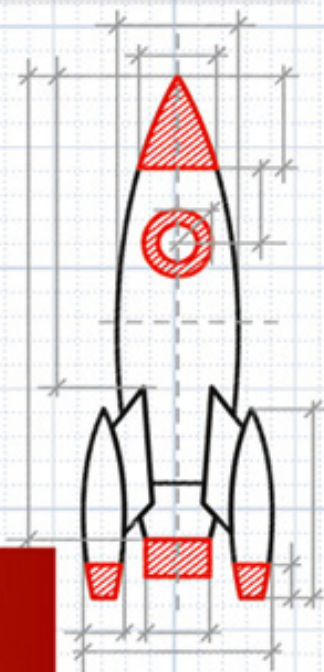
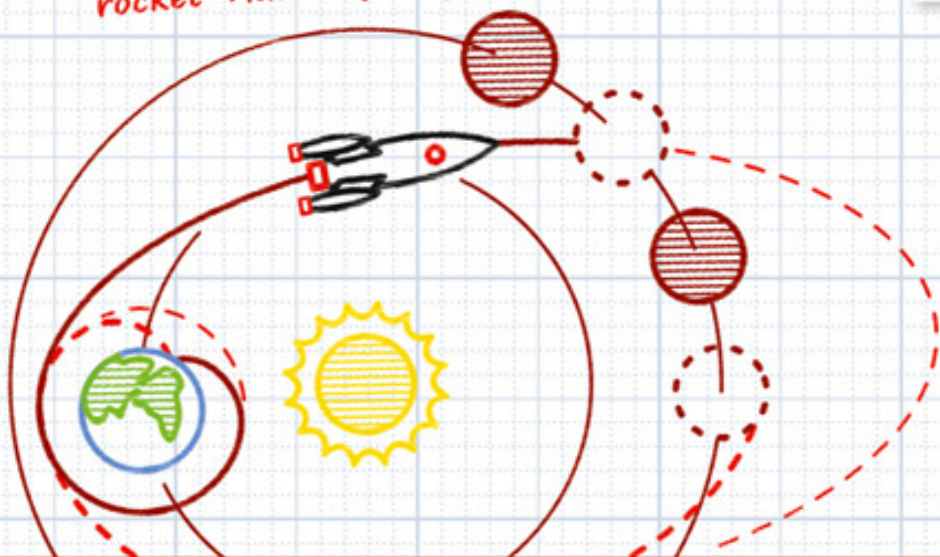


```
DATA(rocket) = zcl_rocket_builder=>build( ).  
rocket->launch( abap_true ).
```

SAP

PRESS



ABAP[®] to the Future

- ▶ Discover the latest and greatest features in the ABAP universe
- ▶ Explore the new worlds of SAP HANA, BRFplus, BOPF, and more
- ▶ Propel your code and your career into the future

Paul Hardy



Rheinwerk
Publishing



SAP PRESS is a joint initiative of SAP and Rheinwerk Publishing. The know-how offered by SAP specialists combined with the expertise of Rheinwerk Publishing offers the reader expert books in the field. SAP PRESS features first-hand information and expert advice, and provides useful skills for professional decision-making.

SAP PRESS offers a variety of books on technical and business-related topics for the SAP user. For further information, please visit our website: www.sap-press.com.

Miroslav Antolovic
Getting Started with SAPUI5
2015, 462 pages, hardcover
ISBN 978-1-59229-969-0

Schneider, Westenberger, Gahm
ABAP Development for SAP HANA
2014, 609 pages, hardcover
ISBN 978-1-59229-859-4

James Wood
Getting Started with SAP HANA Cloud Platform
2015, approx. 575 pp., hardcover
ISBN 978-1-4932-1021-3

James Wood
Object-Oriented Programming with ABAP Objects
2016, approx. 450 pp., hardcover
ISBN 978-1-59229-993-5

Paul Hardy

ABAP® to the Future

To my mother, Mary, who would have loved to have seen this book in print

Dear Reader,

If you're at all familiar with the average SAP PRESS publication, you'll be unsurprised to hear that titling a book *ABAP to the Future* was...shall we say...controversial. Well, I tell you this: SEO-friendly or not, the book lives up to its name. So if you're looking to bring your code, your career, and your programming philosophy (there is such a thing as a programming philosophy!) into the future, you've found your guide.

In my ten years in book publishing, I have learned that there are many different ways to edit: to specification, for concision, with abandon, and even—yes, sometimes—in frustration. However, *ABAP to the Future* and Paul Hardy gave me the opportunity to experience one of the best and rarest ways to edit: with delight. His obvious respect for and fascination with ABAP is on display on every page, and his charmingly irreverent tone is one that will have you chuckling your way out of SE80. Enjoy your journey, dear reader.

Of course, once you've reached your destination, let us know how you found the ride. What did you think about *ABAP to the Future*? As your comments and suggestions are the most useful tools to help us make our books the best they can be, we encourage you to visit our website at www.sap-press.com and share your feedback.

Thank you for purchasing a book from SAP PRESS!

Kelly Grace Weaver

Editor, SAP PRESS

Rheinwerk Publishing

Boston, MA

kellyw@rheinwerk-publishing.com

www.sap-press.com

Notes on Usage

This e-book is **protected by copyright**. By purchasing this e-book, you have agreed to accept and adhere to the copyrights. You are entitled to use this e-book for personal purposes. You may print and copy it, too, but also only for personal use. Sharing an electronic or printed copy with others, however, is not permitted, neither as a whole nor in parts. Of course, making them available on the Internet or in a company network is illegal as well.

For detailed and legally binding usage conditions, please refer to the section [Legal Notes](#).

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy:

Imprint

This e-book is a publication many contributed to, specifically:

Editor Kelly Grace Weaver

Copyeditor Melinda Rankin

Cover Design Graham Geary

Photo Credit Shutterstock.com/46874443/© BiterBig

Production E-Book Kelly O'Callaghan

Typesetting E-Book III-satz, Husby (Germany)

We hope that you liked this e-book. Please share your feedback with us and read the [Service Pages](#) to find out how to contact us.

The Library of Congress has cataloged the printed edition as follows:

Hardy, Paul, 1968-

ABAP to the future / Paul Hardy. -- 1st edition.

pages cm

Includes bibliographical references and index.

ISBN 978-1-4932-1161-6 (print : alk. paper) -- ISBN 1-4932-1161-7 (print : alk. paper) -- ISBN

978-1-4932-1163-0 (print

and ebook : alk. paper) -- ISBN 978-1-4932-1162-3 (ebook) 1. ABAP/4 (Computer program language) 2. Object-oriented

programming languages. 3. SAP ERP. I. Title.

QA76.73.A12H37 2015

005.1'17--dc23

2014046248

ISBN 978-1-4932-1161-6 (print)

ISBN 978-1-4932-1162-3 (e-book)

ISBN 978-1-4932-1163-0 (print and e-book)

© 2015 by Rheinwerk Publishing, Inc., Boston (MA)

1st edition 2015

Contents

Foreword	19
Acknowledgments	21
Introduction	23

PART I Programming Tools

1 ABAP in Eclipse	35
1.1 Installation	37
1.1.1 Installing Eclipse	37
1.1.2 Installing the SAP-Specific Add-Ons	39
1.1.3 Connecting Eclipse to a Backend SAP System	41
1.2 Features	42
1.2.1 Working on Multiple Objects at the Same Time	47
1.2.2 Bookmarking	48
1.2.3 Creating a Method from the Calling Code	50
1.2.4 Extracting a Method	53
1.2.5 Deleting Unused Variables	58
1.2.6 Creating Instance Attributes and Method Parameters	59
1.2.7 Creating Class Constructors	60
1.2.8 Getting New IDE Features	61
1.3 Testing and Troubleshooting	63
1.3.1 Unit Testing Code Coverage	63
1.3.2 Debugging	67
1.3.3 Runtime Analysis	69
1.4 Customization Options with User-Defined Plug-Ins	71
1.4.1 UMAP	73
1.4.2 Obeo	78
1.5 Summary	79

2 New Language Features in ABAP 7.4	81
2.1 Database Access	82
2.1.1 New Commands in OpenSQL	82
2.1.2 Buffering Improvements	85
2.1.3 Creating while Reading	87
2.1.4 Inner Join Improvements	88

2.2	Declaring and Creating Variables	90
2.2.1	Omitting the Declaration of TYPE POOL Statements	90
2.2.2	Omitting Data Type Declarations	92
2.2.3	Creating Objects Using NEW	92
2.2.4	Filling Structures and Internal Tables while Creating Them Using VALUE	93
2.2.5	Filling Internal Tables from Other Tables Using FOR	94
2.2.6	Creating Short-Lived Variables Using LET	95
2.3	String Processing	96
2.3.1	New String Features in Release 7.02	96
2.3.2	New String Features in Release 7.4	97
2.4	Calling Functions	98
2.4.1	Method Chaining	98
2.4.2	Avoiding Type Mismatch Dumps when Calling Functions	99
2.4.3	Using Constructor Operators to Convert Strings	100
2.4.4	Functions That Expect TYPE REF TO DATA	101
2.5	Conditional Logic	102
2.5.1	Using Functional Methods in Logical Expressions	103
2.5.2	Omitting ABAP_TRUE	103
2.5.3	Using XSDBOOL as a Workaround for BOOLC	105
2.5.4	The SWITCH Statement as a Replacement for CASE	106
2.5.5	The COND Statement as a Replacement for IF/ELSE	108
2.6	Internal Tables	109
2.6.1	Using Secondary Keys to Access the Same Internal Table in Different Ways	109
2.6.2	Table Work Areas	112
2.6.3	Reading from a Table	114
2.6.4	CORRESPONDING for Normal Internal Tables	115
2.6.5	MOVE-CORRESPONDING for Internal Tables with Deep Structures	116
2.6.6	New Functions for Common Internal Table Tasks	120
2.6.7	Internal Table Queries with REDUCE	122
2.6.8	Grouping Internal Tables	122
2.6.9	Extracting One Table from Another	124
2.7	Object-Oriented Programming	126
2.7.1	Upcasting/Downcasting with CAST	126
2.7.2	CHANGING and EXPORTING Parameters	127
2.7.3	Changes to Interfaces	128
2.8	Search Helps	129
2.8.1	Predictive Search Helps	129
2.8.2	Search Help in SE80	130

2.9	Unit Testing	131
2.9.1	Creating Test Doubles Relating to Interfaces	131
2.9.2	Coding Return Values from Test Doubles	132
2.9.3	Creating Test Doubles Related to Complex Objects	133
2.10	Cross-Program Communication	134
2.11	Summary	136
3	ABAP Unit and Test-Driven Development	137
3.1	Eliminating Dependencies	139
3.1.1	Identifying Dependencies	140
3.1.2	Breaking Up Dependencies	141
3.2	Implementing Mock Objects	143
3.2.1	Creating Mock Objects	144
3.2.2	Injection	145
3.3	Writing and Implementing Unit Tests	147
3.3.1	Defining Test Classes	148
3.3.2	Implementing Test Classes	153
3.4	Automating the Test Process	161
3.4.1	Automating Dependency Injection	162
3.4.2	Automating Mock Object Creation via mockA	167
3.4.3	Combining Dependency Injection and mockA	170
3.5	Behavior-Driven Development	171
3.6	Summary	173
4	ABAP Test Cockpit	175
4.1	Automatic Run of Unit Tests	177
4.2	Mass Checks	178
4.2.1	Setting Up Mass Checks	180
4.2.2	Running Mass Checks	182
4.2.3	Reviewing Mass Checks	185
4.2.4	Dismissing False Errors	188
4.3	Recent Code Inspector Enhancements	192
4.3.1	Unsecure FOR ALL ENTRIES (12/5/2)	193
4.3.2	SELECT * Analysis (14/9/2)	195
4.3.3	Improving FOR ALL ENTRIES (14/9/2)	196
4.3.4	SELECT with DELETE (14/9/2)	198
4.3.5	Check on Statements Following a SELECT without ORDER BY (14/9/3)	199
4.3.6	SELECTs in Loops across Different Routines (14/9/3)	200
4.4	Summary	202

5 Debugger Scripting 203

5.1 Script Tab Overview 204
 5.2 Coding the SCRIPT Method 209
 5.3 Coding the INIT and END Methods 215
 5.4 Summary 222

6 The Enhancement Framework and New BAdIs 225

6.1 Types of Enhancements 227
 6.1.1 Explicit Enhancements 227
 6.1.2 Implicit Enhancements 228
 6.2 Creating Enhancements 229
 6.2.1 Procedural Programming 229
 6.2.2 Object-Oriented Programming 233
 6.3 Defining BAdIs 235
 6.3.1 Creating an Enhancement Spot 237
 6.3.2 Creating the BAdI Definition 237
 6.3.3 Creating the BAdI Interface 244
 6.4 Implementing BAdIs 245
 6.5 Calling BAdIs 249
 6.6 Summary 252

PART II Business Logic Layer

7 Exception Classes and Design by Contract 255

7.1 Types of Exception Classes 257
 7.1.1 Static Check (Local or Nearby Handling) 258
 7.1.2 Dynamic Check (Local or Nearby Handling) 260
 7.1.3 No Check (Remote Handling) 260
 7.1.4 Deciding which Type of Exception Class to Use 262
 7.2 Designing Exception Classes 263
 7.2.1 Creating the Exception 264
 7.2.2 Declaring the Exception 266
 7.2.3 Raising the Exception 267
 7.2.4 Cleaning Up after the Exception Is Raised 268
 7.2.5 Error Handling with RETRY and RESUME 270
 7.3 Design by Contract 274
 7.3.1 Preconditions and Postconditions 276

7.3.2	Class Invariants	278
7.4	Summary	281
8	Business Object Processing Framework (BOPF)	283
8.1	Defining a Business Object	284
8.1.1	Creating the Object	285
8.1.2	Creating a Header Node	286
8.1.3	Creating an Item Node	288
8.2	Using BOPF to Write a DYNPRO-Style Program	290
8.2.1	Creating Model Classes	291
8.2.2	Creating or Changing Objects	294
8.2.3	Locking Objects	304
8.2.4	Performing Authority Checks	305
8.2.5	Setting Display Text Using Determinations	306
8.2.6	Disabling Certain Commands Using Validations	316
8.2.7	Checking Data Integrity Using Validations	318
8.2.8	Responding to User Input via Actions	325
8.2.9	Saving to the Database	335
8.2.10	Tracking Changes in BOPF Objects	342
8.3	Custom Enhancements	350
8.3.1	Enhancing Standard SAP Objects	351
8.3.2	Using a Custom Interface (Wrapper)	354
8.4	Summary	355
9	BRFplus	357
9.1	The Historic Location of Rules	360
9.1.1	Rules in People's Heads	360
9.1.2	Rules in Customizing Tables	362
9.1.3	Rules in ABAP	364
9.2	Creating Rules in BRFplus: Basic Example	365
9.2.1	Creating a BRFplus Application	365
9.2.2	Adding Rule Logic	373
9.2.3	BRFplus Rules in ABAP	386
9.3	Creating Rules in BRFplus: Complicated Example	388
9.4	Simulations	394
9.5	SAP Business Workflow Integration	397
9.6	Options for Enhancements	401
9.6.1	Procedure Expressions	401
9.6.2	Application Exits	402

9.6.3	Custom Frontends	402
9.6.4	Custom Extensions	403
9.7	Summary	403

PART III User Interface Layer

10 ALV SALV Reporting Framework 407

10.1	Getting Started	409
10.1.1	Defining a SALV-Specific (Concrete) Class	410
10.1.2	Coding a Program to Call a Report	411
10.2	Designing a Report Interface	414
10.2.1	Report Flow Step 1: Creating a Container (Generic/Optional)	416
10.2.2	Report Flow Step 2: Initializing a Report (Generic)	416
10.2.3	Report Flow Step 3: Making Application-Specific Changes (Specific)	424
10.2.4	Report Flow Step 4: Displaying the Report (Generic)	435
10.3	Adding Custom Command Icons Programmatically	441
10.3.1	Creating a Method to Automatically Create a Container	442
10.3.2	Changing ZCL_BC_VIEW_SALV_TABLE to Fill the Container	443
10.3.3	Changing the INITIALIZE Method	444
10.3.4	Adding the Custom Commands to the Toolbar	445
10.3.5	Sending User Commands from the Calling Program	446
10.4	Editing Data	447
10.4.1	Creating a Custom Class to Hold the Standard SALV Model Class	448
10.4.2	Changing the Initialization Method of ZCL_BC_VIEW_SALV_TABLE	449
10.4.3	Adding a Method to Retrieve the Underlying Grid Object	449
10.4.4	Changing the Calling Program	450
10.4.5	Coding User Command Handling	451
10.5	Handling Large Internal Tables with CL_SALV_GUI_TABLE_IDA	454
10.6	Summary	456

11 ABAP2XLSX 457

11.1	The Basics	459
11.1.1	How XLSX Files are Stored	459

11.1.2	Downloading ABAP2XLSX	461
11.1.3	Creating XLSX Files Using ABAP	462
11.2	Enhancing Custom Reports with ABAP2XLSX	466
11.2.1	Converting an ALV to an Excel Object	466
11.2.2	Changing Number and Text Formats	468
11.2.3	Establishing Printer Settings	471
11.2.4	Using Conditional Formatting	474
11.2.5	Creating Spreadsheets with Multiple Worksheets	482
11.2.6	Using Graphs and Pie Charts	484
11.2.7	Embedding Macros	487
11.2.8	Emailing the Result	493
11.2.9	Adding Hyperlinks to SAP Transactions	496
11.3	Tips and Tricks	501
11.3.1	Using the Enhancement Framework for Your Own Fixes ...	501
11.3.2	Creating a Reusable Custom Framework	504
11.4	Summary	505

12 Web Dynpro ABAP and Floorplan Manager 507

12.1	The Model-View-Controller Concept	508
12.1.1	Model	508
12.1.2	View	511
12.1.3	Controller	514
12.2	Building the WDA Application	515
12.2.1	Creating a Web Dynpro Component	517
12.2.2	Declaring Data Structures for the Controller	518
12.2.3	Establishing View Settings	522
12.2.4	Setting Up the ALV	529
12.2.5	Defining the Windows	531
12.2.6	Navigating between Views Inside the Window	532
12.2.7	Enabling the Application to Be Called	535
12.2.8	Coding	535
12.3	Using Floorplan Manager to Modify Existing WDA Components	546
12.3.1	Creating an Application Using Floorplan Manager	548
12.3.2	Integrating BOPF with Floorplan Manager	556
12.4	Summary	559

13 SAPUI5 561

13.1	Architecture	563
13.1.1	Frontend: What SAPUI5 Is	564
13.1.2	Backend: What SAP Gateway Is	564

13.2	Prerequisites	565
13.2.1	Requirements in SAP	565
13.2.2	Requirements on Your Local Machine	566
13.3	Backend Tasks: Creating the Model Using SAP Gateway	567
13.3.1	Configuration	567
13.3.2	Coding	581
13.4	Frontend Tasks: Creating the View and Controller Using SAPUI5 ...	592
13.4.1	View	596
13.4.2	Controller	608
13.4.3	Testing Your Application	613
13.5	Adding Elements with OpenUI5	615
13.6	Importing SAPUI5 Applications to SAP ERP	620
13.6.1	Storing the Application in Releases Lower than 7.31	620
13.6.2	Storing the Application in Releases 7.31 and Above	622
13.6.3	Testing the SAPUI5 Application from within SAP ERP	624
13.7	SAPUI5 vs. SAP Fiori	626
13.8	Summary	627

PART IV Database Layer

14 Shared Memory 631

14.1	The Promises of Shared Memory	632
14.1.1	Database Access	633
14.1.2	Memory Usage	633
14.2	Creating and Using Shared Memory Objects	634
14.2.1	Creating the Root Class	635
14.2.2	Generating the Broker Class	640
14.2.3	Using Shared Memory Objects in ABAP Programs	643
14.3	Updating the Database and Shared Memory Together	647
14.4	Troubleshooting	649
14.4.1	Data Inconsistency between Application Servers	649
14.4.2	Short Dumps	652
14.5	Summary	654

15 ABAP Programming for SAP HANA 655

15.1	Introduction to Code Pushdown	657
15.2	Top-Down Development	658
15.2.1	Building and Calling CDS Views	659
15.2.2	Building and Calling ABAP Managed Database Procedures	671

- 15.3 Bottom-Up Development 677
 - 15.3.1 Building and Calling External Views 678
 - 15.3.2 Building and Calling Database Proxies 679
 - 15.3.3 Transporting Changes 680
- 15.4 Locating Code that Can Be Pushed Down 681
 - 15.4.1 How to Find Custom Code that Needs to Be Pushed Down 682
 - 15.4.2 What Technique to Use to Push the Code Down 684
 - 15.4.3 Example 684
- 15.5 Other Modifications to ABAP for SAP HANA 691
 - 15.5.1 Database Table Design 691
 - 15.5.2 Avoiding Database-Specific Features 695
 - 15.5.3 Changes to Database SELECT Coding 697
- 15.6 Summary 700

16 Conclusion 703

Appendices 705

- A Improving Code Readability 707
 - A.1 Readability vs. Concision 707
 - A.2 The What vs. the How 708
- B Making Programs Flexible 711
- C The Author 719
- Index..... 721

- Service Pages I
- Legal Notes III

Foreword

When I first met Paul Hardy in 1997, he worked as an asset accountant in our UK subsidiary. At the time, we were using a large consultancy to help us implement SAP, and we required some important ABAP programs to be written, mostly custom reports, to assist our business through the migration. They had the SAP expertise, and we had the business knowledge.

Suffice to say, many years later we have both, and many of the ABAP programs the consultancy firm wrote for us at the time were appalling, with little regard to good programming techniques or system performance.

Paul's hobby of dabbling in computer programming at home soon became evident when he taught himself ABAP, and within months his ABAP programs were of much higher quality than those previously written for us. We managed to entice Paul to transfer to Australia when we first implemented SAP there, back in 2000.

Over the ensuing years, Paul has demonstrated time and again that impossible tasks take a little longer to achieve. We continue periodically to run our own version of custom program optimization sessions since the first CPO service was conducted by SAP Australia on our site more than 10 years ago. Paul and his colleagues have taken that process and repeated it no less often than annually to ensure that our system continues to run optimally. It now only takes a day to analyze our entire system, and generally results in only a handful of changes to be performed.

Paul has a unique humor and a way of getting his message across. One of our executive team called him "The Wild Thing" when he was quickly able to whip up a sales-related report in a matter of days in response to a napkin specification.

I've read Paul's book, and I must say that even though he has discussed many of these topics with me in the kitchen, I still found all 15 chapters compelling reading. What I came to appreciate is that this is not only a book for programmers (the unsung heroes of many companies) but also really is a must-read for CIOs and development managers to help them set strategies around what programming

toolsets your company should be adopting and what techniques should be considered and mandated within your development teams, rather than leaving it to the programmers alone.

It's now obvious to me that ABAP functionality and the associated toolsets have not stood still over the past decade, and there are many tools that have been around for 10 years or more that are still not exploited. It would appear that we need to invest in the education of our ABAP programmers (and our web programmers as these technologies converge) to ensure that we are doing things optimally, allowing for ease of future program maintenance, and providing an efficient and self-documenting environment. After all, many of us will retire in the next 10, 20, or 30 years, and we need to ensure that our IT assets are given the best chance of survival long after we have departed.

Educating your programmers and giving them the extra time to come up to speed could sound like a delay for your current program of work, but, as Paul explains, over a computer program's lifetime the initial writing accounts for only 5% of the effort. The remaining 95% of the effort relates to future maintenance (through enhancements, bug fixes, programmers needing to acquaint or reacquaint themselves with the logic, testing, upgrades, etc.). Rather than taking shortcuts, it's worth the initial effort to write optimally the first time around.

I recommend this book to anyone in an IT department running SAP and hope you find it as enlightening as I did. Hanson is fortunate to have the services of a great programming team, and Paul has helped shine a light on it by his passion for assisting others.

Rob Downing

Chief Information Officer

Hanson Australia Pty Ltd.

Acknowledgments

Writing this book has been like sticking your hand into a plug socket. There are just so many advances happening in each area of SAP every single month that it's always important to get a second opinion. For this book, I found it very handy to "phone-a-friend" and have someone look over my draft chapters and provide input.

I would like to thank the following people for their invaluable help on various chapters:

- ▶ The staff at Obeo and Mathias Märker for their help on Chapter 1, ABAP in Eclipse.
- ▶ Uwe Kunath, for his help on Chapter 3, ABAP Unit and Test-Driven Development.
- ▶ Carsten Ziegler, for his help on Chapter 9, BRFplus.
- ▶ Ivan Femia, for his help on Chapter 11, ABAP2XLSX.
- ▶ Dominik Ofenloch, for his advice on Chapter 12, Web Dynpro ABAP and Floorplan Manager.
- ▶ Graham Robinson and Thomas Meigen for their help on Chapter 13, SAPUI5. Graham created the example used in the chapter, and Thomas explained the SAP Gateway naming strategy.

Naturally, any remaining mistakes are mine.

I would also like to thank my editor Kelly Grace Weaver; this is my first book, so she had to do a lot more work on the manuscript than is probably the case with more seasoned authors!

In addition, many thanks to the staff—Elaine, Joy, Colm, and Seamus—at Cookies Lounge Bar in North Strathfield, NSW, where the vast bulk of this book was written.

Last but not least, I'd like to thank my wife for putting up with my spending vast chunks of my spare time over the last year on this book!

It may be hard for an egg to turn into a bird: It would be a jolly sight harder for it to learn to fly while remaining an egg. We are like eggs at present. And you cannot go on indefinitely being just an ordinary, decent egg. We must be hatched or go bad.

—C. S. Lewis

Introduction

Eggs are no doubt lovely places to be inside, full of food and warm from mum sitting on top of the egg. However, living inside an egg most likely does not give you a very good idea of the changes taking place in the outside world. In the world of SAP, such change can be horrifyingly scary: There is an army of programmers at SAP constantly disgorging a stream of new goodies. Developers around the world would probably love them—but often 99% of such programmers remain hidden inside their eggs, blissfully ignorant of the new treasure chest in their SAP system basement.

To avoid this all-too-common situation, the purpose of this book is to shine a spotlight on such improvements. As such, it is aimed at any and all inquisitive ABAP developers. While I cover new technology, I also cover some topics that might not be considered “new”; this is because I have noticed that many people who are unaware of SAP-delivered improvements that came out six months ago are equally unaware of improvements that came out five years ago. I will give these topics some attention because it’s very possible that you are discovering them for the first time.

The idea for this book comes from an observed reluctance to use new technology and, at the same time, a compulsion to use new technology. One of the best blog posts ever on the SAP Community Network website was one by Graham Robinson called “A Call to Arms for ABAP Developers.” In this post, Robinson provides a (supposedly fictitious) example of an ABAP developer who joins a consulting company at the start of the SAP wave in the late nineties. He learns all there is to know about SAP development as it was at that point—all at the client’s expense—

and then, once the boom dies down, gets a permanent job at one of the former clients. For a while, all is good, but after 10 years, the developer realizes that he knows nothing about any of the new technology SAP has come out with, because it was always possible to get by just using the same techniques that worked in the year 2000.

Sure, if I had a dollar for every time I read the words “game changing” or “inflection point” in the IT press, then I could use that pile of money to build a ramp to the moon. But what cannot be denied is that in recent years SAP has been coming out with radical new ideas at a faster and faster pace. A lot of this new technology does not use the ABAP language or SE80 development environment, and the pace of change has moved from getting used to a new version of something every five years to getting used to a new version of something every 12 weeks. Either of those things in isolation would be enough to scare people who aren't used to change, and the combination gets them hiding under the sofa and hoping all this will go away. (It won't.) Like it or not, it's probably a good idea to, at the very minimum, have a rough idea of what all these scary things—like SAP HANA—are all about. The alternative is crossing your fingers and hoping really hard that they won't affect you for another 10 years.

Of course, trying to keep up with the nonstop stream of innovations coming out of SAP these days is like drinking from the proverbial fire hose: Since SAP adopted the agile methodology, the time between releases of various products has decreased noticeably, and SAP is such a large organization that the new features come at you from all sides at once.

Still, if you try to push the latest and greatest SAP invention onto your colleagues, you may hear the following: (a) you should not use this just because it's new; and (b) because it's new, it's risky, and it probably won't work anyway. As always, there is no black and white position on this. A lot of developers do in fact like using new things because they are new, regardless of whether it is appropriate. And some of the new things really are good!

Regardless of the specific technology being discussed, you'll notice two major themes that run through this book: object-oriented (OO) programming and anti-fragile programming. To get you ready for this, I want to say a few brief words about these concepts.

SAP introduced ABAP Objects, the OO version of ABAP, in version 4.6, which came out in the year 2000. After 15 years, you would expect most, if not all, ABAP programmers to be using this style of programming. As far as I can see, nothing could be farther from the truth. I would say that even now most new code at customer sites (and a fair bit from SAP itself) is written in a procedural way. Some people haven't even started procedural programming yet and write everything in one huge block with no subroutines. There have been huge debates on the SAP Community Network as to whether OO programming is better than procedural. I truly believe it is, but regardless of what I think, you will find that if you want to take advantage of some of the latest advances from SAP that I describe later in this book, then you really have no choice. If you're new to OO, here's a tip: The two standout books I have read on OO programming that have actually helped me in my day-to-day work are *Clean Code* and *Head First Design Patterns* (see the "Recommended Reading" box at the end of this introduction). Read them!

The concept of creating antifragile programs was introduced by Robert Martin (though not in those exact words), who noted how computer software tends to rot over time: The constant changes you have to make tend to add more and more conditional logic and the like, bloating subroutines and making the code more and more complex until no one has any idea how to maintain it without breaking something (i.e., it gets more brittle as time goes on). Robert Martin wants us programmers to fight back against this sort of entropy by applying the boy scout rule, which is "Always leave the campsite cleaner than you found it." The idea is that because you have to change the program anyway, make tiny changes each time to the area you're changing to make it clearer (i.e., rename obscure variables so you can tell what they do, split up a huge subroutine into two so that it's easier to understand, and so on). Over time, your code base should get better instead of slowly turning from steel to glass.

What in the world does all this have to do with the list of new goodies I'm about to discuss? More than you might think. The traditional position has been to avoid new tools like the plague, because they're "risky"—that is, new. In this book, in almost every chapter you'll see the reverse argument popping up. I'm going to try to show how these new tools can be used to make your programs simpler to read or more robust: in short, more resistant to change. Why is it important to try to shield your programs from the changes you have to make? Maybe because the rate of change in IT has already accelerated to a breakneck speed and is only getting faster.

What I Am Not Going to Talk About

I think it is fairly safe to say that no company in the world relies solely on its SAP system to get business done. There will always be assorted external systems, some of which are interfaced to SAP ERP, some not. SAP itself has quite a wide variety of products, stemming from the time just after the year 2000 when the new dimension products (like SAP CRM, SAP SCM, and so on) appeared.

Where I work, we have a fair few non-SAP ERP products in use: SAP BW, SAP Business-Objects BI, SAP PI, and Ariba, to name just a few. I could easily talk about advancements in those areas as well, but I'm going to concentrate on the bit in the middle of the Venn diagram: This book will focus on the latest ABAP programming-related features that you can take advantage of if you have an SAP ERP system. I am not going to talk about anything for which you need a separately licensed system outside of SAP ERP.

Structure of the Book

The book is split into sections that mirror the great strides SAP has made recently in the areas of development tools, business logic, user interfaces (UIs), and database technology. This book will not only explain how to use these new technologies—with an example application running throughout to keep the focus concrete rather than abstract—but will also note how these technologies fit in with best practice programming philosophies. In addition, where appropriate, the book highlights custom ways to get around perceived problems in these new tools or ways to enhance them.

This book is organized into four parts. First, you'll look at the actual tools you need when writing programs in the first place. Then, the last three sections mirror the layers of an application: the business logic layer, the UI/presentation layer, and the database layer. Each part and its chapters are described in more detail next.

Part I: Programming Tools

Chapter 1: ABAP in Eclipse

The obvious place to start is the development environment itself, and in recent times SAP has pushed its customers to start using Eclipse as opposed to SE80. You'll explore why this is a good thing and how this tool enables you to do the same things you have always done, only faster. In addition, some of the scary new

technologies developed by SAP (e.g., SAPUI5 and SAP HANA) require the use of ABAP in Eclipse. There are also specialized integration tools in Eclipse for some topics covered in other sections of the book, such as the Business Object Processing Framework (BOPF) and Web Dynpro ABAP (WDA). Therefore, if you really want to be at the cutting edge, then you need to become familiar with Eclipse in a hurry.

Chapter 2: New Language Features in ABAP 7.4

Fish is no good without chips, and ABAP in Eclipse would likewise not be much good without ABAP. The good news, for the nervous among us, is that despite all the horrifying new tools SAP has invented, you will still do the bulk of your programming in ABAP. The even better news is that SAP continues to enhance the ABAP language. Chapter 2 covers recent innovations before the 7.4 release that do not seem to be widely used yet, and then moves on to the radical changes in the 7.4 release that will pretty much allow you to write programs with half the lines of code you needed before.

Chapter 3: ABAP Unit and Test-Driven Development

ABAP Unit has been around for a while but is not widely used. I feel that this is a crying shame, which is why this topic is near the start of the book. The idea is that you write your tests first before coding any actual business logic, and if followed properly, then you can make your applications cast iron in regard to their ability to not buckle under the constant stream of changes required.

This applies to every single other topic in this book: Everything you create needs to be testable. (You will find that some technologies mentioned in the book, like BOPF and BRFPplus, have their own test tools attached as well, but that is just icing on the unit testing cake. ABAP in Eclipse also has excellent support for ABAP Unit, for both helping you create the tests and running them.)

Chapter 4: ABAP Test Cockpit

When you follow the recommended development lifecycle of an application (which I didn't make up, by the way!), first you write the tests, then you write production code until those tests pass, and then, once everything is working, you refactor the code (i.e., make it better without changing the function).

The first part of refactoring can be a static code check. SAP has come up with the ABAP Test Cockpit, which is in some senses an extension of the Extended Program Check and the Code Inspector, with extra bits on top. It can also help you prepare your programs for migration to an SAP HANA database.

Chapter 5: Debugger Scripting

After dealing with static checks of code at design time, you move on to debugging the code at runtime. The debugger in ABAP has always been one of its strongest features, and in this chapter you'll look at a relatively new enhancement: the ability to write programs within the debugger to automate common or labor-intensive tasks.

Debugging is also available from ABAP in Eclipse and can in fact do one amazing thing that debugging in the SAP GUI cannot. Curious? Read Chapter 5!

Chapter 6: The Enhancement Framework and New BADIs

I couldn't leave the area of development tools without taking a quick look at the enhancement framework. This is usually viewed solely as a means to stick user exits all over the place, but it also has a role to play in your own custom developments.

Part II: Business Logic Layer

Chapter 7: Exception Classes and Design by Contract

As you start to write your business logic, you've already got a bunch of unit tests and have performed static code checks and debugged running code. However, you know that unexpected situations are bound to occur and have to decide how to handle this in your applications. The technical way to deal with such problems is to make use of the exception classes that SAP introduced to replace traditional error handling by means of return codes. I'll also talk about one philosophical way to look at this problem: the design by contract pattern invented by Bertrand Meyer.

Chapter 8: Business Object Processing Framework (BOPF)

Over the years, SAP has come up with many ways to model real-world objects inside the ABAP system. The latest is the Business Object Processing Framework

(BOPF), which provides a modeling transaction in which there are places for all common tasks, such as queries, consistency checks, and derived values.

Chapter 9: BRFplus

Due to the fact that SAP ERP is a business-focused application, you're likely used to constantly reading values out of the configuration tables and the IMG, along with your own Z tables. BRFplus is a decision rules engine, which on a technical level vastly extends the capacity of the IMG to store the actual business rules in a way that is easily visualized. On a philosophical level, it devolves responsibility for those rules from IT to the people who actually create them.

Part III: User Interface Layer

Chapter 10: ALV SALV Reporting Framework

Until web applications perform as fast as desktop ones, there will still be a place for reports that run inside the SAP GUI. Given that fact, the SALV reporting framework was created to speed up development of such reports. In this chapter, you'll see how to create a reusable framework to automate the more monotonous tasks that need to be performed when creating a SALV report and look at how to work around some of its perceived limitations. You'll also take a quick look at a new version of the SALV class SAP has invented specifically for situations in which there is an SAP HANA database in the backend but you're still running a SALV-style report.

Chapter 11: ABAP2XLSX

Despite years of attempts to stop people from using Microsoft Excel, a large number of business users spend a large part of their working day playing with spreadsheets. In this chapter, you'll look at the open-source project ABAP2XLSX and how you can improve integration between SAP ERP and Excel enormously, thus saving a huge amount of time.

Chapter 12: Web Dynpro ABAP and Floorplan Manager

Web Dynpro ABAP has been around for a fair while now, but how many people are actually using it? The Floorplan Manager (a specialized Web Dynpro technology) is even newer and is still rapidly evolving. In this chapter, you'll look at how

Web Dynpro ABAP works with the model-view-controller (MVC) design pattern, and then get an introduction to Floorplan Manager.

Chapter 13: SAPUI5

SAPUI5 is the latest and most successful attempt to cure SAP's poor reputation in the UI space. You'll look at how SAPUI5 also uses the MVC pattern, enforced much more than it is with Web Dynpro ABAP, and how it looks better and runs faster as well. You'll also see how to expose data that lives in SAP ERP via the use of an SAP Gateway service and how to create an SAPUI5 application to consume that data.

Part IV: Database Layer

Chapter 14: Shared Memory

For those of you not yet able to work with an SAP HANA database, it's worth taking a look at the shared memory framework to see if it can help prevent a whole heap of unneeded database access.

Chapter 15: ABAP Programming for SAP HANA

People are often terrified of SAP HANA; it's portrayed as so new and revolutionary that ABAP programmers wonder if it will put them out of a job. In this chapter, I'll set everyone's mind at rest and explain what code pushdown really is and how your programming skills are going to be just as relevant as ever in this brave new world.

The Example Application

In order to keep my feet (and yours) on the ground, I'm going to base the example application used throughout this book on a real application I've worked on—with the real subject matter hidden by changing all the class, method, and attribute names via find and replace to protect the innocent and avoid giving away trade secrets.

Given that I can't say what I'm working on and that I'm not going to use the good old SAP practice of using `SFLIGHT`, this book's example is based on a project for

Baron Frankenstein to create a better monster. His prior monsters didn't work too well, so SAP convinced him to sign up for its rapid deployment service for cloud-based, mobile, in-memory monster making.

One million LEU (the currency in Transylvania) later, he realized that the out-of-the-box solution didn't give him everything he wanted, so he hired you to write a monster-making program in SAP ERP. In this application, he inputs all the attributes he wants from a monster, and your program uses algorithms as complicated as the human genome project to prepare the final data, which then gets interfaced to his monster-making machine via web services during a thunderstorm.

During the course of the following chapters, you'll see (with code examples) that, luckily for the baron, you can take advantage of assorted new innovations recently delivered from SAP to aid in his monster-making quest.

Now, it's time to raise the curtain....

Recommended Reading

- ▶ A Call to Arms for ABAP Developers: <http://scn.sap.com/community/career-center/blog/2012/10/24/a-call-to-arms-for-abap-developers> (Graham Robinson)
- ▶ *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin, Prentice Hall, 2008)
- ▶ *Head First Design Patterns* (Freeman et al., O'Reilly, 2004)
- ▶ Antifragile Software: <http://scn.sap.com/community/abap/blog/2013/12/01/antifragile-software> (Vikas Singh)

PART I
Programming Tools

It's best not to stare at the sun during an eclipse.
—Jeff Goldblum

1 ABAP in Eclipse

ABAP's main development environment, the ABAP Workbench (Transaction SE80), has improved with each new release and has often added features found in another popular integrated development environment: Eclipse. Now, SAP has bitten the bullet and adopted Eclipse itself for ABAP programming. In fact, you'll see that a lot of the topics covered in this book—the ABAP Test Cockpit, the Business Object Processing Framework (BOPF), Web Dynpro, Floorplan Manager, and SAPUI5—are supported by specific tools within ABAP in Eclipse. Some of the more advanced SAP functionalities—such as creating ABAP-related objects for use in SAP HANA—can *only* be done by using ABAP in Eclipse.

Eclipse is an open-source development environment that started life in its current form around about 2001. It really started to rock in about 2004, and since then a new version has been released near the end of June every year, each named after a planet or a satellite.

To oversimplify, Eclipse has traditionally been a much-used development platform for Java programmers. (To check if this was true, I asked a Java programmer at the pub if she used Eclipse; she told me yes, she did, and it was “tops”—so that settles that.) In fact, Eclipse even mentions Java on its loading screen. (It also mentions Oracle; you can imagine what SAP thinks about that.) Of course, Eclipse is not limited to Java, and you should check out Wikipedia if you're interested in a full list of compatible languages. (My personal favorite is Groovy, mostly because it must be wonderful to go to a party and—in the unlikely event that anybody cares—be able to say that you are a Groovy programmer.)

There is a massive support community for Eclipse, from conferences to online magazines. To understand why, you need to realize that ABAP programmers have been rather spoiled by having the Transaction SE80 development environment entirely inside the ABAP repository. In other languages, even relatively

small programs can generate a large number of files, which are initially developed and stored on your local machine. Then you have to deploy them somewhere and make sure, for large projects, that the different versions of the program on different machines do not overwrite each other. That all sounds rather painful, and no one likes to be burdened with mundane tasks while developing programs. Therefore, you need a really good development environment to take care of these tasks for you; Eclipse is that environment.

If you've been paying attention, you might notice that more recent features in the ABAP Workbench look suspiciously like features that you find in Eclipse (e.g., automatic code completion or coloring keywords differently from variables). Eventually, SAP decided to make the leap; in July 2012, SAP NetWeaver Development Tools for ABAP (ADT) was released. Everyone calls it "ABAP in Eclipse," because (a) that's what it is and (b) the official name makes you sound like you have swallowed a dictionary when you say it. (You may be familiar with this phenomenon from other SAP product names.)

Although Eclipse is a step in the right direction, change isn't always easy, and SAP quite rightly suspected that traditional ABAP developers would be horrified by the very thought of not performing development tasks in Transaction SE80 or its subset transactions, like SE24, SE37, or SE38. To try and cushion the blow, SAP gave advance warning that it was about to release ABAP in Eclipse about a year before the release, and naturally the SAP Community Network (SCN) website exploded with "I would rather die than use this" comments (a slight exaggeration, but only slight). These comments were virtually always from people who had never even heard of Eclipse before, let alone used it to develop an application. As a response, I had to go and put a cat amongst the pigeons by publishing a very short blog post called "SE24 is Rubbish" in which I extolled the virtues of Eclipse. (SE24 is not rubbish; I was just stirring people up.) A torrent of abuse descended on my head—but now it's only two years later, and judging from the blogs on SCN, the tide has started to turn as more and more people try ABAP in Eclipse and discover they like it (though to be fair, even now there are still a lot of negative comments).

The purpose of this chapter is to show you the features of ABAP in Eclipse and let you decide for yourself if it might speed up your day-to-day work. My position is—naturally—that it will, so let us see if I can convince you. Section 1.1 will explain the process of installing Eclipse on your local machine and look at the

very basics of how to use it. Once you've got that down, you're ready for Section 1.2, which will examine some of the things you can do while developing ABAP programs in Eclipse that you cannot do at all (or not do very easily) when performing the equivalent tasks in Transaction SE80 within the SAP GUI. Section 1.3 will talk about how ABAP in Eclipse handles a vital part of the development process: the testing and troubleshooting of your custom programs. Finally, Section 1.4 explains how to extend the functionality of Eclipse by using custom plug-ins to add extra functionality, either by using freely available open-source tools or by creating your own tools.

Warning: Houston, We Have a Problem

To install Eclipse, your backend system needs to be at least an SAP NetWeaver 7.31 SP 4 system. If you're one of the SAP customers still on 7.02 or below, Eclipse is not for you! There's a reason this book is called *ABAP to the Future*, after all.

1.1 Installation

There are three parts to setting up an Eclipse development environment so that you can use it to create and change ABAP programs:

1. Install Eclipse itself on your local PC.
2. Add the SAP-specific tools.
3. Connect your local Eclipse environment to a backend SAP system.

This section will guide you through each of these steps.

1.1.1 Installing Eclipse

You might expect to start by visiting <https://www.eclipse.org>, where you will see many links to news items about the annual Eclipse conference, newsletters, community groups, and so on; most importantly, you'll see a big button marked DOWNLOAD.

The gotcha is that this page will give you the latest version of Eclipse, and SAP typically runs about three months behind; thus, it may be that if you download the latest version of Eclipse, it will be literally too good to use. So, before getting all excited and pressing the exciting DOWNLOAD button, visit <https://>

tools.hana.ondemand.com/#abap; there you will see what the latest supported version of Eclipse is and the link you should follow to download it (Figure 1.1).

The screenshot shows the SAP Development Tools for Eclipse website. The navigation bar includes links for HOME, ABAP TOOLS, BW TOOLS, CLOUD TOOLS, GATEWAY TOOLS, SAP HANA TOOLS, SAP HCI TOOLS, and SAPUI5 TOOLS. The main content area is titled "ABAP Development Tools for SAP NetWeaver" and provides instructions on installing and updating the front-end components of ABAP Development Tools for SAP NetWeaver (ADT). It also provides information on how to prepare the relevant ABAP back-end system for working with ADT.

Prerequisites

Eclipse Platform	Kepler (4.3)
Operating System	<ul style="list-style-type: none"> Windows OS (XP*, Vista, or 7) 32- or 64-Bit, or Apple Mac OS X 10.6, Universal 64-Bit, or Linux distribution <p>* The compatibility is no more tested by the Eclipse Community since Eclipse Kepler (4.3)</p>
Java Runtime	JRE version 1.6 or higher, 32-Bit or 64-Bit
SAP GUI	<ul style="list-style-type: none"> For Windows OS: SAP GUI for Windows 7.30 For Apple Mac or Linux OS: SAP GUI for Java 7.30
Microsoft VC Runtime	<p>For Windows OS: DLLs VS2010 for communication with the back-end system is required.</p> <p>NOTE: Install either the x86 or the x64 variant, accordingly to your 32- or 64-Bit Eclipse installation.</p>

Procedure

To install the front-end component of ADT, proceed as follows:

1. Get an installation of [Eclipse Kepler](#).

Figure 1.1 Installing Eclipse: Part 1

When you follow the link—for Eclipse Kepler in the example shown in Figure 1.1—you will arrive at a slightly different version of the screen mentioned at the start of this section, but the important thing is that it still has a big DOWNLOAD button for you to click. Below the DOWNLOAD button, you are presented with a dazzling array of choices. Figure 1.2 shows only the first two, but you can keep paging down and the options keep on coming.

Either of the options shown in Figure 1.2 are okay for using ABAP in Eclipse, but as it turns out you are best off choosing the one called IDE FOR JAVA EE DEVELOPERS. In Chapter 13, you will need the extra features this option provides when you start working with SAPUI5.

Warning: Houston, We Have a Problem

Each option has a 32-bit and a 64-bit option. Make sure you choose the one that is compatible with the version of Java that is running on your machine; i.e., a 32-bit version of Java and a 64-bit version of Eclipse do not play well together.

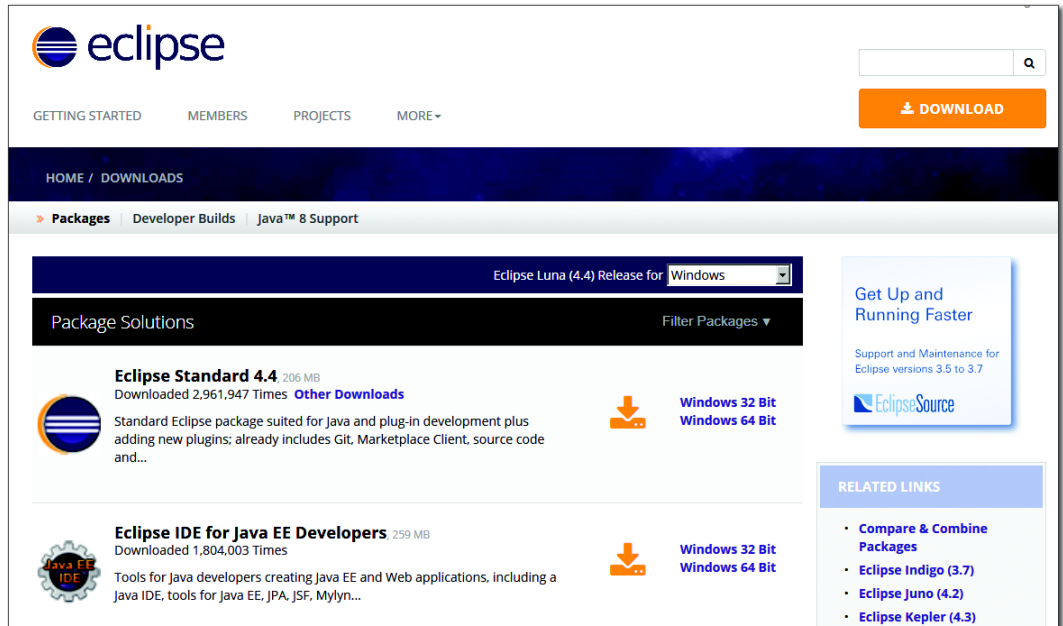


Figure 1.2 Installing Eclipse: Part 2

Downloading this program to your local machine is so straightforward that there is no need to go into further detail, so let's move straight on to what happens once the install is finished and you have a lovely, planet-shaped icon sitting on your desktop.

1.1.2 Installing the SAP-Specific Add-Ons

From any screen in Eclipse, you can choose the **HELP • INSTALL NEW SOFTWARE** menu option, and a pop-up box will appear, asking you to enter a website. The URL takes the following format: *https://tools.hana.ondemand.com/[name of latest supported release]*. For this example, it is *https://tools.hana.ondemand.com/kepler*.

Figure 1.3 shows a tree structure of all the possible goodies you can install that relate to SAP development. You could download everything in the tree if you wanted (you will certainly download the SAPUI5 options later), but for now download everything in the **ABAP DEVELOPMENT TOOLS FOR SAP NETWEAVER** section.

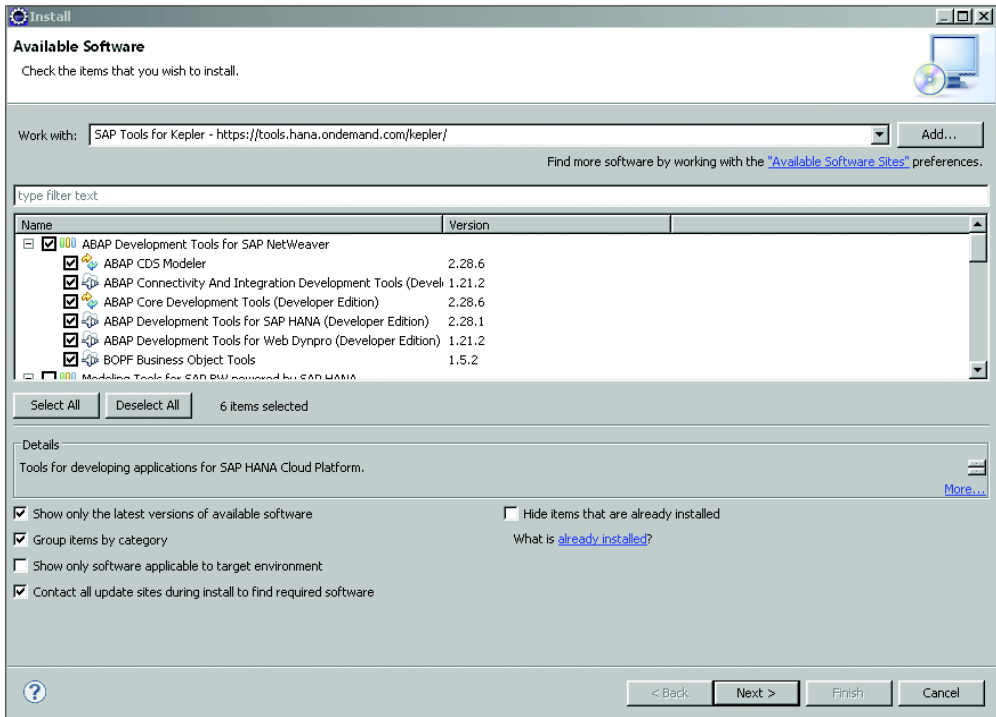


Figure 1.3 Installing SAP Add-Ons

If you try to install the SAP add-ons for the Luna release of Eclipse, you may be presented with all sorts of strange error messages. You can fix the problem by first installing the Eclipse Modeling Framework, which the SAP add-ons seem to need; install this via the same INSTALL NEW SOFTWARE path, but choose the main Eclipse download site from the dropdown options. Look for an option with “EMF” and “SDK” in the title, and install that.

If you do not receive an error message when asking to install the SAP add-ons, then the next thing you see is a list of all the options you chose in a very slightly different format (as part of an ARE YOU SURE prompt). Next comes the ever-popular license agreement. Then you will be asked if you want to restart Eclipse, which you do; otherwise, the SAP tools will not be active.

Now, the Eclipse WELCOME page opens. If you scroll down, you will see lots of SAP-specific options. First, there are several SAP HANA-specific ones (of course), and after that there is the ABAP DEVELOPMENT section (Figure 1.4).

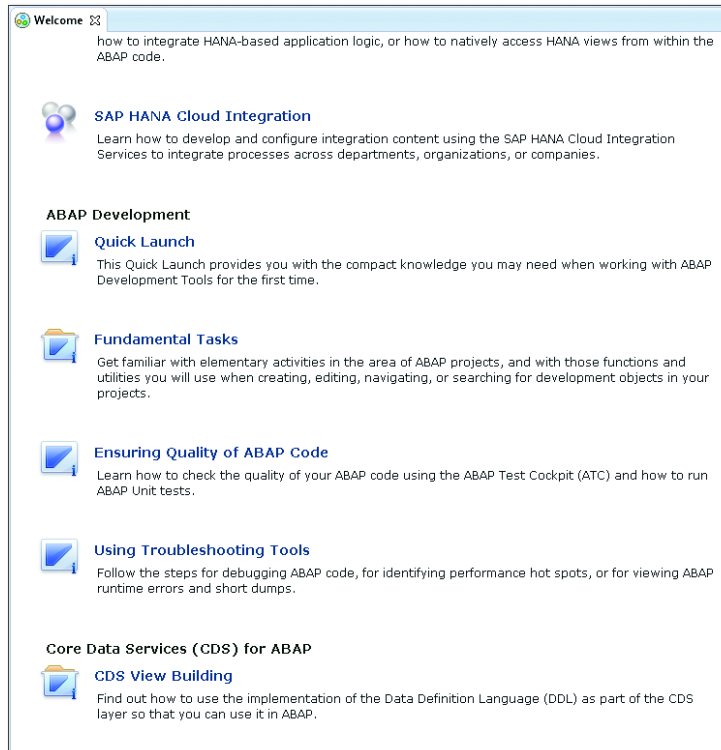


Figure 1.4 Eclipse Welcome Page with SAP-Specific Options

If you open one of those options—for example, QUICK LAUNCH—then the online help will open up and present a bucket load of information. A sizeable chunk of the chapters on the left-hand side of the screen address ABAP or SAP HANA.

1.1.3 Connecting Eclipse to a Backend SAP System

As mentioned earlier, you need a backend system that is at least SAP NetWeaver 7.31 SP 4 in order to connect to Eclipse. To connect that backend to Eclipse, follow the Eclipse menu path FILE • NEW • OTHER • ABAP • ABAP PROJECT. (A “project” in this context is nothing more or less than a connection to a backend ABAP system.)

In Figure 1.5, you will see that the SELECT CONNECTION FROM SAP LOGON radio button is selected. This means that when you click BROWSE, a list of available SAP systems from our locally installed logon pad appears. When we select a system, the bottom half of the pop-up box is populated with the system details.

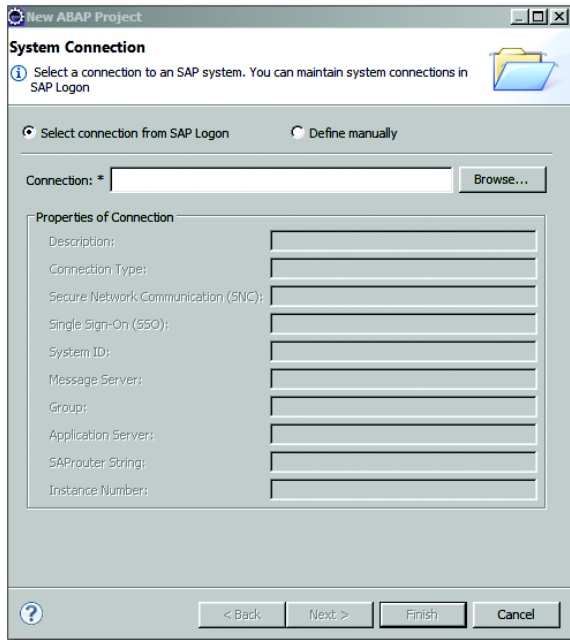


Figure 1.5 Connecting ABAP to Eclipse

SELECT CONNECTION FROM SAP LOGON is the option to take if the files for your SAP logon pad are on your local machine as opposed to on some sort of central server. If your logon pad has the list of SAP systems centrally stored on a server, you can instead choose the DEFINE MANUALLY option and fill in all the boxes in the pop-up box (application server, etc.) with the needed values.

On the next screen, enter the client number and your user name and password. Click FINISH. At the bottom of the screen, you will see that Eclipse is checking whether or not the target system is compatible. If it is, then another box appears asking for favorite packages and the like. For the time being, ignore that box, and click on NEXT and then FINISH. Just like that, you have a new tool to develop your various ABAP repository objects.

1.2 Features

When you first connect your backend system to Eclipse, you will see a welcome message on the right side of the screen (Figure 1.6).

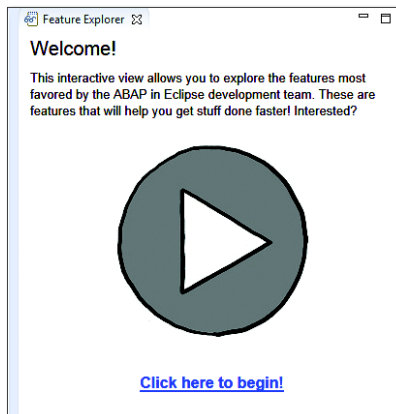


Figure 1.6 ABAP in Eclipse Feature Explorer

Clicking `CLICK HERE TO BEGIN!` launches a tour of ABAP in Eclipse—quite a comprehensive tour at that, so large that you will have to take some time to work through it all. Luckily, you can do this at your own pace, marking each area as “explored” once you have to come to grips with it so that you can continue at a later date from where you left off.

The tutorial covers the whole topic of ABAP in Eclipse in far more detail than anyone could hope to cover in one chapter of a book, so the remainder of the chapter won’t talk about how to do in Eclipse the things you can already do; instead, the primary focus will be on the things that you can do in ABAP in Eclipse that you cannot normally do in an SE80 environment. However, before diving into those details, you should learn how to perform a basic, but very important, action in Eclipse: calling and modifying repository objects. With these basics under your belt, you’ll be better prepared to dive into the more advanced features that are the focus of this section.

In Figure 1.7, you will see the left-hand side of the first screen shown after connecting Eclipse to ABAP. Just like the SAP menu, there is a `FAVORITES` tree at the top for the `Z` packages you are working on at any given time and a big list of every single package in the SAP system at the bottom. (This latter piece is not much use, really, because the tree is so huge.)

Just like the favorites menu in the normal SAP system, you can right-click the `FAVORITE PACKAGES` star to add a new development package. An entry for local objects is added automatically.

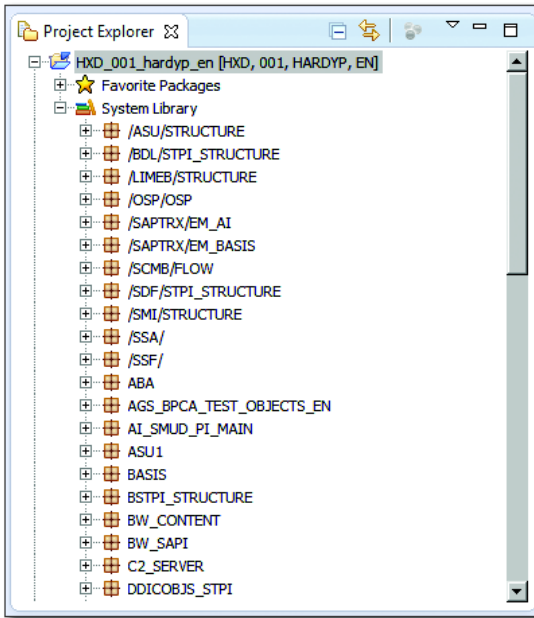


Figure 1.7 Eclipse View of ABAP Packages

Say that you want to change one of your Z classes, and you cannot remember its exact name. Normally, you would go to SE24, type "ZCL_BC*" (as an example), and open the dropdown. (If you're anything like me, sometimes you would not see any result. Then you'd suddenly realize you were in SE37 by mistake.) The equivalent action for searching for custom objects in Eclipse is to press **CTRL** + **SHIFT** + **A**. A search box appears. You type in the start of your object, and with every letter that you add the list of possible matches is narrowed (Figure 1.8). The speed at which Eclipse searches the ABAP repository is far faster than one might expect, often faster than the same search within the actual SAP system, which seems like black magic.

Select one of your Z classes, and the screen shown in Figure 1.9 appears; hopefully, it doesn't look so different from SE80 that you will die of culture shock on the spot. You will notice that there are various windows in an Eclipse view, and you can close them or resize them and even drag and drop them all over the screen. The system will remember the window layout if you exit Eclipse and come back later.

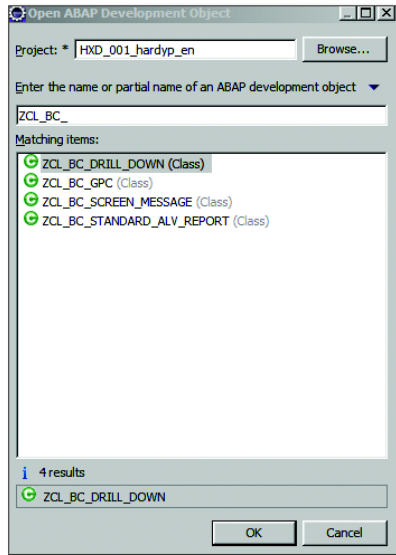


Figure 1.8 Searching for an ABAP Repository Object

That last bit is important, because usually you log off from SAP at the end of the day, and the next day you log back on and go looking for the line of the program where you left off yesterday. In Eclipse, you start where you left off.

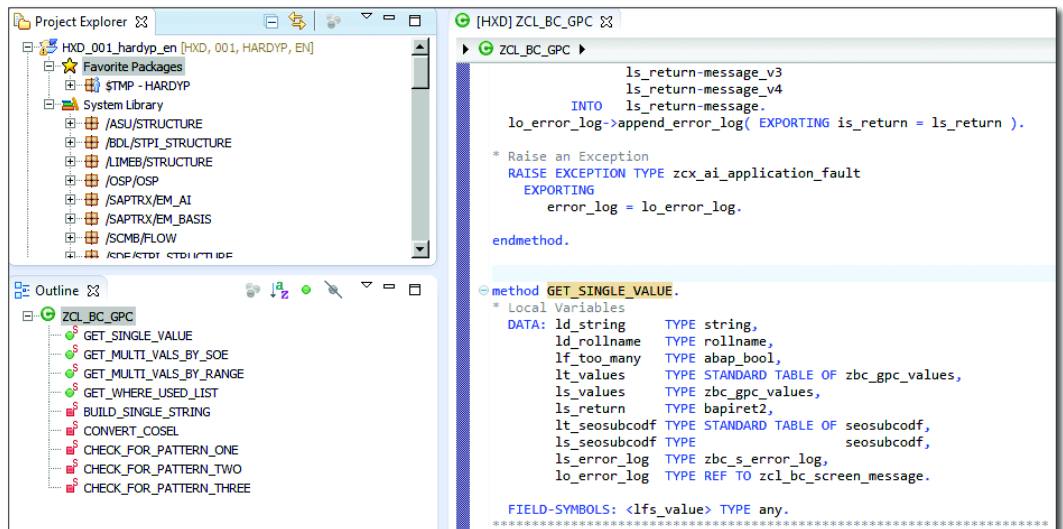


Figure 1.9 Editing a Class in Eclipse

You will note that the class is displayed in what would be the source code view inside SE24. Eclipse always displays objects in this way; it has no concept of the formatted screens like those in SE37 and SE24. To Eclipse, everything is just a big program. As an example, when viewing a function module the signature appears at the start of the code, as shown in Listing 1.1.

```
FUNCTION ZBC_GET_USERS_COUNTRY
  EXPORTING
    E_LAND LIKE TTZ5-LAND1.
```

Listing 1.1 Signature at Start of Function Module Code

You change the signature by changing the source code just like you would when changing the signature of a FORM routine in SE38 or SE80, instead of having to jump between different tabs in SE37. Something else that will take some getting used to is that you are always in change mode. You will also notice that your comments are spell-checked.

At this point, the time has come to prove that this is real and not some sort of dream. Choose one of your Z classes, and add a comment, such as "I changed this code in Eclipse." Then click the ACTIVATE icon at the top of the screen. A pop-up with a progress bar will display while the change is propagated to the SAP system. Next, go into that SAP system, and use SE24 to see if the change actually occurred. You will find that it has, at which point everything becomes amazingly real.

Note

As an aside, at the time of writing I have a very flaky connection to my SAP system. Eclipse keeps telling me when the connection drops out, but I can still keep coding; when the connection comes back 30 seconds later, I can save my changes back to SAP. That is far better than being thrown out of the system every few minutes.

At this point in the chapter, you have learned the absolute basics: what Eclipse is, how to install it and connect it to your ABAP development system, and how to call up and change ABAP repository objects via Eclipse. It is quite probable at this point that a lot of developers are thinking, "So what? This looks pretty much like SE80. It's a lot of extra effort to make the exact same changes I would make in SE80 in this new external framework."

If that were true, then you would imagine that no one would ever develop programs using ABAP in Eclipse twice, let alone on an ongoing basis. Therefore, now it's time to examine some of the features that make life easier for an ABAP devel-

oper to such an extent that the extra minute or so spent opening up Eclipse in the morning while you go and get a cup of coffee is more than compensated for by less development time spent on common tasks throughout the day.

7.4 vs. 7.31

When new Eclipse features come out in new releases of SAP NetWeaver, they come out in the new release first and then get backported to previous releases at a later date. For example, the features that were added in 7.4 SP 5 were later backported to 7.31 in SP 11. Accordingly, most of the features discussed in this section will work in both 7.31 and 7.4. However, if this is not the case, it will be mentioned specifically.

1.2.1 Working on Multiple Objects at the Same Time

The maximum number of sessions you can have open in the SAP GUI is seven. (The original designers probably thought that was more than enough. After all, human beings can pretty much only do one thing at a time, so you could not possibly be working on eight or more different things simultaneously.) As it turns out, developers often want to be able to look at lots of different things at the same time: maybe several methods that call each other in a chain while also having similar objects open to check how they (or someone else) achieved the same sort of task before. Then, of course, someone might come up behind you to ask you something, and to sort out his problem you need to open an unrelated program, and you do not want to shut down anything you are currently working on. Before you know it, you get to the maximum number of sessions and have to decide what you want to close. This is not the end of the world, but it would be nice to have as many objects open as you wanted just to save that little bit of time (because, of course, the session you shut down turns out to be the one you suddenly have an urgent need to look at again). In addition, you'll probably eventually have a need to look at two or more sessions at once without having to keep toggling between them. In recent years, the answer to this has been to have two monitors or even three. I suppose you could fill up the wall with monitors if you wanted. This is all wonderful, but it could be described as a hardware solution to a software problem.

Naturally, I wouldn't have brought all this up if Eclipse did not offer a solution to this problem—which, of course, it does. Because you are not in the SAP GUI, you can have as many windows containing program objects open as you want and can drag and drop and resize the windows to see as many or as few as you want at any given time (Figure 1.10).

The screen in Figure 1.10 looks overly crowded, but when programming you're not that interested in beauty, only in seeing as much information as you need at any given point. Naturally, if you have two (or more) monitors, then you can fill them up with boxes as well (rather like the cop shows on TV in which they fill up the wall with clues to the crime).

You may notice that one of the boxes open in the screenshot in Figure 1.10 was not an ABAP program at all but a JavaScript program from an SAPUI5 application you will be developing later in this book. The point to be made here is that as SAP evolves you will find yourself developing parts of the application in (gasp) languages other than ABAP. That concept may fill you with horror, but as time goes by you are more and more likely to find yourself in such a situation, and if you do you can probably see the benefit in being able to work on every part of the program in the same development environment, rather than having to keep switching between environments.

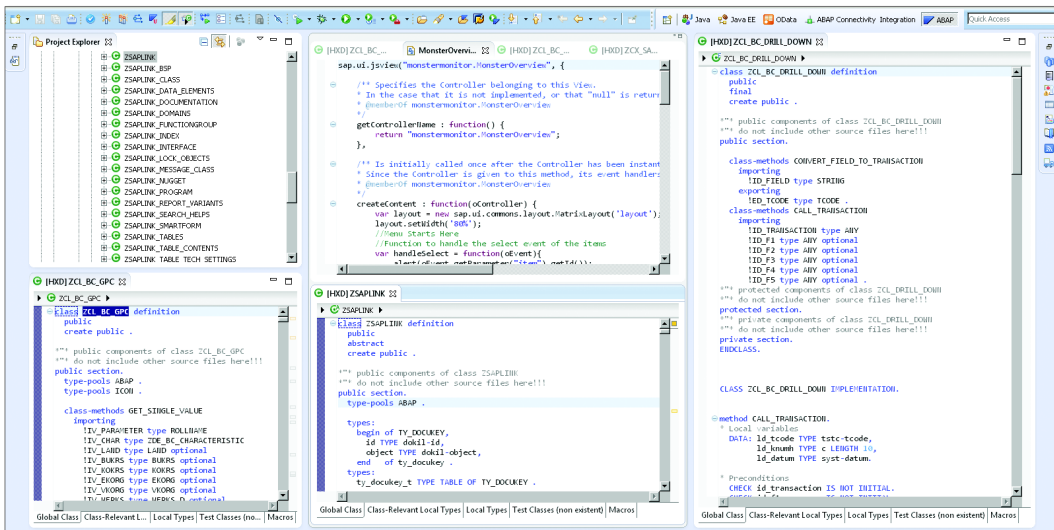


Figure 1.10 Busy Eclipse Screen Full of Objects

1.2.2 Bookmarking

At any given time, you are really never working on only one thing but on a (hopefully) finite number of ABAP constructs (called “artifacts” in Eclipse): programs, classes and function modules, and the like. However, the number can be quite large in a complex development project, and we humans have a horrible tendency

to forget the exact name of a class—even if we work with it twenty days in a row—and to need to go searching for it in SE24.

In ABAP in Eclipse, you can right-click the artifact you want to bookmark, and when you do you'll see a huge context menu, with ADD BOOKMARK near the bottom (Figure 1.11).

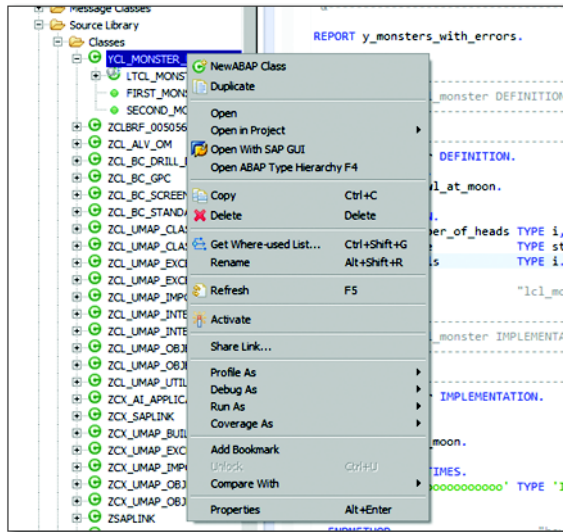


Figure 1.11 Bookmarking an ABAP Artifact

Then, a few hours or days later you can click on a little box on the right of the screen that says BOOKMARKS when you hover your cursor over it; the box shown in Figure 1.12 appears.

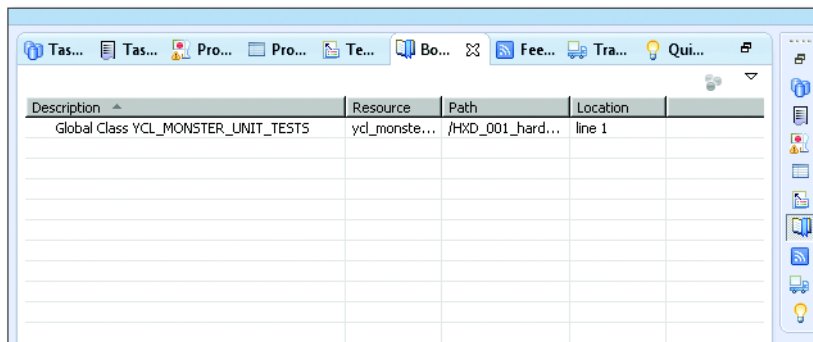


Figure 1.12 List of Bookmarks

Double-click the one you want, and—bingo—it opens up, and you can start changing things to your heart's content.

1.2.3 Creating a Method from the Calling Code

ABAP Objects was introduced with release 4.6C of SAP, and ever since that point SAP has been pushing developers to use it. Even now, there is a lot of resistance. If SAP wants programmers to switch to object-oriented (OO) programming, then OO programming has to be just as easy as or easier than the procedural programming that programmers have been used to.

Sadly, even after all these years, in your SE80 environment there is one thing you can do with procedural programming that you cannot do with OO programming using local methods: FORM routines. For example, consider the sample procedural code in Listing 1.2, which calls a FORM routine.

```
PERFORM create_monster USING    1d_number_of_heads
                           CHANGING 1d_monster_number.
```

Listing 1.2 FORM Routine

If there were no such FORM routine, then when you double-click CREATE_MONSTER the system would ask you if you wanted to create that missing FORM routine. Then, it would create it for you, along with the signature (though you would have to manually code the TYPES of the signature parameters).

Now, consider the object-oriented code calling a method in Listing 1.3.

```
1d_monster_number = lo_laboratory->create_monster( 1d_number_of_heads).
```

Listing 1.3 Method

You may expect that, if there were no such method, then the exact same thing would happen, and a skeleton method implementation and definition would be created for you when you double-clicked the CREATE_MONSTER method. You would, unfortunately, be wrong.

People have moaned and groaned about that on the Internet for 14 years to no avail. What happens when you try the same thing in ABAP in Eclipse? First of all, write a small program (Listing 1.4).

```
CLASS lc1_laboratory DEFINITION.
PUBLIC SECTION.
```

```

METHODS: main.

ENDCLASS. "Laboratory Definition

CLASS lcl_laboratory IMPLEMENTATION.

METHOD main.
* Local Variables
DATA: ld_monster_number TYPE i,
      ld_number_of_heads TYPE i.

      create_monster( id_number_of_heads = ld_number_of_heads ).

ENDMETHOD.

ENDCLASS. "Laboratory Implementation

```

Listing 1.4 Sample Program with Missing Method

As you can see, the `create_monster` method does not exist. In the past, you couldn't just double-click it; you had to create a definition, and then create an implementation, and then find your way back to the place you first started, making you think that maybe `FORM` routines weren't so bad after all.

In Eclipse, you put your cursor on the not-yet-extant `CREATE_MONSTER` and press `[CTRL] + [1]`. A little box appears and asks you if you want to automatically create the missing method. I know I do; I have been waiting 14 years for this, so I say yes.

In Figure 1.13, the definition of the `IMPORTING` parameter came from the variable definition you made in the code, which is one better than the `FORM`-based equivalent in which you have to manually add the variable type.

Sadly, you may have to manually add the `RETURNING` parameter; in my version of ABAP in Eclipse, the autocreation did not work if I entered `ld_monster_number = create_monster(ld_number_of_heads)`. Nonetheless, that is probably just a bug which will be fixed in time. In any event, the end result is as shown in Listing 1.5.

```

CLASS lcl_laboratory DEFINITION.

PUBLIC SECTION.
METHODS: main.

PRIVATE SECTION.

METHODS create_monster

```

```

IMPORTING
    id_number_of_heads TYPE i
RETURNING
    value(rd_monster_number) TYPE i.
ENDCLASS. "Laboratory Definition

CLASS lcl_laboratory IMPLEMENTATION.

METHOD main.
* Local Variables
DATA: ld_monster_number TYPE i,
      ld_number_of_heads TYPE i.

      ld_monster_number = create_monster( ld_number_of_heads ).

ENDMETHOD.

METHOD create_monster.

ENDMETHOD.
ENDCLASS. "Laboratory Implementation

```

Listing 1.5 Automatically Created Method Implementation

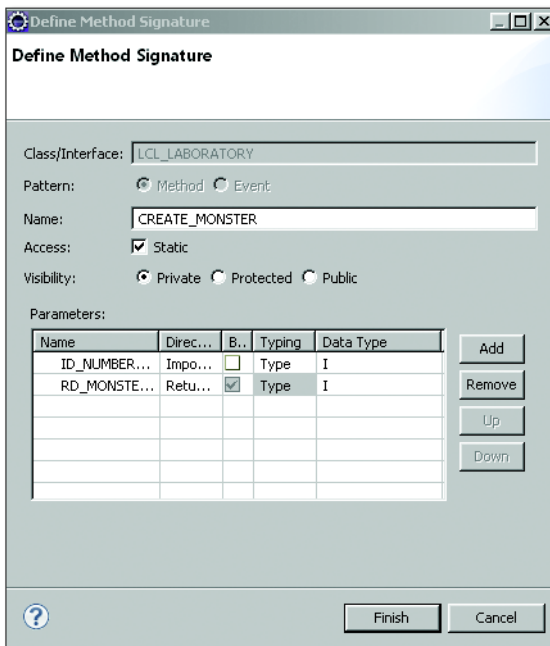


Figure 1.13 Autocreation of a Method

The definition and implementation have been created for you, the cursor jumps to the method implementation (which of course is where you want to be), and at long last local methods have caught up with FORM routines and even jumped ahead due to the signature TYPE definitions being extracted from the variable declarations. You can also create the method implementation after you have created the method definition via the same menu path—`CTRL` + `I`—something else that has been missing in the standard ABAP Workbench.

Quick Assist

The Eclipse term for `CTRL` + `I` is Quick Assist. You can open an area of the screen (view), leave it open in the bottom-left corner or wherever you want, and then you can see what proposals the system has for you in regard to creating missing things automatically or other types of refactoring (Figure 1.14).

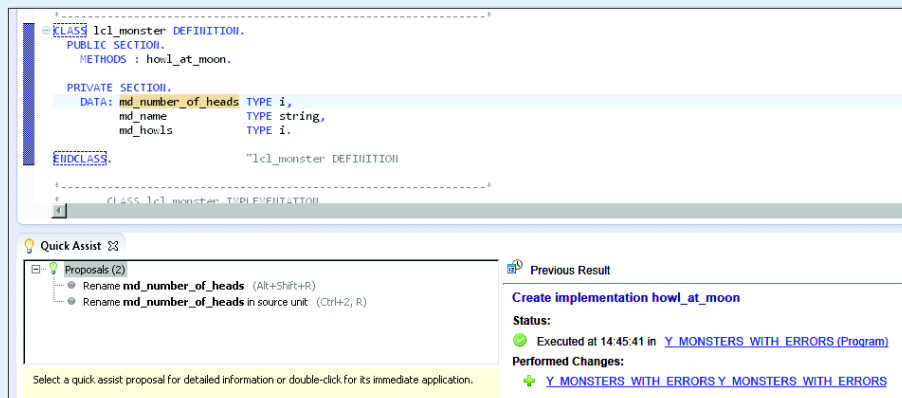


Figure 1.14 Quick Assist View

You won't need to have such a screen area open once you're used to all the things ABAP in Eclipse can do for you, but at the start this is very helpful to remind you that not everything has to be done manually any longer.

1.2.4 Extracting a Method

Programming is a battle against the code rotting over time, and part of the anti-fragile principle discussed in the book's introduction is that programmers must constantly look for ways to make their programs resistant to the ever-increasing sea of changes those programs must undergo. One of the principle causes of pro-

gram fragility is *duplicate code*. It's so easy to cut and paste a big lump of code from one place to the other and just change the variables, leaving the logic exactly the same. Before you know it, you have the same piece of code repeated all over the program or, even worse, in many different programs.

Then the change comes: the logic needs to be adjusted. You now have to make that change in 54 different places, and *of course* you are going to forget to make the change in some of them. This can lead to unpredictable results. The user will experience different behavior when doing the same thing in different contexts; in other words, the change has snapped the program in two, like a glass twig being jumped up on and down on by an elephant during an earthquake. The obvious solution is that when you want to copy and paste a chunk of code and make some small changes you should extract the bulk of the code into its own method, with `IMPORTING` parameters for the parts that vary. That sounds all fine and dandy in principle, but it's a bit of an effort, and it's so much easier to highlight the code and do the `[CTRL] + [C]/[CTRL] + [V]` trick, and people are in a hurry, so that's what they do, little realizing they've just made a rod for their own (and their colleagues', present and future) backs.

Fortunately, ABAP in Eclipse gives you a tool to automatically extract chunks of code into their own methods. Listing 1.6 shows sample code that you might suddenly realize you need in lots of different places, with the variables changed but the logic the same.

```
CLASS lcl_monster IMPLEMENTATION.

    METHOD main.
    * Local Variables
      DATA: ld_monster_madness1 TYPE i,
             ld_monster_madness2 TYPE i,
             ld_monster_madness3 TYPE i,
             ld_description1     TYPE string,
             ld_description2     TYPE string,
             ld_description3     TYPE string.

      ld_monster_madness1 = 25.
      ld_monster_madness2 = 50.
      ld_monster_madness3 = 100.

    * Derive Monster Sanity
      IF ld_monster_madness1 LT 30.
        ld_description1 = 'FAIRLY SANE'.
      ELSEIF ld_monster_madness1 GT 90.
```

```

        ld_description1 = 'BONKERS'.
ELSE.
        ld_description1 = 'AVERAGE SANITY'.
ENDIF.

IF ld_monster_madness2 LT 30.
        ld_description2 = 'FAIRLY SANE'.
ELSEIF ld_monster_madness2 GT 90.
        ld_description2 = 'BONKERS'.
ELSE.
        ld_description2 = 'AVERAGE SANITY'.
ENDIF.

IF ld_monster_madness3 LT 30.
        ld_description3 = 'FAIRLY SANE'.
ELSEIF ld_monster_madness3 GT 90.
        ld_description3 = 'BONKERS'.
ELSE.
        ld_description3 = 'AVERAGE SANITY'.
ENDIF.

ENDMETHOD.

ENDCLASS.

```

Listing 1.6 Sample Code with Changing Variables

How many times have you seen code like that—the same construct again and again, with only the variables changing? This is in fact criminal behavior—and the victim is the programmer who wrote the code—when there are so many ways to make your life easier (e.g., macros).

To fix this problem, highlight the first IF/THEN/ELSE block—be sure to start with the comment line—and press **ALT** + **SHIFT** + **M**, or choose the menu option **SOURCE • EXTRACT METHOD**. Up pops the box shown in Figure 1.15. You may wonder where the proposed name came from; it is in fact the comment line, which was why it was important to include the comment in the highlighted block. You may have no comment or a bizarre comment, so of course you can change the proposed name.

You will notice that Eclipse has looked at the variables in the highlighted section and turned them into parameters. It sometimes gets confused, but generally the result is fairly good. You can change the name of the proposed parameters and the direction. In this case, you do not want to change the monster madness level, so turn the proposal into an **IMPORTING** parameter.

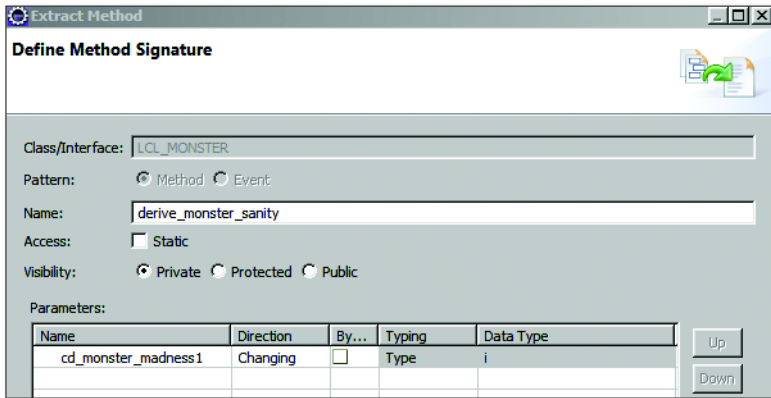


Figure 1.15 Extracting a Method

Next, you will see a box asking for a transport request (if this is not a local object) and whether you want to activate everything after the extraction is complete. The final—and perhaps best—part is that before the change is made you see a before and after comparison, showing what your code will look like after the proposed change has been made (Figure 1.16). If you do not like the look of what the system is about to do, then you can cancel.

After the extraction, three changes have been made. First, a definition of a new private method has been created (Listing 1.7).

```
PRIVATE SECTION.

METHODS derive_monster_sanity
IMPORTING
    id_monster_madness1 TYPE i.
```

Listing 1.7 Definition of New Private Method

Next, the block of code you highlighted has been replaced by a method call (Listing 1.8).

```
ld_monster_madness1 = 25.
ld_monster_madness2 = 50.
ld_monster_madness3 = 100.

derive_monster_sanity( ld_monster_madness1 ).
```

Listing 1.8 Method Call Replacement

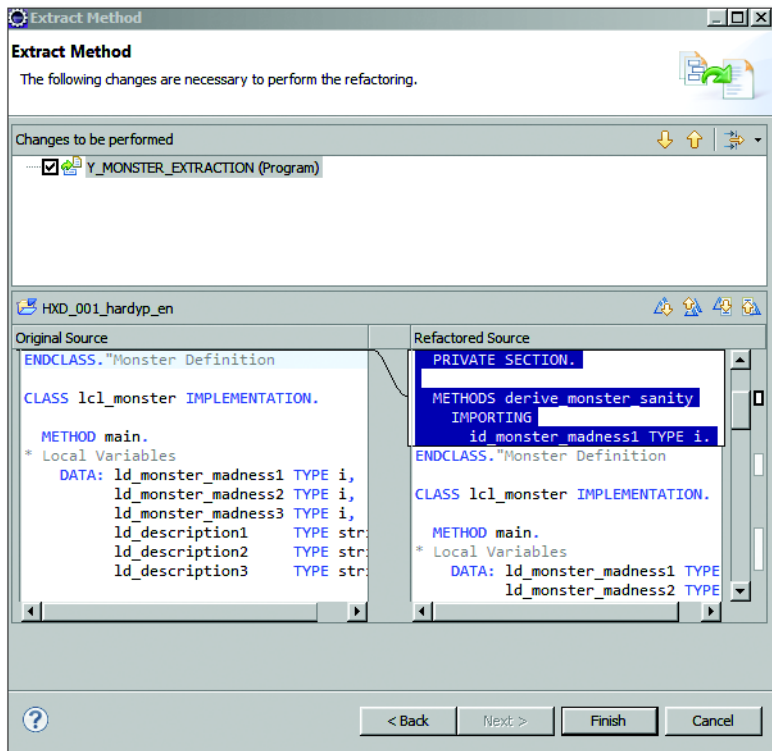


Figure 1.16 Extracting a Method: Before and After Comparison

Last, of course, there is a new method implementation (Listing 1.9).

```

METHOD derive_monster_sanity.
  DATA ld_description1 TYPE string.
  * Derive Monster Sanity
  IF id_monster_madness1 LT 30.
    ld_description1 = 'FAIRLY SANE'.
  ELSEIF id_monster_madness1 GT 90.
    ld_description1 = 'BONKERS'.
  ELSE.
    ld_description1 = 'AVERAGE SANITY'.
  ENDIF.
ENDMETHOD.

```

Listing 1.9 New Method Implementation

Careful observers will note that the Extraction Wizard is not as clever as all that, because you really wanted the description as an `EXPORTING` parameter, and you do

not need your comment any more—but that's a minor nitpick. It's easy enough to change the newly created method, and certainly this process involves a lot less effort than manually creating the implementation and definition.

Furthermore, it's likely that in higher releases you will be able to tell Eclipse to go hunting for identical blocks and replace them all with calls to your new method (i.e., in the preceding example all three IF/THEN/ELSE constructs would have been replaced with method calls).

1.2.5 Deleting Unused Variables

A common programming mantra is “First make it work—then make it good.” Following this principle means messing around until you get whatever it is working and then clearing up the mess you made while experimenting.

Traditionally, when programming inside the SAP GUI using SE80 or a related transaction, the standard extended program check (SLIN) is good at helping with this; you are given a list of variables that have been declared but never get used. This happens because you needed the variables at some point while the program was evolving but then deleted the code that used them because you no longer needed that code. Unused variables are bad, because they take up some memory at runtime for no purpose and also because they clutter the code, making it harder to read, which is a cardinal sin.

Say that as a part of your monster project you're called upon to write a program to perform the most complicated mathematical operation in the world. It takes months of sweat and tears, but at the end you have a program that comes up with the correct result; it looks like Listing 1.10.

```
METHOD main.  
* Local Variables  
  DATA: ld_use_this  TYPE i,  
         ld_also_this TYPE i,  
         ld_result   TYPE i,  
         ld_not_used  TYPE i,  
         ld_not_used2 TYPE i,  
         ld_not_used3 TYPE i.  
  
  ld_use_this  = 1.  
  ld_also_this = 1.
```

```
ld_result = ld_use_this + ld_also_this.
```

```
ENDMETHOD.
```

Listing 1.10 Most Complicated Mathematical Operation in the World

The result is now correct, but over the thousands of versions of this program you have created some variables that are not used. Inside SAP, you usually run the extended syntax check, which provides a list of these unused variables, and then you go into each one, one at a time, and delete them.

In Eclipse, you choose the menu option **SOURCE • DELETE UNUSED VARIABLES (ALL)**, and in one millionth of a second all of your unused friends are gone, crying all the way that their purpose in life has vanished. You also have the option to delete unused variables only in the area you have highlighted.

Just How Clever is the Refactoring Tool in Eclipse?

As a test, I made sure that if I added `##NEEDED` at the end of an unused variable declaration it was not removed, and indeed it is not, which is good. Before using this option, then, make sure that all variables that are accessed dynamically have such a pragma alongside them.

1.2.6 Creating Instance Attributes and Method Parameters

Another valuable feature of Eclipse is the ability to create instance attributes and method parameters. For example, say that you've written the code inside a method without declaring any `IMPORTING` parameters for the method or any variables in the class definition (Listing 1.11).

```
METHOD howl_at_moon.

    DO md_howls TIMES.
        MESSAGE 'Oooooooooooooo' TYPE 'I'.
    ENDDO.

ENDMETHOD.                                "howl_at_moon
```

Listing 1.11 Missing `IMPORTING` Parameters and Variables

If you position your cursor on the `MD_HOWLS` variable and press `CTRL + I`, you will see a list of options. You can choose to make `MD_HOWLS` a class attribute, in which case a `DATA` declaration is created in the `PRIVATE` section of the class definition

(how the TYPE is chosen is black magic), or you can say that the variable is an IMPORTING or CHANGING (or whatever) parameter, and then the method signature in the class definition is changed accordingly.

1.2.7 Creating Class Constructors

A common object-oriented rule is that you should never create objects using the CREATE OBJECT statement; rather, you should have a factory method that gives you the object. The advantage of this is that if your class variable is referenced to an interface rather than a concrete class, then the factory method will decide the exact subclass for you, which makes the program more resistant to change.

In ABAP in Eclipse, if you put your cursor on the class name and press **CTRL** + **1** then you get assorted options, including creating the factory method, creating a class constructor, or creating an instance constructor (Figure 1.17).

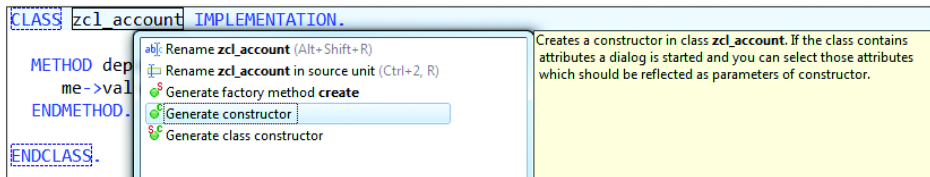


Figure 1.17 Automatically Generating a Constructor

Moreover, if you have declared some variables in the class definition before you do this, then a wizard pops up and asks you which of these variables you want to have as IMPORTING parameters in the constructor. Therefore, if you had member variables for a logging class and the inventor name, for example, which you wanted passed in every time a new instance was created, you would choose them from the list presented. The generated result would look like Listing 1.12.

```
CLASS zcl_monster DEFINITION.
  CONSTRUCTOR IMPORTING i_logger TYPE REF TO zcl_bc_logger
                  i_inventor_name TYPE zde_inventor_name.

CLASS zcl_monster IMPLEMENTATION.
  METHOD constructor.
    Super->constructor( ).
    me->logger = i_logger.
    me->inventor_name = i_inventor_name.
  ENDMETHOD.
```

Listing 1.12 Automatically Generated Constructor

Automatically creating a factory method that returns an instance of the class is a really useful feature as well. All the purists say that you should try and put your `CREATE OBJECT` statements inside factory classes so that you can return a subclass without disturbing the calling program. This is one of the features that is only available in some higher levels of the standard ABAP Workbench; you need a 7.4 system (SP 5) until such time as these features are backported to 7.31 (which is in 7.31 SP 11). The point here is that features are available in Eclipse before they arrive in the “real” ABAP Workbench, often a long while in advance. Conveniently, this is a point that leads nicely into the next section.

1.2.8 Getting New IDE Features

As was just mentioned, in certain cases what functionality you get—for example, in the Extraction Wizard—is dependent both on the ABAP in Eclipse level you are on (you can update to the latest version 10 seconds after it comes out, as it lives on your PC) and on what your backend SAP NetWeaver version is (you are a bit more stuck here).

For example, when you call up the wizard to extract a method, in the bottom-left-hand corner is a question mark, which brings up the online help. Here, you can browse through all the wonderful things you can do, with code examples, but you also will see a matrix listing the minimum level your ABAP system needs to be on to take advantage of each feature (Figure 1.18).

This can be quite frustrating, because it means that whatever version of ABAP in Eclipse you have has the possibility to do far more than your backend ABAP system allows, and new versions of ABAP in Eclipse come out quite frequently.

Nonetheless, there are new features added to ABAP in Eclipse that are not tied to the backend release level, and more importantly Eclipse as a framework gets a new major release every year and, like all open-source projects, gets minor updates regularly. Therefore, you can take advantage of new features that relate to the development environment as a whole on an ongoing basis—as opposed to having to wait five to seven years for your company to do an SAP upgrade that will improve your SE80 experience as a by-product (for some reason, companies do not perform upgrades based on what us developers want). As an example of how fast ABAP in Eclipse advances, when I started writing this chapter I wanted to be sure I had the latest version of ABAP in Eclipse, so I downloaded it on a Tuesday—and a new version came out that Thursday. (That was somewhat of a coincidence, but new versions are released a lot more often than once a year, and are sometimes only two months apart.)

SAP - ABAP Development User Guide > Tasks > Fundamental Tasks and Tools > Editing ABAP Source Code > Refactoring ABAP Source Code

Extracting Variables

In the source code editor, you have the following options for extracting variables:

Function	Description	Available since Release	
		NW 7.31	NW 7.40
Extracting Local Variables from Expressions	Assigning the selected expression to a new local variable. The selected expression is replaced with the new local variable.	-	SP05
Assigning a Statement to a New Variable	Assigning the value of the selected statement to a new local variable or attribute.	SP11	SP05
Converting Locals to Class Members	Converting a local constant, local variable, or local type to a class member such as a member constant, attribute, or member type of the current class.	SP11	SP05
Converting Local Variables to Parameters	In a certain method, converting an existing local variable to a new parameter.	SP11	SP05
Declaring Variables from Usage	Creating a declaration for an attribute within a method.	SP11	SP05
Declaring Inline Variables Explicitly	In the method signature, converting an existing inline declaration of a local variable into an explicit declaration.	-	SP05
Deleting Unused Variables	Supported deletion of unused data declarations and variables.	SP06	SP02

Figure 1.18 Feature Availability Matrix

In order to make the idea of development environment–related improvements a bit more real, it's time for a specific example. If you download the Luna release of Eclipse, you'll see a welcome page containing a **WHAT'S NEW** section, which gives you a list of the most important improvements between this new version of Eclipse and the last one. As might be expected, most of the items in this list do not have anything to do with SAP—but when it comes to improvements in how you can customize the screen layout in Eclipse and new features for the code editor, these are indeed improvements we can take advantage of.

One of the new features that was added in Luna is the ability to show (and edit) different parts of the same program at once (Figure 1.19). If you press **CTRL** + **SHIFT** + **-**, then the editor gets split in two vertically, and if you press **CTRL** + **SHIFT** + **J**, then it gets split in two horizontally. In the first case, you might want to have the variable declarations at the start of a method in the top box and the code that uses those variables in the second box. This helps to avoid paging up and down or double-clicking a variable to reach its definition.

```

1  sap.ui.jsview("monstermonitor.MonsterOverview", {
2
3  /** Specifies the Controller belonging to this View.
4   * In the case that it is not implemented, or that "null" is returned, this View does not have a Controller.
5   * @memberOf monstermonitor.MonsterOverview
6   */
7  getControllerName : function() {
8      return "monstermonitor.MonsterOverview";
9  },
10
11  /** Is initially called once after the Controller has been instantiated. It is the place where the UI is constructed.
12   * Since the Controller is given to this method, its event handlers can be attached right away.
13   * @memberOf monstermonitor.MonsterOverview
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70  var rTitle = new sap.ui.commons.Title("rTitle");
71  rTitle.setText("List of All Monsters");
72  resultPanel.setTitle(rTitle);
73  var oTable = new sap.ui.table.DataTable();
74  oTable.addColumn(new sap.ui.table.Column({ label: new sap.ui.commons.Label({text: "MonsterNumber"}), template: new sap.ui.commons.TextField()}));
75  oTable.addColumn(new sap.ui.table.Column({ label: new sap.ui.commons.Label({text: "Name"}), template: new sap.ui.commons.TextField()}));
76  oTable.addColumn(new sap.ui.table.Column({ label: new sap.ui.commons.Label({text: "Sanity"}), template: new sap.ui.commons.TextField()}));
77  oTable.addColumn(new sap.ui.table.Column({ label: new sap.ui.commons.Label({text: "Color"}), template: new sap.ui.commons.TextField()}));

```

Figure 1.19 Splitting the Source Code Editor

Such improvements apply to all programming languages used by Eclipse. Because ABAP is now a member of that happy family, you can take advantage of such improvements the instant they come out without waiting for an upgrade of your backend system.

1.3 Testing and Troubleshooting

All throughout this book, you will see a recurring theme: new ABAP tools allow you to better test your programs, debug them, and analyze what went horribly wrong. This section examines each of these three areas in turn and discusses what ABAP in Eclipse brings to the table.

1.3.1 Unit Testing Code Coverage

In Chapter 3, you will learn about unit testing in ABAP and test-driven development as a philosophy in some detail. Rather than getting ahead of things and discussing why unit tests are a good thing (they are; they are the best thing in the world), this section will instead talk about how ABAP in Eclipse confronts two of the main problems with writing unit tests.

Problem 1: Creating Test Methods is Difficult

As you'll see, in SE80 there is a wizard to create test methods from your real methods. However, that's the wrong way around—you want to create the test methods first.

To do this in Eclipse, create a new global class. At the bottom is a series of tabs, one of which is TEST CLASS. Note that you have not created anything in the real class yet. Now, go to the TEST CLASS tab, type in the word "test", and press `CTRL` + `SPACE`.

You are asked a question about whether you want to create a transport request or a test class. The test class is the correct answer, and if you choose that option, you'll see the content of Listing 1.13.

```

*** use this source file for your ABAP unit test classes
CLASS ltc1_definition FINAL FOR TESTING
  DURATION SHORT
  RISK LEVEL HARMLESS.

PRIVATE SECTION.
METHODS:
  first_test FOR TESTING RAISING cx_static_check.
ENDCLASS.

CLASS ltc1_IMPLEMENTATION.
METHOD first_test.
  cl_abap_unit_assert=>fail( 'Implement your first test here' ).
ENDMETHOD.
ENDCLASS.

```

Listing 1.13 Source File for ABAP Unit Test Class

Creating such a skeleton template for a test class isn't the holy grail of creating unit tests, but it follows test-driven development much better than the standard SE80 process. Naturally, you need to change the generated code to fill in the name of your test class and test method and add extra test methods as needed. The point is that you are creating the test methods *first* and then copying the definition to the real class.

As you saw in Section 1.2.3, once you have a definition you can generate the implementation skeleton just by pressing `CTRL` + `1`. What the automatic generation is supposed to do is add a definition, such as `DATA: mo_cut TYPE REF TO yc1_monster_unit_tests.` (This actually did not work for me no matter what I tried, so I added it manually. The documentation indicates that this should be done automatically, so it will probably be fixed in a later release.)

In the renamed `first monster` unit test, first define the test result, and then add the line `mo_cut->first_monster().`, which is a method in the main class that

does not yet exist. If you press `CTRL` + `1`, you are asked if you want to create the method in the real class, which you do. Remember, you created the test before you created the real method, which is what test-driven development is all about.

In summary, the whole test-driven development process is faster and easier (and the right way around) when using ABAP in Eclipse than when using the equivalent procedure in the traditional ABAP Workbench.

Problem 2: You Cannot Tell How Much of the Program Is Being Tested

The aim of unit testing is to be as sure as humanly possible that when you change one part of your program—be it a bug fix or adding extra functionality—the change does not break other parts of the program. In real life, people tend to find that a change in one area *always* breaks something in another area, which should be a red flag to indicate that maybe there's something wrong with creating such fragile programs.

Unit tests enable you to perform automated regression tests so that when you change even one line of code you can be sure that you have not broken something somewhere else. Naturally, you can only be really safe if you know that every single line of code in every routine and method in your application is subject to such a regression test. Creating such tests in the first place is not the easiest thing in the world, which is why so few people do it—this will be discussed much more in Chapter 3. However, if you have seen the light and do wish to create such tests, the next task is to see how much of your code is actually getting tested. If that figure turns out to be less than 100%, then see what can be done about it.

There is a tool available in Java called Clover, which measures the percentage of code covered by unit tests. When I read about it, I thought it would be really good for ABAP and that maybe I should write something along those lines myself. It turns out that I don't need to, because ABAP in Eclipse has just such a tool (and, to be fair, so does SE80 in the latest releases of SAP NetWeaver).

In the last section, you created one test method. Now, you'll create the real method, which will be tested, and then you'll create another real method, which will not have a corresponding unit test—as if you're being naughty and trying to fool yourself into thinking that creating methods without tests will save time. (They! Will! Not!)

They say that little things please little minds, but I take great pleasure in being able to write the method definition for the method that will not be tested, pressing `CTRL` + `1` and having the implementation created for me, and having the cursor jump into the implementation. Another cliché says that every second counts, and if these tools save a few seconds here and there all throughout your working day, then over time that does all add up.

```
CLASS ycl_monster_unit_tests IMPLEMENTATION.
```

```
  METHOD first_monster.
    WRITE:/ 'I am the First Monster'.
  ENDMETHOD.
```

```
  METHOD second_monster.
    WRITE:/ 'I am the Second Monster'.
  ENDMETHOD.
```

```
ENDCLASS. "Monster Unit Tests Implementation
```

Listing 1.14 The Method that Will Not Be Tested

If you were in SE80, you would now follow the menu path `PROGRAM • TEST • UNIT TEST`; the equivalent in ABAP in Eclipse is to press `CTRL` + `SHIFT` + `F10`. That's wonderful; you'll see any errors that may have cropped up as a result of any changes you might have made. However, you want to take this to the next level, so instead press `CTRL` + `SHIFT` + `F11`. (Eleven is obviously a better number than 10, in the same way that they started calling hotels six-star hotels because five stars weren't good enough anymore.)

Now, you'll see two results (Figure 1.20). First, all the code that was tested is highlighted; more importantly, you'll see a summary view, which indicates what percentage of each method was tested by the automated unit tests.

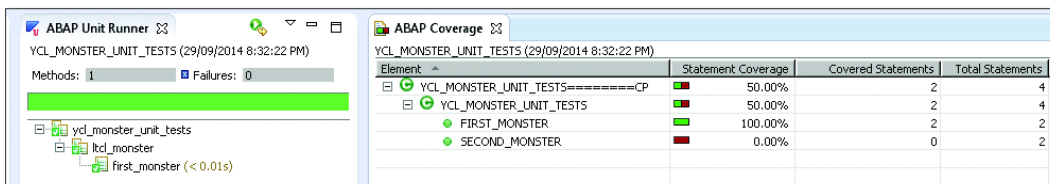


Figure 1.20 Unit Test Coverage

This is the sort of information you want shoved in your face like a custard pie, first because you want to make sure all the vital parts of your program are covered

by regression tests and second to move toward making sure that every single line of code gets test coverage.

1.3.2 Debugging

One of the best features in the ABAP Workbench is the debugger, and Chapter 5 will take a quick look at the latest features that have crept into the debugger recently that you may be unaware of. For the moment, though, the focus is on ABAP in Eclipse, so you may be wondering what relevance debugging has in a pure development environment. This comes back to unit tests again: if you have an error, then you want to debug the program to see what is going wrong.

There is bad news and good (though weird) news here. The bad news is that, as of the time of writing, the debugger in ABAP in Eclipse cannot do everything that the debugger in a real SAP system can do, though of course as time goes by those gaps are being plugged. Now, the good news—though when I tell you what the good news is you might think “That can’t be true, he must be on drugs”—the good news is that it is possible to change source code *while it is in the process of being debugged*.

In the next example, you will put an obvious error in your code and then debug it to see what’s wrong. In the UK, we use the phrase “how many beans make five,” to which the answer is, shockingly, five. In your program, you want to add monsters until you get five, but you’ve missed one and so only end up with four, which is a clear error (Listing 1.15).

```
CLASS lcl_how_many_monsters DEFINITION.
  PUBLIC SECTION.
    METHODS how_many_make_five RETURNING VALUE(rd_how_many) TYPE i.
ENDCLASS. "How Many Monsters Definition
```

```
CLASS lcl_how_many_monsters IMPLEMENTATION.
  METHOD how_many_make_five.
    DO 100 TIMES.
      ADD 1 TO rd_how_many.
      IF rd_how_many = 4.
        RETURN.
      ENDIF.
    ENDDO.
  ENDMETHOD.
ENDCLASS. "How Many Monsters implementation
```

```
DATA: ld_how_many TYPE i,
```

```

    lo_counter TYPE REF TO lcl_how_many_monsters.

START-OF-SELECTION.
  CREATE OBJECT lo_counter.
  ld_how_many = lo_counter->how_many_make_five( ).
  WRITE:/ ld_how_many.

```

Listing 1.15 Only Four Monsters

This causes your unit test to fail, so you debug it. While you're in the debugger, it becomes obvious what's wrong: you need to add another line of code. At this point, normally you would exit the debugger, change your program, and then test or debug it again after the correction.

In ABAP in Eclipse, however, it is in fact possible to change the source code while it's being debugged; that is, you can add the missing line to add the final monster while in the debugger and then step through the new line to see if everything works as expected. This is a bit of a radical change, and when you first see this working you wonder if someone has slipped some LSD into your drink. Nonetheless, this falls into the “strange but true” category—and as soon as SAP has brought the rest of the debugger in ABAP in Eclipse up to par with the standard debugger, this facility will be a clear point in ABAP in Eclipse's favor.

Just like in the normal ABAP editor, you can control where the program will switch to debugger mode when the program is run. You can add soft breakpoints if you so desire, via the menu or **CTRL** + **SHIFT** + **B**; you can also put in hard breakpoints, such as `BREAK BLOGGSJ`. When you run the program (menu option `RUN • RUN AS ABAP APPLICATION`) or execute the unit tests, as soon as a breakpoint is reached a box will appear and ask if you want to look at the program in debug mode. As stupid questions go, this takes the cake; *obviously* you do, so you say yes and tick the box that makes the system remember this wonderful decision you made.

The debugger screen opens up; it looks slightly different from what you have been used to in the past (Figure 1.21). It's possibly slightly easier to change the values of variables at runtime (double-click in the box in which the current value is displayed to change a value).

Warning: Houston, We Have a Problem

If your backend SAP system is not on a high enough level—that is, the kernel has to be at least 721—you will get an error message saying something like “ABAP in Eclipse debugging is not available in system XYZ, please debug inside the SAP GUI.”

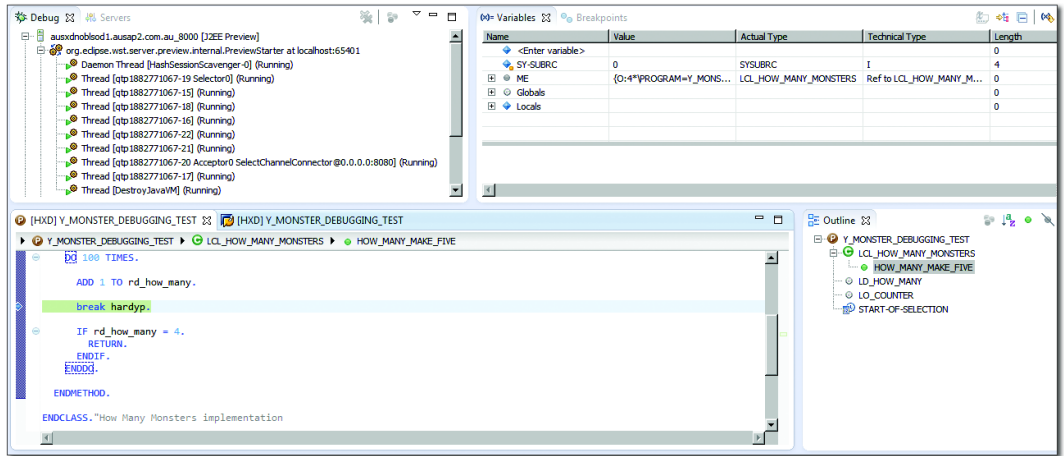


Figure 1.21 Debugging in Eclipse

You can see the error; the code exits with a value of 4, not 5. Change the statement to say `IF RD_HOW_MANY = 5`. This has no effect on the current execution (that would be black magic), but when you're finished and you run the program again, everything is OK; the fix has taken effect.

The results appear in what is called the "embedded SAP GUI," which is a standard SAP screen that appears in an Eclipse window (Figure 1.22). You will see this often when working with Eclipse; whenever Eclipse cannot display something (such as the results of a `WRITE` statement) it will outsource the task to the SAP GUI.

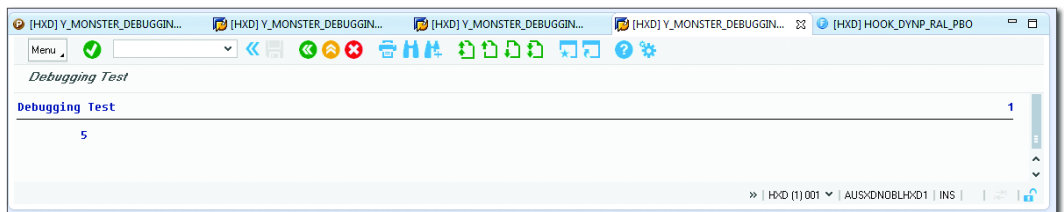


Figure 1.22 Embedded SAP GUI

1.3.3 Runtime Analysis

Of all the great features discussed to this point, what really stands out about ABAP in Eclipse is the graphical view of the runtime analysis of an ABAP program. The famous saying notes that "a picture is worth a thousand words," but traditionally

SAP has not been known for the quality of the graphics within the SAP GUI. You are probably used to the runtime analysis within the ABAP Workbench—Transaction SE30 or more recently SAT—which gives detailed information about how a given application spends its processing time. The problem is that there is just so much information that you tend to get swamped. You'll see a nice set of three bars indicating whether the most time is spent in the database or in the application server, and if it turns out that the server is the problem, then you have to try and get to grips with the big tree that indicates how much time is spent in each routine.

That is not the end of the world, but it takes some getting used to, and the equivalent in ABAP in Eclipse is much friendlier to the eye. To demonstrate this, up next is a silly example in which a small fraction of the program performs mathematical calculations, and the bulk of the program performs `SELECT` statements within a loop. (In fact, the example isn't as silly as all that, because you do tend to find this situation all too often in real-life programs.)

Say that every time a monster learns that there is such a thing as American Airlines, the monster instantly wants to go on a flight with them. Thus, Listing 1.16 combines the popular `SFLIGHT` example with a monster example.

```
DO 100 TIMES.

  SELECT COUNT( * )
    FROM sflight
   WHERE carrid = 'AA'.

  CHECK sy-subrc = 0.

  ADD 1 TO gd_flying_monsters.

ENDDO.
```

Listing 1.16 `SFLIGHT` and Monster Mash-Up

This example is only good for showing what not to do; obviously, we only need to read the database once. To invoke the runtime analysis, select your ABAP program, follow the menu path `PROFILE AS • ABAP APPLICATION`, and then type "trace" into the top-right-hand corner of the screen, where it says `QUICK ACCESS`.

You'll see the trace file as per usual. When you expand it and click on it, instead of bars saying how much time was spent in the database and in the server, you'll see a pie chart showing the exact same thing. That probably does not impress you too much, but as a next step click the `CALL TIMELINE` tab at the bottom. Then,

you'll see a screen in which time will be the axis at the top of the screen, and you'll see a graphical representation of what the program is spending its time doing. This is, let's agree, much easier for the brain of a human to follow.

Change the code such that you only have one database access, and then repeat the exercise. This time, the pie chart shows that you're spending the bulk of your time in the ABAP environment, which is where you want to be.

1.4 Customization Options with User-Defined Plug-Ins

In this section, you'll learn about the general enhancement concept within Eclipse and then look at some specific examples relating to object-oriented programming. (I am a big advocate of object-oriented programming in its place, a fact which will become apparent through this book.)

A big part of Eclipse is its software development kit (SDK), which is meant to allow people to create plug-ins that can be installed in the base Eclipse system to enable extra functionality. All of the ABAP development tools for Eclipse are plug-ins that you manually install. This is why Eclipse can be used as a development platform for so many languages.

Eclipse is an open-source project—which means that, if there is something missing, the idea is not to just sit there whining about it but to create the missing tool yourself and then make it publicly available as a plug-in. This sort of altruistic behavior puzzles the hell out of a lot of people (you're giving away your work for free?). However, in practice this process works really well, and it results in products that evolve at the speed of light.

Creating Your Own Plug-In

What makes user-defined plug-ins so useful is that the SAP system can expose a tool written in ABAP as a URL. Frameworks like Eclipse can then send a request to the SAP system by means of a URL, which is then processed by an ABAP handling class you nominate in Transaction SICF. (In Chapter 13, you'll see that this is precisely how SAPUI5 applications work.) Creating your own plug-in is not a trivial exercise, and it is beyond the scope of this book to go into detail, but you need to be aware that this can be done. (If you're interested in pursuing this further, SAP has published a how-to guide for this purpose; see the "Recommended Reading" box at the end of the chapter.) The focus of this section is not to explain how to create your own plug-ins, but to give you practical examples of what can be achieved with such user-defined plug-ins.

One example of the value of open-source is the publicly available SAPlink plug-in, which was developed by a group of SAP Mentors, and which is a tool for moving development objects between SAP systems via a download/upload process. You'll need the SAPlink plug-in to follow the content in this section. To add it, choose the menu option **HELP • INSTALL NEW SOFTWARE**, enter the URL <http://eclipse.sap-link.org>, and download the components listed. The result can be seen in Figure 1.23: An extra menu item appears in the standard menu. Normally, you have to run SAPlink as a standalone program via SE38 or a transaction code, but now it has been embedded in the development environment.

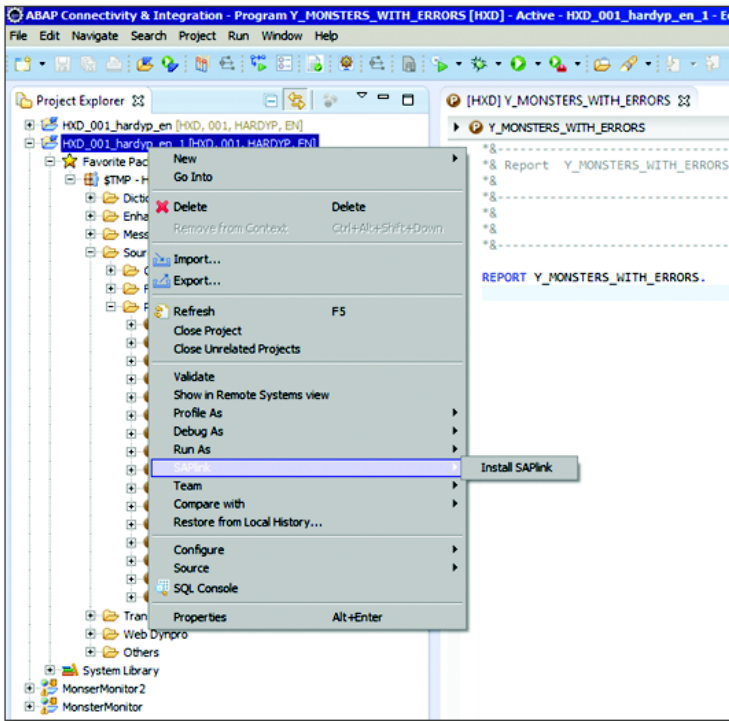


Figure 1.23 Extra Menu Item via the SAPlink Plug-In

The nature of the extra menu items changes depending on what you are doing; if you have a program or other object open, then you get an option to export the object to your local drive as a “nugget,” for example.

Object-oriented programmers often say that when you create a program, the first step is to create a UML (Universal Modeling Language) diagram that describes

how various classes will interact with each other. Does this ring a bell? If so, it may be because you know about a facility inside SE80 through which you can take an existing ABAP program and generate a hideously ugly diagram that is supposed to be a UML model of your application. Actually, though, that's the wrong way around. What you would want is the reverse: to take a UML diagram and use that to create the classes, attributes, methods, and so forth inside SAP (though naturally you still have to write the actual code inside the methods yourself!).

As it turns out, there are two such plug-ins—UMAP and Obeo—that both deal with creating ABAP constructs from UML diagrams. UMAP was created by a gentleman named Mathias Märker and is publicly available at no charge. Obeo is from a French company (also called Obeo), and is a product that you have to pay for. There is nothing like a bit of healthy competition to keep the IT world buzzing, so this section will take a quick look at both products.

1.4.1 UMAP

UMAP is an open-source project created with the hope of getting other people involved; it's a prototype just starting out in life. UMAP uses some of the freely available technology that the aforementioned Obeo provides, and the examples in this section do the same. (Obeo is discussed in more detail in Section 1.4.2.)

To see what UMAP can do, the first step is to go to the website www.uml2abap.org and download the UMAP Code Generator, an SAPlink nugget that creates some classes and an upload program in your ABAP system. In this case, you have to download a file from Eclipse and then upload it again into SAP. The download file contains two files. The first is an SAPlink nugget that you turn into ABAP classes and programs via report ZSAPLINK. The second is a text file, which you need to turn into an XML transformation by going into Transaction XLST_TOOL, creating a transformation called ZUMAP_FILE_PARSER, and then pasting in the code from the text file.

The website talks about also downloading an Eclipse plug-in, but that is a red herring. In actual fact, you need to open up Eclipse, choose the HELP • INSTALL NEW SOFTWARE option, and enter the URL http://uml2abap.org/eclipse_plugin. The screen shown in Figure 1.24 opens.

Select the UMAP box and then click NEXT to install the plug-in. The next step in the process is to create your UML diagram. As mentioned earlier, you need some

sort of modeling tool installed in your Eclipse environment. I chose to install the UML designer from Obeo (because it was free) by choosing **HELP • ECLIPSE MARKETPLACE** and searching for the Obeo plug-in. After your modeling tool is installed, return to the main Eclipse screen, and choose **FILE • NEW • OTHER • UML DESIGNER • UML PROJECT**.

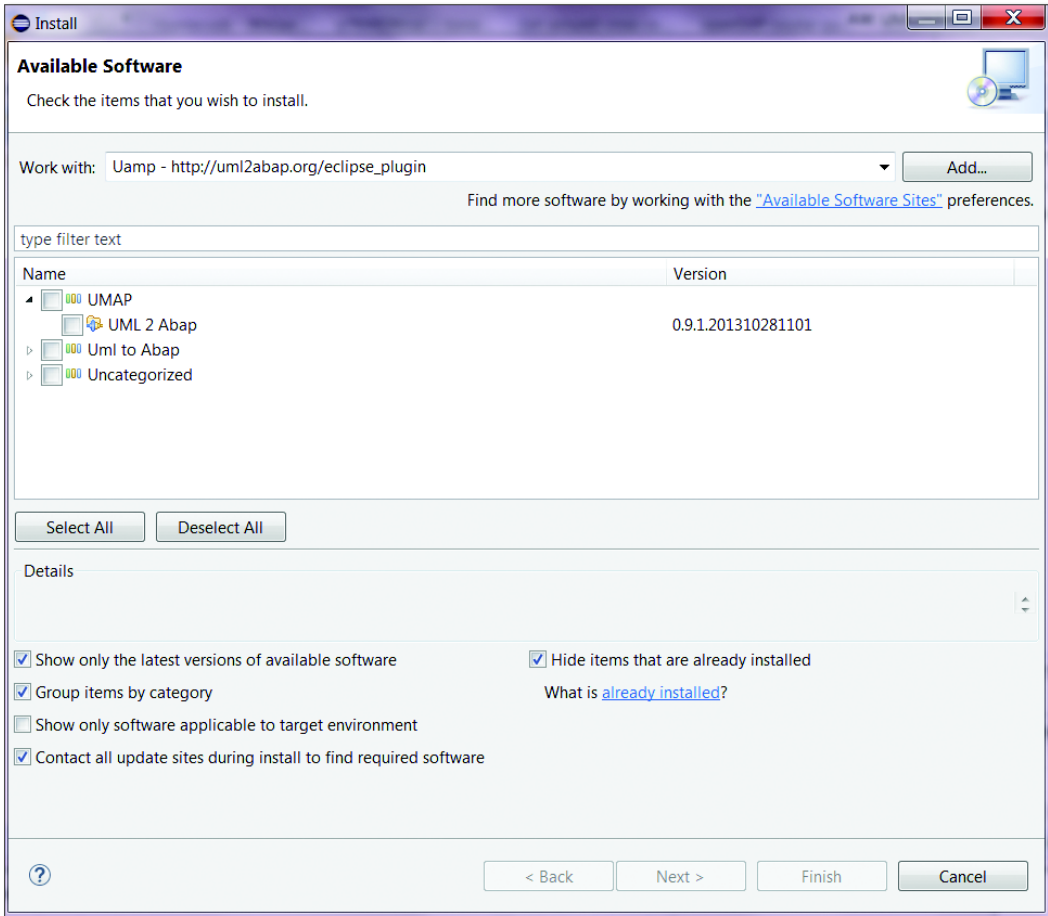


Figure 1.24 Installing the UML2ABAP Eclipse Plug-In

Call the project "ReallyScaryMonster." When asked for a model object, take the default option model. (At this point, for me, everything vanished. Just in case this happens to you, you should be able to fix it by opening a new window and selecting **SHOW VIEW • MODEL EXPLORER**.)

Right-click the REALLYSCARYMONSTER project, and choose CREATE REPRESENTATION; you'll see the screen shown in Figure 1.25.

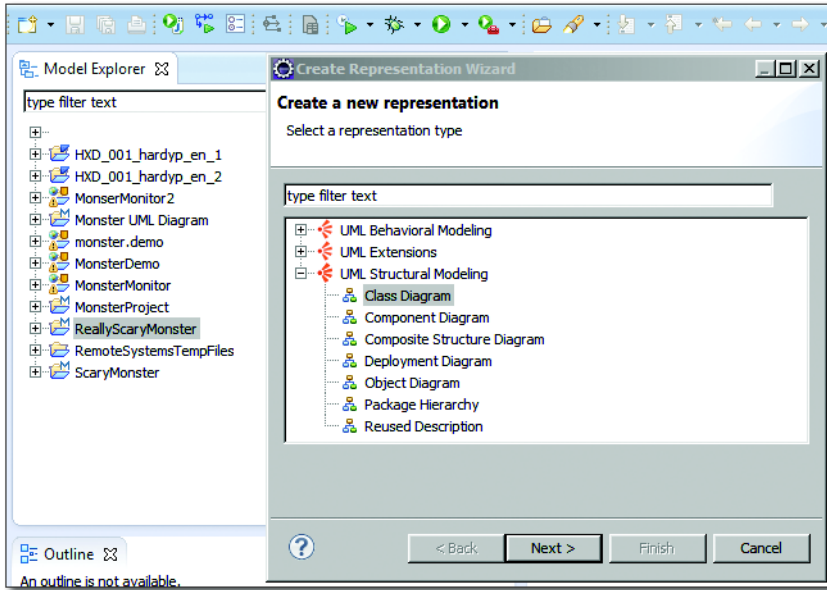


Figure 1.25 Creating a Class Diagram

After pressing the NEXT button, you'll see a screen with three sections (Figure 1.26). In the middle is the area where you'll create your class diagram, on the right are all the things you can drag onto the screen, and at the bottom are the properties of whatever the cursor is on at any given instant, which you can change. (In Chapter 12, you will see that the graphical editor in Web Dynpro ABAP is just like this.)

Create a base class with two subclasses that inherit from the base class. Give all three some properties; in the subclass, make some of the green and blue monsters' attributes protected or private, and give the main class one method (called an "operation" in UML speak). You can do all of this by dragging things around the screen; when you place your cursor on anything (such as an attribute), you will see a list of properties at the bottom of the screen.

Select the REALLY_SCARY_MONSTER Model.UML node, right-click it, and choose UML 2 ABAP • GENERATE ABAP CODE. (Reminder: this menu option would not have been there if you hadn't installed the UML2ABAP plug-in!)

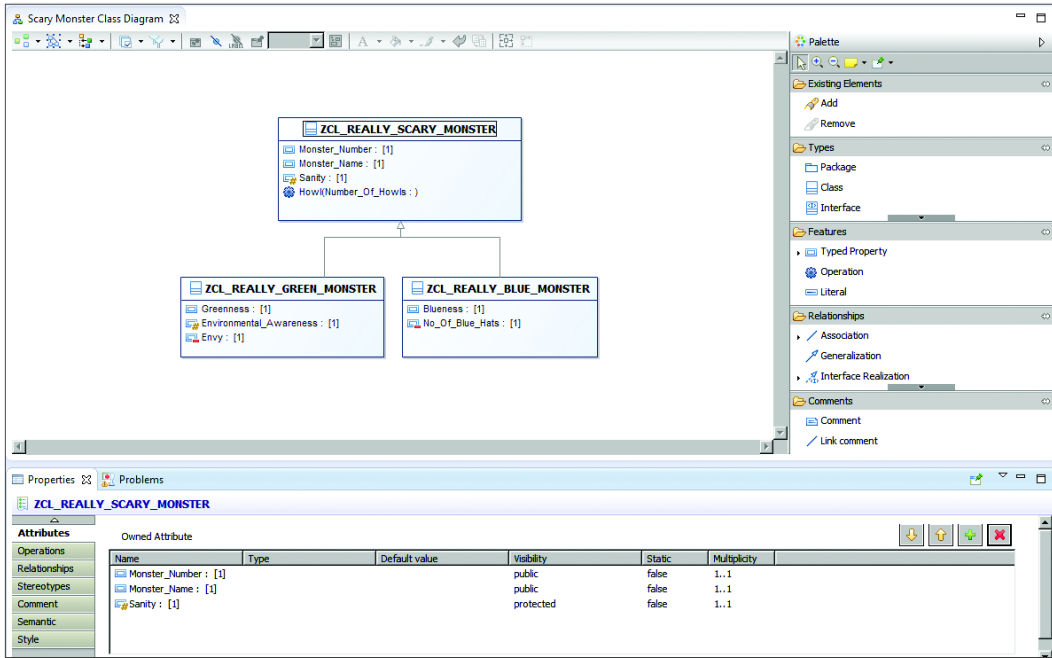


Figure 1.26 Monster Diagram in Eclipse

The hourglass whirls, and a box titled ACCLEO GENERATION RESULTS appears for a few seconds. The Accleo framework is also a creation of Obeo. The important thing is that now in your REALLYSCARYMONSTER project you have a new node, called SRC-GEN; this has a child node called `NewModel.umap`. You need to export this `NewModel.umap` file to your local machine; there's nothing difficult here—just right-click on the file and choose EXPORT and then a folder on your local drive. Then, run the `Z_UMAP_IMPORT` program in your ABAP system, and nominate the file you just saved.

The screen shown in Figure 1.27 appears; click IMPORT OBJECT. Next, you're asked what packages the new classes should live in (or if they are local objects). If all is well, there will be some green lights indicating that the new classes have been created at the bottom of the screen.

If, like me, you distrust everything a computer tells you and have a burning need to check everything yourself, then go into the actual ABAP system via the normal logon pad to make sure the classes are there in SE24; indeed they are. With that

worry off your mind, go back into Eclipse to have a look at the generated code (Listing 1.17). Remember that in Eclipse everything is a source code–based view of what you might see in SE24 or SE37 in a form-based view.

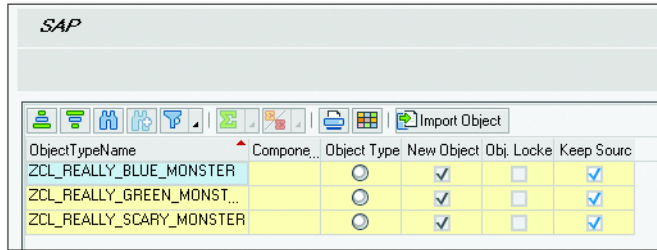


Figure 1.27 Importing a UML Diagram to the ABAP System

```

class ZCL_REALLY_SCARY_MONSTER definition
  public
  create public .

public section.
  data MONSTER_NAME type INVALID .
  data MONSTER_NUMBER type INVALID .

  methods HOWL
    importing
      !NUMBER_OF_HOWLS type INVALID .
protected section.

  data SANITY type INVALID .
private section.
ENDCLASS.

CLASS ZCL_REALLY_SCARY_MONSTER IMPLEMENTATION.

method HOWL.
endmethod.

ENDCLASS.

```

Listing 1.17 Generated Code in Eclipse

Sure enough, Eclipse has successfully generated the code for you, including the subclasses (Listing 1.18).

Note

As of yet, there is no way of getting the correct TYPES for attributes, and the parameters of the method did not copy over. This is to be expected; this is a very rudimentary prototype, just a proof of concept at the moment. That is what open-source projects are all about: If something is missing, then you can add it yourself.

```
class ZCL_REALLY_GREEN_MONSTER definition
  public
  inheriting from ZCL_REALLY_SCARY_MONSTER
  create public .

public section.

  data GREENNESS type INVALID .
protected section.

  data ENVIRONMENTAL_AWARENESS type INVALID .
private section.

  data ENVY type INVALID .
ENDCLASS.

CLASS ZCL_REALLY_GREEN_MONSTER IMPLEMENTATION.
ENDCLASS.
class ZCL_REALLY_BLUE_MONSTER definition
  public
  inheriting from ZCL_REALLY_SCARY_MONSTER
  create public .

public section.
  data BLUENESS type INVALID .
protected section.
private section.
  data NO_OF_BLUE_HATS type INVALID .
ENDCLASS.

CLASS ZCL_REALLY_BLUE_MONSTER IMPLEMENTATION.
ENDCLASS.
```

Listing 1.18 Generated Subclasses

1.4.2 Obeo

The Obeo UML plug-in is just a prototype at the time of writing, not yet released to the public. It's an Eclipse plug-in designed to achieve two goals: first to let you

design UML diagrams inside Eclipse, and then to turn those UML diagrams into ABAP programs.

The prototype Obeo has provided looks like a specialized version of Eclipse, and is called the Obeo Design Studio. Both the Obeo product and UMAP do the same sort of thing, but the following list highlights their differences:

- ▶ You do not have to have anything installed on the backend ABAP system with the Obeo product; it generates the code straight into the ABAP system by using a menu option in ABAP in Eclipse. No download and upload is needed.
- ▶ In the UMAP product, the `TYPES` come up as `INVALID`; in the Obeo version, the `TYPES` are blank, and you have to fill them out yourself, which isn't the end of the world.

As mentioned, as of the time of writing this is just a prototype and not yet released. When the plug-in is released, Obeo will charge for it.

Hopefully, this section has given you an idea of what sort of plug-ins are possible for Eclipse that could make your programming life easier; maybe you will even be motivated to read the how-to document mentioned at the end of this chapter and create your own tool.

1.5 Summary

This chapter covered the basics of ABAP in Eclipse or, as SAP likes to call it, ABAP Development Tools. After reading it, you should have an understanding of what you can do with Eclipse and how it can do things that SE80 can't.

Because ABAP in Eclipse enables programming in ABAP, it somehow needs to make sure that the syntax check is right and that code completion works properly—that is, that it knows everything there is to know about the language. This naturally leads to the next chapter, which is all to do with the somewhat staggering enhancements that have been applied to the ABAP language in SAP NetWeaver 7.4.

Recommended Reading

- ▶ SAP NetWeaver How-To Guide: SDK for the ABAP Development Tools: <http://scn.sap.com/docs/DOC-40668> (Wolfgang Wöhrle)

- ▶ UML2ABAP—Code Generation for ABAP:
<http://www.youtube.com/watch?v=RfFStA2s8OA> (YouTube)
- ▶ UMAP—UML2ABAP Presentation:
<http://www.youtube.com/watch?v=53SsEd6spcs> (YouTube)

It's a beautiful thing, the Destruction of words. Of course the great wastage is in the verbs and adjectives, but there are hundreds of nouns that can be got rid of as well.
—George Orwell, 1984

2 New Language Features in ABAP 7.4

Back in the early eighties, when I was still a teenager, my parents bought me a BBC Microcomputer to replace my ZX81. Even then, my primary interest was programming as opposed to playing games, and what I liked most about my new toy was the broad range of commands that were then available to me in the BBC's version of BASIC (for example, constructs such as `REPEAT UNTIL`). Even now, many years later, I look on with interest when something is added to the ABAP language.

As time has gone by, more and more commands and constructs have been added to ABAP—while nothing has been taken away, to ensure backward compatibility—and the rate of change seems to be violently accelerating. A fair number of changes came in as a result of the introduction of SAP NetWeaver 7.02, which this chapter will touch on, but this is nothing compared to the cascade of changes that came with version 7.4. This proves beyond a doubt that ABAP is not a dead language, which was a fear of many people in 2001, when there was talk (from the very top of SAP) about replacing ABAP with Java.

The quote at the beginning of the chapter from George Orwell's novel 1984 is about the destruction of words. The character working on the language "Newspeak" says to the main character, Winston Smith, that the ruling party is hell-bent on destroying existing words, as opposed to creating new ones. The idea of ABAP 7.4 is, in some ways, the same: Many of the 7.4 changes allow you to achieve the exact same tasks as before, but with half the code or less. For example, in Newspeak "that was wonderful, fantastic, the best thing ever" becomes "++good." In ABAP, `CONCATANATE ld_this ld_that INTO ld_the_other SEPARATED BY '_'` becomes `ld_the_other = ld_this && '_' && ld_that`.

Of course, it is not all about destroying words—release 7.4 also brings a lot of brand new functionality as well. This chapter will focus on the large number of new features that have been introduced into ABAP in release 7.4. These features are divided into the following categories, based on their general area of functionality: database access, creating data, string processing, calling functions, conditional logic, internal tables, object-oriented programming, search helps, unit testing, and cross-program communication.

2.1 Database Access

As you know, programmers tend to spend an inordinate amount of time accessing the database when developing ABAP programs. This section will explain the recent changes to ABAP that affect this task—hopefully making things a bit less painful.

2.1.1 New Commands in OpenSQL

You will most likely be aware that there are two sorts of SQL queries allowed in ABAP. One of them is OpenSQL, which you use the vast majority of the time (see Listing 2.1).

```
SELECT monster_number monster_name
FROM zt_monsters
INTO CORRESPONDING FIELDS OF lt_monsters
WHERE monster_number = ld_monster_number.
```

Listing 2.1 OpenSQL

OpenSQL is a subset of every command that can be used by all the databases that SAP supports (rather like the area in the middle of a Venn diagram). In other words, in OpenSQL the statements you can attach to a `SELECT` in ABAP, such as `ORDER BY`, `INNER JOIN`, `SELECT DISTINCT`, are only a subset of what is available from any given database vendor—that is, the list of possible options that could be added after a `SELECT` statement mentioned in the documentation from that vendor.

The other sort of SQL query is NativeSQL, which is a query written between the `EXEC` and `ENDEXEC` commands. NativeSQL lets you write any query commands supported by the database installed in your system. For example, in Figure 2.1, you'll see the Microsoft `SELECT` syntax options, as an example of what can be done in that company's implementation of NativeSQL.

```

SELECT
[ALL | DISTINCT | DISTINCTROW ]
[HIGH_PRIORITY]
[STRAIGHT_JOIN]
[SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
[SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
select_expr [, select_expr ...]
[FROM table_references
[WHERE where_condition]
[GROUP BY {col_name | expr | position}
[ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_condition]
[ORDER BY {col_name | expr | position}
[ASC | DESC], ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
[PROCEDURE procedure_name(argument_list)]
[INTO OUTFILE 'file_name' export_options
| INTO DUMPFILE 'file_name'
| INTO var_name [, var_name]]
[FOR UPDATE | LOCK IN SHARE MODE]]

```

Figure 2.1 Microsoft SQL Syntax

Because it provides more options, NativeSQL is more powerful. However, if your organization ever decides to perform a database migration, then you are in trouble; you will almost certainly have to rewrite any NativeSQL queries you have coded. Even though database migrations are not a walk in the park, they do happen every so often, and in order to avoid such potential extra work you will find people like me: I have not written one NativeSQL query in the 17 years I have been programming in ABAP.

The good news is that in release 7.4, SAP has expanded the range of options available to use within ABAP programs when you are running an OpenSQL query. You may not have `STRAIGHT JOINS` and `SQL_BIG_RESULTS`, as in the Microsoft SQL syntax in Figure 2.1, but there are some new useful things, and the next sections will take a look at the two most important: `CASE` statements embedded into SQL queries and performing calculations inside SQL queries.

CASE Statements Inserted into SQL Queries

One of the new features of ABAP 7.4 is the ability to insert `CASE` statements into SQL queries. Listing 2.2 shows an example of this. In this example there is a field in an internal table called `SCARINESS_STRING`, and it should be filled with a string describing how scary the monster is based on the values in the database columns `SANITY` and `STRENGTH`.

```

CONSTANTS: lc_scary1 TYPE STRING VALUE 'SLIGHTLY SCARY',
           lc_scary2 TYPE STRING VALUE 'REALLY SCARY',
           lc_scary3 TYPE STRING VALUE 'NORMAL'.

SELECT monster_name, monster_number
  CASE
    WHEN sanity <= 10 AND strength >= 75 THEN @lc_scary2
    WHEN sanity <= 25 AND strength >= 50 THEN @lc_scary1
    ELSE @lc_scary3
  END AS scariness_string
FROM zt_monsters
WHERE monster_number = @ld_monster_number
INTO CORRESPONDING FIELDS OF @lt_monsters.

```

Listing 2.2 CASE Statement within an SQL Statement

You will notice that you have to put an @ symbol in front of your variables (or constants) when using the fancy new features, such as CASE, in order to let the compiler know you are not talking about a field in the database. You also have to put commas between the fields you are bringing back from the database and put the INTO statement at the end, just like in the Microsoft diagram you saw in Figure 2.1. The @ and the commas and so on are as a result of a new “strict” syntax check that comes into force when the compiler notices you are using one of the new features. In other words, if you try and use a new feature like a CASE statement in an SQL query but do not put an @ symbol beside the variable name, then you will get a hard error. In this way, SAP can force you to program in a slightly different way when using new features while still being backward compatible.

That’s all well and good, but what is the CASE statement doing there in the first place? Is it lost? Surely you use CASE statements after you have the data back from the database? Basically, what the CASE statement has allowed you to do is to out-source the job of calculating some conditional logic to the database—instead of performing these calculations on the application server. As you’ll see in Chapter 15, pushing some of this sort of thing into the SAP HANA database to be processed can result in huge performance improvements (if done correctly).

Calculations within SQL Statements

Another functionality new to release 7.4 is the ability to perform arithmetical operations inside of SQL statements. In earlier releases, you had to get the data from the database first and then play with it. As an example of how it worked before, say that you want to divide strength by sanity to get a scariness ratio: the

lower the sanity, the higher the ratio. Listing 2.3 shows what would have been done prior to ABAP 7.4.

```
"Before
DATA: ld_converted_strength TYPE fltp,
      ld_converted_sanity  TYPE fltp.
SELECT monster_name monster_number strength sanity
FROM zt_monsters
INTO CORRESPONDING FIELDS OF lt_monsters.
LOOP AT lt_monsters ASSIGNING <ls_monsters>.
  ld_converted_strength = <ls_monsters>-strength.
  ld_converted_sanity  = <ls_monsters>-sanity.
  <ls_monsters>-scary_ratio =
  ld_converted_strength / ld_converted_sanity.
ENDLOOP.
```

Listing 2.3 Operations Inside SQL Statements: Before

However, as of ABAP 7.4, it's now possible to perform these operations within the SQL query, as shown in Listing 2.4.

```
SELECT monster_name, monster_number
  CAST( strength AS fltp ) / CAST( sanity AS fltp )
  AS scary_ratio
FROM zt_monsters
WHERE monster_number = @ld_monster_number
INTO CORRESPONDING FIELDS OF @lt_monsters.
```

Listing 2.4 Operations Inside SQL Statements: After

As you can see, the example also included some type conversions. This saves you from declaring two helper variables and more importantly from having to loop over the internal table performing a calculation on each line.

Moreover, if you're not interested in the strength and sanity on their own and only care about the ratio, you can have two fewer fields in your internal table (which you would have to expend extra effort to declare as "technical" in an ALV report), and less information will be sent back from the database to the application server (the ratio as opposed to strength and sanity). The less often such data gets sent, the faster the program runs.

2.1.2 Buffering Improvements

One of the tasks I most enjoy is going through the ST05 trace for a program and finding out that by making some minor changes I can take advantage of the table

buffering in table DDIC to dramatically reduce the database access, thus making the program run much faster and cheering up my users. Although buffering has been around for a long time, there have also been some problems with it; mainly, that there are a whole raft of situations in which the buffer is bypassed and you don't get any performance improvement at all. (You can mitigate this problem by using the Code Inspector. If you run a Code Inspector check on your program after creation and after every subsequent change—which you should always do—you get alerted to any such database reads you may have coded in the expectation that the buffer would be used.)

To demonstrate a common situation where the buffer is bypassed, consider an example where half of your queries get data from the table `ZT_MONSTERS` using the `MONSTER_NUMBER` (which is the primary key), and the other half get data using the `MONSTER_NAME` (on which you have an index). Assume that this is a master data table that is relatively small, read often, and rarely changed—so naturally you've defined it as fully buffered in the technical settings.

If you perform a SQL query using `MONSTER_NUMBER`, then no database access happens at all (after the first read after the database is started), because you have specified the primary key in your query. However, if you perform a `SELECT SINGLE` using `MONSTER_NAME`, then the Code Inspector warns you that table buffering will not be used, because you're not specifying the primary key.

People used to get around this by reading the whole table from the database into an internal table and then sorting it by `MONSTER_NAME`. Then, whenever someone wanted to find a monster by name, he'd perform a `BINARY SEARCH` to get the desired record. Before 7.02, people sometimes used to have two (or more) internal tables, sorted differently based on different keys. As of 7.02, you can have secondary indexes on internal tables, and so you only needed one table. However, that's still one table too many from a memory-consumption perspective.

As of release 7.4, this is no longer a problem. If you perform a `SELECT SINGLE` on `MONSTER_NAME` and then run an `ST05` performance trace, you will not see any database access at all, even though you didn't specify the primary key. This is because, as of 7.4, if you've defined a secondary index on `MONSTER_NUMBER` in table `ZT_MONSTERS`, then that index is magically taken into account as well. Secondary indexes are taken into account not only on fully buffered tables but also on generically buffered tables, which is a giant leap forward performance wise. Buffers taking

account of secondary indexes could be described as a *passive* improvement; if you had `SELECT` statements on an indexed field that previously bypassed the buffer, then as of release 7.4 your code will speed up without you having to lift a finger—and a lot of standard SAP code will speed up as well.

Another passive example of buffering improvements in ABAP 7.4 is that up until now you could not perform a `FOR ALL ENTRIES` on a buffered table, even if you used the primary key, without having to go to the database; you had to do a loop of `SELECT SINGLES` and then get warned about doing `SELECTS` in a loop by the extended syntax check. This is now a thing of the past. Because the data is all in memory, the query treats the `FOR ALL ENTRIES` as a loop of `SELECT SINGLES`, and thus database access is totally avoided. As an example, a lot of standard SAP reports do a `FOR ALL ENTRIES` on buffered table `SETLEAF`, which relates to the profit center hierarchy. Under a 7.4 environment, all such standard programs will run much faster.

2.1.3 Creating while Reading

In Chapter 10 of this book, you will learn about `CL_SALV_TABLE` and how it is clever enough to look at the definition of your internal table and turn that definition into an ALV grid without you having to manually define each column again. ABAP 7.4 has taken this one step further by doing away with the need to perform the data declaration for the internal table in the first place.

To illustrate what this means, look at the following example. Say that you want a list of all the monsters named Fred. Traditionally, you would go about this as shown in Listing 2.5.

```
TYPES: BEGIN OF l_typ_monsters,
        monster_name TYPE zt_monsters-monster_name,
        monster_number TYPE zt_monsters-monster_number,
        sanity TYPE zt_monsters-sanity,
      END OF l_typ_monsters.

DATA: lt_monsters TYPE STANDARD TABLE OF l_typ_monsters.

SELECT monster_name monster_number sanity
  FROM zt_monsters
 INTO CORRESPONDING FIELDS OF TABLE lt_monsters
 WHERE monster_name = 'FRED'.
```

Listing 2.5 List of All Monsters Named Fred

Tip

Note that the code includes a `TYPE` declaration. To save time, some programmers would just define the internal table as being of the data dictionary type. However, this is a bit of a waste of memory.

In older versions of ABAP, if you declared a `TYPE` and then suddenly wanted to retrieve an extra field, then you would need to make the change in two places: in the definition and in the `SELECT` statement. You might forget to make one of those changes—I know I have—and although the syntax check will warn you of this it might slip through the net.

In 7.4, however, you can not only skip the `TYPE` definition but the internal table declaration as well. An example of this is shown in Listing 2.6.

```
SELECT monster_name monster_number sanity
  FROM zt_monsters
 WHERE monster_name = 'FRED'
 INTO TABLE DATA(lt_monsters).
```

Listing 2.6 Defining an Internal Table Based on the SQL Query

As you can see, there is no need to declare either a `TYPE` or an internal table. The table is created on the spot at the instant the database is read, and the format of the table is taken from the types of the data fields you are retrieving. This way, there is no way that two lists of fields can get out of sync, because there is only one list.

This new technique also works for structures if you are doing a `SELECT SINGLE` on multiple database fields and for elementary data elements if you are doing a `SELECT SINGLE` on just one database field. Furthermore, if you use a construct such as `KUNNR AS SHIP_TO` in your select statement, then the name of the column in your internal table will be `ship_to`. (This sort of thing is useful, because you might be doing a join, for example, and `KUNNR` might have different meanings in different tables: the customer in `VBAK` and the ship-to in `LIKP`.)

2.1.4 Inner Join Improvements

If you've ever coded before (and if you're reading this book, I guess you have!), then you know that it's fairly common to need to read every single field of one database table and also one or two fields from another related table. The most

efficient way to do this is by way of an inner join. However, it is somewhat tedious to list all of the fields from the main table, as you can see in Listing 2.7.

```
SELECT zt_monsters~monster_name zt_monsters~monster_number
      zt_monsters~sanity      zt_monsters~strength
      zt_monsters~hat_size    zt_monsters~number_of_heads
      zt_hats~bar_code
FROM zt_monsters
INNER JOIN zt_hats
ON zt_monsters~hat_size = zt_hats~hat_size
INTO CORRESPONDING FIELDS OF TABLE lt_monsters
WHERE monster_name = 'FRED'.
```

Listing 2.7 All Fields from Main Table

The preceding code reads everything from the monster table but just one minor detail from the hat table. In 7.4, you can achieve the same thing in a different way, as shown in Listing 2.8.

```
SELECT zt_monsters~*
      zt_hats~bar_code
FROM zt_monsters
INNER JOIN zt_hats
ON zt_monsters~hat_size = zt_hats~hat_size
WHERE monster_name = 'FRED'
INTO DATA(lt_monsters).
```

Listing 2.8 Reading Everything from One Table during an Inner Join

The key here is the asterisk. It acts just like the wild card in `SELECT *` and as such can be abused if you really want not all the fields of the table, but just half of them.

Warning: Houston, We Have a Problem

As always, using asterisks in such cases is the easy way out, but it slows down database performance and—when you are creating the table at the same time—memory consumption as well. This just proves that the more tools are at one's disposal, the greater the damage you can do with them if they are not used properly.

In addition to the asterisk functionality in ABAP 7.4, the syntax of inner joins has also been extended so that now you have much more flexibility in defining the relationship between the tables. An example of this is shown in Listing 2.9.

```
INNER JOIN zt_monster_pets
ON zt_monster_pets~owner EQ zt_monsters~monster_number
```

```
AND zt_monster_pets~type LIKE 'GIGANTIC%'
AND zt_monster_pets~species IN ('SHARK','CROCODILE','DINOSAUR')
```

Listing 2.9 Joining Two Tables in Ways Not Possible Before

If you want a table of all the pets of various monsters along with some details about the owning monsters, but you're only interested in gigantic dinosaurs and the like, then the query shown in Listing 2.9 is the one for you.

2.2 Declaring and Creating Variables

One of the main differences between ABAP and other languages is that in ABAP we have been taught to declare all our variables at the start of the program—for example, in a `TOP INCLUDE`. Other languages declare them just before they are used, and such variables tend to be more local in scope—for example, within a loop. Despite the official ABAP programming guidelines, many ABAP programmers have taken to declaring variables just before they are used for the first time, for purposes of making the program more readable by humans and hence easier to change.

The good news is that as time goes on, changes in the ABAP language make this unofficial practice more and more acceptable. This section will discuss several features that contribute to this shift.

2.2.1 Omitting the Declaration of TYPE POOL Statements

7.02 Feature

This feature came about with ABAP 7.02, but is discussed here because it's little known.

Many programs use the ever-popular `ABAP_TRUE` and `ABAP_FALSE` constants, and even more use the built-in icon constants to make ALV reports more visually appealing to users. This used to involve declaring a `TYPE POOL` statement in the `TOP INCLUDE`, or at the start of a program, or in the forward declaration property of the global class. An example of this is shown in Listing 2.10.

```
TYPE-POOLS: abap,
            icon.
```

Listing 2.10 Declaring a TYPE POOL

Traditionally, if you created a huge application with locally defined internal tables and structures and constants and the like and then suddenly realized that you wanted to reuse a lot of these somewhere else, then using a `TYPE POOL` was the only way to create such reusable constructs. If you didn't declare a `TYPE POOL`, then the compiler would complain that your `ABAP_TRUE` statement was unknown. Even though the introduction of Z table types in ABAP 6.40 removed part of the need for `TYPE POOLS`, they were still used, because there was no other way to have a globally available constant like `ABAP_TRUE`. However, as of release 7.02, this is no longer the case; you can use `ABAP_TRUE` anywhere you want, and the compiler will no longer complain. In other words, all existing `TYPE POOL` statements are now global.

However, there's a catch: you're no longer supposed to create any new `TYPE POOL` statements. You may be thinking to yourself, "Why shouldn't I create `TYPE POOLS` any more just because SAP says so?" That's a valid question. The answer is that `TYPE POOLS` were created to address a specific gap in functionality—a gap which no longer exists. This is like taking a picture with a Kodak camera or sending a fax; you could still do these things if you wanted to, but there is no need to do so any longer.

As it turns out, there is now a solution that deals with both constants and internal table definitions that have not been created in the form of Z table types: public type definitions of global Z classes. An example of no longer having to create a Z table type is when the global class has a public attribute that is a custom table type defined within the Z class, and the calling program wants to call a method that needs that table type passed in. The calling program can do something like the example in Listing 2.11.

```
DATA: lt_monsters TYPE zcl_monsters=>mtt_monster_table.
zcl_monsters->make_monsters_dance( lt_monsters ).
```

Listing 2.11 Using a Custom Table Type Declared within a Z Class

This has the advantage that there is now no need to bloat the repository with Z table types. It is exactly the same with constants; if they are a public attribute of a class, then they can be used anywhere in the system. (In fact, there are standard SAP classes that contain nothing except constant definitions; you could call them the younger brothers of `TYPE POOLS`.)

2.2.2 Omitting Data Type Declarations

Another reason ABAP 7.4 has made it less important to declare variables at the start of your routine or method is that the need for the vast majority of data declarations has melted away. The compiler knows what data type it wants (it has to know to be able to perform a syntax check), so why not let it decide the data type of your variable? In the examples ahead, you will see how letting the compiler decide the data type—instead of declaring it yourself—can save you several lines of code. For example, here is a piece of code declaring a data type:

```
DATA: ld_monster_instructions TYPE string.
      ld_monster_instructions = 'Jump up and down and howl'.
```

Without the data type declaration, you can simplify this to just the following:

```
DATA(ld_monster_instructions) = 'Jump up and down and howl'.
```

Similarly, the following piece of code also declares a data type:

```
DATA: ld_number_of_monsters TYPE i.
      ld_number_of_monsters = LINES( lt_monsters )
```

And, again, you can simplify this to:

```
DATA(ld_number_of_monsters) = LINES( lt_monsters ).
```

As you can see, this has the potential to dramatically reduce the number of lines in your programs. You'll see various examples of such inline declarations throughout this chapter, because they have applications in many different areas of ABAP programming.

2.2.3 Creating Objects Using NEW

In the programming language Java, the command `NEW` is used to create instances of objects (they say that everything in Java is an object). For example, `MonsterFred = NEW(Monster)` in Java creates an instance called `Fred` of the class `Monster`. In the same way that the ABAP runtime environment has been shamelessly stealing features from Eclipse, now the ABAP language is stealing keywords from Java—so you now have a `NEW` command as well.

Instead of

```
DATA: lo_monster TYPE REF TO zcl_monster.
      CREATE OBJECT lo_monster EXPORTING name = 'FRED'.
```

you'll use

```
lo_monster = NEW zcl_monster( name = 'FRED' ).
```

and instead of

```
DATA: ld_sanity TYPE zde_monster_sanity.
ld_sanity = 0.
```

you'll use

```
ld_sanity = NEW zde_monster_sanity( 0 ).
```

As can be seen, this halves the number of lines of code you need. More importantly, the debate about whether you declare the variables at the start of a routine (per the official ABAP guidelines) or just before the variable (in order to aid humans who might be reading the code) is no longer relevant. You have the type of the variable right in your face at the instant the variable is created.

2.2.4 Filling Structures and Internal Tables while Creating Them Using VALUE

No doubt, you are familiar with the common statement

```
DATA: ld_monster_name TYPE string VALUE 'FRED'.
```

In this statement, you create a variable and give it an initial value, which can then later be changed. So far, the ability to change initial values has only been available for elementary data types. However, as of 7.4 the `VALUE` statement has come of age, and you can do the same for structures and internal tables.

Listing 2.12 contains a pre-7.4 example of querying a database. In this example, you create a selection table to be used in an SQL query, but you're only interested in laboratories where monsters will be created. However, because you can't use the `VALUE` statement, you have to fill up one or more work areas, and then append them to the selection table.

```
DATA: lr_plant_type TYPE
      RANGE OF zsmm_plant_master-plant_type,
      ls_plant_type LIKE LINE OF lr_plant_type.
ls_plant_type-option = 'EQ'.
ls_plant_type-sign = 'I'.
ls_plant_type-low = 'L'. "Laboratory
APPEND ls_plant_type TO lr_plant_type.
lt_plants = lo_plant_query( lr_plant_type ).
```

Listing 2.12 Database Query without VALUE

However, in version 7.4, you can use the `VALUE` statement and thus achieve the same effect with fewer lines of code (Listing 2.13).

```
TYPES: lt_typ_plant_type TYPE RANGE OF zsmm_plant_master.
lt_plants = lo_plant_query(
VALUE lt_typ_plant_type(
option = 'EQ'
sign   = 'I'
low    = 'L' ) ) ). "Laboratory
```

Listing 2.13 Database Query with `VALUE`

You could add more than one line to the query table if you wanted—for example, if you were interested in retrieving graveyard data as well. You've avoided the need for an intermediate internal table (`LR_PLANT_TYPE`) and avoided the work area to build up the lines of that intermediate table.

2.2.5 Filling Internal Tables from Other Tables Using `FOR`

How well I remember starting to program when I was 14, with the good old ZX81. The BASIC language I programmed in then had constructs like `FOR x = 1 TO 10`, which meant you were going to loop ten times, with the variable `X` increasing by one each time. Well, the `FOR` command has now arrived in the ABAP world; take a look at what it is for (if you'll forgive the pun).

You read about the `VALUE` statement in the last section; you can use this to fill an internal table, as in Listing 2.14.

```
DATA: lt_monsters TYPE STANDARD TABLE OF ztvc_monsters.
lt_monsters = VALUE#(
( monster_name = 'FRED' monster_number = 1 )
( monster_name = 'HUBERT' monster_number = 2 ) ).
```

Listing 2.14 Fill Internal Table

That's great as an example, but in real life you either fill internal tables from the database or from other internal tables; almost never do you fill them with hard-coded values. Prior to 7.4, you could only fill one internal table from another table if the two tables had identical column structures, and you had to add all the lines of one table to another:

```
APPEND LINES OF lt_green_monsters TO lt_all_monsters.
```

Fortunately, thanks to the `FOR` command introduced in ABAP 7.4, you can now do this in a much more elegant way: the tables can have different columns, and you can limit what gets transferred based upon conditional logic. Using the `VALUE` and `FOR` keywords, this will look like Listing 2.15.

```
DATA( lt_neurotic_monsters ) = VALUE lt_neurotic_monsters(
FOR ls_monsters IN lt_all_monsters WHERE (sanity < 20 )
monster_name = ls_monsters-monster_name
monster_number = ls_monsters-monster_number ) ).
```

Listing 2.15 Filling Internal Tables from Other Tables

2.2.6 Creating Short-Lived Variables Using LET

ABAP 7.4 also allows you to declare variables with short lifespans; this is done with the `LET` statement. These variables only exist while creating data using constructor expressions. (A constructor expression is a mechanism to apply assorted logic when creating a variable, such as an internal table. This is similar to the sort of logic you would find inside a constructor of an OO class. `VALUE` is an example of a constructor expression.)

To illustrate the use of the `LET` statements, say that you have a table of deadly weapons, and you want to arm some monsters. You're not particularly concerned about which monster gets which weapon, as long as each monster gets one. Listing 2.16 shows an example of code that creates some very short-lived local variables.

```
DO LINES( lt_monsters[] ) TIMES.
DATA( ld_weapon_description ) = VALUE string(
LET weapon_name = lo_iterator->get_next_weapon( )
monster_name = lt_monsters[ sy-index ]-monster_name
date_string =
|{ sy-datum+6(2) } / { sy-datum+4(2) } / { sy-
datum(4) }| IN |Monster { monster_name } was issued a { weapon_name }
on { date_string } ).
WRITE:/ ld_weapon_description.
ENDDO.
```

Listing 2.16 Creating Short-Lived Variables

Note

In real life, you would store the description in some sort of internal table. However, the focus here is the `LET` statement.

In this example, the first variable just gets the description of the next weapon in a list of weapons via a functional method call. Then, you read a line of the internal table of monsters—the syntax here may puzzle you, but it will be covered later in Section 2.6—and finally you fill a string variable with the system date, formatted in a human-friendly manner.

You can now use those variables you've just declared to fill up the result string. Thanks to the `LET` statement, after the statement is over those variables (`weapon_name`, etc.) don't exist any longer, as opposed to being accessible from anywhere inside the routine like regular variables.

Field Symbols

You can also declare field symbols as well as variables. The type of the variable or field symbol is dynamically determined by looking at the value that is being passed into it.

2.3 String Processing

The English comedy trio The Goodies once sang:

*"String, string, string, string, everybody loves string,
String for your pants, string for your vest!
Everybody knows—string is best!"*

Some fairly major changes to string processing were introduced with release 7.02 and 7.4 of ABAP; they are discussed next.

2.3.1 New String Features in Release 7.02

You can replace the `CONCATENATE` statement with something like `LD_THIS = LD_THAT && LD_THE_OTHER`. This definitely saves a bit of room in your code, but it does not seem to handle the spaces you often want between variables very well. Much better are the "pipes" and "curly brackets," which came in with release 7.02. Instead of the code shown in Listing 2.17, you can simplify to the code shown in Listing 2.18.

```
CONCATENATE 'Monster Number' LD_NUMBER INTO LD_HELPER SEPARATED BY SPACE.
CONCATENATE LD_HELPER LD_STATUS INTO LD_RESULT
SEPARATED BY ' / '.
```

Listing 2.17 Building Up a String Using `CONCATENATE`


```
LD_RESULT = |Monster Number { ld_number } / { ld_status }|.
```

Listing 2.18 Building Up a String Using Pipes

This gives you much better control of spaces and punctuation and the like; it's slightly more difficult to use text symbols but not impossible. Moreover, you can pass such constructs (the text between the starting | and the ending |) into parameters of method calls that are expecting strings. Previously, you had to build up the string in a helper variable and then pass that into the method call.

2.3.2 New String Features in Release 7.4

If you're anything like me, you may spend half your life calling the functions `CONVERSION_EXIT_ALPHA_INPUT` and `CONVERSION_EXIT_ALPHA_OUTPUT` to add and remove leading zeroes from document numbers like delivery numbers; for example, you might remove the zeroes when showing messages to the user, but then then add them back before you read the database (Listing 2.19).

```
“Remove leading zeroes before output to user
CALL FUNCTION ‘CONVERSION_EXIT_ALPHA_OUTPUT’
  EXPORTING in = ld_delivery_number
  IMPORTING out = ld_delivery_number.
ld_message = |Problem with delivery number { ld_delivery_number }|.
MESSAGE ld_message TYPE ‘I’.
“Now add the leading zeroes back before database read
CALL FUNCTION ‘CONVERSION_EXIT_ALPHA_INPUT’
  EXPORTING in = ld_delivery_number
  IMPORTING out = ld_delivery_number.
SELECT *
  FROM LIKP
  INTO CORRESPONDING FIELDS OF ls_delivery_header
  WHERE vbeln = ld_delivery_number.
```

Listing 2.19 Removing and Adding Leading Zeroes by a Function Call

In ABAP 7.4, this can be done in a more compact manner by using the `ALPHA` formatting option, which does the exact same thing as the two function modules used in Listing 2.19. To remove the leading zeroes in 7.4, you can write code such as that in Listing 2.20.

```
ld_message = |{ ld_delivery ALPHA = OUT }|.
MESSAGE ld_message TYPE ‘I’.
SELECT *
  FROM LIKP
```

```

INTO CORRESPONDING FIELDS OF ls_delivery_header
WHERE vbeln = ld_delivery_number.

```

Listing 2.20 Removing Leading Zeroes via a Formatting Option

When using the method shown in Listing 2.20, you no longer have to add the leading zeroes back—as they were never actually removed from the delivery variable in the first place.

2.4 Calling Functions

Often, you have to jump through hoops to play around with the variables in your programs before you can call other methods or functions. Luckily for you, this only gets easier over time. This section will discuss the new ABAP functionalities that make calling functions much easier than it has been in the past.

2.4.1 Method Chaining

7.02 Feature

This feature came about with ABAP 7.02, but is discussed here because it's little known.

All through this chapter, you have been reading about getting rid of data declarations. Now, you'll turn to the case in which you have dozens of helper variables, all storing intermediate steps, and you store the result of one method call for a short time solely for the purpose of passing that value into another method call. In traditional programming terms, such variables are called “tramp data,” because they are only there to pass on values from one place to another, as in the song “that's why the data is a tramp.”

With the advent of release 7.02, a new concept called *method chaining* was born, whereby you can directly pass the result of one method into the input parameter of another method without the need for a helper variable. Normally, you would declare a helper variable, fill it with the result of a method call, and pass that helper variable into another method, as in Listing 2.21.

```

CATCH zcx_excel INTO lo_exception.
ld_helper = lo_exception->get_text( ).
zcl_bc_screen_message=>output( ld_helper ).

```

Listing 2.21 Helper Variable

However, now you can do away with having to declare the helper variable by chaining the two method calls together, as in Listing 2.22.

```
CATCH zcx_excel INTO lo_exception.
zcl_bc_screen_message=>output(
id_text = lo_exception->get_text( ) ).
```

Listing 2.22 Method Chaining

In this simple example, that may not seem like much of a saving, but the benefit increases proportionately to the number of related method calls that are made one after the other.

2.4.2 Avoiding Type Mismatch Dumps when Calling Functions

It can drive you up the wall when you have declared a variable that you want to either pass into or get back from a method of function module, and the variable type does not match the expected parameter type. With a method, you get a syntax error; with a function module, you get a short dump at runtime. To be safe, you have to keep jumping in and out of the function module signature to see how the variables were typed. This process is tedious, so it has led to people creating custom *patterns*, in which you enter a function module name, the signature is read, and then a whole list of variables are declared with the same type as all the function parameters. Well, half of the problem has now gone away, because if the sole purpose of a variable is to receive a value from a method or function, then the variable can be declared inline and the type read from the parameter definition.

To demonstrate this new functionality, take a look at Listing 2.23. This is an example of what most people have always done: you declare some local variables to be passed into a method, cross your fingers, and hope you have declared the variable the same as the input parameter. If not, you get an error.

```
DATA: ld_number_of_heads TYPE i,
      ld_number_of_hats TYPE i.
lo_monster=>get_ahead_get_a_hat(
EXPORTING id_monster_number = ld_monster_number
IMPORTING ed_number_of_heads = ld_number_of_heads
          ed_number_of_hats = ld_number_of_hats ).
```

Listing 2.23 Declaring Variables with (Hopefully) the Same Type as Method Parameters

With ABAP 7.4, however, you can accomplish the same thing by declaring the variables returned from the method not at the start of the routine but rather at the instant they have their values filled by the method, as shown in Listing 2.24.

```

lo_monster=>get_ahead_get_a_hat(
EXPORTING id_monster_number = ld_monster_number
IMPORT  ed_number_of_heads = DATA( ld_number_of_heads )
        ed_number_of_hats = DATA( ld_number_of_hats ) ).

```

Listing 2.24 Using Inline Declarations to Avoid Possible Type Mismatches

This approach has several advantages:

- ▶ There are fewer lines of code.
- ▶ You cannot possibly get a type mismatch error or dump.
- ▶ If you change the signature definition, then the result variable adapts itself accordingly.

Therefore, this approach is more compact and hopefully easier to read (and thus maintain), safer, more resistant to change, and all-in-all less fragile.

This approach comes into its own when creating object instances using factory methods; the factory will return a different subclass of the base object depending on assorted logic, but the calling program should not care. For example, the pre-7.4 code in Listing 2.25 would become the code shown in Listing 2.26.

```

DATA: lo_monster TYPE REF TO zcl_green_monster.
lo_monster = zcl_laboratory=>build_new_monster( ).

```

Listing 2.25 Calling a Factory Method for a Specific Class

```

DATA( lo_monster ) = zcl_laboratory=>build_new_monster( ).

```

Listing 2.26 Letting the Factory Method Decide the Exact Subclass

Later in this book, you'll see that in frameworks such as Web Dynpro (Chapter 12) and the Business Object Processing Framework (Chapter 8), you're always creating lots of objects with complicated data types. Not having to declare really complicated variable types before the call to fill up such variables with values cleans up a lot of the boilerplate code and allows you to concentrate on what's really important.

2.4.3 Using Constructor Operators to Convert Strings

Many times, one routine consists largely of a call to several smaller routines (such as FORM routines, function modules, and methods). The problem with this is that

sometimes the result of one routine has to have its type converted before it can be passed into another routine (e.g., the period from a standard SAP ERP Financials Financial Accounting function tends to be two characters long, but if you want to pass that period into a Controlling function, then it needs to be three characters long).

Another even more common example is that often you have a variable that is a string, and you want to pass that variable into a function that only accepts input of a certain type (say CHAR20). You cannot pass the variable directly; you have to move it into a helper variable. This is shown in Listing 2.27.

```
DATA: ld_helper TYPE CHAR20,
      ld_monster_name TYPE string.
ld_monster_name = 'HUBERT'.
ld_helper = ld_monster_name.
lo_monster->invite_to_party( ld_helper ).
```

Listing 2.27 Moving a Variable into a Helper Variable

In ABAP 7.4, however, this can be simplified by use of a specific type of constructor operator, CONV, the job of which is to convert values from one type to another. In Listing 2.28, the CONV function reads the target data type from the IMPORTING parameter definition and then converts the string into the type the parameter is expecting. Once again, this is shorter, safer, and more adaptive.

```
DATA: ld_monster_name TYPE string.
ld_monster_name = 'HUBERT'.
lo_monster->invite_to_party( CONV#( ld_monster_name ) ).
```

Listing 2.28 Converting a String with a Constructor Variable

2.4.4 Functions That Expect TYPE REF TO DATA

When you import parameters of functions or methods, a specific type of parameter is usually required. However, in some situations you do not know the variable type until runtime. In such cases, the only way to achieve what you want is to use dynamic programming.

In cases where you don't know the exact data type until runtime, often the importing parameter of the method is typed as TYPE REF TO DATA. This is what happens, for example, when you call methods that build up a dynamic signature to pass into a dynamically defined method. You may have also used parameters

declared `TYPE REF TO DATA` when you needed to store an arbitrary number of values of different data types in a log along with their descriptions.

In pre-7.4 ABAP, you would do something along the lines of the code in Listing 2.29 in order to satisfy the requirement of the `LOG_VALUE` method, which expects to receive a parameter of the `TYPE REF TO DATA` type.

```
DATA: lo_do_value TYPE REF TO DATA.
"VALUE is an IMPORTING parameter TYPE ANY
IF value IS SUPPLIED.
    CREATE DATA lo_do_value LIKE value.
    GET REFERENCE OF value INTO lo_do_value.
    lo_monster_log->log_value( lo_do_value ).
ENDIF.
```

Listing 2.29 Filling a `TYPE REF TO DATA` Parameter before 7.4

However, now you can get rid of most of the code and still achieve the same thing by using the constructor operator `REF` and the `#` symbol. Because the runtime system knows the data type of `VALUE`, the `REF#` function can read this and create the so-called data object, which is then passed in the logging method, which expects a `TYPE REF TO DATA` object to be passed in. As shown in Listing 2.30, this is slightly simpler.

```
"VALUE is an IMPORTING parameter TYPE ANY
IF value IS SUPPLIED.
    lo_monster_log->log_value( REF#( value ) ).
ENDIF.
```

Listing 2.30 Using Constructor Operator `REF` to Fill a `TYPE REF TO DATA` Parameter

2.5 Conditional Logic

By now, it will not come as a shock that I am again going to mention the ZX81. The language it used back in 1981 was able to handle statements like `IF (A + B) > (C + D) THEN ...`. Later on, in 1999, when I started programming in ABAP, I missed this ability. Luckily, with the advent of ABAP 7.02, which I first had access to in 2012, I was once again able to do this sort of thing. (It might have taken 31 years for them to work out how to do this, but it was worth the wait.)

In ABAP 7.40, the `IF/THEN` and `CASE` constructs you know and love keep on getting easier. This section will explain how.

2.5.1 Using Functional Methods in Logical Expressions

7.02 Feature

This feature came about with ABAP 7.02, but is discussed here because it's little known.

One of the great features of ABAP 7.02 was the concept of functional methods and how you could use them in logical expressions. Functional methods allow you to eliminate the use of a helper variable. Thus, instead of the code shown in Listing 2.31, you can simplify with the code shown in Listing 2.32.

```
LD_HELPER = STRLEN( LD_STRING ).
IF LD_HELPER > 10 THEN...
```

Listing 2.31 Helper Variable

```
IF STRLEN( LD_STRING ) > 10 THEN...
```

Listing 2.32 No Helper Variable

Moreover, the same principle applies to return variables from method calls, so you can have statements like this:

```
CHECK zcl_bc_system_environment=>is_production( ) = abap_false.
```

This looks somewhat like the method chaining we talked about in Section 2.4.1. The differences here are that (a) a lot of functional methods supplied from SAP (like `STRLEN`) are built in, as opposed to being real ABAP methods, and (b) functional methods do one thing only—they return a result—whereas the methods you chained together in Section 2.4.1 can do pretty much anything.

2.5.2 Omitting ABAP_TRUE

When functional methods were introduced to ABAP, part of the idea was that this would make the code read a bit more like English. Over the years, the general consensus among ABAP programmers both inside and outside of SAP was that if you were creating a method that returned a value saying whether or not something is true, then the returning parameters should be typed as `ABAP_BOOL`.

For example, the `ZCL_MONSTER->IS_SCARY` method should return `ABAP_TRUE` if the monster is in fact scary but `ABAP_FALSE` if it's not quite as monstrous as it should be. So far, so good. However, as Listing 2.33 shows, something is rotten in the state of Denmark.

```

IF zcl_bc_system_environment=>is_production( ) = abap_true.
    "In production we never want a short dump, but the "design by
    "contract things would just confuse the user
TRY.
    lcl_application=>main( ).
    CATCH cx_sy_no_handler INTO go_no_handler.

```

Listing 2.33 ABAP_TRUE

Why do you need the `= ABAP_TRUE` at the end? It doesn't make the sentence any more readable, just longer. As any English teacher will tell you, adding words that do not change the meaning to a sentence only makes you sound long winded. The answer is that you had to do this, because otherwise the syntax check would fail.

As of release 7.4 (SP 8), however, you can now do just what you would expect: omit `ABAP_TRUE`. This is shown in Listing 2.34.

```

IF zcl_bc_system_environment=>is_production( ).
    "In production we never want a short dump, but the "design by
    "contract things would just confuse the user
TRY.
    lcl_application=>main( ).
    CATCH cx_sy_no_handler INTO go_no_handler.

```

Listing 2.34 Omitting ABAP_TRUE

In Listing 2.34, what is happening from a technical point of view is that if you do not specify anything after a functional method the compiler evaluates it as `IS_PRODUCTION() IS NOT INITIAL`. An `ABAP_TRUE` value is really the letter X, so the result is not initial, and so the statement is resolved as true.

Opinion is divided as to whether it is a Good Thing to have a true Boolean data type in a programming language. SAP says no, and the creators of every other programming language ever invented in the history of the universe say yes. This is why there are workarounds like this in ABAP.

That being said, you have to be really careful though when using this syntax; it only makes sense when the functional method is passing back a parameter typed as `ABAP_BOOL`. As an example, consider the code in Listing 2.35.

```

IF user_wants_to_blow_up_world( ).
    lo_massive_atom_bomb->explode( ).
ENDIF.

```

Listing 2.35 IF Statement without a Proper Check.

If the functional method in Listing 2.35 returns a string and that string is NO NO! Do not blow up the world, whatever you do, do not blow up the world, then the result is NOT INITIAL and thus evaluated as true, and so it is curtains for all of us.

2.5.3 Using XSDBOOL as a Workaround for BOOLC

Another common situation with respect to Boolean logic (or the lack thereof) within ABAP is a case in which you want to send a TRUE/FALSE value to a method or get such a value back from a functional method. In ABAP, you cannot just say something like the following:

```
RF_IS_A_MONSTER = ( LD_STRENGTH > 100 AND LD_SANITY < 20 )
```

Although, in some programming languages you can do precisely that (can you guess which computer could do that in 1981?). Again, we have a workaround in the form of the built-in function `BOOLC` (Listing 2.36).

```
* Postconditions
zcl_dbc=>ensure( id_that = 'A result table is returned'(005)
               if_true = boolc( et_return[] IS NOT INITIAL ) ).
```

Listing 2.36 `BOOLC`

In Listing 2.36, you pass in a TRUE/FALSE value based on whether an internal table has any entries. This works fine.

However, what if you want to test for a negative using this method? Say, for example, that you want to pass in to parameter `IF_TRUE` a TRUE/FALSE value that's true if the table is empty. If you use the previous technique using `BOOLC`, then things start going horribly wrong. This can be demonstrated by running the code in Listing 2.37.

```
DATA: lt_empty TYPE STANDARD TABLE OF ztvc_monsters.
```

```
IF boolc( lt_empty[] IS NOT INITIAL ) = abap_false.
  WRITE:/ 'This table is empty'.
ELSE.
  WRITE:/ 'This table is as full as full can be'.
ENDIF.
```

```
IF boolc( 1 = 2 ) = abap_false.
  WRITE:/ '1 does not equal 2'.
ELSE.
```

```
WRITE:/ '1 equals 2, and the world is made of snow'.
ENDIF.
```

Listing 2.37 Testing for a Negative

When you run this code, the output is as follows:

1. This table is as full as full can be.
2. 1 equals 2, and the world is made of snow.

Oh, dear! The reason for this is a fundamental design flaw in the built-in function `BOOLC`. Instead of returning a one-character field defined in the same way as `ABAP_BOOL`, back comes a string. If the string is an X (`TRUE`), then all is well, but in `ABAP` comparing a string of one blank character with the blank character inside `ABAP_FALSE` means that the comparison fails, even though the values are identical.

Therefore, given that a real Boolean variable is out of the question for whatever reason, a new workaround is needed. Fixing `BOOLC` so that it returns an `ABAP_BOOL` value would have been too easy, so in `ABAP 7.4` a newly created built-in function was added, called `XSDBOOL`, which does the same thing as `BOOLC` but returns an `ABAP_BOOL` type parameter. The function was not invented for this purpose, but it works, and that is all that matters.

2.5.4 The SWITCH Statement as a Replacement for CASE

How many times have you seen code like Listing 2.38? Here, you're getting the day of the week and using a `CASE` statement to turn the number into a string, such as `Monday`, to output at the top of the screen. The problem is that you need to keep mentioning what variable you're filling in every branch of your `CASE` statement.

```
* using the date get the day of the week.
data: l_indicator like scal-indicator,
      l_day(10) type char01.

call function 'DATE_COMPUTE_DAY'
  exporting
    date = p_date
  importing
    day = l_indicator.

case l_indicator.
  when 1.
    l_day = 'Monday'(326).
  when 2.
```

```

    l_day = 'Tuesday'(327).
when 3.
    l_day = 'Wednesday'(328).
when 4.
    l_day = 'Thursday'(329).
when 5.
    l_day = 'Friday'(330).
when 6.
    l_day = 'Saturday'(331).
when 7.
    l_day = 'Sunday'(332).
Else.
    Raise exception type zcx_day_problem.
endcase.

```

Listing 2.38 Filling in a Variable Using a CASE Statement

The day is then added to the month, and you end up with a pretty display for the user, showing what the date is. In 7.4, this can be slightly simplified by using the new SWITCH constructor operator, as shown in Listing 2.39.

```

DATA(L_DAY) = SWITCH char10( l_indicator
    when 1 THEN 'Monday'(326)
    when 2 THEN 'Tuesday'(327)
    when 3 THEN 'Wednesday'(328)
    when 4 THEN 'Thursday'(329)
    when 5 THEN 'Friday'(330).
    when 6 THEN 'Saturday'(331)
    when 7 THEN 'Sunday'(332)
    ELSE THROW zcx_day_problem( ) ).

```

Listing 2.39 Filling in a Variable Using a SWITCH Statement

As you can see from this example, the data definition for L_DAY (CHAR10 in this case) has moved into the body of the expression, thus dramatically reducing the lines of code needed. In addition, Java fans will jump up and down with joy to see that instead of the ABAP term RAISE EXCEPTION TYPE we now have the equivalent Java term THROW. The usage is identical, however: the compiler evaluates the keywords RAISE EXCEPTION TYPE and THROW as if they were one and the same. As an added bonus, this actually makes more grammatical sense, because THROW and CATCH go together better than RAISE EXCEPTION TYPE and CATCH. (It's lucky that exception classes have to start with CX; otherwise some witty programmer at SAP would create an exception class called up.)

2.5.5 The COND Statement as a Replacement for IF/ELSE

All throughout this book, you will be reading about how resistance to change is bad. Despite that, let's admit it: Change can be annoying, especially when it affects your code. One great example of this involves coding a `CASE` statement based on the assumption that one value is derived from the value of another. Then, someone comes along and tells you that the rules have changed, and the last value in the `CASE` statement is only true if it is a Tuesday. On Wednesday, the value becomes something else.

`CASE` statements can only evaluate one variable at a time, so, in the case of this example, you have to change the whole thing into an `IF/ELSE` construct. That is not the end of civilization as we know it, but the more changes you have to make, the bigger the risk.

Say that you're really scared that the logic you've been given might change at some point in the future, but nonetheless you start off with a `CASE`, such as the code in Listing 2.40, which is pre-7.4 code that evaluates the description of a monster's sanity based on a numeric value.

```
* Fill the Sanity Description
CASE cs_monster_header-sanity.
  WHEN 5.
    cs_monster_header-sanity_description = 'VERY SANE'.
  WHEN 4.
    cs_monster_header-sanity_description = 'SANE'.
  WHEN 3.
    cs_monster_header-sanity_description = 'SLIGHTLY MAD'.
  WHEN 2.
    cs_monster_header-sanity_description = 'VERY MAD'.
  WHEN 1.
    cs_monster_header-sanity_description = 'BONKERS'.
  WHEN OTHERS.
    cs_monster_header-sanity_description = 'RENAMES SAP PRODUCTS'.
ENDCASE.
```

Listing 2.40 CASE Statement to Evaluate Monster Sanity

In 7.4, you can achieve the same thing, but you can do this in a more compact way by using the `COND` constructor operator. This also means that you do not have to keep specifying the target variable again and again (see Listing 2.41).

```
* Fill the Sanity Description
cs_monster_header-sanity_description =
COND text30(
```

```

WHEN cs_monster_header-sanity = 5 THEN 'VERY SANE'
WHEN cs_monster_header-sanity = 4 THEN 'SANE'.
WHEN cs_monster_header-sanity = 3 THEN 'SLIGHTLY MAD'.
WHEN cs_monster_header-sanity = 2 THEN 'VERY MAD'.
WHEN cs_monster_header-sanity = 1 THEN 'BONKERS'.
ELSE.
  cs_monster_header-sanity_description = 'RENAMES SAP PRODUCTS'.
ENDIF.

```

Listing 2.41 Using the COND Constructor Operator

That looks just like a `CASE` statement, and the only benefit in this change at this point is that it's a bit more compact. However, when the business decides that you need to take the day into account when saying if a monster is bonkers or not, you can just change part of the `COND` construct. In the pre-7.4 situation, you had to give up on the whole idea of a `CASE` statement and rewrite everything as an `IF/ELSE` construct. The only change needed to the `COND` logic is shown in Listing 2.42.

```

WHEN cs_monster_header-sanity = 1 AND
  ld_day = 'Tuesday' THEN 'HAVING AN OFF DAY'.
WHEN cs_monster_header-sanity = 1 THEN 'BONKERS'.
ELSE.
  cs_monster_header-sanity_description = 'RENAMES SAP PRODUCTS'.
ENDIF.

```

Listing 2.42 COND Constructor Operator with Updated Logic

2.6 Internal Tables

Internal tables are the bread and butter of programming in ABAP. A lot of other languages, like Java, can only dream of having such a thing; they have to deal with various sorts of arrays and stacks and hash browns and what have you. In this section, you'll learn about some new ABAP functionalities that relate to internal tables.

2.6.1 Using Secondary Keys to Access the Same Internal Table in Different Ways

7.02 Feature

This feature came about with ABAP 7.02, but is discussed here because it's little known.

Internal tables are a bit like database tables, and with release 7.02 SAP began the long march of adding operations only possible for database tables into internal table processing as well. The process started with the idea that if database tables can have secondary keys, then why can't internal tables have them as well?

Traditionally, if you wanted to read an internal table in two different ways (e.g., looking for a monster by name or by number), then you either had to keep sorting the table just before a read, which is a waste of processing time, or have two identical tables sorted differently, which is a waste of memory.

For example, say you have monsters set up as material masters (table `MARA`), and the `MATNR` is an internally generated SAP number, but the `BISMT` is the reference number that mad scientists often use to refer to a given monster. In a given query, half the queries will be on `MATNR` and half on `BISMT`. Moreover, some monster data is plant specific (table `MARC`). Sometimes, you only want the details of one monster at one plant; other times you want the details for that monster at all plants. One final requirement is that when you have the primary key of either `MARA` or `MARC` you want access to be lightning fast, so a `HASHED` table is the way to go.

Previously, you could not satisfy all these requirements at once with only two tables, but as of 7.02 you can. The table definitions would be as shown in Listing 2.43.

```
data:
MT_MARA TYPE HASHED TABLE OF mara WITH UNIQUE KEY matnr
WITH NON-UNIQUE SORTED KEY sort COMPONENTS bismt .
data:
MT_MARC TYPE HASHED TABLE OF marc
WITH UNIQUE KEY matnr werks
WITH NON-UNIQUE SORTED KEY sort COMPONENTS matnr .
```

Listing 2.43 Table Definitions

The first one (`MARA`) is just like having a secondary Z index on `BISMT` in the database table definition. What has been done with `MARC` is a bit more obscure, but all will be clear in a second or two.

First up, when you have a query coming in on `MARA`, you either have `BISMT` or `MATNR` specified, and you have to react accordingly. In Listing 2.44, you first of all determine which of the two values have been passed in; if it is the material, then that's great, and you can read the material internal table using the primary key (a

unique hashed key). However, if the `BISMT` has been passed in, then you cannot use the primary key, so you have to use a secondary key to get the record.

```

IF id_matnr IS INITIAL AND
  id_bismt IS INITIAL.
  RETURN.
ENDIF.

IF id_matnr IS NOT INITIAL.
  READ TABLE mt_mara INTO es_mara
  WITH TABLE KEY matnr = ld_matnr.
ELSE.
  READ TABLE mt_mara INTO es_mara
  WITH KEY sort COMPONENTS bismt = id_bismt ##primkey[sort].
ENDIF.

```

Listing 2.44 Example of Internal Table Accesses with Primary and Secondary Keys

Note that even though `MT_MARA` is a `HASHED` table it is also in some sense a `SORTED` table with the key `BISMT`, so when you go looking for the record using `BISMT` a `BINARY SEARCH` is automatically performed. That is not quite as fast as a hashed access, but it's an order of magnitude better than a sequential search and a lot faster than the possible alternative of re-sorting the table each time the search changes.

Next, turn to `MARC`. Either you have both the plant and the material specified, in which case you return a single record, or just the material, in which case you return a table of the data from all plants. The problem is that `HASHED` tables are not really designed for being looped through. Listing 2.45 once again takes advantage of the internal table having two indexes. If you have both plant and material specified, then you can use the primary key, which is `HASHED`. But if you only have the plant, then you can use the secondary index to find the first valid record in the table via the implicit `BINARY SEARCH`, and then loop through all valid records in the order they are defined in the secondary index.

```

"Plant specified, return single record"
IF id_werks IS NOT INITIAL.

  READ TABLE mt_marc INTO es_marc WITH TABLE KEY matnr = ld_matnr
  werks = id_werks.

ELSE.
  "No plant, return all plant data for material"
  READ TABLE mt_marc TRANSPORTING NO FIELDS
  WITH KEY sort COMPONENTS matnr = ld_matnr ##primkey[sort].

```

```

CHECK sy-subrc = 0.
ld_start_point = sy-tabix.
LOOP AT mt_marc INTO ls_marc
USING KEY sort FROM ld_start_point.
  IF ls_marc-matnr GT ld_matnr.
    EXIT."From Inner Loop
  ENDIF.
  CHECK ls_marc-matnr = ld_matnr.
  APPEND ls_marc TO et_marc.
ENDLOOP.
SORT et_marc BY matnr werks.
ENDIF.

```

Listing 2.45 Looping Through a HASHED Table with a Secondary SORTED Index

At this point, the code defining MT_MARC (Listing 2.44) should be clear: depending on what key you choose, the table can act as a HASHED table or as a SORTED table.

2.6.2 Table Work Areas

Some time back, SAP decreed that header lines in internal tables were the work of the devil. This was because the use of header lines led to the existence of two data objects in your program with the exact same name, and this was viewed as confusing. For example, it wouldn't be uncommon to find a program in which you had an ITAB variable that referred to the internal table as a whole and also a work area called ITAB that referred to the current line of the table being processed. Clearly, that is wrong from an academic point of view—but in real life ABAP programmers were so used to the idea of the header line that it was a hard habit to shake. This was especially true because you had to explicitly declare a variable to act as the header line. For example:

```
DATA: ls_itab LIKE LINE OF itab.
```

The good news is that now you can avoid having to make that extra variable declaration and still have two differently named variables, one for the table and one for the work area. In release 7.4, the syntax for reading into a work area and looping through a table is as shown in Listing 2.46.

```

READ TABLE lt_monsters WITH KEY monster_number = ld_monster_
number INTO DATA(ls_monsters).
LOOP AT lt_monsters INTO DATA(ls_monsters).

```

Listing 2.46 Reading Into a Work Area and Looping Through a Table

In Section 2.2.2, you learned that from 7.4 onwards you no longer need to do a `DATA` declaration for elementary data types. It is exactly the same for the work areas, which are of course structures. If you are looping into a work area called `LS_MONSTERS` and the work area structure does not match the type of the table `LT_MONSTERS`, you would get a syntax error.

Therefore, the compiler knows the type it wants—so why do you have to tell it that type in an explicit data declaration? The answer is that now you don't, and this change will remove all those extra data declaration lines you had to insert when everyone moved away from header lines (if they ever did) and had to explicitly declare work areas instead.

In the same way that you no longer need `DATA` declarations for table work areas, you also no longer need `FIELD-SYMBOL` declarations for the (common) situations in which you want to change the data in the work area while looping through an internal table. In release 7.4, if you want to use field symbols for the work area, then the syntax is as shown in Listing 2.47.

```
READ TABLE lt_monsters WITH KEY monster_number = ld_monster_
number ASSIGNING FIELD-SYMBOL(<ls_monsters>).
LOOP AT lt_monsters ASSIGNING FIELD-SYMBOL(<ls_monsters>).
```

Listing 2.47 Field Symbols for Work Area

One vitally important point to note is that if you declare a work area and then loop through an internal area, after the loop is over the work area will be filled with the value of the last row of the internal table that was processed. However, if you loop through an internal table into a work area created via the `DATA` statement, then you might wonder if the work area ceases to exist after you leave the loop.

From an academic point of view, not being able to access the work area outside of the loop is a Good Thing, because the textbooks all say that variables should not exist outside their scope (the loop, in this case). And, indeed, at SAP this behavior is very much desired. Unfortunately, the fact is that the variable does exist after you leave the loop. SAP has got around this problem by publishing guidelines saying that if you access such a variable outside of its scope, then you are *very naughty*, and the staff at SAP will say “tut, tut” if they find out. (Personally, I am not convinced that this dire threat will have much effect, but I thought I would let you know.)

2.6.3 Reading from a Table

If I told you that you would never have to use the statement `READ TABLE` again to get a line out of an internal table, then the bottom would perhaps drop out of your world. In fact, that is exactly what happens in ABAP 7.4. You can remove the keywords altogether, and replace the English phrase `READ TABLE` with a more computer-like pair of square brackets: `[]`.

Prior to ABAP 7.4, calling a line from an internal table would look like Listing 2.48.

```
READ TABLE lt_monsters INTO ls_monsters WITH KEY monster_name = ld_
monster_name.
lo_monster = zcl_monster_factory( ls_monsters-monster_number ).
```

Listing 2.48 Calling a Line from an Internal Table before 7.4

However, in ABAP 7.4, this is simplified, as shown in Listing 2.49.

```
lo_monster = zcl_monster_factory( lt_monsters[ monster_name = ld_
monster_name ]-monster_number ).
```

Listing 2.49 Calling a Line from an Internal Table in 7.40

Note

Some would say that making code less like English and more like machine code actually *prevents* clarity. For example, if after reading your internal table line you wanted to add it to a string and pass the string to a method, then you could go overboard using punctuation marks in your code. The part where you read the internal table gets buried in a morass of code, and someone reading the code might struggle to work out what is happening:

```
zcl_bc_output( |{ ld_monster_name }'s Monster Number is { lt_
monsters[ monster_name = ld_monster_name ]-monster_number }| ).
```

However, so as not to offend the brackets, that discussion will wait until Appendix A.

What happens if such an expression cannot find the internal table line you are looking for? The answer is not good news; an exception is raised, whereas you were probably expecting a blank (initial) value to be returned. SAP must have received a lot of complaints about this, because as of 7.4 (SP 8) they delivered a workaround: If you add the word `OPTIONAL` at the end of your query, then suddenly the system does not care if no record is found, and an initial value of whatever data type you are looking for is returned, which is what would happen with

a traditional `READ TABLE xyz INTO ls_abc` when the read failed. An example of this is shown in Listing 2.50.

```
zcl_bc_output( |{ ld_monster_name }'s Monster Number is { lt_monsters[
  monster_name = ld_monster_name ]-monster_number OPTIONAL }| ).
zcl_bc_output( |{ ld_monster_name }'s Monster Number is { lt_monsters[
  monster_name = ld_monster_name ]-monster_number DEFAULT '999999' }| ).
```

Listing 2.50 OPTIONAL

The addition `DEFAULT` fills in a hard-coded value when the read fails. Then, you can check if the read failed by comparing the result with that hard-coded value in the same way you would if `SY-SUBRC` was 4 when checking if a traditional read on an internal table succeeded.

2.6.4 CORRESPONDING for Normal Internal Tables

You no doubt use the ABAP keyword `MOVE-CORRESPONDING` a lot, moving variable values from one structure to another.

Warning: Houston, We Have a Problem

In some standard SAP documentation, you are advised never to use `MOVE-CORRESPONDING` at all, most likely to avoid the problem of moving strings into number fields, highlighted at the start of this section. However, if you do use it, then the functionality discussed here should be of help to you.

New in 7.4 is a constructor operator called `CORRESPONDING` without the word `MOVE` in front. This operator takes moving data between two internal tables to a whole new level. Say that you have two internal tables full of monster-related data, but they are defined with a different set of columns. What you want to do is copy over the fields with the same names from one table to the other, with two exceptions:

- ▶ You do not want to copy the `EVILNESS` value from one table to the other, even though both tables have an `EVILNESS` column.
- ▶ You want to copy the column named `MOST_PEAASANTS_SCARED` from one table into a similar column called `PEOPLE_SCARED` in the second table.

Prior to 7.4, you would declare a bunch of helper variables to store the work areas of the two tables, and then loop through the first table, moving everything from

one table to another. You'd then perform some logic to deal with your exceptions. This can be seen in Listing 2.51.

```
FIELD-SYMBOLS:
<l_s_green_monsters> LIKE LINE OF gt_green_monsters,
<l_s_blue_monsters> LIKE LINE OF gt_blue_monsters.
LOOP AT gt_green_monsters ASSIGNING <l_s_green_monsters>.
  APPEND INITIAL LINE TO gt_blue_monsters
  ASSIGNING <l_s_blue_monsters>.
  MOVE-CORRESPONDING <l_s_green_monsters> TO <l_s_blue_monsters>.
  CLEAR <l_s_blue_monsters>-evilness.
  <l_s_blue_monsters>-people_scared =
  <l_s_green_monsters>-most_peasants_scared.
ENDLOOP.
```

Listing 2.51 Moving One Table to Another before 7.4

In release 7.4, you can use the constructor operator `CORRESPONDING` to do the exact same thing, but this time you do not have to declare the field symbols. You tell the `CORRESPONDING` operator what the rules are, and it loops through the table for you while executing those rules. Moreover, you don't even need to define the target table; if you put a `#` after the `CORRESPONDING` operator, then it creates the target table with the same columns as the source table but without the `EVILNESS` column and with the `SCARED` column having a different name but the same type as the source column. This can be seen in Listing 2.52.

```
gt_green_monsters = CORRESPONDING #(
gt_blue_monsters
MAPPING people_scared = most_peasants_scared
EXCEPT evilness ).
```

Listing 2.52 Moving One Table to Another in 7.4

2.6.5 MOVE-CORRESPONDING for Internal Tables with Deep Structures

In ABAP, a deep structure is not a structure that thinks a lot and has a complex personality but rather a structure that—in addition to elementary data types, such as strings and numbers—has internal tables. Listing 2.53 shows an example of this. `T_RESULTS` is defined as a table type, so you have an internal table in which each row has a column that is itself an internal table.

```
TYPES: BEGIN OF l_typ_monster_results,
        monster_number TYPE zde_monster_number,
```

```

    monster_name TYPE zde_monster_Name
    t_results    TYPE ztyp_monster_results,
END OF l_typ_monster_results.
DATA: lt_results TYPE STANDARD TABLE OF l_typ_monster_results.

```

Listing 2.53 Deep Structure

Recently, SAP has clearly been thinking about what happens when you perform a `MOVE-CORRESPONDING` between two slightly different structures that are both deep but that are defined slightly differently. As you know, `MOVE-CORRESPONDING` compares the field names of two structures, and if a field called `RESULT` is a string in one and a number in another and the value is `XYZ` in the first structure, then trouble looms (i.e., a short dump). Take this one step further, and say you have two internal tables in two deep structures with the same name but defined differently. What is going to happen when you perform a `MOVE-CORRESPONDING`?

To start off with, the next examples examine what happens in release 7.02 when moving from one structure to another, and then expand this to look at how the same process behaves in 7.4. Spoiler: In 7.4, you can do a `MOVE-CORRESPONDING` between internal tables as a whole, which was not possible in lower versions.

Say that in Europe it is important to keep track of something called an `IBAN` code for each monster, which is meaningless anywhere else in the world, and in the United States it is important keep track of a `LOCKBOX` code for each monster, which likewise is meaningless anywhere else in the world. Listing 2.54 sets up structures to store data for each region and then tries to do a `MOVE-CORRESPONDING` to move some European monster data to the US equivalent.

```

TYPES: BEGIN OF l_typ_european_monsters,
    monster_name TYPE string,
    monster_iban_code TYPE string,
END OF l_typ_european_monsters.

DATA: ls_type_iban_codes TYPE l_typ_european_monsters.

TYPES: BEGIN OF l_typ_us_monsters,
    monster_name TYPE string,
    monster_lockbox_code TYPE string,
END OF l_typ_us_monsters.

DATA: BEGIN OF ls_european_monsters,
    laboratory TYPE string,
    t_result TYPE STANDARD TABLE OF l_typ_european_monsters,

```

```

END OF ls_european_monsters.

DATA: BEGIN OF ls_us_monsters,
      laboratory TYPE string,
      t_result  TYPE STANDARD TABLE OF l_typ_us_monsters,
END OF ls_us_monsters.

ls_european_monsters-laboratory = 'SECRET LABORATORY 51'.

ls_type_iban_codes-monster_name  = 'FRED'.
ls_type_iban_codes-  monster_iban_code =
'AL47212110090000000235698741'.
APPEND ls_type_iban_codes TO ls_european_monsters-t_result.

MOVE-CORRESPONDING ls_european_monsters TO ls_us_monsters.

```

Listing 2.54 Attempt at MOVE-CORRESPONDING

What do you think happens? In fact, in the US monster structure the results table gets the LOCKBOX field filled with the IBAN code from the European equivalent, because they have the same name.

Usually, this is not what you want. For identical tables, you can always get around this using `LT_ONE[] = LT_TWO[]`, which changes the first table into an identical copy of the second table. However, this doesn't work in cases where, say, you have a database table with 10 fields and an internal table with those 10 fields plus five more fields with text descriptions or calculated fields. Instead, you would need to read the database table into one internal table with just the 10 fields, and loop through this first internal table, moving the corresponding elements to some sort of second output table, where you fill in extra data, such as names for sales offices and the like and calculated fields.

Thus, instead of the code shown in Listing 2.55 you would use the code shown in Listing 2.56.

```

LOOP AT lt_european_monsters ASSIGNING <ls_european_monsters>.
  APPEND INITIAL LINE TO lt_us_monsters
  ASSIGNING <ls_us_monsters>.
MOVE-CORRESPONDING <ls_european_monsters> TO <ls_us_monsters>.
ENDLOOP.

```

Listing 2.55 Copying between Internal Tables with Different Structures before 7.4

```

MOVE-CORRESPONDING lt_european_monsters TO lt_us_monsters.

```

Listing 2.56 Copying between Internal Tables with Different Structures in 7.4

This saves you a few lines of code and is wonderful for when the line structures of the two tables are flat. However, it still doesn't solve the problem for deep structures. As you've seen, `MOVE-CORRESPONDING` on its own will dump the contents of an internal table component, such as `LT_RESULT`, into the identically named component in the target structure. If the `LT_RESULT` in the target has differently named columns, then all sorts of bizarre results might ensue. To counteract this, SAP has come up with two new additions to `MOVE-CORRESPONDING` in release 7.4. The next subsections look at them first one at a time and then combine them.

MOVE-CORRESPONDING EXPANDING NESTED TABLES

In `MOVE-CORRESPONDING EXPANDING NESTED TABLES`, anything that might happen to be in the target internal table already is deleted. Columns with simple values, like numbers, are copied to their identically named friends. However, when it comes to complex columns such as `LT_RESULT`, only fields that have the same column name inside the nested tables would get copied. For example, the `MONSTER_NAME` would get copied, because there is an equivalently named column in the target, but `MONSTER_IBAN_CODE` would not, because there is no column with the same name, only the `LOCKBOX`.

MOVE-CORRESPONDING KEEPING TARGET LINES

What happens in `MOVE-CORRESPONDING KEEPING TARGET LINES` is quite strange; if there is anything already in the target internal table, then it stays there. Then, at the end of the target table, extra rows are added—which is what would happen if you just performed a `MOVE-CORRESPONDING` between the two tables. This is rather like `APPEND LINES OF itab1 TO itab2`, except that the two tables have different structures, and only identically named components come across.

MOVE-CORRESPONDING EXPANDING NESTED TABLES KEEPING TARGET LINES

As might be imagined, `MOVE-CORRESPONDING EXPANDING NESTED TABLES KEEPING TARGET LINES` does both of the previous tasks at once; it acts like `APPEND LINES OF itab1 TO itab2`, copying only identically named components to the new rows, but is also a bit clever with any internal table components and only copies identically named components within these structures.

2.6.6 New Functions for Common Internal Table Tasks

When working in ABAP code, there are cases in which you want to know in exactly what line of an internal table the data you are interested in lives. To find this information out in pre-7.4 ABAP, you would have written code that declared a helper variable to store the row number of the target record, read the table for no other purpose than to find that row number, and then transferred the system variable that contained the result of the table read into your helper variable. An example of this is shown in Listing 2.57.

```
DATA: ld_tabix TYPE sy-tabix.
READ TABLE lt_monsters WITH KEY monster_number = ld_monster_
number TRANSPORTING NO FIELDS.
IF sy-subrc = 0.
  ld_tabix = sy-tabix.
ENDIF.
```

Listing 2.57 Reading an Internal Table to Get the Row Number

You see the logic in Listing 2.57 a lot when processing nested loops—you follow such code with a `LOOP AT itab STARTING AT ld_tabix`—but also in a myriad of other use cases. In any event, in release 7.4 this can be simplified by using the built-in function `LINE_INDEX`, which does the exact same task but without the need for looking at the values of `SY-SUBRC` and `SY-TABIX` (`STARTING AT LINE_INDEX`, etc.). This is shown in Listing 2.58.

```
DATA(ld_tabix) =
LINE_INDEX( lt_monsters[ monster_number = ld_monster_number] ).
LOOP AT lt_monsters STARTING AT ld_tabix.
```

Listing 2.58 `LINE_INDEX`

Throughout this chapter, one aim has been to get rid of helper variables. Because `LINE_INDEX` is a so-called built-in function, it can be used at operand positions, thus negating the need for the helper variable `LD_TABIX`. So when looping at an internal table from the row formerly stored in `LD_TABIX` instead, use the code shown in Listing 2.59.

```
LOOP AT lt_monsters STARTING AT LINE_INDEX(etc)
```

Listing 2.59 Built-In Function `LINE_INDEX` at an Operand Position

This new built-in function also has a friend: `LINE_EXISTS`. Say that you want to see if an internal table already has an entry for the monster at hand. If it does, then

you want to modify the existing entry; if not, then you want to add a new entry. The way this was done prior to 7.4 was to first read the internal table to see if there was an existing entry for the monster. If there were no system variable, then SY-SUBRC would not be zero. You would react to that by adding a new entry to the table for your monster. An example of this is shown in Listing 2.60.

```
READ TABLE lt_monsters ASSIGNING <ls_monsters>
WITH KEY monster_number = ld_monster_number.

IF sy-subrc NE 0.
  APPEND INITIAL LINE TO lt_monsters ASSIGNING <ls_monsters>.
ENDIF.

ADD 1 TO <ls_monsters>-scariness.
```

Listing 2.60 Seeing Whether an Internal Table Line Exists before 7.4

By adding the new built-in function `LINE_EXISTS`, the code can be changed as shown in Listing 2.61.

```
IF LINE_EXISTS( lt_monsters[ monster_number = ld_monster_number ] ) =
  abap_false.
  APPEND INITIAL LINE TO lt_monsters ASSIGNING <ls_monsters>.
ELSE.
  READ TABLE lt_monsters ASSIGNING <ls_monsters>
  WITH KEY monster_number = ld_monster_number.
ENDIF.
```

Listing 2.61 Seeing Whether an Internal Table Line Exists in 7.4

“Why is that better?” I hear you cry. For one thing, it makes it more obvious what you are trying to achieve. In addition, using `LINE_EXISTS` instead of `SY-SUBRC` is more reliable; `SY-SUBRC` can be dodgy, because you never know when someone is going to have the bright idea of inserting some code between your `READ` statement and the `IF` statement that evaluates `SY-SUBRC`. If you use `LINE_EXISTS` to have the table read and the evaluation all bundled up in the one statement, as in Listing 2.62, then this means that no one can break the statement with a well-intentioned change.

```
IF LINE_EXISTS( lt_monsters[ monster_number = ld_monster_number ] ) =
  abap_true.
```

Listing 2.62 Code All in One Line, with No Reliance on SY-SUBRC

2.6.7 Internal Table Queries with REDUCE

In Section 2.2.5, you learned a new way to create internal tables by using a `FOR` loop. After you've filled up your internal tables, though, you often want to query them. Say, for example, that you want to know how many really mad monsters you have. As of 7.4, you can do this by using a constructor operator called `REDUCE`, which contains logic to process an internal table and return a single result. In the example in Listing 2.63, you will first say what variable you want created and what type that variable is going to be, then set an initial value for your result variable, and finally loop over the table, performing assorted logic. That sounds really complicated, but as you will see the code is not.

```
DATA:( ld_mad_monsters_count ) = REDUCE sy-tabix(
INIT result = 0
FOR ls_monsters IN lt_neurotic_monsters
NEXT result = result +
lo_checker->is_it_mad( ls_monsters-monster_number ).
```

Listing 2.63 How Many Really Mad Monsters?

The `SY-TABIX` after the `REDUCE` statement defines the type of the variable being created, `LD_MAD_MONSTER_COUNT`. It also defines the type of the temporary variable after the `INIT` statement, which is going to be the result; i.e., both `LD_MAD_MONSTER_COUNT` and `RESULT` have the type `SY-TABIX`. You then loop over your internal table, and every time your `is_it_mad` method comes back with a value of 1, you increase the count of the `RESULT` variable. After the `FOR` loop is finished, the value of `RESULT` gets copied into `LD_MAD_MONSTERS`.

2.6.8 Grouping Internal Tables

You are most likely familiar with the `GROUP BY` addition you can add to a database `SELECT`, which condenses similar records into a single row. In the past, when you wanted to do something similar to an internal table you would maybe use the `COLLECT` statement or the very dodgy `AT NEW` statement, which did not work very well in a lot of circumstances. In fact, the `AT NEW` statement was so unpredictable that it was wiser to avoid it altogether and use other means to process related chunks from your internal table. ABAP 7.4 brings good news: a `GROUP BY` option has been added to looping at internal tables.

To understand what this means, let's look at an example. Say that you have a huge great table of open customer items. You only want to process one customer

at a time; somehow you need to extract the records for that customer, do something with those records, and then move on to the next customer. Technically, you could copy the whole internal table into a similar table each time and delete all the customers you weren't interested in each time—but that seems rather wasteful.

Alternatively, you could have an internal table of customers, do an outer loop of those customers, and for each row of that outer table do an inner loop of your open item table, looking for records relating to the customer in the outer loop. There are ways to do this well—for example, make sure the inner table is a SORTED table—but it is still a lot of effort.

Fortunately, the `GROUP BY` saves the day. Listing 2.64 demonstrates how applying the `GROUP BY` construct to looping over an internal table lets you look at all the open items for one customer at a time without having to go through the agony of nested loops. You loop through the table one customer at a time, and on each pass you're handed all the open items for that customer on a plate.

```

LOOP AT lt_open_items INTO DATA( ls_open_items )
  GROUP BY ( kunnr = ls_open_items-kunnr )
  ASCENDING
  ASSIGNING FIELD-SYMBOL( <customer_items> ).
  CLEAR lt_open_items_for_customer.
LOOP AT GROUP <customer_items>
  ASSIGNING FIELD-SYMBOL( <ls_items> ).
  APPEND <ls_items> TO lt_open_items_for_customer.
ENDLOOP. "Items for one customer
process_open_items->(
EXPORTING it_items   = lt_open_items_for_customer
CHANGING ct_proposals = ct_proposals ).
ENDLOOP. "Grouped Open Items

```

Listing 2.64 GROUP BY

As you can see, the field symbol `<CUSTOMER_ITEMS>` is a single row, such as you would get back from the equivalent SQL statement. However, by using `LOOP AT GROUP` to see the lines that make up the whole, you can break it open again. This might look like it is a geometric loop, but the inner loop is only processed once for each group of similar items that the outer loop finds. Therefore, the runtime only increases in a linear fashion, which means that your program is not going to get into the situation in which it will be fine for a small number of records but will time out when there is a lot of data.

Tip

In the SQL version of `GROUP BY`, you can only use column names. Because the internal table version of `GROUP BY` is processed wholly within ABAP, you can do all sorts of groovy things in addition—comparison, method calls, and the like, for example:

```
LOOP AT lt_monsters INTO DATA( ls_monsters )
  GROUP BY( type = ls_monsters-type
            crackers = is_it_mad( ls_monsters-number )
            ASSIGNING FIELD-SYMBOLS( <ls_monster_group> ).
```

This takes quite a bit of experimentation before you get comfortable with it. The most difficult part, as with all of these new constructs, is marrying new abilities to real-world problems, such as the customer open item example presented earlier.

2.6.9 Extracting One Table from Another

There are two new ways of extracting one internal table from another that were introduced with 7.4, and both use the constructor operator `FILTER`. The next subsection will talk first about using the `FILTER` operator with conditional logic and then as a `FOR ALL ENTRIES` operation on an internal table.

FILTER with Conditional Logic

To understand the process of extracting one table from another in ABAP 7.4, return to the monsters example. Once again, you have a big table of all the monsters, and you want to extract a smaller internal table with just the averagely mad monsters. Normally, you would just loop through the big table and append lines to your new table, as shown in Listing 2.65.

```
LOOP AT lt_all_monsters INTO ls_all_monsters
  WHERE sanity > 25
  AND sanity < 75.
CLEAR ls_averagely_mad_monsters.
MOVE-CORRESPONDING ls_all_monsters TO ls_averagely_mad_monsters.
APPEND ls_averagely_mad_monsters TO lt_averagely_mad_monsters.
ENDLOOP. "All Monsters
```

Listing 2.65 Extracting One Table from Another before 7.4

As of ABAP 7.4 (SP 8), you can do the same thing by using the constructor operator `FILTER` in the fashion shown in Listing 2.66.

```
DATA( lt_averagely_mad_monsters ) =
FILTER #( lt_all_monsters USING KEY sanity
        WHERE sanity > 25 AND
          sanity < 75 ).
```

Listing 2.66 Extracting One Table from Another in 7.40

Therefore, in 7.4 you can do something you could not do before. However, there's a gotcha. The problem is that the `FILTER` operation introduced in 7.4 only works if the large table has either a `HASHED` or `SORTED` key. In the preceding example, `lt_all_monsters` has a `SORTED` key on `sanity`, so the gotcha does not apply. (If the internal table `LT_ALL_MONSTERS` did not have a `HASHED` or `SORTED` index then it would not be such a huge problem; remember that ever since 7.02 internal tables can have several secondary keys, which can be `HASHED` or `SORTED`.)

FILTER Used as a FOR ALL ENTRIES on an Internal Table

When reading from the database into an internal table, if you don't want all of your data in one massive table or you can't merge all the tables together by inner joins, the solution has always been—since the year 2000 and still to this day—to perform a `FOR ALL ENTRIES`. An example of this is shown in Listing 2.67.

```
SELECT *
FROM zt_monsters_pets
INTO CORRESPONDING FIELDS OF lt_monster_pets
FOR ALL ENTRIES IN lt_monsters
WHERE owner = lt_monsters-monster_number.
```

Listing 2.67 FOR ALL ENTRIES during a Database Read

If, earlier in the program, you already had read the entire `lt_monster_pets` table from the database into an internal table for some reason, then you could now do the equivalent of a `FOR ALL ENTRIES` but on an internal table rather than the database (Listing 2.68).

```
DATA( lt_pets_of_our_monsters )
= FILTER #( lt_monster_pets IN lt_monsters
        WHERE owner = monster_number ).
```

Listing 2.68 FOR ALL ENTRIES on an Internal Table

Once again, the filter table (`LT_MONSTERS` in this case) must have a `SORTED` or `HASHED` key that matches what we are searching for (monster number in this case).

ABAP Test Cockpit

Speaking of `FOR ALL ENTRIES`, any programmer with even a little experience will have fallen into the trap in which `FOR ALL ENTRIES` can bring down your program when you pass in an empty table. Happily, when Chapter 4 looks at the ABAP Test Cockpit, you will see that the syntax check now warns you to test that `LT_MONSTERS` is not empty when doing such a thing to avoid the bug in SAP whereby an empty table retrieves the whole database table.

2.7 Object-Oriented Programming

SAP introduced the concept of object-oriented programming in ABAP in the year 2000. Since then, it has been the recommended method of programming. This section will describe the new features of ABAP 7.4 that relate to object-oriented programming.

2.7.1 Upcasting/Downcasting with CAST

In OO programming, a downcast is a process in which you turn a generic object, like a monster, into a more specific object, like a green monster. An upcast is the reverse. This functionality has been available in ABAP for a long time, but it gets a lot easier in 7.4.

To take a non-monster-related example for once, consider a situation where you need to get all the components of a specific dictionary structure. Listing 2.69 shows how you would do this prior to ABAP 7.4. First of all, you call a method of `CL_ABAP_TYPEDESCR` to get metadata about a certain structure. However, to get the list of components of that structure into an internal table, you need an instance of `CL_ABAP_STRUCTDESCR`; this is a subclass of `CL_ABAP_TYPEDESCR`. Thus, you need to do a downcast to convert the instance of the parent class into an instance of the subclass.

```
DATA lo_structdescr TYPE REF TO cl_abap_structdescr.
lo_structdescr ?= cl_abap_typedescr=>describe_by_name( 'ZSC_MONSTER_
HEADER' ).
DATA lt_components TYPE abap_compdescr_tab.
lt_components = lo_structdescr->components.
```

Listing 2.69 Components of a Specific Dictionary Structure without CAST

In 7.4, you can do this all in one line by using the `CAST` constructor operator.

```
DATA(lt_components) = CAST c1_abap_structdescr(
c1_abap_typedescr=>describe_by_name( 'ZSC_MONSTER_HEADER' ) )->
components.
```

Listing 2.70 Components of a Specific Dictionary Structure with `CAST`

The code in Listing 2.69 and Listing 2.70 performs exactly the same function, but in the latter case you no longer need the helper variable `LO_STRUCTDESCR` and you also do not need the line where you define the type `LT_COMPONENTS`.

2.7.2 CHANGING and EXPORTING Parameters

In ABAP, a functional method has until now been defined as a method with one returning parameter and zero to many importing parameters, such as the following:

```
ls_monster_header = lo_monster->get_details( ld_monster_number ).
```

Many ABAP programmers liked the fact that you could put the result variable at the start rather than having to put that variable in an `EXPORTING` parameter. However, they wanted to be able to also have `CHANGING` and `EXPORTING` parameters as well—that is, to have their cake and eat it too.

In 7.40, SAP has waved its magic wand, and now you can have it both ways. An example is shown in Listing 2.71.

```
ls_monster_header = lo_monster->get_details(
EXPORTING id_monster_number    = ld_monster_number
IMPORTING ed_something_spurious = ld_something_spurious
CHANGING cd_something_unrelated = ls_something_unrelated).
```

Listing 2.71 `CHANGING` and `EXPORTING` Parameters

There is no doubt many people will be happy with this, but purists who are used to other languages will be *horrified*. (Although I do not feel quite that strongly, I can see their point.) Good OO design leads you toward small methods that do one thing, and the one thing for functional methods is to output one result. If a functional method suddenly starts giving you back all sorts of other exporting parameters and changes something else, then the method is clearly doing more than one thing, and that is probably bad design. (For example, there are methods that are designed to return four values of a polynomial equation, and you could use

the new design to put the first value in the `RETURNING` parameter and the last three values in `EXPORTING` parameters, but that seems a bit silly. You could just return a structure of four values instead.) Nonetheless, because this is a new feature of 7.4 that you might occasionally find useful, it was important to mention it here.

2.7.3 Changes to Interfaces

This section discusses how SAP has tried to take some of the pain out of your daily usage of interfaces in OO programming in 7.4. As you know, an interface is a collection of data declaration and method names and signatures. If a class implements any given interface, then it has to redefine all the interface methods. This is all good, but prior to 7.4 the problem was that some standard interfaces had a really big list of methods, only some of which were relevant, and so you had to go through the irrelevant methods, redefining them to have blank implementations.

As of 7.4, if you are creating an interface and think that some of the methods might not be needed by all classes that implement the interface, then you can say so in the interface definition. An example of this is shown in Listing 2.72.

```
INTERFACE scary_behavior.
  METHODS: scare_small_children,
           sells_mortgages    DEFAULT FAIL,
           hide_under_bed    DEFAULT IGNORE,
           is_fire_breather
           DEFAULT IGNORE
           RETURNING rf_yes_it_is TYPE abap_bool.
ENDINTERFACE. "Scary Behavior
```

Listing 2.72 Defining an Interface with Optional Methods

This is an interface all monster classes should implement. Naturally, all monsters should be able to scare children; thus, do not give that method definition any addition. This means that each class implementing that interface is forced to redefine the method by the syntax check. On the other hand, most monsters will *not* sell mortgages (just the worst of the worst monsters), so do not force all the classes to implement this method. Because you have added `DEFAULT FAIL` if a program using an instance of a monster that implements this interface tries to make the monster sell mortgages, and the method has not been implemented, then a runtime error occurs (`CX_SY_DYN_CALL_ILLEGAL_METHOD`).

Similarly, do not force all monster classes to hide under beds; obviously, the ones that are a thousand feet tall have problems in this area. By adding `DEFAULT IGNORE`

to the end of the definition, we can make sure these classes aren't forcibly implemented. If the program tells such a monster to hide under the bed, then nothing will happen—just as if a call had been made to an implemented method with no lines of code inside it.

In the same way, not all monsters breathe fire. For the ones that do, the `IS_FIRE_BREATHER` method can be implemented to return `ABAP_TRUE`. If the method is not implemented in any given monster class, then the `DEFAULT IGNORE` addition is used, and the `RETURN` parameter will bring back an initial value, which in this case is `ABAP_FALSE`.

2.8 Search Helps

Search helps are one of the strengths of the SAP system. You can define one and attach it to a data element, and then all throughout the system wherever that data element is referenced an `F4` dropdown of possible values is available instantly. In release 7.4, things have become slightly better, and this section will explain how.

2.8.1 Predictive Search Helps

If you have read anything in the IT media in the last few years, then you will have become sick to death of writers saying that business users now expect at work the same level of usability they get in their spare time. At this point, you've probably seen children with iPads; they swipe away at the screen, calling up this and that, already more adept at this than many adults. If that is the level of user friendliness children are exposed to, then what are they going to expect from enterprise software like SAP when they grow up and get jobs? What would someone who had just left school expect from an SAP system they were being shown how to use on their first day of work?

For one thing, they would expect that when they start typing something in a search field—a customer or material name, for example—after a letter or two has been typed, a dropdown box of potential candidates would appear for them to choose from. That's what happens on Google and many other websites, and you can see why recent students would be shocked when this does not happen on an SAP screen.

The good news is that if your SAP system is 7.4 SP 3 or above and your GUI is 730 patch level 5 or above, then you will find that such predicative search helps are now possible in SAP. Half of the functionality is in the backend, and half is in the GUI, which is why you need both to be on the correct level. (In fact, you really need to be on 7.4 SP 6 for this to work automatically; otherwise you have to fluff around manually with the `CL_DSH_DYNPRO_PROPERTIES` class, as described in SAP Note 1861491.)

Assuming you are on the right version, open Transaction SE11 and go into the SEARCH HELP option about halfway down the screen. You will see the DATA COLLECTION area (Figure 2.2).

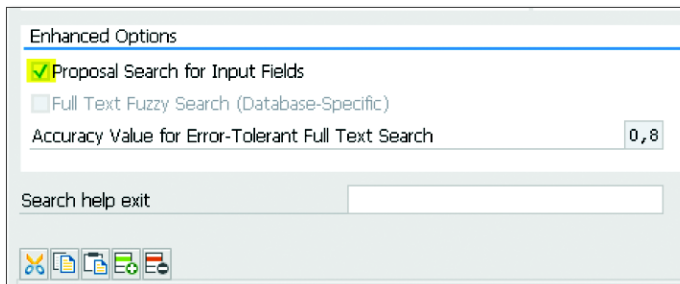


Figure 2.2 Search Help Definition

In release 7.4, directly above the SEARCH HELP EXIT box is a new box called ENHANCED OPTIONS, in which you have two checkboxes and an input field. The latter two are to do with fuzzy text searches on an SAP HANA database and are grayed out, but the first one says PROPOSAL SEARCH FOR INPUT FIELDS.

If you select that checkbox, then your search help will magically start behaving differently—not quite like Google, but a pop-up list of candidates will appear as the user types, which is a step up from before.

2.8.2 Search Help in SE80

You may have noticed, and been bothered, that when you are in SE80 and you cannot remember the name of your program, you cannot just press `[F4]` and see the result list. You have to press the dropdown arrow to the right of the input field, which presumably is why that dropdown arrow is there in the first place. (After all, if `[F4]` worked, then you would not need a separate dropdown arrow.)

So, you can imagine my delight when I was playing around in one of the latest SAP releases and was in SE80 and had typed in half of my program name and pressed **F4** by reflex. I had not even put in a wild card asterisk, but what should I see but a dropdown list of results (Figure 2.3).

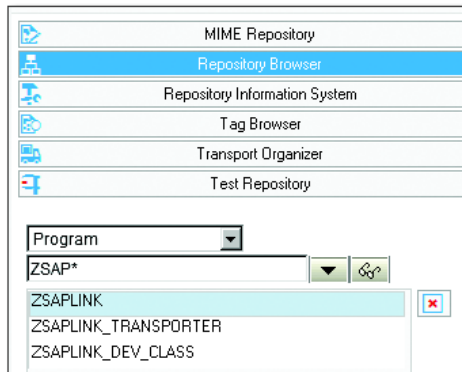


Figure 2.3 F4 Search Help in SE80: Working at Long Last

Mere words cannot explain how happy this made me. It must be true that little things please little minds. It turns out that this is available as of release 7.31—but not everyone knows about it, so it's worth calling out here.

2.9 Unit Testing

Chapter 3 of this book is going to talk about unit testing in some detail. Here, you will look at three areas in which the new features in ABAP 7.40 make unit testing just that little bit easier. Specifically, you'll learn how to create mock classes in which the original class implemented an interface, how to code the contents of a test method faster, and how to create complicated mock objects with fewer lines of code.

2.9.1 Creating Test Doubles Relating to Interfaces

As you'll recall from Section 2.7.3, interface methods can now be made optional, so you do not have to redefine them. The same change has been made to the definition and implementation of test classes, but here the syntax is a little different. Prior to 7.4, you would have to create an empty implementation for every

method defined in the interface, even if you were never going to call that method at any point in your test class. An example of this is shown in Listing 2.73.

```
CLASS lcl_mock_monster DEFINITION FOR TESTING.
  PUBLIC SECTION.
    INTERFACES if_really_big_standard_interface.
ENDCLASS.
```

```
CLASS lcl_mock_monster IMPLEMENTATION.
  METHOD one_i_want_to_use.
  ENDMETHOD.
  METHOD one_i_do_not_want.
  ENDMETHOD.
  METHOD another_i_do_not_want.
  ENDMETHOD.
  Etc.
```

Listing 2.73 Defining and Implementing an Interface in a Test Class before 7.4

From 7.4 on, however, you can write something like Listing 2.74 in your test class. Now, you only have to create implementations of the methods the real class actually uses—that is, not all the blank ones.

```
CLASS lcl_mock_monster DEFINITION FOR TESTING.
  PUBLIC SECTION.
    INTERFACES if_really_big_standard_interface
      PARTIALLY IMPLEMENTED.
ENDCLASS.
CLASS lcl_mock_monster IMPLEMENTATION.
  METHOD one_i_want_to_use.
  ENDMETHOD.
ENDCLASS.
```

Listing 2.74 Defining and Implementing an Interface in a Test Class in 7.4

2.9.2 Coding Return Values from Test Doubles

Usually, when performing unit tests you have some test doubles (mock objects) that return hard-coded values based upon other hard-coded values. Prior to ABAP 7.4, you would end up with something that looks like the code in Listing 2.75.

```
METHOD get_monster_chemical_dose.

  IF id_chemical_group = 'TRANSLYVANIA' AND
     id_monster_strength = 35 AND
     id_chemical_type = 'SNAILS'.
     rd_chemical_dosage = '350'.
  ENDIF.
```

```

IF id_chemcial_type = 'PUPPY_DOG_TAILS'.
  rd_chemical_dosage = 80.
ENDIF.

```

```
ENDMETHOD.
```

Listing 2.75 Mock Objects

However, by using the `COND` construct you learned about in Section 2.5.5, such code can be simplified to the code shown in Listing 2.76. For your return value, instead of a never ending string of `IF/ELSE` statements specifying the variable `RD_CHEMICAL_DOSAGE` variable inside each branch, you have a slightly more compact structure.

```
METHOD get_monster_chemical_dose.
```

```

rd_chemical_dosage = COND zde_chemical_dsoage(
WHEN id_chemical_group = 'TRANSLYVANIA' AND
  id_monster_strength = 35      AND
  id_chemical_type = 'SNAILS'.
  THEN '350'
WHEN id_chemcial_type = 'PUPPY_DOG_TAILS' THEN 80 ).

```

```
ENDMETHOD.
```

Listing 2.76 COND Method

Because such methods tend to have long strings of hard-coded logic like this, even such small reductions in code for each `IF` branch can all add up.

2.9.3 Creating Test Doubles Related to Complex Objects

In Chapter 3, you will read about *injection* in the context of unit testing. Injection makes it easier to set up objects that need to have other objects passed into them upon creation (and some of those objects need other objects passed into them upon creation, and so on). A pre-7.4 example of such an object setup is shown in Listing 2.77.

```

DATA: lo_model  TYPE REF TO zcl_monster_model,
      lo_view   TYPE REF TO lcl_view,
      lo_controller TYPE REF TO lcl_controller,
      lo_logger  TYPE REF TO zcl_bc_logger,
      lo_pers_layer TYPE REF TO lcl_mock_pers_layer.

CREATE OBJECT lo_logger.
CREATE OBJECT lo_mock_pers_layer.

```

```

CREATE OBJECT mo_model
EXPORTING
  io_logger = lo_logger
  io_pers_layer = lo_mock_pers_layer
CREATE OBJECT lo_view.
CREATE OBJECT lo_controller
EXPORTING
  io_model = lo_model
  io_view = lo_view.

```

Listing 2.77 Building Up a Complex Object before 7.4

Well-designed OO programs need a lot of small classes that work together in order to make them more resistant to change—but what a lot of lines of code you need to create an instance of your application controller! If you will forgive the pun, with release 7.4 of ABAP you can replace all this with one “monster” line of code, as shown in Listing 2.78. The `CREATE OBJECT` statements can be replaced by the `NEW` constructor operator each time you want to pass in a parameter to an object’s constructor—and you no longer need the data declarations, because the type of the newly created object is taken from the definition of the importing parameter of the constructor.

```

lo_controller = NEW lcl_controller(
lo_model = NEW zcl_monster_model(
  io_logger = NEW zcl_logger( )
  io_pers_layer = NEW lcl_mock_pers_layer( ) )
lo_view = NEW lcl_view ( )).

```

Listing 2.78 Building up a Complex Object in 7.4

This time, the new process knocked out almost two out of three lines of code: the data declarations and the `CREATE OBJECT` statements.

2.10 Cross-Program Communication

Imagine that Igor the Hunchback is sitting in front of the SAP screen, looking at an ALV report that lists requests for monsters to go out and terrorize villages. Igor has to select a monster, click the GO icon, and then enter some instructions into a pop-up box, and then the monster drops off of the unprocessed part of the list. Meanwhile, in a distant part of the castle, Baron Frankenstein is using CICO to take calls from customers, like evil landowners in top hats who want to hire monsters to scare their rebellious villages back into line.

The problem is that the only way Igor can know when the baron has entered a new request is by clicking the REFRESH icon at periodic intervals, which is not particularly efficient. As a workaround, the baron has taken to banging a large gong whenever he has entered a new request into SAP to let Igor know to click the REFRESH button. However, the gong noise is sometimes drowned out by the thunder and lightning and by the screams of the peasants drifting up from the fields, so it's not really effective.

Furthermore, the Early Watch report from SAP has identified CICO as the most performance-intensive transaction in the baron's entire SAP system. The problem with CICO is that it cannot know when a new phone call has come in from outside the SAP system, so it has to keep constantly polling, which is very performance intensive. The same problem exists with ALV reports within the SAP system; you could have an autorefresh using the `CL_GUI_TIMER` or some such, but that would also spend an awful lot of system time looking for changes that just aren't there.

The baron has an SAP system with many application servers, and chances are he is entering data on one application server while Igor is running his report on another application server. Why does that matter, I hear you ask. It doesn't, if the baron's data is sent to the database and then retrieved a few seconds later by Igor's report performing a database query. This is exactly what's happening in the scenario described thus far—but, as pointed out, it's extremely performance intensive.

Clearly, something needs to be done.

The great news for both the baron and Igor is that release 7.4 of ABAP brings something to the table that will make all of these problems melt away like snow that has just been doused in Tabasco sauce in a blast furnace that has just been hit by a thermonuclear bomb. The solution is *ABAP channels*, and they come in two flavors:

- ▶ ABAP Messaging Channels, which are designed to send messages between programs running on different application servers
- ▶ ABAP Push Channels (which push it real good), which are designed to send messages to and from the Internet using something called *web sockets*

These two channels can work together; for example, a message could come in from the outside world and land in a program in one application server, and that program could promptly decide to send that message to all its friends on the

other application servers. In the case of CICO, the program could subscribe to be notified about incoming phone calls as opposed to having to constantly poll for them.

In the same way, the ALV report has subscribed to be notified about new orders being entered into SAP, the program that processes the new order on one application server publishes that fact to all the application servers, and running instances of the ALV report on any given server can react to this and refresh their displays, again without having to constantly poll.

The mechanics of how to do this are quite complicated and outside the scope of this book. For more information, we recommend the series of blogs mentioned at the end of the chapter, which cover everything from Ping-Pong games to SAPUI5 applications, all enabled by this technology.

2.11 Summary

In this chapter, you have read about the somewhat radical changes that have been introduced in the ABAP language in recent years, starting with 7.02 but most notably in release 7.4. This chapter cataloged the vast array of new language constructs that have been introduced into ABAP and showed examples of where they might be useful in your day-to-day work.

Now that you have learned about the development environment (Eclipse) and the increased capabilities of the language you program in, it's time to move on to the first step of creating a program, which—contrary to what you might expect—is all to do with testing.

Recommended Reading

- ▶ WebSocket Communication with ABAP Channels: <http://scn.sap.com/community/abap/blog/2013/07/18/abap-news-for-release-740--abap-channels> (Horst Keller)
- ▶ Real-Time Notifications and Workflow using ABAP Push Channels: <http://scn.sap.com/community/abap/connectivity/blog/2014/10/16/real-time-notifications-using-abap-push-channels-websockets> (Brad Pokroy)
- ▶ WebSocket Communication Using ABAP Push Channels: <http://scn.sap.com/community/abap/connectivity/blog/2013/11/18/websocket-communication-using-abap-push-channels> (Masoud Aghadavoodi Jolfaei)

Code without tests is bad code. It doesn't matter how well-written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.
—Michael Feathers, Working Effectively with Legacy Code

3 ABAP Unit and Test-Driven Development

During the process of creating and changing custom programs, one of the most important aspects to consider is how to make such changes with minimal risk to the business. The way to do this is via test-driven development (TDD), and the tool for this is ABAP Unit. This chapter explains what test-driven development is and how to enable it via the ABAP Unit framework.

In the traditional development process, you write a new program or change an existing one, and after you are finished you perform some basic tests, and then you pass the program on to QA to do some proper testing. Often, there isn't enough time and this aspect of the software development lifecycle is brushed over—with disastrous results. Test-driven development, on the other hand, is antimatter to the traditional development process; you write your tests before creating or changing the program. That turns the world on its head, which can make old-school developers' heads spin and send them running for the door, screaming at the top of their voices. However, if they can summon the courage to stay in the room and learn about TDD, then they will be a lot better off.

The whole aim of creating your tests first is to make it so that once the tests have all been written as you create—and more importantly change—your application, at any given instant you can follow the menu path PROGRAM • TEST • UNIT TEST to see a graphical display of what is currently working in your program and what is either not yet created or broken (Figure 3.1). This way, when the time comes

to move the created or changed code into test, you can be confident that it is correct.

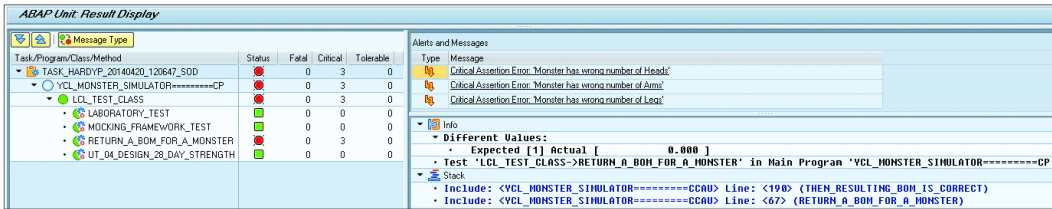


Figure 3.1 ABAP Unit Results Screen

Because that is a highly desirable outcome from everybody's perspective, this chapter explains how to transform your existing code into test-driven code. This process has three main steps:

1. Eliminating dependencies in your existing programs
2. Implementing mock objects in your existing programs
3. Using ABAP Unit to write and implement test classes

Each step will be covered in detail. After that, some additional advice on how to improve your test-driven development via automated frameworks will be presented. Finally, the chapter will conclude with a brief word on behavior-driven development, which is a variation on test-driven development.

Unit Testing Procedural Programs

The examples in this chapter deal with testing object-oriented code (i.e., methods), and most of the examples of ABAP Unit you will find in books or on the web will also focus on OO programming. This does not mean that you can't test procedural programs; it's far easier to test an OO program, but it's not the end of the world to add unit tests to a function module or a procedural program.

You most likely have huge monolithic procedural programs in your system that it would be too difficult to rewrite in an OO fashion, because doing so would take a lot of time and not give you any new functionality. In such cases, whenever you are called on to maintain a section of that program—fixing a bug or adding new functionality—you can make relatively minor changes to break up the dependencies as described in this chapter, and then slowly add test methods to bring little islands of the program under test. As time goes by, more and more islands of the program will have tests, making it more and more stable, until one day the whole continent of the program will be covered.

3.1 Eliminating Dependencies

In the US show “Jeopardy!,” the contestant’s answer must be in the form of a question.

GAME SHOW HOST: Something that stops you dead in your tracks when you want to write a test.

CONTESTANT: What is a dependency?

The contestant is correct. But to go into a little more detail, a dependency is what you have when you want to do a test for your productive code but you can’t—because the productive code reads from the database or writes to the database or accesses some external system or sends an email or needs some input from a user or one of a million other things you cannot or do not want to do in the development environment.

As an example, consider a monolithic SAP program that schedules feeding time at the zoo and makes sure all the animals get the right amount and type of food. Everything works fine. The keepers have some sort of mobile device that they can query to see what animals need feeding next, and there is some sort of feedback they have to give after feeding the animals. They want to keep track of the food inventory, and so on; none of the details are important for this example.

All is well until one day they get two pandas on loan from China, and the programmer has to make some changes to accommodate their specific panda-type needs (e.g., bamboo shoots) without messing up all the other animals. The programmer can do a unit test on the panda-specific changes he has made, but how can he be sure that an unforeseen side effect will not starve the lions, causing them to break out and eat all the birds and all the monkeys before breaking into the insect house and crushing and eating the beehive?

Normally, there’s no way to do it. There are just too many dependencies. The program needs to read the system time, read the configuration details on what animal gets fed when, read and write inventory levels, send messages to the zookeepers, receive and process input from those same zookeepers, interface with assorted external systems, and so on.

However, you really don’t want to risk the existing system breaking and leading the lions to dine on finch, chimps, and mushy bees, so how can we enable tests?

The first step is to break up the dependencies. In this section, you will learn how to do exactly that, a process that involves two basic steps:

1. Looking at existing programs to identify sections of code that are dependencies (and are thus candidates to be replaced by mock objects during a test)
2. Changing your production code to separate out the concerns into smaller classes that deal with each different type of dependency

3.1.1 Identifying Dependencies

Listing 3.1 is an example of a common ABAP application. In this case, our good old friend the baron (remember him from Chapter 2?) does not want any neighboring mad scientists building monsters and thus encroaching on his market, so as soon as he hears about such a competitor he drops a nuclear bomb on him (as any good mad scientist would). Listing 3.1 illustrates this by first getting the customizing settings for the missiles and making sure they are ready to fire, then confirming with the operator that he really wants to fire the missile, then firing the missile, and finally printing out a statement saying that the missile was fired. At almost every point in the code, you will deal with something or somebody external to the SAP system: the database, the operator (user), the missile firing system, and the printer.

```
FORM fire_nuclear_missile.

* Read Database
CALL FUNCTION 'READ_CUSTOMISING'.

* Query Missile Sensor
CALL FUNCTION 'GET_NUCLEAR_MISSILE_STATUS'.

* Business Logic
IF something.
    "We fire the missile here
ELSEIF something_else.
    "We fire the missile there
ENDIF.

* Ask user if he wants to fire the missile
CALL FUNCTION 'POPUP_TO_CONFIRM'.

* Business Logic
CASE user_answer.
    WHEN 'Y'.
        "Off we go!
```

```

        WHEN 'N'.
            RETURN.
        WHEN OTHERS.
            RETURN.
    ENDCASE.

* Fire Missile
    CALL FUNCTION 'TELL_PI_PROXY_TO_FIRE_MISSILE'.

* Print Results
    CALL FUNCTION 'PRINT_NUCLEAR_SMARTFORM'.

ENDFORM."Fire Nuclear Missile

```

Listing 3.1 Common ABAP Application

In this code, you want to test two things:

- ▶ That you direct the missile to the correct target
- ▶ That if the user aborts halfway through, the missile is not actually fired

However, because of the way the routine is written, you can't do a meaningful test unless the following points are true:

- ▶ You have an actual database with proper customizing data.
- ▶ You have a link to the sensor on the missile.
- ▶ You have a user to say yes or no.
- ▶ You actually fire the missile to see what happens (possibly resulting in the world being destroyed).
- ▶ You have a printer hooked up.

These are all examples of *dependencies*. As long as they are part of your code, you can't implement test-driven development.

Unfortunately, most ABAP code traditionally looks like the preceding example. Thus, when preparing an existing program to be unit tested, the first thing to do is make a list of anything that is not pure business logic (i.e., calls to the database, user input, calls to external systems, etc.), exactly as in the bullet list above.

3.1.2 Breaking Up Dependencies

Once you've identified your dependencies, you can resolve them by adopting a *separation of concerns* approach. This approach dictates that you have one class for

database access, one for the user interface layer, and one for talking to an external system; that is, each class does one thing only and does it well. This is known as the *single responsibility principle*. Designing an application this way enables you to change the implementation of, say, your user interface layer without affecting anything else. This type of breaking up (which is hard to do) is vital for unit tests.

Warning: Houston, We Have a Problem

As an aside, and a warning, I have seen a great example in which someone split out all the database access into its own class, presumably following the separation of concerns model. However, that person also made every single variable and method static—and you can't subclass static methods. As a result, the end program still wasn't eligible for unit testing.

To illustrate the separation of concerns approach, take database access as an example. This means that you would go through your program looking for every `SELECT` statement and extract them out in a method of a separate database access class. Repeat the process for other functions of the programs, such as processing user input, communicating with external systems, and anything else you can identify as a dependency, each having one specific class that serves each such purpose. (You may be wondering how to decide which functions need to be split into their own class. Luckily, this is an iterative process; when you start writing tests and the test fails because it does not really have access to proper database entries, user input, or an external system, it will become clear what is a dependency and thus needs to be isolated into its own class.)

The next step is to change the production code to make calls to methods of these newly created classes. The end result will look like Listing 3.2.

```
FORM fire_nuclear_missile.

* Read Database
mo_database_access->read_customising( ).
mo_missile_interface->get_nuclear_missile_status( ).

* Business Logic
IF something.
    "We fire the missile here
ELSEIF something_else.
    "We fire the missile there
ENDIF.

* Ask user if they want to fire the missile
```

```

mo_user_interface->popup_to_confirm( ).

* Business Logic
CASE user_answer.
  WHEN 'Y'.
    "Off we go!
  WHEN 'N'.
    RETURN.
  WHEN OTHERS.
    RETURN.
ENDCASE.

* Fire Missile
mo_missile_interface->tell_pi_proxy_to_fire_missile( ).

* Print Results
mo_printer->print_nuclear_smartform( )..

ENDFORM."Fire Nuclear Missile

```

Listing 3.2 Calling Methods of Classes

Note that the functionality has not been changed at all; the only difference is that the calls to various external systems (the dependencies) are now handled by classes as opposed to functions or FORM routines. All that remains untouched is the business logic.

With the dependencies successfully eliminated, you can now set about implementing mock objects.

3.2 Implementing Mock Objects

After you have isolated each dependency into its own class, you can change your existing programs to take advantage of the ABAP Unit framework. There are two steps to this:

1. Create *mock* objects that appear to do the same thing as real objects dealing with database access and the like, but which are actually harmless duplicates solely for use in unit tests.
2. Make sure that all the class under tests (often a unit test will use several classes, but there is always one main one that you are testing—the *class under test*) is able to use these mock objects instead of the real objects, but only when a test is under way. This is known as *injection*.

Mock Objects vs. Stub Objects

When talking about mock objects, the terms *stub* and *mock* are often used interchangeably; technically, though, there is a difference. If you are testing how your class affects an external system, then the fake external system is a mock, and if you are testing how the fake external system affects your class, then the fake external system is a stub. (Either way, the point is that you use a fake external system for testing.)

3.2.1 Creating Mock Objects

For testing purposes, you want to define *mock classes* and mock objects. Mock classes are classes that run in the development environment. They don't really try to read and write to the database, send emails, fire nuclear missiles, and so on, but they test the business logic nonetheless. Mock objects follow the same principles as regular objects; i.e., in the same way that a monster object is an instance of the real monster class, a mock monster object is an instance of a mock monster class.

This is where the basic features of object-oriented programming come into play: subclasses and interfaces. To continue the previous example, you'll next create a subclass of the database access class that doesn't actually read the database but instead redefines the database access methods to return hard-coded values based upon the values passed in. In Listing 3.3, you'll see some possible redefined implementations of methods in mock subclasses that could replace the real classes in the example.

```
METHOD read_customising. "mock database implementation
*-----*
* IMPORTING input_value
* EXPORTING export_vlaue
*-----*
  CASE input_value.
    WHEN one_value.
      export_value = something.
    WHEN another_value.
      export_value = something_else.
    WHEN OTHERS.
      export_value = something_else_again.
  ENDCASE.
ENDMETHOD. "read customising mock database implementation

METHOD popup_to_confirm. "mock user interface implementation
*-----*
* RETURNING rd_answer TYPE char01
*-----*
```



```

rd_answer = 'Y'.

ENDMETHOD. "mock user interface implementation

METHOD fire_missile. "Mock External Interface Implementation
* Don't do ANYTHING - it's just a test

ENDMETHOD. "Fire Missile - Mock Ext Interface - Implementation

```

Listing 3.3 Mock Method Redefinitions of Assorted Real Methods

In this example, you create subclasses of your database access class, your user interface class, and your external system interface class. Then, you redefine the methods in the subclasses such that they either do nothing at all or return some hard-coded values.

Object-Oriented Recommendation

In order to follow one of the core OO recommendations—to favor composition over inheritance—you should have created an interface that is used by your real database access class and also have the mock class be a subclass that implements that interface. In the latter case, however, you have to create blank implementations for the methods you are not going to use, and that could be viewed as extra effort. Nevertheless, interfaces are a really Good Thing and actually save you effort in the long run. Once you read books like *Head First Design Patterns* (see the “Recommended Reading” box at the end of the chapter), you will wonder how you ever lived without them.

3.2.2 Injection

Usually, classes in your program make use of smaller classes that perform specialized functions. The normal way to set this up is to have those helper classes as private instance variables of the main class, as shown in Listing 3.4.

```

CLASS lcl_monster_simulator DEFINITION.

PRIVATE SECTION.
    DATA:
        "Helper class for database access
        mo_pers_layer TYPE REF TO ycl_monster_pers_layer,
        "Helper class for logging
        mo_logger     TYPE REF TO ycl_logger.

ENDCLASS.

```

Listing 3.4 Helper Classes as Private Instance Variables of the Main Class

These variables are then set up during construction of the object instance, as shown in Listing 3.5.

METHOD constructor.

```

CREATE OBJECT mo_logger.

CREATE OBJECT mo_pers_layer
EXPORTING
    io_logger    = mo_logger    " Logging Class
    id_valid_on = sy-datum.    " Validaty Date

ENDMETHOD.                                "constructor

```

Listing 3.5 Variables Set Up During Construction of Object Instance

However, as you can see, this design does not include any mock objects, which means that you have no chance to run unit tests against the class. To solve this problem, you need a way to get the mock objects you created earlier inside the class under test. The best time to do this is when an instance of the class under test is being created.

When creating an instance of the class under test, you use a technique called *constructor injection* to make the code use the mock objects so that it behaves differently than it would when running productively. With this technique, you still have private instance variables of the database access classes (for example), but now you make these into optional import parameters of the constructor. The constructor definition and implementation now looks like the code in Listing 3.6.

```

PUBLIC SECTION.
METHODS: constructor IMPORTING
    io_pers_layer TYPE REF TO ycl_monster_pers_layer OPTIONAL
    io_logger     TYPE REF TO ycl_logger                 OPTIONAL.

METHOD constructor."Implementation

    IF io_logger IS SUPPLIED.
        mo_logger = io_logger.
    ELSE.
        CREATE OBJECT mo_logger.
    ENDIF.

    IF io_pers_layer IS SUPPLIED.
        mo_pers_layer = io_pers_layer.

```

```

ELSE.
  CREATE OBJECT mo_pers_layer
  EXPORTING
    io_logger      = mo_logger      " Logging Class
    id_valid_on    = sy-datum.      " Validity Date
ENDIF.

```

ENDMETHOD."constructor implementation

Listing 3.6 Constructor Definition and Implementation

The whole idea here is that the constructor has optional parameters for the various classes. The main class needs these parameters in order to read the database, write to a log, or communicate with any other external party that is needed. When doing a unit test, you pass in (inject) mock objects into the constructor that simply pass back hard-coded values of some sort or do not do anything at all. (In the real production code, you don't pass anything into the optional parameters of the constructor, so the real objects that do real work are created.)

Arguments against Constructor Injection

Some people have complained that the whole idea of constructor injection is horrible, because a program can pass in other database access subclasses when executing the code for real outside of the testing framework. However, I disagree with that argument, because constructor injection can give you benefits outside of unit testing. As an example, consider a case in which a program usually reads the entire monster-making configuration from the database—unless you are performing unit testing, when you pass in a fake object that gives hard-coded values. Now, say a requirement comes in that the users want to change some of the configuration values on screen and run a what-if analysis before saving the changes. One way to do that is to have a subclass that uses the internal tables in memory as opposed to the ones in the database, and you pass that class into the constructor when running your simulator program in what-if mode.

At this point, you now have mock objects and a way to pass them into your program. This means that you are ready to write the unit tests.

3.3 Writing and Implementing Unit Tests

At last, the time has come to talk about actually writing the test classes and how to use the ABAP Unit framework. In this section, you will walk through this process, which involves two main steps:

1. Set up the definition of a test class. The definition section of a class, as always, is concerned with the “what,” as in “What should a test class do?” This is covered in Section 3.3.1.
2. Implement a test class. Once you know what a test class is supposed to do, you can go into the detail of how it's achieved technically. This is covered in Section 3.3.2.

Executable Specifications

Some people in the IT world like to call unit tests *executable specifications*, because they involve the process of taking the specification document, breaking it down into tests, and then, when the program is finished, executing these tests. If they pass, then you are proving beyond doubt that the finished program agrees with the specification. If you can't break the specification into tests, then it means that the specification is not clear enough to turn into a program. (Actually, most specifications I get fall into that category. But to be fair to the business analysts, there is only so much that you can write on the back of a post-it note.)

3.3.1 Defining Test Classes

There are several things you need to define in a test class, and the following subsections will go through them one by one. Broadly, the main steps in defining your test class are as follows:

- ▶ Enable testing of private methods.
- ▶ Establish the general settings for a test class definition.
- ▶ Declare data definitions.
- ▶ Set up unit tests.
- ▶ Define the actual test methods.
- ▶ Implement helper methods.

Start the ball rolling by creating a test class. Start with a global Z class that you have defined, and open it in change mode via SE24 or SE80. In this global class, follow the menu option GOTO • LOCAL DEFINITIONS • IMPLEMENTATIONS • LOCAL TEST CLASS.

Enabling Testing of Private Methods

To begin with, enable testing of private methods. The initial screen shows just a blank page with a single comment starting with `use this source file`. In Listing

3.7, you add not only the normal definition line but also a line about `FRIENDS`; this is the line that enables you to test private methods of your main class. In the following example, the global class you will be testing is the monster simulator, and the only way you can test its private methods is if you make it friends with the test class.

```
*** use this source file for your ABAP unit test classes
CLASS lcl_test_class DEFINITION DEFERRED.

CLASS ycl_monster_simulator DEFINITION LOCAL FRIENDS lcl_test_class.
```

Listing 3.7 Defining a Test Class

A lot of people say that testing private classes is evil and that you should only test the methods the outside world can see. However, over the course of time so many bugs have been found in any method at all, be it public or private, that you should have the option to test anything you feel like.

General Settings for a Test Class Definition

Once you have created the class, it's time to establish the general settings for the class, as shown in Listing 3.8.

```
CLASS lcl_test_class DEFINITION FOR TESTING
    RISK LEVEL HARMLESS
    DURATION SHORT
    FINAL.
```

Listing 3.8 Test Class General Settings

Take this one line at a time. In the first line, tell the system this is a test class and thus should be invoked whenever you are in your program and take the `UNIT TEST` option from whichever tool you are in (the menu path is subtly different depending on which transaction you are in).

Now, you come to `RISK LEVEL`. For each system, you can assign the maximum permitted risk level. Although unit tests never run in production, it is possible for them to run in QA or development. By defining a risk level, you could, for example, make it so that tests with a `DANGEROUS` risk level can't run in QA but can run in development. (Leaving aside that I feel unit tests should *only* be run in development, how could a unit test be dangerous? I can only presume it's dangerous if it really does alter the state of the database or fire a nuclear missile. Try as I might, I can't think why I would want a test that was dangerous. Tests are supposed to

stop my real program being dangerous, not make things worse. Therefore, I always set this value to `HARMLESS`, which is what the tests I write are.)

Next is `DURATION`—how long you think the test will take to run. This is intended to mirror the `TIME OUT` dump you get in the real system when a program goes into an endless loop or does a full table scan on the biggest table in the database. You can set up the expected time periods in a configuration transaction.

How long should those time periods be? Well, I'll tell you how long I think a unit test should take to run: it should be so fast that a human cannot even think of a time period so small. The whole point of unit tests is that you can have a massive amount of them and not be afraid to run the whole lot after you've changed even one line of code. It's like the syntax check: most developers run that all the time, but they wouldn't if it took ten minutes. Hopefully, the only reason a unit test would take a minute or more to run is if it actually did read the database or process a gigantic internal table in an inefficient way. If you *are* worried about the method under test going into an endless loop or about having to process a really huge internal table—sequencing a human genome or something—then you could set the `DURATION` to `LONG`, and it would fail due to a time out. Thus far, though, I have never found a reason to set it to anything other than `SHORT`.

Declaring Data Definitions

Continuing with the definition of your test class, you've now come to the data declarations. The first and most important variable you declare will be a variable to hold an instance of the class under test (a main class from the application you're testing various parts of).

This class will be, in accordance with good OO design principles, composed of smaller classes that perform specific jobs, such as talking to the database. In this example, when the application runs in real life, you would want to read the database and output a log of the calculations for the user to see. In a unit test, you want to do neither. Luckily, your application will be designed to use the injection process described in Section 3.2.2 to take in a database layer and a logger as constructor parameters, so you can pass in mock objects that will pretend to handle interactions with the database and logging mechanism. As you are going to be passing in such mock objects to a constructor, you need to declare instance variables based on those mock classes (Listing 3.9).

```

PUBLIC SECTION.

PRIVATE SECTION.
  DATA: mo_class_under_test TYPE REF TO ycl_monster_simulator,
         mo_mock_pers_layer TYPE REF TO ycl_mock_pers_layer,
         mo_logger           TYPE REF TO ycl_mock_logger.

```

Listing 3.9 Defining Mock Classes to Be Injected into the Test Class

In Listing 3.6, you saw how in the constructor in production the class would create the real classes, but during a unit test mock classes are passed into the constructor of the class under test by the `SETUP` method, which runs at the start of each test method.

The full list of the data definitions in the test class are shown in Listing 3.10. In addition to the mock classes, there are some global (to a class instance) variables for things such as the input data and the result. It's good to set things up this way because passing parameters in and out of test methods can distract someone looking at the code (e.g., a business expert) from making sure that the names of the test methods reflect what is supposed to be tested.

```

PRIVATE SECTION.
  DATA: mo_class_under_test TYPE REF TO ycl_monster_simulator,
         mo_mock_pers_layer TYPE REF TO ycl_mock_pers_layer,
         mo_logger           TYPE REF TO ycl_mock_logger,
         ms_input_data      TYPE ys_monster_input_data,
         mt_bom_data        TYPE ytt_monster_bom_data,
         md_creator         TYPE string.

```

Listing 3.10 Variables for the Test Class Definition

After defining the data, you now need to say what methods are going to be in the test class.

Defining the `SETUP` Method

The first method to define is always the `SETUP` method, the job of which is to reset the system state so that every test method behaves as if it were the first test method to be run. Therefore, any of those evil global variables knocking about must either be cleared or set to a certain value, and the class under test must be created anew. This is to avoid the so-called *temporal coupling*, in which the result of one test could be influenced by the result of another test. That situation would

cause tests to pass and fail seemingly at random, and you wouldn't know if you were coming or going.

This method cannot have any parameters—you will get an error if you try to give it any—because it must perform the exact same task each time it is run, and importing parameters might change its behavior. The code for defining the `SETUP` method is very simple:

```
METHODS: setup,
```

Defining the Actual Test Methods

After defining the `SETUP` method, you'll move onto defining the actual test methods. At this stage, you haven't actually written any production code (i.e., the code that will actually run in the real application), and all you have is the specification. Therefore, next you're going to create some method definitions with names that will be recognizable to the person who wrote the specification (Listing 3.11). The `FOR TESTING` addition after the method definition says that this method will be run every time you want to run automated tests against your application.

```
return_a_bom_for_a_monster FOR TESTING
make_the_monster_sing FOR TESTING
make_the_monster_dance FOR TESTING
make_the_monster_go_to_france FOR TESTING
```

Listing 3.11 Unit Test Methods

These are the aims of the program to be written (sometimes these are called *use cases*), because you want to be sure the application can perform every one of these functions and perform them without errors. This is why you need the unit tests.

Implementing Helper Methods

The last step in defining the test class is to implement helper methods. The purpose of helper methods is to perform low level tasks for one or more of the actual test methods; in normal classes, the public methods usually contain several small private methods for the same reason. Helper methods usually fall into two categories:

- ▶ Helper methods that contain boilerplate code that you want to hide away (because although you need this code to make the test work, it could be a distraction to someone trying to understand the test)

- ▶ Helper methods that call one or more ABAP statements for the sole purpose of making the core test method read like plain English

Inside each unit test method (the methods that end with `FOR TESTING`), you will have several helper methods with names that have come straight out of the specification. As an example, the specification document says that the main purpose of the program is to return a bill of materials (BOM) for a monster, and you do that by having the user enter various desired monster criteria, which are then used to calculate the BOM according to certain rules. This can be boiled down into three sentences, and you can create helper methods with the same names as those sentences, as shown in Listing 3.12.

```
given_monster_details_entered,
when_bom_is_calculated,
then_resulting_bom_is_correct,
```

Listing 3.12 Defining Helper Methods to Be Used in Test Methods

A method can of course do two things at the same time: hide boiler plate code and make the remaining code look more like English. Ahead, you'll see such an example; you're going to be looking up the BOM again and again in various tests, and you do not want the reader of the test to be distracted by the mechanics of reading from an internal table. In addition, you want the tests to read like English sentences, so if you use a single `IMPORTING` parameter and a `RETURNING` parameter, then you will be able to refer to `BOM_LINE_ITEM(5)`, which, although not quite grammatically perfect, makes it clear enough to the reader what is being referred to. The code for this is shown in Listing 3.13.

```
*-----*
* Helper Methods
*-----*
    bom_line_item
    IMPORTING      id_line_number TYPE sy-tabix
    RETURNING VALUE(rs_bom_line)  LIKE LINE OF mt_bom_table.

ENDCLASS. "Test Class Definition
```

Listing 3.13 Helper Method to Query a BOM Line Item

3.3.2 Implementing Test Classes

Now that you've defined the test class, you can go ahead with the process of implementing it. At the end of this step, you'll be able to show the business

expert who wrote the initial specification that you've made the specification into a program that does what it says on the side of the box.

Given this, each implementation of a test method should look like it jumped straight out of the pages of the specification and landed inside the ABAP Editor (Listing 3.14).

```
METHOD return_a_bom_for_a_monster.
    given_monster_details_entered( ).
    when_bom_is_calculated( ).
    then_resulting_bom_is_correct( ).
ENDMETHOD. "Return a BOM for a Monster (Test Class)
```

Listing 3.14 Implementation of a Test Class

The steps in implementing the test classes are as follows:

1. Setting up the test.
2. Preparing the test data.
3. Calling the production code to be tested.
4. Evaluating the test result.

Step One: Setting up the Test

To start, set up the class under test in the normal way by manually creating lots of small objects and passing them into the constructor method, in order to get the general idea of constructor injection. Later on, you will find out how to reduce the amount of code needed to do this.

As there is no guarantee about in what order the test methods will run, you want every test method to run as if it were the first test method run, to avoid tests affecting each other. Therefore, when setting up the test, you create each object instance anew and clear all the horrible global variables. This is shown in Listing 3.15.

```
METHOD: setup.
*-----*
* Called before every test
*-----*
    CREATE OBJECT mo_mock_logger.
    CREATE OBJECT mo_mock_monster_pers_layer
```

```

EXPORTING
    io_logger    = mo_logger
    id_valid_on  = sy-datum.

CREATE OBJECT mo_class_under_test
EXPORTING
    id_creator    = md_creator
    io_pers_layer = mo_mock_pers_layer
    io_logger     = mo_mock_logger.

CLEAR: ms_input_data,
       md_creator.

ENDMETHOD. "setup

```

Listing 3.15 Create Class Under Test and Clear Global Variables

At this point, you can be sure that during the test you won't actually read the database or output any sort of log, so you can proceed with the guts of the actual tests. These guts can be divided into the three remaining steps: preparing the test data, calling the production code to be tested, and evaluating the test result.

Step Two: Preparing the Test Data

You now want to create some test data to be used by the method being tested. In real life, these values could come from user input or an external system. Here, you'll just hard-code them (Listing 3.16). Such input data is often taken from real problems that actually occurred in production; for example, a user might have said, "When I created a monster using these values, everything broke down." There could be a large number of values, which is why you hide away the details of the data preparation in a separate method, to avoid distracting anybody reading the main test method.

```

METHOD given_monster_details_entered.

    ms_input_data-monster_strength    = 'HIGH'.
    ms_input_data-monster_brain_size  = 'SMALL'.
    ms_input_data-monster_sanity      = 0.
    ms_input_data-monster_height_in_feet = 9.

    md_creator = 'BARON FRANKENSTEIN'.

ENDMETHOD. "Monster Details Entered - Implementation

```

Listing 3.16 Preparing Test Data by Simulating External Input

Step Three: Calling the Production Code to Be Tested

The time has come to actually invoke the code to be tested: you are calling precisely one method or other type of routine, you pass in your hard-coded dummy values, you call the routine, and you usually get some sort of result back. The important thing is that the routine being called does not know it is being called from a test method; the business logic behaves exactly as it would in production, with the exception that when it interacts with the database or another external system it is really dealing with mock classes.

In this example, when you pass in the hard-coded input data, the real business logic will be executed, and a list of monster components is passed back (Listing 3.17).

```
"WHEN.....
METHOD when_bom_is_calculated.

mo_class_under_test->simulate_bom(
EXPORTING is_bom_input_data = ms_input_data

IMPORTING et_bom           = mt_bom_data ).

ENDMETHOD. "when_bom_is_calculated
```

Listing 3.17 Calling the Production Code to Be Tested

The method that calls the code to be tested should be very short—for example, a call to a single function module or a method—for two reasons:

► **Clarity**

Anyone reading the test code should be able to tell exactly what the input data is, what routine processes this data, and what form the result data comes back in. Calling several methods in a row distracts someone reading the code and makes them have to spend extra time working out what is going on.

► **Ease of maintenance**

You want to hide the implementation details of what is being tested from the test method; this way, even if those implementation details change, the test method does not need to change.

For example, in a procedural program, you may call two or three `PERFORM` statements in a row when it would be better to do a call to a single `FORM` routine—so that if you were to add another `FORM` routine in the real program, you wouldn't have to go and add it to the test method as well. With procedural programs, it would be good if you had a signature with the input and output values, but a lot

of procedural programs work by having all the data in global variables. Such a program can still benefit from unit testing; it just requires more effort (possibly a lot more effort) in setting up the test to make sure the global variables are in the correct state before the test is run.

Step Four: Evaluating the Test Result

Once you have some results back you will want to see if they are correct or not. Next you will learn about the standard way of evaluating such results and then how you can enhance the standard framework when the standard mechanism doesn't do everything you want. Finally, you'll see how to achieve 100% test coverage.

Evaluating Test Results in the Normal Way

When you ran the test method that called the production code, either you got a result back—table `MT_BOM_DATA` in the example—or some sort of publicly accessible member variable of the class under test was updated—a status variable, perhaps. Next, run one or more queries to see if the new state of the application is what you expect it to be in the scenario you're testing. This is done by looking at one or more variable values and performing an evaluation (called an assertion) to compare the actual value with the expected value. If the two values do not match, then the test fails, and you specify the error message to be shown to the person running the test (Listing 3.18).

```
"THEN
METHOD then_resulting_bom_is_correct.
* Local Variables
  DATA: ls_bom_line LIKE LINE OF mt_bom_data.

  ls_bom_line = bom_line_item( 1 ).

  cl_abap_unit_assert=>assert_equals(
    act = ls_bom_line-quantity
    exp = 1
    msg = 'Monster has wrong number of Heads'
    quit = if_aunit_constants=>no ).

  ls_bom_line = bom_line_item( 2 ).

  cl_abap_unit_assert=>assert_equals(
    act = ls_bom_line-quantity
    exp = 2
    msg = 'Monster has wrong number of Arms'
    quit = if_aunit_constants=>no ).
```

```

ls_bom_line = bom_line_item( 3 ).

cl_abap_unit_assert=>assert_equals(
act = ls_bom_line-quantity
exp = 2
msg = 'Monster has wrong number of Legs' ).

```

ENDMETHOD."*Then Resulting BOM is correct - Implementation*

Listing 3.18 Using Assertions to See if the Test Passed

When evaluating the result, you use the standard class `CL_ABAP_UNIT_ASSERT`, which can execute a broad range of tests, not just test for equality; for example, you can test if a number is between 5 and 10. There is no need to go into all the options here. It's easier if you just look at the class definition in SE24. (For a bit more about `ASSERT`, see the box ahead.)

You will see that Listing 3.18 includes a helper method; all this does is read a given line of an internal table. In this example, that does not make much sense, because you could achieve the same end with one `READ` statement. In real life, however, you may well have to do something much more complicated, like extracting data out of a deeply nested structure, and a helper method can hide that complexity so as to focus attention on the test itself.

Multiple Evaluations (Assertions) in One Test Method

Many authors writing about test-driven development have stated that you should have only one `ASSERT` statement per test. As an example, you should not have a test method that tests that the correct day of the week is calculated for a given date and at the same time tests whether an error is raised if you don't supply a date. By using only one `ASSERT` statement per test, it is easier for you to quickly drill down into what is going wrong.

I would change that rule slightly, such that you are only testing one outcome per test. Although in this case your test might need several `ASSERT` statements to make sure it is correct, the fact that you are only testing one outcome will still make it easy to figure out what's wrong. The default behavior in a unit test in ABAP, however, is to stop the test method at the first failed assertion and not even execute subsequent assertions within the same method. Every test method will be called, even if some fail—but only the first assertion in each method will be checked.

You can avoid this problem by setting an input parameter. You will see in Listing 3.18 that there are three assertions. By adding the following code, you can make the method continue with subsequent assertions even if one fails:

```
quit = if_aunit_constants=>no
```

Defining Custom Evaluations of Test Results

The methods supplied inside `CL_ABAP_UNIT_ASSERT` are fine in 99% of cases, but note there is an `ASSERT_THAT` option that lets you define your own type of test on the result. Look at an example of specialized assertion `ASSERT_THAT` in action. First, create a local class that implements the interface needed when using the `ASSERT_THAT` method (Listing 3.19).

```
CLASS lcl_my_constraint DEFINITION.

    PUBLIC SECTION.
        INTERFACES if_constraint.

ENDCLASS.                                "lcl_my_constraint DEFINITION
```

Listing 3.19 ASSERT_THAT

You have to implement both methods in the interface (naturally); one performs whatever tests you feel like on the data being passed in, and the other wants a detailed message back saying what went wrong in the event of failure. In real life, you would have a member variable to pass information from the check method to the result method, but the example here just demonstrates the basic principle. The example is shown in Listing 3.20.

```
CLASS lcl_my_constraint IMPLEMENTATION.

    METHOD if_constraint~is_valid.
*-----*
* IMPORTING data_object TYPE data
* RETURNING result      TYPE abap_bool
*-----*
* Local Variables
    DATA: ld_string TYPE string.

    ld_string = data_object.

    result = abap_false.

    CHECK ld_string CS 'SCARY'.

    CHECK strlen( ld_string ) GT 5.

    CHECK ld_string NS 'FLUFFY'.

    result = abap_true.

ENDMETHOD.                                "IF_CONSTRAINT~is_valid
```

```

METHOD if_constraint~get_description.
*-----*
* RETURNING result TYPE string_table
*-----*
* Local Variables
  DATA: ld_message TYPE string.

  ld_message = 'Monster is not really that scary'.

  APPEND ld_message TO result.

ENDMETHOD.                "IF_CONSTRAINT~get_description
ENDCLASS. "My Constraint - Implementation

```

Listing 3.20 Implementation

All that remains is to call the assertion in the THEN part of a test method, as shown in Listing 3.21.

```

DATA: lo_constraint TYPE REF TO lcl_my_constraint.

CREATE OBJECT lo_constraint.

cl_abap_unit_assert=>assert_that(
  exp = abap_true
  act = lo_constraint->is_valid( lo_monster->scariness )
  msg = lo_constraint->get_description( ) ).

```

Listing 3.21 Call the Assertion

As you can see, the only limit on what sort of evaluations you can run on the test results is your own imagination.

Achieving 100% Test Coverage

When evaluating your results, you want to make sure that you've achieved 100% test coverage. In the same way that the US army wants no man left behind, you want no line of code to remain untested. That is, if you have an IF statement with lots of ELSE clauses or a CASE statement with 10 different branches, then you want our tests to be such that every possible path gets followed at some point during the execution of the test classes, to be sure nothing ends in tears and causes an error of some sort.

That's not as easy as it sounds. Aside from the fact that you need a lot of tests, how can you be sure you have not forgotten some branches? Luckily, there is tool

support for this. From release 7.31 of ABAP, you can follow the menu path LOCAL TEST CLASSES • EXECUTE • EXECUTE TESTS WITH • CODE COVERAGE. As mentioned earlier in the book, the same feature is available with ABAP in Eclipse.

3.4 Automating the Test Process

In the example presented in this chapter, the code was deliberately simple in order to highlight the basic principles without getting bogged down with unnecessary detail. However, in the real world programs are never simple. Even if they start off simple, they keep growing and mutating, the ground of the original purpose getting buried under the ever-falling snow of new functionality.

If you think about what you have read above, you will see that this can lead to quite a large effort in writing the test code, in two areas:

- ▶ **Setting up the class under test via the SETUP method**

Well-designed object-oriented programs use lots of small reusable classes, and these often need to be inserted into the main class during construction. The smaller classes often need still smaller classes inserted into them during their construction, and so on. In practical terms, this can mean a lot of lines of code to build up your test class, passing in assorted mock doubles as well as elementary data parameters, such as date, and organizational elements, such as plant or sales organization.

- ▶ **Creating mock objects**

Although you need mock objects to take the place of real objects during a test, in order to avoid actual database access and the like it can take a good deal of effort to create these and to write the logic inside them to return hard-coded values. Even worse, if you are passing in constructor objects as interfaces as opposed to actual classes (which all the OO experts recommend), then you need to expend even more effort, because you have to create an implementation for every method in your mock object that uses the interface, including the methods you are not interested in.

Luckily, there are solutions for both problems. Neither are anything you will find in the SAP standard, but both solutions use the multitude of tools available within the ABAP environment.

Ahead, you'll read about two frameworks to solve those problems:

- ▶ A framework to automate dependency injection (that I created myself)
- ▶ A framework to automate creating mock objects (called mockA)

The section will end with an examination of how to combine both techniques at once.

3.4.1 Automating Dependency Injection

The main class under test needs two object instances to be passed into it during construction, and one of those objects needs some parameters itself. In other words, the main class has *dependencies*: the main class depends on the fact that those object instances need to be created before it itself can be created.

In real life, this could lead to dozens of lines of code in your `SETUP` method, creating objects all over the place. This is bad, because the more `CREATE OBJECT` statements you have in your program, the less flexible it becomes. The logic goes as follows: Creating an object via `CREATE OBJECT` forces the resulting instance to be of a specific class. Having objects of a specific class makes your program less resistant to change. If your main class contains lots of little helper classes—as it should—then you may need to have a great deal of `CREATE OBJECT` statements, meaning that your program is full of rigid components.

So you have two problems: First, you have to clutter up your code with a large number of `CREATE OBJECT` statements. Second, those `CREATE OBJECT` statements usually create instances of a hard-coded class type rather than a dynamically determined subclass.

Now, you are always going to have to state the specific subclass you want somewhere. However, you will see that it is possible to decouple this from the exact instant you call the `CREATE OBJECT` statement, and, as a beneficial by-product, have fewer `CREATE OBJECT` statements in the first place.

The process of passing the objects a class needs to create itself is known as *dependency injection*; you saw this at work earlier in the chapter when you manually passed mock objects into a constructor method. Here you seek to automate such a process by dynamically working out what objects (dependencies) an instance of the main class needs to create itself and creating and passing those objects in all at once. This will drastically reduce the amount of `CREATE OBJECT` statements—and

while you are going to be using a program to dynamically determine *what* objects need to be created, that same program might as well also dynamically determine *what type* (subclass) those created objects should be.

Listing 3.22 is an example of this approach; this code attempts to achieve the same thing as you saw in the `SETUP` method in Listing 3.15, but with fewer `CREATE OBJECT` statements. If you look at the definition of the `SETUP` method earlier in the chapter, you'll see three `CREATE OBJECT` statements, and in each case the object you're creating has to be defined by a `DATA` statement. (Some objects also need data parameters that are elementary, my dear Watson.)

Listing 3.22 rewrites the same `SETUP` method, using a Z class created to use dependency injection. Look at this like a newspaper: look at the four high level method calls first (which represent the headlines), and then dive into the implementation of each component method (representing the actual newspaper article) one at a time.

First, set some values for elementary data parameters. Then, specify any subclasses you want to substitute for real classes. Finally, create an instance of the class under test.

```
DATA: mo_class_under_test TYPE REF TO ycl_monster_simulator.
```

```
zcl_bc_injector=>during_construction( :
for_parameter = 'ID_CREATOR' use_value = md_creator ),
for_parameter = 'ID_VALID_ON' use_value = sy-datum ).
```

"We want to use a test double for the database object

```
zcl_bc_injector=>instead_of(
using_main_class = 'YCL_MONSTER_PERS_LAYER'
use_sub_class    = 'YCL MOCK_MONSTER_PERS_LAYER' ).
```

```
zcl_bc_injector=>instead_of(
using_main_class = 'YCL_LOGGER'
use_sub_class    = 'YCL MOCK_LOGGER' ).
```

```
zcl_bc_injector=>create_via_injection(
CHANGING co_object = mo_class_under_test ).
```

Listing 3.22 `SETUP` Method Rewritten Using Z Statement

As you can see, when using this approach you do not have to have `DATA` statements to declare the database access object or the logging object. In addition, you only have one `CREATE` statement. This may not seem like much in this simple

example, but the advantage increases proportionally with the complexity of the class being tested.

Note

You can also use this same methodology if the importing parameter of the object constructor is an interface. You just pass the interface name in to the `INSTEAD_OF` method rather than the main class name.

The first method called in Listing 3.22 is the `DURING CONSTRUCTION` method. This is shown in Listing 3.23; it analyzes elementary parameters and then does nothing fancier than adding entries to an internal table.

```
METHOD during_construction.
* Local Variables
DATA: lo_description      TYPE REF TO cl_abap_typedescr,
      ld_dummy           TYPE string ##needed,
      ld_data_element_name TYPE string,
      ls_parameter_values LIKE LINE OF mt_parameter_values.

ls_parameter_values-identifier = for_parameter.

CREATE DATA ls_parameter_values-do_value LIKE use_value.
GET REFERENCE OF use_value INTO ls_parameter_values-do_value.

CHECK sy-subrc = 0.

CALL METHOD cl_abap_structdescr=>describe_by_data_ref
  EXPORTING
    p_data_ref      = ls_parameter_values-do_value
  RECEIVING
    p_descr_ref     = lo_description
  EXCEPTIONS
    reference_is_initial = 1
    OTHERS            = 2.

IF sy-subrc <> 0.
  RETURN.
ENDIF.

SPLIT lo_description->absolute_name AT '=' INTO ld_dummy ld_data_
element_name.

ls_parameter_values-rollname = ld_data_element_name.
```

```

INSERT ls_parameter_values INTO TABLE mt_parameter_values.

ENDMETHOD.

```

Listing 3.23 DURING CONSTRUCTION Method

The next method called as part of the rewritten `SETUP` method is the `INSTEAD_OF` method (Listing 3.24). This method takes in as parameters the subclasses you want to create instead of a super class, and that relation is stored in a hashed table.

```

METHOD instead_of.
* Local Variables
DATA: ls_sub_classes_to_use LIKE LINE OF
      mt_sub_classes_to_use.

ls_sub_classes_to_use-main_class = using_main_class.
ls_sub_classes_to_use-sub_class  = use_sub_class.

"Add entry at the start, so it takes priority over previous
similar entries
INSERT ls_sub_classes_to_use INTO mt_sub_classes_to_use
INDEX 1.

"A specific object instance can be passed in
CHECK with_specific_instance IS SUPPLIED.
CHECK with_specific_instance IS BOUND.

ls_created_objects-clsname = id_class_type_to_create.
ls_created_objects-object  = with_specific_instance.
INSERT ls_created_objects INTO TABLE mt_created_objects.

ENDMETHOD.

```

Listing 3.24 INSTEAD_OF Method

The last part of the rewritten `SETUP` method is the `CREATE_BY_INJECTION` method (Listing 3.25). This is written as close to plain English as possible so that the code is more or less self-explanatory. In essence, you are passing the input values you just stored into the constructor method when creating your class under test and any smaller classes it requires.

```

METHOD create_via_injection.
* Local Variables
DATA: lo_class_in_reference_details
      TYPE REF TO cl_abap_refdescr,
      lo_class_in_type_details

```

```

TYPE REF TO cl_abap_typedescr,
lo_class_to_create_type_detail
TYPE REF TO cl_abap_typedescr,
ld_class_passed_in           TYPE seoclass-clsname,
ld_class_type_to_create      TYPE seoclass-clsname,
ls_created_objects
LIKE LINE OF mt_created_objects,
lt_signature_values         TYPE abap_parmbind_tab.

```

```

* Determine the class type of the reference object that was passed in
lo_class_in_reference_details ?=
cl_abap_refdescr=>describe_by_data( co_object ).
lo_class_in_type_details =
lo_class_in_reference_details->get_referenced_type( ).
ld_class_passed_in =
lo_class_in_type_details->get_relative_name( ).

```

"See if we need to create the real class, or a subclass

```

determine_class_to_create(
  EXPORTING
    id_class_passed_in           = ld_class_passed_in
    io_class_in_type_details     = lo_class_in_type_details
  IMPORTING
    ed_class_type_to_create      = ld_class_type_to_create
    eo_class_to_create_type_detail =
    lo_class_to_create_type_detail ).

```

```

READ TABLE mt_created_objects INTO ls_created_objects
WITH TABLE KEY clsname = ld_class_type_to_create.

```

```

IF sy-subrc = 0.

```

"We already have an instance of this class we can use

```

co_object ?= ls_created_objects-object.

```

```

RETURN.

```

```

ENDIF.

```

*"See if the object we want to create has parameters, and if so,
"fill them up*

```

fill_constructor_parameters(
  EXPORTING io_class_to_create_type_detail =
    lo_class_to_create_type_detail " Class to Create Type Details
  IMPORTING et_signature_values = lt_signature_values ).
  " Constructor Parameters

```

```

create_parameter_object(
  EXPORTING id_class_type_to_create =

```

```

ld_class_type_to_create    " Exact Class to Create
it_signature_values = lt_signature_values " Parameter Values
CHANGING co_object = co_object ).      " Created Object

```

ENDMETHOD."Create Via Injection

Listing 3.25 CREATE_BY_INJECTION Method

If you want to drill into this even more, you can download this code at www.sap-press.com/3680 and run it in debug mode to see what's happening.

In summary, dependency injection provides a way to set up complicated classes while using a lot less code, which will enable you to create test classes with less effort.

Error Handling

There is virtually no error handling in the code just discussed (except for throwing fatal exceptions when unexpected things occur). This could be a lot more elegant—but it is the basic principle of automating dependency injection, not elegance, that is currently our focus.

3.4.2 Automating Mock Object Creation via mockA

Unit testing frameworks have been around for quite some time in other languages, such as Java and C++. ABAP has joined the club rather late in the day. One advantage of this is that ABAP developers can look at problems other languages encountered—and solved—some years ago, and if they find the same problem, then they can implement the same sort of solution without having to reinvent the wheel. Mock objects are a great example of this: Many different mock object frameworks for Java were born to take a lot of the pain out of the process. An SAP developer named Uwe Kunath started an open-source project to achieve the same solution in ABAP and called his project the mockA framework. mockA can be downloaded at <https://github.com/uweku/mockA>.

ABAP Open-Source Projects

There are several times where this book will refer to open-source ABAP projects that started life on the SAP Code Exchange section of the SAP Community Network but nowadays live on sites like GitHub. The obvious benefit of these projects is that they are free. Some development departments have rules against installing such things, but I feel they are just cutting off their nose to spite their face.

The important point to note is that these are not finished products, so installing them is not like installing the sort of SAP add-on you pay for. It is highly likely that you will encounter bugs and that the tool will not do 100% of what you want it to do. In both cases, I strongly encourage you to fix the bug or add the new feature yourself and then update the open-source project so that the whole SAP community benefits.

The very first open-source project you will need to install is SAPlink, which will help you to download any others you need easily. You can get SAPlink at www.saplink.org.

Good object-oriented design recommends that virtually every class has its public signature defined via an interface. The reasons for this are too many and too complicated to go into here, but suffice it to say that this helps you follow the OO principle of favoring composition over inheritance. `mockA` works by using classes that have interfaces describing their signature.

Listing 3.26 adapts the demonstration program in the `mockA` download package to the running example.

```

METHOD mocking_framework_test.
* Local Variables
  DATA: lo_mocker                TYPE REF TO zif_mocka_mockер,
         ld_name                  TYPE abap_abstypename
         VALUE 'YIF_MONSTER_SIMULATOR',
         lo_monster_simulator
         TYPE REF TO yif_monster_simulator,
         lf_has_been_called       TYPE abap_bool,
         ld_scariness             TYPE string.

  lo_mocker = zcl_mocka_mockер=>zif_mocka_mockер~mock(
    ld_name ).

  given_monster_details_entered( ).

  lo_mocker->method( 'CALCULATE_SCARINESS'
  )->with( i_pl = ms_input_data
  )->returns( 'REALLY SCARY' ).

  lo_monster_simulator ?= lo_mocker->generate_mockup( ).

  ld_scariness = lo_monster_simulator->calculate_scariness( ms_input_
data ).

  lf_has_been_called = lo_mocker->has_method_been_called( 'CALCULATE_
SCARINESS' ).

```



```

        cl_abap_unit_assert=>assert_equals(
exp = abap_true
act = lf_has_been_called
msg = 'Test method has not been called' ).

        cl_abap_unit_assert=>assert_equals(
exp = 'REALLY SCARY'
act = ld_scariness
msg = 'Monster is not scary enough' ).

    ENDMETHOD. "Mocking Framework Test

```

Listing 3.26 Coding a Unit Test without Needing Definitions and Implementations

As you can see, the mock object framework does away with the need to create definitions and implementations of the class you want to mock. You only need to focus on what output values should be returned for what input values for what class. This methodology uses the ability of ABAP to generate temporary programs that live only in memory and only exist as long as the mother program is running. In effect, the framework is writing the method definitions and implementations for you at runtime.

Note also the test for `CALCULATE_SCARINESS` in the preceding code. Even if a mock method doesn't do anything at all in a test situation, you still want to be sure that it has been called.

Method Chaining

Listing 3.26 also uses method chaining, a feature we discussed in Chapter 2. Three methods in a row are called on the instance of `LO_MOCKER`: `METHOD`, `WITH`, and `RETURNS`. Before release 7.02 of ABAP, you would have had to use three separate lines here and to repeat the `LO_MOCKER` declaration on each line.

Alternatives to mockA

Without this framework, the way to proceed is to create mock classes that are subclasses of the real class (e.g., `YCL_MOCK_DATABASE_LAYER`), redefine some methods, and put some hard-coded logic inside the redefined method to return certain values based upon input values. You could also create a mock class that implements an interface—but this is even more work, because you have to have an implementation for every method in the interface.

3.4.3 Combining Dependency Injection and mockA

The real value of mockA comes to light when you pass your generated mock object into a larger class that is being tested. To demonstrate this, create a `YCL_MONSTER_LABORATORY` class that takes the `SIMULATOR` object as input and then uses a complex set of business logic to say whether the monster is any good. That business logic is what we want to test. The code is shown in Listing 3.27.

```

METHOD laboratory_test.
* Local Variables
  DATA:
lo_mocker TYPE REF TO zif_mocka_mocker,
ld_name TYPE abap_abstypename VALUE 'YIF_MONSTER_SIMULATOR',
lo_monster_simulator TYPE REF TO yif_monster_simulator,
lf_monster_ok      TYPE abap_bool,
lo_laboratory      TYPE REF TO ycl_monster_laboratory,
lo_mocker_method   TYPE REF TO zif_mocka_mocker_method.

  lo_mocker =
    zcl_mocka_mocker=>zif_mocka_mocker~mock( ld_name ).

  lo_mocker_method = lo_mocker->method( 'CALCULATE_SCARINESS' ).

  given_monster_details_entered( ).

  lo_mocker->method( 'CALCULATE_SCARINESS'
  )->with( i_pl = ms_input_data
  )->returns( 'REALLY SCARY' ).

  lo_monster_simulator ?= lo_mocker->generate_mockup( ).

  CREATE OBJECT lo_laboratory.

  lf_monster_ok = lo_laboratory->evaluate_monster(
  is_bom_input_data = ms_input_data
  io_simulator      = lo_monster_simulator ).

  cl_abap_unit_assert=>assert_equals(
  exp = abap_true
  act = lf_monster_ok
  msg = 'Monster is just not good enough' ).

ENDMETHOD. "Laboratory Test

```

Listing 3.27 Passing In a Mocked Up Interface to a Real Class

In Listing 3.27, large chunks of code that you would normally need have been removed; you didn't need to code either a definition or an implementation for the mock monster simulator class. Nonetheless, the end result is just as good as if

you had gone down the longer route. The laboratory object neither knows nor cares that what has been passed into it is not a real instance of a class but instead a generated mock object.

Take this one step further and combine it with injection: in the injection framework class that is part of Listing 3.27, the method `INSTEAD_OF` has an optional parameter via which you can pass in the generated object. You need this so that you can enable your mockA generated instance to be used when creating the class under test via injection. This is shown in Listing 3.28.

```
DATA: mo_class_under_test TYPE REF TO ycl_monster_laboratory.

... code to set up the mock object, as per above examples ...

lo_monster_simulator ?= lo_mockers->generate_mockup( ).

"We want to use a test double for the database object
zcl_bc_injector=>instead_of(
  using_main_class = 'YIF_MONSTER_SIMULATOR'
  use_sub_class    = 'YCL_MONSTER_SIMULATOR'
  with_specific_instance = lo_monster_simulator ).

zcl_bc_injector=>create_via_injection(
  CHANGING co_object = mo_class_under_test ).
```

Listing 3.28 Combining mockA with Dependency Injection

You're passing in a generated object that will get used when the injection class creates the class under test.

What the examples that culminate in Listing 3.28 show is that it is possible to simplify the `SETUP` method dramatically when creating the class under test. You can use mockA to set up test doubles with a lot less effort and injection to avoid having to code long strings of `CREATE OBJECT` statements that pass the results into each other before handing the end result into the class under test when it's finally created. The two frameworks were not created with the intention of working together, but by a happy accident they fit together like the pieces of a jigsaw puzzle.

3.5 Behavior-Driven Development

This chapter on test-driven development concludes by briefly introducing you to the concept of *behavior-driven development*. This approach is a variation on test-driven development, with more of a business focus; it is the idea that tests should

be managed by both business experts *and* developers. You will notice that the naming conventions for test methods used throughout this chapter adhere to the BDD recommendations that are about to be explained.

When using behavior-driven development, the general recommendation is to start all the test methods with `IT SHOULD`, with the `IT` referring to the application being tested (the class under test). Thus, you would have names such as `IT SHOULD FIRE A NUCLEAR MISSILE`, such names coming straight out of the specification that describes what the program is supposed to achieve.

In ABAP, you are limited to 30 characters for names of methods, variables, database tables, and so on—so you have to use abbreviations, which potentially makes ABAP less readable than languages like Java. In Java, you would have really long test method names, like `It Should Return a BOM For a Purple Monster`, but in ABAP you can't afford to add the extra characters of `IT SHOULD` to the method name. Instead, you can put the `IT SHOULD` in a comment line with dots after it, padding the comment line out to 30 characters to make it really obvious what the maximum permitted length of the names of the test methods are. You will then declare the test methods underneath the dotted line and be aware of when you are running out of space for the name. An example of this is shown in Listing 3.29.

```
*-----*
* Specifications
*-----*
  "IT SHOULD.....
  "User Acceptance Tests
  return_a_bom_for_a_monster    FOR TESTING,
```

Listing 3.29 Test Methods That Describe What an Application Should Do

Other Names for Behavior-Driven Development

Sometimes, these sorts of behavior-driven development unit tests are described as user acceptance tests. Although there is no actual user involved, the reason for the terminology is that this sort of test simulates the program exhibiting a behavior that the user would expect when he performs a certain action within the program. Outside of SAP, the testing framework called FitNesse describes itself as such an automated user acceptance testing framework.

You may also see behavior-driven development referred to as the Assemble/Act/Assert way of testing (which doesn't read as much like natural language, but it makes some people very happy, because every word starts with the same letter).

Whatever you want to call these types of automated tests, they usually involve several methods—and often several classes as well—all working together. An example of this is shown in Listing 3.30.

```
*-----*
* Low-Level Test Implementation Methods
*-----*
    "GIVEN.....
    given_monster_details_entered,
    "WHEN.....
    when_bom_is_calculated,
    "THEN.....
    then_resulting_bom_is_correct,
```

Listing 3.30 The GIVEN/WHEN/THEN Pattern for Unit Tests

As you can see in the preceding code, unit test methods that follow the behavior-driven development approach have three parts:

- ▶ GIVEN describes the state of the system just before the test to be run.
- ▶ WHEN calls the actual production code you want to test.
- ▶ THEN uses ASSERT methods to test the state of the system after the class under test has been run.

Use Natural Language

A test method is supposed to be able to be viewed by business experts to see if the test matches their specifications, so it has to read like natural language. Often, if business experts see even one line of ABAP, their eyes glaze over and you have lost them for good.

3.6 Summary

Mountain climbers will tell you their pastime is not easy, but it is all worth it once you have achieved the incredibly difficult task of climbing the mountain and are standing on the summit, on top of the world, able to see for miles. It may not seem similar on the surface, but unit testing is like that. It's not easy at all—quite the reverse—but once you have enabled your existing programs with full test coverage and you create all new programs using this methodology, then you too suddenly have a much-improved view.

Quite simply, you can make any changes you want to—radical changes—introduce new technology, totally refactor (redesign) the innards of the program, anything at all, and after you change even one line of code you can follow the menu path TEST • UNIT TEST and know—within seconds—if you have broken any existing functions. This is not to be sneezed at. It is in fact the Holy Grail of programming.

One question that has not come up yet is this: What if someone else comes along and changes your program by adding a new feature, but accidentally breaks something else and doesn't bother to run the unit tests, and thus doesn't realize that he's broken something? Clearly, you somehow need to embed the automated unit tests into the whole change control procedure. Conveniently, this leads nicely to the subject of the next chapter: the ABAP Test Cockpit.

Recommended Reading

- ▶ *Head First Design Patterns* (Freeman et al., O'Reilly Media, 2004)
- ▶ Behavior-Driven Development: <http://dannorth.net/introducing-bdd> (Dan North)
- ▶ The Art of Unit Testing: <http://artofunittesting.com> (Roy Osherove)
- ▶ Dependency Injection: <http://scn.sap.com/community/abap/blog/2013/08/28/dependency-injection-for-abap> (Jack Stewart)
- ▶ mockA: <http://uwekunath.wordpress.com/2013/10/16/mocka-released-a-new-abap-mocking-framework> (Uwe Kunath)

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

—Gerald Weinberg

4 ABAP Test Cockpit

Unless you are a newcomer to working with SAP systems, you'll be familiar with the concept of Early Watch reports, which assess the performance of your system with the goal of identifying potential problems before they impact you. One of the major focuses of these reports is the amount of custom code in your system, which is illustrated with a pie chart. The reason for this is that traditionally custom code has been far dodgier than standard SAP code, and the higher the percentage of dodgy code in your system, the more in danger you are. Although it's easy as a non-SAP developer to feel offended by this, SAP does have a point. Many companies have ended up in serious trouble over custom code quality.

To prevent potential problems with custom code, customers were supplied with the Extended Program Check (Transaction SLIN), which can be called from Transaction SE80 and its friends (SE38, SE37, SE24, etc.) to highlight a whole raft of common problems, such as unused variables and procedures, nontyped parameters, and the like. This functionality was later extended via the Code Inspector (Transaction SCI), which also looks at performance issues—for example, a `SELECT` statement with no `WHERE` clause. In even later releases, this also included various usability checks of DYNPRO screens (at the exact same time SAP started telling you not to use DYNPRO screens).

DYNPRO vs. Web Dynpro ABAP

This book uses the capitalized term "DYNPRO" in order to distinguish classic DYNPRO screens from Web Dynpro ABAP. (Chapter 12 discusses Web Dynpro ABAP in more detail.)

This is all wonderful—if these tools are used. However, all too often, the real-world situation is different from the ideal, and the tools are overlooked. SAP had

clearly thought about this sort of situation, and as a result the ABAP Test Cockpit (ATC) was born, which is intended to make such checks mandatory in the development process. The ATC was built based on the principle of *separation of concerns*: one part of a program does one job only and—here's the important bit—does it well. When designing the ATC, SAP had two roles in mind, each with a different concern: the ABAP developer, who writes the programs, and a so-called quality manager, who makes sure the programmers are doing their jobs well. If your company is in such a state that errors like short dumps keep popping up in production, then you most likely do need some sort of central quality management to get to the bottom of this. The aim of the game though is to get all your developers to run so many tests in development—as a matter of routine—that some sort of central quality management is no longer needed, at least on a micro-management level.

The ATC was made available in SAP NetWeaver 7.02 SP 12 and SAP NetWeaver 7.31 SP 5, and it comes as standard in SAP NetWeaver 7.4. Can you guess what this means, dear reader? It's time you learned how to use it.

A cockpit by its very nature is not a tool in itself; it is an easier way to control several underlying tools, and hopefully the whole is greater than the sum of the parts. Any extra static checks on code SAP comes up with are not added to the ATC but rather to the underlying components—for example, the Code Inspector—or even to totally new components that are added with higher support stack levels. However, having a cockpit makes such changes with each new release or support stack much more transparent and thus easier to take advantage of.

Because the actual components of the ATC are not new in themselves, the purpose of this chapter is to look at the most useful functionalities that the cockpit itself can perform that the individual components can't do on their own. The chapter will start with a discussion of automatic runs of unit tests (Section 4.1) and move on to mass checks (Section 4.2). Finally, Section 4.3 will outline some of the most recent enhancements to ABAP Test Cockpit so that you can make sure that you're taking advantage of all the checks that are available to you.

SAP HANA Migration

Please note that many of the functionalities discussed in this chapter—particularly some of the more recent ABAP Test Cockpit enhancements—become especially useful if you're migrating to SAP HANA. However, you should consider using them in all databases!

4.1 Automatic Run of Unit Tests

In Chapter 3, you learned about test-driven development and how after making some changes to a program you can use ABAP Unit for a regression test to make sure you haven't broken anything else inside the program being changed. However, in your average SAP system custom programs can be very inter-dependent. Even though the unit test on one program passes, how do you know that the changes have not messed up a seemingly unrelated program that depends on the object you have just changed?

Good news: ATC has functionality for that. In Transaction SCI, take the CHECK VARIANT option and then expand the following nodes: Checks, Dynamic Checks, and ABAP Unit. The screen in Figure 4.1 appears. Select the checkbox by ABAP UNIT, and click the green arrow next to it. A pop-up box (which is also shown in the screenshot) appears with ABAP Unit–related options; it's best to stick with the default values. (Remember that in Chapter 3 you learned that unit tests will generally all have the same settings.)

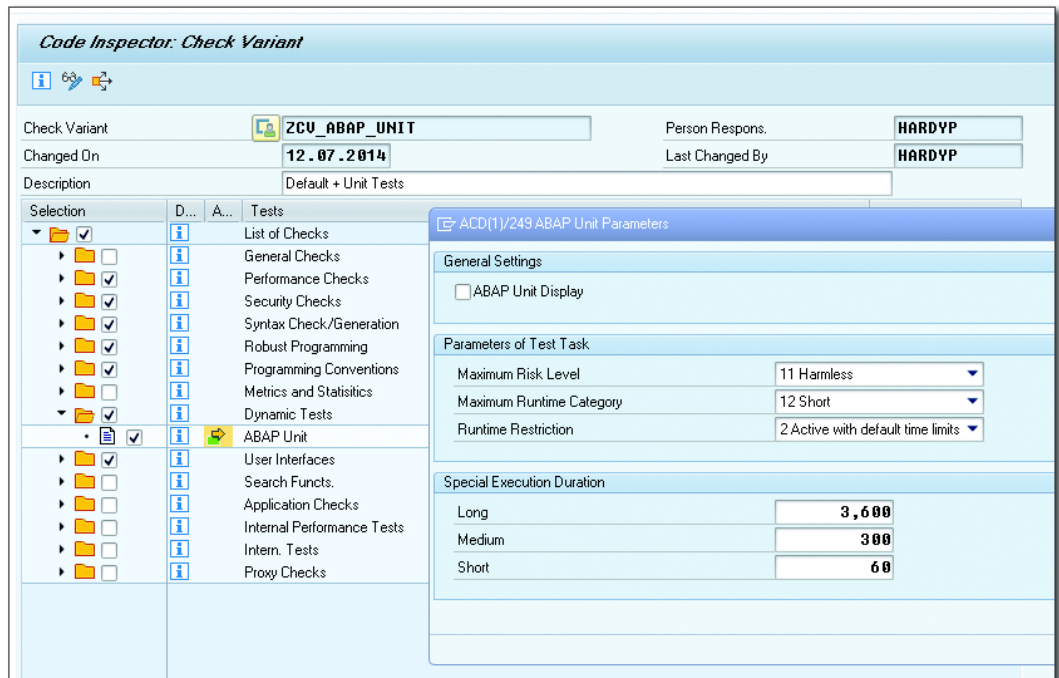


Figure 4.1 Code Inspector Variant with Unit Tests

Once you have saved your settings, this ensures that, during a Code Inspector check, the unit test will run along with the static code checks and amalgamate the results. The result of such a test is shown in Figure 4.2.

ATC: YCL_MONSTER_SIMULATOR (Open Findings)

Pri.	Check Title	Check Message	Object Name	Obj.	Contact Person	Package
1	ABAP Unit	Critical alert	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
1	ABAP Unit	Critical alert	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
1	ABAP Unit	Critical alert	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
1	ABAP Unit	Critical alert	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
2	Extended Program Check	BREAK-POINT Statement	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
3	Extended Program Check	Security Check Without Activated Licence	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
3	Extended Program Check	Text Element Missing in a Character String	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP
3	Extended Program Check	Text Element Missing in a Character String	YCL_MONSTER_SIMULATOR	CLAS	HARDYP	\$TMP

Figure 4.2 ATC: Amalgamated Result List

In the same way that the ATC can perform static code checks on multiple programs at once, it can also do unit test runs on the same set of objects. Provided you have a lot of unit tests in your custom programs, this can be enormously beneficial as a regression test.

RS_AUCV_RUNNER

If you are only interested in the results of unit tests, report `RS_AUCV_RUNNER` can be used to run a batch job for mass checks of unit tests and emailing the results; this report became available in the 7.02 system.

4.2 Mass Checks

Usually, when creating development objects (executable programs, function modules, and classes), you would perform an extended syntax check and then a Code Inspector analysis when you are finished with your changes and are signaling that they are ready for transport to the test system.

This is all good, but it's possible that if (for example) you have added a new obligatory import parameter to a function module or method (which you should not really do) or changed the data type of such a parameter, then you might break a

program that was written a long, long time ago in a galaxy far, far away that is nonetheless still in productive use in your system. The only way you can be 100% certain that a change to an object does not break dependent objects is to check the whole custom code base. Fortunately, ATC can help you with this.

The ATC is based upon the Code Inspector, which is Transaction SCI. In Transaction SCI, you can define a so-called object set (Figure 4.3), in which you can select a whole bunch of objects—everything in your custom name range, if you so desire.

Code Inspector: Object Set

Object Set: **ZOS_ALL_THINGS_CUSTOM** | Vers.: **001** | Person Responsible: **HARDYP**

Deleted On: | Changed On: **12.07.2014** | Last Changed By: **HARDYP**

Description: All Custom Objects | Number of Elements: 0

Save Selections Only:

Object Assignment

Component ID		to		
Software Component		to		
Transport Layer		to		
Package	<input checked="" type="checkbox"/> Z*	to		
Original System		to		
Person Responsible		to		

Object Selection

Classes, Func. Groups... | Free Obj. Choice

<input checked="" type="checkbox"/> Class/Interface	<input checked="" type="checkbox"/> Z*	to		
<input checked="" type="checkbox"/> Function Group	<input checked="" type="checkbox"/> Z*	to		
<input checked="" type="checkbox"/> Program	<input checked="" type="checkbox"/> Z*	to		
<input checked="" type="checkbox"/> Web Dynpro Component	<input checked="" type="checkbox"/> Z*	to		
<input checked="" type="checkbox"/> Dictionary Type	<input checked="" type="checkbox"/> Z*	to		
<input checked="" type="checkbox"/> Type Group	<input checked="" type="checkbox"/> Z*	to		

Figure 4.3 Code Inspector Object Set

With this tool, you can make sure you have not broken anything after you've changed custom objects. In this section, you'll learn how to run these mass checks yourself in development so that you'll find your mistakes before anyone else does. That will help your reputation to no end.

4.2.1 Setting Up Mass Checks

In an ideal world, you would run a mass check on the custom development objects in the test system on a regular basis via a batch job. In order to achieve this, there is some configuration that needs to be done, both in the master (test) system and in every development system that can transport objects to the test system. Start at the beginning: Transaction ATC. When you run this transaction, you'll see a nice menu tree on the left, with pretty icons related to setting up configuration and scheduling batch jobs (Figure 4.4).

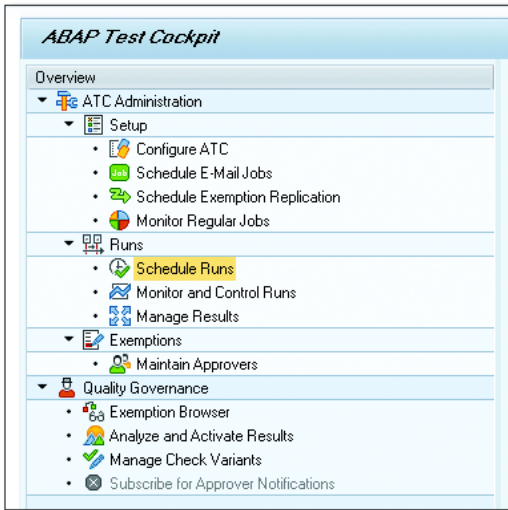


Figure 4.4 Transaction ATC

Start by configuring the master/test system. Follow the menu path for general configuration—ATC ADMINISTRATION • SETUP • CONFIGURE ATC—which has the VARIANT CONFIGURATION icon. Figure 4.5 shows the screen for setting up configuration.

In the GLOBAL CHECK VARIANT field, specify the Code Inspector variant that is to be used by the ATC mass check. As might be expected, this defaults to DEFAULT. However, most companies copy the DEFAULT variant to a Z version via Transaction SCI and change some of the checks that are run. I strongly recommend doing this; naturally, you want to make sure that all the checks you are interested in are active (for example, unit tests). Note that your Z variant has to be a global variant before it appears in the ATC list.

ATC Change Settings

Code Inspector

Global Check Variant:

Exemptions

Do You Want to Enable ATC Exemptions in the System?

No

Yes

For Which Kind of Results?

For Central Results Only

For Any Results

Master System

System ID:

RFC-Destination:

RFC-Dest. (Approval):

Transport Tool Integration

Transport Settings: Checks on transport release depend on user defaults

ATC

Behavior on Release:

Code Inspector

Behavior on Release:

Figure 4.5 ATC Configuration Settings

The next two options give you the opportunity to control the behavior of exemptions. You'll learn about these in detail slightly later on in this section. For now, select YES in answer to DO YOU WANT TO ENABLE ATC EXEMPTIONS IN THE SYSTEM? and select FOR CENTRAL RESULTS ONLY in answer to FOR WHICH KIND OF RESULTS?

The next boxes relate to defining which SAP system is the master (test) system. In the MASTER SYSTEM screen area, fill in the SYSTEM ID field and the RFC DESTINATION field. The remote connection is used for exemption handling.

Lastly, but most importantly, you define the level of integration with the transport system. This is not really that new a concept—it has been available for some time—but it is worth stressing, because a lot of companies are blissfully unaware of this functionality. There are two boxes here: one related to the ATC and one

related to the Code Inspector. What you're doing is defining which tool is going to be called when someone tries to release a transport; that is, you have to choose either the Code Inspector or the ATC as the transport release check tool by disabling the one you don't want.

For this example, select DISABLE "CODE INSPECTOR" AS TEST DRIVER, because SAP recommends that you use the ATC. You could have chosen the Code Inspector as the tool to use by setting the ATC box to DISABLE "ATC" AS TEST DRIVER.

In the dropdown box for whichever tool you have selected, there are two options aside from DISABLE. You can select INFORM ON ERROR, which means that when someone is trying to release a transport, the ATC or Code Inspector is run; the user gets a pop-up box with the warnings and errors, and the release can be aborted if the user wants. If you choose BLOCK ON ERROR, then the same pop-up box appears, but the transport cannot be released if any priority one errors remain.

Once you've set up mass checks, it's time to run them and review them.

4.2.2 Running Mass Checks

To set up a periodic batch job to review all custom objects, you once again call up good old Transaction ATC and follow the menu path ATC ADMINISTRATION • RUNS • SCHEDULE RUNS • CREATE. You will be asked for a name. Enter something like "ZSC_MONSTERS", click the green checkmark, and then you'll see the screen shown in Figure 4.6.

The screenshot shows the 'Configuration: Edit' screen for defining a mass check run. The screen is divided into several sections:

- Check Run:** Description field contains the value `&SYS&: &DOW&, Cw&Cw& &YEAR&`.
- Checks:** Check Variant dropdown is set to **DEFAULT**. There is an unchecked checkbox labeled 'Report Findings Exempted in Code by Pragma or Pseudo Comment'.
- Object Selection:** Two radio buttons are present: 'By Query' (unchecked) and 'By Object Set' (checked).
- Object Set:** CI Obj. Set Variant dropdown is set to **ZOS_MONSTERS**.

Figure 4.6 Defining a Mass Check Run

In the DESCRIPTION field, at the top you will see some variables (for example, &DOW&); these get dynamically replaced by the day of week when the job runs. That is, if you define the job as &DOW& and set it to run every day, then the first run will be called MONDAY, the second TUESDAY, and so forth. You can get a list of such variables by pressing **F1** while in the DESCRIPTION field; there are only four, and they are all shown in Figure 4.6.

Moving down the screen, the next step is to define the CHECK VARIANT field. The CHECK VARIANT field controls what sort of checks are going to be run against the custom objects (a possible check variant definition can be seen in Figure 4.1)—for example, whether you check security, performance, or everything possible.

Finally, you fill out the OBJECT SELECTION area of the screen, which controls what custom objects are going to have the checks defined in the check variant run against them. You can choose BY QUERY to enter objects directly—all Z packages, for example—or BY OBJECT SET to reuse a Code Inspector object set (such an object set can be seen in Figure 4.3). The final box at the bottom of the screen changes deepening on which radio button you choose; for example, if you choose BY OBJECT SET, then the OBJECT SET selection box appears, and you then have to define that set in the OBJ. SET VARIANT field. Otherwise, the OBJECT SELECTION DETAILS box appears, in which you say what packages you are interested in. As an example, you could have your ALL_Z_OBJECTS variant running every month, COMMONLY_USED_OBJECTS running once a week, and VITALLY_IMPORTANT_OBJECTS running every day.

After you save your entry, you are still on the screen that is accessed from Transaction ATC via the menu path ATC ADMINISTRATION • RUNS • SCHEDULE RUNS. This screen has the title RUN SERIES: BROWSE CONFIGURATIONS at the top. On this screen, you have a list of possible mass check runs that you can schedule. Click the job definition you've just created (e.g., ZCS_MONSTERS) and then click the SCHEDULE icon, which looks like a blue wheel; the screen in Figure 4.7 appears.

There are three boxes here: HEADER, EXECUTION, and POSTPROCESSING. In the HEADER box, you choose the SERIES CONFIG. NAME for the batch job that you defined in the screen shown in Figure 4.6. Then, there are some checkboxes to make this "official" (regular jobs scheduled by an administrator), as opposed to a "local" ad hoc check, which is performed by a developer when he's changed something and wants to make sure he hasn't broken anything related. SET TO ACTIVE RESULT means that after this check run has been executed you will see a

lightbulb icon next to it on any list of check runs, indicating that it's the latest and greatest run and that the others are obsolete.

Schedule run series

Header

Series Config. Name: ZCS_MONSTERS

Central Check Run

Set to Active Result

Life Span in Days: 120

Execution

Number Processes: 10

Schedule on Single Server

Schedule on Server Group

Server Name: MY_SERVER_NAME

With Trace (Level 0)

Postprocessing

Distribute Result to: DEV_RFC_DESTINATION

Figure 4.7 Scheduling a Mass Check

Finally, there's the life span in days—here, for example, the results of the ATC check run are deleted after 120 days. Opinions vary, but you probably don't want to make this too long. The idea is to fix everything up ASAP, so why would you be interested in what bugs were in the code three months ago? Conversely, some people like to have a history to see if things are getting better or worse over time.

The EXECUTION area is all about the settings needed when running an intensive batch job, which is what a mass check is. This is just like the settings you make when creating batch jobs to process large amounts of IDocs. Because this job is not supposed to run in production, you're more or less safe to accept the default settings proposed, but if you have a really massive set of objects, then you may need to fiddle with them. The idea is that instead of running a batch job that takes forever, you split up processing—so in effect you have several batch jobs running concurrently.

The first field in the EXECUTION area, NUMBER PROCESSES, refers to the maximum number of work processes that will run in parallel. Therefore, if you have 10

objects and set a value of 10 here, then instead of analyzing each object one after the other in one work process all ten will be analyzed at the same time, each in its own work process. Because there are only a limited number of work processes available at any one time, setting this to a high number and running a huge check during the day will paralyze the system.

The two radio buttons, `SCHEDULE ON SINGLE SERVER` and `SCHEDULE ON SERVER GROUP`, only make sense if the system on which you are going to run the check has more than one application server. If there is only one server, then that is where the job will run; if there are multiple servers, then you can either nominate a specific server (maybe one no actual users log on to) or split the processing between two or more servers so that no individual server is starved of work processes.

Finally, in the `POSTPROCESSING` area, select the RFC destination of the development systems where the results should be seen via Transaction SE80.

Now, you're ready to go! If you want this to be a regular job, choose the menu option `PROGRAM • EXECUTE IN BACKGROUND` and enter the interval, just as you would for any other batch job. Otherwise, press `F8` (or the icon that looks like a ball with a checkmark inside of it), and the job is started in the background immediately as a one-off run.

4.2.3 Reviewing Mass Checks

In recent years, the number of different tools in the `DEVELOPER OPTIONS` part of SE80 has ballooned out, so you have 10 billion tabs along the top to choose from. The ATC adds yet another option; you can now change your settings by going to SE80 and following the menu path `UTILITIES • SETTINGS • ABAP TEST COCKPIT` to control how you see ATC results (Figure 4.8). You can accept the defaults, but you most likely want to set the flag `FINDINGS ASSIGNED TO ME`, which is in the box called `RESULT BROWSER`. Otherwise, you will see objects from all the developers, which would probably be a bit overwhelming.

While you're at it, go to the `WORKBENCH` tab and select the `ATC RESULT BROWSER` checkbox. This gives you another option to see the ATC results (Figure 4.9). Selecting the `ATC RESULT BROWSER` checkbox means that you'll see a new entry in the top-left-hand corner of SE80 called `ATC RESULT BROWSER`, as can be seen at the top of Figure 4.10.

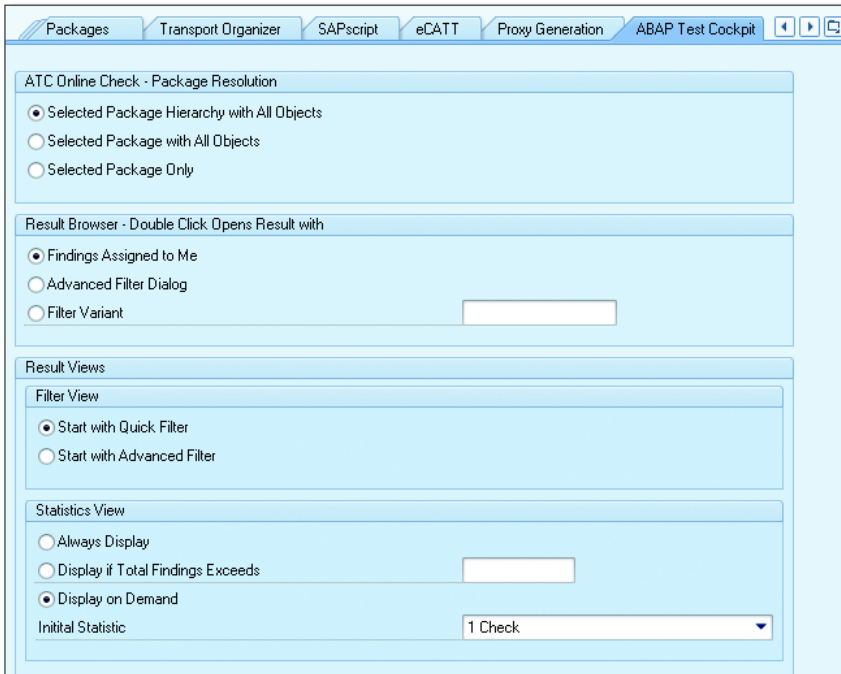


Figure 4.8 Developer Settings: ATC Display Options

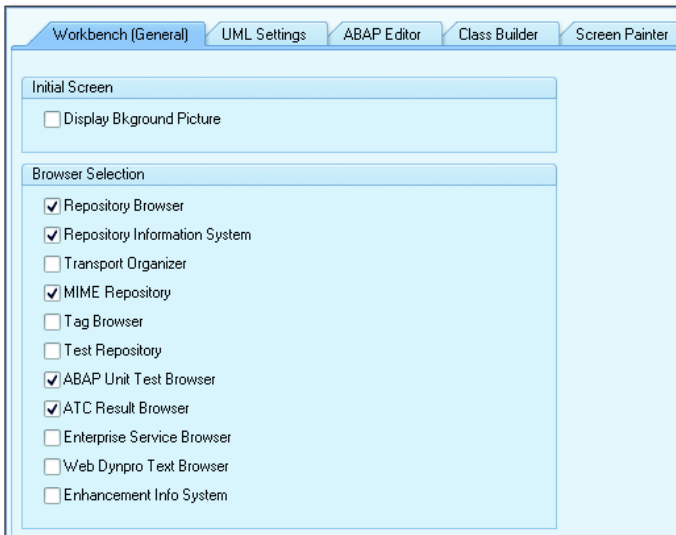


Figure 4.9 Developer Settings: Workbench

Next, go into SE80 to have a look at the result. In Figure 4.10, you'll see the list of batch jobs; the most recent has a pretty light bulb icon beside it.

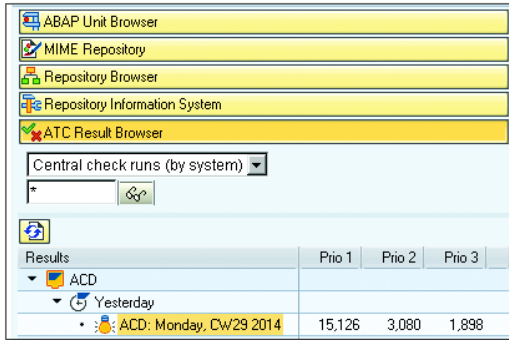


Figure 4.10 SE80: Browsing ATC Check Runs

You can then drill into each batch job to see the ATC results. The Extended Program Check and the Code Inspector have always shown the results in a tree structure. With the ATC, the result list can be quite massive, and so the result screen looks quite different; in essence, it's a large ALV grid, which you can filter based on your needs (refer back to Figure 4.2).

When you double-click a line to get the exact details of the error, the resulting display that appears in the bottom of the screen looks rather different than what you may be used to (Figure 4.11). The display uses an ABAP HTML control to look more like a web page than the SAP GUI, though it's still SAP gray around the edges.

Details			
	Location / Finding	Description	Contact Person
Package	\$TMP		
Class Pool	YCL_MONSTER_SIMULATOR	Monster Simulator	HARDYP
Include	YCL_MONSTER_SIMULATOR=====CCAU		HARDYP
Line Number	114		
Check Title	Extended Program Check		
Check Message	Text element missing in a character string		
Priority	Priority 3		
Found on	12.07.2014 07:06:42		
Strings without text elements are not translated:			
'Monster is not really that scary'			
Display Object	Exemptions disabled by system setup		

Figure 4.11 ATC Detailed Error Display

The ATC error detailed display is more or less the same as the Code Inspector equivalent. You can still navigate to the custom object via the DISPLAY OBJECT

option shown at the bottom left of Figure 4.11. However, as opposed to the Code Inspector equivalent in the ATC screen, there are two documentation lines:

- ▶ CHECK TITLE
Gives a description of the general group of errors you're looking at with a list of the individual checks.
- ▶ CHECK MESSAGE
Gives a description of the specific check that failed. In lower releases, this seems not to work all the time; you sometimes only get the description of the group of checks.

4.2.4 Dismissing False Errors

If you look back briefly at Figure 4.10, you will notice that it shows over 15,000 priority one errors just from one developer. Unfortunately, when you first run this for yourself you are likely to get such a result, and some of the check messages say dire things, like VERY SERIOUS ERROR. When you first see this, you may be horrified and want to jump off a bridge before anyone else discovers the woe-ful state of your code.

In fact, things are in no way as bad as they may appear at first. Obviously, every error message needs to be looked at, but such messages appear in groups of similar errors. Although some problems are crying out to be fixed, some of the "errors" are not in fact problems at all, but just the check tool being overzealous.

This brings us to the subject of false errors and, more specifically, false positives. A *false positive* is an alert that an error has occurred even though everything is actually fine. To take an example from the real world, it was a false positive when, in the former Soviet Union, a pigeon flew into a radar dish and triggered a warning saying that the United States had just launched a massive nuclear attack. (The rules, which the communication officer broke, said that he should have informed his superiors—who would have instructed him to counterattack, thus causing a nuclear war that would have destroyed the world. So it's lucky he didn't trust his country's own technology.)

Fortunately, in SAP terms, false positives are not that drastic. A common situation in which a false positive may occur is when you have a variable that should sometimes, but not always, be changed based on the value of another variable. In order to make sure that this variable is changed as required, you could use a

CASE statement, like the one in Listing 4.1. The rule in the syntax check for CASE statements is that there should always be an OTHERS branch to handle all situations not catered for in the branches where you have given specific values.

The code in Listing 4.1 represents the following business rule: If a monster scares one or two villagers during the course of the day, then his happiness changes to a specific percentage. If the monster has scared no villagers, or more than two, then his happiness remains unchanged.

```
DATA: ld_monster_happiness      TYPE i,
      ld_villagers_scared_today TYPE i.

ld_monster_happiness = lo_monster->get_current_happiness( ).

CASE ld_villagers_scared_today.
  WHEN 1.
    ld_monster_happiness = 25.
  WHEN 2.
    ld_monster_happiness = 50.
ENDCASE.
```

Listing 4.1 CASE Statement that Only Changes a Variable Sometimes

This code is exactly what you want, but nonetheless the extended syntax check is going to give you a false positive warning because there is no OTHERS branch. You have two choices:

1. You can leave the code as it is, in which case you will get a warning that you haven't dealt with the OTHERS possibility in the CASE statement.
2. You can add an OTHERS clause that says "do nothing" as a comment, and then you'll get a warning that there are no executable statements between OTHERS and ENDCASE.

In the case of mass checks, false positives occur when the static code checks fail but there is a good reason why the "rule" set by the system is not being obeyed. A good example is when you create a function module to be called by a *dependency* in Variant Configuration (which changes the value of one or more fields depending upon the value of other fields). In such cases, the interface/signature of the function module is set in stone; you have to have one import structure, two tables parameters, and two defined exceptions. However, when you create the function module, the system starts warning you that tables parameters are obsolete. In this case, you have no choice; the function will not work without tables parameters, so you have to ignore the warning. (Maybe one day SAP will allow

methods to be called from Variant Configuration dependencies, but OO programming has only been available for about 15 years in ABAP, so it's early days yet.)

In this example, it turns out that 75% of the priority one errors are missing text elements. The problem with this is that it prevents translation into another language. Although this is technically a bad practice, it's possible you are in a company that only uses one language, and therefore it may not really affect you. Just for demonstration purposes, say that this is not a real error. Navigate to one of your problems, decide it isn't a problem at all, and click the **APPLY FOR AN EXEMPTION** link at the bottom of the detailed result (Figure 4.12).

Details			
Location / Finding	Description	Contact Person	
Package	ZBC_ABAP_TO_XLST		HARDYP
Class Pool	ZCL_EXCEL_GRAPH_LINE	Bars Graphic	HARDYP
Method	CREATE_AX		HARDYP
Line Number	29		
Check Title	Extended Program Check		
Check Message	Text Element Missing in a Character String		
Priority	Priority 1		
Found on	14.07.2014 17:34:53		
Strings without text elements are not translated:			
'General'			
Display Object	Apply for an Exemption		

Figure 4.12 Detailed ATC Result with Apply for an Exemption at the Bottom

Figure 4.13 shows the pop-up that then appears. For this example, choose to exempt all objects in this package from this particular check, because the (non) errors all stem from the same source. Click **CONTINUE**.

Steps

- Granularity and Scope
- Approver and Reason

Finding

Package: ZBC_ABAP_TO_XLST

Class (ABAP Objects): ZCL_EXCEL_GRAPH_LINE

Method: CREATE_AX

Check: Extended Program Check

Check Message: Text Element Missing in a Character String

Object Restriction: Exemption for

This Sub-Object

This Object

This Package

Check Restriction: Exemption for

This Check Message

This Check

Figure 4.13 Requesting an Exemption: 1 of 2

On the next screen (Figure 4.14), you choose who is going to review the exemption, and you actually get to write the explanation of why this is not a real error.

Figure 4.14 Requesting an Exemption: 2 of 2

In the test system, the quality street manager can call up Transaction ATC and follow the menu path **ATC ADMINISTRATION • QUALITY MANAGEMENT • EXEMPTION BROWSER**. This is now just like a workflow item; the quality manager approves or rejects the exemption and has to provide some sort of text explanation.

Usage Procedure Logging

Section 4.2 talked about running a mass check against all the custom objects in your system. However, it would be a colossal waste of time to use the ATC to perform detailed analysis on code that is never actually used, and SAP surveys suggest that 75% of custom code is never actually run due to business process changes and new versions of existing programs and functions. Fortunately, as of SAP NetWeaver 7.02 SP 9 and SAP NetWeaver 7.31 SP 3 and standard in SAP NetWeaver 7.4, there is a new tool, Usage Procedure Logging (UPL), which is like the Coverage Analyzer (Transaction SCOV). This tool helps you identify unused code.

The idea is that you set up the UPL monitoring job in your SAP Solution Manager system and monitor the execution of all custom code in the productive system for a whole year. There is supposed to be no performance effect on the live system, in case you were worried. It needs to work all year to catch half-year and year-end closing activities.

In order to analyze UPL data, first of all you have to switch on data collection in SAP Solution Manager Transaction /SDA/CD_CCA. When you call up this transaction, there is a box called UPL CONTROL at the top of the screen. When you click this box, you see the screen shown in Figure 4.15.

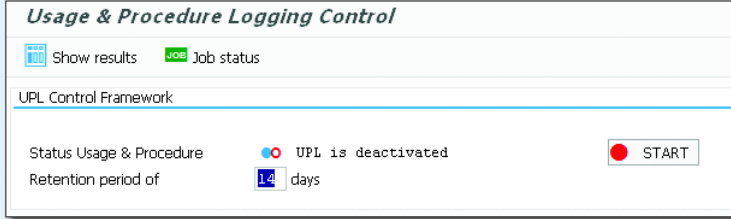


Figure 4.15 UPL Control Screen

Usability experts have said that if you want your users to click a button in an application, then make it big, red, and dangerous looking, preferably with "Do not press this" written next to it. The START button in Figure 4.15 qualifies. Luckily, the world does not blow up if you click it; instead, you see the message UPL ENGINE FOR DATA COLLECTION STARTED, which is a bit of an anticlimax.

Next, click the green JOB STATUS icon, and you are taken to SM37, where you'll see that a job has been scheduled to collect the usage data at one second after midnight each day.

Once the job has run a few times, you can view the data by choosing the box in Transaction /SDA/CD_CCA called UPL SHOW (as in "Let's raise the curtain on the UPL show tonight"), which is above the CODE QUALITY box—the latter being the ABAP Test Cockpit itself.

You end up with a ranked list of every Z object that was executed in production, analyzed down to the subroutine level. By comparing this to your total list of Z objects, you can see what is never used and set about retiring that dead code.

For the code that remains (and you will probably be shocked how little there is as a percentage of the total), you can rank the objects by execution frequency. If the ATC gives you 10 million errors for the (actually used) custom code base, then you know which objects to concentrate on first.

This is exactly like what we all do in production all the time, looking at the most expensive SQL statements ranked in order of number of executions (Transaction ST04) with a view to improving them.

4.3 Recent Code Inspector Enhancements

Because SAP regularly makes enhancements to the Code Inspector, the purpose of this section is to outline some of the more recent checks. The headings in this section identify the functionality by name and then are followed by a series of three

numbers. Each of these numbers corresponds to the support stack in which the functionality was added, and the order of the numbers indicates whether we're talking about SAP NetWeaver 7.02, SAP NetWeaver 7.31, or SAP NetWeaver 7.4. So, for example, "12/5/2" means that the check becomes available in SAP NetWeaver 7.02 SP 12/SAP NetWeaver 7.31 SP 5/SAP NetWeaver 7.4 SP 2. The checks are sorted by the order in which they become available—oldest first.

4.3.1 Unsecure FOR ALL ENTRIES (12/5/2)

As you've probably experienced yourself, there's a bug in SAP that's particularly annoying to programmers: If you use an empty internal table in a FOR ALL ENTRIES database query, then instead of returning nothing, a full table scan occurs. This is counter intuitive. If I gave you an empty list and then said "bring me everything on this list," you would give me nothing rather than trying to get everything in the world to give to me. Nonetheless, this is how the ABAP system responds. For a long time, SAP used the common software vendor trick of saying that if you document a bug then it is no longer a bug. Although they still have not fixed this, there is now a static code check to alert you to cases in which you have forgotten to check for an empty table.

Listing 4.2 shows an example which has not tested if a table is empty before performing a FOR ALL ENTRIES selection.

```
DATA: lt_monsters    TYPE STANDARD TABLE OF ztvc_monsters,
      lt_selections  TYPE RANGE OF zde_monster_number.

SELECT *
  FROM ztvc_monsters
  INTO CORRESPONDING FIELDS OF TABLE lt_monsters
  FOR ALL ENTRIES IN lt_selections
  WHERE monster_number = lt_selections-low.
```

Listing 4.2 No Empty Table Check

In order to fix this problem and do a proper static code check, you have to make sure that the requisite check is active in your default Code Inspector variant. To do this, select the UNSECURE USE OF FOR ALL ENTRIES checkbox, as shown in Figure 4.16.

As you can see in Figure 4.17, this problem is now picked up by the Code Inspector. Make a little change to your program, as in Listing 4.3, in order to check whether or not the internal table of selection criteria is empty.

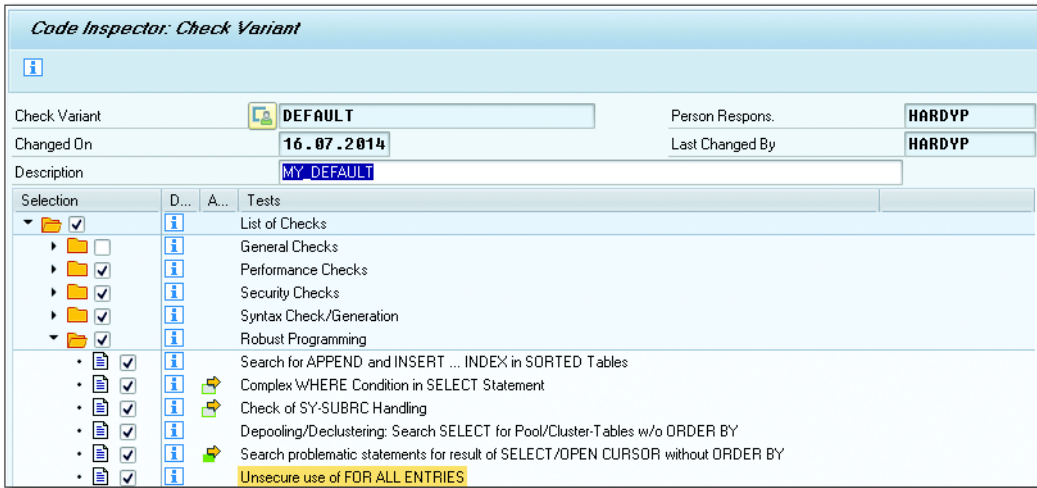


Figure 4.16 Unsecure Use of FOR ALL ENTRIES

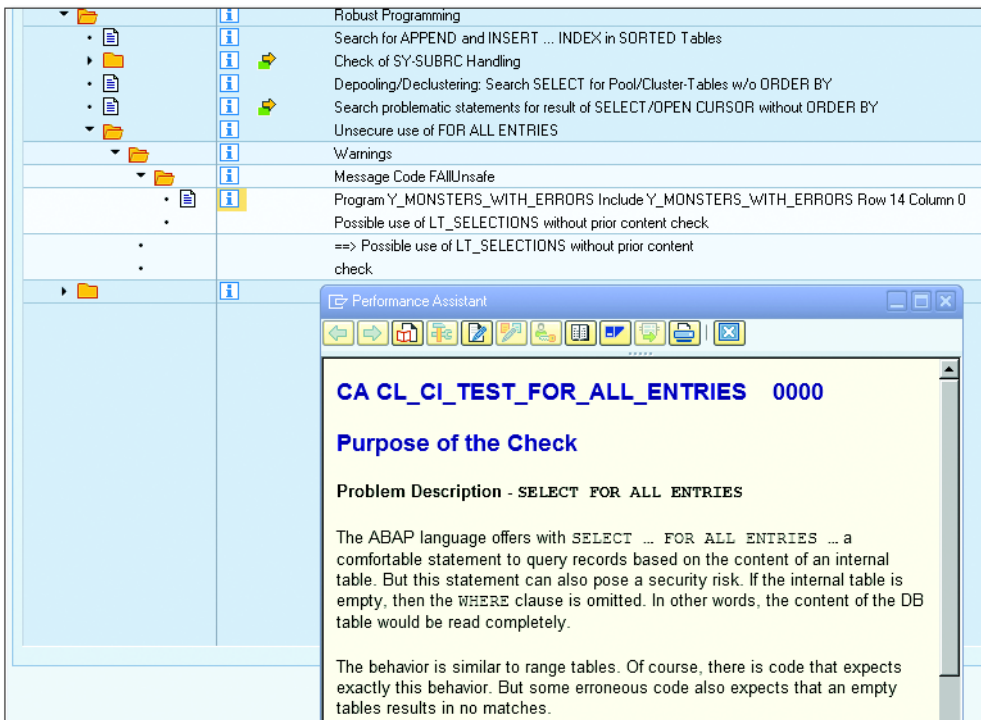


Figure 4.17 FOR ALL ENTRIES Error Message

```

DATA: lt_monsters TYPE STANDARD TABLE OF ztvc_monsters,
      lt_selections TYPE RANGE OF zde_monster_number.

IF lt_selections[] IS NOT INITIAL.

    SELECT *
      FROM ztvc_monsters
      INTO CORRESPONDING FIELDS OF TABLE lt_monsters
      FOR ALL ENTRIES IN lt_selections
      WHERE monster_number = lt_selections-low.

ENDIF.

```

Listing 4.3 Check for Empty Tables

If you rerun the Code Inspector, then you'll find that the error message has vanished like a phantom into the night.

4.3.2 SELECT * Analysis (14/9/2)

SELECT * analysis is a tool that identifies where you have used SELECT * in your code so that you can then make sure that you are only using it when absolutely necessary. The reason you want to be careful with SELECT * is that it brings back every single column from the database table being accessed—and this breaks the golden rule of minimizing the amount of data being transferred from the database to the application server.

In the example, a SELECT * has just been performed on the monsters table to bring back every single column. Now, write some code as shown in Listing 4.4 to output the desired result to the user.

```

LOOP AT lt_monsters INTO ls_monsters.
  WRITE:/ ls_monsters-monster_number,
         ls_monsters-hat_size.
ENDLOOP.

```

Listing 4.4 Outputting the Result

Even though you're only interested in two columns of the table, it turns out that you've retrieved every single column from the database. This is naughty, because it puts undue strain on the system. You should be ashamed of yourself.

Fortunately, you can use the Code Inspector default variant to fix this. Figure 4.18 shows the SEARCH PROBLEMATIC SELECT * STATEMENTS option; all you have to do is make sure this option is checked in your system.

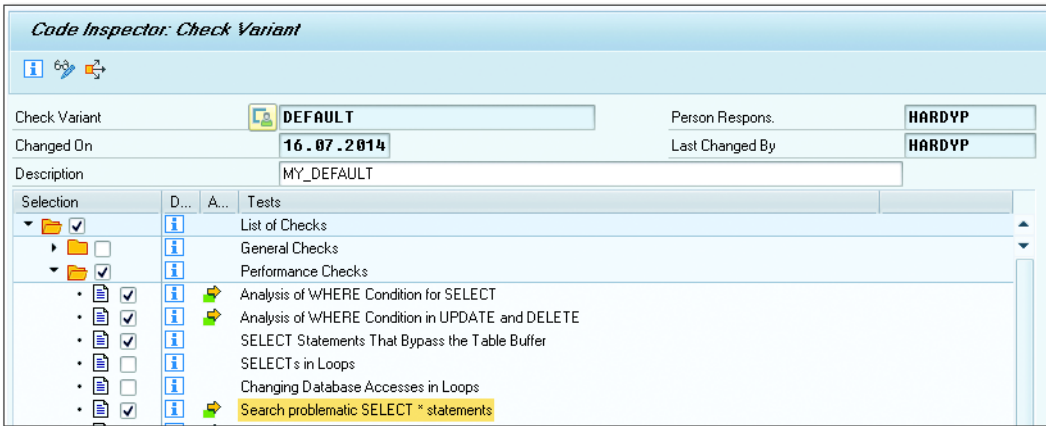


Figure 4.18 Problematic SELECT * Statements

Once that option is active, when you next perform a `SELECT *` a check occurs to see how many of the fields you actually use in subsequent code. If it's less than a percentage you specify when setting up the check (the default is 20%), then you get a slap round the face with a wet fish (Figure 4.19).

Search problematic SELECT * statements	1	0	1
Errors	1	0	0
Message Code FEW ==> Select-Statement can be transformed. 20.0% of fields used	1	0	0

Figure 4.19 Error Message for Problematic SELECT * Query

This is a much more intelligent check than your average bear. In fact, if you don't use any fields that you've retrieved, then you get an existence check warning; that is, you are most likely just seeing if the record exists, so there is no need to get every single column in the table.

4.3.3 Improving FOR ALL ENTRIES (14/9/2)

If the programmers at SAP get any cleverer, their feet will fall off. This check searches for one `SELECT` followed by a `FOR ALL ENTRIES` that uses the result of the first select—and then figures out, and tells you, whether a join would achieve the exact same thing.

Let's turn, once again, to the ongoing monster example. As you know, monsters tend to hide under beds, so you want to do a database query to find out what beds the monsters selected earlier have been hiding under (Listing 4.5).

```
DATA: lt_beds TYPE STANDARD TABLE OF ztvc_monstr_beds.

SELECT *
  FROM ztvc_monstr_beds
 INTO CORRESPONDING FIELDS OF TABLE lt_beds
FOR ALL ENTRIES IN lt_monsters
WHERE monster_number EQ lt_monsters-monster_number.
```

Listing 4.5 SELECT * Check for Monsters under the Bed

If you take a look at this code, you can probably tell that an inner join is a better choice than what's been done here. Will ATC tell you the same thing? Set up the check as in Figure 4.20 to find out.

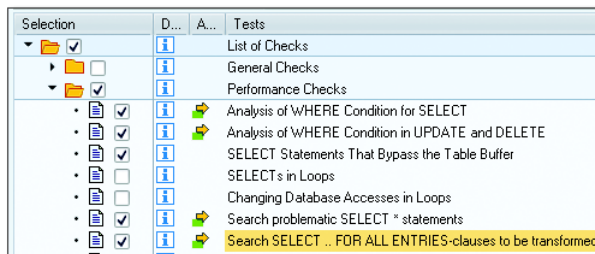


Figure 4.20 Improving FOR ALL ENTRIES

Sure enough, the warning in Figure 4.21 tells you to improve the structure of your code so that you only have one SELECT statement. The reason for the warning is that if you were to take two programs, both selecting the same data, one using an inner join and one using a FOR ALL ENTRIES, and run them one at a time, each time using Transaction ST05 to perform an SQL trace, then you would find that an inner join has only one SQL entry no matter what, whereas a FOR ALL ENTRIES has a large number of SQL entries increasing in a linear fashion with the amount of data to be retrieved.

▶	<input checked="" type="checkbox"/>		Search problematic SELECT * statements	1	0	1
▶	<input checked="" type="checkbox"/>		Search SELECT .. FOR ALL ENTRIES-clauses to be transformed	1	0	1
▼	<input type="checkbox"/>		Errors	1	0	0
▶	<input type="checkbox"/>		Message Code TRANSFORM	1	0	0
•			==> SELECT * FOR ALL statement can be joined with SELECT statement at Include Y_MONSTERS_WITH_ERRORS line 19			

Figure 4.21 FOR ALL ENTRIES Error Message

4.3.4 SELECT with DELETE (14/9/2)

The SELECT with DELETE check is all about situations in which you bring back more records from the database than you really want and then get rid of the surplus ones in your code, which, as you'll see, is a silly thing to do. To understand how this check works, take a look at an example. Listing 4.6 retrieves some database records into an internal table and then deletes the rows you do not like.

```
SELECT *
  FROM ztvc_monstr_beds
 INTO CORRESPONDING FIELDS OF TABLE lt_beds
 FOR ALL ENTRIES IN lt_monsters
 WHERE monster_number EQ lt_monsters-monster_number.
```

```
DELETE lt_beds WHERE monster_number > 500.
```

Listing 4.6 Select Too Many Records and Delete the Unwanted Ones

As you will no doubt have guessed, this is not the optimal way to perform this task. Instead of getting every single bed and then discarding the records relating to certain monsters within the code, the restriction condition should be in the SELECT statement itself, thus minimizing the amount of data transferred between the database and the server. (You might also say no one would ever do this, but I have seen something similar more times than I would like.) Fortunately, there is now a Code Inspector check to detect such nonsense. Figure 4.22 shows the check (SEARCH SELECT STATEMENT WITH DELETE STATEMENT), and Figure 4.23 shows the result of the check.

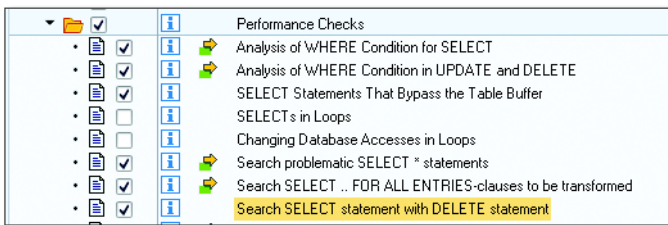


Figure 4.22 Code Inspector: SELECT with DELETE Check

Search SELECT statement with DELETE statement	1	0	1
Errors	1	0	0
Message Code SEL_DEL	1	0	0
==> DELETE statement for result of SELECT statement found			
Information	0	0	1

Figure 4.23 SELECT with DELETE Error Message

4.3.5 Check on Statements Following a SELECT without ORDER BY (14/9/3)

When querying a database, you always want the queries to come back sorted by their primary keys. Although some databases do this automatically, it's best not to rely on it as an assumption. Fortunately, ATC offers a check for this.

The increasingly problematic monster program now does some reads on the GT_BEDS internal table (Listing 4.7).

```
SELECT *
  FROM ztvc_monstr_beds
 INTO CORRESPONDING FIELDS OF TABLE gt_beds
 FOR ALL ENTRIES IN lt_monsters
 WHERE monster_number EQ lt_monsters-monster_number.

READ TABLE gt_beds INTO gs_beds WITH KEY monster_number = 54
 BINARY SEARCH.

PERFORM bed_reader.

*&-----*
*&      Form  BED_READER
*&-----*
FORM bed_reader .

      READ TABLE gt_beds INTO gs_beds INDEX 1.

ENDFORM.                " BED_READER
```

Listing 4.7 Reading a Potentially Unsorted Table

Listing 4.7 makes an assumption that the table is sorted by the primary key. If it is, then the `BINARY SEARCH` will work as expected, and reading the first record will give you the monster with the lowest number. However, to protect yourself, switch on the check to warn you of such problems. Figure 4.24 shows the selected check (`SEARCH PROBLEMATIC STATEMENTS FOR RESULT OF A SELECT/OPEN CURSOR WITHOUT ORDER BY`), and Figure 4.25 shows the results.

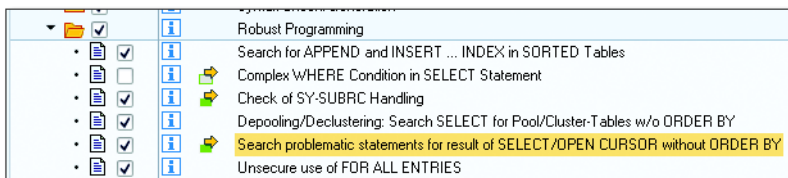


Figure 4.24 Check for Dodgy Statements after SELECT

▼	📁	📄	Search problematic statements for result of SELECT/OPEN CURSOR without ORDER BY	1	0	1
	▼	📄	Errors	1	0	0
	▶	📄	Message Code BIN_SEARCH	1	0	0
	•		==> READ .. BINARY SEARCH for result of statement at Include			
	•		Y_MONSTERS_WITH_ERRORS line 33			
	▼	📄	Information	0	0	1
	▶	📄	Message Code READ_IDX_1	0	0	1
	•		==> READ TABLE ... INDEX 1 for result of statement at			
	•		Include Y_MONSTERS_WITH_ERRORS line 33			

Figure 4.25 Error Message for Dodgy Statements after SELECT

Did you notice that the error for reading the first record was still reported, even though the dodgy statement was in a totally different subroutine to the SELECT statement? This is a new innovation SAP has made in which the Code Inspector becomes clever enough to look across “modularization” units, such as FORM routines, methods, and the like, to hunt for errors. (The next error check you’ll see is an even better example of this.)

4.3.6 SELECTs in Loops across Different Routines (14/9/3)

When you want to retrieve lots of records from the database, the way to go is to get them all at once, because you should always try to minimize traffic between the application server and the database. In other words, the last thing you want to be doing is retrieving the records one at a time. That is what’s happening in Listing 4.8.

```

LOOP AT lt_monsters INTO ls_monsters.
  WRITE:/ ls_monsters-monster_number,
         ls_monsters-hat_size.

  SELECT SINGLE *
    FROM ztvc_monstr_beds
    INTO CORRESPONDING FIELDS OF gs_beds
    WHERE monster_number EQ ls_monsters-monster_number.
ENDLOOP.

```

Listing 4.8 Retrieving Records One at a Time

The preceding code is no problem, because it would automatically raise an error in the Code Inspector. However, now take a look at Listing 4.9.

```

LOOP AT lt_monsters INTO ls_monsters.
  WRITE:/ ls_monsters-monster_number,
         ls_monsters-hat_size.

```



```

PERFORM single_bed_reader USING ls_monsters-monster_number.

ENDLOOP.
*&-----*
*&      Form SINGLE_BED_READER
*&-----*
FORM single_bed_reader USING pus_number TYPE zde_monster_number.

SELECT SINGLE *
  FROM ztvc_monstr_beds
  INTO CORRESPONDING FIELDS OF gs_beds
  WHERE monster_number EQ pus_monsters-monster_number.

ENDFORM.                " SINGLE_BED_READER

```

Listing 4.9 Database Read inside a Loop, Hidden in a Separate Routine

In the preceding code, you're still reading the database in a loop, so you're behaving just as badly as before. However, in this case, the Code Inspector didn't warn you—because the bad database read is happening within a different routine (or method call or function module). Once again, ATC saves the day! You can now activate a check to look out and warn you about exactly this problem. Figure 4.26 shows the check that needs to be activated (SEARCH DB OPERATIONS IN LOOPS ACROSS MODULARIZATION UNITS), and Figure 4.27 shows the results.

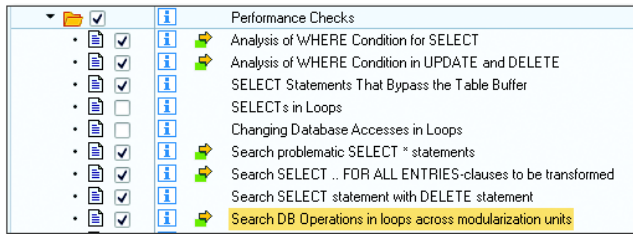


Figure 4.26 Looping SELECTs across Different Units

Folder	<input checked="" type="checkbox"/>	Search DB Operations in loops across modularization units	1	0	1
Folder	<input type="checkbox"/>	Errors	1	0	0
Folder	<input type="checkbox"/>	Message Code DBREAD	1	0	0
Document	<input type="checkbox"/>	Program Y_MONSTERS_WITH_ERRORS Include Y_MONSTERS_WITH_ERRORS Row 61 Column 2	1	0	0
Document	<input type="checkbox"/>	NonLocal Nested Reading DB OP (SELECT) found			
Document	<input type="checkbox"/>	==> NonLocal Nested Reading DB OP (SELECT) found			

Figure 4.27 Looping SELECTs across Different Units Error Message

4.4 Summary

In this chapter, you took a spin around the ABAP Test Cockpit and learned what you can do with it in the latest ABAP releases. Up until this point, you've looked at the process of writing code in the first place and then running assorted static and dynamic tests upon it. In the next chapter, you'll look at how to best debug your programs when you find—to your horror—that despite all this testing there are still errors all over the place.

Recommended Reading

- ▶ Introduction to ATC: <http://scn.sap.com/blogs/cvs/2014/02/28/abap-test-cockpit-atc-your-way-to-secure-abap-code> (Chandan Setty)
- ▶ Rolling Out the ABAP Test Cockpit: <http://scn.sap.com/community/abap/testing-and-troubleshooting/blog/2013/11/19/rolling-out-the-abap-test-cockpit--a-first-experience-report> (Christian Drumm)
- ▶ Getting Started with Usage and Procedure Logging: <http://scn.sap.com/docs/DOC-54826> (Shuge Guo)

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

—Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style*

5 Debugger Scripting

Most of us hate bugs, be they the ones in our programs or the ones with 500 legs and two bright red pincers, those that crawl up our legs when we're not looking. Therefore, in previous chapters you've seen how you can do dynamic tests as an integral part of writing programs with test-driven development and how the ABAP Test Cockpit helps with static checks on your code. These elements represent a two-pronged attack on bugs that tries to kill them before they have taken root.

However, bugs are always with us; a program almost never does everything that was desired with no errors the very first time it is run in Development or QA. (This despite Robert C. Martin's insistence in *The Clean Coder* that "QA should find nothing.") You should always expect the first few tests—by yourself or others—to return unexpected results.

If someone else tells you that the monster simulator program is producing blue monsters with two heads when you were expecting green monsters with three heads, then you need to identify the source of the problem. If you've ever done this—and I suspect you have—then you know that there are days when you spend four hours looking for the problem, and once it finally rears its ugly head the solution is so obvious that you can make the code change to fix up whatever it is in about two minutes flat.

The problem at hand is that you sit in front of a debugger looking for the same sort of thing in different programs, again and again, month after month, year after year, when you would rather spend that time writing programs. What if you could solve this problem by doing exactly that—writing programs—thus turning something really boring into something really fun and solving your employer's

problem faster at the same time? Then, everyone would be a happy bunny. Good news: You can! This chapter will explain how.

To walk you through the process of writing debugger scripts, first you'll see an overview of the `SCRIPT` tab, which is your starting point (Section 5.1). After that, you'll learn about the technical details of coding the `SCRIPT` method (Section 5.2), which is where 99% of the action happens. Finally, Section 5.3 concludes with an example that shows how the `SCRIPT` method's friends, `INIT` and `END`, can help out. Finally, you'll work through an example that shows a full debugger script in action (Section 5.3).

Writing Scripts Outside of a Program

There are actually two ways to create a debugger script: while in the middle of debugging a program and totally independently. These methods are virtually identical, and the focus of this section is explaining the process from within the debugging program. However, if you have an incredibly complicated debugger script that you don't want to write while actually in the middle of debugging a program, you can use Transaction SAS. This allows you to create a debugger script in a standalone manner. (If you are based in the United Kingdom, you may feel an unbearable urge to shout "Who dares, wins!" while using this transaction.)

This transaction looks more or less exactly the same as what's discussed in this section; the only difference is that the list of trace files is in your face, and the actual script writing tool is on the third tab along (this tab is called `SCRIPT EDITOR`).

5.1 Script Tab Overview

Because you're reading a book called *ABAP to the Future*, here's hoping that you already set your debugger settings such that you are in the "new" debugger. To check this, go to Transaction SE80, choose the menu option `UTILITIES • SETTINGS`, and navigate to the `ABAP EDITOR` tab. Here you will get another set of smaller tabs beneath the main `ABAP EDITOR` tab. The one we want is `DEBUGGING`. Here, you can choose between `CLASSIC` (SAP speak for old/rubbish) and `NEW` by selecting the appropriate radio button. You need to select `NEW` in order to walk through the examples in this chapter.

After you've chosen the new debugger, when you are debugging you will have a tab strip at the top of the page showing the various debugger views you can look at (Figure 5.1).

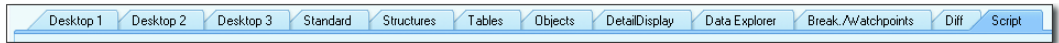


Figure 5.1 Debugger Tab Strip

As might be expected, the SCRIPT tab is the one that lets you create debugger scripts. Clicking on this tab brings you to a screen that looks like Figure 5.2.

Note

If you move to the SCRIPT tab in any system other than the development client where you create your programs, you will get the message SYSTEM SET TO NOT CHANGEABLE. This is because debugger scripts are programs themselves, and it is probably a bad idea to write any sort of program directly in (say) Production. You can load existing scripts from the database into the debugger in production, however, so this is not a problem in any real sense.

Someone might say, "Hang on, I need to run the program in an environment with data, so I never debug in Development." The answer to that is that you are forgetting about unit tests; they run in Development and by their very nature are supposed to cover 100% of the code.

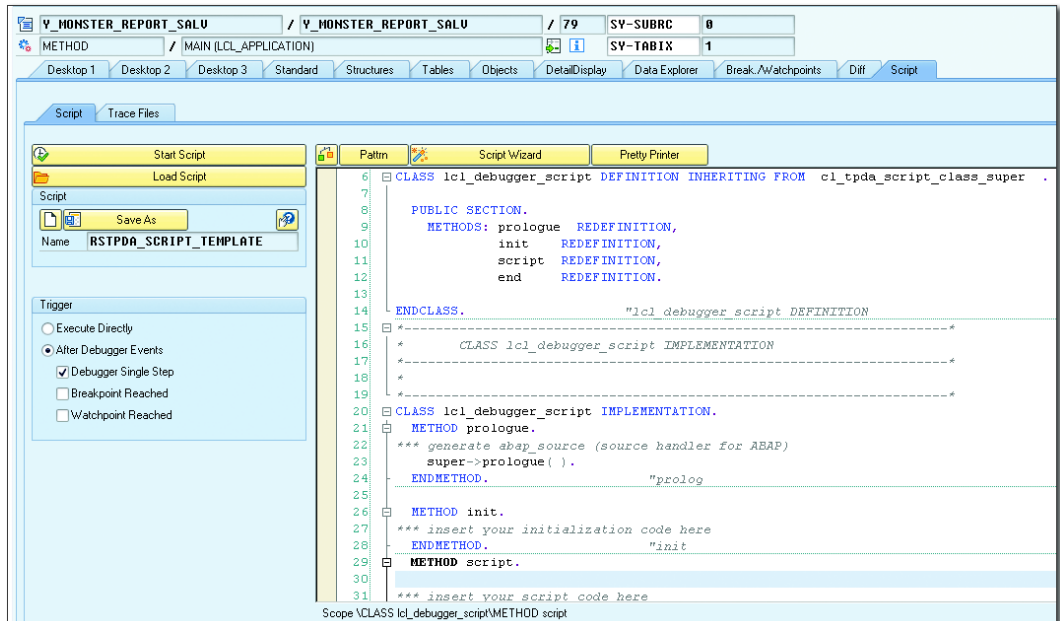


Figure 5.2 Debugger Script Blank Template

The part on the right of the screen in Figure 5.2 is nothing more or less than the normal ABAP Editor—which means that the whole range of ABAP commands are at your disposal.

At the top of the screen in `PUBLIC SECTION`, you can see that there are four methods that you can use in writing debugger scripts. One should not be redefined. Two others can be redefined, but do not have to be. The other one must be redefined.

► **PROLOGUE method** (do not redefine)

The `PROLOGUE` method runs once at the start of the script and does some creepy black magic to make sure the rest of your debugger script can access the variables in the program being debugged. You do not add any of your own code to this method, because it runs automatically without you needing to worry about it.

► **INIT method** (can be redefined)

The `INIT` method runs once at the start of the script and is somewhat like the `SETUP` method in a unit test. It allows you to set some variables to their starting values (for example, to create an instance of the application log).

► **END method** (can be redefined)

The `END` method is like the `TEARDOWN` method in a unit test. It runs once at the end of a script and allows you to do things such as save your application log to the database or email yourself the results. The really important point to note here is that this will only run if the programmer manually clicks the `EXIT SCRIPT` button. (You have to jump through hoops to get that button to appear; you'll see how to do this in Section 5.3.)

► **SCRIPT method** (must be redefined)

The `SCRIPT` method is the vital one; it contains the logic of your program within a program, which will be used to enhance the debugging process. This runs at least once, but more likely many times—usually once every time you step through a line in the code you are debugging.

Another element of Figure 5.2 is the option to configure triggers; these options are shown in more detail in Figure 5.3.

There are two radio buttons on the left of the screen: `EXECUTE DIRECTLY` and `AFTER DEBUGGER EVENTS`. If you choose the first one and then execute your script natu-

rally, then it runs right then and there, using the information available at the exact point you are at in the debugger.

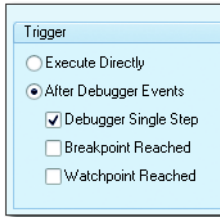


Figure 5.3 Trigger Settings

If you choose AFTER DEBUGGER EVENT and then choose EXECUTE, then you control how many times the script will run, if at all, based on further settings, which you make in the checkboxes below the AFTER DEBUGGER EVENTS button:

▶ **DEBUGGER SINGLE STEP**

If you choose this, then your script will run after every single ABAP statement has been executed. This is for when you have absolutely no idea where in the program the Bad Thing you are investigating occurs.

▶ **BREAKPOINT REACHED**

You know what a breakpoint is, and this is no different. Every time a specified breakpoint is reached, the script will run. This is for when you suspect the problem occurs after a specific statement, such as `AUTHORITY_CHECK`, or after the call of a particular function module. Note that with the advent of the new debugger the breakpoint options were dramatically expanded.

▶ **WATCHPOINT REACHED**

Once again, the watchpoint occurs when a specified variable value changes, and each time this happens your script will run. This is for when you know a problem is caused by a variable being set with a dodgy value, and you want to know what is causing it.

You would be forgiven for thinking that your script would stop at the dynamic breakpoints and watchpoints you have already specified during your debugging session, but you would be wrong. In fact, breakpoints and watchpoints are specific to a given script and have to be defined accordingly. To that end, when you select the breakpoint or watchpoint checkboxes, you will see a little pencil icon pop up to the right of the checkbox (Figure 5.4).

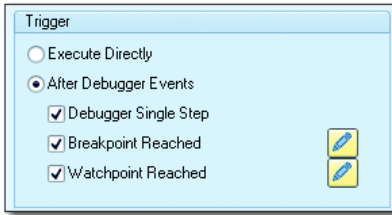


Figure 5.4 Breakpoint/Watchpoint Pencil Icons

When you click on the pencil icon, a new subscreen appears in which you can set up your breakpoints and watchpoints in the exact same way you would in the normal debugger (Figure 5.5 and Figure 5.6).

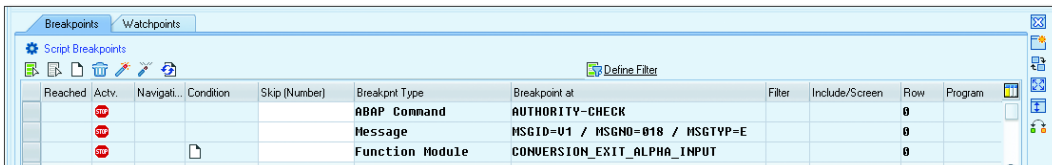


Figure 5.5 Script Breakpoints

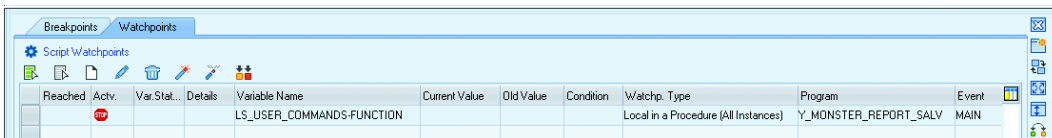


Figure 5.6 Script Watchpoints

While you are defining any breakpoints and watchpoints, you will be reminded of how flexible the existing options already are in regards to where you want the debugger to stop.

However, nothing is ever infinitely flexible; one day you may decide that you only want the debugger to stop at a certain message . . . when a certain variable has a certain value and there is a certain program higher up in the current call stack . . . and a PID has a given value . . . and an external system is offline . . . and Jupiter is in conjunction with Pluto . . . and the cock has crowed four times at midnight and laid four addled eggs. This sort of requirement is quite common, and you can satisfy it by coding extra checks at the start of your main script method whenever it is called by a breakpoint or watchpoint.

5.2 Coding the SCRIPT Method

An unknown person once said: “Don’t anthropomorphize computers; they hate it.” Nonetheless, in this case you’ll need to think about how you can tell the computer to behave as you would when debugging.

To start, think about how you yourself debug something. First, you keep going through the program until you find something you are interested in. To do that, you take a look at the screen. Your debugger script can’t do that, so it needs a way to access the variables in the running program so that it can analyze their values. Next, you may change a variable or value, step forward through the program being debugged a line at a time looking for something else, or decide you’re in the wrong place and press **F8** to move on to the next breakpoint or watchpoint.

Your debugger script needs a way to do all of these things. To start, navigate to the `SCRIPT` method implementation in your debugger script program. There is already one line of code waiting for you (Listing 5.1).

```
*** insert your script code here
me->break( ).
```

Listing 5.1 Generic Command

That is just a generic command to stop the debugger. You’ll usually want to get rid of that line and replace it with something more suited to the purpose at hand, whatever it may be. (If you do want to stop the debugger after your script has run—which you will find is not usually the case—then you can leave that line at the end.)

At the top of the `ABAP EDITOR` section of the debugger `SCRIPT` tab, you’ll see four buttons. Three of them just do what you would expect in an `ABAP` editor: a syntax check, a “pretty print,” and a pattern insert. However, there’s an extra button specific to debugger scripting: it’s called `SCRIPT WIZARD` (though it has a multiple personality disorder; when you hover your cursor over it, you see `SCRIPT SERVICES`). The important thing is that this button has a picture of a magic wand on it. That has to be good.

The Script Wizard is a wonderful wizard, if ever a wizard there was, because, because, because, because it serves as a bridge between the program being debugged and the program that controls the debugger. When you choose an option from the Script Wizard, some code is inserted into your debugger script

program in the exact same way that the PATTERN button inserts a function module or method call into your normal programs and has you fill in the signature (input/output) values.

Figure 5.7 shows some of the most interesting Script Wizard options. Each of these options performs one of the steps that would be done manually by a programmer while debugging a program.

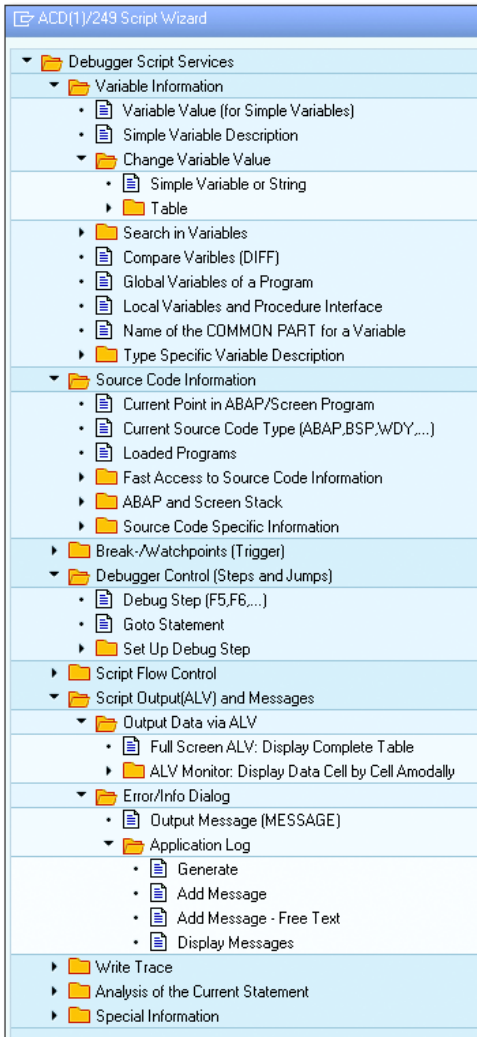


Figure 5.7 Script Wizard Options

For example, some options give the script program the values of various variables in the program being debugged (VARIABLE INFORMATION), system stack information (ABAP and SCREEN STACK), or PID values (SPECIAL INFORMATION)—basically anything you could normally see by poking about manually in the debugger screen. There are also a few options to control stepping through the debugger to simulate pressing, for example, `F5` or `F8` (DEBUGGER CONTROL). Some options let you change the values of a variable in the program being debugged (CHANGE VARIABLE VALUE), and let me tell you, it's a lot easier to append an internal table row via a program than to fill in all the cells by hand. Finally, some options are the equivalent of writing down some information on a piece of paper on your desk—for example, ALV output, messages, or writing to the application log (SCRIPT OUTPUT [ALV] and MESSAGES). This removes the risk of the useful information you have written on your piece of paper getting lost when the fixed asset accountant comes around to complain about your data conversion program not working and then trips up and spills her drink all over your desk.

Frankly, the options on this screen should make you want to jump up and down and start screaming at the top of your voice, "Look at this; look how good it is!" But a more reasonable observation would be that a lot of thought has clearly gone into this, and it seems like most if not all of the manual steps during debugging can be replicated in a debugger script program.

Examine this a little closer to see whether it's true. The manual steps a programmer takes while debugging can be broken into four stages: look, move, act, and log.

The purpose of the look stage is to find out the value of one of the variables in the program being debugged. To accomplish this in the Script Wizard, open it and navigate to VARIABLE INFORMATION • VARIABLE VALUE. A blank template is inserted into the debugger script code, as shown in Listing 5.2.

```
*TRY.
CALL METHOD CL_TPDA_SCRIPT_DATA_DESCR=>GET_SIMPLE_VALUE
  EXPORTING
    P_VAR_NAME =
  RECEIVING
    P_VAR_VALUE =
  .
* CATCH cx_tpda_varname .
* CATCH cx_tpda_script_no_simple_type .
*ENDTRY.
```

Listing 5.2 Variable Value

The idea here is that you modify the template to pass in the variable name from the program being debugged as an uppercase literal string, such as `LD_MONSTER_TYPE`. The method calls move the variable value of the program being debugged into a variable that is local to your debugger script program. Then, an exception is raised if you pass in the name of a variable that does not actually exist, or in this specific example an exception is raised if the variable in question is not an elementary variable. Thus, you've programmatically, instead of manually, completed the look stage.

Now you're at the move stage. When debugging manually, you "move" through the program being debugged until you reach your breakpoint or watchpoint (which is located at the place you're interested in), and once you're there you usually then step through a few (or a lot) of lines of the program by pressing `F5` repeatedly. The equivalent option from the Script Wizard is `DEBUGGER CONTROL • DEBUG STEP`, which generates the code in Listing 5.3.

```
*****
* debugger commands (p_command):
* Step into(F5)  -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_INT0
* Execute(F6)   -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_OVER
* Return(F7)    -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_OUT
* Continue(F8)  -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_CONTINUE
*****
*****
*Interface (CLASS = CL_TPDA_SCRIPT_DEBUGGER_CTRL / METHOD = DEBUG_STEP )
*Importing
*      REFERENCE( P_COMMAND ) TYPE I
*****

*TRY.
CALL METHOD DEBUGGER_CONTROLLER->DEBUG_STEP
EXPORTING
    P_COMMAND =
    .
* CATCH cx_tpda_scr_rtctrl_status .
* CATCH cx_tpda_scr_rtctrl .
*ENDTRY.
```

Listing 5.3 Debug Step

In the generated comments before the `DEBUG_STEP` method, you'll see a nice list of what constants correspond to each of the function keys in the debugger. To advance the debugger by statement, you pass the `DEBUG_STEP_INT0` constant into the `DEBUG_STEP` method. Then, you're one command further along, and you can

call another method to look at the system state and see if you have found the bug. In this example, the bug is that our variable has changed, and this stage will end with you finding that variable.

Now, by a combination of looking and moving, you've found the place where the bug resides, and it's time for the act stage. What you usually do here when manually debugging is change the variable value to the correct value, and press **F8** to see if changing that variable has fixed the problem so that the rest of the program executes correctly. If so, then you know you need to concentrate on fixing the part of the program that messes up the variable value.

To do this via the Script Wizard, choose **CHANGE VARIABLE VALUE • SIMPLE VARIABLE** or **CHANGE VARIABLE VALUE • TABLE**. For this example, use the former, and you'll get the code shown in Listing 5.4.

```
*TRY.
CALL METHOD CL_TPDA_SCRIPT_DATA_DESCR=>CHANGE_VALUE
EXPORTING
    P_NEW_VALUE =
*   p_offset    = -1
*   p_length    = -1
    P_VARNAME   =
.
* CATCH cx_tpda_varname .
* CATCH cx_tpda_scr_auth .
*ENDTRY.
```

Listing 5.4 Programatically Changing a Variable Value in the Debugger

Naturally, this is just the reverse of reading a variable value, with the same sort of error handling in case a bogus variable value is passed in. After you've changed your variable value, return control to the normal debugger with the `ME->BREAK()` command. Hopefully, changing the variable value solved the problem. If not, then at least you have eliminated one possible cause of the error—but it is time to go back to the drawing board and change your script to look for something else. In any event, once back in the real debugger, you press the actual **F8** button, which will bring the program (and hence the debugging session) to a natural end.

Throughout all of the first three stages there is also the log stage. When going through a debugger manually, you sometimes take screenshots of some variable values at a given point in the program and print them out, email them to a colleague, or write them down on the ever-popular piece of paper. (Downloading

the contents of an internal table to a spreadsheet is another fun pastime.) In essence, you are copying information from the debugger screen to a more permanent medium.

When looking at the debugger screen while manually debugging, it's good to check out the call stack to find out exactly how you got the routine that is misbehaving itself. For example, I recall jumping for joy when I finally found out the exact call sequence used by the sales order creation BAPI to get to the code that set the Variant Configuration characteristics. For years, I had never been able to work out where this was occurring, and then one day ("luckily") the BAPI started dumping in the exact spot I was looking for. I wrote the call sequence on a piece of paper, and flagged that piece of paper as really important and to be kept in a safe place. Then, of course, I lost it.

Fortunately, when using a debugger script program, there is no need to step through the program and write down values of variables or the system state (i.e., what exact routine in what exact program changed the value) that are useful to you. The idea is that your program monitors the flow of the program being debugged and gathers all of the information you require and then shows you the final result all in one go. The debugger script framework is designed for creating trace files, but you can also have an ALV report, an application log, or even a spreadsheet of the values from an internal table emailed to yourself via ABAP2XLSX (more about ABAP2XLSX in Chapter 11).

However, if there *is* a point in your debugger script at which you want to make a note of something, simply click the SCRIPT WIZARD button and choose SCRIPT OUTPUT • ERROR/INFO DIALOG • APPLICATION LOG • ADD MESSAGE—FREE TEXT. The result of this is shown in Listing 5.5.

```
*****
*Interface (CLASS = CL_TPDA_SCRIPT_MESSAGES / METHOD =
ADD_MESSAGE_FREE_TEXT )
*Importing
*      REFERENCE( P_MESS_TYPE ) TYPE SYMSGTY
*      REFERENCE( P_TEXT ) TYPE CHAR255
*      REFERENCE( P_PROBLEM_CLASS ) TYPE BALPROBCL OPTIONAL
*****

*TRY.
CALL METHOD APPL_LOG->ADD_MESSAGE_FREE_TEXT
  EXPORTING
    P_MESS_TYPE      =
```

```

P_TEXT          =
*   p_problem_class =
.
* CATCH cx_tpda_application_log .
*ENDTRY.

```

Listing 5.5 Free Text

An application log object has to be created in order to have messages added to it and to output the results. In the example in Section 5.3, you will see how to create your application log in the `INIT` method and display it to yourself in the `END` method.

Using Z Classes in the SCRIPT Method

The Script Wizard contains a large number of useful classes that can do almost anything you can think of when it comes to controlling the debugger. However, you are bound to reach a situation in which you need to do something that is not covered. When that day comes, it's no problem—because you're in a normal ABAP method, so you can use your own classes.

Say you want to save the application log to the database. The standard `CL_TPDA_SCRIPT_MESSAGES` does not give you that option, so instead of using the wizard you just use methods of a class that does have that option, perhaps one that implements `IF_RECA_MESSAGE_LIST`. You usually cannot create a subclass of standard SAP classes, like `CL_TPDA_SCRIPT_MESSAGES`, because 99.99% of the time standard SAP classes are set to `final`. Therefore, creating your own Z classes with the same method interfaces is the way to go most of the time.

5.3 Coding the INIT and END Methods

To further illustrate the process of writing a debugger script, look at another example, this time one that involves the `INIT` and `END` methods as well as the `SCRIPT` method. You have a program in which a monster counts to 10, and every time the monster gets to a number that is divisible by three he has to say “Ha ha ha.” Every time he gets to a number that is divisible by five, lightning flashes. The end users are complaining that there is not enough laughing and lightning. Sounds familiar, right? You probably get almost the exact same situation in your day-to-day work all the time.

The example program can be seen in Listing 5.6. Right away, you will spot the error, which is that the program uses the command `DIV` instead of `MOD`: tut, tut, tut.

```

DATA: ld_current_number TYPE sy-tabix,
      ld_laugh_count     TYPE sy-tabix,
      ld_lightning_count TYPE sy-tabix.

DO 10 TIMES.
  ADD 1 TO ld_current_number.

  WRITE:/ 'Monster has counted to',ld_current_number.

  IF ld_current_number DIV 3 = 0.
    WRITE:/ 'Ha ha ha!'.
    ADD 1 TO ld_laugh_count.
  ENDIF.

  IF ld_current_number MOD 5 = 0.
    WRITE:/ 'Lightning Flashes!'.
    ADD 1 TO ld_lightning_count.
  ENDIF.
ENDDO.

```

Listing 5.6 Laughing Program with Error

Even though you already know what the error is, pretend that you're at your wit's end; you just cannot spot what's wrong. Instead of stepping through the program one line at a time looking for where things go wrong, write a debugger script to show you an application log that analyzes the program flow and tells you what's wrong.

Set a user-specific breakpoint right at the start of the program, and execute it in your development system. Once you're in the debugger, navigate to the SCRIPT tab, and move to the INIT method, where you'll use the Script Wizard to insert a template for creating an application log (Listing 5.7).

** Local Variables*

```

DATA: ls_log_header TYPE bal_s_log.

ls_log_header-object    = 'MONSTER'.
ls_log_header-extnumber = '12345'.

TRY.
  CALL METHOD cl_tpda_script_messages=>create_appl_log
    EXPORTING
      p_log      = ls_log_header
    RECEIVING
      p_ref_log = appl_log.

  CATCH cx_tpda_application_log .

```



```

        "No point going on
me->break( ).
ENDTRY.

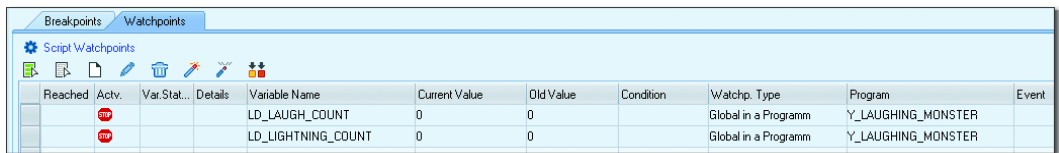
```

```
ENDMETHOD."init
```

Listing 5.7 INIT Method

The `INIT` method only runs once when the debugger script executes, so this is the best place for creating the application log. Note that the instance of the application log is called `APPL_LOG` and is a member variable of the superclass of the debugger script class you're running (put more simply, you don't have to create a variable for the application log).

You then need to set some watchpoints so that the debugger script executes the `SCRIPT` method every time one of the two variables you're interested in changes (Figure 5.8 and Figure 5.9).



Reached	Actv.	Var.Stat...	Details	Variable Name	Current Value	Old Value	Condition	Watchp. Type	Program	Event
	stop			LD_LAUGH_COUNT	0	0		Global in a Programm	Y_LAUGHING_MONSTER	
	stop			LD_LIGHTNING_COUNT	0	0		Global in a Programm	Y_LAUGHING_MONSTER	

Figure 5.8 Laughing Example: Watchpoint Settings

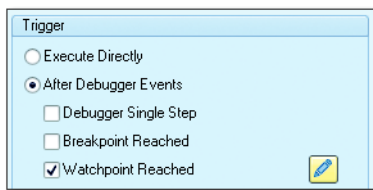


Figure 5.9 Laughing Example: Trigger Settings

Next, when the `SCRIPT` method runs, you need to find out which of the watchpoints has triggered, interrogate the current variable values, and finally write some information to the log. At each stage, you use the Script Wizard to import a template into your code, and the result comes out looking like Listing 5.8.

```

METHOD script.
* Local Variables
  DATA: ld_current_line TYPE i,
        ld_current_number TYPE sy-tabix,

```

```

ld_variable_name TYPE tpda_var_name,
ld_variable_value TYPE tpda_var_value,
ld_info_text     TYPE char255,
ld_message_type  TYPE char01.

```

** What line are we on?*

```

TRY.
  CALL METHOD abap_source->line
    RECEIVING
      p_line = ld_current_line.

  CATCH cx_tpda_src_info .
    RETURN.
  CATCH cx_tpda_src_descr_invalidated .
    RETURN.
ENDTRY.

```

** Get Current Variable Values*

```

ld_variable_name = 'LD_CURRENT_NUMBER'.

TRY.
  CALL METHOD cl_tpda_script_data_descr=>get_simple_value
    EXPORTING
      p_var_name = ld_variable_name
    RECEIVING
      p_var_value = ld_variable_value.

  ld_current_number = ld_variable_value.

  CATCH cx_tpda_varname .
    RETURN.
  CATCH cx_tpda_script_no_simple_type .
    RETURN.
ENDTRY.

```

** Analyse current situation*

```

CASE ld_current_line.
  WHEN 29. "Laugh Count Changed
    ld_info_text = |Laugh trigger { ld_current_
number } is supposed to be exactly divisible by 3|.
    IF ld_current_number MOD 3 <> 0.
      CONCATENATE ld_info_text 'and it is not'
        INTO ld_info_text SEPARATED BY space.
      ld_message_type = 'E'.
    ELSE.
      CONCATENATE ld_info_text 'and it is'
        INTO ld_info_text SEPARATED BY space.
      ld_message_type = 'S'.

```

```

ENDIF.
WHEN 33. "Lightning Count Changed
  ld_info_text = |Lightning trigger { ld_current_
number } is supposed to be exactly divisible by 5|.
  IF ld_current_number MOD 5 <> 0.
    CONCATENATE ld_info_text 'and it is not'
    INTO ld_info_text SEPARATED BY space.
    ld_message_type = 'E'.
  ELSE.
    CONCATENATE ld_info_text 'and it is'
    INTO ld_info_text SEPARATED BY space.
    ld_message_type = 'S'.
  ENDIF.
WHEN OTHERS.
  RETURN.
ENDCASE.

* Write the information into the application log
TRY.
  CALL METHOD appl_log->add_message_free_text
    EXPORTING
      p_mess_type = ld_message_type
      p_text      = ld_info_text.
  CATCH cx_tpd_application_log .
  RETURN.
ENDTRY.

"When done, return control to the real debugger
IF ld_current_number = 10.
  me->break( ).
ENDIF.

ENDMETHOD. "script

```

Listing 5.8 SCRIPT Method for Laughing Example

After the script finishes, the `END` method is called, and that is where you'll display the application log. You will notice in Listing 5.8 that at the end of the script method there is some conditional logic such that after the last expected watch-point has been reached you force a breakpoint and return control to the normal debugger. This is needed so that the `END SCRIPT` button pops up in the developers face. The programmer needs to click that button; otherwise, the `END` method will never be executed (Listing 5.9). (That fact seems really odd; you would expect the `END` method to run at the end of a script, the way the `INIT` method runs automatically at the start of a script, but in real life you have to press a button, and so a button shall be pressed.)

```

METHOD end.
*** insert your code which shall be executed at the end of the
*** scripting (before trace is saved) here
TRY.
    CALL METHOD appl_log->display_log
        EXPORTING
            p_title = 'Laughing Monster Log'.

    CATCH cx_tpd_application_log .
        RETURN.
ENDTRY.

ENDMETHOD."end

```

Listing 5.9 End Script

Now, perform a syntax check on your script to make sure everything is okay, and then you can save your script to the database via the SAVE AS button (the screen shown in Figure 5.10 appears). That way, if you need to execute the program being debugged 500 times in a row, each time you simply press the LOAD SCRIPT button to get to your debugger script, rather than having to write the script again. (You will notice that you can also save debugger scripts to local files, and then upload them to a different SAP system.)

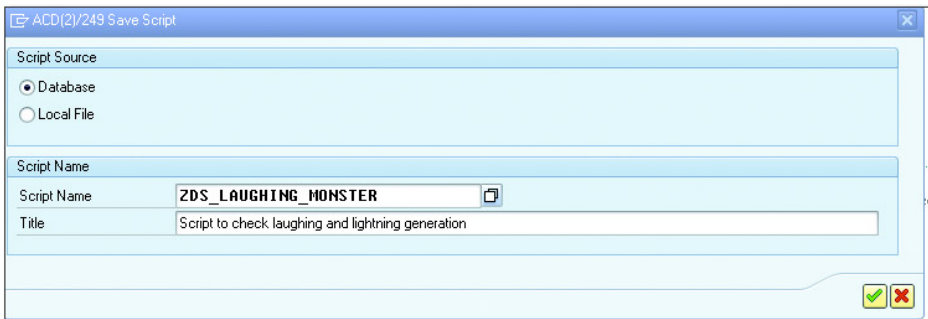


Figure 5.10 Saving a Debugger Script

Last but not least, run your script by pressing the START SCRIPT button. You cannot see that anything is happening until the point is reached in your script in which a breakpoint is forced and control is returned to the normal debugger. When that happens, you'll see a choice of two buttons to click (Figure 5.11).

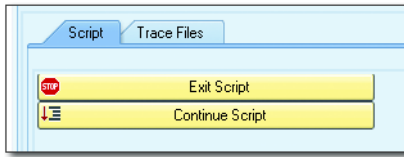


Figure 5.11 Exit or Continue Script

If you press the CONTINUE SCRIPT button, then nothing at all happens; the program ends normally. If you press the EXIT SCRIPT button, then the script does indeed finish, but first the END method is run, resulting in the application log being displayed as shown in Figure 5.12. The log is displayed because of the code in the END method (Listing 5.9) that makes it happen.

One of the great features of the application log is that you can nest messages in a tree structure. Therefore, if you are logging a colossal amount of steps, you can set the initial display to be at a very high level, with a red blob at the top level if any of the lower steps caused an error. In this example, a programmer can look at the error lines on the application log and see at once that the first two trigger points are the numbers one and two instead of three, six, and nine. Then the cause of the error becomes obvious, the programmer changes the DIV to MOD, and all is well.

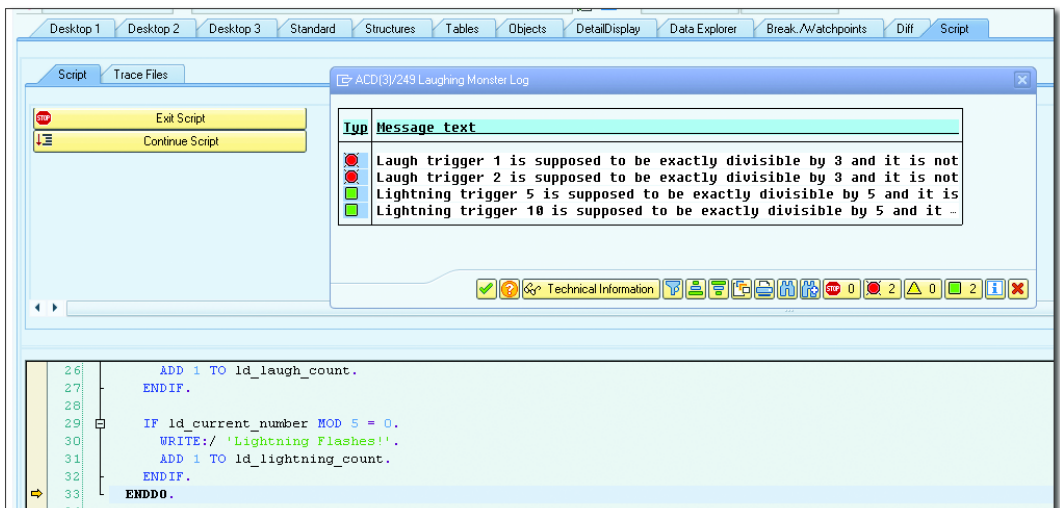


Figure 5.12 Application Log in a Debugger Script

Finally, I would be remiss not to mention that SAP expects you to write your debugger information to trace files as opposed to the application log, and the Script Wizard gives you assorted write trace options to achieve this.

5.4 Summary

This chapter started by acknowledging that sitting in front of the debugger is like death and taxes—always with us—and can be quite boring and time consuming. The very next time you are in the debugger—it could be tomorrow, it could be in 10 minutes—take a step back, look at what you are doing from the outside, and see if you are doing the same steps again and again. If so, debugger scripting can probably help you.

If looking at the list of options the Script Wizard provides makes you feel the paradox of choice—there is so much you can do that you don't know where to start and so end up not using debugger scripting at all—worry not. Here are some tips for how debugger scripting can be used in the real world:

- ▶ **Tracing every line executed by a program**

This is the example you see most often in standard SAP examples about the debugger script. There are a variety of predefined debugger scripts supplied by SAP, and one of them is `RSTPDA_SCRIPT_STATEMENT_TRACE`, which only has one command in the `SCRIPT` method. What this does is to write to a trace file every single line of every single program that gets executed, in the order those lines gets executed.

You can take a copy of such predefined scripts and change them any way you want, so you can have the result list filtered by breakpoints or watchpoints or by any custom criteria you can think of.

- ▶ **Skipping authority checks**

When I'm presenting at SAP conferences, I often have a quick (or not so quick) drink with SAP expert Brian O'Neill from Nevada. He wrote a blog post on SCN in which he gave an example of a common tedious problem when debugging.

Sometimes, the programmer may not have authorization for everything in the program being tested. To get around this, traditionally you had to stop at every `AUTHORITY-CHECK` statement and if the check failed then set `SY-SUBRC` manually to zero so you could continue. This gets boring in a hurry. With a debugger

script, you can tell the debugger to automatically change the value of `SY-SUBRC` to zero just after every `AUTHORITY-CHECK`.

► **Creating a watchpoint for a field symbol**

This is another one from Brian O'Neill. In the standard debugger, you cannot have such a watchpoint that stops the debugger when the value of a field symbol changes. With debugger scripting, he found a way to make this possible.

The important point to note here is that not only can debugger scripting automate manual steps in the debugger, but also it can make the debugger do things it was not previously capable of.

That concludes the discussion of debugging. But don't be sad! In the next chapter, you'll make a stop in the land of the strange life forms called *new BAdIs*, creatures that are half man, half beast.

Recommended Reading

- ABAP Debugger Scripting: Basics: <http://scn.sap.com/people/stephen.pfeiffer/blog/2010/12/14/abap-debugger-scripting-basics> (Stephen Pfeiffer)
- Skip the Authority Check with the ABAP Debugger Script: <http://scn.sap.com/community/abap/blog/2013/03/22/skip-the-authority-check-with-the-abap-debugger-script> (Brian O'Neill)
- How to Create a Watchpoint for a Field Symbol in the ABAP Debugger: <http://scn.sap.com/community/abap/blog/2013/03/08/how-to-create-a-watchpoint-for-a-field-symbol-in-the-abap-debugger> (Brian O'Neill)

I love the baddies. More important, though, is making the baddies somehow, weirdly, understood.
—Mark Strong

6 The Enhancement Framework and New BAdIs

Standard SAP code is like Swiss cheese: full of holes. In these holes, you can add your own code—otherwise known as user exits—and thus alter the standard behavior of the SAP system to fit the exact needs of your business.

User exits haven't always received the attention they deserve. In the past, SAP used to recommend (in certain cases) that instead of creating a user exit you should make a copy of the standard program, creating a clone that started with a Z, and make your changes in that copy. Because this was official SAP policy at the time, many SAP customers did just that and created an army of cloned programs. Then, SAP belatedly realized this was the worst thing you could possibly do, because during an upgrade the original program would change (due to bug fixes and extra functions), but the clone would not. Then came what can best be described as the clone wars, with customers writing clone hunter programs, which eventually became part of standard SAP.

The revised policy was not to have clones, but to use the *Modification Assistant* to insert your changes directly into standard SAP code. Then, during an upgrade, the SPAU process would recognize such changes and make you decide if you wanted to keep them or not. This was a lot better, but if you have a fair number of such modifications to the standard system (and a lot of companies do, even ones with a “no modification” policy), then at upgrade or support stack time the SPAU process could take quite a while.

The current—and clearly the best—option is to use a user exit. Modifications to standard SAP code vanish during an upgrade and have to be manually reapplied, but user exits of any variety remain.

Over the years SAP has tried several varieties of user exits. In order of appearance, they are as follows:

▶ **VOFM routines**

VOFM routines are used for changing the behavior of pricing procedures, controlling the transfer of data between sales documents, splitting up invoices, and a whole bunch of other, seemingly unrelated, activities. The idea is to write a short piece of ABAP code with a number in it, like RV905A, and the functional consultant adds the number 905 into a field in a customizing table to change the behavior of the pricing procedure (for example). (You can tell this type of user exit came first, because it cannot handle decimal places very well.)

▶ **Form-based user exits**

Form-based user exits are just what they sound like: user exits you can create by filling out a standard form. An example is Create Sales Order, Transaction VA01 (Program SAPMV45A). Form-based user exits were incredibly useful but a touch unstable, because you can change global variables of a standard SAP program like nobody's business.

▶ **CMOD framework**

The CMOD framework is comprised of function modules with the word `EXIT` in them—for example, `EXIT_SAPLIEDI_002`. These have a defined interface saying what you are supposed to be changing.

▶ **BAdIs**

This acronym stands for *Business Add Ins*, which were the first user exits to be object oriented and have a defined interface, and they beat the other types into a cocked hat, as you will see in this chapter. The first iteration of BAdIs came in with version 4.7 of SAP ECC. Although no one could argue that this was not a positive step in the evolution of user exits, the main problem was that in order to see if such a user exit existed in a given point in a standard program, a database read was needed. Because this had to happen a lot in standard programs, in some companies the system became more “sluggish” after the upgrade due to the increased number of database reads. In SAP ECC 6.0, the BAdIs became “new” and moved house down into the kernel, so no database read was needed to see if they were there or not. During an upgrade, any custom BAdIs could be migrated to become “new” BAdIs. (In my personal experience, the slowdowns I've seen at companies upgrading to version 4.7 is more than reversed when they upgrade to SAP ECC 6.0, and I think the redesign of the BAdIs played a large part in this.)

In BAdI Transaction SE19, you'll still see two sections: one for new BAdIs and one for "classic" BAdIs. As always, "classic" here means "the terrible way we used to do this, of which we are deeply ashamed" (roughly).

► **Enhancements**

The enhancement framework, introduced with SAP ECC 6.0, lets you add your own code virtually anywhere inside any type of standard SAP program: module pool, function module, method, and so on. Moreover, it lets you add your own fields to the data structures and signatures of standard SAP objects. (Before the enhancement framework, all user exits were called at a specific point in a standard SAP program; a developer at SAP had decided that this part of the program was the sort of place that SAP customers would want to add their own logic and thus an ideal place for a user exit. This worked very well, but people being people they are never satisfied. There is always going to be something in standard SAP that a customer wants to change that was not foreseen by the original programmer.) The positive side of the enhancement framework is that you can bend and twist standard SAP to your will like never before. The negative aspect is that if you get it wrong you can really stuff things up like never before. As Voltaire once said to Spiderman, "With great power comes great responsibility." (Something like that.)

Although there are many types of user exits, the focus of this chapter is on the latest and greatest, as mentioned previously: enhancements and new BAdIs. Section 6.1 and Section 6.2 are devoted to enhancements and discuss their types and how to create them. The remaining sections of the chapter—Section 6.3, Section 6.4, and Section 6.5—are about new BAdIs as they are used in the enhancement framework: defining them, implementing them, and calling them.

6.1 Types of Enhancements

There are two types of enhancements: explicit enhancements and implicit enhancements.

6.1.1 Explicit Enhancements

Explicit enhancements are similar to every one of the types of user exits that existed before the enhancement framework: They are enhancements that are

inserted at preordained places in SAP code. However, before SAP was guessing where customers might want to place user exits; with explicit enhancements, the location is based on where SAP itself needs to make changes for industry-specific or country-specific requirements.

You can identify places for explicit enhancements very easily; if you find the statement `ENHANCEMENT_POINT` tacked on to the word `STATIC` somewhere in a standard program, then that's where SAP thinks you might want to add some logic of your own.

Switch Framework

If you see the statement `ENHANCEMENT_POINT` but not the word `STATIC`, then beware: without the `STATIC` addition, such enhancements are controlled by the *switch framework*, which SAP uses to control extra functionality (for example, functionality specific to industry solutions).

After the `ENHANCEMENT_POINT` line, there will usually (but not always) be one or more `ENHANCEMENT` or `ENDENHANCEMENT` constructs with some standard SAP code already inside. This is shown in Figure 6.1, where an SAP programmer has added a `STATIC` variable. (You can then add your own enhancement in addition to anything SAP may already have done; this is covered in Section 6.2.)

```

28 ENHANCEMENT-POINT CONVERSION_EXIT_MATN1_INPUT_02 SPOTS ES_SAPLOMVC STATIC .
29 *?*-Start: CONVERSION_EXIT_MATN1_INPUT_02-----*?*-
30 ENHANCEMENT : MOV_CONV_EXIT_SAPLOMVC. "active version
31 ~~~~~
32 statics: l_func_exits type c. "SR note 1366176
33 ~~~~~
34 -ENDENHANCEMENT.

```

Figure 6.1 Explicit Enhancement Point

6.1.2 Implicit Enhancements

Whereas the points at which explicit enhancements occur are predefined by SAP, implicit enhancement points occur much more frequently. If you're in the ABAP Editor and looking at an executable program, a function module, or a global method, then take the menu option `EDIT • ENHANCEMENT OPERATIONS • SHOW IMPLICIT ENHANCEMENT OPTIONS`, and you'll start to see lines of double quotation marks appearing all over the place, as can be seen in Figure 6.2. More specifically, they will appear at the start and end of every `FORM` routine, the start and end of every function module and method, at the end of every structure definition, and right at the end of an executable program.

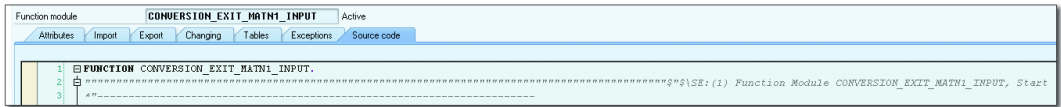


Figure 6.2 Implicit Enhancement Point

However, implicit enhancement points are still limited to the start and end of routines. If you want to add your own code in the middle of a standard SAP routine, then there needs to be an explicit enhancement point added by SAP.

6.2 Creating Enhancements

The process of actually creating an enhancement doesn't change based on whether it is explicit or implicit, although the way to start the creation is slightly different. If you're dealing with an explicit enhancement, then you put your cursor on the line that says `ENHANCEMENT_POINT`; if you are dealing with an implicit enhancement point, then you can position your cursor on the line of double quotation marks.

In both cases, you then navigate to `EDIT • ENHANCEMENT OPERATIONS • CREATE IMPLEMENTATION`, and you will be guided through the process of adding in an enhancement implementation (here, "enhancement" refers to the fact that an area of standard code can be enhanced; an "enhancement implementation" refers to a specific section of custom code that you have written).

You'll now look at creating such an enhancement implementation, first in a procedural program and then in an object-oriented (OO) program.

6.2.1 Procedural Programming

As an example, pretend that one of your custom programs is a standard SAP program—and then butcher it with every possible enhancement option you can imagine. (I would like to stress that I'm not actually recommending you do this; it's just to show what's possible.)

This example will guide you through pretending to change a standard SAP function module. You will add a new optional importing parameter, do something with that value at the start of the function module, and then at the end of the

function module you will send an extra parameter back out based on whatever it was the function module does.

First, go into the function module in display mode, navigate to the **IMPORT** tab that shows importing parameters, and then click the seashell icon (or press **Shift + F4**). This opens the dialog for creating an enhancement implementation (Figure 6.3).

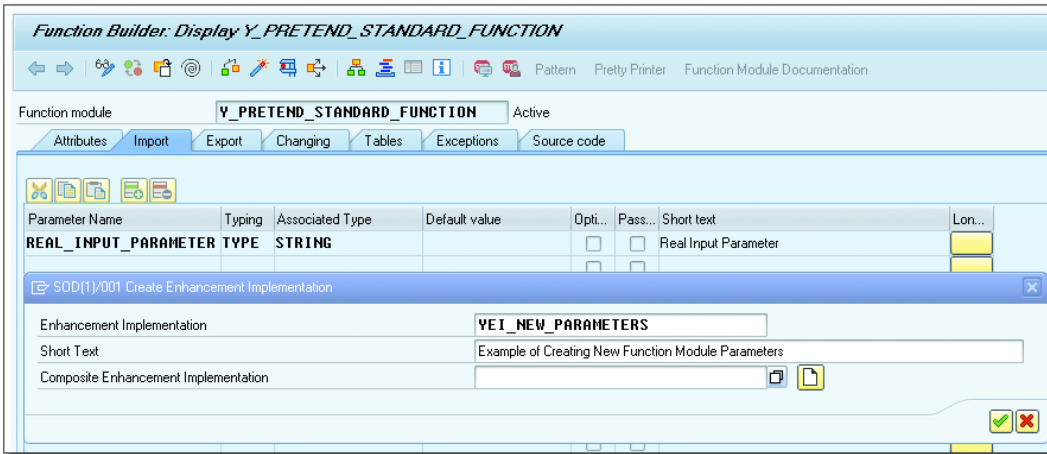


Figure 6.3 Creating an Enhancement Implementation for a Function Module

After entering a name and description for the enhancement implementation and pressing the green checkmark, you find yourself on the screen shown in Figure 6.4. This is where you can add new parameters.

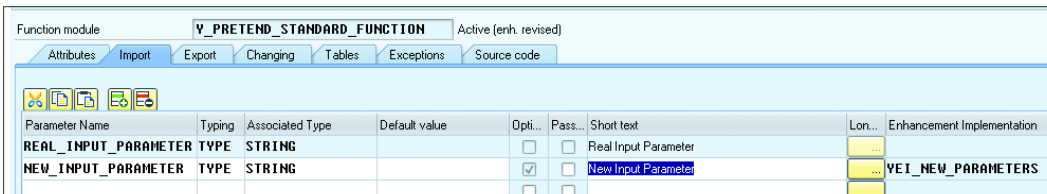


Figure 6.4 Adding a New Parameter to a Standard SAP Function Module

Figure 6.4 shows a list of parameters for the function module. At the top of the list are some gray fields, which represent the standard SAP parameters that you cannot change. The bottom of the list has white fields, where you can add any new parameters that you feel like.

After you have added a new parameter, you will see it has the **OPTIONAL** checkbox selected, in order to avoid breaking any existing programs that may call that module. You can also see that there is a new column called **ENHANCEMENT IMPLEMENTATION** to indicate that the new importing parameter is not SAP standard but was rather added by an enhancement. You can do exactly the same to add a new export parameter (you don't have to create a new enhancement implementation this time; you get asked if you want to use the existing one). As a rule of thumb, all related changes should be in the same enhancement implementation: An enhancement implementation should be able to be switched off without affecting anything, so if you had a new parameter in one implementation and code referring to that parameter in another, then you could not switch them off independently.

This is all lovely, but now you want to do something with these parameters, so you need to change something inside the code of the function module. Go to the **SOURCE CODE** tab of the function module (still in display mode), and navigate to **EDIT • ENHANCEMENT OPERATIONS • SHOW IMPLICIT ENHANCEMENT OPTIONS**. You'll see a line of quotation marks on the first line of the function module. Select that line and navigate to **EDIT • ENHANCEMENT OPTIONS • CREATE IMPLEMENTATION**. You'll see the screen shown in Figure 6.5.

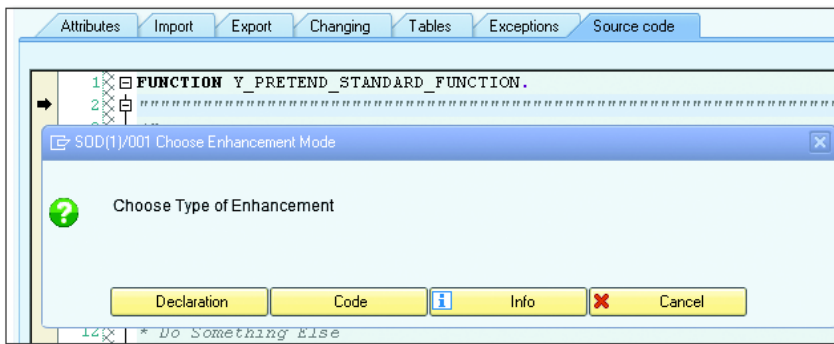


Figure 6.5 Inserting a Source Code Plug-In

Click the **CODE** button, and you will see an **ENHANCEMENT/ENDENHANCEMENT** block in which you can add your own code. Not only do you have access to your new import parameter and the real import parameters of the function module, but you also have access to the global data of the function group and can change it. Be careful!

However, you don't have access to any local variables in the main body of the code, because they're declared after your inserted code. The only way to work with the local variables is if an explicit enhancement spot has been defined in the middle of the code.

In Figure 6.6, you will see two inserted blocks of code: one at the start and one at the end.

```

1  FUNCTION Y_PRETEND_STANDARD_FUNCTION.
2  ~~~~~
3  **$*-Start: 9999-----
4  ENHANCEMENT 1  YEI_FM_SOURCE_CODE.    "inactive version
5  ~~~~~
6  * Do something using the new input parameter
7  ~~~~~
8  ENDENHANCEMENT.
9  ~~~~~
10 **$*-End:  9999-----
11 ~~~~~
12 **"Local Interface:
13 **  IMPORTING
14 **    REFERENCE (REAL_INPUT_PARAMETER) TYPE  STRING
15 **  EXPORTING
16 **    REFERENCE (REAL_OUTPUT_PARAMETER) TYPE  STRING
17 ~~~~~
18 * Do Something
19 * Do Something Else
20 * Do Something Else Yet Again
21 ~~~~~
22 ~~~~~
23 **$*-Start: 9999-----
24 ENHANCEMENT 2  YEI_FM_SOURCE_CODE.    "inactive version
25 ~~~~~
26 * Set new export parameter based on function module results
27 ~~~~~
28 ENDENHANCEMENT.
29 ~~~~~
30 ENDFUNCTION.

```

Figure 6.6 Coding Source Code Plug-Ins

As can be seen, the possibilities are endless here, especially considering that you can do the exact same thing to any FORM routines the function may call—that is, adding code at the start and the end. (Sadly, in some older standard SAP programs there are no FORM statements—the code is in one massive block—and in such cases you're pretty much out of luck.)

It is also possible to go into the TOP include and change that as well. You can not only add your own global variables, you can add extra fields to the structures already defined by SAP (see Figure 6.7).


```

Include          LV_PRETEND_STANDARD_FUNCOP          Inactive
1  FUNCTION-POOL Y_PRETEND_STANDARD_FUNC.           "MESSAGE-ID ..
2
3  □ DATA: BEGIN OF ty_structure,
4      standard_field_one TYPE string,
5      standard_field_two TYPE string,
6  □
7  ~~~~~
8  *$$$-Start: (1)-----
9  ENHANCEMENT 1 YEI_DATA_DECLARATION.           "active version
10 DATA: my_new_field TYPE string.
11 ENDENHANCEMENT.
12 *$$$-End: (1)-----
13 ~~~~~
14 □
15 *$$$-Start: (2)-----
16 ENHANCEMENT 4 YEI_FH_SOURCE_CODE.           "active version
17 DATA: gd_my_new_global_field TYPE string.
18
19 ENDENHANCEMENT.
20 *$$$-End: (2)-----

```

Figure 6.7 Adding New Global Variables

If you want to go to extremes, you could even replace the standard functions with your own custom logic. An example of this is shown in Listing 6.1.

```

CALL FUNCTION 'Z_TOTALLY_DIFFERENT_FUNCTION'
  EXPORTING real_input_parameter = real_input_parameter
            new_input_paramater  = new_input_paramater
  IMPORTING real_output_parameter = real_output_parameter
            new_output_paramater  = new_output_paramater.

RETURN.

```

Listing 6.1 Replacing Standard Functions with Custom Logic

You could do the same with calls to `FORM` routines, but it has to be stressed that this is a last resort. If you do this sort of thing, then you often get the clone problem—in other words, fixes and enhancements to standard SAP code won't apply to the code you are now using.

6.2.2 Object-Oriented Programming

The example discussed thus far only addresses the world of procedural programming, because the bulk of standard SAP programs are written that way. However, in reality, you want all new programs to be object-oriented—so it's important to understand the same concepts in standard classes. In regard to adding extra parameters to existing methods in standard SAP classes, the procedure is exactly the same as it is for the function modules discussed in Section 6.2.1. The result looks like Figure 6.8.

Parameter	Type	Pa...	Op...	Typing Met...	Associated Type	Default value	Description	Enhancement
REAL_IMPORT_PARAMETER	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	STRING		Real Import Parameter	
REAL_EXPORT_PARAMETER	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	STRING		Real Export Parameter	
NEW_IMPORT_PARAMETER	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	STRING		New Import Parameter	YEI_METHOD_PARAMETERS
NEW_EXPORT_PARAMETER	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	STRING		New Export Parameter	YEI_METHOD_PARAMETERS

Figure 6.8 Adding Parameters to a Standard SAP Method

There are also no surprises when enhancing the source code of a standard SAP method. The bonus is that you can see the new parameters in the signature, which you cannot when enhancing function modules. This can be seen in Figure 6.9.

```

Class Builder: Enhancement YEL_METHOD_SOURCE_CODE Change
Method: PRETEND_REAL_METHOD Active

1 METHOD pretend_real_method.
2
3 *$$$-Start: 9999-----
4 ENHANCEMENT 1 YEI_METHOD_SOURCE_CODE. "inactive version
5
6 * This is just the same
7
8 ENDENHANCEMENT.
9 *$$$-End: 9999-----
10
11 * Do Something
12 * Do Something Else
13 * Do Something Else Yet Again
14
15
16 *$$$-Start: 9999-----
17 ENHANCEMENT 2 YEI_METHOD_SOURCE_CODE. "inactive version
18
19 * Also, just the same
20
21 ENDENHANCEMENT.
22 *$$$-End: 9999-----
23 ENDMETHOD.
    
```

Figure 6.9 Enhancing a Standard SAP Method

You can also add your own methods to a standard SAP class; once again, the procedure is just the same, and the result looks like Figure 6.10.

Method	Level	Visib...	Me...	Description	PreExit	PostExit	Overwrite-Exit	Enhancement
PRETEND_REAL_METHOD	Insta...	Pub...		Pretend Real Method				
NEW_METHOD	Insta...	Pub...		New Method added via Enhancement Framework				YEI_METHOD_PARAMETERS

Figure 6.10 Adding your Own Method to a Standard SAP Class

The big bonus is that your new method has access to all the member variables of the class, including the private ones, as you can see in Figure 6.11.

```

Method: NEW_METHOD (Active)
1  METHOD new_method .
2      * MD_PRIVATE_STRING is a private variable of the standard class
3
4      MESSAGE md_private_string TYPE 'I'.
5
6  ENDMETHOD.

```

Figure 6.11 Accessing a Private Method

A Warning from Dr. Who

Many development departments in assorted companies ban the use of the enhancement framework on the grounds that you can do an amazing amount of damage enhancing SAP programs if you don't know what you're doing. As Doctor Who once said, "I've no doubt you could augment an insect to the point where it understood nuclear physics. It would still not be a sensible thing to do" (Robert Holmes, "The Two Doctors"). Take that as a word of warning.

6.3 Defining BADs

As you should know by now, hard-coding of any sort is the work of the devil. Therefore, say that you've avoided hard-coding monster types and (for example) are using a customizing table of some sort to store the specific value for a standard monster, and your program reads these values into a range variable. That removes some hard-coding, but problems come in never ending waves (however,

to quote Steerpike from the novel Gormenghast, “What are problems for, if not for solving?”). The next problem is that in two countries a standard monster is handled the same way, in the United States it's handled a different way, and in Australia nothing needs to be done at that point in the program. One way (not a good way) of addressing this problem is demonstrated in Listing 6.2.

```
IF ( ld_land = 'DE' OR ld_land = 'GB' ) AND
    ld_monster_type IN gr_standard_monster.

    PERFORM something.

ELSEIF ld_land = 'US' AND ld_monster_type IN gr_standard_monster.

    PERFORM something_else.

ELSEIF ld_land = 'AU' AND ld_monster_type IN gr_standard_monster.

* Don't do anything

ENDIF.
```

Listing 6.2 A Suboptimal Way of Coding Logic that Varies by Country

This code has conditional logic based on the country, so you keep having to change the code for each new country added. To fix this problem, this is where you need to take a leaf out of SAP's book and define a BAdI. In such a situation, SAP uses a BAdI, and if it is good enough for SAP, then it's good enough for me.

Defining a BAdI is quite a complicated process, to put it mildly. One of the goals of OO programming is abstraction (to separate the things that change from the things that stay the same), and sometimes to achieve a good level of abstraction things have to be more complicated (have more layers) than you would find in the procedural world. Creating a BAdI is rather like a Russian nesting doll: Each time you create a doll, you then create a bigger one to put the last one into. The outside world (the calling program) only sees the outermost doll; all the complexity is hidden inside.

There are three main steps in defining a BAdI: creating an enhancement spot, creating the BAdI definition, and creating the BAdI interface. Each of these steps is discussed next.

6.3.1 Creating an Enhancement Spot

To create an enhancement spot, which is where you'll be adding the BAdI, open SE80, go to the list of local objects, and then navigate to CREATE • ENHANCEMENT • ENHANCEMENT SPOT. You'll see the pop-up shown in Figure 6.12.

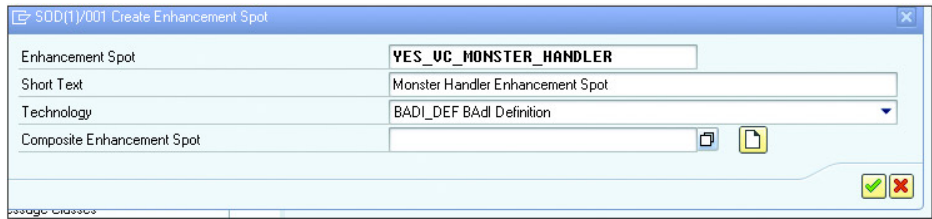


Figure 6.12 Creating an Enhancement Spot

All you have to do on this screen is create a name and description. The TECHNOLOGY field refers to the fact that an enhancement spot is either a source code plugin (i.e., when you add some of your own code inside a standard SAP program), or a BAdI, such as the one you're defining here. The COMPOSITE ENHANCEMENT SPOT field is for grouping several enhancement spots into a tree structure purely for semantic reasons. On the CREATE ENHANCEMENT SPOT screen, once you've filled out all the fields you need, click the good old green checkmark.

6.3.2 Creating the BAdI Definition

Clicking the green checkmark just mentioned brings you to the screen for the next Russian nesting doll you have to create, which is the BAdI definition itself. You'll see a screen with four tabs, one of which is called ENHANCEMENT SPOT ELEMENT DEFINITIONS. On the far left is an icon that looks like a piece of paper, the hover text of which reads CREATE BAdI. What that really means is "Create a BAdI definition," and when you click the icon the pop-up box with the title CREATE BAdI DEFINITION (in the middle of the screen shown in Figure 6.13) appears. Because the steps in creating the BAdI definition are complicated, it's best to break them down into several substeps: configuring the definition, defining a filter, and setting filter checks.

Configuring the Definition

In the screen shown in Figure 6.13, add a name and description for your BAdI. Then, click the green checkmark, and the screen shown in Figure 6.14 appears.

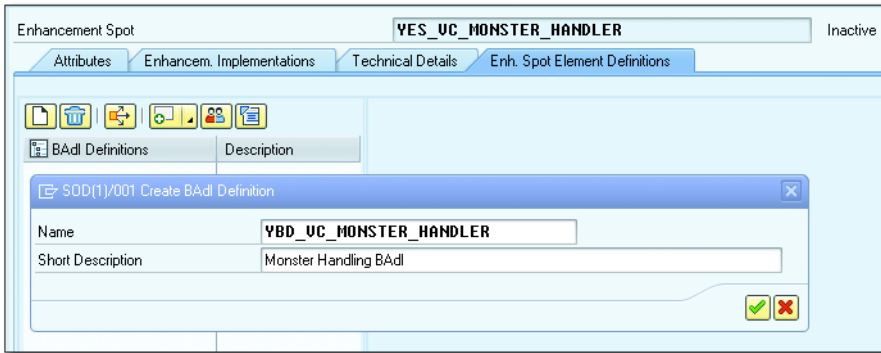


Figure 6.13 Creating a BAdI Definition: Part 1

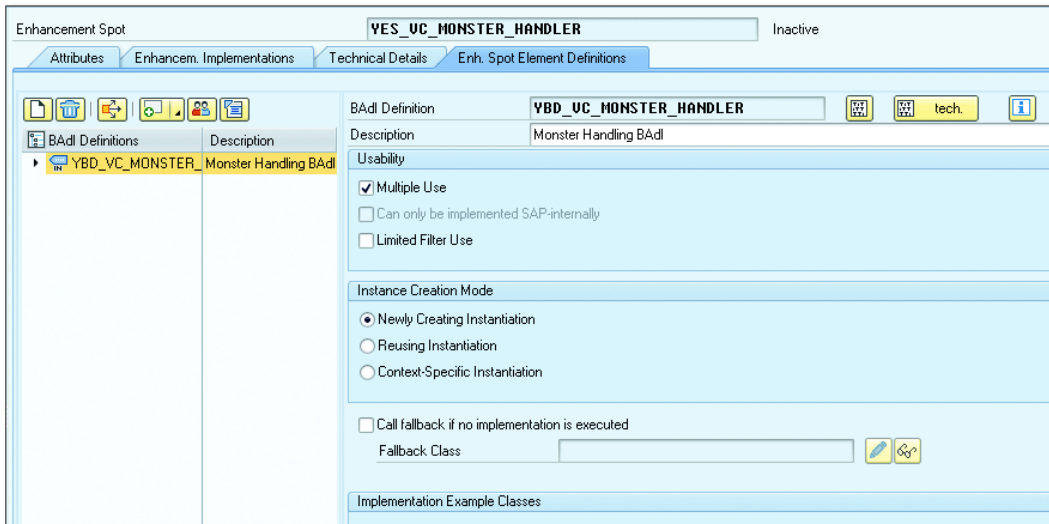


Figure 6.14 Creating a BAdI Definition: Part 2

On the screen in Figure 6.14, there are a lot of nice buttons to press and boxes to check. The first screen area is called **USABILITY**. You will see that by default **MULTIPLE USE** is on. A single-use BAdI is one in which only one implementation of the BAdI is ever going to be called at any one time. A multiple-use BAdI is one in which between one and one trillion billion million (approximately) BAdIs are called in a random order, depending on how many implementations have been created. In OO terms, this is like raising an event, often called the publish-subscribe pattern.

Why would you want to call more than one BAdI implementation? As an example, say that your BAdI is to be called when you're saving a database record. When saving the main record, you may want to update some Z tables, send messages to external systems, send emails, start a workflow, and have SAP recite a poem to the user (you can actually make SAP do just that, but it's probably best not to describe how to do so here). Each of these tasks is unrelated to the others, and not all are done by each country. The United States may send an email and start a workflow, Germany may do only the Z table updates, Australia might do everything, France may just recite the poem, and Poland may do nothing at all. In this case, you define one BAdI implementation for each of these assorted tasks, and then set a filter for the countries that are going to use that task.

Note

In a really complicated application you often find that several developers want to work on the same part of the program at once for different reasons. If the assorted tasks mentioned previously were in one big routine, then any developer changing one part of that routine would lock the whole thing; you could end up in a crazy situation in which you have six development requests all needing to be solved by changing user exit MV45AFZZ and so they have to go into a queue. Solving this problem is just another bonus to the BAdI approach.

The next screen area in Figure 6.14 is titled `INSTANCE CREATION MODE`. Sometimes, you want to buffer the data inside your BAdI so that if it gets called twenty times in a row the program doesn't have to keep going to the database or recalculating all the values. Other times, however, you do want to reread the database or recalculate the values each time.

To make sure the data inside the BAdI is created anew each time, choose `NEWLY CREATED INSTANTIATION`. If you are happy with just reusing the same data each time, then choose `REUSING INSTANTIATION`. The latter is the singleton pattern, in which only one instance of a class can ever exist.

The bottom of the screen shown in Figure 6.14 starts talking about a fallback class; Section 6.4 addresses creating and dealing with such a beast.

Defining a Filter

The next step in creating the BAdI definition is to define a filter (look at the middle of the left-hand side of the screen shown in Figure 6.14). You'll see an icon

with a plus sign, with the hover text CREATE SUBOBJECT. Click that plus sign icon; in the resulting dropdown menu, choose FILTER.

You will notice in Figure 6.15 that the filter types available are the basic data types: character, integer, packed numbers, and the like. These filters are going to end up as input parameters when calling a BAdI, and because everywhere else in the ABAP programming environment you need to be really specific as to what types of data are allowed to be passed in (you usually use data elements), you want the same strong check to be applied here. To achieve this, you need to create some filter checks.

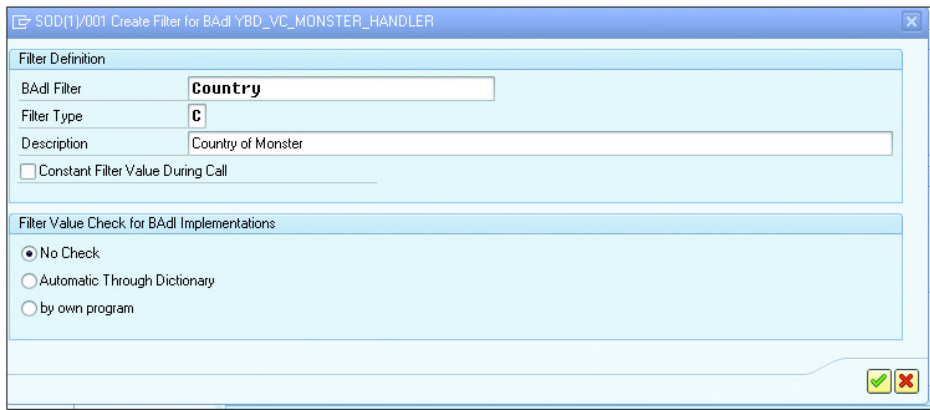


Figure 6.15 Creating a Filter Definition

Creating Filter Checks

You want all the checks you can get to stop programs passing in incorrect data types (e.g., passing in a string to a filter that is expecting a number). Therefore, the next step in setting up filters is to set up a filter value check to make sure the correct type of value is getting passed into the BAdI. If you select the AUTOMATIC THROUGH DICTIONARY radio button, then you can indeed choose a data element (or domain), just as you would when setting the type for an input parameter in a method (Figure 6.16).

You should hope this would then cause a syntax error in the calling program if you tried to pass in, say, a 30-character data element to a value you had defined as a 20-character data element; after all, if you tried to pass in a 30-character value

to a 20-character input field in the signature of a function module or method, you would get a hard error. But no, alas, alack, only the basic type is checked statically; that is, you cannot pass in a number to a filter expecting a string or vice versa. The runtime system must look at the data element and derive whether it's numeric or not and not delve any further.

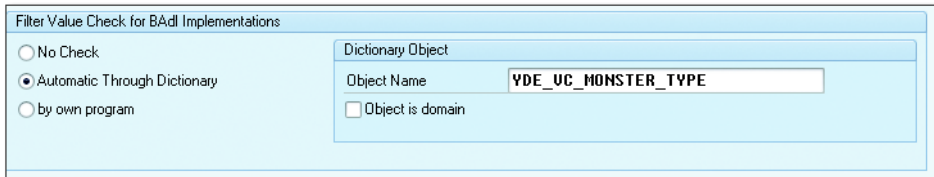


Figure 6.16 Checking Filter Values Based on Data Elements

All is not lost, however. What does happen is that, later on, when you choose the filter value, the `F4` help shows values from the domain, and only those values are allowed. In other words, this check is aimed at the programmer who defines the BAdI implementation as opposed to the programmer who wants to call that implementation.

Turning on the data element–based check does have the advantage that it makes it impossible to define a nonexistent value as a filter value, so it's best not to select the `NO CHECK` option at definition time. A no check option means that anything at all can be passed in, which is a disaster waiting to happen. The BAdI might get called intermittently, seemingly at random, confusing the poor old users no end.

A check you can switch on—which causes a hard syntax error in the calling code—is enabled by selecting the `CONSTANT FILTER VALUE DURING CALL` checkbox that can be seen in Figure 6.15. If this checkbox is selected, then the BAdI will cause a syntax error unless a constant is passed into the filter. Because the programmer who creates the definition defines the constant values, this prevents calling programs from trying to pass in arbitrary values.

With this constant check activated, the only way you can call the BAdI in your program is by something like the code shown in Listing 6.3. In this example, the programmer writing the calling code is forced to use a constant for the country to stop him from writing “UK” instead of “GB”, “UK” being a value that the BAdI does not expect.

```

GET BADI lo_monster_badi
  FILTERS
    country      = yif_vc_monster_handler=>great_britain
    monster_type = ld_monster_type_name.

```

Listing 6.3 Forcing a Constant to Be Passed in to a BAdI Call

Note

Although limiting the possible input values to a filter to hard-coded constants stops the calling program from passing in incorrect values, this approach is not really that useful most of the time, because you tend to want the freedom to pass a variable in. In fact, most of this chapter is about avoiding hard-coding of any sort.

The next option for setting a check on filter values is one that may appeal to you: defining a check via your own program. Ninety-nine times out of 100, a check on a data element ought to be good enough, but there are always going to be cases in which your specific needs aren't handled by the standard system; in effect, what you have here is a user exit for the process of defining user exits!

As you can see in Figure 6.17, this process involves choosing a class name and some information about the filter value data field.

The screenshot shows a dialog box titled "Filter Value Check for BAdI Implementations". On the left, there are three radio buttons: "No Check", "Automatic Through Dictionary", and "by own program" (which is selected). On the right, there is a "Program" section with three input fields: "Class" containing "YCL_FILTER_TEST", "Input Length" containing "30", and "Decimal Places" which is empty.

Figure 6.17 Checking a Filter Value via Your Own Program

If you then double-click on the class name you've chosen, you're taken into SE24 to create the class. The important thing is that the system automatically creates a class that implements the `IF_FILTER_CHECK_AND_F4` interface. Then, you just need to create an implementation of the four methods in that interface, which are as follows:

► SHOW_F4_ICON

Set the returning parameter `SHOW_F4_ICON` to be `ABAP_TRUE`, and then you will be alerting any programmer who wants to choose a filter value while implementing a BAdI that an `F4` help is available for choosing the filter value.

► **F4**

Most ABAP programmers will be familiar with the `PROCESS ON VALUE REQUEST` concept in dialog programs, and this method is exactly the same. The method has four changing parameters for the possible types of filter values—character, integer, and what have you—and you need to program a pop-up from which the programmer can choose a value. As an example, maybe the data element is an integer, but you only want the programmer to choose multiples of five.

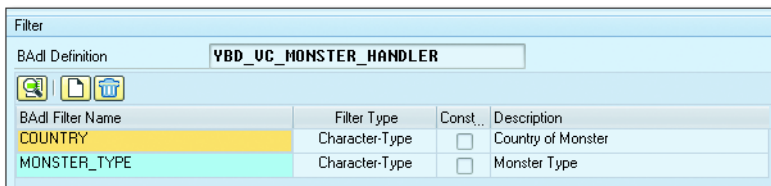
► **VALUE_CHECK**

Here, you are given the filter value input by the programmer, and you program your own logic to see if the value is valid or not; in the example in the previous bullet point, perhaps the value needs to be a multiple of five. If the program decides the value is no good, then exception `CX_ENH_BADI_FLTR_VALUE_INVALID` needs to be raised.

► **INIT**

As you might imagine, this is called before any other method in the class is used. Four different values are passed in: the enhancement spot (e.g., `YES_VC_MONSTER_HANDLER`), the BAdI name (e.g., `YBD_VC_MONSTER_HANDLER`), the enhancement name (e.g., `YEI_VC_MONSTER_HANDLER_GB_EM`), and the implementation name (e.g., `ZBI_VC_MONSTER_HANDLER_GB_EM`). The idea is that if you so desire you could use those input values to set some sort of member variables accessible by the other methods to change the logic based on the exact implementation name (or whatever). This does seem like cracking a nut with a sledgehammer, but it's always better to have options that you probably never need to use than to not have them and suddenly need them.

After you have added a filter for country with no check and a filter for monster type using your own program, the screen will look like the one shown in Figure 6.18.



BAdI Filter Name	Filter Type	Const...	Description
COUNTRY	Character-Type	<input type="checkbox"/>	Country of Monster
MONSTER_TYPE	Character-Type	<input type="checkbox"/>	Monster Type

Figure 6.18 Defining Multiple Filter Fields

6.3.3 Creating the BAdI Interface

The last Russian nesting doll to create in the BAdI definition is the interface for the BAdI—that is, what methods BAdI implementations are going to have and what parameters and exceptions those methods have. To create this interface, click on the arrow on the left of the BAdI definition, and double-click the INTERFACE icon in the tree that pops up.

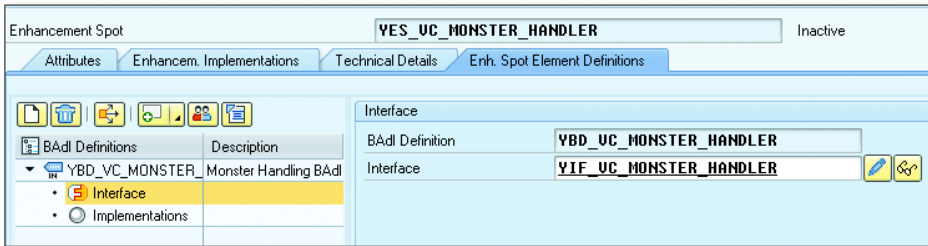


Figure 6.19 Creating a BAdI Interface: Part 1

Most developers are used to SE11 or SE38 or SE24 for creating various repository objects in which the icons are arranged in the order DISPLAY, CHANGE, CREATE. To keep you on your toes when creating the BAdI interface, the first two options are reversed to CHANGE, DISPLAY...and there is no CREATE option in sight (Figure 6.19). What you need to do is create the interface by clicking the CHANGE icon, after which you are asked if you want to create the interface. Then, you'll end up at the standard SE24 screen (Figure 6.20).



Figure 6.20 Creating a BAdI Interface: Part 2

This is exactly the same as defining a normal interface, with the exception that multiple-use BAdIs can only have importing and changing parameters. This is because they are called in a random order, so exporting or returning parameters make no sense, because only the last implementation called would be meaningful. You're now finished with the definition of the BAdI (rather like saying you

want to go to a restaurant); now, it's time to deal with the implementation of the BAdI (rather like saying what specific restaurant you want to go to).

6.4 Implementing BAdIs

After you have completed the steps in the previous section, the BAdI definition is complete. You should still be on the BADI DEFINITION screen shown in Figure 6.14. Now, the time has come to fill in the FALLBACK CLASS section at the bottom of the screen.

What you are about to create is a default implementation (the so-called fallback-class). In Figure 6.21, you can see a ticked checkbox indicating that you want a fallback class, and that class is given a name. (You really only need a fallback class for a single-use BAdI, because in that case at least one implementation is mandatory. With a multiple-use BAdI, it's fine if no implementations at all are called.)

The screenshot shows the SAP BADI DEFINITION screen for the BAdI 'YBD_UC_MONSTER_HANDLER'. The description is 'Monster Handling BAdI' and the interface is 'YIF_UC_MONSTER_HANDLER'. The 'Usability' section has 'Multiple Use' checked. The 'Instance Creation Mode' section has 'Newly Creating Instantiation' selected. The 'Call fallback if no implementation is executed' checkbox is checked, and the 'Fallback Class' is set to 'YCL_UC_MONSTER_HANDLER'. The 'Implementation Example Classes' section shows a table with one entry: 'YCL_VC_MONSTER_HANDLER' with the description 'Class for BAdI: YVC_BD_MONSTER_HANDLER'.

Example Classes	Description
YCL_VC_MONSTER_HANDLER	Class for BAdI: YVC_BD_MONSTER_HANDLER

Figure 6.21 Creating a Fallback Class

This takes you into SE24 again, so nothing to see here; you create the class in the normal way. The system does ask you if you want the fallback class to show up on the BAdI definition table as an example implementation, so you could have an

empty implementation filled solely with comments about how you envisage other programmers creating implementations of this.

Tip

If you do put any code in here, then be sure to remove the `final` indicator in the properties, in case subsequent implementations want to inherit from this example class.

At this point, you already have a default implementation, which you don't really need in this example; now is the time to create one of the real implementations. When you are in the screen that defines the enhancement spot, you will see an icon that looks like two cartoon people standing side by side, with the hover text `CREATE BADI IMPLEMENTATION`.

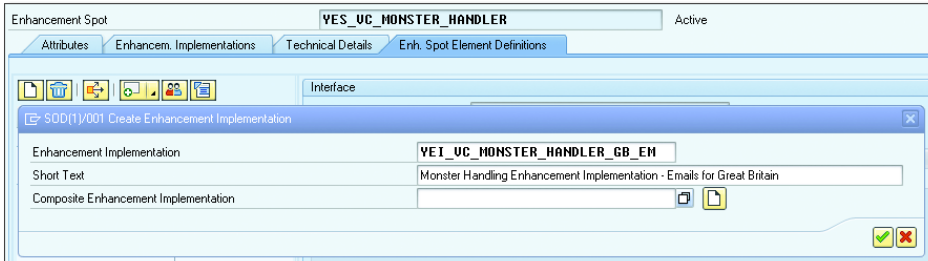


Figure 6.22 Create Implementation

The analogy of Russian nesting dolls is still in full force. There are two parts to creating a BAdI implementation: First, you create an enhancement implementation (Figure 6.22), and then instantly you are asked to create the next Russian nesting doll, which is the BAdI implementation (Figure 6.23).



Figure 6.23 Creating a BAdI Implementation: Part 1

Due to the fact that you created a fallback class earlier, you're asked if you want to copy it as a starting point or create a new empty class. You're going to do the latter in this example, so click the `EMPTY CLASS` button, as shown in Figure 6.24.

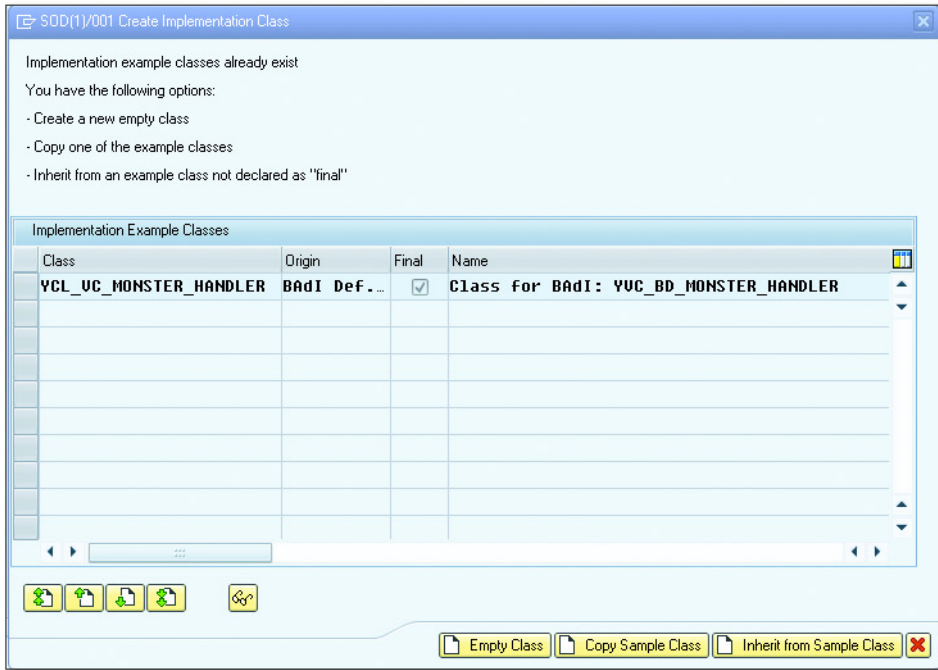


Figure 6.24 Creating an Implementation Class

You'll now see another screen (Figure 6.25), showing the enhancement implementation and how it's linked to the corresponding definition. Expand the tree on the left-hand side so that you can see the words IMPLEMENTING CLASS, and then click on the same.

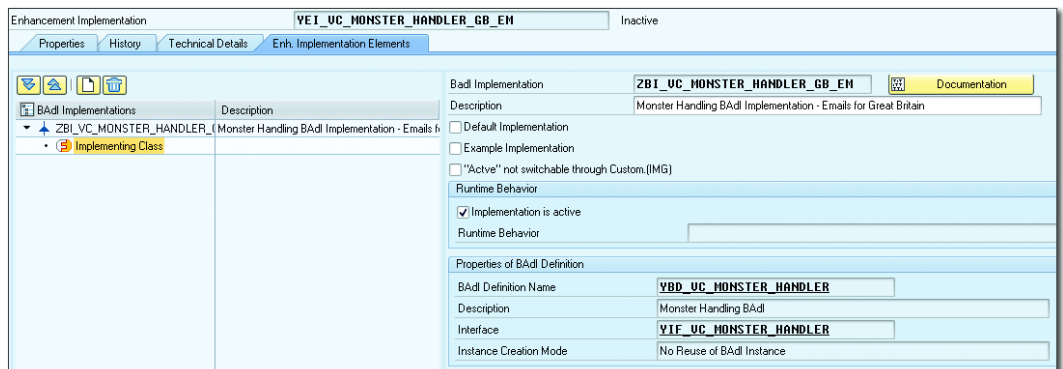


Figure 6.25 Creating a BAdI Implementation: Part 2

You will see an overview of the implementing class (Figure 6.26). On the right-hand side, you can either change the class as a whole (for example, create a new method or attribute) or click an existing method to change (or create) the implementation.

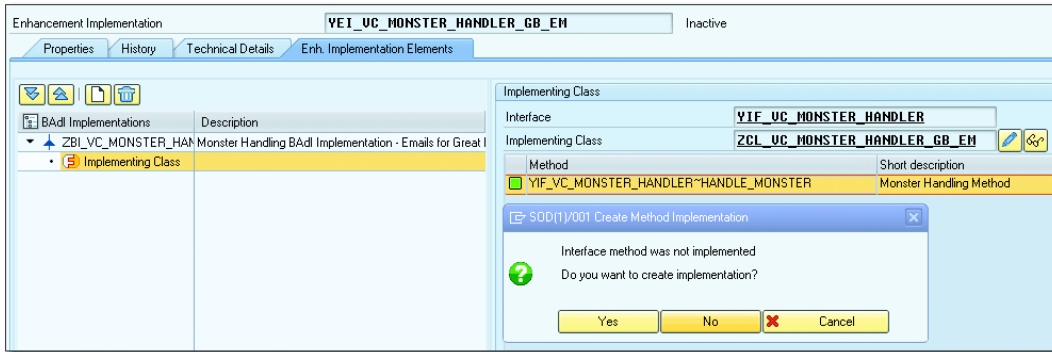


Figure 6.26 Changing a BAdI Class

Now, you are back in SE24 once more and can code as normal. When you're done, back on the ENHANCEMENT IMPLEMENTATION screen, the magic words THE IMPLEMENTATION WILL BE CALLED NOW appear, which means that everything is defined properly.

You set up some filters in the implementation definition earlier; that is, you said what sort of data you were going to use as filters. Now that it's implementation time, you're going to say what specific values to use as filters for this implementation.

In this example, you only want your BAdI called if this is Great Britain and if your monster is a standard monster. On the ENHANCEMENT IMPLEMENTATION screen, expand the tree on the left, and click FILTER VALUES. The subscreen on the right then changes to a blank area ready to be filled with filters. Click the CREATE COMBINATION icon. Because you need both conditions to be true (AND), select both filters in the ensuing pop-up box (Figure 6.27). If you wanted the BAdI to run if either was true (OR), then you would select one at a time.

The screen then fills up with question marks, indicating that you need to add some filter values. Go into each line, and add the specific value needed for this BAdI—for example, the country GB (Figure 6.28).

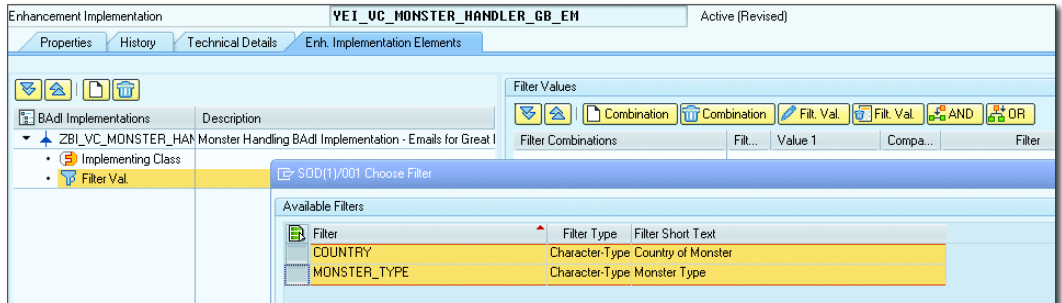


Figure 6.27 Setting Filter Values

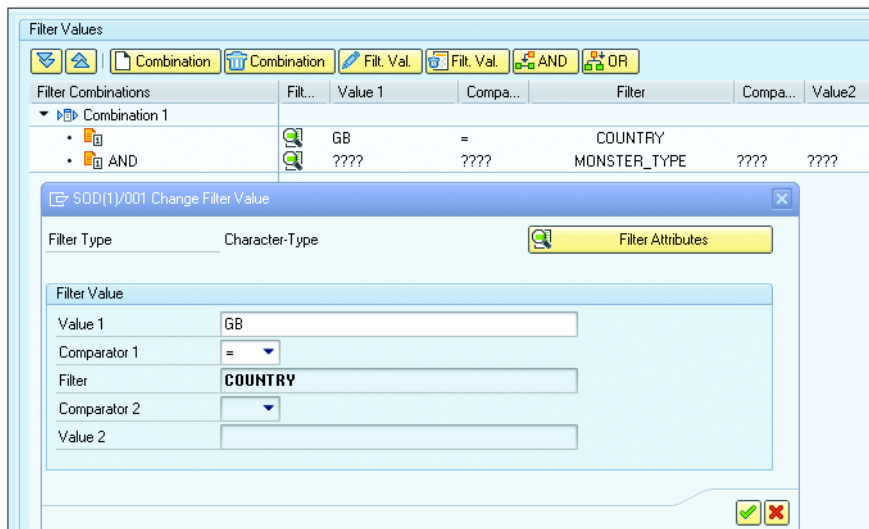


Figure 6.28 Adding a Specific Value for a Filter

That took some doing, but now you're finished. Now it's time to explain how to call this BAdI from your own program.

6.5 Calling BADIs

Listing 6.4 is the code for calling the BAdI. By now, you know that the filter values for the BAdI are constants. Almost all the time, you can derive organizational elements from a running program (e.g., getting the current country from the

company code or plant being processed). In the current example, this enables you to avoid hard-coding the country value that gets passed into the BAdI filter. As you can see, there is also some conditional logic to see what sort of monster you're dealing with, and then code that tasks a BAdI with dealing with the chosen monster type.

This code is yet another example of a separation of concerns, and every time you achieve a separation of concerns, you get to pass GO and collect a million dollars. The part of the program that determines the monster type doesn't need to know what you're going to do with that information, and the part that deals with a specific monster type doesn't need to know how you worked out what monster type you're dealing with.

```

DATA: lo_monster_badi TYPE REF TO ybd_vc_monster_handler.

ld_land = ls_company_code-land."GB

IF ld_monster_type IN gr_standard_monster.
  ld_monster_type_name = 'STANDARD_MONSTER'.
ELSEIF ld_monster_type IN gr_extra_scary_monster.
  ld_monster_type_name = 'EXTRA_SCARY_MONSTER'.
ELSE.
  "Unexpected situation, do some error handling
ENDIF.

TRY.
  GET BADI lo_monster_badi
  FILTERS
    country      = ld_land
    monster_type = ld_monster_type_name.

  CALL BADI lo_monster_badi->handle_monster
  CHANGING
    co_monster = lo_monster.

  CATCH ycx_monster_exception.
    "Do some error handling
  CATCH cx_badi_not_implemented.
    "This exception only happens for single-use BAdIs
ENDTRY.

```

Listing 6.4 Calling a BAdI

When you compare Listing 6.4 to Listing 6.2, you'll see that the logic is no longer welded together. Before, there was a convoluted IF/ELSE construct in which each branch had values for the country, the monster type, and what routine to call based on the first two. Now all three are separated: there's some code to work out the country, some code to work out the monster type, and some code to decide what to do with this information. Each part of the program is doing one thing only, as per good object-oriented design.

Note

You could go even further and put each of the three sections in its own method, thus allowing you to change the logic inside one section without affecting the other two. This example doesn't do this only because it would have obscured the point being made.

The benefits of such an approach are as follows: If a new customizing value is added for a new country, then you don't have to change the part of the program that gets the information about what type of monster you have. In addition, if a country changes the way it deals with a given monster type or a new country comes along with a totally new method of monster handling, then you don't have to change the part of the program that deals with a given type of monster. This is the open-closed principle on steroids.

Instead, if you have to add a new country to the system, then you create a new implementation of the BAdI that handles country-specific behavior. The calling program remains unchanged. If a new activity needs to be performed (such as sending a message to a new external system), then you add a new implementation for any affected countries. Figure 6.29 shows how you define a complex filter such that the BAdI gets called if you're in either the United Kingdom or Australia and are dealing with a standard monster.

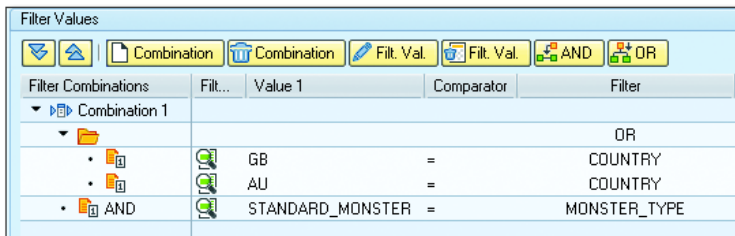


Figure 6.29 Complex Filter

Another example could involve an interface from SAP to a monster-making machine, of which there are 10 possible varieties. Some of them have mandatory data fields, varying depending on the company that made the machine. These mandatory fields would be the same in any country where you were using that variety of machine. However, some (but not all) countries have laws governing what mandatory information must appear on the external machine's display; those laws were created before any machine was even invented, and they are naturally different between countries.

You can handle this by creating one BAdI implementation for each variety of machine to validate that each machine-specific mandatory field is filled. Then, create BAdI implementations for each country that has special laws to validate the relevant fields as well. When you call the BAdI, any machine-specific checks and any country-specific checks are performed; if either set of checks fails, then an error is raised. There may be no country-specific checks or no machine-specific checks, and you may need to introduce a new country-specific check if a new law is created. None of this matters to the calling program: It only needs to handle any error messages sent to it.

6.6 Summary

This chapter gave you a crash course on user exits, focusing on the two most recent types: BAdIs and the enhancement framework. You should now have a good idea of how (and why!) to use these in your own code.

In the last code sample (Listing 6.4), you will notice comments such as "Do some error handling." This was left deliberately vague, because it leads in to the next chapter, which is all about error handling.

Recommended Reading

- ▶ How To Define a New BAdI Within the Enhancement Framework: <http://scn.sap.com/people/thomas.weiss/blog/2006/04/03/how-to-define-a-new-badi-within-the-enhancement-framework--part-3-of-the-series> (Thomas Weiss)
- ▶ Back to the Future: <http://scn.sap.com/community/abap/blog/2012/08/31/back-to-the-future--part-06> (Paul Hardy)

PART II

Business Logic Layer

There is no exception to the rule that every rule has an exception.
—James Thurber

7 Exception Classes and Design by Contract

Most people have heard the phrase, “There are only two sure things in life: death and taxes.” With computer programs, you can introduce another sure thing: unexpected situations. Sure, you’d like the program flow to always follow the happy path, on which everything proceeds according to the original design. However, another common saying is, “No battle plan survives contact with the enemy.” In real life, as soon as the program is running, the user is going to type in something unexpected, or an external system will send in something unexpected, or one of a million possible other things will go wrong. These things are referred to as *exceptions*.

To better understand what an exception really is, imagine a situation where you have 100 monsters to be dispatched, and you ask the user how many he wants every hour. Then, you divide 100 by the user input to see how long it will take to dispatch all the monsters (e.g., 20 monsters per hour = $100/20$, meaning that it takes five hours in total). Here are some possible exceptions:

- ▶ The user enters zero as the input value, leading to a division by zero dump.
- ▶ The user enters something like “hello” instead of a number.
- ▶ The user enters a correct value, but then a database read on a configuration table that you thought would always return a value doesn’t, because someone accidentally deleted the configuration table entries.
- ▶ A wombat chews through the underground cable, causing the user to lose his connection to the SAP central instance.
- ▶ There’s a programming error in part of the program code, which only gets triggered in rare circumstances (e.g., only when the user enters a value higher

than 10 on a Thursday in June), and a short dump happens when this situation does in fact occur.

Not All Problems Are Exceptions!

Sometimes, a program won't work for a reason that does not qualify as an exception. For example:

- ▶ The user clicks the CANCEL button in the pop-up box.
- ▶ The user enters a number when the monster object is locked by another user, which means that the monster cannot be processed at that moment—but can be as soon as the other user releases the lock.
- ▶ A user tries to run the transaction on October 31 and an error is sent. This isn't because of a problem with the program, but because all monsters refuse to work on this day—in case they are mistaken for children dressed up as monsters and going trick-or-treating, which would be the biggest humiliation that could happen to a monster.

You still have to deal with these situations, but because you know for a fact they're going to happen, dealing with such events is not exception handling.

Sadly, crossing your fingers and hoping for the best doesn't cut it when “impossible” things happen. Therefore, because you know for a fact things are going to go wrong, you'd better design your programs to deal with such situations. However, the sad truth is that many programmers gloss over this area, citing a lack of time or some such, which is a false economy in my opinion. Never fear: *ABAP to the Future* to the rescue!

Exceptions can be handled in many ways, but SAP recommends exception classes. The ability to create such classes has been around in ABAP for a while now, but many people still struggle with their use. This chapter shows some practical examples of where and how to use exception classes, including a discussion of the design by contract pattern in ABAP.

Error handling is an exact fit with general object-oriented (OO) principles in that there is a separation of concerns: three different activities. This can best be explained by an analogy: If a house caught fire, there would be several parts to fixing that problem, each of which is addressed by a section in this chapter:

- ▶ Even before the fire breaks out, the emergency services make plans for such events. As and when a call comes in, the telephone operator has to decide

which department is best able to deal with this situation: perhaps the police, the fishmonger, or the fire department. Likewise, your program has to make a similar decision—so this chapter starts by discussing the different types of exception classes, which allow you to choose the appropriate part of the program in which to deal with a raised error signal (Section 7.1).

- ▶ Once you notice your house is on fire, you phone 999 (United Kingdom) or 911 (United States) to report the situation and tell them your address. Then, the fire department comes around and puts out the fire. To express this in programming terms, you need to design an exception class to respond to specific types of situations and store pertinent information for onward transmission to whatever will deal with the problem—and then you have to actually deal with that problem. This is what exception classes are all about (Section 7.2).
- ▶ After the event, when you make an insurance claim, the insurance company will investigate the cause of the blaze. Did you accidentally set your house on fire yourself? Did you hire a cook to prepare a banquet, and the cook accidentally set the house on fire? In programming terms, the concept of design by contract comes into play here. This means determining if the cause of the error is at the start or end of each routine (Section 7.3).

Exception Terminology

This chapter uses five phrases, all with the word *exception* in them. To make sure you're not confused, keep the following in mind:

- ▶ An *exception* is an unexpected situation. This is a real world event.
- ▶ *Exception handling* is how a program recognizes and reacts to such an event.
- ▶ *Exception classes* are one possible way to implement exception handling; as you should already have guessed from the chapter title, this is the focus of the chapter.
- ▶ *Raising an exception* is the process of creating an instance of an exception class during the course of exception handling.
- ▶ An *exception object* is the instance created when raising an exception.

7.1 Types of Exception Classes

The first decision you'll have to make when designing exception classes is what type you need. The type you choose will determine things like whether you handle the error locally or remotely and where you can put handling code. With that

in mind, this section talks about each type of exception class and then concludes with a handy little diagram that shows you how to choose the appropriate one.

7.1.1 Static Check (Local or Nearby Handling)

When you create your own Z exception class, the system proposes a standard exception class to inherit from, and that class is `CX_STATIC_CHECK`. Classes that inherit from base class `CX_STATIC_CHECK` have the following behavior:

- ▶ If you raise an exception within a given routine, then you must either handle (CATCH) it within the routine or declare the exception class in the method signature using the `RAISING` keyword. The exception object cannot be propagated outside the routine unless its class is specifically mentioned in the signature. In order to demonstrate these techniques, Listing 7.1 does both: mention the exception class in the signature and handle it locally. (Usually, however, you would do one or the other.)
- ▶ If you raise the exception but the exception class is not declared in the signature nor caught locally, then when you do the syntax check a hard error is given, and the code will not compile.

```
FORM something_static RAISING cx_salv_error.

IF something.
    "If you don't want to handle the error locally
    RAISE EXCEPTION TYPE cx_salv_error. "Static
ENDIF.

TRY.
    IF something_else.
        RAISE EXCEPTION TYPE cx_salv_error. "Static
    ENDIF.
    CATCH cx_salv_error.
        "If you do want to handle the error locally
    ENDRY.
ENDFORM.          " SOMETHING_STATIC
```

Listing 7.1 How to Handle an Exception Locally or Propagate It up the Stack

Warning: Houston, We Have a Problem

As was just mentioned, if you call a method that has a static exception class in the signature and do not handle it in the calling method, then you get a syntax error. If you do the same thing using `FORM` routines, then there's no syntax error, but the program would

dump at runtime if the exception was raised. This is one of many concrete reasons why using an OO method is better than the FORM routine equivalent.

The advantage of this type of exception class is that it makes it blindingly obvious to any calling programs that a certain type of problem might occur, and so the calling program is forced by the syntax check to implement some error handling code. Declaring an exception class in the signature has been described as putting up a big “Beware of Dog” sign. The called routine has made the caller aware that an exception of this type might be raised, and it's the responsibility of the caller to deal with it; otherwise, the caller will be bitten.

An example of when you might use a `CX_STATIC_CHECK` check is as follows. You have a routine that determines where a given color of monster likes to hide before jumping out to eat people: green monsters like to hide under the bed, blue monsters like to hide in the cupboard, and purple monsters like to hide in the fridge drinking beer and playing the piano. However, if any other color is passed in to the routine, then no hiding place can be determined, and an exception is raised. Because you cannot be sure what program will call your routine in the future, you want to force the calling program to handle the situation in which an exception is raised (e.g., by default, set the hiding place to be behind the curtains, rather than the calling program presuming the routine will always come back with an answer and getting into trouble when it does not). In such a case, a `STATIC` check is the way forward.

That being said, having to declare exception classes in the signature has the potential of making programs more complicated than they need to be. As an example, if you had a program with a really deep nesting level and only the very highest level could reasonably know what to do with the error, then you would have to declare the exception class all the way down. It's possible that the deeper you went, the longer the signature of the methods would become.

In conclusion, `CX_STATIC_CHECK` is best used for local (or nearby) handling that does not require intermediate methods. When you raise such an exception and don't handle it locally, you have to propagate the exception upwards through the call stack until a method that can deal with the problem is found. With a `CX_STATIC_CHECK`, each such intermediate method has to define the exception class in its signature. Each intermediate method needs to be able to catch the exception and propagate it upwards in case something that it calls raises the exception — so when

there are more than one or two intermediate methods, `CX_STATIC_CHECK` is not the ideal exception class to inherit from.

7.1.2 Dynamic Check (Local or Nearby Handling)

Exception classes that inherit from the `CX_DYNAMIC_CHECK` class are identical to classes that inherit from `CX_STATIC_CHECK` in every respect except one. In the case of `CX_STATIC_CHECK`, if you raise an exception within a routine, then you must either handle it locally or pass it upwards by mentioning the exception class in the signature; if you don't, then a syntax error occurs. In the case of `CX_DYNAMIC_CHECK`, however, if you raise an exception within a routine, then you must either handle it locally or pass it upwards by mentioning the exception class in the signature; if you don't, then *no* syntax error occurs. You get no sort of warning at all, much less an error. However, if such an exception is raised at runtime and not handled locally or passed on, then a short dump occurs.

This has confused a lot of people over the years. This type of exception class seems to have no added benefit compared to `CX_STATIC_CHECK` but a really big downside, in that you are likely to get a short dump without the syntax check warning you well in advance. Why in the world would anyone ever use this type of exception class?

The official explanation seems to be that you should use a dynamic check when you can use preconditions to make sure you never get to the line that raises a dynamic exception, meaning that the situation is so rare as to be deemed impossible. This means that the programmer is supposed to try and ensure that the line that raises the dynamic exception never gets reached, which makes raising such an exception and then not dealing with it rather like an `ASSERT` statement.

In conclusion, if you do want to use `CX_DYNAMIC_CHECK` for whatever reason, then the handling ideally has to be done in the method in which the exception is raised (i.e., locally). You could pass it upwards by specifying the exception class in the signature, but for an exception that's never supposed to happen, that seems illogical, Captain.

7.1.3 No Check (Remote Handling)

Exception classes that inherit from `CX_NO_CHECK` are the only ones that can leave the routines in which they are raised without the exception class being explicitly

declared in the routine signature. In essence, you can throw the exception object up in the air, and it will travel up the call stack looking for an appropriate handler. It will stop at the first one it finds, and if it can't find any, then a short dump will occur.

In most documentation, this type of exception class is described as being useful for problems that can occur anywhere (such as the system running out of memory), and it would clearly be impractical to declare a long list of such exception classes in every single method signature throughout the system. However, this also has a much wider application. As an example, say that you have an interactive ALV list program that calls a Monster Monitor. The Monster Monitor displays a list of active monsters, and you can select a monster by clicking the checkbox next to the monster. Then, you can choose one of 15 icons along the top of the screen to issue various commands to that monster. The code for this is shown in Listing 7.2.

```
FORM user_command USING id_ucomm    TYPE sy-ucomm
                    is_selfield TYPE slis_selfield.

TRY.
  PERFORM get_chosen_monster USING    is_selfield
                                CHANGING mo_chosen_monster.

  PERFORM lock_monster USING mo_chosen_monster.

  CASE id_ucomm.
    WHEN 'HOWL'.
      mo_chosen_monster->howl_at_moon( ).
    WHEN 'TERROR'.
      mo_chosen_monster->terrorise_village( ).
    WHEN 'SUBPRIME'.
      "Most monstrous activity possible
      mo_chosen_monster->sell_mortgages( ).
"Etc...
    WHEN OTHERS.
      MESSAGE 'Function is not available' TYPE 'I'.
  ENDCASE.

  CATCH zcx_bc_user_cancelled.
    MESSAGE 'Processing Cancelled'(349) TYPE 'S'.
    PERFORM unlock_monster USING mo_chosen_monster.
    RETURN.
ENDTRY.
```

```
PERFORM unlock_monster USING mo_chosen_monster.
```

```
ENDFORM. "User Command
```

Listing 7.2 Using a NO_CHECK Exception During User Command Processing

In the preceding example, `ZCX_BC_USER_CANCELLED` inherits from `CX_NO_CHECK`. Almost every one of the 15 possible user commands shows the user one or more ARE YOU SURE? pop-up boxes, any of which the user can cancel out of, aborting the whole process, and the code that calls the pop-ups can be many levels down the call stack from the `USER_COMMAND` routine.

Traditionally, in order to deal with users deciding to cancel processing, you'd have to declare a global variable called `GF_CANCEL` or some such, and after coming back from each of the 15 different pop-up boxes you'd have a check, like `CHECK GF_CANCEL = ABAP_FALSE`, and would repeat this statement at every level in the call stack. Remember, for some user commands you might show the user three different sorts of pop-up boxes, one after the other, any one of which the user can cancel out of.

However, the code in Listing 7.2 takes a different approach: the instant the user clicks a CANCEL button, you raise the `ZCX_BC_USER_CANCELLED` exception, and the program flow jumps right back to the top of the call stack, unlocks the monster, and you're done. This prevents the code from being littered with checks for the same thing.

In conclusion, `CX_NO_CHECK` seems ideal when your routine has no idea what to do with the problem encountered and you want to pass the error information upwards until some part of the program does know how to respond.

7.1.4 Deciding which Type of Exception Class to Use

Many people are trying to work out which of the three types of exception class to use in what situation, and most documentation on the subject is very vague. There are no hard and fast rules, but Figure 7.1 is a little decision tree to help you decide what type of exception class to use.

For example, imagine you have created a public method in a global class that is going to be reused all over the place. If this method gets into trouble when an exceptional situation comes along and cannot handle the problem locally, then

maybe it's best to declare a static exception class in the signature so that potential calling programs are aware of the danger.

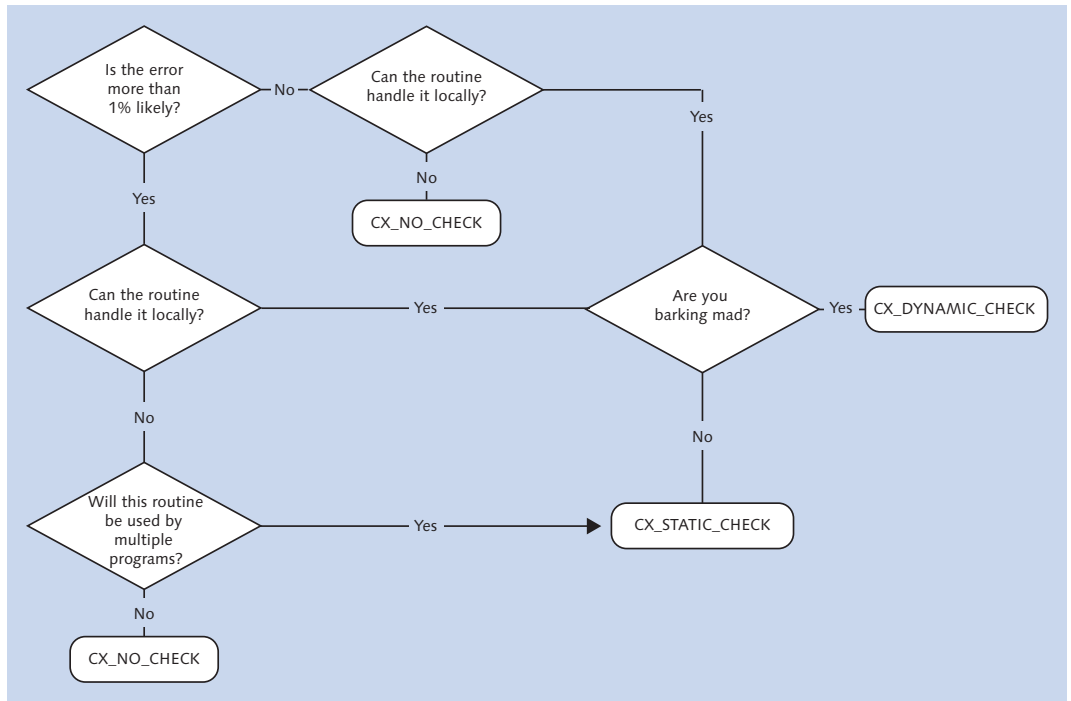


Figure 7.1 Exception Class Decision Tree

7.2 Designing Exception Classes

Java programmers often say that everything is an object. When it comes to exception classes, they're right: exception classes are classes that represent an unexpected situation, and the objects they create are instances of one specific error situation of this type. Realistically, you'll have to design your own custom exception classes so that you can have application-specific attributes. (There are a lot of standard SAP exception classes, but they tend to have names like `CX_JAPANESE_MICE_RACING_ERROR`, which is a bit too specific for general reuse.)

There are several steps in the process of designing an exception class, which can be described as followed:

1. Create an exception class to describe the exceptional situation, and specify what makes it unusual.
2. If necessary, declare the exception class in the signature of a routine in order to warn people to be careful. This is only required for two exception classes: `CX_STATIC_CHECK` and `CX_DYNAMIC_CHECK`.
3. Raise the exception.
4. If necessary, clean up after yourself after you've raised the exception. You only need `CLEANUP` if you're passing the exception object outside of the current routine.
5. Add error handling code.

7.2.1 Creating the Exception

Exception classes are created via SE24 in the same way as normal classes, except that they start with `ZCX` or `YCX` (for customers of SAP) or `CX` for standard SAP classes (Figure 7.2).

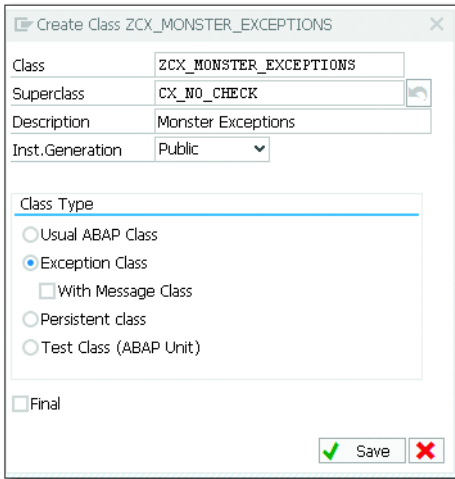


Figure 7.2 Creating an Exception Class

As you can see, the screen asks for some basic information about the class: most notably, what type is being created (refer back to Section 7.1 for an explanation of the different types). It's important to set the `INST.GENERATION` option to `PUBLIC`, because the other options don't make sense. As you can see, it's possible to attach

an existing message class to your exception class, but don't do that. Let's not cling on to the past; let's rush headlong into the future.

The last thing you have to think about when creating an exception class is as follows: At the time this error occurs, is there any specific information that would be really useful to anyone hunting down the cause of the error? Think about all the times you have tried to analyze a short dump in the live system and have thought, "I wish I knew what the value of this variable was, or what input parameters the user had entered" (or whatever). ST22 gives you that information—sort of—but you need to go hunting.

In this example, every time a serious unexpected error happens in one of your monster programs, you really need to know what the number of wibbly wobbly woos were at the time the error occurred. It's probably the same for you in other situations; you spend half your working day trying to guess what this value might have been when an error occurred. Think about it: How can you ever get to the root cause of a problem without knowing such a vital piece of information? Therefore, you add this data as an attribute of your exception class by calling up Transaction SE24, specifying your exception class, and then choosing the ATTRIBUTES tab (Figure 7.3).

Attribute	Level	Visi...	Re...	Typing	Associated Type	Description	Initial value
<IF_T100_MESSAGE>			<input type="checkbox"/>				
DEFAULT_TEXTID	Constant	Public	<input type="checkbox"/>				
T100KEY	Instance A.	Public	<input type="checkbox"/>	Type	SCX_T100KEY	T100 Key with Parameters Mapped to Attribute Names	
<CX_ROOT>			<input type="checkbox"/>				
CX_ROOT	Constant	Public	<input type="checkbox"/>	Type	S0TR_CONC	Exception ID: Value for Attribute TEXTID	*16AA9A39...
TEXTID	Instance A.	Public	<input checked="" type="checkbox"/>	Type	S0TR_CONC	Key for Access to Message Text	
PREVIOUS	Instance A.	Public	<input checked="" type="checkbox"/>	Type Ref.	CX_ROOT	Exception Mapped to the Current Exception	
KERNEL_ERRID	Instance A.	Public	<input checked="" type="checkbox"/>	Type	S380ERRID	Internal Name of Exception, if Triggered from Kernel	
IS_RESUMABLE	Instance A.	Public	<input checked="" type="checkbox"/>	Type	ABAF_B00L	Flag, Whether RESUME Can Be Used in the Exception Handler	
ZCX_MONSTER_EXCEPTIONS	Constant	Public	<input type="checkbox"/>				
HEAD_HAT_DISPARITY	Constant	Public	<input type="checkbox"/>				
NO_HEAD_HOWLING_PROBLEM	Constant	Public	<input type="checkbox"/>				
WIBBLY_WOBBLY_WOOS	Instance A.	Public	<input checked="" type="checkbox"/>	Type	I	Number of Wibbly Wobbly Woos at time error occurred	

Figure 7.3 Exception Class with Context-Specific Attributes

Then it is just a question of adding the attribute to store the error specific information at the end of the list. The exception class constructor will automatically

create an (optional) importing parameter with the same name as the attribute so that every time the exception is raised the programmer can (optionally) pass on that information to the instance of the exception class that is generated.

Later on, when you start processing instances of your exception class, you'll be very grateful that you have a mechanism to store such data and pass it onwards into routines to analyze and respond to the error.

7.2.2 Declaring the Exception

For two of the three types of exception class (`CX_STATIC_CHECK` and `CX_DYNAMIC_CHECK`), you have to specify the exception class in the signature of the method that raises the exception. If you do have to specify the exception, then you do so in Transaction SE24 by navigating to a method name in your class and clicking the EXCEPTIONS button. The screen shown in Figure 7.4 appears.

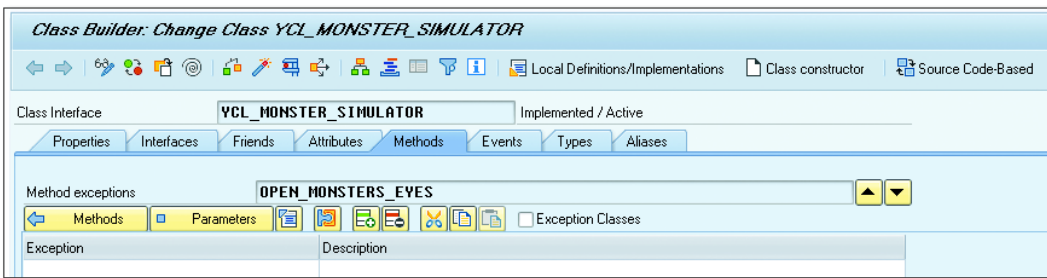


Figure 7.4 Adding Exceptions to Global Class Methods

You just have to add one or more exception classes to the table in Figure 7.4; the descriptions will fill themselves out.

As you can see in Figure 7.4, when you try to add a class-based exception to a global method, amazingly, the `EXCEPTION CLASSES` checkbox is not selected by default. Without that box selected, global methods can have exceptions that will behave just like a function module, giving a list of exceptions that set the value of `SY-SUBRC`. Luckily, the system is clever enough to recognize that a class starting with `ZCX` is an exception class and will therefore select the `EXCEPTION CLASSES` box for you.

If you're going to declare the exception class in the signature of the method, then the name of the exception class must make sense to the caller (i.e., it cannot be

related to any internal implementation details of the method that raises the exception). The caller doesn't need to know how the called routine works, just that it failed in some way; therefore, a name like `CX_NO_RECORD_FOUND` is better than `CX_READ_ON_TABLE_XYZ_FAILED`.

7.2.3 Raising the Exception

In traditional function modules, exceptions were raised via the `RAISE EXCEPTION` construct. Now, there is the obviously better `RAISE EXCEPTION TYPE` construct. (It's obviously better because it has one more word and is close enough to the old syntax to thoroughly confuse everyone.)

In order to raise class-based exceptions using the new `RAISE EXCEPTION TYPE` construct, you need to have a `TRY/CATCH/CLEANUP` block, which some traditional ABAP programmers don't like the look of, because it seems much more complicated than just saying `IF SY-SUBRC <> 0`. An example of catching a class-based exception is shown in Listing 7.3; as you can see, there are no commands that do anything; it just describes the structure of the error handling block. Each section has a clear task to do, and in each section you call a method to do that section's task (i.e., the proper purpose of the method *or* error handling *or* cleaning up). This approach obeys the wise words of one of the fathers of OO programming, Robert Martin, who once said, "Each method should do one thing only, and error handling is one thing."

Thus, there is a clear separation of concerns. In Listing 7.3, the `DO_SOMETHING` method does the real work of the application and cares not about how errors are handled. The `HANDLE_THIS_EXCEPTION` method, on the other hand, is solely concerned about handling an error and cares not about the real business of the application. You should always aim for this approach when writing `TRY/CATCH/CLEANUP` blocks.

```
METHOD main.
* Local Variables
DATA: lcx_monster_exception
      TYPE REF TO ycx_monster_exception,
      lcx_monster_exception_mc
      TYPE REF TO ycx_monster_exception_mc.

TRY.
  do_something( ).
```

```

    CATCH ycx_monster_exception INTO lcx_monster_exception.
        handle_this_exception( lcx_monster_exception ).
    CATCH ycx_monster_exception_mc
    INTO lcx_monster_exception_mc.
        handle_that_exception( lcx_monster_exception_mc ).
    CLEANUP.
        cleanup_main_method( ).
    ENDTRY.

```

Listing 7.3 TRY/CATCH/CLEANUP

In Listing 7.3, you're passing the exception object into the `HANDLE_THIS_EXCEPTION` method. Because this is just a standard instance of an ABAP class, you can query all the class attributes in your handler method (e.g., get the error text and any supplementary data). Meanwhile, to raise the exception in the first place, you would write code similar to that in Listing 7.4 inside the `DO_SOMETHING` method.

```

    RAISE EXCEPTION TYPE zcx_monster_exceptions
    EXPORTING
        wibbly_wobbly_woos = ld_www_count.

```

Listing 7.4 Exporting Vital Information while Raising an Exception

7.2.4 Cleaning Up after the Exception Is Raised

The `CLEANUP` command seems to be one of the most widely misunderstood parts of the whole class-based exception framework. Many programmers try it once, conclude it doesn't work, and then never look at it again. The idea behind the `CLEANUP` construct is that at the start of any given routine, the data is in a consistent state. The routine merrily starts changing some data values, making the data temporarily inconsistent, and if all goes well the routine will fix up the data again before the routine finishes. However, if an error occurs halfway through the routine, then the second half (which fixes up the data) never happens.

Consider the code in Listing 7.5, which is all about conducting an operation to replace a monster's head with a new one. If something goes wrong during the operation (e.g., a power failure), then what you want is to abort the operation and reattach the old head.

```

METHOD head_swap_operation.

    TRY .
        mo_monster->remove_current_head( ).

```

* Code to do assorted operation type activities

```

    mo_monster->add_new_head( ).

CATCH zcx_power_failure.
    mo_candle->light( ).
CLEANUP.
    mo_monster->reattach_old_head( ).
ENDTRY.

```

ENDMETHOD. *"Head Swap Operation*

Listing 7.5 Head Swap Operations

Looking at the code, you would expect that the `CLEANUP` code would reattach the old head before moving on to the `CATCH` code and lighting the candle. However, that's not what happens. If the `ZCX_POWER_FAILURE` exception is raised in between removing the current head and attaching a new head, then the `CLEANUP` code is ignored. This means that the candle is lit, but the monster is left without a head. Not good!

This is because the `CLEANUP` block is only called if the exception is propagated outside of the current routine (here, `HEAD_SWAP_OPERATION`) and handled in another routine. In this example, the `ZCX_POWER_FAILURE` exception is handled inside the routine in which the exception is raised, and so the `CLEANUP` block is not called. One way to get around this problem is to add some code to the end of the routine (outside the `TRY/CATCH` block) to test for data being in an inconsistent state (e.g., monster with no head) and correct the situation. This is shown in Listing 7.6.

```

CATCH zcx_power_failure.
    mo_candle->light( ).
CLEANUP.
    "In case an exception is raised which is
    "not caught inside this routine
    mo_monster->reattach_old_head( ).
ENDTRY.

"In case an exception is caught within this
"routine before the new head is attached
IF mo_monster->number_of_heads = 0.
    mo_monster->reattach_old_head( ).
ENDIF.

```

ENDMETHOD. *"Head Swap Operation*

Listing 7.6 Making 100% Sure a Method Cleans Up After an Exception

Another approach in cases like this—in which we 100% want the `CLEANUP` code to be executed in case of a problem—is to make sure the exception is handled outside of the routine in which the exception is raised.

In Listing 7.7, if the castle is stormed by the villagers during the middle of an operation, then the `CLEANUP` routine in the head swap routine will reattach the old head before moving on to the `CATCH` block in the `REPLACE_EVERYTHING` routine, when the monster will attack the villagers—and for that activity, it will of course need its head back on.

```
METHOD replace_everything.

    TRY.
        mo_monster->head_swap_operation( ).
        mo_monster->leg_swap_operation( ).
        mo_monster->arm_swap_operation( ).

    CATCH zcx_castle_stormed_by_villagers.
        mo_monster->attack_villagers( ).
    ENDTRY.

ENDMETHOD. "Replace everything"
```

Listing 7.7 Catching an Exception in a Different Routine

(It could be argued that a slightly more realistic example is a routine that calls an `ENQUEUE` module to lock a sales order right at the routine start, and if an error occurs, then you want to release the lock. But where's the fun in that?)

7.2.5 Error Handling with `RETRY` and `RESUME`

Exception classes have a feature that sets them apart from old-fashioned error handling methods like checking `SY-SUBRC`. When handling an instance of a class-based exception, you can decide to have the program attempt to fix the underlying cause of the problem and then have another go.

Therefore, handling errors with exception classes can best be described as if at first you don't succeed, try, try again. In real life, an example might be that the Baron tries to terrify a group of villagers by sending out a really foul-smelling monster. It turns out that all the villagers have colds, so they cannot smell the monster and aren't too bothered. He tries again by sending out a really ugly monster that screams a lot. There's nothing wrong with the villagers' eyes and ears, so this time they're really scared: job done.

Technically, how you achieve this is by using the keywords `RETRY` and `RESUME`. `RETRY` is used when the routine in which the exception was raised might be able to fix the problem. `RESUME` is for when a routine higher up the call stack might be able to fix the problem.

RETRY

It's possible that, though the error is unexpected, the program knows how to fix it. If that's the case, then in the `CATCH` block you'll try to repair the error to the best of your ability and then `RETRY`.

In Listing 7.8, the `RETRY` restarts the program flow at the start of the `TRY` block, with the hope that the user will fix the problem. The user gets one chance. The error message will let the user navigate to the Change Monster transaction, where he can correct the number of heads that the monster has. If the user reacts to the information in the error message and manages to fix the problem, then all is well and good.

If the user does nothing or is unable to fix the monster master record, then, after the `RETRY` statement, processing starts again at the start of the `TRY` block. Because nothing has changed, the exception is raised again in exactly the same place as before, but this time a hard error message occurs—because you've set the flag `LF_REPAIR_ATTEMPTED` to `TRUE` during the first attempt at error handling.

```

    TRY.
... some code ...
        IF ld_heads = 0.
            RAISE EXCEPTION TYPE ycx_monster_exception
            EXPORTING
                textid = ycx_monster_exception=>head_error
                heads  = ld_heads.
        ENDIF.
... other code ...
    CATCH ycx_monster_exception.
        IF lf_repair_attempted = abap_false.
            SET PARAMETER ID 'VKORG'    FIELD ld_vkorg.
            SET PARAMETER ID 'ZMONSTER' FIELD ld_monster_number.
            "001 = Monster has &1 heads, which is most unexpected
            MESSAGE i001(yms_monster_errors) WITH ld_heads.
            lf_repair_attempted = abap_true.
            RETRY.
        ELSE.
            MESSAGE e001(yms_monster_errors) WITH ld_heads.

```

```

        ENDIF .
    ENDTRY .

```

Listing 7.8 RETRY after Handling an Exception

Note that every statement between the start of the `TRY` block and the place where the exception occurred gets processed again, so you need to make sure you restore the program to the state it was in when the `TRY` block that raised the exception was entered. `CLEANUP` clauses should help here if the exception was caught further up the stack; if caught locally, then you most likely need to reset local variables and the like.

A possible use case for the `RETRY` statement during error handling is if you were really sure that the data was in an internal table, but it turned out not to be—in which case you could read the database instead to add the entry to the internal table and then `RETRY`. Another possible use case would be if the program tried to access an external system to get the temperature, and the interface was down, in which case you could ask the user for the temperature instead.

If you don't want to start again from the start of the `TRY` block but want to carry on from where the exception was raised, then you also have the `RESUME` option.

RESUME

With the `RETRY` command, after the error handling in the `CATCH` block, processing is restarted at the beginning of the `TRY` block, where the exception was raised. With the `RESUME` command, after the error handling in the `CATCH` block, processing is restarted at the line after the line that raised the exception. If, for example, the routine being called does not know how to fix the problem but the calling program does, then the exception is raised as `RESUMABLE` so that the calling program can fix up the data and then come back to the point things left off in the routine being called.

In Listing 7.9, the exception is raised in the low-level routine `DO_NOT_KNOW`; the exception is propagated to the `CATCH` block in the high-level routine that knows how to fix the error. After fixing the error, the high-level routine calls the `RESUME` statement, and processing restarts in the low-level `DO_NOT_KNOW` routine at the next command after the exception was raised.

Meanwhile, back at the ranch, the `BEFORE UNWIND` statement makes sure that the local variables of the called routine `DO_NOT_KNOW` are preserved so that they are still available when processing restarts after the error has been fixed. If you try to do a `RESUME` statement without the `BEFORE UNWIND`, then you get a syntax error. (A `CLEANUP` block isn't needed in this situation, because it alters the state of the variables—which is exactly what you're trying to avoid, which is why it doesn't get called.)

```

*&-----*
*& Form RESUMABLE
*&-----*
* Routine that knows what to do with incorrect data
*-----*
FORM resumable .

    gf_data_is_rubbish = abap_true.

    TRY.
        PERFORM do_not_know.

        CATCH BEFORE UNWIND cx_ai_application_fault.
            "Repair the data using special knowledge
            "only this level knows
            IF gf_data_is_rubbish = abap_true.
                gf_data_is_rubbish = abap_false.
            ENDIF.
            RESUME.
        ENDTRY.

    ENDFORM. " RESUMABLE
*&-----*
*& Form DO_NOT_KNOW
*&-----*
* This routine does not know how to repair data
*-----*
FORM do_not_know RAISING resumable(cx_ai_application_fault).

    TRY.
        MESSAGE 'I am executed before the exception' TYPE 'I'.

        IF gf_data_is_rubbish = abap_true.
            RAISE RESUMABLE EXCEPTION TYPE cx_ai_application_fault.
        ENDIF.

        MESSAGE 'I am executed after the exception' TYPE 'I'.

    CATCH ycx_monster_exception.

```

```

CLEANUP.
    "This never gets called due to BEFORE UNWIND above
    MESSAGE 'Cleanup' TYPE 'I'.
ENDTRY.

ENDFORM. " DO_NOT_KNOW

```

Listing 7.9 Raising a Resumable Exception

A use case for the `RESUME` statement is a situation in which you are using design by contract exceptions. What are design by contract exceptions, you ask? Read on!

7.3 Design by Contract

When you're sending out a signal that something has gone wrong, naturally you want to say which part of the program is responsible for the error, because that's where the error handling code should probably go. Exception classes contain the program name and line number where the error was detected—but this doesn't necessarily mean that that routine is really the source of the problem. So, what do you do?

It turns out there's a well-known (in IT terms) article by Bertrand Meyer from a long time ago (in IT terms; namely, 1992) in which he talks about designing programs to make use of contracts to determine what part of the program has "breached" the contract and is thus responsible for an error. This approach is known as *design by contract*. The general idea is that any time one part of a program calls another part there is an implicit contract between the two, and when something goes wrong, you determine which part has breached the contract so that you can modify its behavior and ensure that the problem doesn't occur the next time.

As an example, I'm in a pub at the time of writing this and am about to go up to the bar to buy a pint of beer from the Irish barman. The barman and I do not have a contract as such but I *assume* he is going to give me a pint of beer as opposed to a pint of poison, and he *assumes* I am going to pay him with real money, not counterfeit money I forged in my garden shed before coming to the pub.

If all goes well, then both parties benefit: I get some beer and am not poisoned, and he gets paid with real money. If he gives me poison instead of beer, then he's at fault, the police arrest him, and hopefully he will stop poisoning people in the

future. If I give him fake money, then I'm at fault, the police arrest me, and hopefully in the future I'll stop printing bogus money. To put this into technical terms: When the intended design of an interaction between routines in a program is violated, the exact routine where the problem occurred is identified and stopped from happening again.

Here are some examples of program assumptions that, if violated, would cause an exceptional situation that would cause the program to fail if there was no exception handling to deal with the violation:

- ▶ The pension age in Australia has been 65 for over a hundred years, so it will always be 65, right? Certainly, there is no need for programs that deal with pensions to be able to handle an age of anything other than 65. Oh dear! The current government is raising it to 70! Who would have thought?
- ▶ The assumption that a certain database read will always succeed. Surely there is no need to test it with a `SY-SUBRC = 0`! That's just a waste of time. Without exception handling, if the result is 4 for `fail`, then the program will continue on its merry way, unaware that the data it's now using is most likely incorrect.
- ▶ When testing to see if an object is locked by calling a standard `ENQUEUE` (locking) function module and specifying in the code that the record is locked `IF SY-SUBRC <> 0`, assuming that the only possible error could be 1 for `FOREIGN_LOCK`. If this were the case, the system variable `SY-MSGV1` would be filled with the user name of the person locking the object and would send him a message. Unfortunately for the person making this assumption, there are several possible exceptions generated from an `ENQUEUE` module, all with different `SY-SUBRC` values, and one of the possible return codes is 2 for `SYSTEM_FAILURE`. I learned this the hard way: In my case, the production system kept failing, and the function module came back with `SYSTEM_FAILURE` and `SY-MSGV1` being blank. Meanwhile, because `SY-SUBRC` was not zero, my code presumed there was a lock and tried to send a message to the user in `SY-MSGV1`. It turns out that when you send a message to a blank user name using the `TH_POPUP` function, every single user in the system gets that message. As might be imagined, this did not make me popular. My lunch breaks in Germany were halved because I made a failed assumption.

The fact is, we all make assumptions all the time in our programs, similar to the examples just mentioned. Therefore, if eating is important to you, pay close attention. In the rest of this section, you'll look at how the design by contract

approach can be used to improve the error handling in your ABAP programs. Specifically, you'll look at the two main strands of design by contract—preconditions/postconditions and class invariants—and how they can be implemented using exception classes.

ASSERT Statements

It's also possible to implement design by contract using `ASSERT` statements. However, exception classes are much more elegant and the recommended approach, so they are not covered here.

7.3.1 Preconditions and Postconditions

The idea of preconditions and postconditions is that in addition to saying what input variables are mandatory for input and what ones are output in the signature of a method, you also say what conditions at the start of a method (preconditions) would cause an error due to a software bug in the caller and what conditions after the method has finished (postconditions) would cause an error due to a software bug in the method itself.

To understand this more clearly, Listing 7.10 shows an example of how the design by contract theory was implemented in the programming language in which design by contract was first conceived, namely, Eiffel.

```
SET_HOUR (a_hour: INTEGER)
  "Set `hour` to `a_hour`"
  REQUIRE
    valid_argument: a_hour >= 0 and a_hour <= 23
  DO
    hour := a_hour
  ENSURE
    hour_set: hour = a_hour
  END
```

Listing 7.10 Design by Contract in Eiffel

This code is designed such that if a silly value for the hour (like 99) is passed in, then the caller is to blame. However, if the export parameter `HOUR` at the end of the routine has the wrong value, then this indicates a fault of the routine itself. If either condition ever fails, then this indicates a bug in the program that should be corrected.

To demonstrate the design by contract approach in OO ABAP, imagine two exception classes, `ZCX_VIOLATED_PRECONDITION` and `ZCX_VIOLATED_POSTCONDITION`. Both of these classes inherit from `CX_NO_CHECK` on the grounds that the errors described in the previous paragraph are ones that should never happen. Now imagine a `ZCL_DBC` class (DBC for “design by contract”) to implement methods with the same names as the Eiffel commands in the previous example in Listing 7.10; i.e., `REQUIRE` and `ENSURE`. The result in ABAP looks like the code in Listing 7.11.

```
METHOD open_monsters_eyes. "Implementation
*-----*
* CHANGING co_monster TYPE REF TO ycl_monster.
*-----*
* Preconditions
  zcl_dbc=>require( that = 'The Monster has at least one eye'
                  which_is_true_if = boolc(
                    co_monster->md_no_of_eyes GE 1 ) ).

* Some code to open the monsters eyes....

* Postconditions
  zcl_dbc=>ensure( that = 'The Monsters eyes are open'
                 which_is_true_if = boolc(
                   co_monster->mf_eyes_open = abap_true ) ).

ENDMETHOD."Open Monsters Eyes - Implementation
```

Listing 7.11 Design by Contract in ABAP

This routine requires that the caller must supply a monster with at least one eye; otherwise, the caller has broken the contract. In return, this routine ensures that the eyes of the monster will be opened; otherwise, the routine has broken the contract.

In addition, this code forces a comment each time the programmer codes a precondition or postcondition. These comments will then be put into an error message, so it's important that the semantics of the phrase are correct. As an example, Listing 7.12 shows some code from the `ENSURE` method, which builds up an error message (which is intended for a programmer rather than an end user).

```
lo_log->append_error_log( :
  id_text = |{ 'This Diagnosis is intended for IT'(011) }| ),
  id_text = |{ 'Routine(001)' } { ld_server_routine }
```

```

    { 'of program'(002) } { 'has a contract'(003) } | ),
    id_text = |{ 'with calling routine'(004) }
{ ld_client_routine } { 'of program'(002) }
{ ld_client_program } | ),
    id_text = |{ 'Routine'(001) } { ld_server_routine }
{ 'agrees to carry out a certain task'(005) } | ),
    id_text = |{ 'The task is to ensure that'(012) }
{ id_that } | ),
    id_text = |{ 'That task has not been fulfilled by routine'(013) }
{ ld_server_routine } | ),
    id_text = |{ 'Therefore there is an error (bug) in routine'(008) }
{ ld_server_routine }
{ 'that needs to be corrected'(009) } | ).

```

```

RAISE EXCEPTION TYPE zcx_violated_postcondition
EXPORTING
    md_condition = that
    mo_error_log = lo_log.

```

ENDMETHOD. "Ensure

Listing 7.12 Building an Error Message

The `require` method raises an instance of exception class `ZCX_VIOLATED_PRECONDITION`, and the `ensure` method raises an instance of exception class `ZCX_VIOLATED_POSTCONDITION`. Both of those exception objects are merely containers for the error log that was built up and the text description of the logical condition that failed.

7.3.2 Class Invariants

In the original design by contract article by Bertrand Meyer, he also presents the concept of the class invariant. A class invariant describes one or more logical conditions that always remain true at the end of every method called on an instance of a class.

The article gives the following analogy: Although a chef's employment contract may not specifically mention how he or she shouldn't burn down the kitchen, you'd still expect that the chef would not do this. To represent this idea in the Eiffel language, you would define a class invariant for the kitchen class that contained a logical condition checking that the kitchen had not burned down. Then, every time a method was called on the kitchen class, the runtime system would

automatically call the class invariant and raise an exception if the kitchen had gone up in flames.

Turning to implementing class invariants using ABAP, you cannot achieve the same functionality that exists in Eiffel (calling the same test automatically at the end of every method) directly in ABAP, but you can come close.

Coding a class invariant involves using the standard SAP interface `IF_CONSTRAINT` (originally discussed in Chapter 3), which lets you create quite complicated check routines. To demonstrate this, turn once again to the monster example. You want to make sure that after every method call to your monster, it still remains a monster, rather than turning into a fluffy pink bunny or some such (let's agree, fluffy pink things are not scary).

The example code is shown in Listing 7.13. First, import the instance of your monster class. The constraint interface takes a data object as input, so you need to cast that into a local variable representing the monster object before you can check the data. Presume failure, and then run a series of tests on the various attributes of the monster to make sure it has not turned pink or gone all fluffy. If all the tests are passed, then the monster is judged not pink and not fluffy, and you return a `TRUE` value to indicate that the method call has not ruined the essential nature of the monster.

The `get_description` method also has to be implemented to describe what has happened in the event of a failure of the various tests (e.g., in this case, the monster is no longer a monster).

```

CLASS lcl_monster_constraint DEFINITION.

    PUBLIC SECTION.
        INTERFACES if_constraint.

ENDCLASS. "Monster Constraint Definition"

CLASS lcl_monster_constraint IMPLEMENTATION.

    METHOD if_constraint~is_valid.
*-----*
* IMPORTING data_object TYPE data
* RETURNING result      TYPE abap_bool
*-----*
* Local Variables
    DATA: lo_monster TYPE REF TO ycl_monster.

```

```

lo_monster ?= data_object.

result = abap_false.

CHECK lo_monster->scariness      CS 'SCARY'.
CHECK lo_monster->bolts_in_neck EQ 2.
CHECK lo_monster->fluffiness     EQ 0.
CHECK lo_monster->colour        NE 'PINK'.

result = abap_true.

ENDMETHOD.                                "IF_CONSTRAINT~is_valid

METHOD if_constraint~get_description.
*-----*
* RETURNING result TYPE string_table
*-----*
* Local Variables
  DATA: ld_message TYPE string.

  ld_message = 'Monster is no longer a monster!'.

  APPEND ld_message TO result.

ENDMETHOD. "IF_CONSTRAINT~get_description

ENDCLASS. "Monster Constraint Implementation

```

Listing 7.13 Pink and Fluffy is not Scary!

You could then add the code in Listing 7.14 to the end of every method in the monster class. In this way, you can be sure that no method call will have an adverse effect on the essential nature of the monster.

```

zcl_dbc=>ensure(
  that = 'The Monster is still a Monster'
  which_is_true_if = lo_monster_constraint->if_constraint~is_valid( mo_
  class_under_test ) ).

```

Listing 7.14 Calling a Class Invariant at the End of Each Method Call**Note**

Instead of adding the code in Listing 7.14 to the end of every method call, you could alternatively add such a call to the end of every unit test you do on the monster class. It would not make much sense to do both.

7.4 Summary

This chapter introduced you to exception classes, explaining the types that are available, how to design them, and design by contract exceptions. After reading this chapter, you should now be familiar with the myriad ways in which you can deal with an error that has occurred, which is in sharp contrast to the traditional approach of totally ignoring the problem and which will ensure that your programs are robust and your users don't scream in frustration all day long.

Recommended Reading

- ▶ Design By Contract:
<http://se.ethz.ch/~meyer/publications/computer/contract.pdf> (Bertrand Meyer)
- ▶ Message Handling—Finding the Needle in the Haystack:
<http://wiki.scn.sap.com/wiki/display/profile/2007/07/09/Message+Handling+-+Finding+the+Needle+in+the+Haystack>

How can it be that mathematics, being after all a product of human thought independent of experience, is so admirably adapted to the objects of reality?

—Albert Einstein

8 Business Object Processing Framework (BOPF)

Reality is something us developers are often accused of having no grasp of. In this chapter's opening quote, Einstein talks about the "objects of reality" and how they can be represented by mathematics. This is what object-oriented (OO) programming is all about. If you write a program to control a real-world object like an elevator, then you create a mathematical representation of that object as an elevator object in your program, with people objects that get in and out of the elevator on the different floors (attributes) of a building object.

SAP inhabits the all-too-real world of business, and as such the objects it has to deal with are representations of concepts like sales orders, purchase orders, invoices, and journal entries in ledgers. The fact that in most of the world the actual thick ledgers in which accounting entries were written no longer exist in a physical sense does not matter; the concept is exactly the same as it was the day the document principle was invented by Italian monk Pacioli in the year 1494.

Right from the start of its existence, SAP has made various attempts at a programming framework for business objects, such as sales orders or purchase orders. The idea is that the business logic stays the same regardless of whatever flavor of the month framework is wrapped around it, and you achieve this by having a *model* class to handle such business logic, with the current framework wrapped around it. This also ensures that business logic remains separate from user interface logic: the so-called model-view-controller (MVC) pattern.

The Role of Model Classes

Throughout this book—and indeed in most software books that talk about OO programming—you keep hearing about the MVC design pattern made famous by the OO

"bible" Design Patterns: Elements of Resuable Object-Oriented Software, written by the "gang of four." According to the MVC design pattern, the model should know nothing at all about the user interface (UI) technology that is going to be used to expose its data, so adding interfaces specific to BSP or Web Dynpro or any other UI technology to the model is a no-no.

The controller, on the other hand cares a great deal about what UI technology is being used, so that's the ideal place to add UI-specific interfaces. UI technologies change so fast, and the business logic in the model changes so slowly by comparison (business logic might change fast as well, but the two things do not change *at the same time*), you want to keep the two isolated so that you can make changes to one without the other.

That's the whole point of the MVC pattern. As soon as you make a cast iron link from the model to a specific UI technology, you've lost all the benefits in a heartbeat.

This chapter takes a look at the latest framework for business objects, the Business Object Processing Framework (BOPF), which is already embedded in the SAP system. Because the concept is so alien to traditional ABAP programmers, this chapter explains BOPF by comparing how to write a program in DYNPRO versus BOPF. You'll create an SAP-based representation of a real-world object (a monster, obviously) via BOPF.

The chapter starts with the basics, by addressing the transaction in which you define an object using BOPF (Section 8.1). Next, you'll move on to the core of the chapter, in which you'll learn about the various parts of BOPF by comparing them to writing the "old-fashioned" DYNPRO programs beloved of us old SAP programmer types (Section 8.2). (OO purists will have me burned at the stake as a witch for making such a comparison, but I don't care.) Finally, this chapter ends by looking at enhancing BOPF when standard SAP does not deliver exactly what you need out of the box (Section 8.3). (I wonder how many boxes in real life it takes a team of 70 consultants a year to open?)

8.1 Defining a Business Object

In the past, when defining a custom business object, the first thing you would do is create one or more database tables—for example, your custom equivalents of VBAK and VBAP. It's no different here; you start off exactly the same way. In this case, you're going to have a monster header table for attributes common to a monster and a monster parts table that comprises the various components the monster is made of (rather like a bill of materials [BOM]). Instead of SE11, the transaction for this is Transaction BOB, as in "BOB the Builder."

This section will walk you through the three steps of defining the business object: initial creation, header node creation, and item node creation.

8.1.1 Creating the Object

Figure 8.1 shows the initial BOB screen, where there are three categories: business objects, which are the standard SAP-created ones, enhancement objects, which are those in which you add your own extras to a standard SAP object (more on this in Section 8.3), and your custom business objects. Click the CUSTOM BUSINESS OBJECT icon.

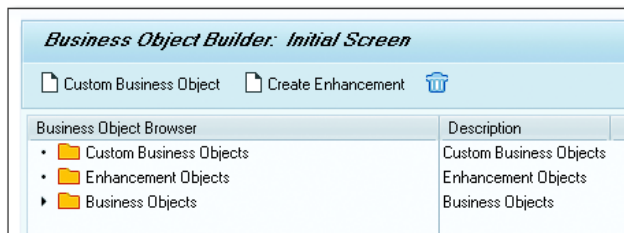


Figure 8.1 BOB Initial Screen

The next two screens are self-explanatory; the first is a “hello” screen, and the second asks you for your customer namespace (Z, unless you're a software company) and the name of your object. The third screen is where things start to get interesting (Figure 8.2).

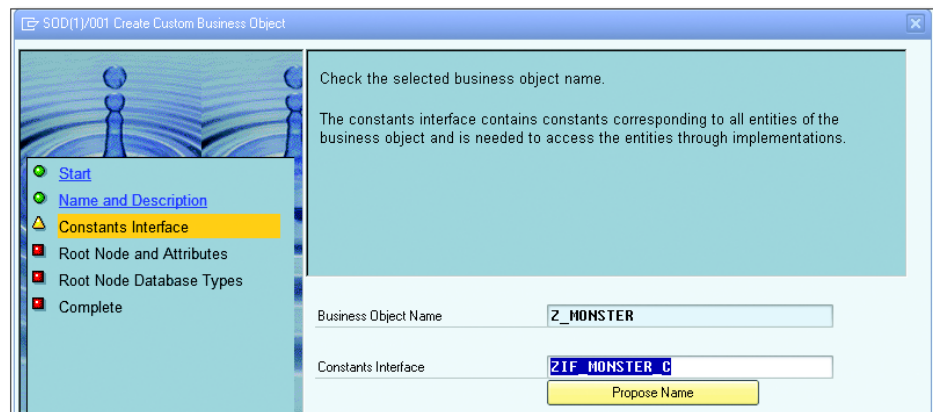


Figure 8.2 Constants Interface

At this stage, you may be asking yourself what in the world a constants interface is. This interface is needed when you start making assorted customizing settings, because there will be quite a few 32-character hexadecimal GUIDs (see box ahead for more info) generated for accessing various parts of your monster object. This is very common in modern SAP frameworks. To hide these GUIDs from programmer eyes, the system will generate constants with English names that you can use instead. These constants will live inside an ABAP interface—in this example, `ZIF_MONSTER_C`. Thereafter, your custom programs can refer to `ZIF_MONSTER_C=>MONSTER` as opposed to `A00765432100EF00`.

Click the **NEXT** button, and you're now ready to define the top-level (header) node.

What is a GUID?

GUID stands for "Globally Unique Identifier," and there is a very good (and very funny) explanation of GUIDs to be found on the web at <http://betterexplained.com/articles/the-quick-guide-to-guids>.

This is a universal idea, but the application within SAP is that instead of having an identification number (or name) for an object such as a customer or ABAP artifact, you have a really long hexadecimal string. This string is then used as the primary key in a database table.

The benefit is that it really is a unique number, and database access using such a number is really fast. The downside is that such numbers are not understandable by humans, and you need a mechanism to present such keys to human beings via some sort of alias (i.e., FRED BLOGGS instead of 30dd879c-ee2f-11db-8314-0800200c9a66).

It's worth noting that virtually all new SAP technology uses GUIDs (e.g., BRFplus, as well as the BOPF framework presented in this chapter).

8.1.2 Creating a Header Node

The next screen that appears is shown in Figure 8.3. This is where you define the so-called root node, like the top level of an XML tree or an IDoc; again, this terminology is increasingly common in SAP. You will see the same term used later on in the book when talking about shared memory. In this case, you can think of this as the header, like table VBAK. (Later on, you'll understand the connection to the item table, but for now just concentrate on the header table.)

You will see you have a field for `ROOT NODE NAME`, which is filled by default with `ROOT`. You have to change this to a different name; in this example, enter `"MONSTER_HEADER"`.

Enter a root node name that is unique in the business object.
Enter a short description of the purpose of the root node.

Enter the name of the persistent structure for persistent attributes.
Enter the name of the transient structure for transient attributes.
If the structures do not already exist, create them using forward navigation.

Start
 Name and Description
 Constants Interface
 Root Node and Attributes
 Root Node Database Types
 Complete

Root Node Name:

Root Node Description:

Persistent Structure:

Transient Structure:

Figure 8.3 Header (Root) Node

In the PERSISTENT STRUCTURE field, enter the name of the persistent structure, which is a list of all fields that are going to be stored in the database for the header record. You create this structure by double-clicking the structure name you have entered, which takes you into SE11. For this example, start the list with `MONSTER_NUMBER`. Traditionally, in a database table a human-readable field like this would be your primary key, but in the BOBF world the real primary key will be a 32-character GUID.

The transient structure (also created by double-clicking the chosen structure name and adding your chosen fields) is a list of the fields that are dynamically determined at runtime: text names for organizational units, calculated fields, and the like. In the past, you may have created DYNPRO screens and first added fields from a table like `VBAK` (sales organization, sales office, and the like) and then manually added text fields, which you later manually fill up with your own code so that humans can know what the values in the database fields mean. In the BOBF world, you think about this in advance, before you even go near the UI layer.

In Figure 8.4, you're about to create the database table for the header record. As you can see, you're going to generate a structure for the combined structures (persistent records and transient records), a table type of this same combined structure, and the database table for the persistent records all in one hit. These are

automating steps that are often done one after the other when creating a new application.

For each field, the wizard will propose a name. However, because every company and developer seems to have its own naming conventions, you're free to change the proposals to whatever makes you happy, for example, ZCS for combined structure, ZTT for table type, and T for table. If you get into trouble, then you can click the PROPOSE NAME button under each field to populate the SAP-proposed name.

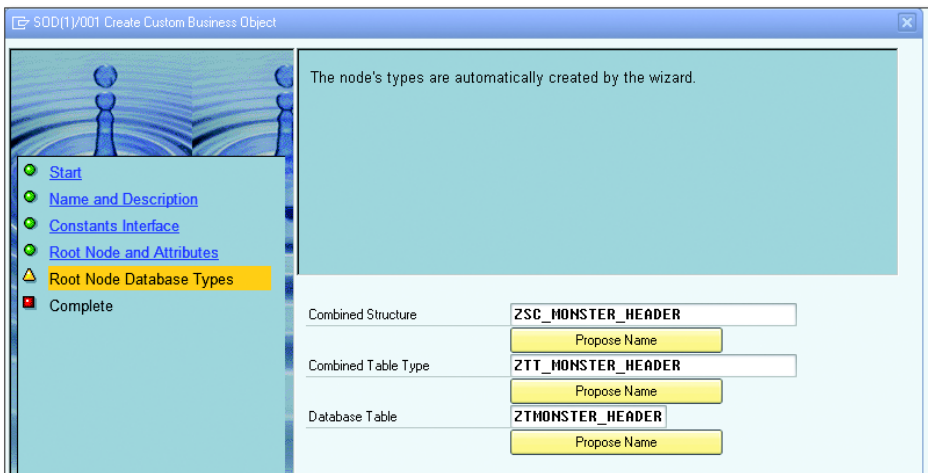


Figure 8.4 Defining the Database Table

In the last window in the wizard, all you see is the COMPLETE button. When you click that button, if you look at the bottom of the screen, you will see a vast array of objects being created.

8.1.3 Creating an Item Node

Once everything is generated, you'll see the screen shown in Figure 8.5; now, you'll add another level to store the item table.

If you've ever defined an IDoc or an XML structure in SAP Process Integration (PI), then you'll be familiar with building up a tree structure in which each node can have zero to many child nodes. BOPF is just like that. There isn't much to this; in the screen shown in Figure 8.5, you just right-click your header-level node and

then choose the CREATE SUBNODE option. The wizard screen pops up, and you repeat the same process as before.

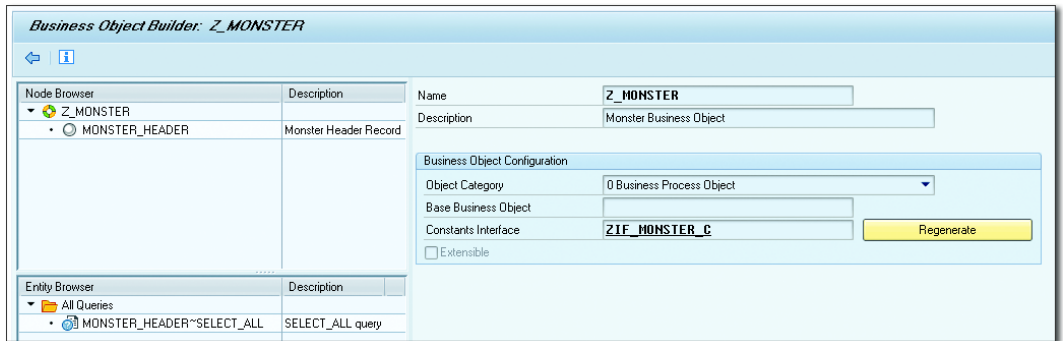


Figure 8.5 Initial Monster Business Object

The first screen just tells you what the parent node is and asks you to give the child node a name and description; nothing exciting here. Next, you're asked for two structures, as before: one to be stored in the database, and a transient structure for things like text descriptions, which are calculated at runtime. You create both structures by double-clicking them and adding assorted fields.

Finally, you're asked for the name of the combined structure, a table type for the combined structure, and a database table to store the items—exactly the same as before. This time, the system doesn't need to generate quite so many objects, and the end result looks like Figure 8.6.

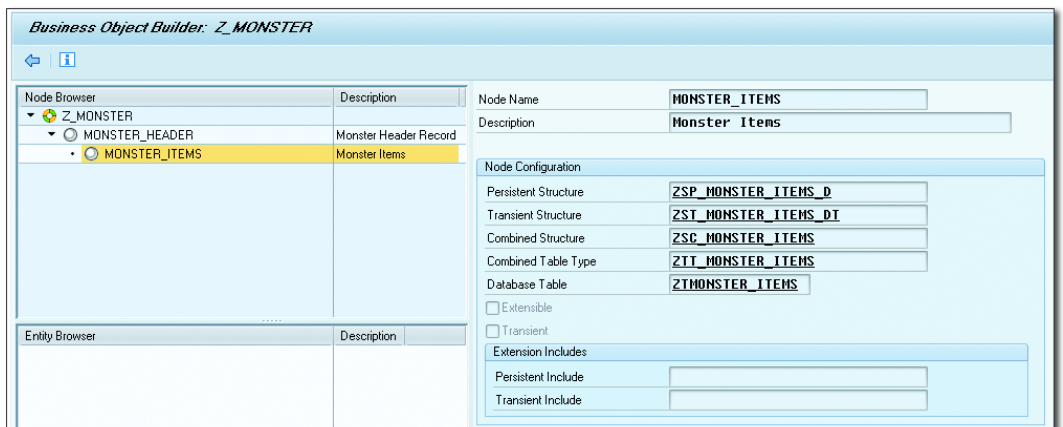


Figure 8.6 Monster Object with Item Node

You could repeat this process as often as you wanted; you can easily imagine creating a tree somewhat like the asset master, with one header table ANLA and 25 other tables all hanging off the main one.

Note

Subnodes are not the only things you can create for a parent node by right-clicking. The full list of options is as follows:

- ▶ Subnodes (associations)
- ▶ Queries
- ▶ Determinations
- ▶ Validations
- ▶ Actions

The next section discusses each option in turn whenever it is useful for the example class you're about to create.

8.2 Using BOPF to Write a DYNPRO-Style Program

Throughout this book you've seen many references to the MVC design pattern, in which you separate the UI coding from the business logic. In this section, you're going to be creating a model class for your monster business object, which can then be used by a controller, which sends the model's data to the UI layer and tells the model how the user has responded.

UI technology has moved on somewhat violently from the traditional DYNPRO UI framework, but the types of data the model sends out and the types of events that come back are still the same. Therefore, in this section you'll build the model as if it was going to be feeding a traditional DYNPRO screen, yet at the same time you'll make the class so generic that it can be used by other interface technologies. In effect, you're having your cake (describing BOPF in DYNPRO terms many programmers are familiar with) and eating it too (ensuring the resulting class is future-proof).

The next sections will mirror the flow of a DYNPRO program. Section 8.2.1 addresses the business logic you would normally write in an `INCLUDE` statement that ends in `F01` (the logic that has nothing to do with the UI). Here, you'll put such logic in a model class. Then, in Section 8.2.2, Section 8.2.3, Section 8.2.4,

and Section 8.2.5, you'll look at elements of BOPF that are analogous to the initial screen on which you enter your monster number and click `DISPLAY` or `CHANGE`. This encompasses tasks such as checking whether the monster record already exists, whether the current user is authorized to view or change it, and whether the record is locked by another user. Next, you'll look at the BOPF equivalents of PBO (process before output) processing (what the program does before the data is presented to the user) in Section 8.2.5 and Section 8.2.6. The tasks in question are filling out certain fields based on the values of others and disabling certain user commands in certain circumstances. Naturally, the section ends with the BOPF equivalents of PAI (process after input) in Section 8.2.7, Section 8.2.8, Section 8.2.9, and Section 8.2.10. Tasks here involve checking user input to make sure the data is consistent, acting upon commands the user has invoked, saving the record to the database, and creating change documents to track what fields the user has changed.

8.2.1 Creating Model Classes

This is a good time to prepare yourself to be puzzled. In the next section, you'll see how to code the existence check for a BOPF object. You probably know that this requires about two lines of code in the way you've always done things (it's a simple SQL query). So when you first see the equivalent BOPF code, which goes on for pages, it's easy to decide that BOPF is overcomplicated, ridiculous, or in the too-hard basket—and switch off and walk away. Try and keep an open mind until you understand how this all works, because only then can you truly make an informed decision as to whether BOPF speeds up or slows down application development.

Note

Although there's a lot of detail presented in the upcoming discussion (so that you can understand what's going on), Section 8.3.2 will talk about wrapping mundane boiler plate tasks in reusable classes with simple interfaces so that you never have to worry about such details again. This is in fact exactly what SAP itself does internally.

Now, have I scared you enough? Yes? Good! In that case, it's time to look at how to start coding a BOPF model class.

You're going to create class `ZCL_MONSTER_MODEL` to hold the routines you'll be using to manipulate your BOPF monster. As you'll see shortly, you need instances

of several different classes before you can even dream of starting to think about dealing with a BOPF object, so you need to have those as attributes that get instantiated in the constructor of the monster model persistence layer.

In a well-designed OO program, there are a lot of small classes, each with a very specific job to do, which is why you need so many helper classes. This is partly why programming this framework is going to seem overcomplicated at first, but it does give you an enormous amount of flexibility. (Also, as mentioned earlier, you're eventually going to be hiding all this complexity away.) These helper classes are shown in Figure 8.7, and Listing 8.1 shows how an instance of each gets created during the construction phase of the monster model class.

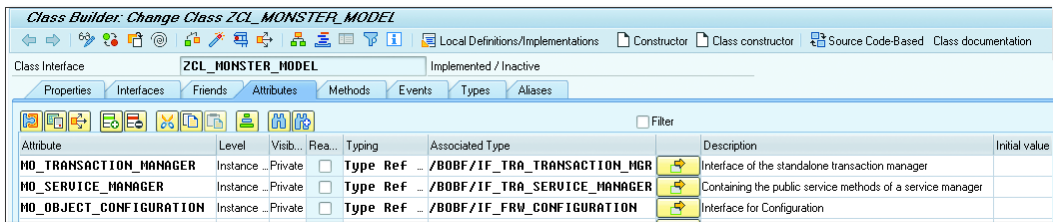


Figure 8.7 Model Class: Helper Classes as Attributes

METHOD constructor.

TRY.

```
mo_transaction_manager =
  /bobf/cl_tra_trans_mgr_factory=>get_transaction_manager( ).
```

```
mo_service_manager =
  /bobf/cl_tra_serv_mgr_factory=>get_service_manager(
  zif_monster_c=>sc_bo_key ).
```

```
mo_object_configuration =
  /bobf/cl_frw_factory=>get_configuration(
  zif_monster_c=>sc_bo_key ).
```

CATCH /bobf/cx_frw.

```
* Oh dear, we've not gotten off to a good start
ENDTRY.
```

ENDMETHOD.

Listing 8.1 Monster Model Class Constructor

You'll notice the attributes that were declared all referred to interfaces as opposed to concrete classes, so in the constructor you'll use factory methods to get actual instances; the factory method takes care of choosing the appropriate concrete class.

A `transaction_manager` class does what the name suggests, taking care of lots of things you would otherwise manually code yourself, like keeping track of what has changed and saving the record.

The `service_manager` class controls the business object itself and has methods to do useful things, such as checking internal consistency and performing actions. You'll see most of these methods shortly, because you'll need to understand the exact place the logic gets coded.

The `object_configuration` class is needed because the BOPF is such a totally generic framework that you have to keep dynamically reading the structures in your object using runtime identification every time you want to read or write data.

The constants interface was mentioned earlier (Figure 8.2); you'll see that Listing 8.1 references this to get the `sc_bo_key` value for the monster object, which is a 32-character hexadecimal GUID that says that this is indeed a monster object.

BOPF generates classes to process its various components like determinations, validations, and actions. Because these are Z classes with public methods, you could call the validation (or whatever) method directly from the controller program—but that's naughty from an OO point of view, because it relies on the caller having lots of details about the internal structure of the BOPF object. Instead, you would call an intermediate method of the business object's service manager and pass the manager a constant that refers to the name of the validation, determination, or action. This presumes the name of the action (or whatever) will not change, but the actual Z class inside the BOPF framework might, and this way the calling program does not need to be changed.

At first glance, traditional programmers will be horrified by things like service managers, which sit between calling programs and actual business objects, saying that this level of complexity just makes the programmer take more time. However, one of the antifragile themes running through this book keeps coming back to the fact that if a structure takes longer to set up in the initial coding phase (5% of the lifecycle of a program) but makes things far easier in the

enhancing and fixing phase (the remaining 95%), then that's the way forward, and it makes the whole program more stable to boot.

8.2.2 Creating or Changing Objects

If you were building a traditional DYNPRO-based transaction (module pool) to create, change, and display monsters, then you would have an initial screen on which the user entered the monster number and clicked CHANGE or DISPLAY or did not enter a number and clicked CREATE. It is of course possible to have external number ranges, in which case the user might also enter a number before clicking the CREATE button.

For all three options, the first step is to see if the number entered by the user actually corresponds to a monster that is stored in the database. First, see what the generated database table looks like (Figure 8.8). The primary key is not the monster number that you would have used in a traditional database table like VBAK or EKKO but is instead a GUID field.

The screenshot shows the SAP database table ZTMONSTER_HEADER. The table has a short description of 'Monster Header Record'. The fields are listed in a table with columns for Field, Key, Initials, Data element, Data Type, Length, Decimal places, and Short Description.

Field	Key	Initials	Data element	Data Type	Length	Decimal...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		Client
DB_KEY	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	ZBOBF/CONF_KEY	RAW	16		ModelID
.INCLUDE	<input type="checkbox"/>	<input type="checkbox"/>	ZSP_MONSTER_HEADER_D	STRU	0		Monster Header Persistent Structure
MONSTER_NUMBER	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_NUMBER	CHAR	10		Monster Number
NAME	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_NAME	CHAR	30		Monster Name
SANITY	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_SANITY	INT4	10		Monster Sanity %age
COLOR	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_COLOR	CHAR	10		Monster Color
STRENGTH	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_STRENGTH	CHAR	20		Monster Strength
HAT_SIZE	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_HAT_SIZE	INT4	10		Monster Hat Size
AGE	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_AGE_IN_DAYS	INT4	10		Monster Age in Days
NO_OF_HEADS	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_HEADS	INT4	10		Number of Monster Heads
MONSTER_COUNT	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_MONSTER_COUNT	INT4	10		Monster Count

Figure 8.8 Generated Monster Table

The user has naturally entered the monster number as opposed to the GUID key. You could put an index on the monster number and perform an SQL query, but then you're fighting BOPF rather than working with it. Instead, take advantage of

an inbuilt mechanism of BOPF whereby you can perform a query that takes in the monster number and comes back with the correct monster without you having to manually handle horrible things like GUIDs. However, you do still need to create an index on the monster number.

Creating a Custom Query

In the bottom-left corner of the BOB screen is a list of possible queries. When you created your monster object, you got one query for free: an empty `SELECT_ALL` query. This is not any use (except when you're testing and want to see all the monsters at once, rather like calling SE16 without filling in any selection fields), so you'll have to create your own that enables you to get records back by specifying a monster number. On the BOB screen, right-click on the `MONSTER_HEADER` node and choose `CREATE QUERY`, and another wizard pops up (Figure 8.9).

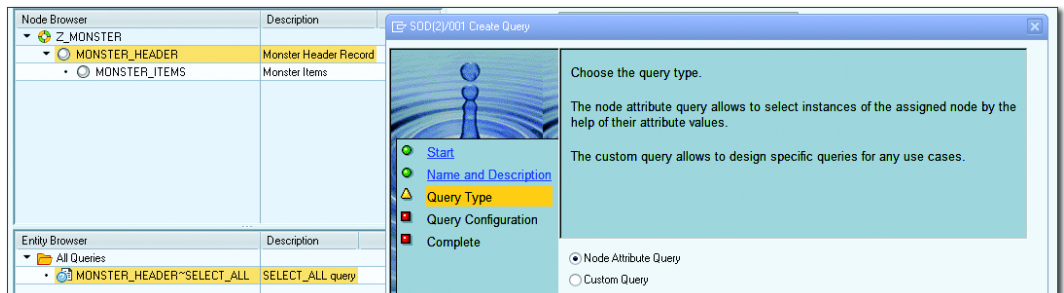


Figure 8.9 Creating a Query

You have two choices: a node attribute query that enables a query by any field in the persistent structure or a custom query in which you can go bananas and define every single aspect yourself. It's good that you can do the latter, but for now stick to the basic node attribute query.

When you select the `NODE ATTRIBUTE QUERY` radio button and click `NEXT`, the query configuration option vanishes, and you're left on the `COMPLETE` screen. Click `FINISH` and you are done. What happens in the background is that the constant interface class has been extended with a new constant called `SC_QUERY_ATTRIBUTE`, which you'll be using in a minute, and that constant is a complicated structure that lists all the fields in your header record.

Using a Custom Query

Next, you'll add a method to your `ZCL_MONSTER_MODEL` that will input a monster number and get the header structure and item table back. A model class should never query the database directly; it should delegate that job to some sort of persistence layer. However, don't panic; that's exactly what you're going to do. Any task that retrieves or updates the database will be outsourced to a specialist class, which can then be mocked for the purposes of unit testing.

To reiterate, when the model class is created it either takes in a persistency layer mock object as a parameter for unit testing or creates one itself that will do all the BOPF-related stuff for real. The model will not know that the BOPF is handling the database retrieval and updates.

There is a huge amount of debate about what to call the method that returns something like a header record and item table. You could call it `QUERY`, `GET`, `LOOKUP`, or any of the wide variety of names used by assorted BAPIs. You could also go with the acronym `CRUD` (despite the unfortunate connotations of the word `CRUD` in some countries of the world), which stands for create, read, update, and delete. Sometimes `CRUD` is expanded with the word *retrieve* instead of *read*; indeed, the word *retrieve* is featured heavily in SAP documentation. Thus, in this example, the method is called `RETRIEVE` (Figure 8.10).

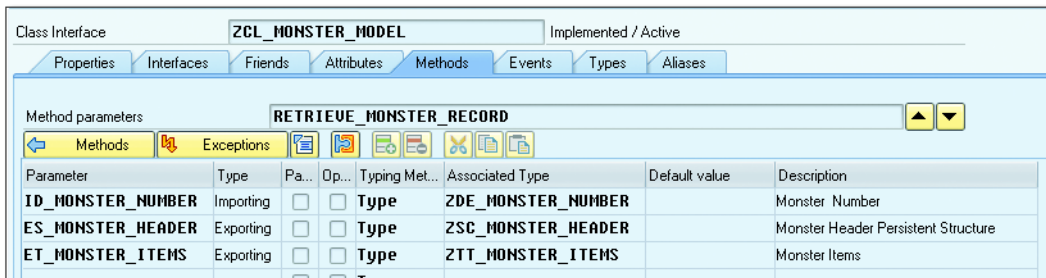


Figure 8.10 Model Class: Retrieval Method Signature

Note

The monster model class passes on this method to an identical method in the `ZCL_MONSTER_MODEL_PERS_LAYER` class, because model classes should never read the database directly. This top-level method should give you an idea of what sort of thing is going on and will call several lower-level methods that deal with the question of how.

As you can see in Listing 8.2, there are three parts to bringing back a monster record from the database. First, you need to transform the monster number into a key (GUID). Then, armed with this information, you can go about getting the header record, and then you can get all items for this header.

```

METHOD retrieve_monster_record.
*-----*
* We could do this the traditional way...
*
* SELECT SINGLE * FROM ztmonster_header
* INTO CORRESPONDING FIELDS OF es_monster_header
* WHERE monster_number = id_monster_number.
*
* But that's just what they're EXPECTING us to do!
* Instead, go down the BOPF path...
*-----*
* Clear Export Parameters
CLEAR: es_monster_header,
      et_monster_items.

* To get a BOPF object, we need a key, not a number
DATA : ld_monster_key TYPE /bobf/conf_key.
      ld_monster_key = get_bopf_key_4_monster_number( id_monster_number ).

* The header record lives in the header(root) node
DATA : lrs_monster_header TYPE REF TO zsc_monster_header.
      lrs_monster_header  ?= mo_bopf_pl_helper->get_node_row(
          id_object_key    = ld_monster_key
          id_node_type     = zif_monster_c=>sc_node-monster_header
          id_node_row_number = 1 ).

FIELD-SYMBOLS: <lrs_monster_header> TYPE zsc_monster_header.

ASSIGN lrs_monster_header->* TO <lrs_monster_header>.

es_monster_header = <lrs_monster_header>.

* The item table records live in one or more child nodes of the
* header-level node
DATA : lrt_monster_items TYPE REF TO ztt_monster_items.

      lrt_monster_items  ?= mo_bopf_pl_helper->get_child_node_table(
          id_object_key    = ld_monster_key
          id_parent_node_type = zif_monster_c=>sc_node-monster_header
          id_child_node_type = zif_monster_c=>sc_association-monster_header-
monster_items ).

FIELD-SYMBOLS: <lrt_monster_items> TYPE ztt_monster_items.

```

```

ASSIGN lrt_monster_items->* TO <lt_monster_items>.

et_monster_items = <lt_monster_items>.

```

```
ENDMETHOD.                                "Retrieve Monster Record
```

Listing 8.2 Model Class Retrieval Method

You will note that due to the fully generic nature of the BOPF classes and methods, you have to devote a lot of effort to converting generic data objects into actual structures and tables you can work with directly.

You will also see that Listing 8.2 includes a call to an instance of a helper class. Section 8.3.2 explains how best to wrap the generic BOPF classes and methods in a more developer-friendly API, but for now anything monster-specific goes in the monster model class (or its persistency-layer friend) and anything fully generic goes into a reusable BOPF helper class; this helper class is named `MO_BOPF_PL_HELPER` in the example code. This all means that how the data is retrieved is several levels down from the monster model class. You now have a nice level of abstraction, because it's not the model's concern how data is obtained; this helps with unit testing.

Next, Listing 8.3 delves into the method that turns the monster number into a key. You're going to be using the query you defined earlier (Figure 8.9), which enables you to do a search on the database for attributes other than the GUID key.

```

METHOD get_bopf_key_4_monster_number.
*-----*
* IMPORTING id_monster_number TYPE ZDE_MONSTER_NUMBER
* RETURN    rd_bopf_key      TYPE /BOBF/CONF_KEY
*-----*
* Local Variables
DATA : lt_parameters    TYPE /bobf/t_frw_query_selparam,
      lt_monster_keys   TYPE /bobf/t_frw_key.

FIELD-SYMBOLS : <ls_parameter>    LIKE LINE OF lt_parameters,
                <ls_monster_keys> LIKE LINE OF lt_monster_keys.

* This builds the dynamic WHERE clause for the database read
APPEND INITIAL LINE TO lt_parameters ASSIGNING <ls_parameter>.
<ls_parameter>-attribute_name =
zif_monster_c=>sc_query_attribute-monster_header- select_monster_by_
attribute_monster_number.
<ls_parameter>-sign    = 'I'.
<ls_parameter>-option = 'EQ'.

```

```

<ls_parameter>-low      = id_monster_number.

* This builds a fully dynamic SQL Statement
CALL METHOD mo_service_manager->query
  EXPORTING
    iv_query_key = zif_monster_c=>sc_query-monster_header-select_
monster_by_attribute
    it_selection_parameters = lt_parameters
  IMPORTING
    et_key          = lt_monster_keys.

* The return is always a table, but we have a unique key, so
* as someone once said THERE CAN BE ONLY ONE!
READ TABLE lt_monster_keys INDEX 1 ASSIGNING <ls_monster_keys>.

CHECK sy-subrc EQ 0.

rd_bopf_key = <ls_monster_keys>-key.

ENDMETHOD. "Get BOPF Key for Monster Number

```

Listing 8.3 Using a Custom Query to Turn a Monster Number into a Key

In Listing 8.3, you will probably recognize the method of creating a generic parameter table (LT_PARAMETERS), because you've been able to do that for some time in order to call methods dynamically.

Put a breakpoint before the call to the standard BOPF query method `MO_SERVICE_MANGER->QUERY` so that you can see exactly what is going on in the guts of the system. You don't have to code this yourself, which is why it isn't in the code shown here—but to get a complete picture you need to know what the standard SAP code does in the background (a fully dynamic SQL statement is getting created). Also do an ST05 trace while testing this method to see the database read. BOPF automatically caches object records in a buffer (something you normally always have to hand code yourself), but it's important to be sure that when the database is accessed nothing bad is happening.

As mentioned when talking about generating the database table to store the monster header record, you need to create an index on `MONSTER_NUMBER` in the database table (via SE11) at the time of the table's creation. This is because, in a query such as this, the dynamic SQL statement does a read based on `MONSTER_NUMBER`; if there were no index on that field, then a full table scan would occur, which is poison from a performance point of view.

You now have the key and can set about getting the header record using this key, so it is time to move on to Listing 8.4. This listing shows the call to the method `GET_NODE_ROW` of the helper class. Here, you have a totally generic method that has no idea what type of business object it's dealing with. Because BOPF nodes can have more than one entry, you always have to start by getting a table. In this case, you know there's only ever one header record, so you read the first row of your own line table.

You'll notice that the method does not need to know the actual structure of the header record; a reference to the data structure is passed back, which is converted back to the real header structure in your monster class, which of course does know all about the correct header structure for a monster.

```
METHOD get_node_row.
* Local Variables
  DATA lt_ref_to_data TYPE REF TO data.

  FIELD-SYMBOLS : <lt_data> TYPE INDEX TABLE,
                 <ls_row>  TYPE any.

  lt_ref_to_data   = get_node_table(
  id_object_key    = id_object_key
  id_node_type     = id_node_type
  id_edit_mode     = id_edit_mode ).

  IF lt_ref_to_data IS NOT BOUND.
    RAISE EXCEPTION TYPE /bobf/cx_dac. "Data Access Exception"
  ENDIF.

  ASSIGN lt_ref_to_data->* TO <lt_data>.
  READ TABLE <lt_data> INDEX id_node_row_number
  ASSIGNING <ls_row>.

  IF sy-subrc EQ 0.
    GET REFERENCE OF <ls_row> INTO rdo_row_data.
  ELSE.
    RAISE EXCEPTION TYPE /bobf/cx_dac. "Error Messages of the data access"
  ENDIF.

ENDMETHOD. "Get Node Row
```

Listing 8.4 Getting a Header Record (Node Row)

Now, take a look at reading the one-line table from the header node. This is shown in Listing 8.5, which contains the three steps needed to get the table containing the header record. First, you need to know the DDIC structure name of the

header record, so use the BOPF configuration class to find this out. You then create a dynamic data table of the correct format. Once you have that, you can complete the third step, which is to fill that (one-line) table, by using the `RETRIEVE` method of the BOPF service manager class to read the data out of the database.

```
METHOD get_node_table.
* Local Variables
  DATA : lt_object_keys          TYPE /bobf/t_frw_key,
          ls_node_configuration TYPE /bobf/s_confro_node,
          lo_message             TYPE REF TO /bobf/if_frw_message.

  FIELD-SYMBOLS : <ls_object_key> LIKE LINE OF lt_object_keys,
                  <lt_data>       TYPE INDEX TABLE.

  "Get all details about the node; what we're interested
  "in is the data table name
  mo_object_configuration->get_node(
    EXPORTING iv_node_key = id_node_type
              IMPORTING es_node      = ls_node_configuration ).

  "Now we know the database table name and can create a dynamic
  "internal table bound to the returning parameter
  CREATE DATA rdo_data_table
  TYPE (ls_node_configuration-data_table_type).
  ASSIGN rdo_data_table->* TO <lt_data>.

  "Retrieve the target node:
  APPEND INITIAL LINE TO lt_object_keys
  ASSIGNING <ls_object_key>.
  <ls_object_key>-key = id_object_key.

  "Off we go! We call a standard BOPF class/method
  mo_service_manager->retrieve(
    EXPORTING iv_node_key = id_node_type
              it_key      = lt_object_keys
    IMPORTING eo_message = lo_message
              et_data     = <lt_data> ).

  "Error Handling
  CHECK lo_message IS BOUND.

  CHECK lo_message->check( ) EQ abap_true.

  RAISE EXCEPTION TYPE /bobf/cx_dac "Error in data access
  EXPORTING mo_message = lo_message.

ENDMETHOD. "Get Node Table
```

Listing 8.5 Getting a Table that Contains the Header Record

Now, move on to getting all the items for one monster, as shown in Listing 8.6. Getting the table of monster items is in one sense simpler; you're getting the whole table rather than a specific row, so you need one less method call. However, it's also slightly more complicated, in that you have to tell BOPF both the parent (header) node and the child (item) node. BOPF thinks of these as 32-character GUIDs, which are referenced by the constants that were generated when you created the monster object.

In Listing 8.6, you're going through the same three steps as before, but with one subtle difference. As the first step, you need the structure for the table, so you ask the BOPF configuration class—but this time you need to specify both the parent and child node types. BOPF calls the parent/child relationship an association, so you'll call a method called `GET_ASSOC`.

Once you know the table structure, once again you can perform the second step of creating a dynamic internal table.

In the third step (`RETRIEVE_BY_ASSOCIATION`), this dynamic table is filled by the BOPF service manager, which this time also needs to know both the parent and child node types.

```
METHOD get_child_node_table.
* Local Variables
DATA: lt_object_key          TYPE /bobf/t_frw_key,
      ls_node_configuration TYPE /bobf/s_confro_node,
      ls_association        TYPE /bobf/s_confro_assoc,
      lo_message            TYPE REF TO /bobf/if_frw_message.

FIELD-SYMBOLS : <ls_key> LIKE LINE OF lt_object_key,
                <lt_data> TYPE INDEX TABLE.

"Find out all about the child node
mo_object_configuration->get_assoc(
  EXPORTING iv_assoc_key = id_child_node_type
            iv_node_key  = id_parent_node_type
  IMPORTING es_assoc    = ls_association ).

IF ls_association-target_node IS NOT BOUND.
  RAISE EXCEPTION TYPE /bobf/cx_dac. "Error of data access
ENDIF.

"The target node contains the name of the database table
ls_node_configuration = ls_association-target_node->*.

"Now we know that we can bind the result parameter
"to a dynamic internal table
```

```

CREATE DATA rdo_data
TYPE (ls_node_configuration-data_table_type).
ASSIGN rdo_data->* TO <lt_data>.

"Have to put the key in a table for some reason
APPEND INITIAL LINE TO lt_object_key ASSIGNING <ls_key>.
<ls_key>-key = id_object_key.

"Off we go!
mo_service_manager->retrieve_by_association(
  EXPORTING iv_node_key      = id_parent_node_type
            it_key          = lt_object_key
            iv_association  = id_child_node_type
            iv_fill_data   = abap_true
  IMPORTING eo_message      = lo_message
            et_data        = <lt_data> ).

"Error Handling
CHECK lo_message IS BOUND.

"The check method makes sure there are actually error messages
"in the message table that came back
CHECK lo_message->check( ) EQ abap_true.

RAISE EXCEPTION TYPE /bobf/cx_dac
  EXPORTING mo_message = lo_message.

ENDMETHOD. "Get Child Node Table

```

Listing 8.6 Getting the Item Table (Child Node Table)

All throughout the example code, you will notice lots of error handling, which just throws the error up through the call stack and out of the model class so that whatever program called the model class can deal with it. Usually, when the read fails it's because the calling program gave rubbish data, so the fault lies with the caller; or perhaps the calling program *wants* the database read to fail, because it may be trying to make sure it does not try to create two monsters with the same number. The latter case is very important in BOPF, because the real primary key is a GUID—so if your actual primary key is the monster number, then you have to enforce uniqueness yourself.

In a DYNPRO-type program, the read on the database is done at the start to get the monster data to display or to change or (in the case of external numbering) to make sure no existing monster has the number you're about to create. In the case of change, you now need to ensure that nobody else is also trying to change the monster you're interested in.

8.2.3 Locking Objects

Usually, when creating a brand-new type of business object that you want your end users to be able to change, you have to create a lock object via SE11 with a name like `EZMONSTER` or `EZONSUNDAYMORNING`. That isn't a very difficult thing to do, but it's an extra manual step. Then, in your program you would call the generated function modules starting with `ENQUEUE` and `DEQUEUE` to make sure that only one person could change a record at any one time.

The BOPF handles all this for you. If the user calls your transaction in change mode, then you want your program to retrieve the existing record and at the same time lock it. In order to make sure this happens, you specify an edit mode during the call to the `RETRIEVE` method, which tells the system what you're trying to do. The possible values are best addressed as constants from the `/BOBF/IF_CONF_C` interface (e.g., `SC_EDIT_READ_ONLY` or `SC_EDIT_EXCLUSIVE`, the latter being the one you want to choose if you want an exclusive lock on the object so that you can change it).

To understand how to do this, consider an example in which you want to change the name of your monster from `HUBERT` to `FRED`. First, you need an exclusive lock, which is shown in Listing 8.7.

```
lo_monster_model->retrieve_monster_record(
  EXPORTING
    id_monster_number = ld_monster_number
    id_edit_mode      = /bobf/if_conf_c=>sc_edit_exclusive
  IMPORTING es_monster_header = ls_monster_header
            et_monster_items  = lt_monster_items ).
```

Listing 8.7 Exclusive Lock

Just to be really sure of what's happening, put a breakpoint just after that piece of code, and then look at Transaction `SM12` to see what's happening in the lock table world (Figure 8.11).

As you can see, the object being locked is a generic lock table used by the BOPF with a key of `Z_MONSTER`, followed by the `GUID`, and so there was no need to manually create a lock object. The lock is released automatically when the BOPF transaction manager either saves the data to the database via `SAVE` or rolls back the changes via the `CLEANUP` method (calling that method `ROLLBACK` would have been too easy; people could have guessed what it did).

The screenshot shows the 'Lock Entry List' window in SAP. It contains a table with the following data:

Cli...	UNAME	Time.....	Lock mode	Table name	Lock Argument	Use Cou...	Use Cou...	
001	HARDYP	16:35:26	E	/BOBF/S_LIB_ENQUEUE_NODE	001Z_MONSTER	005056B900031ED486904E3E9C5FAECO	0	1

Figure 8.11 Lock Table Entry for a Monster

At this point, you might be running around the room screaming in horror, thinking that you've lost control of exactly where in your program you can release the lock. However, if you think about it, you want to lock the record just before the user starts making any changes and unlock the record when either (a) he has finished making changes or (b) he has decided not to make any changes after all. Put like that, the idea of bundling in the `COMMIT/ROLLBACK` with the unlocking of the record does not seem so horrifying after all.

8.2.4 Performing Authority Checks

Another thing you code yourself every time when creating a custom business object is the authority check. You need to know if the user is authorized to display, change, or create a monster. You usually create a new authority check object and then test it against the user's authority at the start of the transaction and, if you're really paranoid, just before the user tries to create or change the record.

Does BOPF let the train take the strain from this manual task? Yes, if you're on SAP NetWeaver 7.4. A little checkbox appears in Transaction BOB when you define a business object: `BUSINESS OBJECT HAS AUTHORITY CHECKS`. You still have to create the authorization objects yourself using SU20 and SU21 (tut, tut; hopefully SAP will automate this in future releases), but then you can link the authorization object to your business object in Transaction BOB. This is the same as the normal authority check transaction; you'll link the primary key of your object to the authority check object.

At that point, BOPF will automatically start making checks at the appropriate point (i.e., is the user authorized to view the monster record, is he allowed to retrieve the data in change mode, etc.). The SAP security person will give user roles activities (change, display, and what have you) against the authority object via the user mechanism.

Just like the locking mechanism, authority checks are a simple thing to create and code for each new business object, but they are a manual step nonetheless, and the idea of BOPF is to remove as many manual steps as possible.

8.2.5 Setting Display Text Using Determinations

It's likely that many times in your programming life you have created some sort of interactive application with a mixture of fields retrieved from one or more database tables directly, and some associated fields where you have either got the value by looking it up from other database tables (e.g., text fields for the name of a sales organization) or calculated the value based on some sort of business logic (e.g., subtracting the time in a database field from the current time). You do this in ALV type reports; you do this in DYNPRO-type reports; some would say that this is the staple diet of an SAP programmer, even if we'd all secretly always like to be doing cutting-edge-type things. In the world of BOPF, naturally you need to do this as well. This time, the process of setting display text or filling other derived fields is achieved via determinations.

You've already seen that when creating a BOPF object you make a distinction between the fields that come out of the database (a persistent structure that provides the values to do the determining) and the fields that you get at runtime (the transient structure in which the fields get determined). This just provides a home for something you've always done but now can do in a consistent way.

You'll now learn how to fill some monster-based text fields based on database values in the monster header table by means of creating a determination. Specifically, you're going to take the monster's hat size, which is stored in the database, and derive a text description, such as "really big hat." You'll also take the monster's sanity and turn it into a description such as "very mad."

In this example, there are two fields in your transient structure that need to be filled based on the values retrieved from the database. The business logic is nice and complicated, as is almost always the case in real life. To set this up, go to your header (root) node (which contains the database structure), and right-click to access the context menu. Choose `CREATE DETERMINATION`.

Figure 8.12 shows the wizard that pops up when you're creating a determination. The screen asks you to name a class that will be used to house the logic to fill in the text fields; note that the class name the system proposes is based on the name you gave to the determination, so don't make that name generic (i.e., in this example, it has to mention monsters).

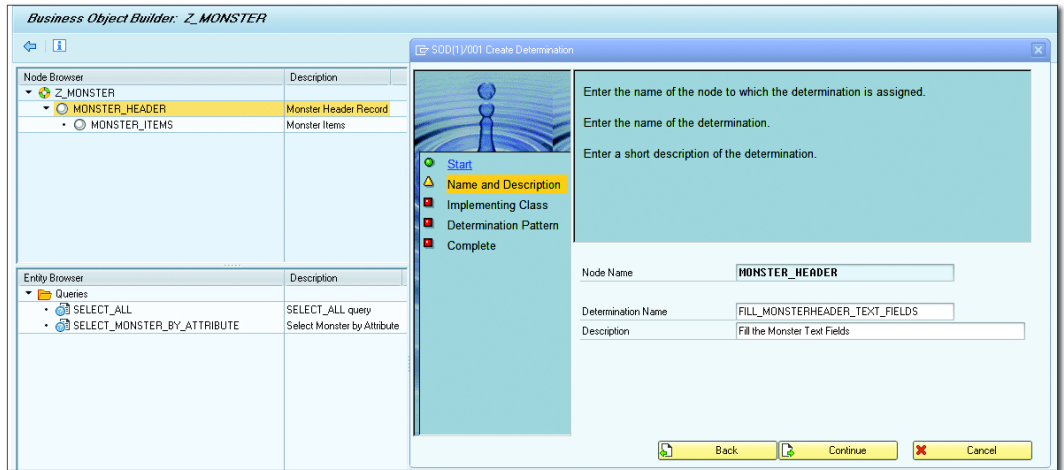


Figure 8.12 Creating a Determination

The name proposed here is `ZCL_D_FILL_MONSTERHEADER_TEXT`, which sounds fine. Click **CONTINUE**, and then the most important decision has to be made (Figure 8.13): what determination pattern to use.

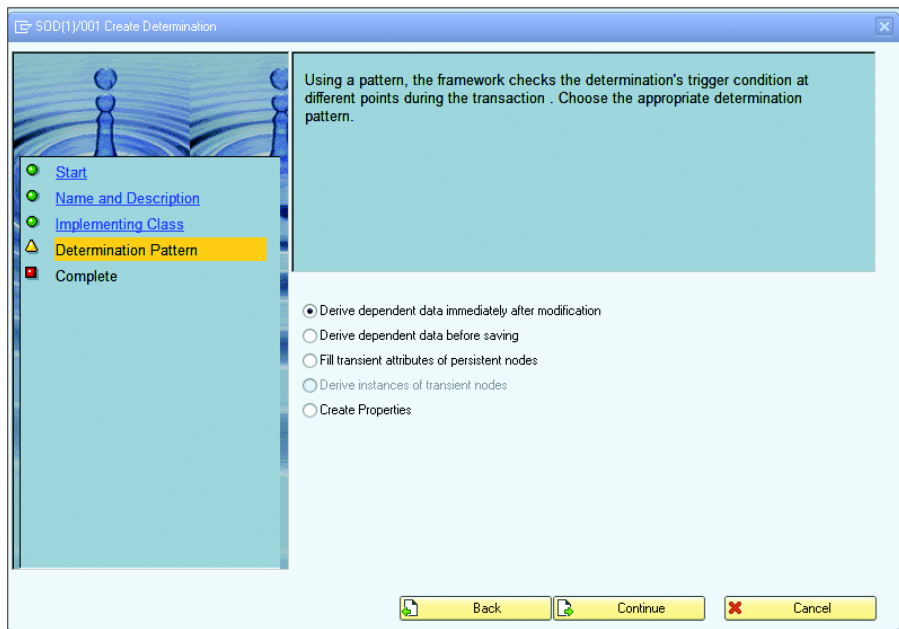


Figure 8.13 Defining when Determinations Run

There are lots of choices here, including the following:

- ▶ **DERIVE DEPENDENT DATA IMMEDIATELY AFTER MODIFICATION**
This is analogous to the equivalent PAI (processing after input) concept you're used to. By selecting this, you're telling the program that after the user changes data you want to update other fields based upon what has been changed.
- ▶ **DERIVE DEPENDENT DATA BEFORE SAVING**
This option refers to a case when you change fields the user doesn't even know about (they're not on the UI, for example) based upon the entered data. By selecting this, you tell the program to change the value of certain fields based on others after the user has chosen the **SAVE** option, and to do so without informing the user that such fields have changed. For example, the technical monster type is not something the user would understand (it only makes sense to a mad scientist), but it's derived from the requirements the user enters.
- ▶ **FILL TRANSIENT ATTRIBUTES OF PERSISTENT NODES**
This is analogous to the PBO (processing before output) concept you know and love. Before showing the user a UI screen, you need to fill in the text fields and calculated fields. If you choose this, you're telling the program that you want to fill out certain derived fields based on other fields such that all the derived fields are full when we show them to the user.
- ▶ **CREATE PROPERTIES**
This option is for when you want to make fields read only or mandatory, based on other data. You often see this in standard SAP; for example, once a delivery has been goods issued a lot of the fields go gray so you can no longer change them. This example does not create such a determination, but you could imagine a scenario where, if the user fills a checkbox saying the monster must eat mountains every day, then the field **HOW MANY MOUNTAINS** becomes mandatory.

The **DERIVE INSTANCE OF TRANSIENT NODES** option is grayed out, so you cannot choose that.

Once you've created your determination, you can see in a nice graphical manner what determinations get run for what nodes (Figure 8.14), and you can even control what order they get run in. You might want to count the number of items first, for example, and then store that number in a transient attribute in the header row.

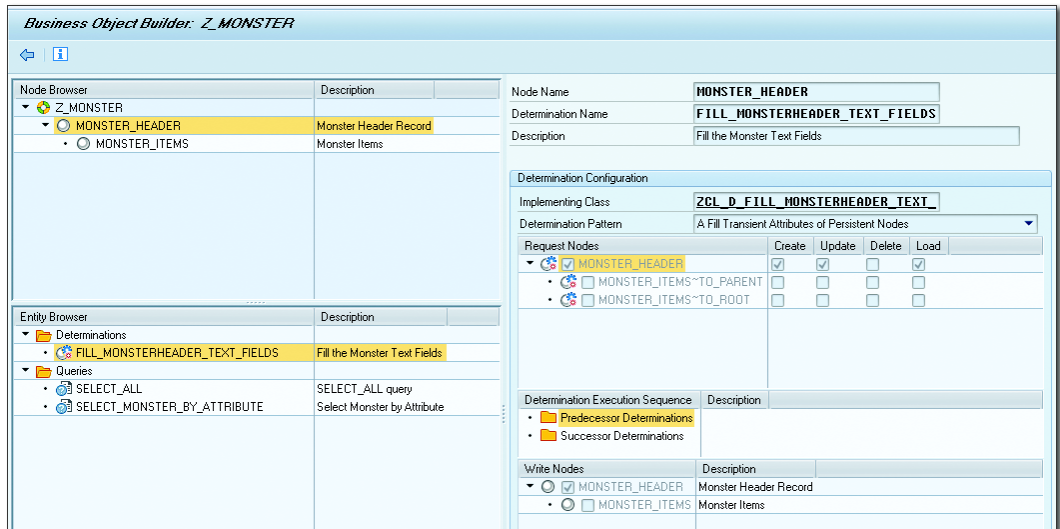


Figure 8.14 Displaying a Determination in Transaction BOB

In this case, you want to do the PBO trick (filling out derived fields before showing them to the user)—you can tell because the DETERMINATION PATTERN field has been set to FILL TRANSIENT ATTRIBUTES OF PERSISTENT NODES—and fill some text fields straight after you have the database fields.

After you finish the wizard, the next task is to actually code `ZCL_D_FILL_MONSTERHEADER_TEXT`. Looking at the generated class, the first thing to notice is that the `/BOBF/IF_FRW_DETERMINATION` interface has been added to the new class. The interface for determinations has the following three methods you can implement (the first two are optional):

▶ `CHECK_DELTA`

This corresponds to the DYNPRO concept of `ON-INPUT`. This method works out what fields have been changed; this is important, because usually only some fields will need to trigger determination if they have been changed. In the DYNPRO framework, this was part of the UI logic, but really this is the job of the model.

▶ `CHECK`

This is very subtly different; this time, the method looks at the new content of the changed fields. This is important, because sometimes you need to work out

if the current values are such that a determination needs to be run to calculate other field values.

► EXECUTE

This is the method that determines the value of certain fields based upon the values of other fields. You could have an `area` field based on height multiplied by width multiplied by length, for example. (The monster example is much more fun.)

To walk you through the implementation of these three methods, take a look at an example. This example is a testimony to the phrase “A place for everything, and everything in its place.” Instead of lumping everything together in one big class, it's better to have lots of small independent units. This way, when (not if!) you need to change something, it's much easier to do than changing one element in a gigantic monolithic structure that can come crashing down all around you. (A monolithic program is like a game of Jenga; every change is like pulling out a wooden block, and eventually the whole structure comes crashing down.)

For that reason, this example keeps the determination logic in the model class itself, and that logic gets called by the BOPF determination class. This approach—having the framework depend on the model and not vice versa—is known as *dependency inversion*. That way, when SAP decides that BOPF is old hat and comes out with a new and better framework (not that they would ever dream of doing such a thing), the model class does not have to change. It also means the model class can work with the myriad of existing business object frameworks in SAP as well as any new ones that may come knocking at your door.

In order to fill in certain fields based on the values of others, you need to add a method called `FILL_HEADER_DERIVED_FIELDS` to your model class. This method has a `CHANGING` parameter based on the combined structure (i.e., database fields plus derived fields). This method will be called automatically by the BOPF framework prior to displaying a data record. (But, again, it could be called directly by any framework.) The business logic to derive the text descriptions is shown in Listing 8.8.

```
METHOD fill_header_derived_fields.
* Fill the Hat Size Description
  IF cs_monster_header-hat_size > 10.
    cs_monster_header-hat_size_description = 'REALLY BIG HAT'.
  ELSEIF cs_monster_header-hat_size > 5.
    cs_monster_header-hat_size_description = 'BIG HAT'.
```

```

ELSE.
    cs_monster_header-hat_size_description = 'NORMAL HAT'.
ENDIF.

* Fill the Sanity Description
IF cs_monster_header-sanity > 75.
    cs_monster_header-sanity_description = 'VERY SANE'.
ELSEIF cs_monster_header-sanity > 50.
    cs_monster_header-sanity_description = 'SANE'.
ELSEIF cs_monster_header-sanity > 25.
    cs_monster_header-sanity_description = 'SLIGHTLY MAD'.
ELSEIF cs_monster_header-sanity > 12.
    cs_monster_header-sanity_description = 'VERY MAD'.
ELSEIF cs_monster_header-sanity > 1.
    cs_monster_header-sanity_description = 'BONKERS'.
ELSE.
    cs_monster_header-sanity_description = 'RENAMES SAP PRODUCTS'.
ENDIF.

ENDMETHOD. "Fill Header Derived Fields

```

Listing 8.8 Filling Derived Header Fields

The code in Listing 8.8 to fill the derived fields is deliberately simple. In real life, you would perform some text lookups to get the sales organization description based on the sales organization code or some such, or you would multiply two fields together to get a third. (The example is also unrealistic in that in the real world surely nobody would be mad enough to rename their entire product portfolio every month.)

Now, you have the business logic code; next, you'll learn how you can tell the BOPF determination class to make use of this logic. As mentioned previously, this involves the implementation of three methods: CHECK_DELTA, CHECK, and EXECUTE.

The CHECK_DELTA Method

I started off programming in 1981 on a computer with 1K of memory and a processor with a tiny fraction of the processing power we are used to today. The restrictions this forced upon me have stayed with me all my life, and even to this day I am obsessed with not executing a single line of code that does not need to be run (in order to put less strain on the CPU). In this example, you can reduce the number of lines of code that run in this way: Instead of redetermining the hat size text every time a user changes anything, only run the code when the hat size value changes. This is somewhat analogous to ON INPUT in the DYNPRO framework.

Stopping determinations being run unless fields like `hat_size` change is the job of the `CHECK_DELTA` method. The `CHECK_DELTA` method starts life as an inherited method with no code inside it. You have to redefine it in your Z class (remember, this is optional). How this works is that a table of changed nodes is passed in, and you delete any entries from that table where you don't want the determination to run.

In Listing 8.9, the first method (`CHECK_DELTA`) is in the generated BOPF determination class, and the second (`IS_DERIVATION_RELEVANT`) is in the monster model class, which is called by `CHECK_DELTA`. In a simple example like this, such an approach seems crazy, but in real life the rules will be far more complicated and the benefit of keeping framework-independent logic in the model far greater.

In the `CHECK_DELTA` method, there are three tasks to be done:

1. Use the BOPF read object (`IO_READ`) that is imported to the method to get a list of every field that has had its value changed.
2. Pass a table of changed fields to the model class method `IS_DERIVATION_RELEVANT` so that it can say if you need to run a determination or not.
3. If you decide there is really no need to run a determination, then delete the entry of your header node from the internal table of nodes that have changed.

```
METHOD /bobf/if_frw_determination-check_delta.
* Local Variables
DATA: lo_changes TYPE REF TO /bobf/if_frw_change,
      lt_changes TYPE /bobf/t_frw_change,
      ls_changes LIKE LINE OF lt_changes,
      lf_determination_needed TYPE abap_bool,
      lo_monster_model TYPE REF TO zcl_monster_model,
      lt_monster_header TYPE ztt_monster_header,
      ls_monster_header LIKE LINE OF lt_monster_header.

io_read->compare(
  EXPORTING iv_node_key = zif_monster_c=>sc_node-monster_header
            it_key      = ct_key
            iv_fill_attributes = abap_true
  IMPORTING eo_change   = lo_changes ).

IF lo_changes->has_changes( ) = abap_true.

  lo_changes->get_changes( IMPORTING et_change = lt_changes ).

  io_read->retrieve(
```



```

EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
           it_key  = ct_key
IMPORTING et_data = lt_monster_header ).

READ TABLE lt_monster_header INTO ls_monster_header INDEX 1.

CHECK sy-subrc EQ 0.

lo_monster_model = zcl_monster_model=>get_instance( ls_monster_
header-monster_number ).

lf_determination_needed =
lo_monster_model->is_derivation_relevant( lt_changes ).

ENDIF. "Are there any changes?

CHECK lf_determination_needed = abap_false.

DELETE ct_key WHERE key = ls_monster_header-key.

ENDMETHOD. "Check Delta
METHOD is_derivation_relevant.
* Local Variables
DATA: ls_changes      LIKE LINE OF it_changes,
      ls_changed_field TYPE string.

LOOP AT it_changes INTO ls_changes.
  LOOP AT ls_changes-attributes INTO ls_changed_field.
    CHECK ls_changed_field = 'SANITY' OR
          ls_changed_field = 'HAT_SIZE'.
    rf_relevant = abap_true.
  RETURN.
ENDLOOP.
ENDLOOP.

ENDMETHOD. "Is Derivation Relevant

```

Listing 8.9 Checking Changed Fields

The CHECK Method

The CHECK method takes a look at the current values and decides if running a determination needs to be done or not based on those values. This is another optional method, because sometimes whether a determination runs or not is not based on any particular field values.

In this method, you're not interested in what has changed but rather in the actual current field values. Although the `CHECK_DELTA` method is called only when the record is changed, the `CHECK` method is called on the initial load of the record and when the record is changed. If the `CHECK_DELTA` method decides that the determination does not need to run, then the `CHECK` method is not called.

It's difficult to come up with any meaningful examples of when you might use the `CHECK` method. (Maybe you would only want to fill certain transient nodes for invoices over \$10,000 or for really tall monsters.) Therefore, just to show how this is done, pretend that you don't want the determination to run if a monster has no head, because hat size and sanity become somewhat irrelevant in such a case.

Listing 8.10 shows how to code the check for a monster having no head. The coding is even simpler than the `CHECK_DELTA` method. First, you get the current values of the header record, then you pass those values to a method of the model class (`ARE_VALUES_DERIVATION_RELEVANT`) to evaluate business rules and decide if you really need to run the determination. Just as before, if you decide that the determination does not need to run, then you delete a record from the table of node keys, thus stopping the determination from running.

```
METHOD /bobf/if_frw_determination-check.
* Local Variables
  DATA: lf_determination_needed TYPE abap_bool,
         lo_monster_model TYPE REF TO zcl_monster_model,
         lt_monster_header TYPE ztt_monster_header,
         ls_monster_header LIKE LINE OF lt_monster_header.

  io_read->retrieve(
    EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
              it_key   = ct_key
    IMPORTING et_data = lt_monster_header ).

  READ TABLE lt_monster_header INTO ls_monster_header INDEX 1.

  CHECK sy-subrc EQ 0.

  lo_monster_model = zcl_monster_model=>get_instance( ls_monster_
header-monster_number ).

  lf_determination_needed =
lo_monster_model->are_values_derivation_relevant( ls_monster_header ).

  CHECK lf_determination_needed = abap_false.
```

```

DELETE ct_key WHERE key = ls_monster_header-key.

ENDMETHOD. "Check
METHOD are_values_derivation_relevant.

    IF is_header_values-no_of_heads > 0.
        rf_relevant = abap_true.
    ELSE.
        rf_relevant = abap_false.
    ENDIF.

ENDMETHOD.

```

Listing 8.10 Coding the CHECK Method for a Determination

The EXECUTE Method

The EXECUTE method is the method that actually does the determination (i.e., changes some values based on others), provided the CHECK_DELTA and CHECK methods allow it to be run. From a technical point of view, the important thing about this method is that not only do you get an imported (via the method parameters) object, READ, which allows you to read the current values, but also another imported object, MODIFY, which lets you modify these values.

In Listing 8.11, you can see that there are three simple steps to executing a determination. First, you use the imported object READ to get the values from the database for a given monster (or many monsters at once, given that a table of the keys is being passed in). Then, create an instance of your model class and perform the monster-specific logic (i.e., logic that would be the same no matter what framework you were using) to fill up the transient fields. Finally, use the imported object MODIFY to pass back the completed combined structure to BOPF.

```

METHOD /bobf/if_frw_determination~execute.
* Local Variables
    DATA: lt_monster_header TYPE ztt_monster_header,
           ls_monster_header LIKE LINE OF lt_monster_header,
           lo_monster_model TYPE REF TO zcl_monster_model.

* Clear Exporting Parameters
    CLEAR: eo_message,
           et_failed_key.

* Get the persistent (database) values
    io_read->retrieve(
        EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
                 it_key   = it_key

```

```

IMPORTING et_data = lt_monster_header ).

LOOP AT lt_monster_header INTO ls_monster_header.

* Use the model to derive the transient values
lo_monster_model =
zcl_monster_model=>get_instance( ls_monster_header- monster_
number ).

lo_monster_model->fill_header_derived_fields(
CHANGING cs_monster_header = ls_monster_header ).

* The combined structure is now full, so we pass it back to BOPF
DATA: ls_ref_to_data TYPE REF TO data.

GET REFERENCE OF ls_monster_header INTO ls_ref_to_data.

io_modify->update(
EXPORTING iv_node = is_ctx-node_key
          iv_key   = ls_monster_header-key
          is_data  = ls_ref_to_data ).

ENDLOOP. "Monsters being Queried
ENDMETHOD. "Execute

```

Listing 8.11 Executing a Determination

The wonderful thing is that you can test this straight away without having to create a little test program or some such. This is because there is a dedicated BOPF test transaction (/BOBP/TEST_UI) that shows the contents of business objects that have been saved to the database and also automatically runs the determinations.

8.2.6 Disabling Certain Commands Using Validations

When designing applications, you often restrict what commands the user can see. (In standard DYNPRO programming, this is done via PBO processing.) For example, you would not want to allow a goods issue action on a delivery object that has already been goods-issued, and you do not want to allow the HOWL command for monsters with no heads.

In these cases, rather than having a button and showing an error when the user presses it, it's common to gray out or hide buttons the user is not currently allowed to press. This can make things less confusing for the user: If there are six logs they can look at, but five of them are blank, then you want to only display the button to show the log that actually exists. Otherwise, the user has to click on

each log in turn until they find the one with the errors. (Can you guess which standard SAP transaction I am talking about here? Oh yes, it's Transaction CO08, Create Production Order. I always thought it would be lovely if, when looking at the list of possible logs you could drill into, the empty logs were grayed out.)

The exact way to gray out such buttons varies with the UI technology you're using, but in every case you can use the BOPF to see if an action is able to be processed or not and then gray out or hide the button accordingly.

In the PBO section of the calling program (the controller), you're in effect going to do a dummy run of the `HOWL` action (like running a `BAPI` in test mode) and only if this passes can the controller go ahead and let the view add a `HOWL` button the user can press. If the monster has no head, then the check will fail, and there will be no button to press.

In Listing 8.12, first you turn your monster number into the `GUID` key that BOPF likes. Then, you fill up the input structure that the `HOWL` action will be expecting; just ask for one `howl`, because if that doesn't work, then no amount of `howls` will work. For technical reasons, you have to turn the input structure into an object of `TYPE REF TO DATA`. Then, call the `check_action` method of the business object's service manager, and pass the manager a constant that refers to the name of the action to be checked (namely, `howl_at_the_moon`). As I mentioned earlier, calling a generated class directly breaks encapsulation.

If the `failed_action_keys` table contains any entries after the `check_action` method call, then the monster is unable to `howl`, and the controller should not allow the view to add the `HOWL` button to the user interface.

```
*Local Variables
TRY.
* To get a BOPF object, we need a key, not a number
DATA : ld_monster_key TYPE /bobf/conf_key.
      ld_monster_key =
      get_bopf_key_4_monster_number( id_monster_number ).

* Now we put the key into a one line table
DATA lt_key TYPE /bobf/t_frw_key.
FIELD-SYMBOLS <ls_key> LIKE LINE OF lt_key.

APPEND INITIAL LINE TO lt_key ASSIGNING <ls_key>.
<ls_key>-key = ld_monster_key.

* Populate the parameter structure that contains parameters passed
* to the action:
```

```

DATA ls_parameters TYPE zsa_howl_at_the_moon.
DATA lr_s_parameters TYPE REF TO data.

* If not even one howl is possible, no point in going on
ls_parameters-no_of_howls = 1.
GET REFERENCE OF ls_parameters INTO lr_s_parameters.

* Check to see if the HOWL action can be invoked:
DATA lo_message TYPE REF TO /bobf/if_frw_message.
DATA lt_failed_key TYPE /bobf/t_frw_key.
DATA lt_failed_act_key TYPE /bobf/t_frw_key.

CALL METHOD mo_service_manager->check_action
EXPORTING
  iv_act_key = zif_monster_c=>sc_action-root-howl_at_the_moon
  it_key      = lt_key
  is_parameters = lr_s_parameters
IMPORTING
  eo_message      = lo_message
  et_failed_key   = lt_failed_key
  et_failed_action_key = lt_failed_act_key.

IF lt_failed_act_key[] IS NOT INITIAL.
  "The action would fail - do not add the button
ENDIF.

CATCH /bobf/cx_frw INTO lx_frw.
  "The action would fail really badly, do not add the button
ENDTRY.

```

Listing 8.12 Invoking an Action Validation Manually**Note**

In Section 8.2.8, you will look at how user commands are processed in BOPF using actions and how an action validation is run automatically by the framework just before a user command (action) is executed.

8.2.7 Checking Data Integrity Using Validations

Before you save your data to the database, you want to be sure it's in a consistent state. In fact, you want to check this not only at the time of saving but often many times throughout the transaction. In BOPF, this is handled via validations. Validations are called automatically during save, but can also be called on demand. Often, in DYNPRO programs the equivalent code to perform a validation is called after every user entry (i.e., during PAI), but that could seem excessive; in any

event, you could run the validation on every roundtrip if you wanted, or only when the user presses some sort of CHECK button, just like when you're entering a supplier invoice in Transaction MIRO.

What actually defines a consistent object is naturally very application-specific; for example, a sales order with no items or no customer is not much good. In this example, say that if you enter a hat size for the monster, then the monster has to have a head. It's always the job of the model to know what data is consistent and what is not. Using this example, you'll learn how to create and then code a validation.

Creating the Validation

To create the validation, open Transaction BOB, and select the MONSTER_HEADER node. Right-click on it, and choose the context menu option CREATE CONSISTENCY VALIDATION. By this point, it will probably not come as much of a shock to you that a wizard appears to guide you through the creation process. The first two screens are nothing special: just choose a name (which will be used to generate the implementing class) and a text description, and then the name of the implementing class is proposed.

This time, you can choose when the validation will be run by selecting checkboxes (Figure 8.15). Note that you can, for example, tell the system to run a validation on the header data when item data is changed or vice versa.

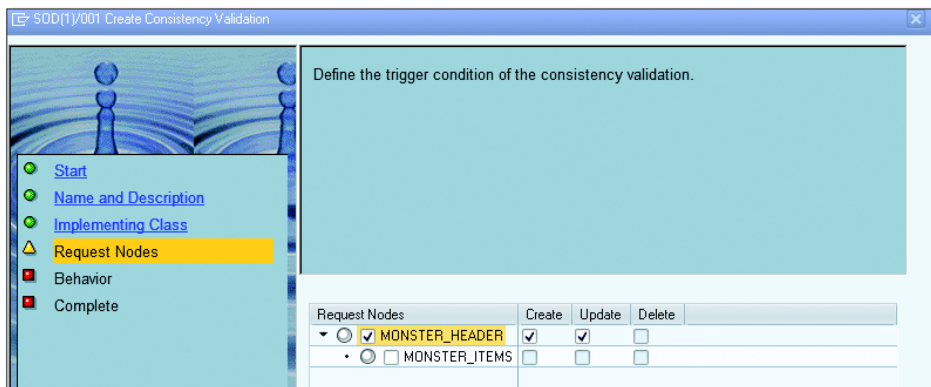


Figure 8.15 Choosing when a Validation Runs

Note that the checkboxes in Figure 8.16 control the automatic running of the validation by the framework at update time. Because the validation is implemented

as a method of a class, you can of course also call it whenever you feel like it via the service manager.

The final screen asks if you want the messages returned to prevent saving or updating the record (Figure 8.16). This is what you would want most of the time (it seems silly to tell the user there is something wrong and then save the record anyway), but it's possible to imagine a situation in which a document is not valid because it isn't approved and the user creating the document also isn't allowed to approve it.

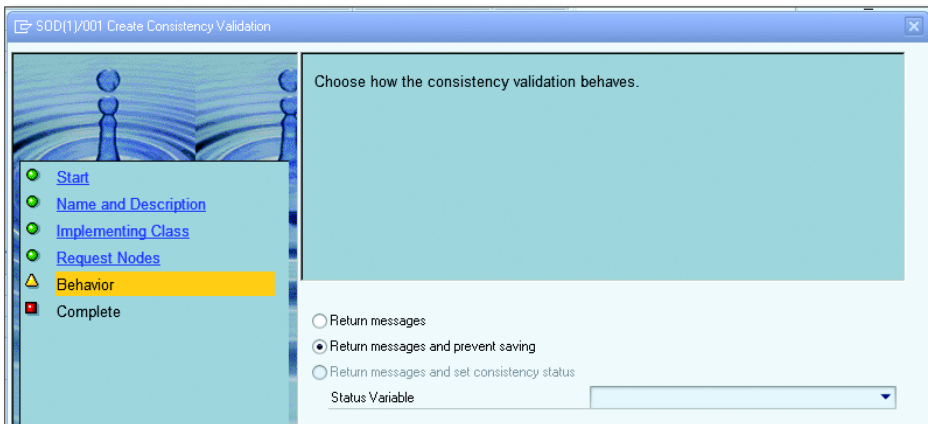


Figure 8.16 Defining Validation Behavior

Click COMPLETE, and the validation is created, along with a generated class. The monster business object is filling out quite nicely (Figure 8.17).

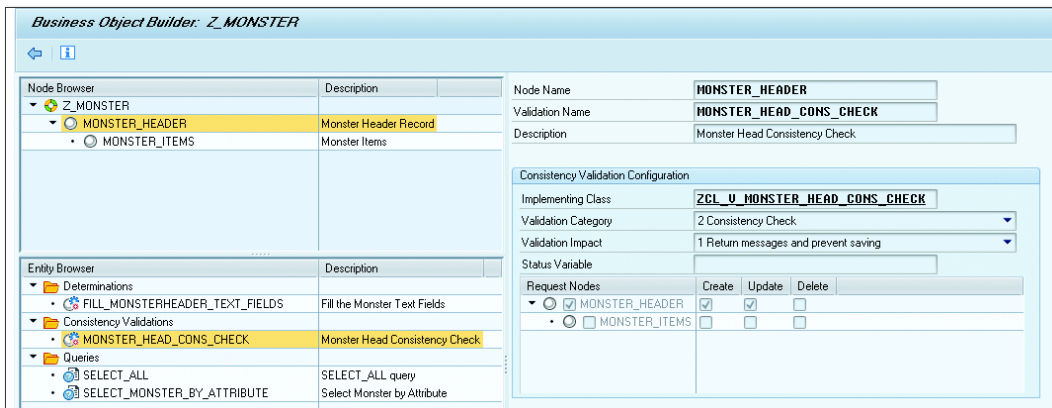


Figure 8.17 Finished Validation in Transaction BOB

Coding the Validation

The process of coding the validation is exactly the same as when you created a determination in Section 8.2.5. A custom class has been generated for you; this time, it implements the `/BOBF/IF_FRW_VALIDATION` interface. If you look at the class in SE24, you will see that it has the exact same methods as the generated determination class: `CHECK_DELTA`, `CHECK`, and `EXECUTE`.

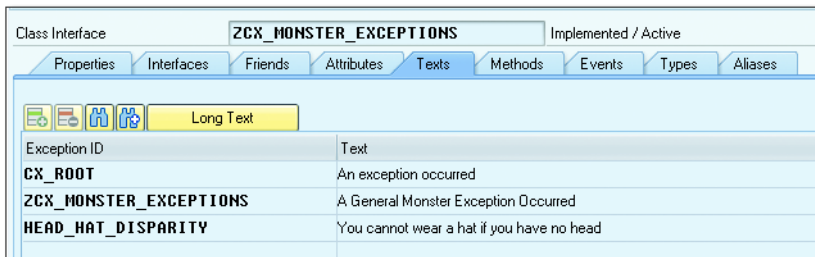
The first two methods are the same as before. To review:

- ▶ `CHECK_DELTA`
This optional method reads the entire object structure to see what has changed since the last time it looked (the last validation) and removes any nodes (from being checked) in which nothing of any interest has changed.
- ▶ `CHECK`
This optional method can look at the current values of any field in the business object and remove any nodes in which one of the fields has a value that means that the validation would serve no purpose.

Those two methods are 100% identical to the ones in the determination, which was discussed at length in Section 8.2.5 and won't be reiterated here.

The `EXECUTE` method in the validation is the one that does the actual validation work, but it's different from the `EXECUTE` method in the determination in that the error handling in a validation needs to be a lot more complicated than that used in a determination.

In a short while, you'll see that the BOPF needs a very specific type of message object. However, the monster model does not need to know that, so define your own monster specific exception class in which you will send out details of what went wrong (Figure 8.18).



Exception ID	Text
CX_ROOT	An exception occurred
ZCX_MONSTER_EXCEPTIONS	A General Monster Exception Occurred
HEAD_HAT_DISPARITY	You cannot wear a hat if you have no head

Figure 8.18 Monster Exception Class

This exception class implements the `IF_T100_MESSAGE` interface (exception classes were discussed back in Chapter 7). The BOPF framework likes traditional error messages and other frameworks (like Web Dynpro) do not, but if our model sends out a message catering to the lowest common denominator, then the receiving framework can convert it to whatever it can deal with.

Define the business logic in a method within the monster model (Listing 8.13). The model is passed the header data, examines it for consistency, and throws an exception if it does not like what it sees.

```
METHOD validate_monster_header.

  IF is_header_values-hat_size    GT 0 AND
     is_header_values-no_of_heads EQ 0.
    RAISE EXCEPTION TYPE zcx_monster_exceptions
      EXPORTING
        textid = zcx_monster_exceptions=>head_hat_disparity.
  ENDIF.

ENDMETHOD. "Validate Monster Header
```

Listing 8.13 Validation Method in Main Monster Model

You now have everything you need to code the `EXECUTE` method in the BOPF generated validation class. In Listing 8.14, you do five things:

- ▶ First, you get the data for the node (header row) that you're interested in.
- ▶ Then, you hand that data to the model so that it can throw an exception if something is amiss. If all is fine, then you move straight to the `ENDMETHOD` statement, and you're done.
- ▶ Next, if it turns out that there's an error (the model has thrown an exception), then you add the key of the failed node to a table of nodes that are failures in life (`ET_FAILED_KEY`), and this table is exported from the method.
- ▶ You then create the `EXPORTING` message parameter (`EO_MESSAGE`) by means of a standard BOPF factory method. You can then add one or many error messages to it.
- ▶ Finally, you have to create a very specific type of message object (`TYPE REF TO /bobf/cm_frw_core`). You are passing into this message object the error message that the monster model sent out by means of an exception, plus a whole bunch of constants and the location of the error. Then, add your message to the exporting parameter, and you're done.

```

METHOD /bobf/if_frw_validation~execute.
* Local Variables
  DATA: lt_monster_header TYPE ztt_monster_header,
         ls_monster_header LIKE LINE OF lt_monster_header,
         lo_monster_model  TYPE REF TO zcl_monster_model,
         lo_monster_exception TYPE REF TO zcx_monster_exceptions.

* Clear Exporting Parameters
  CLEAR: eo_message,
         et_failed_key.

* Get the current header values
  io_read->retrieve(
  EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
            it_key   = it_key
  IMPORTING et_data = lt_monster_header ).

  READ TABLE lt_monster_header INTO ls_monster_header INDEX 1.

  CHECK sy-subrc EQ 0.

* Use the model to actually perform the logic check
  lo_monster_model =
  zcl_monster_model=>get_instance( ls_monster_header-monster_number ).

  TRY.
    lo_monster_model->validate_monster_header( ls_monster_header ).
  CATCH zcx_monster_exceptions INTO lo_monster_exception.
    DATA: ls_key LIKE LINE OF it_key.

    READ TABLE it_key INTO ls_key INDEX 1. "Only one line

    "This key (node) has failed at the job of being consistent
    INSERT ls_key INTO TABLE et_failed_key.

"Create the bottle for our message
    eo_message = /bobf/cl_frw_factory=>get_message( ).

* Now we send the error message in the format the BOPF Framework
* desires
  DATA: ls_location TYPE /bobf/s_frw_location,
         lo_message  TYPE REF TO /bobf/cm_frw_core.

  ls_location-node_key = is_ctx-node_key.
  ls_location-key = ls_key-key. "I heard you the first time

  CREATE OBJECT lo_message
    EXPORTING

```

```

textid = lo_monster_exception->if_t100_message-t100key
severity = /bobf/cm_frw=>co_severity_error
symptom = /bobf/if_frw_message_symptoms=>co_bo_inconsistency
lifetime = /bobf/if_frw_c=>sc_lifetime_set_by_bopf
ms_origin_location = ls_location.

```

```

"Put the message in the bottle
  eo_message->add_cm( lo_message ).
ENDTRY.

```

ENDMETHOD. "Execute Validation

Listing 8.14 Executing a Validation

The final step in the process is to test this using Transaction /BOBF/TEST_UI, by picking a monster and setting the number of heads to zero (Figure 8.19).

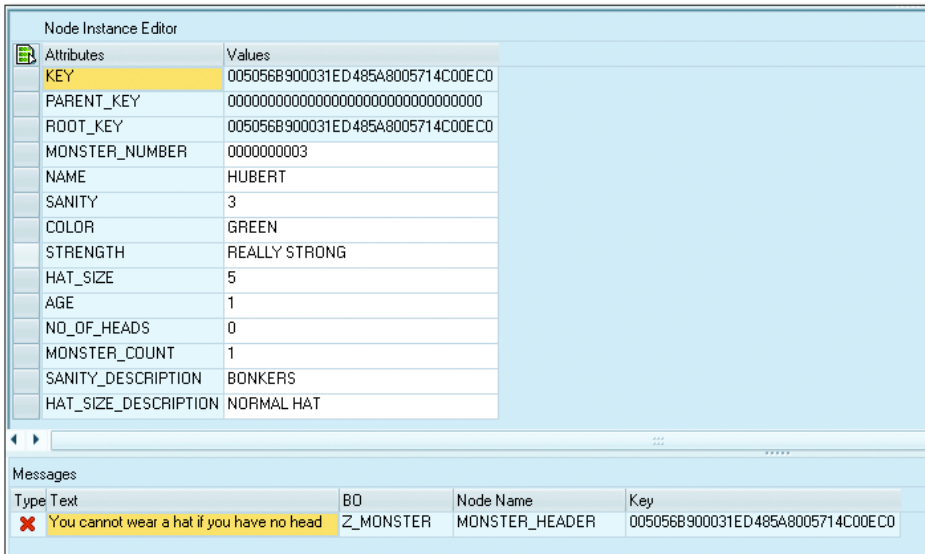


Figure 8.19 Testing the Validation

As a bonus, in Figure 8.19 you can see how in the test transaction the transient nodes, like SANITY_DESCRIPTION, have filled themselves out automatically based on the determination you created earlier without you having to do anything at all.

8.2.8 Responding to User Input via Actions

The nature of business objects is that they not only contain data but they also have behavior. A sales order can create deliveries, or an invoice could be canceled, or you could fire a nuclear missile. In this example, a monster is able to howl at the moon. In general, such actions are triggered by humans sitting in front of some sort of UI, but they could just as easily be triggered by steps in a business workflow or by some sort of batch job.

The consequence of this is that the model class does not know precisely what will trigger these requests for action, and it does not need to know. It exposes methods for each action it can perform in its public interface. That is one side of the coin: the model says what actions it can do. The other side of the coin is that the framework needs to provide a way to call these actions.

First, you'll learn how to define what actions a given business object can have in BOPF; this is known as *creating* an action. Then you'll learn how to actually code those actions—both in the underlying model and in the generated BOPF class that handles the action. Next comes action validations, which were touched on back in Section 8.2.6. Validations determine whether a particular action is allowed to be executed given the current state of the business object. Again, you'll learn both how to define these for a BOPF object and how to write the coding to make these validations work.

Creating the Action

To create the action, go to Transaction BOB and right-click the `MONSTER_HEADER` node (actions can be either at the header or the item level; in this case, the action applies to the monster as a whole). This time, choose the `CREATE ACTION` option.

Once again, the first two screens ask us for a text name and a name that will be used to generate the class that implements the action in BOPF. The next screen is the important one (Figure 8.20) in which, in addition to setting the class name, you make two other settings.

In the `ACTION CARDINALITY` set of radio buttons, you can choose if an action will be executed on lots of nodes at once (e.g., releasing all items of an order), or just one node (such as rejecting an entire invoice), or no nodes at all, which is analogous to calling a static method of a class.

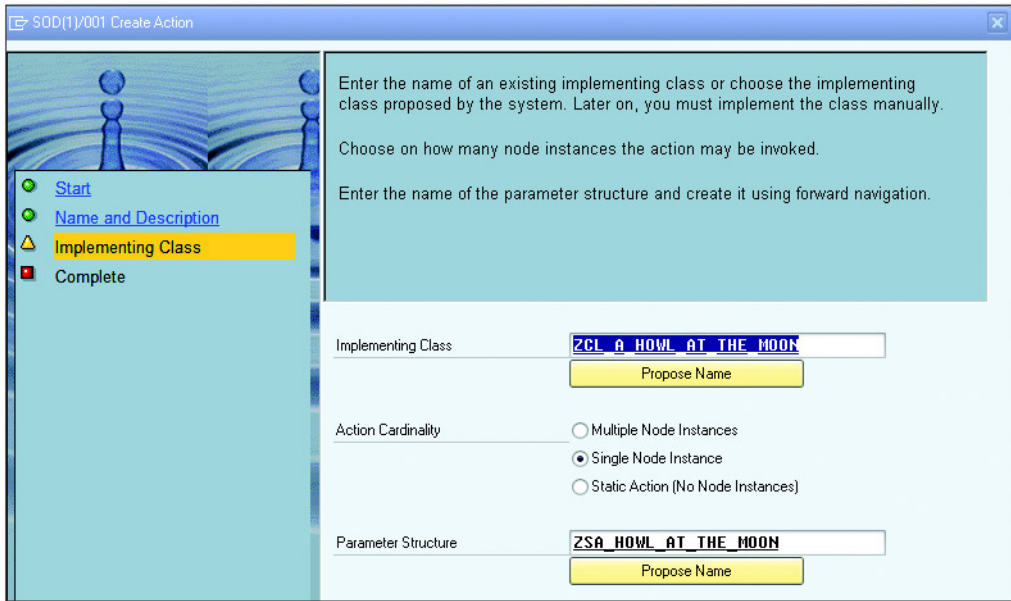


Figure 8.20 Creating an Action

Actions usually need one or more input parameters, which means that you have to define a parameter structure. In PARAMETER STRUCTURE, choose a name for that structure and then double-click that name to create the structure via SE11. Then, click COMPLETE. The result is shown in Figure 8.21.

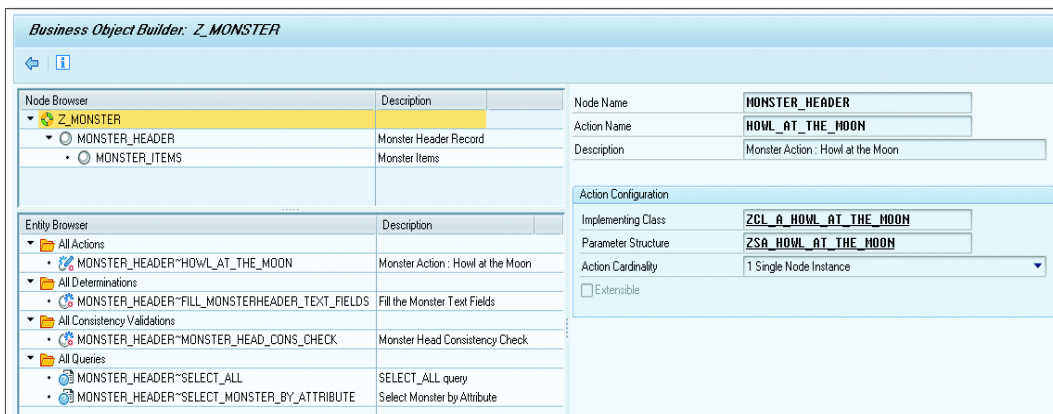


Figure 8.21 Completed Action Definition in Transaction BOB

Coding the Action

Now that the action has been created, you have to code it. The first step is to create a public method for howling at the moon. This will have a single importing parameter that corresponds to the structure you defined when creating your action in BOPF—namely, `ZSA_HOWL_AT_THE_MOON`. (You could have a totally different signature if you wanted, because the BOPF-generated action class can adapt the generated input structure to the signature of the model. This would be the case if you were hooking up an already existing model class to BOPF.)

Because methods of a model class can do literally anything that can be imagined (in real actions, you update the current state of the object, or save it to the database, or send an instruction to an external system), you will just have a method with a few lines of code that sends a message to the SAP GUI (Listing 8.15) in your model class and will concentrate on how to call this from BOPF.

```
METHOD howl_at_the_moon.

    DO is_howl_request-no_of_howls TIMES.
        MESSAGE 'Ooooooooooooooooooooooooooooo' TYPE 'I'.
    ENDDO.

ENDMETHOD. "Howl at the Moon
```

Listing 8.15 Coding the Howl at the Moon Action in the Model Class

With that piece of programming genius out of the way, it's time to turn to coding the generated action class. The generated class implements the `/BOBF/IF_FRW_ACTION` interface, which gives you three methods:

► RETRIEVE DEFAULT PARAM

This method sends out a list of what input fields are needed to call the action. It's remotely possible that a calling application might need the structure of the action input parameter for some reason. If the parameter structure keeps changing dramatically (e.g., if in the BOPF definition the DDIC type of the input parameter structure is changed), then the calling program might want to fill the parameter fields dynamically so that the calling program does not have to change when the BOPF object changes the parameter. That isn't very likely, though. This method is optional.

► PREPARE

An action can be performed on more than one object, and this method lets you limit the number of objects the action is performed upon. The difference

between this and the actual action validation is a bit blurry. This method is called twice when an action is processed. First, it's called just before a validation is performed to see if an action can be executed. Then, it's called again just before the `EXECUTE` method. Naturally, you get the same set of keys each time, so only objects that are going to have the action performed on them have an action validation processed. This method is optional.

► EXECUTE

This is the important—that is, not optional—method. It calls the corresponding action method of your monster model. The signature of the method contains `READ` and `MODIFY` objects for getting and changing any BOPF data, a generic data object coming in to store any user input parameters, and an error message object plus a table of failed keys that you can send out (Figure 8.22).

Parameter	Type	Pa...	Op...	Typing Method	Associated Type	Default value	Description
IS_CTX	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	/BOBF/S_FRW_CTX_ACT		Context Information for Actions
IT_KEY	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	/BOBF/T_FRW_KEY		Key Table
IO_READ	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type Ref To	/BOBF/IF_FRW_READ		Interface to Reading Data
IO_MODIFY	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type Ref To	/BOBF/IF_FRW_MODIFY		Interface to Change Data
IS_PARAMETERS	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type Ref To	DATA		
EO_MESSAGE	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type Ref To	/BOBF/IF_FRW_MESSAGE		Message Object
ET_FAILED_KEY	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	/BOBF/T_FRW_KEY		Action cancelled

Figure 8.22 Action EXECUTE Method Signature

The actual coding is pretty much the same as the validation and is shown in Listing 8.16. The full code is shown here just to make it clear what's happening; normally, you would identify any duplicate code between different classes and encapsulate it in a utility method.

There are several parts to Listing 8.16. First, you get the data in the header row of the monster on which the action is to be performed. You use this to get an instance of the monster model via the factory method of the monster class.

Then, you have to transform the generic data reference object, which is sent to you by BOPF. This object contains the values of the import parameters you need to send to the model. Change this to an actual structured variable, and then you can call the real action method of the model, `HOWL_AT_THE_MOON`.

The rest of the code is related to error handling. Naturally, sometimes you want to send back a success message as well (e.g., DOCUMENT XYZ HAS BEEN CREATED OR LITTLE JOHNNY'S BED HAS BEEN HIDDEN UNDER). You could send back such a success message here as well by having a return code from the monster model method and creating a message object to send to BOPF (this time with the constant /bobf/cm_frw=>co_severity_success).

Finally, because a model class should never try to send messages itself (that's the job of the UI), the model passes back any messages to be dealt with by the controller, which at the moment is BOPF.

```
METHOD /bobf/if_frw_action~execute.
* Local Variables
  DATA: lt_monster_header TYPE ztt_monster_header,
         ls_monster_header LIKE LINE OF lt_monster_header,
         lo_monster_model  TYPE REF TO zcl_monster_model,
         lo_monster_exception TYPE REF TO zcx_monster_exceptions.

* Clear Exporting Parameters
  CLEAR: eo_message,
         et_failed_key.

* Get the current header values
  io_read->retrieve(
    EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
              it_key   = it_key
    IMPORTING et_data = lt_monster_header ).

  READ TABLE lt_monster_header INTO ls_monster_header INDEX 1.

  CHECK sy-subrc EQ 0.

* Get the model
  lo_monster_model =
  zcl_monster_model=>get_instance( ls_monster_header-monster_number ).

* Transform generic import structure into concrete structure
  DATA: ls_howl_request TYPE zsa_howl_at_the_moon.

  FIELD-SYMBOLS: <ls_howl_request> TYPE any.

  ASSIGN is_parameters->* TO <ls_howl_request>.
  ls_howl_request = <ls_howl_request>.

* Off we go!
  TRY.
```

```

lo_monster_model->howl_at_the_moon( ls_howl_request ).

* Error Handling Time
CATCH zcx_monster_exceptions INTO lo_monster_exception.
  DATA: ls_key LIKE LINE OF it_key.

  READ TABLE it_key INTO ls_key INDEX 1. "Only one line

  "This key (node) has failed at the job of being consistent
  INSERT ls_key INTO TABLE et_failed_key.

  "Create the bottle for our message
  eo_message = /bobf/cl_frw_factory=>get_message( ).

* Now we send an error message in the format the BOPF Framework
* desires
  DATA: ls_location TYPE /bobf/s_frw_location,
         lo_message  TYPE REF TO /bobf/cm_frw_core.

  ls_location-node_key = is_ctx-node_key.
  ls_location-key      = ls_key-key. "I heard you the first time

  CREATE OBJECT lo_message
  EXPORTING
    textid      = lo_monster_exception->if_t100_message~t100key
    severity    = /bobf/cm_frw=>co_severity_error
    symptom     = /bobf/if_frw_message_symptoms=>co_bo_inconsistency
    lifetime    = /bobf/if_frw_c=>sc_lifetime_set_by_bopf
    ms_origin_location = ls_location.

  "Put the message in the bottle
  eo_message->add_cm( lo_message ).

ENDTRY.

ENDMETHOD. "Execute Action

```

Listing 8.16 Executing an Action

When you test this in Transaction /BOBF/TEST_UI, the result is truly beautiful. Call up a list of all your monsters, select one, click EXECUTE ACTION, and you'll magically get a pop-up asking you to fill in the import parameter for the action (Figure 8.23). Once you've done that, the howling commences (Figure 8.24).

As you can imagine, being able to test a user command before you have even started writing the main application and are still really at the data modeling stage is a wonderful step forward. The earlier you can do testing, the better.

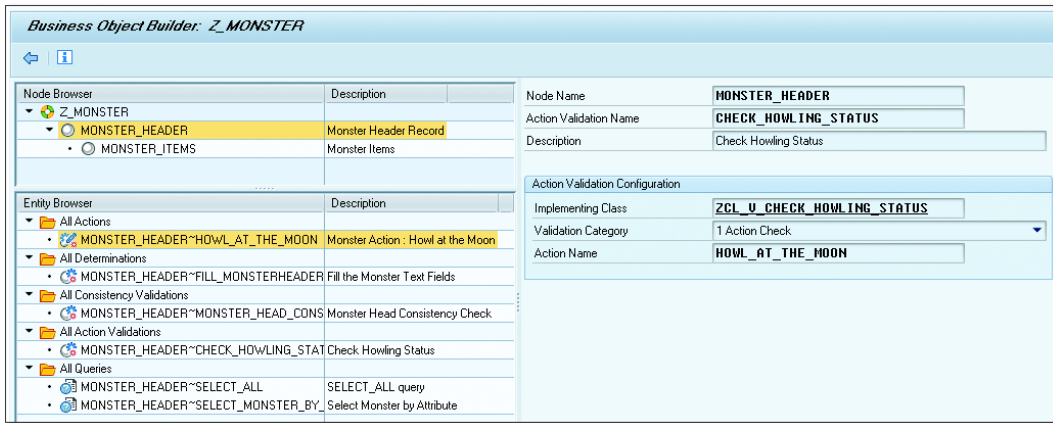


Figure 8.25 Completed Action Validation in Transaction BOB

Coding Action Validations

The generated class for each action validation implements the same interface as the generated class for consistency validations—namely, `/BOBF/IF_FRW_VALIDATION`. Therefore, the `CHECK_DELTA` and `CHECK` methods have the same meaning, and there is no need to restate that information here. However, it's worth taking a look at the coding for the action validation for the sake of completeness.

First off, create a new message, which will be shown to the users if they try to execute the action and the data is incorrect (Figure 8.26).

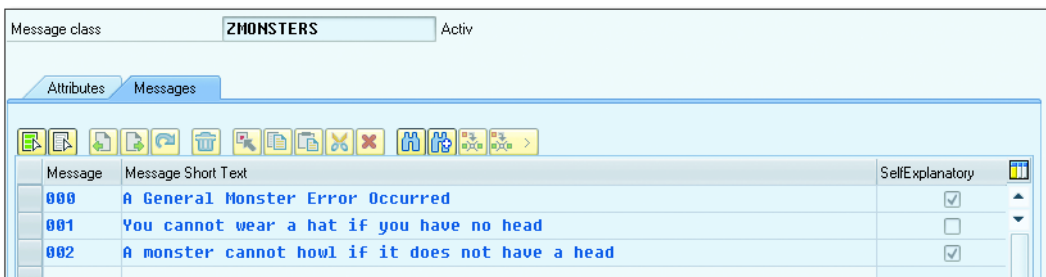


Figure 8.26 Action Validation Error Message

Then, add that message to your monster exception class (Figure 8.27).

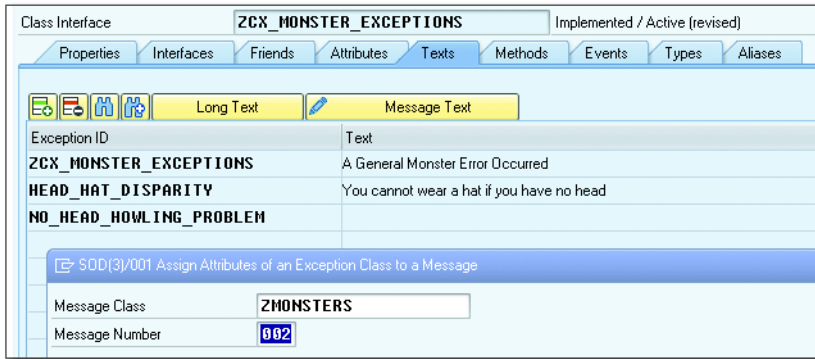


Figure 8.27 Action Validation Exception

Next, code the validation logic in your monster model class. This is shown in Listing 8.17, in which an exception is thrown if the monster has no head.

```
METHOD validate_howl_action.
```

```
    IF is_header_values-no_of_heads EQ 0.
        RAISE EXCEPTION TYPE zcx_monster_exceptions
            EXPORTING
                textid = zcx_monster_exceptions=>no_head_howling_problem.
    ENDIF.
```

```
ENDMETHOD. "Validate Howl Action
```

Listing 8.17 Validating the Howl Action in the Monster Model Class

The final stage is to code the EXECUTE method in your action validation, which is going to look suspiciously like the code in the consistency validation you created earlier (see Listing 8.18).

```
METHOD /bobf/if_frw_validation~execute.
```

```
* Local Variables
```

```
    DATA: lt_monster_header TYPE ztt_monster_header,
           ls_monster_header LIKE LINE OF lt_monster_header,
           lo_monster_model TYPE REF TO zcl_monster_model,
           lo_monster_exception TYPE REF TO zcx_monster_exceptions.
```

```
* Clear Exporting Parameters
```

```
    CLEAR: eo_message,
           et_failed_key.
```

```
* Get the current header values
```

```
    io_read->retrieve(
```

```

EXPORTING iv_node = zif_monster_c=>sc_node-monster_header
           it_key   = it_key
IMPORTING et_data = lt_monster_header ).

READ TABLE lt_monster_header INTO ls_monster_header INDEX 1.

CHECK sy-subrc EQ 0.

* Use the model to actually perform the logic check
lo_monster_model =
zcl_monster_model=>get_instance( ls_monster_header-monster_number ).

TRY.
  lo_monster_model->validate_howl_action( ls_monster_header ).
CATCH zcx_monster_exceptions INTO lo_monster_exception.
  DATA: ls_key LIKE LINE OF it_key.

  READ TABLE it_key INTO ls_key INDEX 1. "Only one line

  "This key (node) has failed at the job of being consistent
  INSERT ls_key INTO TABLE et_failed_key.

  "Create the bottle for our message
  eo_message = /bobf/cl_frw_factory=>get_message( ).

* Now we send the error message in the format BOPF
* desires
  DATA: ls_location TYPE /bobf/s_frw_location,
         lo_message  TYPE REF TO /bobf/cm_frw_core.

  ls_location-node_key = is_ctx-node_key.
  ls_location-key      = ls_key-key. "I heard you the first time

  CREATE OBJECT lo_message
  EXPORTING
    textid = lo_monster_exception->if_t100_message~t100key
    severity = /bobf/cm_frw=>co_severity_error
    symptom = /bobf/if_frw_message_symptoms=>co_bo_inconsistency
    lifetime = /bobf/if_frw_c=>sc_lifetime_set_by_bopf
    ms_origin_location = ls_location.

  "Put the message in the bottle
  eo_message->add_cm( lo_message ).

ENDTRY.

ENDMETHOD. "Execute Howling Status Action Validation
Listing 8.18 Coding an Action Validation

```

If you compare the coding for the normal validation (Listing 8.14) to the coding for the action validation (Listing 8.18), then you will see that there is only one line that is different—namely, the method you call in the monster model class. Even the precise error message is passed back transparently via the exception. Again, every time you see identical code like this it cries out to be in its own helper method, with only the aspect that varies being passed in as a parameter.

Finally, test this using Transaction /BOBF/TEST_UI. Set a monster record to have no head, and then try to execute the HOWL action (Figure 8.28).

Monster Number	Name	Species	Color	Strength	
3	HUBERT	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
4	FRED	3	GREEN	REALLY STRONG	0	1	0	1	BONKERS	NORMAL HAT
5	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
6	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
6	HUBERT	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
7	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
8	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
9	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
9	HUBERT	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT

Attributes	Values
KEY	005056B900031ED485A88B889213EECD
PARENT_KEY	00000000000000000000000000000000
ROOT_KEY	005056B900031ED485A88B889213EECD
MONSTER_NUMBER	0000000004
NAME	FRED

Type	Text	BO	Node Name	Key
✖	A monster cannot howl if it does not have a head	Z_MONSTER	MONSTER_HEADER	005056B900031ED485A88B889213EECD

Figure 8.28 Testing an Action Validation

The most common action validation is checking that a record is ready to be saved or changed in the database. Usually, that's handled by the consistency validation that is called automatically by BOPF prior to the record being saved, but you might conceivably need an action validation as well. You most likely will not, but be aware of the possibility.

8.2.9 Saving to the Database

Once the user has made all his changes or entered all his data for a new record and the program has validated that the data is consistent, the time has come to click the SAVE button and update the database.

Following the CRUD naming convention, the monster model class has `CREATE` and `UPDATE` methods (Figure 8.29). The signature is exactly the same in both cases. The method lives in the monster model class, which then instantly delegates it to an identical method in the model persistency layer class.

Ty.	Parameter	Type spec.	Description
▶	IT_MONSTER_ITEMS	TYPE ZTT_MONSTER_ITEMS	Monster Items
▶	IS_MONSTER_HEADER	TYPE ZSC_MONSTER_HEADER	Monster Header Persistent Structure
▶	EF_UPDATE_SUCCESSFUL	TYPE ABAP_BOOL	Update was Successful
▶	/BOBF/CX_FRW		BOPF Exception Class

Figure 8.29 Create and Update Methods Signature

Next you'll learn how to code a method that uses BOPF to create a new record in the database, and then how to code the method that updates existing records with changed data. Typically, a business object is changed many times after it's created.

Creating a New Record

Without further ado, it's time for you to meet the code for creating a new monster in the database. In Listing 8.19, the purpose of the method as a whole is to fill an internal table with a list of the changes that you want to make. This is a very generic structure, so you never pass in an actual structure or internal table; instead, you pass in data references in several stages.

First, in the section starting with the comment `Create Header Record`, you add an entry to the table of required "changes" (creating something is a change) for the header record, the most important bit being to generate a new GUID. Then, you do the exact same thing for the item table. Every line needs a new key as well; remember, the item lines are child nodes of the header record. Finally, in the section that starts with the comment `Here we go!`, you pass all this data into two helper methods: first, to update the data in memory (you can think of this as running a BAPI in test mode if you want; I won't tell anyone) and then, if all is well, to update the actual database.

```
METHOD create_monster_record.
* Local Variables
DATA : lt_changes_to_be_made TYPE /bobf/t_frw_modification,
```



```

lo_message          TYPE REF TO /bobf/if_frw_message,
lo_change_list      TYPE REF TO /bobf/if_tra_change,
lrs_monster_header  TYPE REF TO zsc_monster_header,
lrs_monster_item    TYPE REF TO zsc_monster_items.

```

```

FIELD-SYMBOLS: <lrs_changes> LIKE LINE OF lt_changes_to_be_made,
               <lrs_monster_header> TYPE zsc_monster_header.

```

```
CLEAR ef_creation_successful.
```

```

*-----*
* Create Header Record
*-----*
"The input parameter for the data structure is TYPE REF TO DATA
CREATE DATA lrs_monster_header.

ASSIGN lrs_monster_header->* TO <lrs_monster_header>.

<lrs_monster_header>      = is_monster_header.
"I've got a brand new pair of roller skates, you've got a brand new key
<lrs_monster_header>-key = /bobf/cl_frw_factory=>get_new_key( ).

APPEND INITIAL LINE TO lt_changes_to_be_made
ASSIGNING <lrs_changes>.
<lrs_changes>-node = zif_monster_c=>sc_node-monster_header.
<lrs_changes>-change_mode = /bobf/if_frw_c=>sc_modify_create.
<lrs_changes>-key       = lrs_monster_header->key.
<lrs_changes>-data      = lrs_monster_header.

*-----*
* Time for the item table
*-----*
DATA: lrs_monster_item LIKE LINE OF it_monster_items.

FIELD-SYMBOLS: <lrs_monster_items> TYPE zsc_monster_items.

LOOP AT it_monster_items INTO lrs_monster_item.

  CREATE DATA lrs_monster_item.

  ASSIGN lrs_monster_item->* TO <lrs_monster_items>.

  <lrs_monster_items>      = lrs_monster_item.
  <lrs_monster_items>-key = /bobf/cl_frw_factory=>get_new_key( ).

  APPEND INITIAL LINE TO lt_changes_to_be_made
  ASSIGNING <lrs_changes>.

```

```

<ls_changes>-node = zif_monster_c=>sc_node-monster_items.
<ls_changes>-change_mode = /bobf/if_frw_c=>sc_modify_create.
<ls_changes>-source_node =
zif_monster_c=>sc_node-monster_header.
<ls_changes>-association =
zif_monster_c=>sc_association-monster_header-monster_items.
<ls_changes>-source_key = lrs_monster_header->key.
<ls_changes>-key      = lrs_monster_item->key.
<ls_changes>-data     = lrs_monster_item.

ENDLOOP. "Monster Items

*-----*
* Here We Go!
*-----*
mo_bopf_pl_helper->change_data_in_memory(
EXPORTING it_changes_to_be_made      = lt_changes_to_be_made
IMPORTING ef_data_in_memory_changed = ef_creation_successful ).

CHECK ef_creation_successful = abap_true.

mo_bopf_pl_helper->change_data_in_database( ).

ENDMETHOD. "Create Monster Record

```

Listing 8.19 Creating a New Monster Record

The methods that change the data in memory and in the database have been moved into a helper class, because they're fully generic and would be used by any type of BOPF object. Speaking of these two data update methods, I've already forgotten what the first method is all about. Hang on, it's coming back...oh yes, it's all to do with memory.

The `change_data_in_memory` method implementation is shown in Listing 8.20; this method is a wrapper for the `MODIFY` call to the BOPF service manager to update the data in memory prior to actually saving the data. Because the error handling code is always going to be the same, move it from the business object-specific `CREATE` method and put it in the helper class methods, which are going to be the same for every object. While you're at it, also rename the externally visible parameters and local variables of the helper methods to be a bit more meaningful.

```

METHOD change_data_in_memory.
* Local Variables
DATA: lo_actual_changes_made TYPE REF TO /bobf/if_tra_change,
      lo_error_messages      TYPE REF TO /bobf/if_frw_message.

```

```

CLEAR ef_data_in_memory_changed.

* Change Data in Memory
mo_service_manager->modify(
  EXPORTING it_modification = it_changes_to_be_made
  IMPORTING eo_change       = lo_actual_changes_made
           eo_message       = lo_error_messages ).

ef_data_in_memory_changed =
boolc( lo_actual_changes_made->has_failed_changes( ) = abap_false ).

CHECK lo_error_messages IS BOUND.

CHECK lo_error_messages->check( ) EQ abap_true.

RAISE EXCEPTION TYPE /bobf/cx_dac
  EXPORTING
    mo_message = lo_error_messages.

ENDMETHOD.

```

Listing 8.20 Method to Change the BOPF Data in Memory

You now actually want to update the database, so code the method to do so. The code in Listing 8.21 is almost identical to the code to update the data in memory. The only difference is that instead of using the service manager class `MODIFY` method, this time you're using the BOPF transaction manager class `SAVE` method, which actually persists the data, so naturally the results are a bit more permanent.

The code in Listing 8.20 and Listing 8.21 is so generic that it can be used for any type of BOPF business object at all. You could use it in your own applications totally unchanged, though of course I beg you to understand what's going on first.

```

METHOD change_data_in_database.
* Local Variables
  DATA: lo_message          TYPE REF TO /bobf/if_frw_message,
        lf_rejected         TYPE abap_bool.

  "Off we go!
  mo_transaction_manager->save(
    IMPORTING eo_message = lo_message
           ev_rejected = lf_rejected ).

  CHECK lf_rejected EQ abap_true.

  RAISE EXCEPTION TYPE /bobf/cx_dac
    EXPORTING
      mo_message = lo_message.

```

ENDMETHOD.

Listing 8.21 Updating a BOPF Object in the Database

You may be puzzled as to why such a big deal is made of completing the transaction in memory prior to replicating that transaction in the database. In essence, it's about improving the already very good Logical Unit of Work concept in SAP, which states that related database changes must occur in a group with none missing. For example, normally if you wanted to create a sales order and then a delivery, you would have to actually create the sales order in the database and do a COMMIT. With BOPF, however, you can create the sales order object in memory, try to create the delivery business object, and if that fails abort the whole thing without going anywhere near the database.

Changing an Existing Record

Changing an existing monster record is almost exactly the same as creating a new monster record. In both cases, you pass in the entire header record and item table and let the framework handle the update.

As can be seen in Listing 8.22, the only real change from the code in Listing 8.19 is that the `change_mode` constant being passed into the changes table now contains the word `update` rather than `create`. In real life, when you find two methods that are identical except for one line, you would want to change this to just one method with the varying part coming in as a parameter.

```
METHOD update_monster_record.
* Local Variables
  DATA : lt_changes_to_be_made TYPE /bobf/t_frw_modification,
         lo_message           TYPE REF TO /bobf/if_frw_message.

  FIELD-SYMBOLS:
    <ls_changes>           LIKE LINE OF lt_changes_to_be_made,
    <ls_monster_header> TYPE zsc_monster_header.

  DATA : lrs_monster_header TYPE REF TO zsc_monster_header.

*-----*
* Update Header
*-----*
  CREATE DATA lrs_monster_header.

  ASSIGN lrs_monster_header->* TO <ls_monster_header>.
```

```

<ls_monster_header> = is_monster_header.

APPEND INITIAL LINE TO lt_changes_to_be_made ASSIGNING <ls_changes>.
<ls_changes>-node      = zif_monster_c=>sc_node-monster_header.
<ls_changes>-change_mode = /bobf/if_frw_c=>sc_modify_update.
<ls_changes>-key       = lrs_monster_header->key.
<ls_changes>-data      = lrs_monster_header.

*-----*
* Update Items
*-----*
DATA: lrs_monster_item LIKE LINE OF it_monster_items,
      lrs_monster_item TYPE REF TO zsc_monster_items.

FIELD-SYMBOLS: <ls_monster_items> TYPE zsc_monster_items.

LOOP AT it_monster_items INTO ls_monster_item.

  CREATE DATA lrs_monster_item.

  ASSIGN lrs_monster_item->* TO <ls_monster_items>.

  <ls_monster_items> = ls_monster_item.

  APPEND INITIAL LINE TO lt_changes_to_be_made
  ASSIGNING <ls_changes>.
  <ls_changes>-node = zif_monster_c=>sc_node-monster_items.
  <ls_changes>-change_mode = /bobf/if_frw_c=>sc_modify_update.
  <ls_changes>-source_node =
  zif_monster_c=>sc_node-monster_header.
  <ls_changes>-association =
  zif_monster_c=>sc_association-monster_header-monster_items.
  <ls_changes>-source_key = lrs_monster_header->key.
  <ls_changes>-key       = lrs_monster_item->key.
  <ls_changes>-data      = lrs_monster_item.

ENDLOOP. "Monster Items

mo_bopf_pl_helper->change_data_in_memory(
  EXPORTING it_changes_to_be_made      = lt_changes_to_be_made
  IMPORTING ef_data_in_memory_changed = ef_update_successful ).

CHECK ef_update_successful = abap_true.

mo_bopf_pl_helper->change_data_in_database( ).

ENDMETHOD. "Update Monster Record

```

Listing 8.22 Updating (Changing) an Existing BOPF Record

8.2.10 Tracking Changes in BOPF Objects

One feature of the SAP system that most users (and especially auditors) love is the fact that it's possible to get a complete history of who changed the value in data fields, both in master data and transactional data. This is called the *change document* mechanism. Thanks to this handy functionality, if someone were to change the bank details of your largest supplier to his own personal bank account and then change the vendor open items for that supplier to be \$100 billion just before a payment run, it would be really obvious whodunit.

Naturally, when you create business objects for custom entities like monsters, the end users will be expecting to have an option in which they can request a list of changes for a given monster record; using that option, up will come a list of who changed what and when. Traditionally, when creating a DYNPRO program to create and change a custom business object, you would create a custom change document object using Transaction SCDO (Figure 8.30).

Name of Table	Copy as internal tab.	Doc. for individual fields at delete	Name of Ref. tab.	Name of old field string
/BOFU/CDTCEXTID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
ZTMONSTER_HEADER	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
ZTMONSTER_ITEMS	<input type="checkbox"/>	<input checked="" type="checkbox"/>		

Figure 8.30 Creating a Change Document Object

As it turns out, you still have to do this for your BOPF object if you want to track changes, so please go ahead, call Transaction SCDO, and make the settings shown in Figure 8.30. You will notice that in addition to the monster header and item tables you also need to add structure /BOFU/CDTCEXTID, which you need to add here in order to get the external ID (monster number in this case) of the business object to appear in the change log together with the GUID.

Then, click the INSERT ENTRIES button, followed by SAVE. You have in effect clicked the SAVE button twice, so you might think you have created the change

document object. April fools! In fact, after saving, you also have to navigate from the menu at the top of the screen to UTILITIES(M) • GENERATE UPDATE PGM. The screen shown in Figure 8.31 appears.

Change document object	Z_MONSTER
Include	Z_MONSTER
Function group	ZMONSTER_CHANGE_DOCUMENTS
Fun.mod. structure prefix	Z
Error Message ID	CD
Error number	600

Processing type

Immediate update
 Delayed update
 Dialog

Special text handling
 Generating DATA for ABAP OO

Generate Cancel

Figure 8.31 Generating a Change Document Function Group

You need to enter a name for the function group (here, ZMONSTER_CHANGE_DOCUMENTS) and say what your namespace prefix is (the default value is Y, but most companies use Z to start off the names of their custom objects). All the other fields can be left set to their default values. Click the GENERATE button, and you will see a list of functions modules, includes, and structures as long as your arm. The SAP system needs all these to handle change documents correctly. When you see that list, click the SAVE button yet again. Now, you're finally done.

When you have created such a change document object in a manually created DYNPRO program, you have to make sure that you call a generated function module during the update task after saving the newly created or changed data (i.e., you're pretty much coding everything yourself using all the constructs the system has generated for you). As might be expected, BOPF takes care of this—by using a so-called delegated object, which in this case has the job of tracking changes to the data in each of your monsters and saving that data to the database.

Delegated Objects

A delegated object is a BOPF business object that encapsulates commonly used functionality used in business object processing in order to make that functionality available for reuse in multiple business objects. The change document is a very good example of such a delegated business object, because virtually all other business objects will want to use it—but there are several others available, such as objects to handle addresses, long texts, and attachments.

This is the same concept you've seen throughout the book in which there are small, reusable helper classes that do one specific thing being used by main classes, such as our monster model class.

The SCDO step explained previously is always needed when dealing with change documents. The steps specific to setting up change documents for a BOPF object are then performed subsequently and are as follows:

1. Create a subnode of your root node, and choose object `/BOFU/CHANGE_DOCUMENT` as the delegated object.
2. Create an association for the new change document node.
3. Create a subclass of standard class `/BOFU/CL_CDO_BO_GENERICCALLBACK`, and redefine two of its methods.
4. Link the custom business object, change document objects (as created in Transaction SCDO), and the callback subclass you just created together using a customizing table in the IMG.
5. As always, you should of course test to make sure everything works correctly.

Each of these steps is discussed next.

Creating a Subnode for a Delegated Object

In this step, you'll attach a delegated object to your monster object to take charge of tracking all data changes. Transaction BOB cannot handle adding delegated objects, so you have to go to the old-fashioned Transaction `/BOBF/CONF_UI`, which was the predecessor of BOB.

Note

Transaction `/BOBF/CONF_UI` will have the `CHANGE` button grayed out unless you set a `PID` for you user called `/BOBF/CONF_PROTOTYPE` to the value `X`.

Go into Transaction /BOBF/CONF_UI, click the icon to go into change mode, and then select your monster object from the tree on the left-hand side. When you cursor is on the MONSTER_HEADER node, right-click and follow the menu path CREATE SUBNODE • BUSINESS OBJECT REPRESENTATION NODE. The screen shown in Figure 8.32 appears.

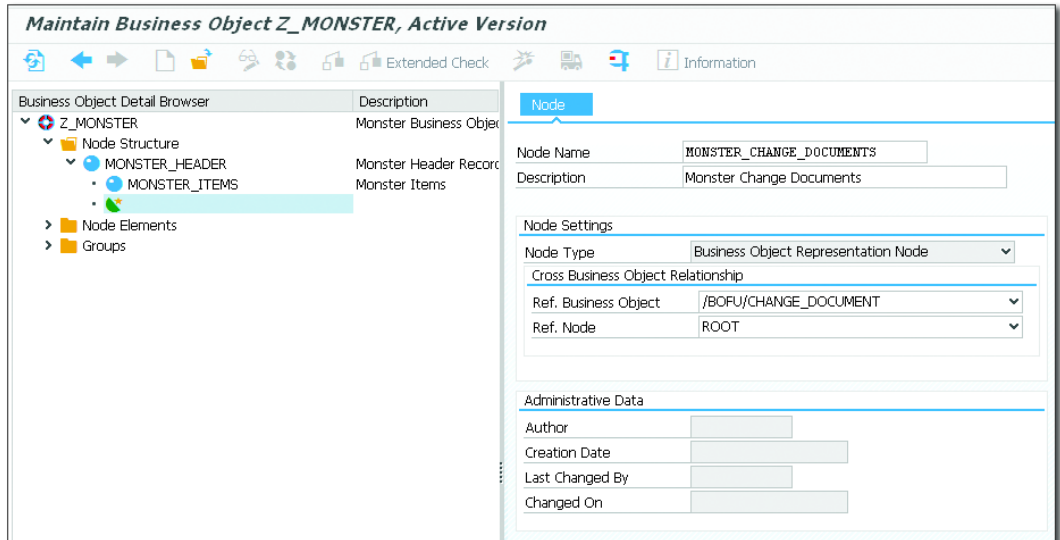


Figure 8.32 Creating a Subnode for Change Documents

You need to enter a name for the node (like MONSTER_CHANGE_DOCUMENTS) and a description. The NODE TYPE field defaults to BUSINESS OBJECT REPRESENTATION NODE. Then, open a dropdown on the REF. BUSINESS OBJECT field and choose /BOFU/CHANGE_DOCUMENT. This is the delegated object you're attaching to your monster object (i.e., you're adding in a standard reusable class to give your monster object some extra commonly used functionality). Once you press the key, you will then be able to open a dropdown on the REF. NODE field, and the value you want to select is ROOT.

Click SAVE, and the name of your new node appears in the tree on the left-hand side of the screen in Figure 8.32. You will notice that the icon is a little different (half a green ball as opposed to a whole blue ball) to reflect that this is a delegated object.

Creating an Association for the Change Document Subnode

Here, you'll link a class that performs change document tasks to your main monster node. Still in Transaction /BOBF/CONF_UI, open up the NODE ELEMENTS folder, and select the MONSTER_HEADER node. Right-click to bring up the context menu, and navigate to CREATE • CREATE ASSOCIATION. The screen shown in Figure 8.33 appears.

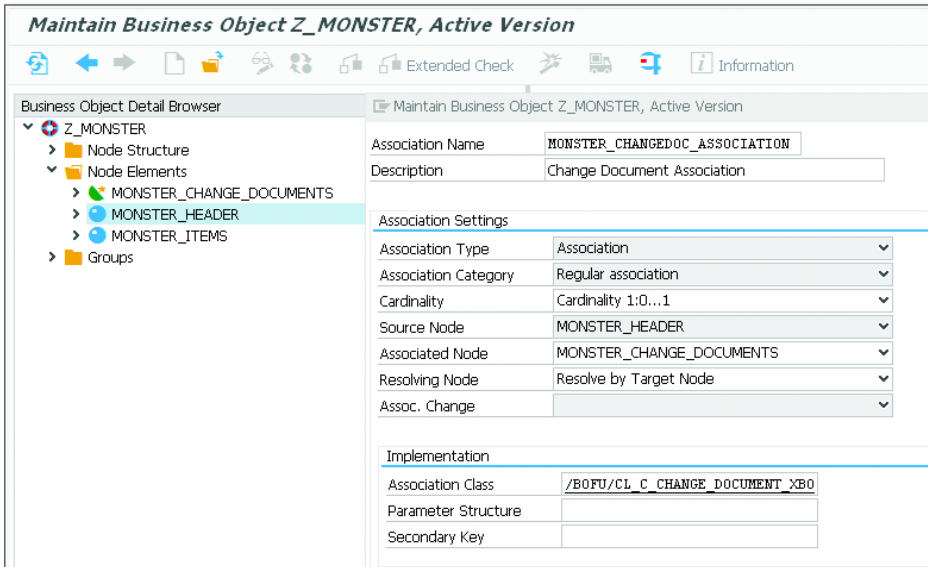


Figure 8.33 Creating a Change Document Association

You have to fill in a name and description and change the CARDINALITY field to CARDINALITY 1:0...1. Then, open the dropdown on ASSOCIATED NODE to select your change documents node. Finally, specify the association class as /BOFU/CL_C_CHANGE_DOCUMENT_XBO. Click the green checkmark at the bottom of the screen, and the pop-up box shown in Figure 8.34 appears.

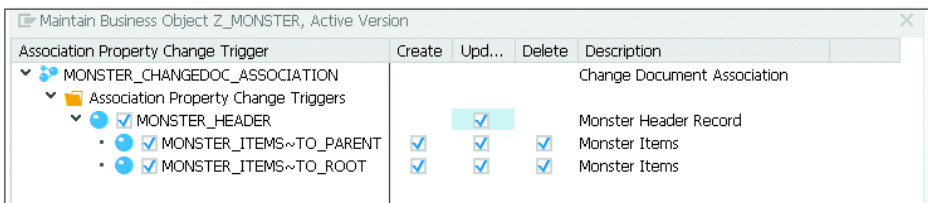


Figure 8.34 Monster Object Change Document Triggers

The pop-up box in Figure 8.34 defines what events will trigger the creation of a change document. You want every single type of event to be tracked, so select every checkbox in sight. After you've finished selecting boxes, use the green checkmark at the bottom of the pop-up box to continue, and you're done. A new entry appears on the left-hand side of the screen to denote your monster change document association.

Creating a Callback Subclass

Next, you need to subclass the standard BOPF change document class so that you can redefine two of its methods if need be. Go into Transaction SE24, and create a subclass of `/BOFU/CL_CDO_BO_GENERICCALLBACK`. You will then see a list of standard methods that can be redefined (Figure 8.35).

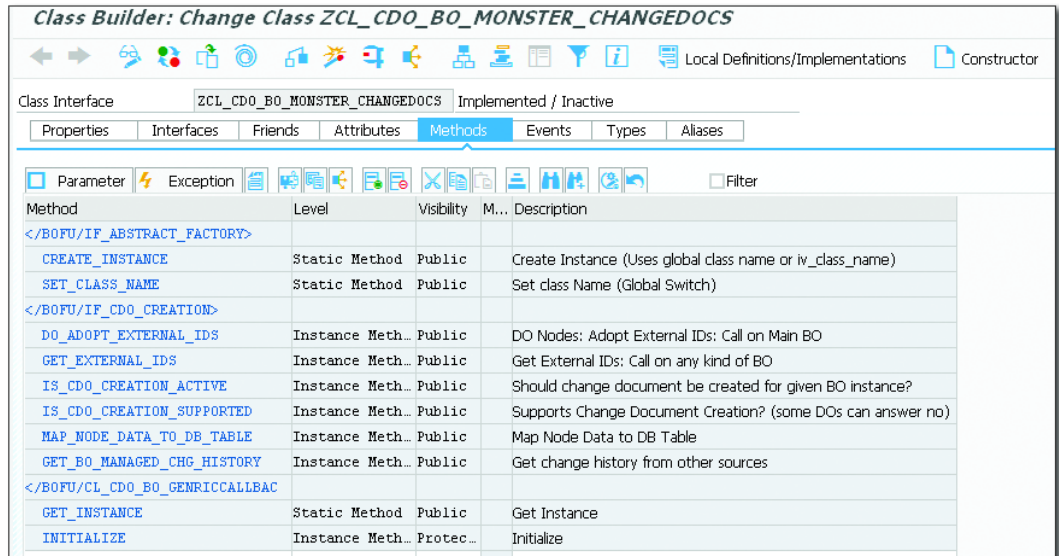


Figure 8.35 Subclassing a CDO Callback Class

The standard help invites you to redefine method `IS_CDO_CREATION_ACTIVE`, and you can then pass back a table of document (object) numbers that will not have change documents generated. I cannot think of a use for this for the life of me, so leave that method as is.

The method you do want to redefine is `GET_EXTERNAL_IDS`. The default behavior of the BOPF change documents is to keep track of object changes by using the GUID. Because this number will not mean anything to anybody, you have to add in a human-friendly reference number as well (in this case, the monster number) so that you can work out what monsters have actually been changed. The code for redefining this method to handle your monster object is shown in Listing 8.23. This method gets called because you added structure `/BOFU/CDTCEXTID` to your monster object in Transaction SCDO. Code in the standard SAP class looks to see if that structure has been added, and if it has, then the `GET_EXTERNAL_IDS` method is called.

In the code in Listing 8.23, first the incoming node is queried to make sure the object being changed is the monster header. If so, then a loop is performed of all header records being changed (usually only one), and the external ID (monster number in this case) is extracted from the header record and exported into table `ET_EXTERNAL_IDS`.

```
METHOD /bofu/if_cdo_creation-get_external_ids.
* Local Variables
  DATA: ls_ext_id          LIKE LINE OF et_external_ids,
         ls_monster_node TYPE zsc_monster_header.

  CASE iv_node_key.
    WHEN zif_monster_c=>sc_node-monster_header.
      LOOP AT it_node_data INTO ls_monster_node.
        ls_ext_id-key      = ls_monster_node-key.
        ls_ext_id-ext_id = ls_monster_node-monster_number.
        INSERT ls_ext_id INTO TABLE et_external_ids.
      ENDOLOOP.
    WHEN OTHERS.
      RETURN.
  ENDCASE.

ENDMETHOD.
```

Listing 8.23 Redefining a Standard BOPF Change Document Method

Later, you'll see that the redefined method shown in Listing 8.23 adds an extra entry into the change document database tables to store the external ID (monster number) and link it to the GUID of the business object.

Linking all the Change Document Components Together in the IMG

Finally, you'll need to link up all the jigsaw pieces you've created relating to change documents and put them all together in a configuration table. Go into the

IMG (Transaction SPRO), and navigate to CROSS-APPLICATION COMPONENTS • PROCESSES AND TOOLS FOR ENTERPRISE APPLICATIONS • REUSABLE OBJECTS AND FUNCTIONS OF THE BOPF ENVIRONMENT • MAINTAIN BO-SPECIFIC CHANGE DOCUMENT OBJECTS. The screen shown in Figure 8.36 appears.

Figure 8.36 BOPF IMG Change Document Configuration

In the screen shown in Figure 8.36, you fill in three fields to link three components together: the business object type for which we want to keep track of changes as created in Transaction BOB, the change document object as created in Transaction SCDO, and the change document class as created in Transaction SE24 as a subclass of the standard BOPF change document class. Save your changes.

Testing

You can now test that what you've done is actually working. You test this in the same way you tested everything else in this chapter: via Transaction BOBF/TEST_UI. Here, you pick a monster and change its age (for example, from one to five days), and save your changes.

When you look in the database via Transaction SE16 and query tables CDHDR and CDPOS, you'll see the screens shown in Figure 8.37 and Figure 8.38.

Data Browser: Table CDHDR Select Entries 1							
Table: CDHDR							
Displayed Fields: 14 of 14 Fixed Columns: 4 List Width 0250							
MANDANT	OBJECTCLAS	OBJECTID	CHANGNR	USERNAME	UPDATE	UTIME	TCODE
001	Z_MONSTER	005056B900031ED485A8D69FFE75AECO	0000151661	HARDYP	09.01.2015	04:29:17	/BOBF/TEST_UI

Figure 8.37 Change Document Header Record

MANDANT	OBJECTCLAS	CHANGENR	TABNAME	FNAME	CHNGIND	VALUE_NEW	VALUE_OLD
001	Z_MONSTER	0000151661	/BOFU/CDTCEXTID	EXT_ID U	U	0000000005	
001	Z_MONSTER	0000151661	ZTMONSTER_HEADER	AGE	U	5	1

Figure 8.38 Change Document Item Records

The header record (Figure 8.37) tells you who changed the monster object and when and the transaction code that was used. The GUID means nothing to a human, so you use the `CHANGENR` field to query the `CDPOS` table (Figure 8.38). Here, you can see that the age was changed from one to five days, and furthermore an entry was added for `EXT_ID`, which tells you that it was monster number 5 that was changed.

SAP provides Transaction `RSSCD100` to call up an ALV report to show a joined view of the two change document tables. You could also code a custom action to query the change document database tables, but if you're using a Web Dynpro application to display your business object, then there's no need to code anything yourself: Web Dynpro components `/BOFU/CHANGE_DOC_TAB` and `/BOFU/CHANGE_DOC_TAB2` come as standard to show change logs. (Web Dynpro is outside the scope of this chapter—but turn to Chapter 12 for more!)

Summary

You've now come to the end of this imaginary transaction. You've chosen the record to create or change, made life easy for the user with some determinations to show text fields and calculated fields, used validations to check the data integrity, and saved the record for later use. After the record has been saved, you can keep track of who created or changed the record.

8.3 Custom Enhancements

In this section, you'll first take a quick look at what SAP has provided as a mechanism for enhancing its own standard business objects, and then learn how you can take the complexity out of BOPF programming.

8.3.1 Enhancing Standard SAP Objects

So far, SAP has only used BOPF in a few areas, including transportation management and environmental health and safety. The head of SAP development has claimed that BOPF is at the core of all new SAP developments, so it seems reasonable to expect the number of standard SAP BOPF business objects to increase over time.

In an ideal world, SAP would provide BOPF objects for obvious things like sales orders and purchase orders. If this ever occurred, we “customers” would still not be happy, because we like to add custom fields to the standard SAP tables, like VBAP, and to add our own business logic. As it turns out, SAP has thought about this, and the good news is that standard (or indeed custom) BOPF objects can be enhanced with your own functionality. The only constraint is that the object has been marked as `extensible` in order to be extended, in the same way that a class that is marked as `final` cannot be subclassed.

As you’ll see, extending a standard BOPF object is pretty much exactly the same as subclassing a standard class. In addition to adding your own fields (and sub-nodes), you can add new determinations, validations, and actions and even enhance the existing ones that SAP has created. When it was not possible to create your own custom business objects in BOPF, the transaction to enhance standard SAP business objects was BOPF_EWB (Figure 8.39)—but now that you can create your own business objects you can use Transaction BOB to both create new business objects and enhance standard SAP ones. (The only advantage Transaction BOPF_EWB had was that it showed a huge picture of toy building blocks spelling out “BOPF.”)

In any event, whatever transaction you are using, the procedure is the same. Both screens have a CREATE ENHANCEMENT button at the top, so select a standard SAP-delivered business object, and click the CREATE ENHANCEMENT button. A wizard will appear (Figure 8.40).

On the NAME screen of the Enhancement Wizard, you are just creating a name and description for a new business object that will wrap the standard SAP business object. Note that there is a separate field for the prefix. Most SAP customers use a Z to prefix their objects, so don’t start the enhancement name with a Z as well; otherwise, the new business object will have two Z’s at the start of its name.

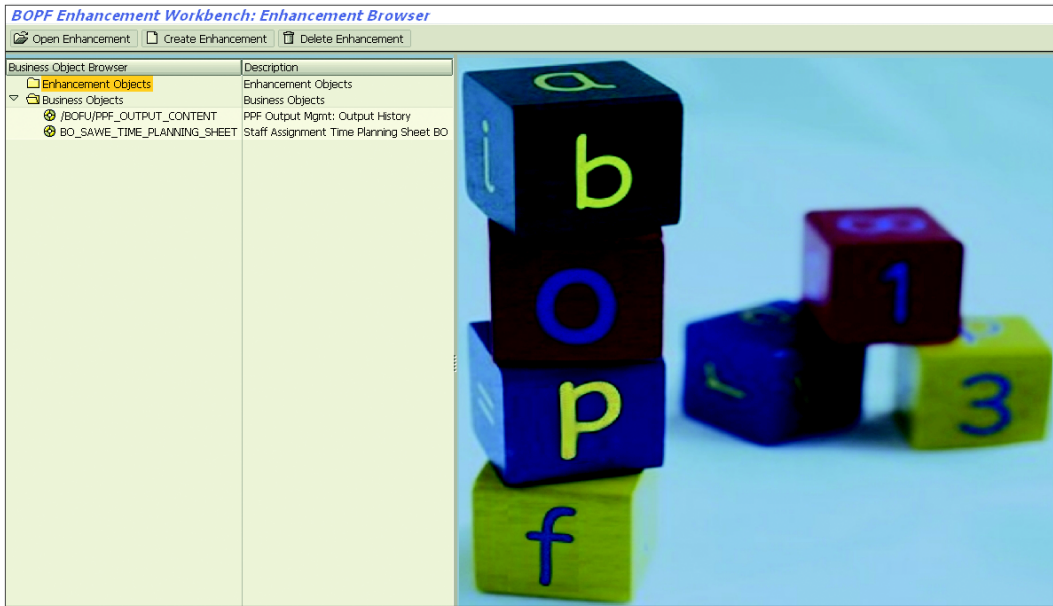


Figure 8.39 BOPF Enhancement Workbench

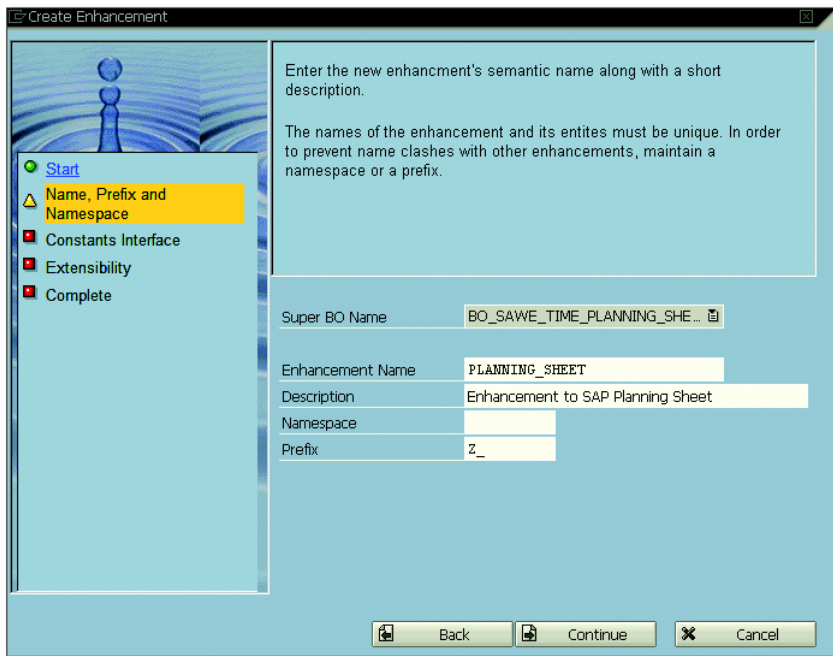


Figure 8.40 Enhancement Wizard

The next two screens are too boring to show. On the next screen in the wizard, you're asked for the name of the constants interface; this is exactly the same as the steps we covered in Section 8.1. The constants interface provides a link between the human-readable names of the business object and the internal GUID references. The last screen of the wizard asks if you want to make your new enhancement object extensible; that is, you could let some future developer come along and make an enhancement to the enhanced object you're now creating. It's wise to say yes to this, for the same reason you should very rarely define a class as `final`; you just never know what requirements will come along in the future, so keep your options open.

When you click the **COMPLETE** button at the end of the wizard, a new business object is created that has the exact same child nodes as the original object (Figure 8.41).

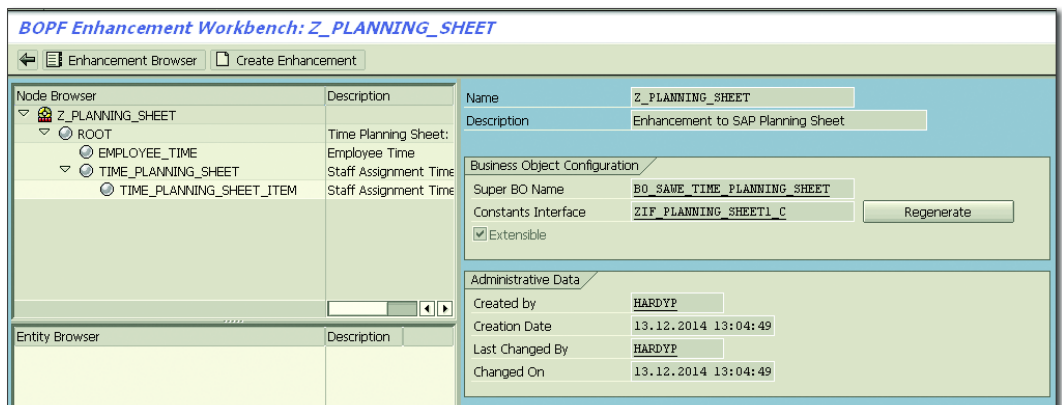


Figure 8.41 Enhanced Business Object

Now the object is yours, and you can do with it what you will. As mentioned previously, in addition to obvious things like adding new Z fields to the existing structure, you can also create new subnodes or actions and validations. There's really no point in going through these one by one, because the procedure is exactly the same as when you created actions and validations for your custom business object earlier in the chapter, and I'm sure I don't need to tell you how to add an extension structure to a standard SAP table!

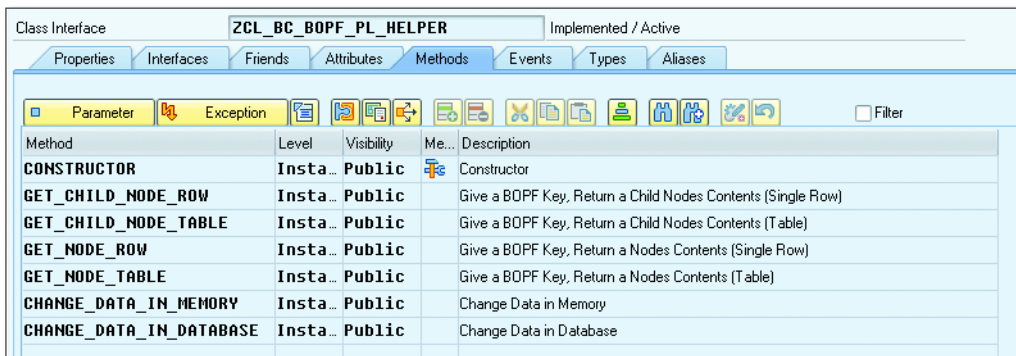
If you're bursting for further information on this topic, I refer you to the fifth of the excellent series of blog posts by James Wood on the subject of BOPF; there's

a link to these blog posts in the “Recommended Reading” box at the end of the chapter.

8.3.2 Using a Custom Interface (Wrapper)

What puts a lot of people off of BOPF is the sheer amount of coding that seems to be needed to perform routine tasks that take only a line or two of code in “traditional” ABAP. However, this is always going to be the case when you have a very generic framework intended to handle just about anything, which is after all SAP’s market: SAP wants to produce software that can be used by any company (or government department) in any country.

When I was writing some code to demonstrate BOPF, first I got it working, and then I split the code into small methods, each of which did one thing only, as per the OO philosophy. That left me with some fully generic methods that had nothing to do with monsters and could be put in a reusable Z class for the next time I wanted to use BOPF for something. Code that never changes is crying out to be abstracted to this or a similar wrapper class. Thus, you will notice that the code samples in this chapter use a custom wrapper class to surround a lot of the class in standard BOPF classes and methods (Figure 8.42).



Method	Level	Visibility	Me...	Description
CONSTRUCTOR	Insta...	Public	Me...	Constructor
GET_CHILD_NODE_ROW	Insta...	Public	Me...	Give a BOPF Key, Return a Child Nodes Contents (Single Row)
GET_CHILD_NODE_TABLE	Insta...	Public	Me...	Give a BOPF Key, Return a Child Nodes Contents (Table)
GET_NODE_ROW	Insta...	Public	Me...	Give a BOPF Key, Return a Nodes Contents (Single Row)
GET_NODE_TABLE	Insta...	Public	Me...	Give a BOPF Key, Return a Nodes Contents (Table)
CHANGE_DATA_IN_MEMORY	Insta...	Public	Me...	Change Data in Memory
CHANGE_DATA_IN_DATABASE	Insta...	Public	Me...	Change Data in Database

Figure 8.42 Custom Wrapper Class

If you look inside the methods of ZCL_BC_BOPF_HELPER using the downloadable code sample you get as part of this book, then you will see that these methods make no reference to monsters or any other type of BOPF object. The idea is that such methods, and indeed the entire class, should be reusable in the next BOPF entity you create without any changes whatsoever.

Therefore, if you work on the computer system of a zoo that contains only mythical animals, say, then next week you might be called upon to create a `unicorn` business object in the SAP system. You could reuse the previous helper class as is without looking at it. However, I would urge you to look at the examples in which it's used (in Section 8.2.9, mainly) and try to understand what's going on. It's highly likely that during your work you'll find yourself writing some code that falls into the generic basket and can be abstracted into a new method of the preceding class or to a different helper class if the code's purpose is not database relevant.

If you do find the need to do this, then when creating wrapper classes it makes sense to change the parameter names of the wrapper methods so that they make it a bit clearer to programmers calling the custom method what was actually required. As an example, the standard method asks for a node key, and you might try to pass in the object key, when what is really required is a GUID that indicates what type of node you're dealing with. Therefore, rename the parameter `node type`. This is the adapter pattern: in this case, you're adapting the standard SAP parameter names into something easier for a human to understand.

For the same reason, it often makes sense to rename the method names. If you find yourself having to put a comment above a call to one of your custom methods in order to explain what it does, you should rename the method to exactly match the comment you have just written, and then get rid of the comment (which is now redundant).

One or more wrapper classes takes away most of the pain of using BOPF, but in some areas you're still going to end up with more code than you're used to.

8.4 Summary

This chapter discussed the new Business Object Processing Framework (BOPF) SAP applies to represent business objects such as sales orders and any custom equivalents you may create. I'm going to have to leave it to you to experiment with writing an application using BOPF and seeing if there is less effort overall than the traditional method; SAP certainly believes this to be case, and I'm inclined to agree.

Recommended Reading

- ▶ *Design Patterns: Elements of Resuable Object-Oriented Software* (Gamma et al., Addison-Wesley, 1994)
- ▶ Navigating the BOPF: Part 1—Getting Started: <http://scn.sap.com/community/abap/blog/2013/01/04/navigating-the-bopf-part-1--getting-started> (James Wood)
- ▶ Project Objectify—Continued: <http://scn.sap.com/community/abap/blog/2014/03/22/project-objectify--continued> (Bruno Esperenca)

*Any fool can make a rule
And any fool will mind it.*
—Henry David Thoreau, Journal #14

9 BRFplus

It is one of the great truths in life that rules are with us always—just like death and taxes—and the same is true in SAP. A *business rule* defines what outcomes should occur based upon the values of one or more input criteria. An example is as follows: “Australians who do not have private health care have to pay 2.5% extra income tax.” In this case, the input criterion is whether any given individual has such insurance; the outcome is whether or not that individual has to pay more tax.

A *business rules framework* (BRF), also known as a business rule management system (BRMS), is a framework that enables you to express such rules within your computer system. It consists of three components: a UI, a rules repository, and a rules engine. An example of a business rules framework—and the subject of this chapter—is, of course, BRFplus.

One of the purposes of a business rules framework is to isolate a process from the decision logic that is a part of that process. A *process* is the high-level end goal achieved by a set of decisions—for example, the process of making monsters. Processes tend to be the same for all companies and very rarely change; in OO programming terms, the process is the abstraction, which cares not about the details.

The *decision logic*, on the other hand, is the individual decisions that go into the bigger process. This can be very different between companies and can also change all the time. No two mad scientists can agree on what the best recipe is for making a monster or which is the best sort of hunchback to assist them. In OO programming terms, these are the details, which depend upon the abstraction.

In Figure 9.1, you can see the difference between a process and the decisions in that process.

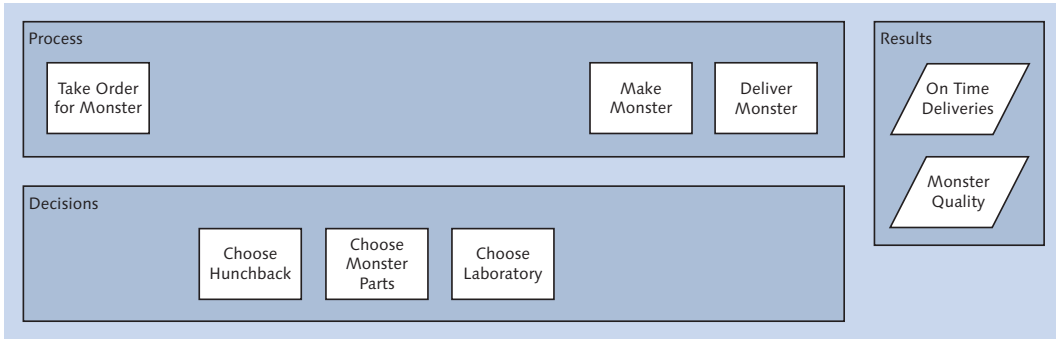


Figure 9.1 Process vs. Decisions

In the end, the results are the same among companies; all companies are judged on the same benchmark. Baron Frankenstein always aims to appear in the top-right-hand corner of the Gartner Magic Quadrant for Monsters. Figure 9.2 adds another level to the diagram: a rules engine.

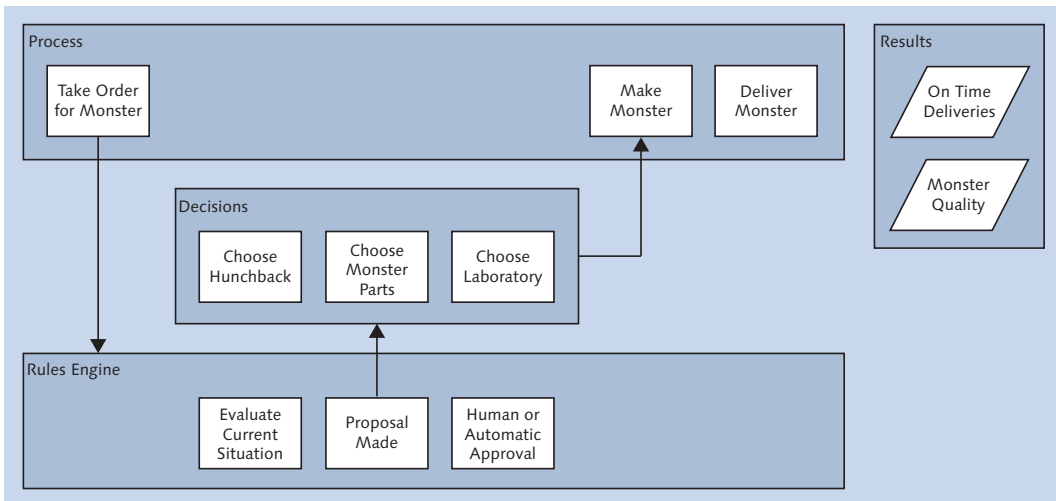


Figure 9.2 Process vs. Decisions vs. Rules Engine

A rules engine splits up the decision logic into two parts: the actual decision itself and how that decision is made. With the use of a rules engine, you can input a bunch of facts about the current situation, they swirl around inside a black box, and out comes a proposal as to what decision you should make—which then gets approved or rejected by something, be it a human or a machine.

The promise of the BRFplus framework is to (a) deliver a platform in which rules can be set up and changed in an easier and more flexible manner than anything before in SAP's long history, (b) enable rules to be more easily visible than was previously the case, and (c) enable rules to be owned by the business in a very real sense. As part of that lofty goal, SAP delivers BRFplus at no extra charge, bundled with your SAP NetWeaver license. This is worth stressing: BRFplus is not a separately licensed product. If you are on ECC 6.0, then you have this already and can use it at no additional cost. Trust me on this! When you start playing with the BRFplus transaction, it might not look like the rest of the ABAP Workbench transactions—such as SE80 and its derivatives (e.g., SE24 and SE37)—but that is only because the user interface is in Web Dynpro. BRFplus is not an external system, and it does not run on an external system. BRFplus is wholly ABAP, just like any program you write.

SAP Decision Service Management

There is also a "premium" version (i.e., a version you have to pay for) of BRFplus called SAP Decision Service Management. This book only covers technologies that come bundled in with the normal SAP license, so this chapter does not discuss SAP Decision Service Management.

Although BRFplus is a useful tool, it also involves extracting a lot of conditional logic and letting business users control business rules—so a lot of programmers tend to avoid it. This chapter discusses the best use cases for BRFplus and how to get around some of its perceived shortcomings.

You will start by looking at how rules are traditionally stored within SAP systems (Section 9.1). Then you'll see two actual business examples (with the names changed to protect the innocent) that show in a very clear way how BRFplus can accomplish something easily that customizing tables and ABAP would have to struggle very hard to match (Section 9.2 and Section 9.3). Next, take a quick look at some very useful features of BRFplus: a way to simulate how any given business rule reaches a particular decision (Section 9.4) and integration with SAP business workflow (Section 9.5). Finally, the chapter will conclude by looking at the ways BRFplus can be enhanced if it does not do everything you want out of the box (Section 9.6). The good news is that the framework was designed with this in mind.

9.1 The Historic Location of Rules

You now know what rules are—but where are they stored? They start, of course, in people's heads. Prior to BRFplus, the rules in people's heads were then converted to rules in SAP by the use of customizing tables. In some cases, when the customizing tables were not enough, they were coded in ABAP. This section discusses each of these rule locations.

9.1.1 Rules in People's Heads

Rules always start out in the head of a business expert—in this case, Baron Frankenstein. Baron Frankenstein has rules inside his head for working out what ingredients to use when making a monster. There are two main questions that need to be answered: How sane does the monster need to be, and how scary does the monster need to be?

There are three levels of sanity: mad (sanity level: 15% and under), average (sanity level: 16% to 74%), and extra sane (sanity level: 75% and over). There are also three levels of monster scariness: normal, scary, and extra scary.

Baron Frankenstein's job is to build a monster that has the customer's required sanity level and scariness level using three possible sets of ingredients:

- ▶ **Sugar/spice/all things nice**

This ingredient is very cheap, but is only suitable for monsters that are not actually extra scary.

- ▶ **Instant monster mix—just add water**

This ingredient is averagely priced, and what you generally would expect to use.

- ▶ **Snips/snails/puppy dog tails**

This ingredient is quite expensive, but it is the only choice for really strong/scary/mad/evil monsters.

How does the baron go about reconciling the opposing aims of trying to use the cheapest ingredients and still keeping his customers satisfied? He says it's really simple (everything is always obvious to the expert with 40 years of experience). Just follow these basic rules:

► **Average monsters (sanity level: 16% to 74%)**

For monsters of average sanity, you cannot ever use sugar/spice/all things nice. You can use the instant monster mix if the customer doesn't require that the monster is extra scary. However, if the customer requires that the monster *does* have to be extra scary, then you have to use the snips/snails/puppy dog tails option.

► **Extra sane monsters (sanity level: 75% and over)**

An extra sane monster can be made with any of the ingredients, so the one you choose depends on the purpose of the monster. If it is to be a ballroom dancer, then it does not need to be very monstrous at all, so you can use sugar/spice/all things nice. If, on the other hand, it is going to be a mortgage salesman, then it needs to be extra scary, so you have to use snips/snails/puppy dog tails. If it is anything of an average monstrousness level, then you can go with the instant monster mix.

► **Mad monsters (sanity level: 15% and under)**

For mad monsters, you can never use sugar/spice/all things nice. Generally, they need the snips/snails/puppy dog tails to give them the desired level of scariness. The only way you can get away with substituting the instant monster mix is if the desired evilness is only moderate and even then only if the monster is only going to fly into an uncontrollable rage once or less per day.

You can now turn these rules into a flowchart, as shown in Figure 9.3.

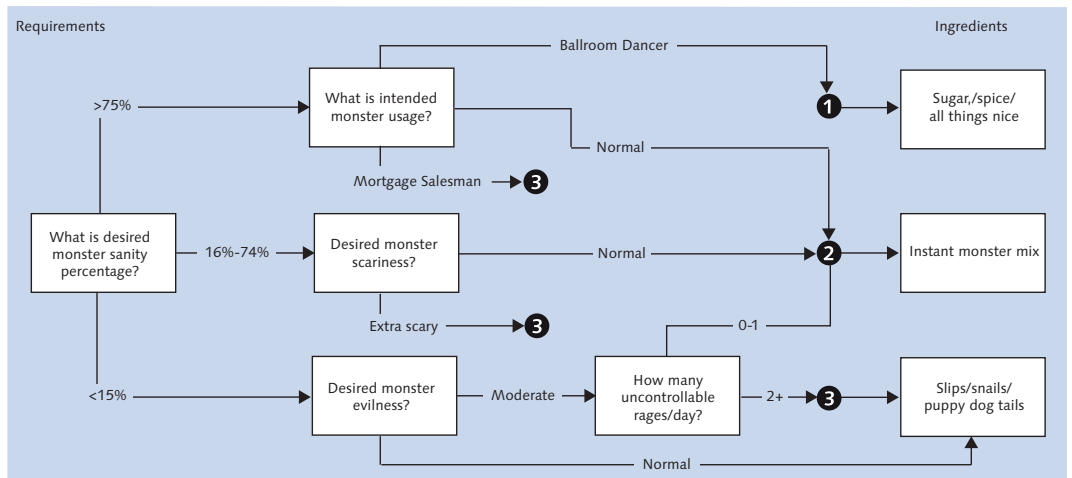


Figure 9.3 Simple Monster Rules Example

The flowchart is shown to the business experts so that they can easily visualize the proposed flow to make sure it's correct. Usually, the analyst also turns the requirement into a spreadsheet, because these days many business experts live and die by spreadsheets. The result will look something like Figure 9.4.

	A	B	C	D	E	F
1	Monster Sanity	Monster Usage	Scariness	Evilness	Rages / Day	Result
2		75 Ballroom Dancer	Any	Any	Any	Option 1
3		75 Normal	Any	Any	Any	Option 2
4		75 Mortgage Salesman	Any	Any	Any	Option 3
5		50 Any	Normal	Any	Any	Option 2
6		50 Any	Extra Scary	Any	Any	Option 3
7		10 Any	Any	Normal	Any	Option 3
8		10 Any	Any	Moderate		0 Option 2
9		10 Any	Any	Moderate		1 Option 2
10		10 Any	Any	Moderate		2 Option 3

Figure 9.4 Monster Rules Spreadsheet

The business analyst shows both the flowchart and the spreadsheet to the expert to be sure the requirements have been captured accurately. Then, it's time to work out how to put these rules into SAP.

9.1.2 Rules in Customizing Tables

When SAP was invented, what set it apart from the pack (aside from it being an integrated system, which was itself a radical idea) was that it was software that could in theory be used by any industry in any country, because you could customize it to your requirements. As you know, this is achieved by having many thousands of customizing tables sitting in the IMG, each one of which can be used as is or—more likely—with company-specific entries added to describe the way any given organization does things (e.g., route determination based on order type and material type). Even better, you can add your Z customizing tables into the actual IMG, so you see all customizing tables (standard and customer) for one area, like sales and distribution, in a single place.

You could attempt to replicate the monster logic in a customizing table, which you would design based on the spreadsheet created earlier. A possible design is shown in Figure 9.5, with five key columns and a results column. All at once, you can start to see possible problems.

The problem with SAP customizing tables is that they are designed such that you really need a value for all of the keys to do a proper database read. If you want to

get a unique record back, then you need one record for each possible percentage value, multiplied by one record for each possible monster usage, multiplied by one record for each possible scariness value, and so on. You can see how such a customizing table can quickly have a gigantic number of records. In some standard SAP tables, you have two keys, like transportation planning point and shipment costs type. If you have 300 transportation planning points and 10 shipment cost types, then you have to enter 3,000 records, and you can only cut and paste 26 records at a time from a spreadsheet into the standard IMG screen; there is no facility to upload an entire spreadsheet at once.

Table: ZMONSTER_RULE						
Displayed Fields: 7 of 7		Fixed Columns: 6		List Width 0250		
HANDT	SANITY	MONSTER_USAGE	SCARINESS	EVILNESS	RAGES_PER_DAY	INGREDIENTS
<input type="checkbox"/>	230	10			Moderate	2
<input type="checkbox"/>	230	10			Moderate	2
<input type="checkbox"/>	230	10			Moderate	3
<input type="checkbox"/>	230	10			Normal	3
<input type="checkbox"/>	230	50		Extra Scary	0	3
<input type="checkbox"/>	230	50		Normal	0	2
<input type="checkbox"/>	230	75	Ballroom Dancer		0	1
<input type="checkbox"/>	230	75	Hortgage Salesman		0	3
<input type="checkbox"/>	230	75	Normal		0	2

Figure 9.5 Monster Rules as a Customizing Table

If you do not want such a large number of entries, then there are naturally ways you can work around this. For example, instead of having one column for the sanity percentage, you could have two columns: from and to. Then, as in Figure 9.5, you can have blank values for the wild card values, those for which the actual value of the field can be anything and the record is still valid. Finally, you would have to retrieve a whole bunch of rows—for example, all the rows in which sanity is below 15%—into an internal table, and then loop through that table with your own ABAP logic to try and see which one is best.

The problem is that the more records you need to retrieve, the less efficient the process is, and the less you can take advantage of buffering. You cannot have single-record buffering if you are going to say `SELECT where X < 15`. Standard SAP tables often get around this by having a view cluster in which there are several different tables, each with a different number of keys, and you try them in sequence until you get a result. (The condition technique in pricing works like this.) However, with all these workarounds, you are spreading the complexity all around—both in the Z table and in the ABAP coding.

If the analyst decides this is really too complicated to maintain in a Z table, then the next option is to farm the rules coding out to ABAP.

9.1.3 Rules in ABAP

Programmers know that they can make ABAP do whatever they want it to do, and quickly at that. As you can see in Listing 9.1, it's not too difficult to whip up some lines of code to achieve the result demanded by the monster business rule requirement.

```

IF ld_sanity GT 75.
  CASE ld_usage.
    WHEN 'BALLROOM_DANCER'.
      ld_ingredients = 1.
    WHEN 'MORTGAGE_SALESMAN'.
      ld_ingredients = 3.
    WHEN OTHERS.
      ld_ingredients = 2.
  ENDCASE.
ELSEIF ld_sanity LE 74 AND
      ld_sanity GE 16.
  CASE ld_scariness.
    WHEN 'EXTRA_SCARY'.
      ld_ingredients = 3.
    WHEN OTHERS.
      ld_ingredients = 2.
  ENDCASE.
ELSE."Sanity is 15% or less
  CASE ld_evilness.
    WHEN 'MODERATE'.
      IF ld_rages_per_day LE 1.
        ld_ingredients = 2.
      ELSE.
        ld_ingredients = 3.
      ENDIF."Rages per Day
    WHEN OTHERS.
      ld_ingredients = 3.
  ENDCASE."Evilness
ENDIF."Sanity Percentage

ASSERT ld_ingredients NE 0.

```

Listing 9.1 Monster Rule in ABAP

Although this code is easy to write, the perceived problem is that the logic is hidden away in ABAP. From the business expert's point of view, and even from that of the poor old business analyst, this really is a black box. However, customizing tables and ABAP code are the traditional way in which business rules have been implemented since the dawn of SAP.

How dare BRFplus come along and tell us it can do this better! Look at how it handles the same task.

9.2 Creating Rules in BRFplus: Basic Example

This section walks you through the basic steps in creating rules in BRFplus, which consists of three main steps: creating an application, adding rule logic, and calling the BRFplus function in ABAP.

9.2.1 Creating a BRFplus Application

To start creating a BRFplus application, call Transaction BRFplus or BRF+ (either will do; both transactions call the exact same program). Depending on your system settings, you may be asked for your user name and password again. The first time you run the transaction you may see a whirling circle for a few minutes; afterwards, the BRFplus screen appears.

Note

The screens in BRFplus look very different depending on what version of SAP you are running. This is one product that still has a large number of enhancements coming through in each support stack.

Like Russian nesting dolls, you have to create many layers when creating an object in SAP, and the same is true here. The innermost doll is the rule, and the outermost doll is the application. You start with the outside—the application—so click the **CREATE APPLICATION** button. The screen shown in Figure 9.6 appears.

In Figure 9.6, you can see the screen for creating an application, with all the standard fields: **NAME**, **SHORT TEXT**, and **TEXT**. Before version 7.4 you did not have to start the name of applications with "Z" in this framework; you still don't have to in 7.40, but you get a warning if you don't. When you get deeper inside of BRFplus it's best to have some sort of naming convention so that you can tell different types of objects apart by looking at their technical names.

In the **STORAGE TYPE** field, you have three options: **MASTER DATA**, which lives in the current system and does not get transported, **CUSTOMIZING**, which is client-dependent, and **SYSTEM**, which is like a workbench object.

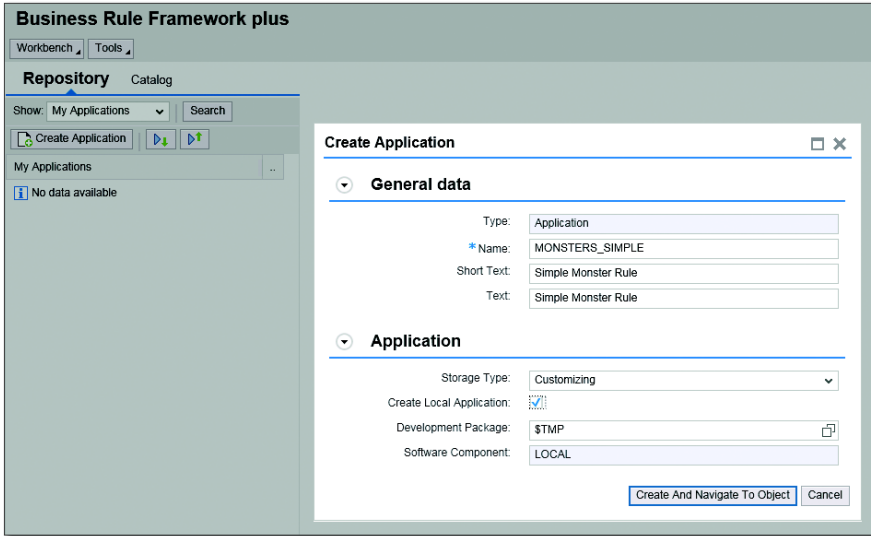


Figure 9.6 Creating a BRFplus Application

In the next box down, select the CREATE LOCAL APPLICATION box, because this is an example not intended for productive use. When creating a rule you do intend to use in the live system, you would have to nominate a development package, and a transport would be created in the normal way so that you could move your changes between systems.

Click the CREATE AND NAVIGATE TO OBJECT button, and the screen shown in Figure 9.7 appears.

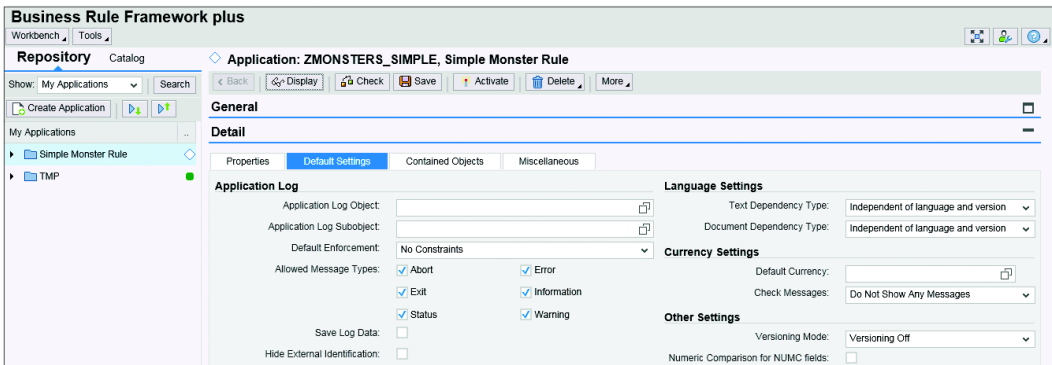


Figure 9.7 BRFplus Application Default Settings

Figure 9.7 shows the screen that controls the application settings. As you can see, the DEFAULT SETTINGS tab shows the application log settings. One way BRFPplus seeks to avoid being a black box is by recording how decisions are made inside the application log so that someone can go back after the event and perform a postmortem on real decisions in the live system.

To create the next layer in your application, a *function*, go to the CONTAINED OBJECTS tab, which you can see in the background of the screen in Figure 9.8 (the gray area behind the pop up box). (You can also create objects such as functions directly from the context menu of the tree on the left side.) A function is what you're going to be calling from within your custom ABAP code to pass in some values and get a result back based on the rule(s) that live inside the function; an application can have one or many functions. Select FUNCTION from the dropdown list in the top left of the CONTAINED OBJECTS tab, and then click the CREATE OBJECT button. The pop up box shown in Figure 9.8 appears.

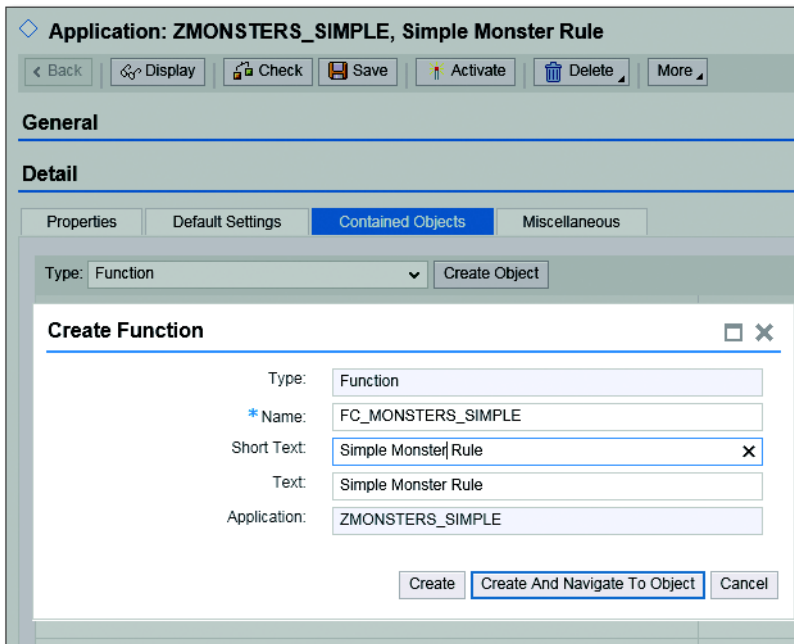


Figure 9.8 Creating a Function

In Figure 9.8, you can see the box for creating a function. The names can be anything you want, but it's good to add a two-digit prefix to remind you later what

sort of object is what—for example, the prefix FC for functions. After entering this information, click **CREATE AND NAVIGATE TO OBJECT**, and the screen shown in Figure 9.9 appears. (If you click **CREATE** instead, then a blank object will be created, and you will have to navigate to it yourself to make the required settings.)

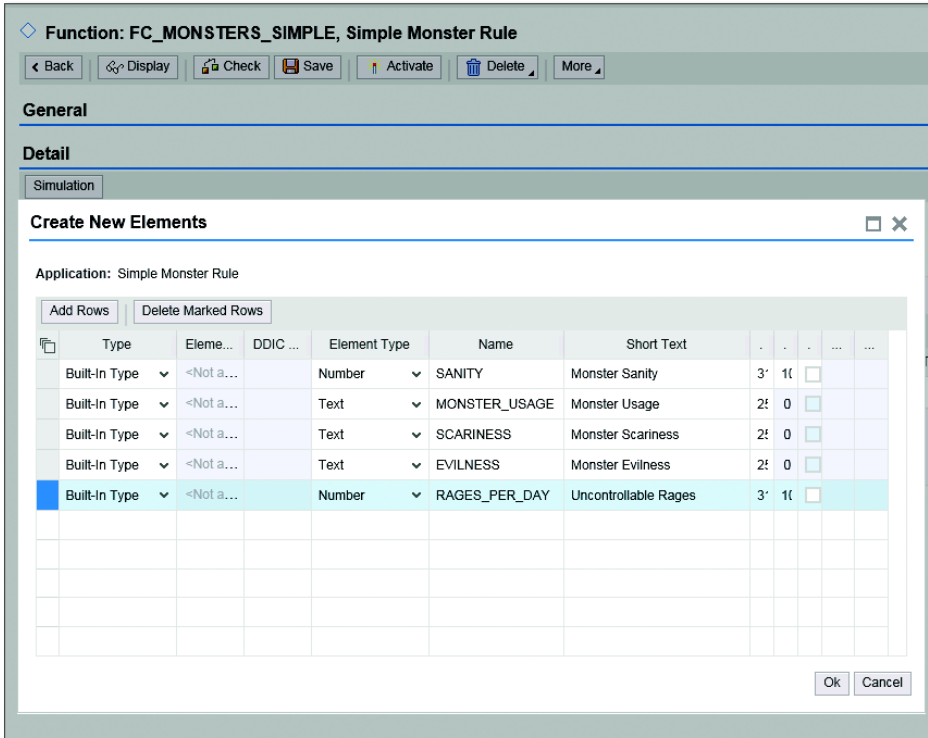


Figure 9.9 Adding Input Parameters to a Function

In this function, you're going to be passing in the customer requirements: what the monster is going to be used for, how scary it needs to be, how many rages it has to fly into each day, and so on. Based on the rules you'll add to the function, a result will come back saying what ingredients to use.

To enable such passing of data backwards and forwards in your new function, you need to create the signature—that is, what data you expect in and what data you're returning. In regard to parameters, a BRFplus function is like a cross between a function module and a method; the name says “function,” but the import and export parameters can be tables, as is usually only possible in methods.

To create such a signature, navigate to the SIGNATURE tab in the function screen shown in Figure 9.10. Go to the CONTEXT area (which means input data) at the top of the screen, and click the ADD NEW DATA OBJECT button. You are then asked if you want to add one at a time, but because you have five input fields for the various customer requirements (scariness and so forth) you'll want to add them all at once.

For each input parameter you can specify a built-in type, like a number, but you can also have an input parameter referring to an existing DDIC data element if you want. Once you have added your data, the parameters are displayed in alphabetical order (which is a pity, because it seems like logical order would be better; however, it does make searching easier).

Once you've finished with the context (input data), it's time to turn to the RESULT DATA OBJECT. As can be seen in Figure 9.10, this object lives just below the CONTEXT area. This is the object that the BRFplus function will return to the calling ABAP program to tell it what the outcome of the rule evaluation has been.

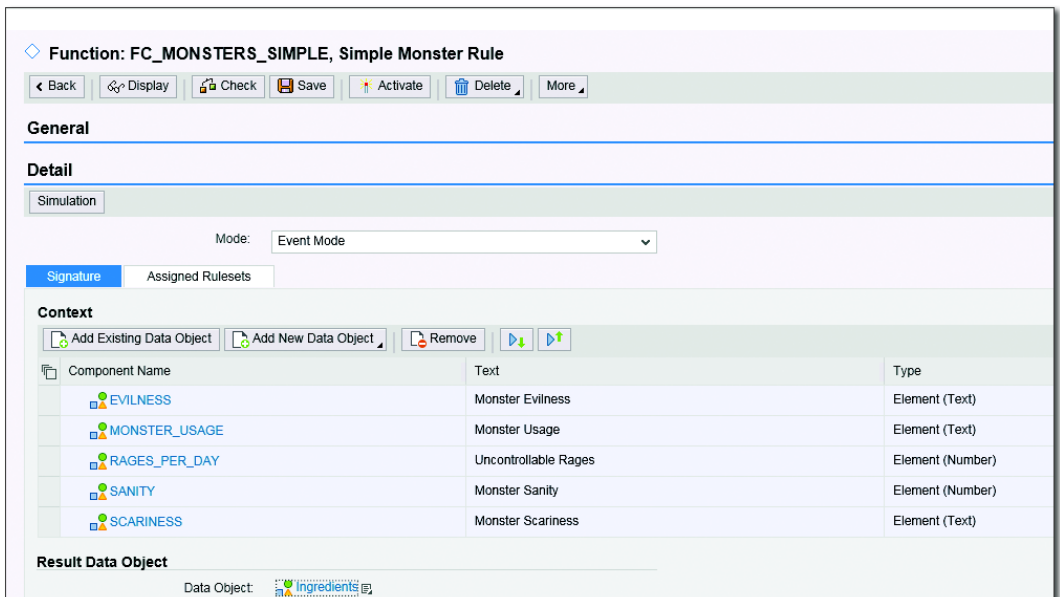


Figure 9.10 Full Signature of a Function

Click the action button to the right of DATA OBJECT, which is INGREDIENTS in our example. Now click CREATE and then specify what form the result should take (the type of ingredients, in this example) in exactly the same way that you specified the data type of the input parameters.

Here, there is only one result allowed, so if you have multiple values coming back (the exception rather than the rule, you would think), then you have to define the result object as a structure or a table rather than a data element. Do this by defining a structure or a table type using SE11, and then set the type of the result data object to be that DDIC structure or table. In Figure 9.10, you'll see the completed signature—input and output parameters—of your function; i.e., everything below `CONTEXT`.

Now that you have a means to receive some data and send back a result, it's time to move on to the next level down, which is a ruleset. You could ask me what a ruleset is, and I could answer that it's a set of rules, but probably that would not make you any better off. The best way of answering is by way of an example. In this example, one of the monsters has been arrested for not killing enough peasants. The sentence is destruction. The baron launches a legal appeal based on the laws of Transylvania—an enormous set of legal rules built up over many centuries.

The judge cannot come to a decision, so the case is referred to the court of the European Union, where another judge will consider the matter based on European laws. That judge cannot come to a decision either, so the case is referred to the United Nations to be tried under international law. This time, a decision is made, and the monster is released on the condition that it does better next time.

In this case, the first (Transylvanian) set of rules cannot come back with a result, so another attempt is made with a different (European) set of rules, which also produces no result, and so a third (international) set of rules is invoked, which does produce a result. In every case, the input data—the facts of the case—was the same.

The BRFplus equivalent of our real-life example is that a BRFplus function can have many rulesets, and each ruleset can have one or many rules; the idea is to loop through the rulesets until you get a result. If a certain set of rules is unable to make a decision, then instead of giving up you can try a totally different strategy, by passing a new set of rules the same input data to see if they can do a better job.

Just to keep this simple for the moment, there will be one ruleset with one rule. Both the ruleset and the rule have the simple job of determining what type of ingredients to use for a monster based on the customer requirements. In order to create a ruleset, navigate from the `SIGNATURE` tab of the function to the `ASSIGNED RULESETS` tab; the screen will now look like Figure 9.11.

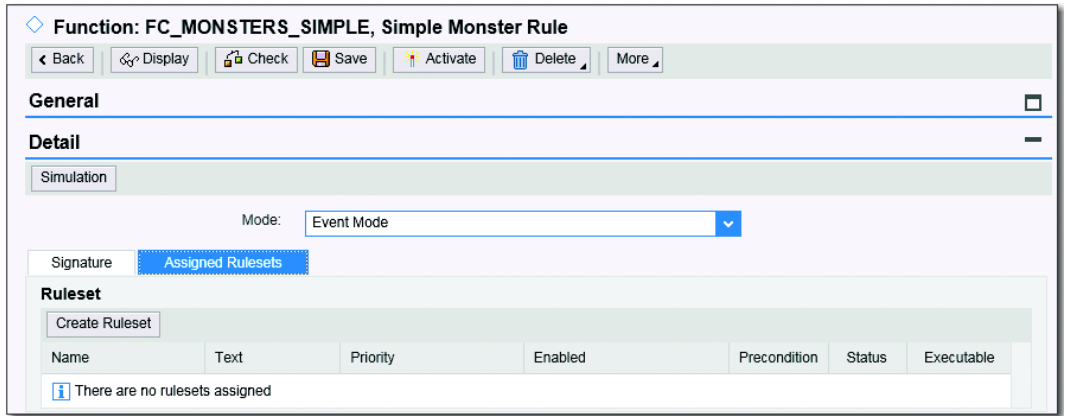


Figure 9.11 Screen Showing Rulesets Assigned to a Function

Click CREATE RULESET on the ASSIGNED RULESETS tab, and a pop-up like the one shown in Figure 9.12 appears.

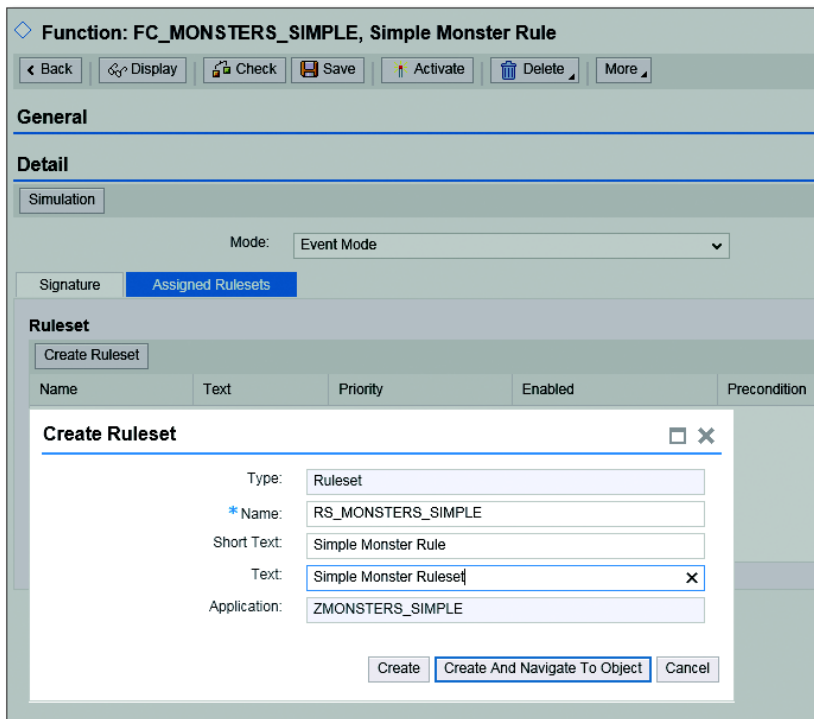


Figure 9.12 Creating a Ruleset

In the pop-up shown in Figure 9.12, you define the name of the ruleset. You will note the “RS_” (for “ruleset”) prefix to the technical name, but you can specify any name you desire. Click **CREATE AND NAVIGATE TO OBJECT**, and the screen shown in Figure 9.13 appears.

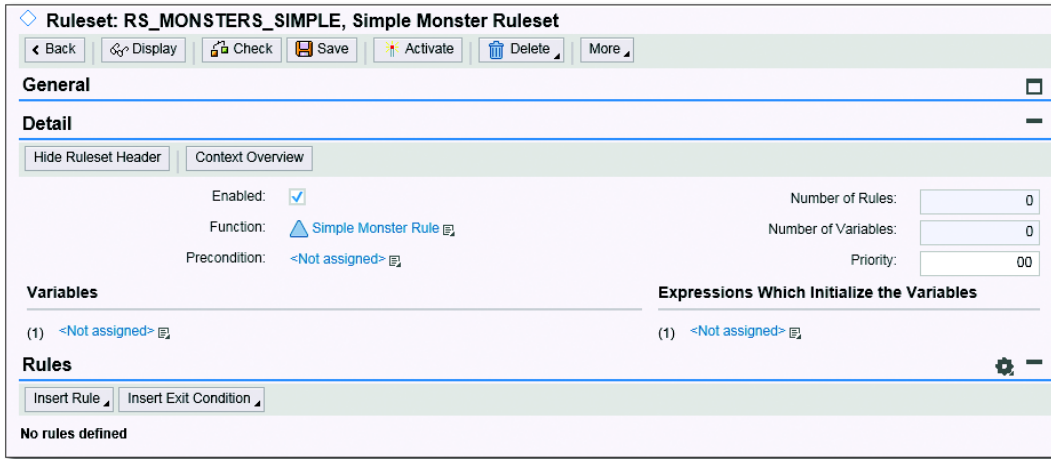


Figure 9.13 Ruleset Definition Screen

In the middle of Figure 9.13, you will see a **VARIABLES** section; this is for defining variables that are used internally within the ruleset and are not part of the function signature. Function context (the input variables), function result (the output variable), and ruleset variables (which you might think of as variables local to the ruleset) make up the execution context of an expression or a rule. As soon as a rule or expression is assigned to a function, those data objects can be provided for use in the rule.

The ruleset definition screen shown in Figure 9.13 has a list of rules at the bottom. Naturally, the list will be empty when you start, so use the dropdown on the **INSERT RULE** button, and choose the **CREATE** option. The screen shown in Figure 9.14 appears.

In this screen, you'll see what looks like an **IF/THEN/ELSE** statement in ABAP. Use the dropdown on the **ADD** text just after the **THEN** statement, and choose **PROCESS EXPRESSION** from the resulting context menu. Hooray! You've reached your destination.

Figure 9.14 Creating a Rule

9.2.2 Adding Rule Logic

Now that the basic BRFplus application has been created, you can start to define the rule logic. Rule logic—in this case, how you choose what ingredients to use—is defined by using an *expression type*, which refers to a generic type of method to make a decision. As an example of such a decision-making method, you could write all the possible choices for a decision you have to make on pieces of paper, stick them on a dartboard, get a monkey to throw darts at the board, and choose the outcome relating to the choice the monkey's dart hit. (This is in fact how most corporate decisions in Fortune 500 companies are made.) You could also flip a coin or hire an astrologer, who would slaughter a goat and look at the entrails to look for signs as to what decision you should make. (This is how the remaining decisions in such large corporations get made.)

In this case, there is a computerized business rules management system available to use—that is, BRFplus—and it provides you with over 20 possible ways to make decisions (expression types), none of them involving monkeys or astrologers. Next, we will look at the two most commonly used expressions in BRFplus, a

decision tree and a decision table, which are probably the most commonly used because when a business user looks at them they are easy to understand.

Decision Tree Rules in BRFplus

A decision tree is basically a series of yes/no questions to answer; after you answer each question, you either get a result or are asked another yes/no question. When you try and draw all the possible routes someone could take through the questions on a piece of paper, it comes out looking a bit like a tree, because it has so many branches.

In this example, you will create just such a structure inside BRFplus. To get started, refer back to the screen shown in Figure 9.14, click ADD, and take the PROCESS EXPRESSION option. The screen shown in Figure 9.15 appears.

The screenshot shows the 'Create Object' dialog box with the following configuration:

- General data:**
 - Type: Decision Tree
 - * Name: DT_MONSTERS_RULE
 - Short Text: Simple Monster Rule
 - Text: Simple Monster Rule
 - Application: ZMONSTERS_SIMPLE
 - Is Reusable:
- Result:**
 - Result Name: INGREDIENTS
 - Text: Ingredients
- Possible Result Data Objects:**

Object	Text	Type
MONSTER_USAGE	Monster Usage	Element
SCARINESS	Monster Scariness	Element
EVILNESS	Monster Evilness	Element
RAGES_PER_DAY	Uncontrollable Rages	Element
INGREDIENTS	Ingredients	Element

At the bottom, there is a checkbox for 'Expression Triggers Actions' and three buttons: 'Create', 'Create And Navigate To Object', and 'Cancel'.

Figure 9.15 Creating a Decision Tree

At the top of the screen shown in Figure 9.15 is a TYPE dropdown box, in which you choose from a vast list of possible expression types. In this case, choose DECISION TREE, and give your tree a name and description. This example starts the name with "DT_" for "decision tree."

Moving down the screen, you'll see a big list of POSSIBLE RESULT DATA OBJECTS, which are all the elements from the function signature. You select the one you want (in this case, INGREDIENTS) and the selected element is copied to the middle of the RESULT section the instant you select the row. You can change your mind after you have selected a row, but you can only have one result element.

Click CREATE AND NAVIGATE TO OBJECT, and the screen shown in Figure 9.16 appears.

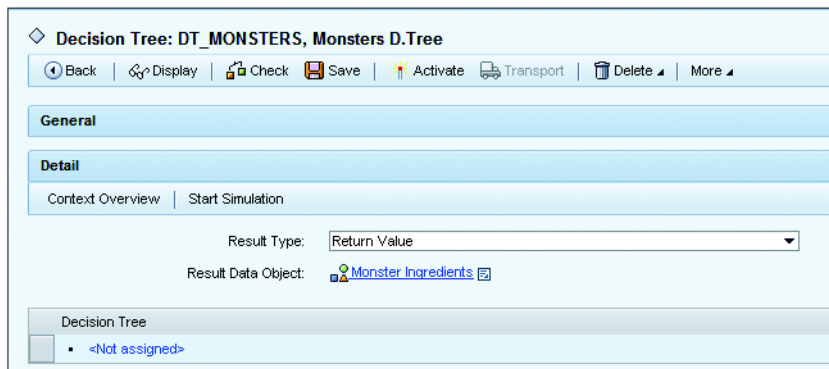


Figure 9.16 Creating a Decision Tree: Empty Screen

The area in the lower half of the screen, under the words DECISION TREE, is blank apart from a top-level tree node that is just a blank icon; this is to be expected, because you haven't entered anything yet. Now it's time to start building up what is effectively the same sort of convoluted IF/THEN statement you would code in ABAP. For each step in the process, you create a TRUE/FALSE branch, represented by green checkmarks and red Xs. First, right-click on the top-level blank node, and follow the menu path SET CONDITION • DIRECT INPUT • NOT ASSIGNED • SELECT CONTEXT PARAMETER. Then, you'll see a list of possible variables you can use as input parameters (this list of variables is the list of customer requirements that are being sent into the BRFplus function: desired scariness, sanity, monster usage, and so on). To start off, choose RAGES PER DAY as the input variable you want, and the screen that asks you to input a logical condition appears (Figure 9.17).

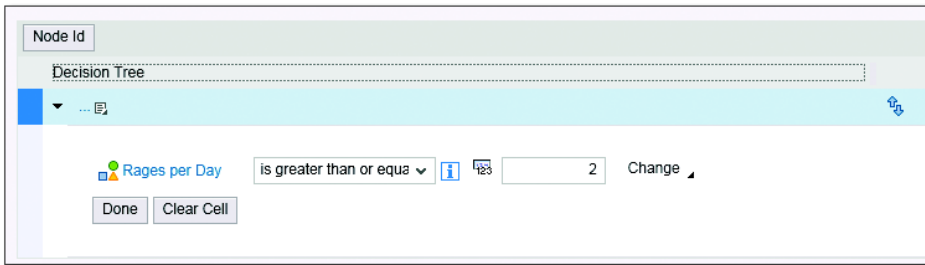


Figure 9.17 Adding a Decision Tree Step

Enter a condition, such as “rages per day are greater or equal to 2,” and then click DONE. The `Not Assigned` node has changed into a line showing the logical condition you just entered, followed by a green checkmark and red X (i.e., green checkmark if the rages are greater than or equal to two and a red X otherwise; see Figure 9.18).

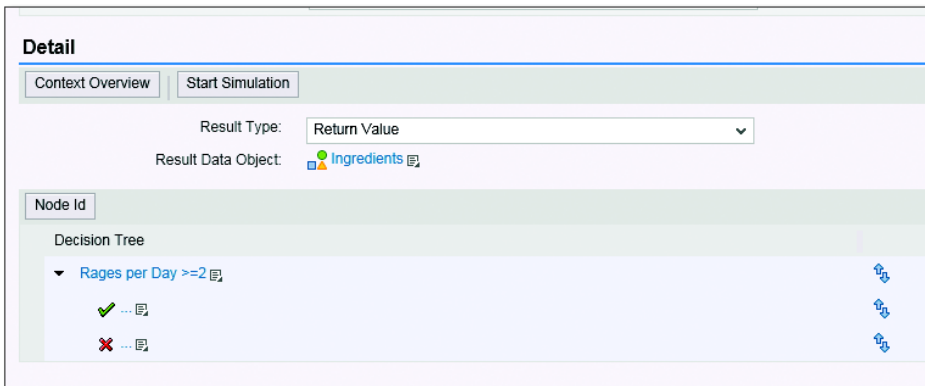


Figure 9.18 Creating the First Decision Tree Node

Now, repeat the process by selecting a `Not Assigned` node. You can input another logical condition in the same way as before, or if you want to end the tree structure at a certain node (i.e., you’re in a position to make a decision), then you can follow the menu path `SET RESULT • DIRECT INPUT` and enter the return value (type of ingredients in this case).

Repeat the process again and again until every `Not Assigned` node has been changed into either a logical condition or an end result. When you’re finished, the result looks like Figure 9.19.

Figure 9.19 Completed Decision Tree

At the top of the screen in Figure 9.19 is a DOCUMENTATION tab. Function modules and methods can have documentation stored inside SAP (not often used), but here you can see the documentation and the actual decision tree in the same screen.

In real life, instead of the numbers one to three, you would probably want to have text names for the result; you want to avoid someone looking at the number 2 and having no idea what it means. In SAP NetWeaver 7.31 and above, the text of a value is shown at the place of usage (e.g., in a decision tree) whenever it is possible to derive a text value (e.g., from a domain or value table).

Decision Tree Look and Feel

If you look at the same decision tree before and after upgrading from SAP NetWeaver 7.02 to 7.31/7.4, then you will see it looks quite different. In 7.02, there are three different icons: a blue question mark along with the green checkmarks and red Xs. There

is also a lot of text on each line saying "If this is true, then the result is" instead of just showing the result variable.

SAP had a usability expert look into this, and the recommendation was to use only two icons—the green checkmark and the red X—and lose the text explanations. This was intended to make the display more readable and user-friendly.

Whether that idea was good or bad I will leave you to decide for yourselves!

The idea is that when a decision tree is looked at by a business expert, that expert should be able to tell if the logic is right or wrong. The decision tree is not as clear as a flowchart and in fact takes some getting used to; you have to work out that each checkmark or X represents either a result or another IF statement, which is not immediately obvious. However, it's supposed to be a lot clearer than a mass of IF and CASE statements hiding away in ABAP code.

Decision Table Rules in BRFplus

A decision table is a way of making a decision based on a grid of information. Such a grid looks rather like a spreadsheet, with the columns on the left being the input data and one or more columns on the right being the decision result.

In the previous example of a decision tree, you will always get a result. However, in order to demonstrate a decision table, pretend that it's possible that the decision tree rule might fail to return a proposal for the ingredients, rather like the judge in the earlier example (the monster on trial for not being monstrous enough) being unable to reach a decision and invoking a higher court. At the moment you only have one rule in your monster ruleset—the rule that processes the decision tree. If that rule were to fail you need to add another rule to cater for the possibility of the first rule failing. You can use any expression at all for this new rule, but here you will be using a decision table.

Now you will add another rule to the ruleset, and this time there will be a precondition for the new rule: only process the new expression if the MONSTER INGREDIENTS option is empty (Figure 9.20). In other words, only process the new rule if the decision tree rule failed.

To create the decision table rule as shown in Figure 9.20, open the dropdown menu of the ADD box to the right of the THEN statement and choose PROCESS EXPRESSION • CREATE. This time, you will create a decision table, so when the screen shown in Figure 9.20 appears, use the dropdown at the top to select DECISION TABLE.

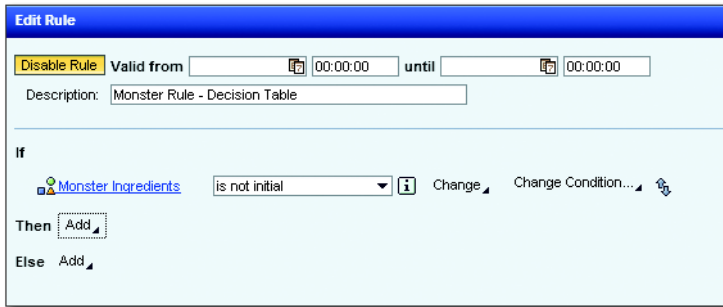


Figure 9.20 Adding a Rule Precondition

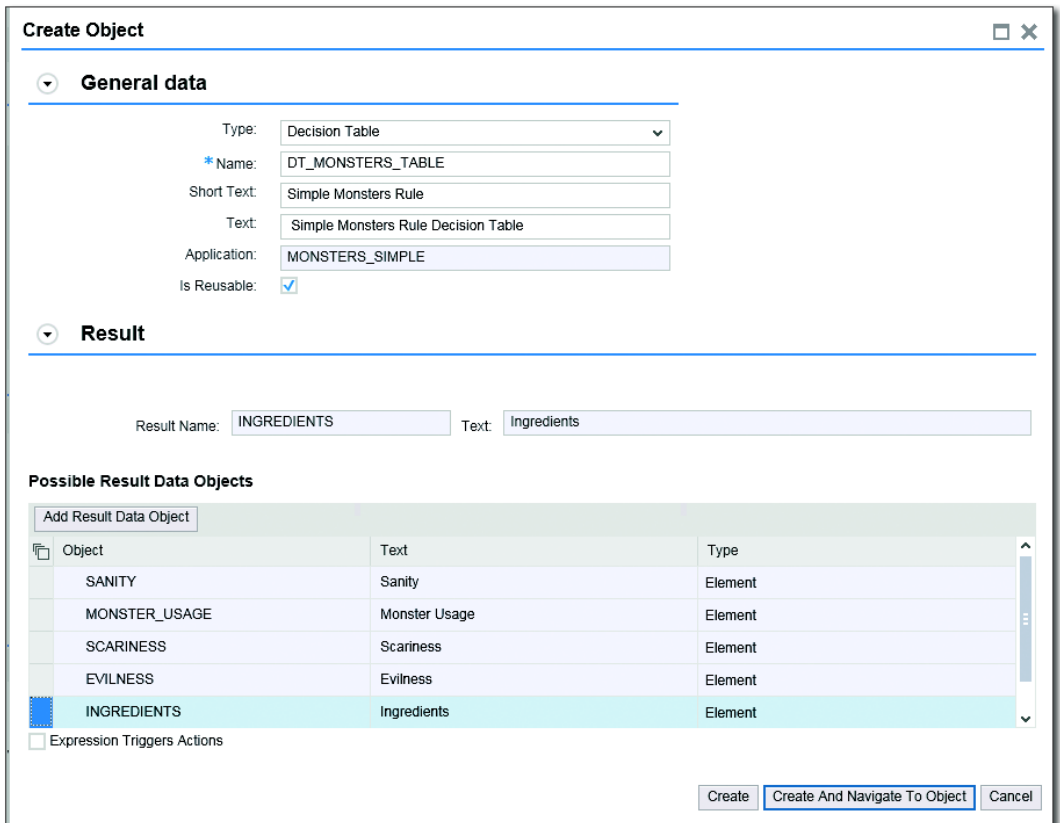


Figure 9.21 Creating a Decision Table

At the bottom of the screen shown in Figure 9.21, you'll see a list of fields that you could use as result parameters. Choose the good old INGREDIENTS parameter, because that's the result of the decision you want to make—that is, what type of ingredients you should use to make your monster based on the customer requirements. Click CREATE AND NAVIGATE TO OBJECT.

You'll see the decision table screen, which is alarmingly blank (the grayish section in the background of Figure 9.22). A table cannot hold its head up in polite society until it has some columns, so you had better add some in a hurry. The input columns you want are going to relate to the customer requirements: desired monster sanity, desired scariness, and so on. To achieve this, click the TABLE SETTINGS button, and a pop-up box appears in which you can add your good old columns (Figure 9.22).

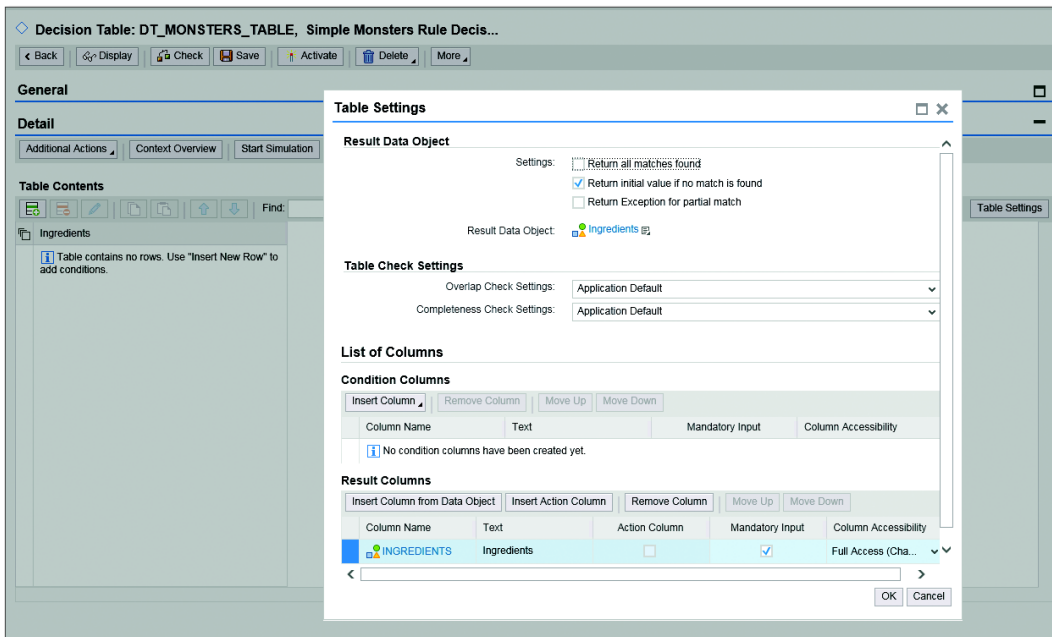


Figure 9.22 Empty Table Settings Pop-Up

On the screen shown in Figure 9.22, go to the area with the title **CONDITION COLUMNS**, and follow the menu option **INSERT COLUMN • FROM CONTEXT DATA OBJECTS**. A list of fields appears (Figure 9.23).

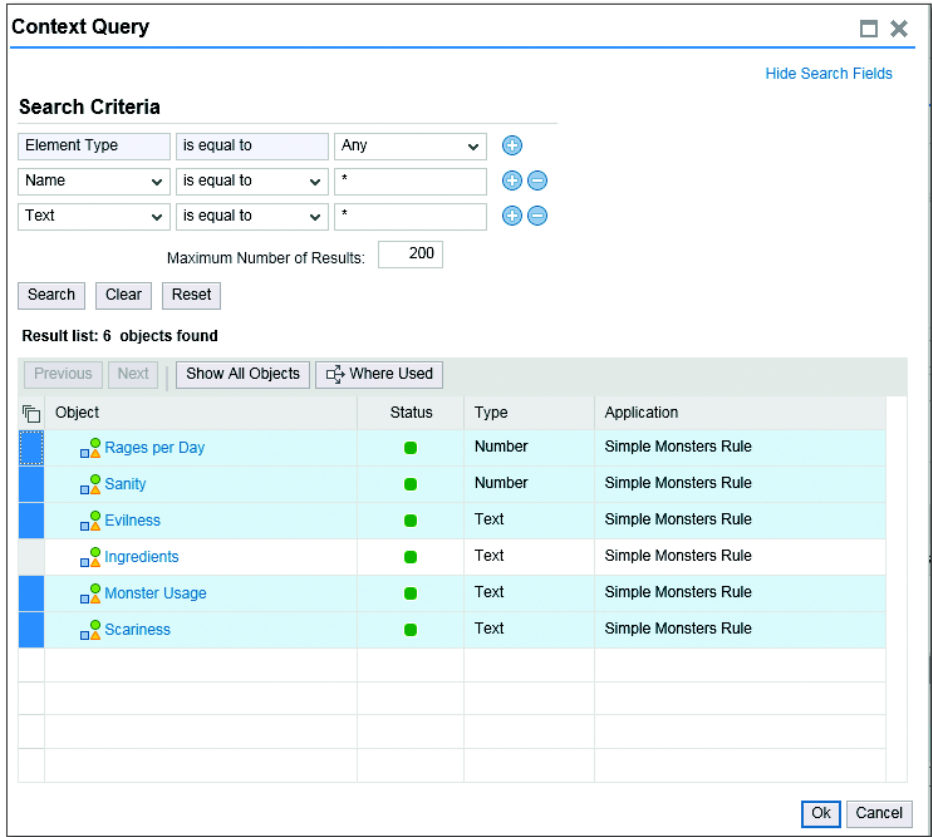


Figure 9.23 Adding Column Fields

These fields are available for your use because they are defined in the function as either context or result variables. Select the ones you want; in this case, you want all the customer requirements, which form the input parameters of your function, but naturally not the output parameters (ingredients). Click OK, and the screen shown in Figure 9.24 appears.

Figure 9.24 shows the TABLE SETTINGS box with all the columns added that you just chose. The little man who lives inside the computer clearly knows what order you want them in; you didn't specify an order when you selected the fields, but on the result screen they're shown in the same order as the data dictionary table you created earlier. Because you didn't actually specify a link to table DDIC and—amazingly—some people claim there is no little man inside the computer,

another explanation could be that the columns were inserted in the order you set them up in the decision tree. Or, it could be that the sequence of the creation resulted in sortable UUID keys, which then make the fields show up in this order. (My money is on the little man.)

Table Settings

Result Data Object

Settings: Return all matches found
 Return initial value if no match is found
 Return Exception for partial match

Result Data Object: ■ Ingredients

Table Check Settings

Overlap Check Settings: Application Default

Completeness Check Settings: Application Default

List of Columns

Condition Columns

Insert Column Remove Column Move Up Move Down

Column Name	Text	Mandatory Input	Column Accessibility
■ SANITY	Sanity	<input type="checkbox"/>	Full Access (Changes A... ▼
■ MONSTER_USAGE	Monster Usage	<input type="checkbox"/>	Full Access (Changes A... ▼
■ SCARINESS	Scariness	<input type="checkbox"/>	Full Access (Changes A... ▼
■ EVILNESS	Evilness	<input type="checkbox"/>	Full Access (Changes A... ▼
■ RAGES_PER_DAY	Rages per Day	<input type="checkbox"/>	Full Access (Changes A... ▼

Result Columns

Insert Column from Data Object Insert Action Column Remove Column Move Up Move Down

Column Name	Text	Action Column	Mandatory Input	Column Accessibility
■ INGREDIENTS	Ingredients	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Full Access (Cha... ▼

< _____ >

OK Cancel

Figure 9.24 Final Table Settings

On this screen, pay attention to the COLUMN ACCESSIBILITY setting, in which you can have hidden columns and read-only columns. Hidden columns could be used for calculations, and read-only columns could display the text name of the value to its left; for example, one column could display the sales office value, and the read-only column could display the text description of the sales office. Click OK,

which moves you to the DECISION TABLE screen (Figure 9.25) and then press the INSERT ROW button (the leftmost icon under TABLE CONTENTS).

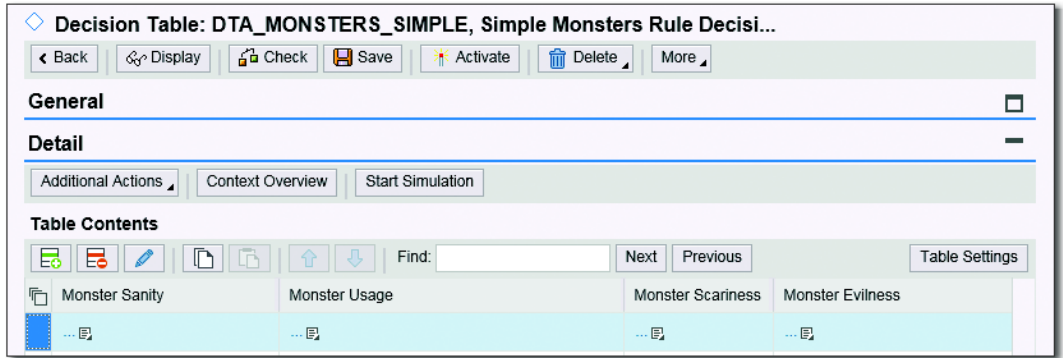


Figure 9.25 Empty Decision Table

To start off, you only have one row, which has a little icon with a few dots before it in each cell. You now have two options: Enter the values in each cell individually, or upload a whole table from Microsoft Excel. The latter is achieved by opening the ADDITIONAL ACTIONS dropdown and choosing either IMPORT FROM EXCEL or EXPORT TO EXCEL (you can either upload an Excel spreadsheet into your decision table in BRFplus or download the contents of the table into Excel).

As noted earlier, often the business analyst creates a spreadsheet to store the rules during the analysis phase, or even receives a spreadsheet directly from the business expert in the first place. In some cases, the process is then as simple as defining an identical structure in the BRFplus decision table, uploading the spreadsheet, and then going back to the expert and saying, “Here is the rule inside SAP. Does it look like that spreadsheet you gave me ten minutes ago?”

In this case, imagine that you don’t want to upload the spreadsheet you were given, because it wasn’t a very good representation of the rules; you would have to have thousands of rows to capture every combination. Instead, click the INSERT ROW icon ((the leftmost icon under TABLE CONTENTS), and in the SANITY column you have two choices. You could click on the square icon to the right of the dots, and a context menu would appear, from which you would choose the option DIRECT VALUE INPUT to enter a formula. Alternatively, to save yourself a few button clicks, just click on the three dots you can see in each cell, and then you can enter a formula without having to choose an option from a menu. In either case, the screen shown in Figure 9.26 appears.



Figure 9.26 Adding Formulas to the Decision Table

In Figure 9.26, you're adding a formula that says that this row is only valid if the sanity is 75% or higher; this information comes straight from the business rules provided by the baron, which state that monsters with a very high sanity level need different ingredients than other monsters.

You just cannot do that in a normal DDIC configuration table—that is, have a formula in a cell. Moreover, if you use the `CREATE EXPRESSION` context menu option, then you can embed another BRFplus object in the cell where you pass in the input value (such as monster usage) and get a true or false value back. The actual value that goes in the cell can be determined by another decision table, a decision tree, a formula, a call to an ABAP function module (remember that you're not in a separate system, so you can call function modules from BRFplus), or any of the 20 or so possible rule definition types in BRFplus.

You can also just enter a direct value by using the `IS EQUAL TO` option; for data objects (columns) linked to DDIC data elements, there is the standard `F4` dropdown for possible entries.

After you've made an entry in every cell, the final result looks like Figure 9.27.

At runtime, BRFplus goes through the decision table one line at a time, looking at each cell. If a cell is empty, then it equates to `TRUE`; otherwise, the formula in the cell is checked, and if it comes out as `FALSE`, then the line as a whole is disregarded. The first line where every cell comes out as `TRUE` is chosen and the result sent back.

Say that the customer wants a monster with a sanity of 30% that is extra scary. BRFplus will start working through the table from the top and will disregard the first four rows, because the first two only fire if the desired sanity is greater than 74%, and the next two only fire if the desired sanity is less than 16%.

The fifth row is chosen, because every cell evaluates to `TRUE`, and a result of ingredient type 3 (snips/snails/puppy dog tails) is returned. If the monster was required to be only averagely scary, then the fifth row would have evaluated to

FALSE because of the requirement in the SCARINESS column, and the sixth row would have been chosen, because every cell is blank and therefore TRUE.

Monster Sanity	Monster Usage	Monster Scariness	Monster Evilness	Rages Per Day	Ingredients
>=75	BALLROOM_DANCER	1
>=75	MORTGAGE_SALESMAN	3
>=75	2
<=15	MODERATE	>=2	3
<=15	MODERATE	...	2
...	...	EXTRA_SCARY	3
...	2

Figure 9.27 Complete Decision Table

When you set up the decision table in the first place, you were asked what reaction you wanted if no result was found; you can also set this to return every single row that evaluates as TRUE. Say that you had set the table to give you a list of every row that was TRUE. In the preceding example with the extra scary monster with a sanity of 30%, the fifth row would have been returned, because every cell evaluates as TRUE—and so would the sixth row, because every cell there evaluates as TRUE as well. So the result would be a table of two lines. This makes no sense in this example, which is trying to be serious, after all, but you could imagine a situation in which there are rules to see if a monster of a particular color is capable of dancing the foxtrot, for example. In that case, you would create a decision table in which the inputs are the customer requirements, and the results would be a table of the colors of monsters that can dance the foxtrot—for example, green and orange ones can, but sky blue and pink ones cannot.

Note

When you download this to Excel, the formulas come out as "=>75". Upload it again with changed values; BRFPplus will know how to interpret the value.

You're finished now; it's just a question of checking and activating each of the objects you've created. Every screen in which you create or change an object will

have a CHECK button at the top, with the scales icon you know and love from all over the SAP system—that is, the same icon you see in SE38 for performing a syntax check. If something is wrong, you'll see a warning or error message at the top of the screen.

In the same way, at the top of the screen for defining objects in BRFplus, there is always an ACTIVATE icon, which looks like a matchstick (again, the same as in SE38 and with the same purpose). Just like in SE38, the ACTIVATE icon does a check first; if there are any problems, you get error messages instead of the object being activated.

9.2.3 BRFplus Rules in ABAP

The last step in the process of creating rules in BRFplus is to call the BRFplus function in ABAP. As it turns out, there is not a great deal involved in calling the BRFplus function; as a programmer, your main responsibility is making sure that you pass suitable variables to and from the function signature. The code for this is shown in Listing 9.2.

```

CONSTANTS: gc_ingredients_dermination TYPE fdt_uuid
VALUE '005056B92F471ED3BEEDEC044A8177D1'.

DATA: ld_sanity          TYPE zde_monster_sanity VALUE 80,
      ld_usage          TYPE zde_monster_usage  VALUE 'BALLROOM_DANCER',
      ld_evilness       TYPE zde_monster_evilness,
      ld_scariness      TYPE zde_monster_scariness,
      ld_rages_pers_day TYPE zde_monster_rages_per_day,
      ld_ingredients    TYPE zde_monster_ingredients,
      "BRFplus Definitions
      lo_function       TYPE REF TO if_fdt_function,
      lo_context        TYPE REF TO if_fdt_context,
      lo_result         TYPE REF TO if_fdt_result,
      lx_fdt            TYPE REF TO cx_fdt.

FIELD-SYMBOLS: <ls_message> TYPE if_fdt_types=>s_message.

TRY.
  "Say which function we want
  lo_function = cl_fdt_factory=>if_fdt_factory~get_instance( )->
    get_function( iv_id = gc_ingredients_dermination ).
  "Get structure of function signature
  lo_context = lo_function->get_process_context( ).
  "Fill up input parameters
  lo_context->set_value( :
  iv_name = 'SANITY'      ia_value = ld_sanity ),

```

```

iv_name = 'MONSTER_USAGE' ia_value = ld_usage ),
iv_name = 'EVILNESS'      ia_value = ld_evilness ),
iv_name = 'SCARINESS'     ia_value = ld_scariness ),
iv_name = 'RAGES_PER_DAY' ia_value = ld_rages_pers_day ).

"Off we go!
lo_function->process( EXPORTING io_context = lo_context
                    IMPORTING eo_result = lo_result ).

"Retrieve Result
lo_result->get_value( IMPORTING ea_value = ld_ingredients ).

CATCH cx_fdt INTO lx_fdt.
LOOP AT lx_fdt->mt_message ASSIGNING <ls_message>.
  MESSAGE <ls_message>-text TYPE 'I'.
ENDLOOP.
ENDTRY.

```

Listing 9.2 Calling a BRFplus Function

The only thing you will find puzzling in Listing 9.2 is the really long hexadecimal string at the start. What's that all about? You've been used to calling function modules and methods with names like `Z_GET_MONSTERS` or `ZCL_MONSTERS->GET_MONSTERS()`—that is, names that are friendly to someone reading the program. However, when you use BRFplus, you need to address the function via a unique, 32-digit ID, which is generated for you when the function is created; you can see this number in the ID field of Figure 9.28.

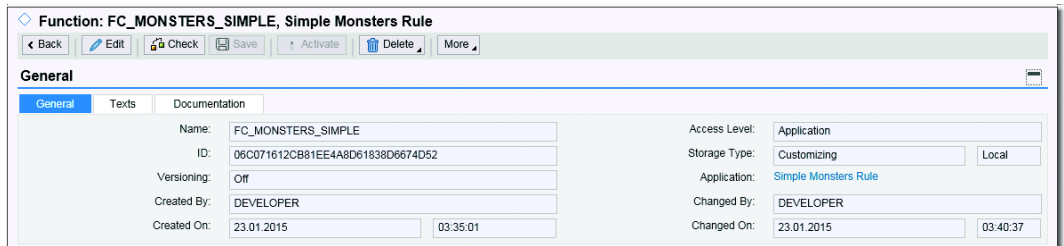


Figure 9.28 Finding the UID for a BRFplus Function

This is rather like when you phone a bank to find your balance, and a machine guides you through 32 steps, and you have to choose an option each time before hearing, "Sorry this service is offline at the moment." I did a quick survey to find out who thought that putting a meaningless 32-digit code in the middle of a program was a good idea, and I only found one person who said that it was—and then he told me to leave him alone, because he claimed to be Napoleon and was busy plotting how to defeat the English at the Battle of Trafalgar.

In case you don't think you are a historical French emperor, you might want to use a constant to turn the UID into a human-readable name. You can wrap the call to BRFplus in a method with the same signature as the BRFplus function and then have the UID as a static constant of the class that owns this method. (In case you can't tell, I am almost fanatic about having code read as close to written English as possible.)

9.3 Creating Rules in BRFplus: Complicated Example

Section 9.2 walked you through the process of creating some basic rules in BRFplus—but life is never simple, right? Inevitably, you'll find that more complicated business logic is required. This section will walk you through an example of exactly this.

Section 9.2 defined the three possible sets of monster ingredients: sugar/spice/all things nice, instant monster mix, and snips/snails/puppy dog tails. If you choose the instant monster mix, there's only one ingredient, but in the other two cases there are three possible ingredients, and you have to determine the split between them—for example, 40% sugar, 35% spice, and 25% all things nice. How do you decide what the correct split should be? This answer depends on the exact configuration of the monster, and it's the focus of this section.

Note

The example in this section is based on a real-life example. Although the product in question—that is, monsters—has obviously been changed, the basic idea of the example remains true to an actual situation.

When ordering their monsters, the baron's customers enter their monster requirements into an online configurator, rather like the one you go through when ordering a BMW, and those requirements feed into the Variant Configuration module of SAP. The online configurator asks for the following information:

► **Region**

This is a mandatory entry: the region the customer lives in. This is important, because some of our ingredients (e.g., snails) are hard to come by in some areas (e.g., France—because they've all been eaten).

► **Brain size**

Also mandatory: the customer has to choose among small, medium, and large.

- ▶ **Color**
Also mandatory: most of the time, the customer chooses green.
- ▶ **Monster model**
Also mandatory: the best-selling model is the ever-popular “bolts through the neck.”
- ▶ **Minimum oxtail soup percentage**
Optionally, a customer can specify the minimum percentage of oxtail soup to be in the monster’s blood. This is needed to keep the monster warm during cold nights.
- ▶ **Growth percentage**
Optionally, a customer can specify that, when angry, the monster increases in size by a specified percentage.
- ▶ **Early age strength**
Normally, a monster achieves its full strength at 28 days old, full strength being defined as being able to carry 25 elephants on one finger without breaking a sweat. Since that is usually good enough, specifying early age strength is optional. If a customer needs a strong monster in a hurry, though, then the customer can specify “early age strength”—for example, 20 elephants by seven days old.

In addition, the business expert has provided requirements about how to split up ingredients appropriately in the form of a spreadsheet; as you can see, the columns in the spreadsheet correspond to the parameters that the customer has to enter when specifying their monsters' requirements. The actual spreadsheet goes on forever; a small subset is shown in Figure 9.29.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Region	Brain Size	Colour	Monster Model	MOS Speed	Growth Speed	EAS Speed	Days	Sugar	Spice	AT Nice	Snips	Snails	PDT
2	NORTH	SMALL	GREEN	BLACK LAGOON	X	X	X	10	75	25	0	2	50	48
3	NORTH	SMALL	GREEN	BLACK LAGOON	X	X	X		74	24	2	4	49	47
4	NORTH	SMALL	GREEN	BLACK LAGOON	X	X			73	23	4	6	48	46
5	NORTH	SMALL	GREEN	BLACK LAGOON		X	X	10	72	22	6	8	47	45
6	NORTH	SMALL	GREEN	BLACK LAGOON		X	X		71	21	8	10	46	44
7	NORTH	SMALL	GREEN	BLACK LAGOON		X			70	20	10	12	45	43
8	NORTH	SMALL	GREEN	BLACK LAGOON	X		X	10	65	19	16	14	44	42
9	NORTH	SMALL	GREEN	BLACK LAGOON	X		X		60	18	22	16	43	41
10	NORTH	SMALL	GREEN	BLACK LAGOON	X				55	17	28	18	42	40
11	NORTH	SMALL	GREEN	BLACK LAGOON			X	10	50	16	34	20	41	39
12	NORTH	SMALL	GREEN	BLACK LAGOON			X		45	15	40	22	40	38
13	NORTH	SMALL	GREEN	BLACK LAGOON		X			40	14	46	25	39	36
14	NORTH	SMALL	GREEN	BLACK LAGOON	X				35	13	52	30	38	32
15	NORTH	SMALL	GREEN	BOLTS_THROUGH_NECK					30	12	58	35	37	28
16	NORTH	SMALL	GREEN						20	11	69	40	36	24
17														

Figure 9.29 Complex Rule Spreadsheet

To find the appropriate percentage split among ingredients using just the spreadsheet, the business logic would be as follows:

- ▶ Remove any lines that do not match the region, brain size, or color specification.
- ▶ If a particular model has been specified, then go through the lines for that model looking for a match. If no result is found, then go through all lines with no model specified, looking for a match.
- ▶ If a minimum oxtail soup percentage has been defined (any value), then a line is only a match if it has an X in the MOS SPECD column.
- ▶ If a growth percentage has been specified (any value), then a line is only a match if it has an X in the GROWTH SPECD column.
- ▶ If an early age strength has been specified, then a line is only a match if it has an X in the EAS SPECD column.
- ▶ If early age strength has been specified and there are two lines in the spreadsheet with an X in the EAS SPECD column, then the one that is chosen is the one in which the DAYS column is less than or equal to the early age chosen by the user. If the user has specified 15 days and the spreadsheet line has 10 days, then choose the line on the spreadsheet in which no days have been specified.

Remember, this is a real-life requirement! The data in question was not so frivolous, but the logic was exactly as complicated as this example.

Traditional Rules Management

An author's confession: I initially tried to solve this by using ABAP. I created a Z table with four primary keys: region, brain size, color, and model, because these fields are mandatory. Then I would read all the records with a match on all four and loop through them, performing all sorts of convoluted logic to determine which record was the best possible match. If I could not find any matches, then I did another database read to get all records with a match on region, brain size, and color but for which the model was blank. I then looped through these records, applying the same complicated logic checks as before.

This worked, but it took a lot of doing, and when wrong results were being returned, debugging the code was a nightmare.

BRFplus to the rescue!

Because that spreadsheet you were given by the business user looks just like a BRFplus decision table, using that expression type in BRFplus is clearly the way to go. (This may seem like a forced example, but if you look back through all the real

requirements you have received in your career, you may be surprised how many turn out to be decision tables of one form or another.)

To start, create a function with a signature that represents the actual values the user enters as selection. The result data object will be a structure with the six possible result columns. The complete signature can be seen in Figure 9.30.

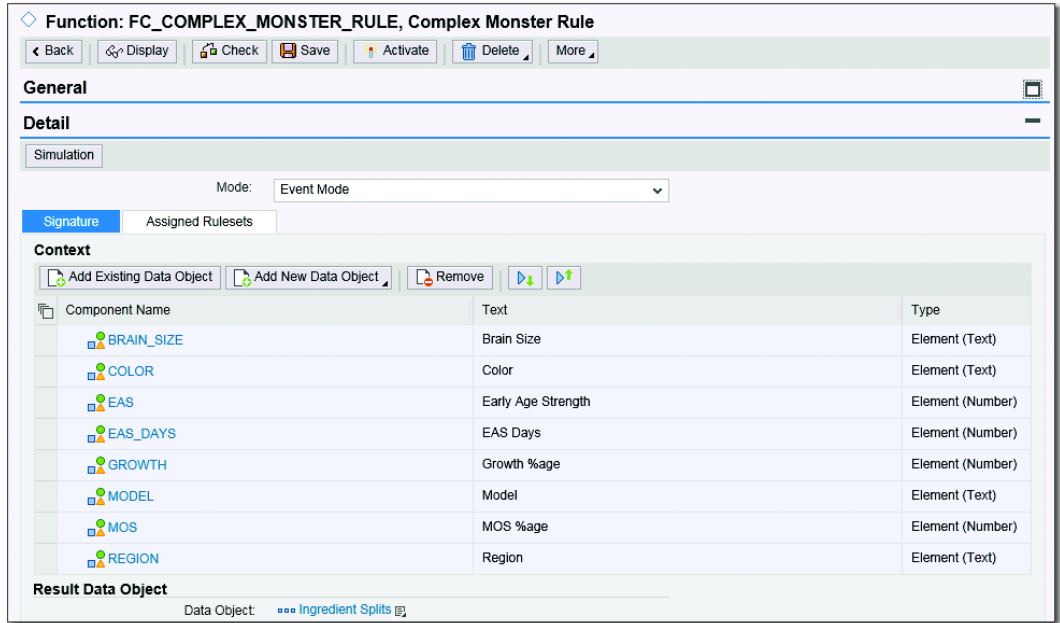


Figure 9.30 Complex Rule Function with Signature

Then, go through the same steps that were discussed in Section 9.2: Create a rule-set and a rule (expression), which is a decision table. There are different columns this time (early age strength, growth percentage, etc.): still customer requirements, but different customer requirements from those used in the earlier example. (Once again, there must be a little man inside the computer, because the columns will come out in the order you want. Either the system remembers the order in which you chose the columns, or it's magic.)

Because you have a large spreadsheet, first download the blank decision table to Excel (via the ADDITIONAL ACTIONS button shown in Figure 9.24) to make 100% sure that you have the format correct for upload. Then, paste in the val-

ues from the spreadsheet the user gave you, and check that everything is in the right column.

Some of the decision criteria you want are not fixed values but rather formulas—for example, early age strength days must be less than or equal to 10. You cannot achieve this in the IMG tables you used to use to make decisions, but you can input such values in a spreadsheet, and BRFplus will recognize them for what they are. Therefore, before uploading the data back into BRFplus, change the cells in the spreadsheet when they need to be formulas rather than the X in the spreadsheet. For example, the X for MOS %AGE is replaced by >0. The result can be seen in Figure 9.31.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Region	Brain Size	Color	Model	MOS %age	Growth %age	Early Age Strength	EAS Days	Sugar	Spice	All Things Nice	Slips	Snails	Puppy Dog Tails
2	NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	>0	<=10	75	25	0	2	50	48
3	NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	>0		74	24	2	4	49	47
4	NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0			73	23	4	6	48	46
5	NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	>0	<=10	72	22	6	8	47	45
6	NORTH	SMALL	GREEN	BLACK_LAGOON		>0	>0		71	21	8	10	46	44
7	NORTH	SMALL	GREEN	BLACK_LAGOON		>0			70	20	10	12	45	43
8	NORTH	SMALL	GREEN	BLACK_LAGOON	>0		>0	<=10	65	19	16	14	44	42
9	NORTH	SMALL	GREEN	BLACK_LAGOON	>0		>0		60	18	22	16	43	41
10	NORTH	SMALL	GREEN	BLACK_LAGOON	>0				55	17	28	18	42	40
11	NORTH	SMALL	GREEN	BLACK_LAGOON			>0	<=10	50	16	34	20	41	39
12	NORTH	SMALL	GREEN	BLACK_LAGOON			>0		45	15	40	22	40	38
13	NORTH	SMALL	GREEN	BLACK_LAGOON		>0			40	14	46	25	39	36
14	NORTH	SMALL	GREEN	BLACK_LAGOON	>0				35	13	52	30	38	32
15	NORTH	SMALL	GREEN	BOLTS_THROUGH_NECK					30	12	58	35	37	28
16	NORTH	SMALL	GREEN						20	11	69	40	36	24

Figure 9.31 Modified Spreadsheet Ready for Upload

Then, upload the spreadsheet back into BRFplus (via the good old ADDITIONAL ACTIONS button), and at once you have the equivalent data in the decision table (Figure 9.32).

The decision table achieves the exact same thing the ABAP code did. However, it took only a fraction of the time to create, there was no need for a Z table and complicated code, and there was less capacity for miscommunication—because you used the spreadsheet you were given by the business expert as a basis for what you uploaded to BRFplus.

Here comes the icing on the cake: With a decision table, you have two options in the ADDITIONAL ACTIONS context menu called CHECK COMPLETENESS and CHECK OVERLAP. The overlap check tells you if any rows in your table will never be chosen in any circumstances. In Figure 9.33, this occurs twice, because—to your horror—some of the lines are identical to each other. This is a problem; you have set the decision table to return only one row, so if there are two identical lines that

satisfy the condition, processing will stop after the first line is returned—so there is no way on Earth that the second line can ever be reached.

Decision Table: DTA_COMPLEX_MONSTER_RULE, Complex Monster Rule Decisi...

General

Detail

Table Contents

Region	Brain Size	Color	Model	MOS %age	Growth %age...	Early Age Strength	EAS Days	Sugar	Spice	All Things Nice	Snips	Snails	Puppy Dog Tails
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	>0	<=10	75	25	0	2	50	48
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	>0	...	74	24	2	4	49	47
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	>0	73	23	4	6	48	46
NORTH	SMALL	GREEN	BLACK_LAGOON	...	>0	>0	<=10	72	22	6	8	47	45
NORTH	SMALL	GREEN	BLACK_LAGOON	...	>0	>0	...	71	21	8	10	46	44
NORTH	SMALL	GREEN	BLACK_LAGOON	...	>0	70	20	10	12	45	43
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	...	>0	<=10	65	19	16	14	44	42
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	...	>0	...	60	18	22	16	43	41
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	55	17	28	18	42	40
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	<=10	50	16	34	20	41	39
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	...	45	15	40	22	40	38
NORTH	SMALL	GREEN	BLACK_LAGOON	...	>0	40	14	46	25	39	36
NORTH	SMALL	GREEN	BLACK_LAGOON	>0	35	13	52	30	38	32
NORTH	SMALL	GREEN	BOLTS_THROUGH_NECK	30	12	58	35	37	28
NORTH	SMALL	GREEN	20	11	68	40	36	24

Figure 9.32 Complex Rule Decision Table in BRFplus

Decision Table: DTA_COMPLEX_MONSTER_RULE, Complex Monster Rule Decisi...

Overlap Check Result (2 Messages)

- DTA_COMPLEX_MONSTER_RULE (Expression) : Line 12 is equivalent to line 6 (unreachable in first match mode) (Technical Details) Display Help
- DTA_COMPLEX_MONSTER_RULE (Expression) : Line 13 is equivalent to line 9 (unreachable in first match mode) (Technical Details) Display Help

Figure 9.33 Overlap Check

The overlap check would also pick up the situation that might occur in which some of the rows near the top of the decision table are less stringent than rows further on down, so the lesser condition is satisfied before the tougher one is evaluated. In the simple decision table, if you add a new line that checks for sanity of greater than 65%, then this line would get evaluated before the ones below it; the overlap check in Figure 9.34 shows the resulting errors.

Decision Table: DTA_MONSTERS_SIMPLE, Simple Monsters Rule Decisi...

Back Display Check Save Activate Delete More

Overlap Check Result (2 Messages)

Close

- **DTA_MONSTERS_SIMPLE (Expression)** : Line 3 is a subset of line 2 (unreachable in first match), move up ([Technical Details](#)) [Display Help](#)
- **DTA_MONSTERS_SIMPLE (Expression)** : Line 4 is a subset of line 2 (unreachable in first match), move up ([Technical Details](#)) [Display Help](#)

General

Detail

Additional Actions Context Overview Start Simulation

Table Contents

Find: Next Previous

Monster Sanity	Monster Usage	Monster Scariness	Monster Evilness	Rages Per Day	Ingredients
>=75	BALLROOM_DANCER	1
>65	2
>=75	MORTGAGE_SALESMAN	3
>=75	2

Figure 9.34 Overlap Check Showing that Rows are in the Wrong Order

In such a circumstance, you can go back to your business expert and query whether the rules are correct after all. The equivalent check would be a lot harder to do in conventional ABAP without a great deal of unit tests, and even then you would probably miss these errors.

The completeness check is intended to tell if you if it's possible to evaluate every single row and still not find a result. The preceding example would cause the check to scream, because the user could put in the region SOUTH, and no result would be found. This would be tedious to achieve in a normal ABAP method or function module—looping through every possible combination of input values looking for a failure to return a result. (As a caveat, the completeness check won't work when the text values are not linked to data elements with domains, because the system has no way of knowing what possible values a user can input.)

9.4 Simulations

This chapter has talked about conventional techniques for storing rules, which can be a bit of a black box; in other words, it is difficult to tell how exactly a given decision was reached. This can be taken further: When you make a change to an ABAP program or a customizing table, you might be able to do a test and see that the result has changed, but can you really be sure why? To get around this, the BRFplus

functions have a simulation function, which is designed as much for business experts as it is for IT. The following example shows this function in action.

First, go into the function definition in BRFplus, and click SIMULATION (Figure 9.35).

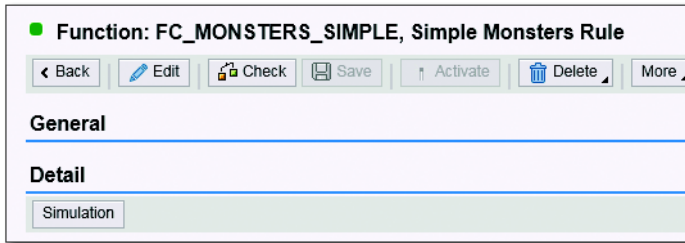


Figure 9.35 Simulation: 1 of 4

Figure 9.36 shows the initial screen of the simulation function. In GENERATION MODE, some actual ABAP code is generated, which is how BRFplus works when evaluating rules for real. For a simulation, you can stick with the default values.

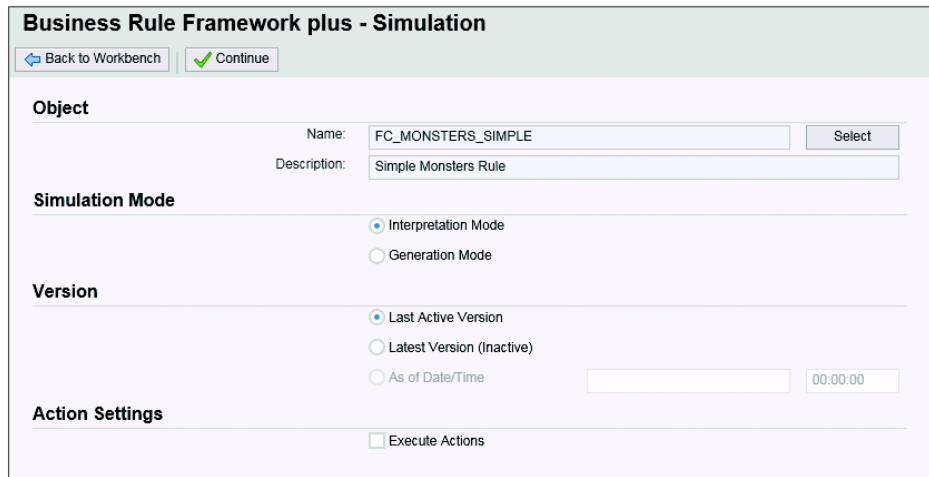


Figure 9.36 Simulation: 2 of 4

Figure 9.37 shows some test values that have been input to see what the result will be. In the past, you may have found yourself writing some small test programs in which you enter test values on a selection screen and then call a function module or class method to see what happens. This is just the same, without the effort.

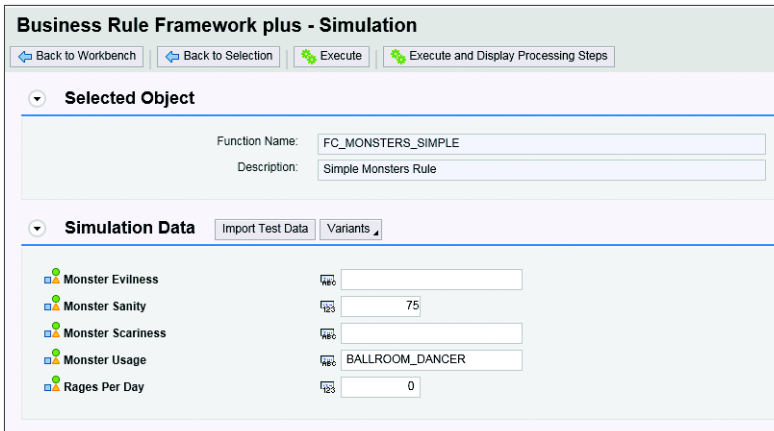


Figure 9.37 Simulation: 3 of 4

Figure 9.38 shows the simulation result screen. At the top, you can expand the CONTEXT VALUES to remind you what test values you entered. Then, you see the result. Finally, you see in excruciating detail the steps that BRFplus took in order to arrive at the result.

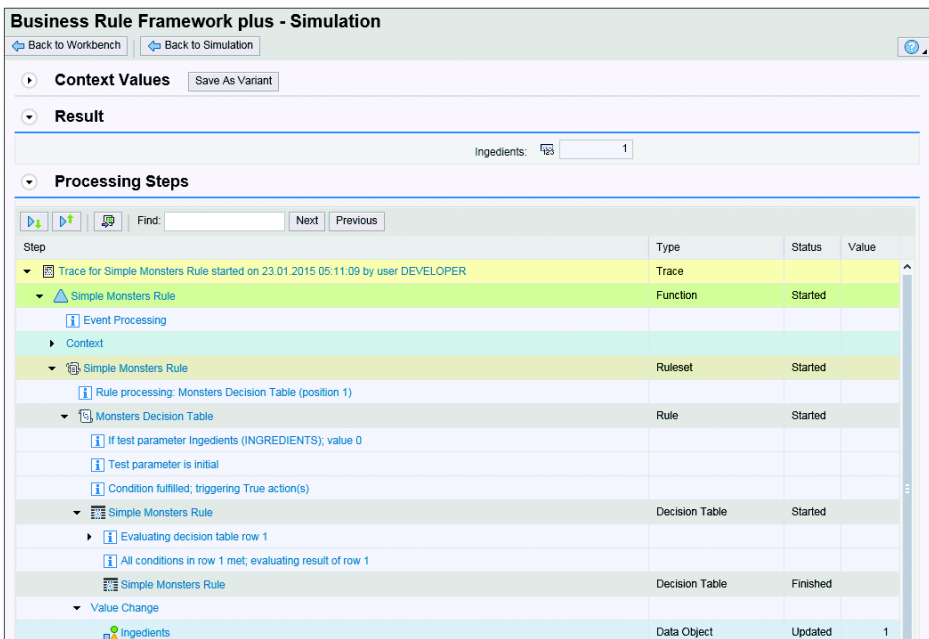


Figure 9.38 Simulation: 4 of 4

This is the exact opposite of the good old black box; instead, the whole decision process is under a microscope, having a flashlight shone on it, to make any logical flaws in the process stand out in letters of fire 1,000 miles high (as Jock McMetaphor, the Scotsman who mixes his metaphors, would say).

9.5 SAP Business Workflow Integration

SAP Business Workflow is one of the most powerful tools within SAP, and naturally it revolves around decisions, made either by those pesky human beings who work in your organization or by the computer. This makes this a natural fit for BRFplus, so SAP has included some out-of-the-box integration.

Warning: Houston, We Have a Problem

There is one prerequisite for integrating BRFplus with SAP Business Workflow: In the BRFplus application, the signature has to be composed of data objects, which are mapped to DDIC data elements. However, ultimately this should not be much of a hurdle. The examples in this chapter create data objects on the fly, but usually you would map all input/output parameters to data elements, because those are what your ABAP program is working with.

When you are in the Workflow Builder and you are creating an activity, you have a new option on the context menu: CREATE BRFPPLUS TASK. Clicking on this option leads to the screen shown in Figure 9.39.

In this screen, you will see two boxes: APPLICATION FOR FUNCTION and APPLICATION FOR RULESET. What is about to happen is that the system is going to generate a new application/function/ruleset to store the new rule you're about to create in the following steps. The boxes are asking if you want to add the generated ruleset and/or function to an existing application. In this case, choose to add the ruleset that is about to be created to your simple monster application, in case you ever want to use this new ruleset outside of workflow.

The rest of the screen has to do with creating a transport request. Choose the \$TMP package to make this a local object (when creating objects for productive use, you would naturally put a real development package in the field and then be prompted for a transport request). After clicking the green checkmark, you are brought to the screen shown in Figure 9.39. Here, you define the binding for the

task that is about to be created, which means the input and output parameters. When a workflow task is created, you have to define a *container*, which is a list of variables with which the workflow task deals. This isn't discussed here, as it is standard workflow and has nothing to do with BRFplus. However, while creating such a container, if the task is to be used with BRFplus, then you need to make sure that the high-level container includes all DDIC elements that you want to use in your BRFplus rule. This ensures that the data elements are available on the left-hand side of the screen, as shown in Figure 9.40. Then it's just a case of dragging and dropping all the parameters from the left-hand side of the screen to the right-hand side of the screen. After you've done that, highlight the return parameter, and click the icon at the top of the screen that says THIS IS A RETURN PARAMETER when you hover your cursor over the icon (which looks like a tiny box with a right pointing arrow underneath). To finish processing and leave the screen, click SAVE.

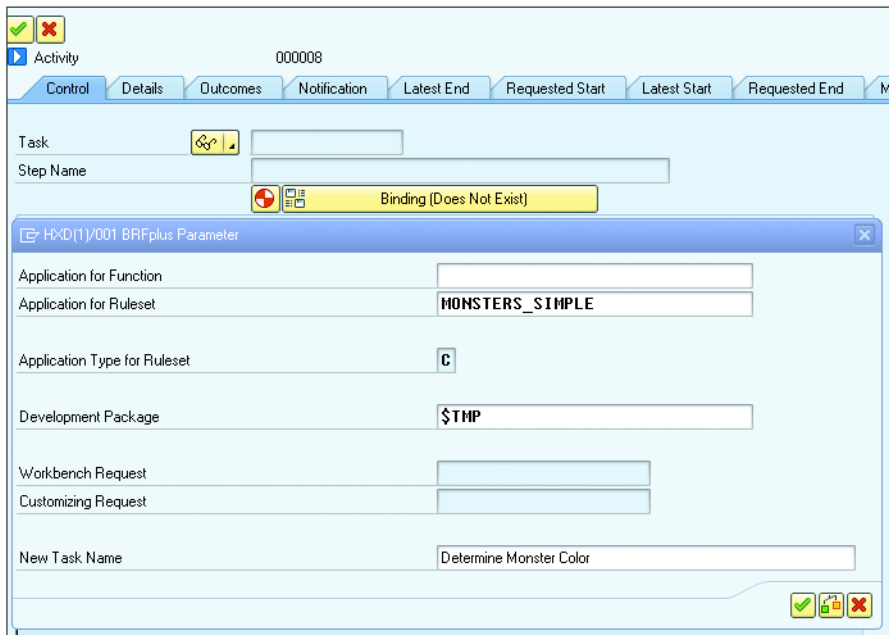


Figure 9.39 Creating a BRFplus task in SAP Business Workflow

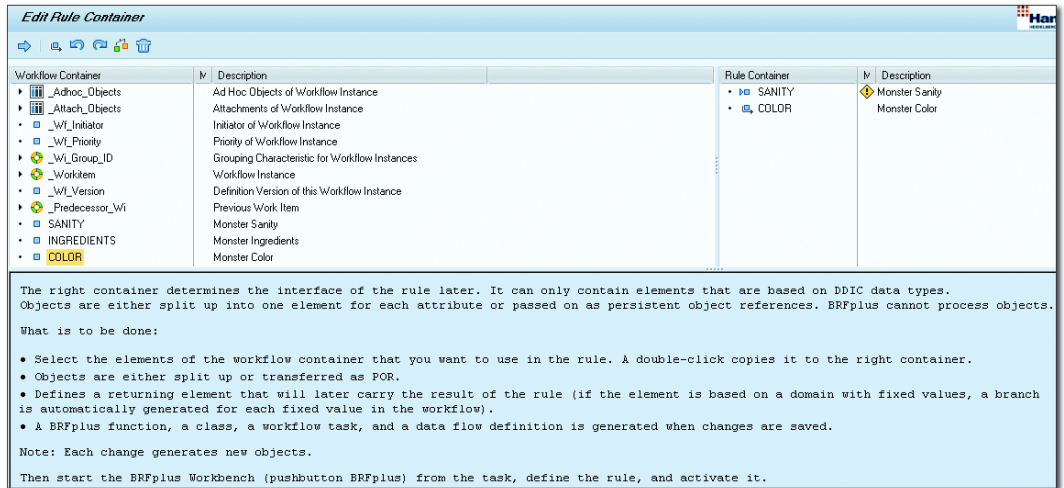


Figure 9.40 Defining the Binding for a BRFplus Workflow Task

You're then returned to the ACTIVITY definition screen. As can be seen in Figure 9.41, there's a lovely new button here that you can click, with BRFPLUS written on it and a picture of a magic wand. Everyone likes buttons; everyone *really* likes buttons with magic wands on them. Click this new one as fast as you can.

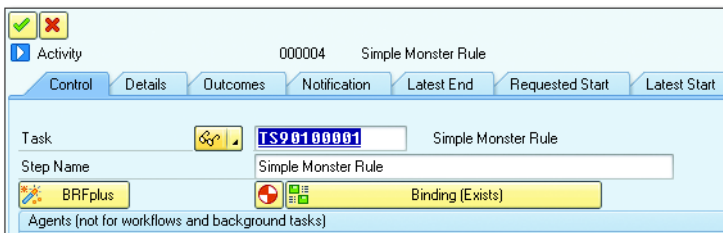


Figure 9.41 An Extra Button in the Activity Definition

Clicking this new button takes you to the screen shown in Figure 9.42, which is the normal screen for creating and changing a workflow task. The only difference is that instead of an SWO1 business object, you have a class, and that class has been generated for you by the system. By double-clicking on the funny-looking class name, you can look at the code itself. In this example, you pass in the monster sanity and get back the monster color. The code in the generated method looks like Listing 9.3.

Standard Task: Display

Standard task: 90100001 SWF_BRFPlus
 Name: Simple Monster Rule
 Package: \$TMP Applic. Component

Basic data | Description | Container | Triggering events | Terminating events | Default rules

Name
 Abbr.: SWF_BRFPlus
 Name: Simple Monster Rule
 Release status: Not defined

Work Item Text
 Work item text: Call of BRFplus Function

Object method
 Object Category: CL ABAP Class
 Object Type: ZCLBRF_005056B92F471ED3BF_0001 (Generated WF-BRFPlus)
 Method: EXECUTE
 Synchronous object method

Figure 9.42 Generated BRFplus Workflow Task

method EXECUTE.

```

call method _set_value(
  name      = 'SANITY'
  fdt_name  = 'SANITY'
  value     = SANITY ).

call method _PROCESS_FUNCTION( ).

call method _get_result(
  IMPORTING value = COLOR ).

endmethod.
```

Listing 9.3 Generated BRFplus Workflow Task Method

Change the generated task so that the binding accepts the sanity percentage and returns the color. The workflow step in the Workflow Builder has already created this binding for you.

The end result is that you can have a workflow with decision rules that are defined within BRFplus. In the running example, whenever you get a request to build a monster with a certain level of sanity, you can use BRFplus to work out the color, and then perform some actions in SAP Business Workflow based upon that result.

9.6 Options for Enhancements

No matter how good someone makes something, we human beings are never satisfied. The positive aspect of this is that it makes your job as a developer safe, because end users will always be crying out for more from your programs.

BRFplus was designed with this part of human nature in mind and thus has built-in facilities for when what you get out of the box is not quite good enough for what you want, and therefore you need to extend the functionality. It would be beyond the scope of this book to go into great detail about the various possible enhancements available in BRFplus, but it's worth a brief discussion of the possibilities for enhancement that do exist.

9.6.1 Procedure Expressions

Instead of a decision table or decision tree, one of the 20 or so other options for a BRFplus expression (type of rule) is a procedure call. Don't be fooled by the name; you are not calling a `FORM` routine, but instead a function module or a public method of a class.

The BRFplus framework lets you do some mapping between the function/method parameters and the data objects in BRFplus. The positive side of this is that because you are calling an ABAP routine you can do anything in the world. The negative side is this: If you have to resort to calling an ABAP routine, then why are you using BRFplus in the first place? In fact, there could be many valid reasons; for example, BRFplus might solve 95% of your problem in a more compact way than using Z tables and ABAP code, but you need to call an ABAP routine to round out the last 5% of the problem—in the same way that you would use a user exit to make a standard SAP transaction go the extra mile.

9.6.2 Application Exits

When you define an application in BRFplus, there is an option field called APPLICATION EXIT, which really means “user exit.” In this field, you can specify a Z class that implements a special BRFplus interface. Thereafter, at certain points during BRFplus processing, the methods of this class are called and can change or replace the standard behavior of the BRFplus framework.

For example, you can totally replace the authorization checks with your own custom ones. This is no small thing, because you will be opening up BRFplus to let business experts make some changes, so you had better make everything expert proof. Another common usage is to add custom formulas (by way of static ABAP methods) to the standard list of available choices in the FORMULA expression.

9.6.3 Custom Frontends

Chapter 12 will talk about Web Dynpro. BRFplus runs with a Web Dynpro frontend, and during construction the inventor of BRFplus thought to himself that the SAP customers (like you and me) might not be happy with the standard BRFplus user interface and might want to write their own.

How correct he was! Developers like me are never happy, and I would rather my business experts be unaware that there is a thing called BRFplus. Instead, I would give them a screen to change their business rules that looks just like the other custom screens our development department has written for them or, to be brutally honest, a screen that looks just like their legacy system, which is what they *really* want to keep on using. (Once I ported an access database to SAP and made the screens in SAP look like the old access database—only grayer—and for once we did not get a peep of complaint from the users about making the change.)

The good news is that you can write custom frontends for maintaining BRFplus rules. I'm not going to say it's easy; you need to know your Web Dynpro, for starters. For more information, check out the SAP PRESS book *BRFplus—Business Rule Management for ABAP Applications* (see the “Recommended Reading” box at the end of the chapter), which covers the public API of BRFplus classes so that you can call them from your own custom programs. That book also tells you how to go about the Web Dynpro aspect of writing your own custom frontends.

9.6.4 Custom Extensions

There are some people in this world who, when they win a million dollars on the lottery, get all upset and say, “Oh no, I wish I had won two million dollars.” In the same way, there are about 20 different expression types in BRFplus you can use to enter and update rules, but sooner or later you are going to wish there were more; the one you really wanted just isn't there. Well, stop moaning, and write the missing expression type yourself.

You can find out how on SCN by reading “Custom Expressions and Action Types” (see the “Recommended Reading” box at the end of the chapter) which gives you incredibly detailed instructions on how to do this. You have to be a top gun ABAP programmer, though—which I'm sure you are, so no problem there.

As an example, SAP has been working on a graphical expression type that would look like a Visio diagram. An unexpected problem reared its head at the last minute (in essence, the proposed solution wasn't compatible with Web Dynpro), so that feature isn't available yet. Why don't you see if you can write one yourself and beat SAP to it? I'm planning on doing just that—the week after I climb Mount Everest and score all the goals for the UK national football team that wins the World Cup final.

9.7 Summary

In some countries with a volatile climate, you often hear the expression “If you don't like the weather, just wait 10 minutes, and then something better may come along.” BRFplus is like that unpredictable weather: it's still a relatively young product and is evolving fast. When you upgrade to a higher SAP NetWeaver level—for example, a move from SAP NetWeaver 7.02 to 7.31—you will notice that the delta change between versions will be a lot more obvious than in mature transactions, like SE24, which stay pretty much the same. Moreover, even when you are not on the latest SAP NetWeaver release, every time you install a support stack, you will see a very large number of notes related to BRFplus. Some of these are bug fixes, but a large number also bring performance improvements and extra functionality. Moving from 7.02 SP 9 to 7.02 SP 14 made an enormous difference in terms of extra features and usability improvements. In the 7.4 release, for example, quite a lot has been added, ranging from minor (but popular) features (like a list of the last 50 objects you changed) to a test

case feature (which requires the DSM add-on) that is rather like ABAP Unit; you create some unit tests in development as to what input values get what expected results, and then you can run them all at once (remotely in production; you can do this because such tests do not alter system state) to see if the results are what you would have expected. This is of course very useful for regression testing—that is, making sure that your new rule change does not break anything that already exists. There is also a drag-and-drop feature coming, with which you can drag objects from the menu on the left over into the main screen area where you define your BRFplus expressions. Naturally, this will speed up creation and maintenance enormously. Because the IMG is not going away any time soon, if you want all customizing rules to be in the same place, then you need to look at accessing BRFplus from the IMG. You can add your own nodes to the IMG, and in the definition of the node you can specify your own BAdI, and also you can call up custom BRFplus frontend screens from your own code, so this should not be an insurmountable barrier. In fact, some of the newer SAP applications already have BRFplus nodes in their IMG.

All this is to say that the pace of new features being added is not going to slow down in the near future, so it could be said that the future for BRFplus is so bright that you have to wear shades. To sum up this chapter in one sentence: The success of SAP to date has been due to the fact that it can be extensively customized by the IMG, and the remaining gaps can be filled in via the ABAP language; BRFplus is a tool that extends the capabilities of both.

This chapter on BRFplus concludes the discussion of business logic. Now, you'll turn to the best way to shove that business logic into the user's face like a custard pie—namely, the user interface layer.

Recommended Reading

- ▶ *BRFplus—Business Rule Management for ABAP Applications* (Ziegler, Albrecht, SAP PRESS, 2011)
- ▶ Custom Expressions and Action Types: <https://scn.sap.com/docs/DOC-48050> (Wolfgang Schaper)
- ▶ SCN Resource Center for Business Rules Management: <http://scn.sap.com/docs/DOC-29158> (Wolfgang Schaper)
- ▶ SAP NetWeaver Decision Service Management—Let's Talk Features: <http://scn.sap.com/community/brm/blog/2013/09/27/sap-netweaver-decision-service-management-let-s-talk-features> (Carsten Ziegler)

PART III

User Interface Layer

Computer science departments have always considered “user interface” research to be sissy work.
—Nicholas Negroponte, founder and director of the Massachusetts Institute of Technology’s Media Laboratory

10 ALV SALV Reporting Framework

To start Part III, which focuses on the user interface (UI) layer, this chapter explains how to make best use of the SALV framework to code reports that run inside the SAP GUI. Although the SAP GUI is not particularly futuristic (quite the opposite), the SALV framework within the SAP GUI replaced the traditional ALV reporting framework and is still widely unfamiliar to many ABAP developers. The good news is that the SALV framework is a technology that old-school ABAPers find easy to get their heads around; it could be compared to Henry Ford’s “faster horse.”

A lot of programmers hate writing reports. They feel they are boring compared to cutting-edge new programs, and if they do have to write a report they want the pain of doing so over as quickly as possible. The first and most obvious way of doing this is to have a template program: When a new report is needed, you copy the template and then insert the report-specific tables and retrieval logic. In the days of procedural programming, you could isolate all repetitive ALV code in an `INCLUDE` program. You could have an `INCLUDE` program for all the data declarations, such as the definitions for the field catalog and sort catalog tables and so on, and another `INCLUDE` for the routines, such as adding a new line to the field catalog—that is, the routines that would always be identical if you created each program normally. As time went on, you also started to see ALV robot types of programs on the Internet that you could download. You ran the program and said what your internal table would look like, and the whole program was generated for you using the ability of SAP to use a program to create another program. You’d think that would be the ideal solution—but generated code tends to be lengthy due to (for example) not encapsulating repetitive code in a routine.

Enter SALV, which is one way to solve this problem. The SALV reporting framework, in the form of `CL_SALV_TABLE` and its friends, are in some senses a direct follow on from the ALV function modules that you're undoubtedly familiar with (although the reporting framework was actually developed after `CL_GUI_ALV_GRID`). The ALV function modules made the programming community sing and dance when they arrived back in the year 2000 but the SALV goes one stage further, making everyone dance even harder and sing even louder. Now with SALV you no longer have to do the same task twice. Previously, you had to (a) define all the fields for the internal table that stores the report data to be output in one place in the data declaration part of the program and then (b) define those exact same fields again in another part of the program when setting up a field catalog for an ALV function module or a `CL_GUI` class.

In the SALV reporting framework, you pass the internal table in, and the class dynamically creates the field catalog for you. In some basic report programs, that action halves the lines of code in one fell swoop. Moreover, you don't need to create a DYNPRO screen for the report output to live in; the SALV class does that for you as well, just like the function module did. (However, if you so desire, you can attach your SALV object to a control just like the `CL_GUI` classes, so you do have the best of both worlds.)

This time, there's no gray area. If you have a simple report, then using `CL_SALV_TABLE` is always going to take fewer lines of code and less programming effort than the function module equivalent—at least for saying what columns you want (and many reports have dozens of columns). This was the nail in the ALV function modules' coffins.

As you saw in Chapter 2, with ABAP 7.4 we dispense with another task (declaring the internal table), so you've moved from the following tasks...

1. Declaring an internal table by listing all the fields
2. Filling the internal table with data by listing all the fields again in a `SELECT` statement
3. Creating a field catalog by listing all the fields again
4. Calling the function module

...to this obviously shorter list of tasks:

1. Filling the internal table by listing all the fields in a `SELECT` statement, which also dynamically creates the internal table structure

2. Calling the `CL_SALV_TABLE` class and passing it to the internal table

This is all wonderful, even before 7.4. However, what causes consternation among programmers the first time they encounter SALV is that, as opposed to having structures as inputs to a function module, a SALV object takes other objects as input: one from sorting, one for subtotaling, one for controlling the fields—in fact, quite a large number of possible input objects. The more features you want in your SALV report, the more objects you have to declare, and then create, and then link to the main SALV object, and then finally feed the information you want. This is normal in object-oriented (OO) programming, but could also be described as a vast horde of boilerplate code statements sweeping out of the graveyard and devouring everything in their path.

Fortunately, there's a way to make this simpler, and this chapter shows you what it is. Section 10.1 and Section 10.2 explain the steps in creating a standard interface (which is designed to work with any UI technology) and wrap that around the SALV framework (thus avoiding the process of declaring all those pesky helper objects; "I'd have gotten away with it if it wasn't for those pesky helper objects!"). This enables you to hide the details of the SALV framework behind a generic interface—that is, to wrap the SALV in the standard interface. This, in turn, gives you the ability to change a report from using one UI technology to another by altering a bare minimum of code.

Even once you've mastered that process, though, you may encounter some other "impossible" things that programmers often struggle with when writing SALV reports: how to add new user commands programmatically to the standard menu and how to make the data editable. You're in luck! These topics are covered in Section 10.3 and Section 10.4.

Finally, in Section 10.5, you'll look at a new version of the SALV framework that came out with ABAP 7.4. This version is designed to handle reports that use very large internal tables to store the data to be displayed.

10.1 Getting Started

In the world of OO programming, the API (interface) we are about to create constitutes the view portion of the model-view-controller (MVC) pattern. The view is the part of the program whose sole concern is outputting data to the user in a

good-looking fashion; it is not concerned with where the data comes from in the first place or how any user input is handled.

Model-View-Controller

In almost any book or article on object-oriented programming, the MVC pattern is mentioned almost at once. *Head-First Design Patterns* (see the “Recommended Reading” box at the end of this chapter) describes this as the Elvis Presley of design patterns. Although we go into a bit more detail on MVC in Chapter 12 of the book, here’s a brief analogy to help you understand the gist.

Imagine that you are making a film. The model is the story writer: the writer has created the story and nothing could happen without it, but at the time of writing the writer could not be sure this would be realized as a movie; it might have ended up as a play or even a book. The writer didn’t actually care about this when writing the story.

The view comprises the actors, make-up artists, special effects people, and so on, who are doing their best to bring the story to life. They don’t actually care who wrote the story in the first place.

The controller is the movie company who takes the story from the model (writer) and gives it to the assorted view people (actors, etc.). The controller might suddenly decide this is better as a radio play, sack all the movie-based view people, and replace them with radio-based view people, and the model (writer and story) would still be the same.

Likewise, the controller might suddenly decide the writer is no good after all and replace the story with a similar one with the same sort of plot submitted the week before by another writer; the actors could stay the same and just learn some new lines.

The MVC design pattern tries to replicate this sort of real-world situation in software by having three different parts of the program all dealing with one aspect: business logic (model), presentation to the user (view), and interaction between the two (controller).

In this section, I’ll talk about creating a SALV-specific (concrete) class that contains SALV-specific attributes (Section 10.1.1) and writing a small example program to call that class (Section 10.1.2). These steps will prepare you to design the actual report interface, which is the focus of this chapter and the subject of Section 10.2.

10.1.1 Defining a SALV-Specific (Concrete) Class

In Section 10.2 you will define an interface that is framework-agnostic. Interfaces do not contain any code, so you need to define a concrete class that implements the abstract interface—in this example, a SALV-specific class.

The first problem to solve while defining the concrete class is this: In programs that make use of `CL_SALV_TABLE` for displaying reports, if you want to use anything above and beyond the absolute basics, then you end up having to declare a large number of object variables. This is because the SALV class is designed according to OO principles; instead of being one class that does everything (a so-called *god class*), it's composed of many small objects, each of which has one specific function.

To prevent writing the same code again and again in different programs, create a class called `ZCL_BC_VIEW_SALV_TABLE`, which will implement the `ZIF_BC_ALV_REPORT_VIEW` interface that you will define in Section 10.2. In the attributes section of the concrete class, you can then declare all the helper objects for the SALV class, as can be seen in Figure 10.1.

Attribute	Level	Visib...	Rea...	Typing	Associated Type	Description	Initial value
MO_ALV_GRID	Instance	Public		Type Ref	CL_SALV_TABLE	Basis Class for Simple Tables	
MO_CONTROLLER	Instance	Public		Type Ref	ZCL_BC_CONTROLLER	Base Class for Application Controllers	
MO_AGGREGATIONS	Instance	Private		Type Ref	CL_SALV_AGGREGATIONS	All Aggregation Objects	
MO_COLUMN	Instance	Private		Type Ref	CL_SALV_COLUMN_TABLE	Column Description of Simple, Two-Dimensional Tables	
MO_COLUMNS	Instance	Private		Type Ref	CL_SALV_COLUMNS_TABLE	Columns in Simple, Two-Dimensional Tables	
MO_EVENTS	Instance	Private		Type Ref	CL_SALV_EVENTS_TABLE	Events in Simple, Two-Dimensional Tables	
MO_FUNCTIONS	Instance	Private		Type Ref	CL_SALV_FUNCTIONS_LIST	Generic and User-Defined Functions in List-Type Tables	
MO_LAYOUT	Instance	Private		Type Ref	CL_SALV_LAYOUT	Settings for Layout	
MO_SELECTIONS	Instance	Private		Type Ref	CL_SALV_SELECTIONS	Selections in List-Type Output Tables	
MO_SORTS	Instance	Private		Type Ref	CL_SALV_SORTS	All Sort Objects	
MO_SETTINGS	Instance	Private		Type Ref	CL_SALV_DISPLAY_SETTINGS	Appearance of the ALV Output	
				Type			

Figure 10.1 SALV View Class: Attributes

In the calling program, you now only have to declare one object variable as opposed to about 10, which gets rid of a lot of boilerplate code statements and means that you don't have to declare a lot of variables in every report program you write. This is less effort for you the programmer and also makes life easier for anyone reading the code, because they don't get distracted by big chunks of data declarations.

10.1.2 Coding a Program to Call a Report

With `CL_SALV_TABLE` in its basic form, you only need two commands in order to call up a report: calling the `FACTORY` method to pass in the data table and then call-

ing the `DISPLAY` method. If all you want to do is show a basic report, then there is no need to go through the process of actually creating a report interface. However, in reality, 99 times out of 100 you want to modify the standard display in some way, and this is why you need to have a concrete SALV-specific class that implements your custom interface.

If you do have a concrete SALV-specific class that implements a custom interface, then you do not call the `FACTORY` method of the `CL_SALV_TABLE` but rather create an instance of your concrete SALV class and call a method of that to display the report.

Note

In his book *Clean Code* (see the “Recommended Reading” box at the end of this chapter), Robert Martin says that a program should be like a newspaper: You start off looking at the headlines and then drill into anything you find interesting. In the following sections, whenever there is a code sample, that sample will start off with a small routine that calls several methods. Then each of these methods is examined in detail (which usually involves branching off into several more methods).

Listing 10.1 shows how to call a report using a class that implements the common API—in this case, your SALV class. You start off creating a local class that inherits from the concrete SALV class so that you can redefine the `APPLICATION_SPECIFIC_CHANGES` method. Next, the data to be displayed is read into an internal table. Finally, a method of your SALV-specific class is called in order to display the report. This is only about 1% more complicated than calling the `FACTORY` method, but it gives you much more flexibility.

```
REPORT y_monster_report_salv.

DATA: gt_monsters TYPE STANDARD TABLE OF ztvc_monsters.

PARAMETERS: p_vari TYPE disvariant-variant.

CLASS lcl_view DEFINITION INHERITING FROM zcl_bc_view_salv_table.
  PUBLIC SECTION.
    METHODS : application_specific_changes REDEFINITION.
ENDCLASS.

DATA :lo_view TYPE REF TO lcl_view,
      ld_repid TYPE sy-repid.

START-OF-SELECTION.

  SELECT *
```

```

FROM ztvc_monsters
INTO CORRESPONDING FIELDS OF TABLE gt_monsters.

"It is bad news to pass system variables as parameters
ld_repid = sy-repid.

CREATE OBJECT lo_view.

lo_view->prepare_display_data(
EXPORTING
    id_report_name = ld_repid           " Calling program
    id_variant     = p_vari " Display Variant as specified by user
CHANGING
    ct_data_table = gt_monsters ).

```

Listing 10.1 Generic Code to Call the Custom API

There is only method being called in Listing 10.1: the `PREPARE_DISPLAY_DATA` method. This method is inherited from the abstract class and implemented in the concrete class but has a totally generic nature. Listing 10.2 examines the coding for this method. First, initialize the underlying report object in whatever technology you're using. Then, pass control to the calling report to make changes specific to the application. Finally, display the report using your chosen technology.

```

METHOD zif_bc_alv_report_view~prepare_display_data.
* Step One - Generic - Set up the Basic Report
    initialise(
        EXPORTING
            id_report_name = id_report_name " Calling program
            id_variant     = id_variant     " Layout
            io_container   = io_container
            it_user_commands = it_user_commands " Toolbar Buttons
        CHANGING
            ct_data_table = ct_data_table ).

* Step Two - Application Specific
    application_specific_changes( ).

* Step Three - Generic - Actually Display the Report
    display( ).

ENDMETHOD.

```

Listing 10.2 Preparing and Displaying Data Method

The `PREPARE_DISPLAY_DATA` method orchestrates the three mandatory process flow steps that are part of creating a report interface. These steps are discussed in detail in the next section.

10.2 Designing a Report Interface

Now that you've defined an SALV-specific class and coded the program to call a report, you'll create an interface for the concrete class to implement. This interface is the public API for disparate report technologies; the calling program knows what the report technology can give it and cares not how it's achieved.

An interface is composed of attribute definitions and method definitions. The attribute definition section of the interface contains very little (in fact, nothing) we can include that is statically typed; each technology uses very different structures (or objects) to get the job done. In fact, the only thing we can say for certain is that there will always be a table of data to be displayed, and it will be defined differently in every case.

Interface methods are supposed to be an abstract representation of tasks in a high-level process—in this case, the process a program has to go through when preparing and displaying a report. Therefore, in order to know what method definitions to create, you need to think of the high-level steps (Figure 10.2) executed by the calling program when a report is executed.

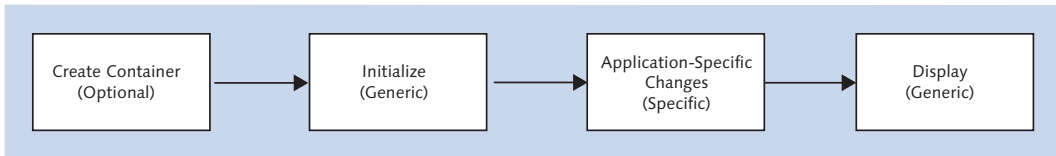


Figure 10.2 Steps in Report Execution

As you can see in Figure 10.2, there are four high-level steps executed by the program. Once designed, the interface will represent this high-level view expressed in SAP terms. The calling report will only know about this high-level framework. Classes specific to each UI technology will determine how to implement each step, thus separating the “what” from the “how to.”

Note

The designation of “Specific” or “Generic” in parentheses in Figure 10.2 refers to whether or not each step requires that you write unique code for an individual program. Logically, application-specific changes must be written differently for each individual program. All the other steps are generic, which mean they apply to every single report you create.

The following information describes the steps in a bit more detail:

1. Create Container (Optional)

For some reports, the program has to create an object to represent a container and pass it on to the UI framework; this container defines where the report lives on an SAP screen. In some cases, the container and screen are created by the programmer. (However, it's also possible to automate the creation of both the screen and container.)

2. Initialize (Generic)

The program executes an initialization routine, filling variables with a mixture of hard-coded values and user input from the selection screen. These variables are filled in based on a routine created by the programmer (the nature of the routine varies wildly depending on the UI framework in use).

3. Application-Specific Changes (Specific)

The program executes application-specific routines (column definitions, sorting, etc.) that were written by the programmer.

4. Display (Generic)

The program displays the report, which, again, is done based on code written by the programmer.

As you can see, you've got your work cut out for you! Each step will need one or more method definitions in order to get the job done (these method definitions are listed in Figure 10.3). But don't worry—the rest of this section will walk you through the process of defining these methods.

Method	Level	Me...	Description
INITIALISE	Insta...		Initialise the View
PREPARE_DISPLAY_DATA	Insta...		Prepare then Display the Data
DISPLAY	Insta...		Display Data
SET_COLUMN_ATTRIBUTES	Insta...		Set Hotspots, lengths, new names etc
ADD_SORT_CRITERIA	Insta...		Add Sort Criteria
OPTIMISE_COLUMN_WIDTHS	Insta...		Optimise Column Widths
SET_LIST_HEADER	Insta...		Set List Header
REFRESH_DISPLAY	Insta...		Refresh Screen Display with Updated Data

Figure 10.3 View Interface: Methods

10.2.1 Report Flow Step 1: Creating a Container (Generic/Optional)

The vast majority of reports you are called upon to create will take up the whole screen, but occasionally you will need to have two or more tables of data on the same screen. Each table of data to be displayed will have its own report object (e.g., an instance of `CL_SALV_TABLE`), and when that object is being created you can pass in a container object that represents an area of a DYNPRO screen inside which the data table will appear.

Note

Creating such a container is an optional step for the SALV framework; if no such container object is passed in, then the report will take up the whole screen without you having to do anything.

If you do need to create a container, though, it can be quite painful: you have to define a DYNPRO screen, then paint a big blank custom control in the area the report will be in, and then do some coding to link this big blank area to an ABAP object that will represent it. If this was your least favorite part of creating a report using `CL_GUI_ALV_GRID`, you'll like `CL_SALV_TABLE` (and you'll never want to go back to creating containers manually again).

Luckily, it turns out that you don't have to worry about this at all when defining your generic interface. For now, don't bother creating a container. If you don't pass a container into your `INITIALIZE` method, then the SALV will create the report in full screen mode, which is generally fine. (However, in Section 10.3.1 you'll see that when you do need to create a container, this step can be automated.)

10.2.2 Report Flow Step 2: Initializing a Report (Generic)

As noted earlier, three of the four steps in the high-level process flow are generic, which means that they apply to every single report you create. No matter what UI technology you're using, there's some basic information you need in order to make some high-level report settings, and we need a method to pass in that basic information. This is the `INITIALIZE` method.

In Figure 10.4, you can see the parameters of the `INITIALIZE` method, in which you pass in some high-level parameters for the report as a whole.

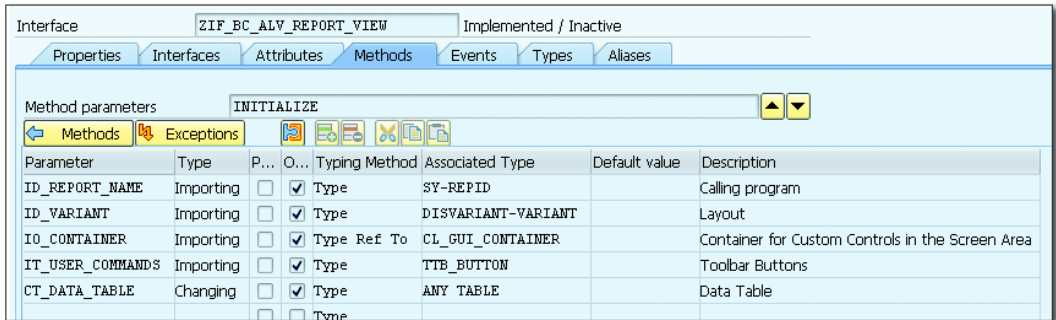


Figure 10.4 INITIALIZE Method Parameters

Take a look through the parameters in the signature for the INITIALIZE methods. Start with the importing parameters `ID_REPORT_NAME` and `ID_VARIANT`. You've most likely already worked out a reason you need a variant; no two users ever quite like to see the exact same columns in the same report. Because you don't want to have 23 virtually identical reports all with similar column layouts, you tend to write reports with a large number of columns, most of them hidden. Then the users can choose the ones they like to look at and save this information in a variant. Any report worth its salt has the display variant as an input parameter on the selection screen. (The reason you pass in both the name of this display variant and also the report name is that this is needed to distinguish identically named display variants for different reports.)

Then, there's the optional parameter `IO_CONTAINER`, which is a container, as discussed in Section 10.1.1—that is, an area of the screen where you want the report to appear. You could have four different reports appearing on the one page, or an area at the top for unprocessed items and an area at the bottom for unprocessed items with the two areas having different layouts, or a tree at the top and a grid at the bottom and the user drags items from the grid onto the tree. In all of these examples, you need to specify where the report being initialized should appear. If you just want the whole screen as per normal, then you don't need to pass anything in.

Next comes an importing table: `IT_USER_COMMANDS`. As noted earlier, all the standard SAP report technologies give you a nice inbuilt series of buttons that appear at the top of the screen to sort and filter the data, export the data to Excel, and so on. A lot of the time, you also want to add some of your own buttons to invoke another transaction or report, be it custom or standard, using data from this

report. You might have a report that's a list of monsters, and you can select a checkbox by a particular monster, and then click an icon at the top of the screen, which displays a map showing the monster's location in real time. In a slightly more realistic example, you might have a list of customers whose orders are going to be delivered late, and you select a customer, and then click a button, which triggers an outgoing phone call to the phone number stored in SAP ERP so that you can tell them the bad news. Neither of those examples would have been feasible 10 years ago, which shows how far technology has come.

Anyway, you want to be able to pass in a list of these extra user commands into your report object, and have extra buttons appear at the top of the screen, and have extra options on the context menu the users get when they select a record and right-click. A table of the structure `STB_BUTTON` is ideal for this purpose, because it has fields for the name of the new command, what it should look like (icon, button, etc.), and text description fields to tell the user what the button does: a short description that appears next to the button and a longer description that appears via hover text.

In a good OO program, this list comes from the application model, which says to the controller something like, "Hello, here's a list of commands I can respond to. I'll give you the descriptions and a rough guide to how they should look, but I'll leave the actual details to you." The controller passes the instructions to the view with a similar comment. This way, when a new and better view technology comes along, the controller can cast aside the old view and run off with the new younger one but still give it the exact same command list.

The actual table is passed in changing parameter `CT_DATA_TABLE` typed as a generic table, due to the fact that the structure of the table to be displayed is different for every report. This example uses a changing parameter for the technical reason that some technologies want this table as a changing parameter and others want it as a pure importing parameter. A changing parameter is the lowest common denominator; you can have a changing parameter and not change it, but you can't have an importing parameter and then try to change it in an OO method.

In order to code the `INITIALIZE` method, you have to split up the creation of the view object and the initialization, because you create the view object, then link it into the MVC framework, and only at a later date when you have the finished report data from the model do you actually create the underlying report object.

The idea here is to perform each step you would manually perform when creating a SALV object, provided that each such step would always be performed without exception and always in the same way.

Setting Up the Basic Report

First, you always need to create the SALV object itself; without it, you're not going anywhere (Listing 10.3).

```
cl_salv_table=>factory(
  IMPORTING
    r_salv_table = mo_alv_grid
  CHANGING
    t_table      = ct_data_table[] ).
```

Listing 10.3 Creating the SALV Object

In contrast to the older technologies, the `factory` method makes no presumptions as to whether you want all the lovely buttons at the top of the screen for sorting and changing the display variant and the like. This is the single responsibility principle: it's `factory`'s job to create the report object, but it's not its place to make any other decisions, like whether you want useful buttons at the top of the screen. In fact, you always do want such buttons, so call the method shown in Listing 10.4 directly after creating the SALV object.

```
display_basic_toolbar( ).

METHOD display_basic_toolbar.

  mo_functions = mo_alv_grid->get_functions( ).
  mo_functions->set_all( if_salv_c_bool_sap=>true ).

ENDMETHOD.
```

Listing 10.4 Making Sure a Toolbar Appears at the Top of the Report

In case you want to change some column attributes later, now is a good time to create the object that looks after the columns:

```
mo_columns = mo_alv_grid->get_columns( ).
```

If you don't do this early on (and construction is nice and early), then you'd have to keep calling the `GET_COLUMNS` method every time you wanted to change a column, which is painful, boring, and repetitive.

Next, if the user has chosen a display variant, then you want to make sure the report comes out with the chosen column order and sorting and what have you:

```
set_layout( id_variant ).
```

In Listing 10.5, you're setting up the layout by calling a big bunch of SALV methods. First, you have to access the `layout` object so that you can make changes to it. Then, you pass in the calling program: without this information, the program cannot determine which variant to retrieve from the database, because there might be identically-named display variants for different programs. Pass this into the `layout` object, tell this object that if there is no user variant, then it should use the default variant, and finally pass in the variant the user selected. You'll notice that Listing 10.5 includes a custom authority object to say whether a user can change global display variants; generally, only central people (like business analysts) should be authorized to change global variants.

```
METHOD set_layout.
* Local Variables
  DATA: ls_key TYPE salv_s_layout_key.

  mo_layout = mo_alv_grid->get_layout( ).

* Set the Layout Key
  ls_key-report = sy-cprog.

  mo_layout->set_key( ls_key ).
* set usage of default Layouts
  mo_layout->set_default( 'X' ).

* set initial Layout
  IF id_variant IS NOT INITIAL.
    mo_layout->set_initial_layout( id_variant ).
  ENDIF.

* Set save restriction
* Check authority to change display variants.
  AUTHORITY-CHECK OBJECT 'Z_VARIANT1' ID 'ACTVT' FIELD '*'.

  IF sy-subrc = 0.    " does he ride a white horse?
    mo_layout->set_save_restriction( if_salv_c_layout=>restrict_none ).
    " yes, allow user and global display variants
  ELSE.
    mo_layout->set_save_restriction( if_salv_c_layout=>restrict_user_
dependant ).
  ENDIF.

ENDMETHOD."Set Layout
```

Listing 10.5 Setting up the Layout

Setting Up Event Handling

Listing 10.6 sets the handlers, which means that it defines how user commands are handled in the program.

```
set_handlers( ).
METHOD set_handlers.

    mo_events = mo_alv_grid->get_event( ).

    SET HANDLER handle_link_click    FOR mo_events.

    SET HANDLER handle_user_command FOR mo_events.

ENDMETHOD.
```

Listing 10.6 Custom ALV Object: SET_HANDLERS Method

Forging the link between the events that are raised by the SALV objects is quite complicated. The main SALV class contains an attribute within it (CL_SALV_EVENTS_TABLE) that raises an event (RAISE_LINK_CLICK) when a user clicks on a hotspot. You need the code in Listing 10.6 to tell your newly created SALV object that the HANDLE_LINK_CLICK method in your custom SALV class is going to respond to that event. That declaration does half the job; the other step you need to do is to make sure that the definition of the event handler method links the event from CL_SALV_EVENTS_TABLE to your custom event handling method. In SE24, select the method HANDLE_LINK_CLICK of your custom class and click on the DETAIL VIEW icon at the top of the list of methods. The result can be seen in Figure 10.5.

Setting up such a definition in the class gives the method the potential to be used as an event handler, but it's not actually used until a SET_HANDLER command is used at runtime to subscribe to an event; that is, you activate the linkage between a specific instance of the class that raises the event and a specific instance of the class that is to handle the event.

Now you've handled the case in which the user has clicked a hotspot, such as clicking a monster number to display the monster master record. The other case you have to deal with is one in which you've added your own buttons to the top of the screen (the steps for doing this are explained in Section 10.3) and the user has clicked one of them. Naturally, if the user clicks one of the standard buttons, then the SALV class deals with that situation itself.

Object type	ZCL_BC_UVIEW_SALV_TABLE
Method	HANDLE_LINK_CLICK
Description	User has clicked on a hotspot
Visibility	Method
<input checked="" type="radio"/> Public <input type="radio"/> Protected <input type="radio"/> Private	<input type="radio"/> Static <input checked="" type="radio"/> Instance
<input type="checkbox"/> Abstract <input type="checkbox"/> Final <input checked="" type="checkbox"/> Event handler for	
Class/interface	CL_SALV_EVENTS_TABLE
Event	LINK_CLICK
<input type="checkbox"/> Modeled <input type="checkbox"/> Editor lock <input checked="" type="checkbox"/> Active	

Figure 10.5 Hotspot Event Handling Class: Detail View

There is a sort of added bonus to handling the `LINK_CLICK` event. This is because the way you put checkboxes into a SALV report is more complex than for the older technologies. In a SALV report, the checkbox does not just turn itself on and off when the user clicks on it. Instead, it raises the `ON_LINK_CLICK` event, and you have to do some code to turn the underlying field into a space or an X. However, you just defined a handler for `ON_LINK_CLICK`, so you can just put the code for switching on and off the check into your main `USER_COMMAND` routine.

If you thought it was quite convoluted setting up the last event link, then things are going to get worse. As mentioned before, there is an attribute hiding inside `CL_SALV_TABLE`: an instance of `CL_SALV_EVENTS_TABLE`. The `CL_SALV_EVENTS_TABLE` class inherits from `CL_SALV_EVENTS`, which raises an event called `ADDED_FUNCTION` when the user clicks a user-defined button (what SAP calls an *application-defined function*). In the detail view of your `HANDLE_USER_COMMAND` method, define a linkage to the event of the superclass, as shown in Figure 10.6.

That may seem overcomplicated at first glance, and indeed it is if you have to do this over and over again every time you write a report program—but now that you've set this up in a reusable class, you never have to worry about it ever again. The effect of drilling down on a particular field may vary from program to program, as does how many (if any) user-defined buttons there are—but the

mechanism for responding to them never changes, and this is handled for the calling program when it calls the `CREATE OBJECT` mechanism.

Object type	ZCL_BC_VIEW_SALV_TABLE	
Method	HANDLE_USER_COMMAND	
Description	User has pressed a menu button	
Visibility	<input checked="" type="radio"/> Public <input type="radio"/> Protected <input type="radio"/> Private	
Method	<input type="radio"/> Static <input checked="" type="radio"/> Instance	
<input type="checkbox"/> Abstract <input type="checkbox"/> Final <input checked="" type="checkbox"/> Event handler for		
Class/interface	CL_SALV_EVENTS	
Event	ADDED_FUNCTION	
<input type="checkbox"/> Modeled <input type="checkbox"/> Editor lock <input checked="" type="checkbox"/> Active		

Figure 10.6 Self-Defined Function Event Handling Method: Detail View

In total, Listing 10.7 shows what the code looks like for the `INITIALIZE` method.

```
METHOD zif_bc_alv_report_view~initialize.
```

```
TRY.
```

```

    cl_salv_table=>factory(
      IMPORTING
        r_salv_table = mo_alv_grid
      CHANGING
        t_table      = ct_data_table[] ).

    display_basic_toolbar( ).
    mo_columns = mo_alv_grid->get_columns( ).
    set_layout( id_variant ).
    set_handlers( ).

```

```

CATCH cx_salv_msg.
    "Raise fatal error - there is a serious bug in the program
    RETURN.
ENDTRY.

```

```
ENDMETHOD.
```

Listing 10.7 INITIALIZE Method

Note

Section 10.3 and Section 10.4 of this chapter discusses souping up this method to make it do "impossible" things, but for now you've got the basics.

10.2.3 Report Flow Step 3: Making Application-Specific Changes (Specific)

You now come to the part of the program that will vary with each and every report: the application-specific changes. For incredibly simple reports that only display a read-only representation of every column of an internal table, you may well have to do nothing here; however, it's likely that sooner or later the business users are going to demand something extra. You achieve this by calling one or more methods to do things like optimize the column widths, change various aspects of the columns of data, or sort and subtotal the data in a specific way.

Since you are pretty much always going to have to make application-specific changes, there needs to be a method to do this. Good news; there is! There is a definition of an `APPLICATION_SPECIFIC_CHANGES` method in the interface that is an abstract method in the concrete `ZCL_BC_VIEW_SALV_TABLE` class; that is, the method is always redefined by the calling program. Therefore, in the calling program, you will have code like that in Listing 10.8.

In this code, first the column widths are optimized to make them look pretty. Then, the column definitions are changed in various ways to satisfy various common business requirements. Next, the code specifies how the data is to be sorted. Finally, there's a bucket load of error handling.

```
CLASS lcl_view IMPLEMENTATION.

    METHOD application_specific_changes.
    * Local Variables
    DATA: lo_error      TYPE REF TO cx_salv_msg,
           lo_not_found TYPE REF TO cx_salv_not_found,
           lo_data_error TYPE REF TO cx_salv_data_error,
           ls_message    TYPE bal_s_msg.
    TRY.

        lo_view->optimise_column_width( ).

        lo_view->set_column_attributes( :
            id_field_name = 'MANDT'
```



```

if_is_technical = abap_true ),
id_field_name   = 'MONSTER_NUMBER'
if_is_hotspot   = abap_true
id_tooltip     = |'Click here to see the Monster Master Record'| ),
id_field_name   = 'HAT_SIZE'
if_is_visible   = abap_false ),
id_field_name   = 'MONSTER_COUNT'
id_long_text    = 'Monster Count'
if_is_subtotal  = abap_true ).

CATCH cx_salv_not_found INTO lo_not_found.
  ls_message = lo_not_found->get_message( ).
  IF ls_message-msgid IS INITIAL.
    MESSAGE s290(zbc_horizontool_mc01). "Report in Trouble
  ELSE.
    MESSAGE ID
ls_message-msgid TYPE 'I' NUMBER ls_message-msgno
    WITH ls_message-msgv1 ls_message-msgv2
         ls_message-msgv3 ls_message-msgv4.
  ENDIF.
CATCH cx_salv_data_error INTO lo_data_error.
  ls_message = lo_data_error->get_message( ).
  IF ls_message-msgid IS INITIAL.
    MESSAGE s290(zbc_horizontool_mc01). "Report in Trouble
  ELSE.
    MESSAGE ID
ls_message-msgid TYPE 'I' NUMBER ls_message-msgno
    WITH ls_message-msgv1 ls_message-msgv2
         ls_message-msgv3 ls_message-msgv4.
  ENDIF.
CATCH cx_salv_msg INTO lo_error.
  IF lo_error->msgid IS INITIAL.
    MESSAGE s290(zbc_horizontool_mc01). "Report in Trouble
  ELSE.
    MESSAGE ID
lo_error->msgid TYPE 'I' NUMBER lo_error->msgno
    WITH lo_error->msgv1 lo_error->msgv2
         lo_error->msgv3 lo_error->msgv4.
  ENDIF.
RETURN.
ENDTRY.
ENDMETHOD. "Application Specific Changes

ENDCLASS.

```

Listing 10.8 Application-Specific Changes Method in Calling Program

Each of the methods called in Listing 10.8 are discussed next; you'll see how they're defined in the abstract interface and implemented in the concrete class.

Optimizing the Column Width

The method `OPTIMIZE_COLUMN_WIDTHS` has no parameters, because what it does is very straightforward. Ninety-nine times out of 100, your report will have columns such as customer names, and more often than not the longest customer name in the report will not be long enough to fill the entire length of the customer name field in the data dictionary. To avoid empty space in the report and to get the maximum number of columns on one screen to prevent the users having to scroll to the right (they hate doing that), you can tell the view technology to work out the maximum length needed for each column dynamically at runtime based on the actual contents of the table to be displayed.

Note

There are rare occasions when it's vital that you maintain total control over how wide each column is, and in such cases you do not want to call this method.

Setting the Column Attributes

The `SET_COLUMN_ATTRIBUTES_METHOD` is probably the most commonly used method for making application-specific changes; it's a way to define the special attributes of the various columns in the internal table to be displayed. As you can see in Figure 10.7, this consists of a number of optional input parameters that control the following elements:

- ▶ The identity of the column in question. To define this, you pass in the field (column) name, which is the name of the field in the internal table structure. You have to have the table name as an optional parameter as well, because the interface has to cater to several UI technologies. One of which, `CL_GUI_ALV_GRID`, cannot live without the field name—so you have to pander to the lowest common denominator.
- ▶ Whether the user can double-click on a column value and trigger an action, such as viewing the underlying business document (a hotspot).
- ▶ Whether the column is hidden by default and the user can choose to display it or whether the column is never to be displayed under any circumstances (so-called *technical columns*, which are only used internally by the program).

- ▶ Whether the column is to be displayed as an icon (technically, this is a push button, because you “push” it by clicking it, and something happens as a result).
- ▶ Whether the contents of the column can be subtotaled. Obviously, this only makes sense for numeric fields, but you don’t want it active for all numeric fields by default. Some numeric fields should not be subtotaled (e.g., averages, unit rates, etc.).
- ▶ Whether the column is to have a different column heading than the one that the SAP system would normally propose based upon the data element used in the definition of the field in question. In later technologies, you also have a tooltip, which is a little box that appears when the user hovers their cursor over the column heading to give a lengthier explanation of what the column contents represent.

Parameter	Type	Pa...	Op...	Typing Met...	Associated Type	Default value	Description
ID_FIELD_NAME	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	LUC_FNAME		ALV control: Field name of internal table field
ID_TABLE_NAME	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	LUC_S_FCAT-REF_...		ALV control: Reference table name for internal table field
IF_IS_HOTSPOT	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	ABAP_BOOL		Can you drill down?
IF_IS_VISIBILE	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	ABAP_BOOL		Is the field hidden?
IF_IS_TECHNICAL	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	ABAP_BOOL		Is the field for internal use only?
IF_IS_A_BUTTON	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	ABAP_BOOL		Is this field a push button?
IF_IS_SUBTOTAL	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	ABAP_BOOL		Do we want a subtotal?
ID_LONG_TEXT	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	SCRTEXT_L		Replace Long Text
ID_MEDIUM_TEXT	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	SCRTEXT_M		Replace Medium Text
ID_SHORT_TEXT	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	SCRTEXT_S		Replace Short Text
ID_TOOLTIP	Importing	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Type	LUC_TIP		ALV control: Tool tip for column header

Figure 10.7 Set Column Attributes Method Parameters

Listing 10.8 called the `SET_COLUMN_ATTRIBUTES` method several times, exporting the name of each report output column whose nature needed to be modified in some way. For each call, in addition to the column name, the code also exported one or more of the optional parameters, depending on what exactly it was that needed to be changed. Technically, the way the SALV class modifies report columns is quite different, depending on what’s being changed, but the calling report doesn’t need to know or worry about that.

Listing 10.9 shows the entire code of the `SET_COLUMN_ATTRIBUTES` method. You will see that this doesn’t do anything dramatic; it just delegates the various types of requested tasks to helper methods. (The helper methods are discussed next, so you can see how exactly the SALV class accomplishes each task.)

```

METHOD zif_bc_alv_report_view~set_column_attributes.
* Preconditions
  CHECK id_field_name IS NOT INITIAL.

  IF if_is_a_checkbox = abap_true.
    set_checkbox( id_field_name ).
  ENDIF.

  IF if_is_hotspot = abap_true.
    set_hotspot( id_field_name ).
  ENDIF.

  IF if_is_visible IS SUPPLIED.
    set_visible( id_field_name = id_field_name
                 if_is_visible = if_is_visible ).
  ENDIF.

  IF if_is_technical = abap_true.
    set_technical( id_field_name ).
  ENDIF.

  IF if_is_a_button = abap_true.
    set_column_as_button( id_field_name ).
  ENDIF.

  IF if_is_subtotal = abap_true.
    set_subtotal( id_field_name ).
  ENDIF.

  IF id_long_text IS NOT INITIAL.
    set_long_text( id_field_name = id_field_name
                  id_long_text = id_long_text ).
  ENDIF.

  IF id_medium_text IS NOT INITIAL.
    set_medium_text( id_field_name = id_field_name
                    id_medium_text = id_medium_text ).
  ENDIF.

  IF id_short_text IS NOT INITIAL.
    set_short_text( id_field_name = id_field_name
                   id_short_text = id_short_text ).
  ENDIF.

  IF id_tooltip IS NOT INITIAL.
    set_tooltip( id_field_name = id_field_name
                id_tooltip = id_tooltip ).
  ENDIF.

ENDMETHOD.

```

Listing 10.9 Custom ALV Class: SET_COLUMN_ATTRIBUTES Method

In Listing 10.10, you'll see the `SET_CHECKBOX` method, which will turn the field selected (which one would hope is defined as a character length of one) into a checkbox. Note that the checkbox will not change when the user clicks on it, unless you have coded this in the handler for `ON_LINK_CLICK`. Also note that an exception here should stop the report with a hard error, so hopefully testing will prevent the report even getting out of development until the problem is addressed. This is because an error here indicates a bug in the program; realistically, the only thing that could be going wrong here is that a column name that doesn't exist is being passed in. This is clearly an error; either the column name was misspelled when calling the method, or that particular column was deleted from the internal table containing the report data.

Chapter 7 introduced you to the concept of *design by contract*. If think you back to that discussion, you'll see that, though the exception handling is at the end of the method, this is a violated precondition—because the routine that called this routine has fed in nonsense data and thus is at fault.

```
METHOD set_checkbox.
* Local Variables
  DATA: lo_not_found      TYPE REF TO cx_salv_not_found,
        lf_error_occurred TYPE abap_bool.

TRY.
  mo_column ?= mo_columns->get_column( id_column_name ).

  CALL METHOD mo_column->set_cell_type
    EXPORTING
      value = if_salv_c_cell_type=>checkbox_hotspot.

  CATCH cx_salv_not_found INTO lo_not_found.
    lf_error_occurred = abap_true.
    "Object = Column
    "Key   = Field Name e.g. VBELN
    zcl_dbc=>require( id_that =
  |{ lo_not_found->object } { lo_not_found->key } must exist|
  if_true = boolc( lf_error_occurred = abap_false ) ).
ENDTRY.

ENDMETHOD. "Set Checkbox
```

Listing 10.10 SET_CHECKBOX Method

There is nothing complicated in the `SET_HOTSPOT` method in Listing 10.11—just creating an object and setting an attribute—but what you're avoiding are the

repetitive tasks of creating a COLUMNS object out of the SALV object, and then creating a COLUMN object out of the COLUMNS object, and then setting the attribute, again and again.

```
METHOD set_hotspot.

    TRY.
        mo_column ?= mo_columns->get_column( id_column_name ).

        CALL METHOD mo_column->set_cell_type
            EXPORTING
                value = if_salv_c_cell_type=>hotspot.

        CATCH cx_salv_not_found.
            "Raise fatal exception
    ENDTRY.

ENDMETHOD. "Set Hotspot
```

Listing 10.11 SET_HOTSPOT Method

Listing 10.12 shows the code for the SET_VISIBLE method. Note that all columns in the internal table passed into the SALV object during creation are visible by default; you have to pass ABAP_FALSE into this method to hide the ones you don't want to be seen on the initial screen, but the lucky old user can add them back in later if he so desires.

```
METHOD set_visible.

    TRY.
        mo_column ?= mo_columns->get_column( id_field_name ).
        mo_column->set_visible( if_is_visible ).
    CATCH cx_salv_not_found.
        "Raise Fatal Exception
    ENDTRY.

ENDMETHOD.
```

Listing 10.12 SET_VISIBLE Method

Listing 10.13 is for when you don't want the user to ever see the column in a report. Often, there are helper fields in your internal tables to aid in interim calculations. It would be painful to have two almost identical internal tables, one to prepare the data and one just with the display columns. That would take up twice as much memory, and you would have to go to the effort of moving data from one to the other.

```

METHOD set_technical.

    TRY.
        mo_column ?= mo_columns->get_column( id_field_name ).
        mo_column->set_technical( abap_true ).

        CATCH cx_salv_not_found.
            "Raise Fatal Exception
    ENDTRY.

ENDMETHOD.

```

Listing 10.13 SET_TECHNICAL Method

The code in Listing 10.14 is for when you want a column field to be an icon rather than a piece of data. This could be for information (e.g., a traffic light) or to really draw attention to certain rows that have incorrect data. This could also be an icon the user could drill down on (e.g., a picture of a house the user could click to pop up a box showing an address or a picture of a phone he could click on to initiate an outbound call to the customer that line refers to). Note that the underlying internal table field must have the appropriate definition that supports icons (e.g., data element `ICON_TEXT`).

```

METHOD set_column_as_button.

    TRY.
        mo_column ?= mo_columns->get_column( id_field_name ).
        mo_column->set_icon( if_salv_c_bool_sap=>true ).
        CATCH cx_salv_not_found.
            "Raise Fatal Exception
    ENDTRY.

ENDMETHOD.

```

Listing 10.14 SET_COLUMN_AS_BUTTON Method

In Listing 10.15, you define the subtotals in your SALV report. You will notice that a different object is used for subtotalling than was used in the previous examples about other application-specific changes. This is because, rather than telling a column object that it is to be subtotaled, you're telling a subtotal object what columns are to be subtotaled. The method encapsulates this complexity, hiding it from the calling program. Note also that there is another optional parameter of the standard method `ADD_AGGREGATION` called `AGGREGATION`, which defaults to a constant from interface `IF_SALV_C_AGGREATION`, saying that the type of subtotalling being performed is a straight total. You could also have minimum, maximum,

or average—but in 14 years I've never needed any of those in a report, so I've left my method set to use the default total value. (Now that I've said this, tomorrow a business user will come to me and say he really needs a column with the minimum value at the bottom.)

```
METHOD set_subtotal.

    mo_aggregations = mo_alv_grid->get_aggregations( ).

    TRY.
        mo_aggregations->add_aggregation( columnname = id_field_name ).
    CATCH cx_salv_not_found
          cx_salv_data_error
          cx_salv_existing.
        "Raise Fatal Exception
    ENDTRY.

ENDMETHOD.
```

Listing 10.15 SET_SUBTOTAL Method

You can also, of course, change the headings at the top of each column in the report. You may find that when you do change the name of the column, you end up changing the long, medium, and short texts to exactly the same string. You can define separate methods for changing the long, medium, and short texts in case they need to be different, but if the value is constant (i.e., short), you can just pass it in to the SET_LONG_TEXT method, and it changes all three values.

In Listing 10.16, you 'll see that if the string length is short enough, then the SET_LONG_TEXT method copies the value to the medium text and to the short text method, if it can. (The medium text and short text methods are not shown here, but they follow the same principle.)

```
METHOD set_long_text.
* Local Variables
    DATA: ld_medium_text TYPE scrtext_m,
          ld_short_text  TYPE scrtext_s.

    TRY.
        mo_column ?= mo_columns->get_column( id_field_name ).
        mo_column->set_long_text( id_long_text ).

        IF strlen( id_long_text ) LE 20.
            ld_medium_text = id_long_text.
            mo_column->set_medium_text( ld_medium_text ).
        ENDIF.
```



```

IF strlen( id_long_text ) LE 10.
  ld_short_text = id_long_text.
  mo_column->set_short_text( ld_short_text ).
ENDIF.

CATCH cx_salv_not_found.
  "Raise Fatal Exception
ENDTRY.

ENDMETHOD.

```

Listing 10.16 SET_LONG_TEXT Method

In Listing 10.17, you create the tooltip. You can pass in up to 40 characters to give the user some more detailed information about what exactly the column is showing.

```

METHOD set_tooltip.

  TRY.
    mo_column ?= mo_columns->get_column( id_field_name ).
    mo_column->set_tooltip( id_tooltip ).

    CATCH cx_salv_not_found.
      "Raise Fatal Exception
    ENDTRY.

  ENDMETHOD.

```

Listing 10.17 SET_TOOLTIP Method

Adding Sort Criteria

The idea of the `ADD_SORT_CRITERIA` method will be familiar to ABAP programmers; they're used to building up an internal table saying what columns in a report should be used as sort criteria and whether each such column should be subtotaled. Each different SAP UI technology has essentially the same concept but achieves this end in a different way. In this method, you need input parameters that can add a new sort criterion. These input parameters are shown in Figure 10.8.

The following list explains what each of the parameters in Figure 10.8 is for:

► ID_COLUMNNAME

First, the program needs to know which column you're talking about: sales order number, monster number, and so on.

► **ID_POSITION**

Then, you need to determine the priority. For example, if you want to sort by monster color first and then sort all monsters of the same color by sanity, the color has to have priority one and sanity priority two.

► **IF_DESCENDING**

Next, you need to say if you wish to sort the values in ascending or descending order. Most often, the vast majority of data is sorted in ascending order in a report.

► **IF_SUBTOTAL**

Then, you need to say if a change of value of the column will trigger a subtotal for numeric fields (e.g., does the report say how many green monsters there are).

► **ID_GROUP**

If you don't want subtotals, then you can put a line or a page break in the report output when the value of the field changes by using the `group` import criteria. You use the constant `IF_SALV_C_SORT=>GROUP_UNDERLINE` to say you want a line and the constant `IF_SALV_C_SORT=>GROUP_NEWPAGE` for a page break.

► **IF_OBLIGATORY**

If for some reason you do not want your users to be able to override the sort criteria you've defined programmatically, then you can set the `obligatory` parameter to lock them in place. (However, this is not a feature you would ever normally use.)

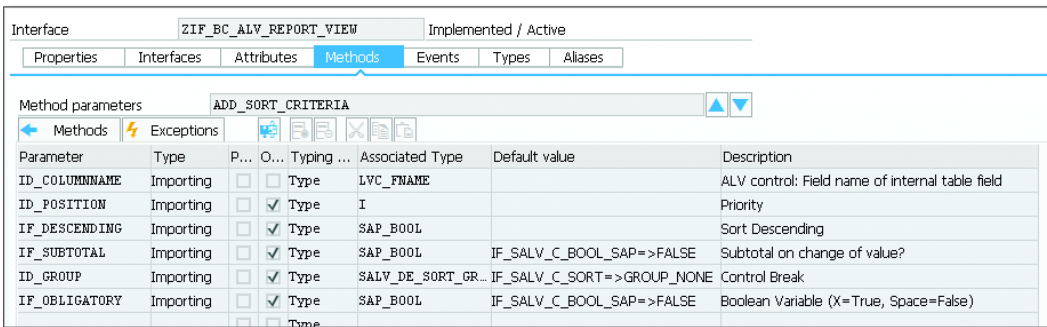


Figure 10.8 Add Sort Criteria Method Parameters

You usually want to have a default sort order for the report data that's independent of how the internal table might be sorted when it gets passed into the report.

The code in Listing 10.18 takes the same parameters as the standard SALV method and relieves the calling program of the burden of having to declare an object to do the sorting.

```
METHOD add_sort_criteria.
* Local Variables
DATA: ld_sequence TYPE salv_de_sort_sequence.

IF if_descending = abap_true.
    ld_sequence = if_salv_c_sort=>sort_down.
ELSE.
    ld_sequence = if_salv_c_sort=>sort_up.
ENDIF.

TRY.
    mo_sorts = mo_alv_grid->get_sorts( ).

    mo_sorts->add_sort( columnname = id_columnname
                       position    = id_position
                       sequence    = ld_sequence
                       subtotal    = if_subtotal
                       group       = id_group
                       obligatory  = if_obligatory ).

CATCH cx_salv_data_error
      cx_salv_existing
      cx_salv_not_found.
    "Raise Fatal Exception
ENDTRY.
```

Listing 10.18 Custom SALV Class: Adding a Sort Criteria

10.2.4 Report Flow Step 4: Displaying the Report (Generic)

With all ALV-type technologies, from the ALV function modules through to CL_SALV_TABLE, when you're finished defining your report settings, you shout "Go!" and the framework you are using takes charge of displaying the report and sends back information about things the user has done (e.g., double-clicking a cell), which the calling program can respond to if it so desires.

In a standard UI interface, you need a method to tell the framework that you are done defining the report settings and that it's time to display the report. Then, you need an event to pass information about what the user has done from the UI framework to the calling program. Finally, in case the action the user has taken has somehow caused the report data to change, you need a `refresh` method to show the user that the data has in fact changed.

Creating the Display Method

This method has no parameters; what you're saying here is that when this method is invoked, you've finished telling the view the specifics of how you want the report to appear, and now is the time to show the report to the user.

Listing 10.19 is possibly the least exciting code sample in the world!

```
METHOD zif_bc_alv_report_view~display.

    mo_alv_grid->display( ).

ENDMETHOD.
```

Listing 10.19 DISPLAY Method

Defining the User Command Received Event

As mentioned a few sentences ago, once the SALV framework (or whatever framework you use) has picked up the report ball and is running with it (i.e., showing it to the user), the framework sends back events to the calling program when the user does something to the report. Your interface (which can work with multiple UI technologies) needs to have an event of its own to pick up the event in the form in which the given UI technology raises that event and to resend that event in a more generic format. In this way, the calling program doesn't need to change when the UI technology hiding behind the interface changes.

In Figure 10.9, you can see the single event defined in the interface. The one and possibly only event you can be certain of is that the user will want to click on some part of the report output and expect something to happen in response.

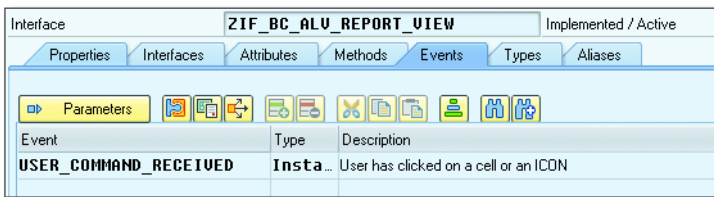


Figure 10.9 View Interface: Events

The parameters that can be seen in Figure 10.10 reflect the fact that there are two possible situations. The first situation is that the user could have clicked a button at the top of the screen without selecting a row; in this case, parameter `ED_USER_COMMAND` will be filled whereas `ED_ROW` and `ED_COLUMN` will be blank.

Parameters	Op...	Typing	Associated Type	Default value	Description
ED_USER_COMMAND	<input checked="" type="checkbox"/>	Type	SALV_DE_FUNCTION		ALV Function
ED_ROW	<input checked="" type="checkbox"/>	Type	SALV_DE_ROW		Line
ED_COLUMN	<input checked="" type="checkbox"/>	Type	SALV_DE_COLUMN		Column

Figure 10.10 User Command Received Event Parameters

However, the more common situation is that the user double-clicks a specific cell in the report. In this case, parameters `ED_ROW` and `ED_COLUMN` will be filled with the coordinates of the cell that was clicked on and `ED_USER_COMMAND` will be filled with a system generated constant; it's the wittily named `&IC1`, which is the value the SAP system automatically sets when a user double-clicks something. In this case, the row number and column number become vitally important so that the part of the program that deals with the user action can know exactly what the user wants dealt with (e.g., which sales order—or monster—to display).

Note that this is not about responding to what the user has done; this is just about broadcasting the fact that the user has done something. The problem of how to respond to that something is the responsibility of another part of the program. The view is ideally placed to recognize such an event, because only the view knows exactly what the screen looks like and what technology is being used to interact with the user. Once the view has raised such an event, its work here is done.

As mentioned earlier, it's not the job of the view to respond to any commands the user makes; its job is to pass the information about what the user just did on to the parts of the program best suited to deciding what the proper response should be. Half the articles on MVC say the view should talk directly to the model, and half say the view and the model should know nothing about each other and only communicate via the controller. In the latter case, it becomes easy to replace the view without affecting the model (e.g., when a new user interface technology comes along) and likewise to use the same view for various similar models (e.g., you have lots of different models for approving various things, like leave requests and purchase orders, and you want them all to have the same look and feel, so you use the same view each time).

With the ABAP event mechanism, it doesn't matter which of the two approaches is taken; the view raises the event that says the user has done something, and it's up to the programmer to decide which parts of the program respond to this event. My recommendation is to have the controller react to an event from the view; the controller does something with that information if it wants, and then it passes the event on to the model to see if the model wants to react in some way.

In any event, all that the event handling methods in the view do is to pass on the event raised by the SALV object after user interaction. Figure 10.11 and Figure 10.12 show the parameters of the event handling methods in your view class.

Parameter	Type	Pa...	Dp...	Typing Met...	Associated Type	Default value	Description
ROW	Importing	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type	SALV_DE_ROW		Zeile
COLUMN	Importing	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type	SALV_DE_COLUMN		Spalte

Figure 10.11 Handle Link Click Method Parameters

Parameter	Type	Pa...	Dp...	Typing Met...	Associated Type	Default value	Description
E_SALV_FUNCTION	Importing	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Type	SALV_DE_FUNCTION		ALV Funktion

Figure 10.12 Handle User Command Method Parameters

You'll notice that every time SAP comes out with a new technology to do the same thing, fields that do similar tasks are renamed and/or have their lengths changed. The obvious example is the dozen or so structures that are used to store error messages: all have the message class, message number, and message variables and all are named slightly differently (e.g., NUMBER in BAPIRET2, MSGNR in BDCMSGCOLL, and MSGNO in SYST, all of which store the same value).

In the same way, each SAP UI technology passes on the same three pieces of information: the user command chosen, the row chosen (if applicable), and the column chosen (if applicable). For every technology, the data elements used to store those values have different names and different lengths.

In order to make life easier for the calling program, you want to use the *adapter pattern*; the subclass has event handler methods that accept parameters in the

format that the technology the subclass is dedicated to uses, and these parameters are adapted to the format that the interface event uses. Because the calling program only knows about the interface, it's blithely unaware that different technologies use different formats to store rows and columns.

The code in Listing 10.20 and Listing 10.21 handle the events raised when a user interacts with the SALV report. In both cases, they adapt the parameters of the data the view works with into a uniform data format and then pass the event onwards, where it will be picked up by the controller.

```
METHOD handle_link_click.
* No type conversions needed
  RAISE EVENT user_command_received
  EXPORTING ed_user_command = '&IC1'
           ed_row           = row
           ed_column        = column.
ENDMETHOD. "Handle Link Click
```

Listing 10.20 HANDLE_LINK_CLICK Method

```
METHOD handle_user_command.
* Local Variables
  DATA: ld_command TYPE sy-ucomm.

* Convert view specific data type to generic data type
  ld_command = e_salv_function.

  RAISE EVENT user_command_received
  EXPORTING ed_user_command = e_salv_function.

ENDMETHOD. "Handle User Command
```

Listing 10.21 HANDLE_USER_COMMAND Method

With the `HANDLE_LINK_CLICK` method, you're not doing any type conversions, because you're using the SALV definitions as the baseline. For the user command, however, the generic data element is set to be defined as `SY-UCOMM`, because (a) that is the system field that always has the user command written to it automatically in every single SAP program and (b) it just feels right. Who knows why the authors of the SALV object felt the need to define their user command data element differently—but they did, so this needs to be adapted to prevent a runtime error, even though both definitions are 70 characters long. (Perhaps SAP disallows treating data elements of the same length identically in routine parameters in case one data element definition changes down the track.)

Refreshing the Display Method

The code for refreshing the `DISPLAY` method is self-explanatory. Often, a user action changes the data (e.g., the user chose a purchase order from a list of unreleased purchase orders and released it, and so you want to remove the processed item from the list).

The actual changing of the database is not the job of the view, but after the application model makes the change, it raises a *data changed* event, which causes the controller to tell the view to call this `refresh` method so that the user sees the updated values in the report.

The `REFRESH` method shown in Listing 10.22 (which you want to call when the report data has changed for whatever reason) is a bit meatier than the code to display the report, but just as simple. The important thing here is to make the refresh stable—so even though the data may change, the user's cursor stays where it is, which is not the default system behavior.

```
METHOD zif_bc_alv_report_view~refresh_display.
* Local Variables
  DATA: ls_stable_refresh TYPE lvc_s_stbl.

"I am going to be a madman and suggest that when a user refreshes
"the display because data has changed, they want the cursor to
"stay where it is as opposed to jumping six pages up to the start
"of the report, which is the default behavior
  ls_stable_refresh-row = abap_true.
  ls_stable_refresh-col = abap_true.

  mo_alv_grid->refresh( s_stable = ls_stable_refresh ).

ENDMETHOD.
```

Listing 10.22 REFRESH_DISPLAY Method

Future Proofing

The logical extension of the ability to wrap the SALV in the standard interface is that if and when SAP comes up with the successor to the SALV reporting framework, all you'll need to do is create a new subclass that implements the generic view interface, create implementations for each of the interface methods to deal with the specific way the new technology will deal with each task, and then go around changing existing reports to use the new technology at the drop of a hat—rather than having to rewrite them all.

10.3 Adding Custom Command Icons Programmatically

One thing you almost always want when creating a report program is to have a few of your own icons at the top of the screen to perform application-specific tasks. For example, you may have a report that shows a list of monsters, and you want the user to be able to select a monster using a checkbox and then click the RAMPAGE icon at the top of the screen to send the selected monster into a blind frenzy, destroying everything in its path and becoming so angry that you lose control of it and it kills every single living thing on the planet, including you. That's a reasonably common user requirement.

What you *do not* want if at all possible is to have to paint icons on some sort of screen. This is what you have to do when using the REUSE_ALV function modules; if you want to add extra icons at the top of the screen to respond to application-specific functions (like rampage), then you have to create a custom STATUS and change it each time you have a new command. (This isn't the end of the world and you can exclude any functions you don't want at runtime, but it still isn't very elegant.)

CL_SALV_TABLE cannot say at runtime what icons you want programmatically. With CL_GUI_ALV_GRID, there's no need for a custom STATUS at all. If you want extra commands in the SALV reporting framework, then you can send a list of icons programmatically to the view in a pop-up box—but not in the default full screen view, which is the one you usually want for a report. In the full screen case, you have to have a custom STATUS, which is a quantum leap backwards and contrary to the whole OO initiative SAP is pushing.

I don't take this sort of thing lying down, so I thought to myself, "I need a container in order to add commands to my SALV report, and I don't want to have to manually create that container for each report." Accordingly, this section walks you through the steps needed to change your custom ZCL_BC_VIEW_SALV_TABLE class and calling program such that you can add custom icons programmatically. The steps in this process are as follows:

1. Create a method to automatically create a container.
2. Add a method to fill the container object needed by the SALV with the container just created; this also involves adding an interface to your view class and creating a function module.

3. Change the `INITIALIZE` method to pass in the container object to the SALV `factory` class.
4. Code the method that adds a list of user commands received from the calling program to the SALV report object.
5. Pass in the list of user commands from the calling program.

This process of creating a screen automatically is going to seem quite complicated; it involves calling a method, which calls a function module, which calls another method; it's enough to make your head spin. Nonetheless, this only has to be set up once, and all of this complexity is hidden from the calling program forevermore.

If you want the screen and associated container to magically create themselves, then first you split the initialization into two steps: creating such a screen (Section 10.3.1) and then the rest of the initialization steps you normally have to do manually in each report (Section 10.3.2 and Section 10.3.3). Then, once the screen and container are in place, you move on to looking at how the SALV-specific view class goes about programmatically adding new custom commands to the toolbar (Section 10.3.4) and how the calling program tells the view what custom commands it wants the view to display (Section 10.3.5).

10.3.1 Creating a Method to Automatically Create a Container

In order to automatically create a container, you have to create a method called `CREATE_CONTAINER_PREP_DISPLAY` and add it to the concrete `ZCL_BC_VIEW_SALV_TABLE` class, which is going to be invoked from the calling program instead of the usual `INITIALIZE` method.

The first half of Listing 10.23 puts the input parameters into global (to the class instance) variables, and then a function module is called, the purpose of which revolves around creating a screen and container. The call needs to be to a function module, because function modules can call screens, but methods cannot.

```
METHOD create_container_prep_display.

    md_report_name      = id_report_name.
    ms_variant-report   = id_report_name.
    ms_variant-variant  = id_variant.
    mt_user_commands[] = it_user_commands[].
```

```
CREATE DATA mt_data_table LIKE ct_data_table.  
GET REFERENCE OF ct_data_table INTO mt_data_table.  
  
CALL FUNCTION 'ZSALV_CSQT_CREATE_CONTAINER'  
  EXPORTING  
    r_content_manager = me  
    title              = id_title.  
  
ENDMETHOD.
```

Listing 10.23 Creating a Container Automatically

10.3.2 Changing ZCL_BC_VIEW_SALV_TABLE to Fill the Container

There is a standard SAP function module called `SALV_CSQT_CREATE_CONTAINER` that automatically creates a screen and a container. Make an identical Z copy of this function module and then make one change: select the `WITHOUT APPLICATION TOOLBAR` checkbox so that the resulting screen doesn't have an ugly hole at the top. The code in Listing 10.23 makes a call to this new Z function module, passing in the calling class so that the function knows where to return control to.

The function module has one big screen, on which is a big container that fills the whole screen. The function module creates an object to represent that container, and then in that screen's PBO processing a method call is made to the class that called the function module (i.e., `ZCL_BC_VIEW_SALV_TABLE`).

`ZCL_BC_VIEW_CLASS` cannot respond to the call from within `ZSALV_CSQT_CREATE_CONTAINER` unless it implements interface `IF_SALV_CSQT_CONTENT_MANAGER` (which has the somewhat enigmatic description “manages content”), so you need to add this interface to class `ZCL_BC_VIEW_SALV_TABLE`. When added, this interface brings with it a method called `FILL_CONTAINER_CONTENT`.

Because that class has the `FILL_CONTAINER_CONTENT` method, it can receive the newly created container object passed in from the function module. This enables the SALV framework to show the report on the screen inside the function module.

The `FILL_CONTAINER_CONTENT` code shown in Listing 10.24 receives a container object, `R_CONTAINER`, and then passes this object on to a method that orchestrates the rest of the initialization steps—almost the exact same call as was made in Section 10.1.2 in Listing 10.1 in order to call a full screen report, but this time you're passing in a container object as well.

```

METHOD if_salv_csqt_content_manager~fill_container_content.
*-----*
* This gets called from function SALV_CSQT_CREATE_CONTAINER PBO
* module
* which creates a screen and a container, and passes us that container
*-----*

* Local Variables
FIELD-SYMBOLS: <lt_data_table> TYPE ANY TABLE.

ASSIGN mt_data_table->* TO <lt_data_table>.

prepare_display_data(
  EXPORTING
    id_report_name      = md_report_name " Calling program
    id_variant          = ms_variant-variant
    io_container        = r_container
    it_user_commands    = mt_user_commands
  CHANGING
    ct_data_table      = <lt_data_table> ). " Data Table

ENDMETHOD.

```

Listing 10.24 FILL_CONTAINER_CONTENT Method

10.3.3 Changing the INITIALIZE Method

Now, you have to make some changes in the INITIALIZE method, as shown in Listing 10.25, to accommodate the facts that (a) you now have a container being passed in and (b) you want to add some user commands programmatically

The table of user commands is passed in from the calling program via the CREATE_CONTAINER_PREP_DISPLAY method, which fills up global table MT_USER_COMMANDS before calling the function module.

In the FILL_CONTAINER_CONTENT method, this table is passed into the PREPARE_DISPLAY_DATA method, which in turn calls the INITIALIZE method.

Exhausted after all this travel, the table of user commands finally arrives inside the INITIALIZE method in the form of importing parameter table IT_USER_COMMANDS.

In Listing 10.25, if you have a container, then the factory method of CL_SALV_TABLE is called, this time passing in the container object. Next, the program has to specifically ask for the basic toolbar to appear at the top of the screen; SALV will not do this automatically.

Finally, the program calls a method to add the table of your custom commands to the toolbar.

```

*-----*
* If we have a container, then we can add our own user-defined
* commands programmatically, but we cannot edit the data
*-----*
    IF io_container IS SUPPLIED AND
       io_container IS BOUND.
       cl_salv_table=>factory(
           EXPORTING
               r_container = io_container
           IMPORTING
               r_salv_table = mo_alv_grid
           CHANGING
               t_table      = ct_data_table[] ).

       display_basic_toolbar( ).
       IF it_user_commands[] IS NOT INITIAL.
           add_commands_to_toolbar( it_user_commands ).
       ENDIF.

```

Listing 10.25 Revised INITIALIZE Method

10.3.4 Adding the Custom Commands to the Toolbar

Listing 10.26 shows how you actually pass the list of user commands into the `MO_FUNCTIONS` object of SALV, which is in charge of keeping track of any user-defined functions that have been added. For each command, you pass in what the icon should look like, a text description, and a tooltip (if you so desire).

```

METHOD add_commands_to_toolbar.
* Local Variables
    DATA: ls_commands LIKE LINE OF it_commands,
           ld_icon      TYPE string,
           ld_tooltip   TYPE string,
           ld_text      TYPE string.

    TRY.
        LOOP AT it_commands INTO ls_commands.
            CHECK ls_commands-function <> '&IC1'.
            ld_icon      = ls_commands-icon.
            ld_text      = ls_commands-text.
            ld_tooltip   = ls_commands-quickinfo.
            mo_functions->add_function(
                name = ls_commands-function
                icon  = ld_icon
                text  = ld_text
            ).
        ENDLOOP.
    CATCH cx_sy_zif_unavailable.

```

```

                                tooltip = ld_tooltip
                                position = if_salv_c_function_
position=>right_of_salv_functions ).
    ENDLLOOP.

    CATCH cx_salv_wrong_call.
        "Raise Fatal Exception
    CATCH cx_salv_existing.
        "Raise Fatal Exception
    ENDTRY.

ENDMETHOD.

```

Listing 10.26 Adding Commands Programmatically

Sadly, it's impossible to add a separator line programmatically like you can with `CL_GUI_ALV_GRID` to make your new commands distinct from the standard ones. At one point, rumors abounded that this was somehow going to be made possible as of a certain enhancement pack level, but I have come to the conclusion this is not going to happen. A separator line does appear all by itself, but as a programmer I like control; when the artificial intelligences take over the world (as predicted by Professor Stephen Hawking and Bill Gates), I'm not going to be happy.

10.3.5 Sending User Commands from the Calling Program

When setting up the various elements that make each report unique, if you have actions a user can take specific to this report (i.e., user commands above and beyond the standard SAP-supplied ones), then the calling program will send a list of these extra user commands to the view. Usually, this list originates from the model and makes its way to the view via the controller—but if you wanted to add a custom user command directly from the controller, then you would use something like the code in Listing 10.27.

```

DATA: lt_user_commands TYPE zty_ttb_button,
      ls_user_commands TYPE zsbc_stb_button.

CLEAR ls_user_commands.
ls_user_commands-function = 'ZZZZ'.
ls_user_commands-icon    = icon_phone.
ls_user_commands-butn_type = 0. "Normal Button
ls_user_commands-text    = 'My New Command'.
APPEND ls_user_commands TO lt_user_commands.

lo_view->create_container_prep_display(
EXPORTING

```

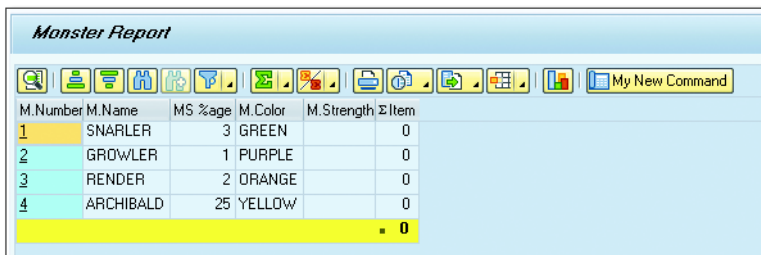
```

id_title           = 'Monster Report'
id_report_name    = ld_repid " Calling program
id_variant        = p_vari " Display Variant specified by user
it_user_commands  = lt_user_commands
CHANGING
ct_data_table     = gt_monsters ).

```

Listing 10.27 Method in the Model Class to Define User Commands

Figure 10.13 shows the end result. You've added a new user command button programmatically to a normal full screen SALV report without having to manually create a new screen or a new custom STATUS.



M.Number	M.Name	MS %age	M.Color	M.Strength	ΣItem
1	SNARLER	3	GREEN		0
2	GROWLER	1	PURPLE		0
3	RENDER	2	ORANGE		0
4	ARCHIBALD	25	YELLOW		0
					0

Figure 10.13 Monster Report with New Button

10.4 Editing Data

In this section, you're once again going to look at a (very) common user requirement, realize that `CL_SALV_TABLE` cannot cut the mustard on its own, and work out a way to plug the gap.

Once upon a time, there was a clear separation between reports (which showed end users lists of data) and transactions (which enabled end users to be able to change that data). These days, most business users will not settle for that. If they see a list of data, then they want to be able to interact with that data: remove billing blocks, change the price, add an extra head to a monster, and so forth. As a result, over the last decade there have been a lot of SAP GUI reports moving from read-only to being able to change the data—technically using an editable ALV grid.

Business users have become used to this, and they *love it*. It makes their lives so much easier, because they no longer have to go into 5,400 contracts individually

to make changes one by one. Using an interactive ALV grid is almost as easy as (gasp) a spreadsheet!

Here, SAP is very clear: with `CL_SALV_TABLE`, you cannot have an editable grid. It is impossible. It cannot be done. From a purely technical point of view, this seems really strange, because at the heart of `CL_SALV_TABLE` there's an embedded instance of `CL_GUI_ALV_GRID`, which of course can be editable. The problem (for us programmers) is that the business will come to you with a request for a read-only report, you'll program it using the SALV reporting framework (as SAP recommends), and right at the end—just before you think the project is over—the business will say, “Oh, and can we make a few columns editable?”

SAP's official position is that you should always use `CL_GUI_ALV_GRID`, not the SALV framework, for this purpose. But here's a little secret: you can make the SALV object editable.

In order to do so, you have to go through the following steps:

1. You need to create custom class `ZCL_SALV_MODEL`, inheriting from the standard SAP class but able to export the underlying grid object from a linked SALV report object.
2. Next, the `INITIALIZE` method of `ZCL_BC_VIEW_SALV_TABLE` is changed to link together the instance of the SALV report object and the custom `ZCL_SALV_MODEL` class.
3. Then, you code a method in the `ZCL_SALV_MODEL` to extract the underlying grid object from the linked SALV report object.
4. Next, you can change the calling program to invoke the report using a method that automatically creates a container; you want to programmatically change the user commands to add an `edit_me` command.
5. Finally, you bring all this together and code some user command handling such that when a user clicks a `MAKE THIS EDITABLE` button, he can in fact edit the data.

10.4.1 Creating a Custom Class to Hold the Standard SALV Model Class

In order to make a SALV object editable, you need to be able to access the underlying `CL_GUI_ALV_GRID` instance. Where that grid object normally lives is in standard

SAP class `CL_SALV_MODEL_LIST`, but the problem is that the grid instance is private and thus cannot be read.

To get around this, create a new class: `ZCL_SALV_MODEL`, inheriting from `CL_SALV_MODEL_LIST`. Then give this new class a private attribute, `MO_MODEL`, which is typed as an instance of `CL_SALV_MODEL`. This attribute is populated during construction, as shown in Listing 10.28. The actual value coming in will be a `model` object that knows all about the SALV report object.

```
METHOD constructor.
    super->constructor( ).
    mo_model = io_model.
ENDMETHOD.
```

Listing 10.28 Adding the Standard SALV Model to a Custom Class

10.4.2 Changing the Initialization Method of `ZCL_BC_VIEW_SALV_TABLE`

Section 10.4.1 explained the process of adding an attribute, `MO_SALV_MODEL`, to the custom SALV view class. Now change the end of the `INITIALIZE` method in order to link the `CL_SALV_TABLE` instance with the `CL_SALV_MODEL` instance object (Listing 10.29).

```
set_handlers( ).

DATA: lo_salv_model TYPE REF TO cl_salv_model.

"Narrow casting
"CL_SALV_MODEL is a superclass of CL_SALV_TABLE
"Target = LO_SALV_MODEL = CL_SALV_MODEL
"Source = MO_ALV_GRID = CL_SALV_TABLE
lo_salv_model ?= mo_alv_grid.

"Object to access underlying CL_GUI_ALV_GRID
CREATE OBJECT mo_salv_model
    EXPORTING io_model = lo_salv_model.
```

Listing 10.29 Changing the `INITIALIZE` Method to Store an ALV Grid Object

10.4.3 Adding a Method to Retrieve the Underlying Grid Object

In the class `ZCL_SALV_MODEL`, add a method called `GET_ALV_GRID`, which returns an instance of `CL_GUI_ALV_GRID`. In Listing 10.30, the instance `MO_MODEL` is linked to the SALV report object. You now have a lovely MVC construct. The view in this

case is the `CL_GUI_ALV_GRID` at the heart of a SALV object, the model is the SALV object itself, and the controller knows whether the report is running in full screen mode or in container mode.

In this case, the report is running in container mode and always will be, but nonetheless, because the theoretical possibility exists that it might not be, you can only access the underlying grid object by passing the controller object through two adapters before you can extract the underlying grid object. The end result is a `CL_GUI_ALV_GRID` instance that hides within `CL_SALV_TABLE`.

```
METHOD get_alv_grid.
* Local Variables
  DATA: lo_grid_adapter TYPE REF TO cl_salv_grid_adapter,
         lo_controller   TYPE REF TO cl_salv_controller_model,
         lo_adapter      TYPE REF TO cl_salv_adapter.

  lo_controller = mo_model->r_controller.

  "Target = LO_ADAPTER type CL_SALV_ADAPTER
  "Source = R_ADAPTER type CL_SALV_ADAPTER
  "R_ADAPTER may actually be a subclass however
  lo_adapter ?= lo_controller->r_adapter.

  "We are in container mode
  lo_grid_adapter ?= lo_adapter.
  ro_alv_grid = lo_grid_adapter->get_grid( ).

ENDMETHOD.
```

Listing 10.30 Returning an ALV Grid Object from `ZCL_SALV_MODEL`

10.4.4 Changing the Calling Program

In order to let the user make the cells editable, you need a user command to be added programmatically. As explained in Section 10.3, that means you need a container. Therefore, in the example monster report that uses the custom SALV class, now create the SALV object (called `GO_VIEW` in the code) with reference to a container (Listing 10.31).

```
"No custom status, added user commands
"Editing possible
go_view->create_container_prep_display(
EXPORTING
  id_title      = 'Monster Report'
  id_report_name = gd_repid " Calling program
  id_variant    = p_vari "Variant specified by user
```

```

        it_user_commands = lt_user_commands
CHANGING
        ct_data_table     = gt_monsters ).

```

Listing 10.31 Creating a SALV Report Using a Container

10.4.5 Coding User Command Handling

Next, you need to change the user command handling to make the grid editable. In this case, you want the user command handling to be in the controller part of the program, reacting to a user command that the view (SALV grid) has sent it.

In Listing 10.32, the user has clicked a button to make the grid editable, so the parameter `ED_USER_COMMAND` has the value `ZEDIT`. First, a method of `ZCL_BC_VIEW_SALV_TABLE` is called (`GET_ALV_GRID`). This method returns an instance of `CL_GUI_ALV_GRID`, which hides inside the SALV report object. Then, the program gets the field catalog (list of columns in the report) and controls which ones are to be made editable. The important point is that the internal table that stores the display data has to have a column called `CELLTAB`, defined as `celltab TYPE lvc_t_styl`. You need this in each row of your table to define what cells in that row can be made editable, because this setting is made on a cell-by-cell basis.

```

METHOD on_user_command.
*-----*
* FOR EVENT added_function OF cl_salv_events
* IMPORTING ed_user_command
*           ed_row
*           ed_column.
*-----*
* Local Variables
DATA: lo_alv      TYPE REF TO cl_gui_alv_grid,
      ls_layout  TYPE lvc_s_layo,
      ls_celltab TYPE lvc_s_styl,
      lf_valid   TYPE abap_bool ##needed,
      lt_fcat    TYPE lvc_t_fcat,
      ld_answer  TYPE char01.

FIELD-SYMBOLS: <ls_output> TYPE g_typ_over_ride,
               <ls_celltab> TYPE lvc_s_styl.

CASE ed_user_command.
  WHEN 'ZEDIT'. "A command to make the grid editable

      lo_alv = mo_view->get_alv_grid( ).

      IF lo_alv IS NOT BOUND.

```

```

        RETURN.
    ENDIF.

    lo_alv->get_frontend_fieldcatalog(
        IMPORTING
            et_fieldcatalog = lt_fcat ).

    make_column_editable( :
        EXPORTING id_column_name = 'MONSTER_HATS'
        CHANGING ct_fcat = lt_fcat ),
        EXPORTING id_column_name = 'MONSTER_HEADS'
        CHANGING ct_fcat = lt_fcat ),

    lo_alv->set_frontend_fieldcatalog( lt_fcat ).

    ls_layout-stylefname = 'CELLTAB'.

    lo_alv->set_frontend_layout( ls_layout ).

    lo_alv->refresh_table_display( ).

    cl_gui_cfw=>flush( ).

    WHEN etc etc

```

Listing 10.32 User Command to Make a SALV Grid Editable

Naturally, now you need to code the custom methods called by Listing 10.32. First, code the `GET_ALV_GRID` method in the `ZCL_BC_VIEW_SALV_TABLE` class (Listing 10.33), which has a single returning parameter typed as an instance of `CL_GUI_ALV_GRID`. The method gets this instance by calling the equivalently named method `GET_ALV_GRID` of the custom `ZCL_SALV_MODEL` class that was defined in Section 10.3.3.

```

METHOD get_alv_grid.
    * We have a container, so
    * then the grid needs to be retrieved via the adapter
    ro_alv_grid = mo_salv_model->get_alv_grid( ).

ENDMETHOD.

```

Listing 10.33 `GET_ALV_GRID` of `ZCL_BC_VIEW_SALV_TABLE` Method

Next, back in the controller, define the `MAKE_COLUMN_EDITABLE` method, which does the dirty work of changing the field catalog so that cells can be edited. This code can be seen in Listing 10.34. Here, you need to loop through the internal

table containing the data to be displayed, and for each selected column fill up the `CELLTAB` table. If the column is read-only, then make it editable, and vice-versa. Then, you also need to set the `EDIT` component of the field catalog for the selected column to `TRUE`. These two settings work in hand in hand to make the desired cells editable.

```

METHOD make_column_editable.
*-----*
* IMPORTING id_column_name TYPE dd031-fieldname
* CHANGING ct_fcat          TYPE lvc_t_fcat.
*-----*
* Local Variables
  DATA : ls_celltab TYPE lvc_s_styl.

  FIELD-SYMBOLS: <ls_output> TYPE g_typ_over_ride,
                 <ls_celltab> TYPE lvc_s_styl.

  LOOP AT mo_model->mt_output_data ASSIGNING <ls_output>.

    READ TABLE <ls_output>-celltab ASSIGNING <ls_celltab>
    WITH KEY fieldname = id_column_name.

    IF sy-subrc <> 0.
      ls_celltab-fieldname = id_column_name.
      APPEND ls_celltab TO <ls_output>-celltab.
      READ TABLE <ls_output>-celltab ASSIGNING <ls_celltab>
      WITH KEY fieldname = id_column_name.
    ENDIF.

    IF <ls_celltab>-style EQ cl_gui_alv_grid=>mc_style_enabled.
      <ls_celltab>-style = cl_gui_alv_grid=>mc_style_disabled.
    ELSE.
      <ls_celltab>-style = cl_gui_alv_grid=>mc_style_enabled.
    ENDIF.

  ENDLOOP.

  FIELD-SYMBOLS: <ls_fcat> LIKE LINE OF ct_fcat.

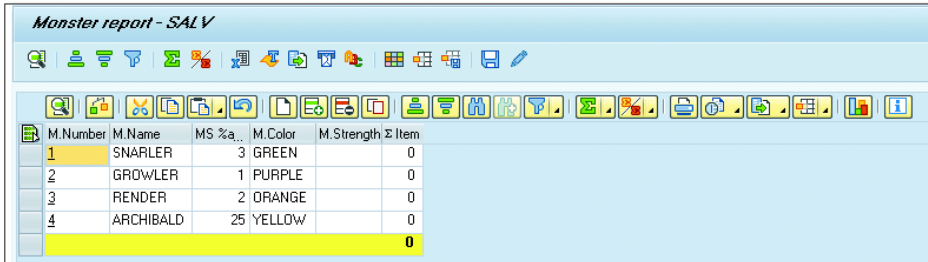
  LOOP AT ct_fcat ASSIGNING <ls_fcat>
  WHERE fieldname = id_column_name.
    <ls_fcat>-edit = abap_true.
  ENDLOOP.

ENDMETHOD. "MAKE_COLUMN_EDITABLE

```

Listing 10.34 MAKE_COLUMN_EDITABLE Method

At this point, everything is complete. If you run the report and click the **MAKE DATA EDITABLE** button, then (as Figure 10.14 shows) the grid becomes editable.



M.Number	M.Name	MS %a...	M.Color	M.Strength	Σ Item
1	SNAPLER	3	GREEN		0
2	GROWLER	1	PURPLE		0
3	RENDER	2	ORANGE		0
4	ARCHIBALD	25	YELLOW		0
					0

Figure 10.14 Editable SALV Grid

Note

In fact, there should be no need for a workaround like this, because since February 8, 2008, the SAP Community Network has been petitioning SAP to make `CL_SALV_TABLE` editable by default, but to no avail. (I've proposed making February 8 International Editable SALV Day and making a big fuss about this topic on this day each year.)

10.5 Handling Large Internal Tables with `CL_SALV_GUI_TABLE_IDA`

You may have presumed that ALV in all its forms, even new ones like SALV, is dead in the water, that in the future all our UIs will use SAPUI5 or some such, and that the poor old ALV will be ancient history. Although that may well be the case a long time down the track, the fact is that currently it's likely that all your reports will be based on ALV technology of one sort or the other and run in the SAP GUI.

In recognition of this, SAP has not stood still in regard to innovation in the area of SALV and has come out with the concept of SALV with IDA, where "IDA" stands for *integrated data access*. A new class, `CL_SALV_GUI_TABLE_IDA`, has been created for the purpose of dealing with really large internal tables (more than 100,000 rows of data). Traditional programs get into trouble dealing with so much data at once. You might run out of memory, and paging down will most probably cause a delay; even with small tables, an ALV grid sometimes goes all white for a second as you page down.

With the new class, not only do you outsource the creating of the column definitions to SALV, you also outsource the data retrieval. Normally, you fill up an internal table with a `SELECT` statement and then pass that internal table to SALV. Now, you skip coding the data retrieval yourself and instead tell SALV what table you want and what selection parameters you want (see Listing 10.35).

```
"Tell the ALV what database table we want
DATA(lo_alv_display) = cl_salv_gui_table_ida=>create(
iv_table_name = 'ZT_MONSTERS' ).
"Tell the ALV the select-options
DATA(lo_collector) = NEW cl_salv_range_tab_collector( ).
lo_collector->add_ranges_for_name(
iv_name = 'MONSTER_NUMBER' it_ranges = s_mnum[] ).
lo_collector->add_ranges_for_name(
iv_name = 'MONSTER_NAME' it_ranges = s_mnam[] ).
lo_collector->get_collected_ranges(
IMPORTING et_named_ranges = DATA(lt_name_range_pairs) ).
lo_alv_display->set_select_options(
it_ranges = lt_name_range_pairs ).
"Off we go!
lo_alv_display->fullscreen( )->display( ).
```

Listing 10.35 Coding the SALV with IDA with Selection Criteria

If you have no selection parameters (not very likely), then you can code an entire SALV-based ALV report in one line, as shown in Listing 10.36.

```
cl_salv_gui_table_ida=>create( iv_table_name = 'ZT_MONSTERS' )
->fullscreen( ) ->display( ).
```

Listing 10.36 Coding the SALV with IDA without Selection Criteria

What happens is that if only 20 rows can fit on the screen, then only 20 rows are passed back from the database. When you scroll down, another database trip occurs, and the next 20 rows are sent back. Only those 20 (or however many fit on the screen) rows are ever in memory at any given instant. The database knows about every record that was queried but only sends back to the application server what's required at any given instant. Because you cannot drill down on a row that isn't on the screen, you don't actually need that row to be in memory.

In summary, this dramatically speeds up the time between running the query and the data appearing on the screen, speeds up the response time when paging down, and lowers memory consumption. Moreover, in addition to being faster, the end user will not notice the difference (thus, no extra training costs); everything

looks the same as before, and all the functions at the top of the screen are still available.

`CL_SALV_GUI_TABLE_IDA` can be used with any database, but SAP says it works best with SAP HANA (the class was designed to use code push down where possible, which means that more work is done inside the database). This means you probably won't notice that much performance difference with other databases.

10.6 Summary

This chapter explained how to create a custom API so that a calling program doesn't need to know the gruesome details of what technology is being used to create the report output (SALV in this case, naturally) and, moreover, can switch between technologies easily, in case a new one come along. It also explained a few "impossible" things that you supposedly can't do with SALV (but actually can), as well as a recent SALV improvement that helps in dealing with large internal tables.

If you've worked in IT for any length of time, then you'll know that once a user has a nice report on his SAP screen that he's happy with, the next thing he'll do is to download it to Excel to play with the data. A lot of IT departments have fought against this, but it's rather like King Canute ordering the tide to go back out. Therefore, the next chapter looks at an open-source project to vastly improve SAP and Microsoft Excel integration.

Recommended Reading

- ▶ International Editable SALV Day: <http://scn.sap.com/thread/3567633> (Paul Hardy)
- ▶ *Head First Design Patterns* (Freeman et al., O'Reilly, 2004)
- ▶ *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin, Prentice Hall, 2008)

Q. How do you identify an extroverted spreadsheet user?

A. He looks at your shoes when he talks to you.

—Unknown

11 ABAP2XLSX

Many times, when SAP programmers venture out of their ivory towers, go down to the “coalface,” and meet the people who actually use the software they write (the so-called users), they are amazed to find that a large number of jobs that people perform revolve around the following three tasks:

- ▶ Periodically running some sort of report in SAP and dumping the results into a spreadsheet
- ▶ Transforming that data in weird and wonderful ways—often using macros written by shadow IT Microsoft Excel experts, macros that are so complicated that they put your ABAP programs to shame and that have big colorful buttons that make it obvious what the spreadsheet user has to do
- ▶ Uploading that transformed data back into SAP again

A killer application: Programmers always want to create such a thing, and it's universally acknowledged that Excel is just such a beast. Nonetheless, IT departments have been fighting this tooth and nail for many years; the reasoning goes that ERP systems were invented to get away from all this. My observation is that the decade-long battle to stop people using spreadsheets has been well and truly lost. Given that premise, the only thing you can do is to make it easier for your users to do what they're going to do anyway.

SAP has always been able to talk to Excel, but this chapter covers the open-source ABAP2XLSX framework that dramatically enhances the data exchange between ABAP and Excel. The reason I included ABAP2XLSX in a book about new innovations in ABAP tools is that, in 2012, I gave a presentation to some programmers and IT managers about all the new things available in SAP ECC 6.0: ABAP Unit, shared memory, Web Dynpro, and so on. The only thing that seemed to impress

them was ABAP2XLSX, because there was an obvious business use for it. To reiterate, although a lot of people deny it, in most companies three quarters of the work is done in Excel. I don't think there has been one day in 23 years of working for my company in which I haven't opened a spreadsheet at some point during the day, and I am supposed to be on the side that wants to get rid of spreadsheets. In addition, as far as I can see, ABAP2XLSX is still widely unknown in the ABAP community. I draw this conclusion from the fact that on the SCN almost every single week someone posts a blog about a new way they have discovered of downloading SAP data to Excel. Then I point them to ABAP2XLSX, and they say, "That's interesting; I've never heard of that."

In an effort to make sure I never have to read that response again, this chapter introduces you to ABAP2XLSX. Section 11.1 introduces you to the basics you need to know to work with the technology. The heart of the chapter is Section 11.2, which provides assorted practical examples of the ABAP2XLSX features that programmers would want to take advantage of. In each case, you'll see how ABAP2XLSX lets SAP take the strain for these repetitive tasks, leaving users free to concentrate on value-adding tasks. Finally, Section 11.3 discusses some tips and tricks to help you get the most out of this framework.

A Word on Open-Source Projects

ABAP2XLSX is an open-source community project and as such is written and provided not by SAP but by one original author (in this case, Ivan Femia) and a large number of contributors working on it in their spare time away from their day jobs. Sometimes, such projects are looked down on as second-class citizens, but I can't help but giggle when they are described as "just" community projects. It reminds me of the time one of the IT department members told the CIO's PA that she was "just" a secretary. In retrospect, he wished he hadn't.

I find that the open-source community projects tend to address actual business needs that people have encountered in their day-to-day business work, whereas the "real" innovations invented by SAP tend to be guesses as to what their customers might want. Although usually these projects are on target, SAP often gets it spectacularly wrong.

The perceived downside of open-source ABAP projects is that they are (naturally) not supported by SAP; thus the quality is not guaranteed, and often there are bugs. The upside is that the source code is free, there are no license considerations, and new versions come out regularly (often every day). In addition, if there is a feature missing, then you can—and are in fact encouraged to—add the missing element yourself. The same is true of bugs; if you find one, then you fix it yourself. In both cases, the fix or new feature should be communicated to the open-source project so that the whole ABAP commu-

nity can benefit. That's what open source is all about—not just “take, take, take,” though of course some people do just that.

11.1 The Basics

In this section, you'll take a look at the basics of both Excel and ABAP2XLSX that you need to understand in order to use ABAP2XLSX effectively. Section 11.1.1 explains how special formatting and the like get stored in Excel spreadsheets. (This is probably common knowledge in Microsoft circles, but not so much in the ABAP world.) Section 11.1.2 explains how to download the ABAP2XLSX software and how to provide feedback and any improvements you might want to make to the open-source project. Finally, Section 11.1.3 explains how ABAP2XLSX generates the required data out of SAP using ABAP. The code to call this from your own programs is very simple, but you need to understand the underlying mechanism if you ever want to make any improvements or fix any problems.

11.1.1 How XLSX Files are Stored

The best way to demonstrate the storage of XLSL files is to create a spreadsheet with a particular setting and then save it in such a way that you can see the underlying XML structure. As an example, create a spreadsheet, and have the header rows repeat at the top while printing, as can be seen in Figure 11.1.

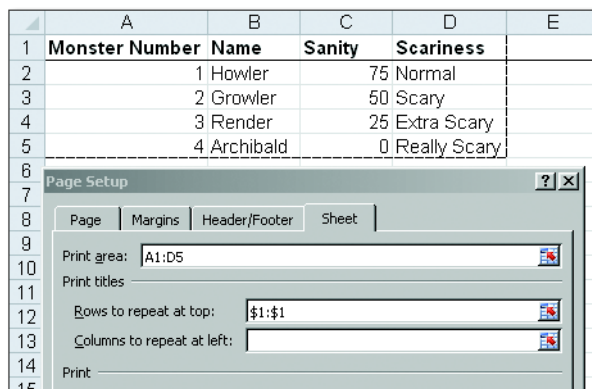


Figure 11.1 Example Spreadsheet with Print Settings

Instead of saving this spreadsheet normally, choose the `SAVE AS` option, and save it as "Monster Spreadsheet.zip"; it's important to have the quotation marks around the name. You can see this in Figure 11.2.

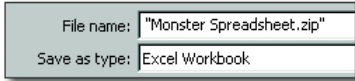


Figure 11.2 Saving a Spreadsheet as a ZIP File

This appears in the Explorer as a normal compressed ZIP file. However, when you double-click it, instead of the contents being a spreadsheet file, you get a tree structure of folders containing XML files (Figure 11.3). Each level in the tree will have one or more XML files and possibly some lower-level folders. The more features the spreadsheet has, the more complicated the tree structure will be.

Name ^	Type
_rels	File folder
docProps	File folder
xl	File folder
[Content_Types]	XML Document

Figure 11.3 File Tree Structure for a Spreadsheet

Now, you have to go on a big game hunt to find out which XML file the feature in which you're interested lives. In this example, you're looking for the details of the print area and which rows repeat at the top, so double-click the `xl` folder, which results in the screen shown in Figure 11.4. Then, open the workbook XML file, which results in the screen shown in Figure 11.5.

Name ^	Type
_rels	File folder
printerSettings	File folder
theme	File folder
worksheets	File folder
sharedStrings	XML Document
styles	XML Document
workbook	XML Document

Figure 11.4 Expanded `xl` Folder Contents

```

<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <workbook xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships" xmlns:
  <fileVersion r:upBuild="9303" lowestEdited="5" lastEdited="5" appName="xl"/>
  <workbookPr defaultThemeVersion="124226"/>
  - <bookViews>
    <workbookView windowHeight="9600" windowWidth="23505" yWindow="45" xWindow="240"/>
  </bookViews>
  - <sheets>
    <sheet r:id="rId1" sheetId="1" name="Sheet1"/>
    <sheet r:id="rId2" sheetId="2" name="Sheet2"/>
    <sheet r:id="rId3" sheetId="3" name="Sheet3"/>
  </sheets>
  - <definedNames>
    <definedName name="_xlnm.Print_Area" localSheetId="0">Sheet1!$A$1:$D$5</definedName>
    <definedName name="_xlnm.Print_Titles" localSheetId="0">Sheet1!$1:$1</definedName>
  </definedNames>
  <calcPr calcId="145621"/>
</workbook>

```

Figure 11.5 Worksheet XML File

As you can see, near the bottom of Figure 11.5 there are two lines that store the print area and repeating rows. The information about which XML file stores which type of information is not a Da Vinci code type of secret; if you search online for something such as “Microsoft Open XML,” you will find more hits than you can shake a stick at. Microsoft has published the structure and definition of each element with the express intention of enabling tools such as ABAP2XLSX to generate spreadsheets and other documents. This is why the format is called Open XML.

11.1.2 Downloading ABAP2XLSX

The home page of ABAP2XLSX is www.abap2xlsx.org, which will redirect you to the GitHub home page (formerly, it used the SAP Community Network Code Exchange, back when I was proud to get on the list of top active members of the project, albeit at the bottom). When you are on the GitHub site, simply click the big blue button to download a ZIP file containing the nugget of all the ABAP2XLSX classes and demonstration programs. You will need to have SAPlink installed to upload this material into your SAP system (this was discussed in Chapter 3 when you read about the open-source mockA unit testing project). You will also need to sign up for a GitHub account (it’s free). Once you have uploaded the nugget file containing the ABAP2XLSX objects and activated them, you’re good to go.

11.1.3 Creating XLSX Files Using ABAP

At a very high level, the process for creating XLSX files using ABAP is incredibly simplistic, as simplistic as Jock McSimplistic the simplistic Scotsman, winner of the all-Scotland "being simplistic" contest. This process is depicted in Figure 11.6.

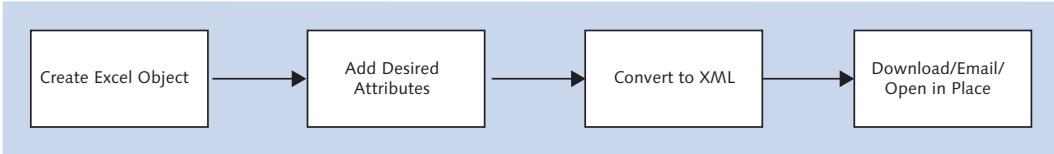


Figure 11.6 High-Level ABAP2XLSX Process Flow

Each of the steps in the process are discussed in more detail next.

Creating the Excel Object

The first step in the process flow is to create an Excel object. (Section 11.2.1 adds another step at the start of the process flow: taking an existing SAP report object and converting it into an Excel object. However, for now you will start by creating the Excel object from scratch.) In OO programming, classes represent real-world objects, and in this case you have a class that is a representation of the real-world object that is an Excel spreadsheet. Because building up an XML structure manually can be quite cumbersome, working with an instance of such a class is far easier.

A spreadsheet consists of one or more worksheets, so after you create your Excel object you have to specify which worksheet you're talking about. Naturally, the default is the first worksheet, so that is the active one. In most spreadsheets, you only have one worksheet. The code for creating the Excel object and navigating to the first worksheet is shown in Listing 11.1.

```

DATA: lo_excel      TYPE REF TO zcl_excel,
      lo_worksheet TYPE REF TO zcl_excel_worksheet.

CREATE OBJECT lo_excel.

lo_worksheet = lo_excel->get_active_worksheet( ).
  
```

Listing 11.1 Creating the Excel Object and Navigating to the First Worksheet

Adding Desired Attributes

Now, you'll add the contents of the spreadsheet (fill up the cells with values), and then add all the fancy elements like you normally do, such as changing column widths and the like. The code for filling in one cell with a value and changing the size of the column and rows is shown in Listing 11.2.

```
lo_worksheet->set_title( 'Example Worksheet' ).
lo_worksheet->set_cell( ip_column = 'A' ip_row = 1
                      ip_value = 'I am a Spreadsheet!' ).

DATA: column_dimension
      TYPE REF TO zcl_excel_worksheet_columndime,
      row_dimension
      TYPE REF TO zcl_excel_worksheet_rowdimensi.

column_dimension = lo_worksheet->get_column_dimension( ip_column = 'A' ).
column_dimension->set_width( ip_width = 19 ).

row_dimension = lo_worksheet->get_row_dimension( ip_row = 1 ).
row_dimension->set_row_height( ip_row_height = 20 ).
```

Listing 11.2 Adding Desired Attributes to a Spreadsheet Object

As you can see, this code pulls out tiny objects from inside the main Excel object and changes their attributes, which should give you a big feeling of *déjà vu*. This is because this is exactly the sort of thing you do when setting up an instance of `CL_SALV_TABLE` for displaying a report on the screen, which was discussed back in Chapter 10. Therefore, there is nothing all that new and radical going on here. You will also notice how good the naming convention is: Everything reads like plain English, something you don't see very often with standard SAP functions, classes, and methods.

Converting to XML

Once you're happy with the contents of the spreadsheet and the way it will be formatted, it's time for the spreadsheet object to leave the cozy world of ABAP and become a real spreadsheet that can be viewed and manipulated using Microsoft Excel. To enable this, you have to turn the Excel object into an XML file. This is done via one simple method call, as shown in Listing 11.3.

```
DATA: ld_xml_file      TYPE xstring,
      lo_excel_writer TYPE REF TO zif_excel_writer.
```

```
CREATE OBJECT lo_excel_writer TYPE zcl_excel_writer_2007.
ld_xml_file = lo_excel_writer->write_file( lo_excel ).
```

Listing 11.3 Converting an Excel Object to XML

The interface `ZIF_EXCEL_WRITER` defines the methods and attributes for converting an Excel object into an XML file. Therefore, if a new version of Excel comes out and it turns out to be radically different, then a new class is created that handles the new version, and the calling program does not need to change at all, except that the `CREATE OBJECT` statement has to refer to `ZCL_EXCEL_WRITER_2057` or whatever the new Excel version will be. As an example, if the spreadsheet is to be macro-enabled, then the object has to be created as `TYPE ZCL_EXCEL_WRITER_XLSM`.

If you drill down into the `WRITE_FILE` method, then you'll see a series of steps that build up the exact folder structure you saw earlier (Figure 11.3) when you looked at your zipped spreadsheet file—that is, first a ZIP object is created, and then the folders and XML files are appended to that ZIP object. The `ZCL_EXCEL_WRITER` class uses standard SAP mechanisms to build up an XML structure, with the result coming out as an `XSTRING` object, which is essentially an enormous amount of information in a really long string.

As mentioned previously, the settings for the rows that repeat the top when printing were stored in the workbook file within the `xl` folder. If you look inside the `WRITE_FILE` method, you'll see a submethod called `CREATE_XL_WORKBOOK`, which does what you would expect—that is, it creates the XML file called “workbook” that lives in the `xl` folder.

Downloading the Spreadsheet to the Frontend

Once you have an XML file containing the spreadsheet information, you can export this file from the ABAP system to a directory on your local machine, where the file will look just like a spreadsheet that has been created manually in Excel and then saved.

The code in Listing 11.4, which shows how to download the spreadsheet, consists entirely of calls to standard SAP methods. First, the XML string is converted into so-called raw data; then, the user is asked where he wants to save the file; finally, the file is downloaded in the normal manner.

```
DATA: lt_rawdata TYPE solix_tab,
      ld_bytecount TYPE i,
      ld_filename TYPE string,
```



```

        ld_path      TYPE string,
        ld_fullpath  TYPE string.

* This is the important bit
lt_rawdata  = cl_bcs_convert=>xstring_to_solix( iv_xstring = ld_xml_file ).
ld_bytecount = strlen( ld_xml_file ).

* From now on this is all bog standard SAP
cl_gui_frontend_services=>file_save_dialog(
    EXPORTING
        window_title      = 'Choose where to save file'
        default_extension  = 'XLSX'
    CHANGING
        filename           = ld_filename " File Name to Save
        path                = ld_path     " Path to File
        fullpath            = ld_fullpath " Path + File Name
    EXCEPTIONS
        cntl_error         = 1
        error_no_gui       = 2
        not_supported_by_gui = 3
        OTHERS              = 4 ).

IF sy-subrc <> 0.
    RETURN.
ENDIF.

cl_gui_frontend_services=>gui_download(
    EXPORTING bin_filesize = ld_bytecount
              filename     = ld_fullpath
              filetype     = 'BIN'
    CHANGING data_tab     = lt_rawdata ).

```

Listing 11.4 Downloading the Spreadsheet

In a similar manner, you can open up the spreadsheet inside SAP or email it out. (Section 11.2.8 goes into detail about email options.) Opening the spreadsheet in place uses standard SAP mechanisms based on standard class `CL_OI_CONTAINER_CONTROL_CREATOR`, which can be seen in the example programs that came with `ABAP2XLSX`.

Examples

When you install `ABAP2XLSX` onto your system, in addition to the base classes you get an executable program called `ZABAP2XLSX_DEMO_SHOW`, which is a collection of about 40 individual demonstration programs, each showing a feature of `ABAP2XLSX` and how to code it. By looking at the code in these sample programs, you can quickly learn how to make cells bold, underline cells, or perform any of the other common Excel tasks.

11.2 Enhancing Custom Reports with ABAP2XLSX

If you were to perform an inventory of what your users were doing with your custom SAP reports, you would possibly find that they were running large reports in batch every night and then calling up the spool request the next day and downloading it to Excel. SAP lets you do that easily, but then you have to remove a few blank columns from the top and the left. The user will then make the report “pretty” by making the headers bold and adding colors (e.g., negative numbers in red) and so forth and then will set up the print formatting so that the report prints in landscape orientation. Quite often, conditional formatting is used to make exceptional values jump out right into the user's face, making use of red traffic lights and the like. Sometimes, several reports are merged into one spreadsheet with multiple worksheets, or a worksheet is added that contains pretty bar and pie charts. A common enhancement is to add macros to spreadsheets to enable all sorts of groovy functionality, like sorting a column when you double-click the header (just like what happens inside SAP ALV reports).

Finally, the report is emailed out to various interested parties. When those interested parties get the spreadsheet and open it up, a user might look at some figures and say: “Oh, that's interesting; that sales order has a red light to indicate a massive negative margin. I wonder what's going on there?” Then, that user goes into SAP, runs a report to try and find that sales order, and then drills into the underlying document to get to the root of the problem.

All in all, this is a whole lot of work—and it doesn't have to be. This section will go over each one of those tasks in turn to see how ABAP2XLSX makes each one better.

11.2.1 Converting an ALV to an Excel Object

It is highly unlikely that you will create a report that only creates a spreadsheet. The normal state of affairs is that you have an already existing custom report that you want to enhance by using ABAP2XLSX to add extra options, such as to download the report to a spreadsheet or email it to one or more people with all of the formatting and features that would normally be manually added.

The normal situation starts when you have an `SALV` object or a `CL_GUI_ALV_GRID` object, and you have finished adding sort criteria and hiding columns and saying what columns should be subtotaled and so forth. You now want to turn this into

an Excel object so that you can make some further changes and then download the spreadsheet or email it out.

The good news—the wonderful news—is that there is a class called `ZCL_EXCEL_CONVERTER` that transforms a report object into an Excel object. Listing 11.5 first creates an SALV object by passing in a table of monster data to the `CL_SALV_TABLE` factory method. Once you have the SALV object, you create an instance of the `CONVERTER` class, pass in the ALV object (and the data table again), and out pops an Excel object.

```
DATA: lo_converter TYPE REF TO zcl_excel_converter,
      lo_excel     TYPE REF TO zcl_excel,
      lo_exception TYPE REF TO zcx_excel,
      lo_salv      TYPE REF TO cl_salv_table.

cl_salv_table=>factory(
  IMPORTING
    r_salv_table = lo_salv
  CHANGING
    t_table      = gt_monsters[] ).

* Add sort criteria and what have you...

CREATE OBJECT lo_converter.

TRY.
  lo_converter->convert(
    EXPORTING
      io_alv          = lo_salv
      it_table        = gt_monsters[]
      i_row_int       = 5
      i_column_int    = 6
      i_table         = abap_true
      i_style_table   = zcl_excel_table=>builtinstyle_medium2
    CHANGING
      co_excel        = lo_excel ).

  CATCH zcx_excel INTO lo_exception.
    "Raise Fatal Exception
ENDTRY.
```

Listing 11.5 Transforming a Report Object into an Excel Object

In Listing 11.5, you can see an SALV object being created and then transformed into an Excel object. Interestingly, you could have created and prepared a `CL_GUI_ALV_GRID` object instead and passed that in; the `IO_ALV` parameter is typed as `TYPE REF TO OBJECT`, so you can pass any object in, and within the method the dynamic

runtime type identification works out what type of object is being passed in and creates an appropriate `CONVERTER` object (if one exists for that class). This means that if a new UI technology comes along you only have to create a new subclass for the purposes of conversion to Excel.

In your custom report program, the four steps to spreadsheet heaven are as follows:

1. Create your report object (`SALV/CL_GUI_ALV_GRID`), and choose your formatting options for displaying this within SAP.
2. Convert the report object into an Excel object by using the built-in `CONVERTER` class.
3. Use the ABAP2XLSX methods to add further spreadsheet-specific formatting options, such as print settings, conditional formatting, and the like.
4. Download the spreadsheet, email it out, or open the spreadsheet in place within the SAP GUI.

11.2.2 Changing Number and Text Formats

I have a horrible confession to make. For the first seven years of my career, I had nothing to do with IT; I was—dramatic pause—an accountant. I did bank reconciliations, looked after fixed assets, and administrated the payroll for truck drivers, and my working life revolved around month end.

Although I much prefer programming, my experience with accounting enables me to empathize with business users who are themselves accountants. The first thing an accountant will usually do with a spreadsheet he has generated from out of SAP will be to change the formatting so that negative numbers are (a) red and (b) surrounded by brackets. Zero values have to be replaced by dashes. The currency sign needs to be added for values relating to money. Only then is the report fit to be looked upon by human eyes.

In Section 11.2.1, you converted your SALV report into an Excel object. Now, you'll start making application-specific changes—in this case, changing all the currency amounts into the format that accountants like. First, you need a bunch of data declarations; these aren't very exciting, but they are presented in Listing 11.6 for completeness.

```

DATA: lo_worksheet      TYPE REF TO zcl_excel_worksheet,
      ld_no_of_columns  TYPE zexcel_cell_column,
      ld_no_of_rows     TYPE zexcel_cell_row,
      ld_column_alpha   TYPE zexcel_cell_column_alpha,
      ld_column_integer TYPE zexcel_cell_column,
      ld_row_integer    TYPE zexcel_cell_row,
      ls_stylemapping   TYPE zexcel_s_stylemapping,
      ld_cell_style     TYPE zexcel_cell_style.

```

Listing 11.6 Data Declarations

Now, to get on with the actual work. You'll have to use a sledgehammer approach here, checking every cell in the spreadsheet to see if it's a numeric value with two decimals (quantities have three decimals) and then applying the accountant-specific rules to that cell. This will demonstrate quite a few ABAP2XLSX functions along the way.

Listing 11.7 first determines the occupied area of the spreadsheet to figure out what cells need to be checked. The code also contains a nested loop that moves from column to column, checking each cell in each column. The code accesses the cells in the way in which they appear to someone looking at the spreadsheet—that is, (A,1) instead of (1,1).

```

lo_worksheet = lo_excel->get_active_worksheet( ).

ld_no_of_rows    = lo_worksheet->get_highest_row( ).
ld_no_of_columns = lo_worksheet->get_highest_column( ).

DO ld_no_of_columns TIMES.
  ADD 1 TO ld_column_integer.
  ld_column_alpha =
    zcl_excel_common=>convert_column2alpha( ld_column_integer ).
  ld_row_integer = 0.
  DO ld_no_of_rows TIMES.
    ADD 1 TO ld_row_integer.

```

Listing 11.7 Looping Through All the Cells in a Spreadsheet

You're interested not in the cell's contents but rather in its *style*, which is the way it is currently formatted (e.g., three decimal places or negative numbers show as red). This format is stored as a string that exactly matches what you would type in if you were creating a custom format in Excel. The code for determining a cell's style is shown in Listing 11.8.

```

lo_worksheet->get_cell(
EXPORTING ip_column = ld_column_alpha
          ip_row      = ld_row_integer
IMPORTING ep_guid    = ld_cell_style ).

TRY.
ls_stylemapping = lo_worksheet->excel->get_style_to_guid( ld_cell_style ).
CATCH zcx_excel.
  CLEAR ls_stylemapping.
ENDTRY.

```

Listing 11.8 Finding the Style of a Spreadsheet Cell

Once you know the style, you can see the current number of decimal places. If it's two, then you're dealing with a currency number; if there are three decimal places, then you're dealing with a quantity.

Listing 11.9 makes changes based on the current style (determined in Listing 11.8). If there are two decimal places (i.e., a currency), then the code sets the cell to turn red when negative, to not have a minus sign, and to display a dash when the value is zero (quantities are unchanged so the end user can tell that numbers with three decimal places are quantities). Changing the number of decimal places and so forth is done by setting the `format_code` variable to the same value as the string you would manually type into Excel to achieve the same goal and then passing this variable into a method called `CHANGE_CELL_STYLE`. Once the code has made the change, it moves on to the next cell.

```

IF ls_stylemapping-complete_style-number_format-format_code = '#,##0.00'.
  " This is a currency amount, use the accounting conventions
  " which are to have negative numbers as red in brackets, and show zero values
  " as dashes, so as to focus the eye on the real numbers
ls_stylemapping-complete_style-number_format-format_code = '$#,##0.00;
[Red]($#,##0.00);-'.
lo_worksheet->change_cell_style(
ip_column      = ld_column_alpha
ip_row         = ld_row_integer
ip_number_format_format_code = ls_stylemapping-complete_style-number_
format-format_code ).
ENDIF."It's a number with two decimals
ENDDO."Rows
ENDDO."Columns

```

Listing 11.9 Setting a Cell to be Formatted Appropriately

The class `ZCL_EXCEL_STYLE_NUMBER_FORMAT` has a whole raft of constants for the most common formats for dates and percentages and the like, but as you can see you can mix and match, and within SAP you are able to do anything that's possible with the Excel formatting options.

\$295,488	\$56,926	\$195,871	\$59,411	\$123,959	\$8,879	-	-	\$740,535
\$75,280	\$9,467	(\$2,149)	-	-	-	-	-	\$86,896
\$4,984	-	\$81,389	-	-	\$949	-	-	\$87,321
-	(\$60,983)	-	-	-	-	-	-	\$60,983
\$3,154	\$8,531	\$5,207	-	\$4,090	-	-	-	\$20,982
-	-	-	-	\$35,304	-	-	-	\$35,304
\$922	-	-	-	-	-	-	-	\$922
\$1,818	-	-	-	-	-	-	-	\$1,818

Figure 11.7 Cells Formatted Using Accounting Conventions

As can be seen in Figure 11.7, by the time the accountant has downloaded the spreadsheet from SAP, none of the usual work to reformat the cells in the way that he likes is needed.

11.2.3 Establishing Printer Settings

Even while I was still at school, I heard talk about the dawn of the “paperless office,” in which nothing would ever be printed, because everything could be done on a computer. Technology has moved on in leaps and bounds since then, but even now I don't think that the ink companies need to be too worried. Realistically, the first thing people do when they have downloaded a report from SAP onto a spreadsheet is to print it out so they can scribble all over it.

Given that premise, the first problem your user is going to have is that in 99% of reports there are more columns than there are rows, and because the default printing mode for a spreadsheet is `PORTRAIT`, there will be more columns than can fit on the page. If the user tries to print it out without changing the orientation, then the result will look horrible, with the information from one row being on two different pieces of paper. Therefore, the first thing someone would do before printing is to change the orientation to `LANDSCAPE`, then usually change the setting so that all the columns fit on one page, and finally add some headers and footers and set some rows as header rows so that each printed page has header information above each column saying what the column represents.

Why not do all of that in SAP before downloading the spreadsheet so that your users don't have to? First, change the orientation and make a few other general

settings. There is an attribute of the worksheet object that is a structure called `SHEET_SETUP` that's filled with fields in which you can change assorted print settings. Listing 11.10 first makes the page margins narrow (so that more can fit on the printed page) by passing in various values to a method called `SET_PAGE_MARGINS`. Then it changes some more fields in the `SHEET_SETUP` structure to make the spreadsheet columns fit all on one page and to change the orientation to landscape (which is the most important thing accomplished in this code).

```

" Page printing settings
" Margins are to be set to the narrow values. Just copy
" the values in the narrow option on the print preview.
lo_worksheet->sheet_setup->set_page_margins(
ip_top    = '1.91'
ip_bottom = '1.91'
ip_left   = '0.64'
ip_right  = '0.64'
ip_header = '0.76'
ip_footer = '0.76'
ip_unit   = 'cm' ).
lo_worksheet->sheet_setup->black_and_white = abap_true.

"Requirement is to fit all columns on one sheet
lo_worksheet->sheet_setup->fit_to_page = abap_true. " you should
            * turn this on to activate fit_to_height and fit_to_width
lo_worksheet->sheet_setup->fit_to_width = 1.      " used only if ip_fit_
            " to_page = 'X'

lo_worksheet->sheet_setup->orientation
            = zcl_excel_sheet_setup=>c_orientation_landscape.
lo_worksheet->sheet_setup->page_order
            = zcl_excel_sheet_setup=>c_ord_downthenover.
lo_worksheet->sheet_setup->paper_size
            = zcl_excel_sheet_setup=>c_papersize_a4.
lo_worksheet->sheet_setup->scale
            = 80. " used only if ip_fit_to_page = SPACE
lo_worksheet->sheet_setup->horizontal_centered = abap_true.

```

Listing 11.10 Programatically Changing the Print Orientation to Landscape

As mentioned earlier, the method names and attributes are all in plain English, so it's easy to tell what is going on.

Next, set up some headers and footers to be printed at the top and bottom of every page; naturally, you can put anything you want in the headers and footers. In this example, put the spreadsheet name in the center of the header, the date the spreadsheet was generated on in the left of the footer, the spreadsheet file

path in the middle of the footer, and the current page number (as in, page X of Y) on the right in the footer.

Listing 11.11 fills up two structures: LS_HEADER and LS_FOOTER. These structures have various fields for the left, center, and right of the header and footer. Change the structures such that the worksheet name is in the center of the header, and then specify that the date goes on the left of the footer, the spreadsheet path goes in the middle of the footer (there is a secret code used here called '&Z&F', which stands for path/filename), and that the current page number should go on the right of the footer (again using a secret code, 'page &P of &N'). Once the two structures are full, pass them into the SET_HEADER_FOOTER method to update the worksheet object.

```
DATA: ls_header TYPE zexcel_s_worksheet_head_foot,
      ls_footer TYPE zexcel_s_worksheet_head_foot,
      ld_string TYPE string.

"Put Tab Name in Header Centre
ls_header-center_value   = lo_worksheet->get_title( ).
ls_header-center_font    = ls_header-right_font.
ls_header-center_font-size = 8.
ls_header-center_font-name = zcl_excel_style_font=>c_name_arial.

"Put generated date on footer left
CONCATENATE sy-datum+6(2) sy-datum+4(2) sy-datum(4) INTO ld_
string SEPARATED BY '/'.
ls_footer-left_value = ld_string.
ls_footer-left_font  = ls_header-center_font.

"Put Spreadsheet path and name in Footer Centre
ls_footer-center_value = '&Z&F'.           "Path / Filename
ls_footer-center_font  = ls_header-center_font.

"Put page X of Y on Footer Right
ls_footer-right_value = 'page &P of &N' ##no_text. "page x of y
ls_footer-right_font  = ls_header-center_font.

lo_worksheet->sheet_setup->set_header_footer(
ip_odd_header = ls_header
ip_odd_footer = ls_footer ).
```

Listing 11.11 Coding Header and Footer Print Settings

In this example, the first row is the header row; say you want this to repeat at the top of every sheet that's printed. The code for this is shown in Listing 11.12.

```

to_worksheet->zif_excel_sheet_printsettings~set_print_repeat_rows(
    iv_rows_from = 1
    iv_rows_to   = 1 ).

```

Listing 11.12 Making the Header Row Repeat on Every Printed Sheet

As can be seen in Figure 11.8, setting up the print settings has been automated, saving users a few seconds, because now they don't have to make the settings manually. Each time saving on its own may not seem like much, but if you multiply this by the number of users and by the number of reports printed, then the savings quickly add up.

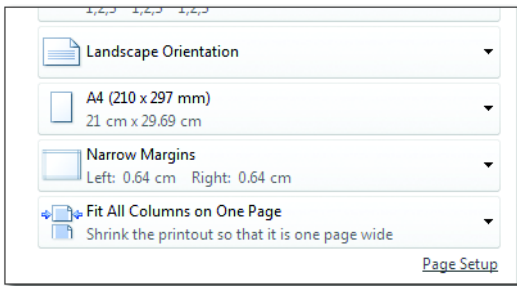


Figure 11.8 Automated Print Settings

11.2.4 Using Conditional Formatting

There are a lot of ways you can use conditional formatting; a basic example and a more advanced example are discussed next.

Basic

Sometimes, when you look at a spreadsheet, you are looking only for rows that are unusual in some way (i.e., managing by exception). A way to draw people's attention to the guilty rows on a spreadsheet is to use conditional formatting to color in cells with unusual values.

In this example, you will color in the MONSTER STRENGTH column so that if the monster is strong, then the cell is green; if it's weak, then the cell is red; otherwise, the cell is yellow. First, determine which column is the strength column in your spreadsheet (column E in this example). Set this as a constant to avoid hard-coding anything; see Listing 11.13.

```
CONSTANTS: lc_strength_column
TYPE zexcel_cell_column_alpha VALUE 'E'.
```

Listing 11.13 Strength Column

Next, set up the three colors needed in this example; as you will see in Listing 11.14, this is quite complicated, which is often the case when you have a large amount of flexibility. The flexibility here is that although there are constants for common colors such as green via the eight-digit hexadecimal codes that Excel uses for colors, you can have any color under the sun. The ZDEMO_EXCEL21 program, which comes with ABAP2XLSX, demonstrates all the colors available to you. Each color will be represented by a GUID, which, as you've seen throughout this book, is a very long string that is only understandable by a computer.

For each color, set up a `style` object, indicate that there will be a solid background, and specify what color that background will be. Then, call the `GET_GUID` method of the `style` object to turn the human-friendly instructions into something a computer can work with. You will note that the colors are set up to be used based on the Excel object as opposed to the worksheet object, as they can be used in any worksheet.

```
DATA:
lo_style_conditional TYPE REF TO zcl_excel_style_conditional,
lo_style_1           TYPE REF TO zcl_excel_style,
lo_style_2           TYPE REF TO zcl_excel_style,
lo_style_3           TYPE REF TO zcl_excel_style,
ld_green_guid        TYPE zexcel_cell_style,
ld_red_guid          TYPE zexcel_cell_style,
ld_yellow_guid       TYPE zexcel_cell_style,
ls_cellis            TYPE zexcel_conditional_cellis.

lo_style_1
lo_style_1->fill->filltype      = lo_excel->add_new_style( ).
lo_style_1->fill->filltype      = zcl_excel_style_fill=>c_fill_solid.
lo_style_1->fill->bgcolor-rgb   = zcl_excel_style_color=>c_green.
ld_green_guid                  = lo_style_1->get_guid( ).

lo_style_2
lo_style_2->fill->filltype      = lo_excel->add_new_style( ).
lo_style_2->fill->filltype      = zcl_excel_style_fill=>c_fill_solid.
lo_style_2->fill->bgcolor-rgb   = 'FFFF99FF'. "Soft Red Pink Really
ld_red_guid                     = lo_style_2->get_guid( ).

lo_style_3
lo_style_3->fill->filltype      = lo_excel->add_new_style( ).
lo_style_3->fill->filltype      = zcl_excel_style_fill=>c_fill_solid.
lo_style_3->fill->bgcolor-rgb   = 'FFFF9900'. "Orange
ld_yellow_guid                  = lo_style_3->get_guid( ).
```

Listing 11.14 Creating Colors to Use in a Spreadsheet

The next step is to consider the most basic type of conditional formatting, in which you look for specific values and set the color accordingly. In Listing 11.15, you'll see the code to set up conditional formatting. Three times, the code sets up a rule—in this case, an “is equal to” formula—and applies it to the same range (i.e., the cells in the MONSTER STRENGTH column). If more than one of the formulas turns out to be true (impossible in this example, but you could have one rule that is true if the value is greater than 5 and another rule that is true if the value is greater than 10), then the rule with the highest priority (1 being higher than 3 in this case) will win.

```
DATA: ld_first_data_row TYPE sy-tabix,
      ld_last_data_row  TYPE sy-tabix.

ld_first_data_row = 2. "i.e. first row after header
ld_last_data_row  = lines( gt_monsters ) + 1.

"High Strength Monster - Green for 'Good'
lo_style_conditional = lo_worksheet->add_new_conditional_style( ).
lo_style_conditional->rule = zcl_excel_style_conditional=>c_rule_cellis.
ls_cellis-formula          = "HIGH".
ls_cellis-operator        = zcl_excel_style_conditional=>c_operator_equal.
ls_cellis-cell_style      = ld_green_guid.
lo_style_conditional->mode_cellis = ls_cellis.
lo_style_conditional->priority    = 1.
lo_style_conditional->set_range( ip_start_column = lc_strength_column
                                ip_start_row    = ld_first_data_row
                                ip_stop_column  = lc_strength_column
                                ip_stop_row     = ld_last_data_row ).

"Low Strength Monster - Red for 'Bad'
lo_style_conditional = lo_worksheet->add_new_conditional_style( ).
lo_style_conditional->rule = zcl_excel_style_conditional=>c_rule_cellis.
ls_cellis-formula          = "LOW".
ls_cellis-operator        = zcl_excel_style_conditional=>c_operator_equal.
ls_cellis-cell_style      = ld_red_guid.
lo_style_conditional->mode_cellis = ls_cellis.
lo_style_conditional->priority    = 2.
lo_style_conditional->set_range( ip_start_column = lc_strength_column
                                ip_start_row    = ld_first_data_row
                                ip_stop_column  = lc_strength_column
                                ip_stop_row     = ld_last_data_row ).
```

```

"Medium Strength Monster - Yellow for 'nothing special'
lo_style_conditional = lo_worksheet->add_new_conditional_style( ).
lo_style_conditional->rule = zcl_excel_style_conditional=>c_rule_cellis.
ls_cellis-formula      = '"MEDIUM"'.
ls_cellis-operator    = zcl_excel_style_conditional=>c_operator_equal.
ls_cellis-cell_style  = ld_yellow_guid.
lo_style_conditional->mode_cellis = ls_cellis.
lo_style_conditional->priority    = 3.
lo_style_conditional->set_range( ip_start_column = lc_strength_column
                                ip_start_row    = ld_first_data_row
                                ip_stop_column  = lc_strength_column
                                ip_stop_row     = ld_last_data_row ).

```

Listing 11.15 Creating Conditional Formatting

One point to note is that the rule that you set—which is a constant from `ZCL_EXCEL_STYLE_CONDITIONAL`, with values like `CELLIS`, `ICONSET`, and so on—must match the structure you then fill up (e.g., `MODE_CELLIS`, `MODE_ICONSET`, etc.). If these two do not match up, then the resulting spreadsheet will not open up properly.

Monster Numb	Monster Name	Monster Sanity %a	Monster Color	Monster Strength	Monster Age in Days	Monster Count
1	SNARLER	3	GREEN	LOW	2	0
2	GROWLER	1	PURPLE	MEDIUM	6	0
3	RENDER	2	ORANGE	MEDIUM	8	0
4	ARCHIBALD	25	YELLOW	HIGH	20	0
						0

Figure 11.9 Conditional Formatting: Basic Example

In Figure 11.9, you can see the result. The important point to note is that you haven't only set a color; you've set a formula. If the value in the spreadsheet changes, then the color of the cell will respond accordingly.

You may wonder where the blue background and the autofilters in the header line came from. You haven't actually asked for them in the code, have you? In fact, you did implicitly; when you called the `CONVERTER`, you set the importing parameter `I_TABLE` to be `ABAP_TRUE`, and this defaults the range the data appears in within the spreadsheet to be called `Table1` and to have the autofilters. Furthermore, at the bottom of each column you can open a dropdown to change the totaling option from `NONE` to `AVERAGE`, `MINIMUM`, or a wide variety of other choices.

Advanced

A more advanced example of conditional formatting is to take advantage of the full range of formulas available in Excel to test if a value is bigger or smaller than a certain value. For instance, Listing 11.16 causes a cell to become red if it starts with a 1.

```
lo_style_conditional = lo_worksheet->add_new_conditional_style( ).
lo_style_conditional->rule = zcl_excel_style_conditional=>c_operator_
beginswith.
ls_cellis-formula      = '1'.
ls_cellis-operator    = zcl_excel_style_conditional=>c_operator_beginswith.
ls_cellis-cell_style  = ld_red_guid.
```

Listing 11.16 Conditional Formatting: Testing the Start of a String

Testing for cells that begin with something is not standard in ABAP2XLSX, but it can be done. First, manually create a spreadsheet and set up the conditional formatting the way you want it—in this example, with colors changing based on a cell starting with a 1. Then save the spreadsheet in the way described in Section 11.1.1 so that you can see the structure of the resulting XML file. You'll notice that in the `sheet1` section of the XML file, the XML code shown in Listing 11.17 had been generated.

```
-<conditionalFormatting sqref="G6:G12">-
  <cfRule operator="beginsWith" priority="4" dxfid="4"
    type="beginsWith" text="1">
    <formula>LEFT(G6,LEN("1"))="1"</formula>
  </cfRule>
</conditionalFormatting>
```

Listing 11.17 Generated XML Code

If you want this same functionality in ABAP2XLSX, you have to find a way to manually generate this XML code based on the attributes of the worksheet object. It turns out the XML for conditional formatting is generated inside the `ZCL_EXCEL_WRITER_2007` class in the `CREATE_XL_SHEET` method. Inside that method, there is a `CASE` statement on `LO_STYLE_CONDITION->RULE`, which is a value that was set in Listing 11.16. Listing 11.18 adds a new branch to the case statement and calls various methods to build up the XML structure.

The top node is set outside of the `CASE` statement by the standard ABAP2XLSX class; this is represented by an object called `LO_ELEMENT`, which contains the node that starts and ends with `<conditional formatting>`.

Another object, LO_ELEMENT_2, is going to be a child node of the <conditional_formatting> node; LO_ELEMENT_2 represents the node that starts and ends with <cfRule>. This in turn has another child node, the node that starts and ends with <formula>, and in the code this is represented by LO_ELEMENT_3.

Inside each element, add the values to match the example generated code. For example, inside the formula node, dynamically add the string that looks like LEFT(G6LEN("1"))-"1". Likewise, inside the rule node, add the type="begins with" and text="1" strings by calling the SET_ATTRIBUTE_NS method.

Finally, at the end of Listing 11.18, the child node for the formula is linked to its parent by calling the APPEND_CHILD method.

```
*-----*
* Begin of Insertion PDH
*-----*
* <conditionalFormatting sqref="G6:G12">-
*   <cfRule operator="beginsWith" priority="4" dxfid="4"
*           type="beginsWith" text="1">
*     <formula>LEFT(G6,LEN("1"))="1"</formula>
*   </cfRule>
* </conditionalFormatting>
*-----*
  WHEN zcl_excel_style_conditional=>c_operator_beginswith.
    ls_cellis = lo_style_conditional->mode_cellis.
    READ TABLE me->styles_cond_mapping INTO ls_style_cond_mapping
  WITH KEY guid = ls_cellis-cell_style.
    lv_value = ls_style_cond_mapping-dxf.
    CONDENSE lv_value.
    "Element 2 = <cfRule> node
    lo_element_2->set_attribute_ns( name = lc_xml_attr_dxfid
                                  value = lv_value ).
    lv_value = ls_cellis-operator."Begins With
    lo_element_2->set_attribute_ns( name = lc_xml_attr_operator
                                  value = lv_value ).

    lv_value = ls_cellis-formula."The number 1
    lo_element_2->set_attribute_ns( name = 'text'
                                  value = lv_value ).

    "Element 3 = <Formula> Node
    lo_element_3 = lo_document->create_simple_element(
                                  name = lc_xml_node_formula
                                  parent = lo_document ).
    lv_value = lo_style_conditional->get_dimension_range( ).
    SPLIT lv_value AT ':' INTO ld_first_half ld_second_half.
    lv_value = 'LEFT(' && ld_first_half && ',LEN(''
    && ls_cellis-formula && ''))="'
```

```

    && ls_cellis-formula && ''.
    lo_element_3->set_value( value = lv_value ).
    lo_element_2->append_child( new_child = lo_element_3 ).
    " formula node
*-----*
* End of insertion PDH
*-----*

```

Listing 11.18 Building Up XML for New Type of Conditional Formatting

At the end of the `CASE` statement, the standard ABAP2XLSX code (Listing 11.19) links the new nodes to higher-level nodes by repeated calls to the `APPEND_CHILD` method.

```

ENDCASE.

    lo_element->append_child(
        new_child = lo_element_2 ). " cfRule node

    lo_element_root->append_child(
        new_child = lo_element ). " Conditional formatting node
ENDWHILE.

```

Listing 11.19 Standard ABAP2XLSX Code Linking Nodes Together

Now, use a different sort of conditional formatting and have different colored icons appear in your `MONSTER AGE` column based upon the age of your monster in days. This is not going to be as difficult as you might think, but you do have to do something rather strange, as you can see at the start of Listing 11.20, where you set the value for days to `-9999`.

In Listing 11.20, you're filling up a structure of type `ZEXECECEL_CONDITIONAL_ICONSET`. Set up three structures each containing a pairs of values: one (`CFV01_VALUE`) saying you're about to declare a value that is a number and another (`CFV02_VALUE`) saying what the number is. These numbers will control the relationship between the age of the monster in number of days and the color of the icon that appears.

Once the structure is full, create a new condition formatting object attached to the worksheet, tell this object that the icons will look like traffic lights, and then pass in the completed structure so that the formatting object knows what values trigger what icons.

Finally, specify what range of cells will be affected by the conditional formatting.


```

DATA: ls_iconset3 TYPE zexcel_conditional_iconset,
      ld_number_as_string TYPE string.

"Green if below 7 days
ls_iconset3-cfvo1_type = zcl_excel_style_conditional=>c_cfvo_type_
number.
ls_iconset3-cfvo1_value = '-9999'."What on Earth???"
"Red if above 14 days
ls_iconset3-cfvo2_type = zcl_excel_style_conditional=>c_cfvo_type_
number.
ld_number_as_string      = 14.
ld_number_as_string     = '-' && ld_number_as_string.
CONDENSE ld_number_as_string.
ls_iconset3-cfvo2_value = ld_number_as_string.
"Yellow otherwise
ls_iconset3-cfvo3_type = zcl_excel_style_conditional=>c_cfvo_type_
number.
ld_number_as_string     = 7.
ld_number_as_string     = '-' && ld_number_as_string.
CONDENSE ld_number_as_string.
ls_iconset3-cfvo3_value = ld_number_as_string.
"Show the value as well as the ICON
ls_iconset3-showvalue   = zcl_excel_style_conditional=>c_showvalue_
true.
lo_style_conditional = lo_worksheet->add_new_conditional_style( ).
"We are going to show ICONS
lo_style_conditional->rule = zcl_excel_style_conditional=>c_rule_
iconset.
lo_style_conditional->priority      = 1.
"The ICONS are going to look like Traffic Lights
ls_iconset3-iconset = zcl_excel_style_conditional=>c_iconset_
3trafficlights.
lo_style_conditional->mode_iconset = ls_iconset3.
lo_style_conditional->set_range(
ip_start_column = lc_age_column
ip_start_row    = ld_first_data_row
ip_stop_column  = lc_age_column
ip_stop_row     = ld_last_data_row ).

```

Listing 11.20 Conditional Formatting with Traffic Lights

When looking at Listing 11.20, which covers the conditional formatting for the traffic lights, you may well be puzzled as to what the -9999 value is doing. The problem is that in standard Excel the green traffic light always relates to the highest value. If a high value is bad, then you have to multiply all your values by minus one. However, you do not want the value to appear as negative on the spreadsheet, so in your internal table you can multiply the value by minus one just before converting it to an Excel object and then change the formatting (Listing 11.21) so that the negative figure looks positive.

```

ls_stylemapping-complete_style-number_format-format_code = '#,##0;#,##
0'."i.e. no minus sign when negative
lo_worksheet->change_cell_style(
ip_column          = lc_age_column
ip_row             = ld_sheet_row "this would be a loop
ip_number_format_code = ls_stylemapping-complete_style-number_
format-format_code ).
    
```

Listing 11.21 Changing the Formatting to Make Negatives Look Positive

The result can be seen in Figure 11.10. What's more, you can set the value thresholds (i.e., when a green traffic light turns into a yellow or red traffic light) by having them on the selection screen of the ABAP report with default values that the user can then change. Certain values might be more stringent at year end, for example.

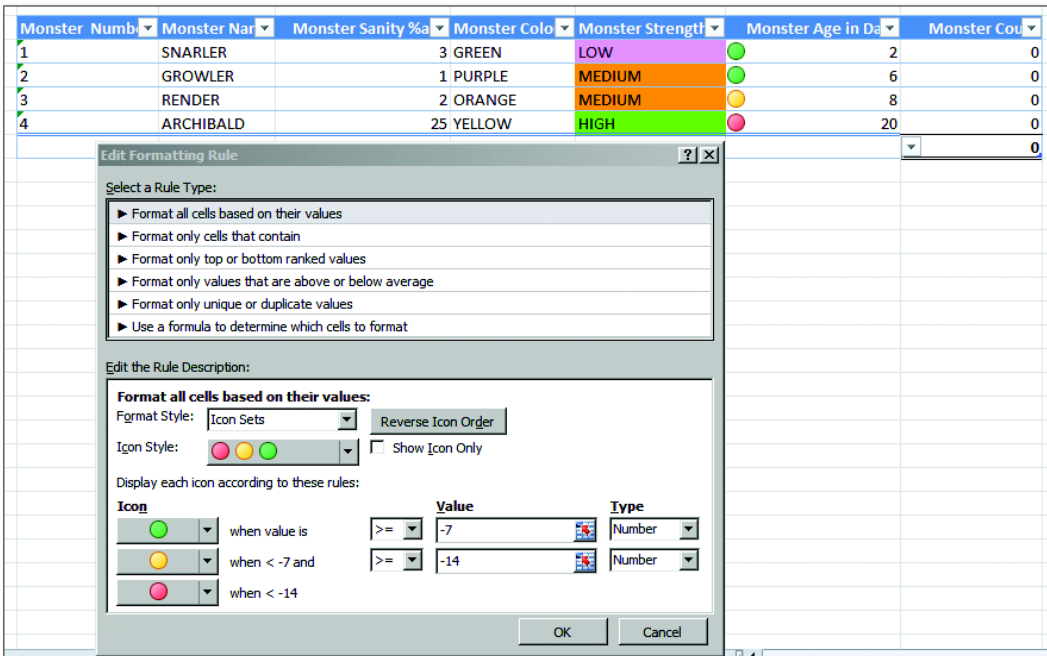


Figure 11.10 Conditional Formatting with Icons

11.2.5 Creating Spreadsheets with Multiple Worksheets

Many times, preparing a spreadsheet involves downloading several reports and then merging them so that each report lives on a separate workbook in the

spreadsheet. Perhaps you want the same data to be displayed in different ways on different sheets: a detail view, a summary view, and a list of records requiring attention, for example. This requires a lot of manual work, usually the exact same work each month or week or even day. Often, people get friendly Excel experts to write macros to help them do this. Next, you will get to go one stage further and have ABAP2XLSX do this work inside of SAP.

In this example, you will download a spreadsheet with two worksheets, both containing the same report. (In real life, of course, you could have as many different reports as you wanted.) Listing 11.22 shows the code to create a second worksheet and fill it with data from a report. In Listing 11.22, you start from the position of already having one worksheet that is full of data. You then give that original worksheet a name so that you can tell it apart from any others you create. Then, create a second worksheet, and call the `CONVERTER` again to fill the new worksheet with an SAP report, this time passing in the exact worksheet you want to populate. (You didn't need to perform that last step of passing in the worksheet object back when there was only one worksheet.)

```
lo_worksheet->set_title( 'First Worksheet' ).

lo_worksheet = lo_excel->add_new_worksheet( ).

lo_worksheet->set_title( 'Second Worksheet' ).

lo_converter->convert(
    EXPORTING
        io_alv      = go_view->mo_alv_grid
        it_table    = gt_monsters[]
        i_table     = abap_true
        i_style_table = zcl_excel_table=>builtinstyle_medium2
        "Specify newly created worksheet
        io_worksheet = lo_worksheet
    CHANGING
        co_excel    = lo_excel ).
```

Listing 11.22 Creating a Second Worksheet

Note

In fact, you do not need a report object at all; you can pass an internal table in via the `BIND_TABLE` method of the worksheet object, and then you do not need a `CONVERTER` at all. The worksheet object also has a `BIND_ALV` method that wraps the `CONVERTER`. However, this chapter explains the long way of doing things so that you can see what's really happening.

The result is shown in Figure 11.11; this facility can save somebody a lot of manual messing about, such as downloading several reports and merging them together. You can even hide some of the added worksheets; a common use of this is a worksheet that is only of interest to a small subset of people to whom this spreadsheet will be mailed out, which is equivalent to having hidden columns in an ALV report.



Figure 11.11 Creating Multiple Worksheets

11.2.6 Using Graphs and Pie Charts

Sometimes, people spend time each month downloading reports from SAP into Excel and then converting the data into pretty graphs, which are then presented to senior management. Indeed, one area in which Excel trumps the SAP GUI is the realm of bar charts, pie charts, and the like. (The equivalent SAP Business Graphics facility within standard reports is unknown to most people, and when they accidentally click an icon to call these “graphics” up, they look at the resulting screen in puzzlement and swear a vow never to press that button again.)

In Section 11.2.5, you worked with multiple worksheets. Now, you’ll amend the example program to have three worksheets: the normal report, values for a pie chart, and the pie chart itself. The worksheet with the values will be hidden, because the chart itself is all you usually need to see. In Listing 11.23, you create a worksheet and manually fill it with the data you need to create a pie chart, which is a column with labels describing the data and a column with the actual values. In this example, you manually fill each worksheet cell; in a real program, you would create an internal table with this data and loop over it, populating each row of the worksheet.

```
lo_worksheet = lo_excel->add_new_worksheet( ).
lo_worksheet->set_title( 'Pie Chart Values' ).

"In real life you would loop over an internal table to
"populate the data values
"Pie Chart - Monster Types
lo_worksheet->set_cell( ip_column = 'A' ip_row = 1 ip_value =
  'Blue Monsters' ).
```

```

lo_worksheet->set_cell( ip_column = 'A' ip_row = 2 ip_value =
'Red Monsters' ).
lo_worksheet->set_cell( ip_column = 'A' ip_row = 3 ip_value =
'Green Monsters' ).
lo_worksheet->set_cell( ip_column = 'A' ip_row = 4 ip_value =
'Sky Blue Pink Monsters' ).

"Pie Chart - Number of each Monster Type
lo_worksheet->set_cell(
ip_column = 'B' ip_row = 1 ip_value = 5 ).
lo_worksheet->set_cell( ip_column = 'B' ip_row = 2 ip_value = 10 ).
lo_worksheet->set_cell( ip_column = 'B' ip_row = 3 ip_value = 15 ).
lo_worksheet->set_cell( ip_column = 'B' ip_row = 4 ip_value = 20 ).

```

Listing 11.23 Setting Up the Pie Chart Data Worksheet

Next, create the worksheet in which the pie chart itself will live. In Listing 11.24, first you create a new worksheet, and then you create a special pie object, `LO_PIE`, which encapsulates pie chart-specific attributes. Tell the pie chart where its data is to come from, and make pie chart-specific settings, such as showing the category names.

Then, create a more generic drawing object, `LO_DRAWING`, which can contain various sorts of charts. Tell this drawing object that it is dealing with a pie chart, and pass in the pie object. Indicate where the top-left-hand cell and bottom-right-hand cell of the drawing object are going to be (in fact, the top left defaults to A1), and add this drawing to the worksheet.

Finally, do a bit of housekeeping to hide the workbook with the pie chart data and to make sure that when the user opens the spreadsheet he is on the first worksheet. Figure 11.12 shows the result.

```

"Add the worksheet with the actual pie chart on it
lo_worksheet = lo_excel->add_new_worksheet( ).

lo_worksheet->set_title( 'Monster Pie Chart' ).

DATA: lo_drawing TYPE REF TO zcl_excel_drawing,
      lo_pie      TYPE REF TO zcl_excel_graph_pie.

CREATE OBJECT lo_pie.

"Tell the Pie Chart where it gets its data from
CALL METHOD lo_pie->create_serie
  EXPORTING
    ip_order      = 0

```

```

"The sheet the data comes from
  ip_sheet =
  'Pie Chart Values' "Range where the labels live
  ip_lbl_from_col = 'A'
  ip_lbl_from_row = '1'
  ip_lbl_to_col = 'A'
  ip_lbl_to_row =
  '4' "Range where the data values live
  ip_ref_from_col = 'B'
  ip_ref_from_row = '1'
  ip_ref_to_col = 'B'
  ip_ref_to_row = '4'
  ip_sername = 'Monsters by Color'.

lo_pie->set_style( zcl_excel_graph=>c_style_15 ).

"Show the category names next to the pie chart
lo_pie->set_show_cat_name( zcl_excel_graph_pie=>c_show_true ).

"Time to create a generic "drawing" object
lo_drawing = lo_worksheet->excel->add_new_drawing(
ip_type = zcl_excel_drawing=>type_chart
ip_title = 'Monster Pie Chart' ).

lo_drawing->graph = lo_pie.
lo_drawing->graph_type = zcl_excel_drawing=>c_graph_pie.

DATA: ls_upper TYPE zexcel_drawing_location,
      ls_lower TYPE zexcel_drawing_location.

ls_lower-row = 20. "Bottom left ia A20
ls_lower-col = 10. "Bottom right is J20 (ten columns)

lo_drawing->set_position2(
  EXPORTING
    ip_from = ls_upper
    ip_to = ls_lower ).

lo_drawing->set_media(
  EXPORTING
    ip_media_type = zcl_excel_drawing=>c_media_type_xml ).

lo_worksheet->add_drawing( lo_drawing ).

"The value sheet for the pie chart is hidden by default
lo_excel->set_active_sheet_index( 2 ).
lo_worksheet = lo_excel->get_active_worksheet( ).
lo_worksheet->zif_excel_sheet_properties~hidden = zif_excel_sheet_
properties=>c_hidden.

```

"You want the user to start on the first worksheet
 lo_excel->set_active_sheet_index(1).

Listing 11.24 Setting Up the Pie Chart Worksheet

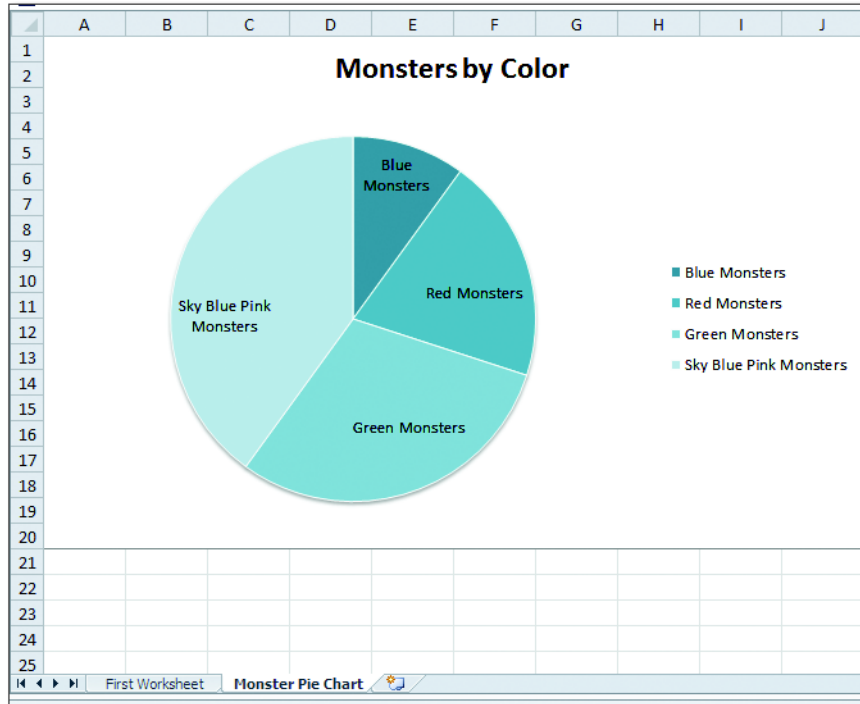


Figure 11.12 Pie Chart

The drawing object can also accept bar charts and line charts. The code to do so is slightly different than the pie chart example, but it follows the exact same principles: creating a specific chart type object, setting its specific properties, and then adding it to a generic drawing object. The examples that come with ABAP2XLSX show how to do this.

11.2.7 Embedding Macros

Most companies are crawling with spreadsheets that have had all sorts of fancy macros added. Usually, once a month (or however frequently there is a need for a given SAP report to be run) a report is downloaded into a spreadsheet from SAP, and then a macro is run to perform some conversions on the data to save

somebody from having to go through this process manually each time. As you might imagine, the whole purpose of ABAP2XLSX is to remove the need for this sort of thing. However, that still leaves the other main purpose of macros, which is to let the user navigate around complicated spreadsheets in the same way that you can click icons within SAP and jump into related reports and transactions.

Realistically, the VBA (Visual Basic for Applications) programming language used inside Excel spreadsheets is powerful enough that the range of things the macros might be doing is limitless. The inventor of ABAP2XLSX decided it would not be a very good idea to replicate the VBA language inside of SAP and thus enable direct creation of a spreadsheet with a macro in the same way that you can generate a spreadsheet with formatting and colors and charts.

You still want your users to be able to use their macros, though; if they cannot, they'll burst into tears. Some of these macros are no doubt works of art, making SAP programs seem simplistic by comparison. The approach to take using ABAP2XLSX is as follows:

1. Create a blank spreadsheet that contains all the desired macros.
2. Upload this blank, macro-enabled spreadsheet into SAP via ABAP2XLSX.
3. Download the blank sheet into the SAP database, where it will serve as a template.
4. When the SAP report is run, upload this template from the database; it becomes your Excel object.
5. Populate the Excel object with the report data, and start formatting it as per usual, with fonts, print settings, and all the trimmings, and then save the result to the local drive or email it out.

When the final consumer opens the spreadsheet, he will find his macros magically working and will therefore not burst into tears, unless they're tears of joy.

The next sections will cover each of the preceding steps one at a time.

Creating the Blank, Macro-Enabled Worksheet

On my very first day at work at my organization, I was a student working during the summer prior to my final year at university. I was given a really boring task to do, which involved manipulating a spreadsheet so that the data would come out in a format that could then be given to a room full of people, who would punch

the results into the mainframe. Within half an hour, I was bored and started looking for an easier way to do what I was doing; thus, I discovered macros. This was not even in Excel; back in those days, it was Lotus 1-2-3. Nonetheless, the principle was the same, and when Excel came along I was writing macros all over the place. In 1997, I made the jump to SAP, and I haven't written a macro since.

Therefore, I'm not going to give a fancy example here; instead, I'll present the bog-standard Microsoft equivalent of the "hello world" macro, which you'll find right at the start of their training documentation. In this example, you will use a macro to set the spreadsheet name. To do so, open a blank spreadsheet, go to the macro section, and type the code in Listing 11.25.

```
Private Sub Workbook_Open()
    Sheets("Sheet1").Select
    Sheets("Sheet1").Name = "Name Set by Macro"
End Sub
```

Listing 11.25 Macro to Change the Name of a Worksheet

Save the blank spreadsheet as an XLSM file (macro-enabled format) to your local drive.

Uploading the Blank Spreadsheet into SAP

I will not show any code for making the user (which is you, in this case) select the file path, because that is 100% normal SAP code that you have probably written a million times before. Suffice to say that there is a selection screen to choose the file path and the name of the template to be stored.

In Listing 11.26, you start off knowing where the file lives: the file path is in `LD_FULL_PATH`. Load the blank worksheet with the macros off of the desktop into SAP, and then transform it into an Excel object. For once, you don't want to modify that Excel object, so leave it as is, and transform it into an XML representation stored inside an `XSTRING`, which is the form the data needs to be in if you're going to be able to save it to the database.

```
START-OF-SELECTION.
  DATA: lo_excel          TYPE REF TO zcl_excel,
        lo_excel_writer  TYPE REF TO zif_excel_writer,
        lo_excel_reader  TYPE REF TO zif_excel_reader,
        ld_object_as_xstring TYPE xstring.

  CREATE OBJECT lo_excel_reader TYPE zcl_excel_reader_xlsm.
```

```
CREATE OBJECT lo_excel_writer TYPE zcl_excel_writer_xlsm.
```

```
"Where did I say the file lived?  
ld_full_path = p_path.
```

```
"Load the empty template with the macros  
lo_excel = lo_excel_reader->load_file( ld_full_path ).
```

```
"Transform the resulting EXCEL object into a whacking  
"great XSTRING  
ld_object_as_xstring = lo_excel_writer->write_file( lo_excel ).
```

Listing 11.26 Turning a Blank Spreadsheet into an XSTRING

Saving the Template in the SAP Database

Create a transparent table to store your templates inside SAP. You could get all fancy and store them in the MIME repository, but a quick and dirty solution is to go into Transaction SE11 and create a table like the one shown in Figure 11.13.

The screenshot shows the SAP SE11 table definition for ZTEXCEL_TEMPLATE. The table is active and has a short description of 'Excel Templates'. The 'Fields' tab is selected, showing the following table structure:

Field	Key	Initia...	Data element	Data Type	Length	Decimal...	Short Description
TEMPLATE_NAME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	CHAR30	CHAR	30	0	30 Characters
RAW_DATA	<input type="checkbox"/>	<input type="checkbox"/>	XSTRING	RAWSTRING	0	0	XString

Figure 11.13 Excel Template Storage Table

Then, you need to store the XSTRING version of the Excel object in the database (Listing 11.27). There's nothing fancy here; set the primary key of the table entry to be the template name, add the XSTRING data, and then use the MODIFY statement to store the data.

```
DATA: ls_template TYPE ztexcel_template.
```

```
"What did we say the template name was?  
ls_template-template_name = p_temp.  
ls_template-raw_data      = ld_object_as_xstring.
```

```
MODIFY ztexcel_template FROM ls_template.
```

Listing 11.27 Storing the Spreadsheet Template in the Database

Retrieving the Template from a Custom Report

Now, you'll modify the monster example report so that instead of creating a new workbook you will instead start by retrieving your example template from the database. Put a checkbox on the front screen to let the user decide if he wants to use the template with macros. Listing 11.28 performs a database read to get the template in XSTRING format and then converts it into an Excel object. Then, create a worksheet object; the Excel and worksheet objects, taken together, will make all the difference when you call the CONVERTER.

```
*-----*
* Retrieve template from the database
*-----*
DATA: ls_template      TYPE ztexcel_template,
      lo_excel_reader  TYPE REF TO zif_excel_reader,
      lo_worksheet     TYPE REF TO zcl_excel_worksheet.

IF p_macro = abap_true.

    CREATE OBJECT lo_excel_reader TYPE zcl_excel_reader_xlsm.

    SELECT SINGLE *
          FROM ztexcel_template
          INTO CORRESPONDING FIELDS OF ls_template
          WHERE template_name = 'MONSTER_EXAMPLE'.

    lo_excel = lo_excel_reader->load( ls_template-raw_data ).

    lo_worksheet = lo_excel->get_active_worksheet( ).

ENDIF. "Do we want to use a macro?"
```

Listing 11.28 Uploading an Excel Template

Now you have an Excel object and a worksheet object; they're empty, but they have macros hiding within them. Listing 11.29 fills up the blank worksheets with report data by using the CONVERTER mentioned in Section 11.2.1.

```
lo_converter->convert(
    EXPORTING
        io_alv      = go_view->mo_alv_grid
        it_table    = gt_monsters[]
        i_table     = abap_true
        i_style_table = zcl_excel_table=>builtinstyle_medium2
        io_worksheet = lo_worksheet
    CHANGING
        co_excel    = lo_excel ).
```

Listing 11.29 Filling the Macro-Enabled Worksheet with Data

When you call the `CONVERTER` in Listing 11.29, you're passing in bound instances of the Excel and worksheet objects (because you just created them in Listing 11.28), so the `CONVERTER` will use those and add the report data to the specified worksheet. If the worksheet and Excel objects were not yet created when you called the `CONVERTER`, then the `CONVERTER` would have created them for you.

To recap, when you want to use macros in a spreadsheet generated from ABAP2XLSX, create the Excel object from a template stored in the database so that you retain any macros therein. Otherwise, if you're not interested in macros, then create a new, blank Excel object.

Modifying and Saving the Excel Object

Once created, the Excel object can be modified in the exact same way as has been demonstrated in Section 11.2.1 through Section 11.2.6, by using the standard ABAP2XLSX classes and methods. The only tricky bit is that you need to do some jiggery pokery just before converting the Excel object into XML to make sure that the spreadsheet is downloaded with the correct extension. You cannot trust the users to get this right; you have to force the issue.

This jiggery pokery is shown in Listing 11.30. You want to know if the Excel object contains any macros, and the easiest way to find out is to ask the object itself (this is an OO programming principle called "ask, don't tell"). The object has an attribute with the words `VBA_PROJECT` in its name; this attribute will only be populated if the spreadsheet has a macro.

Once you know if the Excel object is macro enabled or not (see the `LF_MACRO_ENABLED` flag in Listing 11.30), you can check the file extension the user has entered, correct it if needed, and also create the correct subclass for converting the object into XML. If you get the extension or subclass wrong, then the user will not be able to open the downloaded spreadsheet; he will only get an error message about corruption in high places.

```
*-----*
* Convert to XML
*-----*
DATA: ld_xml_file      TYPE xstring,
      lo_excel_writer  TYPE REF TO zif_excel_writer,
      lf_macro_enabled TYPE abap_bool.

IF lo_excel->zif_excel_book_vba_project~codename_pr IS NOT INITIAL.
```

```

    lf_macro_enabled = abap_true.
ELSE.
    lf_macro_enabled = abap_false.
ENDIF.

IF ld_fullpath CS 'XLSM' AND lf_macro_enabled = abap_false.
    REPLACE 'XLSM' WITH 'XLSX' INTO ld_fullpath.
ELSEIF ld_fullpath CS 'XLSX' AND lf_macro_enabled = abap_true.
    REPLACE 'XLSX' WITH 'XLSM' INTO ld_fullpath.
ELSEIF ld_fullpath CS 'XLSX' OR
    ld_fullpath CS 'XLSM'.
    "Do nothing - everything is fine"
ELSEIF ld_fullpath CS 'XLS' AND lf_macro_enabled = abap_false.
    REPLACE 'XLS' WITH 'XLSX' INTO ld_fullpath.
ELSEIF ld_fullpath CS 'XLS' AND lf_macro_enabled = abap_true.
    REPLACE 'XLS' WITH 'XLSM' INTO ld_fullpath.
ENDIF.

IF ld_fullpath CS 'XLSM'.
    CREATE OBJECT lo_excel_writer TYPE zcl_excel_writer_xlsm.
ELSE.
    CREATE OBJECT lo_excel_writer TYPE zcl_excel_writer_2007.
ENDIF.

ld_xml_file = lo_excel_writer->write_file( lo_excel ).

```

Listing 11.30 Making Sure the Spreadsheet File is Saved Correctly

Note

At time of writing, drawings do not appear to work in macro-enabled workbooks. I was hoping that ABAP2XLSX V7 would fix this problem, but it did not—so this is crying out to be fixed by somebody who wants to contribute to the project.

11.2.8 Emailing the Result

Traditionally, if you wanted to download a spreadsheet from an ALV report in SAP, then naturally you had to be logged on and had to run the report online. As might be imagined, some reports generate so much data that they take quite a while to run, so if executed online they would terminate with a `TIME OUT` short dump.

Therefore, if you wanted to download a long-running report to a spreadsheet (and these types of reports are generally the ones people do want to download), then you ran the report as a batch job, and either the result had to be downloaded

to the application server (which is painful, because IT doesn't really want your average person to have access to the application server) or the user had to open up the spool request and download it (which gives you random blank columns at the front and top, with the headers repeating throughout the sheet; it involves tons of manual formatting of the downloaded spreadsheet).

With ABAP2XLSX, there's a miracle solution to this problem. Run the report in the background and then email the resulting spreadsheet; you can send it to yourself or to a distribution list. The spreadsheet generated from the SAP report appears as an attachment in the email.

There is nothing extraordinary about the code that follows (Listing 11.31). You can see it on the Internet in a million places, almost line for line in every case. The only thing happening differently here is that the XML string has been generated from an Excel object.

```
* Local Variables
DATA: lo_send_request TYPE REF TO cl_bcs,
      lo_document     TYPE REF TO cl_document_bcs,
      lo_recipient    TYPE REF TO if_recipient_bcs,
      lo_bcs_exception TYPE REF TO cx_bcs,
      lt_main_text    TYPE bcsy_text,
      lf_sent_to_all  TYPE os_boolean,
      ld_bytecount    TYPE i,
      ld_filelen      TYPE so_obj_len,
      lt_file_tab     TYPE solix_tab,
      ld_email        TYPE ad_smtpadr.

"Start off with the XSTRING derived from the EXCEL object
"Convert this to a format that the standard SAP emailing
"class understands
  lt_file_tab = cl_bcs_convert=>xstring_to_solix(
    iv_xstring = id_file ).
  ld_bytecount = strlen( id_file ).

  TRY.
    "Off you go with the standard emailing code
      lo_send_request = cl_bcs=>create_persistent( ).

*The email body is a table with lines of element SOLI
  lt_main_text = lt_email_body.
  lo_document = cl_document_bcs=>create_document(
    i_type      = 'RAW'
    i_text      = lt_main_text
    i_subject   = id_email_subject ).
```

```

* Fill a table with any attachments the email you have,
* which in this case is only one, the spreadsheet
  ld_filelen = ld_bytcount.

  DATA lt_att_head   TYPE soli_tab.
  DATA lv_text_line  TYPE soli.
  CONCATENATE '&SO_FILENAME=' id_attachment_subject
  INTO lv_text_line.
  APPEND lv_text_line TO lt_att_head.

lo_document->add_attachment(
i_attachment_type   = 'EXT' "#EC NOTEXT
i_attachment_subject = id_attachment_subject
i_attachment_size   = ld_filelen
i_att_content_hex   = lt_file_tab
i_attachment_header = lt_att_head ).

* Separation of concerns - the document class's job is to be a
* document. The send request class is concerned with sending things
  lo_send_request->set_document( lo_document ).

* You have a table of email addresses. Usually for multiple people
* you would define a distribution list
  LOOP AT it_email_addresses INTO ld_email.
* Email recipients are objects too, everything is an object
  lo_recipient =
  cl_cam_address_bcs=>create_internet_address( ld_email ).

* One more object for the send request
  lo_send_request->add_recipient( lo_recipient ).

  ENDLLOOP."Email addresses

* The actual send method has a return parameter that is blank if
* the send fails
  lf_sent_to_all =
  lo_send_request->send( i_with_error_screen = 'X' ).

  COMMIT WORK.

  IF lf_sent_to_all IS INITIAL.
    "Document not sent to &l
    MESSAGE i500(sbcoms) WITH ld_email
  ELSE.
    MESSAGE s022(so). "Document sent
    "kick off the send job so the email goes out immediately
    WAIT UP TO 2 SECONDS. "ensure the mail has been queued
    SUBMIT rsconn01
      WITH mode   = '*' "process everything you find.
      WITH output = ' '

```

```

        AND RETURN.
    ENDIF.

    CATCH cx_bcs INTO lo_bcs_exception.
    * If something goes wrong you probably want to send a simple failure
    * message to the user and a more detailed one to the application log
    ENDTRY.

```

Listing 11.31 Sending a Spreadsheet by Email

You will find that the ability to send out SAP-generated spreadsheets by email is really the jewel in the crown of ABAP2XLSX. You will want to give people a `P_EMAIL` parameter on the selection screen of all ABAP2XLSX-enabled reports together with a `SEND SPREADSHEET BY EMAIL` checkbox so that when the report runs (in the foreground or the background) a spreadsheet is emailed to the address specified in the `P_EMAIL` selection parameter.

You can default to the email address of the current user by using `BAPI_USER_GET_DETAIL`, or if you are using SAP ERP Human Capital Management (HCM), then usually you can find the email address in Infotype 0105. Because the email address is a selection parameter, the user can change his email address and send the spreadsheet to someone else, or even to a distribution list.

11.2.9 Adding Hyperlinks to SAP Transactions

One of the best features of SAP reports (and SAP in general) is that when you see something on the screen you are interested in, like a sales order number, you can double-click it and be directed to the underlying document. However, if you send people a spreadsheet and they go through it and find a sales order number they really want to look at, then naturally they can't just double-click the cell and have the SAP screen pop up to show them—for example—Transaction VA03 (Display Sales Order). That would be impossible.

Or would it?

In a normal Excel spreadsheet, you can have hyperlinks that, when clicked, take you to a web page of some sort. ABAP2XLSX has a hyperlink object, which is concerned with generating such URLs for embedding in spreadsheets generated from SAP.

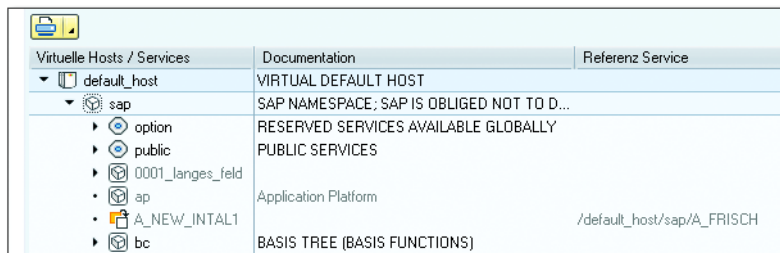
It has been said that one of the great leaps forward SAP has made in recent years is the introduction of the ICF (Internet Connection Framework), which enables

the SAP system to respond to incoming messages from the Internet. If a spreadsheet can send a message via a URL and SAP can respond to such a message, then you have all the building blocks in place to achieve what's needed.

In this section, you will kill two birds with one stone. The primary focus is on enabling hyperlinks from out of Excel into SAP, but this means that you have to touch on the basics of using the ICF. In fact, you will learn about setting up the ICF settings first; the section will end with the ABAP2XLSX coding that you need.

Setting Up the ICF for Hyperlinks

It should not come as too much of a life-changing shock to find out that ICF settings are made using Transaction SICF. When you run that transaction, you will see a tree structure, as shown in Figure 11.14, with the headings in an endearing mixture of English and German.



Virtuelle Hosts / Services	Documentation	Referenz Service
default_host	VIRTUAL DEFAULT HOST	
sap	SAP NAMESPACE; SAP IS OBLIGED NOT TO D...	
option	RESERVED SERVICES AVAILABLE GLOBALLY	
public	PUBLIC SERVICES	
0001_langes_feld		
ap	Application Platform	
A_NEW_INTAL1		/default_host/sap/A_FRISCH
bc	BASIS TREE (BASIS FUNCTIONS)	

Figure 11.14 Transaction SICF

In essence, each node in the tree corresponds to a part of the incoming URL, so if you add a node under the `SAP/BC` node, then the URL should contain the words `sap/bc`. This will become more obvious when you start to build up the URL from within SAP a bit later.

As it turns out, SAP is expecting you to want to call a transaction from outside of SAP and has provided a handy home from which you can create a custom service to handle the incoming call. The path to this service (i.e., what nodes you navigate to in the tree before creating your own custom entry) is shown at the top of Figure 11.15.

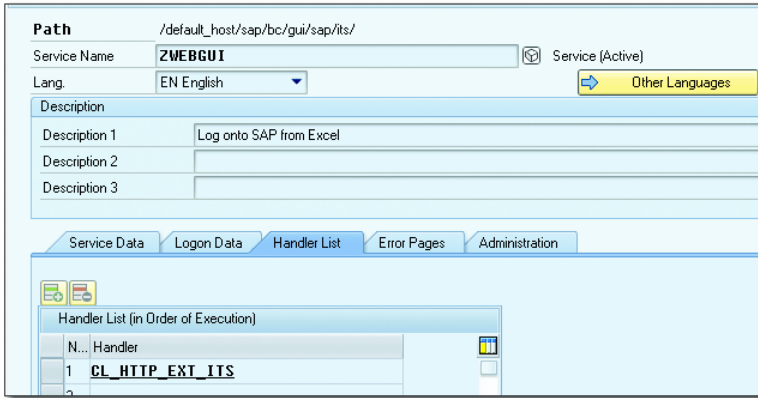


Figure 11.15 Custom ICF Service for Call Transaction

You might think that you would have to create a custom class to do the call transaction, but as it turns out `CL_HTTP_EXT_ITS` was created with this task in mind. If you did have to create a custom handling class, then it would have to implement the `IF_HTTP_EXTENSION` interface. As you can see, you can have as many handling classes as you like, so if the first one can't take the pressure of dealing with the incoming URL, then it can give up and pass the task on to the next handler. You do not even have to create a Z service, really; there's a standard ICF service under the menu path `/default_host/sap/bc/gui/sap/its/` with the name `webgui` that is designed for handling incoming transaction codes. (However, you might want to make a copy in case you feel the need to change anything down the track.)

The standard SAP class `CL_HTTP_EXT_ITS` is expecting a transaction code to be passed into it and will then execute a `CALL TRANSACTION`, displaying the results in a browser (SAP GUI for HTML).

ABAP2XLSX Coding for Hyperlinks

Now, you'll enhance the monster report so that when a user double-clicks the monster number in the generated spreadsheet a call will be made to Transaction `ZMONSTER` in SAP to display the monster master record. (All the other examples of this functionality involve passing your user name to `SU01` to display your own name, but (a) that's boring, and (b) you already know what your own name is.)

You want to make sure that your internal report table is sorted in the same order that the data will appear in the spreadsheet; otherwise, this next bit will not work

at all. Listing 11.32 shows the code to put the hyperlinks into the spreadsheet so that you get a nice blue line under the monster numbers in the spreadsheet. That blue line just screams out to the user, "Click me!"

```
*-----*
* Hyperlinks
*-----*
CONSTANTS: lc_monster_column TYPE zexcel_cell_column_alpha
VALUE 'A'.

DATA: lo_hyperlink TYPE REF TO zcl_excel_hyperlink,
      ld_url        TYPE string,
      ld_parameter  TYPE string,
      ld_tcode      TYPE sy-tcode,
      ld_ok_code    TYPE string,
      ls_monsters   LIKE LINE OF gt_monsters.

* Now loop through the spreadsheet, adding hyperlinks so the
* user can drill down into the original document in SAP
ld_sheet_row = ld_first_data_row.
LOOP AT gt_monsters INTO ls_monsters.
* Drill down into the monster master record
IF ls_monsters-monster_number IS NOT INITIAL.
  ld_parameter = 'ZTVC_MONSTERS-MONSTER_NUMBER=' &&
ls_monsters-monster_number && ';'.
  ld_url        = build_hyperlink_url( id_transaction = 'ZMONSTER'
id_parameters  = ld_parameter
id_ok_code     = '=ONLI'
id_user_name   = sy-uname ).
  lo_hyperlink =
zcl_excel_hyperlink=>create_external_link( iv_url = ld_url ).
  "In goes the hyperlink
  lo_worksheet->set_cell(
ip_column      = lc_monster_column
ip_row         = ld_sheet_row
ip_value       = ls_monsters-monster_number
ip_hyperlink   = lo_hyperlink ).
  "Now make it look like a hyperlink
  lo_worksheet->change_cell_style(
ip_column      = lc_monster_column
ip_row         = ld_sheet_row
ip_font_color_rgb = zcl_excel_style_color=>c_blue
ip_font_underline = abap_true ).
ENDIF.
ADD 1 TO ld_sheet_row.
ENDLOOP. "Monster Table
```

Listing 11.32 Inserting Hyperlinks in a Spreadsheet

You're halfway there; you'll notice that the creation of the URL is encapsulated in its own method, the code of which is shown in Listing 11.33. This code builds up a URL to log onto an SICF service. The SICF service will redirect to a Java system, which will authenticate the user via the Kerberos logon ticket created when the user logged onto Windows.

After authentication, the Java stack will redirect back to the SICF service, which will then log the user on without asking for his user name or password, just like normal SAP SSO. If the Java service is not there, then the user is asked for his name and password (twice).

```
METHOD build_hyperlink_url.
```

```
* Local Variables
```

```
DATA: ld_logical_system TYPE t000-logsys,
      ld_rfc_destination TYPE rfcdst,
      ld_server_name     TYPE rfchost_ext,
      ld_url              TYPE string.
```

```
* T000 - Fully Buffered
```

```
SELECT SINGLE logsys
  FROM t000
  INTO ld_logical_system
  WHERE mandt = sy-mandt.
```

```
CHECK sy-subrc = 0.
```

```
ld_rfc_destination = ld_logical_system.
```

```
CALL FUNCTION 'DEST_RFC_ABAP_READ'
  EXPORTING
    name          = ld_rfc_destination
  IMPORTING
    server_name   = ld_server_name
  EXCEPTIONS
    read_failure = 1
    OTHERS       = 2.
```

```
IF sy-subrc <> 0.
  RETURN.
ENDIF.
```

```
* First half of the URL - SICF path and first parameter i.e.
```

```
* transaction code
```

```
rs_url =
'https://' &&
ld_server_name &&
':44300/sap/bc/gui/sap/its/zwebgui?' &&
```

```

"44300 is the standard port for HTTPS
 '~transaction=*' &&
 id_transaction.

* Second half of the URL i.e. what data to be pass to the
* transaction code
  ld_url =
  id_parameters &&" e.g. USR02-BNAME=hardyp;
  'DYNP_OKCODE=' &&
  id_ok_code &&
  '&sap-client=' &&
  sy-mandt .

  CONCATENATE rs_url ld_url INTO rs_url SEPARATED BY space.

ENDMETHOD. "Build Hyperlink URL

```

Listing 11.33 Building the Hyperlink URL

Once the code in Listing 11.33 has built up the URL and this URL appears in the spreadsheet as a hyperlink, the user can click on that link to call up the transaction. However, the main problem with using hyperlinks to call SAP transactions is the single sign-on mechanism. To say this is complicated is the greatest understatement in the history of the universe, and explaining it is well outside the scope of this book. In order to set the system up so that a user is not asked for his password every time he uses a hyperlink to access SAP, you have to find an SAP Basis person who is a megagenius. Otherwise, the user does indeed have to enter his user name and password while drilling down.

11.3 Tips and Tricks

The last major section of the chapter will be about making even more out of the ABAP2XLSX framework. First, you will see what to do when you have a fix that you need that is perhaps not applicable to the open-source project as a whole. Then, you will discover how to achieve the functionality described all throughout Section 11.2 without having to add bucket loads of code to all of your custom reports.

11.3.1 Using the Enhancement Framework for Your Own Fixes

Someone once said about cloud platforms such as SuccessFactors and Ariba that "The best thing is that you get updates every three months, and the worst thing is

that you get updates every three months." Traditionally, updating software is a Bad Thing, because you have to retest everything.

With open-source projects, this "problem" is even worse, because you generally have a daily build—yes, it changes frequently—as well as the latest stable release. If you run into a problem with ABAP2XLSX or other open-source frameworks and have a look on the Internet, then you may well find that someone in the Outer Hebrides has encountered and fixed that very same issue five minutes ago. That's great, you may think; all you need to do is download the latest daily build, and all will be well.

That's fine—unless you've been experimenting with the framework and have made some changes yourself to fix or enhance assorted classes in order to fix problems specific to your business. You may not have them working properly yet, or the change may not yet have been merged with the global open-source project, or your change may have been rejected by the project owner as not universally applicable.

When you download the latest daily build, you receive the latest version of ABAP2XLSX as a nugget file, which will update all classes and thus wipe out any changes you have made—just like an SAP upgrade, but without the SPAU (though the version management still tells you what has changed). Naturally, you want to keep your changes and not have to list them on a spreadsheet (as much as we all love spreadsheets) and then manually reapply them after installing the new version.

Does this problem sound familiar? It should, because it's encountered during normal SAP upgrades and support stack applications. Earlier in this book (in Chapter 6), you read about how to get around this using the enhancement framework. This works just as well for open-source projects like ABAP2XLSX. In the following example, you will make a change to a standard ABAP2XLSX class that was rejected as inappropriate by the project at large. You want to keep your change, but you also want to upgrade from V6 of ABAP2XLSX to V7, overwriting everything yet keeping your change.

The change in this example is in regard to calculating the column widths; in essence, the problem is that dates are appearing in the spreadsheet with a column width slightly too small, so the user cannot see the entire date and has to widen the column manually. Because the fix would not work in all circumstances—fair enough—it cannot be incorporated in the project as a whole.

However, it works for you, so you want to keep it. What do you do? You do just what you would if you had “repaired” a standard SAP program. You use the Enhancement framework to isolate your change, as described in Chapter 6, and then when the update happens your change is magically still there.

In this example, go to the standard ABAP2XLSX class `ZCL_EXCEL_WOKSHEET` in display mode, and click the ENHANCEMENT icon (which looks like a seashell), or press `[Shift] + [F4]`. Then, right-click the method you want to replace—in this case, `CALCULATE COLUMN WIDTHS`—and choose the `INSERT OVER WRITE METHOD` option from the resulting context menu, which once again is a mix of English and German. The result is a new column, as seen in Figure 11.16.

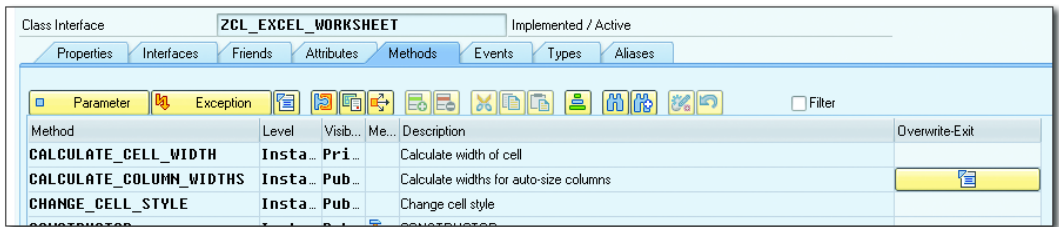


Figure 11.16 Adding an Overwrite Exit Method via the Enhancement Framework

Clicking the icon in the overwrite column takes you to a source code editor, in which you can write some code to totally replace the standard method. The replacement method will naturally have the same signature. Delegate the method call to your own `EXTENDER` class, which is where you can store all your ABAP2XLSX extension code. The code for this is shown in Listing 11.34.

```
METHOD iow_zei_excel_extensions~calculate_cell_width.
*-----*
* Declaration of Overwrite-method, do not insert any comments here please!
*
* methods CALCULATE_CELL_WIDTH
* importing
*   !IP_COLUMN type SIMPLE
*   !IP_ROW type ZEXCEL_CELL_ROW
*   !IP_CURRENT_MAX type FLOAT optional
* returning
*   value(EP_WIDTH) type I
* raising
*   ZCX_EXCEL .
*-----*

zcl_excel_extender=>calculate_cell_width(
```

```

EXPORTING
  io_worksheet   = core_object " Worksheet
  ip_column      = ip_column   " Cell Column
  ip_row         = ip_row      " Cell Row
  ip_current_max = ip_current_max " Current maximum Width
RECEIVING
  ep_width      = ep_width ). " New Width

ENDMETHOD.

```

Listing 11.34 Coding an Overwrite Method

You will notice that in Listing 11.34 you cannot refer to the class you are enhancing by using the `me` keyword to refer to the class instance; instead, you have to use the term `CORE_OBJECT`. This is because when the overwrite method is called you aren't inside the main `ZCL_EXCEL_WORKSHEET` class; you're inside a generated class called `LCL_ZEI_EXCEL_EXTENSIONS`, which has an instance of the real class called `CORE_OBJECT` passed into it upon creation.

11.3.2 Creating a Reusable Custom Framework

When looking at the code samples throughout this chapter, you may have noticed that often it takes a large amount of code to achieve each function; the conditional formatting code is a case in point. Despite this, every time you find yourself wanting to cut and paste a large chunk of code out of one program and into another one and then change one or two input values, you should take a leaf out of the Spice Girls' book and stop right now. Obviously, this applies to programming as a whole: not just ABAP2XLSX, not just SAP, but all code everywhere. However, I'm talking about ABAP2XLSX specifically here, and although it's good to work out how to do each task, once you have that worked out it's time to hide the complexity in your own custom method of some sort of generic ABAP2XLSX Z custom class.

For this example, assume there is such a class in the system, and (for example) the code in Listing 11.33 is encapsulated in a method in that reusable Z class. It is the same for emailing or downloading a spreadsheet: In each case, you have about 20 lines of code with just one value varying for each report, so it's best to have a reusable Z method with all the code that never changes in it, taking the variable data as input parameters.

As an example, in Section 11.2.4 it was noted that you have to set two different input values for the conditional object: one saying what the rule is plus one of four possible input structures. You have to make sure that you fill up the correct structure—that is, the one with the same name as the chosen rule. You may get that wrong, but if you pass the variable values into a method, then you can write logic inside that method to make sure the correct structure is populated based on the rule type. You only need to get it right once, and then the possibility for errors of this sort going forward is removed.

None of what was just described is one of Stephen Hawking's deep mysteries of the universe, but the details are important to mention—in case anyone is scared off by the perceived large amount of code that would seem to be needed to use ABAP2XLSX based on the code samples in this chapter.

11.4 Summary

In this chapter, you were introduced to the open-source ABAP2XLSX framework, as invented by Ivan Femia and then improved by the open-source community; the purpose of the ABAP2XLSX framework is to vastly improve the integration of SAP and Excel.

This chapter concludes the discussion of desktop user interfaces that live in PCs, such as the SAP GUI and Excel. The next chapters turn to the world of transactions and reports that appear inside web browsers.

Recommended Reading

- ▶ ABAP2XLSX—Generate Your Professional Excel Spreadsheet from ABAP: <http://scn.sap.com/community/abap/blog/2010/07/12/abap2xlsx--generate-your-professional-excel-spreadsheet-from-abap> (Ivan Femia)
- ▶ Introducing the Office (2007) Open XML File Formats : http://msdn.microsoft.com/en-us/library/aa338205.aspx#office2007aboutnewfileformat_structureoftheofficexmlformats

*Can he swing from a web?
No he can't; he's a pig.
—Homer Simpson*

12 Web Dynpro ABAP and Floorplan Manager

Perhaps the most difficult thing to achieve in life is to make an ABAP developer jump into a new UI technology. Although Web Dynpro ABAP (WDA) has been around since 2005, it is still a technology that not all ABAPers use, and not by a long stretch. (Indeed, I once heard a story that on the same day one development department of a multinational company was having WDA training their colleagues in another country were being berated by their development manager for using ALV grids instead of `WRITE` statements.) To take matters further, Floorplan Manager (FPM), a technology that enhances WDA, is even less widely used—which is ironic, because FPM is where quite a lot of SAP development effort is focused. In any event, this chapter introduces ABAPers to both WDA and FPM.

Before diving into any technical content, it's important to get some naming problems out of the way. The following three statements are all equally true:

- ▶ WDA is rather like "classic" DYNPRO.
- ▶ JavaScript is rather like Java.
- ▶ A carpet is rather like a car.

In other words, to paraphrase Sesame Street, none of these things is just like the other, even though the names are similar. It's a big jump from module pool programming to WDA, which is one reason people don't want to make that jump.

Note

To make sure there's no confusion between classic DYNPRO screens and Web Dynpro ABAP clear, this chapter refers to the former in uppercase and to Web Dynpro ABAP as "WDA."

The second reason often given against jumping out of DYNPRO is that some people argue that WDA is obsolete even before it has entered widespread adoption due to the advent of SAPUI5. This book looks at WDA in this chapter and SAPUI5 in the next and lets you make up your own mind (i.e., treats you like an adult).

But let's not get ahead of ourselves. This chapter will teach you about WDA (including Floorplan Manager); where applicable, it uses the familiar concepts of classic DYNPROs in order to do so. Section 12.1 looks at the idea of the model-view-controller (MVC) concept and how SAP has attempted to bake this in to the WDA framework. Section 12.2 revisits the monster model from the previous chapters and the model you developed to display the Monster Monitor in an ALV grid. It also explains how to reuse this model to display data in a WDA application. Finally, Section 12.3 takes a peek at FPM, which is a WDA-based technology that aims to speed up WDA application development and at the same time provide a sort of uniformity of look and feel to disparate applications. As part of this, you will see how the BOPF monster business object created earlier integrates into FPM.

12.1 The Model-View-Controller Concept

WDA is based on the MVC concept of programming, which you've seen mentioned several times throughout this book. The MVC model helps to maintain a separation of concerns so that each part of the application has one job to do only and can change independently of the others; the WDA framework in SAP enforces the MVC pattern. (In some other programming languages, you literally do not have a choice and must do things this way. However, ABAP was designed for the monolithic programming style, so the WDA model is supposed to enforce "best practices," a common SAP catchphrase.)

Because you're now reading a chapter on a technology that holds this MVC concept at its core, you should have a basic grasp of what the three elements are. This section describes these elements and where they should reside inside a WDA application.

12.1.1 Model

A *model* is a class that contains the business logic of an object. It exposes to the outside world all one would wish to know about an object's attributes and behav-

ior. No presumptions are made about what sort of software application might make use of such a model or how the model's data might be presented to the outside world. In the recurring monster example, the model class would contain the business logic of the monster object.

In the various articles that you may read on the subject of WDA, there is almost complete agreement that the model class should not know or care that it is being accessed by a WDA application. Instead, you should create a model class before you have even decided what UI technology you are going to use to let the user view and manipulate the data the model encapsulates. In this way, the model is totally decoupled from the UI technology (in this case, WDA).

This decoupling provides two advantages:

- ▶ You can reuse existing model classes in new WDA applications (or indeed any sort of framework).
- ▶ You can change the innards of the model class, safe in the knowledge that this can have no possible effect on any WDA applications that may be using it.

In your WDA application, you are going to be using the `ZCL_MONSTER_MODEL` class found throughout this book. This class does not care what uses it, because a properly designed model class has no UI-related code in any of its methods. Queries can be made on the class or instructions given to it about data changes to be made, and results or exceptions are passed back, but it is not the job of the model class to care about how such results are to be presented.

Once the model class is created, the next question is where to put it. There are three options:

- ▶ Put the model inside the view.
- ▶ Use the model as an assistance class.
- ▶ Declare the model in the controller.

Each of these options is discussed next.

Model Inside the View

In general, putting the model inside the view is not considered a good approach. Because you should be able to swap out the model for the view (or vice versa) in your application, the model and the view should not know about each other. It is far easier to replace a view when, for example, the UI technology changes than it

is to change the model (although the latter is not impossible either). Therefore, putting the model inside the view is not the way forward; the model should go nowhere near this area. (Strangely, it has been said that “SAP forbids `SELECT` statements within views,” but this is not enforced by the syntax check—you don’t get a warning and it all works fine. So by “forbid,” it would seem that SAP presumes that you won’t do it if you’re told not to.)

Another recommendation is to not put any references to the model class (or any `SELECT` statements) in the view. Even if it is technically possible to do so, it paints you into a corner. You will see later on in this chapter how you use the code in the view to delegate tasks upwards (which, as an aside, is exactly what you do in classical DYNPRO modules as well).

Model as an Assistance Class

Using the model as an assistance class is another option. A WDA application looks somewhat like a tree structure, and right at the top you can declare an assistance class. This class then pops up automatically as a member variable of all the controllers within the application.

At first glance, this looks like the ideal place to put your model class, and some articles about WDA suggest just this approach. However, housing a model class does not seem to be what an assistance class was intended to do; its purpose seemed to be very WDA-specific and to revolve around handling texts (such as field labels) throughout the application. If you’re a purist, you’d also be uncomfortable with one class doing two disparate things; this would mean that you would have to make your model inherit from a WDA-specific abstract class. Your model class would then be tightly coupled to the WDA framework (the opposite of what you generally want to achieve) as well as doing two different things: model-type things and WDA-specific text things.

It is also possible to use the assistance class as a go-between linking the real model to the WDA application. In this scenario, the assistance class would still be doing two things, but you could reuse existing models and they would not be tied to the WDA framework. However, if you take this approach, then the views can call the model methods directly, and you are back to the problem with the model and the view tied together, so it is difficult to change one without the other. It seems that all those OO books (see the end of the chapter for some recommendations) actually put in all these rules about separation of concerns for a very good reason. As

a general rule, if you have two choices regarding how to do something, then you should usually veer toward the one that follows the OO principles.

Model Declared in the Controller

This leaves one last option: declaring the model in the controller. If you declare the model class in the main controller, it is instantiated right at the start when the controller is being created. That might be a backward-looking attitude, but it beats the other two options into a cocked hat: there is a clear separation of concerns and there is no way the view can know about the model and vice versa—so you can change one without having to change the other. This should make OO purists bounce up and down with joy, so this is the approach taken in the example later in this chapter.

12.1.2 View

A *view* is a part of the program that brings data up on the screen for the user to look at or interact with. The best examples inside SAP are functions such as `REUSE_ALV_GRID` or classes such as `CL_SALV_TABLE`. The job of a view is very simple in one sense; it displays data and controls what sort of commands a user can perform based on that data. Where this becomes an art form is that the view must present this information in a way that does not make the user physically sick. Thus far, the vast majority of applications have failed in this task.

In an ideal world, a view would be totally generic, like an ALV grid: pass in any internal table you want, and the view will display it in the best way it knows how.

Note

Just to confuse you totally, there are two elements that correspond to the classical definition of the term "view" in WDA: windows and views. The important thing to remember is that windows are the glue that stick views together. A window is like an empty fridge door onto which you stick several fridge magnets (the views).

If you can think of the window in WDA as sort of the same as an empty main screen in classical DYNPRO and the view as a DYNPRO subscreen, then you will recognize the same sort of concepts. (Now, some WDA fans are going to be foaming at the mouth with rage and chasing me from the town with tar and feathers, screaming "It's *completely* different!" Maybe it is, but drawing analogies always helps me understand things better; maybe it will help you also.) Hereafter, when this chapter refers to views, it is referring to WDA views. When it refers to WDA windows, it will specifically say so.

Using the analogy that views (which live inside windows) in a WDA application are somewhat similar to their DYNPRO counterparts (they are not really, as you will see), you'll soon see how the concepts you know and love in the DYNPRO world translate into their WDA equivalents. First, you'll learn about the WDA equivalent to Screen Painter, which is the good old tool you use to define views in the classic DYNPRO world. You will be relieved to know there is a WDA equivalent. Then, you'll learn about data in a view. In a DYNPRO screen, you work with global data; because there is no such thing in a WDA application, you'll learn how the data gets to and from the screen. Finally, just as a classic DYNPRO screen revolves around the PBO/PAI concepts defined in the screen flow, there is a very similar concept in WDA, and you will also learn about this.

Graphical Screen Painter

You create a DYNPRO screen using a graphical screen painter. When defining a WDA view there is still a graphical screen painter (though it has no name; it is just part of the WDA framework). As with DYNPROs, you have a little box popping up on the right in which you can change various properties of the screen element (e.g., right justified, read only, or what have you).

Section 12.2.3 talks about when and how you call this tool. You'll also see what it looks like, which is not all that different from what you're used to.

Storing Data in the Context

In classic DYNPRO, you declare a table area or global variables (usually both) and then paint fields on the screen with the exact same name; at runtime, the system automatically copies the values from the screen field to the identically named variable in your program. Although it looks like the data is only stored in one place, it isn't; during processing after user input, the global variable value might not change to the screen value until several statements are executed.

Conversely, in WDA it looks like the data is stored in two places; it looks like the context (a structure containing assorted data fields) in the controller corresponds to the data in the program, whereas the context in the view is the same structure, only this time holding the values on the screen that the user sees. In fact, when you map the two data structures (controller and view) together while creating the program, at runtime they both refer to the same area of memory—so a change to one affects the other and vice versa. However, the controller must explicitly

transfer that screen data to and from the model, which means that the screen data is global data no longer—which was the aim of the game.

Processing Before and After User Input

In DYNPRO screens, you manually code modules to validate data and handle user commands and the like. You split these modules into ones that get called before the user sees the screen (PBO) and modules that get called after the user has done something such as pressing a button (PAI).

PBO and PAI processing is much more structured in WDA than in traditional DYNPRO. There are assorted methods you can define that will get called before the screen is shown and assorted methods you can define that get called after the user has done something. For example, there are so-called Captain Hook methods, which correspond to points in the transaction flow such as just before the data is displayed, and buttons (for example) that have a direct link to an action, which is like handling a user command.

When it comes to the processing before the screen is shown, you can use the monster model as a base class and thus reuse the logic in the monster model without recoding the same thing in the WDA application. When the application retrieves data from the model class, derived items (like text descriptions) fill themselves out based on methods within the model. (In the case of the examples in this book, you are using the BOPF framework, but you could be just doing direct reads on text tables within the model class.) In a traditional DYNPRO application, you would most likely have PBO modules that go and get the text descriptions, but here you don't have to do a thing. The monster model is nice and reusable. (In the next chapter, you'll see how this also works the same way with an SAPUI5 application.)

When it comes to processing after user input, the navigation between screens is much more explicit in WDA than you will be used to. Instead of `CALL SCREEN` or `LEAVE TO SCREEN`, you leave the WDA screen via an outbound plug, which connects to an inbound plug on the screen you're jumping to.

A. A. Milne once wrote, "Where are you going to? I don't know. What does it matter where people go?" In WDA navigation, you always know where you are going and where you have come from, because you have to specify the target screen explicitly rather than saying `LEAVE TO SCREEN 0`—which could be anywhere.

12.1.3 Controller

A *controller* is any sort of computer program that knows about one or more models and one or more views; it's a sort of broker that handles communication between them. Inside a controller, you need to code business logic to decide if a certain view should be used to display the data of a certain model. When a view sends back a message saying the user has done something, you need to add code to the controller so that it can decide if the model needs to know this, or if the controller can deal with the input itself (e.g., an instruction to display exactly the same data in a different format can be handled by the controller simply calling up another view without having to bother the model). The controller must be careful never to let the view and the model know of each other's existence (otherwise the entire universe will explode—or something almost as terrible).

The controller concept in WDA is even more complicated than you might expect, because there isn't only one controller but a number of them. The thing to remember is that out of all the myriad controllers you may see, the component controller is the head honcho, numero uno, the leader of the pack, the king of the hill, and the undisputable top cat.

The controllers in WDA are generated automatically: the component controller at the start and other controllers as and when the windows and views are created. It is also possible to create additional controllers manually.

What is the "component" that is being controlled? A component can be thought of as the equivalent of the module pool program (it isn't really, but it's close enough). If you treat the two terms as interchangeable, then the component controller becomes the program controller, and the waters are slightly less muddy. The component controller can be accessed by all the views and can talk back to them, as might be expected; any other controllers knocking around are just willing accomplices.

There are "view controllers" and "window controllers," but they are not controllers in any meaningful sense of the word, because such controllers are one with their respective windows and views and never talk to the model like proper controllers do.

Thus, as mentioned earlier, just because something is called a controller, walks like a controller, and quacks like a controller, you shouldn't treat it like a controller if it isn't one; make sure all controller-type tasks are delegated to a real controller.

You decide you're interested in monster number seven, so select that row and click **SHOW SELECTED MONSTER**. Double-clicking the record should have the same effect, and users are used to this in the standard ALV. The result is shown in Figure 12.3.

Monster Header Record

Monster Number: 7

Monster Name: FRED

Monster Sanity %age: 3

Monster Color: GREEN

Monster Strength: REALLY STRONG

Monster Hat Size: 5

Monster Age in Days: 1

Monster Heads: 1

Sanity:

Hat Size:

View: * [Standard View] Print Version Export Filter Settings

Monster Number	Monster Item Number	Monster Part Category (Head /Wings / Tail etc.)	Monster Part Quantity e.g. 6 Heads	Text
7	10	HD	1	
7	11	AR	1	
7	12	AR	1	
7	13	LG	1	
7	14	LG	1	
7	15	TN	1	

Figure 12.3 WDA Monster Monitor: Displaying a Single Record

There are a fair few steps to this. In the rest of this section, you'll go through all of these steps, which are as follows:

1. Create the component, which is going to be the high-level structure under which you'll assemble all the building blocks needed for the Monster Monitor.
2. Set up some data structures to be used by the controller to pass data between the model and the view.
3. Make some settings for the views to define what the various screens are going to look like.
4. Use an ALV grid display in one of your screens to show a list of monsters.
5. Define some windows for your views to live inside.

6. Configure settings to let the user navigate from one window to another in order to see a different view.
7. Create the WDA equivalent of a transaction code, which is in fact a URL; this allows users to call the application.
8. (Here's the fun bit!) Do the necessary ABAP coding to bring the Monster Monitor to life.

12.2.1 Creating a Web Dynpro Component

Your first step in building a WDA application is to create a component. To do this, open Transaction SE80. In the dropdown box in the top left of SE80, choose WEB DYNPRO COMPONENT/INTERFACE. You're going to call the WDA component (program) `ZWDC_MONSTER_MONITOR`. When you click the DISPLAY button, naturally you get this message: WEB DYNPRO COMP./INTF. `ZWDC_MONSTER_MONITOR` DOES NOT EXIST. CREATE OBJECT? As it turns out, you do want to create the object, so say YES.

In Figure 12.4, you can see the resulting pop-up box, in which you have some questions to answer. Apart from obvious things like the text description, the system proposes names for the default window and a default view.

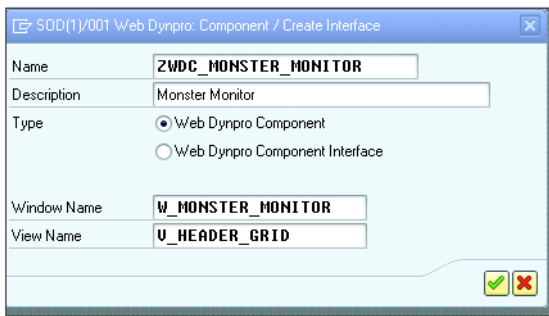


Figure 12.4 Creating a WDA Component

The proposed name of the default window is the same as the component itself. It's a good practice to change the name, because having different constructs with the exact same names referring to different things can make your head spin. The proposed name for the default view is `MAIN`, which is fair enough, but a more meaningful name would make it obvious that it's a view and would be related to what it's going to be displaying.

When you create a module pool transaction, you specify which screen you are going to be starting on; the same idea applies here. As noted earlier, when the WDA application is called, you're going to start on the `V_HEADER_GRID` view inside the `W_MONSTER_MONITOR` window. As soon as you're done specifying this and click the good old green checkmark, a whole bunch of objects are generated for you, as can be seen in Figure 12.5.

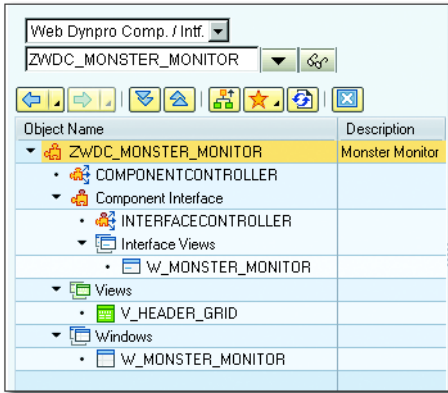


Figure 12.5 Empty WDA Component

You have your proper controller, another controller called the *interface controller* (which we are going to use in order to reuse the standard ALV functionality available in WDA), and your window and view (which are both views in the MVC sense). If you went into the root node and specified an assistance class, then that would be in the tree structure as well, and it would look like you had child nodes for a model, controller, and views. However, even if that looks nice, remember that an assistance class is not really the model but a sinister impostor that rises above its text-handling station to try and subjugate the entire universe and torture us all to death slowly in its salt mines—so don't do that.

12.2.2 Declaring Data Structures for the Controller

If you were writing a DYNPRO program, then right about now you'd be declaring `TABLE` statements in your `TOP INCLUDE` for DDIC structures and maybe some global variables that you knew you'd be using in your UI screen. In WDA, there is a clear separation between the data in the view, controller, and model layers, and you have to explicitly say that you want to move data between them. Even given that life is not so different in the WDA world, at this point you're still going

to declare the same sort of data, this time in the CONTEXT tab in the component (program) controller.

In this application, you're going to have three structures: an input structure (with two fields so that the user can search for monsters by name or by number), a table of monster headers, and a table of monster items. The first two will be used on the main screen and the last one when the user is looking at an individual monster record.

Navigate to the COMPONENTCONTROLLER node in your application. Move to the CONTEXT tab, and you'll see a little ball labeled CONTEXT. Right-click this rubber ball, which keeps bouncing back to you, and choose CREATE • NODE.

In the screen shown in Figure 12.6, you do not need to do anything; just accept all the default settings. (If you are curious what they all mean, `F1` help is actually helpful here—especially with respect to the SINGLETON field, where you are pointed to a blog that basically says, “Never switch this on ever or the universe will explode.”) Since you do not need to change any setting, just click the ADD ATTRIBUTES FROM STRUCTURE button.

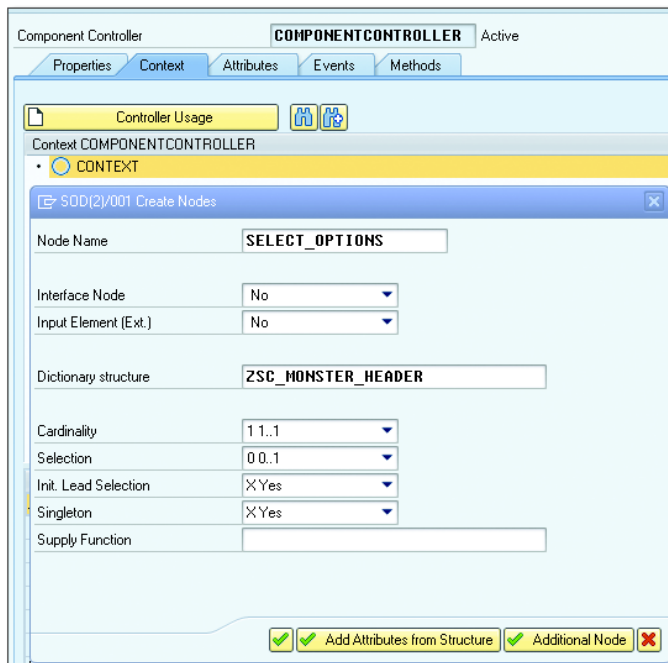
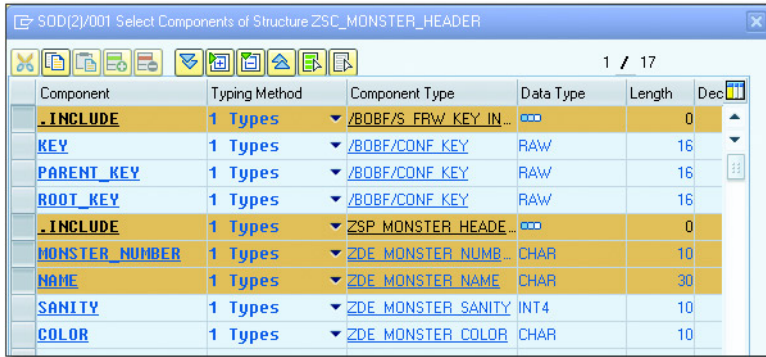


Figure 12.6 Creating a Selction Option Context: Part 1

In the screen shown in Figure 12.7, you only need two fields from the monster header structure, so select them and you're done. A `SELECT_OPTIONS` node appears under the root context node. Now, you need to create the two tables that are going to be used for ALV grid displays: one for a table of monster header records and one for a table of monster items for a single monster. In both cases, the procedure is the same. First, you go back to the root context node and do the `CREATE • NODE` trick again, just as before.



Component	Typing Method	Component Type	Data Type	Length	Dec
.INCLUDE	1 Types	/BOBF/S FRW KEY IN...	...	0	
KEY	1 Types	/BOBF/CONF_KEY	RAW	16	
PARENT_KEY	1 Types	/BOBF/CONF_KEY	RAW	16	
ROOT_KEY	1 Types	/BOBF/CONF_KEY	RAW	16	
.INCLUDE	1 Types	ZSP MONSTER HEADE	...	0	
MONSTER_NUMBER	1 Types	ZDE MONSTER NUMB...	CHAR	10	
NAME	1 Types	ZDE MONSTER NAME	CHAR	30	
SANITY	1 Types	ZDE MONSTER SANITY	INT4	10	
COLOR	1 Types	ZDE MONSTER COLOR	CHAR	10	

Figure 12.7 Creating a Selection Option Context: Part 2

Press the green checkmark to leave the screen shown in Figure 12.7. Next, you need to create the two tables that are going to be used for ALV grid displays: one for a table of monster header records and one for a table of monster items for a single monster. In both cases, the procedure is the same. First, you go back to the root context node and do the `CREATE • NODE` trick again, just as before.

This next node will be called the `MONSTER_HEADER_TABLE` node. The only change here is to change the `CARDINALITY` to be `0..N` (i.e., there can be no records, lots of records, or anything in between). Once again you click `ADD ATTRIBUTES FROM STRUCTURE`—but this time, when adding the attributes from the `ZSC_MONSTER_HEADER` structure, you select everything except the administrative fields. At this point you have defined the two node structures for the overview screen: a structure to hold the selection options and a structure to hold a table of selected monster header records.

Now you move on to creating the node structures for the detail screen: a structure to hold the header details of one selected monster, and a structure to hold a table of the items (components) of the selected monster. First, create the node to hold

the header details of one monster. Create a new node from the context root node and this time call the node `MONSTER_HEADER`. Define this node with the exact same fields as the header table has. The difference is going to be the usage; i.e., this will be used to display a single record rather than a table.

Next, create the node for the table of monster items. Go back to the root context node and repeat the exact same procedure you went through before, defining the `MONSTER_HEADER_TABLE` node for the `MONSTER_ITEM_TABLE` node. However, this time, use the `ZCS_MONSTER_ITEM` structure as opposed to the `ZSC_MONSTER_HEADER` structure. Once you've done this, the `CONTEXT` tab in your controller looks like the one shown in Figure 12.8.

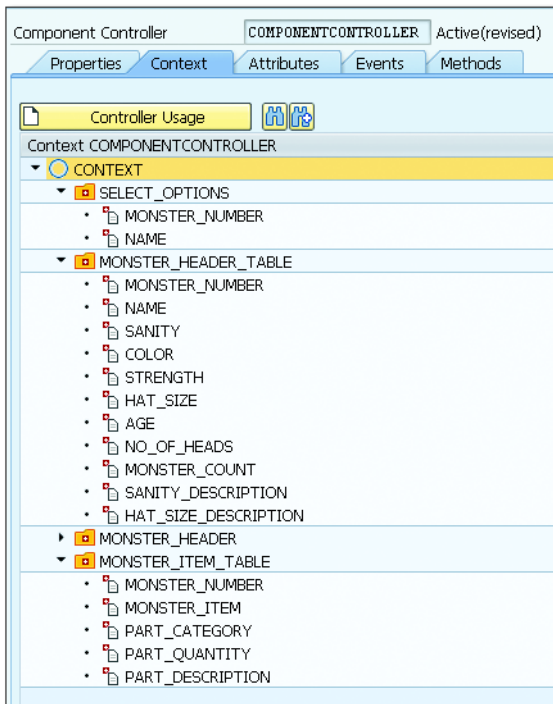


Figure 12.8 Complete Context Hierarchy

In WDA terms, you've created four nodes; this is important to remember, because from here on in when you want to read or write to these structures, you have to use the term "node." (WDA experts would wonder why I am belaboring this point; to them it's as plain as the node on your face.)

12.2.3 Establishing View Settings

Now, you'll be defining various screens (views) and parts of those screens in order to enable your transaction flow. First, you'll create a selection option field via which the user can enter a monster number and/or name. Then, you'll set up a button the user can press after he's entered his selections so that a monster search is triggered. Next, you'll define the view that will store the resulting list of monsters. Finally, you'll set up the view that will be used to display header details of a monster selected from the list and the associated list of monster items.

Defining the Select Options Part of the View

Now is the time to take the structure that is being used to store the select options and replicate that structure in the view (which, again, is how you'd start in a DYN-PRO program—by setting up the initial screen). Normally, you'd have at least a dozen fields on a selection screen; here, you're just going to use two for simplicity's sake: name and number.

Technically, the selection options are going to live in a view all of their own, so select your `ZWDC_MONSTER_MONITOR` root node, right-click, and choose `CREATE • VIEW`. Be radical: call this view `SELECT_OPTIONS`. You'll land on the `LAYOUT` tab of your new view. Navigate to the `CONTEXT` tab, which is where data structures have their home (Figure 12.9).

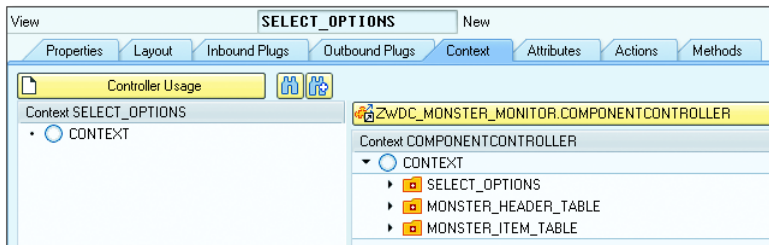


Figure 12.9 Creating a View for the Select Options

Select the `SELECT_OPTIONS` node from the controller box on the right and drag it over to the view box on the left. The result is that the `SELECT_OPTIONS` node is replicated inside the view box, and you get a message to the effect that a mapping has been defined. This means that there are now two identical structures (one in the view and one in the controller), and they are joined at the hip. You will also have

to drag the `MONSTER_HEADER_TABLE` over as well, as later on the `SELECT_OPTIONS` node will need this in order to properly handle the user pressing the `FIND SELECTED MONSTERS` button.

Section 12.2.2 talked about data structures, and now you'll turn to the real job of a view, which is looking after the appearance of such data. Navigate to the `LAYOUT` tab of your view, and you'll find that your cursor has landed on a node called `ROOTUIELEMENTCONTAINER`. Right-click on this, and choose `CREATE FORM CONTAINER`. A pop-up box appears that's just crawling with options. In this case, you want to use the context you just created, so click the `CONTEXT` button in the top-right-hand corner of the pop-up box.

In the resulting screen (Figure 12.10), you can choose the `SELECT_OPTIONS` node (not that you actually have much choice at the moment). The two fields appear in the pop-up box in a tabular display; click the green checkmark.

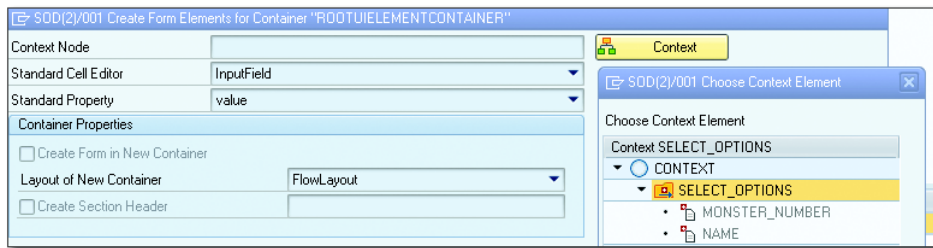


Figure 12.10 Creating a Form Container

Using Standard WDA Elements

In this example you have manually coded the two selection option boxes. In real life you would want to make use of a standard reusable Web Dynpro component `WDR_SELECT_OPTIONS` that defines “real” select options such as the ones you see on a selection screen, with all the extra functionality that entails, rather than having to code it all yourself. The “Recommended Reading” box at the end of the chapter has a link to instructions on how to do this.

Now, it's time to meet the WDA equivalent of the `DYNPRO` Screen Painter. Either press `CTRL` + `F10` or click the button that says `SHOW/HIDE LAYOUT PREVIEW`. The resulting screen (Figure 12.11) probably looks very familiar, at least in terms of structure.

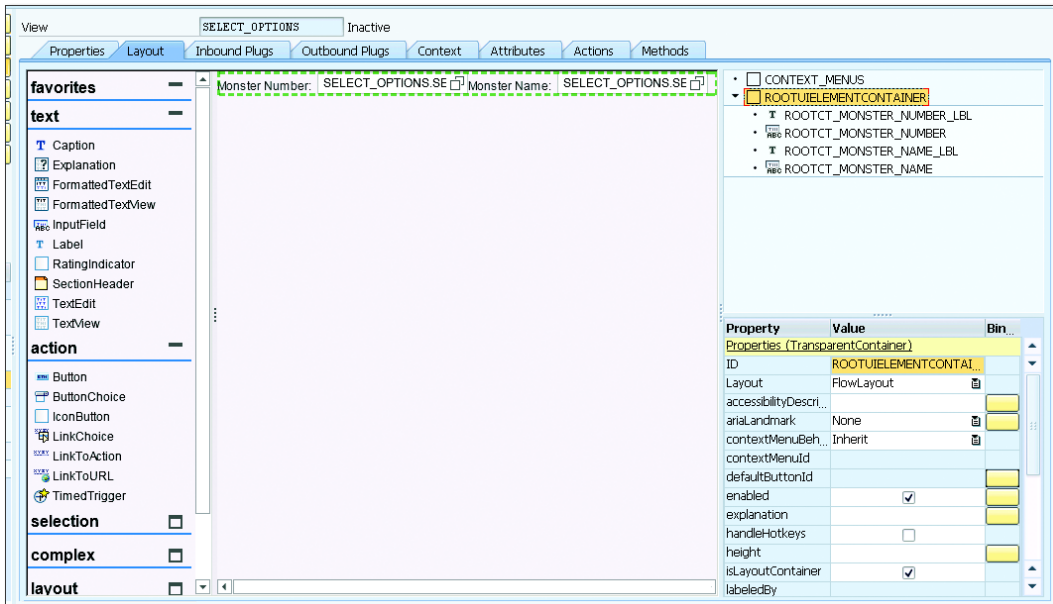


Figure 12.11 WDA Version of the Screen Painter

First of all, there is still a lot of battleship gray, a color that SAP developers find exciting. Just like the DYNPRO Screen Painter, there are lots of UI element things in a list to the left of the main screen. In the traditional Screen Painter, the list of properties of the current field floats in its own pop-up box; here, it's on the right-hand-side of the screen. There are far more options in the WDA equivalent.

Defining the Button that Triggers the Search

You need a button to trigger the database selection once the user has entered his requirements. Your cursor is still on the `ROOTUIELEMENTCONTAINER`, so right-click again; this time, choose `INSERT ELEMENT`. A tiny pop-up box appears with two fields: what you're going to call the new element (in this case, `FIND_MONSTERS`) and what sort of element this is (when you use the `[F4]` search, you can see that there are many things you can choose; this time, choose `BUTTON`).

Now, we all love pressing buttons, but a button that doesn't do anything is no sort of button at all. As it turns out, your newly created button element has a property near the bottom of its list of properties called `onAction`, next to an icon

that looks like a piece of paper. If you click that icon, then a pop-up box appears in which you can give your action a name (Figure 12.12). This is not that radically different from adding an icon to the application toolbar in the traditional DYNPRO Screen Painter.

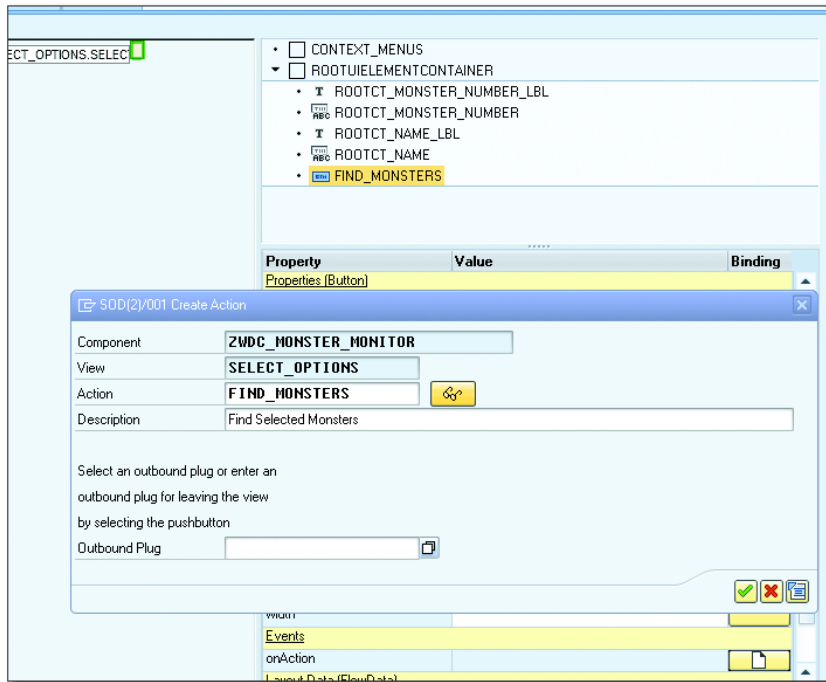


Figure 12.12 Defining an Action

The first thing you do when adding a button in the old DYNPRO Screen Painter is give it an icon. Naturally, you can do that as well here by via a dropdown on the IMAGE SOURCE property field. You also want to fill in the text and tooltip fields, in order to make the button a proper button that can hold its head up high in polite society. You can choose a standard SAP GUI icon, choose one from the new list that WDA gives you, or add your own.

In traditional DYNPRO, you then add some code to your USER_COMMAND routine. Here, a method to handle your newly created action is automatically created for you inside the view. You'll code this a bit later, but what you'll be doing is telling the controller that the button has been clicked. Anything else is outside of the

view's job description. (If it tried to handle the response itself, then it would find itself surrounded by a picket line of angry members of the controllers union accusing it of trying to put honest, hardworking controllers out of a job.)

Defining the View for the List of Monsters

Now turn to the `V_HEADER_GRID` view, which is the view on which users will find themselves when your application starts (the default view). You want two things on this screen: the selection options and the ALV grid with the results. (In normal ALV reports, the selection criteria are on a different screen, but users love having the selection fields at the top of the result list to remind them what they were actually searching for.)

Navigate to the `V_HEADER_GRID` view, and click the `LAYOUT` tab. Your cursor is on the `ROOTUIELEMENTCONTAINER` once again; right-click, and create two elements, one after the other. This time, they will both be of the type `ViewContainerUIElement`, which means that other UI elements will be living inside them. Name them with the element type at the start, followed by what the container is going to contain (i.e., `VCUIE_SELECT_OPTIONS` and `VCUIE_MONSTER_HEADER_TABLE`).

Note

In WDA, there are lots of options for how the system will automatically arrange the various elements on the screen. There is not space in this book to go into all the various options and what they mean, but they are all documented in great detail throughout the Internet. For example, visit <http://sapuniversity.eu/usage-and-significance-of-various-layouts-in-webdynpro-abap>.

Here on the `V_HEADER_GRID` screen, you're going to have the selections at the top with the `SELECT MONSTER` button just to the right of the selection options, and the grid showing the selected monster header records directly under those selection fields. To achieve this, select the `ROOTUIELEMENTCONTAINER` node on the `LAYOUT` tab, and change the `layout` property to `MatrixLayout`. Then, go to the monster table node and change the layout to `MatrixHeadData` (if you don't do this, then the grid appears to the right of the selections). If the field to change the layout of the monster header table layout data is not there, click the `SAVE` button, and it will magically appear.

While you're here, you're also going to add a button to display the monster record that is currently highlighted in the ALV grid of monster headers. The

procedure for creating the button is exactly the same as before; right click on the `ROOTUIELEMENTCONTAINER` node, choose `CREATE ELEMENT • BUTTON` and call it `SHOW_MONSTER`. Once the element is created you can change the attributes to create the corresponding action and add a text description and an icon. (This example uses the `DECEASED PATIENT` icon, as it has a gravestone on it. This seems appropriate for a monster.) The layout data property button for the button has to be changed to `MatrixHeadData` as well. The result is shown in Figure 12.13.

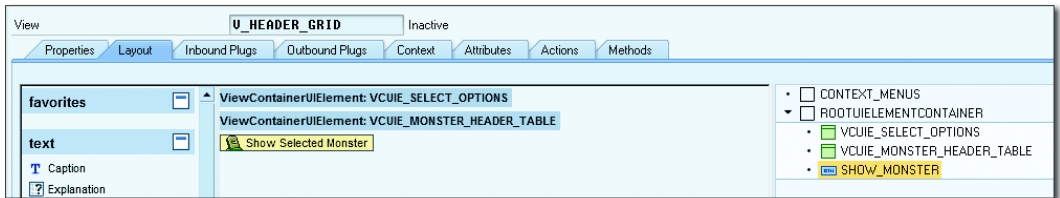


Figure 12.13 V_HEADER_GRID View to Show Table of Monster Headers

Then, move to the `CONTEXT` tab of the `V_HEADER_GRID` view. Just as you did earlier, map the data structures in the controller to the ones in your view; this time, you will be mapping both the monster header table as well as the selection options. That is, you drag both those nodes from the right-hand side of the screen to the left-hand side. You will get a confirmation prompt when you drag the table definition over to the view part of the context to ask if you actually do want to map the data from one node to the other. Once the nodes are mapped, if you click on a target node (on the left-hand side), you will see a field called `MAPPING PATH` at the bottom of the screen; this says where the source data is coming from.

Defining the View for the Selected Monster

There is a final view, `V_DISPLAY_MONSTER`, in which you'll see the header record of the selected monster and, below it, an ALV grid showing a table of that monster's items (components). The process for defining the context and view is exactly the same as before. Select the `ZWDA_MONSTER_MONITOR` root node and choose `CREATE • VIEW`. This view will be called `V_DISPLAY_MONSTER`.

Now go to the `LAYOUT` view and create two `ViewContainerUIElements`; call them `VCUIE_MONSTER_HEADER` and `VCUIE_MONSTER_ITEM_TABLE`. This time, you can keep the default flow layout that the system proposes—this means that the header fields of the selected monster will be displayed one per line, with the item table at the bottom after the list of fields.

Now navigate to the `CONTEXT` tab of the `V_DISPLAY_MONSTER` view. In order to set up automatic mapping, drag the `MONSTER_HEADER` and `MONSTER_ITEM_TABLE` nodes from the right-hand side of the screen to the left-hand side.

Lastly, the `MONSTER_HEADER` structure gets its own view, which is going to live inside the `VCUIE_MONSTER_HEADER` in the same way an ALV grid of monster items is going to live inside the `VCUIE_MONSTER_ITEM_TABLE`. Go to the root `ZWDC_MONSTER_MONITOR` node, and choose `CREATE • VIEW`. This view will be called `V_MONSTER_HEADER`. Go to the context node and drag over the `MONSTER_HEADER` structure to the left-hand side of the screen.

On the `LAYOUT` tab in `ROOTELEMENTUICONTAINER`, choose `CREATE CONTAINER FORM` as you did earlier in the chapter (see Figure 12.10). This time, when you click the `CONTEXT` button and choose the `MONSTER_HEADER` element, you get a list of every field in the header. When you press the green checkmark, these fields are copied to the graphical screen editor, just as when you were defining the `V_SELECT_OPTIONS` view at the start of Section 12.1.3.

Adding the Single Record View to the Window

Because you have just created a new view—to display a single monster—you'll need to make sure that the window knows about it; in a while you'll be defining how to jump from the `V_HEADER_GRID` view to the `V_DISPLAY_MONSTER` view. Therefore, navigate to the `WINDOW` tab of the default window (`W_MONSTER_MONITOR`), right-click the root node, and choose `EMBED VIEW` (Figure 12.14).

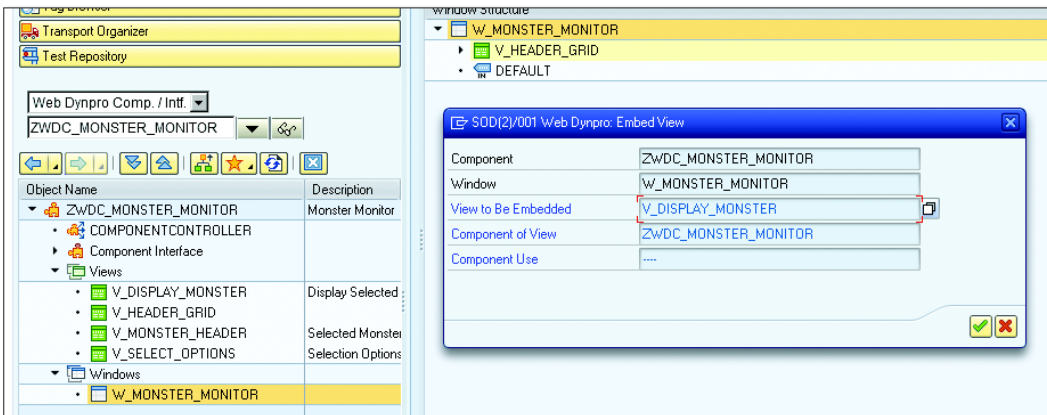


Figure 12.14 Adding a New View to the Default Window

A window shows only one view at a time—rather like when you define a main screen in traditional DYNPRO with lots of subscreens, and the user sees each subscreen one at a time by choosing different tabs. So embedding a new view is like adding a subscreen to a classical DYNPRO screen. And, just as in classic DYNPRO, you have to add some code to navigate between the subscreens to make sure the correct one is showing after a user navigation command.

Section 12.2.6 comes back to navigating between the two views. First it's time to set up the ALV lists in your application.

12.2.4 Setting Up the ALV

In the same way that you call the `REUSE_ALV_GRID` or `CL_SALV_TABLE` function modules, here you're going to make use of the WDA equivalent. To do this, navigate to your root WDA component, and go to the **USED COMPONENTS** tab. In the lower half of the screen is a table called **USED WEB DYNPRO COMPONENTS**. This is where you can reuse standard functionality—in this case, the ALV. Here, you're going to declare two ALV grids: one for the main selection view `V_HEADER_GRID` and one for the single monster record display `V_DISPLAY_MONSTER` (Figure 12.15).

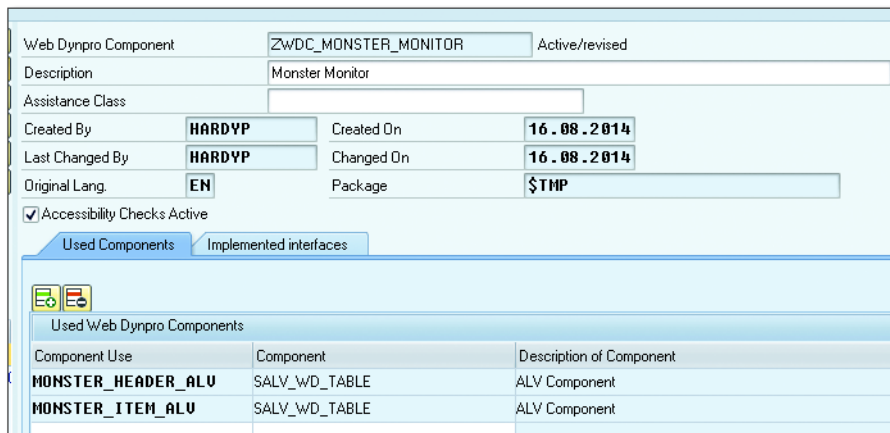


Figure 12.15 Declaring Usage of ALV

Once you've saved that, your WDA application tree expands. At the bottom of the tree, there are two new nodes under **COMPONENT USAGES** that correspond to our two ALV grids (Figure 12.16).

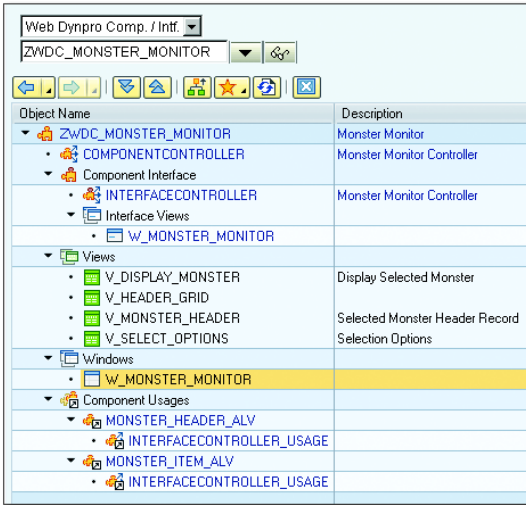


Figure 12.16 ALV Component Usages

For both grids, first go into the `INTERFACECONTROLLER_USAGE` node, and make sure you're on the `CONTEXT` tab. There, you'll see an icon labeled `CONTROLLER USAGE` that looks like a piece of paper. Everyone likes pressing buttons! Good news; you can press this one. At first you will be really puzzled: A dropdown list of components appears where it looks like the only option for the monster header ALV is the item table and vice versa. The trick is to pick the `ZWDC_MONSTER_MONITOR` component at the top of the list.

Suddenly, on the right of the screen you'll see the context (data structures) of the controller. Map the monster header table and item table respectively to the appropriate component usage nodes called `DATA` (Figure 12.17).

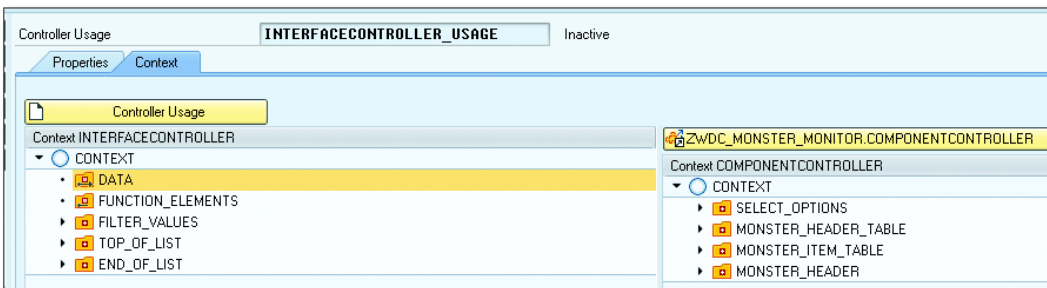


Figure 12.17 Mapping a Controller Table to an ALV Grid

12.2.5 Defining the Windows

As George Formby would sing, "You should see what I can see when I'm defining windows!" Both of our windows are going to have two containers, each with a view inside of it. On the main screen, you'll show the selection options, and then the main ALV grid. On the single monster record display, you'll show the header record, and then a grid of monster items. You set up the two containers when creating the `V_HEADER_GRID` and `V_DISPLAY_MONSTER` views earlier, but they're empty at the moment.

Navigate to the `W_MONSTER_MONITOR` window, choose the `WINDOW` tab, choose the `SELECT_OPTIONS` node of the `V_HEADER_GRID`, right-click, and choose `EMBED VIEW`.

Figure 12.18 shows the popup box that appears when you want to embed a view. All the fields are grayed out, so you have to choose the view you want via the `F4` help. This gives you a list of all the views you have defined thus far and a load of entries that are a result of the two ALV components you declared earlier. Just to be wild and daring, choose the `SELECT_OPTIONS` view for the `SELECT_OPTIONS` container.

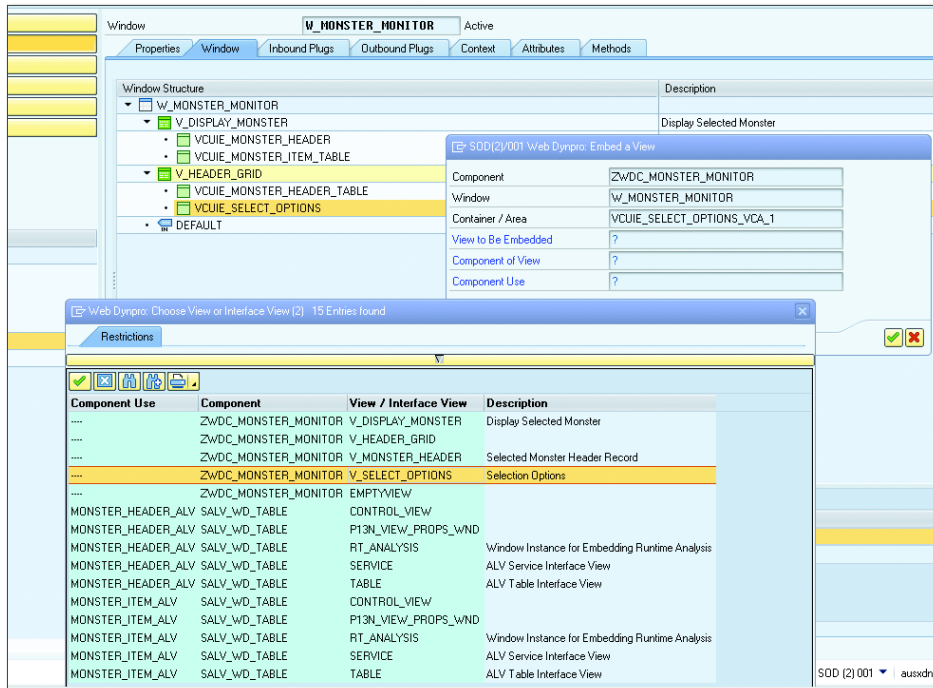


Figure 12.18 Embedding a View inside a Window

Then, simply repeat this procedure three times, adding the `MONSTER_HEADER` view to its related container and the table entries (e.g., the row that reads `MONSTER_HEADER_ALV / SALV_WD_TABLE / TABLE`) from the `[F4]` list of the two ALV grids into their respective containers. You'll end up with the screen shown in Figure 12.19.

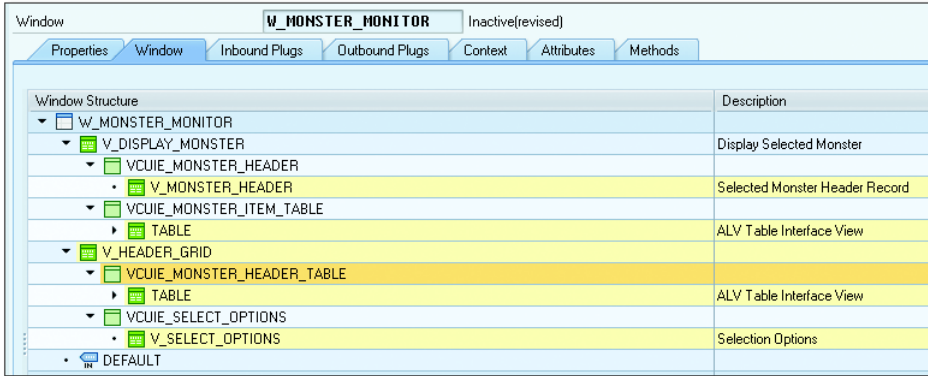


Figure 12.19 Completed Window Definition

12.2.6 Navigating between Views Inside the Window

What you want in life is that, when the user chooses a line on the monster header ALV grid and presses the `SHOW SELECTED MONSTER` button, he is taken to a new screen that shows the selected monster header record and an ALV grid of monster items. The view definition is all sewn up, which just leaves defining the navigation from one view to another.

Only one view can be shown at any given time inside a window, which is why you embedded some smaller views to leave just two possible views: `V_HEADER_GRID` (for the overview) and `V_DISPLAY_MONSTER` (for single record display). To navigate between these views, use an outbound plug to leave one view, and a corresponding inbound plug to go into the desired view (rather than using `CALL SCREEN XXX` as you would in a classic DYNPRO).

Go the view you want the user to be able to navigate away from (`V_HEADER_GRID`), and then move to the `OUTBOUND PLUG` tab (Figure 12.20). Add an entry called `OP_TO_DIS_MONSTER`; `OP` stands for “outbound plug,” and the rest of the name reflects the fact that when the navigation is invoked it will take the user to the `DISPLAY_MONSTER` view.

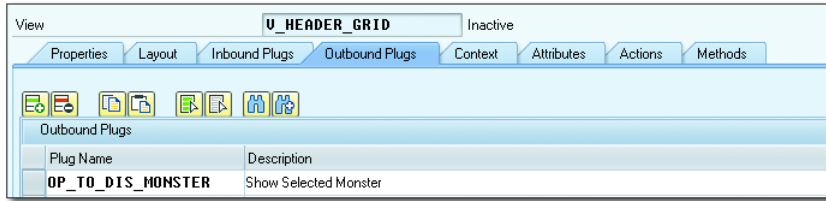


Figure 12.20 Creating an Outbound Plug

Once you've defined an outbound plug, you can enter data in the lower half of the OUTBOUND PLUG tab (Figure 12.21), in the section with IMPORTING PARAMETER FROM at the top. You almost always want to send data between views so that you can have as many parameters as you want, rather like events in OO programming. This is an outbound parameter from the sending view's perspective, but the WDA framework has chosen to call it an "importing" parameter on the grounds that it will be imported into other views. You could send the monster number and have the system look up the monster header record, but as you already have all the fields, that seems rather wasteful—so send the whole record.

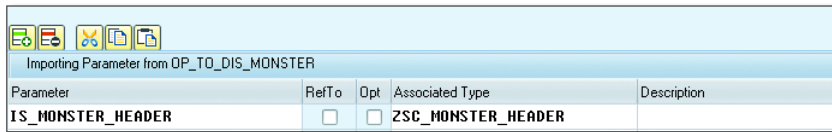


Figure 12.21 Creating an Outbound Plug Parameter

As we all know, what goes up, must come down. Now you have an outbound plug looking for a home, so you need to define a corresponding inbound plug in the target view. Simply navigate to the target view (`V_DISPLAY_MONSTER`) and select the INBOUND PLUG tab (Figure 12.22).

The difference between the outbound and inbound plugs is that when you add an entry in the INBOUND PLUG tab called `IP_FROM_MONSTER_TABLE`, an event handler method called `HANDLEIP_FROM_HEADER_TABLE` is created for you (you'll be coding this method in Section 12.2.8).

Now, create the linkage between the two plugs by going to the WINDOW tab of the `W_MONSTER_MONITOR` window, selecting the outbound plug leaving the `V_HEADER_GRID`, right-clicking, and choosing `CREATE NAVIGATION LINK`.

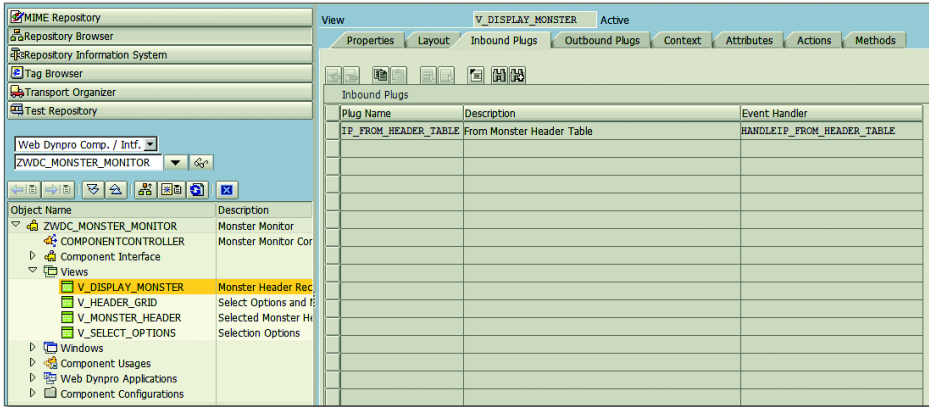


Figure 12.22 Inbound Plug to the DISPLAY_MONSTER View

You'll see the pop-up shown in Figure 12.23. From this pop-up, you can use **F4** to select possible target views. Select the `V_DISPLAY_MONSTER` view; because it only has one inbound plug, that plug is selected automatically, and the two plugs are now chained together. An icon that looks like a chain appears below the outbound plug in the tree structure.

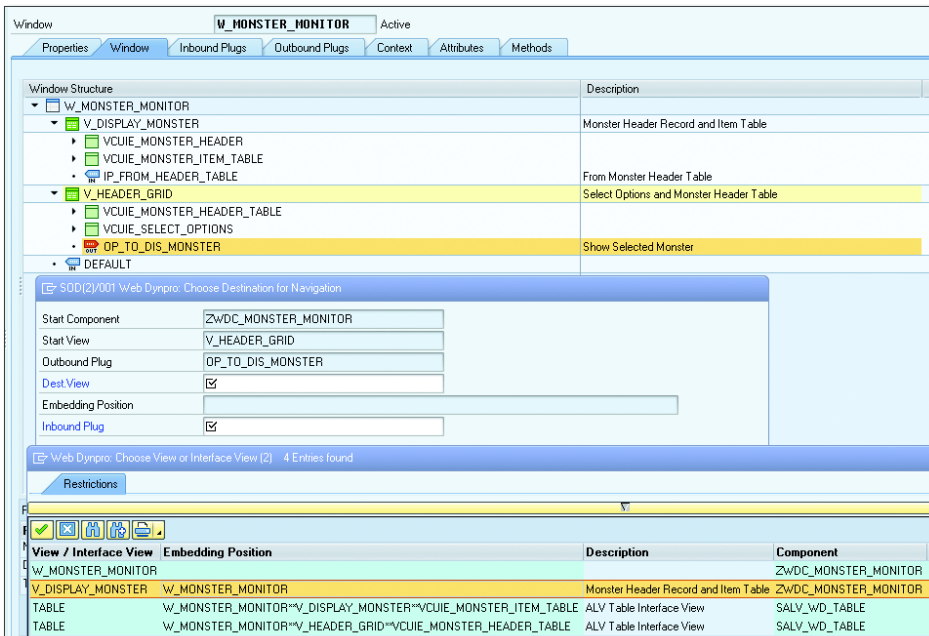


Figure 12.23 Linking Inbound and Outbound Plugs

12.2.7 Enabling the Application to Be Called

When you are creating an executable program or a module pool program, you also create a transaction code to call it. WDA is different, however; because the application is going to run in a web browser, it needs a URL. To start the process of creating such a URL, you need a so-called *application*. Go to the root component, right-click, and choose CREATE • WEB DYNPRO APPLICATION.

Figure 12.24 does not look too much different than creating a transaction code for a module pool program: you choose a program (component) and the initial screen (view), and there you go.

The screenshot shows the 'Web Dynpro Application Wizard' dialog box. At the top, the 'Application' field contains 'ZWDC_MONSTER_MONITOR' and a 'New' button. Below this are two tabs: 'Properties' (selected) and 'Parameters'. The 'Properties' tab contains several fields: 'Description' (with a yellow highlight and a red border) containing 'Monster Monitor', 'Component' containing 'ZWDC_MONSTER_MONITOR', 'Interface View' containing 'W_MONSTER_MONITOR', and 'Plug Name' containing 'DEFAULT'. Below these fields is a 'Help Links' button. The next section is 'Authorization Check' with two radio buttons: 'Check for Application' (selected) and 'Check for Application and Application Configuration'. The 'Handling of Messages' section has two radio buttons: 'Show Message Component on Demand' (selected) and 'Always Display Message Component'. The 'Administrative Data' section contains fields for 'Created By' (DEVELOPER), 'Created on' (09.01.2015), 'Last Changed By', 'Changed On', 'Package', 'Language', and 'URL' (http://abapci.dummy.nodomain:50000/sap/bc/webdynpro/sap/zwdc_monster_...).

Figure 12.24 Creating the Application (Transaction Code)

This is all well and good, but without any ABAP code your application is not going to do all that much. Move on to that task with all due haste!

12.2.8 Coding

In almost everything you'll ever read on WDA, there is talk about how "purists" insist that there is no code in the view and then go on to justify why there should

be. I don't mind admitting that I'm a purist, and we're about to start doing some coding in the view here. I have no problem at all with the view having code, as long as that code relates solely to view-specific tasks. The view's job is to send any user input somewhere else for processing, to get the results, and then to show those results on the screen, making them look as nice as possible. If the code in the view is restricted to those tasks, then you can have 20 billion lines of code if you want to, and my delicate sensibilities will not be offended.

So what does this coding entail? First, you'll learn how to make sure that the controller knows about the model so that the view can delegate non-view-type tasks. Then you'll learn about the coding needed in the view in order to bring up the list of monsters once the user has made some selections. Finally, you'll learn about the coding needed once the user selects a specific monster in order to bring up the view showing the details of that monster.

Linking the Controller to the Model

As there is going to be some coding inside the views, you will see that whenever any non-view-related tasks are to be performed, they will be delegated to the controller, which will often delegate that task to the model. Thus, you need the controller to know about the model. Go to the component controller ATTRIBUTE tab, and declare `MO_MONSTER_MODEL` as `TYPE REF TO ZCL_MONSTER_MODEL`.

Then, in the `WDDOINIT` method of the controller, add the following statement:

```
CREATE OBJECT wd_this->mo_monster_model.
```

Selecting Monster Records

You should understand each block of code that needs to be created in the order in which the user interacts with the application. On the initial screen, there is a blank set of selection options and a blank grid. The user says he wants a list of all monsters named Fred and presses the `FIND MONSTERS` button.

When you defined that button, you created an action called `FIND_MONSTERS`. If you navigate to the `METHODS` tab of the `V_SELECT_OPTIONS` view, then you can see that there is an automatically generated method called `ONACTIONFIND_MONSTERS` for that action sitting there. If you double-click that method, then you'll enter a screen that looks a bit like Transaction SE24 (but which is subtly different), from which you can do some coding.

First, you want to read the selection values the user has entered on the screen. The easiest way to do this is to use the Web Dynpro Code Wizard (**CTRL** + **F7**) or the icon that looks like a magic wand; Figure 12.25) to automatically generate the code. However, this generated code is only the first step; you'll need to manually fiddle with that code until you like the look of it. This often requires changing the generated code quite dramatically; this is certainly going to be the case for the following examples, because you need to make clear what is going on. (It's usually at this point when traditional ABAP programmers start screaming and never stop.) In particular, two major changes should be made:

- ▶ The generated comments talk a lot about *lead selection*. This term means that, if there is more than one node or UI element, the retrieval methods get the currently selected node or element. The revised code should remove all references to lead selection from the comments and replace them with comments that describe what is actually happening.
- ▶ The generated code contains the phrase `@TODO`, which means that you have to code something yourself (e.g., error handling). Again, you should replace such sections with actual code.

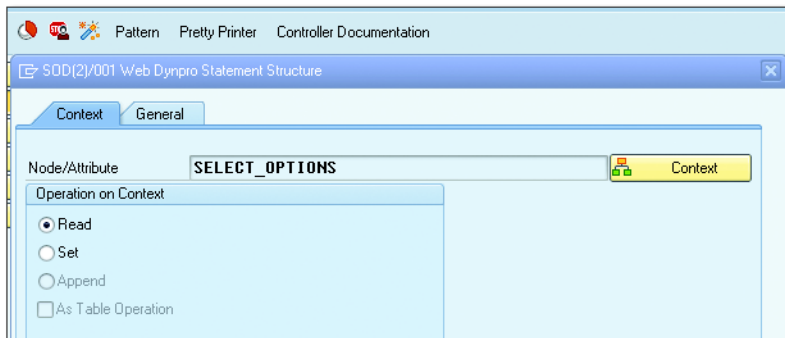


Figure 12.25 WDA Code Wizard

The generated code, with the previously noted changes included, is shown in Listing 12.1. As you can see, there are four main sections in the code:

1. The view gets the values the user has entered. This takes three steps (which is what scares and confuses people when they first see WDA code): You have to get the node in order to get the element in order to get the actual values. It is easy enough to grasp that the node structure is like a directory or the data

model, where VBAP is a child of VBAK, but the idea that the elements of the node and the contents of those elements are two different things starts to make people sweat. An understanding of basic OO concepts is needed: The node is like a class, the elements are instances of that class, and the contents of the element are the attributes of that instance.

2. Once the selection options have been retrieved and stored in the `LS_SELECT_OPTIONS` structure, the program changes the structure into a table for the purposes of sending that table of selection options to the controller.
3. The view then tells the controller what values the user has chosen and asks what values it should therefore put in the ALV grid table.
4. Once the call to the controller method has finished, the view has the data it needs to populate the ALV grid stored inside `LT_MONSTER_HEADERS` and it binds that data table to the node that houses the ALV grid.

```
METHOD onactionfind_monsters .
* First, find the node (screen area) where the selection options
* live. Ask the context for this node
DATA: lo_node_select_options TYPE REF TO if_wd_context_node.

lo_node_select_options =
wd_context->get_child_node( name = wd_this->wdctx_select_options ).

CHECK lo_node_select_options IS NOT INITIAL.

* Next step is to get the "element" associated with that node
* There might be more than one element per node. In this case
* there is only one
DATA lo_element_select_options TYPE REF TO if_wd_context_element.

lo_element_select_options = lo_node_select_options->get_element( ).

CHECK lo_element_select_options IS NOT INITIAL.

* Only now can you get the selection values in a way you can
* work with them
DATA ls_select_options TYPE wd_this->element_select_options.

lo_element_select_options->get_static_attributes(
IMPORTING
static_attributes = ls_select_options ).

* Now you have the data. Send it to the
* controller
DATA: lt_selections          TYPE /bobf/t_frw_query_selparam,
      lt_monster_headers    TYPE ztt_monster_header.
```

```

FIELD-SYMBOLS: <ls_selections> LIKE LINE OF lt_selections.

* Package up selections for sending to controller
IF ls_select_options-monster_number NE ''.
APPEND INITIAL LINE TO lt_selections ASSIGNING <ls_selections>.
  <ls_selections>-attribute_name = 'MONSTER_NUMBER'.
  <ls_selections>-option          = 'EQ'.
  <ls_selections>-sign           = 'I'.
  <ls_selections>-low = ls_select_options-monster_number.
ENDIF.

IF ls_select_options-name NE ''.
APPEND INITIAL LINE TO lt_selections ASSIGNING <ls_selections>.
  <ls_selections>-attribute_name = 'NAME'.
  <ls_selections>-option        = 'EQ'.
  <ls_selections>-sign         = 'I'.
  <ls_selections>-low = ls_select_options-name.
ENDIF.

* Ask the friendly controller to get the data you want.
wd_comp_controller->retrieve_headers_by_attribute(
  EXPORTING
    it_selections      = lt_selections
  IMPORTING
    et_monster_headers = lt_monster_headers ).

* Now, send the table data to the screen
DATA: node_monster_header_table TYPE REF TO if_wd_context_node.

* Set a link to the on-screen variable that holds the data
node_monster_header_table =
wd_context->get_child_node(
  name = if_v_select_options=>wdctx_monster_header_table ).

* Send the internal table to the on-screen variable
node_monster_header_table->bind_table( lt_monster_headers ).

ENDMETHOD. "On Action Find Monsters

```

Listing 12.1 Reacting to FIND_MONSTERS Action

You will notice in Listing 12.1 that the view handled three of the four tasks mentioned previously on its own, but when it came to getting the data it delegated the task to the controller. You could have done a database `SELECT` right there inside the view, even though that is “forbidden” by SAP. However, a purist would say that you should not execute any business logic on the surface of the application (the view) but instead send that request on a journey down the call stack to the

center of the Earth. In fact, the controller is going to pass the request straight on to the model, which is going to pass it on yet again (Figure 12.26).

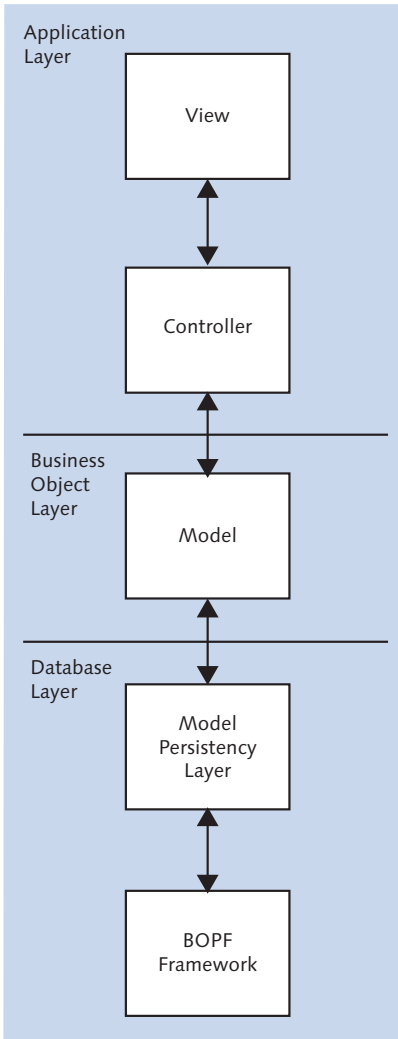


Figure 12.26 Inception-Style Data Request Flow

If you have ever seen the film *Inception*, then you will recall that it was all about going inside a dream inside a dream inside a dream. Near the end of the film, the lead female character wakes up from the deepest dream, opens her eyes in the

next dream, wakes up from that, opens her eyes in the next dream, wakes up, and repeats the process until she wakes up and opens her eyes back in reality. The ALV grid data in our program must feel a bit like that. The request rushes down from the view, through the controller, and into the model, which instantly asks its database layer to handle this request, which in return delegates that task to, in this example, the BOPF framework. Regardless of how the data is retrieved, once it's there it then starts its journey back up toward the view.

If You Are Not Using BOPF (But You Should Be Using BOPF!)

Even though this book uses the BOPF framework, it's more than possible in real life that you may be reusing an existing model class that has some other sort of data retrieval mechanism, the most obvious being direct database reads using `SELECT` statements. If this is the case, the only difference is that the box labeled `BOPF Framework` gets replaced by a box saying `SELECT STATEMENTS OR SHARED MEMORY OR READ FROM EXTERNAL SYSTEM` or whatever. Due to the highly encapsulated design approach used in this chapter, nothing else needs to change.

An alternative to the somewhat convoluted process of passing a request through several layers is to have one line of code inside the view (the `SELECT` statement). At this point, anyone not used to the OO concept of abstracting everything will think I've gone fruit loops, barking at the moon, free energy crazy. What do you get from this seemingly ludicrous, overcomplicated level of abstraction? It's all about making the program antifragile so that you can change one part without breaking the others. Namely, it has the following advantages:

- ▶ The database layer can decide to use something else than the BOPF, such as shared memory, plain SQL reads, or something exotic, like calls to an external system.
- ▶ At the other end of the scale, you can add a lot of new views that all show the same data in different ways, and because they all farm off the data request to the other layers they're all automatically synchronized rather than having the lookup logic in each view.
- ▶ In the middle, the model is framework independent and can be used by other frameworks, like ALV grid-based programs or "old-fashioned" DYNPRO programs.

In each case, due to the separation of concerns, you can only break one part at a time. The parts of the application that are not adjacent to the box you have

changed will be unaware of what you have changed and thus not dependent on it in any way (and so they cannot break).

The end result of all this is that in Listing 12.1 you saw that the view was getting the data by calling a method of its controller—so you need to go to the `COMPONENTCONTROLLER` node and create such a method, with the exact same parameters as the corresponding method in `ZCL_MONSTER_MODEL`. Then add the coding as shown. The view only knows about the controller and no other layers.

In Listing 12.2, you see the listing of that method in the controller, which is not very exciting; all it does is make the same request of a method in the model with the exact same name and signature.

```
METHOD retrieve_headers_by_attribute .

    wd_this->mo_monster_model->retrieve_headers_by_attribute(
        EXPORTING
            it_selections      = it_selections
        IMPORTING
            et_monster_headers = et_monster_headers ).

ENDMETHOD.
```

Listing 12.2 Controller Method to Retrieve Monster Header Data

This leads to the stage in which the user can enter a monster name such as Fred and get an ALV grid of monster header records. Now, you need to allow the user to drill down into a single monster record.

Navigating to the Single Record View

Earlier, when you created a button to show the selected monster, you also created a corresponding action. A method is automatically generated for you to handle that action, so go to the `V_HEADER_GRID` view and fill in the code for the `ONACTION-SHOW_MONSTER` method. The code for this is shown in Listing 12.3.

```
METHOD onactionshow_monster .
* Find the node where the monster header table lives
    DATA lo_node_monster_header_table TYPE REF TO if_wd_context_node.

    lo_node_monster_header_table = wd_context->get_child_node(
        name = wd_this->wdctx_monster_header_table ).

* Get all the selected elements (lines). In fact there is only one
* but as in some applications the user can select more than one
```

```

* the returning parameter is a table
DATA lt_selected_monster_elements TYPE wdr_context_element_set.

lt_selected_monster_elements =
lo_node_monster_header_table->get_selected_elements( ).

CHECK lo_node_monster_header_table IS NOT INITIAL.

* As this is a table with one line, you have to read that line
DATA lo_element_monsterheadertable
TYPE REF TO if_wd_context_element.

READ TABLE lt_selected_monster_elements INTO
lo_element_monsterheadertable INDEX 1.

CHECK lo_element_monsterheadertable IS NOT INITIAL.

* Now, you can read the contents of the element into a
* structure
DATA: ls_monster_header TYPE zsc_monster_header.

lo_element_monsterheadertable->get_static_attributes(
IMPORTING
static_attributes = ls_monster_header ).

* Fire the "plug" to take you to the display monster view
* and pass that view the selected monster header record
wd_this->fire_op_to_dis_monster_plg(
is_monster_header = ls_monster_header ).

ENDMETHOD. "On Action Show Monster

```

Listing 12.3 Coding the Action Method to Display a Single Monster

In Listing 12.3, you will already see a pattern beginning to emerge in the way that WDA view methods are coded. First, you get the node you want, then the element inside that node that you want, and finally you extract the content from that element into the type of structure you normally deal with in programs. This time, the process is slightly different, in that you're reading the selected line in a table. The system knows what line has been selected—which is called the lead selection—so you don't have to jump through hoops to work out what the user has done. Moreover, this bit of code can be generated for you by the Code Wizard.

At the end of the `onactionshow_monster` method, you ask the controller to fire the outbound plug, which means that the application navigates to the screen defined when you created the outbound and inbound plugs and linked them together earlier in the chapter (Section 12.2.6).

You don't have to code the firing method (FIRE_OP_TO_DIS_MONSTER_PLG) yourself (the framework takes care of it), but you do have to code the method in the target view that responds to the inbound plug. In V_DISPLAY_MONSTER, note that a method (HANDLEIP_FROM_HEADER_TABLE) was generated when you created the inbound plug; you now have to fill that method.

The code for this is shown in Listing 12.4, where (as will come as no surprise by now) you'll start by looking for nodes and elements. This time, you'll find the header record, which you'll then fill up with the structure that was passed as a parameter into this method. The second half of the listing is all about getting the node where the ALV grid for the item table lives, asking the controller for the item data, and filling up that grid.

```
METHOD handleip_from_header_table .

* Get the node where the header record lives
DATA lo_node_monster_header TYPE REF TO if_wd_context_node.

lo_node_monster_header =
wd_context->get_child_node( name = wd_this->wdctx_monster_header ).

CHECK lo_node_monster_header IS NOT INITIAL.

* Get the element that lives in that node
DATA lo_element_monster_header TYPE REF TO if_wd_context_element.

lo_element_monster_header =
lo_node_monster_header->get_element( ).

CHECK lo_element_monster_header IS NOT INITIAL.

* Fill a structure with the values that were passed in
DATA ls_monster_header TYPE wd_this->element_monster_header.

ls_monster_header = is_monster_header.

* Pass those values into the node element
lo_element_monster_header->set_static_attributes(
    static_attributes = ls_monster_header ).

* You're done with the header record. Time for the items
* Get the node you want to fill
DATA lo_node_monster_item_table TYPE REF TO if_wd_context_node.

lo_node_monster_item_table =
wd_context->get_child_node( name = wd_this->wdctx_monster_item_table ).
```



```

CHECK lo_node_monster_item_table IS NOT INITIAL.

* Ask the controller for the item data for this monster
DATA:lt_monster_item_table TYPE ztt_monster_items,
      lf_read_failed       TYPE abap_bool.

wd_comp_controller->retrieve_monster(
  EXPORTING
    id_monster_number = is_monster_header-monster_number
  IMPORTING
    ef_read_failed    = lf_read_failed
    et_monster_items  = lt_monster_item_table ).

CHECK lf_read_failed = abap_false.

* Fill up the node
lo_node_monster_item_table->bind_table(
  new_items          = lt_monster_item_table
  set_initial_elements = abap_true ).

ENDMETHOD. "Handle IP from Header Table

```

Listing 12.4 Handling an Inbound Plug

You will also have to manually add the importing parameter `IS_MONSTER_HEADER` of type `ZSC_MONSTER_HEADER` to the method. WDA methods are not quite like SE24; the signature and the code are defined in the same place.

As before, the controller passes the request straight on to the model, as shown in Listing 12.5. (Again you will have to add this method to the `COMPONENTCONTROLLER` with the same parameters as the correspondingly named method in `ZCL_MONSTER_MODEL`).

```

METHOD retrieve_monster .

  TRY.
    wd_this->mo_monster_model->retrieve_monster_record(
      EXPORTING
        id_monster_number = id_monster_number
      IMPORTING
        es_monster_header = es_monster_header
        et_monster_items  = et_monster_items ).

    CATCH zcx_monster_exceptions.
      ef_read_failed = abap_true.
  ENDTRY.

ENDMETHOD.

```

Listing 12.5 Controller Method that Passes a Data Request to the Model

At this point, you can navigate to the `Web Dynpro Application` node, which is somewhat like a transaction code, and click the `TEST` icon (or press `[F8]`). Up pops a browser in which you'll see the screen flow shown at the start of this section (Figure 12.27).

The screenshot shows a web application interface for 'Monster Monitor'. At the top, there are input fields for 'Monster Number' and 'Monster Name' (containing 'FRED'), and a 'Find Monsters' button. Below this is a 'View' dropdown set to 'Standard View' and buttons for 'Print Version' and 'Export'. The main part of the screen is a table with 12 rows and 12 columns. The first row is highlighted in yellow. The table contains the following data:

Monster Number	Monster Name	Monster Sanity %age	Monster Color	Monster Strength	Monster Hat Size	Monster Age in Days	Number of Monster Heads	Monster Count	Text	Text
4	FRED	3	GREEN	REALLY STRONG	0	1	0	1		
5	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
6	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
7	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
8	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
9	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
1	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
10	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
11	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT
12	FRED	3	GREEN	REALLY STRONG	5	1	1	1	BONKERS	NORMAL HAT

At the bottom left of the table area, there is a 'Show Selected Monster' button.

Figure 12.27 Testing the WDA Monster Monitor

Obviously, this application needs lots more bells and whistles. This example deliberately omits things that did not add value to this discussion.

12.3 Using Floorplan Manager to Modify Existing WDA Components

In the last section, you performed two common developer tasks: building an overview screen with a list of objects and displaying a single business object record (in this case, the business object was a monster). If, over the course of his career, a programmer creates applications to display such data for 10 different custom business objects, those objects will most likely come out looking 10 different ways (if only subtly) and most probably will not look like standard SAP screens that show such data.

Internally, SAP has historically been even more inconsistent; one of the most common complaints about SAP is its UI, specifically how the transactions in the different modules have different looks and feels. This criticism is not unwarranted: posting a journal in SAP ERP Financials does not look much like creating a sales order, and creating a sales order looks nothing like creating a purchase order. One can only presume that the development teams in SAP ERP Financials,

Sales and Distribution, and Materials Management worked utterly independently of each other.

FPM aims to reduce inconsistency by standardizing the way data is presented in applications. The FPM framework also caters for various ways to improve existing applications from a developer, administrator (business analyst), or end user point of view. (Naturally, end users can change a lot less than developers, but it is good that they can change *anything* if it makes their lives easier.)

FPM is organized around (you'll be shocked to hear) the concept of floorplans. The idea of a *floorplan* is best described by using an architectural analogy. If your job in life was to design houses rather than starting from scratch each time, then the logical approach would be for you to develop several templates in which the kitchen and toilet are in the same place each time. This is why McDonalds (or some pub chains in the UK) is so successful: no matter what country you are in, when you enter such a store it looks the same, and everything is pretty much where you expect it to be. Conversely, if you have spent any appreciable time in hotels, you may have noticed that the shower controls are never the same—so the first part of every morning is spent thinking, how in the world does this one work?

A floorplan in an SAP context is an extension of the same principal. When users see a new application for the first time, you don't want them to have to struggle to work out how to do basic things (e.g., expand an item in a list). If all applications look the same and the menu options are in the same place each time, then life becomes a lot easier.

At the time of writing, SAP provides four floorplans to choose from:

- ▶ OVP (Overview Floorplan): An overview screen to show a big list of records and let the user choose one.
- ▶ OIF (Object Instance Floorplan): A business object display screen to show a single object (like a monster).
- ▶ GAF (Guided Activity Floorplan): A wizard-type application in which you have X number of steps along the top, with the current step in the process highlighted. (This is somewhat like the idea SAP once came out with called "xApps," in which you could purchase a product to build composite applications, which also had steps along the top.)

- ▶ QAF (Quick Activity Floorplan): A simple application to enter one or two pieces of data only (the sort of thing you might see on your mobile phone).

In this section, you'll examine how you might build the Monster Monitor that you built using WDA by using FPM instead. You'll then see how the BOPF monster business object integrates straight into the FM framework via the FBI (we love acronyms—or, rather, WLA).

Creating Empty FPM Applications Using the FPM Workbench

In addition to modifying existing WDA components to build an FPM application around them, it is also possible to create an empty FPM application using the FPM Workbench and then insert existing WDA components. In both cases, the WDA components have the functionality, and the FPM overlays this functionality with a standard look and feel; however, the process steps involved are different. Because most developers will be modifying existing WDA components, instead of creating FPM applications from scratch, this is the approach that is the focus of this chapter.

For more information about creating empty FPM applications, you can take a look at a book by James Wood on this subject (see the "Recommended Reading" box at the end of the chapter).

12.3.1 Creating an Application Using Floorplan Manager

In order to create an FPM application, the first thing you have to do is—you got it—open the tool. How you open FPM varies by system. In version 7.02, there are no specific transaction codes related to FPM; you have to create a WDA application that implements a specific interface, and only then can you call up FPM. In version 7.31 and above, you can open FPM by calling Transaction FPM_WB (FPM Workbench). The initial screen of Transaction FPM_WB is shown in Figure 12.28.

The screen in Figure 12.28 displays an enormous amount of options, some of which are deliberately designed for interfacing with other SAP technologies. For example, in the lower-left-hand corner you get extra options for creating applications based on the SAP CRM BOL business object framework and also for OData services, which are used for communicating SAP data models to the outside world (e.g., SAPUI5).

The main pieces used for creating FPM applications are—remember, we all love acronyms—UIBBs (Unreasonably Intolerant Bionic Bullfrogs). Amazing as it may seem, some people say that the acronym stands for "User Interface Building Blocks."

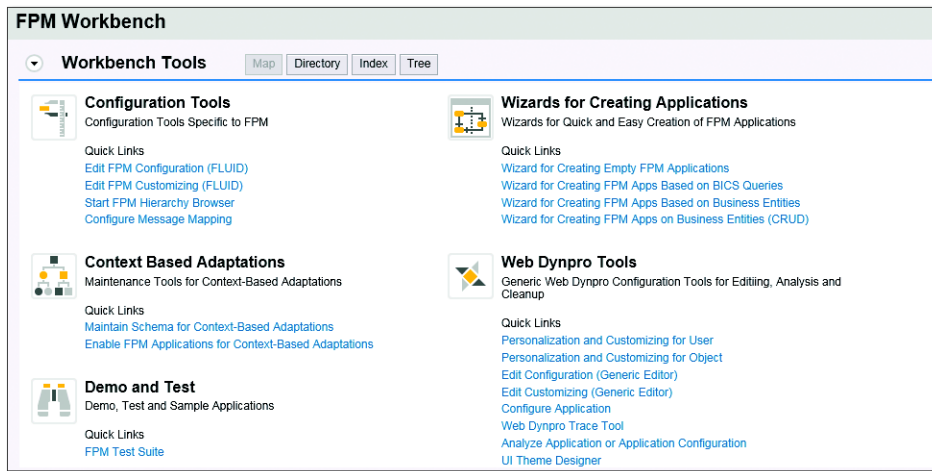


Figure 12.28 FPM Workbench

What might a UIBB be? There is no way you could guess from the name. SAP Help defines a UIBB in the following manner:

From an FPM perspective, UIBBs are the interface views (Web Dynpro ABAP windows) that are provided by the external application and not by FPM itself.

The building blocks are the windows in your existing WDA application, and, in order to use such windows in an FPM application, you need to make these visible to the FPM framework.

UIBBs vs. GUIBBs

This section focuses on *freestyle* UIBBs. That sounds a bit like a swimming contest, and reflects the fact that when you are exposing existing WDA applications (like the Monster Monitor), they can look like anything at all.

There are also GUIBBs (Generic User Interface Building Blocks) for common UI elements like lists, trees, and searches. These GUIBBs take the form of WDA components such as `FPM_LIST_UIBB`, which contain no logic and perform view-like tasks in a consistent manner. Section 12.3.2 takes a quick look at the integration of BOPF with FPM, which involves creating an FPM application from scratch using GUIBBs.

As it turns out, you can enable your existing Monster Monitor WDA component for use with FPM, turning the windows into UIBBs like turning a pumpkin into a horse and carriage at (almost) the drop of a hat by going to the root

node, navigating to the IMPLEMENTED INTERFACES tab, and adding the entry IF_FPM_UI_BUILDING_BLOCK (Figure 12.29).

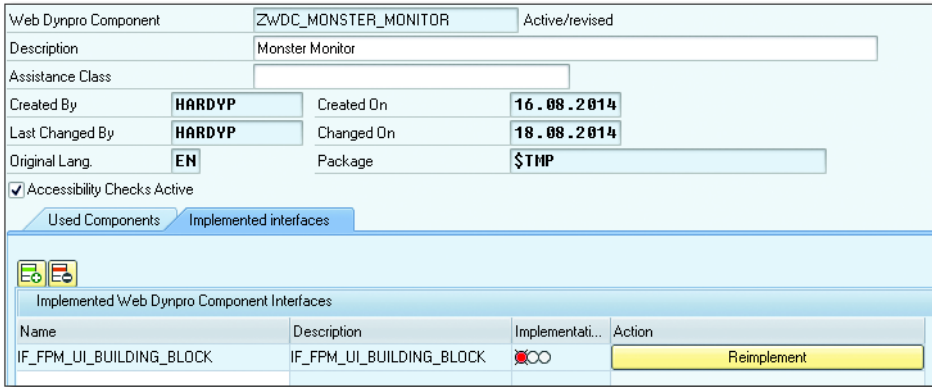


Figure 12.29 Turning a WDA Component into a UIBB

Without such an interface, there is no way FPM can talk to normal WDA components. You will no doubt notice the red light; when you implement an interface, all the inherited methods are naturally empty, and you have to fill them up. If you press the REIMPLEMENT button, then the light changes to green. That was easy! What happened when you pressed that button was that the controller in your monster application gained some new methods (Figure 12.30).

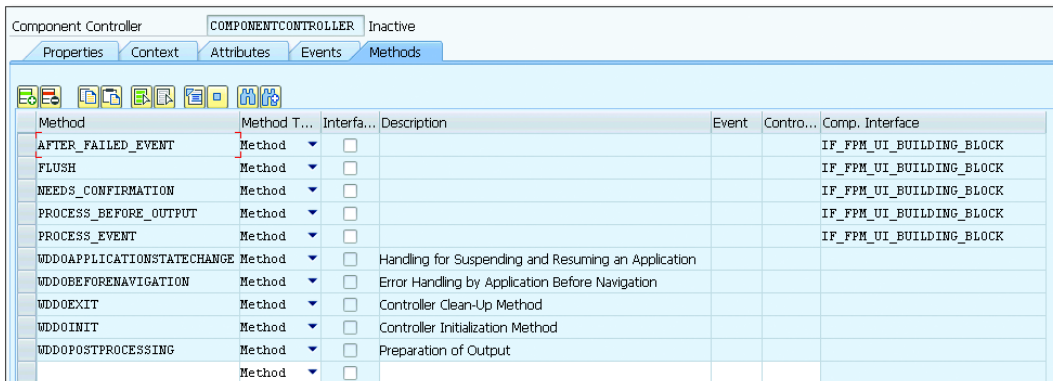


Figure 12.30 New UIBB-Related Methods

To liken those methods to traditional DYNPRO processing, they're all rather similar to what you manually code to handle what happens during PAI when a user

presses a button. You will see that the list of methods in Figure 12.30 is shown in alphabetical order. The following list is a rundown of what each new method means in the order in which they get processed at runtime:

- ▶ FLUSH
This goes first! The flush is the data transfer method from the UIBB to the backend. The controller is the backend in this case—it needs to know from the view layer (the UIBB) what the user just did.
- ▶ NEEDS_CONFIRMATION
You can pop up a box, such as one that says, ARE YOU SURE YOU WANT TO LAUNCH THAT NUCLEAR MISSILE?
- ▶ PROCESS_EVENT
Fire the missile.
- ▶ AFTER_FAILED_EVENT
This is for error handling, and it's nice to see this built into the framework. Here you can deal with what happens if something goes wrong with the missile launch.
- ▶ PROCESS_BEFORE_OUTPUT
Process before output is a term we all know and love. In this case, before you show the user the screen again, you want to update the display (e.g., add a success or error message, or maybe remove the missile that was just fired from the list of possible missiles that can be fired).

The important thing is that once you've added this interface to a WDA component it then becomes visible to FPM.

The next step is creating another WDA application for our WDA component. Go to the root node, and choose CREATE • WEB DYNPRO APPLICATION. The screen shown in Figure 12.31 appears.

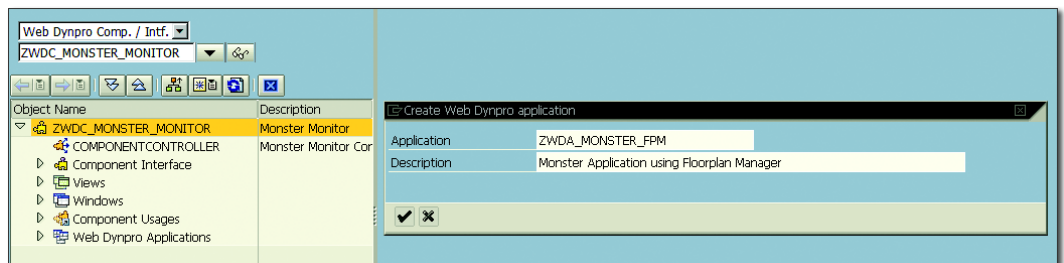


Figure 12.31 Creating a New WDA Application for FPM: Part 1

Enter a technical name and a description, and click the green checkmark. A much more interesting box now appears (Figure 12.32).

The screenshot shows the configuration screen for a new WDA application. The application name is `ZWDA_MONSTER_FPM`. The description is `Monster Application Using Floorplan Manager`. The component is `FPM_OIF_COMPONENT` and the interface view is `FPM_WINDOW`. The plug name is `DEFAULT`. The interface includes sections for Authorization Check, Handling of Messages, and Administrative Data.

Field	Value
Application	ZWDA_MONSTER_FPM
Description	Monster Application Using Floorplan Manager
Component	FPM_OIF_COMPONENT
Interface View	FPM_WINDOW
Plug Name	DEFAULT
Authorization Check	<input checked="" type="radio"/> Check for Application
Handling of Messages	<input checked="" type="radio"/> Show Message Component on Demand
Created By	DEVELOPER
Created on	09.01.2015
Last Changed By	
Changed On	
Package	
Language	
URL	http://abapci.dummy.nodomain:50000/sap/bc/webdynpro/sap/zwda_monster_...

Figure 12.32 Creating a New WDA Application for FPM: Part 2

The screen will default to the `ZWDA` component, but this time instead of the custom WDA component change the entry to a standard SAP floorplan, like the values shown in Figure 12.32. The component is `FPM_OIF_COMPONENT` (for overview) and the interface view (start screen) is `FPM_WINDOW`.

Save your entries. You'll be asked first if this is an administration service (it is not) and then if this is to be a package or a local object (in this example, a local object). Next, you need to create an application configuration. To do so, first navigate to the newly created `ZWDA_MONSTERS_FPM` WDA application. The trick here is that you have to change the component in the SE80 search box to `FPM_OIF_COMPONENT`. Once you've found your newly created WDA application, right-click it, and choose `CREATE/CHANGE CONFIGURATION`. A web browser opens (Figure 12.33).



Figure 12.33 Editor for WDA Application Configuration

Create a name for the CONFIGURATION ID, and click NEW. Once again, you're asked if this is a local object and, if not, what package to place it in; take the same option as the last time you were asked. The end result looks like Figure 12.34.

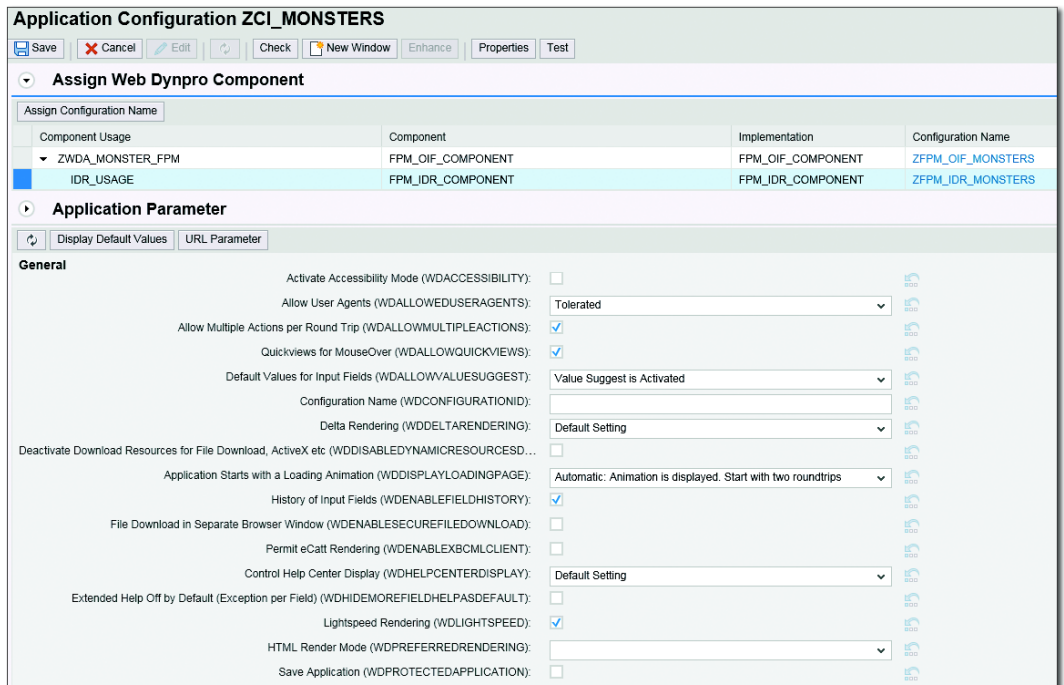


Figure 12.34 Monsters Application Configuration

In the screen shown in Figure 12.34, you'll notice some blank entries at the right of the screen in the column titled CONFIGURATION. Blank squares are crying out to be filled, so highlight each line and click the ASSIGN CONFIGURATION NAME button. Names could be, for example, ZFPM_OIF_MONSTERS and ZFPM_IDR_MONSTERS. ("OIF" means overview screen, and "IDR" stands for "identification region.") Click SAVE, and you'll see a bunch of warnings that these configurations do not

exist. That is because you haven't set them up, and you cannot do so until you have pressed the SAVE button: a catch-22 situation.

You will notice at the bottom of the screen shown in Figure 12.34 that there are approximately 10 million options that control the look and feel of the application (e.g., whether the fields remember input history). You may have a standard SAP WDA transaction you are enhancing using FPM, and the standard transaction may not remember field input history; now, you can enable this feature.

After clicking SAVE, the links on the right-hand side of the screen shown in Figure 12.34 have come to life. Click on the first link. You are prompted to enter a description and then are asked for a package once again. A screen appears (Figure 12.35) that is a sort of cut down version of the screen you saw earlier when looking at the FPM Workbench (refer back to Figure 12.30).

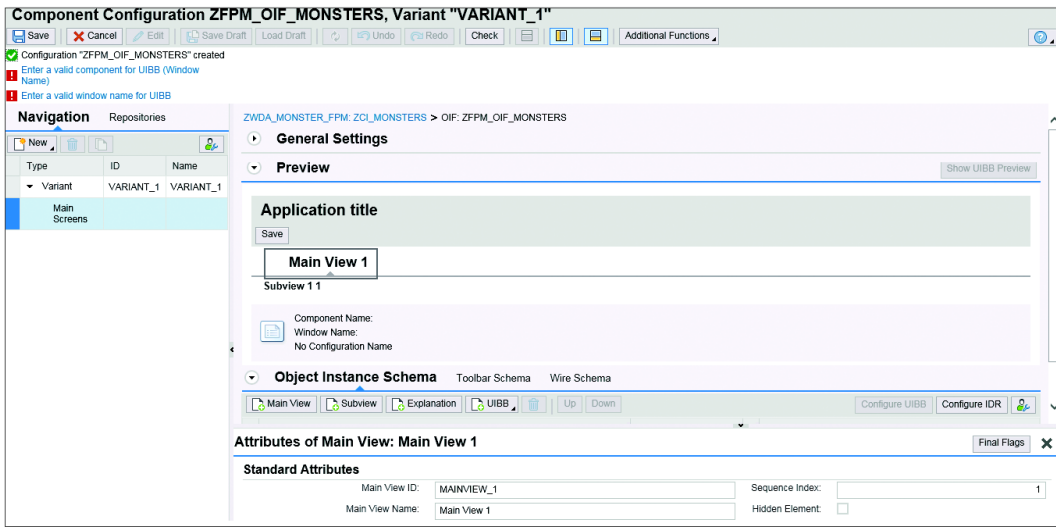


Figure 12.35 Configuring the Monster Overview: Part 1

Click the box that starts with COMPONENT NAME: below the word SUBVIEW in the main part of the screen shown in Figure 12.35. You'll see fields for COMPONENT and WINDOW NAME at the bottom of the screen. In these fields, enter the details of the WDA Monster Monitor component (Figure 12.36).

Click SAVE, and then click the CONFIGURE IDR button on the right of the screen. You are prompted to enter a description, and then once again you are asked for a package.

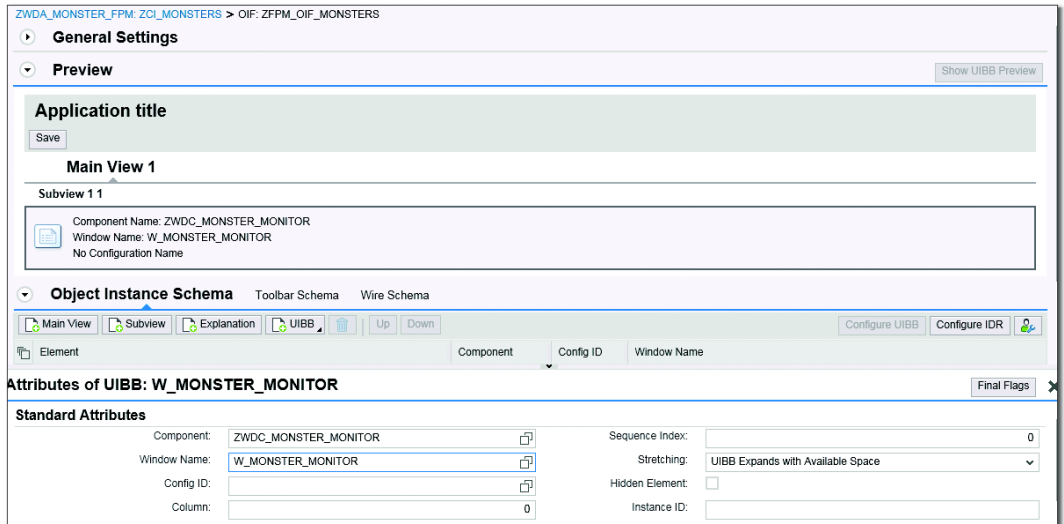


Figure 12.36 Configuring the Monster Overview: Part 2

The screen that pops up now is a bit more basic than before (Figure 12.37). You only have to fill in the title of the application (which will appear at the top of the screen) and the tooltip (which will appear when you hover your cursor over the title).

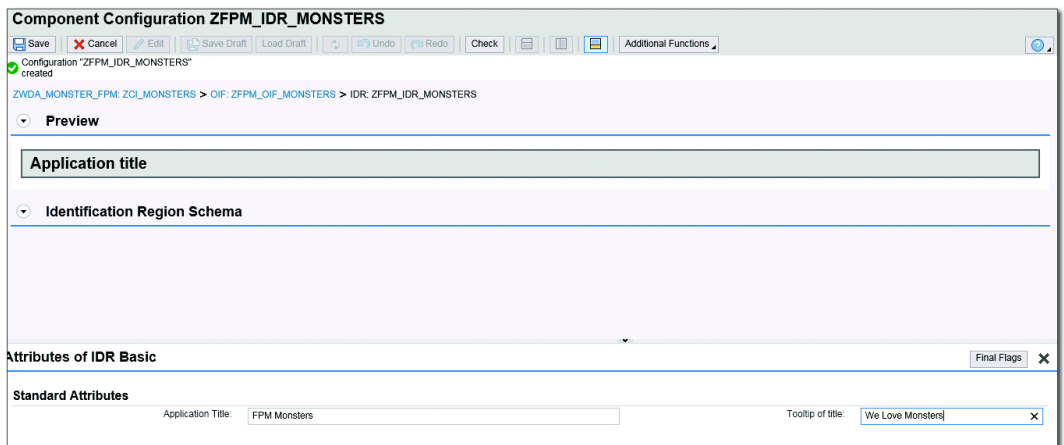


Figure 12.37 Configuring the Monster IDR

Click **SAVE**, and you're done. Because all of the setup was done in a web browser back in the SAP GUI, you'll still have SE80 open and the ZWDA_MONSTERS_FPM WDA application selected. Expand this application, and select the ZCI_MONSTERS node under APPLICATION CONFIGURATION. Right-click this node, and select **TEST**. The screen shown in Figure 12.38 appears.

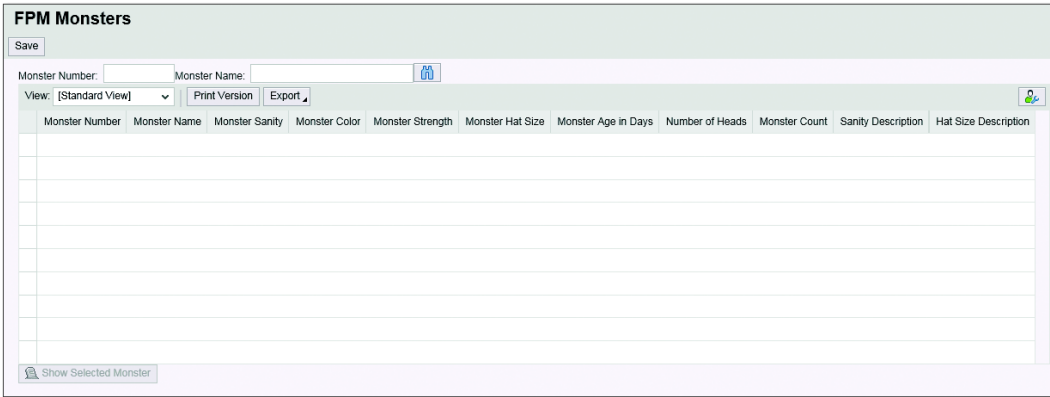


Figure 12.38 FPM Monsters Screen

Because you changed an existing WDA application into something that uses FPM, naturally the end result looks pretty much like the original WDA application. The only differences are the title at the top, a tooltip, and some new options in the top-right-hand corner that have to do with personalization.

In this example, you created the WDA application first and then bolted it into the FPM framework. The original WDA application is untouched. This means that you can use FPM to enhance existing WDA applications, taking components from them and arranging them however you want. Thus, if you use any standard WDA screens, then you can use this technique to make modification-free enhancements.

12.3.2 Integrating BOPF with Floorplan Manager

One of the great things about FPM is its easy integration with BOPF. Now is the time to examine the monster model framework you created using the BOPF framework from Chapter 8 and to see how it integrates into FPM. The general idea is that first you create your model using the BOPF and then, once you have

that working, you can plug the BOPF object straight into FPM and have a working application to show an object instance using the floorplan provided for that purpose. (In fact, it's best to keep the real model class separate from the BOPF framework, but that's not important here.)

You've created a BOPF monster object (which uses a separate model class), and you can create an FPM application based upon this. There are two ways to do this: the easy way, in which the results don't look so good and you cannot add your own bells and whistles, or the almost as easy way (called an *FBI view*; FBI stands for "Floorplan BOPF Integration"), which still doesn't look all that good, but you can add as many extra things as you feel like until the cows come home. Naturally, the second method is best, but for the sake of completeness this chapter discusses both.

Earlier in this chapter it was mentioned that reusing existing WDA components involves adding an interface to them to make them visible to the FPM, turning those components into freestyle UIBBs. With the integration of BOPF and the FPM, you would be creating an FPM application from scratch using the other type of UIBBs: namely, GUIBBs, which are reusable UI elements designed to give disparate applications the same look and feel.

GUIBBs have no business logic as per the MVC pattern; a GUIBB gets its information from a feeder class. Here the BOPF framework serves as the model, holding the business logic, and at the other end of the scale the GUIBBs do view-like tasks such as arranging data in a grid in an attractive and consistent manner. The configuration of FPM acts as the controller, linking the business logic and the view tasks together.

In the floorplan for displaying a single business object record, there are two screens: an initial screen in which you enter the monster number (or, on the off chance that you are not dealing with monsters, a sales order number or some other such number) and a screen that shows the header record and a table of item details. (The definition of the monster business object and the methods to get the header and item data were set up in Chapter 8.)

Start with the initial screen. Earlier in this chapter, you looked at the `COMPONENT CONFIGURATION` screen for defining FPM applications. In that screen, you can choose `NEW • INITIAL SCREEN`, and the resulting screen will look like Figure 12.39.

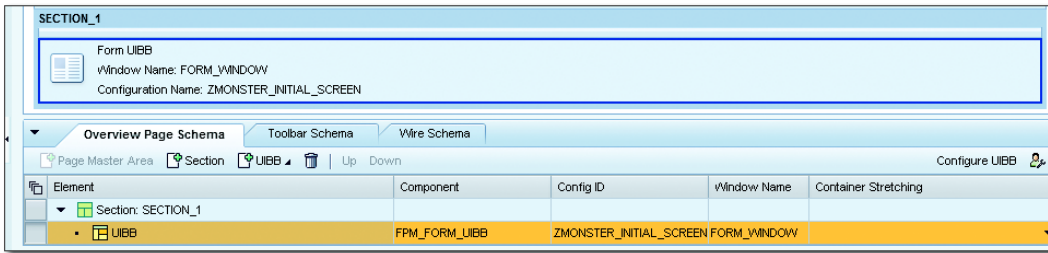


Figure 12.39 Configuring the FBI Initial Screen

There is a CONFIGURE UIBB button in the top-right-hand corner. If you click it, then the system asks for a bottom feeder class. For BOPF purposes, the class is always going to be the same (`/BOFU/CL_FBI_GUIBB_ALTKEY_FDR`), and the name of the BOPF object type is going to be an input parameter to that class. (You're never going to want the users to enter the silly 32-character BOPF key; they will always enter a realistic value, like a monster number.) After entering the class, click EDIT PARAMETERS and enter the Z monster object as the business object, ROOT as the node, and MONSTER_NUMBER as the alternate key. In the same way, when the time comes to set up the main screen, the feeder class for the header record is `/BOFU/CL_FBI_GUIBB_FORM`, and for the item table it will be `/BOFU/CL_FBI_GUIBB_LIST`.

For both the header table and the item table, when you edit the feeder parameters you'll see a screen in which the first two entries are FBI VIEW and BUSINESS OBJECT. These are the two options mentioned at the start of the section; putting in a business object is easiest, but if you do that, then you are limited to the fields defined in the BOPF definition of the object. Usually, when building an application you want to put all sorts of extra stuff on the screen as well, so the FBI view is the way to go.

To create an FBI view, you have to create a new component configuration using the WDA component `/BOFU/FBI_VIEW`. (Remember: In an FPM application there is an application configuration, which relates to the application as a whole, and component configurations, which relate to parts of the application. These two sorts of configurations were what we were creating in Section 12.3.1.)

After you specify the FBI view, you can choose data from other BOPF objects when defining the UI. More to the point, you are allowed to stick in anything else you feel like, which is the normal course of events. Even more importantly, you can hide data from the BOPF that you may not actually want in this application. In short, it's normally not a choice at all; FBI views are the way to go.

Again, a more detailed discussion of this topic is outside of the scope of the book, but do check out the “Recommended Reading” box at the end of this chapter for more resources about how to link the BOPF with FPM.

FPM Outlook

FPM is evolving rapidly, and with each new SAP NetWeaver release quite a lot of new functionality is added. In particular, SAP seems to be quite committed to integrating some of its disparate technologies via FPM. As an example, in the “Recommended Reading” box at the end of the chapter you will see a reference to a how-to guide that talks about creating an FPM application using CDS views, which (as you’ll see in Chapter 15) are an SAP HANA–related technology. In addition, as you’ve seen already, there is native integration with BOPF and also the SAP CRM BOL equivalent. FPM is an SAP technology that is very much alive and kicking and constantly evolving.

12.4 Summary

This chapter introduced you to WDA technology, which (although created back in 2005) is still not very widely used by many programmers. For almost ten years, the position of SAP has been to make WDA the go-to technology for user interfaces. In recent times, however, a new competitor has strolled onto the scene: SAPUI5. You’ll learn more about this in the next chapter.

Recommended Reading

- ▶ *Web Dynpro ABAP: The Comprehensive Guide* (Wood and Parvaze, SAP PRESS, 2013)
- ▶ Select Options in Web Dynpro ABAP: <http://scn.sap.com/docs/DOC-52017> (Sankar Gelivi)
- ▶ *Floorplan Manager: The Comprehensive Guide* (Wood, Bowdark Press, 2013)
- ▶ Getting Started with FPM Integration with BOPF: www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/e0864a37-45f1-3010-ae81-c7076358a039?Quick-Link=index&overridelayout=true&58652073398668 (Matan Taranto)
- ▶ How To Create FPM Application Consuming CDS View Using ACT: <http://scn.sap.com/docs/DOC-53596> (Gopalakrishnan Ramachandran)
- ▶ *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin, Prentice Hall, 2008)
- ▶ *Head First Design Patterns* (Freeman et al., O'Reilly Media, 2004)

*What if you could find brand new worlds right here on Earth where anything's possible? Same planet—different dimension.
I've found the gateway!
—Sliders (television show)*

13 SAPUI5

I once read an article from the head of SAP usability in which he described an experiment: SAP had created a new application and had some guinea pig users in one room being trained on this new product and some SAP staff in another room watching them via camera and counting how many times the users smiled while using the software. Well, forgive me for being cynical, but if the SAP people were using their fingers to count, then I don't think they would need both hands. Or even one hand. Maybe the users would smile when they were let out of the room.

Due to this sort of reaction, SAP is determined to shed its image of having the most hideous, user-unfriendly user interface (UI) ever. After many false starts, SAPUI5 could be the breakthrough SAP's looking for. This chapter talks about the basics of SAPUI5, including its enabler, which is the SAP Gateway add-on for SAP ERP and which comes bundled with your existing license. SAP has finally taken this problem seriously and (in my opinion, at least) nailed it.

Before getting into the details, though, there are two misconceptions to clear up here. The first centers on the crazy fact that there are two utterly different programming languages that both start with the word "Java." There are historical reasons for this, but Java resembles JavaScript in the same way a duck resembles duct tape. JavaScript is not Java, but it's still a fully functional programming language. (That's sort of a pun, because JavaScript is a functional language, whereas Java and ABAP are imperative languages.)

The second misconception is much more common and seems to be shared by developers and management alike. When confronted with the fact that SAPUI5 (or anything else for that matter) is written using JavaScript, people throw up

their hands and say, “Well, we cannot use our ABAP programmers, then, because they just don’t have the right skillset.” Do you know, amazing as it may seem, that there was a time when I didn’t know ABAP? When I was 14, I programmed in BASIC, and then at university Pascal was the go. When I started work (as an accountant, as opposed to as a programmer), on my very first day I discovered how to record macros and then use the programming language in the spreadsheet (which would become Visual Basic) to automate boring tasks. Then, when SAP came along it was time to start writing programs in ABAP, and then the jump from procedural programming to OO programming was rather like learning a whole new language. (Also there are bits of Java all over SAP products now, such as the graphical mappings in SAP PI, and I had to adjust those from time to time.) The point is, nowhere in all of this did I throw up my hands and scream “I only know BASIC! There’s no point in going on in life. I might as well throw myself off the nearest bridge.” You are a *programmer*. This is more a way of thinking and problem solving than knowledge of the syntax of a particular programming language. If you need to learn something new to solve a problem, learn something new! And, you should learn JavaScript.

The killer argument is that if you can program on both sides of the fence—the ABAP side that produces the data and the JavaScript side that consumes and formats the data—then that puts you in an incredibly small minority and that has to be a good thing for your career. At the end of this chapter is a link to a wonderful article from SAP aimed at introducing ABAP programmers to JavaScript. There are many differences (e.g., variables that change their data types when you give them a different value), but it’s not the end of the world. It may be worth taking a look at that so that the next section (which shows some JavaScript code) makes a bit more sense.

In this chapter, Section 13.1 looks at the technical architecture of how you can expose your SAP applications to appear as SAPUI5 applications in web browsers and mobile devices. It also describes SAP Gateway, which is an SAP add-on that can be used for this purpose, among many others. Section 13.2 discusses the prerequisites for creating an SAPUI5 application: what you will need installed in your SAP system and what you will need installed on the local machine on which you will be developing.

The bulk of the chapter will then describe the process of creating the ever-popular Monster Monitor as an SAPUI5 application. This process has several facets:

1. Use SAP Gateway to expose the data model to the outside world, a process that involves both configuration and ABAP coding (Section 13.3).
2. Create the SAPUI5 part of the application (the view and the controller), which needs to be done on your local machine (the frontend; Section 13.4).
3. At this point, you'll have a working SAPUI5 application, and you'll see how easy it is to enhance such an application with all sorts of fancy UI elements (Section 13.5).
4. Next, you'll look at how you can import your finished SAPUI5 application into the ABAP system so that it can hang out with all its non-UI application friends (Section 13.6).

Finally, the chapter ends by clearing up some confusion between the terms "SAPUI5" and "SAP Fiori," which seem to mean different things to different people (Section 13.7).

13.1 Architecture

When writing programs within SAP, one of the traditional ways to avoid performance problems was to minimize the amount of data transferred between the database and the server, because the effort involved in moving from one place to another created bottleneck. The SAP HANA platform avoids this entirely by having all the data in memory.

In the same way, the curse of web-based applications (at least from the end user's point of view) is the roundtrip that occurs every time the user presses a button or enters a piece of data. The information about what the user has done is sent back to the backend system, which then interrogates the database or performs some business logic and sends back instructions to the frontend as to what to display next. This time, the bottleneck is the time spent going to and from the browser to the server and back again, which is why you often spend so much time in web applications looking at a whirling circle (hourglasses are clearly old-fashioned).

Naturally, then, in an application that is going to run in a browser (be it on a desktop or a mobile device) you only want to do a roundtrip when you have a question only the backend can answer (e.g., some information from the database or some decision that has to be made based on complicated business logic, like whether to approve a loan or how many snails to put inside a monster).

In this section, you'll see how a UI application deals with the frontend processing on the device running the application and the backend processing inside the SAP system.

13.1.1 Frontend: What SAPUI5 Is

I could easily say "SAPUI5 is a JavaScript library," but would that make any sense to a traditional ABAP developer? Probably not. JavaScript is a programming language—quite different from ABAP, but a programming language nonetheless. A library is simply a collection of reusable classes people have built over time. We do the same thing with reusable Z utility classes and function modules in our ABAP systems, but most SAP programmers don't group these classes and modules into packages.

The SAPUI5 application is a JavaScript program that runs on the client side—that is, within the web browser or mobile device (i.e., on the machine that's physically right in front of you, as opposed to the SAP system that could be on the other side of the world).

In model-view-controller (MVC) terms, the view and the controller are on the frontend. The view handles anything explicitly related to what the screen looks like, and when the user does something, the controller decides if the view can handle the request by itself (e.g., shrinking an area of the screen or showing some other static screen) or if it needs to bother the model, which is lurking in the backend SAP system. The idea is that the total amount of roundtrips should therefore be reduced, because you don't have to go to the backend every single time the user does anything (as is the case with Web Dynpro, for example). This should lead to faster performance times and happier users.

13.1.2 Backend: What SAP Gateway Is

Most articles that talk about SAP Gateway show a diagram with three boxes—the SAP system, SAP Gateway, and the outside world—with arrows pointing between them. I could have included such a diagram, but I'm not sure what it would have proved. You get the point: SAP Gateway lets SAP talk to the outside world.

You are probably thinking to yourself at this point that SAP already has *millions* of ways to talk to the outside world: IDocs, remote function calls (RFCs), proxy calls

to SAP PI, direct invocation of web services, and so on. What's so special about SAP Gateway?

According to SAP, the problem with all the other integration techniques is that they are proprietary—that is, SAP-specific. In recent years, SAP has made a 180-degree turn away from such bespoke solutions in favor of open-source equivalents, which are much more widely used and not owned by anybody in particular. In the case of SAP Gateway, the protocol it uses to expose SAP data is called OData, as in, “Oh, data, you're so fine, you're so fine you blow my mind, oh data.” (Some people say it actually stands for “Open Data Protocol.”)

To go acronym crazy for a second, SAP used to hope that its enterprise system architecture (ESA) vision would catch on, but that was based on SOAP, and in recent times no one likes SOAP anymore. People prefer something called REST, and OData is based on REST. SAP decided that made the ESA thing pretty much dead in the water. If you can't beat them, join them, and so SAP came up with SAP Gateway to expose SAP data as OData.

You could say that OData is the glue that binds SAP Gateway to SAPUI5. SAPUI5 expects data to be exchanged using such a protocol, and SAP Gateway enables such an exchange to be performed from within an SAP system.

Later on in the chapter, this will all make much more sense, when you see how you actually go about exposing a model class using transactions within SAP.

13.2 Prerequisites

Before you can get up and running with SAPUI5, you have to make sure your system has everything it needs. This section discusses the prerequisites for SAPUI5.

13.2.1 Requirements in SAP

SAP Gateway started life in 2011; it's a component of the ABAP system, and it can be used in any SAP system based on SAP NetWeaver 7.0 and above. Before 7.4, you had to download and install it explicitly, but in SAP NetWeaver 7.4 and beyond it comes as standard.

Software Component	Release	Level	Highest Support Packa...	Short Description of Software Component
GW_CORE	200	0007	SAPK-20007INGW...	SAP GW CORE 200
IW_BEP	200	0007	SAPK-20007INIW...	Backend Event Provider
SAP_BS_FND	702	0012	SAPK-70212INSA...	SAP Business Suite Foundation
SAP_BW	702	0014	SAPKW70214	SAP Business Warehouse
UISAPUI5	100	0006	SAPK-10006INUI...	SAP UI5
UI_INFRA	100	0006	SAPK-10006INUI...	SAP UI INTEGRATION INFRASTRUCTURE
IW_FND	250	0007	SAPK-25007INIW...	SAP IW FND 250
IW_GIL	100	0003	SAPK-10003INIW...	Generic Interaction Layer

Figure 13.1 SAP Gateway Components

In systems earlier than 7.4, you will see some of the components shown in Figure 13.1 when taking the `SYSTEM • STATUS` option and looking at the component information—specifically, `IW_FND`, `GW_CORE`, and `IW_BEP`. In a 7.4 system, you will just see `SAP_GWFND`. Leaving meaningless names aside, what this means is that if you see those components in your system, then you will have some new transaction codes, which you can use to model your SAP data so that SAPUI5 applications and the like can use this data. You will note the phrase “and the like”; this is not just for SAPUI5. Data exposed in this format can be used by a wide variety of consumers. One example is the new incarnation of Duet, which allows SAP to talk to Microsoft Office applications.

Eagle-eyed readers will notice that in Figure 13.1 there are two components with “UI” in their names: `UISAPUI5` and `UI_INFRA`. These have nothing to do with SAP Gateway (the job of which is to send and receive information to and from the SAP system); they’re concerned with enabling frontend SAPUI5 components to be stored inside the SAP system. This is discussed in Section 13.7.

13.2.2 Requirements on Your Local Machine

In order to use SAPUI5, you have to have Eclipse installed on your local PC (refer back to Chapter 1 to see how to do this). Open Eclipse, navigate to `HELP • INSTALL NEW SOFTWARE`, and enter the following in the box at the top of the screen (Figure 13.2): “<https://tools.hana.ondemand.com/>[name of latest Eclipse version supported by SAP]”. Then click the `ADD` button. (You can find the latest Eclipse version supported by SAP by visiting <https://tools.hana.ondemand.com/>.)

It takes a little while to load, but once done you have both sides of the coin installed and are ready to start developing an SAPUI5 application.

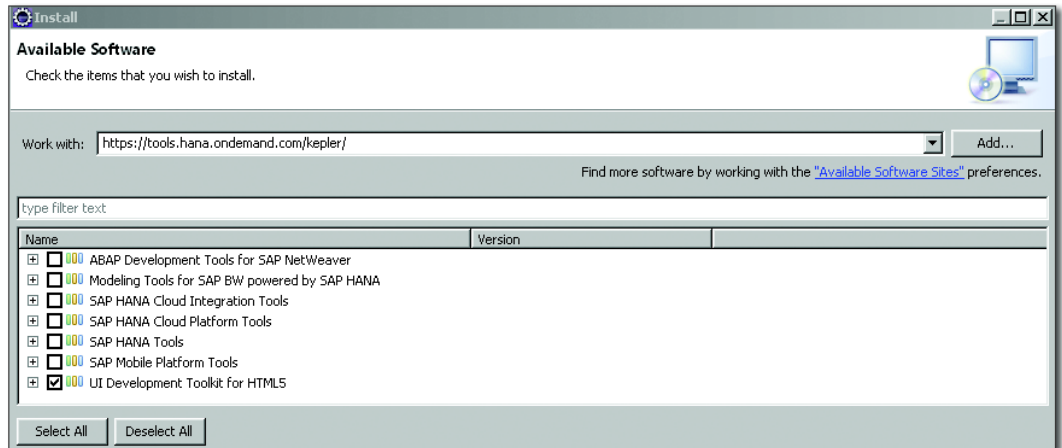


Figure 13.2 Installing SAPUI5 in Eclipse

13.3 Backend Tasks: Creating the Model Using SAP Gateway

Now, you'll find out how to take an existing business object that lives inside SAP (a monster, in this example) and enable it to be exposed for use in SAPUI5 applications.

There are two parts to this:

1. The configuration part, in which you define the data structures used in the model that represents the business object and the relationships between them.
2. The coding part, in which you implement methods that will be used to create, retrieve, update, and delete instances of your business object (monster) in the SAP database.

13.3.1 Configuration

In this section, you'll see how to use SAP Gateway to set things up such that when a URL request is made from a web browser or from a desktop or mobile device for information that resides in our model in SAP that information gets sent back. The steps are as follows:

1. Create a project, which will contain the details of your monster data model.
2. Create so-called entities, which describe the data structure of the monster header and the monster items.

3. Link together these two data structures by means of an association.
4. Create some generated classes based upon that data model and a related service.
5. Make sure that the system configuration is set up correctly so that you can proceed with adding the new service.
6. Add the new service to the list of available services.
7. Perform a basic test to make sure that the service has been added correctly and thus is able to be called by an SAPUI5 application.

Each of these steps is discussed in more detail next.

Creating a Project Using the Service Builder

When you install the SAP Gateway add-on into your SAP system, you get a new transaction code—specifically, Transaction SEGW, which is the Service Builder. Run this transaction now. (Admittedly, it doesn't look very impressive—just a big gray empty screen with a few icons in the top-left-hand corner.)

On this screen, the top nodes are called projects (which are groupings of one or more business objects). Start by choosing the piece of paper icon in the top-left-hand corner that says CREATE PROJECT (Figure 13.3).

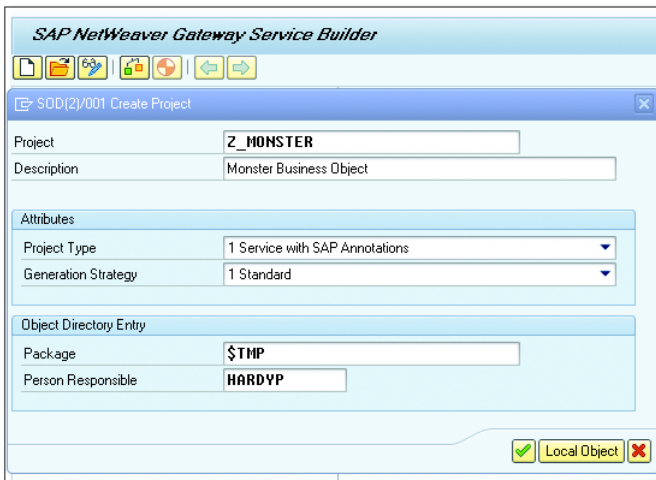


Figure 13.3 Creating an SAP Gateway Project

Now, you can get cracking!

Creating Entities for the Monster Header and Monster Items

As you know by now, your monster object has both a header structure and an item structure. You're going to declare both of these as entities, and then link them together.

As always, there are lots of ways to do this. In this case, right-click the `Data Model` node and choose `IMPORT • DDIC STRUCTURE`. While doing that, you'll no doubt notice the other options, most notably the option to use a remote-enabled function module or a BOR object. (A BOR object from Transaction SWO1 is a bona fide business object as far as I'm concerned, and the fact that this shows up in a brand-spanking-new SAP technology means that the BOR is back from the dead, despite SAP trying to unsuccessfully kill it off in 2004.)

Once you've created your monster project via the screen shown in Figure 13.3, click the green checkmark; the screen shown in Figure 13.4 appears.

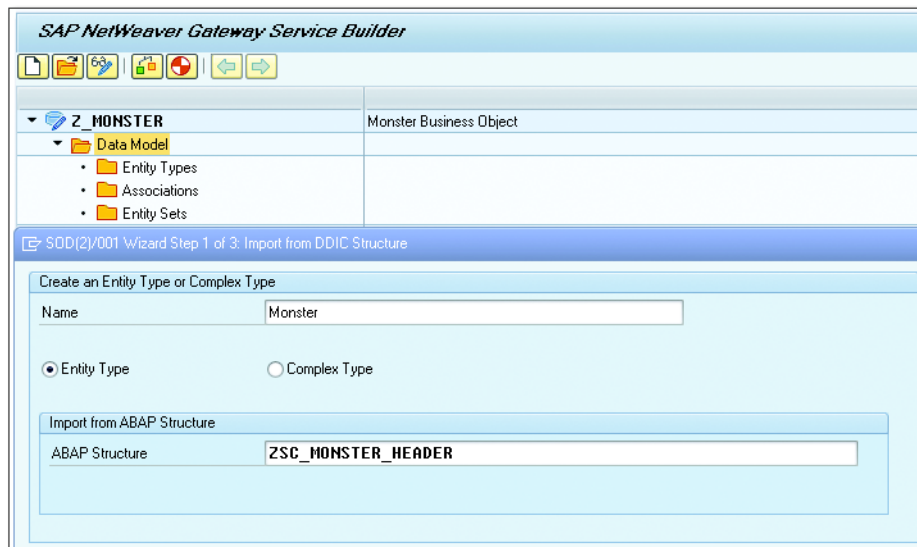


Figure 13.4 Creating a Monster Entity: 1 of 3

In Figure 13.4, you give your entity a name and pick a data structure. As always, use the BOPF combined header structure. The name here will be used in URLs, so sticking to the convention of lowercase except the first letter is the way to go. Click the green checkmark, and Figure 13.5 appears.

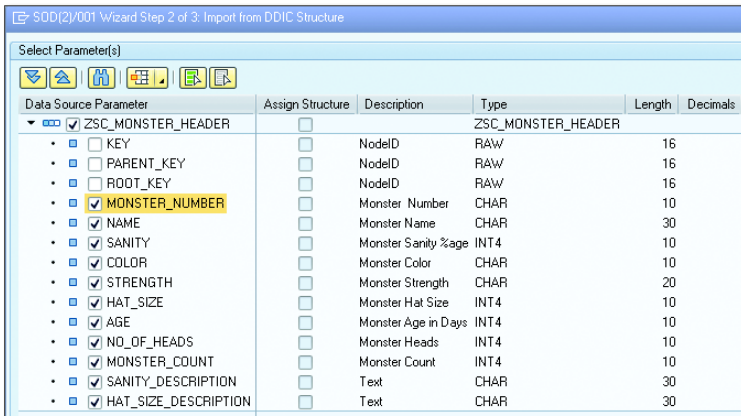


Figure 13.5 Creating a Monster Entity: 2 of 3

In Figure 13.5, you indicate what fields you want. The 16-digit UIDs are not of much interest to the outside world; they're only used internally by BOPF. The rest of the data is what you want to show people. Again, click the good old green checkmark, and the screen shown in Figure 13.6 appears.

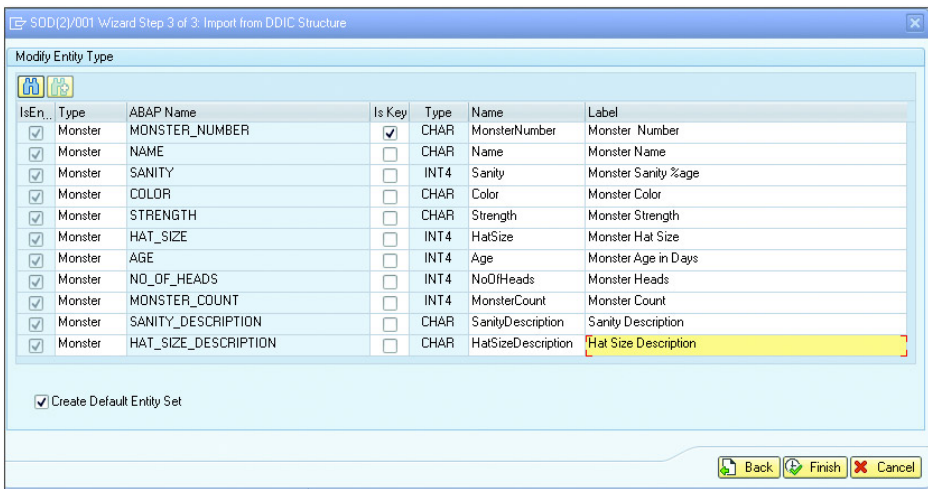


Figure 13.6 Creating a Monster Entity: 3 of 3

In Figure 13.6, you nominate what the key field is (this is a structure, so the system has no idea). You can change the text descriptions if you want; they default

from the DDIC descriptions, but if you've been naughty and used generic data elements in your structure definition, then you can change the name here to something meaningful.

When you click FINISH, your empty screen looks a lot better. Next, navigate to the Entity Sets node. An entity set is a grouping of monsters in this case, so an entity set is like a table of class instances, and an entity is one row in that table, a single instance of that class. Tell the system you want to be able to do CRUD operations in regard to monsters and also to search for monsters (Figure 13.7).

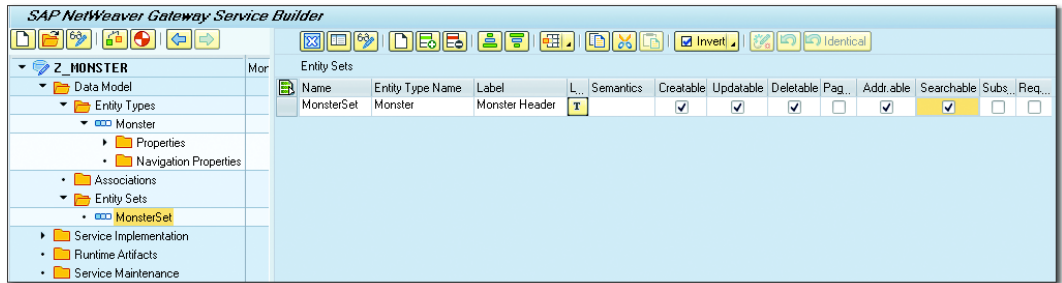


Figure 13.7 Defining CRUD Availability for a Monster Entity

Repeat the exact same sequence of steps for your monster item structure; this too will be an entity. As an aside, if you get something wrong, you'll be confronted by the usual meaningless error messages, as in "Ha, ha, ha, you've done something wrong, and I'm not going to tell you what." Let me take this opportunity again to stress the importance of presenting the users with meaningful information to help them work out what they did wrong and how to correct it. As an example of what not to do with error messages, if you give your entity the same name as one of the fields in the structure, then you get a weird message about children and parents. Clicking the question mark to get the long text on the message is no help, because it's "self-explanatory," but luckily Google will help you out.

Calming down for a second, you want to pay attention to the field on the far right side when you click on the Properties node of an entity: This field is called SEMANTICS. If a DDIC field is an email address, a geocoordinate, the start date of a holiday, or another one of many such things (the `F4` help will give you a list), then you should state that here. You may wonder why this is important. What it's all about is that SAP Gateway is communicating with the outside world, and external applications like Microsoft Outlook care very much if something is an

email address or a telephone number, and some applications care about geocoordinates. Because you have no idea what types of things may end up accessing this data, you're writing in letters of fire 1,000 miles high that something is a telephone number—so that any application that does something with telephone numbers is in no doubt about what field to look at.

Linking Header and Item via an Association

Next, you need to say that the items are related to the header, rather like defining a foreign key relationship in a DDIC table. Right-click the `Data Model` node, and choose `CREATE • ASSOCIATION`.

The screen shown in Figure 13.8 appears. Here, choose what to link together (header to items in this case), and fill in the `CARDINALITY` fields to say that one header record needs at least one item record but can have many.

Figure 13.8 Creating an Association: 1 of 3

When you're done, click the green checkmark. The screen shown in Figure 13.9 appears. Here, link the key fields together (i.e., both `Monster_Header` and `Monster_Item` have the same key field, `MonsterNumber`, and that's how you can tell which items relate to which header record).

Principal Entity	Principal Key	Dependent Entity	Dependent Property
Monster	MonsterNumber	Monster_Item	MonsterNumber

Figure 13.9 Creating an Association: 2 of 3

Green checkmark time again, and then your association is complete. When you look at your monster project again (Figure 13.10), you'll see that the system has added a few more nodes to your tree structure and chained together the header and item.

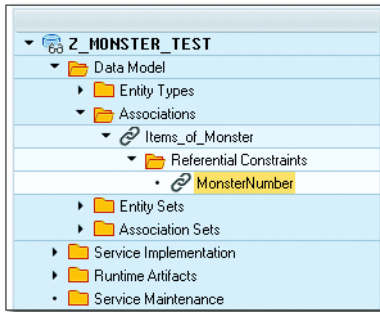


Figure 13.10 Creating an Association: 3 of 3

At this point, if you expand the node called `Service Implementation`, then you will see the CRUD operations: `Create`, `Read`, `Update`, and `Delete`. It makes me want to cry that whoever invented the OData design decided to call the read operations `GET`, thus turning CRUD into CGUD. There are two read operations: `GETENTITY` (Read), which returns either a single object or an error, and `GETENTITYSET` (Query), which returns between zero and multiple objects. However, would it have hurt the OData inventor (I think his name was Seamus O'Data) so much to go with the industry standard? Obviously, yes.

Anyway, you cannot perform any of those operations without some sort of ABAP coding, and for that you need the system to generate some ABAP classes, which will form a service.

Creating the Service and Classes for Your Monster Project

Now that you're done with defining everything, the data is in a form in which you can let the system generate classes and a related service based upon that data. Do this by clicking the red and white crash test dummy icon at the top of the SEGW screen (you can see it at the top of the screen in Figure 13.11, behind the pop-up box) with the hover text `GENERATE RUNTIME OBJECTS`. When you click this button, the pop-up box shown in Figure 13.11 appears.

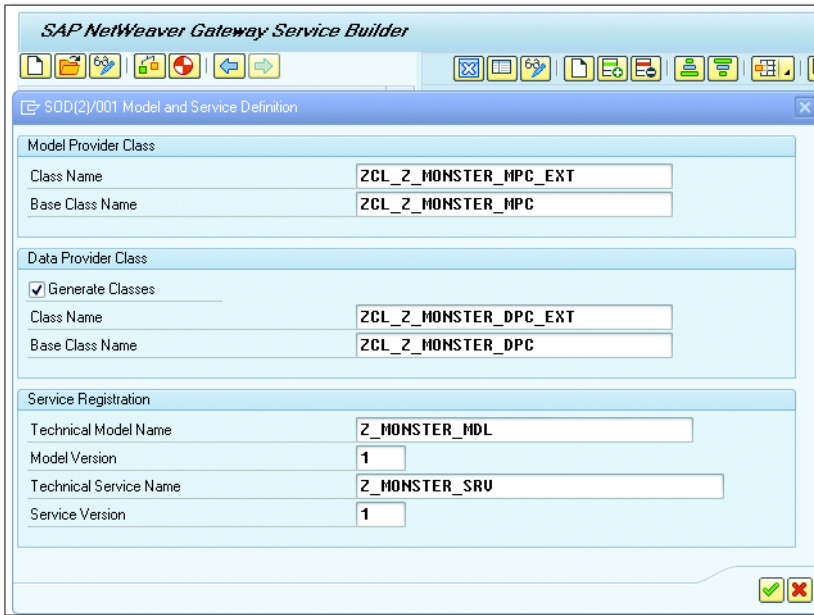


Figure 13.11 Generating Runtime Objects and Services

In Figure 13.11, you'll see the pop-up box proposing the names of some generated classes that will be created in SE24 world and a service that will be created in SICF world. The classes will do the work within the SAP system, and the service will enable the outside world to access the functionality these classes provide in a controlled manner.

When you click the ever-popular green checkmark, all these things (classes and the service) are generated, and afterwards there are a lot of green lights at the bottom of the screen to tell you that this has occurred.

You'll find out what to do with the generated classes in Section 13.3.2, which deals with coding. For the time being, you need to do some configuration to get the service into an active state.

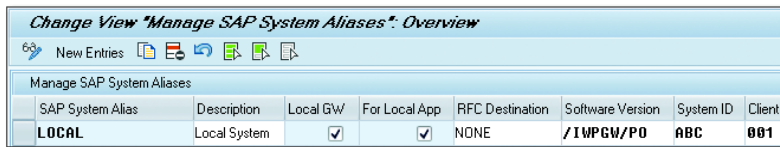
Configuring the Service: General Settings

The very first time you create an SAP Gateway service, there are some configuration steps that need to be taken, which will remain valid for all subsequent services that you create.

To start, you may have heard it said that talking to yourself is a sure sign of madness. Given that logic, then, your SAP system needs to be mad in order for any SAP Gateway services to work. You're going to set things up so that your system can indeed talk to itself if it so desires.

First, you need to set up a system alias, which is a human-friendly name for the RFC destination (SAP system) where SAP Gateway lives. In this case, `LOCAL` means that SAP Gateway is installed and running on the same SAP ERP system where the business data lives, as opposed to some separate hub system.

The system alias is set up via the IMG. You know that IMG menu paths tend to change frequently, so the path in this example won't look like the one in your system, but if you do a search for "system alias", then you should find some entries called `MANAGE SAP SYSTEM ALIASES` and from there be able to create an entry called `LOCAL` to connect to the current system (Figure 13.12).

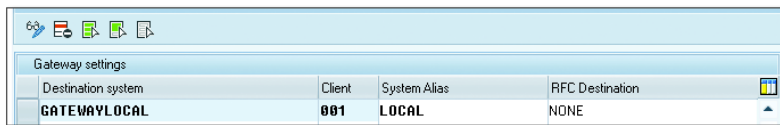


The screenshot shows the 'Manage SAP System Aliases' overview table in the SAP IMG. The table has the following columns: SAP System Alias, Description, Local GW, For Local App, RFC Destination, Software Version, System ID, and Client. A single entry is visible with the alias 'LOCAL', description 'Local System', and checked boxes for 'Local GW' and 'For Local App'. The RFC Destination is 'NONE', Software Version is '/IWPGW/PO', System ID is 'ABC', and Client is '001'.

SAP System Alias	Description	Local GW	For Local App	RFC Destination	Software Version	System ID	Client
LOCAL	Local System	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	NONE	/IWPGW/PO	ABC	001

Figure 13.12 Creating a System Alias

In Figure 13.12, on the left of the screen, select some boxes to say that your SAP Gateway system is installed locally and not on a hub (so you don't need an RFC destination to reach that hub). On the right of the screen, enter the details of the system (system ID and client number) where the SAP Gateway system resides, which is the same as the system in which the SAP data lives.



The screenshot shows the 'Gateway settings' table in the SAP IMG. The table has the following columns: Destination system, Client, System Alias, and RFC Destination. A single entry is visible with Destination system 'GATEWAYLOCAL', Client '001', System Alias 'LOCAL', and RFC Destination 'NONE'.

Destination system	Client	System Alias	RFC Destination
GATEWAYLOCAL	001	LOCAL	NONE

Figure 13.13 SAP Gateway Configuration Settings

You're halfway there; there's one more configuration setting needed to enable your SAP system to talk to itself. In the IMG, navigate to `SAP NETWEAVER • GATEWAY SERVICE ENABLEMENT • BACKEND ODATA CHANNEL • CONNECTION SETTINGS TO SAP NETWEAVER GATEWAY • SAP NETWEAVER GATEWAY SETTINGS`. (What a nice, simple path to follow.)

Add an entry exactly the same as the one shown in Figure 13.13. The word `GATEWAYLOCAL` appears in the menu of project nodes (such as your monster node) in Transaction `SEGW` under `SERVICE MAINTENANCE`, and later you'll see how this can be used to register the service.

This is going to sound silly, but you also have to ensure that SAP Gateway is actually active. In order to do this, go into a slightly different version of Transaction `SPRO`, called `SIMGH`. There, you have to choose the `IMG Structure Project Gateway 1.0`. Then, choose `GATEWAY • ODATA CHANNEL • CONFIGURATION • ACTIVATE OR DEACTIVATE SAP NETWEAVER GATEWAY`. If it's not active, then you can switch it on; if it's already active, then wonderful—nothing more to do here.

Adding the New Service

Earlier in the chapter, when you processed the screen shown in Figure 13.11 you sort of created the new SICF service. However, to really bring it out of the womb you need to go to another transaction that is delivered with the SAP Gateway add-on—namely, `/IWFND/MAINT_SERVICE`.

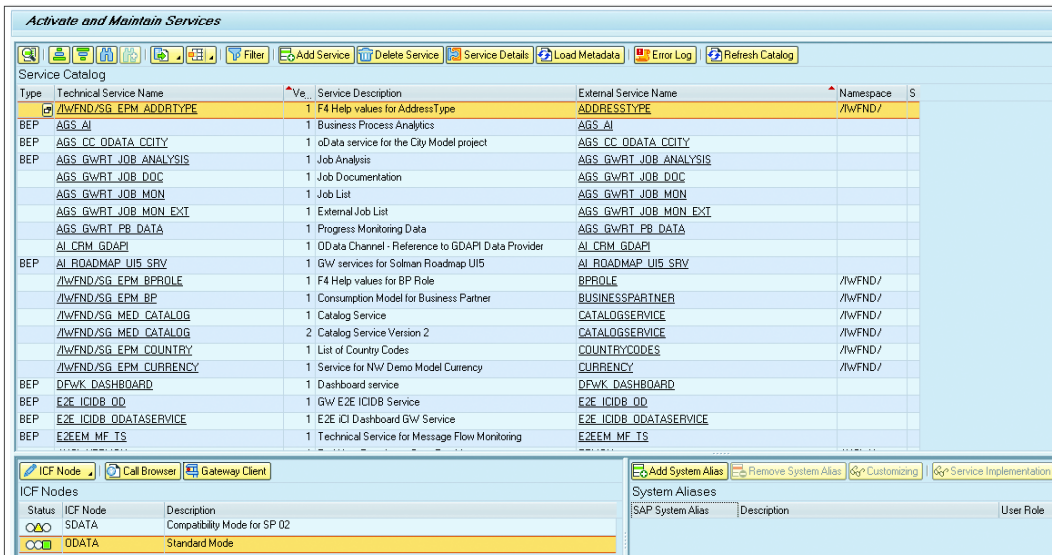


Figure 13.14 Adding the Service: Part 1

In Figure 13.14, you can see the initial list of SAP-supplied SAP Gateway services. Before you add your own, you need to make sure that the green light in the bottom-left-hand corner is on; otherwise, you're not going anywhere. If by any chance that light is not green, then right-click on the ICF node icon and choose CONFIGURE SICF. That will take you into the relevant section of SICF, where you need to activate everything in sight—but especially the `ADDRESSTYPE` node that lives under `ODATA` and everything above it in the tree.

Back on the `ACTIVATE AND MAINTAIN SERVICES` screen (Figure 13.14), click the `ADD SERVICE` button at the top; the screen shown in Figure 13.15 appears.

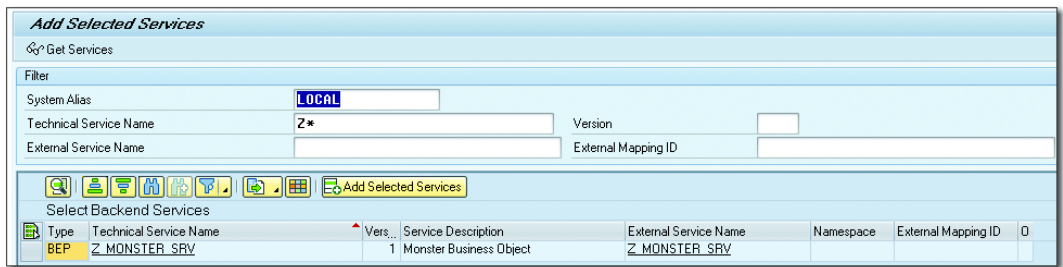


Figure 13.15 Adding the Service: Part 2

When this screen first appears, the list of services at the bottom will be blank. To find your service, do an `[F4]` dropdown search on the `SYSTEM ALIAS` field; you'll see the entry for `LOCAL`, which you created just now, so select that.

Enter `"Z*"` in the `TECHNICAL SERVICE NAME` field, because you're only interested in custom services as opposed to standard, SAP-delivered ones.

Those are all the fields you need to fill out. Click the `GET SERVICES` button at the top of the screen, and any created services that aren't yet active are listed, as shown at the bottom of Figure 13.15. Select your monster service by clicking the selection box to the left of the service and pressing the `ADD SELECTED SERVICES` button. The screen shown in Figure 13.16 appears.

In the screen shown in Figure 13.16, accept all the default values. You can rename the service name and model name if you want, but there really isn't any benefit to doing so. In this example, the `PACKAGE ASSIGNMENT` field is set as a local object. When creating a service for productive use, you would nominate an actual package.

The screenshot shows the 'SOD(1)/001 Add Service' dialog box with the following data:

Section	Field	Value
Service	Technical Service Name	Z_MONSTER_SRU
	Service Version	1
	Description	Monster Business Object
	External Service Name	Z_MONSTER_SRU
	Namespace	
	External Mapping ID	
	External Data Source Type	C
Model	Technical Model Name	Z_MONSTER_MDL
	Model Version	1
Creation Information	Package Assignment	\$TMP
		Local Object
ICF Node	Standard Mode	<input checked="" type="radio"/>
	Compatibility Mode for SP 02	<input type="radio"/>
	Set Current Client as Default Client in ICF Node	<input checked="" type="checkbox"/>

Figure 13.16 Adding a Service

After you've added your service, you can go back to the `/IWFND/MAINT_SERVICE` screen (Figure 13.14), and your monster service will have appeared somewhere on the list of services. You can search for this newly added service by using the `FILTER` option—for example, to look for services with “monster” in the description. Once you've selected your monster service, you're ready to test it.

Tip: Faster Way to Add a New Service

You've seen how to add a new service by going to Transaction `/IWFND/MAINT_SERVICE`, pressing the `ADD` button, searching for a service to add, adding that service, and then searching for the newly created service. You'll have noticed that there was a whole lot of searching going on.

A much easier way to add a new service (possible due to the configuration settings you made earlier to link destination `GATEWAYLOCAL` to the current SAP system) is to jump straight from Transaction `SEGW` into the correct part of Transaction `/IWFND/MAINT_SERVICE`, bypassing all the searching.

In the `SEGW` screen (Figure 13.17), the `Service Maintenance` node of the monster project is expanded. On the right-hand side of the screen, you can see a `REGISTER` button.

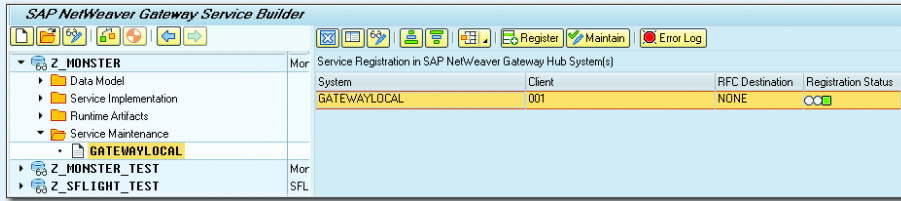


Figure 13.17 Jumping from SEGW to Service Maintenance

Clicking the REGISTER button in the SEGW screen in Figure 13.17 is a shortcut through the steps you followed before. First, the screen shown in Figure 13.16 appears, allowing you to add the service (without having to search for it), and then you end up on the /IWFND/MAINT_SERVICE screen (Figure 13.14), with the newly created monster service selected (again, without having to search for it).

Testing the New Service

You've got to the stage where you've added a new service to the list of available services. You're on the /IWFND/MAINT_SERVICE screen, with the monster service selected. When you added the service in the last section, a service node in Transaction SICF was created and activated. You can see the details of this service node by making sure your monster service is selected in the top half of the /IWFND/MAINT_SERVICE screen, making sure the line with the green light and the word ODATA next to it is selected in the bottom half of the screen, and then choosing ICF NODE • CONFIGURE SICF. The screen shown in Figure 13.18 appears. This screen shows the entry in Transaction SICF for your monster node.

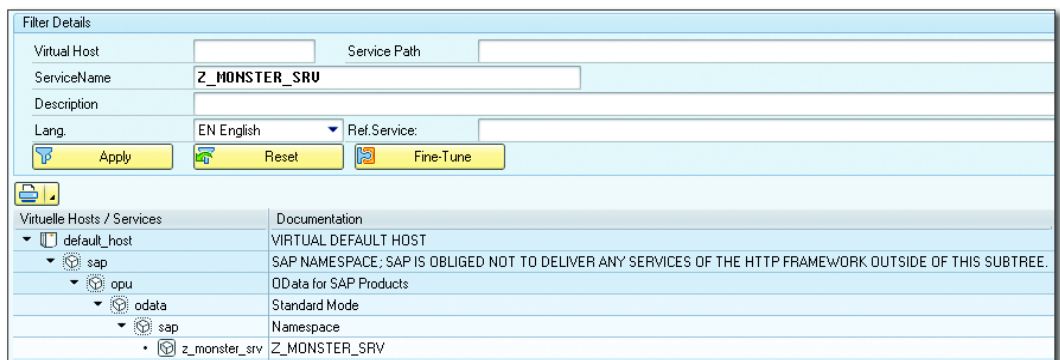


Figure 13.18 SICF Monster Node

Transaction SICF controls how incoming URLs are dealt with in SAP. A URL starts with the name of your SAP system, then the number 8000 (to be more precise, 8000 is the default value, but your SAP Basis people may have configured this to be something else), and then a bunch of parameters (e.g., *SAP System/8000/this/that/the other*).

The tree structure in an SICF node corresponds to a URL that someone (human or program) enters into a web browser, which then contacts your SAP system and uses the URL to find the SICF node and to call one or more ABAP handling classes, which are defined in the node definition. In this case, the system has defined a generic class called `/IWFND/CL_SODATA_HTTP_HANDLER` to respond to the incoming URL and forward the request to your SAP Gateway monster entity.

You could have a class that decides it cannot deal with the incoming request, but rather than raising an error just decides to let the next class in the chain have a go to see if it has better luck. This whole approach of using service nodes frees you from the task of having to create half a billion lines of code each time you want to expose a business object to the outside world by using the SICF incoming URL framework.

Note

For the examples in this chapter, make sure the services `/sap/bc/ui5_ui5` and `/sap/opu` are active. You do this by going into Transaction SICF and expanding the menu path for each of these services (e.g., SAP • BC) until you find the service listed. If the service description is in bold, then it's already active.

If the description is not in bold, then the service is dormant. Right-click it, choose the **ACTIVATE** option, and then say **YES** to the **ARE YOU SURE?** prompt.

Moreover, not only can you see the details of your new service from within SICF, but you can also test it. At the bottom of the `/IWFND/MAINT_SERVICE` screen is a **CALL BROWSER** button. Click it, and up will pop a web browser with the URL of the SICF node filled in (Figure 13.19).

Browser Support

Not all browsers seem to like OData and SAPUI5 very much, though they all promise to support SAPUI5 in their latest versions. If you get into trouble in your version of Internet Explorer (for example), then try Google Chrome.

```
<app:workspace>
  <atom:title type="text">Data</atom:title>
  <app:collection sap:label="Monster Header" sap:searchable="true" sap:pageable="false" sap:content-version="1" href="MonsterSet">
    <atom:title type="text">Monster Header</atom:title>
    <app:member title="Monster"><app:member-title>
      <atom:link href="MonsterSet/OpenSearchDescription.xml" rel="search" type="application/opensearchdescription+xml" title="SearchMonsterSet" />
    </app:member>
  </app:collection>
  <app:collection sap:searchable="true" sap:pageable="false" sap:content-version="1" href="Monster_ItemSet">
    <atom:title type="text">Monster_ItemSet</atom:title>
    <app:member title="Monster_Item"><app:member-title>
      <atom:link href="Monster_ItemSet/OpenSearchDescription.xml" rel="search" type="application/opensearchdescription+xml" title="SearchMonster_ItemSet" />
    </app:member>
  </app:collection>
</app:workspace>
```

Figure 13.19 Calling the Browser Test Tool

This is all wonderful: An external request can see that you have various monsters that can be searched for, each with some items. However, at the moment, if the external application (which is going to be an SAPUI5 application in this case) wants to look up a particular monster, or all the monsters, or a subset of them, then it's right out of luck—because your SAP Gateway monster entity doesn't have any ABAP code in it yet, just a bunch of empty generated classes.

13.3.2 Coding

In the last section, you not only created and activated a service to expose your monster model data to the outside world, you also generated some ABAP classes that are to be used to manipulate your monster data within the SAP system. These classes won't do anything much until you add some code to them. There are several steps to this process:

1. First, you have to understand the structure of an SAP Gateway service implementation, such as the one you will be creating for your monster data model. This will help you understand what code we need to add and where.
2. Code the data retrieval method `GET_ENTITY_SET`.
3. Test the data retrieval method `GET_ENTITY_SET`. This is going to be different than the testing you've been used to, because you have to test this from a web browser.
4. Code `GET_ENTITY_SET` for associated entries, where the entity set is the list of items associated with a monster entity.
5. Code error handling (i.e., how to pass error messages back to the calling application) so that they can appear in the SAPUI5 display.

Each of these steps is discussed in more detail next.

Understanding the Structure of an SAP Gateway Service Implementation

To get an idea of what an SAP Gateway service implementation looks like, go back to your SEGW transaction, select your `Z_MONSTER` service, and expand the `Service Implementation` and `Monster Set` nodes. You'll see the CRUD operations (Figure 13.20): Each one is represented by a generated ABAP method, into which you have to put some code.

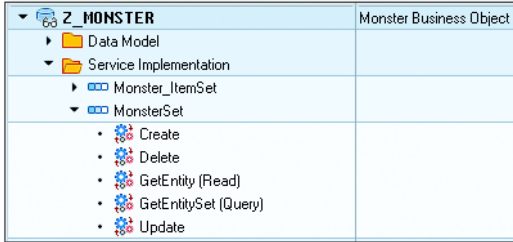


Figure 13.20 CRUD Classes that Need Some Code

In fact, there are four classes that have been generated by the system for each project. The first two are an abstract class called (in this case) `ZCL_Z_MONSTER_DPC` (where DPC stands for “data provider class”) and a class that inherits from this called `ZCL_Z_MONSTER_DPC_EXT`, which you can and will change. In the abstract class, there are methods for create, read, and so on, but all they do is raise an exception, so you need to redefine them all. The second two classes have almost the same names as the first two, except they have MPC instead of DPC (MPC stands for “model provider class”); take a look in the code for `ZCL_ZMONSTER_MPC`, and you will see generated code that sets up the data model based upon the configuration settings you've made.

To reiterate, the data provider class stores transactional methods, which you need to redefine, and the model provider class stores the data model, which you do not need to change, because it was created based on the settings you made setting up the data model.

Note that there are two types of entity (monster and monster item), but both entities live in the same project and are thus manipulated by the same service. Thus, the CRUD methods for each type of entity are all in the same DPC class.

Coding the CRUD methods is not actually going to be a major problem, because you already have the monster model class, which does all of these operations. The

only task is to translate the interface of the generated OData classes to the equivalent signatures of the corresponding monster model methods.

If you are coding CRUD methods and you do not have an existing model class, then (a) shame on you and (b) you will have to code the database reads and updates directly in the respective methods—which is not very reusable.

Take a look at a few examples of how to code some of these transactional methods, starting with the query operation `GET_ENTITY_SET`.

Coding the `GET_ENTITY_SET` Query Method

Start by selecting the `GET_ENTITY_SET (Query)` node, right-clicking, and choosing `Go To ABAP WORKBENCH`. That takes you to SE24 (after a message telling you that you haven't yet created the thing that you're trying to create, which you already knew). You'll see the `ZCL_Z_MONSTER_DPC_EXT` class, with half a million methods.

The first thing you want to do is give is to give this class the attribute `MO_MONSTER_MODEL`, which references your `monster_model` class, and a `CONSTRUCTOR`, in which you set up a monster model as soon as an instance of this is created by the SAP Gateway framework. The code for this is shown in Listing 13.1.

```
METHOD constructor.  
  
    super->constructor( ).  
  
    CREATE OBJECT mo_monster_model.  
  
ENDMETHOD.
```

Listing 13.1 Embedding the `monster_model` Class into the Data Provider Class

Now, you can move along to the inherited methods; start with the `MONSTERSET_GET_ENTITYSET` method. As is normal in SE24, select the method and click the `REDEFINE` icon. Then, you can add some code.

Delete the generated code (all that does is call an empty method that raises an exception), and then proceed to add your own coding. You will have noticed by now in this book the benefits of abstracting our monster logic into its own model class: for every new framework you look at, you can reuse the same class as before.

The purpose of the `GET_ENTITY_SET` method is to pass back a list of all the entities (monsters) that have been requested by an incoming URL. As can be seen in Listing 13.2, first you start dealing with instructions that have been passed in to the method (from the URL), such as the sort order (like the `SORTCAT` table in the ALV), which is here stored in `LT_SORTORDER`, then the selection options, and finally some more obscure things, like paging.

In regard to the selection options, they're listed in the URL in the form of a string, but the handler class in the SICF service node transforms this into the form of a table of selections (`IT_FILTER_SELECT_OPTIONS`), which is passed into the `GET_ENTITY_SET` method. Translate that selection table into the format that your monster model class data retrieval method likes, and then delegate getting the actual data to the monster model method `RETRIEVE_HEADERS_BY_ATTRIBUTE`.

Once you have the data, check if the incoming URL requested the data to be sorted in a particular order. If so, then build up a dynamic table to sort the retrieved data. Now check to see if the incoming URL specified that you only bring back certain lines (e.g., the first 10 rows of the table).

Next, an export table is filled up with all the entities (monster headers) that you want to send back to the caller. Right at the end is some code to enable testing using Eclipse.

Warning: Houston, We Have a Problem

Often, you try and test your application and get the screen but no data due to something called *CORS* (Cross Origin Resource Sharing). Adding the preceding call to `set header` solves the SAP Gateway (server) side of the equation. The local device (client) settings needed will be discussed in Section 13.4.3.

```
METHOD monsterset_get_entityset.
* Extract any instructions as to how to sort the result list from
* the incoming request
DATA: lt_techorder TYPE /iwbep/t_mgw_tech_order,
      lt_sortorder TYPE abap_sortorder_tab.

IF io_tech_request_context IS BOUND.
    lt_techorder = io_tech_request_context->get_orderby( ).
ENDIF.

* See if we have any selection criteria passed in
* We adapt the ODATA structure to the BOPF selection structure
DATA: ls_filter_select_options
```



```

        LIKE LINE OF it_filter_select_options,
        ls_select_options
        LIKE LINE OF ls_filter_select_options-select_options.
DATA: lt_selections TYPE /bobf/t_frw_query_selparam,
      lt_result      TYPE ztt_monster_header.

FIELD-SYMBOLS: <ls_selections> LIKE LINE OF lt_selections.

IF it_filter_select_options[] IS NOT INITIAL.
  LOOP AT it_filter_select_options
    INTO ls_filter_select_options.
    APPEND INITIAL LINE TO lt_selections
    ASSIGNING <ls_selections>.
    <ls_selections>-attribute_name =
    ls_filter_select_options-property.
    LOOP AT ls_filter_select_options- select_options INTO ls_select_
options.
      <ls_selections>-option = ls_select_options-option.
      <ls_selections>-sign   = ls_select_options-sign.
      <ls_selections>-low    = ls_select_options-low.
      <ls_selections>-high   = ls_select_options-high.
    ENDLLOOP. "Selection options for field being queried
  ENDLLOOP.
ELSE.
* No selection criteria have been passed in
* Set selection criteria so that all records are returned
  APPEND INITIAL LINE TO lt_selections
  ASSIGNING <ls_selections>.
  <ls_selections>-attribute_name = 'MONSTER_NUMBER'.
  <ls_selections>-option         = 'GT'.
  <ls_selections>-sign           = 'I'.
  <ls_selections>-low            = '0000000001'.
ENDIF. "Were any selection criterai passed in?

mo_monster_model->retrieve_headers_by_attribute(
  EXPORTING
    it_selections      = lt_selections
  IMPORTING
    et_monster_headers = lt_result ).

* Now we build a dynamic table which we will then use to
* sort the result a la SORTCAT in the ALV
FIELD-SYMBOLS: <ls_tech_order> LIKE LINE OF lt_techorder,
               <ls_sort_order> LIKE LINE OF lt_sortorder.

LOOP AT lt_techorder ASSIGNING <ls_tech_order>.
  APPEND INITIAL LINE TO lt_sortorder
  ASSIGNING <ls_sort_order>.
  <ls_sort_order>-name = <ls_tech_order>-property.
  IF <ls_tech_order>-order = 'desc'.
    <ls_sort_order>-descending = abap_true.
  
```

```

ENDIF.
IF <ls_tech_order>-property = 'NAME' OR
   <ls_tech_order>-property = 'COLOR' OR
   <ls_tech_order>-property = 'STRENGTH'.
  <ls_sort_order>-astext = abap_true.
ENDIF.
ENDLOOP. "Sort Order from Incoming Request

SORT lt_result BY (lt_sortorder).

* Query the incoming URL to see if we have to start from
* a specific point, and how many rows to display
DATA: ld_start_row TYPE sy-tabix,
      ld_end_row   TYPE sy-tabix.

"The URL may contain text like "$skip=5"
IF is_paging-skip IS NOT INITIAL.
  ld_start_row = is_paging-skip + 1.
ELSE.
  ld_start_row = 1.
ENDIF.

"The URL may contain text like "$top=10"
IF is_paging-top IS NOT INITIAL.
  ld_end_row = is_paging-skip + is_paging-top.
ELSE.
  ld_end_row = lines( lt_result ).
ENDIF.

* Export the final result
FIELD-SYMBOLS : <ls_entity_set> LIKE LINE OF et_entityset,
               <ls_result>      LIKE LINE OF lt_result.

LOOP AT lt_result FROM ld_start_row TO ld_end_row
  ASSIGNING <ls_result>.
  APPEND INITIAL LINE TO et_entityset
  ASSIGNING <ls_entity_set>.
  MOVE-CORRESPONDING <ls_result> TO <ls_entity_set>.
ENDLOOP.

* In development only change security settings to allow local testing
* http://scn.sap.com/community/gateway/blog/2014/09/23/solve-cors-with-gateway-and-chrome
DATA: ls TYPE ihttnvp.
ls-name = 'Access-Control-Allow-Origin'.
ls-value = '*'.
/iwbep/if_mgw_conv_srv_runtime~set_header( is_header = ls ).

ENDMETHOD. "Monster Set - GET_ENTITY_SET

```

Listing 13.2 Coding the Monster GET_ENTITY_SET

When you're adding the coding in Listing 13.2, please look at the signature of the method, where you will see quite a few `IMPORTING` parameters for things like selection options, filter values, sort order, and the like. It's fairly obvious what to do with such values once you have them, but you may be wondering where they come from in the first place. The comments in the code give the answer: The instructions come from the end of the URL.

There is an official OData document that details the structure of the URL that you pass in, which can be found at www.odata.org/documentation/odata-version-2-0/uri-conventions. Although this document is pretty heavy going at the start, toward the end you get to a list of suffixes to put at the end of the URL, like `$TOP`, `$SKIP`, and `$FILTER`.

For example, if the URL ended with `$skip=1$top=2$`, then you would get the second and third monster in the list (i.e., skip one row, then show the top two rows in the result set). This is for when you have a large number of rows but can only show about 10 at a time, because the display is appearing on a smartphone of some sort. Normally, you wouldn't know what data was where in the table, so instead you would fill in the selection options by ending the URL with `Z_MONSTER_SRV/MonsterSet?$filter=Name eq 'FRED'`. This will automatically fill the `IMPORTING` parameter `IT_FILTER_SELECT_VALUES`, which, as can be seen in the code, is transformed into something our monster model understands.

The `$TOP` and `$SKIP` (is there one called `$JUMP?`) examples are in the method because they tie in with concepts discussed earlier in the book (the expanded set of SQL options that comes with ABAP 7.4) and a concept that is discussed later, which is moving some things you would traditionally do in ABAP down into the database layer (i.e., down into SAP HANA).

Note

Say that you really did want the second and third rows only for some reason (e.g., you have a table of days of the week, and you want Tuesday and Thursday). In such a case, it would be crazy to do what was discussed previously (i.e., get all the records and then throw most of them away). That goes against everything you've ever been taught, most specifically to minimize the amount of data transferred between the database and the application server.

If you really wanted to do that, then you would need to have an extra field in the database table (day number, for example) and to use that field as a selection.

OData queries have been described as very database-centric. Because they are open source and not SAP-specific, they assume you have the full range of SQL options available, which is not the case in ABAP. For example, the `$TOP/$SKIP` options have a direct counterpart in full SQL, which is the `LIMIT` addition to the SQL query, as in `SELECT * FROM monster table LIMIT 1 OFFSET 2`.

SAP HANA

In an SAP HANA environment, if you didn't have such a SQL option available to use in native ABAP, then you'd push down the `$TOP/$SKIP` logic into a procedure in the database layer so that the discarding of unwanted rows would happen before sending the result back to the application server, thus complying with the golden rule. This is discussed in more detail in Chapter 15.

Testing the GET_ENTITY_SET Method

For now, go back into the `/IWFND/MAINT_SERVICE` transaction, and look for your monster service. Once you've found it, once again make sure that your cursor is on the ODATA line at the bottom of the screen, and click `CALL BROWSER`. Naturally, the exact same screen as before pops up, showing the data structure you've defined.

Now, take things to the next stage: Change the URL so that the suffix (i.e., the bit after `/Z_MONSTER_SRV/`) is `$metadata`. Now, you can see a more detailed description of the data structures (Figure 13.21).

```
<?xml:Edmx xmlns:sap="http://www.sap.com/Protocols/SAPData" xmlns:ms="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx" Version="1.0">
  - <edmx:DataService Namespace="Z_MONSTER_SRV"
  - <Schema xmlns:lang="en" xmlns="http://schemas.microsoft.com/ado/2009/09/edm" Namespace="Z_MONSTER_SRV">
  - <EntityType sap:content-version="1" sap:is-thing-type="true" Name="Monster">
    - <Key>
      - <PropertyRef Name="MonsterNumber"/>
    - </Key>
    <Property Name="MonsterNumber" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Number" MaxLength="10" Nullable="false" Type="Edm.String"/>
    <Property Name="Name" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Name" MaxLength="90" Nullable="false" Type="Edm.String"/>
    <Property Name="Sanity" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Sanity %age" Nullable="false" Type="Edm.Int32"/>
    <Property Name="Color" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Color" MaxLength="10" Nullable="false" Type="Edm.String"/>
    <Property Name="Strength" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Strength" MaxLength="20" Nullable="false" Type="Edm.String"/>
    <Property Name="HatSize" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Hat Size" Nullable="false" Type="Edm.Int32"/>
    <Property Name="NoOfHeads" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Heads" Nullable="false" Type="Edm.Int32"/>
    <Property Name="MonsterCount" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Monster Count" Nullable="false" Type="Edm.Int32"/>
    <Property Name="SanityDescription" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Sanity Description" MaxLength="30" Nullable="false" Type="Edm.String"/>
    <Property Name="HatSizeDescription" sap:filterable="false" sap:sortable="false" sap:updatable="false" sap:creatable="false" sap:label="Hat Size Description" MaxLength="30" Nullable="false" Type="Edm.String"/>
    <NavigationProperty Name="Monster ItemSet" ToRole="ToRole ItemsOfMonster" FromRole="FromRole ItemsOfMonster" Relationship="Z_MONSTER_SRV.ItemsOfMonster"/>
  - </Schema>
  - </EntityType>
  - </edmx:DataService>
  - </edmx:DataServiceVersion>
  - </edmx:Edmx>
```

Figure 13.21 Monster Metadata in a Browser

That may not seem very exciting, but the `$metadata` bit is important, because it's how the OData service implements its self-describing characteristic. For example,

you can see that `MonsterNumber` is the key; it's a string field of a maximum length of 10 characters, and it cannot be null. This is how a UI control can dynamically figure out how big to make a field on the screen and so on.

Now, take things one step further and change the URL once again so that it ends in `/odata/sap/Z_MONSTER_SRV/MonsterSet?sap-ds-debug=true`. You have to be very careful here; URL entries are case sensitive, and you need to make sure you type in the name of the entity set exactly as it appears in the `$metadata` list.

As can be seen in Figure 13.22, the results are more impressive this time. You'll see a big list of all the monsters, which is proof that the ABAP code has been called. You could (and you really should) put an external breakpoint in the method to see for yourself what's happening.

```

Body Request Response Server URI Runtime
<feed xmlns:=""http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns=""http://www.w3.org/2005/Atom"
xml:base=""http://ausxndnoblso01.ausap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_TEST_SRV/" xmlns:d=""http://schemas.microsoft.com/ado/2007/08/dataservices">
  <id>http://ausxndnoblso01.ausap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_TEST_SRV/MONSTERSet</id>
  <title type=""text"">MONSTERSet</title>
  <updated>2014-09-06T00:26:54Z</updated>
  - <author>
    <name/>
  </author>
  <link title=""MONSTERSet" href=""MONSTERSet" rel=""self"/>
  - <entry>
    <id>http://ausxndnoblso01.ausap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_TEST_SRV/MONSTERSet('4')</id>
    <title type=""text"">MONSTERSet('4')</title>
    <updated>2014-09-06T00:26:54Z</updated>
    <category scheme=""http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" term=""Z_MONSTER_TEST_SRV.MONSTER"/>
    <link title=""MONSTER" href=""MONSTERSet('4')" rel=""edit"/>
    - <content type=""application/xml">
      - <m:properties xmlns:=""http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns:d=""http://schemas.microsoft.com/ado/2007/08/dataservices">
        <d:HatSizeDescription/>
        <d:SanityDescription/>
        <d:MonsterCount>1</d:MonsterCount>
        <d:NoOfHeads>0</d:NoOfHeads>
        <d:Age>1</d:Age>
        <d:HatSize>0</d:HatSize>
        <d:Strength>REALLY STRONG</d:Strength>
        <d:Color>GREEN</d:Color>
        <d:Sanity>3</d:Sanity>
        <d:Name>FRED</d:Name>
        <d:MonsterNumber>4</d:MonsterNumber>
      </m:properties>
    </content>
  </entry>

```

Figure 13.22 Monster List in a Browser

If anything goes wrong, there's an error handling transaction called `/IWFND/ERROR_LOG`, which is not useful at all. It just keeps pointing you to the same SAP Note.

Coding GET_ENTITY_SET for Associated Entities

Now you want to be able to see the items for an individual monster, so it's time to redefine another method in your entity set. Listing 13.3 reads `IT_KEY_TAB`,

which gets filled in based on what you pass into the URL, and uses this to display the item details.

```

METHOD monster_itemset_get_entityset.
* Local Variables
  DATA : ls_key_tab          LIKE LINE OF it_key_tab,
         ld_monster_number TYPE zde_monster_number,
         lt_result          TYPE ztt_monster_items.

  READ TABLE it_key_tab INTO ls_key_tab
  WITH KEY name = 'MONSTER_NUMBER'.

  CHECK sy-subrc = 0.

  ld_monster_number = ls_key_tab-value.

  TRY.

      mo_monster_model->retrieve_monster_record(
        EXPORTING
          id_monster_number = ld_monster_number
        IMPORTING
          et_monster_items  = lt_result ).

      et_entityset[] = lt_result[].

  CATCH zcx_monster_exceptions.
    RETURN.
  ENDTRY.

ENDMETHOD. "Monster Item Set - GET_ENTITY_SET"

```

Listing 13.3 Coding the Method to Get All the Monster Items

As before, put an external breakpoint in the method so that you can see what's happening, and call up the browser as described previously. This time, change the URL so that it ends in `Z_MONSTER_SRV/MonsterSet('0000000007')/MonsterItemSet?sap-ds-debug=true`, and off you go again. You have to make sure that you get the uppercase letters right and the underscores in the correct positions.

The result (Figure 13.23) looks very similar to what you saw before. The method you coded brought the data back correctly from the database, and you can see all the items for a monster with one call.

Body	Request	Response	Server	URI	Runtime
<pre> <feed xml:base="http://ausxndnobl1.sap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_SRV/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom"> <id>http://ausxndnobl1.sap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_SRV/Monster_ItemSet</id> <title type="text">Monster_ItemSet</title> <updated>2014-09-06T01:05:21Z</updated> - <author> <name/> </author> <link title="Monster_ItemSet" rel="self" href="Monster_ItemSet"/> - <entry> <id>http://ausxndnobl1.sap2.com.au:8000/sap/opu/odata/sap/Z_MONSTER_SRV/Monster_ItemSet('7')</id> <title type="text">Monster_ItemSet('7')</title> <updated>2014-09-06T01:05:21Z</updated> <category scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" term="Z_MONSTER_SRV.Monster_Item"/> <link title="Monster_Item" rel="edit" href="Monster_ItemSet('7')"/> - <content type="application/xml"> - <m:properties xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"> <d:PartDescription/> <d:PartQuantity>1</d:PartQuantity> <d:PartCategory>HD</d:PartCategory> <d:MonsterNumber>7</d:MonsterNumber> </m:properties> </content> </entry> </pre>					

Figure 13.23 Monster with One Head

Coding Error Handling

Now you need to redefine all the other methods of both entities (create, delete, and so forth), but because all that involves is passing the call on to the equivalent method in your monster model class, there's no need to go through each one here in any detail.

This example puts a small piece of code in the `delete` method for you to see how to pass back exceptions to the calling application. It also sets things up so that trying to delete a monster always fails, so that you can see an error message in your SAPUI5 application later on. The code for this is shown in Listing 13.4.

```

METHOD monsterset_delete_entity.
* Local Variables
DATA: lo_message_container
      TYPE REF TO /iwbp/if_message_container.

lo_message_container =
/iwbp/if_mgw_conv_srv_runtime-get_message_container( ).

lo_message_container->add_message_text_only(
  EXPORTING
    iv_msg_type
      = /iwbp/if_message_container=>gcs_message_type-error
    iv_msg_text = `This monster does not want to be deleted` ).

RAISE EXCEPTION TYPE /iwbp/cx_mgw_busi_exception

```

```
EXPORTING
    message_container = lo_message_container.
```

```
ENDMETHOD.
```

Listing 13.4 Passing an Exception Back to the Calling Application

If you look at the definition of exception class `/IWBEP/CX_MGW_BUSI_EXCEPTION`, you'll see that there are a whole bunch of parameters that you can pass into the constructor, as well as some standard error messages that you could use, like `resource not found`. Exception classes were discussed back in Chapter 7.

Once you've redefined the CRUD methods for your monster entity, you've finished the first half of the process. Any sort of web application can now send a URL that points to your SAP system, and the SICF framework will invoke the entity set methods you've written to read or change the data in SAP.

13.4 Frontend Tasks: Creating the View and Controller Using SAPUI5

Section 13.2.2 talked about installing the add-ons in Eclipse that are needed for SAPUI5 development. You're now at the stage in which you'll create the frontend components that will run on the device where the SAPUI5 application is displayed. These components will be a view and a controller, and they live together inside an SAPUI5 application project.

To start this process, open Eclipse, and navigate to `NEW • OTHER • SAPUI5 APPLICATION DEVELOPMENT • APPLICATION PROJECT`. You may be puzzled by the labels on the radio buttons in Figure 13.24 in the `LIBRARY` box. In some versions of Eclipse, the labels tell you what they mean, but in others (like the one shown), you get secret codes. To put you out of your misery, the first one (`SAP.UI.COMMONS`) means that your target device is a desktop and the second one (`SAP.M`) means that your target device is a mobile. Choose the second (mobile) option so that later on you can see how the screen rearranges itself as the display area gets smaller.

The `OPTIONS` area in Figure 13.24 contains a `CREATE AN INITIAL VIEW` checkbox. Leave it selected, because you really need a view in order for the user to be able to see anything. As a result, another pop-up box appears asking questions about the view (this time, asking for the name of the view, e.g., `MONSTER OVERVIEW`),

and then it asks you what language you want to program the view logic in. In fact, although you want the controller to be written in JavaScript, you want the view to be in XML. Choose the XML option, and you'll see that an XML file will be created for the view and a JavaScript file will be created for the controller.

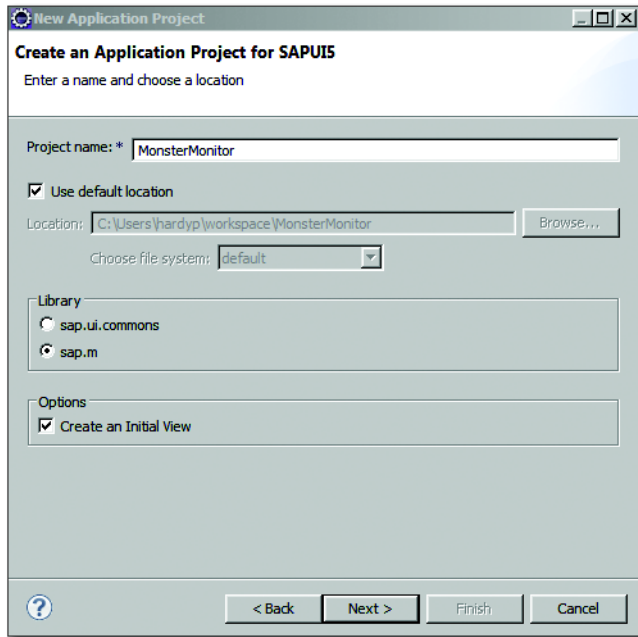


Figure 13.24 Creating an SAPUI5 Project: Part 1

After you click NEXT, the box shown in Figure 13.25 appears to tell you what's going to happen. In ABAP, you're used to programs being saved straight in the database. In other languages, the code is usually stored in a file structure on your local device. Here, you can see where the files are going to be stored. Then, you'll see the target device, which is a mobile device in this case (not that you would know from the secret code).

Next, there are a whole bunch of libraries that are going to be added to your SAPUI5 program. As mentioned earlier, you can think of this as a bunch of INCLUDE statements giving you access to assorted functions, the way you add an INCLUDE statement at the start of function modules or classes that are going to add data to a workflow container so that you can use assorted macros.

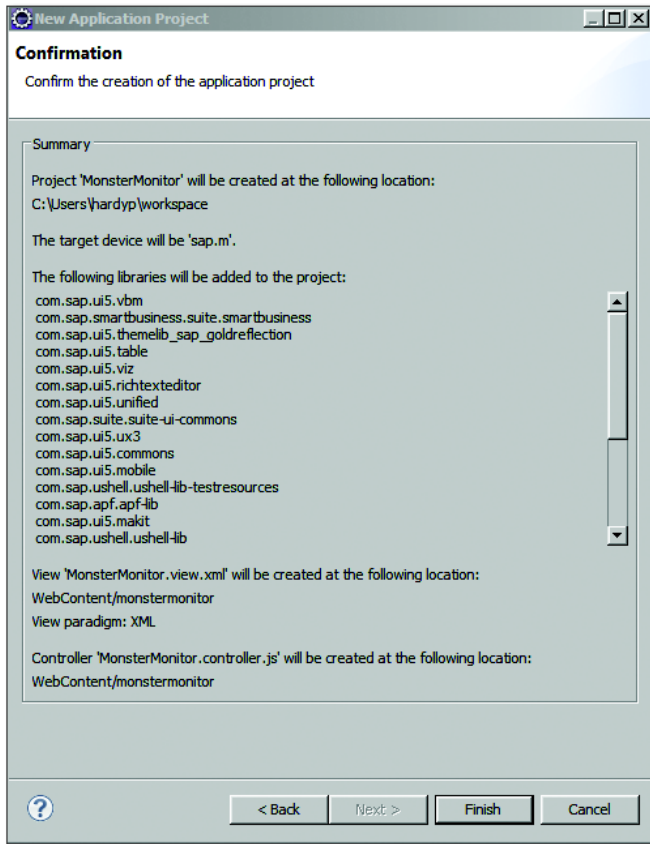


Figure 13.25 Creating an SAPUI5 Project: Part 2

Finally, you'll see that a view file and a controller file are going to be created. Your model lives in SAP and uses ABAP, and the view and controller will run on the local device. Hopefully, you can now see how valuable you can become if you know more than one language.

You might be asked if you want to open the project in a Java EE perspective, which is a puzzling question for the average ABAPer. The correct answer is yes. (Once you understand that the word "perspective" in that sentence refers to the way the screen will look, things make a lot more sense.)

If you remember the discussion of Eclipse back in Chapter 1, the screen in Figure 13.26 shouldn't be too scary. There are tabs for the view and controller, in the

same way that you would have two classes in ABAP to represent these constructs. You can see that SAP has already generated some code (e.g., to tell the view what its controller is), and there is explanatory text at the start of each method definition.

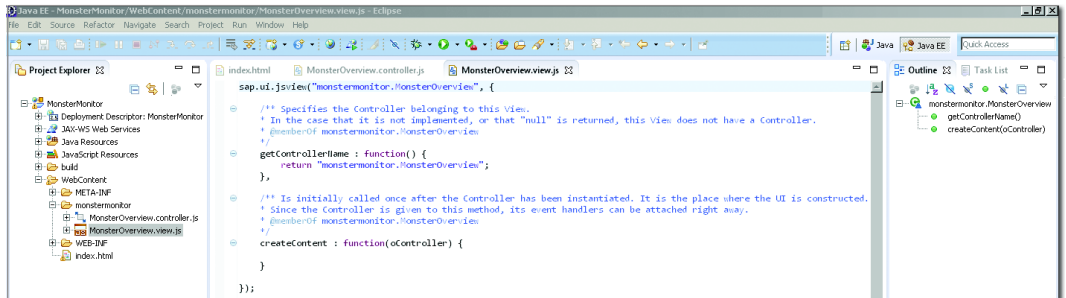


Figure 13.26 Eclipse Screen for Coding View and Controller

Just to recap, the model lives inside your SAP system, written in ABAP. The controller is on the local device, written in JavaScript. The view is also on the local device, but is written in XML, and as you'll see it will contain no code at all—just instructions on how to look pretty. Having the model, view, and controller in totally different languages enforces the separation of concerns by putting 1,000-mile high walls between them. That's a more effective way of doing this than saying to the programmer "Do not do such and such in the view; it's naughty."

Now the time has come to start coding, and this is where the fact that you're in a new language will start to make some people shake and sweat. If you're starting to worry about your job, remember that all the complicated business logic is still being programmed inside the ABAP system, and the SAPUI5 coding is solely concerned with making the display look nice (and SAP provides functions to make this as easy as possible) and handling user interaction.

This section first talks about the view (the part of the application in charge of making sure the display looks nice) and how it's organized technically. It then moves on to the controller, which uses JavaScript code to handle communication between the view and the model living in your SAP backend system. Finally, it explains how to test your newly created SAPUI5 application in order to make sure everything is working as expected.

13.4.1 View

There are three different files that will be created in your SAPUI5 application: an HTML file, an XML file, and another XML file, called a fragment.

The HTML file is the web page that every SAPUI5 application needs in order to run in a web browser on a laptop or on a mobile device. The XML file is the “proper” view; the example presented only has one, but complex applications can have many. The XML fragment is sort of like an `INCLUDE` file that can be reused by other view files. Fragments are used in the majority of the SAPUI5 examples presented in Section 13.5.

HTML File

An SAPUI5 application is, for all intents and purposes, a web page—so you need to define an HTML file that calls up your view when you access it via a URL. This was created for you automatically when you created a new SAPUI5 application; so in that new application follow the path `MONSTER DEMO • WEB CONTENT • INDEX.HTML`. The code to create this file is shown in Listing 13.5.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta http-equiv='Content-Type' content='text/html; charset=UTF-8' />

    <script src="resources/sap-ui-core.js"
      id="sap-ui-bootstrap"
      data-sap-ui-libs="sap.m"
      data-sap-ui-xx-bindingSyntax="complex"
      data-sap-ui-theme="sap_bluecrystal">
    </script>

<!-- only load the mobile lib "sap.m" and the "sap_bluecrystal" theme -->

  <script>
    sap.ui.localResources("monster");
    var app = new sap.m.App({initialPage:"idmyView1"});
    var page = sap.ui.view({id:"idmyView1",
      viewName:"monster.myView",
      type:sap.ui.core.mvc.ViewType.XML});
    app.addPage(page);
    app.placeAt("content");
  </script>
```

```
</head>
<body class="sapUiBody" role="application">
  <div id="content"></div>
</body>
</html>
```

Listing 13.5 Monster HTML File

In Listing 13.5, there are several sections, including two `<script>` sections. Somewhat like `ON_LOAD_OF_PROGRAM`, the first `<script>` section loads the core JavaScript library and any other libraries (groups of functions) you need. Sometimes, you have to add these manually; if you do, it will be mentioned in the generated comments. In this example, the comments tell you to only load one library and a theme, and the lines to do this are generated for you.

Then there is another `<script>` section, which performs the following tasks:

- ▶ States the folder where the view and controller live (in this case, `monster`).
- ▶ Creates a new mobile application with the variable name `APP` and specifies the name of the view that is going to appear first. This is somewhat like when you create a `DYNPRO` application in ABAP and say what screen comes first.
- ▶ Another variable, `PAGE`, says to use the `monster.myView` XML file.
- ▶ Links the two variables (they're objects, really) together by adding the initial page to the application.
- ▶ Indicates that the application is going to live in the `content` section of the web page.
- ▶ Defines the body of the web page, and inside that declares that the content of the web screen is indeed called `content`.

XML File

In Listing 13.5, you defined the view with the witty name of `MyView`, and that is indeed what the view file is called (its full name is `MyView.view.xml`). At this point, you're going to write some XML code to create a screen with a search box at the top and a list of monster header records in the main area of the screen, which can be filtered based on what the user puts in the search box. Once you're done with this process, the finished screen will look like Figure 13.27.

Name and Number	Color	Sanity	Hat Size
HUBERT Monster Id 2	GREEN	BONKERS	NORMAL HAT
HUBERT Monster Id 3	GREEN	BONKERS	NORMAL HAT
HUBERT Monster Id 3	GREEN	BONKERS	NORMAL HAT

Figure 13.27 Monster View Output

In order to end up with Figure 13.27, you have to do some coding. Every view needs a controller, so the very first thing that happens in your view file is to say what the name of the controller file is. The controller is going to have all the code; all you're doing in the view is laying out the screen and sometimes saying that a screen element will respond to being clicked. The controller will worry about what to do if a user does click such a field or button.

In Listing 13.6, you set the header level information for the view—not only the name of the controller but also some SAPUI5 libraries that it will need to run, such as `SAP.M`. This is similar to having `INCLUDE` files in old-fashioned ABAP programs full of reusable subroutines.

```
<core:View xmlns:core="sap.ui.core"
  xmlns:mvc="sap.ui.core.mvc"
  xmlns="sap.m"
  controllerName="monster.myView"
  xmlns:html="http://www.w3.org/1999/xhtml">
```

Listing 13.6 A View Defining the Name of Its Controller

All the code in Listing 13.6 was automatically generated for you at the time the view file was created (which was when you clicked the `FINISH` button in Figure 13.25), so now you can move on to defining your own code to define what your monster screen looks like. All good screens need a title at the top; for this one, the title will be `SEARCH FOR MONSTERS`. Also, put a button next to the title, for no good reason other than to show that you can put buttons in the title bar, just like the icons at the top of an ALV screen. Listing 13.7 shows the code for adding the title, and Figure 13.28 shows the result.

```

<Page title="Search for Monsters" class="marginBoxContent">
  <headerContent>
    <Button icon="sap-icon://action" />
  </headerContent>

```

Listing 13.7 Adding a Title



Figure 13.28 SAPUI5 Application Header Row

The first line on your screen (header) is the title. The next line down on that screen (subheader) is going to be a box in which you can enter the name of a monster. When you type "F", for example, the list will filter to show all the monsters with names starting with "F". This is a standard SAPUI5 screen element: a box with the word SEARCH on the left and a magnifying glass on the right. The good thing about using standard elements is that it gives your applications a uniform look.

Listing 13.8 presents the code to add the search field. You're going to place the search field inside the tool bar. You could also add a button next to the search field inside the tool bar if you wanted, just like in standard SAP screens that have two lines of icons at the top of the screen. The cumulative result is shown in Figure 13.29.

```

<subHeader>
  <Toolbar>
    <SearchField width="200px" liveChange="onSearch" />
  </Toolbar>
</subHeader>

```

Listing 13.8 Adding a Search Field to the Screen



Figure 13.29 Header with Search Box inside the Toolbar

You're done with the header section of the screen (headerContent in Listing 13.7 above). Now, you've come to the body (<content>) of the screen as shown in Listing 13.9, which will be somewhat like an ALV grid—only better looking. You can have as many different tables of data as you want on your screen, but in this case there's just the one. Time to start determining what this table is going to look like.

In Listing 13.9, you indicate that you're starting a new table called `MonstersTable` and then say where you're getting the data from. The view says that it wants to display the data that lives in a URL that ends with `/MonsterSet`. The controller takes care of working out the rest of the URL (i.e., where the model data comes from, which you know is inside your SAP system, but the view doesn't need to know any of this).

Next, there's another toolbar. You had one in the header of your screen, and now you have one for your table. This is just the same as when you have a GUI screen with several ALV grids. The main screen has an application toolbar and each grid has a title and its own row of icons at the top as well. In your SAPUI5 screen, set the title to `MONSTERS`. You could add some buttons here as well, if you so desired. Some are all in favor of filling the top of the screen with icons (e.g., Microsoft), and some say doing so confuses the user. In the SAPUI5 world, we're free to try and strike a happy medium.

```
<content>
  <Table id="idMonstersTable" inset="false" items="{path: '/MonsterSet'}">
    <headerToolbar>
      <Toolbar>
        <Label text="Monsters"></Label>
      </Toolbar>
    </headerToolbar>
```

Listing 13.9 Creating a Table and Giving It a Title

Next, in Listing 13.10 you will define the column headings, which naturally you can name anything you want. In traditional ALV reports, you're used to the column heading defaulting from the texts of the data elements that are used in the columns, but half the time you end up manually changing these anyway. In SAPUI5 world, you explicitly define the texts for all the column headings.

```
<columns>
  <Column>
    <Text text="Name and Number" />
  </Column>
  <Column>
    <Text text="Color" />
  </Column>
  <Column>
    <Text text="Sanity" />
  </Column>
  <Column>
    <Text text="Hat Size" />
```



```

    </Column>
</columns>

```

Listing 13.10 Defining the Column Titles

Now, it's time to decide what values should be displayed inside each column. In Listing 13.11, the first thing that you do is to say that the rows in this column can be navigated; that is, they do something when someone clicks the row. In this case, they tell the controller to invoke its `OnMonsterSelected` function.

Now, move on to the contents of each cell in the rows in the table. Start with the identifier: the title is the monster name, which people can search for, and the text description is the number. Usually, in SAP reports this is the other way around, but in real life nobody knows monster numbers—but they do know names (i.e., no one ever screams, "Oh no! Here comes Number 12678!", but they do scream, "Oh no! Here comes the Creeping Terror!").

The next three lines say what fields from your monster structure are going to be queried for their values to go into the row cells. There are two things to note here. First, you'll notice that each field from the structure (e.g., `color`), which is going to be replaced by a value, is enclosed in curly brackets (`{ }`). This is just the same as the way string processing works in ABAP now: If you enclose a variable in a string, then in that same set of brackets it gets replaced with the variable's value at runtime. Second, you may notice that there are fields from the monster header structure that have been used throughout this book: `Color` is a field stored in the database, but `SanityDescription` and `HatSizeDescription` are transient fields that get determined at runtime by looking up the text description of a related database field. Normally, you'd have to do some coding to get those text values. Here, you don't, because your model uses BOPF, which fills out such fields automatically without your having to worry your pretty little head about anything. The cumulative result is shown in Figure 13.30.

```

<items>
  <ColumnListItem type="Navigation"
                 press="onMonsterSelected">
    <cells>
      <ObjectIdentifier title="{Name}"
                      text="Monster Id {MonsterNumber}"
                      class="sapMTableContentMargin" />
      <Text text="{Color}" />
      <Text text="{SanityDescription}" />
      <Text text="{HatSizeDescription}" />
    </cells>
  </ColumnListItem>
</items>

```

```

        </ColumnListItem>
    </items>
</Table>
</content>

```

Listing 13.11 Defining the Cell Values for Each Column

Monsters			
Name and Number	Color	Sanity	Hat Size
HUBERT Monster Id 2	GREEN	BONKERS	NORMAL HAT
HUBERT Monster Id 3	GREEN	BONKERS	NORMAL HAT

Figure 13.30 SAPUI5 Application: Including Monster Table

At the bottom of the page is the footer. Once again, you can put a toolbar with buttons here, and this is just what you do in Listing 13.12. As in the header, the buttons don't do anything. They're just there to prove that they can be there. The buttons are shown in Figure 13.31.

```

        <footer>
            <Toolbar>
                <ToolbarSpacer/>
                <Button text="Accept" type="Accept" />
                <Button text="Reject" type="Reject" />
                <Button text="Edit" />
                <Button text="Delete" />
            </Toolbar>
        </footer>
    </Page>
</core:View>

```

Listing 13.12 Defining the Footer: Adding Some Buttons

GREEN	BONKERS	NORMAL HAT	>
GREEN	BONKERS	NORMAL HAT	>
GREEN	BONKERS	NORMAL HAT	>

Accept
Reject
Edit
Delete

Figure 13.31 Action Buttons in Footer of SAPUI5 Application

You're finished with the main view!

Fragment XML File

Maybe that last exclamation mark was a tad strong; you should know that few, if any, applications have only one screen. In this case, what you want when a user clicks on a monster is for a dialog box to open with the full monster header details at the top and a table of monster items at the bottom. This is technically possible to achieve in the SAP GUI, though when someone asks you to code this, you probably waggle your finger at the requester and say "Tut, tut, now, now, come, come," or words to that effect, and try to talk them out of the request because there's a fair bit of work involved. Happily, in the SAPUI5 world this is fairly painless—which is just as well, because this is the sort of extra requirement that pops up ten minutes before go-live on a regular basis.

To optimize the performance of your application, load only the main view to start off with. Any other views you need are loaded on demand; to do this, you define the other views (pop-up boxes, in this case) as fragments. Here, you're going to display your monster detail information in some style using a fragment, a process that could be described as a fragment style monster. Ultimately, you'll see that the controller function `onMonsterSelected` calls the fragment that you're about to define—a fragment the purpose of which is to pop up a big dialog box full of monster details (Figure 13.32).

Monster Details

Monster Name (Number): FRED (12)
Color: GREEN
Strength: REALLY STRONG
Days Old: 1
Number of Heads: 1

Monster Items

Item	Part Category	Part Category Description	Part Qty
000010	HD	Head	1
000011	AR	Arm	1
000012	AR	Arm	1
000013	LG	Leg	1
000014	LG	Leg	1
000015	TN	Tentacle	1

Rampage Delete Cancel

Figure 13.32 Monster Item Pop-Up Box

Start your fragment definition in Listing 13.13 by indicating that this will be a dialog box, giving this box a title, and saying that it takes up 60% of the screen. The result is shown in Figure 13.33.

```
<?xml version="1.0" encoding="UTF-8"?>
<core:FragmentDefinition xmlns="sap.m"
  xmlns:core="sap.ui.core" xmlns:f="sap.ui.layout.form">
  <Dialog title="Monster Details" contentWidth="60%">
    <content>
```

Listing 13.13 Defining Header Details of the Fragment (Pop-Up Box)

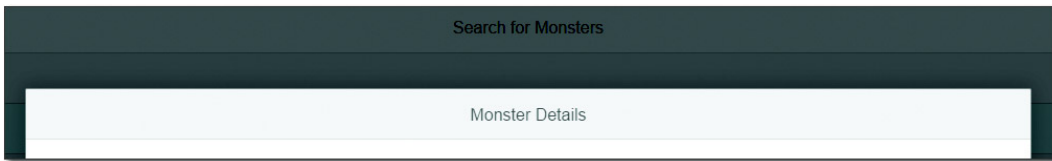


Figure 13.33 Dialog Box Title

In Listing 13.14, you'll set up the header details of the selected monsters. At the start of the code, declare that those details are going to be laid out in `SimpleForm`, one of the many UI elements available for you to use. Then, say how many columns the form is going to have (one in this case), and so on; this is just like passing values into a function module or method.

You'll recognize from Listing 13.11 the technique for declaring what the name of the fields are going to be. For each such field, add the name of an SAP DDIC field with `{}` brackets around it to store the value that is going to be dynamically determined at runtime. The runtime system knows what monster header you're talking about, because you've just navigated from that particular monster, and the code in the `onMonsterSelected` handler passes in the correct monster. The result is shown in Figure 13.34.

```
<f:SimpleForm minWidth="1024" maxContainerCols="2"
  editable="false" layout="ResponsiveGridLayout" labelSpanL="3"
  labelSpanM="3" emptySpanL="4" emptySpanM="4" columnsL="1" columnsM="1">
  <f:content>
    <Label text="Monster Name (Number)" />
    <Text text="{Name} ({MonsterNumber})" />
    <Label text="Color" />
    <Text text="{Color}" />
    <Label text="Strength" />
    <Text text="{Strength}" />
    <Label text="Days Old" />
```

```

    <Text text="{Age}" />
    <Label text="Number of Heads" />
    <Text text="{NoOfHeads}" />

  </f:content>
</f:SimpleForm>

```

Listing 13.14 Defining Layout and Field Texts for the Pop-Up Box

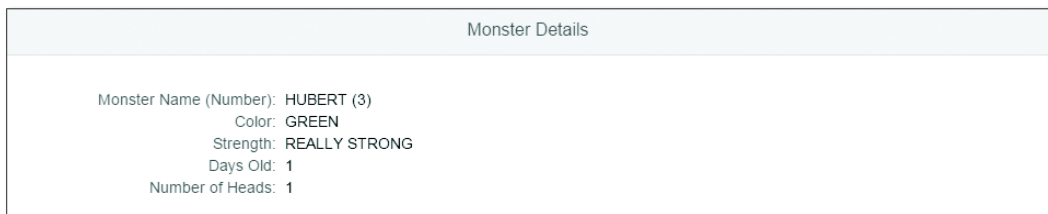


Figure 13.34 Dialog Box Header Details

Next, in Listing 13.15 you'll define a table that shows all the items for the monster you're drilling into: how many arms, heads, tails, and so on. The way to define such a table is exactly the same as the way you did this for the table of header details in Listing 13.10 and Listing 13.11. First, declare a `path` so that the controller can ask the model to get some monster item data. Then, there's a title at the top. Finally, define the column names and their widths, and say what values need to go in the columns at runtime. The result is shown in Figure 13.35.

```

<Table inset="false" items="{path: 'Monster_ItemSet'}">
  <headerToolbar>
    <Toolbar>
      <Label text="Monster Items" />
    </Toolbar>
  </headerToolbar>
  <columns>
    <Column width="100px">
      <Text text="Item" />
    </Column>
    <Column width="100px">
      <Text text="Part Category" />
    </Column>
    <Column width="100px">
      <Text text="Part Category Description" />
    </Column>
    <Column width="100px">
      <Text text="Part Qty" />
    </Column>
  </columns>

```

```

<items>
  <ColumnListItem>
    <cells>
      <Text text="{MonsterItem}" />
      <Text text="{PartCategory}" />
      <Text text="{PartDescription}" />
      <Text text="{PartQuantity}" />
    </cells>
  </ColumnListItem>
</items>
</Table>
</content>

```

Listing 13.15 Defining the Table of Monster Items for a Given Monster

Monster Details			
Monster Name (Number): HUBERT (3) Color: GREEN Strength: REALLY STRONG Days Old: 1 Number of Heads: 1			
Monster Items			
Item	Part Category	Part Category Description	Part Qty
000010	HD	Head	1
000011	AR	Arm	1
000012	AR	Arm	1
000013	LG	Leg	1
000014	LG	Leg	1
000015	TA	Tail	1

Figure 13.35 Dialog Box with Item Table

In Listing 13.16, you add some buttons at the bottom. Two of them ask the controller to call a function when the button is pressed; the RAMPAGE button is just there for fun. The completed dialog box, along with the buttons at the bottom, is shown in Figure 13.36.

```

<buttons>
  <Button text="Rampage"
    icon="sap-icon://physical-activity" />

```

```

<Button text="Delete" type="Reject"
  icon="sap-icon://delete"
  press="onMonsterDelete" />
<Button text="Cancel"
  press="onMonsterDetailDialogCancel" />
</buttons>
</Dialog>
</core:FragmentDefinition>

```

Listing 13.16 Adding Action Buttons at the Bottom of the Pop-Up Box

You may be wondering how to know what value to pass into the `ICON` field. In normal SAP, you use Transaction `<ICON>` to call up a list of all the pretty icons you can use in your screens. The equivalent in SAPUI5 is to go to the following URL, where you will get such a list:

<https://openui5.hana.ondemand.com/test-resources/sap/m/demokit/icon-explorer/index.html>

Monster Details

Monster Name (Number): HUBERT (3)
 Color: GREEN
 Strength: REALLY STRONG
 Days Old: 1
 Number of Heads: 1

Monster Items

Item	Part Category	Part Category Description	Part Qty
000010	HD	Head	1
000011	AR	Arm	1
000012	AR	Arm	1
000013	LG	Leg	1
000014	LG	Leg	1
000015	TA	Tail	1




 Rampage
  Delete
  Cancel

Figure 13.36 Dialog Box with Action Buttons at the Bottom

In this case, I wanted a picture that looked like a monster for my RAMPAGE button and the nearest I could find was the PHYSICAL ACTIVITY button. Rampaging through the village terrorizing the peasants is indeed a physical activity.

13.4.2 Controller

That does it for the view files; now it's time to code the controller. As mentioned earlier, the views are in XML and the controller is in JavaScript, so the file ends in "JS". In a functional language, which JS is, functions are first-class citizens, so don't be surprised to see lots of functions in the following coding strutting all about the place and looking pleased with themselves.

Code in the controller consists of a series of function definitions—and unlike ABAP methods, here this means the definition (signature) and implementation (coding) all together. The steps for coding the controller are as follows:

1. Create the initialization method that's called when the controller is first created.
2. Define the search box, which filters the list of monsters.
3. Define the dialog box that appears when a user clicks on a particular monster.
4. Define the functions to respond to any buttons the user presses once that dialog box is open.
5. Define a function you need in the controller to help with testing.

Initializing the Controller

The controller brokers communication between the model and the view, so when you start coding the controller the first thing you're going to do is say what model you're interested in, and then move on to handling events that the view tells you have happened.

Listing 13.17 starts off by saying "Hello, I am a controller and my name is..." so that the view can refer to it. Then, you swiftly move on to saying what model you want. When the SAPUI5 application queries your SAP system to get the SAP Gateway service, it needs a URL that can get to the correct place. What you see in the declaration of variable `OMODEL` (starting with `/SAP/OPU/ODATA`) is the path to the SICF node where your SAP Gateway service is located. The bit at the start of the URL is determined by another function at the end of the controller file (this will be covered in due course).


```
jQuery.sap.require("sap.m.MessageBox");

sap.ui.controller("monster.myView", {

  /**
   * Called when a controller is instantiated and its View controls
   * (if available) are already created.
   * Can be used to modify the View before it is displayed, to bind event
   * handlers and do other one-time initialization.
   * @memberOf monster.myView
   */
  onInit: function() {
    var oModel = new sap.ui.model.odata.ODataModel(this.getUrl("/
sap/opu/odata/sap/Z_MONSTER_SRV/"), true);
    this.getView().setModel(oModel);
  },

```

Listing 13.17 Controller Declaring the Model It Uses

You will see a whole bucket load of generated code in the controller file, including a lot of commented-out function skeletons with comments above them saying what those functions are used for. For example, you can see the comments that describe the `onInit` function, which is pretty much the same as its namesake in Web Dynpro. (Indeed, the example in Chapter 12 involved the creation of an instance of the monster model when the `WDDOINIT` method was called, which was then used throughout the application; this is what's happening here as well.)

There's no need to go through all those generated comments here, because they're all self-explanatory, so skip straight to the functions you define that are specific to your monster application.

Defining the Search Box Function

Back in your view, you defined a search field at the top of the screen and said `liveChange = onSearch`. The right-hand side of the assignment (`onSearch`) is the name of the function you are about to define, and the left-hand side of the assignment (`liveChange`) indicates that the function is called each time the user changes even one letter in the search box (as opposed to waiting for the user to click the `SEARCH` icon).

To recap what the search box does, if you type in "F", then all monsters with names containing "F" pop up, and all others vanish from the list. If you type in "D", then all monsters with D in their name pop up, and so forth. This would be quite a complicated bit of coding normally. Here, you have to do virtually nothing.

You code the `onSearch` function in Listing 13.18, and as you can see, you're not passing very much into the boilerplate code. You say what field you want to filter on (monster name, in this case). In the second part, you say what table you're talking about from the view (the table of monsters). The other two input parameters are generic values.

In the first part of Listing 13.18, the variable `SQUERY` is filled with the contents of the search box. If there is anything in the box, then the filter object is filled with a list of all monsters whose names contain the value in the search box. In the second part, the result list is retrieved into variable `OTABLE`, and that table is filtered by the filter object created in the first half of the listing.

```
onSearch: function(oEvent) {

    // add filter for search
    var aFilters = [];
    var sQuery = oEvent.getSource().getValue();
    if (sQuery && sQuery.length > 0) {
        var filter = new sap.ui.model.Filter("Name",
sap.ui.model.FilterOperator.Contains, sQuery);
        aFilters.push(filter);
    }

    // update list binding
    var oTable = this.getView().byId("idMonstersTable");
    var binding = oTable.getBinding("items");
    binding.filter(aFilters, "Application");
},
```

Listing 13.18 Coding the `onSearch` Event Handler

Defining the Dialog Box Function

Next, you may recall that in the view you set the `column list items` to be of type `navigation`, which would call `onMonsterSelected` when a user selected a given row. This is what controllers are all about; this function is called from one view and then goes about linking another view (the dialog box) with the model. Now, it's time to code that function using the code in Listing 13.19.

At the start of the code, you check if the dialog box is `undefined` (i.e., this is the first time anyone has clicked on a monster). If so, then the `fragment` is loaded so that you know what the dialog box has to look like.

Once you're sure your dialog box is good to go, it's time to get the model and then get the `source`, which is the monster that has been selected, and then the context path, which is any nodes in your SAP Gateway model linked to the monster header (you only have the monster items). What you're doing is setting the context of the dialog box and all its children to the same context as the item that was selected, which is why the data bindings in the dialog box change as you select each monster.

Finally, the `expand` parameter tells the model to return not just the monster entity but also the list of items for that monster. Technically, this is a list of the monster entity's associated `Monster_ItemSet` entities.

Once again, this is quite simple in comparison to setting up a DYNPRO modal subscreen with header fields at the top and an ALV grid at the bottom.

```
onMonsterSelected: function(oEvent) {
    console.log('onMonsterSelected ' + oEvent.getSource().getBindingContextPath());
    if (this._monsterDetailDialog === undefined) {
        this._monsterDetailDialog = new
sap.ui.xmlfragment("monster.MonsterDetail", this);
        this._monsterDetailDialog.setModel(this.getView().getModel());
    }
    this._monsterDetailDialog.open();
    this._monsterDetailDialog.bindElement({
        path: oEvent.getSource().getBindingContextPath(),
        parameters: {
            expand: 'Monster_ItemSet'
        }
    });
},
```

Listing 13.19 Coding the Function `onMonsterSelected`

Defining the Buttons in the Dialog Box

Next, you have to deal with the buttons at the bottom of your dialog box. Only two buttons do anything; the definitions of those buttons point to `onMonsterDetailDialogCancel` and `onMonsterDelete`, respectively. Naturally, these are functions you have to code as well.

Listing 13.20 shows the `onMonsterDetailDialogCancel` function, and as you can see, the name is almost as long as the code inside the function. You just call a standard SAPUI5 method to close the dialog box.

```
onMonsterDetailDialogCancel: function(oEvent) {
    this._monsterDetailDialog.close();
},
```

Listing 13.20 Coding the onMonsterDetailDialogCancel Function

That wasn't very exciting; time to move on to deleting a monster. The code to do this in Listing 13.21 is a bit meatier. First, once again you get the model and then the source to see which monster you're talking about. The source is the line in the monster list you selected, the context of that line is what refers to the specific monster, and the path of that context is what you need to pass to the model. Hence, `oEvent.getSource().getBindingContext().getPath()`.

When the `remove` method of the model is invoked, the application goes to the SAP system and calls the `MONSTERSET_DELETE_ENTITY` method of your SAP Gateway service. This will raise an exception if something goes wrong, and it certainly will in this case. This example codes the delete method in such a way that an exception is always raised, and thus the error function will be called.

```
onMonsterDelete: function(oEvent) {
    var oModel = this.getView().getModel();
    var oPath = oEvent.getSource().getBindingContext().getPath();

    oModel.remove(oPath, {
        success: this._monsterDeleteSuccess,
        error: this._monsterDeleteError
    });
},

_monsterDeleteSuccess: function(oEvent) {
    this.onMonsterDetailDialogCancel();
    sap.m.MessageToast.show('Deleted', {
        duration: 5000,
        my: 'center center',
        at: 'center center'
    });
},

_monsterDeleteError: function(oEvent) {
    var oResponse = JSON.parse(oEvent.response.body);
    if (oResponse.error.message) {
        sap.m.MessageBox.alert(oResponse.error.message.value);
    } else {
        sap.m.MessageBox.alert('An Unknown Error Detected');
    }
},
```

Listing 13.21 Coding Functions Involved in Deleting a Monster

As it turns out, you cannot test the `DELETE` function from Eclipse. The online SAP help gives all sorts of excuses as to why this is. Regardless, you have to wait until you have imported the application into your SAP system, which is discussed in Section 13.6.

Had the deletion succeeded, then the success message would have appeared as a toast message, which means the notification pops up (hence the name) on the screen for a given number of seconds and then fades away, like the notification you get when a new email arrives. In case of an error, the alert is like an information message; that is, it stays on your screen until you click on the pop-up box to acknowledge that you've seen the message.

Defining a Function to Help with Testing

The final function in your controller (Listing 13.22) has nothing to do with user interaction. It concerns making sure the start of the URL is correct when trying to contact the SAP system.

```
getUrl : function(sUrl) {  
    if ( sUrl == "" )  
        return sUrl;  
    if (window.location.hostname == "localhost") {  
        return "proxy" + sUrl;  
    } else {  
        return sUrl;  
    }  
}  
  
});
```

Listing 13.22 Coding the `getUrl` Function to Aid with Testing

To reiterate, the first 95% of the controller code was doing the normal job of a controller, but Listing 13.22 related solely to helping you test the application, which leads in nicely to the next section.

13.4.3 Testing Your Application

Right at the start of the controller, when the model was being retrieved, the `getUrl` function was called, which specified the last half of the URL (the path to the SICF node) but was looking for the first half of the URL, which says what SAP system you're pointing to.

When testing earlier in the chapter, you just typed in a big long URL with the address of your SAP host at the start and the path to your SAP Gateway service at the end. When testing the application, the Eclipse environment is going to put the words "local host" at the start of the URL, so you need to substitute this for the actual host name of your SAP system.

How do you do that? What is this "proxy" of which I speak?

You need to navigate to `WEBCONTENT • WEB-INF • WEB.XML`, and make a quick change to that file. When looking at this, you'll see that everything has been generated automatically. The bit you need to change is shown in Listing 13.23.

```
<!-- ===== -->
<!-- UI5 proxy servlet -->
<!-- ===== -->

<servlet>
  <servlet-name>SimpleProxyServlet</servlet-name>
  <servlet-class>com.sap.ui5.proxy.SimpleProxyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SimpleProxyServlet</servlet-name>
  <url-pattern>/proxy/*</url-pattern>
</servlet-mapping>
<context-param>
  <param-name>com.sap.ui5.proxy.REMOTE_LOCATION</param-name>
  <param-value>http://my_sap_host_name:8000</param-value>
</context-param>
```

Listing 13.23 Defining the Actual SAP System for a Local Host

You can find the host name of a given SAP system by following the menu path `SYSTEM • STATUS` and looking at the `DATABASE DATA` section. You can ask your SAP Basis person if the number at the end is 8000 or something else.

Why go through all this rigmarole, you may ask? Because to start off with you want to test this on your local device, but eventually your application will become productive, and the local host prefix will not apply. More importantly, if you look on the Internet you will see dozens of people trying to test SAPUI5 applications, getting screens with no data, and wondering why. It's because they're not using the simple proxy servlet just described and are running into the so-called CORS problem. (Recall that you have to make a change in the code to get the server side to work; for a refresher, refer back to Listing 13.2.)

Due to all the effort you just made to define proxy settings and the like, you can now select **MONSTER MONITOR** in the top-left-hand corner of the Eclipse screen, and choose **RUN AS • WEB APP PREVIEW**. A new tab appears, and after a few seconds your application screen appears (Figure 13.37).

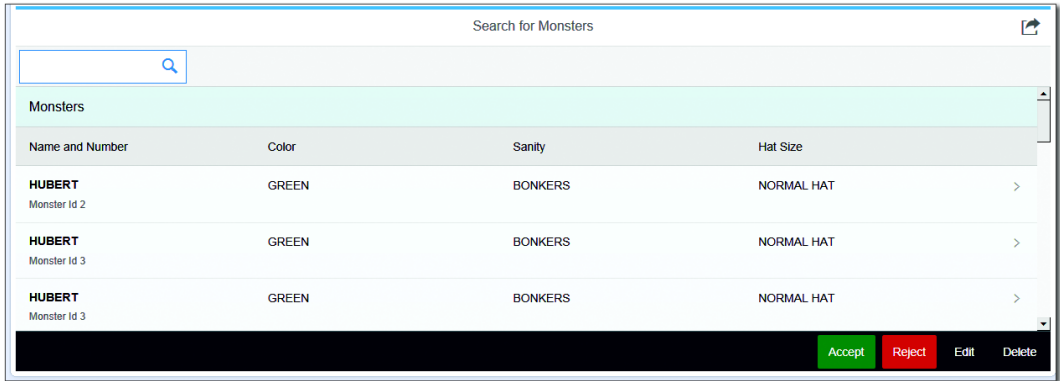


Figure 13.37 Testing an SAPUI5 Application in Eclipse

If you click on the blue ball in the top-right-hand corner, then your default Internet browser will open (as mentioned earlier, if your default browser is Internet Explorer, then this may not work), and you can see the screen in a full web page.

13.5 Adding Elements with OpenUI5

You may have heard that the Eskimos have three million words for snow. In the same way, there are often many different terms referring to the same (or almost the same) thing in the SAP sphere, and this confuses people to no end. For example, you may have heard the terms “UI Development Toolkit for HTML5,” “SAPUI5,” and “OpenUI5” and wondered what the difference was.

The first two are one and the same. SAPUI5 is a JavaScript library of assorted functions that relate to UI elements. OpenUI5 is the open-source version of SAPUI5. The only difference is that SAPUI5 contains functions that solely relate to SAP-specific concepts and wouldn’t make sense in any other environment. The functions in OpenUI5 can be used by anybody at all, even if they never have and would not touch SAP with a 10-foot bargepole. As might be imagined, OpenUI5 contains a very large percentage of the SAPUI5 functions.

OpenUI5 and Open Source

OpenUI5 is often described as “open source,” but many people would say that’s not quite true. The product is open source in that it’s free to download and you can use it any way you want without having to pay anybody a license, but that’s only half of the open-source story.

With real open-source products, not only can you view the source code, you are actively encouraged to fix bugs and make enhancements and to send those changes to the project so that the whole world can benefit. SAP is not quite willing to go that far yet, but SAP claims that it will make this a proper open-source product eventually.

Thanks to OpenUI5, there is an absolutely stunning online resource to help you start adding all sorts of exotic features in no time at all, available at <http://sap.github.io/openui5/index.html> (Figure 13.38). This will really come in handy in real life—because, let’s face it, your users are going to want the moon on a stick when it comes to the look and feel of their applications.

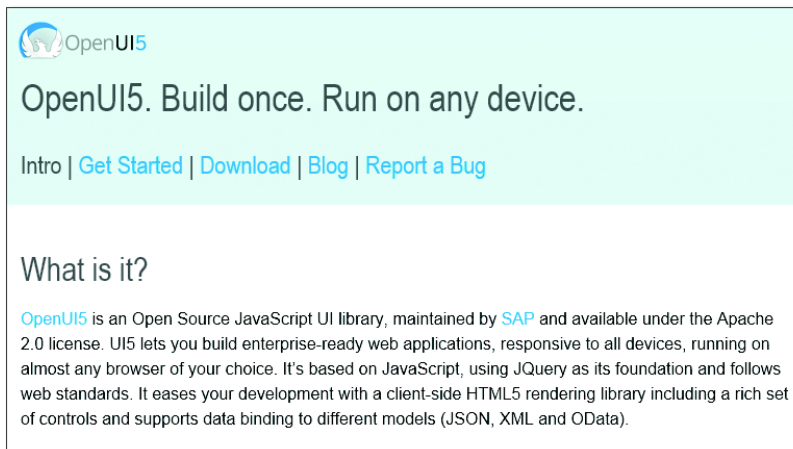


Figure 13.38 OpenUI5 Launch Page

If you click the **DOWNLOAD** link shown in Figure 13.38, you can get the latest version of OpenUI5 (a new version comes out every few weeks), and more importantly for the purpose at hand there’s a link to the developer guide. (When you open the developer guide, it seems like all references to “open” UI5 are gone, and the logo and description at the top of the screen refer to SAPUI5, as seen in Figure 13.39. However, you’re really still in the OpenUI5 developer guide.)

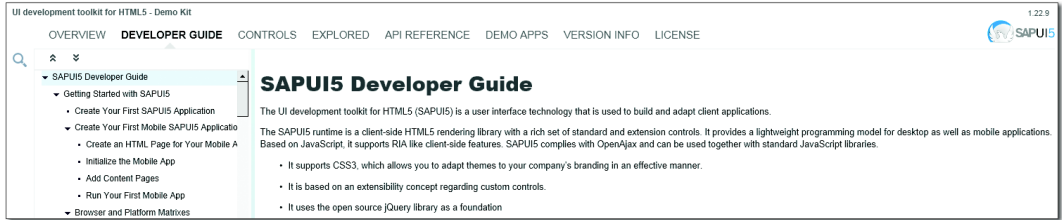


Figure 13.39 OpenUI5 Developer Guide

At this point, to explore the vast array of options open to you, navigate to the EXPLORED tab. As you can see in Figure 13.40, on the left is a huge list of UI elements. If you select one, then the right-hand side of the screen displays a description of what this entity does. If you click SAMPLES, you can see a working example, and then you can click the CODE button to see the code needed to make this work. Usually, you'll see some code samples: a view, a fragment, and a controller. By now, you hopefully understand which one does what.

Just as an example, if you want to add a dropdown menu at the top of your monster application in the form of an action sheet (as shown in Figure 13.40), it's the work of minutes to copy the sample code from the website into your controller and view files, add a fragment file, and then make some monster-related changes.

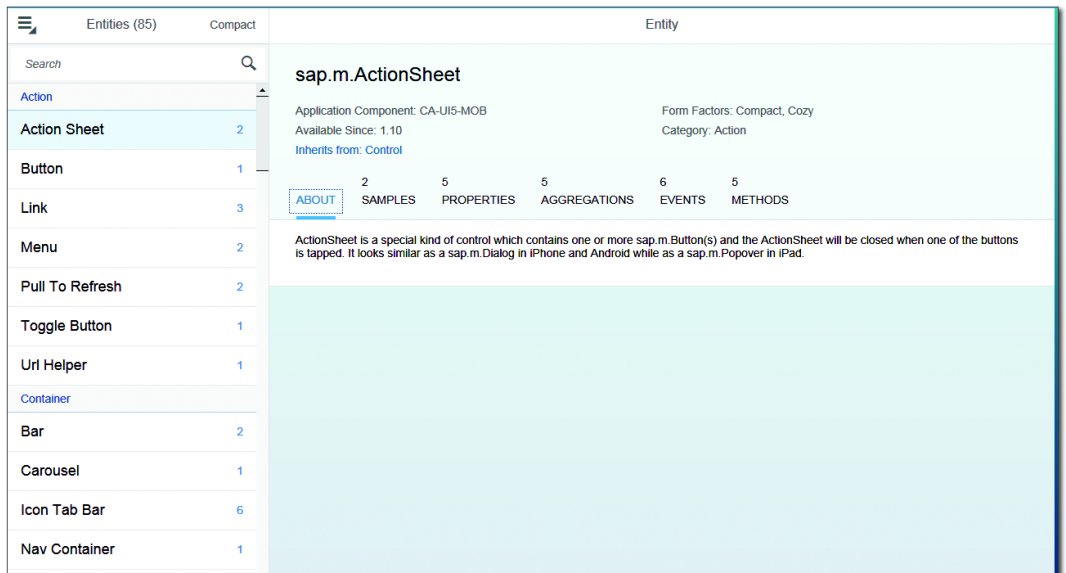


Figure 13.40 List of UI Entities with Example Code

First, create a new XML file called `ActionSheet.fragment.xml`, as shown in Listing 13.24.

```
<?xml version="1.0" encoding="UTF-8"?>
<core:FragmentDefinition
  xmlns="sap.m"
  xmlns:core="sap.ui.core">
  <ActionSheet
    title="Choose Monster Action"
    showCancelButton="true"
    placement="Bottom">
    <buttons>
    <Button text="Create" icon="sap-icon://lab" />
    <Button text="Give Award" icon="sap-icon://competitor" />
    <Button text="Remove Head" icon="sap-icon://wrench" />
    <Button text="Award Degree" icon="sap-icon://study-leave" />
    <Button text="Inject" icon="sap-icon://syringe" />
    <Button text="Other" />
    </buttons>
  </ActionSheet>
</core:FragmentDefinition>
```

Listing 13.24 `ActionSheet.fragment.xml`

This code defines a list of push buttons with icons that will appear in a context menu when the user clicks the ACTION button. Now, change the code near the top of your view. You had a button that did nothing; change it to one that invokes an action from the controller (Listing 13.25).

```
<headerContent>
  <Button
    text="Open Action Sheet"
    press="handleOpen" />
</headerContent>
<subHeader>
```

Listing 13.25 Coding a Button that Triggers an Action when You Press It

Finally, add a function to your controller to respond to the user pressing the button and to call up your fragment of possible monster actions (Listing 13.26).

```
handleOpen : function (oEvent) {
  var oButton = oEvent.getSource();

  // create action sheet only once
  if (!this._actionSheet) {
    this._actionSheet = sap.ui.xmlfragment(
```

```

        "monster.ActionSheet.", this);
    this.getView().addDependent(this._actionSheet);
}

    this._actionSheet.openBy(oButton);
}

```

Listing 13.26 Call Up Fragment

This is just cutting and pasting for the main part, though naturally you always have to change *something*, and the more experienced you get, the more you will change things.

The assorted buttons that appear when you call up the action sheet do nothing. In real life, the fragment would have `press` definitions like the one in Listing 13.25 to call functions in the controller based on what icon the user clicked (e.g., for the REMOVE HEAD button, you'd say `press = handle head removed` and then code a function in the controller called `handle head removed`).

Adding that code and changing a few things took maybe six or seven minutes, and when you test your application again you can see that a dropdown menu has appeared at the top, as shown in Figure 13.41.

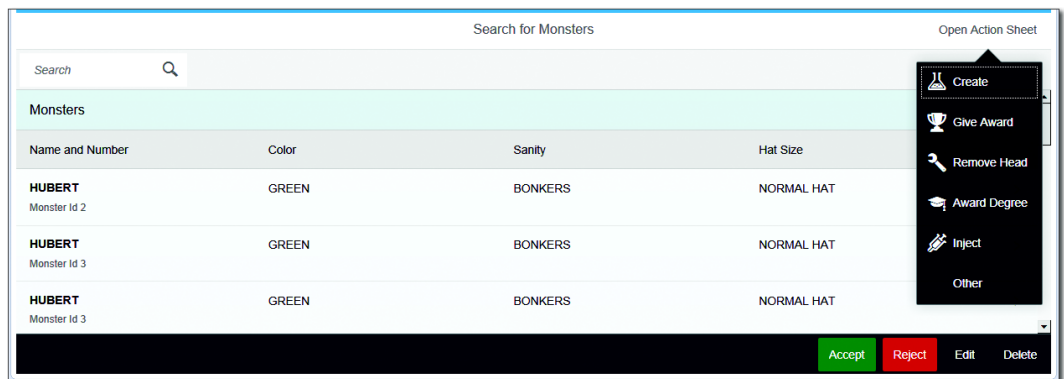


Figure 13.41 Adding a Dropdown Menu

As you can imagine, this makes adding features to an application a breeze, but I would urge you to actually look at any code you copy and try to figure out what's actually going on. If you do, it will make troubleshooting so much easier, and before you know it you'll have your head round XML and JavaScript syntax.

13.6 Importing SAPUI5 Applications to SAP ERP

It's possible that throughout this chapter you've been horrified by the idea of developing in Eclipse, where the files are stored on your local computer, as opposed to in the SAP ABAP repository, which is where everything you normally develop lives.

The good news is that it is in fact possible to move your new SAPUI5 application inside the SAP system, where it can join the repository object club. Technically, how this is done is that the SAPUI5 runtime leverages the BSP framework to store the SAPUI5 artifacts on the ABAP server. BSP applications look a bit like HTML pages with other programming languages inside them, and by usage of the mime repository and a special ICF handler, the SAPUI5 application can be stored in such a way that it looks like a BSP application, even if it isn't one really—like those insects that disguise themselves as bees to get inside the hive and steal the honey.

This section explains how to store SAPUI5 applications in SAP ERP for releases both prior to and after 7.31. The section ends by discussing how to test SAPUI5 applications from within SAP ERP.

13.6.1 Storing the Application in Releases Lower than 7.31



For releases below 7.31, you cannot directly move an SAPUI5 application definition into the SAP ERP system. However, the good news is that SAP Note 1793771 provides a workaround, giving you an ABAP report to upload and download files. After about SP 3 for release 7.02, this report is available as standard in the SAP ERP system and is called `/UI5/UI5_REPOSITORY_LOAD` (the irony here is that this standard SAP report supplied to upload SAPUI5 applications contains an awful lot of `WRITE` statements).

To get started with the process of moving your SAPUI5 application definition into the ABAP system, select your main monster node in Eclipse, right-click on it, choose `EXPORT`, select `FILE SYSTEM`, and the whole application structure will be saved in a folder on your local machine. Next, go to SE38 and run program `/UI5/UI5_REPOSITORY_LOAD`; there is no transaction code.

The screen shown in Figure 13.42 appears. Be careful here; the length of the input box might make you think you can have a really long name, but only 12 charac-

ters are allowed. If you put a long name in, then you will only get an error at the end of the process.

SAPUI5 Repository Load: Load SAPUI5 App from and to local File System

Up- and Download your SAPUI5 Application

Use this report to upload a SAPUI5 application to the SAPUI5 ABAP Repository.
 Use it to download a SAPUI5 application to the local file system.
 Use it to delete an existing SAPUI5 application from the repository.

Name of SAPUI5 Application

Upload
 Download
 Delete

* Specify SAPUI5 Application Name *

Remarks:

- > In case you would like to transport:
 Prepare an ABAP workbench request and note down its id for later entry.
- > Your default code page is "Cp1252"
- > Files to be ignored may be specified in file '!UI5RepositoryIgnore!'
- > Additional text files may be specified via '!UI5RepositoryTextFiles!'
- > Additional binary files may be specified via '!UI5RepositoryBinaryFiles!'
- > Each line in the files above specifies a file name pattern:
 Either in form a sub string or as a regular expression.
 Example: '^:~/\build[/\]?\$' starting with '^' and ending with '\$'.
- > For operation you may need authorization for the following auth. objects:
 S_DEVELOP, S_ICF_ADM, S_TCODE, S_TRANSPRT, S_CTS_ADMI and S_CTS_SADM
 Check the security guide.

Figure 13.42 Uploading SAPUI5 Application to the ABAP Repository: Part 1

Click EXECUTE, and the screen shown in Figure 13.43 appears. As you can see, this shows you all the things the system is about to do. Click [CLICK HERE TO UPLOAD](#).

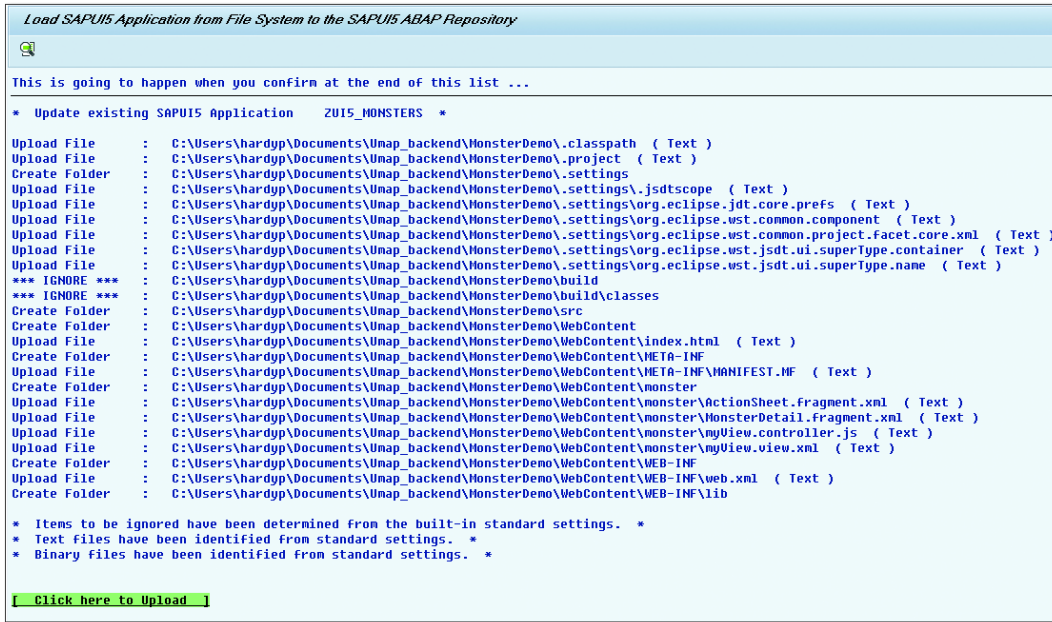


Figure 13.43 Uploading SAPUI5 Application to the ABAP Repository: Part 2

Another box appears, asking for a name for the BSP application and a development package (or you could choose \$TMP for a local object). After this point, your application will be snugly inside the SAP ERP system.

13.6.2 Storing the Application in Releases 7.31 and Above

Release 7.31 SP 4 and above can import SAPUI5 application definitions directly from Eclipse into the SAP ERP system. In Eclipse, select your main Monster Monitor node, right-click, and choose TEAM • SHARE PROJECT. The reason for this naming is that the component that moves things from Eclipse to SAP is called a team provider, on the grounds that if something is in Eclipse it's on your local machine and only available to you, but if it's inside the SAP system, it's available to anyone on the programming team.

The first pop-up is shown in Figure 13.44. There are a few options; choose SAPUI5 ABAP REPOSITORY. In the next box, you will be asked to browse to the SAP logon pad on your local machine to get the details of the SAP system you want to connect to. All the logon pad details will be copied over, and then you

will be asked for the client and user name and password (unless single sign-on is active). If all goes well, then the connection is established. Remember, if your backend SAP system is not on a high enough level, then you will get error messages at this point about not supporting Eclipse tools or not supporting team providers.

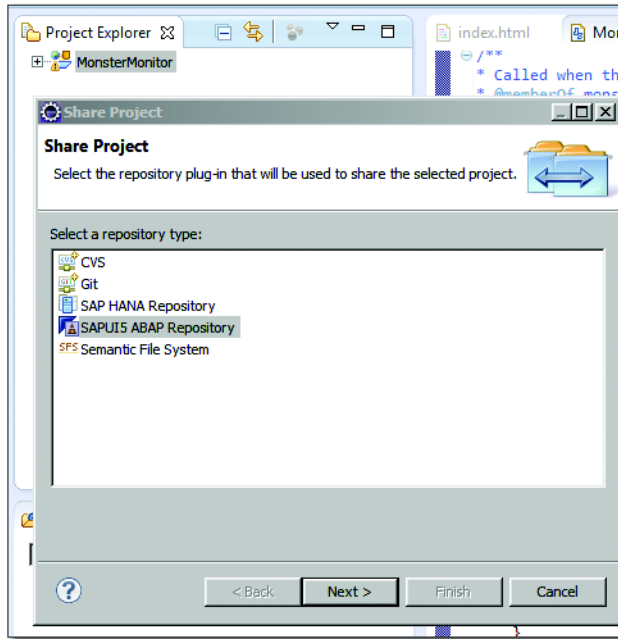


Figure 13.44 Moving an SAPUI5 Application into SAP

If there are no errors, then the next box will talk about creating a new BSP application and what package you want to put it in. If this is not a local object, then you will be asked for a transport request, as might be expected. After that, the dialog is finished, and you would expect that you've moved your application into ABAP—but no, you haven't. All you've done is to express your strong desire to do so.

You have to select the main Monster Monitor node in Eclipse once more, and this time choose **TEAM • SUBMIT**. Then, you'll see a box with a big list of files to move into the SAP system. You may wonder why this is a two-step process. The whole idea of Eclipse is strange to an ABAP developer; you can have several developers changing the same program on their local machines and then they both try to

import their new versions into the SAP system. You just don't have that problem inside ABAP, but tools like Eclipse have to live with this on a daily basis and as such provide a means to deal with such conflicts.

13.6.3 Testing the SAPUI5 Application from within SAP ERP

Once the SAPUI5 application has arrived inside your SAP system disguised as a BSP, you can test it in the exact same way you would test any other BSP. You can use SE80 and choose BSP APPLICATION and then hunt for the name of the application you just created, and there it will be (Figure 13.45).

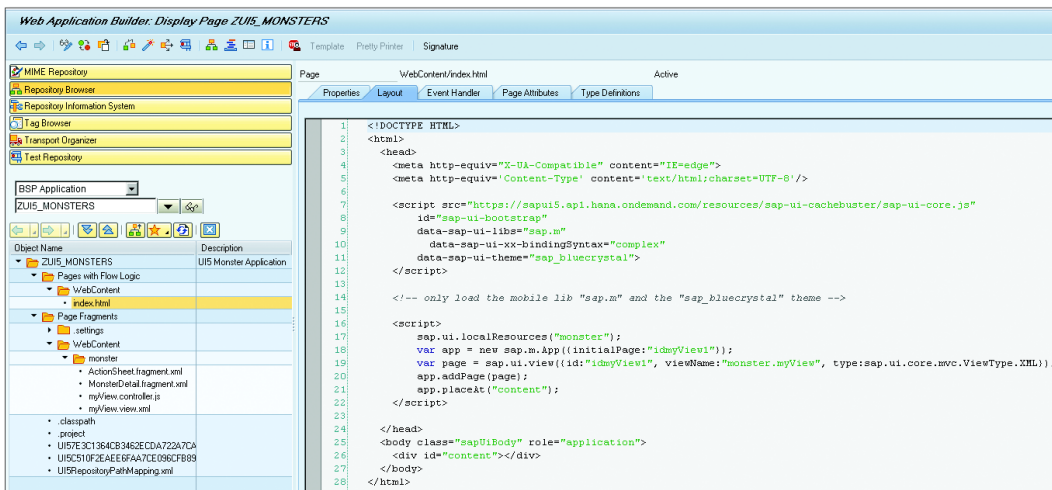


Figure 13.45 SAPUI5 Application as BSP Page

The eagle-eyed amongst you may have noticed that after the application code was uploaded, it was necessary to make a change to the Index.html file definition inside SAP. Specifically, you have to replace the bootstrap section with the code in Listing 13.27.

```

<script src="https://sapui5.api.hana.ondemand.com/resources/sap-ui-
cachebuster/sap-ui-core.js"
  id="sap-ui-bootstrap"
  data-sap-ui-libs="sap.m"
  data-sap-ui-xx-bindingSyntax="complex"
  data-sap-ui-theme="sap_bluecrystal">
</script>

```

Listing 13.27 Changes Needed to the Bootstrap File Once It Lives inside SAP

The code in Listing 13.27 will not work if you put it in the index file inside Eclipse; conversely, the index file inside SAP ERP will not work without the code being changed. This is because both environments need to get their resources in slightly different ways when starting the application.

After transferring the data to the SAP ERP system, you'll also see the name of your application in Transaction SICF, under the path `sap/bc/ui5_ui5`. This means that if you type "`<myhost>/<myport>/sap/bc/ui5_ui5/monster_monitor`" into a web browser (note that Internet Explorer doesn't seem to work on versions 9 and below; Chrome works just fine), then the application will start.

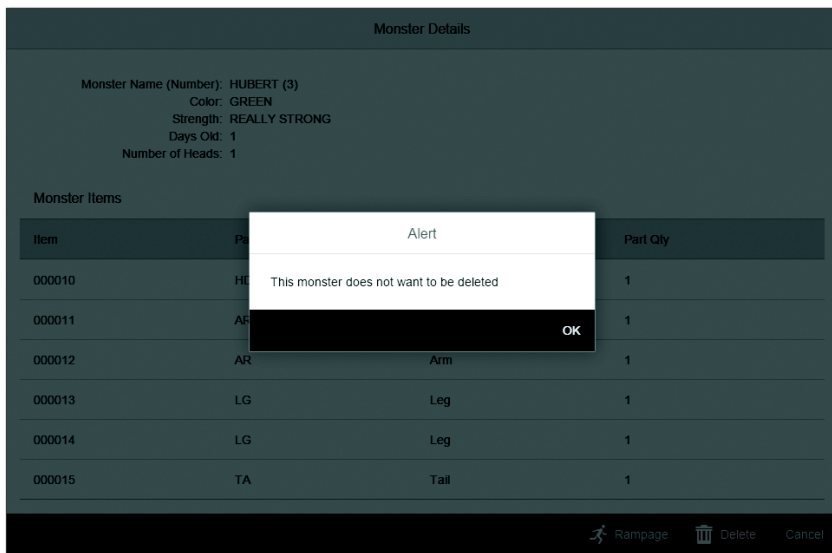


Figure 13.46 Testing the BSP SAPUI5 Application

You could also select the top node of your BSP application in SE80 and click the TEST button. In Figure 13.46, you can see the result when selecting a monster and clicking the DELETE button. The error message is shown, which did not work when you tested this inside Eclipse.

Clearly, the benefit of having the SAPUI5 definition within the ABAP repository is that you can have this in the same transport as the pure ABAP objects that make up your SAP Gateway service. This way the model, view, and controller can all be transported from development to test together, even if they're in different programming languages.

13.7 SAPUI5 vs. SAP Fiori

If you've heard about SAPUI5, it's likely that you've also heard about SAP Fiori. Although it seems like the terms are sometimes used interchangeably, SAPUI5 is not the same thing as SAP Fiori. SAP Fiori refers to a set of applications that were built using SAPUI5 and that can be customized using SAPUI5. (Kind of like how all the standard SAP transactions are written in ABAP, and you can write your own Z ones.)

The idea here is the diametric opposite of how SAP has approached building UIs in the past. Traditional SAP screens are viewed using your desktop, and they are crawling with fields and tabs and context menus, enough to satisfy every company in every industry in every country of the world. Each individual company would only use a few of those fields and tabs, however. The applications using those screens have an enormous amount of functionality (i.e., you can do about 34 different things from one screen).

An SAP Fiori screen, in contrast, is designed to be viewed on a smartphone or some such and performs one task only, showing you the absolute bare minimum of screens and buttons; one screen and one button is the ideal (e.g., to approve a leave request).

SAP initially came out with about 25 SAP Fiori apps to handle common SAP tasks, and this was going to change the way people thought about using SAP. Sadly, initially SAP was going to charge for this in addition to the normal license fee, when their competitors were giving out UI improvements for free as part of standard maintenance. In early 2014, common sense prevailed, and since then SAP Fiori and another product called Screen Personas have been bundled into the annual maintenance fee with no extra charge. More SAP Fiori applications are being released over time in "waves."

SAP Screen Personas

Since about 1998, there has been an SAP partner company called Synactive, which made a product called GuiXT, of which the core product was free, with separately licensed add-ons. What this did was enable you to move fields around the screen on standard SAP transactions, rename fields to suit the language of your company, add pictures, and merge screens to some extent. The underlying DYNPRO application was unchanged; GuiXT was just a layer on top.

Synactive must have been very sad when SAP came out with its own equivalent, called SAP Screen Personas, which enables you to do the same sort of thing but to a much greater extent and in a web browser. You have a tool to drag and drop elements of SAP screens around, and then you can write code (JavaScript) to add whatever functionality you feel like (e.g., changing a three-screen transaction into one screen). Using SAP Screen Personas used to cost you money, but it's now as free as GuiXT has always been.

13.8 Summary

This chapter introduced you to the latest player in SAP's UI strategy, SAPUI5, and showed you how to go about building applications with it. This concludes the book's coverage of UI technologies; the last part of the book will focus on the database aspect of ABAP programming—ending with everyone's favorite lady, SAP HANA.

Recommended Reading

- ▶ Getting Started with SAPUI5: <http://scn.sap.com/docs/DOC-48897>
- ▶ End-to-End How-To Guide: Building SAPUI5 Applications on SAP NetWeaver AS ABAP 7.31 Consuming Gateway OData Services: <http://scn.sap.com/docs/DOC-33792> (Betram Ganz and Bernhard Siewert)
- ▶ JavaScript for ABAP Developers: <http://scn.sap.com/community/abap/blog/2014/02/18/javascript-for-abap-developers> (Chris Whealy)
- ▶ *Getting Started with SAPUI5* (Antolovic, SAP PRESS, 2014)

PART IV
Database Layer

The worst part of holding the memories is not the pain. It's the loneliness of it. Memories need to be shared.

—Lois Lowry, *The Giver*

14 Shared Memory

Unless you've been hiding under a very large rock for several years, you may have noticed SAP pushing the in-memory database SAP HANA with all its might nonstop. (Arguably, the high spot was when Hasso Plattner published a video online of himself interviewing himself: <https://www.youtube.com/watch?v=UrJ8mJ3cxvs>.)

However, for those SAP customers who don't yet have SAP HANA in their systems (which is about 99% of them at the moment) but do have a gigantic amount of memory floating about unused, maybe the technology known as shared memory is worth a bit of investigation. I heard a story about a company that went from a 32-bit architecture to a 64-bit one. The CIO said to a programmer, "I have a vast amount of memory now sitting there doing nothing; are you going to tell me what I can use it for?" Yes, the programmer could. The irony of course is that the use of shared memory actually reduces overall memory consumption.

More specifically, there are two problem areas shared memory is designed to address: database access and memory usage.

Problems with database access arise when a vast number of users execute the exact same database query simultaneously, clogging up the system. If you look at Transaction SM66, then you can see what database tables are being accessed by programs that users are running at any given instant. If the screen is full of red blobs against users running transactions in dialog mode, then that is a Bad Thing, because it means you have assorted users staring at hourglasses on the screen and cursing SAP in general and the IT department (i.e., you) specifically for stopping them from getting their jobs done. If the SM66 list gets long enough, then all the work processes fill up, and then no one can do anything until a new process becomes available. (This would be even worse if the user had a customer on the

phone, because then you have *two* people waiting for the hourglass to finish. This must be what happens whenever I phone a bank or similar institution.)

Problems with memory usage arise when there is a duplication of the exact same data in a large number of individual user sessions. For example, you may have been called on to investigate runtime errors in which the user's session terminates with a `ROLL_NO_AREA` dump or one of its close friends. If you read the text in the short dump, then you will see that an internal table has reached a very large number of rows and then tried to append another row, and the SAP system says "this is the straw that breaks the camel's back" and declares that there is just not enough room in memory to store such a huge amount of data in an individual user session. If an individual user session is hogging a huge amount of memory, then it can affect the system as whole. If the system is to remain healthy enough for everything to go swimmingly for everybody, then programmers should try to minimize the amount of data held in memory at any one time. This is why SAP has tools such as the Memory Inspector, which watches out for memory leaks in long-running transactions like `CICO`, in which if you are not careful the amount of memory used just goes up and up until the out-of-memory dump occurs.

At the end of the day, both problems with database access and memory usage cause runtime errors or short dumps, which not only are annoying to the individual users but also often interfere with the running of the system as a whole, which is annoying to every single user. Shared memory can address both issues, and this chapter will show you how.

Section 14.1 gives a general introduction to the problem the shared memory framework was created to address and how the framework goes about solving those problems. Section 14.2 talks about the mechanics of how shared memory objects are created and used in the ABAP language, which is quite complicated. Once you've got the data in shared memory, you have to make sure that it is exactly the same as the equivalent data in the database; this process is the subject of Section 14.3. Finally, Section 14.4 provides some tips for troubleshooting common problems.

14.1 The Promises of Shared Memory

As noted in the chapter introduction, the goal of shared memory is to solve two big problems: database access and memory usage. This section will outline, briefly and on a high level, how shared memory goes about this.

14.1.1 Database Access

As is most likely common knowledge, reading from data in memory is much, much faster than reading that same information from the database. SAP gets around this in many ways, the most common of which is allowing transparent tables to be buffered wholly or partially, so that after the database has been started up only the first access of a data record will need to go to the database; subsequent accesses will be read from the buffer.

The best solution, of course, would be to store the entirety of the database in memory. However, unless you have SAP HANA, this is only practical for master data tables that are small and hardly ever change—for example, configuration tables and some pricing tables. (A possible solution is to start hacking standard SAP tables and buffering them, but that doesn't seem like a wise thing to do; otherwise, SAP would have done this itself years ago.)

The shared memory framework solves this problem by reading the database once and holding the data in one place that everyone can get to. Say that you have a business object—a monster, for example—that takes quite an effort to set up, reading from six different database tables and then having to use `READ_TEXT` to get some texts. A lot of database access occurs every time an instance of the monster is created, and yet the underlying data changes very rarely, if at all. Moreover, you often have many users executing the same transaction for the same monster and thus all performing the same database query all at the same time. By storing that database data in a single place in memory, anyone who was interested could access that data without a database read at all.

14.1.2 Memory Usage

The second promise of shared memory is that instead of 100 user sessions running the same program and storing the same data in internal tables you just have one internal table that is read by them all. You not only eliminate the database access, but you also only occupy enough memory for one internal table instead of 100 identical ones.

It is worth noting that you only get this benefit if the consuming program does not need every record in the shared memory object. If the calling program needs every record, then it still has to take a local copy. Therefore, shared memory has more benefit when the calling programs generally only need single records.

14.2 Creating and Using Shared Memory Objects

The best way to describe how to use shared memory is to use an example. This section will first present a basic example and then will present more complicated scenarios.

In Chapter 9, you learned that objects in SAP often resemble Russian nesting dolls in that an observer only sees the outermost doll, but inside are a series of smaller dolls. This matches the OO principle of having each class do one thing and one thing only.

Therefore, an application built using OO principles that uses shared memory will have absolutely no idea that it does in fact use shared memory, because it does not need to know or care about where the data it uses comes from. If you start from the point of view of the human sitting in front of the computer or mobile device looking at the screen and pressing buttons, you will pass through the following layers:

- ▶ The outermost layer of the software is the *user interface* (the view), which of course is the software from the end user's point of view, because that's all that he sees. That is why ugly gray screens don't go over so well, even if those ugly gray screens are rendered in a web browser.
- ▶ The view only knows about the next layer down, which is the *controller*. The controller acts as a mediator, passing user input down to the business application (the model) and receiving back instructions from the model as to how the state of affairs has changed in response to that user input so that the controller can tell the view to let the user know.
- ▶ The model (one level down from the controller) knows about the controller it talks to but has no further knowledge of the outside world. It is solely concerned with business logic. That business logic almost always needs to access information from the database, but the model does not know or care if the information is actually coming from the database or an external system or from the Planet Bong in the Constellation of Horseradish Sauce. The model delegates the task of getting the data it needs to yet another class, whose sole task is to retrieve data.
- ▶ The next layer is the *persistency layer*, and usually the class that performs this layer's function simply performs a big bunch of database reads in its various methods. It could of course use RFC calls to other systems or call SAP Process

Integration (PI) proxies to achieve the same end. In this example, the data is going to come from shared memory. It turns out that this will require two further layers, both of which are known to the persistency layer: the broker class and the root class.

- ▶ The *broker* class is a class that handles shared memory areas for the given set of data. The persistency class uses this shared memory handling class to access the innermost layer. The broker class is a generated class, and you don't do any custom coding inside it.
- ▶ Finally, the innermost layer is the so-called *root class*, and this is where you do our own coding for where in the database you want to get the data from initially. You also can add a whole bunch of whatever methods you see fit to use in order to control what data gets stored and how it gets changed and accessed.

Broker Class: What's in a Name?

They say that the Eskimos have 200 words for snow; in the same way, SAP often uses multiple terms to mean the same thing. In the realm of shared memory, the terms "broker class," "area class," and "handle" are used interchangeably. To avoid confusion, this chapter sticks to "broker class"; it's arguable the best of the three phrases, because it describes the task of the class in human terms—that is, to act as an intermediary, a broker, between the calling program and the root class.

The user interface and controller layers were, of course, the focus of earlier chapters in the book. In this section, we concentrate on the layers from the persistency layer inward. First of all, you will create the root class, in which all the action (your own code) happens. Then, you will learn about creating the broker class, which serves as the interface between the custom code in the root class and the persistency layer in any ABAP programs that want to take advantage of shared memory. Finally, you'll discover how to call the broker class from a persistency layer class.

14.2.1 Creating the Root Class

This type of class does not want to change the world; it only has one job, which is to store data and then let anyone in the system who is authorized to do so access that data. Just as Spiderman can do whatever a spider can, this root class can do whatever a normal class can, which makes it a lot cleverer than a database table. It cannot swing from a web, but it can perform complicated authorization

checks on requests, perform calculations (return calculated values), and store really complicated data structures. In other words, the root class is a normal class, and thus you create it via SE24 as you would any other class. The important part, as shown in Figure 14.1, is to select the SHARED MEMORY-ENABLED checkbox.

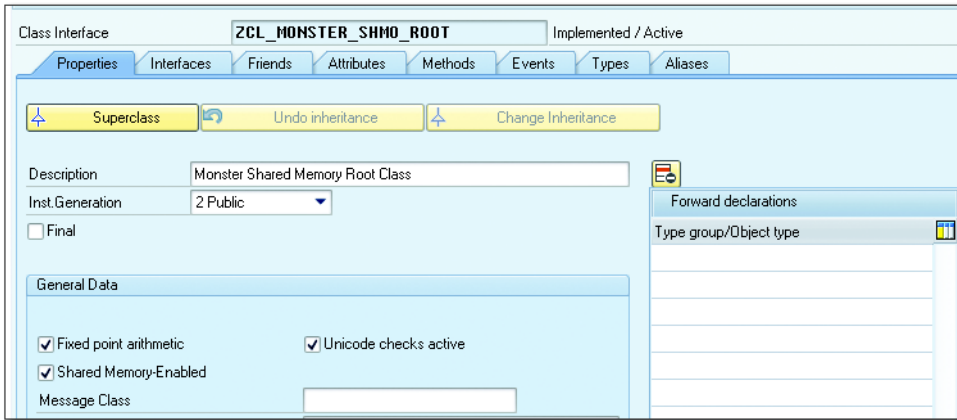


Figure 14.1 Shared Memory-Enabled Check Box

In this example, you will buffer the entire monster table from the running monster example. That may not be the most realistic example the world has ever seen, but it will demonstrate the basic principles. The idea is that you will store the entire monster table in shared memory and then have an access method to get one record at a time. Initially, this will seem pointless, because you can slay this vampire of a problem by setting the table to be buffered. However, later you will move on to using the same technique for more complicated data structures that you cannot buffer, so you need to start with the basics.

First, you need to add an attribute to store a copy of the database table. Up until SAP ERP 6.0, you would have had to create a DDIC table type with a line type that matched the table definition, such as table type VBAP_T. However, that sort of thing just bloats the data dictionary; nowadays, you can define your own type by clicking on the TYPES tab, as shown in Figure 14.2.

After you click the DIRECT TYPE ENTRY button, you are taken to the source code-based view of SE24 (one of the best improvements to SE24 ever), where you can directly type the table just as you would inside an ABAP program. The code for this is shown in Listing 14.1.

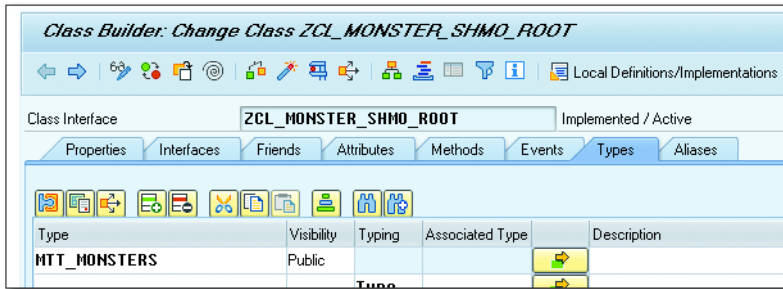


Figure 14.2 Declaring a Custom Table Type

```

PUBLIC SECTION.
*** public components of class ZCL_MONSTER_SHMO_ROOT
*** do not include other source files here!!!

INTERFACES if_shm_build_instance .

TYPES: mtt_monsters TYPE HASHED TABLE OF ztvc_monsters
      WITH UNIQUE KEY monster_number.

DATA: mt_monsters TYPE mtt_monsters.
    
```

Listing 14.1 Data Declarations in the Root Class

You have a place for the data to live, but how does the data get there in the first place? Surely, you have to go to the database at least once? Yes you do, and in order to enable this you need your root class to implement the `IF_SHM_BUILD_INSTANCE` interface, which has the `BUILD` method. `BUILD` is called automatically by the SAP system, but you cannot code the method until you've created the broker class, so Section 14.2.2 will return to this topic.

In the meantime, you can define your own `LOAD` method (Figure 14.3) to transfer the entire database table to an equivalent internal table.

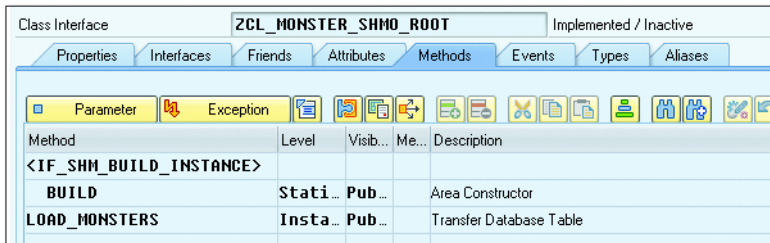


Figure 14.3 Defining the LOAD Method

There are no parameters for this method and nothing too mysterious inside it, as you can see in Listing 14.2.

```
METHOD load_monsters.
```

```
    SELECT * FROM ztvc_monsters
    INTO CORRESPONDING FIELDS OF TABLE mt_monsters.
```

```
ENDMETHOD.
```

Listing 14.2 LOAD Method

There is only one (meaningful) unique key field in table ZTVC_MONSTERS—namely, `monster_number`—so make it a hashed table. If it were a standard table, you would sort it after getting the data, because the only way to make your code future-proof is to not rely on the underlying database to sort your data for you before sending it back to the application server. (I've seen a database send back the data sorted one way in development and the opposite way around in test.)

Now, you need a method to get the data back. One (bad) way to do this is to allow calling programs to retrieve the whole table, because sometimes that's what you think you need. However, that would defeat the whole point of the exercise, which is to prevent the whole table being stored in two places in memory at once. Instead, you want the calling program to ask for one record at a time. To enable this, there is an optional `monster_number` parameter.

Figure 14.4 allows the whole table to be retrieved because the table is small. You can see that the `id_monster` importing parameter is optional; if it isn't supplied, then the whole table is returned. If this were a huge table, you should seriously consider making single-record access the only way to go, just like setting single-record buffering in a database table. You have to ask yourself: Do the calling programs really ever need the whole table at once? If not, make the importing parameter with the record key (monster number, in this case) mandatory.

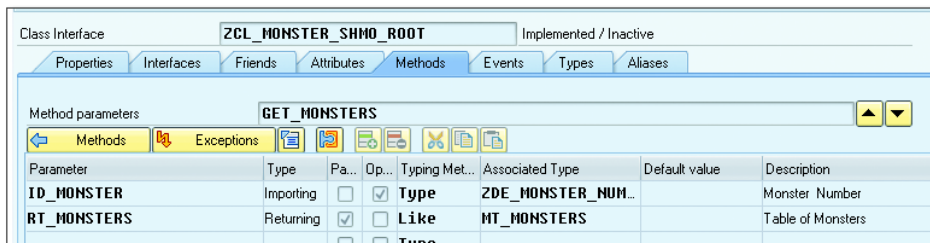


Figure 14.4 Get Monster Method: Parameters

```

METHOD get_monsters.
* Local Variables
DATA: ls_monsters LIKE LINE OF mt_monsters.

IF id_monster IS SUPPLIED.

    READ TABLE mt_monsters INTO ls_monsters WITH TABLE KEY
    monster_number = id_monster.

    INSERT ls_monsters INTO TABLE rt_monsters.

ELSE.
    rt_monsters[] = mt_monsters[].
ENDIF.

ENDMETHOD.

```

Listing 14.3 Root Class Monster Retrieval Method

The code in Listing 14.3, in which you fill up a table with the required monster details, is a very simplistic example, but you could opt for something more complicated with no effort at all. You could have lots of different retrieval methods; for example, there could be one that returned all the monsters of a certain color. In fact, because this is all standard ABAP, it's not difficult to imagine a situation in which the root class gets information from a dozen database tables and three external systems during its `LOAD` method and in the retrieval method combines all that assorted data into one return structure based on the most complicated business logic ever invented.

For example, often you end up with an internal table in your program that comprises the contents of one database table plus some text descriptions of sales offices (from customizing table TVKBT) and the like, maybe some address details from ADRC, and some calculated fields. Perhaps also one of the fields in the internal table contains a reference to an object. Normally, you do a database read on a single table, and then loop through the resulting internal table, filling in the missing bits. That might be quite a lot of work, and in some circumstances it may make sense to simply go through this process once and store the result in shared memory.

Remember, though, that retrieving, manipulating, and sending back data is all that the root class is supposed to do. This is the single responsibility principle. There is nothing in the code that is done differently because an instance of the class is going to live in shared memory. If you had an existing class that just dealt

with data, then you could just select the `SHARED MEMORY-ENABLED` checkbox, and that would be that. The root class doesn't need to know anything about the mechanics of shared memory; that's the job of the broker class.

14.2.2 Generating the Broker Class

The broker class will be automatically generated by the system, and you do not add any custom code of your own. In order to generate the broker class, call Transaction SHMA, which is used to create the shared memory area and generate the broker class in one step. Because this is going to generate a class, name the shared memory area as if it were a class. In this case, call it `ZCL_MONSTER_SHMO`, where "SHMO" stands for "shared memory object."

The screenshot shows the configuration window for a Shared Memory Area (SHMA). The window is titled "Area" and contains the following fields and options:

- Name:** `ZCL_MONSTER_SHMO`
- Description:** `Monster Data Shared Memory Area`
- Attributes:**
 - Root Class:** `ZCL_MONSTER_SHMO_ROOT`
 - Client-Specific Area
 - Aut. Area Creation
 - Transactional Area
- Fixed Properties:**
 - Binding:** `109200001 Application Server`
 - With Versioning
- Dynamic Properties:**
 - Constructor Class:** `ZCL_MONSTER_SHMO_ROOT`
 - Displacem. Type:** `1208200200 Displacement Not Possible`
- Runtime Setting (Default):**
 - Area Structure:** `1107197102 Autostart for Read Request and Every Invalidation`
 - Version Size:** `UNLIMITED No Limit` (kByte)
 - No. of Versions:** `UNLIMITED No Limit`
 - Lifetime:** `1125600000 No Entry` (Minutes)

Figure 14.5 Transaction SHMA: Creating a Shared Memory Area

The SHMA screen is shown in Figure 14.5; the following paragraphs work down the page, telling you the options to select and why.

First, tell the shared memory area what root class it will be responsible for. Then, select the `AUT. AREA CREATION` checkbox; this means that if the broker class gets a request for information and no instance of the root class exists in shared memory, then such an instance is automatically created in the shared memory area and the `LOAD` method is called.

Select the `TRANSACTIONAL AREA` checkbox, because in this example the root class represents the whole database table. Your programs have to ensure that when the database changes, the shared object changes accordingly. (You'll learn more about how this works later on in Section 14.3 and will also look at a shared memory area without this checkbox selected.) Generally, all shared memory areas correspond to database data to a greater or lesser extent, but some cannot be transactional, for reasons that will become clear.

Leave the `BINDING` option at the default value of `APPLICATION SERVER` and keep the `WITH VERSIONING` box checked. The `CONSTRUCTOR CLASS` should be the same as the root class—here you are specifying the class that is going to fill the shared memory root class with data. Although you could in theory have a separate class that fills up the root class, it makes life a lot easier to just have one.

Moving down the screen, the displacement option (`DISPLACEM. TYPE`) is obsolete on 64-bit systems; the idea of this option was that if the entire shared memory in the system were running out, you could displace rarely used shared memory areas to make room for the popular ones.

Recall that you specified the constructor class as the root class; this is the class that has its `BUILD` method called when automatically creating an instance during the very first read request. To enable that `BUILD` method to be called automatically, set the `AREA STRUCTURE` field to `AUTO START FOR READ REQUEST AND EVERY INVALIDATION`.

Note that there are several options when choosing the value for the `AREA STRUCTURE` field. The default is `NO AUTOSTART`, which is bad, because if a read request comes in and no instance of the root class exists, then a short dump occurs. The second option is `AUTOSTART FOR READ REQUEST`, which creates an instance the first time a read request comes in and calls the `BUILD` method. The one chosen in this example creates an instance when the first read request comes in and also if the current existing instance of the root class becomes invalidated—that is, no longer

agrees with the database contents. (The term *invalidate* here has the same meaning as when used in relation to SAP database tables that use buffering.)

You will have seen by now that there are three settings scattered throughout the screen that all relate filling up the root class with data the first time it is accessed: AUT. AREA CREATION, CONSTRUCTOR CLASS, and AREA STRUCTURE. If these three fields are not consistent, then you get all sorts of error messages.

The LIFETIME option at the bottom enables you to expire certain shared memory areas if they have not been read for (as an example) 180 minutes, thus freeing up memory. You usually would not want to do this, unless maybe you have objects which were only accessed at certain times of the year (which does not seem very likely).

When you are happy with the settings you have defined, click the SAVE button to create the shared memory area. You will see messages popping up at the bottom of the screen telling you that a class with the same name as the area is being generated: ZCL_MONSTER_SHMO, in this case.

As mentioned earlier, you do not have to add any of your own custom coding into the broker class. Each broker class is generated automatically with 99% identical code; some small parts change based on the choices you made during the SHMA transactions. For example, in the ATTRIBUTES tab, as shown in Figure 14.6, you can see that some of the attributes (such as the name of the root class) have been set to the values you just entered.

Attribute	Level	Visib...	Rea...	Typing	Associated Type	Description	Initial value
PROPERTIES	Instance	Public	<input checked="" type="checkbox"/>	Type	SHM_PROPERTIES	Area Attributes	
INST_NAME	Instance	Public	<input checked="" type="checkbox"/>	Type	SHM_INST_NAME	Name of a Shared Object Instanc...	
CLIENT	Instance	Public	<input checked="" type="checkbox"/>	Type	MANDT	Client	
ATTACH_MODE_WAIT...	Constant	Prote...	<input type="checkbox"/>	Type	SHM_ATTACH_MODE	(Internal) Repeat Lock Request	1302197003
_LOCK	Instance	Prote...	<input type="checkbox"/>	Type	%_C_POINTER	(Internal) ModelLockRefId	
INST_TRACE_SERVICE	Instance	Prote...	<input type="checkbox"/>	Type Ref	IF_SHM_TRACE	(Internal) Reference to Trace Cla...	
INST_TRACE_ACTIVE	Instance	Prote...	<input type="checkbox"/>	Type	ABAP_BOOL	(Internal) Flag: Trace Active?	ABAP_FALSE
AREA_NAME	Constant	Public	<input type="checkbox"/>	Type	SHM_AREA_NAME	Name of an Area Class	'ZCL_MONSTER_SHMO'
ROOT	Instance	Public	<input checked="" type="checkbox"/>	Type Ref	ZCL_MONSTER_SHMO_ROOT	SHM: Model of a Data Class	
VERSION	Constant	Private	<input type="checkbox"/>	Type	I	(internal)	20
_TRACE_SERVICE	Static Attr.	Private	<input type="checkbox"/>	Type Ref	IF_SHM_TRACE	(Internal) Reference to Trace Cla...	
_TRACE_ACTIVE	Static Attr.	Private	<input type="checkbox"/>	Type	ABAP_BOOL	(Internal) Flag: Trace Active?	ABAP_FALSE
_TRANSACTIONAL	Constant	Private	<input type="checkbox"/>	Type	ABAP_BOOL		ABAP_TRUE
_CLIENT_DEPENDENT	Constant	Private	<input type="checkbox"/>	Type	ABAP_BOOL		ABAP_FALSE
_LIFE_CONTEXT	Constant	Private	<input type="checkbox"/>	Type	SHM_LIFE_CONTEXT	Lifetime of an Area (Constants in ..._CL_SHM_AREA=>LIFE_CONTEXT_APPSERVER	

Figure 14.6 Broker Class Attributes

The broker class has a number of public static methods, which can be seen in Figure 14.7. The best way to understand these is to see an example of how to use them in ABAP programs; conveniently, this is the subject of Section 14.2.3.

Method	Level	Visib...	Me...	Description
CLASS_CONSTRUCTOR	Stati...	Pub...		CLASS_CONSTRUCTOR
GET_GENERATOR_VERSION	Stati...	Pub...		Query Generator Version
ATTACH_FOR_READ	Stati...	Pub...		Request a Read Lock
ATTACH_FOR_WRITE	Stati...	Pub...		Request a Write Lock
ATTACH_FOR_UPDATE	Stati...	Pub...		Request a Change Lock
DETACH_AREA	Stati...	Pub...		Release all locks on all instances
INVALIDATE_INSTANCE	Stati...	Pub...		Active version of one instance will be set to obsolete
INVALIDATE_AREA	Stati...	Pub...		Active versions of all instances will be set to obsolete
PROPAGATE_INSTANCE	Stati...	Pub...		OBSOLETE: Use INVALIDATE_INSTANCE
PROPAGATE_AREA	Stati...	Pub...		OBSOLETE: Use INVALIDATE_AREA
FREE_INSTANCE	Stati...	Pub...		Deletion of an Instance
FREE_AREA	Stati...	Pub...		Delete all instances
GET_INSTANCE_INFOS	Stati...	Pub...		Returns the names of all instances
BUILD	Stati...	Pub...		Direct Call of Area Constructor
SET_ROOT	Insta...	Pub...		Sets Root Objects

Figure 14.7 Broker Class: Methods

14.2.3 Using Shared Memory Objects in ABAP Programs

Now that you have your root class and broker class set up, you can start using them in your lovely ABAP programs. As might be imagined, there are two things you want to do with persistent data (remember: persistent data isn't data that keeps on and on at you and never gives up; it's data that is permanently stored in the database). You want to read such data and to create or change such data (a write request).

The following subsections look at the two types of database requests you make—read and write—one at a time, with examples of how to code such requests.

Coding a Read Request

In Listing 14.4, you see the code for getting the data out of shared memory; it's very simple—just a few lines. First, you need to create an instance of your broker

class, and once you have that you can access the root class within the broker to get your monster data.

```

DATA: lt_monsters TYPE STANDARD TABLE OF ztvc_monsters.
      lo_monster  TYPE REF TO zcl_monster_shmo.

DO 10 TIMES.
  TRY.
    "We create an instance of the broker class which
    "contains within it a reference to the instance of
    "the root class which lives in shared memory
    "A (non exclusive) read lock is also set
    lo_monster = zcl_monster_shmo=>attach_for_read( ).

    "The root class contains our custom methods
    lt_monsters[] = lo_monster->root->get_monsters( ).

    "Release read lock ; ending the transaction would
    "have the same result
    lo_monster->detach( ).
    EXIT. "From Do-End-Do

  CATCH cx_shm_no_active_version.
    "The root instance has not been created - the BUILD
    "method will be called asynchronously
    "If after ten seconds the object has not initialised itself we
    "give up
    WAIT UP TO 1 SECONDS. "Takes a while to load up
    CONTINUE. "With do-end-do

  CATCH cx_shm_inconsistent "Root class is out of date
        cx_shm_exclusive_lock_active "Someone trying to change
        cx_shm_change_lock_active "Someone trying to change data
        cx_shm_read_lock_active. "Amazingly, this can be an error
    EXIT. "From do-end-do, we will use a real database read

ENDTRY.
ENDDO.

```

Listing 14.4 A Shared Memory Read Request

As is normal in programming, you can see that the bulk of the code in Listing 14.4 is in regard to error handling. There are two possible error situations:

- ▶ Another user may have the instance of the root class locked in such a way that you're not allowed access; someone may be in the middle of changing the data, for example. In this case, it's better to read the database directly. The calling class is in the persistency layer, so reading from the database is not outside its job description.

- ▶ This is the very first read request, and the root class instance has not yet been created. Due to the way you set up the shared memory area, the system throws the exception and at the same time calls a task asynchronously to call the `BUILD` method to fill the root class with data.

The problem in the second error situation (the first read request) is that the calling program does not know when the `BUILD` task is finished. Listing 14.4 used the standard technique of using the `WAIT UP TO X SECONDS` command to give the `BUILD` method time to finish. This solution isn't ideal, however; it's not guaranteed to work and can cause short dumps. (In Section 14.4.2, you'll see an alternative way to deal with this.)

Coding a Write Request

Now, it's time to actually code the `BUILD` method in your example class; as a by-product, you'll find out how to create the instance of the root class. First, fill up the `LOAD` method, which is a public instance method, to retrieve the data from the database. The code for this is shown in Listing 14.5.

```
METHOD load_monsters.

    SELECT *
      FROM ztvc_monsters
      INTO CORRESPONDING FIELDS OF TABLE mt_monsters.

ENDMETHOD.
```

Listing 14.5 Fill LOAD Method

Next is something of actual interest: the public static `BUILD` method. The code for this is shown in Listing 14.6.

```
METHOD if_shm_build_instance~build.
* Local Variables
DATA: lo_broker    TYPE REF TO zcl_monster_shmo,
      lo_root      TYPE REF TO zcl_monster_shmo_root,
      lo_exception TYPE REF TO cx_root.

"Open the shared memory area for write access
TRY.
    lo_broker = zcl_monster_shmo=>attach_for_write( ).
    "CX_SHM_ERROR is a superclass of all the
    "lock error exception classes
CATCH cx_shm_error INTO lo_exception.
```

```

        RAISE EXCEPTION TYPE cx_shm_build_failed
        EXPORTING
            previous = lo_exception.
    ENDTRY.

    "Read the database and fill up the internal table
    TRY.
        "The broker class is sometimes called a "handle"
        "as in 'Handle and Gretel'
        CREATE OBJECT lo_root AREA HANDLE lo_broker.
        lo_broker->set_root( lo_root ).
        lo_root->load_monsters( ).
        lo_broker->detach_commit( ).
    CATCH cx_shm_error INTO lo_exception.
        RAISE EXCEPTION TYPE cx_shm_build_failed
        EXPORTING
            previous = lo_exception.
    ENDTRY.

    IF invocation_mode = cl_shm_area=>invocation_mode_auto_build.
        CALL FUNCTION 'DB_COMMIT'.
    ENDIF.

ENDMETHOD. "Build

```

Listing 14.6 Coding the BUILD Method

Most of the code in Listing 14.6 is straightforward enough, but notice how the instance of the root class is created. Instead of a normal `CREATE OBJECT` statement, which creates the instance in the memory of the running program, there is the `AREA HANDLE` addition, which means that instead the instance is created in shared memory and stays there even after the calling program ends.

The other point to note is the `DETACH_COMMIT` method. For a transactional shared memory area (such as the one in this example), the shared memory area only gets updated once a `COMMIT WORK` occurs, which makes updating the shared memory part of a logical unit of work fall under the “all shall change or none shall change” principle. For a nontransactional area, `DETACH_COMMIT` updates the shared memory at once.

In any event, as can be seen during a build, a `DB_COMMIT` function is called to ensure that both types of shared memory area are updated. The build is running in a separate session, so the `COMMIT` can have no side effects.

14.3 Updating the Database and Shared Memory Together

Now, your shared memory objects are defined and some custom programs are using them. This makes it vitally important that the data in shared memory is exactly the same as the equivalent data in the database. With standard database tables, the SAP runtime system takes care of this for you, but with shared memory you have to look after it yourself.

When it comes to updating the database, it's more than likely that you will be using a custom program, and in that case you could code a write request (refer back to Section 14.2.3) in your custom program to make sure that the shared memory was updated at the same time as the database.

But what if you had a Z table that was updated via SM30? The SAP system generates the code for such table maintenance, so how is the shared memory going to get updated? It turns out that the coding is identical; it's just where the coding lives that differs, and in this section you'll find out what you need to do in the SM30 situation.

In SE11 for table ZTVC_MONSTERS (in change mode), go to the Table Maintenance Generator, and follow the menu path ENVIRONMENT • MODIFICATION • EVENTS. You'll see a message informing you that this is a standard SAP table—but it isn't, so you don't need to worry about that message. You'll then see a screen in which you can add code routines for the various events in the lifetime of an SM30-based transaction—for example, before database save, after database save, after checking that data has changed, and a whole bunch of others. In this case, you want to call some code just after the data is saved, so add an entry as shown in Figure 14.8.

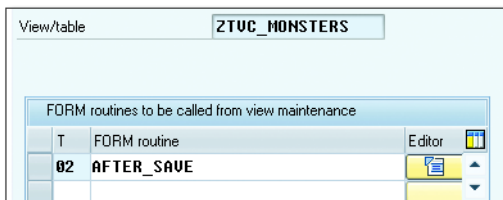


Figure 14.8 Table Maintenance Event

Go into the editor and write some code, as shown in Listing 14.7.

```

FORM after_save.

DATA: lo_monsters TYPE REF TO zcl_monster_shmo.

TRY.
    lo_monsters = zcl_monster_shmo=>attach_for_read( ).
    CATCH cx_shm_attach_error.
        "If the area does not exist, no need to invalidate it
        RETURN.
    ENDTRY.

CHECK lo_monsters IS BOUND.

TRY.
lo_monsters->invalidate_area( affect_server = cl_shm_area=>
affect_all_servers ).
    CATCH cx_shm_parameter_error
            cx_sy_ref_is_initial
            cx_dynamic_check.
        RETURN.
    ENDTRY.

TRY.
    lo_monsters->detach( ).
    CATCH cx_shm_parameter_error
            cx_sy_ref_is_initial
            cx_dynamic_check.
        RETURN.
    ENDTRY.

ENDFORM.                    "after_save

```

Listing 14.7 Coding an SM30 Event

The code in Listing 14.7 is called after the user has just updated table ZTVC_MONSTERS via SM30. In such a case, you want to make sure that the data in shared memory also gets updated, so invalidate the shared memory area on every application server. You can only do this for shared memory areas that have been defined as TRANSACTIONAL. Other shared memory areas can only invalidate the data on the current application server.

Earlier, you set up the shared memory area such that upon invalidation a new area is automatically created and the BUILD method is called again to get the new data from the database. This will happen once on every application server; that is, if you have six application servers, then each server will have its instance of the root class rebuilt. This is a bit of a sledgehammer approach, but there is no way to

send data from a shared memory object on one application server to its counterparts on the other application servers, so a global invalidation ensures that the shared memory always matches the database.

Having to reload the data on every application server may seem wasteful, but the type of information you would want to put into shared memory is by definition the type of information that would hardly ever get changed.

14.4 Troubleshooting

There are two very common problems that arise when working with shared memory. This section explains how to fix both of them.

14.4.1 Data Inconsistency between Application Servers

If a database read on one application server returned a different result than a database read on another application server, then no one would have faith in the computer system anymore, and you would have a riot on your hands. With standard SAP buffered tables, the runtime system takes care of making sure that the data buffer is the same on all application servers. With shared memory, you have to take care of this task yourself.

Section 14.3 talked about how to keep the shared memory area in line with the database and mentioned the way that transactional areas handle this by affecting all servers. However, many shared memory areas are not defined as transactional, as they do not mirror an actual database table in its entirety. This section talks about what to do when you do not have a transactional area.

In the following example, there is a shared memory area to buffer a subset of `MARA` records. A user changes the material master record. Even if you have a routine in a user exit that occurs on save of Transaction `MM02` and you send out an invalidate command, only the shared memory instance on the current application server gets refreshed, because this is not a transactional area. This means that if you have six application servers, then only one has the current data in shared memory, and the other five have incorrect data. That is a Bad Thing, because anything reading the shared memory object on the other servers thereafter comes back with the wrong result, which could have disastrous results.

The ability of a program to only talk to the shared memory root instance on the current application server has led to shared memory being perceived to be impossible to use. False! You can get around the problem.

First, create a method in your root class that takes in a material as the input and then deletes that record from the buffer table. Naturally, that method only affects the application server on which the program is running. Then, wrap the code inside a remote-enabled function module; the code looks like Listing 14.8.

```

FUNCTION zmm_shm_invalidate_material.
*-----
***"Local Interface:
*  IMPORTING
*    VALUE(ID_MATERIAL) TYPE  MARA-MATNR
*-----
DATA: lo_material TYPE REF TO zcl_material_master_shmo.

DO 2 TIMES.
  TRY.
    "We only want one user updating the shared memory at once
    "If two try at once, only let the first one do so
    lo_material = zcl_material_master_shmo=>attach_for_update( ).

    lo_material->root->delete_material( id_matnr = id_material ).

    lo_material->detach_commit( ).

  RETURN. "All done

  CATCH cx_shm_inconsistent
        cx_shm_no_active_version
        cx_shm_exclusive_lock_active
        cx_shm_version_limit_exceeded
        cx_shm_change_lock_active "that is what I am looking for
        cx_shm_parameter_error
        cx_shm_pending_lock_removed.
    WAIT UP TO 5 SECONDS.
    CONTINUE. "Try Again - in case two users trying to
              "update at same time
    CATCH cx_shm_out_of_memory.
    "We are in trouble. The only way to be sure the data is updated
    "is to free up the entire material shared memory
    zcl_material_master_shmo=>invalidate_area( ).
    RETURN.

  ENDTRY.
ENDDO.

* If we get here we could not delete the record that has been

```

```

* updated, so we have to use the sledgehammer approach and
* delete everything. That's better than the slightest risk of
* having wrong data in shared memory
  zcl_material_master_shmo=>invalidate_area( ).

```

```
ENDFUNCTION.
```

Listing 14.8 Remote-Enabled Function for Shared Memory

Finally, add some code known as a business transaction event to the user exit; this code fires whenever a material is saved. In this user exit, the added code looks like Listing 14.9.

```

DATA: lt_application_servers TYPE STANDARD TABLE OF rfchosts
      WITH HEADER LINE,
      ld_message TYPE char80 ##needed.

"If certain data gets changed, notify shared memory of this fact
CHECK upd_mara IS NOT INITIAL.

* Obtain a list of all active servers in the system.
CALL FUNCTION 'RFC_GET_LOCAL_DESTINATIONS'
  TABLES
    localdest      = lt_application_servers[]
  EXCEPTIONS
    not_available = 1
    error_message = 2
    OTHERS       = 3.

IF sy-subrc <> 0.
  "At least remove the material on this server
  CALL FUNCTION 'ZMM_SHM_INVALIDATE_MATERIAL'
    EXPORTING
      id_material = i_mara_new-matnr.
  RETURN.
ENDIF.

* Make sure material is invalidated on all application servers
* Call the function in a new task to avoid an implicit database
* COMMIT
LOOP AT lt_application_servers.
  CALL FUNCTION 'ZMM_SHM_INVALIDATE_MATERIAL'
    STARTING NEW TASK i_mara_new-matnr
    DESTINATION lt_application_servers-rfcdest
    EXPORTING
      id_material      = i_mara_new-matnr
  EXCEPTIONS
    system_failure     = 16 MESSAGE ld_message
    communication_failure = 17 MESSAGE ld_message.

```

```

IF sy-subrc <> 0.
  "At least remove the material on this server
  CALL FUNCTION 'ZMM_SHM_INVALIDATE_MATERIAL'
    EXPORTING
      id_material = i_mara_new-matnr.
  CONTINUE.
ENDIF.

ENDLOOP. "Application Servers

```

Listing 14.9 User Exit on Material Save

The code in Listing 14.9 finds all the application servers on the system and then loops through them, each time performing an RFC to tell the application server to update the shared memory object. If the program gets into trouble deleting a single record, then it invalidates the whole area. In this way, the program can ensure that when a record is changed, the change is propagated to all application servers. Once again, this code does not directly change the data; it only deletes a record so that it gets reloaded on the next read request.

Note

Cynics will still say that you're going through a lot of effort to manually replicate what SAP does in the normal course of events for buffered tables, which is why buffering tables is still the preferred option if possible. Nonetheless, there are some situations where table buffering just does not cut the mustard and shared memory neatly fills that gap. (You will also notice that if there is some sort of massive system failure, then the propagation will not work—but in such an event, this would be the least of your worries.)

You may recall that in Section 2.10 of Chapter 2 we talked about a new feature in ABAP 7.4 called ABAP Messaging Channels, which sends messages between application servers. This facility would seem ideally suited to shared objects, because the object instances on the application servers could send messages to each other whenever one of them has its data changed. You would most likely still want to delete the records in the other servers.

14.4.2 Short Dumps

When you use a new technology and get unexplained short dumps all over the place, it can put you off in a hurry. All of the examples in this chapter have been following the SAP party line, and when they cannot instantly attach a shared

memory root object for read or update they have used `WAIT UP TO X SECONDS` and tried again. When you start to use this process in real life, you'll run into two problems:

- ▶ No matter how many seconds you wait and no matter how many loops you do, you cannot be sure that the build will be complete, the lock will be released, or whatever. The only way to ensure this is an infinite loop, and that is not a viable workaround. Pausing for 10 seconds is not much fun for the user either, especially if he has a customer on the phone.
- ▶ The second and far more serious problem is that you cannot be sure where the request will be called from. There are certain places in which `WAIT UP TO X SECONDS` causes a short dump; an update task springs to mind. The update task performs a read request, which is usually harmless, and a short dump ensues.

Taking all this into consideration, when a user performs the first read request and the instance does not yet exist, it's better for that user to lose the battle than for the whole system to lose the war. To make this happen, you can change the code from Listing 14.4 to that shown in Listing 14.10.

```
CATCH cx_shm_no_active_version      "Root class being built
      cx_shm_inconsistent         "Root class is out of date
      cx_shm_exclusive_lock_active "Someone trying to change data
      cx_shm_change_lock_active   "Someone trying to change the data
      cx_shm_read_lock_active.    "Amazingly, this can be an error
      EXIT. "From do-end-do, we will use a real database read
ENDTRY.
```

Listing 14.10 Amended Read Lock Error Handling

Back in Chapter 7, you read about exception handling, and Listing 14.10 is an example of a controlled panic response to an unexpected situation. You expect the data to be in shared memory, but in the exceptional case in which it is not you try an alternative strategy—that is, getting the data directly from the database—and then resume where you left off.

Warning: Houston, We Have a Problem

One problem—without a solution, but worth noting: Whereas the SAP runtime system looks after memory application for buffered tables, shared memory is on its own. If a program calls a shared memory object and the shared memory is full, then an exception is raised. In your own programs, you can add exception handling to prevent a short dump, but a lot of the standard SAP programs have no such exception handling. Therefore, if you aren't careful, then you can start to cause standard SAP code that uses

shared memory to fail and cause short dumps if your custom shared memory areas get over a certain size. You need to do a lot of testing in QA to make sure that the total shared memory usage (which can be seen via monitoring Transaction SHMM) does not come anywhere near the maximum size allocated by your Basis department.

If you have a shared memory object being used as a buffer, then one possible control mechanism is to have an attribute that controls the maximum size of the buffer—an attribute readout of a Z table that can be changed directly in production to reduce the buffer size in the event of an emergency.

14.5 Summary

In this chapter, you learned about the shared memory framework by describing the problems it's meant to solve, how to configure it, and some troubleshooting tips.

At some point in the future—still many years away for most—the whole shared objects framework will become redundant for those SAP customers who move to using the SAP HANA database, which is the topic of the final chapter.

Recommended Reading

How to Work with ABAP Shared Memory Objects: <http://scn.sap.com/docs/DOC-10461>
(Trond Stroemme)

When you innovate, you've got to be prepared for everyone telling you you're nuts.

—Larry Ellison

When SAP, and, specifically Hasso Plattner, said they're going to build this in-memory database and compete with Oracle, I said, God, get me the name of that pharmacist, they must be on drugs.

—Larry Ellison

15 ABAP Programming for SAP HANA

Here's a fascinating fact that ABAP developers may not know and may not like the sound of: In the same way that an engine is not much use without a car to surround it, a database is no good without some sort of development environment to query the data and do something with it. In the SAP world, you're used to having the user interface and the application server where the business logic is processed all in the same place, and this all talks to the database—a database that can be swapped out if you feel like it.

Enter SAP HANA. (Because SAP has been talking about nothing else but SAP HANA for several years now, you're probably sick to death of hearing about it, but no self-respecting SAP-based book could hold its head up in public without a chapter devoted to the subject.) SAP HANA is different in that it's designed to be able to have the business logic run within it, and it also contains a standalone development environment (XS) using JavaScript and SQLScript as programming languages. Thus, it's possible for a company that doesn't use SAP to build fully functional applications via SAP HANA, and ABAP doesn't even get a look in.

This is what scares some ABAP programmers: They get a confused message that SAP HANA is the future, and SAP HANA doesn't need ABAP, and they start looking for the nearest bridge to jump off. However, just like in 2001 when it was said that Java would replace ABAP, rumors of the death of ABAP are greatly exaggerated (to paraphrase Mark Twain). Indeed, in the rest of this chapter you'll see

how ABAP can be used to work with SAP HANA to make the most out of the new opportunities provided.

The biggest change SAP HANA brings to ABAP programming is the way that you access data in database tables. SAP HANA brings the new concept of code pushdown, which is moving business logic out of the application server into the database. This concept scares a lot of ABAP programmers, because they fear it will impact their jobs. Because this concept has the most significant impact on ABAP developers, the majority of the chapter is devoted to it. Section 15.1 introduces you to the basic concept of code pushdown. Then, Section 15.2 and Section 15.3 explains the two ways code pushdown is accomplished. The first is top-down development (the process of creating artifacts in ABAP and replicating them in SAP HANA), which is fairly clearly the approach all ABAP programmers will use eventually when using ABAP to control what SAP HANA does. The second is bottom-up development (the process of creating artifacts in SAP HANA and calling them from ABAP), which is already being discouraged by SAP—but this chapter will look at it anyway, because some companies are not yet on a high enough level to do the top-down approach. The discussion of code pushdown is rounded off in Section 15.4, which explains how to identify code that should be pushed down to the database.

Finally, the chapter concludes with Section 15.5, which offers a discussion of a number of other ways in which SAP HANA will affect your ABAP programming: changes not related to the idea of code pushdown but important nonetheless.

SAP HANA Being Fast Is Not the Point

When it comes to the benefits of SAP HANA, it could be said that SAP marketing has been prioritizing the wrong message. What you hear about is the fact that queries run 10,000 times faster on SAP HANA—hence the “10,000 club” that SAP talks about in conferences referring to companies that have such benefits. But is that what SAP HANA is all about?

The truth is that if a report takes ages to execute, then people are used to setting up such reports to run overnight and logging on the next morning to have a look at them. With SAP HANA, they run in seconds, so you could run them all throughout the day if you wanted—but are people going to do this? Is just doing the exact same thing as before really fast actually a benefit? SAP would say there has to be some sort of behavior change as well: companies, by which we mean the individuals in the companies, have to take advantage of the new capabilities by actually doing something they were not doing before.

15.1 Introduction to Code Pushdown

One strange thing in life is that often you get told for years on end “This is the way to do (whatever it is), this is the only way, and nothing else will work—so do it this way,” and just when you’re comfortable with this, everything changes overnight and the very same experts start telling you to do the same thing in a different way.

When it comes to SAP HANA, the change is as follows: For years, computer programming textbooks have gone to great lengths to convince everyone to separate the code that does database access and the code that does the business logic. You have all done this, usually by having a separate class for the database accesses. Now, with SAP HANA, because of the gigantic performance benefits, you’re being encouraged to look for situations in which you can push down (or you could say “outsource”) large chunks of business logic to the database layer.

In the ABAP world, you’ve been used to doing everything in the application layer. There are two ways to move some of this logic to the database layer: stored procedures and views. In the case of stored procedures, you write some code that is going to be executed directly in the database layer; this is called a stored procedure. This could be just a SQL statement to select some data (though that would be a bit pointless), but it’s usually one or more database queries sprinkled with some business logic on top. The idea, as always, is to reduce the amount of data transferred between the database and the application server—something you always strive for, because this is one of the golden rules. In the case of the views, you create a view, a concept that should not be that alien to ABAP programmers.

ABAP vs. SAP HANA Database Views

You know what a database view is in the ABAP world: You define a view in SE11 in which you nominate one or more database tables, declare all the fields you are interested in, the join conditions where appropriate, and maybe some selection conditions (e.g., a view on VBAK in which you are only interested in quotations). A database view in SAP HANA is pretty much exactly the same, except that the options you have within SAP HANA are far greater.

In case you’re already getting nervous about delegating some ABAP tasks to the database, you should realize that, most likely, you are already doing just that to some extent. For example, if you want to know how many purple monsters with

green hats there are, and that's all you want to know (just the total number), you most likely don't read all the data into an internal table, see how big that table is, and then throw away the internal table. Instead, you would say `SELECT COUNT(*)` and thus tell the database to go and count the records and transfer back the total (i.e., minimize data transfer between the database and the server). If you think of stored procedures and views as just fancier versions of the aggregation features you use every day, then suddenly they no longer seem as strange and scary as space aliens from the Planet Bong.

OpenSQL in ABAP

The OpenSQL capabilities in ABAP have been vastly extended in release 7.4. If you make use of some of the fancy new features when doing a `SELECT` statement in ABAP, then you are effect already doing code pushdown—that is, outsourcing some work to the database generally with the aim of reducing the amount of data sent back. Therefore, normally you only need to define special views and stored procedures when the extended OpenSQL capabilities just cannot cut the mustard.

What's more, this is something of a race; extra capabilities are being introduced into SAP HANA views and stored procedures with each support pack. But at the same time, the capabilities of OpenSQL in ABAP are also increasing with each support pack

15.2 Top-Down Development

With top-down development (Figure 15.1), ABAP is the king of the castle. All artifacts (views and stored procedures) are created in the ABAP system and then replicated inside SAP HANA. This means you can keep using ABAP for all development and use the normal change and transport system to move your changes through the system landscape.

In order to develop SAP HANA artifacts from within the ABAP development environment, you have to build and call two new types of ABAP repository object: core data services (CDS) views and ABAP Managed Database Procedure (AMDP) objects. This section introduces you to both objects. You'll learn how to build and call the CDS view object, which gets created in the database via a so-called DDL in the ABAP system. Then you'll implement the code in an ABAP method using the SQLScript language so that that method generates and calls an equivalent stored procedure, the AMDP object, in the SAP HANA database at runtime.

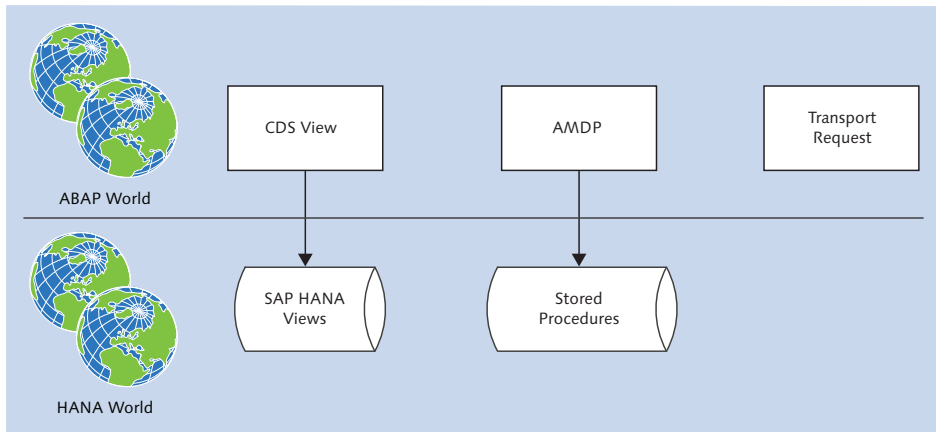


Figure 15.1 Top-Down Development

CDS View vs. AMDP

Initially, the difference between a CDS view and an AMDP may seem puzzling. In a nutshell, when coding an AMDP you use SQLScript, whereas the language used when creating a CDS view is DDL, which is an enhancement of SQL. This means that you can do a whole lot more in a stored procedure than in a CDS view, because SQLScript has built-in calculation engine functions as well as interim steps, local variables, and conditional logic, such as IF/ELSE. A CDS view is just a big SELECT statement on steroids; it can only return one result (a row or table of data). A stored procedure is much more like a method and can return as many result parameters as it feels like.

15.2.1 Building and Calling CDS Views

If you look up “CDS” on the Internet you will find that it stands for “Country Dance and Song.” In SAP HANA terms, though, it stands for “core data services,” which doesn’t seem to be quite as exciting as shouting “Yeehaw!” while riding a bull. But once you’ve seen what is possible inside a CDS view, maybe you’ll be just as excited. It certainly beats the SE11 equivalent into a cocked hat.

CDS views could be described as “database views—the next generation.” SAP claims that whereas a traditional database view is just a linkage of one or more tables, a CDS view is a fully fledged data model, which, in addition to having extra features that SE11-defined views do not, can be used even by applications outside of the SAP domain. You’ll recall the idea is that SAP HANA is also a development platform and that non-SAP systems should be able to use artifacts created in SAP

HANA, like CDS views. You can create a CDS view in SAP HANA alone, but in this example you're going to create one from the ABAP environment.

Say that you want to create a view in SE11 to get all the details of monsters who own certain pets. The first thing you would do is to create a database view in SE11 and then choose what tables to join—for example, connect the monster items to the main monster table (which the system achieves by comparing foreign key relationships and thus doing a join on `MONSTER_NUMBER`) and then make a big list of fields from one or more of the linked database tables. Next, you might add some hard-coded selection criteria (e.g., you're only interested in green monsters or monsters with sanity less than 10%).

However, if you wanted to do an outer join between the monster table and the monster pets table and also on `MONSTER_NUMBER` to get the details of what pets, if any, the monster owns, then you'd be out of luck. You can't do outer joins in an SE11 database view. You have to create the view linking the first two tables, and then do the outer join in your program via a `SELECT` statement between the newly created view and the monster per table. View creation in SE11 works well, but like all form-based development in ABAP, it's somewhat clunky and involves clicking a lot of buttons, choosing options on pop-up screens, and moving from tab to tab.

Luckily, thanks to SAP HANA, there is now a new ABAP repository object that you can create that not only is less clunky but also gives you more options. The name of the ABAP repository object is DDL (Data Definition Language), and it looks like a tiny program consisting of one complicated `SELECT` statement. When the DDL object is generated in the ABAP system, an equivalent view is generated within SAP HANA, called a CDS view. (The cut-down version of a CDS view can be seen in SE11 but not changed.) What's even better is that this technique works with any database, not just SAP HANA.

To recap, instead of the form-based approach of view creation using SE11, you're specifying everything using code.

The steps in building and calling a CDS view can be broken down as follows:

1. Creating a DDL in Eclipse.
2. Coding the DDL.
3. Generating the CDS view in the database from the DDL.

Each step is discussed in more detail next.

Creating a DDL in Eclipse

The first step in coding a CDS view is to create a new DDL in your lovely ABAP in Eclipse development environment (you cannot create this in SE80). You do this by selecting the package in which you want to create the view and following the menu path **NEW • OTHER ABAP REPOSITORY OBJECT • DICTIONARY • DDL SOURCE** (see Figure 15.2).

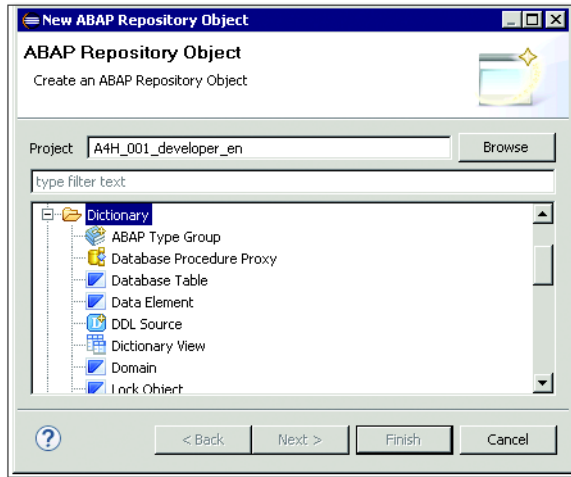


Figure 15.2 Creating a DDL Source: Part 1

After selecting the DDL source option and clicking **NEXT**, the screen shown in Figure 15.3 appears.

Enter a name, such as `ZCDS_MONSTERS`, and a description, and click **NEXT**. If this is not a local object, then you'll be asked for a transport request. After you choose one, click **FINISH**. For a local object, just click **FINISH**. A blank screen appears with the CDS view name at the top.

Why the Blank Screen?

As has been stressed often before, in Eclipse there is no concept of form-based programming like you see with transactions such as SE37 and SE24, in which you have to put parameters and method names in certain boxes. In Eclipse, everything is based on source code, and naturally CDS views are no exception. You have to code everything yourself, and that's a Good Thing.

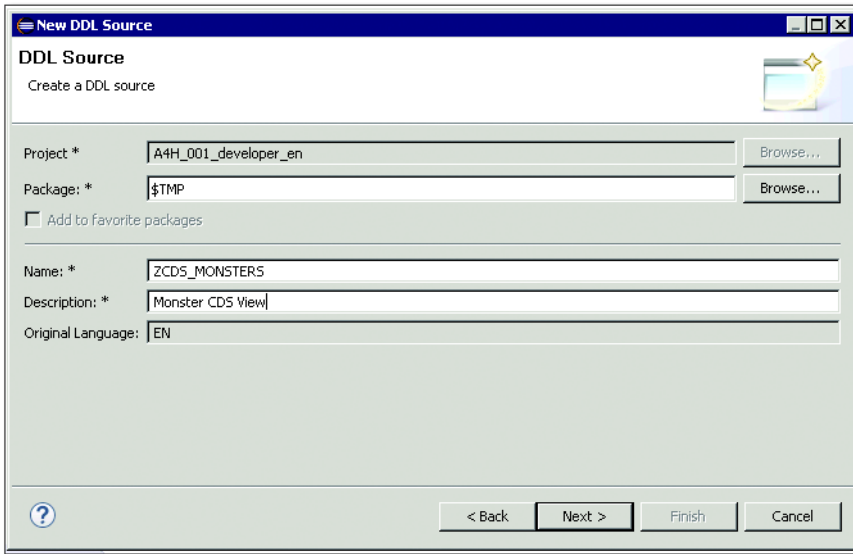


Figure 15.3 Creating a DDL Source: Part 2

Your CDS view already has a name, but having one name is just not good enough; you need two names. The new name is for the SQL view that is going to be created in the dictionary (the one you will be able to look at in SE11), and the name you already have is a name for the CDS view entity, which is viewed and changed via Eclipse. You could name both the SQL view and the CDS view the same, but...don't. They're different things, so the name should reflect that. Call the SQL view `ZV_MONSTERS` and the entity `ZCDS_MONSTERS`.

The former can be seen in SE11; the latter is the one you should refer to in `SELECT` statements in your ABAP programs. (You could refer to the former in your programs as well, but that would be naughty.)

Coding the DDL

Now, it's time to code the DDL, which will eventually create the CDS view when complete. As mentioned earlier, this will consist of one big `SELECT` statement. You have to start the coding with one or more so-called annotations, which can be thought of as header settings (or metadata, as in "Mutants in Metadata One") for the view. An annotation is a line of code that starts with an `@` sign.

You always need at least one annotation to define the SQL view name, so the first line will be as follows:

```
@AbapCatalog.sqlViewName: 'ZMONSTERS'
```

Then, if you so desire (and only if the underlying SAP table or tables that you'll be accessing allow buffering), you can set some buffering settings for the view on the second line:

```
@AbapCatalog.buffering.status #ACTIVE
@AbapCatalog.buffering.type #SINGLE
```

Buffering

In the same way that a DDIC table can be set to have single-record buffering, full buffering, or whatever, a CDS view can be set the same (you could of course always do this in SE11 for traditional views). You would think such things no longer make a lot of sense in an SAP HANA database, because everything is in memory anyway, but actually buffering is just as important as ever. The rule of thumb is that if you want to get some stored data, then here are your options, from fastest to slowest:

- ▶ Read an internal table
- ▶ Read from the table buffer
- ▶ Read from the DB cache
- ▶ Read from an SAP HANA database
- ▶ Read from a boring, square, not-in-memory database

Therefore, if you have a small database table that doesn't change often, buffering it in the world of SAP HANA is a good thing to do, just as it has always been.

Then, in the same way that a FORM routine starts with FORM and a method starts with METHOD, the CDS view starts with the words `define view`, as in:

```
define view zcds_monsters as
```

After all this, the first few lines of the view will then look like the code shown in Listing 15.1.

```
@AbapCatalog.sqlViewName: 'ZMONSTERS'
@AbapCatalog.buffering.status #ACTIVE
@AbapCatalog.buffering.type #SINGLE
define view zcds_monsters as
```

Listing 15.1 Header Settings of a CDS View

From a technical standpoint, a CDS view entity is defined as a `SELECT` from one or more data sources (with a data source being an SAP table or view or another CDS

view), so naturally the next part of the coding is the `SELECT` statement, which looks almost the same as the usual `SELECT` statement you write in your ABAP programs (Listing 15.2).

```
SELECT FROM zt_monsters AS monster_header
INNER JOIN zt_monster_items AS monster_items
ON monster_header.monster_number = monster_items.monster_number
LEFT OUTER JOIN zt_monster_pets AS monster_pets
ON monster_header.monster_number = monster_pets.owner
```

Listing 15.2 `SELECT` Statement

There are two differences between a `SELECT` statement in traditional ABAP OpenSQL and the CDS view equivalent. First, the aliases (as in `TABLE_NAME AS ALIAS`) have to be declared, whereas this is optional in ABAP (and not really used outside of SAP examples or code cut and pasted from SAP examples). Second, in traditional ABAP, after the `SELECT` statement is either an asterisk to denote “everything” or a list of fields that you want from the database tables.

One aspect that remains the same as coding a regular `SELECT` statement in ABAP is that when coding a DDL that generates a CDS view you don't have to put the client (`MANDT`) into the `SELECT` statement; the runtime system handles this for you, just like you've always been used to. In Section 15.2.2, you'll see that this is not the case for ABAP Managed Database Procedures.

You will also notice that the example in Listing 15.2 contains an outer join, and you cannot do an outer join in an SE11 database view. In fact, in the DDL that creates a CDS view you can not only do a `LEFT OUTER JOIN`, but also a `RIGHT OUTER JOIN`, `UNION`, `UNION ALL`, and `UNION CITY BLUE`, the latter four being unknown concepts in traditional ABAP `SELECT` statements.

Next, you want to declare a list of fields, just as you would in a traditional SE11 view. In SE11, you have two columns: on the left are human-friendly names, like `CUSTOMER` and `CUSTOMER_NAME`; on the right are the data element names, like `KUNNR` and `NAME1`. In CDS views, things are slightly different; you're going to make a big list of the human-friendly names for the fields that will be available in the view (you can still use German abbreviations if you want), and how they're going to be typed will be determined later on by the system based on what values from the database (or from a calculation) are going to be placed into them. This is rather like the `DATA()` statement you looked at in Chapter 2 when defining variables in ABAP at the place in the code where they have a value placed into them.

To recap, you're going to define the name of each view field and how it is to be filled together, one at a time. Instead of listing all the fields after the `SELECT` statement as you would do in an SQL query, the list of fields and how to fill them comes in the body of the code, after the `SELECT` and `JOIN` criteria but before the `WHERE` clause. This list of fields is enclosed in curly-wurly brackets (`{ }`), as shown in Listing 15.3.

```
{
Key monster_header.monster_number AS monster_number,
Monster_header.Monster_name AS monster_name
LPAD( monster_header.hat_size, 10, "0" ) AS hat_size
SUBSTRING( monster_header.monster_name, 0, 1 ) AS first_initial
}
```

Listing 15.3 List of Fields for the CDS View to Get from the Database

Once again, you are required to declare an alias every time (e.g., `AS monster_number`). The target field has no explicit type declaration; it is typed according to the value that is being passed into it. You will notice in Listing 15.3 that there are some built-in functions: for when you want to return a hat size of 3 as 0000000003 and return the first initial of a monster called Fred as "F". There are bucket loads of such functions available, and more creep in with each support stack.

Back in Chapter 2, you saw that `CASE` statements had crept into OpenSQL statements in ABAP, and much as you might shout "Get back! Get back! Get back to where you once belonged!" at them, it's a fact of life in ABAP 7.4 that `CASE` statements go where they please when they please (swaggering, arrogant bullies that they are). Here they can be used inside a CDS view, as demonstrated in Listing 15.4. You will note they can also be nested.

```
CASE monster_header.sanity_description
  WHEN 'BONKERS' THEN 'REALLY SCARY'
  WHEN 'MAD' THEN
    CASE monster_header.strength_description
      WHEN 'REALLY STRONG' THEN 'SCARY'
      ELSE 'NOT SO SCARY REALLY'
    END
  WHEN 'SLIGHTLY MAD' THEN 'SLIGHTLY SCARY'
  ELSE 'NOT REALLY SCARY AT ALL'
END AS scariness
}
```

Listing 15.4 CASE Statement within a CDS View

The closing curly bracket at the end of Listing 15.4 means that the list of required fields is at an end, and you can have the `WHERE` clause now if you want, as shown in Listing 15.5.

```
WHERE monster_header.sanity < 10
      AND monster_header.color = 'GREEN'
```

Listing 15.5 `WHERE` Clause

Often, in SE11 views you don't have hard-coded restrictions, such as the ones shown in Listing 15.5. Maybe you would restrict a view on VBAK to only have sales orders by saying `AUART = C`. It's the same in a CDS view; hard-coded restrictions will most likely be used sparingly.

One point to note is that, in the DDL coding, a `WHERE` statement can have values that have no real equivalent in ABAP (i.e., `IS NULL` and `IS NOT NULL`). Almost every other programming language apart from ABAP has the `NULL` concept to indicate that a data value does not exist in the database.

In ABAP, you're used to saying "equals space" for a text field or "equals zero" for a number or quantity. That works fine, but it's made IT purists burst into tears and bang their heads on the table for many years.

To end the view code, you can have a `GROUP BY` clause if you want. (This would not make a lot of sense in this example, because you're interested in details of individual monsters as opposed to aggregated data for groups of monsters.)

In an OpenSQL query, you always have an `INTO` clause, but there's no need for an `INTO` clause in a view definition, because the target for the retrieved data will be specified by whatever code queries the view.

Just like an SE11 view, the definition resides in ABAP but causes a `CREATE` operation in the database attached to the ABAP system. Therefore, the database will always be in sync with the ABAP system, and when you transport the CDS view definition via the normal transport system, you don't need to worry about manually keeping the database up to date.

Finally, the last step in the process is to call the CDS view from your ABAP system. When doing so, you don't use the name of the SQL view you can see in SE11 but rather the name of the CDS view itself. This is shown in Listing 15.6.

```
SELECT *
FROM zcds_monsters
INTO lt_specialized_monsters.
```

Listing 15.6 Calling a CDS View from within an ABAP Program

Latest CDS View Built-in Functions

Just to give an indication of how fast the functionality of CDS views are evolving, at the time of writing two support packs were released in fairly quick succession: SP 8 and SP 9. In SP 8, a whole bunch of new built-in functions became available—for example, FLOOR, REPLACE, MOD, ABS, ROUND, and COALESCE. Also in SP 8, support was added for when the data in the database table is comprised of amounts in different currencies or quantities in different units of measures. There are built-in functions CURRENCY_CONVERSION and UNIT_CONVERSION, which enable the database to convert all the currency values into Vietnamese Dongs before sending the data back to the application server or to convert everything into kilograms so that you get a meaningful total quantity.

In SP 9, improvements were made to the lifecycle management of CDS views: Before, it was very difficult to change an existing CDS view when the existing database table had data inside it, and now that problem has gone away. In addition, in SP 9 the concept of an enumeration was introduced into the DDL syntax. Most other programming languages have this concept; it can best be thought of as a domain with several possible values for a data element, only defined inside a block of code as opposed to inside SE11.

It's also possible to code a CDS view with parameters. Again, the process is slightly different than in traditional ABAP. Transaction SE11 views allow you to define selection parameters, but they are hard-coded (e.g., the monster color is set as GREEN). When dynamic selections are required (e.g., the user chooses what he wants on a selection screen), you would add WHERE clauses in your ABAP program to restrict what the view brought back. An example of this is shown in Listing 15.7.

```
SELECT *
FROM zv_monster_views
WHERE color EQ p_color
AND sanity IN s_sanity.
```

Listing 15.7 Coding Parameters in Traditional ABAP

You can do the exact same thing when referring to a CDS view within your program, but in addition, starting with SP 8, in CDS views you can define them with parameters that are passed in via the ABAP program at runtime, as shown in Listing 15.8.

```

@ABAPCatalog.sqlViewName: 'ZV_MONSTERS_PARAMETERS'
Define view zcds_monster_parameters
  with parameters p_sanity_low:ZDE_MONSTER_SANITY,
                 p_sanity_high:ZDE_MONSTER_SANITY,
                 p_color:ZDE_MONSTER_COLOR
As select from zt_monsters
  { key monster_number,
    monster_name,
    color,
    sanity,
  strength }
WHERE color = p_color AND
       sanity between p_sanity_low and p_sanity_high;

```

Listing 15.8 Defining a CDS View with Parameters

From your program, you then write a statement like the one in Listing 15.9.

```

SELECT *
FROM zcds_monster_parameters(
  p_color      = @p_color,
  p_sanity_low = @s_sanity-low
  p_sanity_high = @s_sanity-high )
INTO TABLE @(lt_colorful_mad_monsters).

```

Listing 15.9 Calling a CDS View with Parameters

You could also have a mixture of parameters and normal `WHERE` clauses. The main advantage of parameters is that inside CDS views you can call functions, and an input parameter to the CDS view as a whole can be passed into function calls made within the CDS view. With `WHERE` clauses, you can only refer to the list of database fields defined in the view; parameters can be anything you feel like.

As of ABAP version 7.4 SP 8, there is only support for some databases (e.g., SAP HANA) to enable an ABAP program to do a `SELECT` on CDS views with parameters. The roadmap from SAP says that, ultimately, all databases will be able to take advantage of this feature.

View Appends

Another gap that has been plugged is that, previously, if SAP had created a view, then in SE11 you could add an `APPEND` structure to it with some fields of your own (e.g., Z fields you have added to VBAK). You could not do this in SAP-supplied CDS views until ABAP 7.4 SP 8; thereafter, you can get the same result by creating a DDL as follows:

```

@AbapCatalog.sqlViewAppendName: 'ZA_STANDARD_EXTENSION'
  extend view standard_thing with zcds_standard_extension

```

```
{ vbak.zz_new_field_one
  vbak.zz_new_field_two };
```

Generating the CDS View in the Database from the DDL

When you're finished coding your DDL in Eclipse, click the GENERATE icon, and the CDS view is generated inside the database (and you can only view and change this from Eclipse via the DDL object). Also, an SE11 view is created in the ABAP system. In your code, you should only refer to the CDS view entity, not the SE11 view, as shown in Listing 15.10.

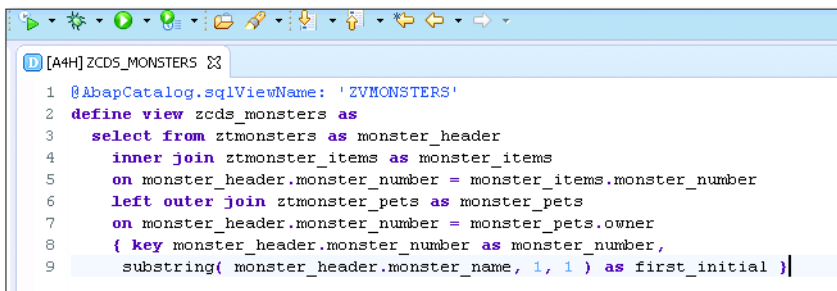
```
SELECT * FROM zcds_monsters
  INTO TABLE @lt_monsters "Use CDS entity name
```

Listing 15.10 Reading Data from a CDS View Entity in ABAP

Note

When doing a SELECT on your CDS view entity, you have to use OpenSQL syntax (as described in Chapter 2), which means that variables have to have an @ sign in front of them, and the INTO clause has to go at the end.

I suspect SAP doesn't even want you to look at the SE11 view, but I'm going to show you just that. First, in Figure 15.4 you can see a really simple CDS view defined in Eclipse.



```
[A4H] ZCD5_MONSTERS Σ
1 @AbapCatalog.sqlViewName: 'ZVMONSTERS'
2 define view zcds_monsters as
3   select from ztmonsters as monster_header
4     inner join ztmonster_items as monster_items
5       on monster_header.monster_number = monster_items.monster_number
6     left outer join ztmonster_pets as monster_pets
7       on monster_header.monster_number = monster_pets.owner
8   { key monster_header.monster_number as monster_number,
9     substring( monster_header.monster_name, 1, 1 ) as first_initial }
```

Figure 15.4 CDS View in Eclipse

As you've learned from this section, this view defines the join conditions for the various database tables, and then between the { } brackets the fields of the view are listed together with instructions on how those fields are to be filled. In SE11,

you can see the representation of the database joins for the ZVMONSTERS SQL view (Figure 15.5).

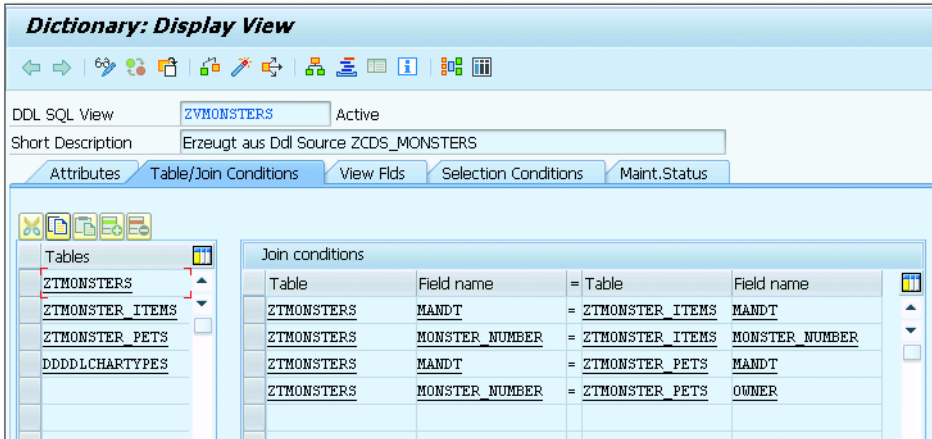


Figure 15.5 SE11 View: Database Join Conditions

Did you notice that the generated name for the view is in German? Anyway, you can then look at the fields that can be retrieved from the view by moving to the VIEW FLDS tab (Figure 15.6).

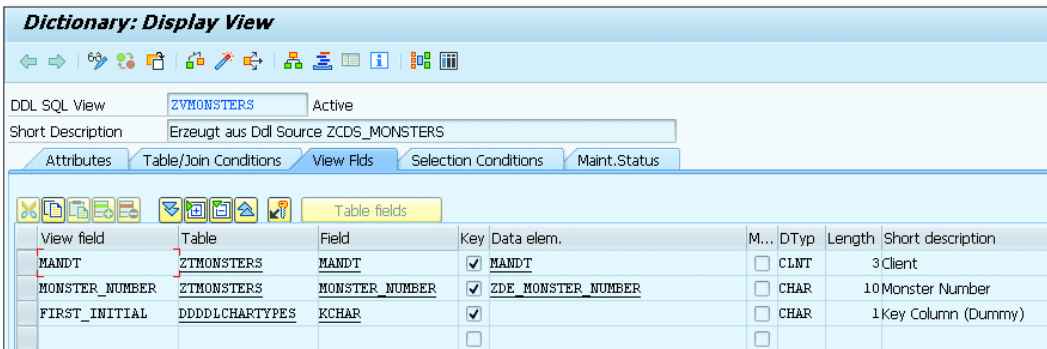


Figure 15.6 SE11 View: Database Fields

You'll notice that the SAP system has been very clever and worked out the type and length of the FIRST_INITIAL field from the CDS view, a field that doesn't directly refer to a database table field and hence is not tied to a data element.

Summary

In this section, you've created a DDL object in Eclipse, in which you wrote a much more complicated SQL statement than would be possible in ABAP. When the DDL object activated, an SE11 view was created on the ABAP system, and a CDS view was created in the SAP HANA database.

Before the CDS view concept came along, you had to retrieve the data from the database using a view and then do some manipulation of that data in your ABAP program. Now, because the CDS view (which runs in the database) has taken over some of the tasks that formerly you would have done in the ABAP program (which runs on an application server), those tasks—or, more accurately, the code to do those tasks—has been pushed down into the database.

15.2.2 Building and Calling ABAP Managed Database Procedures

In Section 15.3, you'll see that, in the bottom-up development approach, the call to a stored procedure is a bit like using a function module—that is, very procedural (old-fashioned). With the top-down approach, you have instead an ABAP Managed Database Procedure (AMDP), which is a method of a class and thus fully object oriented.

As you know, when you create classes and methods of a class, there are two things you need to code: the definition and the implementation. The process for an AMDP is exactly the same. First, you'll look at how to define a class such that one or more of its methods can be executed directly on the database by means of an SAP HANA stored procedure. Then, you'll learn how to code the implementation of such a method using SQLScript, such that although the code is written in the ABAP system it will actually execute inside the SAP HANA database at runtime.

Defining the ABAP Method

If you have a class that has at least one method in which you would like to delegate processing to the database, then the only change you need to make to the definition of that class is to include the interface `IF_AMDP_MARKER_HDB`, as shown in Listing 15.11.

```
CLASS zcl_monsters DEFINITION.  
PUBLIC SECTION.  
    INTERFACES IF_AMDP_MARKER_HDB.
```

```

METHOD get_complex_monster_data
  IMPORTING VALUE( this_and_that ) TYPE whatever
            VALUE( id_client )     TYPE sy-mandt
  EXPORTING VALUE( the_other )     TYPE something_else.

```

Listing 15.11 Class Definition with Interface to Enable Use of AMDPs

Although the CDS views discussed in Section 15.2.1 are more or less database independent, AMDPs are specific to SAP HANA. This is why `_HDB` (for “HANA database”) appears at the end of the interface name shown in Listing 15.11. The other point you may be puzzled about in Listing 15.11 is that you have to tell the SAP HANA database what client you’re talking about, and thus the method needs to get the client (`MANDT`) as an importing parameter. In the ABAP world, you’re used to the system automatically reading or writing to the client where the program is running—but in the database world, the client is a vital piece of information. (As mentioned in Section 15.2.1, when coding a DDL for a CDS view, you don’t have to pass in the client; the runtime system handles it.)

Did you also notice that you have to pass everything by value? That’s compulsory if you want a method to make use of an AMDP.

Implementing the ABAP Method Using an AMDP in SQLScript

If you just did a normal implementation of a method of a class defined as shown in Listing 15.11 with normal ABAP code, then it would run in the application server as normal. However, this isn’t what you want, so you have to tell the method implementation to behave differently.

You can control whether a method is run on the normal ABAP application server or within the database by adding some extra lines to the implementation of the method. This is shown in Listing 15.12.

Warning: Houston, We Have a Problem

If you put the word `SQLSCRIPT` in lowercase, Eclipse gets really upset. Conversely the word `hdb` has to be in lowercase.

```

CLASS zcl_monsters IMPLEMENTATION.
  METHOD get_complex_monster_data
  BY DATABASE PROCEDURE
  FOR hdb
  LANGUAGE SQLSCRIPT

```



```

    USING zt_monsters zt_monster_items.
        " SQLScript code goes here
    ENDMETHOD.
ENDCLASS.

```

Listing 15.12 Implementing a Method so that It Runs inside the Database

After the method implementation in Listing 15.12, you'll see the words `BY DATABASE PROCEDURE`, which says that this method will generate and execute a stored procedure in the database at runtime. Having the control in the implementation as opposed to the definition is a stroke of genius: It means you can redefine the method so that subclasses can run the method in the server or in the database, or during unit tests a test double can just return hard-coded values. In addition to the obvious benefit for unit testing, if you're dubious as to whether pushing a certain method down to the database is really worthwhile, then you can create one class that runs in the server and a subclass that runs in the database and do some tests where you swap between them to compare the runtime.

The next line in Listing 15.12 says `FOR hdb`, which means that you want to use an SAP HANA database, and `LANGUAGE`, to say what language you want. As Henry Ford once said, "You can have any color you want, as long as it is black." In the same way, you have these options to specify what database you can use (it has to be SAP HANA) and what language you are going to code the AMDP in (it has to be SQLScript).

Finally, in Listing 15.12 you have the `USING` statement, in which you have to give an explicit list of what database tables (or views) you're going to use within the procedure. It's also possible for one database procedure to call another, and if you want such a thing, then you have to mention the database procedure you're going to invoke in this list as well. This whole concept will seem strange to ABAP programmers, who are used to just reading from whatever database table takes their fancy and calling global functions and methods at will, but remember: You're in another world now, the world of the database, where the laws of physics are different, and water flows uphill; this is programming, but not as you know it.

Restrictions

You cannot create the AMDP coding (or any sort of SAP HANA artifact, for that matter) in SE24 or SE80. You can display it, but to create or change the code you need to use ABAP in Eclipse. (Hopefully Chapter 1 convinced you that is the best tool to use any-

way.) Another restriction is that in a method that is to be implemented as an AMDP, you cannot have returning parameters, and all parameters have to be passed as values. The ABAP in Eclipse aspect of this is unlikely to ever change, but I wouldn't be shocked if the other restrictions were lifted at some point in the future. For example, before ABAP 7.4 SP 8, you couldn't have `CHANGING` parameters in a method that was going to be implemented as an AMDP; now you can. In the same way, before SP 8, any sort of exception caused a short dump; after SP 8, the exception can be caught by `CATCH cx_admp_error`.

In a stored procedure, you have two choices when retrieving data from the database: SQL statements or calculation engine (CE) functions. A year ago, SAP's position was that you should use CE functions whenever possible. Now, the instruction is not to touch them with a 10-foot bargepole. CE functions were once supposed to perform faster for basic operations, but over time the performance of real SQL statements has improved to the extent that the overhead of wrapping them in a CE function is not worth it.

In any event, the important thing is to keep the two (SQL statements and CE functions) apart. As an example, you may have encountered a dual-stack SAP Process Integration (PI) system in which half of the code was in ABAP and half in Java. The ABAP code ran fast, the Java code not quite so fast but okay, but the performance killer was when data had to keep moving between the two stacks. In the same way, if you have both SQL statements and CE functions in one stored procedure, then the poor old database is thrown into a tizzy, and the performance is much worse than if you had used only one or the other.

An SQL statement in SQLScript does not look that radically different from its ABAP equivalent; you just have lots more options available and the syntax (punctuation marks) changes slightly. You saw this just now when we talked about CDS views. The change here is that CDS views manage the client for you; in an AMDP, you have to specify the client.

In Listing 15.13, you'll see some SQLScript code to do a database `SELECT` from within an AMDP method implementation.

```
METHOD get_people_scared_today BY DATABASE PROCEDURE
FOR HDB
LANGUAGE SQLScript
OPTIONS READ-ONLY
USING zt_monsters zt_scary_event_header zt_scary_event_items.
```

```

-- declare a local variable
declare ld_today date;
-- get current date
select current_date into ld_today from dummy;
-- select relevant scary events
lt_scary_events =
select
event_items.mandt          as client,
event_header.event_number as event_number,
monster.monster_number    as monster_number,
event_header.created_at   as scaring_date,
event_items.people_scared,
from zt_scary_event_items as event_items
join zt_scary_event_header as event_header
on
event_items.mandt = event_header.mandt
and
event_items.event_number = event_header.event_number
join zt_monsters as monster
on
event_header.client = monster.client
and
event_header.monster_number = monster.monster_number
where
event_header.mandt = :id_mandt
and monster.number = :id_monster_number
and event_header.scare_date = :ld_today
and event_header.scary_status = 'REALLY_SCARY';

```

Listing 15.13 Coding a Database SELECT inside an AMDP

Most of the code in Listing 15.13 is just business as usual, except for the following:

- ▶ The syntax for declaring the local variable for today's date was not quite what you're used to; for example, you don't need the word `TYPE`. However, such minor syntax differences are part and parcel of learning to code in a new language.
- ▶ You have to put a colon before variables in the `WHERE` clause.
- ▶ You'll notice that the local table for storing the scary events is declared as it's being created, just like the new `DATA` constructs in ABAP 7.40.
- ▶ You're in the database now, so the "internal table" `LT_SCARY_EVENTS` is not really a local internal table, but rather a temporary database table that exists for

the duration of the procedure. As such, later on in the stored procedure you can do a `SELECT` upon that table, just like any other database table.

Even though you're not supposed to use them, a few words on CE functions: There are a fair few CE functions available in SQLScript, but only a subset of them are currently usable in AMDPs. The one most often (in fact, usually the only one) mentioned in the AMDP examples you'll see on the Internet is `CE_CONVERSION`, which is for when you have a database table full of amounts in different currencies and you want to unify them. Such an example can be seen in Listing 15.14.

```
lt_final_table =
ce_conversion( :lt_items_in_different_currencies,
[ family = 'currency',
  method = 'ERP',
  steps = 'shift,convert,shift_back',
  source_unit_column = "CURRENCY_CODE" ,
  output_unit_column = "CURRENCY_CODE_CONV",
  target_unit       = 'VND', "Vietnamese Dongs
  reference_date    = :ld_today,
  client            = :id_mandt ],
[total_field] ) ;
```

Listing 15.14 Calling a CE Function

As a final point, even though you're not coding in ABAP inside an AMDP, the syntax check will know that the language inside the method is SQLScript and adapt accordingly (i.e., you will get an error if you have syntax errors in your SQLScript).

Summary

You've now defined an ABAP class that implements `IF_AMDP_MARKER_HDB` to allow methods of that class to be executed inside the SAP HANA database and defined a method for that purpose. Then, you created a method implementation with special annotations so that the runtime system knows that this method should run in the database as opposed to the application server. Finally, you coded the method—in SQLScript, as opposed to ABAP—using assorted features that duplicate the business logic you would normally code inside your ABAP method to manipulate data that you have retrieved from the database.

As the AMDP method runs in the database, all the business logic is run before the results are returned to the calling program that runs in the application server. Thus, the code—just as in the last section on CDS views—has been pushed down to the database.

15.3 Bottom-Up Development

So-called bottom-up development (Figure 15.7) involves creating your artifacts (view and stored procedures) directly in the SAP HANA development environment and then making those SAP HANA artifacts available to the ABAP environment.

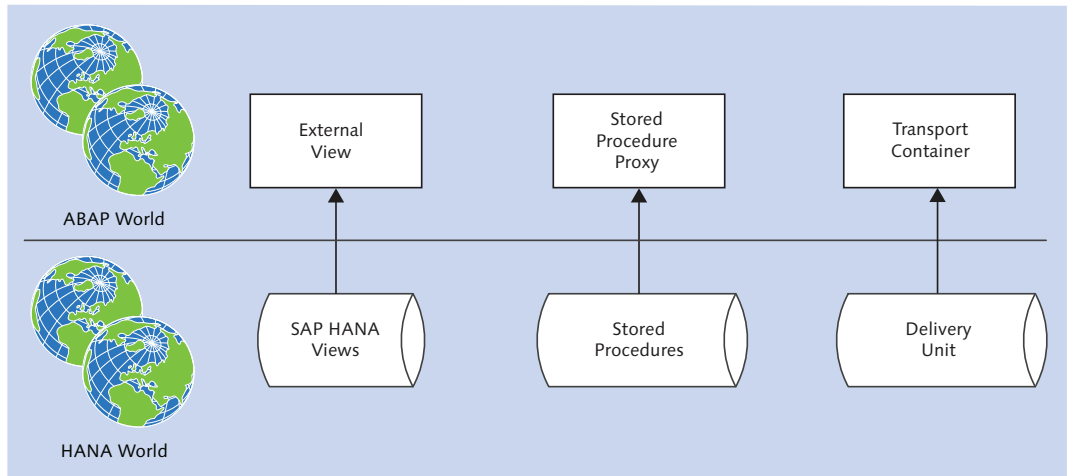


Figure 15.7 Bottom-Up Development

As mentioned earlier, bottom-up development has been superseded by top-down development, which was discussed in detail in Section 15.2). SAP recommends top-down development if you're on a release that supports it (ABAP 7.4 SP 5). Nonetheless, you may be in the uncomfortable position of being between 7.4 SP 2 (in which bottom-up development was first enabled) and SP 5 (in which top-down development was enabled), with no prospect of moving to a higher support stack for some time. In addition, SAP warns that there may still be situations in which a top-down approach doesn't do what you require and thus you're forced down the bottom-up path, although such gaps are reduced with every support stack released. Therefore, this section will talk a bit about how the bottom-up approach works.

As you can see in Figure 15.7, there are three boxes (related to views, stored procedures, and transports) that together make up the bottom-up approach. This section discusses each of these boxes in turn. Section 15.3.1 talks about how to

create external views in SAP HANA and how to enable the ABAP system to be able to access such a view. Section 15.3.2 talks about how to call stored procedures that have been created in SAP HANA from ABAP.

The main drawback to the bottom-up approach is that you have to work in two unrelated development environments and manually keep the changes between the two systems synchronized. Therefore, Section 15.3.3 outlines the fairly convoluted transport mechanism you need when following such an approach.

Note

Because bottom-up development is no longer the recommended development approach from SAP, this section provides only an overview of the general process.

15.3.1 Building and Calling External Views

There are two halves to creating an external view: defining the view in SAP HANA and making such a view visible inside the ABAP system.

An external view is “external” in that the definition is controlled outside of the ABAP system. Therefore, the first thing you need do is to create a view within SAP HANA, which means opening a development environment such as SAP HANA Studio. Inside here, you can build up a graphical representation of the view you want to create, dragging and dropping the database tables you’re interested in and linking the columns for the join.

The actual discussion of how to create SAP HANA views is outside the scope of an ABAP book, so this isn’t discussed here. Instead, assume that your view has been created—and now you’re ready for the ABAP side of things. In ABAP in Eclipse, you can right-click the `Dictionary` node of one of your packages and follow the path `NEW • DICTIONARY VIEW`. The pop-up box shown in Figure 15.8 appears.

You can see a radio button called `EXTERNAL VIEW` at the bottom (grayed out in systems without SAP HANA) and a box to enter the name of an SAP HANA view (or an option to browse for the one you want). When you click `FINISH`, a DDIC view is generated, which can be seen in SE11. The only difference between this sort of view and a traditional SE11 DDIC view is that this will have the words `EXTERNAL VIEW` at the top (as opposed to `PROJECTION VIEW` or `MAINTENANCE VIEW` or any of the other types of view you’re used to).

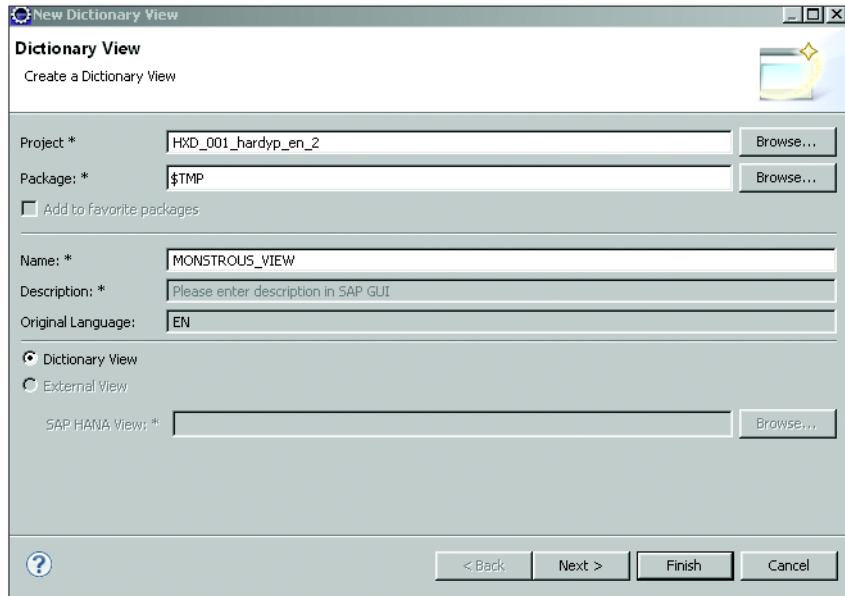


Figure 15.8 Creating an External View

Once a view exists in SE11, inside your ABAP code you can define internal tables based upon the view and do a normal SELECT statement, as in Listing 15.15.

```
DATA: lt_monsters TYPE STANDARD TABLE OF zev_monstrous_view.
SELECT *
FROM zev_monstrous_view
INTO CORRESPONDING FIELDS OF lt_monsters.
```

Listing 15.15 Calling an External View from an ABAP Program

15.3.2 Building and Calling Database Proxies

The first step in the process of creating and coding a stored procedure via bottom-up development is to create the procedure directly inside the SAP HANA development environment. During the creation of a stored procedure, you need to specify the import and export parameters and write the SQLScript to retrieve the data; the code in the stored procedure will look rather like that inside an AMDP implementation (refer back to Section 15.2.2). The creation of this stored procedure in SAP HANA is, of course, not an ABAP task and thus won't be discussed in more detail.

The second step in the process is to replicate the import and export structures from the stored procedure in the ABAP dictionary. To do this, you need to go into ABAP in Eclipse and create a database procedure proxy. Open Eclipse, and select the package in which the proxy is to be created. Right-click this package, and follow the menu path **NEW • OTHER ABAP REPOSITORY OBJECT • DICTIONARY • DATABASE PROCEDURE PROXY**.

You will be asked for a name (e.g., `ZMONSTER_PROXY`) and what SAP HANA stored procedure you want to link the new ABAP proxy object to. A name is then proposed for an ABAP interface (e.g., `ZIF_MONSTER_PROXY`), which will be created to store the `TYPE` definitions for the input and output parameters.

When you click **FINISH**, the interface is created in your ABAP system along with a database procedure proxy that can be called from an ABAP program. An example of a generated interface is shown in Listing 15.16.

```
INTERFACE zif_monster_proxy PUBLIC.
  TYPES: in TYPE zs_monster_inputs,
         out TYPE zs_monster_outputs.
ENDINTERFACE.
```

Listing 15.16 Generated Interface

Once the interface and proxy have been generated, you can call the newly created `ZMONSTER_PROXY`, which will in turn call the SAP HANA stored procedure from an ABAP program (Listing 15.17).

```
DATA: in TYPE zif_monster_proxy=>in,
      out TYPE zif_monster_proxy=>out.

CALL DATABASE PROCEDURE zmonster_proxy EXPORTING in = in
                                       IMPORTING out = out.
```

Listing 15.17 Calling a Database Procedure Proxy

That's simple enough, but you're working in two development environments, and the whole concept is not really very object oriented. You can see why this was only intended as a stopgap measure until the real solution (top-down development) was ready.

15.3.3 Transporting Changes

With the bottom-up approach, there's no automatic link of changes made in the SAP HANA environment and the ABAP environment. This is problematic,

because the ABAP artifacts depend on the equivalent versions of the SAP HANA artifacts to exist in the same system (e.g., the version in the SAP HANA test box must match the ABAP version in the SAP test box).

Because there's no link, moving changes between development and test systems has a fair few steps, as follows:

1. Create the SAP HANA artifacts (views and/or stored procedures) in the SAP HANA development system via SAP HANA Studio.
2. Create the ABAP equivalents in the SAP development system as described in Section 15.4.1 and Section 15.4.2. These are normal repository objects and thus will create a transport request. As you make changes to the ABAP objects, those changes are stored in that transport request.
3. Just before you are ready to move everything to the test system—minutes before—create a repository object in the ABAP system called an *SAP HANA transport container*. During the creation process, the system will ask you what SAP HANA artifacts you want to move. The definitions of these artifacts from the SAP HANA system are then stored against your transport request. Because there's no automatic link, if you change one of your SAP HANA artifacts, then a few minutes later the ABAP transport request will not know about the change. What's in the transport request is a snapshot of what the artifact looked like at a given moment in time.
4. Release the ABAP transport request; off it goes to test via STMS in the normal fashion.
5. Once the transport is in test, the SAP HANA-related content must be exported from the test ABAP system to the test SAP HANA system.

I imagine you're reading this with horror, but if you don't yet have an ABAP system on 7.4 SP 5, then this is the only way you can proceed.

15.4 Locating Code that Can Be Pushed Down

Now you know how to push your code down into the database—but how do you decide what code deserves that fate and what technique is appropriate? Fear not! This section explains how to find custom code that needs to be pushed down and how to work out which of the techniques you've read about should be used to

implement the code pushdown. This section will conclude with an example that shows a case in which code should be pushed down.

15.4.1 How to Find Custom Code that Needs to Be Pushed Down

Luckily for you, SAP has realized that you might need some help locating code to be pushed down, and provides tool support for exactly this purpose. The tools provided to you enable an analysis of the SAP environment at your organization and spit out a list of what sections of your custom code you need to have a look at with a view to pushing down the code.

There are three transactions working in tandem that enable this analysis:

- ▶ The ABAP Test Cockpit (SATC)
- ▶ The SQL Monitor (SQLM)
- ▶ The SQL Worklist (SWLT)

The ABAP Test Cockpit (which, as you'll recall, was the subject of Chapter 4) does a static check of your custom code in the development system. SQLM is a transaction for analyzing database calls in production; the results look just like ST05. The big differences are as follows:

- ▶ ST05 is designed for a one-off analysis of a single program; this is designed for analyzing everything in the system and holding on to that data for a protracted period (e.g., everything that happened in the system for a year).
- ▶ This information can be retrieved from the development system via RFC.

The last member of this trio is another new transaction called the SQL Worklist (Transaction SWLT; Figure 15.9), which retrieves the production data from SQLM via RFC. Then, coupled with a static analysis of the ABAP code via the ATC, Transaction SWLT gives you a ranked worklist of what routines could do with being improved generally and, once you're considering a database migration, what code could do with being pushed down into the SAP HANA database.

In a nutshell, ATC + SQLM => SWLT.

Exactly how to go about the SWLT process to hunt down code that could be improved by moving to the database could fill up a book all on its own. Luckily, there has been so much written on this subject that you do not have to go hunting far (see the "Recommended Reading" box at the end of this chapter for a starting point).

SQL Performance Tuning Worklist: Selection Screen

Application Help

Object Set

Package: Z* to []

Object Type: [] to []

Object Name: [] to []

Static checks may return findings outside the object set

SQL Monitor Settings

Request Type: [] to []

Request Entry Point: [] to []

Minimum Number of Executions: 5000

Show All Results
 Show Top n Executions
 Show Top n Times

SQL Monitor Snapshot Selection

No snapshot selected or available
 Latest Snapshot by System
 Overall Latest Snapshot
 Individual Snapshot

Static Check Settings

Code Inspector
 ABAP Test Cockpit
 None

RFC Destination: NONE ✓ (SOD)

No ATC run series matching the selection criteria found

Additional Options

Figure 15.9 Transaction SWLT

ATC/SQLM/SWLT Availability

The ATC tool was made available in SAP NetWeaver 7.02 SP 12 or SP 5 of SAP NetWeaver 7.31, and comes as standard in SAP NetWeaver 7.4.

The SQL Monitor is available as of SAP NetWeaver 7.02 SP 14 or SAP NetWeaver 7.31 SP 9, and comes as standard with SAP NetWeaver 7.4. You also need the ST_PI plug-in to be on level 2008, support stack 8.

The SQL Performance Tuning Worklist comes with SAP NetWeaver 7.02 SP 14, SAP NetWeaver 7.31 SP 9, or SAP NetWeaver 7.4 SP 2.

15.4.2 What Technique to Use to Push the Code Down

Once you have a list of what custom code might benefit from being pushed down to the database, for each such block of code you need to decide how exactly to achieve this. There are in fact three code pushdown methods to choose from: one of those methods is OpenSQL (which you have always used to push down logic to the database, even though you might not have thought that you were doing such a thing), and the other two methods are database views and stored procedures, which as we have seen can be implemented using either bottom-up or top-down development.

To decide which technique is appropriate, perform the following steps for each chunk of code:

1. See if you can use the extended syntax of OpenSQL (as discussed in Chapter 2) in a normal ABAP program to outsource the extra logic to the database. This technique works with any database.
2. If you cannot, then see if you can achieve the same with a CDS view. These work with any database and thus are “open.”
3. If a CDS view cannot achieve what you are looking for and you have an SAP HANA database, then a stored procedure is what you need to look into. The stored procedure offers more functionality than the CDS view (e.g., it can return multiple result sets).

You will notice that this is far from the popular perception that SAP wants you to push everything down into the database. In fact, the motto at SAP is “stay open,” which is why OpenSQL is at the top of the list.

15.4.3 Example

The best way to show an example of hunting for code that can be pushed down is to take a real-life example (changed slightly to protect the innocent) that I came across while peer reviewing the work of another programmer the other day. Nothing was wrong with the code he had written, but I thought to myself, “This is a job for SAP HANA.” This example will walk you through that code and explain why it is a good candidate for code pushdown.

Like any good businessman, Baron Frankenstein expects to be paid for the monsters he makes for his clients, and he issues invoices saying that payment is due

30 days after the end of the month in which the invoice date falls. However, some customers go bankrupt and thus cannot pay the invoice; in SAP terms, the customer open item must be classified as a bad debt. An accountant (e.g., Igor) has to clear the open item by writing off the debt from the balance sheet and charging it to the profit and loss section of the accounts.

To make matters worse for the customer (as if going bankrupt was not bad enough), the baron presses a button on his control panel, and the monster delivered to the customer goes into a frenzy, destroying everything in sight and then killing and eating the customer. As a result, the Baron has a remarkably low bad debt ratio.

Nonetheless, customers who have been eaten by monsters are not very good at paying off their debts afterwards, so there remain a number of cleared open items in the baron's SAP system relating to such bad debts. Luckily for the baron, the Transylvanian government has recognized this problem. If the baron sends an electronic extract of such bad debts to the government each month, then he can get tax relief for them.

Now imagine there is a program that looks at the cleared open items relating to bad debts; it runs each month and sends that data to the government in the format it requires. The most important thing naturally is not to claim the same bad debt twice, because that would be fraud—much worse than killing and eating your customers.

The program identifies which customers are bad debt customers (i.e., did not pay their bills and thus have been eaten), loops through each such customer, and reads that customer's cleared open items. Once this data has been retrieved from the database, the program loops through each record and applies some business logic to determine if the record should be sent to the government.

There is a variable called `GD_WOFF_DATE` that stores the date on which the customer came to a gruesome end. Because this is the government the baron's dealing with, the rules are quite complicated: any cleared items with a creation date (`CPUDT`) equal to or later than the date of the customer's horrible, horrible death (`GD_WOFF_DATE`) can be claimed as a tax write-off. For open items with a creation date earlier than the write-off date, only items with a clearing date (`AUGDT`) greater or equal to the write-off date can be claimed.

The first time the program runs, all such items are sent to the government, and the variable `GD_LASTEXTRACT` is updated to the date those items were sent to the government so that they don't get sent again. However, what if some new items are added later? They need to be sent as well. Therefore, the program reads the entry date of the cleared item (`CPUDT`), and if the entry date is later than the last extract, then the new record needs to be sent to the government and the latest extract date updated accordingly. The ABAP code to perform this logic on the records retrieved from the database is shown in Listing 15.18.

```

if not gt_output[] is initial.

LOOP AT gt_output.
IF gt_output-cpudt GE gd_wofff_date AND
  "Cleared item created after customer went bad and
  gt_output-cpudt GT gd_lastextract.
  "Cleared item created after last extract
  "Extract the record
  CONTINUE.
ELSEIF gt_output-cpudt LT gd_wofff_date AND
  "Cleared item created before customer went bad and
  gt_output-cpudt GT gd_lastextract AND
  "Cleared item created after last extract
  gt_output-augdt GE gd_wofff_date.
  "The item has a clearing date after the customer went bad
  "Extract the record
  CONTINUE.
ELSE.
  "Do not extract the record
  DELETE gt_output.
ENDIF.
ENDLOOP."Cleared Items
  if gt_output[] is not initial.
    gd_baddebt_flag = 'X'.
    perform populate_openitems.
  endif.

endif.

endloop. "Dead Customers

```

Listing 15.18 Removing Bad Debt Records from a Table on the Application Server

You will notice in Listing 15.18 that every single record that could possibly be sent to the government is retrieved from the database, and then some business logic is done to remove the ones that are not required. This breaks one of the SQL golden rules—namely, minimize the amount of data sent from the database to the application server. Getting records you don't want is like ordering 10 elephants

when you only want one and then, when the elephants arrive, having to arrange for nine of them to be shipped back to Africa.

The bottleneck (even in an SAP HANA database) is the transfer of such data, and so the idea is to remove the records that are not required *before* they are sent from the database to the requesting ABAP program. So, why do you see code like Listing 15.18 all the time? It's because up until now there has been no choice at all in the matter. The only way to achieve a business requirement such as the one in the example is to get every record from the database, examine those records one at a time, and disregard the ones you don't want.

However, technology has moved on and now you have other options at your disposal. I'll explain how you can handle this problem in other ways by examining your options in the following order: OpenSQL, CDS views, and AMDP methods.

OpenSQL

As you saw in Chapter 2, the options when coding an OpenSQL `SELECT` statement in a traditional ABAP program have increased dramatically and continue to increase with each support stack. You can, for example, have `CASE` statements embedded within the `SELECT` query, but not (as of the time of writing) a convoluted `IF` statement like the one in the preceding example.

The correct way to proceed is to initially pretend to yourself that the extended OpenSQL syntax is the only option you have. Then, you have to think: Is it possible to limit the number of records coming back by using a `CASE` statement or one of the other new features? If so, then go for it; as noted earlier, OpenSQL is always the preferred choice. However, in this case it seems the business logic is a bit too complicated. You *might* be able to use a `CASE` statement to flag bad records by setting the clearing date to 12.31.9999 or some such, but you would still have to send those records back to the application server to be deleted—rather defeating the purpose of the exercise, which is to remove those records during database processing.

Therefore, you have to start researching other options.

CDS Views

As mentioned earlier, a CDS view is rather like a `SELECT` statement on steroids. The so-called DDL artifact that lives in the ABAP repository and is created via

ABAP in Eclipse is a small file consisting of the code for one complicated `SELECT` statement, with features that OpenSQL cannot yet handle.

Again, the mental exercise you should go through is to pretend that the AMDP option doesn't exist. The reason for this is that although an AMDP has far more options, a CDS view is *open*; that is, it works with any database. This has been one of SAP's strengths over the years: regardless of what database you chose to use underneath SAP, your ABAP code would still work, and you could even change databases with no negative consequences. This sort of benefit is too good to give up lightly, which is why SAP strongly urges you to stay open whenever possible, only using SAP HANA-specific features as a last resort.

Now, look at what's possible in the DDL code, which will generate a CDS view. Can you have `IF` statements here? As it turns out, although the options available to you when using a DDL to create a CDS view have expanded greatly, `IF` statements are not yet among them; you're still limited to the `CASE` statement.

Just to be clear about this: If the CDS view can handle your desired logic, then this is the option you should take. In addition, if CDS views (or OpenSQL) are ever enhanced with extra features, then you should reexamine everything you've coded in AMDP methods or CDS views to see whether you can reimplement your solution in a more open way.

AMDP Methods

As you now know, an AMDP is a method that runs in the database, as opposed to on the application server like a traditional method. Also, you will have by now realized that the language used in such AMDP methods is SQLScript (as opposed to ABAP), which makes some people quake in their boots.

In this example, you know what you want: to remove certain records before the result set is sent back to the application server. Wouldn't it be nice to just have the exact same ABAP code as before, this time running in the database? Sorry; it doesn't actually work like that.

You can have `IF` statements inside an AMDP for setting local variables and the like, but that doesn't help much in this situation. You cannot have loops, either. The easiest solution is to create a method (the definition of which is shown in Listing 15.19) with two returning parameters: a table of cleared items created

before the write-off date (that match the rules for such items) and a table of cleared items created after the write-off date.

```

CLASS zcl_monster_debts DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    INTERFACES if_amdp_marker_hdb.

    TYPES: m_tt_cleared_items TYPE STANDARD TABLE OF zmonster_debts.

    METHODS get_cleared_items
      importing
        value(id_mandt) type mandt
        value(ID_WOFF_DATE) type SY-DATUM
        value(ID_LAST_EXTRACT_DATE) type SY-DATUM
      exporting
        value(ET_OLD_BAD_DEBTS) type M_TT_CLEARED_ITEMS
        value(ET_NEW_BAD_DEBTS) type M_TT_CLEARED_ITEMS.

  PROTECTED SECTION.
  PRIVATE SECTION.

ENDCLASS.

```

Listing 15.19 AMDP Method Definition to Get Monster Cleared Items

Inside the AMDP implementation (see Listing 15.20), you just need two database reads to fill up the two export tables. Naturally, you could have used equivalent SELECT statements in a method that runs in the application server, but that involves going on a trip to the database twice, and that can sometimes take longer than just going once to get a load of records and then discarding some. This way, all the work is done in the database, and only the desired records are sent back to the application server.

```

CLASS zcl_monster_debts IMPLEMENTATION.

  method GET_CLEARED_ITEMS BY DATABASE PROCEDURE
    FOR hdb
    LANGUAGE SQLSCRIPT
    USING zmonster_debts.

    et_new_bad_debts = SELECT
      mandt as mandt,
      belnr as belnr,

```

```

    cpudt as cpudt,
    augdt as augdt,
    amount as amount,
    waers as waers
from zmonster_debts
where mandt = :id_mandt
and cpudt >= :id_wofff_date
and cpudt >= :id_last_extract_date;

et_old_bad_debts = SELECT
    mandt as mandt,
    belnr as belnr,
    cpudt as cpudt,
    augdt as augdt,
    amount as amount,
    waers as waers
from zmonster_debts
where mandt = :id_mandt
and cpudt < :id_wofff_date
and cpudt >= :id_last_extract_date
and augdt >= :id_wofff_date;

endmethod.

```

ENDCLASS.

Listing 15.20 AMDP Method Implementation to Get Monster Cleared Items

If you do an ST05 SQL trace on that method when the data was retrieved from the database, you'll find that there was no `FETCH` operation, as is normally the case when performing a `SELECT` statement against a traditional database. Instead, the result looks like Figure 15.10, which shows a call to the AMDP inside the database.

07:18:02.159	221	0	CL_SEDI_ABAP_DOC_CHECK-----CP	REPOSRC	SELECT WHERE "PROGNAME" = "ZMONSTER_DEBTS-----" AND "RSSTATE" = "A"
07:18:02.162	12	0	ZCL_MONSTER_DEBTS-----CP		SET CLIENT INFO (EPP_COUNTER=09, SAP_PASSPORT="(230 bytes)")
07:18:02.163	12-413	0	ZCL_MONSTER_DEBTS-----CP	"ZCL_MONSTER_DEBTS->GET_CLEAR	CALL "ZCL_MONSTER_DEBTS->GET_CLEARED_ITEMS#1tb2#20150202071427" (?, ?, ?)
07:18:02.176	534	1	CL_ABAP_LIST_PARSER-----CP	TRDIR	SELECT WHERE "NAME" = "ZMONSTER_DEBTS"
07:18:02.177	2-221	0	SAPMSYD		COMMIT WORK ON CONNECTION 0

Figure 15.10 ST05 Trace on an AMDP

That was a very simplistic example, but hopefully you get the idea: Whenever you find that the only way to get what you want from the database using traditional methods like OpenSQL cannot work without breaking a golden rule (such as getting back more data than you actually need), then it's time to think about pushing the code down to the database using the guidelines in this section (i.e., use the extended syntax of OpenSQL if possible; if not, then use a CDS view if possible;

if not, then use an AMDP). In every case, the idea is to do work you would have normally performed on the application server (like filtering out records) on the database.

15.5 Other Modifications to ABAP for SAP HANA

Regardless of how much or little code pushdown you end up doing, there are some further changes you need to make to your ABAP program design to get the most out of your shiny new SAP HANA database.

This is rather like when electric light and central heating were invented: The architects could have kept on designing houses with big candelabras on the ceiling and huge open fireplaces, and sometimes this is still appropriate (e.g., for the baron's castle)—but not for one-bedroom apartments. You have to change the way you build things to take advantage of the new technology.

We programmers spend an awful lot of time worrying about performance, which influences the design of both our programs and any Z database tables we create, and such performance-related features need to be examined to see if they are really appropriate anymore once we're in the SAP HANA world. In this section, you'll learn three ways you should adjust your ABAP coding to account for SAP HANA: the way you design your database tables, the way you avoid coding database-specific features, and the way you make changes to databases using `SELECT` statements.

15.5.1 Database Table Design

There are two major ways that SAP HANA changes how you will design database tables; one has to do with redundant storage and the other with secondary indexes. Each of these is discussed next.

Redundant Storage

From the time SAP was first created to the present day, ABAP developers have often been forced (for performance reasons) to store the same data redundantly in many different places. This of course means that the total space occupied by the database is much larger than it needs to be.

The example SAP gives is in the area of finance, where traditionally (in order to satisfy the needs of assorted reporting requirements) the data for each financial transaction had to be stored in literally dozens of tables—BKPF, BSEG, BSID, BSIK, COSP, the list goes on and on—and also in an SAP BW system, and also in indexes within SAP, and aggregates in SAP BW. In other words, the same data was stored again and again and again, because otherwise there was no way to get that data out in a timely manner.

The advent of SAP HANA changes everything: You only need to store the data in one place, because any query on anything comes back in no time at all. The fewer redundant fields (and tables) you have, the less database space you occupy, and so the cheaper things are for the company as a whole. SAP claims that it managed a 75% reduction in database size when it did its own internal migration to an SAP HANA database. In addition, during every business transaction there's an overhead, however small, to updating the database in more than one place with the same data; therefore, removing redundant storage improves performance when writing new records to the database.

Many times, you see the same field in transactional tables as in a related master data table (e.g., material group in the sales order line item table VBAP). Why does that need to be there? In theory, you can read that value from the material number, which is also in VBAP. Someone must have thought that there would be so many queries needing the material group value that it would be much faster just to have that value in the VBAP table. In the same way, the customer number does not really need to be in VBAK; it could be derived from VBPA using the sales order number. However, it's such an important piece of information that it got replicated in the main header table.

You may have done the same thing yourself: If you can derive a field from a value in a line item table but you need to go to one or more other database tables to retrieve what you want, then it seems so much easier to just store that value redundantly in the line item table. Take characteristics in a sales order, for example. It's fairly painful to get them back in a custom program, and if one characteristic is really important and needed all the time, then why not put it in a Z field in VBAP? However, if suddenly it didn't matter how many database tables you queried to derive one value from another, then you could knock out a whole bunch of redundant fields from your database tables.

All of this has three implications for what you need to do when designing ABAP programs:

- ▶ First, you no longer need to add redundant fields to any new database tables that you create for the sole purpose of improving performance.
- ▶ Next, due to the absence of such redundant storage fields, you don't have to spend any time considering which table to get any given piece of data from, because it will only be in one place. For example, in the traditional way of doing things, you might have to query both BSID and BSAD—because you know a database record exists in one of these tables, but not in which one.
- ▶ Finally, because there are fewer (hopefully no) redundant fields in transactional tables, you'll need to read more master data tables to fill in the blanks with information such as the material group for a material.

Avoiding Inefficient Retrieval of Master Data

It has always been (and still will be, in the SAP HANA world) a no-no to do an inner join between transactional data and master data. This is because there's no point in having 100 sales order lines all with the same material group joined with the material master; the result would be the same for each line.

Such a query would be a lot faster with SAP HANA, but it still would not be a sensible thing to do. Just because the extra database access time is now negligible, that's no excuse for abandoning common sense! (Though I suspect a lot of people will do just that.)

Secondary Indexes

In traditional SAP database tables, the primary key is often some sort of number: sales order number, material document number, monster number, and so on. In more modern SAP database tables (e.g., BOPF, BRFplus, etc.) the primary key is some sort of GUID. That's wonderful if you happen to know the sales order number or the GUID, but more often than not you don't. Database queries are usually along the lines of "Give me all the sales orders for customer Joe Bloggs for May" or "Give me all the monsters named Fred who went on a rampage in June."

Doing a database `SELECT` on a field other than the primary key would have caused a full table scan, and so you would have created secondary indexes on such fields as customer number or monster name. This made data retrieval very fast, but the downside was twofold: Every time a record was saved, there was an additional overhead of updating the index, and in some tables with lots of indexes, the stor-

age space for the indexes was almost as big as for the whole table itself. In addition, the SAP Basis department had to spend a lot of time each month reorganizing indexes that became fragmented over time.

To use an analogy, consider a book about monsters, and the chapters are organized from least scary type of monsters to the scariest. If you know the scariness level you desire, you can jump straight to the correct chapter. However, if you want to know about green monsters, you either have to flick through the whole book looking for the word "green" (a full table scan) or use the index at the back of the book, look up the word "green" there, and then look at the pages the index recommends. However, if suddenly you were able to read a book at 10 million billion words per second, then you could flick through the book and find all the green monsters in no time at all. You wouldn't need the index, so the book wouldn't need the index, so the book could be 5% shorter. In other words: Removing indexes means that you need less database space.

SAP HANA can read the database table book at such a high speed because it's very different from the traditional databases you're used to, not least because it's column based as opposed to row based. This means that you do not need indexes at all anymore; it's as if you've manually created an index on every field, but without any of the drawbacks just mentioned.

Of course, everything has a caveat. If you read the standard SAP documentation, then you will see SAP hedging its bets and saying that secondary indexes are *usually* not required. That was a bit wooly, so I went hunting and found this quote from SAP expert John Appleby (for more details, see the "Recommended Reading" box at the end of this chapter):

In certain obscure scenarios where you find a performance problem, a secondary index can help. This is only in OLTP scenarios, where you have complex joins and only return a small subset of data from the table.

Ninety-nine percent of scenarios will never benefit from an index. They take up space and will slow insert operations, so should be avoided.

What practical implication does this have on your actual database table design? You will find when you create a database table in SE11 that a new tab has appeared: DB-SPECIFIC PROPERTIES (Figure 15.11). Here, you can see that the default setting for the new table is for column-based storage inside the database, as opposed to the row-based storage that was always the case in the past.

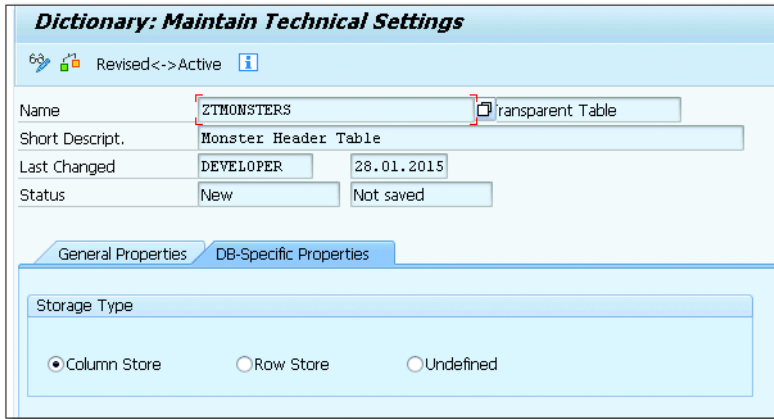


Figure 15.11 SE11 Technical Settings with an SAP HANA Database

All new tables default to column store, and you should leave that setting as is. During the migration from your old database to the SAP HANA database, the database tables are migrated from row based to column based according to rules determined by SAP based on SAP's knowledge of the usage of that table. After migration, you can see what tables are still row based by looking at the ROWORCOLST field in table DD09L (R = row based, C = column based).

No table should have the UNDEFINED radio button selected! This is like when you create a new transport request: It starts off in an undefined state, and it's unusable until you pick DEVELOPMENT/CORRECTION or REPAIR. In the same way, a database table is no good until either COLUMN STORE or ROW STORE is specified. Luckily, in this case UNDEFINED is never the default value for new objects.

The only time I've ever heard experts advising to use row-based storage is if (a) the data is never actually going to be stored in the database (which would make it an odd sort of database table) or (b) SAP Support personnel instruct you to make that setting.

15.5.2 Avoiding Database-Specific Features

The vast majority of your existing custom code most likely uses OpenSQL for all the database retrieval. However, it's worth hunting through your custom code base for database-specific code, because this is 100% for sure going to break when you migrate to SAP HANA.

This is, unfortunately, partly a manual task. When you have to resort to using database-specific features, it's due to a specific problem only your organization's SAP system is having, and thus it's difficult for SAP to provide an automated tool to look for such occurrences. (Having said that, there are some automated checks that can be performed using the Code Inspector; these are covered in Section 15.5.3.)

One example of database-specific code in a custom program (it can also be found in SAP standard programs) is "database hints," which you can add to your ABAP `SELECT` statements, as in Listing 15.21.

```
SELECT SINGLE monster_number
  FROM zt_monsters
  INTO ld_monster_number
  WHERE monster_name EQ ld_monster_nmae
  %_HINTS ORACLE 'index(ZT_MONSTERS "ZT_MONSTERS~Z01")'.
```

Listing 15.21 `SELECT` with a Database Hint

Normally, the database optimizer chooses what index to use, but if that optimizer keeps choosing the wrong index, then you can force the issue using a `HINT` (i.e., make the optimizer use the index you think is best).

The recommendation from SAP has always been never to use database hints, on the grounds that one day you might want to do a database migration. In this situation, the cows have certainly come home to roost; this is precisely what you're doing—moving to SAP HANA—so you need to get rid of all these `HINTS` statements.

The other database-specific type of coding you might come across in custom code is NativeSQL. This is when the specific database you are using has SQL commands not available in OpenSQL, and you want to take advantage of this, so you write some Oracle- or Microsoft-specific SQL inside the ABAP constructs `EXEC SQL/ENDEXEC`.

Naturally, SAP recommends against this as well, and there are two good reasons that you should replace such NativeSQL with OpenSQL:

- ▶ Such a query will no longer work once you've migrated to SAP HANA.
- ▶ With all the advances that ABAP 7.4 brings to OpenSQL, it's more than likely that the missing functionality that caused the original programmer to use

NativeSQL is missing no longer, and now you can achieve the same thing using OpenSQL.

15.5.3 Changes to Database SELECT Coding

There are various changes you need to make to the `SELECT` statements you code, to make sure you don't get any unpleasant surprises in your current custom programs when you switch to an SAP HANA database.

First, there is some good news: When you move to an SAP HANA database, virtually all existing code you've written to access database tables will still work, and a lot of it will run a lot faster without you lifting a finger. Moreover, as time goes by SAP keeps making improvements under the hood (e.g., in SAP NetWeaver 7.4 SP 5, the performance in an SAP HANA database of `FOR ALL ENTRIES` and `SELECT SINGLE` was improved without needing any code changes at all).

However, life is not all beer and chocolate. If your `SELECT` statement is badly written in the first place, then moving to SAP HANA will most likely make the performance a billion times worse. This is the opposite of what you might expect. Two examples of badly-written `SELECT` statements are those that select lots of columns you do not need and nested `SELECT` statements. Those are bad practices that have a negative impact on any database system but have a really bad impact on an SAP HANA database. In addition, in some rare situations you may find your existing programs behaving very strangely after the switch to SAP HANA, coming up with all sorts of incorrect results.

For this reason, you should know about some Code Inspector checks that will help you find and fix such problems. An important thing to note is that you can run these checks before you migrate to SAP HANA. In fact, as soon as those checks are available in your system, run them anyway and make the recommended fixes, even if you're not planning to move to SAP HANA anytime soon (or ever). The benefits of such corrections are greatest for organizations that are transitioning to SAP HANA, but the changes will make your programs more robust no matter what database system you're using. The checks discussed can be switched on by going into the Code Inspector transaction (SCI) when you're setting up your check variant (i.e., making a list of possible errors or performance killers in your code that you want to check for).

As in Chapter 4, which talked about the ABAP Test Cockpit, the headings in this section identify the Code Inspector check by name and then are followed by a series of three numbers. Each of these numbers corresponds to the support stack in which the new Code Inspector check was added, and the order of the numbers indicates whether the system in question is SAP NetWeaver 7.02, SAP NetWeaver 7.31, or SAP NetWeaver 7.4. For example, 12/5/2 means that the check becomes available in SAP NetWeaver 7.02 SP 12/SAP NetWeaver 7.31 SP 5/SAP NetWeaver 7.4 SP 2.

Select for Pool/Cluster without Order By (14/9/2)

As every programmer who has dealt with the financial area of SAP knows, table BSEG is not a real, transparent table (even though it looks like one in SE16) but is in fact a cluster table; that is, all the data except the key fields is stored in the database in one huge string.

In a similar fashion, some tables in SAP are called pool tables; they are green and have pockets, and you try to knock balls into those pockets. No, wait, that's not correct. In fact, they're several similar tables stored inside the database in one big table.

When you migrate to SAP HANA, both types of table become real, transparent tables so that what you see currently in SE16 will actually be the way the data is stored in the database. The difference this makes for your programming is that with a cluster or pool table you could be sure that the data read from the database would come back in the order of the primary key—so there was no need to sort the data in your program. With SAP HANA, all bets are off: The rows of data can come back in any order. This can cause problems if you then try to do a `BINARY SEARCH` or similar, so an error is now raised if you don't put an `ORDER BY` addition to your database selection on such tables.

The Code Inspector check you need to switch on in Transaction SLIN is shown in Figure 15.12.

After that check has been turned on, whenever you do an extended check on one of your custom programs you will be alerted to the fact you have broken this rule. More realistically, if you do regular mass checks of all your custom programs as described in Chapter 4, then you will be alerted to every single custom program that has this problem, so that you can work through them all and correct them.

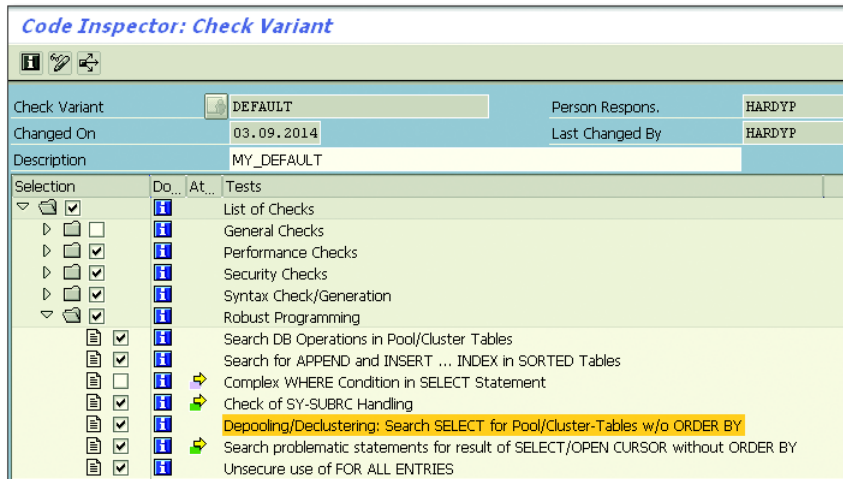


Figure 15.12 Check for SELECT on Cluster/Pool without Order By

Check on Statements Following a SELECT without Order By (14/9/3)

This check was mentioned back in Chapter 4 when discussing the ABAP Test Cockpit; now the subject is revisited with an SAP HANA twist. You will by now be very familiar with the idea that in most databases (Oracle, for example) the result of the query (usually) comes back already sorted by the primary key. You can't rely on that, however; sometimes the results come out in the expected order in DEV and production, but backwards in TEST. You can't tell what will happen when using other databases than the one you normally use, and you certainly can't rely on it using SAP HANA. Always sort it yourself to be sure.

Therefore, if after a SELECT the ABAP code does a BINARY search on the retrieved data, expecting it to be sorted already—or reads the first line expecting that to be the row with the lowest primary key—then everything will fall to pieces. To protect yourself from this sort of situation, switch on the check to warn you of code in which you retrieve data but then do not sort it (Figure 15.13).

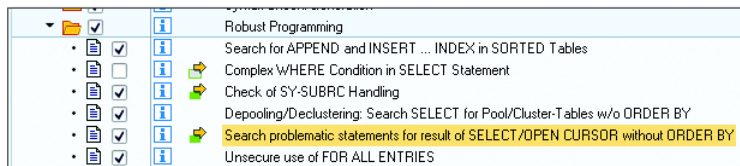


Figure 15.13 Check for Dodgy Statements after SELECT without Order By

Naughty Reads on Underlying Tables (15/10/4)

As you'll recall from a few paragraphs ago, cluster and pool tables are not actually real database tables, even if they look the same as normal tables from SE11 and SE16. In fact, traditionally these have been stored in the database by real, transparent tables; for example, the BSEG data actually lives in cluster RFBLG, so it's possible in ABAP code to do a direct SQL read on RFBLG as opposed to BSEG. Neither one is a real table, like VBAK—but after conversion to SAP HANA, BSEG is a real table and RFBLG is empty.

The Code Inspector check for database reads on such false tables is shown in Figure 15.14.

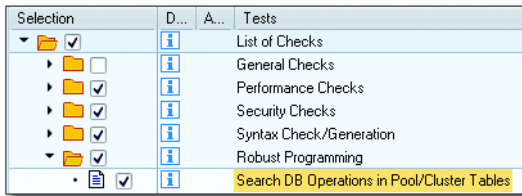


Figure 15.14 Check for Direct Reads on Underlying Tables

The check shown in Figure 15.14 has been created to spot such SQL statements that will work fine up until a migration to SAP HANA and then not at all afterwards.

15.6 Summary

As you might imagine, in this day and age it would be virtually impossible to write an SAP-related book (especially a highly technical one) without mentioning the (relatively) new, all-singing, all-dancing, in-memory database SAP HANA. This chapter showed you how ABAP programming is changing because of SAP HANA. You can also expect fast developments in this area; ABAP version 7.4 was the first to be specifically enabled for SAP HANA, and it entered general availability in May 2013. With SP 2, bottom-up development was enabled, and in February 2014, with SP 5, top-down development came to town. SP 9 was released at the end of November 2014, many announcements about the new features were made at SAP TechEd (or what was briefly named "d-code" in a sort of "New Coke" moment), and already SAP is hinting about what's going to come as far ahead as

SP 12. So SAP clearly has a long-term plan of what's intended to be enabled in each of the next three support packs. Beyond SP 12, it's (to list just one example) planned to have an automatic connection to an authority check inside a CDS view.

Historically, these support packs have been released twice a year—around about May or June (for SAPHIRE) and around about November or December (for SAP TechEd). Recently, these support packs appear to be coming out faster and faster, which is what you would expect in order for SAP to be able to stay ahead of the game—now that the competition has belatedly realized what's going on and is now attempting to catch up.

All this to say: It's very clear that even though you can already do an enormous amount in both SAP HANA views and AMDPs, both are going to have new features added continually, at a fairly rapid pace.

Recommended Reading

- ▶ *ABAP Development for SAP HANA* (Schneider, Westenberger, Gahm, SAP PRESS, 2014)
- ▶ Best Practice Guide—Considerations for Custom ABAP Code During a Migration to SAP HANA: <http://scn.sap.com/docs/DOC-46714> (Eric Westenberger)
- ▶ ABAP Managed Database Procedures—Introduction: <http://scn.sap.com/docs/DOC-51612> (Jens Weiler)
- ▶ New Data Modeling Features in SAP NW ABAP 7.4 SP 5: <http://scn.sap.com/community/abap/eclipse/blog/2014/02/04/new-data-modeling-features-in-abap-for-hana#comment-561176> (Christiaan Edward Swanepoel)
- ▶ Six Golden Rules for New SAP HANA Developers: <http://scn.sap.com/community/hana-in-memory/blog/2013/12/29/6-golden-rules-for-new-sap-hana-developers> (John Appleby)

We've got twenty-first-century technology and speed colliding head on with twentieth- and nineteenth-century institutions, rules, and cultures.
—Amory Lovins, *A 40-Year Plan for Energy* (Ted Talks)

16 Conclusion

In a paraphrase of a famous saying, back in 2001 Shai Agassi promised to “kill all ABAP programmers.” I don’t think he meant it literally, but at that point it looked like Java was going to be king. There was even talk of running some automated process to transform the entire SAP ABAP code base to Java (I think that was a little optimistic).

With the benefit of 20/20 hindsight, we can see that this did not happen; SAP programmers still use ABAP the vast majority of the time (and Agassi’s electric car company did not do too well either). Instead, as we have seen throughout this book, SAP has been throwing an enormous amount of effort into improving the ABAP language with new features and adding whole new chunks of technology, like BRFPplus and BOPF.

The killer argument that ABAP is future-proof is the fact that it can be used to take advantage of SAP’s new poster child, SAP HANA. When a lot of ABAP programmers realize this, they breathe a sigh of relief and say, “Thank goodness for that! My job is safe, and everything is business as usual.” Well it’s not, and there are two reasons for this.

If you only stick to the technology that was around the year you started programming in SAP—such as DYNPRO programs, SAPscript, and ALV reports (I have even seen development departments in which they still favor `WRITE` statements)—and don’t even consider “weird,” “new” things (such as object-oriented programming and assorted new technologies, like the ones showcased in this book), then you are committing professional suicide.

People born today (and, indeed, the people entering the workforce right now) don’t tolerate ugly gray screens and years between new versions of programs.

They want everything to look as good as the apps on their plethora of mobile devices and updates to come out every nanosecond.

The SAP tools from circa the year 2000 simply can't deliver either of those requirements, so hopefully the contents of this book have shown how you can use various new or "not so new, but still new to lots of people" tools to do three things, in increasing order of importance:

- ▶ Make the user interface look better
- ▶ Dramatically speed up the development cycle
- ▶ Make programs less fragile so that they do not break when exposed to that dramatically increased development cycle

The worst-case scenario is that you stick with DYNPRO screens and what-have-you because "we've always done it this way, and it works," and you are made redundant after fifteen years at age 50, and then you try and get another SAP programming job. Chances are that at the interview your potential new employer will start asking you questions full of alphabet soup (like AMDP and BOPF and BRF) and will talk about "the decorator pattern" and as far as you are concerned they might as well be asking you questions in Klingon.

As much as it fills people with horror, if you accept that these new technologies are vital for your future, then you *must* step out of your comfort zone. Some SAP HANA artifacts, for example, can only be created using ABAP in Eclipse, a development environment that was strongly opposed when it came out on the basis of "What's wrong with SE80?" In the same way, working with user interface technology such as SAPUI5 really requires an understanding of JavaScript.

Hopefully, in this book you will have seen that all of this new technology is nowhere near as scary as it might seem at first glance. What I would like you to do is not simply put this book down and say "That was interesting." Instead, actually think of ways to use the contents—tomorrow, in your day-to-day job. That might hurt, and at the start it might even slow you down, but even in the medium term (let alone the long term), you will be really glad that you did.

We are authors. And one thing about authors is that they have readers. Indeed, authors are responsible for communicating well with their readers. The next time you write a line of code, remember that you are an author, writing for readers who will judge your effort.

—Robert Martin, Clean Code

Appendices

A	Improving Code Readability	707
B	Making Programs Flexible	711
C	The Author	719

A Improving Code Readability

As Robert Martin said about programmers, “We are authors.” This is true in that other programmers (and indeed our future selves) will read the code we have written and will need to understand it in order to make the never-ending stream of changes that the business will require. With that in mind, this appendix will examine whether the improvements to the language to make programs more compact are counterproductive due to making programs more difficult to understand and thus change. Section A.1 will examine the nature of the problem, and Section A.2 will present a possible approach to having the best of both worlds.

A.1 Readability vs. Concision

If you paid attention in Chapter 2, you will have noticed that most of the new ABAP 7.4 constructs involve performing tasks we have always been able to perform, only with fewer lines of code. My mother was a shorthand typist, an unknown concept today. The idea was that the boss dictated a letter to the secretary, and because the boss was speaking so fast the secretary had to write down what the boss was saying in a sort of secret code that abbreviated words and sentences dramatically.

The following is an example of this shortening of sentences being applied in ABAP 7.40, with a change made that maybe does not achieve all it set out to do:

```
MOVE-CORRESPONDING ls_blue_monsters TO ls_green_monsters.
```

can be shortened to

```
ls_green_monsters = CORRESPONDING#( ls_blue_monsters ).
```

The second version is not really that much shorter, and it also works slightly differently, because `MOVE-CORRESPONDING` leaves differently named columns in the target structure untouched, whereas `CORRESPONDING #` blanks them out. The latter behavior is often not desirable, so in SP 8 a new addition was added. The syntax is now as follows:

```
Ls_green_monsters = CORRESPONDING#( BASE (ls_green_monsters) ls_blue_monsters ).
```

This now works the same as `MOVE-CORRESPONDING`, but it's longer—so if the target structure is already partially filled and you want to keep the values, then you might as well not bother and should stick with `MOVE-CORRESPONDING`.

The most compact code of all is of course machine code, which reads like this: `A098E564C63E231A45C`. If you write a program in this fashion, then it ends up very short indeed, but of course nobody can understand what it means. This is why we have high-level languages like ABAP, which read like English and then get compiled into a form the computer can understand but humans cannot. ABAP has often been attacked for being very verbose when compared to other languages—for example, `ADD 1 TO 1d_number.`, as opposed to `Number++`, which a lot of experts seem to think is better. On the other side of the fence, academics such as Donald Knuth have aimed for *literate programming*, in which the program is supposed to read like a novel or a newspaper.

In essence, the benefit of “verbose” language such as we have been used to in ABAP is that the more like English it is, the easier it is to understand and thus to change without breaking something. Contrariwise, as Tweedledum would say, the benefit of the shorthand statements being introduced in ABAP 7.4 is to enable you to achieve programming tasks in fewer lines of code, thus boosting productivity; also, you can see more of the program at once, so you don't have to page up and down so often.

Are these benefits mutually exclusive? Can you have the positive aspects of each approach all at once—that is, have your cake and eat it too?

A.2 The What vs. the How

I think that we can, and the reason is that there are two things that we describe in a program, each for different audiences.

When you're tasked with adding some functionality or fixing a bug in someone else's program, a program you've never seen before, clearly the first thing you need to understand is *what* it is doing (either what it is trying and failing to do or what it needs to do in addition to what it currently does).

This “what I am doing” information needs to be in plain English so that you can grasp it in a hurry. Programmers don't read written documentation in Word documents and the like—no one who works in IT does—so the information has to be in the code.

When it comes to writing the code for *how* to do any given task, the audience is quite different; this time, you're talking to a machine. Obviously, a programmer is still going to read that code as well and try to get his head around it, but at this point you're looking at the nuts and bolts of programming and can be as technical as you want, as opposed to the abstract concepts talked about in the "what" context.

Here's an example: In Chapter 2, you learned how the `CAST` expression could simplify the code for building an internal table containing the components of a structure. In traditional ABAP, the code was as follows:

```
DATA lo_structdescr TYPE REF TO cl_abap_structdescr.
lo_structdescr ?= cl_abap_typedescr=>describe_by_name( 'ZSC_MONSTER_
HEADER' ).
DATA lt_components TYPE abap_compdescr_tab.
lt_components = lo_structdescr->components.
```

As noted, in 7.4 you can do this all in one line by using the `CAST` constructor operator.

```
DATA(lt_components) = CAST cl_abap_structdescr(
cl_abap_typedescr=>describe_by_name( 'ZSC_MONSTER_HEADER' ) )-
>components.
```

The second block of code is certainly shorter, but is it less readable? In this case, neither is particularly readable, but in some examples you have seen, the new shortened code makes traditional ABAPers struggle, so much so that they might not even be able to maintain code written by their more "thoroughly modern" colleagues.

Look at this a different way: Why do you want the components? It turns out that every field in this structure is on a DYNPRO screen, and you want to read all the values in. We don't want to do this one at a time, so you need the components so that you can loop through them reading each value.

Robert Martin said that some programming languages force the following sort of structure:

- ▶ *To process the user input, we need* the current screen values.
- ▶ *To get the current screen values, we need* to read them off the screen.
- ▶ *To read the values of the screen, we need* a list of fields to read.
- ▶ So forth and so on.

The idea is that the top line in the structure is a routine, which calls a routine represented by the second line in the structure, and so on. You don't need to be quite as specific as in this example, but try and structure the code as follows:

```
ls_monster_header = the_values_on_the_screen( ).  
METHOD the_values_on_the_screen.  
    DATA( needed_fields ) = fields_of( 'ZSC_MONSTER_HEADER' ).  
    rs_monster_header = screen_values_of( needed_fields ).  
ENDMETHOD. "The Values on the Screen"
```

The method name tells you that you're filling up the header structure from the screen values, and the body of the method tells you that in order to do this you first need to know what fields you're looking for and then get their values. Inside the lower-level methods, you can have the references to the SAP-specific classes that actually do this work for you, such as `CL_ABAP_STRUCTDESC` and the `DYNPRO_READ_SCREEN` function. The higher-level methods are the "what," and the lower level methods are the "how."

In conclusion, my recommendation is to compress the "how" part, even if it is a bit difficult to follow—but make sure to keep the "what" part nice and clear.

B Making Programs Flexible

One of the golden rules of writing good code is *make your programs flexible*, and one of the most important ways to do this is to avoid hard-coding. In a presentation I once saw from someone from the oil company Shell, they noted, "Once you put in electronic concrete, it is hard to get it out." What do they mean by that? As an example, look at the below code, which is fairly representative of what you might see in an SAP program, custom or standard.

```
SELECT SINGLE auart
  FROM vbak
  INTO ld_order_type
  WHERE vbeln EQ ld_order_number.

CASE ld_order_type.
  WHEN 'ZX23'.
    "Do Something
  WHEN 'ZX99'.
    "Do Something else yet again
  WHEN 'ZPAB'.
    "Do yet another thing
  WHEN OTHERS.
    "Raise an error message
ENDCASE.
```

This is fairly straightforward; the logic varies by order type. However, this is "electronic concrete" (in OO terms, it violates the open-closed principle), because you have to change the code in the following situations:

► **When customizing values change**

Customizing values can change over time. One day, a business analyst may create a new order type, ZX24, which is really similar to ZX23, so the program needs to treat it the same way. You have to change your program to say `WHEN 'ZX23' OR 'ZX24'`. (The removal of customizing values is less common and less of a problem, though it can create dead branches of code that never get executed but still get checked during an upgrade.)

► **When customizing values are different in different systems**

In this case, you might copy the program from the master system to a subsidiary's SAP system and find that the subsidiary has the exact same concept as a ZX23 but calls it a Y999. You could change the hard coding in the program in the target system or have a constant and change the value of that constant in the

target program, but either way the two programs are now different, and copying over any updates now becomes fraught with danger.

► **When the same customizing value means different things in different SAP systems**

Once again, you might copy your program from the master system to a subsidiary system—and this time the target system does have the value ZX23, but it means something totally different. This is a worse problem, because your program will react to the value but do something unexpected. You can do the same fix when customizing values are different in different systems, but with the same problem.

As an aside, this approach also makes code hard to read. After all, what is a ZX23 anyway? You tend to have a limited number of characters in customizing values (sometimes only one character), so it's hard to make those values meaningful. Other programmers will look at the code and struggle to understand what's going on.

Once my company sold a business unit to another company, and because I had written most of the software I was hired out for a week to help the new company modify my programs for use in their system. It turned out that one of the major programs only had about five lines that needed changing, because I had hard-coded some customizing values of delivery types and the like, and it was just a question of replacing those values with the new company's equivalent values, and that was that. There are three ways I could have reacted to this:

- I could have thought, "Oh no, if I had hard-coded much more and written the program in an inflexible way, we could have had a lot more consulting hours out of this."
- I could have said, "Whoopee! How good is this! I wrote this program to be so portable that it needs minimal changes when transported into to a totally foreign SAP system. Hang out the flags!"
- What I actually thought was, "Hang on; if I can write a program that's 95% portable, then why not one that's 100% portable? After all, that's what SAP does. Programs like SAPMV45A (VA01) work in any company."

When looking through custom programs, it's often amazing how many assumptions are made; programmers take things for granted because they are always true in their company.

Example

The point should now be clear: hard-coding is bad. Now consider an example that shows how to fix this problem. As bizarre as it sounds, you're going to create a customizing table of customizing entries. Say that you have five different categories of monsters. The categories are pretty much the same all over the world, but the customizing values in each country's SAP system are different. You need a table to say what data element you're talking about and what possible categories can apply to that data element.

In Figure B.1, you can see how you assign values to a data element. You could also have other values, like for an extra scary monster or gigantic monster. The underlying database table would look like Figure B.2.

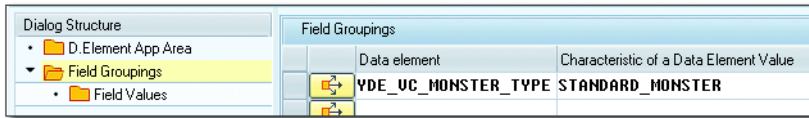


Figure B.1 Adding a Meaningful Value for a Data Element

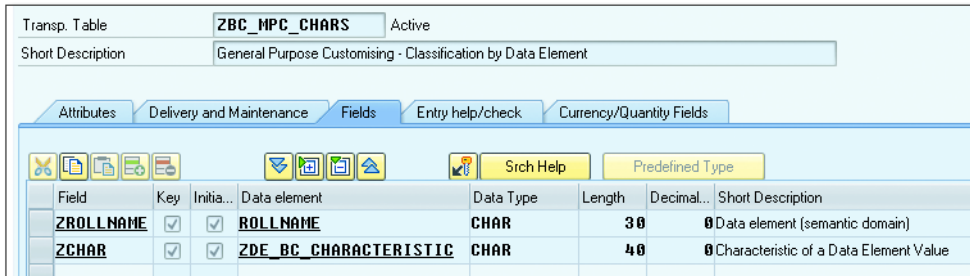


Figure B.2 Database Table for Linking Meaningful Values to Data Elements

The next step is to store the exact values each country is using to describe the same sort of monster. As you can see in Figure B.3, each country has chosen a different four-digit code to describe the same thing, and each of those codes is meaningless, in that it's impossible to guess that ZZT5 represents a standard monster. However, a program specification would say something like "When you encounter a standard monster, do such and such," as opposed to "When you encounter a ZZT5, do such and such," so you want to see the former approach in your code.

Counter var.	Value of a Data Element	Chr	CoCd	COAr	POrg	SOrg	Plnt	BusA	DChl	Dv	Other value	Program Name
1	4321	DE										
2	P45X	GB										
3	ZZT5	US										
4	MSTR	AU										

Figure B.3 Adding a Country-Specific List of Customizing Values

The underlying database table would look like Figure B.4.

Field	Key	Initia...	Data element	Data Ty...	Length	Decimal...	Short Description
ZROLLNAME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	ROLLNAME	CHAR	30		0 Data element (semantic domain)
ZCHAR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	ZDE_BC_CHARACTERISTIC	CHAR	40		0 Characteristic of a Data Element Value
ZCOUNTER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	COUNTER1	INT1	3		0 Counter
ZVALUE	<input type="checkbox"/>	<input type="checkbox"/>	ZDE_BC_VALUE	CHAR	80		0 Value of a Data Element
ZLAND	<input type="checkbox"/>	<input type="checkbox"/>	LAND	CHAR	3		0 Country of company
ZBUKRS	<input type="checkbox"/>	<input type="checkbox"/>	BUKRS	CHAR	4		0 Company Code
ZKOKRS	<input type="checkbox"/>	<input type="checkbox"/>	KOKRS	CHAR	4		0 Controlling Area
ZEKORG	<input type="checkbox"/>	<input type="checkbox"/>	EKORG	CHAR	4		0 Purchasing Organization
ZUKORG	<input type="checkbox"/>	<input type="checkbox"/>	UKORG	CHAR	4		0 Sales Organization
ZWERKS	<input type="checkbox"/>	<input type="checkbox"/>	WERKS_D	CHAR	4		0 Plant
ZGSBER	<input type="checkbox"/>	<input type="checkbox"/>	GSBER	CHAR	4		0 Business Area
ZUTWEG	<input type="checkbox"/>	<input type="checkbox"/>	UTWEG	CHAR	2		0 Distribution Channel
ZSPART	<input type="checkbox"/>	<input type="checkbox"/>	SPART	CHAR	2		0 Division
ZOTHER	<input type="checkbox"/>	<input type="checkbox"/>	ZBC_DTE_EXIT_PARAM_OTHER	CHAR	40		0 Other parameter value
ZPROGRAMM	<input type="checkbox"/>	<input type="checkbox"/>	PROGRAMM	CHAR	40		0 ABAP Program Name

Figure B.4 Database Table for List of Customizing Values by Organizational Element

As you can see, I've added as many different organizational elements I can think of that might have different rules (e.g., company codes or sales organizations might do things differently). Because it's impossible to guess exactly what might cause the rules to vary, there is also an OTHER category, which can represent anything. For example, there might be a customizing value that's different for DEV, QA, and PROD (archive settings are sometimes like this), so you could have the client number as an OTHER value.

This approach can also be used for the “magic numbers”—for example, you have a variable that is typed as the standard SAP data element for dates `DATUM`. You would firstly want to give that variable a meaningful name, like `MAXIMUM ANNUAL HOLIDAYS`. Then at runtime you would programmatically fill the value of that variable, which would be 30 in Germany and 21 in Australia.

Magic Numbers

A *magic number* is a number that appears in a program seemingly for no logical reason. For example, you may see code that says that, “If variable XYZ is greater than 65, do something; otherwise, do something else.” Naturally, you would think to yourself, “What is so wonderful about 65?”

If, instead, the code said, “If variable XYZ is greater than `RETIREMENT_AGE`, do something; otherwise, do something else” then suddenly things make a lot more sense. Moreover if the retirement age ever changes, it is easier to search code for an English phrase than the number 65, as the numeric value could pop up for all sorts of other reasons.

Now, you need to ascertain the correct value. In some programs, you can set the organizational elements based on what the user enters on an initial screen—for example, the select options for a report program or the sales organization for a dialog transaction, like `VA01`. Sometimes, this option isn’t available, but at a minimum you can usually get the country from the user’s master record: The default time zone is usually a dead giveaway as to what country the user is in.

As soon as you know the organizational element (or straight away if that isn’t relevant), your program will call a routine called something like `READ_GLOBAL_CUSTOMIZING`, which will set all such values needed by the logic in the program. This could be in a constructor for an object—for example, the code below could be inserted into a constructor method:

```
PERFORM get_customizing_range :  
  USING 'YDE_VC_MONSTER_TYPE' 'STANDARD_MONSTER'  
        CHANGING gr_standard_monster[],  
  USING 'YDE_VC_MONSTER_TYPE' 'EXTRA_SCARY_MONSTER'  
        CHANGING gr_extra_scary_monster[].
```

The vast majority of the time, you’ll want the values in a range, even if you only have one value at the moment, because you never know when more values are going to be added further down the track. In the example below, you see what would be code in the `GET_CUSTOMIZING_RANGE`; namely, a call to a method that will fill up the values in the range based on organizational-element-specific rules.

```

TRY.
  CALL METHOD zcl_bc_gpc=>get_multi_vals_by_soe
    EXPORTING
      iv_parameter = pud_paramater
      iv_char       = pud_char
      iv_vkorg      = p_vkorg
    CHANGING
      et_values    = pt_values[].

* Error Handling
  CATCH zcx_ai_application_fault INTO lo_fault.
    CALL METHOD lo_fault->error_log->show_error_log.
  ENDTRY.

```

In the preceding example, the user has entered the sales organization on the initial screen, and a class is created to read from the table. This code will try several strategies to get the value (starting with the sales organization entered, then trying the company code based on the sales organization, then the country based on clues from the organizational elements, and then the user master record if all else fails). Finally, it will look for a default entry (i.e., one for which no organizational elements at all are specified). (If everything fails, then this is an exception situation.)

The range of values is put into a global range in a procedural program or a private range in an OO program. What benefit do you get from this? Say that you wanted to keep a program the same across various different SAP systems so that you could easily move updates from the master system to subsidiary systems. Normally, you would have to write something like the following:

```

IF ( ld_land = 'DE' AND ld_monster_type = '4321' ) OR
  ( ld_land = 'AU' AND ld_monster_type = 'MSTR' ) OR
  ( ld_land = 'GB' AND ld_monster_type = 'P45X' ) OR
  ( ld_land = 'US' AND ld_monster_type = 'ZZT5' ).

  PERFORM something.

ENDIF.

```

The normal way to avoid the situation in the preceding example is to have four different versions of the program, each one changing the routine to something like this:

```

IF ld_monster_type = 'ZZT5'.

  PERFORM something.

ENDIF.

```

Both examples are horrible: The first one needs to change every time a new country comes on board or a new monster type is created, and the second one is even worse, because you have four virtually identical programs— which is bad no matter how you look at it. In both cases, anyone looking at the program still doesn't know what a ZZT5 is. Instead, use the global customizing range:

```
IF 1d_monster_type IN gr_standard_monster.  
    PERFORM something.  
ENDIF.
```

This doesn't seem like rocket science, does it? Yet, now you have some code that's readable by a human and doesn't need to be changed when a new monster type is created in customizing. Should you need to move a bunch of programs into a new SAP system, instead of spending ages hunting through all those programs changing hard-coded values, all you need to do is add any missing entries in a central place (i.e., the global customizing table). Furthermore, a lot of those entries might already be there due to another program that already makes use of them.

C The Author



Paul Hardy joined Heidelberg Cement in the UK in 1990. For the first seven years, he worked as an accountant. In 1997, a global SAP rollout came along; he jumped on board and has never looked back since. He has worked on country-specific SAP implementations in the United Kingdom, Germany, Israel, and Australia.

After starting off as a business analyst configuring the good old IMG, Paul swiftly moved on to the wonderful world of ABAP programming. After the initial run of data conversion programs, ALV reports, interactive DYNPRO screens, and (urrgh) SAPscript forms, he yearned for something more and since then has been eagerly investigating each new technology as it comes out. Particular areas of interest in SAP are business workflow, B2B procurement (both point to point and Ariba based), logistics execution, and variant configuration, along with virtually anything new that comes along.

Paul can regularly be found blogging away on the SCN site and presenting at SAP conferences in Australia (Mastering SAP Technology and the SAP Australian User Group annual conference). If you happen to ever be at one of these conferences, Paul invites you to come and have a drink with him at the networking event in the evening and to ask him the most difficult questions you can think of (preferably SAP-related).

Index

A

ABAP 7.02, 90, 96, 98, 103, 109
ABAP 7.4, 81, 97, 102, 114, 127, 131, 365,
675, 707
 new features, 81
 recommended reading, 136
ABAP Managed Database Procedures
 (→ AMDP)
ABAP Messaging Channels, 135
ABAP Push Channels, 135
ABAP Test Cockpit, 175–176
 and SAP HANA, 176
 recommended reading, 202
ABAP Unit, 137
ABAP Workbench, 35, 53, 359
ABAP_TRUE, 103
ABAP2XLSX, 457, 465
 conditional formatting, 474
 download, 461
 email, 493
 enhancement framework, 501
 enhancing custom reports, 466
 example programs, 465
 hyperlinks, 496, 498
 macros, 487
 multiple worksheets, 482
 printer settings, 471
 recommended reading, 505
 templates, 491
 testing, 478
Accleo, 76
AFTER_FAILED_EVENT, 551
ALV, 466
ALV grid, 515
ALV SALV, 407
AMDP, 659, 671–672
 Eclipse, 673
Antifragile programs, 65, 541, 711
ASSERT, 158, 276

B

BAdIs, 225–226, 235
 calling, 249
 defining, 235, 237
 filter checks, 240
 filters, 239
 implementing, 245
 interface, 244
Behavior-driven development, 171
BOOLC, 105
Boolean logic, 105
BOPF, 283–284, 541, 570
 action validations, 331
 actions, 325, 327
 and FPM, 556
 authority checks, 305
 callback subclass, 347
 change document subnode, 346
 create header node, 286
 create item node, 288
 create model classes, 291
 create object, 285
 creating/changing objects, 294
 CRUD, 336
 custom enhancements, 350
 custom queries, 295–296
 delegated objects, 344
 determinations, 306
 locking objects, 304
 recommended reading, 356
 testing, 349
 tracking changes, 342
 validations, 316, 318
 wrappers, 354
BOR object, 569
Breakpoints, 207
BRFplus, 357, 365
 call in ABAP, 386
 create application, 365
 decision tables, 378
 decision trees, 374, 377

BRFplus (Cont.)
enhancements, 401
example, 388
recommended reading, 404
rule logic, 373
SAP Business Workflow, 397
simulations, 394

Broker class, 635

Buffering, 85

Business Object Processing Framework
 (→ BOPF)

Business rules, 357, 360
ABAP, 364
BRFplus, 365
customizing tables, 362

Business rules framework, 357

C

CASE, 83, 102, 106, 665

CDS view
buffering, 663
built-in functions, 667

CDS views, 659
building, 660

CE functions, 674

CHANGING parameters, 127

CHECK, 309, 313, 321

CHECK_DELTA, 309, 311, 321

CL_SALV_TABLE, 87, 409, 411

Class under test, 161

CLEANUP, 269

CMOD framework, 226

Code Inspector, 175, 200
new features, 192

Code pushdown, 682
AMDP, 688
CDS views, 687
locating code, 684
OpenSQL, 687
techniques, 684

Complex objects, 133

COMPONENTCONTROLLER, 545

Concision, 707

COND, 108

Conditional logic, 102

Constructor operators, 100

CORRESPONDING, 115, 707

CORS (Cross Origin Resource Sharing), 584

Cross-program communication, 134

CRUD, 296, 336, 582

CX_DYNAMIC_CHECK, 260

CX_STATIC_CHECK, 258

D

Data definitions, 150

Data type declarations, 92

Database access, 82, 633

DDIC table, 572

DDL, 661, 666, 669
coding, 662

Debugging, 203
recommended reading, 223
Script tab, 204
Script Wizard, 209, 213
skipping authority checks, 222
tracing every line, 222

Decision logic, 357

Delegated objects, 344

Dependencies, 138
breaking up, 141
eliminating, 139
identifying, 140

Design by contract, 255, 274
class invariants, 278
postconditions, 276
preconditions, 276

DISPLAY method, 412

Downcast, 126

DYNPRO, 507

E

Early Watch reports, 175

Eclipse, 35–36
AMDP, 673
and SAP HANA, 661
and SAPUI5, 592
bookmarking, 48
class constructors, 60

Eclipse (Cont.)
connect to backend system, 41
create attributes, 59
create parameters, 59
debugging, 67
extract method, 53
Extraction Wizard, 57
features, 42, 61
installation, 37
Modeling Framework, 40
multiple objects, 47
plug-ins, 71
prerequisites, 37
Quick Assist, 53
recommended reading, 79
refactoring, 59
release cycle, 35, 47
runtime analysis, 69
SAP add-ons, 39
SDK, 71
unit tests, 63
unused variables, 58

Ellison, Larry, 655

END, 206, 215

Enhancement framework, 225
recommended reading, 252

Enhancement spot, 237

Enhancements, 227
creation, 229
explicit, 227
implicit, 228
object-oriented programming, 233
procedural programming, 229
types, 227

Error handling, 270
RESUME, 272
RETRY, 271

Excel, 457
and ABAP, 462
and XML, 464
_EXCEL_WRITER, 464

Exception classes, 255, 257
choosing which type, 262
design, 263
types, 257

Exception handling, 257

Exceptions, 255, 257
declaring, 266
raising, 257, 267
recommended reading, 281

EXECUTE, 310, 315, 328

EXPORTING parameters, 127

Extended Program Check, 175

F

FACTORY method, 411

FILTER, 124

FitNesse, 172

Floorplan Manager (→ FPM)

FLUSH, 551

FOR, 94

FOR ALL ENTRIES, 193, 196

FPM, 507, 546
and BOPF, 556
floorplans, 547
FPM Workbench, 548
GUIBBs, 549
Guided Activity Floorplan, 547
modify WDA components, 546
Object Instance Floorplan, 547
Overview Floorplan, 547
Quick Activity Floorplan, 548
recommended reading, 559
UIBBs, 549

Functional methods, 103

G

GET_ENTITY_SET, 583, 588–589

GitHub, 167

GUID, 286, 302

GuiXT, 626

H

HASHED table, 112

Helper methods, 152

I

ICF, 497
 IF/ELSE, 108
 IF/THEN, 102
 INIT, 206, 215
 INITIALIZE, 423, 444
 Injection, 145

- automation*, 162
- constructor*, 147

 Inner joins, 88
 Internal tables, 109, 302

- grouping*, 122
- new functions*, 120

J

JavaScript, 564

L

LET, 95

M

Mass checks, 178

- reviewing*, 185
- running*, 182
- setup*, 180

 Memory, 633
 Method chaining, 98, 169
 Microsoft Excel (→ Excel)
 Microsoft Open XML, 461
 MIME repository, 490
 Mock objects, 138, 143, 161
 Model classes, 283
 Model-view-controller (→ MVC pattern)
 Modification Assistant, 225
 MOVE-CORRESPONDING, 116, 708
 MVC pattern, 284, 290, 409–410, 508, 557, 564

- controller*, 514
- location of model*, 509
- model*, 508
- view*, 511

N

NEEDS_CONFIRMATION, 551
 NEW, 92

O

Obeo, 73, 78
 object_configuration, 293
 Object-oriented programming, 50, 126, 144, 233, 283, 354, 409, 533, 711
 OData, 565
 Open source, 458, 616
 OpenSQL, 82, 658, 666

- new commands*, 82

 OpenUI5, 615

- open source*, 616

 ORDER BY, 199

P

Persistency layer, 634
 Plattner, Hasso, 631
 PREPARE, 327
 Private methods, 148
 Procedural programming, 138
 PROCESS_BEFORE_OUTPUT, 551
 PROLOGUE, 206

R

READ TABLE, 114
 Readability, 707
 REDUCE, 122
 Report programming, 407
 REST, 565
 RETRIEVE DEFAULT PARAM, 327
 Return values, 132
 Root class, 635
 RS_AUCV_RUNNER, 178
 Rules engines, 358

S

- SALV, 408
 - add custom icons*, 441
- SALV (Cont.)
 - application-specific changes*, 424
 - CL_SALV_GUI_TABLE_IDA*, 454
 - create container*, 416, 442
 - design report interface*, 414
 - display the report*, 435
 - editing data*, 447
 - event handling*, 421
 - initialize report*, 416
 - recommended reading*, 456
 - SAP HANA, 456
- SAP Code Exchange, 167
- SAP Decision Service Management, 359
- SAP Fiori, 626
- SAP Gateway, 561, 564
 - coding*, 581
 - create model*, 567
 - create service*, 574
 - data provider class*, 582
 - error handling*, 591
 - model provider class*, 582
 - service builder*, 568
 - service implementation*, 582
 - testing*, 579
- SAP GUI, 407
- SAP HANA, 176, 456, 588, 631, 655
 - ABAP table design*, 691
 - AMDP*, 659, 671
 - and Eclipse*, 661
 - artifacts*, 658
 - bottom-up development*, 677
 - CDS views*, 659
 - CE functions*, 674
 - code pushdown*, 657, 681
 - database proxies*, 679
 - database views*, 657
 - database-specific features*, 695
 - DDL*, 661
 - external views*, 678
 - programming changes*, 691
 - recommended reading*, 701
 - redundant storage*, 691
 - secondary indexes*, 693
- SAP HANA (Cont.)
 - SELECT coding*, 697
 - top-down development*, 658
 - transporting changes*, 680
- SAP NetWeaver Development Tools for ABAP (ADT) (→ Eclipse)
- SAP Screen Personas, 626
- SAPUI5, 508, 561
 - and Eclipse*, 566, 592
 - architecture*, 563
 - browser support*, 580
 - buttons*, 611
 - controller*, 608
 - fragment XML file*, 603
 - functions*, 609–610
 - HTML file*, 596
 - importing applications*, 620
 - JavaScript*, 564
 - prerequisites*, 565
 - recommended reading*, 627
 - storing applications*, 622
 - testing*, 613, 624
 - view*, 596
 - view and controller*, 592
 - XML file*, 597
- SCRIPT, 206, 209, 215
- Search helps, 129
 - predictive*, 129
- SELECT, 199
- SELECT *, 195
- Separation of concerns, 142
- service_manager*, 293
- SETUP, 151
- Shared memory, 631
 - broker class*, 640
 - business transaction event*, 651
 - data inconsistency*, 649
 - objects*, 634
 - read request*, 643
 - recommended reading*, 654
 - remote-enabled function module*, 650
 - root class*, 635
 - short dumps*, 652
 - updating database*, 647
 - write request*, 645
- Short dumps, 99–100, 632, 652
- SORTED table, 112

SQL, 588
 calculations, 84
 queries, 83
 SQLScript, 659, 671, 674
 String processing, 96
 Stub objects, 144
 SWITCH, 106
 Switch framework, 228

T

Table work areas, 112
 Test classes, 148, 153
 Test doubles, 131
 Test methods, 152
 Test-driven development, 137
 Transaction
 /*BOBF/CONF_UI*, 345
 /*WVND/MAINT_SERVICE*, 576
 <*ICON*>, 607
 ATC, 183
 BOB, 285, 295, 325
 /*BOBF/TEST_UI*, 349
 SAT, 70
 SATC, 682
 SCDO, 342
 SCI, 175
 SE11, 130, 660
 SE19, 227
 SE24, 77, 242, 264, 359, 636
 SE37, 77, 359
 SE80, 35, 44, 65, 70, 130, 359
 SEGW, 576
 SICF, 497, 580
 SLIN, 175
 SM30, 647
 SQLM, 682
 ST05, 682
 SWLT, 682
 XLST_TOOL, 73
 transaction_manager, 293
 Transport request, 56
 TRUE/FALSE, 105
 TRY/CATCH/CLEANUP, 267
 TYPE POOL, 90
 TYPE REF TO DATA, 101

U

UMAP, 73
 UML diagram, 72
 Unit testing, 147
 ABAP 7.4, 131
 automation, 161
 executable specifications, 148
 mockA, 162, 167, 170, 174
 recommended reading, 174
 Usage Procedure Logging, 191
 User exits
 form-based, 226

V

VALUE, 93
 Variables, 90
 VBA (Visual Basic for Applications), 488
 VOFM routines, 226

W

Watchpoints, 207, 223
 WDA, 507
 ALV grid, 507, 529
 calling application, 535
 coding, 535
 component controller, 514
 create component, 517
 data structures, 518
 defining view, 527
 graphical screen painter, 512
 interface controller, 518
 nodes, 521
 PAI, 513
 PBO, 513
 recommended reading, 559
 standard elements, 523
 storing data, 512
 view settings, 522
 windows vs. views, 511
 Web Dynpro ABAP (→ WDA)

X

XML, 460, 463
XSDBOOL, 105

Z

Z classes, 163, 215, 293, 564
Z indexes, 110
Z tables, 390, 647
ZCL_BC_VIEW_SALV_TABLE, 443
ZCX_NO_CHECK, 260

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it, for example, by writing a review on <http://www.sap-press.com>. If you think there is room for improvement, please get in touch with the editor of the book: kellyw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also navigate to our web catalog page for this book to submit feedback or share your reading experience via Twitter, Facebook, email, or by writing a book review. Simply follow this link: <http://www.sap-press.com/3680>.

Supplements

Supplements (sample code, exercise materials, lists, and so on) are provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: <http://www.sap-press.com/3680>. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at SAP PRESS, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website <http://www.sap-press.com> provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at <http://www.sap-press.com>.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2015 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All of the screenshots and graphics reproduced in this book are subject to copyright © SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany. SAP, the SAP logo, mySAP, mySAP.com, SAP Business Suite, SAP NetWeaver, SAP R/3, SAP R/2, SAP B2B, SAPtronic, SAPscript, SAP BW, SAP CRM, SAP EarlyWatch, SAP ArchiveLink, SAP HANA, SAP GUI, SAP Business Workflow, SAP Business Engineer, SAP Business Navigator, SAP Business Framework, SAP Business Information Warehouse, SAP interenterprise solutions, SAP APO, AcceleratedSAP, InterSAP, SAPoffice, SAPfind, SAPfile, SAPtime, SAPmail, SAP-access, SAP-EDI, R/3 Retail, Accelerated HR, Accelerated HiTech, Accelerated Consumer Products, ABAP, ABAP/4, ALE/WEB, Alloy, BAPI, Business Framework, BW Explorer, Duet, Enjoy-SAP, mySAP.com e-business platform, mySAP Enterprise Portals, RIVA, SAPPHIRE, TeamSAP, Webflow, and SAP PRESS are registered or unregistered trademarks of SAP SE, Walldorf, Germany.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.