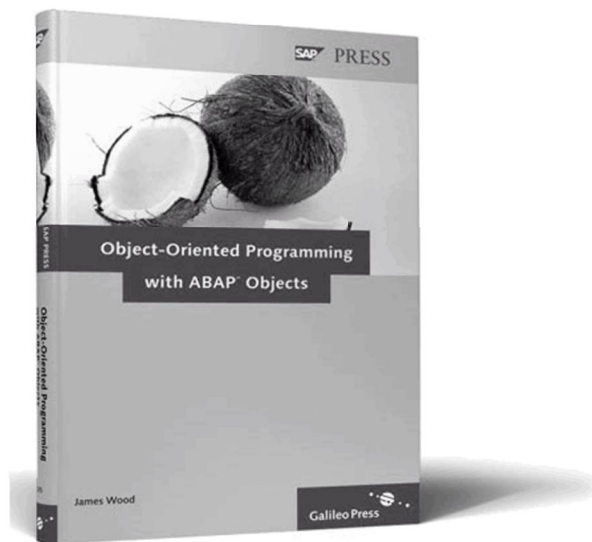


James Wood

Object-Oriented Programming with ABAP™ Objects



 Galileo Press®

Bonn • Boston

Introduction

With all of the hype surrounding object-oriented programming, it can be difficult to separate the truth from fantasy. If you have picked up this book, it is likely that you have developed an interest in learning more about what the excitement is all about. This book provides the answers that you are looking for.

The goal of this book is to teach you how to think about writing ABAP™ software from an object-oriented point-of-view. After reading this book, you will be equipped to work with many of the new and exciting ABAP-based technologies based on ABAP Objects such as Web Dynpro, ABAP Object Services, SAP® Business Workflow, and Web Services.

Target Group and Prerequisites

This book is intended for ABAP application developers that have some basic experience writing ABAP programs using the ABAP Development Workbench. Basic ABAP language concepts are not covered in this book, so you have not worked with ABAP before, read *ABAP Objects – ABAP Programming in SAP NetWeaver* (SAP PRESS, 2007). Of course, in an introductory book such as this, no prior object-oriented experience is expected.

The object-oriented extensions to the ABAP programming language (i.e., the *Objects* part of ABAP Objects) were made available in SAP R/3 4.6C. Therefore, you do not have to have the latest version of the SAP NetWeaver Application Server ABAP (AS ABAP) to start working with most of the object-oriented concepts described in this book. However, additions to the standard that were added in subsequent releases are pointed out where appropriate.

If you want to reproduce the examples in the book and don't have access to an AS ABAP instance, you can download a trial version from the SAP Developer Network (<http://sdn.sap.com>) that you can install on your local PC. From the main page, select DOWNLOADS • SOFTWARE DOWNLOADS • SAP NETWEAVER MAIN RELEASES. There, you will find several versions of the AS ABAP that you can install depending on your preferred operating system, and so on. Each download pack-

age comes with a set of instructions to help you get started. The SAP Developer Network forums can also provide useful tips if you run into problems.

Structure of the Book

In many ways, this was a very difficult book to write. Doing a topic like this justice requires a healthy balance between theoretical and practical concepts, so many practical examples are included that illustrate theoretical concepts.

The first part of the book helps get you started quickly by describing basic object-oriented concepts using a series of simple object-oriented programs. The second part of the book covers core object-oriented concepts such as encapsulation, inheritance, and polymorphism. The final part of the book teaches you how to apply these concepts using the tools and services available in the SAP NetWeaver Application Server.

The end of each chapter includes a brief tutorial on the Unified Modeling Language (UML). These tutorials show you how to express your object-oriented designs using a graphical notation that is commonly used throughout the industry.

In detail, the chapters provide the following content:

- ▶ **Chapter 1: Introduction to Object-Oriented Programming**
Object-oriented programming is steeped in theory. Therefore, before we delve into the creation of classes, we need to review this theory so that you can understand how everything fits together. The concepts described in this chapter provide you with the foundation you need to start developing classes.
- ▶ **Chapter 2: Working with Objects**
This chapter reinforces the theoretical concepts covered in Chapter 1 by allowing you to get your hands dirty by creating some simple object-oriented programs written in ABAP Objects. Here, we will spend a lot of time looking at the ABAP Objects syntax for defining classes, methods, and so on. This syntax is highlighted by a series of examples that illustrate how classes can be used in practical situations.
- ▶ **Chapter 3: Encapsulation and Implementation Hiding**
This chapter introduces you to two important concepts in object-oriented design: encapsulation and implementation hiding. First, the importance of these concepts is demonstrated by observing some problems with code librar-

ies developed using procedural methods. Then, you will learn how to avoid these problems in class libraries through the use of access specifiers. Finally, we will take a step back and look at ways to develop reusable classes using a technique called design-by-contract.

▶ **Chapter 4: Object Initialization and Cleanup**

This chapter walks you through the lifecycle of objects from creation to deletion. Along the way, you will learn how to interact with this process to maximize performance and improve the integrity of your designs.

▶ **Chapter 5: Inheritance**

One of the potential side effects of good object-oriented designs is the ability to reuse code. In this chapter, you will learn how to reuse classes using the concept of inheritance. We will also consider an alternative form of class reuse known as composition.

▶ **Chapter 6: Polymorphism**

This chapter shows you how to exploit inheritance relationships described in Chapter 5 using a technique referred to as polymorphism. This discussion is highlighted by the introduction of interfaces, which are pure elements of design.

▶ **Chapter 7: Component-Based Design Concepts**

After covering the basics of object-oriented programming in Chapters 1–6, this chapter broadens the focus a bit by showing you how the ABAP Package Concept can be used to organize your class libraries into coarse-grained development components.

▶ **Chapter 8: Error Handling with Exceptions**

This chapter explains how to deal with exceptions in your classes and programs using the ABAP class-based exception handling concept.

▶ **Chapter 9: Unit Testing with ABAP Unit**

This chapter shows you how to develop automated unit tests using the ABAP Unit test framework. These tests help you ensure that your classes deliver on the functionality described in their API contracts.

▶ **Chapter 10: Working with the SAP List Viewer**

This chapter is the first of three case study chapters that show you how ABAP Objects classes can be used in many common development tasks. In this chapter, you see how to create interactive reports using the new object-oriented

ALV Object Model. This chapter also provides a practical example for working with events in ABAP Objects.

► **Chapter 11: ABAP Object Services**

This chapter demonstrates the use of the services provided by the ABAP Object Services framework. In particular, you will learn how to use these services to develop persistent classes whose state can be stored in the database without having to write a single line of SQL.

► **Chapter 12: Working with XML**

This chapter concludes the case study series by showing you how to work with XML documents using the object-oriented iXML library provided with the SAP NetWeaver Application Server. This discussion also provides you with an opportunity to develop an abstract data type that uses most of the concepts described throughout the course of the book.

► **Chapter 13: Where to Go From Here**

In this final chapter, we will look ahead to see how to apply the object-oriented concepts learned in this book in real-world projects.

► **Appendix: Debugging Objects**

In this appendix, we will look at how to use the ABAP Debugger tool to debug object-oriented programs.

Conventions

This book contains many examples demonstrating syntax, functionality, and so on. Therefore, to distinguish these sections, we use a font similar to the one used in many integrated development environments to improve code readability:

```
CLASS lcl_test DEFINITION.  
    PUBLIC SECTION.  
        ...  
ENDCLASS.
```

As new syntax concepts are introduced, these statements are highlighted using a bold listing font (i.e., the **PUBLIC SECTION** statement in the preceding code snippet).

Acknowledgments

I am a firm believer in the saying "you are what you read." As such, I am indebted to so many great authors whose works have planted the seeds from which this book took form. I have similarly been fortunate enough to have the opportunity to work with so many talented software development professionals who have taught me so much. It is my sincere hope that this work represents a small token of my appreciation for all of their hard work and dedication to the field.

I would like to thank Dr. Stephen Yuan and Dr. Jason Denton for opening my eyes to the world of software engineering. I would also like to thank Russell Sloan at IBM and Colin Norton at SAP America for giving me a chance to spread my wings.

Much of the inspiration from this book came during my time working on a project at Raytheon. A special thanks to the good people there who allowed me to play "mad scientist" with their development methodology. I am also grateful for the fresh perspective offered by members of the OneAero development team at Lockheed Martin. In particular, I would like to thank Greg Hawkins from SAP America, who graciously offered valuable insight whenever I needed someone to look at this from a different angle.

To my editor, Stefan, thank you so much for your support throughout this process. I would not have been able to do this without you.

To my Dad, and my Mom who still corrects my grammar to this day, thanks for always being there for me.

To my children, Andersen and Paige, thank you for all of your love and support. I will always cherish our writing breaks playing on the floor in my office.


To my wife Andrea, I just want to say how much I feel loved and supported by you. Without your influence on my life, I would not be where I am today. I consider myself very fortunate to have met someone as special as you.

And finally, to Him who is, and was, and is to come again: Soli Deo Gloria.

James Wood

Principal SAP NetWeaver Software Consultant,
Bowdark Consulting, Inc., Flower Mound, TX

PART I
Basics



This chapter provides an overview of object-oriented programming from a conceptual point of view. The concepts described in this chapter lay the foundation for the remainder of the book.

1 Introduction to Object-Oriented Programming

Object-oriented programming (OOP) is a programming methodology that is used to simplify software designs to make them easier to understand, maintain, and reuse. Like procedural programming before it, OOP represents a different way of thinking about writing software. The beauty of OOP lies in its simplicity. As you will see, the expressiveness of OOP makes it easier to deliver quality software components on time and under budget.

The purpose of this chapter is to introduce you to the basic concepts that you need to understand to effectively design and develop object-oriented programs. These concepts apply to most modern OOP languages such as C++, Java, and, of course, ABAP Objects. This chapter also begins an introduction to the Unified Modeling Language (UML), which is the de facto object modeling language used in the industry today.

1.1 The Need for a Better Abstraction

The most important goal for any software development project is to deliver a product that solves the problem(s) it was designed to address. To be effective in this endeavor, developers must work collaboratively with business analysts to formulate a good design approach. Many projects fail at this stage because it can be very difficult for functional and technical team members to communicate in terms that are well understood by their counterparts. A way to simplify complex designs into a form that can be easily interpreted by all project stakeholders — a common language — is needed. Over the years, numerous attempts have been made to develop programming languages and development methodologies to

bridge this communication gap. Most of these approaches failed because the languages and methods were either too hard to learn or not flexible enough to be used to articulate all of the various viewpoints within the team.

In his book *Thinking in Java* (Prentice Hall, 2006), Bruce Eckel argues that “the complexity of the programs you are trying to solve is directly related to the kind and quality of abstraction you are trying to work with.” Early programming languages (e.g., assembly languages) provided a thin layer of abstraction on top of the underlying machine. Consequently, developers working with those languages spent almost as much time worrying about “bit twiddling” as they did thinking about the problem they were trying to solve.

The next-generation procedural programming languages were much more expressive but still required a considerable amount of translation between the conceptual problem domain and the physical solution space (i.e., the program code). This conversion process is not only time consuming but also prone to error because requirements can easily become lost in translation.

With an object-oriented approach, solutions are designed from the outset in terms of real-world “objects” modeled from the problem domain. Therefore, it becomes much easier for business analysts and programmers to exchange information and ideas about a design that uses a common domain language as opposed to one based on technical constructs such as integers, structured data types, procedures, and so on. The improvements in communication help to bring out hidden requirements, identify risks, and reduce confusion. At the end of the day, all of this helps to improve the quality of the software being developed.

1.2 Classes and Objects

Students learning pure object-oriented languages such as Java are often taught that “everything is an object.” Although this is not necessarily the case in a hybrid language such as ABAP Objects (where it is still possible to use procedural constructs), it is still a good way to start thinking about how to design programs using an object-oriented approach. Of course, it helps if you know what an object is.

An *object* is a special kind of variable that has distinct characteristics and behaviors. The characteristics (or attributes) of an object are used to describe the *state* of an object. For example a car object might have attributes to capture information such as color, make, or current driving speed. Behaviors (or methods) represent

the *actions* performed by an object. In our car example, there might be methods that can be used to drive, turn, and stop the car.

As you might expect, a considerable amount of the object-oriented design process is focused on identifying the objects you will need to model a given problem domain. This process is an inexact science that often requires some trial and error before you get it right. One fairly typical approach for initiating this process is to identify the nouns (i.e., a person, place, thing, or idea) used to describe various aspects of the problem domain. The semantic meaning of these nouns provides a basis for classifying and defining the objects you need to build a working program model.

Early object-oriented researchers drew a parallel between this process and classification techniques used by biologists to identify and explain the relationships between plants and animals. The term *class* was used to describe abstract data types that could be created to simulate real-world phenomena. Consequently, most object-oriented languages use the `CLASS` keyword to define these abstract data types. A class declaration defines a blueprint that describes how to create objects. Figure 1.1 shows an example of a class definition for the Car object described previously.

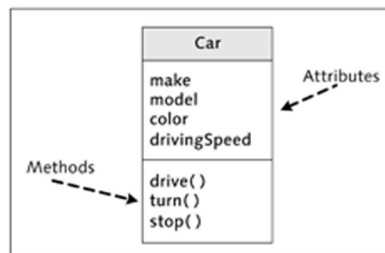


Figure 1.1 Class Definition for the Car Class Example

One good analogy to explain the difference between a class and an object is to think about the relationship between a set of architectural blueprints and the houses that are built in reference to those blueprints. In this case, the blueprints provide instructions that can be used as a guide for constructing new houses (e.g., the layout of the floor plan, room dimensions, what materials to use, etc.). Each of the homes that are built will have its own unique street address along with specific customizations such as paint color (see Figure 1.2). This uniqueness gives a

home an identity unto itself. In other words, a home is referred to as an *instance* of a particular set of blueprints. Of course, although each house is distinctive in its own way, it also shares certain commonalities with all of the other houses that have been built using the same set of blueprints. These commonalities are often exploited by homebuilders looking to reuse materials and assembly expertise in an effort to reduce costs. You will see how these same concepts can apply to object-oriented software development in later chapters.

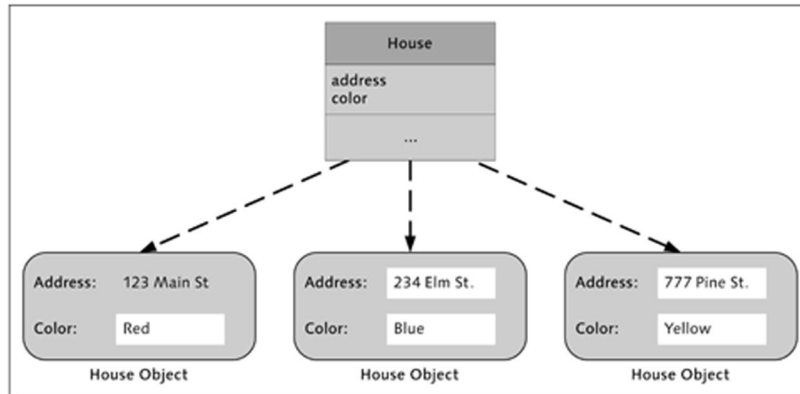


Figure 1.2 Relationship Between Classes and Objects

Now that you have a basic understanding of the relationship between classes and objects, let's briefly summarize these concepts in object-oriented parlance. A class is a blueprint that can be used to construct *object instances*. Each object instance is controlled by an *object reference variable* that is designed to point to objects of a particular class type. A class's type defines an *interface* that describes how to communicate with object instances of that class. Objects communicate with one another by sending messages to each other. In object-oriented terms, an object sends a message to another object by calling a *method* on that object. In this sense, you can think of methods as the *services* provided by a class/object. Because each object maintains its own internal state information, it is self-aware and therefore able to process these service requests in context.

Put another way, objects *know* how to do their job. This innate knowledge allows you to delegate tasks to objects and trust that they will be carried out correctly. For this reason, OOP pioneer Alan Kay described object-oriented programs as "a

bunch of objects telling each other what to do by sending messages [to each other]."¹

1.3 Establishing Boundaries

Every healthy relationship needs boundaries, and the relationship between objects working together inside of a computer system is no different. As an object-oriented design begins to take shape, each class assumes specific role assignments within the system. This division of labor helps to simplify the overall program model, allowing each class to specialize in solving a particular piece of the problem at hand. Such classes are said to have *high cohesion* in the sense that each of the class's operations are closely related in some intuitive way. Defining boundaries within classes helps to maintain the integrity and cohesiveness of the program model, making sure that classes are used correctly.

Object-oriented languages allow you to establish boundaries within a class using the concept of *visibility sections*. Visibility sections clearly separate a class's *interface* from its *implementation* — a process commonly referred to as *encapsulation* or *implementation hiding* (see Figure 1.3).

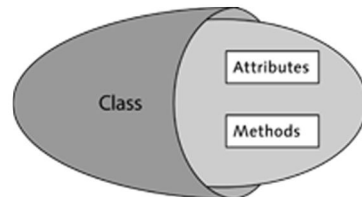


Figure 1.3 Encapsulation of Data and Behavior in a Class

Class components can be defined in *public*, *private*, or *protected* visibility sections that control how these components are accessed. The private visibility section is used to deny access to a class's components from outside of the class. Such components can only be accessed from inside the class (e.g., in a method) — they are completely hidden from the outside world. Components defined in the public visibility section can be accessed from any context. We will discuss the protected visibility section in Chapter 5, Inheritance.

¹ Quote taken from *Thinking in Java* (Prentice Hall, 2006).

There are a couple of important reasons for hiding the implementation details of a class using visibility sections:

- ▶ First of all, hiding implementation details makes life easier for clients wanting to leverage existing classes in their programs. Here, clients only need to familiarize themselves with the components defined in the public interface of a class — everything else is just details. This significantly shortens the learning curve for client developers wanting to understand how to work with a class by allowing them to concentrate on the *what* and not the *how*.
- ▶ Secondly, implementation hiding significantly reduces the side effects associated with making changes to a class. After all, if the internal details of a class are hidden to the outside world, then you can change the implementation of a class without having to worry about affecting any client code currently using that class.

Client developers accustomed to having total access to everything within their programs often find this concept to be highly restrictive (or indeed punitive) at first glance. However, the important thing to realize here is that the intent is not to make the user's life more difficult but rather to hide the aspects of the class that are most likely to change. You can think of a class's public interface as a service contract between instances of the class (objects) and its clients. A developer of a class is free to change the underlying implementation of that class in any way (e.g., to make it run faster, use a different data source, etc.) as long as he does not violate this contractual agreement. This design approach is commonly referred to as design by contract².

Encapsulated classes do not have a lot of dependencies to the outside world. Moreover, the interactions that they do have with external clients are controlled through a stabilized public interface. In other words, an encapsulated class and its clients are *loosely coupled*. For the most part, classes with well-defined interfaces can be plugged into another context without a lot of "re-wiring." Therefore, when designed correctly, encapsulated classes become reusable software assets that should be able to be leveraged in many contexts. We will investigate some best practices for designing encapsulated classes in Chapter 3, Encapsulation and Implementation Hiding.

² This term was originally coined by Bertrand Meyer in a technical report entitled *Design by Contract* (Interactive Software Engineering, 1986). We will consider this design approach in more detail in Chapter 3, Encapsulation and Implementation Hiding.

Over time, the accumulation of these software assets makes it possible to quickly compose solutions using quality components that have been proven to work through exhaustive testing. This composition-based approach to solution design is similar to various component-based approaches used in other engineering disciplines. For example, automotive manufacturers simplify the manufacturing process by splitting up the design of a car into a series of discrete parts. These parts compartmentalize various complexities into smaller units that are easier to understand and maintain.

For instance, the intricacies related to properly mixing fuel and air in an internal combustion engine can be encapsulated into a *fuel injector* part. The fuel injector part can then be effortlessly integrated into the engine assembly by a mechanic without detailed knowledge of complex injection schemes, and so on. Such parts, when designed with common interfaces, also become interchangeable. This touches several levels of economics, allowing manufacturers to reuse parts across product lines and also to replace faulty parts with better ones without requiring an engine overhaul. We will look at ways to perform component-based software development in Chapter 7, Component-Based Design Concepts.

1.4 Reuse

One of the most compelling reasons for adopting an object-oriented approach to program design is the significant capability for reusing code. Although it is easy to allow yourself to become dazzled by promises of huge productivity gains, it is important to keep things in perspective. Learning how to develop reusable classes takes time and experience. The following subsections describe some basic techniques for reusing classes. We will cover each of these topics at length in Chapter 5, Inheritance, and Chapter 6, Polymorphism.

1.4.1 Composition

The easiest way to reuse a class is to simply create an object instance and begin calling its methods. Classes can also be reused as attributes of new classes that you are building. This usage type is often referred to as *composition*, where new classes are composed from existing classes whose types are used to define member attributes, and so on. These classes are aggregates, using existing classes as building blocks (think LEGO®) for constructing arbitrarily complex assemblies.

Designs based on composition are easy to understand and highly flexible. Because member objects can be hidden just like any other attribute, it is easy to change the way you use these objects both at design time and at runtime.

1.4.2 Inheritance

Another way to reuse a class is through *inheritance*. The concept of inheritance is a continuation of the classification metaphor used to describe the nature of classes and their relationships. Here, we are interested in defining specialization relationships between families of related classes. These relationships begin to reveal themselves as an object-oriented design matures.

The idea of inheritance is best explained by an example. Let's imagine that you are working on an object-oriented design for a banking system. Initially, you come up with a series of classes, including one to represent a bank account. After studying the requirements further, you discover that there are certain peculiarities unique to checking and savings accounts. At this point, you are faced with a dilemma. On one hand, you could copy the code you have put together for the account into new checking and savings account classes. However, this seems wasteful because this would introduce a lot of redundant code. Another option would be to use inheritance to describe this specialization relationship. In this case, you still create new checking and savings account classes, but you create them as *subclasses* derived from the original account class (which is the parent or superclass). The checking and savings account subclasses are said to *inherit* the attributes and behaviors (and indeed the *type*) of the account superclass (see Figure 1.4). Now, the relevant changes can be made to each of the subclasses independently without having to reinvent the wheel.

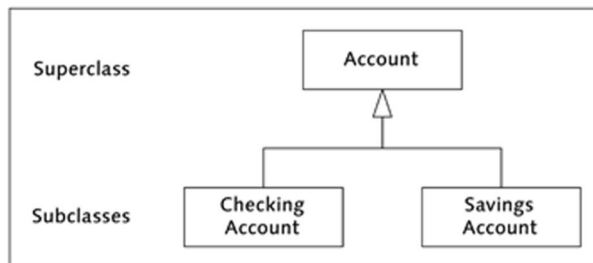


Figure 1.4 Inheritance Tree for Bank Accounts

It is important to remember that inheritance describes a relationship; it is not simply a fancy term for copying and pasting one piece of code into another. Initially, a subclass looks like a clone of the superclass. However, over time, a subclass can be extended to add new attributes and methods as needed. Additionally, changes to the superclass are automatically applied to the subclass (you will see exceptions to this rule in Chapter 5, Inheritance). It is possible to create class hierarchies with arbitrarily deep inheritance relationships.

The connection between a subclass and its parent is often described using the “is a” relationship. Looking at the preceding example, a checking account *is an* account, and so on. The is-a relationship is a simple way of saying that the subclass and superclass share the same type. As you will recall, a class’s type describes how you can communicate with objects of that class. Therefore, because objects of a superclass and subclass share the same type, it is possible to communicate with both of them in the exact same way. Polymorphism exploits this capability, allowing for code reuse in multiple dimensions.

1.4.3 Polymorphism

The definition of an inheritance relationship implies that a subclass is inheriting both the *type* and the *implementation* of its superclass. In the subclass, however, it is possible to *redefine* a method’s implementation to further specialize certain behavior. Redefining a method does not change the interface of the method (i.e., the way it is called); it simply changes the behavior inside the method in some way.

Polymorphism allows you to work with subclasses in the exact same way that you deal with superclasses. To show how this works, let’s consider an example. Figure 1.5 depicts an `Employee` class hierarchy that might be used to model the types of employees managed within a certain company. In this case, the `Employee` superclass is used to describe the basic characteristics and behaviors for all types of employees. The three specialized subclasses (`HourlyEmployee`, `CommissionEmployee`, and `SalariedEmployee`) are used to represent employees paid by the hour, employees working on commission, and salaried employees, respectively. Also, for the purposes of this example, let’s assume that the `calculateWage` method has been redefined in each of the subclasses to properly calculate the employee’s wage based on the actual employee type.

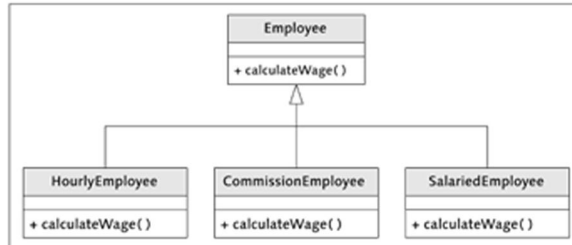


Figure 1.5 Employee Class Hierarchy

Now, let's imagine that the company wants to use this `Employee` class hierarchy to enhance its accounts payable (AP) system by automating the creation of monthly paychecks. Listing 1.1 shows an example of the pseudo code for an enhancement such as this.

```

For Each Employee
    Call "calculateWage" to Calculate the Employee's Wage
    Print the Paycheck
End For
  
```

Listing 1.1 An Example of an Algorithm Using Polymorphism

From an AP perspective, the logic really is that simple. The fact that monthly wages are calculated differently for each employee type is mainly a problem for the human resources (HR) department. Managing these concerns in two places introduces a maintenance nightmare. Fortunately, the concept of *polymorphism* provides a way to design the AP system to work with generic `Employee` types and not get bogged down with a lot of HR-specific details.

The term polymorphism literally means *many forms*. In the preceding example, each subclass represents a different form (or type) of `Employee`. However, because the subclasses take part in an inheritance relationship with the `Employee` superclass, each subclass *is* an `Employee`. In other words, because both the superclass and subclass share the same public interface, any method that can be called on the superclass can also be called on the subclass. The AP system can take advantage of this feature by simply working with generic `Employee` instances. At runtime, these instances could be of type `Employee` or any of its subclasses. The runtime system takes care of making sure that the proper method implementation is called. This is another example of how an object is "smart enough" to know how to do its job.

Polymorphism introduces a capability for creating reusable algorithms that are designed to work on generic objects. In Chapter 6, Polymorphism, we will look at how to take advantage of this functionality in your designs.

1.5 Object Management

An object-oriented program typically consists of a series of objects that call on one another to perform various tasks. Because each object is depended on to fulfill a specific role within the system, it is important that the object has everything it needs to do its job when called upon.

To ensure that an object is properly initialized, most object-oriented languages allow you to create a special method called a *constructor* that is called whenever a new object is created. The constructor's job is to make sure that the object is initialized in a consistent state before it is used.

The lifecycle of an object is typically quite different from traditional program variables. Often, it is impossible to determine exactly how many instances of an object you will need in a program until runtime. This presents a challenge to object-oriented language designers needing to develop a mechanism for managing program resources. A common solution to this problem for many modern object-oriented language implementations is to dynamically allocate objects from a *memory heap*. This approach is beneficial for the programmer because it takes the problem of memory management and places it squarely on the shoulders of the runtime system. We will investigate the details of an object's lifecycle in Chapter 4, Object Initialization and Cleanup.

1.6 UML Tutorial: Class Diagram Basics

Object-oriented software development places a considerable amount of emphasis on design. Before you can start coding, it is imperative that you have a plan. For instance, you must figure out what kind of objects you will need as well as how those objects will interact with one another at runtime.

Object-Oriented Analysis and Design (OOAD) is a software development methodology used to analyze system requirements and formulate a system design from an object-oriented perspective. OOAD practitioners often use graphical modeling

techniques to communicate their designs more effectively. The *Unified Modeling Language* (UML) contains a set of graphical notations for building diagrams that depict various aspects of the system model. The UML is used extensively throughout the software development industry, so it is important that you understand how to use UML diagrams to express and interpret object-oriented designs.

Throughout the remainder of this book, we will examine the usage types of various UML diagrams at the end of each chapter. Our discussions will be based on version 2.0 of the UML standard³. In this chapter, let's begin by looking at the class diagram. For now, we will simply reinforce the concepts covered in this introductory chapter. In Chapter 5, Inheritance, and Chapter 6, Polymorphism, more advanced features of class diagrams will be considered.

A class diagram is used to illustrate the static architecture of an object-oriented system. Here, you can depict the various classes used in the system, as well as their relationships. Figure 1.6 shows a simple class diagram that describes a scaled-down model of a sales order system used to process orders for an e-commerce website. In the following subsections, we will consider some of the basic features of a UML class diagram.

1.6.1 Classes

The diagram in Figure 1.6 contains five classes: `Order`, `OrderItem`, `Product`, `Customer`, and `Address`. Classes are represented in class diagrams as rectangular boxes partitioned into three sections: the top section contains the class name, the middle section contains the attributes associated with the class, and the bottom section contains the operations (or methods) of the class (see Figure 1.7).

Of course, the rules here are not very strict. For example, in the diagram shown in Figure 1.6, no operations are defined for class `Address`. This could be because there were no operations identified for this class when it was designed, or the creator of this diagram thought that the operations were insignificant when describing the system architecture. The point is to not get too carried away with the details because this can complicate the model to the point that the diagram is not readable.

³ The UML standard is maintained by the Object Management Group (OMG). For more information on the OMG, check out <http://www.uml.org>.

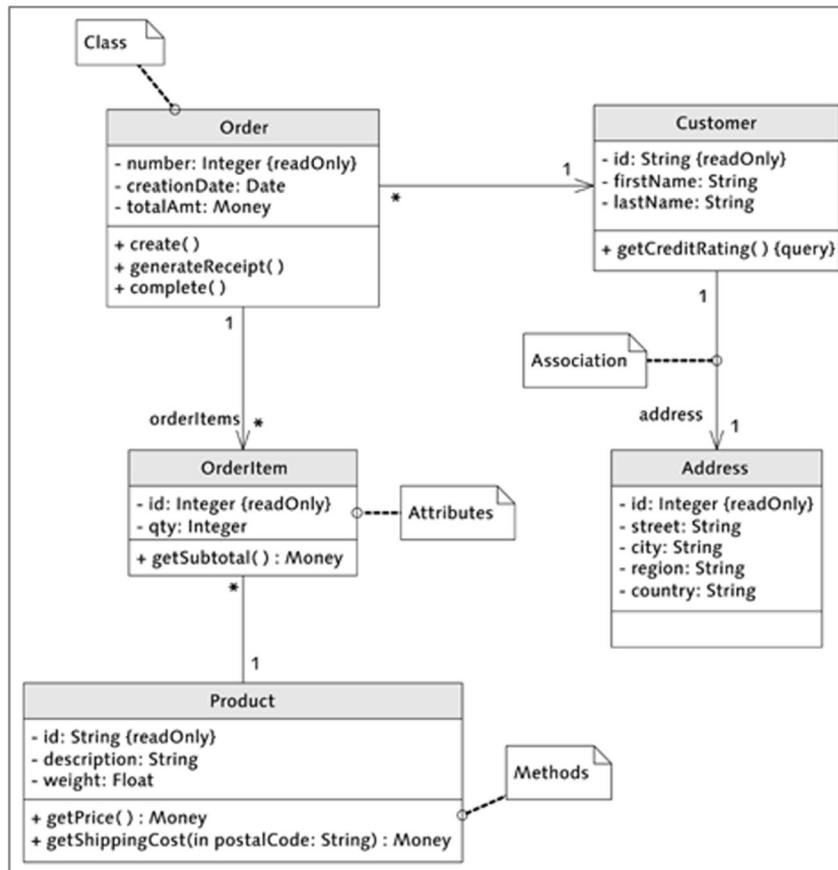


Figure 1.6 Basic UML Class Diagram

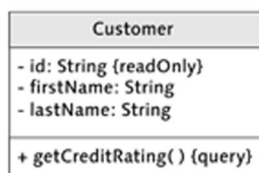


Figure 1.7 UML Class Notation

Some developers new to UML fall into this trap, worrying that there isn't enough information in their class diagram to start writing code. If you find yourself in this position, remember that the UML provides a multitude of diagrams that can be used to express the various aspects of your design; class diagrams only tell one part of the story.

1.6.2 Attributes

Attributes can be specified on the class diagram using the syntax shown in the following Listing 1.2.

```
visibility name: type-expression = initial-value  
                (property-string)
```

Listing 1.2 Attribute Notation for a UML Class Diagram

You are only required to provide the name when specifying an attribute in a class diagram. However, the other syntax elements shown in Listing 1.2 can be used to provide some additional information about the attribute:

- ▶ The *visibility* of an attribute describes the accessibility of the attribute from an external perspective. Possible values include + (plus) for public attributes, - (minus) for private attributes, and # (sharp) for protected attributes.
- ▶ The *type-expression* is used to describe the attribute's type. The UML defines some standard types such as integer or string, but you can also specify custom types here. The *type-expression* can also be used to express the cardinality of an attribute (e.g., for an internal table), and the initial value of the attribute (if one is assigned).
- ▶ The *property string* is an optional element that can be used to describe certain additional properties for an attribute. For example, in the `OrderItem` class from Figure 1.6, the `id` attribute has the `readOnly` property assigned to indicate that an item's ID number never changes. Values for these properties can be defined at the discretion of the person designing the class diagram. The primary purpose here is to provide additional details that are helpful to the developer responsible for actually implementing the class using an OOP language.

Listing 1.3 shows an example of the syntax described in Listing 1.2 using the `id` attribute specified in the `OrderItem` class from Figure 1.6. This syntax declares `id` as a private, read-only attribute of type `Integer`.

```
- id: Integer (readOnly)
```

Listing 1.3 An Example of an Attribute Definition

1.6.3 Operations

Operations can be expressed using the syntax shown in Listing 1.4.

```
visibility name(parameter-list) : return-type
                               (property-string)
```

Listing 1.4 Operation Notation in a UML Class Diagram

For brevity's sake, developers will often just specify the name of an operation when creating a class diagram. The remaining optional syntactical elements from Listing 1.4 are typically used strategically to emphasize a certain aspect of the operation:

- ▶ The *visibility* of an operation defines its accessibility. Possible values include `+` (plus) for public operations, `-` (minus) for private operations, and `#` (sharp) for protected operations.
- ▶ The *parameter-list* in parentheses can be used to specify a comma-separated list of parameters for the operation. Each parameter is of the form shown in Listing 1.5.
 - ▶ Here, *kind* signifies the type of parameter. Valid values include `in` for inbound parameters passed by value, `out` for outbound parameters passed by value, and `inout` for inbound parameters passed by reference.
 - ▶ The *name* token symbolizes the parameter name.
 - ▶ Each parameter can optionally have a type associated with it using the *type* token. The type can be a generic type or a type specific to a particular programming language.
 - ▶ Finally, you can specify an initial value for the parameter using the *default-value* expression.

```
kind name : type = default-value
```

Listing 1.5 Specifying the Parameters of an Operation

- ▶ The *return-type* element is used to specify the data type of values returned by functional operations.

- ▶ The optional `property-string` indicates certain properties assigned to an operation. An example of this is the `{query}` property string assigned to the `getCreditRating` operation of class `Customer`. Such operations are *read-only* operations that do not alter the state of the object. Applying these property strings can give hints to aid the developer in implementing the class in a particular programming language.

An example of the syntax described in Listing 1.4 is given in Listing 1.6. This example declares a public operation called `getShippingCost` that receives a single inbound parameter called `postalCode` (which is of type `String`). The operation returns a value of type `Money` to represent the derived shipping cost.

```
+ getShippingCost(in postalCode: String) : Money
```

Listing 1.6 An Example of an Operation Definition

1.6.4 Associations

The lines drawn between classes in a class diagram represent a type of *association*. You can think of an association as another way to specify an attribute for a class. For example, the directed line drawn between the `Customer` and `Address` classes in Figure 1.6 describes an attribute of type `Address` for class `Customer`. The arrow in the association between classes `Customer` and `Address` indicates that instances of class `Address` can be reached through an attribute defined in class `Customer`.

If the association line had contained arrows pointing in both directions, then the association would have been *bidirectional*. In this case, an attribute of type `Customer` would also have been defined for class `Address`, making it possible to navigate between attributes in both directions. The numbers affixed to each endpoint represent the cardinality of the association from the perspective of the nearby class (see Table 1.1). For example, in Figure 1.6, the association between classes `Order` and `OrderItem` denotes a *one-to-many* relationship between an order and its items. In this case, an order can contain zero or more items, and any given item can exist for exactly one order.

At this point, you might be wondering why you would need to build an association when you could just use a simple attribute instead. There is no hard-and-fast rule for using one approach instead of the other. However, a good rule of thumb to apply here is to use an association whenever you are using composition to reuse a class inside of another class. This illustrates the composition relationship

more clearly, and makes it easier to rework the diagram as you experiment with your class model.

Cardinality	Description
0..1	Zero or one instances of a class
1	Exactly one instance of a class
*	Zero or more instances of a class
m..n	A range of instances with lower/upper bounds (e.g., 2..4)

Table 1.1 UML Cardinality Notation

1.6.5 Notes

You can add comments to a UML diagram using notes. Notes are represented using an element that resembles a sticky note that has been dog-eared in the top-right corner (see Figure 1.8). These notes can be used in any kind of UML diagram to include comments related to a particular element (linked via a dashed line) or to the diagram as a whole. Notes are often used to help clarify a certain requirement that is too difficult to express using standard UML notation.

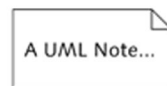


Figure 1.8 UML Note Notation

1.7 Summary

In this chapter, you learned that a class is a kind of blueprint that can be used to describe how to create object instances. Classes combine attributes and methods together to model real-world phenomena in a software setting. Rules and constraints for these models can be enforced inside visibility sections that control how attributes and methods within the class are used. We also considered some of the basic reuse capabilities associated with classes. Finally, we concluded this introductory chapter by initiating our discussion of the UML and, specifically, class diagrams.

1 | Introduction to Object-Oriented Programming

This chapter covered a lot of ground very quickly. If you are finding yourself a little lost, don't worry; you will learn much more about each of these topics in the coming chapters.

This chapter introduces you to some basic ABAP Objects syntax and the relevant development tools that you will need to start building object-oriented programs in ABAP.

2 Working with Objects

In this chapter, we will start getting our hands dirty by creating some simple object-oriented programs using ABAP Objects. Because the primary unit of development for object-oriented programs is the class, we will spend quite a bit of time examining the ABAP Objects syntax for defining new class types. Classes can be defined as global ABAP Repository objects (class pools), or locally within an ABAP program. Throughout the course of this chapter (and the rest of the book), you will see how to create and use both types of classes through a series of examples.

2.1 Syntax Overview

Before you begin writing object-oriented programs in ABAP, you must first learn about the syntax that is used to define ABAP Objects classes. An ABAP Objects class definition consists of a *declaration part* and an *implementation part*.

- ▶ The declaration part of the class definition is used to define all of the *components* of a class (i.e., attributes, methods, etc.).
- ▶ The implementation part of the class definition is used to provide implementations for the methods specified in the declaration part of the class definition.

In the following subsections, you will learn about the ABAP statements used to define local classes in an ABAP program. However, as you study these statements, keep in mind that the same syntax is being generated "behind the scenes" in the Class Builder tool when you edit global classes. You will see evidence of this in Section 2.4.5, Editing the Class Definition Section Directly.

2.1.1 Defining Classes

Listing 2.1 demonstrates the syntax used to define a local class called `lcl_myclass`. The `CLASS DEFINITION` statement is used to describe the properties and structure of the class. This example only declares the properties of the *components* of the class (e.g., attributes, method interfaces, etc.) – the implementation part comes later. The components of a class can be created within three *visibility sections*: the `PUBLIC SECTION`, the `PROTECTED SECTION`, or the `PRIVATE SECTION`. We will discuss these visibility sections in much more detail in Chapter 3, Encapsulation and Implementation Hiding. The discussion of the various `[class_options]` that can be applied to the `CLASS DEFINITION` statement is deferred to Chapter 3, Encapsulation and Implementation Hiding, Chapter 4, Object Initialization and Cleanup, and Chapter 5, Inheritance, where these optional features are described in context.

```
CLASS lcl_myclass DEFINITION [class_options].
    PUBLIC SECTION.
        [components]
    PROTECTED SECTION.
        [components]
    PRIVATE SECTION.
        [components]
ENDCLASS.
```

Listing 2.1 ABAP Class Definition Part Syntax

2.1.2 Declaring Components

The properties of a class are specified through its component definitions. You can define two different types of components within a class: *instance components* and *class components*.

- ▶ Instance components define the internal state and behavior of individual object instances. For example, an `Employee` class might have an instance attribute called `id` that uniquely identifies an employee within a company. Each instance of class `Employee` maintains its own copy of attribute `id`, which has a distinct value.
- ▶ A class component (or static component) is valid for all instances of a class. You use class components whenever it makes sense to share a component across all object instances.

All of the component names within a class belong to the same namespace. Therefore, for example, it is not possible to define an attribute and a method using the same name — even if they belong to different visibility sections. The following subsections describe the types of components that can be created within an ABAP Objects class.

Attributes

Attributes are used to describe the internal state of an object (or class). This state is represented in the form of data fields that can be declared using any valid ABAP data type.

- ▶ Instance attributes are declared using the familiar `DATA` keyword. These attributes define the instance-specific state of the object.
- ▶ Class attributes can be declared using almost the exact same syntax as that used to declare instance attributes. The only difference is the use of the `CLASS-DATA` keyword in lieu of the normal `DATA` keyword. All object instances share a single copy of a class attribute, which can come in handy in certain situations (see Section 2.2.5, Working with Class Components, for more details).
- ▶ Within a class definition, you can also create special class attributes called *constants*. Constants are declared using the `CONSTANTS` keyword. Constants must be assigned an initial value when they are declared, and this value cannot be subsequently changed. A constant is shared across all object instances just like normal class attributes. Constants should be named using the `CO_<constant name>` convention.

To demonstrate how to declare various types of attributes, consider the definition of local class `lcl_customer` in Listing 2.2. This class declares four private instance attributes: `id`, `customer_type`, `name`, and `address`. Some native data types, as well as some data elements and structures defined in the ABAP Dictionary, are used to declare these instance attributes. You can also create additional attributes using more complex types (e.g., table types, reference types, etc.) to model other properties of a customer, but you get the basic idea.

```
CLASS lcl_customer DEFINITION.  
  PUBLIC SECTION.  
    CONSTANTS: CO_PERSON_TYPE TYPE c VALUE '1',  
              CO_ORG_TYPE    TYPE c VALUE '2',  
              CO_GROUP_TYPE  TYPE c VALUE '3'.
```

```

PRIVATE SECTION.
  DATA: id TYPE numc10.
         customer_type TYPE c.
         name TYPE string.
         address TYPE adrc.
  CLASS-DATA: next_id TYPE numc10.
ENDCLASS.

```

Listing 2.2 Declaring Attributes in a Class Definition

The attribute `next_id` is a class attribute that keeps track of the next available customer ID number. Because all instances of class `lcl_customer` share the same copy of `next_id`, this attribute provides a convenient way for caching the customer ID number range. Three public constants, `CO_PERSON_TYPE`, `CO_ORG_TYPE`, and `CO_GROUP_TYPE`, represent the different types of customers supported in this simple class definition. Constants improve the readability of the class, giving semantic meaning to literal values that would otherwise have no significance to the naked eye.

The basic ABAP variable naming rules apply when defining attributes (see the context-sensitive help for the `DATA` keyword for more details). Of course, to improve the readability of the code, it is a good idea to give attributes meaningful names. Keep in mind that the semantic meaning of these attributes is defined in terms of the surrounding object (or class), so it is not necessary to qualify each and every attribute name.

- ▶ For instance, notice that the customer ID number attribute is not named `customer_id`. Whenever the `id` attribute is accessed, it is always accessed in the context of an object instance of type `lcl_customer`, so there is no need for such qualification.
- ▶ Also, notice that each of the attributes declared in Listing 2.2 were not given names with prefixes such as `G` or `L` to identify global or local variable scope, and so on. Because attributes are defined inside of an internal class namespace, there is only one scope, so this convention is not needed.
- ▶ Finally, attribute names share the same namespace as method names. Consequently, you should avoid the use of verbs in attribute names because this can conflict with potential method names.

Methods

The behavior of an object is expressed through its methods. Typically, methods are defined using the syntax shown in Listing 2.3.

```
METHODS my_method
  [IMPORTING parameters]
  [EXPORTING parameters]
  [CHANGING parameters]
  [RETURNING VALUE(parameter)]
  [EXCEPTIONS...].
```

Listing 2.3 General Method Declaration Syntax

This syntax defines a method called `my_method` that optionally supports various types of parameters. You can define the parameter interface for a method using the `IMPORTING`, `EXPORTING`, `CHANGING`, or `RETURNING` additions as shown in Listing 2.3. The `IMPORTING` addition is used to define input parameters that cannot be changed inside the method. The `EXPORTING` addition is used to define output parameters whose value is derived inside the method. Parameters defined using the `CHANGING` addition are input/output parameters that can be changed inside the method. We will explore parameters defined using the `RETURNING` addition shortly when we look at functional methods.

Regardless of the type, the syntax for declaring a parameter `p1` is given in the following Listing 2.4.

```
{ p1 | VALUE(p1) } TYPE type [OPTIONAL | {DEFAULT def1}]
```

Listing 2.4 Formal Parameter Declaration Syntax

Method parameters should be named according to the SAP naming conventions shown in Table 2.1.

Parameter Type	Naming Convention
IMPORTING	IM_<parameter name>
EXPORTING	EX_<parameter name>
CHANGING	CH_<parameter name>
RETURNING	RE_<parameter name>

Table 2.1 SAP Method Parameter Naming Conventions

The name of a method, along with its parameter declarations, represents the method's *signature*. The signature of a method defines how a method is invoked (or called). When a method that contains parameters is invoked, the calling program passes parameters by matching *actual parameters* (e.g., local variables in the calling program, literal values, etc.) in the method call with the *formal parameters* declared in the method signature (see Figure 2.1).

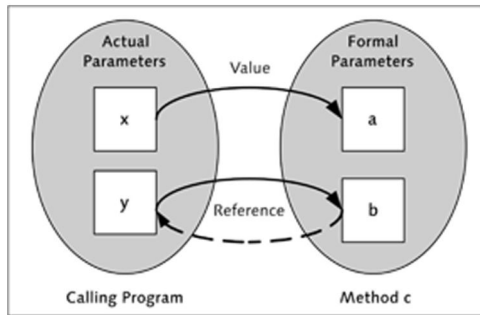


Figure 2.1 Mapping Actual Parameters to Formal Parameters

Parameters can either be passed into methods *by reference* (default behavior) or *by value*. The `VALUE(...)` addition implies that a parameter is passed into the method by value. This means that a copy of the actual parameter is created and passed to the method. Changes made to value parameters inside the method only affect the copy; the contents of the variable used as the actual parameter are not disturbed. In Figure 2.1, the formal parameter `a` of method `c` is defined as a value parameter. Therefore, when the calling program invokes method `c`, a copy of the actual parameter `x` is made, and the value is assigned to the formal parameter `a`.

Reference parameters contain a reference (or pointer) to the actual parameter (i.e., variable) that was used in the method call. Changes made to reference parameters *are* reflected in the calling program. In Figure 2.1, the formal parameter `b` of method `c` is defined as a reference parameter. This means that `b` points back to the actual parameter used in the method call in the calling program (`y` in this case). Therefore, if we change the value of parameter `b` inside method `c`, the change is actually made to the contents of variable `y` in the calling program.

By default, method parameters are defined as reference parameters. This improves the performance of method calls because it can be expensive (computa-

tionally) to make copies of large data objects at runtime whenever a method is called. In some programming languages, it is considered risky to pass parameters by reference because it is not always obvious where changes are being made to a given variable. ABAP Objects eliminates this subtle distinction by restricting changes to reference parameters passed as `IMPORTING` parameters to a method. If a calling program wants to directly manipulate the contents of a variable inside a method, then it must declare its intentions explicitly by mapping the variable to a `CHANGING` formal parameter. Note, however, that this does not apply to reference types such as object or data reference variables (more on these in Section 2.2.1, Object References).

It is also possible to define *functional methods* in classes. Functional methods are used to compute a single value (hence the use of the term *function*). Listing 2.5 shows the syntax used to declare functional methods. Here, as before, you can declare `IMPORTING` parameters to provide input to the method. However, notice that `CHANGING` or `EXPORTING` parameters cannot be defined in functional methods because the functional method only returns a single value — the `RETURNING` value parameter. You will see some interesting uses for functional methods in Section 2.2.6, Creating Complex Expressions Using Functional Methods.

```
METHODS func_method
  [IMPORTING parameters]
  RETURNING VALUE(rval) TYPE type
  [EXCEPTIONS...].
```

Listing 2.5 Functional Method Declaration Syntax

The `math` utility class `lcl_math` in Listing 2.6 declares four methods that demonstrate the method definition syntax described in this section:

- ▶ Method `max` receives two input value parameters `a` and `b` (of type integer) and returns the greater of the two values as a single exporting value parameter called `result` (also of type integer).
- ▶ Method `round` is used to round a floating-point parameter `a` to the nearest whole number. In this case, the changing parameter `a` is passed by reference and modified directly inside the method.
- ▶ Method `log` applies the logarithmic function to importing parameter `x` using base `b`. Notice that parameter `b` was defined using the `OPTIONAL` addition. This specification implies that callers are not required to pass a value for importing

parameter `b` (which might be defaulted to base 10 inside the method implementation, for example).

- ▶ Method `power` is a functional method that receives two importing parameters `base` and `exponent` and returns the value of the base raised to the exponent (value parameter `result`). The importing parameter `exponent` is also specified to contain a default value of 2.

The `DEFAULT` addition is similar to the `OPTIONAL` addition in the sense that they both create optional parameters. However, in the event that a caller does not pass a value for a `DEFAULT` parameter, the compiler will implicitly pass an actual parameter containing the specified default value (e.g., 2 in this example).

```

CLASS lcl_math DEFINITION.
  PUBLIC SECTION.
    METHODS:
      max IMPORTING VALUE(a) TYPE i
          VALUE(b) TYPE i
          EXPORTING VALUE(result) TYPE i,
      round CHANGING a TYPE f,
      log IMPORTING x TYPE f
          b TYPE i OPTIONAL
          EXPORTING y TYPE f,
      power IMPORTING base TYPE f
          exponent TYPE f DEFAULT 2
          RETURNING VALUE(result) TYPE f.
ENDCLASS.

```

Listing 2.6 Example Class Demonstrating Method Declarations

Class methods can be declared using almost the exact same syntax as that used to declare instance methods. The only difference is the use of the `CLASS-METHODS` keyword in lieu of the `METHODS` keyword used for instance methods.

Method names typically begin with a verb to emphasize the type of behavior that is being carried out in the method implementation. For example, a method used to create a sales order in a class called `lcl_sales_order` might be called `create`. Developers accustomed to creating verbose function module names might look at this name and find it too generic. However, remember that method names belong to the internal namespace of the class and are not subject to the potential naming clashes of global Repository objects such as function modules. In class `lcl_sales_order`, the method could have been named `create_order`, but this is

somewhat redundant because the `create` operation is being invoked on a sales order object. Getting used to the reflexive relationship between objects and methods takes a little time, so don't worry if it doesn't seem intuitive to you yet — there will be plenty of examples that will help you understand this relationship as we move forward.

Events

Classes can declare and trigger *events* that are handled by special *event handler methods*. Event handler methods can be defined within the same class that declared the event, or in a completely separate class. Events are defined within a class using the syntax shown in Listing 2.7.

```
EVENTS evt
  [EXPORTING parameters].
```

Listing 2.7 Instance Event Declaration Syntax

The syntax for defining `EXPORTING` formal parameters in events is identical to the syntax used to define formal parameters for methods (refer to Listing 2.4). However, it should be noted that these `EXPORTING` parameters must always be passed *by value*. Event parameters are used to pass additional information about the event to event handler methods. Events also pass an implicit parameter called *sender* that contains a reference to the *sending object* (i.e., the object that raised the event).

Class events can be created using the `CLASS-EVENTS` keyword. Other than the difference in keywords, the syntax for declaring class events is identical to that of regular instance events (see Listing 2.8).

```
CLASS-EVENTS evt [EXPORTING parameters].
```

Listing 2.8 Class Event Declaration Syntax

Events are handled by special event handler methods that are defined using the syntax shown in Listing 2.9.

```
METHODS evt_handler
  FOR EVENT evt of CLASS lc]_some_class
  [IMPORTING p1 p2 ... [sender]].
```

Listing 2.9 Event Handler Method Declaration Syntax

The syntax shown in Listing 2.9 declares an event handler method called `evt_handler` for an event `evt` defined in class `lcl_some_class`. The names of the importing parameters for an event handler method must match the signature of the exporting parameters defined in the event itself. However, unlike normal method declarations, you must not specify the types of importing parameters for event handler methods because this has already been specified in the event declaration.

In Chapter 10, Working with the SAP List Viewer, you will see how all of this fits together in an example report program. In particular, you will see how to register event handler methods to *listen* for events that are triggered using the `RAISE EVENT` statement.

Types

Custom data types can be defined within a class using the ABAP `TYPES` statement. These types are defined at the class level and are not specific to any object instance. You can use these custom types to define local variables within methods, and so on. It is also possible to declare the use of global type pools defined within the ABAP Dictionary using the `TYPE-POOLS` statement.

The definition of class `lcl_person` in Listing 2.10 provides an example that demonstrates how types can be declared and used in a class definition. The custom type `ty_name` is used to define the person's name attribute. Custom types have the naming convention `TY_<type name>`. The class also declares the use of type group `SZADR` from the ABAP Dictionary. This type group is defined within the SAP *Business Address Services* (BAS) package and contains various types related to addresses. BAS offers a streamlined API for working with addresses in applications. Here, type `SZADR_ADDRI_COMPLETE` from the `SZADR` type group is used to declare the `address` attribute for the `lcl_person` class.

```
CLASS lcl_person DEFINITION.
  PRIVATE SECTION.
    TYPES: BEGIN OF ty_name,
            first_name TYPE char40,
            middle_initial TYPE char1,
            last_name TYPE char40,
          END OF ty_name.
    TYPE-POOLS: szadr.      *Business Address Services
```

```

        DATA: name    TYPE ty_name,
              address TYPE szadr_addr1_complete.
    ENDCLASS.

```

Listing 2.10 Defining and Working with Types

2.1.3 Implementing Methods

In Section 2.1.2, *Declaring Components*, you learned how to define the various components of a class. If the declaration part of the class defined methods, then you must also create an implementation part that provides implementations for each of these methods to complete the class definition. The implementation part essentially contains the source code for the methods defined in the declaration part of the class. Listing 2.11 shows how to create the implementation part for the `lcl_math` class defined in Listing 2.6.

```

CLASS lcl_math IMPLEMENTATION.
    METHOD max.
        IF a > b.
            result = a.
        ELSE.
            result = b.
        ENDF.
    ENDMETHOD.
    METHOD round.
        "Implementation goes here...
    ENDMETHOD.
    METHOD log.
        "Implementation goes here...
    ENDMETHOD.
    METHOD power.
        "Implementation goes here...
    ENDMETHOD.
ENDCLASS.

```

Listing 2.11 Example Implementation for Class `lcl_math`

Each method defined in the declaration part of a class definition must be implemented inside of a `METHOD...ENDMETHOD` processing block within the implementation part of the class definition. Notice that no parameter specifications are included in the method processing block. These are not needed here because the declaration part of the class has already specified the method interface. Inside the

method processing block, you can implement the behavior of the class using regular ABAP statements in much the same way you would implement a procedural subroutine or function module. However, note that many obsolete/deprecated statements cannot be used in ABAP Objects classes. If you're not sure which statements have become deprecated over the years, don't worry, the compiler will tell you where you've gone wrong. Specific details concerning individual language elements are also described in the online help documentation (<http://help.sap.com>).

Method implementations can define local variables internally using the `DATA` keyword. Local variables are used to support the implementation of the method (e.g., as counters, temporary value placeholders, etc.). A general convention for defining local variable names is to prefix the name with an `L`. For example, a counter variable might have the name `lv_counter`. This naming convention is useful for avoiding potential naming conflicts with attributes created in the declaration part of the class definition. Although it is possible to create a local variable with the same name as an attribute, this is considered bad form because the local variable *hides* the attribute inside the scope of the method. Subtle scoping usages like this are hard to read and often cause errors that are difficult to debug.

2.2 Creating and Using Objects

Now that you have learned how to define classes in ABAP Objects, let's take a look at how you can create and use objects based on those class definitions. The following subsections will show you how to declare object reference variables, create object instances, and access components of those objects/classes. In Section 2.3, Building Your First Object-Oriented Program, we will put all of these pieces together to create a fully functional program.

2.2.1 Object References

The ABAP runtime environment does not allow direct access to objects inside a program. Therefore, to obtain access to an object at runtime, you must first declare an object reference variable. An object reference variable contains a reference (or pointer) to an object. When we examine the object creation process in Chapter 4, Object Initialization and Cleanup, you will come to appreciate the

need for this kind of indirection. For now, it is enough to know that you access objects through object reference variables that are declared using the syntax shown in Listing 2.12.

```
DATA: oref TYPE REF TO some_class.
```

Listing 2.12 Declaring Object References

The syntax in Listing 2.12 declares an object reference variable called `oref` that refers (or points to) objects of type `some_class`. The `REF TO` extension to the `TYPE` addition of the `DATA` statement designates that variable `oref` is an object reference variable that has the static type `some_class`. Object reference variables can be defined in any context where the declaration of variables is permitted (e.g., as global variables, local variables in subroutines or methods, attributes in a class, etc.).

2.2.2 Creating Objects

Object instances are created using the `CREATE OBJECT` statement. The code snippet in Listing 2.13 creates an object instance of type `some_class` and assigns a pointer to that instance to the object reference variable `oref`. The object creation and assignment process is completely controlled by the ABAP runtime environment. In Chapter 4, Object Initialization and Cleanup, you will see how special methods called *constructors* can be used to influence the creation process to perform attribute initializations, and so on.

```
DATA: oref TYPE REF TO some_class.  
CREATE OBJECT oref.
```

Listing 2.13 Syntax for Creating Objects

2.2.3 Object Reference Assignments

Object reference variables can be reassigned to point to different object instances using the `MOVE` statement or the assignment (`=`) operator. When assigning object reference variables, it is important to remember *what* you are assigning — namely references. Figure 2.2 depicts the relationship between two object reference variables (`Ref_1` and `Ref_2`) and the objects they point to (`Object_1` and `Object_2`, respectively).

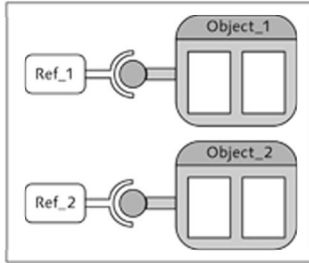


Figure 2.2 Reassigning Object References — Part 1

If, for example, you assign the value of `Ref_2` to `Ref_1` (see Figure 2.3), then both reference variables will contain an address that points to the same object instance (i.e., `Object_2`). In this case, if there are no other object reference variables pointing to `Object_1`, the object will be orphaned. In Chapter 4, Object Initialization and Cleanup, you will see how the ABAP runtime environment automatically cleans up these orphaned objects using a process known as *garbage collection*.

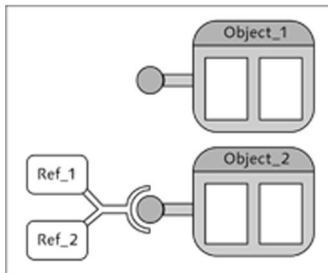


Figure 2.3 Reassigning Object References — Part 2

2.2.4 Working with Instance Components

To interact with an object instance in an ABAP program, you must use an object reference variable that *points* to that object instance because direct access to the object is strictly forbidden. One way to think about the relationship between an object reference variable and the object instance it refers to is to consider the connection between a remote control and a TV. Here, the remote control provides an interface that can be used to communicate with the TV (e.g., by pressing buttons).

Similarly, you can use an object reference variable to access instance components. This is achieved through the use of the *object component selector* operator. The object component selector (`->`) allows you to access the instance components of an object.

To demonstrate how to work with the object component selector, let's consider an example of a *Point* object in a Cartesian coordinate system. If you've slept since your last high school geometry class, a Cartesian coordinate system (or plane) is a two-dimensional grid that contains a horizontal x-axis and vertical y-axis (see Figure 2.4). You can plot points on a graph by specifying an x-coordinate and a y-coordinate, for example, point (1,2) in the graph of Figure 2.4.

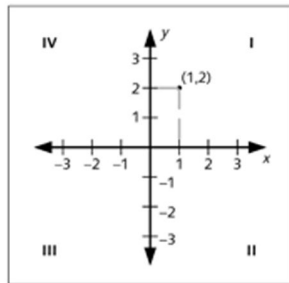


Figure 2.4 The Cartesian Coordinate System

The code in Listing 2.14 defines a class called `lcl_point` that represents a single point in a Cartesian coordinate system. The instance method `get_distance` is used to calculate the Euclidean distance between two points in the Cartesian plane. Because `get_distance` is an instance method, it must be accessed using an object reference variable. The object pointed to by this object reference variable implicitly becomes the first point; the second point is provided via the importing reference parameter `im_point2`.

```

CLASS lcl_point DEFINITION.
  PUBLIC SECTION.
    DATA: x TYPE i,           *X-Coordinate
           y TYPE i.         *Y-Coordinate

    METHODS get_distance IMPORTING im_point2
                       TYPE REF TO lcl_point

```

```

                                EXPORTING ex_distance
                                    TYPE f.
ENDCLASS.

CLASS lcl_point IMPLEMENTATION.
    METHOD get_distance.
    *   Method-Local Data Declarations:
        DATA: lv_dx TYPE i,          "Diff. X
              lv_dy TYPE i.          "Diff. Y

    *   Calculate the Euclidean distance between the points:
        lv_dx = im_point2->x - me->x.
        lv_dy = im_point2->y - me->y.
        ex_distance =
            SORT( ( lv_dx * lv_dx ) + ( lv_dy * lv_dy ) ).
    ENDMETHOD.
ENDCLASS.

```

Listing 2.14 Using the Object Component Selector — Part 1

To perform the calculation, the x- and y-coordinates for both points must be evaluated. To clarify the use of the attributes associated with the implicit first point, the *self-reference variable* `me` was used. Each object instance implicitly contains an object reference attribute called `me`. The `me` reference variable contains a reference to the object in which it is enclosed. The use of the self-reference variable is optional for accessing attributes within a method; the system will quietly insert it behind the scenes when omitted. The self-reference variable is primarily used for emphasis but is also occasionally used to pass a reference of the current object to another method, and so on.

The code snippet in Listing 2.15 shows how the object component selector is used to access public attributes (e.g., `x` and `y`) and public methods of class `lcl_point` using object reference variables. In this example, two object reference variables (`lr_point_a` and `lr_point_b`, respectively) are instantiated, assigning their x- and y-coordinates, and calculating the distance between the points. Methods are typically invoked in ABAP using the `CALL METHOD` statement, as evidenced by the call that you see to method `get_distance`.

```

* Local Data Declarations:
DATA: lr_point_a TYPE REF TO lcl_point.
      lr_point_b TYPE REF TO lcl_point.

```

```

lv_distance TYPE f.

* Instantiate both of the points:
CREATE OBJECT lr_point_a.
lr_point_a->x = 1.
lr_point_a->y = 1.

CREATE OBJECT lr_point_b.
lr_point_b->x = 3.
lr_point_b->y = 3.

* Calculate the distance and display the results:
CALL METHOD lr_point_a->get_distance
EXPORTING
    im_point2 = lr_point_b
IMPORTING
    ex_distance = lv_distance.

WRITE: 'Distance between point a and point b is: ',
       lv_distance.

```

Listing 2.15 Using the Object Component Selector — Part 2

2.2.5 Working with Class Components

During the analysis process, you will sometimes stumble across attributes and behaviors that are only associated with a class and not necessarily any particular instance of that class. Such components should be defined as *class components*. To demonstrate the use of class components, the `lcl_point` class has been revised from Listing 2.14 to make use of some class components (see Listing 2.16).

```

CLASS lcl_point DEFINITION.
PUBLIC SECTION.
    CONSTANTS: CO_QUADRANT_1 TYPE i VALUE 1,
               CO_QUADRANT_2 TYPE i VALUE 2,
               CO_QUADRANT_3 TYPE i VALUE 3,
               CO_QUADRANT_4 TYPE i VALUE 4.

    CLASS-DATA: next_point_no TYPE i. "Next Point Number

    DATA: point_no TYPE i,          "Point Number
           x TYPE i,                "X-Coordinate

```

```

        y TYPE i,                "Y-Coordinate

METHODS:
  constructor,
  get_point_number RETURNING value(re_number)
                    TYPE i,
  get_quadrant RETURNING value(re_quadrant)
                    TYPE i.

CLASS-METHODS:
  get_distance IMPORTING im_point1
                TYPE REF TO lcl_point
                im_point2
                TYPE REF TO lcl_point
                RETURNING value(re_distance)
                TYPE f.
ENDCLASS.

CLASS lcl_point IMPLEMENTATION.
  METHOD constructor.
    next_point_no = next_point_no + 1.
    point_no = next_point_no.
  ENDMETHOD.

  METHOD get_point_number.
    re_number = point_no.
  ENDMETHOD.

  METHOD get_quadrant.
    IF x > 0.
      IF y > 0.
        re_quadrant = CO_QUADRANT_1.
      ELSE.
        re_quadrant = CO_QUADRANT_4.
      ENDIF.
    ELSE.
      IF y > 0.
        re_quadrant = CO_QUADRANT_2.
      ELSE.
        re_quadrant = CO_QUADRANT_3.
      ENDIF.
    ENDIF.
  ENDMETHOD.

```

```

ENDMETHOD.

METHOD get_distance.
*   Method-Local Data Declarations:
    DATA: lv_dx TYPE i,           "Diff. X
           lv_dy TYPE i.         "Diff. Y

*   Calculate the distance between the two points:
    lv_dx = im_point2->x - im_point1->x.
    lv_dy = im_point2->y - im_point1->y.

    re_distance =
        SQRT( ( lv_dx * lv_dx ) + ( lv_dy * lv_dy ) ).
ENDMETHOD.
ENDCLASS.

```

Listing 2.16 Revising Class `lcl_point` to Use Class Components

As you can see in Listing 2.16, class `lcl_point` has been enhanced to include several different kinds of class components. The attribute `next_point_no` is a class attribute that is used to keep track of the next available point number that can be assigned to a point when it is created. To exploit this functionality, a special method called `constructor` has been defined. This method is called automatically by the ABAP runtime environment whenever an object of class `lcl_point` is created (see Chapter 4, Object Initialization and Cleanup, for more details). Inside the `constructor` method, the next available point number (i.e., `next_point_no`) is incremented, and the value is assigned to the instance attribute `point_no`. The value of the assigned point number can be retrieved using the `get_point_number` method.

In this implementation of class `lcl_point`, a new instance method has also been created called `get_quadrant` to return the current quadrant location for the point. Inside method `get_quadrant`, the public constant attributes `CO_QUADRANT_x` are used to specify the quadrant values rather than using hard-coded literal values. Within the bounds of class `lcl_point`, the use of the class attributes do not have to be qualified. However, outside of class `lcl_point`, these public constant attributes must be accessed using the *class component selector* (`=>`) operator.

For example, to access the constant value for Quadrant 1 in the coordinate plane, you could use the following expression: `lcl_point=>CO_QUADRANT_1`. The class component selector can be used to access any type of class component, including

methods, types, and so on. It is possible to use the class component selector in conjunction with an object reference variable (i.e., `oref=>class_component`). However, this syntax can become confusing as you will see later in the book. Therefore, this book sticks to the convention of binding the class component selector to the class name. This convention emphasizes the fact that the components are related to the class itself rather than a particular instance of the class.

If you look closely at class `lcl_point`, you will notice that the method `get_distance` has been redefined as a class method. Because `get_distance` is no longer invoked on a particular instance of class `lcl_point`, the method signature had to be changed to provide importing parameters for both points. Here, keep in mind that class methods cannot implicitly access instance attributes or instance methods within the class because there is no `me` self-reference associated with the static context of the class. Rather, class methods can only access instance attributes through object reference variables created locally or passed in as input parameters. Class methods can, of course, access other class methods and attributes without any such restrictions.

The example code shown in Listing 2.17 illustrates some of the updated features of class `lcl_point`. First, four object instances have been created: `lr_point_a`, `lr_point_b`, `lr_point_c`, and `lr_point_d`. Each time an object is created, the constructor method is invoked, and a unique point number is assigned using the class attribute `next_point_no`. This functionality is evidenced by the call to method `get_point_number` for the point referenced by `lr_point_c`, which should produce the expected value 3. Similarly, the call to method `get_quadrant` should determine that the point referenced by `lr_point_c` is located in Quadrant 3 based on the assigned x- and y-coordinate values. Finally, the call to class method `get_distance` calculates the distance between the points referenced by `lr_point_c` and `lr_point_d`. Here, notice the use of the class component selector in the `lcl_point=>get_distance` method call.

```
DATA: lr_point_a TYPE REF TO lcl_point.
      lr_point_b TYPE REF TO lcl_point.
      lr_point_c TYPE REF TO lcl_point.
      lr_point_d TYPE REF TO lcl_point.
      lv_point_number TYPE numc1.
      lv_quadrant     TYPE numc1.
      lv_distance     TYPE f.
```

```

* Create some sample point objects:
CREATE OBJECT lr_point_a.
lr_point_a->x = 2.
lr_point_a->y = 3.

CREATE OBJECT lr_point_b.
lr_point_b->x = 2.
lr_point_b->y = -1.

CREATE OBJECT lr_point_c.
lr_point_c->x = -3.
lr_point_c->y = -4.

CREATE OBJECT lr_point_d.
lr_point_d->x = -5.
lr_point_d->y = 1.

* Determine the point number & quadrant of point C:
lv_point_number = lr_point_c->get_point_number( ).
lv_quadrant = lr_point_c->get_quadrant( ).
WRITE: / 'Point C has Point #', lv_point_number,
        'and resides in quadrant', lv_quadrant.

* Calculate the distance between points C and D:
lv_distance =
    lcl_point=>get_distance( im_point1 = lr_point_c
                           im_point2 = lr_point_d ).
WRITE: / 'Distance between point C and point D is: ',
        lv_distance.

```

Listing 2.17 Working with Class Components

2.2.6 Creating Complex Expressions Using Functional Methods

The use of the `CALL METHOD` statement is optional for functional methods. This relaxed syntax supports the use of functional methods as *operands* in an expression. When used in an expression, functional methods are invoked, and the value of the `RETURNING` parameter is substituted into the expression *before* it is evaluated.

To demonstrate how this works in code, a simple class called `lcl_material` has been created in Listing 2.18 that is used to represent a material that might be produced by a given manufacturer. For the purposes of this basic example, only two

attributes are defined that store the material's ID number and a flag that indicates whether or not the material can be dangerous to handle. These instance attributes are called `material_number` and `hazardous_ind`, respectively. The instance method `is_hazardous` is a *boolean method* that returns the value of the `hazardous_ind` flag in the returning value parameter `re_result`. Boolean methods return a value of either *true* or *false* and should normally be named according to the convention `IS_<adjective>`.

```

CLASS lcl_material DEFINITION.
  PUBLIC SECTION.
    TYPE-POOLS: abap.
    DATA: material_number TYPE string,
           hazardous_ind  TYPE abap_bool.
    METHODS: is_hazardous RETURNING VALUE(re_result)
              TYPE abap_bool.
ENDCLASS.

CLASS lcl_material IMPLEMENTATION.
  METHOD is_hazardous.
    re_result = hazardous_ind.
  ENDMETHOD.
ENDCLASS.

```

Listing 2.18 Defining a Functional Method for a Material Class

The sample test code in Listing 2.19 creates an object of type `lcl_material` and assigns a reference to that object in object reference variable `lr_material`. After the material object is created, some test values are assigned to the instance attributes `material_number` and `hazardous_ind`. In the IF statement, the `is_hazardous` method is called to determine whether or not the material should be handled with care. The results of this method call are returned *before* the logical expression is evaluated. Thus, in the example, the logical expression evaluates to *true* because the `hazardous_ind` attribute was initialized to *true* right before the `is_hazardous` method was called. As you would expect, this causes the program flow to branch to the IF processing block of the IF statement.

```

DATA: lr_material TYPE REF TO lcl_material.

CREATE OBJECT lr_material.
lr_material->material_number = '1234567890'.
lr_material->hazardous_ind = abap_true.

```

```

IF lr_material->is_hazardous( ) EQ abap_true.
  WRITE: / 'Material', lr_material->material_number,
         'should be handled with caution!'.
ELSE.
  WRITE: / 'Material', lr_material->material_number,
         'can be handled normally.'.
ENDIF.

```

Listing 2.19 Using Functional Methods in Expressions

In the case of method `is_hazardous` from Listing 2.19, no parameters were passed in because there were no importing parameters defined. Of course, if a functional method does define importing parameters, these parameters must be provided in the method call. However, the syntax rules for passing these parameters are somewhat flexible. For example, if the functional method only provides a single importing parameter, you can simply provide the importing parameter using the syntax shown in Listing 2.20.

```
method_name( actual_parameter ).
```

Listing 2.20 Functional Method Calls with One Parameter

Similarly, if there is more than one importing parameter defined for a functional method, you can pass the parameters using the form shown in Listing 2.21.

```
method_name( p1 = f1 ... pn = fn ).
```

Listing 2.21 Functional Method Calls with Multiple Parameters

Obviously, the implementation of the `is_hazardous` method in Listing 2.19 is extremely contrived. Nevertheless, even a simple example such as this demonstrates the potential power in being able to wrap the result of a complex calculation neatly into an ABAP expression. Table 2.2 shows other places where you can use functional methods in ABAP expressions.

ABAP Expression	Where Used
MOVE	In the source field of the expression. Example: MOVE oref->meth() TO...

Table 2.2 Using Functional Methods in Expressions

ABAP Expression	Where Used
COMPUTE	In the arithmetic expressions. Example: COMPUTE c = oref->get_a() + oref->get_b(). or C = oref->get_a() + oref->get_b().
Logical expressions (e.g., IF)	As an operand in a boolean expression. Example: IF oref->get_weight() GT 100. ... ENDIF.
CASE/WHEN	As the operand in a CASE or WHEN statement. Example: CASE oref->get_type(). WHEN oref->get_value1(). ... ENDCASE.
LOOP AT/DELETE/MODIFY	In the WHERE clause. Example: LOOP AT itab WHERE field EQ oref->get_val(). ... ENDLOOP.

Table 2.2 Using Functional Methods in Expressions (cont.)

2.3 Building Your First Object-Oriented Program

In this section, we will create and test our first object-oriented program. In this example, we will integrate a local class called `lcl_date` inside of an executable program (i.e., a report program) named `YDATE_DEMO`. The `lcl_date` class encapsulates the concept of a date, providing some utility methods for displaying the date in various formats, and so on.

1. To create the test driver program, start up an SAP session, and open the Object Navigator (Transaction SE80).

- In the object list selection of the Repository Browser, select the PROGRAM option, enter the name of the program (see Figure 2.5), and press .

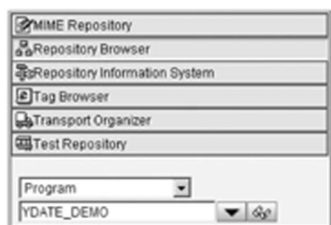


Figure 2.5 Creating a Test Driver Program

- When prompted to create the new Repository object, select the YES button (see Figure 2.6).

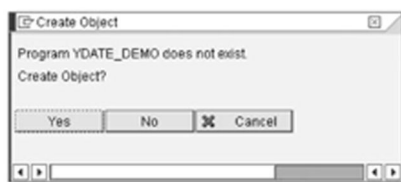


Figure 2.6 Create Repository Object Confirmation Dialog Box

- Next, in the CREATE PROGRAM dialog box (see Figure 2.7), you will be prompted to determine whether or not you want to create a *Top Include* for your program. Because we are only creating a simple report, you can de-select the WITH TOP INCL. checkbox, and press .

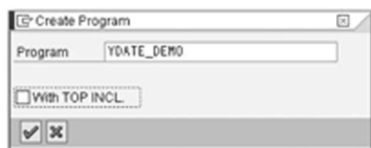


Figure 2.7 Create Program Dialog Box

- In the ABAP PROGRAM ATTRIBUTES dialog box (see Figure 2.8), you can confirm the various attributes for the program. Here, program type EXECUTABLE PRO-

GRAM has been selected. INCLUDE PROGRAM, MODULE POOL, and so on could also have been selected – local classes can be defined in many different types of programs. For now, accept the defaults, and click the SAVE button.

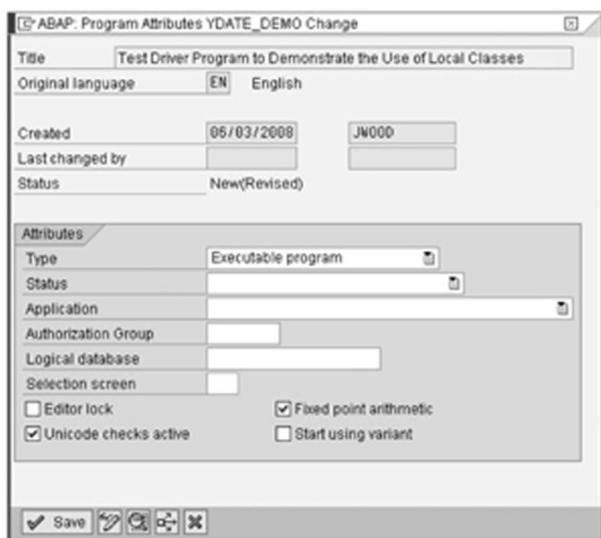


Figure 2.8 ABAP Program Attributes Dialog Box

- In the CREATE OBJECT DIRECTORY ENTRY dialog box (see Figure 2.9), you can specify a package that you want to create the program in, or you can select LOCAL OBJECT to create the program in the temporary objects (\$TMP) package. Objects created in this package are never transported, so this is a good place to work on examples.

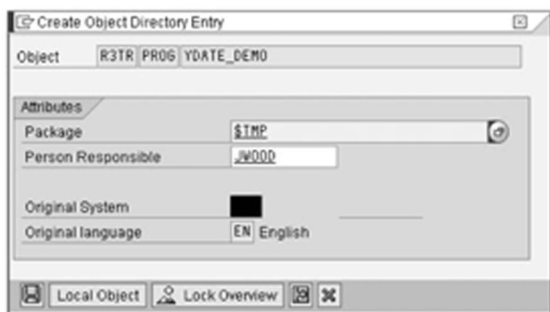


Figure 2.9 Create Object Directory Entry Dialog Box

7. Finally, you should be taken to an ABAP Editor screen where you can start coding. The source code for the example is given in Listing 2.22.

```
REPORT YDATE_DEMO.

CLASS lcl_date DEFINITION.
  PUBLIC SECTION.
    METHODS:
      set_date IMPORTING im_month TYPE numc2
                im_day   TYPE numc2
                im_year  TYPE numc4,
      as_native_date RETURNING value(re_date)
                    TYPE sydatum,
      display_short_format RETURNING value(re_date)
                          TYPE string,
      display_long_format RETURNING value(re_date)
                          TYPE string,
      get_day_of_week RETURNING value(re_weekday)
                      TYPE string,
      get_month_name RETURNING value(re_month)
                    TYPE string.

  PRIVATE SECTION.
    DATA: month TYPE numc2,      *Month: 1-12
          day   TYPE numc2,      *Day: 1-31
          year  TYPE numc4.      *Year
ENDCLASS.

CLASS lcl_date IMPLEMENTATION.
  METHOD set_date.
    month = im_month.
    day = im_day.
    year = im_year.
  ENDMETHOD.

  METHOD as_native_date.
    CONCATENATE year month day INTO re_date.
  ENDMETHOD.

  METHOD display_short_format.
    CONCATENATE month day year INTO re_date
      SEPARATED BY '/'.
  ENDMETHOD.
ENDCLASS.
```

```

ENDMETHOD.

METHOD display_long_format.
*   Local Data Declarations:
    DATA: lv_weekday TYPE string,    "Week Day (String)
           lv_month   TYPE string,    "Month Name

*   Determine the day of the week & the month name:
    lv_weekday = get_day_of_week( ).
    lv_month   = get_month_name( ).

*   Format the date string in longhand format:
    CONCATENATE lv_weekday ' '
                lv_month ' ' day ' ' year
                INTO re_date
                RESPECTING BLANKS.
ENDMETHOD.

METHOD get_day_of_week.
*   Local Data Declarations:
    DATA: lv_date TYPE sydatum,      "Date in Native Fmt.
           lv_day  TYPE p,            "Day Integral Value
           lt_day_names TYPE STANDARD TABLE
                OF t246,             "Day Names
           ls_day_name TYPE t246.     "Day Name

*   Use the functionality of the ABAP native date type
*   "D" to determine the week day (as an integer):
    lv_date = as_native_date( ).
    lv_day = lv_date MOD 7.
    IF lv_day GT 1.
        lv_day = lv_day - 1.
    ELSE.
        lv_day = lv_day + 6.
    ENDIF.

*   Use the standard function module DAY_NAMES_GET to
*   determine the name of the derived day value:
    CALL FUNCTION 'DAY_NAMES_GET'
        TABLES
            day_names          = lt_day_names
        EXCEPTIONS

```

```

        day_names_not_found = 1
        OTHERS                = 2.

    READ TABLE lt_day_names INTO ls_day_name
        WITH KEY wotnr = lv_day.
    IF sy-subrc EQ 0.
        re_weekday = ls_day_name-langt.
    ENDIF.
ENDMETHOD.

METHOD get_month_name.
*   Local Data Declarations:
    DATA: lt_month_names TYPE STANDARD TABLE
           OF t247, "Month Names
           ls_month_name TYPE t247. "Month Name

*   Determine the month name:
    CALL FUNCTION 'MONTH_NAMES_GET'
        TABLES
            month_names          = lt_month_names
        EXCEPTIONS
            month_names_not_found = 1
            OTHERS                = 2.

    READ TABLE lt_month_names INTO ls_month_name
        WITH KEY mnr = month.
    IF sy-subrc EQ 0.
        re_month = ls_month_name-ltx.
    ENDIF.
ENDMETHOD.
ENDCLASS.

* Global Data Declarations:
    DATA: gr_date    TYPE REF          "Date Object Ref.
           TO lcl_date,
           gv_display TYPE string.     "Date Display Value

START-OF-SELECTION.
*   Create an instance of class lcl_date:
    CREATE OBJECT gr_date.

*   Initialize the date to 9/13/2009:

```



```

CALL METHOD gr_date->set_date
EXPORTING
    im_month = '9'
    im_day   = '13'
    im_year  = '2009'.

* Display the date in shorthand format (e.g. mm/dd/yyyy):
gv_display = gr_date->display_short_format( ).
WRITE: / 'Date in shorthand format:', gv_display.

* Display the date in longhand format:
gv_display = gr_date->display_long_format( ).
WRITE: / 'Date in longhand format:', gv_display.

```

Listing 2.22 Program YDATE_DEMO

As you can see in Listing 2.22, program YDATE_DEMO creates an instance of class `lcl_date`, initializes the date value, and displays the date in a couple of different formats. The output of this program is shown in Figure 2.10.

Test Driver Program to Demonstrate the Use of Local Classes
Test Driver Program to Demonstrate the Use of Local Classes
Date in shorthand format: 09/13/2009
Date in longhand format: Sunday, September 13, 2009

Figure 2.10 Output of Example Program YDATE_DEMO

For demonstration purposes, the definition of class `lcl_date` is included inside the main executable program shown in Listing 2.22. In a real-world scenario, you would probably create an *include program* to define the local class just as you would normally do for other type declarations. You should do this partly because to avoid clutter in the main program, but the primary benefit comes from an increased opportunity for reuse. Local classes are only visible in the context in which they are defined. Therefore, if you define a local class in an include program, then that class can be included in other executable programs, module pools, and so on. So, if you want to work with local classes in your programs, it is highly recommended that you define them in a separate include program. Of course, normally you should to define such classes as global Repository objects. You will see how to do that in the following section 2.4, Getting Started with the Class Builder.

2.4 Getting Started with the Class Builder

The Class Builder is a fully integrated development environment inside the ABAP Workbench that can be used to edit global classes within the ABAP Repository. Most of the time, you will define your classes globally so that they can be reused in other programs more easily. Therefore, it is important to get comfortable working with the Class Builder because you will likely be spending a lot of time there.

2.4.1 Class Pools

Global classes are stored within the ABAP Repository inside of a class pool. A class pool is a special ABAP program type that defines a single global Repository class along with related local type definitions used to support the implementation of the class. Class pools are similar to function groups in the sense that they cannot be executed directly. Instead, runtime object instances are created in reference to the global class type using the same `CREATE OBJECT` statement described in Section 2.2.2, Creating Objects.

2.4.2 Accessing the Class Builder

The Class Builder can be accessed via Transaction SE24, or using the menu path **TOOLS • ABAP WORKBENCH • DEVELOPMENT • SE24 – CLASS BUILDER**, as shown in Figure 2.11.



Figure 2.11 Selecting the Class Builder in the SAP Easy Access Menu

It is also possible to access the Class Builder from within the Object Navigator (Transaction SE80). Here, you can select the `CLASS/INTERFACE` entry in the object list selector drop-down list in the Repository Browser, enter the name of the class in the corresponding object type field, and click the `DISPLAY` button. This will load

the class inside an instance of the Class Builder that is embedded inside the content area of the Object Navigator (see Figure 2.12).

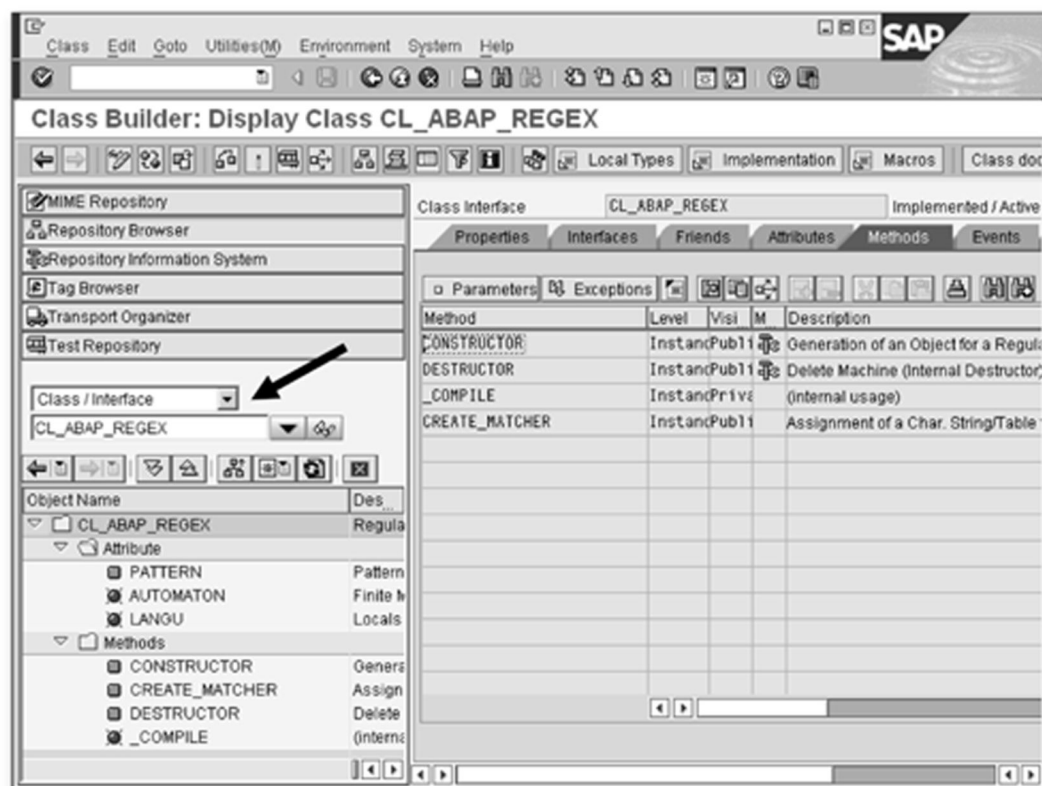


Figure 2.12 Accessing the Class Builder Inside the Object Navigator

2.4.3 Creating Classes

After starting the Class Builder via Transaction SE24, you will be presented with the overview screen shown in Figure 2.13.

1. To create a new class in the ABAP Repository, enter the name of the class in the OBJECT TYPE field. Class names can be up to 30 characters long and should be named according to the convention shown in Listing 2.23.

```
[Y|Z|Namespace]CL_<class_name>
```

Listing 2.23 General Syntax for Defining Global Classes

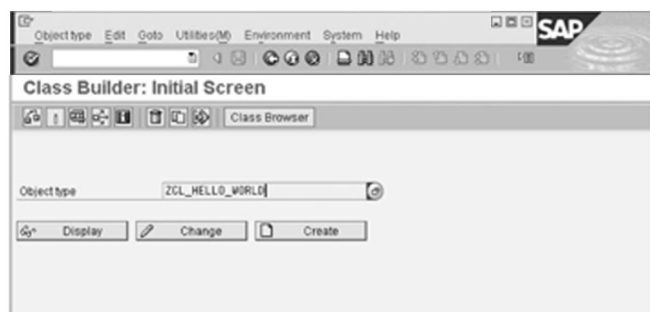


Figure 2.13 Class Builder – Initial Screen

Because classes are meant to represent some kind of real-world phenomena, the `class_name` portion of the naming convention should be made up of *singular nouns*. For example, a purchase order class might be called `ZCL_PURCHASE_ORDER`.

2. After entering the class name, click on the **CREATE** button. Next, you will be prompted with the **CREATE CLASS** dialog box shown in Figure 2.14. Here, you can enter a short description of the class (**DESCRIPTION**), the instantiation type (**INSTANTIATION**), and the class type (**CLASS TYPE**). We will discuss the instantiation type in Chapter 4, Object Initialization and Cleanup; for now, accept the default **PUBLIC** value.



Figure 2.14 Create Class Dialog Box

3. Accept the default value for class type (e.g., **USUAL ABAP CLASS**) as well. The remaining class types are described in Chapter 8, Error Handling with Excep-

tions, Chapter 11, ABAP Object Services, and Chapter 9, Unit Testing with ABAP Unit, respectively. The `FINAL` checkbox is related to inheritance, which is covered in Chapter 5, Inheritance; for now, leave it de-selected.

4. Finally, the `ONLY MODELED` checkbox is used to exclude the class from the class pool that gets saved to the ABAP Repository. This attribute can be used to make sure that classes in the early stages of development (i.e., modeled classes) are not used in the runtime environment. Such classes can be implemented later by selecting `CLASS • IMPLEMENT` from the menu bar of the Class Editor.
5. When you click the `SAVE` button, you are prompted with the `CREATE OBJECT DIRECTORY ENTRY` dialog box shown in Figure 2.9. Here, you can select the relevant package name (or the local package `$TMP`) and click the `SAVE` button to save the class.

After the class is saved, the new class is displayed in an inactive state in the Class Editor screen (see Figure 2.15).

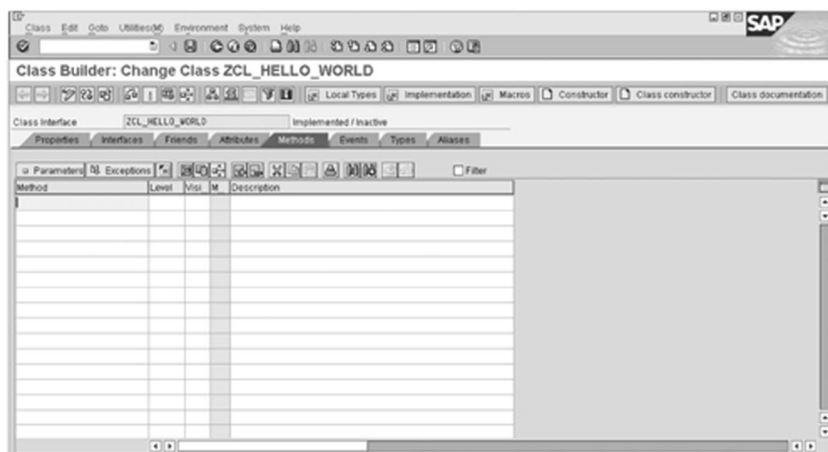


Figure 2.15 Class Editor Screen

2.4.4 Defining Class Components

You can define the components of a global class in the Class Editor part of the Class Builder as shown in Figure 2.15. The following subsections describe how to create various types of components within the Class Editor.

Attributes

Attributes are defined on the ATTRIBUTES tab of the Class Editor. This tab of the Class Editor provides an entry table in which you can specify all of the various attributes for a class (see Figure 2.16).

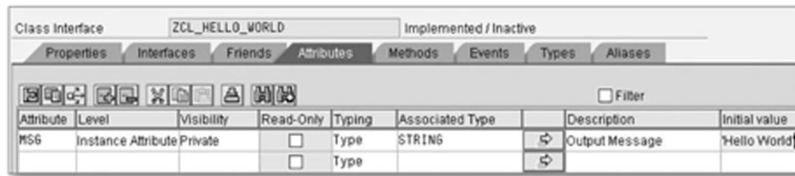


Figure 2.16 Defining Attributes in the Class Editor

The name of an attribute is specified in the ATTRIBUTE column. The LEVEL column is used to define the declaration type of the attribute. Figure 2.17 shows a list of possible declaration types available in the input value help for this column. As you can see, global class attributes can be declared as an INSTANCE ATTRIBUTE, a STATIC ATTRIBUTE, or a CONSTANT.



Figure 2.17 Setting the Declaration Type for an Attribute

The VISIBILITY column is used to define the visibility section assignment for an attribute (for a list of possible values, see Figure 2.18).

The TYPING/ASSOCIATED TYPE columns are used to declare the data type for an attribute. The TYPING field qualifies the type assigned to the attribute by specifying the *typing method* (see Figure 2.19). These typing methods are described in more detail in Table 2.3.



Figure 2.18 Setting the Visibility Area for a Component

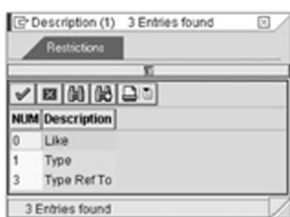


Figure 2.19 Setting the Typing Method for an Attribute

Typing Method	Description
LIKE	Defines an attribute's type by referencing the type associated with a class attribute from any globally defined class.
TYPE	Defines an attribute's type using an ABAP native type or a global type defined in the ABAP Dictionary.
TYPE REF TO	Defines an attribute's type in reference to an object type (i.e., another global class) or a data type (i.e., a data reference).

Table 2.3 ABAP Objects Typing Methods

It is also possible to define an attribute's type directly by clicking on the **DIRECT TYPE ENTRY** button in the column to the right of the **ASSOCIATED TYPE** column. This button brings you to an ABAP Editor screen where you can specify the attribute type in the same way that you learned how to define attributes for local classes in Section 2.1.2, *Declaring Components*.

In the **DESCRIPTION** column, you can enter a basic descriptive text that describes the purpose of the attribute. The **INITIAL VALUE** column can be used to define an initial value for elementary attributes in a class.

Methods

Methods are defined on the **METHODS** tab in the Class Editor. Here, much like you saw on the **ATTRIBUTES** tab, the Class Editor provides you with an input table for defining the methods of a class (see Figure 2.20).

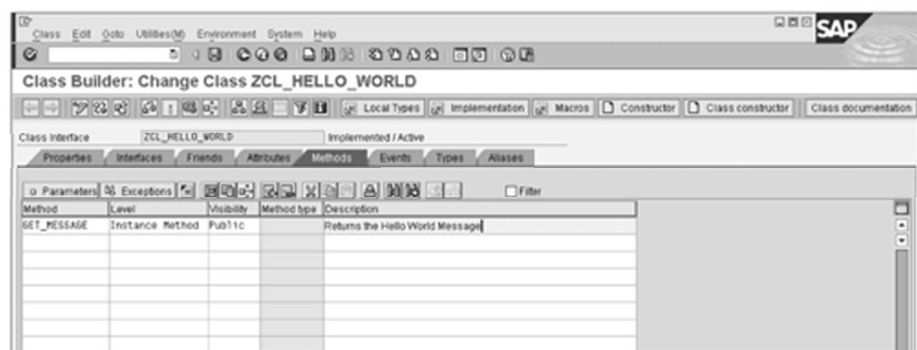


Figure 2.20 Defining Methods in the Class Editor

The method name is entered in the **METHOD** column. The **LEVEL** column allows you to specify whether the method is an *instance method* or a *static method* (see Figure 2.21). In the **VISIBILITY** column, you can determine which visibility section that you want to assign the method to (refer to Figure 2.18 for a list of possible visibility section options). You can also enter an optional descriptive text for the method in the **DESCRIPTION** column.

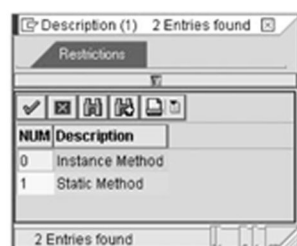


Figure 2.21 Setting the Declaration Type for a Method

After you have specified the basic attributes of the method, you can complete its signature definition by declaring its parameter interface.

1. To define the parameters for a method, place your cursor on the name of the method that you want to edit in the METHOD column.
2. Click on the PARAMETERS button in the toolbar above the method input table. This will open up the METHOD PARAMETERS input screen for the selected method (see Figure 2.22).

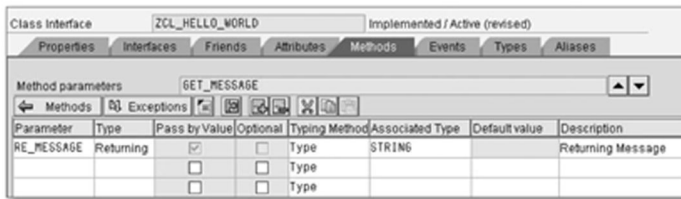


Figure 2.22 Defining Method Parameters

3. In the METHOD PARAMETERS input table, you can specify the following:
 - ▶ The name of a parameter in the PARAMETER column; refer to Table 2.1 for a list of naming conventions.
 - ▶ The TYPE column is used to specify whether the parameter is an importing, exporting, changing, or returning parameter.
 - ▶ The PASS BY VALUE column contains a checkbox that indicates whether or not the parameter should be passed by value or by reference.
 - ▶ The OPTIONAL column contains a checkbox that can be used to mark a parameter as optional when the method is called. When selected, you can optionally specify a default value for the parameter in the DEFAULT VALUE column.
 - ▶ The TYPING METHOD/ASSOCIATED TYPE columns are used to define the type of the parameter in the same way that attribute types are defined on the ATTRIBUTES tab. You can also enter an optional descriptive text to describe the parameter in the DESCRIPTION column.
4. After you have finished specifying the method parameters, you can click on the METHODS button in the toolbar above the method parameters to return to the normal method editor view.
5. To edit the method's implementation, you can either double-click on the method name, or click on the SOURCE CODE button (see Figure 2.23).



Figure 2.23 Navigating to the Method Source Code Editor

6. In the source code editor, you can implement the method inside of a METHOD... ENDMETHOD processing block within the ABAP Editor as per usual (see Figure 2.24).



Figure 2.24 Editing a Method Implementation in the ABAP Editor

Events

Events are defined on the EVENTS tab of the Class Editor. Here, you are provided with an input table similar to the ones used to specify components on the ATTRIBUTES and METHODS tabs (see Figure 2.25).

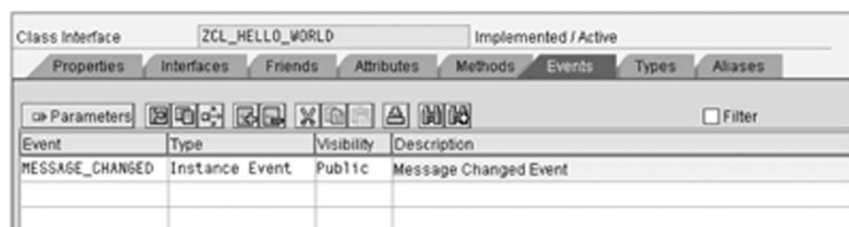


Figure 2.25 Defining Events in the Class Editor

- In the event input table, you can enter the following:
 - ▶ The name of an event in the EVENT column.
 - ▶ The TYPE column is used to specify the declaration type of the event (i.e., *instance event* versus *static event*).
 - ▶ In the VISIBILITY column, you can assign the event to a visibility section in the class (refer to Figure 2.18 for a list of possible visibility section options).
 - ▶ The optional DESCRIPTION field can be used to enter a descriptive text about the event.
- To declare the parameters for the event, place your cursor on the name of the target event in the EVENT column, and click the PARAMETERS button in the toolbar above the input table. This opens up the EVENT PARAMETERS input screen shown in Figure 2.26.

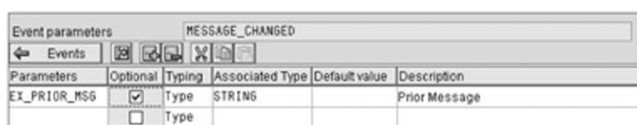


Figure 2.26 Defining Event Parameters

- Here, you can specify the exporting parameters of the event in much the same way that you learned how to define method parameters in the previous section.

Types

Custom data types can be defined on the TYPES tab of the Class Editor. As you can see in Figure 2.27, the TYPES tab provides an input table that you can use to specify the details of the custom types.

- In the TYPE column, you can enter type names according to the naming convention `TY_<type name>`.
- The VISIBILITY column is used to assign the custom type to a particular visibility section with the class (refer to Figure 2.18 for a list of options).
- The TYPING METHOD/ASSOCIATED TYPE fields are used to define the type in the same way that attribute types are defined on the ATTRIBUTES tab.

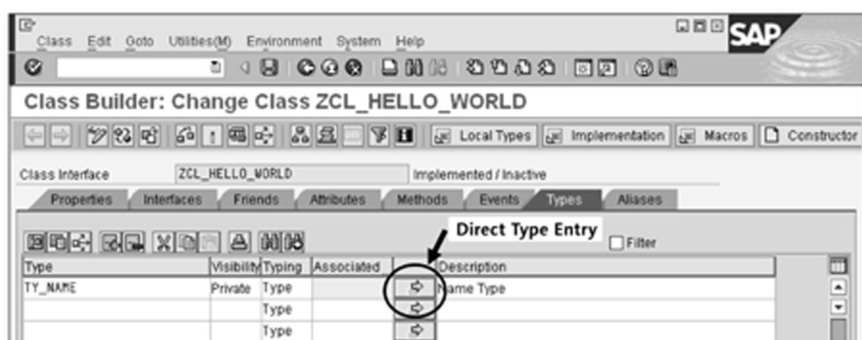


Figure 2.27 Defining Custom Types in the Class Editor

As was the case with attributes, it is also possible to specify custom types directly using the `DIRECT TYPE ENTRY` button (see Figure 2.27). Clicking on this button takes you to an ABAP Editor screen that allows you to edit the declaration part of your class for the related visibility area (e.g., `PUBLIC SECTION`, `PRIVATE SECTION`, etc.).

In Figure 2.28, type `TY_NAME` is declared as a structure containing three components to represent the first name, middle initial, and last name, respectively.

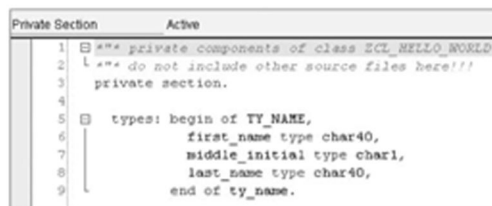


Figure 2.28 Defining Custom Types Using Direct Type Entry

Local types can also be defined outside of the class in the class pool. These types are used to support the creation of local variables in method implementations, and so on. To define these types, click on the `LOCAL TYPES` button in the top-level toolbar of the Class Editor.

Local Inner Class Definitions

The Class Builder also allows you to create local helper classes inside the class pool. These classes are useful for implementing internal details of the class and

should be used to simplify the interface of the global class by removing an overabundance of private helper methods.

You can edit local classes by clicking on the `IMPLEMENTATION` button in the top-level toolbar of the Class Editor. This will take you to an ABAP Editor screen where you can create local classes using the same local class definition syntax described earlier in this chapter.

2.4.5 Editing the Class Definition Section Directly

As stated previously, the Class Builder tool quietly generates ABAP Objects class definition syntax behind the scenes as you edit global classes. You can access the definition sections of a global class by selecting the menu options `GOTO • PUBLIC SECTION`, `GOTO • PROTECTED SECTION`, and `GOTO • PRIVATE SECTION`, respectively, in the Class Builder menu bar. For example, if you look closely at the ABAP Editor screenshot shown in Figure 2.28, you will notice that the custom type `TY_NAME` is being edited directly in the `PRIVATE SECTION` of the global class definition.

2.5 Case Study: Working with Regular Expressions

The demonstrative classes that we have reviewed thus far have been extremely straightforward and uncomplicated. However, one of the beauties of object-oriented programming is the fact that you can encapsulate complex logic inside of a class that can be used by programmers who may or may not understand how it works.

SAP provides many useful global classes in the standard distribution that can be used right out of the box. An example of this is the *regular expression* API provided in classes `CL_ABAP_REGEX` and `CL_ABAP_MATCHER`. A regular expression is a text string that is used to describe a search pattern in text. A regular expression consists of literal characters and *metacharacters*. You can think of metacharacters as a type of shorthand for describing certain character patterns — similar to the use of the asterisk for representing one or more characters in a search help lookup (see Figure 2.29). However, regular expressions are much more powerful, providing a more generalized syntax that can be used to express many different types of complex text patterns.

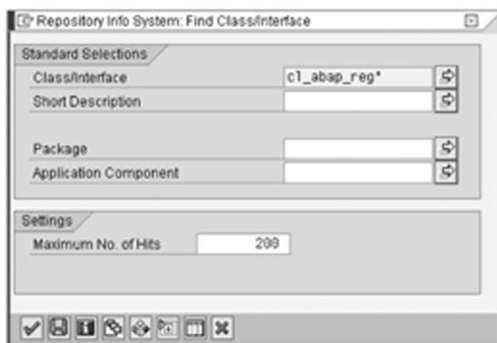


Figure 2.29 Using Metacharacters in a Search Help Query

An example of a regular expression that could be used to match a telephone number in the (xxx)xxx-xxxx format is shown in Figure 2.30.

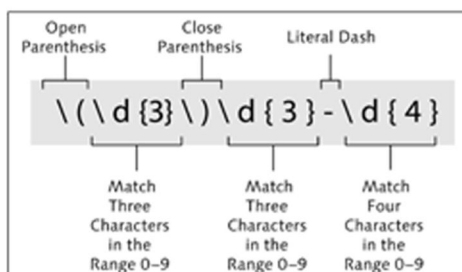


Figure 2.30 Regular Expression for Matching Phone Numbers

The report program YREGEX_DEMO shown in Listing 2.24 shows how to use the ABAP regular expression class library to validate a phone number entered as a parameter on a selection screen.

```
REPORT yregex_demo.

SELECTION-SCREEN BEGIN OF BLOCK blk_main.
  PARAMETERS:
    p_phone TYPE ad_tlnmbr1.
SELECTION-SCREEN END OF BLOCK blk_main.

AT SELECTION-SCREEN ON p_phone.
  PERFORM check_phone_number.
```

```

START-OF-SELECTION.
  WRITE: / 'You entered: ', p_phone.

FORM check_phone_number.
* Local Data Declarations:
  DATA: lr_regex TYPE REF TO cl_abap_regex,
        lr_matcher TYPE REF TO cl_abap_matcher.

* Create the regular expression:
  CREATE OBJECT lr_regex
    EXPORTING
      pattern = '\(\d{3}\)\d{3}-\d{4}'.

* Check to see if the phone number matches the
* regular expression:
  CREATE OBJECT lr_matcher
    EXPORTING
      regex = lr_regex
      text = p_phone.

  IF lr_matcher->match( ) NE abap_true.
    MESSAGE 'Enter phone number in (xxx)xxx-xxxx format.'
      TYPE 'E'.
  ENDIF.
ENDFORM.

```

Listing 2.24 Using Global Classes in ABAP Programs

As you can see in Listing 2.24, you can perform a fairly complex validation with a handful of statements — the classes perform the heavy lifting. Therefore, even if you are not yet an expert in creating classes, you can still put classes to work for you immediately in your programs.

2.6 UML Tutorial: Object Diagrams

Section 1.6, UML Tutorial: Class Diagram Basics, showed how class diagrams can be used to specify the static architecture of an object-oriented system. Most of the time, these diagrams are straightforward and easy to interpret. However, sometimes the relationship between certain classes is not so intuitive. In these cases, *object diagrams* can be used to depict a snapshot or simulation of the actual objects created in reference to these classes at runtime. Often, just seeing an

example of how the actual objects are configured at runtime can shed some light on the nature of complex class relationships.

Figure 2.31 illustrates a portion of a class diagram that shows the recursive aggregation relationship between a bill of material (BOM) document and its items. The diamond on the `MaterialBOM` side of the association is used to indicate that the BOM is an *aggregate*, containing 0 or more items.

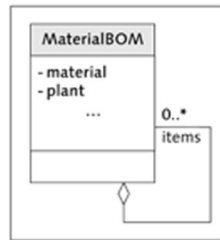


Figure 2.31 Class Diagram Showing Material BOM Aggregation

A BOM contains a series of items (or components) that are used to assemble a finished product. In complex engineering scenarios, it is not uncommon for a BOM to contain items that are also complex assemblies. Figure 2.32 demonstrates an object diagram that shows a BOM object for a laptop computer that is being produced by a computer hardware manufacturer. The `laptop` object is comprised of multiple components (e.g., a hard drive, motherboard, LCD display, etc.) The `motherboard` object is also an aggregate, containing a CPU and chipset.

As you can see from Figure 2.32, object diagrams are very similar to class diagrams in many respects. However, in an object diagram, the rectangular boxes represent object instances instead of classes. Figure 2.33 shows the basic notation for specifying objects in an object diagram. In the top box, you provide the name of the object as well as the type of class the object is created in reference to. The lower box is optional, allowing you to provide additional runtime details about the object (i.e., its current state).

Object diagrams can display as many objects as needed to illustrate the class/object relationships. The diagram is considered to be a viewport into the system at a particular point in time. Objects are created and destroyed in programs all of the time, so it is important not to get hung up on trying to illustrate every possible object that will be created in the system at runtime. If one diagram cannot

fully describe the relationship, additional diagrams can be used to show the progression of the object configuration as the program continues to run.

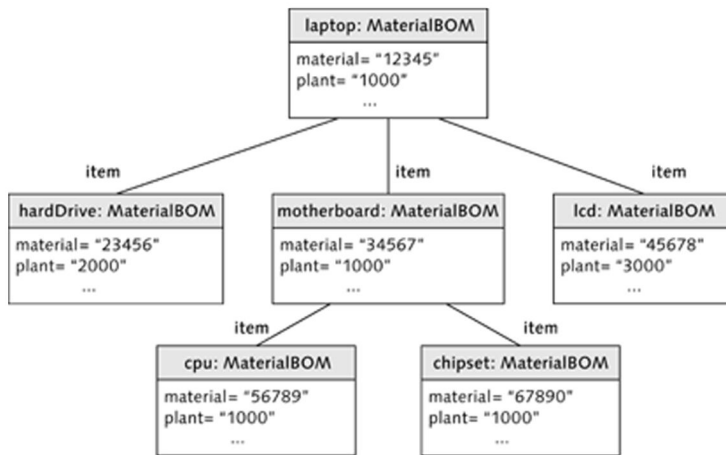


Figure 2.32 Object Diagram Showing BOM for a Laptop Computer

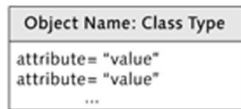


Figure 2.33 Object Instance Notation in Object Diagrams

2.7 Summary

At this point, you should have learned enough ABAP Objects syntax to begin writing object-oriented programs in ABAP. However, although this chapter provided a nuts-and-bolts description of classes and objects, it only scratched the surface with regards to the potential benefits that can be obtained by adopting an object-oriented approach to your program designs.

In the next chapter, we will begin to explore some of these features by examining the concepts of encapsulation and implementation hiding.

PART II
Core Concepts

Classes are abstractions that can be used to extend the functionality of a programming language by introducing user-defined types. Encapsulation and implementation hiding techniques are used to simplify the way developers interact with these types, making object-oriented programs easier to understand, maintain, and enhance. In this chapter, you will learn how to apply these techniques to your ABAP Objects classes.

3 Encapsulation and Implementation Hiding

One of the most obvious ways to speed up the software development process is to leverage pre-existing code. However, while most projects strive to build reusable code libraries, few actually succeed in delivering modules that can be used outside of the context in which they were originally conceived. In most cases, this is because the module is too tightly coupled with its surrounding environment. Of course, without clairvoyance, it is difficult to anticipate how and when a given module might be reused in other contexts. Instead of trying to predict future usage types, pragmatic developers look for ways to build autonomous components that can *think* and *act* on their own within a defined set of boundaries.

In this chapter, we will investigate ways to breathe life into objects by learning how to exploit some of the potential benefits associated with encapsulating data and behavior together within a class. Along the way, we will explore the use of access control mechanisms that help to shape the interfaces of these classes to make them easier to modify and reuse in other contexts.

3.1 Lessons Learned from the Procedural Approach

Contrary to popular belief, many core object-oriented concepts are based on similar notions that are rooted in the procedural programming paradigm. In both disciplines, the basic goal is to bring order and reliability to the software development process. Clearly, there is a lot to be learned from procedural and structured

programming techniques. However, as you will see in this section, there are certain limitations to the procedural approach that you must overcome to improve the overall quality of your software designs.

3.1.1 Decomposing Functional Decomposition

Procedural developers typically formulate their program designs using a process called *functional decomposition*. This term comes from the mathematics world, where a mathematical function is broken down into a series of smaller functions that are easier to understand. From a development perspective, functional decomposition refers to the process of *decomposing* a complex program into a series of smaller modules (or procedures). One common approach for discovering these procedures is to scan through the verbs used to describe the *actions* of a program within the functional requirements. These actions represent the *steps* a program must take to meet its objectives. After all of the steps have been identified, they must be *composed* into a main program that is responsible for making sure that procedures are called in the right order, and so on. The process of organizing and refining the main program is sometimes called *step-wise refinement*.

For small- to medium-sized programs, this strategy works pretty well. However, as programs start to branch out and grow in complexity, the design tends to become unwieldy as the main program becomes saddled with too many responsibilities. Much of this burden stems from the fact that the main program must be accountable for all of the data used by the various procedures.

Ideally, you want to be able to delegate some of these management duties to procedures, but for that to happen, the procedures need to be smart enough to figure certain things out on their own — and that requires data. Of course, a main program can pass instructions to a procedure via parameters, but the downside to this approach is that cluttering up a procedure's parameter interface causes it to be *tightly coupled* to the calling program that provides the data. These kinds of dependencies cause all kinds of maintenance problems and also make it much more difficult to reuse the procedures in other environments. On the other hand, liberal use of global data within procedures is also dangerous. For example, think about a program that has a series of procedures that all share and manipulate a piece of global data. If you are asked to change the sequence of the procedure calls, can you be certain that this change will not result in some kind of unpredictable data corruption scenario?

As you can see, although functional decomposition helps to break a program down into smaller chunks (i.e., procedures) that are easier to understand, it does not necessarily guarantee that a program will hold up to the changing requirements that inevitably creep in over time. Clearly, a better way of structuring software is needed.

3.1.2 Case Study: A Procedural Code Library in ABAP

To demonstrate some of the problems associated with functional decomposition, let's consider an example that shows how you might construct a Date utility library using procedural function groups written in ABAP. Prior to SAP R/3 Release 4.0, most reusable code libraries written in ABAP were built using function groups. In some respects, function groups are analogous to classes in the sense that you can define global data (attributes) and function modules (methods) centrally within a function pool inside the ABAP Repository. However, this analogy breaks down when you consider the fact that you cannot load multiple *instances* of a function group inside your program¹. This limitation makes it difficult for function module developers to work with global data in the function group because additional logic is required to partition the data into separate work areas (i.e., instances).

The typical workaround for this shortcoming is to store the data locally within the calling program and create a series of *stateless* function modules to operate on the data. Stateless function modules do not have any recollection of prior invocations; each time that you call them, you must pass in all of the data that they will need to perform their task(s).

Listing 3.1 shows a simple function group called `ZDATE_API` that defines the Date library. Normally, this library would contain many other function modules to maintain the components of the date, display the date in various localized formats, and so on. However, for the purposes of this discussion, it simply describes a single function module called `Z_DATE_SET_DAY` that is used to assign the day value for the date.

¹ Whenever you call a function module from a particular function group inside your program, the global data from the function group are loaded into the memory of the internal session of your program. Any subsequent calls to function modules within that function group will share the same global data allocated whenever the first function module was called.

3 | Encapsulation and Implementation Hiding

```
FUNCTION-POOL zdate_api.  
FUNCTION z_date_set_month.  
    ...  
ENDFUNCTION.  
FUNCTION z_date_set_day.  
    * Local Interface IMPORTING VALUE (iv_day) TYPE I  
    *                   CHANGING (cs_date) TYPE SCALS_DATE  
    *                   EXCEPTIONS invalid_date  
    DATA: lv_month_end TYPE i.          "Last Day of Month  
  
    CASE is_date-month.  
        WHEN 1.  
            lv_month_end = 31.  
        WHEN 2.  
            ...  
    ENDCASE.  
    IF iv_day LT 1 OR iv_day GT lv_month_end.  
        RAISE invalid_date.  
    ELSE.  
        cs_date-day = iv_day.  
    ENDIF.  
ENDFUNCTION.  
...  
...  
...
```

Listing 3.1 A Simple Date Library Built with Function Groups

The logic inside function module `Z_DATE_SET_DAY` is pretty straightforward. First, the value of importing parameter `IV_DAY` is examined to ensure that the `DAY` field in structure `CS_DATE` (see Figure 3.1) is not assigned an invalid value based on the current month assignment (handling leap year situations, etc.). Assuming all of the validations are passed, the `DAY` field is updated in the `CS_DATE` structure; otherwise, an exception is thrown, and the update does not occur.

Now that we have established our simple Date library, let's think about how we might handle a couple of maintenance scenarios that might pop up over time:

- ▶ First, imagine that you are asked to expand the functionality of the Date library to also keep track of time. Here, you are faced with a dilemma. Changing the representation of the date (i.e., switching to a structure containing both date and time components) requires wholesale changes not only to the function modules but also to the programs that call them.

- ▶ Next, consider a program that is using the Date library to output a date in various formats. You are assigned a defect for this program because it is displaying invalid date values (e.g., 02/31/2009). During your investigation, you discover that the invalid day value was not set by function `Z_DATE_SET_DAY`, but rather by an invalid assignment that was made to the `DAY` field in the local date structure maintained in the calling program.

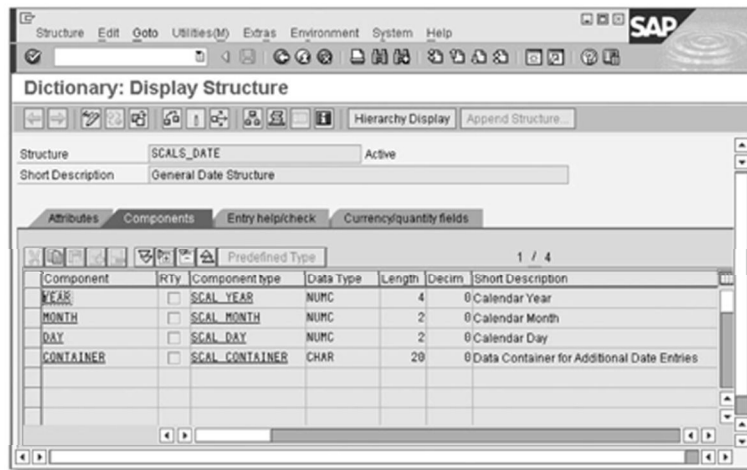


Figure 3.1 ABAP Dictionary Structure SCALS_DATE

The previous maintenance scenarios demonstrate some serious flaws in our date library. Because the data object containing the date fields is managed outside of the function group (i.e., in the calling programs), we cannot control the values that are assigned to it externally. Within the `SCALS_DATE` structure definition, the field `DAY` is simply a two-digit numeric character with a valid number range of 00–99; the *semantic meaning* of the `DAY` component is defined within the logic of function `Z_DATE_SET_DAY`. Given the amount of effort invested in getting the logic for function `Z_DATE_SET_DAY` right, we want to make sure that all updates to the `DAY` field pass through this function so that we can avoid data corruption and code duplication. Moreover, we also need to figure out a way to simplify the function interface so that changes to the internal data representation of the date do not affect programs that are already using this Date library in production.

3.1.3 Moving Toward Objects

The function group `ZDATE_API` shown in Listing 3.1 is an example of an *abstract data type*. An abstract data type defines a set of data along with the operations that can be performed on that data. For the abstraction to be effective, you must keep the data and operations as close to one another as possible. In our case study, this was not the case because programs kept track of the date data locally. This separation of data and behavior limited the usefulness of the abstraction, making library use awkward and error prone. These problems become even more pronounced as the size and complexity of a code library expands.

In many ways, all of the problems that we have encountered in this section can be traced back to one common theme: poor support for data. This is problematic because data is the foundation upon which we build our code. This is the reason procedural programs eventually begin to break apart over time. In the following sections, we will investigate ways that the OOP paradigm can be used to solve these problems.

3.2 Data Abstraction with Classes

Recognizing many of the limitations outlined in Section 3.1, Lessons Learned from the Procedural Approach, software researchers developed the OOP paradigm from the ground up with a strong emphasis on data *and* behavior. As you have already learned, classes encapsulate data (attributes) and behavior (methods) together inside a self-contained package. Encapsulation improves the organization of the code, making object-oriented class libraries much easier to understand and use than their procedural counterparts.

Think about how clumsy our Date library was in Listing 3.1. Each time we called function module `Z_DATE_SET_DAY`, we had to pass it an externally defined structure containing all of the date data it needed to work with. The interfaces for class libraries (i.e., method signatures) are much more elegant because object instances take care of keeping up with the relevant data objects that methods need to do their jobs. In other words, you don't have to provide a lot of instructions to tell an object how to do its job — it simply *knows* how to do it.

Objects created in reference to encapsulated classes take on their own identity, allowing developers to start thinking about their designs in more conceptual

terms. For example, a `Date` object would be responsible for maintaining internal attributes such as `month`, `day`, and `year`. Inside the class definition, we might choose to store those attributes in a structure of type `SCALS_DATE`, or as three distinct integer variables. In either case, this is only a concern for the developer of a class library; clients should be blissfully unaware of these minute details. From a user perspective, the `Date` object is a black box that can be used to work with dates more efficiently.

Using visibility sections (discussed in the next section), you can ensure that operations on a `Date` object (such as changing the day, etc.) can only be *requested* by sending a *message* to the object. The message is realized as a method call on the object. Inside the method implementation, you can apply all of the relevant business rules to ensure that the internal state of the object is not compromised. These business rules give meaning to primitive data types, allowing users of the class library to concentrate on the abstraction as a whole instead of worrying about the nitty-gritty details behind it.

Of course, objects are not magical; it still takes a lot of coding effort to make the abstraction work. The difference is that we have isolated this complexity inside of a concept that is much easier to work with.

3.3 Defining Component Visibilities

The term *encapsulation* refers to the idea of enclosing something inside of a *capsule*. In object-oriented terms, we are enclosing attributes and methods inside an object. The verbal imagery associated with words such as *capsule* implies that we are setting some kind of boundary between the internal components of a class and the outside world. The purpose of this boundary is to protect (or hide) the inner mechanisms of the object that are sensitive to change, and so on.

Most of the time, the most vulnerable parts of an object are its attributes (i.e., the object's state). However, in this book, we will look at ways to hide *any* design decisions that are subject to change. This section describes the ABAP Objects language constructs that you can use to establish boundaries within your classes. In the next section, we will consider how to use these boundaries to build robust classes that can easily be adapted to ever-changing functional requirements.

3.3.1 Visibility Sections

ABAP Objects provides three visibility sections that can be used to control access to the internal components of a class: the `PUBLIC SECTION`, the `PROTECTED SECTION`, and the `PRIVATE SECTION`. Within a class definition, component declarations (e.g., attributes, methods, events, types, etc.) must be assigned to one of these three visibility sections. Listing 3.2 shows a simple class called `lcl_visible` with attributes defined in each of the three visibility sections.

```
CLASS lcl_visible DEFINITION.
  PUBLIC SECTION.
    DATA: x TYPE i.
  PROTECTED SECTION.
    DATA: y TYPE i.
  PRIVATE SECTION.
    DATA: z TYPE i.
ENDCLASS.
```

Listing 3.2 Defining Class Components in Visibility Sections

Components defined within the `PUBLIC SECTION` of a class are accessible from any context in which the class itself is visible (i.e., anywhere you can use the class type to declare an object reference variable). These components make up the *public interface* of the class. The `PRIVATE SECTION` of a class is used to define components that are only accessible from within the class itself. For example, if you were to attempt to compile the code in Listing 3.3, you would receive a compilation error, indicating that access to a private attribute is not allowed. Here, the private attribute `z` can only be accessed inside methods of class `lcl_visible`.

```
DATA: lr_visible TYPE REF TO lcl_visible.
CREATE OBJECT lr_visible.
WRITE: lr_visible->z.
```

Listing 3.3 Attempting Access to Private Components of a Class

The `PROTECTED SECTION` defines components that are only accessible within a class and its subclasses. You will learn more about protected components in Chapter 5, Inheritance.

You can assign components of global classes to visibility sections using the `VISIBILITY` field on the Class Editor screen (see Figure 3.2).

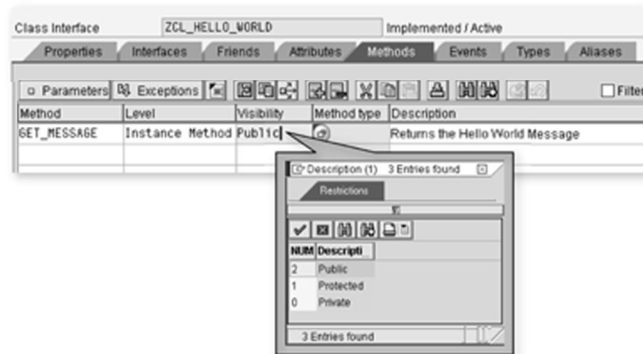


Figure 3.2 Setting the Visibility of Components in Global Classes

When designing the visibility of class components, it is important that you keep the public interface clean and concise. Users of your class should be on a “need-to-know basis.” In other words, if a user doesn’t need direct access to a component, there’s no need for them to even be concerned with its existence. Declaring such components with private visibility makes life easier for everyone because client programmers can concentrate on working with a simplified public interface, and developers can focus on improving the internal implementation of a class without fear of breaking existing client code.

Most of the time, attributes should be defined within the `PRIVATE SECTION` of a class. The primary reason for hiding data attributes is to ensure that the state of the object cannot be tampered with haphazardly. If a client needs to update the state of the object, then he can do so through a method defined in the public interface. The advantage of this kind of indirection is that you can control the assignment of the attribute using business rules that are defined inside the method. This eliminates a lot of the guesswork in troubleshooting data-related errors because you know that any and all changes to an attribute are made through a single method. Methods that update the value of private attributes are sometimes called *setter* or *mutator* methods. The syntax shown in Listing 3.4 demonstrates the general convention used for naming mutator methods.

```
SET_<attribute name>
```

Listing 3.4 Naming Convention for Mutator Methods

If read-only access for an attribute is needed, then you can also provide a *getter* or *accessor* method for that attribute. Accessor methods should be named according to the syntax shown in Listing 3.5.

```
GET_<attribute name>
```

Listing 3.5 Naming Convention for Accessor Methods

ABAP Objects also allows you to define read-only attributes using the `READ-ONLY` addition to the `DATA` keyword. Listing 3.6 shows how to use this addition to create three public, read-only attributes for a class called `lcl_time`. Of course, this option should be used sparingly because it exposes the internal implementation details of your class to the outside world.

```
CLASS lcl_time DEFINITION.
  PUBLIC SECTION.
    DATA: hour    TYPE i READ-ONLY,
          minute  TYPE i READ-ONLY,
          second   TYPE i READ-ONLY.
ENDCLASS.
```

Listing 3.6 Defining Read-Only Attributes in Classes

You can define read-only attributes in global classes by clicking the `READ-ONLY` checkbox for the attribute on the `ATTRIBUTES` tab of the Class Builder (see Figure 3.3).

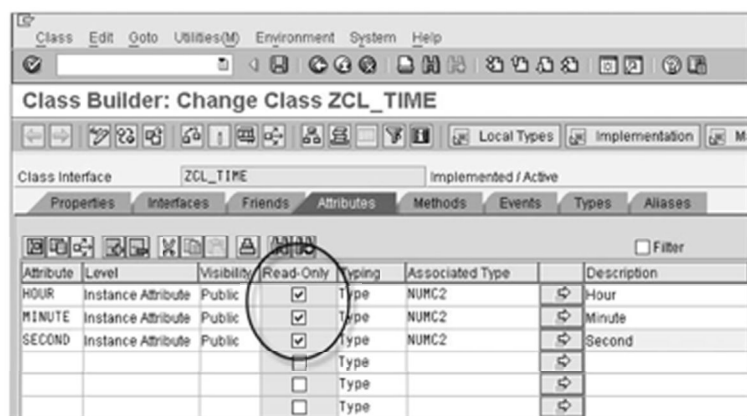


Figure 3.3 Creating Read-Only Attributes in Global Classes

3.3.2 Friends

In the previous section, you learned that components defined within the private and protected sections of a class are not visible outside of that class (or subclasses in the case of protected components). However, in some cases, it is advantageous to grant special access to certain named classes. Such classes are called *friends* of the class that grants them access. You can specify friendship relationships in the class definition using the syntax shown in Listing 3.7.

```
CLASS some_class DEFINITION FRIENDS c1 c2 i3 i4.
```

Listing 3.7 Defining Friendship Relationships in Classes

Within the `FRIENDS` addition, you can specify multiple classes (separated by spaces) as well as interfaces (covered in Chapter 6, Polymorphism). Listing 3.8 shows an example that demonstrates a friendship relationship between two classes called `lcl_parent` and `lcl_child`. Here, class `lcl_child` is declared as a friend of class `lcl_parent`. In method `buy_toys`, class `lcl_child` takes advantage of this friendship relationship by accessing the private attribute `credit_card_no` in class `lcl_parent` to purchase some new toys.

```
CLASS lcl_child DEFINITION DEFERRED.
CLASS lcl_parent DEFINITION FRIENDS lcl_child.
  PRIVATE SECTION.
    DATA: credit_card_no TYPE string.
  ENDClass.
CLASS lcl_child DEFINITION.
  PUBLIC SECTION.
    METHODS buy_toys.
  ENDClass.
CLASS lcl_child IMPLEMENTATION.
  METHOD buy_toys.
    DATA: lr_parent TYPE REF TO lcl_parent.
    CREATE OBJECT lr_parent.
    WRITE: lr_parent->credit_card_no.
  ENDMETHOD.
ENDCLASS.
```

Listing 3.8 Bypassing Access Control Using Friends

There are a couple things to consider when it comes to friendship relationships:

- ▶ First of all, it is important to note the direction and nature of the friendship relationship. In Listing 3.8, class `lcl_parent` explicitly granted friendship access to class `lcl_child`. This relationship definition is not reflexive. For example, class `lcl_parent` cannot access the private components of class `lcl_child` without the `lcl_child` class granting friendship access to `lcl_parent` first.
- ▶ Secondly, notice that classes cannot arbitrarily declare themselves as friends of another class. For instance, class `lcl_child` cannot declare itself a friend of class `lcl_parent`. If this were the case, access control would be a waste of time because any class could bypass this restriction by simply declaring itself a friend of whatever class it is trying to access.

The example shown in Listing 3.8 also introduced a new addition to the `CLASS` statement that we have not discussed before: the `DEFINITION DEFERRED` clause in the `CLASS DEFINITION` statement for class `lcl_child`. This addition is needed to instruct the compiler of the existence of a class `lcl_child` that will be defined later on in the program. Without this clause, the compiler would complain that class `lcl_child` was unknown whenever you tried to establish the friendship relationship in the definition of class `lcl_parent`.

You can define friendship relationships for a global class on the `FRIENDS` tab of the Class Editor (see Figure 3.4). To do so, both of the classes in the friendship relationship must be defined and implemented as global classes in the ABAP Repository. If the `MODELED ONLY` checkbox is selected, the friendship relationship cannot be used at runtime.

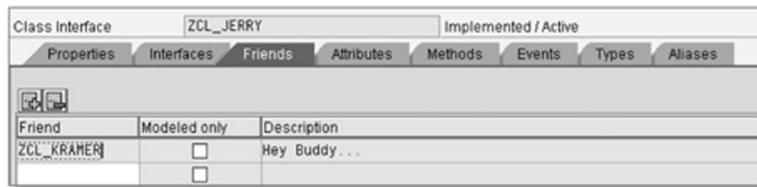


Figure 3.4 Defining Friendship Relationships Between Global Classes

Many purists argue that the use of friends should not be allowed in object-oriented languages because they bypass traditional access control mechanisms. Like

many hotbed topics of this nature, the truth lies somewhere in the middle, and the best approach is to use friendship relationships sparingly in your designs. In particular, you should only develop friendship relationships whenever they are absolutely needed to fulfill a requirement. Most of the time, there are design alternatives that can achieve the same results with more flexibility.

3.4 Hiding the Implementation

Now that you have a nuts-and-bolts understanding of encapsulation techniques, let's try to reimplement the Date library from Listing 3.1 using an object-oriented approach. Here, function group ZDATE_API is replaced by class `lcl_date`, and function `Z_DATE_SET_DAY` is replaced by method `set_day` (see Listing 3.9).

```

CLASS lcl_date DEFINITION.
  PUBLIC SECTION.
    METHODS set_day IMPORTING VALUE(iv_day) TYPE i
              EXCEPTIONS invalid_date.
  PRIVATE SECTION.
    DATA: date TYPE scals_date.
ENDCLASS.

CLASS lcl_date IMPLEMENTATION.
  METHOD set_day.
    DATA: lv_month_end TYPE i.      "Last Day of Month
    CASE date-month.
      WHEN 1.
        lv_month_end = 31.
      WHEN 2.
        ...
    ENDCASE.
    IF iv_day LT 1 OR iv_day GT lv_month_end.
      RAISE invalid_date.
    ELSE.
      date-day = iv_day.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

```

Listing 3.9 Reimplementing the Date Library Using Classes

As you can see in Listing 3.9, the code inside class `lcl_date` is almost identical to that of function module `Z_DATE_SET_DAY` from Listing 3.1. However, one major difference between the two approaches is the fact that method `set_day` works on an internal date data object (i.e., private attribute `date`) rather than an external one provided via its method interface. By encapsulating this data element inside the class, we have eliminated the need for maintaining a local `Date` data structure within the calling program. Now, programs can simply work with object references of type `lcl_date` without having to worry about keeping up with other related data objects (i.e., structures containing the date value). This also simplifies the interface for method `set_day` because we no longer need to pass in a reference to a `Date` data structure for the method to have data to work on.

With our new class-based `Date` library in tow, let's now revisit some of the maintenance scenarios described in Section 3.1.2, Case Study: A Procedural Code Library. Here, if we are asked to enhance the `lcl_date` class to support timestamps, we can either change the type of attribute `date` or add an additional attribute to keep track of the time component of the timestamp. Obviously, this changes the *implementation* of class `lcl_date`, but it does so without disturbing the existing *interface*.

In other words, because we are no longer passing `Date` data structures back and forth, method calls from client programs are not affected by this change. Instead, the public interface for the class can simply expand to incorporate additional methods that can be used to manipulate the time element of the date. We have also addressed data corruption problems by encapsulating the data inside the boundaries of the object. The use of the private visibility section ensures that these boundaries are upheld. By removing the `Date` structure from the method interface and encapsulating it internally as a private attribute, we have effectively *hidden* this implementation concern from the user. Separating the interface of a class from its implementation is a very effective design technique that helps you develop classes that are much more responsive to change.

3.5 Designing by Contract

Encapsulation and implementation hiding techniques can be used to define very precise public interfaces for a class. These interfaces help to form a *contract* between the developer of a class and users of that class. The contract metaphor is

taken from the business world, where customers enter into contractual agreements with suppliers providing goods or services.

In his book *Object-Oriented Software Construction* (Prentice-Hall, 2000), Bertrand Meyer described how this concept could be adapted into object-oriented software designs to improve the reliability of software components that are "... implementations meant to satisfy well-understood specifications." In this context, objects are subject to a series of *invariants* (or constraints) that specify the valid states for the object. To maintain these invariants, methods are defined using *preconditions* (what must be true before the method is executed) and *postconditions* (what must be true after the method is executed). In Chapter 8, Error Handling with Exceptions, we will look at ways to deal with exceptions to these rules.

The primary goal for using the *design-by-contract* approach in your software designs is to produce components that deliver *predictable* results. The boundaries set by the visibility sections ensure that *loopholes* are not introduced into the contract. For instance, we observed that the Date library from Listing 3.1 had many loopholes that made it possible to bypass the business rules implemented inside the function module(s). The encapsulation techniques applied in the class-based implementation of the Date library in Listing 3.9 eliminated these loopholes by encapsulating the data objects for the date as private attributes inside the boundaries of the class.

Client programmers using classes based on these principles know what to expect from the class based on the provided public interface. Similarly, class developers are free to change the underlying implementation as long as they continue to honor the contract outlined in the public interface.

3.6 UML Tutorial: Sequence Diagrams

So far, our study of the UML has been focused on diagrams that are used to describe the static architecture of an object-oriented system. This chapter introduces the first of several *behavioral diagrams* that are used to illustrate the behavior of objects at runtime. The *sequence diagram* depicts a message sequence chart between objects that are interacting inside a software system. Some of the more advanced features of sequence diagrams will be considered in Section 11.5, UML Tutorial: Advanced Sequence Diagrams.

Figure 3.5 shows a simple sequence diagram that is used to illustrate a cash withdrawal transaction in an ATM machine. A sequence diagram is essentially a graph in two dimensions:

- ▶ The various objects involved in the interaction are aligned along the horizontal axis.
- ▶ The vertical axis represents time.

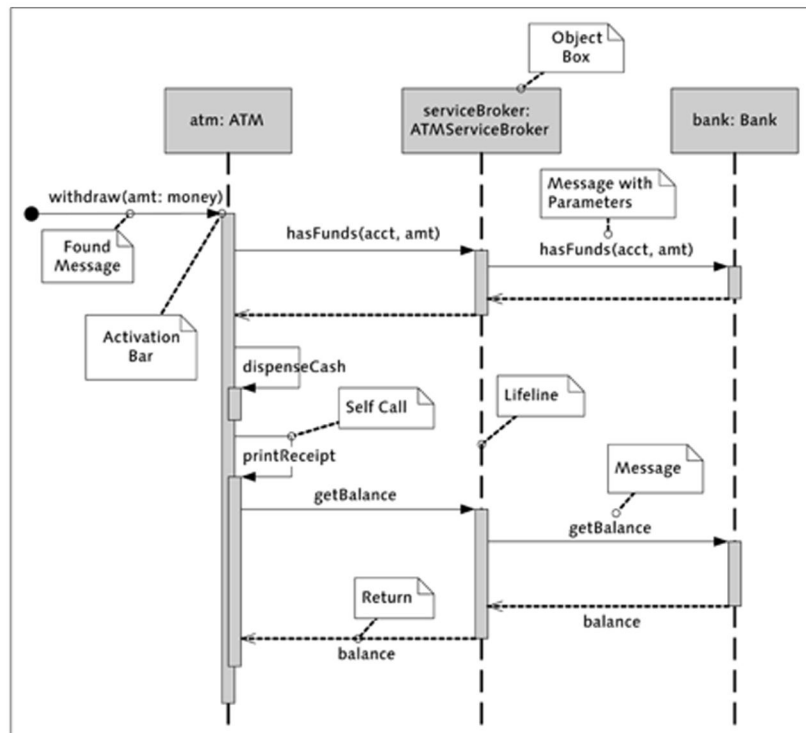


Figure 3.5 Sequence Diagram for Withdrawing Cash from an ATM

Sequence diagrams are initiated by a request message from some kind of external source. In the example in Figure 3.5, the external source is a user interfacing with the ATM. This initial message is called a *found message*. In object-oriented terms, a message is analogous to a method call. Messages are sent to objects (depicted in

the familiar object boxes seen on the object diagrams described in Section 2.6, UML Tutorial: Object Diagrams). The dashed line protruding from underneath the object box represents the object's *lifeline*. In Chapter 11, ABAP Object Services, we will examine situations where objects are created and destroyed within the interaction; for now, we will only consider sequences where all of the objects exist throughout the entire interaction.

The intersection of a message and an object's lifeline is depicted with a thin rectangular box called an *activation bar*. The activation bar shows when an object is active during the interaction. Objects are activated via messages (i.e., method calls). Messages can include parameters that help clarify the operation to be performed by the object. However, it is not a good idea to try and fully specify the method interface in a sequence diagram — that's what a class diagram is for. Here, we only use parameters for emphasis or clarity. Synchronous method calls can have a *return* message that can also have optional parameters.

In some cases, a method might need to call other local helper methods to complete its task. In this case, a *self call* can be illustrated by drawing a circuitous arrow to another activation bar that is stacked on top of the current activation bar. For example, in Figure 3.5, messages `dispenseCash` and `printReceipt` are both represented as self calls on the `atm` object inside method `withdraw`.

Sequence diagrams are very useful for explaining complex interactions where the order of operations is difficult to follow. One of the reasons that sequence diagrams are so popular is that the notation is very intuitive and easy to read. To maintain this readability, it is important to avoid cluttering a sequence diagram with too many interactions. In the coming chapters, we will look at other types of interaction diagrams that can be used to illustrate fine-grained behavior within an object or more involved interactions that span multiple use cases.

3.7 Summary

In this chapter, you learned about the many advantages of applying encapsulation and implementation hiding techniques to your class designs. Encapsulating data and behavior in classes simplifies the way that users work with classes. Hiding the implementation details of these classes strengthens the design even further, making classes much more resistant to data corruption. The combination of these two design techniques helps you to design intelligent classes that are highly self-suffi-

3 | Encapsulation and Implementation Hiding

cient. Such classes are easy to reuse in other contexts because they are loosely coupled to the outside world.

In the next chapter, we will examine the basic lifecycle of an object. Here, we will also learn about special methods called *constructors* that can be used to ensure that object instances are always created in a valid state.

Some of the most elusive bugs to detect in a program can be traced back to missing or invalid variable initializations. This chapter explores how classes can be enhanced to ensure that objects are properly initialized prior to their use in programs.

4 Object Initialization and Cleanup

In the previous chapter, you learned how encapsulation and implementation hiding techniques can be used to protect the integrity of an object. Such objects produce consistent and reliable results, freeing developers from constantly worrying about data correctness issues in their programs. However, all of these measures are wasted if we fail to properly initialize the object in the first place.

In this chapter, we will investigate how special methods called *constructors* can be used to ensure that an object will always be created in a valid state. We will also examine the overall object lifecycle, paying particular attention to how object resources are managed by the automatic memory management functionality built into the ABAP runtime environment.

4.1 Creating Objects

One of the primary goals of the object-oriented design process is to identify ways to delegate responsibilities to objects. This approach transfers complexity from the main program into objects that are intelligent enough to handle the tasks they are assigned. To coordinate these efforts, the main program needs to be able to create and destroy objects on demand. As you can imagine, this dynamic allocation process can get pretty involved for complex object types. Fortunately, the ABAP runtime environment takes care of most of the details, making it relatively painless for developers to create and work with objects in their programs.

Of course, there are costs associated with creating objects dynamically. To recognize how these costs can affect the performance of your program, it is important

to understand what is going on behind the scenes whenever you request the creation of an object using the `CREATE OBJECT` statement.

To put all of this in perspective, let's consider an example of a report program running inside a SAP GUI window that needs to create an object at runtime. However, before we start investigating this particular scenario, we first need to understand how memory is organized and used inside of an SAP NetWeaver Application Server ABAP (AS ABAP) instance.

Figure 4.1 gives a high level overview of the memory architecture of the AS ABAP, showing the local memory used by the individual work processes along with the shared memory that is used throughout the application server. Whenever you log on to the SAP system, it creates a user session (or context) inside the roll buffer area of shared memory. This user session keeps track of administrative items such as your assigned authorizations, and so on. It also keeps track of the program(s) you are running along with the data objects that are being used by those programs.

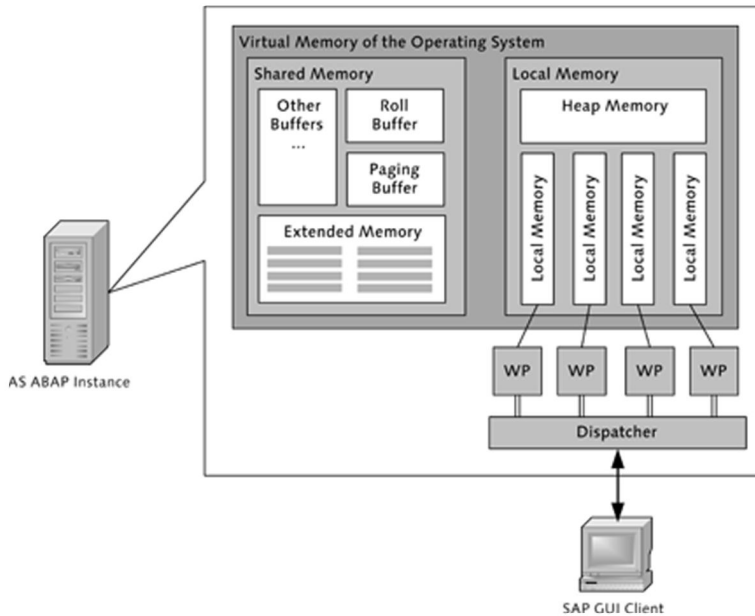


Figure 4.1 Basic Memory Architecture of AS ABAP

Each SAP GUI window that you open allocates a separate memory area inside the user session called a *main session*. Whenever you execute a program in that window, the system creates yet another memory area called an *internal session* inside this main session. The internal session manages the data objects of the program that is running, as well as the data objects of other programs (e.g., function pools, class pools, etc.) that are being used by that program. As you can see in Figure 4.2, it is possible to have multiple internal sessions inside of a main session. Additional internal sessions are created whenever a program calls another program, creating a type of program call stack inside of the main session.

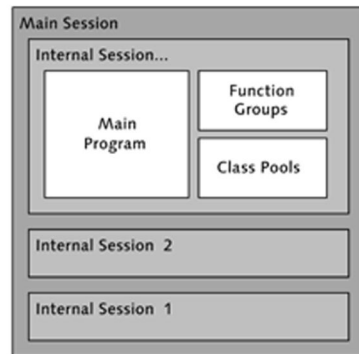


Figure 4.2 Logical Memory Areas in a Main Session

In an effort to optimize system resources, SAP decided to distribute the load of executing programs on an application server by *multiplexing* work processes. The basic idea here is to share a limited amount of work processes across a large number of users that are only interacting with the system part of the time. For example, you might open up an SAP session, execute a report transaction, and then sit there reviewing it for a few minutes while sipping a cup of coffee. Rather than tie up a work process while you are sitting idle, the system will simply take a note of where you are in your program (by storing information in your user session) and then reassign the work process to another user that has made a request.

The process of assigning and unassigning work processes to a user is generally described as a *roll-in* or *roll-out*. During a roll-in, the user session is loaded from the roll buffer into the local memory of a work process (refer to Figure 4.1). Similarly, when a roll-out occurs, information about the state of the currently execut-

ing program is copied back into the roll buffer. One of the ways the system speeds up the roll-in/roll-out process is to keep the user session small by making use of *pointers*. Pointers, as the name suggests, are special lightweight variables that contain a memory address that is used to *point* to data objects that are stored elsewhere. In the case of the AS ABAP, pointers typically point to data objects stored in the extended memory area of the shared memory.

Now that you have a better appreciation for how the various memory areas of an AS ABAP instance are used, let's get back to our report program example. Inside of a processing block in the report program, you would request the creation of an object using the `CREATE OBJECT` statement as shown in Listing 4.1.

```
DATA: oref TYPE REF TO lcl_some_class.
CREATE OBJECT oref.
```

Listing 4.1 Creating an Object Using the `CREATE OBJECT` Statement

At this point, the ABAP runtime environment uses the definition information from class `lcl_some_class` to determine how much memory it needs to allocate for an instance of this class type. After it knows how much memory it needs, the runtime environment then needs to scan through the extended memory area to find a chunk of memory large enough to store the object and its data. It also needs to allocate some additional memory to store a *header* data structure that is used to keep track of various administrative details about the object. After the memory is allocated, the address of the header structure is returned to the object reference variable `oref` (see Figure 4.3).

The primary consequence of this approach for dynamically generating objects is the additional time required to allocate the proper amount of memory for an object. As multiple programs create and destroy objects, the extended memory area can start to become fragmented, making it difficult to locate a contiguous chunk of memory large enough to hold an object. Skeptics sometimes point to these performance costs as a reason for not using objects in their programs, claiming that they can't afford the additional overhead at runtime. However, if you look carefully at your existing programs, you will likely find that you are already using many types of dynamic data objects¹.

¹ For an excellent description of dynamic data objects, check out Horst Keller's blog entitled *ABAP Geek 12 – The Deep* (<https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/2016>).

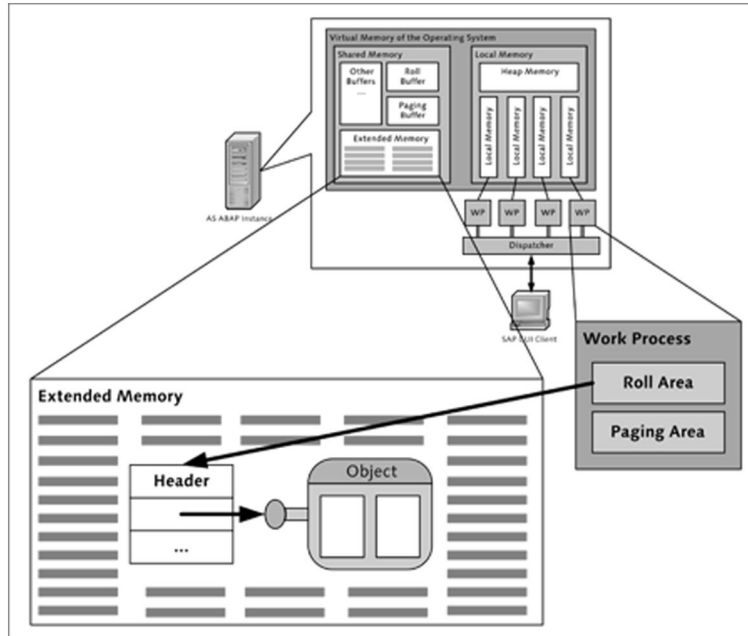


Figure 4.3 Memory Allocation for Objects

For example, internal tables are dynamic data objects that require additional memory to be dynamically allocated as additional rows are appended to them. Most design decisions involve some kind of trade-off, and in the case of objects, you might have to sacrifice a little bit of performance to realize the many benefits of Object-Oriented Programming (OOP). Fortunately, SAP has optimized the performance of the ABAP runtime environment so that these performance issues are rarely a concern. We will investigate some basic guidelines for tuning performance in Section 4.5, Tuning Performance.

4.2 Controlling Object Initialization with Constructors

Encapsulated objects rely on data stored in hidden attributes to keep track of their internal state so that they can respond to method requests intelligently. There-

fore, it is vitally important to come up with a way to reliably supply the object's attributes with data. Otherwise, you end up back in a procedural mode where each method has to receive data via parameters or derive it from an external data source (e.g., a database lookup, etc.). In either of these cases, the method becomes more complex because you have to validate and perhaps derive data before you can do any useful work.

Obviously, you want to avoid this kind of situation. However, to do so, you must figure out a way to *guarantee* that the attributes of a class are properly initialized before *any* calls are made to methods that depend on these attributes. Of course, we could try to be disciplined in our approach and make sure that we call all of the appropriate setter methods before using the object, but then we have to remember to do it every time an object is instantiated. In the best case, a lot of redundant code is introduced. In the worst case, forgetting to call a method here and there creates an even bigger problem by making it more difficult to figure out where things went wrong. Clearly, we need a better method for initializing objects.

Normally, whenever you declare a variable in an ABAP program, you don't have to worry about *creating* it at runtime because this is handled for you automatically by the ABAP runtime environment. For example, if you declare a global variable in a program using a structure data type, you can immediately begin assigning values to the components of that structure without having to issue a `CREATE` statement, and so on. As you have seen, this is not the case for anonymous data objects such as object instances. Here, we must use the `CREATE OBJECT` statement to *request* that an object be created dynamically by the ABAP runtime environment. Prior to that request, we cannot use object reference variables in our program because they do not point to a valid object instance.

One of the benefits of this kind of indirection is that it lets the ABAP runtime environment take complete control of the creation process. Class developers are allowed to interact with this process by creating a special method called a *constructor*. Constructors are called automatically by the ABAP runtime environment *after* the object has been created but *before* control is returned to the program. Inside the constructor, you can initialize all of the attributes in your class to ensure that the object is created in a valid state.

Constructors are defined using a similar syntax to the one that you have used to define normal methods in a class (see Listing 4.2). The only restriction here is that

you can only define importing parameters in the method signature. If you think about it, this makes sense because the constructor is called by the ABAP runtime environment and not by the normal `CALL METHOD` statement. In fact, it is not possible to call constructors directly in your programs.

```
METHODS constructor
  IMPORTING [VALUE()]i1 i2 ...[]
  TYPE type [OPTIONAL]...
  EXCEPTIONS ex1 ex2.
```

Listing 4.2 Syntax for Defining an Instance Constructor

Constructors can be created in global classes by clicking the **CONSTRUCTOR** button in the Class Editor (see Figure 4.4).

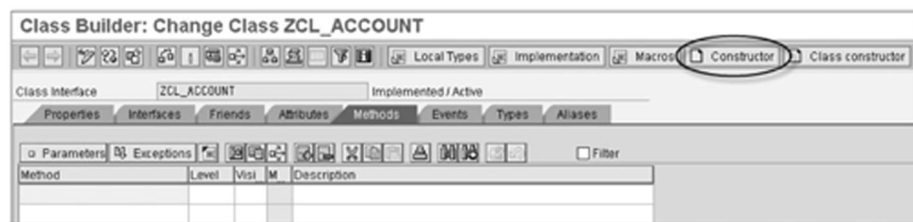


Figure 4.4 Creating Constructors for Global Classes

To demonstrate the usefulness of constructors, let's consider an example based on an `Account` class that is used to represent various types of accounts at a bank. In Listing 4.3, a class called `lcl_account` has been created that provides a few basic methods to view the current balance of the account as well as to withdraw or deposit funds. The account details are stored in private attributes that do not have associated setter methods. Instead, these attributes are initialized via a database lookup inside the `constructor` method. Whenever a user instantiates an object of type `lcl_account`, the user must provide a valid account number that is used to lookup the relevant account details. In Chapter 8, Error Handling with Exceptions, you will see how *exceptions* can be used to enforce this rule; for now, you can simply assume that this is the case.

```
CLASS lcl_account DEFINITION.
  TYPE-POOLS: abap.
  PUBLIC SECTION.
  METHODS:
```

4 | Object Initialization and Cleanup

```
        constructor IMPORTING im_account_no
                       TYPE string,
get_balance RETURNING VALUE(re_balance)
                       TYPE bapicurr_d,
deposit     IMPORTING im_amount
                       TYPE bapicurr_d,
withdrawal  IMPORTING im_amount
                       TYPE bapicurr_d
                       RETURNING VALUE(re_result)
                       TYPE abap_bool.
PRIVATE SECTION.
    DATA: account_no TYPE string,
           balance    TYPE bapicurr_d.
ENDCLASS.

CLASS lcl_account IMPLEMENTATION.
    METHOD constructor.
    *   Query database tables to retrieve account details:
    *   SELECT FROM ...
    *   WHERE account_no = im_account_no.
    ENDMETHOD.

    METHOD get_balance.
        re_balance = balance.
    ENDMETHOD.

    METHOD deposit.
        balance = balance + im_amount.
    ENDMETHOD.

    METHOD withdrawal.
        IF im_amount LE balance.
            balance = balance - im_amount.
            re_result = abap_true.
        ELSE.
            *Exception handling code...
        ENDIF.
    ENDMETHOD.
ENDCLASS.
```

Listing 4.3 Initializing an Account Class Using a Constructor

As you look at the code in Listing 4.3, notice how the addition of method `constructor` has effectively put a lock on the front door of class `lcl_account`. Clients requesting objects in reference to this class must provide a valid account number, so we don't have to worry about which account we are working on inside the various methods in the public interface.

For example, notice how methods `deposit` and `withdraw` did not contain any logic to determine which account to post the transactions against. Here, this is not needed because we have guaranteed proper initialization of the object and restricted access to the sensitive attributes of the class. The object lifecycle begins by pulling up the latest account details from the database and can only be further influenced through public methods containing business logic to ensure that users do not compromise the integrity of the object in any way.

Most of the time, whenever we talk about constructors, we are typically talking about *instance constructors* that are used to initialize an instance of an object that is being created. However, it is also possible to supply a *class constructor* for a class. Class constructors provide a mechanism for initializing the class (or static) attributes of a class. A class constructor is called implicitly by the system before any accesses are made to the class inside your program. Class constructors are defined using the syntax shown in Listing 4.4.

```
CLASS-METHODS class_constructor.
```

Listing 4.4 Syntax for Defining a Class Constructor

As you can see in Listing 4.4, you cannot specify any parameters for a class constructor because it is being called implicitly by the system. It is also worth mentioning here that you cannot access instance components within the class constructor because no instances of the class exist when it is being called. Of course, it is possible to instantiate an object of the class at this point. Here, you simply access the instance components of the object through a local object reference variable just as you would in any normal method implementation.

Class constructors can be created for global classes by clicking the **CLASS CONSTRUCTOR** button on the Class Editor screen (see Figure 4.5). The report program `ZCOUNTER_DEMO` shown in Listing 4.5 demonstrates how class constructors can be used to initialize class attributes.

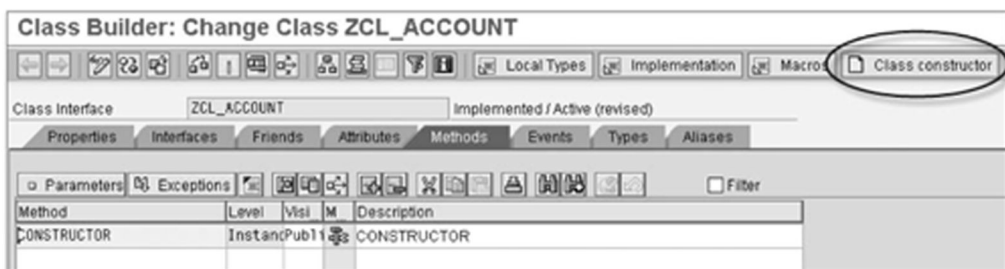


Figure 4.5 Creating Class Constructors for Global Classes

```
REPORT zcounter_demo.

CLASS lcl_counter DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS: class_constructor.
    METHODS: increment,
               get_count RETURNING VALUE(re_count)
                  TYPE i.
  PRIVATE SECTION.
    CLASS-DATA: count TYPE i.
ENDCLASS.

CLASS lcl_counter IMPLEMENTATION.
  METHOD class_constructor.
    count = 10.
  ENDMETHOD.

  METHOD increment.
    count = count + 5.
  ENDMETHOD.

  METHOD get_count.
    re_count = count.
  ENDMETHOD.
ENDCLASS.

DATA: lr_counter1 TYPE REF TO lcl_counter,
      lr_counter2 TYPE REF TO lcl_counter,
      lv_count     TYPE i.
```

```

START-OF-SELECTION.
  CREATE OBJECT lr_counter1.
  lv_count = lr_counter1->get_count( ).
  WRITE: / lv_count.

  DO 10 TIMES.
    lr_counter1->increment( ).
  ENDDO.

  lv_count = lr_counter1->get_count( ).
  WRITE: / lv_count.

  CREATE OBJECT lr_counter2.
  lr_counter2->increment( ).
  lv_count = lr_counter2->get_count( ).
  WRITE: / lv_count.

```

Listing 4.5 An Example Program Using a Class Constructor

If you run the `ZCOUNTER_DEMO` report, you will see that the class constructor has taken care of pre-initializing the `count` class attribute to the value 10 prior to the creation of the `lr_counter1` object. After the `lr_counter1` object is initialized, the instance method `increment` is called 10 times, incrementing the value of the counter by 5 each time. After this loop is completed, the value of the `count` attribute will be 60. Next, another object named `lr_counter2` is created. Here, the class constructor is not called again, so the `count` attribute is not affected. Consequently, whenever the call is made to `increment` for the `lr_counter2` object, `count` is incremented to 65.

As you can see in the example, class constructors provide a handy way to initialize class attributes prior to the creation of object instances. Often, class constructors are used to initialize common resources that are used inside instance constructors.

4.3 Taking Control of the Instantiation Process

Until now, it has been possible to create instances of the classes we have developed anywhere that the class itself is visible using the `CREATE OBJECT` statement. This is the default behavior for local and global classes. However, in certain cases, it is useful to take control of the instantiation of objects within the class itself.

This kind of behavior can be specified using the `CREATE` addition of the `CLASS DEFINITION` statement. This syntax is depicted in Listing 4.6.

```
CLASS lcl_some_class DEFINITION
    CREATE (PUBLIC | PROTECTED | PRIVATE).
    ...
ENDCLASS.
```

Listing 4.6 Specifying the Instantiation Context of a Class

You can set the instantiation context for global classes on the `PROPERTIES` tab of the Class Editor (see Figure 4.6).

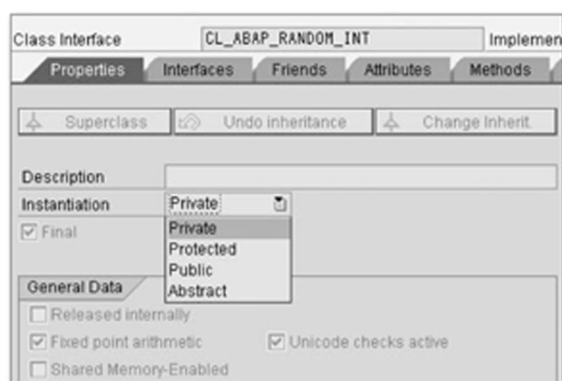


Figure 4.6 Setting the Instantiation Context for Global Classes

Table 4.1 describes each of the possible instantiation contexts that can be defined for classes.

Instantiation Context	Visibility
PUBLIC	These classes can be instantiated anywhere that the class itself is visible without restrictions.
PROTECTED	These classes can only be instantiated inside methods of the class itself and its subclasses.
PRIVATE	These classes can only be instantiated inside methods of the class itself.

Table 4.1 Instantiation Contexts for Classes

Perhaps the best way to illustrate the utility of controlled instantiation is to look at how it might be used in a practical example. Let's imagine that you want to build a class library that can be used to process XML documents. To maximize the usefulness of your `XMLDocument` class, you want to be able to load XML documents from a number of different types of data sources. For instance, you might want to build the XML document using a file, a byte stream, a tree-like data structure, and so on.

In many object-oriented languages, this problem can be solved by *overloading* the `constructor` method to support different method signatures. Here, each overloaded constructor method has the same name but a different set of parameters. Unfortunately, this is not a feature supported in ABAP Objects. One common workaround for this limitation is to create multiple optional parameters in the constructor's method signature and try to figure out which scenario you are dealing with inside the constructor's implementation using conditional logic. Listing 4.7 shows an `IF` statement that uses the `IS SUPPLIED` option to determine if parameter `im_param1` was supplied to the method during the method call.

```
IF im_param1 IS SUPPLIED.
  ...
ENDIF.
```

Listing 4.7 Determining if Parameters Are Passed in a Method Call

The problem with this technique is that the signature of the `constructor` method becomes quite large and difficult to work with. The logic in the constructor code also becomes obscured by all of the various input permutations.

As you have seen, our typical approach for dealing with complexity is to figure out a way to encapsulate (or hide) it. In this case, we want to encapsulate the instantiation process to make it more straightforward and intuitive. One way to encapsulate this process is to configure the XML document class to have a protected/private instantiation context. This implies that users will no longer have the ability to directly instantiate XML document objects. Instead, they must work with public *creational* class methods that are customized to build XML documents using various types of inputs (see Listing 4.8). These methods behave like a constructor, taking control of the initialization process. This is important because the primary goal here is to simplify the initialization process so that a single constructor is not responsible for supporting all of the various initialization variants that we want to provide.

4 | Object Initialization and Cleanup

```
CLASS lcl_xml_document DEFINITION
    CREATE PRIVATE.
    PUBLIC SECTION.
        CLASS-METHODS:
            create_from_scratch RETURNING VALUE(re_xml_doc)
                                TYPE REF TO lcl_xml_document.
            create_from_file    IMPORTING im_filename
                                TYPE string
                                RETURNING VALUE(re_xml_doc)
                                TYPE REF TO lcl_xml_document.
            create_from_stream IMPORTING im_stream
                                TYPE xstring
                                RETURNING VALUE(re_xml_doc)
                                TYPE REF TO lcl_xml_document.
            *Other Creational Methods...
            *Utility Methods...
            as_string           RETURNING value(re_string)
                                TYPE string.
    PRIVATE SECTION.
        METHODS: constructor.
ENDCLASS.

CLASS lcl_xml_document IMPLEMENTATION.
    METHOD constructor.
    *   Default initialization code goes here...
    ENDMETHOD.

    METHOD create_from_scratch.
    *   Use the basic constructor logic to create
    *   an empty XML document:
    CREATE OBJECT re_xml_doc.
    ENDMETHOD.

    METHOD create_from_file.
    *   Use the private constructor to build a
    *   basic XML document:
    CREATE OBJECT re_xml_doc.

    *   Read the given file and load the XML
    *   document using utility/setter methods:
    *   OPEN DATASET im_filename...
    ENDMETHOD.
```

```

METHOD create_from_stream.
*   Use the private constructor to build a
*   basic XML document:
  CREATE OBJECT re_xml_doc.

*   Load the byte stream into the XML document:
*   ...
ENDMETHOD.

METHOD as_string.
*   re_string = ...
ENDMETHOD.
ENDCLASS.

```

Listing 4.8 Controlling Instantiation Through Creational Methods

Notice that each of the creational methods shown in Listing 4.8 use the `CREATE OBJECT` statement to create a base XML document object. As stated before, the private instantiation context only restricts external users of the class from accessing the constructor; internal methods are still free to use it. In the case of class `lcl_xml_document` the basic initialization logic can still reside inside the private constructor. The creational methods simply build on this base object to initialize the XML document using the desired data source.

4.4 Garbage Collection

After you have finished using an object in your program, you need to make sure that you restore its resources to the system. In some languages, it is the programmer's responsibility to make sure that objects are properly destroyed. Fortunately, this is not the case with ABAP Objects because object resources are automatically cleaned up by a special memory management feature of the ABAP runtime environment called the *garbage collector*. The garbage collector's job is to scan through memory and delete objects that no longer have any references associated with them (i.e., "orphans").

Much of the time, these references are destroyed automatically whenever an object reference variable goes out of scope (i.e., when a subroutine or method terminates). However, if you are done with an object, it is a good idea to explicitly remove the reference using the `CLEAR oref` statement (see Figure 4.7).

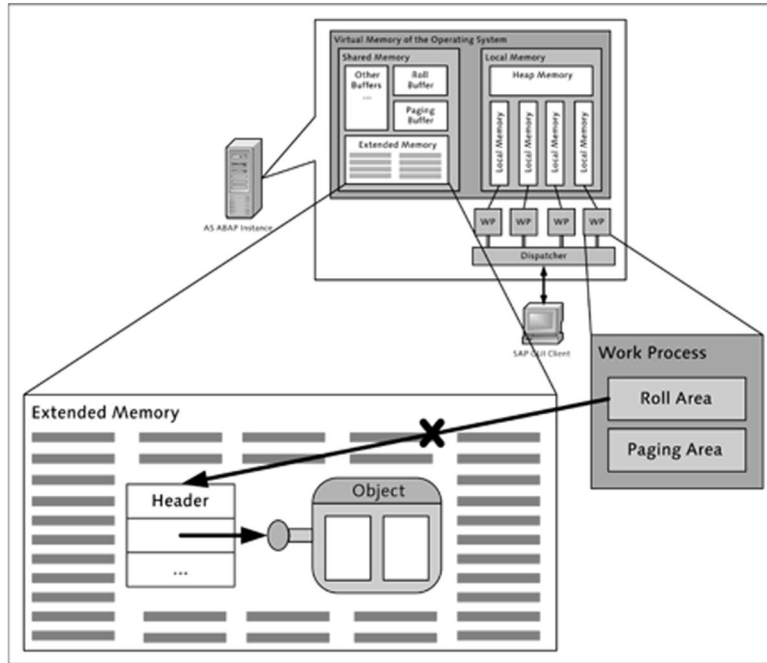


Figure 4.7 Removing References to Objects Using the CLEAR Statement

Some programming languages allow you to create a special lifecycle method called a *destructor* that gets called before an object is destroyed. This method can be used to manage any internal resources for the object that need to be released gracefully (e.g., open file handles, etc.). Regrettably, this method is not used in ABAP Objects, so it is important to remember to clean up any of these kinds of internal object resources via an explicit method call before allowing the object to be destroyed.

4.5 Tuning Performance

The advanced memory management features of the ABAP runtime environment provide a safe environment for creating and destroying objects. However, it is

important to remember that these features will not prevent you from making poor design decisions that consume excessive amounts of memory. In this section, we will provide some basic tips that you can use to avoid these performance traps.

4.5.1 Design Considerations

Even if you don't anticipate performance problems for a given class, it is always a good idea to modularize the initialization logic of the class so that you can implement performance tuning measures later without disturbing core functionality, and so on. The following list contains some basic modularization tips that you should consider when developing your classes:

- ▶ Keep the logic inside the `constructor` method to a minimum by delegating initialization tasks to modularized private *helper* methods.
- ▶ Provide yourself a public `reset` method that can be used to clear the values of a class's attributes.
- ▶ Try to avoid adding too many parameters to the `constructor` method's interface. Instead, encapsulate the initialization process inside of a series of creational methods as shown in Section 4.3, Taking Control of the Instantiation Process.

4.5.2 Lazy Initialization

Sometimes, you may have large composite objects that contain lower-level details that are not frequently used. For example, let's imagine that you are tasked with designing a purchasing system. One of the main classes for this system is a `PurchaseOrder` class that contains an internal table attribute of `PurchaseOrderItem` objects that also in turn contain a table of `ScheduleLine` objects. Based on the functional requirements, you observe that this class is primarily used to query header level information such as the PO status, partner details, and so on.

In a situation such as this, it can be advantageous to delay the initialization of the lower-level details until they are needed. Here, once again, you are protected by encapsulation because you can control access to these lower-level attributes through getter methods that initialize the attributes on demand whenever they are first requested. This technique is referred to as *lazy initialization*. The basic

idea here is to avoid having to create (and ultimately garbage collect) objects that may never be used.

For example, in the `PurchaseOrder` class analogy, you might choose to implement the constructor method in such a way that purchase order header-level details are only loaded when an object is created. If a user wants to access the line item details of the purchase order, then he may do so at any time using an instance method (e.g., `get_items()`). At this point, the line item data is brought into context on demand. Clearly, the first call to this instance method incurs a bit of a performance hit. Still, keep in mind that this task would have to be carried out anyway if you chose to initialize everything in the constructor. In the case of lazy initialization, however, you can avoid this processing overhead altogether if the user never accesses the lower-level details. This improves the performance of the constructor and also keeps the size of the process order objects in check.

4.5.3 Reusing Objects

The easiest way to avoid the performance hits associated with creating/destroying objects is to simply avoid this process altogether by recycling objects. Some typical candidates for recycling include temporary objects created inside loops, and objects created in utility methods. You may discover that you could have created an object in a higher scope that could be reused in the loop or method calls, you might be working with a lightweight object in a loop that simply needs to be reinitialized based on the loop index, and so on. Rather than create a new object each time, you might be able to call a `reset` method to reuse the object.

4.5.4 Using Class Attributes

As you design your classes, you should think about whether or not each object instance will require its own local copy of an attribute. If a local copy of an object is not required for an object instance, defining the attribute as a class attribute can help avoid the creation of a lot of redundant objects. Here, as you will recall, the ABAP runtime environment will only create a single copy of the data object represented by a class attribute that is shared across all object instances. Therefore, the potential reduction in memory usage can be exponential when you are creating many objects at runtime.

4.6 UML Tutorial: State Machine Diagrams

The sequence diagrams introduced in the previous chapter are good for showing the behavior of multiple objects interacting in a particular use case. Another type of behavioral diagram in the UML is the *state machine diagram*, which are useful for showing the behavior of a single object throughout its lifetime.

Figure 4.8 shows a state machine diagram for a class that could be used to represent a batch job that is created using the SAP Job Scheduler tool. Whenever a new job object is created, it is initialized in the *Scheduled* status. This is depicted on the diagram by an *Initial Pseudostate* node that points to the *Scheduled* state box. Each of the possible statuses of a job are shown using rounded boxes called *states*. Changes in state are represented with directed *transition* arrows. Transitions can optionally be labeled with special transition label strings using the syntax shown in Listing 4.9.

```
event(s) [guard conditions]/activity
```

Listing 4.9 Syntax Diagram for Defining Transition Labels

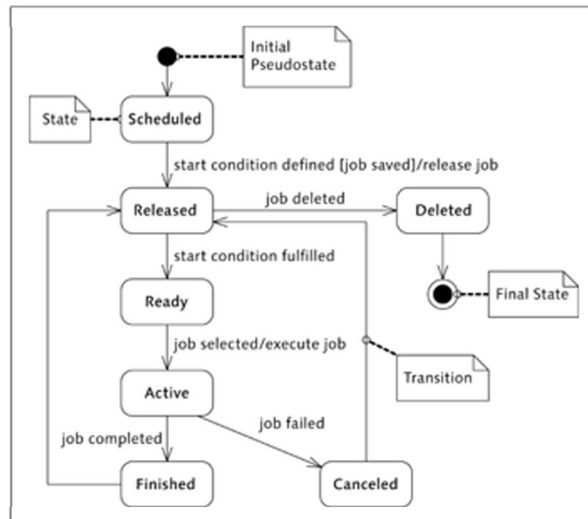


Figure 4.8 UML State Diagram for an SAP Batch Job

The `event(s)` portion of the transition label is used to describe the event (or events) that would trigger a change in state for the object. If `guard conditions` are included in the transition label, then those conditions must be true for the transition to occur. The `activity` option can be used to specify some behavior that takes place during the transition.

As an example, let's consider the transition between the `Scheduled` and `Released` statuses in Figure 4.8. In this case, the triggering event occurs whenever a user defines a start condition for the job in Transaction SM36. However, for the job to be released in the system, it must first be saved.

If a job is deleted in the system, then the state machine (i.e., the object) will reach its *final state*. In Figure 4.8 this is shown via the arrow that points to the circular node with a dot in it.

Like many of the diagrams that you will see throughout this book, the state machine diagram fulfills a distinct purpose. In this chapter, we investigated some of the various ways that objects are created in the system. Most of the time, the lifecycle of an object is pretty straightforward. However, for objects with complex lifecycles, state machine diagrams can be very useful in showing how an object will interact with the environment around it.

4.7 Summary

In this chapter, you learned how to use constructors to ensure that objects are always properly initialized before they are used in a program. Combining the use of constructors with the encapsulation techniques described in Chapter 3, Encapsulation and Implementation Hiding, helps you to build reliable and robust class libraries.

In the next chapter, we will investigate ways for reusing these class libraries in other contexts.

As your understanding of a problem domain matures, so also does your comprehension of the relationships and responsibilities of the classes that are being used to model software systems based on that domain. This chapter begins our discussion of inheritance, which is a key object-oriented concept that can be used to expand and refine your object model to evolve with ever-changing functional requirements.

5 Inheritance

In Chapter 3, Encapsulation and Implementation Hiding, we examined how you might try to construct a reusable code library using procedural design techniques. During our investigation, we observed some of the problems that can make it difficult to reuse these libraries in other environments. In the past, whenever developers encountered these kinds of challenges, they typically either tried to enhance/rework the library to accommodate the new requirements, or they cut their losses and salvaged as much of the code as possible by copying and pasting it into new development objects. Unfortunately, neither one of these approaches works very well in practice:

- ▶ Modifying the code library to handle new requirements threatens the integrity of pre-existing programs using the library because it is possible that errors could be introduced into the system along with the changes.
- ▶ The copy-and-paste approach is less risky initially but ultimately increases the cost of long-term maintenance efforts because redundant code makes the overall code footprint bigger, often requiring enhancements/bug fixes to be applied in multiple places that can be difficult to locate.

The reusability predicaments described here are not unique to procedural programming. In fact, just because a class has been well encapsulated does not mean that it is immune to the types of errors that could be introduced whenever changes are made to the code. However, there are measures that you can take in your object-oriented designs to avoid these pitfalls.

In this chapter, we will examine how the concept of *inheritance* can be used to make copies of a class without disturbing the source class or introducing redundant code. You will also learn about another technique called *composition* that provides a way to reuse classes in situations where inheritance doesn't make sense.

5.1 Generalization and Specialization

One of the most difficult parts of the object-oriented design process is trying to identify the classes that you will need to model a domain, what the relationships between those classes should be, and how objects of those classes will interact with one another at runtime. Even the most knowledgeable object-oriented developers rarely get it all right the first time. Often developers new to Object-Oriented Programming (OOP) are troubled by this, fearing the long-term consequences of early design mistakes. Fortunately, the use of good encapsulation and implementation hiding techniques should minimize the "ripple effects" normally associated with changing modularized code.

Nevertheless, certain changes force us to look at the problem domain in a whole new way. Here, for instance, you may discover that your original design was not sophisticated enough to handle specialized cases. Frequently, during gap analysis, you may realize that you have either failed to identify certain classes in the domain or that you have defined particular classes too generically.

For example, let's say you take a first pass through a set of requirements for a human resources system. During this analysis process, you discover a need for an `Employee` class, among others. However, during the implementation cycle of the project, more requirements come out that describe specific functionalities relevant for certain types of employees. At this point, you could try and enhance the original `Employee` class to deal with these added features, but this seems counter-intuitive because it clutters the class with too many responsibilities. On the other hand, abandoning the `Employee` class altogether in favor of a series of specialized classes (e.g., `HourlyEmployee`, etc.) leads to the kind of code redundancy issues that you want to avoid. Fortunately, object-oriented languages such as ABAP Objects provide a better and more natural way for dealing with these kinds of problems.

The concept of inheritance can be used to *extend* a class so that you can reuse what is already developed (and hopefully tested) to expand the class metaphor to better fit specialized cases. The newly created class is called a *subclass* of the original class; the original class is called the *superclass* of the newly created class. As the name suggests, subclasses *inherit* components from their superclass. These relationships allow you to build a hierarchical inheritance tree with superclasses as parent nodes and subclasses as child nodes (see Figure 5.1). In Chapter 6, Polymorphism, you will see how members of this inheritance tree can be used interchangeably, providing for some interesting generic programming options.

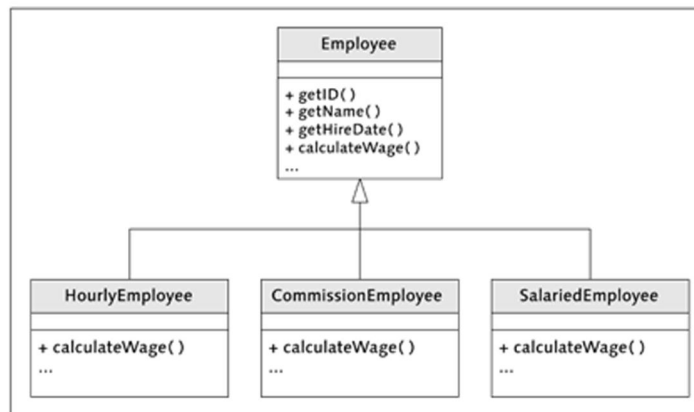


Figure 5.1 Inheritance Hierarchy for Employees

The root of every inheritance tree is the predefined empty class `OBJECT`; thus every class that we have created so far has implicitly inherited from this class. To demonstrate how to establish explicit inheritance assignments, let's consider the example code shown in Listing 5.1.

```

REPORT zemployee_test.

CLASS lcl_employee DEFINITION.
  PUBLIC SECTION.
    DATA: id TYPE numc10 READ-ONLY. *Demo Purposes Only!!
ENDCLASS.
  
```

5 | Inheritance

```
CLASS lcl_hourly_employee DEFINITION
    INHERITING FROM lcl_employee.
    PUBLIC SECTION.
    METHODS:
        constructor IMPORTING im_id TYPE numc10
                        im_wage TYPE bapicurr_d.

        calculate_wage.
    PRIVATE SECTION.
    CONSTANTS: CO_WORKWEEK TYPE i VALUE 40.
    DATA: hourly_wage TYPE bapicurr_d.
ENDCLASS.

CLASS lcl_hourly_employee IMPLEMENTATION.
    METHOD constructor.
    * Must call the constructor of the superclass first:
      CALL METHOD super->constructor( ).

    * Initialize the instance attributes:
      id = im_id.
      hourly_wage = im_wage.
    ENDMETHOD. "constructor

    METHOD calculate_wage.
    * Method-Local Data Declarations:
      DATA: lv_wages TYPE bapicurr_d. "Calculated Wages

    * Calculate the weekly wages for the employee:
      lv_wages = CO_WORKWEEK * hourly_wage.

      WRITE: / 'Employee #', id.
      WRITE: / 'Weekly Wage:', lv_wages.
    ENDMETHOD. "calculate_wage
ENDCLASS.

START-OF-SELECTION.
* Create an instance of class lcl_salaried_employee
* and call method "calculate_wage":
DATA: gr_employee TYPE REF
      TO lcl_hourly_employee.

CREATE OBJECT gr_employee
```

```

EXPORTING
  im_id = '1'
  im_wage = '10.00'.

CALL METHOD gr_employee->calculate_wage( ).

```

Listing 5.1 Example Report Showing Inheritance Syntax

The report program `ZEMPLOYEE_TEST` in Listing 5.1 contains two simple classes: `lcl_employee` and `lcl_hourly_employee`. In this example, class `lcl_hourly_employee` is a subclass of class `lcl_employee` and therefore inherits its public `id` attribute. Note that the `id` attribute is only defined in the `PUBLIC SECTION` of class `lcl_employee` for the purposes of this demonstrative example. You will learn about a better alternative for providing access to sensitive superclass components in Section 5.2.1, *Designing the Inheritance Interface*.

The inheritance relationship is specified using the `INHERITING FROM` addition to the `CLASS DEFINITION` statement that was used to define class `lcl_hourly_employee`. Inside class `lcl_hourly_employee`, several references are made to the `id` attribute from the `lcl_employee` superclass. Here, notice that we didn't have to do anything special to access this component in the subclass because it has been automatically inherited from the superclass.

You can define inheritance relationships in global classes by clicking on the `CREATE INHERITANCE` button on the `CREATE CLASS` dialog box (see Figure 5.2). This adds an additional `SUPERCLASS` input field that can be used to enter the superclass (see Figure 5.3).

You can also maintain the inheritance relationship on the `PROPERTIES` tab of the Class Editor (see Figure 5.4). Here, you can remove the relationship or define a new superclass as well. Inheritance is more than just a fancy way of copying classes into new classes. Inheritance defines a natural relationship that will likely expand over time.

To appreciate the nature of this relationship, let's consider a situation where you are asked to start keeping track of addresses for employees. Furthermore, let's imagine that you have extended the class hierarchy from Listing 5.1 to include various other subclass types. In this case, you need to maintain addresses for all employees. You could add an `address` attribute to each of the subclasses, but that would be redundant because every type of employee should have an address. The logical place to create the `address` attribute is in the superclass `lcl_employee`.



Figure 5.2 Defining Inheritance for Global Classes – Part I



Figure 5.3 Defining Inheritance for Global Classes – Part II

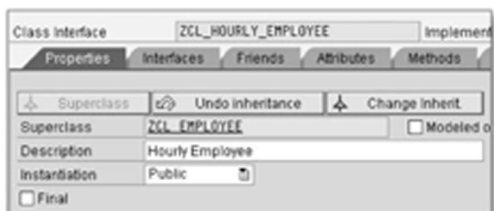


Figure 5.4 Editing Inheritance Relationships for Global Classes

Making the change in the `lcl_employee` superclass ensures that the `address` attribute is automatically inherited by each of the subclasses because of the inheritance relationship that they maintain with the superclass. However, if you make changes in the subclasses (e.g., `lcl_hourly_employee`), these changes *are not* reflected in the superclass. This functionality allows you to expand your code libraries with subclasses that do not jeopardize the integrity of their superclasses and the production code that is depending on them.

5.2 Inheriting Components

So far, our discussions on the subject of component visibility have focused on designing a class's public and private interface from an external user point-of-view. However, inheritance adds a new dimension into the mix because we now also need to consider how to define the interface between a superclass and its subclasses. Sometimes, you might want to provide access to a component in subclasses without having to expose the component in the public interface.

For example, in Listing 5.1 shown earlier, if the `id` attribute of class `lcl_employee` had been placed in the `PRIVATE SECTION` of the class, that attribute could not be addressed inside the `lcl_hourly_employee` subclass. Therefore, `id` was exposed as a public, read-only attribute. Of course, good implementation hiding techniques would call for exposing access to that private attribute through a getter method, but you get the idea.

5.2.1 Designing the Inheritance Interface

To address this middle ground, ABAP Objects provides another alternative by allowing you to define components within the `PROTECTED SECTION` of a class definition.

The components defined within the `PROTECTED SECTION` of a class make up the interface between a superclass and its subclasses. Subclasses can access components defined in the `PROTECTED SECTION` of a superclass in the same way that they would access components defined in the `PUBLIC SECTION` of that superclass. To the outside world however, components defined in the `PROTECTED SECTION` of a class behave just like components defined in the `PRIVATE SECTION` of the class. Listing 5.2 redefines the `lcl_employee` class from Listing 5.1 to use the `PROTECTED SECTION` visibility area.


```

CLASS lcl_employee DEFINITION.
  PROTECTED SECTION.
    DATA: id          TYPE numc10,
           hire_date  TYPE sydatum.
ENDCLASS.

CLASS lcl_hourly_employee DEFINITION
  INHERITING FROM lcl_employee.
  PUBLIC SECTION.
  METHODS:
    constructor IMPORTING im_id          TYPE numc10
                im_hire_date TYPE sydatum,
    display.
ENDCLASS.

CLASS lcl_hourly_employee IMPLEMENTATION.
  METHOD constructor.
*   Must call the constructor of the superclass first:
    CALL METHOD super->constructor( ).

*   Initialize the instance attributes:
*   Notice that we can access these attributes directly:

    id = im_id.
    hire_date = im_hire_date.
  ENDMETHOD.          "constructor

  METHOD display.
    WRITE: / 'Employee #', id,
           'was hired on', hire_date.
  ENDMETHOD.          "display
ENDCLASS.

```

Listing 5.2 Defining and Accessing Protected Components

As you start to design your inheritance interfaces, it is important not to get carried away with defining components in the `PROTECTED SECTION` of the class. Sometimes, we tend to think of subclasses as having special privileges that should allow them full access to a superclass. Here, it is essential that you employ the encapsulation concept of *least privilege* when designing your subclasses.

The concept of least privilege implies that if a subclass doesn't really need to access a component, then it shouldn't be granted access to that component. For

example, imagine that you have defined some superclass that contains certain components that you want to change in some way. If these components are defined in the protected visibility section of the superclass, it is quite possible that these changes cannot be carried out without affecting all of the subclasses that may be using these components. The general rule of thumb here is to always define attributes in the private visibility section. If a subclass needs to be granted access to these components, then the access should be provided in the form of getter/setter methods that are defined in the `PROTECTED SECTION` of the class. This little bit of additional work ensures that a superclass is fully encapsulated.

5.2.2 Visibility of Instance Components in Subclasses

Subclasses inherit the instance components of *all* of the superclasses defined in their inheritance tree. However, not all of these components are *visible* at the subclass level. A useful way of understanding how these visibility rules work is to imagine that you have a special instance attribute pointing to an instance of the superclass inside of your subclass. You can use this reference attribute to access public components of the superclass, but access to private components is restricted just as it would be for any normal object reference variable.

As it turns out, this imaginary object reference metaphor is not too far off from what is actually implemented in subclasses behind the scenes. Subclasses contain a special *pseudo reference* variable called `super` that contains a reference to an instance of an object of the superclass's type. This reference is used to access components of a superclass inside a subclass. The primary difference between the `super` pseudo reference variable and a normal reference variable is that the `super` pseudo reference can also be used to access components defined in the `PROTECTED SECTION` of the superclass it points to.

The use of the `super` pseudo reference variable is optional (as was the case with the `me` self-reference variable discussed in Chapter 2, *Working with Objects*) but can be used in situations where explicit reference to superclass components is needed. Normally, you will simply access the components of the superclass directly, but it is important to remember that the compiler is implicitly plugging in the `super` pseudo reference behind the scenes to properly address these components. If you operate in this mindset, the visibility rules for accessing superclass components should be pretty intuitive.

Public and protected components of classes in an inheritance tree all belong to the same internal namespace. This implies that you cannot create a component in a subclass using the same name that was used to define a component in a superclass. There is no such restriction on the naming of private components, however. For example, if you define a private component called `comp` in a superclass, you can reuse this same name to define components in subclasses without restriction.

5.2.3 Visibility of Class Components in Subclasses

Subclasses also inherit all of the class components of their superclasses. Of course, as was the case with instance components, only those components that are defined in the public or protected visibility sections of a superclass are actually visible at the subclass level. However, in terms of inheritance, class attributes are not associated with a single class but rather to the overall inheritance tree. The change in scope makes it possible to address these class components by binding the class component selector operator with any of the classes in the inheritance tree.

This can be confusing because class components are defined in terms of a given class and probably don't have a lot of meaning outside of their defining class's context. To avoid this kind of confusion, it is recommended that you always address class components by applying the class component selector to the defining class's name (e.g., `lcl_superclass=>component`). That way, your intentions are always clear.

5.2.4 Redefining Methods

Frequently, the implementation of an inherited method needs to be changed at the subclass level to support more specialized functionality. You can redefine a method's implementation by using the `REDEFINITION` addition to the method definition in your subclass.

The code example in Listing 5.3 shows how class `lcl_hourly_employee` is redefining the default (dummy) implementation of method `calculate_wage` from the `lcl_employee` superclass. In Section 5.3.1, Abstract Classes and Methods, we will demonstrate a better approach for defining methods like `calculate_wage` at a generic superclass level.

```

CLASS lcl_employee DEFINITION.
  PROTECTED SECTION.
  METHODS:
    calculate_wage RETURNING VALUE(re_wage)
                  TYPE bapicurr_d.
ENDCLASS.

CLASS lcl_employee IMPLEMENTATION.
  METHOD calculate_wage.
*   Empty for now...
  ENDMETHOD.
ENDCLASS.

CLASS lcl_hourly_employee DEFINITION
  INHERITING FROM lcl_employee.
  PUBLIC SECTION.
  METHODS:
    calculate_wage REDEFINITION.
ENDCLASS.

CLASS lcl_hourly_employee IMPLEMENTATION.
  METHOD calculate_wage.
*   re_wage = hours worked * hourly rate...
  ENDMETHOD.
ENDCLASS.

```

Listing 5.3 Redefining Methods in Subclasses

To redefine a method in a global class, place your cursor in the **METHOD** column for the method that you want to redefine, and click on the **REDEFINE** button (see Figure 5.5).

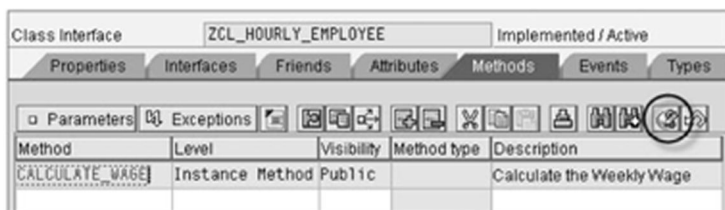


Figure 5.5 Redefining Methods in Global Classes

Whenever you redefine a method, you are only allowed to redefine its implementation — the method interface (or signature) must remain the same. The redefinition obscures the superclass's implementation in the subclass, which means that whenever a call is made to the method for an object of the subclass type, the redefined implementation will be used instead. Sometimes, the redefinition simply needs to add on to what was previously implemented in the superclass. In this case, you can use the `super` pseudo reference to invoke the superclass's method implementation so that you don't have to reinvent the wheel.

5.2.5 Instance Constructors

Unlike other normal instance components, constructors *are not* inherited. If you think about it, this makes sense because each class only knows how to initialize objects of its own type. To ensure that the instance attributes of its superclasses are also properly initialized, a subclass is required to explicitly call the constructor of its superclass *before* it starts to initialize its own instance attributes. This is achieved using the syntax shown in Listing 5.4. Here, the use of parameters is optional depending upon whether or not the constructor of the immediate superclass requires them.

```
CALL METHOD super->constructor
  [EXPORTING
    im_param1 = value1
    im_param2 = value2
    ...].
```

Listing 5.4 Syntax for Calling the Constructor of a Superclass

Whenever you instantiate a subclass using the `CREATE OBJECT` statement, the ABAP runtime environment will take care of recursively walking up the inheritance tree to make sure that the constructor of each superclass is called. At each level in the inheritance hierarchy, a superclass's constructor will only have visibility to its own components and those defined in its superclasses. This implies that a method call inside the superclass constructor will be bound to the implementation defined for that superclass and not a redefined version defined at the subclass level.

This complex sequence of events is best demonstrated using an example. In Listing 5.5, the subclass `lc1_child` redefines the `message` method that was inherited

from class `lcl_parent`. As you can see, the `message` method is called in the constructors for both classes. However, if you instantiate an object of type `lcl_child`, you will see that the constructor of the `lcl_parent` class called its own implementation rather than the redefined version in class `lcl_child`.

```

CLASS lcl_parent DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor,
             message.
ENDCLASS.

CLASS lcl_parent IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD me->message.
  ENDMETHOD.           "constructor

  METHOD message.
    WRITE: / 'In parent...'.
  ENDMETHOD.           "message
ENDCLASS.

CLASS lcl_child DEFINITION
  INHERITING FROM lcl_parent.
  PUBLIC SECTION.
    METHODS: constructor,
             message REDEFINITION.
ENDCLASS.

CLASS lcl_child IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD super->constructor.
    CALL METHOD me->message.
  ENDMETHOD.           "constructor

  METHOD message.
    WRITE: / 'In child...'.
  ENDMETHOD.           "message
ENDCLASS.

```

Listing 5.5 Example Showing Constructor Call Sequence and Scope

5.2.6 Class Constructors

Each subclass is also allowed to define its own unique class constructor. This constructor gets called right before the class is addressed in a program for the first time. However, before it is executed, the ABAP runtime environment walks up the inheritance tree to make sure that the class constructor has been called for each superclass in the inheritance hierarchy. These class constructor calls occur in the proper order.

For example, let's imagine that you have a class hierarchy with four classes A, B, C, and D. When a program tries to access class D for the first time, the runtime environment will first check to see if the class constructors have been called for classes A, B, and C. If the class constructor has already been called for class A, but not for B and C, then the order of class constructor calls will be B, C, and D. This ensures that the class attributes of a superclass are always properly initialized before a subclass is loaded.

5.3 The Abstract and Final Keywords

Occasionally, you may identify an occurrence where you need to define a class whose functionality cannot be fully implemented on its own. Such classes must be completed by subclasses that fill in the gaps.

5.3.1 Abstract Classes and Methods

A crude way of dealing with these gaps is to create "dummy" methods to completely define the class. However, this can be dangerous because often these methods really don't make much sense in the context of a generic superclass. In these situations, it is better to define an *abstract class* that explicitly delegates unknown features to subclasses. Because their implementation is incomplete, abstract classes cannot be instantiated on their own. Their purpose is to provide a common *template* that makes it easier to implement specialized subclasses.

To understand how all this works, let's revisit the *Employee* example initially shown in Listing 5.1. There, method `calculate_wage` was not created at the superclass level but rather at the subclass level (i.e., in class `lcl_hourly_employee`). However, if you think about it, this method is really applicable for all types of employees. Of course, at the generic superclass level (i.e., in class

`lcl_employee`), we do not know how to calculate the wages of an employee. Nevertheless, as you will see in Chapter 6, Polymorphism, it is advantageous to define this behavior at the appropriate level in the inheritance hierarchy.

The code in Listing 5.6 shows how the class hierarchy has been reworked (or *refactored*) by defining `lcl_employee` as an abstract class. Method `calculate_wage` has also been defined as an abstract method inside class `lcl_employee`. These changes force any subclasses of `lcl_employee` to either provide an implementation for method `calculate_wage` or be defined as abstract (thus further propagating the functionality down the inheritance hierarchy). In this case, method `calculate_wage` has been fully implemented in class `lcl_hourly_employee`.

```

CLASS lcl_employee DEFINITION ABSTRACT.
  PUBLIC SECTION.
    METHODS:
      constructor IMPORTING im_id TYPE numc10,
      calculate_wage abstract.
  PROTECTED SECTION.
    DATA: id TYPE numc10.
ENDCLASS.

CLASS lcl_employee IMPLEMENTATION.
  METHOD constructor.
    id = im_id.
  ENDMETHOD.
ENDCLASS.

CLASS lcl_hourly_employee DEFINITION
  INHERITING FROM lcl_employee.
  PUBLIC SECTION.
    METHODS:
      constructor IMPORTING im_id TYPE numc10
      im_wage TYPE bapicurr_d,
      calculate_wage REDEFINITION.

  PRIVATE SECTION.
    CONSTANTS: CO_WORKWEEK TYPE i VALUE 40.
    DATA: hourly_wage TYPE bapicurr_d.
ENDCLASS.

CLASS lcl_hourly_employee IMPLEMENTATION.
  METHOD constructor.

```



```

* Must call the constructor of the superclass first:
CALL METHOD super->constructor( im_id ).

* Initialize the instance attributes:
hourly_wage = im_wage.
ENDMETHOD. "constructor

METHOD calculate_wage.
* Local Data Declarations:
DATA: lv_wages TYPE bapicurr_d. "Calculated Wages

* Calculate the weekly wages for the employee:
lv_wages = CO_WORKWEEK * hourly_wage.

WRITE: / 'Employee #', id.
WRITE: / 'Weekly Wage:', lv_wages.
ENDMETHOD. "calculate_wage
ENDCLASS.

```

Listing 5.6 Defining Abstract Classes and Methods

You can create abstract global classes by setting the instantiation type to **ABSTRACT** on the **PROPERTIES** tab of the Class Editor (see Figure 5.6).

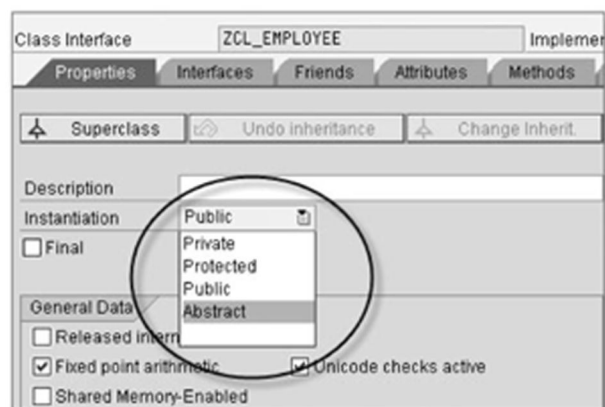


Figure 5.6 Creating Abstract Global Classes

To create abstract methods for global classes in the Class Editor, place your cursor in the **METHOD** column, and click on the **DETAIL VIEW** button. This opens up a dialog box that allows you to modify various attributes for the method. In this case,

click the **ABSTRACT** checkbox. This opens a prompt advising you that the method implementation was deleted (see Figure 5.7).

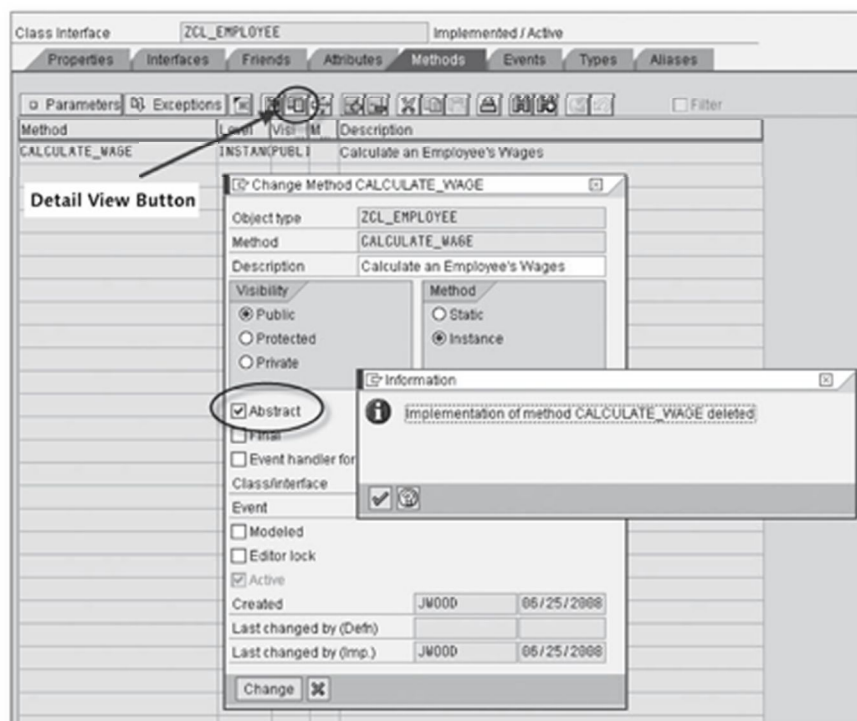


Figure 5.7 Defining Abstract Methods for Global Classes

5.3.2 Final Classes

Sometimes, you may refine a class hierarchy to the point where it no longer makes sense to extend it. At this stage, it is best to announce this situation formally by marking the class's inheritance tree as complete using the `FINAL` modifier. Final classes cannot be extended in any way, effectively concluding a branch of an inheritance tree. The syntax for creating final classes is shown in Listing 5.7.

```
CLASS lcl_ender DEFINITION FINAL.
    ...
ENDCLASS.
```

Listing 5.7 Syntax for Defining Final Classes

Global classes are marked as final by selecting the **FINAL** checkbox on the **PROPERTIES** tab in the Class Editor (see Figure 5.8).

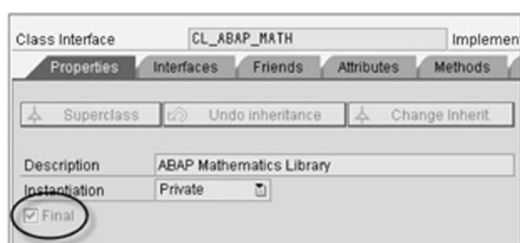


Figure 5.8 Marking Global Classes as Final

You should exercise extreme caution when deciding to mark a class as final. Although you may think that you have reached the end of the class hierarchy, it is hard to know for certain. The bottom line is that if you have any doubts about what you are doing, it's probably best that you don't do it.

5.3.3 Final Methods

A less risky approach to finalizing a class is to mark individual methods as final. This way, you still leave yourself the option for extending the class without allowing users of that class to redefine specific methods that you believe to be complete. The syntax for defining final methods is shown in Listing 5.8.

```
CLASS lcl_ender DEFINITION.
  PUBLIC SECTION.
    METHODS: complete FINAL.
ENDCLASS.
```

Listing 5.8 Defining Final Methods

You can mark the final indicator for methods in global classes in the same method **DETAIL VIEW** screen shown in Figure 5.7. Here, you simply click the **FINAL** checkbox to mark the method as complete (see Figure 5.9).



Figure 5.9 Defining Final Methods for Global Classes

5.4 Inheritance Versus Composition

Developers sometimes get confused by the hype surrounding inheritance, assuming that they must make extensive use of it in their designs to be *true* object-oriented programmers. Note that while inheritance is powerful, it is not always the best solution for reusing code from existing classes. In fact, one of the worst mistakes you can make is to try to *stretch* classes to fit into some sort of loosely formed inheritance relationship.

Whenever you are thinking of defining a new class in terms of some pre-existing class, you should ask yourself whether or not the relationship between the subclass and superclass fits into the is-a relationship mold. To illustrate this, let's consider an inheritance tree for various types of order objects (see Figure 5.10). At each level of the tree, you should be able to apply the is-a relationship between a subclass and its superclass, and it should make sense. For example, a *SalesOrder* is an *Order*, and so on.

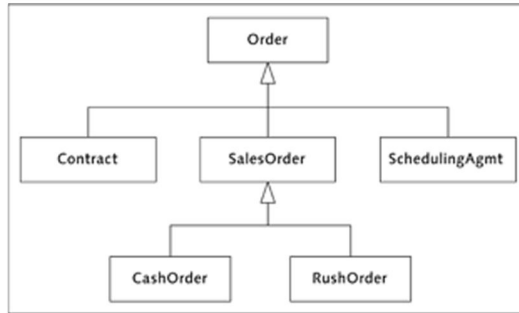


Figure 5.10 Inheritance Tree for Order Types

Most of the time, the application of the is-a test should make inheritance relationships between classes pretty obvious. For example, if we try to extend the Order class in Figure 5.10 to create a Delivery subclass, the is-a relationship would not make sense (i.e., a *Delivery is not an Order*).

Although this observation should be clear to even the novice developer, it is not uncommon to encounter situations where developers have tried to create inheritance relationships like this in an effort to leverage classes that have useful features or similarities to the ones they are trying to implement. Whenever you find yourself stuck trying to figure out ways to define an inheritance relationship between two classes, it is a good idea to take a step back and think about the relationship between the classes from a logical perspective. If you think about it, a *Delivery is not an Order*, but an *Order does have one or more Deliveries associated with it*. This has-a association is commonly referred to as a *composition* relationship.

The term *composition* basically describes the reuse of existing functionality in classes by integrating objects of those classes as attributes in your new class. You can use these attributes in the same way that you have used ordinary attributes based on elementary types, structures, and so on. Listing 5.9 shows how you could define a composition relationship between an Order object and a Delivery object.

```

CLASS lcl_delivery DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor.
  
```

```

        get_delivery_date RETURNING value(re_date)
                           TYPE sydatum.

PRIVATE SECTION.
    DATA: delivery_date TYPE sydatum.
ENDCLASS.

CLASS lcl_delivery IMPLEMENTATION.
    METHOD constructor.
        delivery_date = sydatum.
    ENDMETHOD.

    METHOD get_delivery_date.
        re_date = delivery_date.
    ENDMETHOD.
ENDCLASS.

CLASS lcl_order DEFINITION.
    PUBLIC SECTION.
        METHODS: constructor IMPORTING im_id TYPE i,
                 release,
                 track.
    PRIVATE SECTION.
        DATA: id          TYPE i,
               delivery    TYPE REF
               TO lcl_delivery.
ENDCLASS.

CLASS lcl_order IMPLEMENTATION.
    METHOD constructor.
        id = im_id.
    ENDMETHOD.
    "constructor

    METHOD release.
*   Arbitrarily create a delivery for the order...
        CREATE OBJECT delivery.
    ENDMETHOD.
    "release

    METHOD track.
*   Local Data Declarations:
        DATA: lv_delivery_date TYPE sydatum.

```

```

    lv_delivery_date = delivery->get_delivery_date( ).
    WRITE: / 'Order #', id, 'was shipped on',
           lv_delivery_date.
    ENDMETHOD.           "track
ENDCLASS.

```

Listing 5.9 Reusing Classes with Composition

You should favor the use of composition over inheritance unless the inheritance relationships between classes are obvious. In Chapter 6, *Inheritance*, you will see how inheritance can bring along some unwanted baggage that can lead to inflexible designs if you are not careful.

5.5 Using the Refactoring Assistant

Inheritance provides a natural way for extending classes to adapt to changing functional requirements. However, sometimes you may not discover inheritance relationships until later on in the software development lifecycle. At that point, it is likely that you have not defined classes at the right level of granularity.

For instance, let's revisit the `Employee` class hierarchy example that we have considered throughout this chapter. In this case, let's imagine that the initial set of requirements only described functionality related to employees paid by the hour. Based on the information available at the time, you might decide that you simply need to create a single class called `HourlyEmployee`. At a later stage in the project, you are confronted with new requirements that are related to salaried employees, and so on. At this point, you realize that you probably need to incorporate a more generic `Employee` class at the root of the inheritance tree. Such changes will certainly affect the internal structure of class `HourlyEmployee` (although hopefully the use of encapsulation techniques will make these changes transparent to the outside world). In any event, any time you make changes such as this, you run the risk of introducing errors into the system. However, if you ignore these architectural observations, the effectiveness of your design will ultimately deteriorate over time.

In his famous book *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), Martin Fowler describes a process called *refactoring* that can be used to incorporate these kinds of structural changes into a design. The basic idea here is to improve the underlying structure of a system without affecting its external behavior. The *Refactoring* book describes a series of refactorings (or patterns)

that can guide you toward making good design decisions whenever you alter the structure of your classes. In many cases, these refactorings must be performed manually and therefore require careful attention to detail to ensure that the changes are propagated consistently throughout the system.

Fortunately, SAP has provided a useful tool inside the Class Builder to assist you with your refactoring efforts for global classes. The *Refactoring Assistant* tool can be used to automatically perform some of the most common refactorings. This automation helps to ensure that you don't accidentally make a mistake by omitting some of the manual steps involved with moving components between classes, and so on.

To demonstrate the functionality of the Refactoring Assistant tool, let's try to perform a *Move Method* refactoring to move method `CALCULATE_WAGE` from class `ZCL_HOURLY_EMPLOYEE` to a newly derived superclass called `ZCL_EMPLOYEE`.

1. To start the Refactoring Assistant tool, select **UTILITIES • REFACTORING ASSISTANT** from the top menu bar of the Class Editor screen (see Figure 5.11).

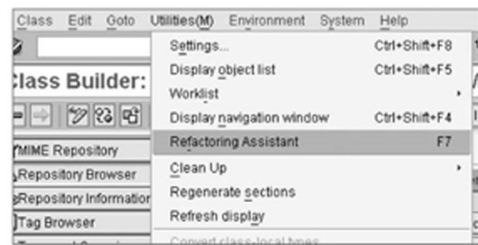


Figure 5.11 Opening the Refactoring Assistant Tool

2. Inside the Refactoring Assistant tool, you are presented with a tree control containing the subclass (`ZCL_HOURLY_EMPLOYEE`), its components (e.g., `CALCULATE_WAGE`), and its superclass (`ZCL_EMPLOYEE`) as shown in Figure 5.12.
3. To move the `CALCULATE_WAGE` method up to the base class level, select and drag the method name up onto the `ZCL_EMPLOYEE` node. Click on the **SAVE** button in the Refactoring Assistant toolbar to save these changes. At this point, both classes need to be activated for the changes to be fully committed.

SAP has plans to expand the functionality of the Refactoring Assistant in future releases, promising tighter integration with the new ABAP Editor. These features

will make the process of refactoring even more reliable and efficient, helping to ease the concerns of management types who fail to see the value in “fixing something that isn't broken.”

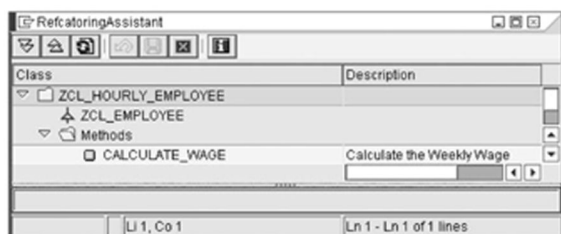


Figure 5.12 The Refactoring Assistant Editor

5.6 UML Tutorial: Advanced Class Diagrams Part I

Section 1.6, UML Tutorial: Class Diagram Basics, introduced some of the basic elements of a class diagram, showing you how to define rudimentary classes along with their attributes and behaviors. In this chapter and the next one, we will expand our discussion of class diagrams to incorporate some of the more advanced concepts that have been described in the past few chapters.

5.6.1 Generalization

Most of the time, our discussions on inheritance tend to focus on specializations at the subclass level. However, if you look up the inheritance tree, you see that superclasses become more generalized as you make your way to the top of the tree. Perhaps this is why the creators of the UML decided to describe the notation used to depict inheritance relationships between classes in a class diagram as a *generalization* relationship.

Figure 5.13 shows a basic class diagram that depicts a superclass called `Account` along with two subclasses (`CheckingAccount` and `SavingsAccount`). Notice that each subclass has a connector drawn upward toward their superclass. The triangle at the top of the association identifies the relationship between the two classes as a generalization.

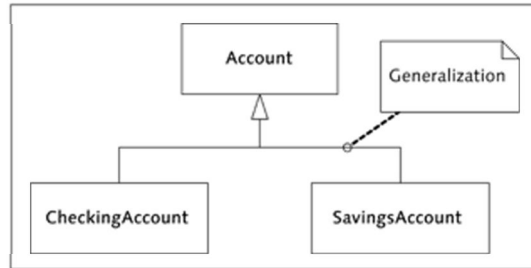


Figure 5.13 UML Class Diagram Notation for Generalizations

5.6.2 Dependencies and Composition

In Section 5.4, Inheritance versus Composition, we described the concept of inheritance in terms of a has-a relationship between two classes. In Chapter 1, Introduction to Object-Oriented Programming, we looked at how associations could be used to depict a composition relationship between classes. However, an association depicts a fairly loose relationship between two classes. Sometimes, you will want to define a composition relationship in more detail.

For example, often a composing class is highly dependent on a supplier class. In that case, it is a good idea to depict this tight coupling by creating a *dependency* relationship between the composing class and the supplier class. Figure 5.14 shows the dependency relationship between an *Order* class and a *Delivery* class as described in Section 5.4, Inheritance versus Composition.

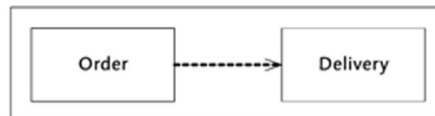


Figure 5.14 Defining a Dependency Relationship Between Classes

The UML also provides a specific notation for depicting composition relationships. In Figure 5.15, this notation is used to show that an instance of class *Address* can be embedded inside either class *Customer* or class *Vendor*, but not both. This notation also implies that any instances of class *Address* will be deleted whenever the instance of the composing *Customer* or *Vendor* class is deleted.



Figure 5.15 Defining Composition Relationships in Class Diagrams

As you can see in Figure 5.15, the filled-in diamond in the association line between the two classes in a composition relationship is always affixed to the composing class. The direction and cardinality of the association lines further describes the nature of the composition relationship. For example, in Figure 5.15, classes *Customer* and *Vendor* can reference zero or more instances of class *Address*.

Looking back at Section 5.4, Inheritance versus Composition, you can see that the UML interpretation for composition relationships is much more specific than the more common view of composition used in normal development scenarios. Consequently, you should be careful to only use the UML composition notation whenever you intend for composed objects to be completely managed by their composing objects.

5.6.3 Abstract Classes and Methods

Figure 5.16 shows the UML notation for depicting abstract classes and methods. The only requirement here is to italicize the class or method name to indicate that the class or method is to be defined as abstract.

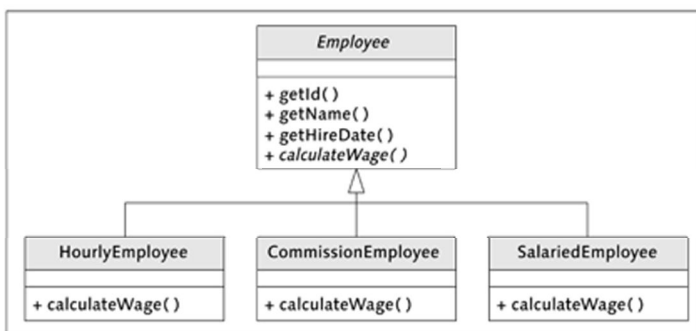


Figure 5.16 Defining Abstract Classes and Methods

Because italics are sometimes hard to read, you will often see developers tag abstract classes using the `<< abstract >>` keyword (see Figure 5.17).

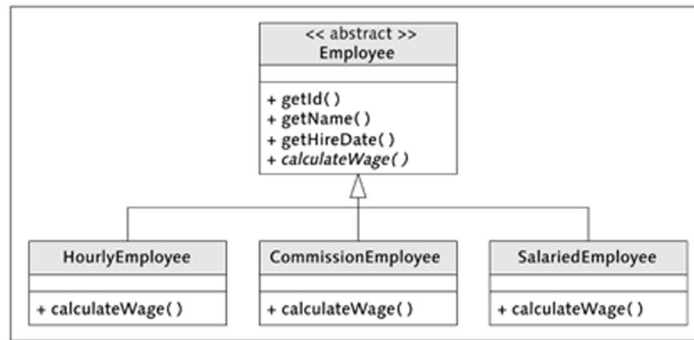


Figure 5.17 Non-Normative Form for Defining Abstract Classes

5.7 Summary

In this chapter, you have learned how inheritance and composition techniques can be used to quickly and safely reuse the implementations of existing classes. In this chapter, we concentrated our focus on inheriting a class's implementation.

However, there is another dimension of the inheritance relationship that we have not yet considered. In the next chapter, you will see how type inheritance can be used to further exploit inheritance relationships to make your designs more flexible.

The term polymorphism literally means “many forms.” From an object-oriented perspective, polymorphism works in conjunction with inheritance to make it possible for various types within an inheritance tree to be used interchangeably. In this chapter, you will learn how to implement flexible designs in your ABAP Objects programs using polymorphism.

6 Polymorphism

In the previous chapter, you learned how to create inheritance relationships between related classes. If you recall, the basic litmus test for identifying these relationships is to ask whether or not a subclass is a *type of* a superclass. For example, a *Dog is a type of Mammal*, and therefore shares common characteristics and behaviors with other mammals. Subclasses can take advantage of this commonality by reusing the implementation of their superclasses. However, as it turns out, there is another important dimension to an is-a relationship that we have not yet considered.

Classes in an inheritance tree share a common public interface, making it possible for a given subclass to respond to any request (i.e., method call) that could be submitted to its superclass. This aspect of an inheritance relationship is referred to as *interface inheritance*. The ability for a subclass to redefine the implementation of its inherited methods adds an interesting twist to this functionality, allowing subclasses to respond to requests in a specific way. In this chapter, we will investigate how these features can be exploited to develop highly flexible designs.

6.1 Object Reference Assignments Revisited

In Chapter 2, Working with Objects, we learned how to use the `MOVE` statement and the assignment operator (`=`) to assign the contents of one object reference variable to another. As you may recall, object reference assignments copy the pointer stored in the source reference variable into the contents of the target reference variable. After an assignment takes place, the target reference variable will

point to the same object pointed to by the source reference variable. Of course, this kind of assignment only makes sense if the types of the two reference variables are *compatible*.

Strictly speaking, two variables are compatible if they share the same type. In spite of this, we frequently make assignments between variables having incompatible types (e.g., between built-in types such as an integer and a floating-point number, etc.). Such types are said to be *convertible* in the sense that there exists some kind of conversion rule that tells the ABAP runtime environment how to convert the contents of the source variable into a format compatible with the target variable.

However, conversions don't make sense for object reference assignments because an object reference stores a pointer and not the object itself. Instead, an object reference variable must be *enhanced* with additional type information that provides visibility to the components of the actual object that it points to. In this section, you will learn how to perform object reference assignments between families of related types. Understanding how these assignments work is a prerequisite for learning how to implement generic designs using polymorphism.

6.1.1 Static and Dynamic Types

So far, we have only performed assignments between object reference variables that have shared the same *static type*. The static type of an object reference variable is the class type (or interface type, as you will see in Section 6.3, Interfaces) used to define the object reference variable. For example, in Listing 6.1, the static type of object reference variable `lr_oref` is the class type `lcl_class`.

```
DATA: lr_oref TYPE REF TO lcl_class.
```

Listing 6.1 Determining the Static Type of an Object Reference

Sometimes, you may want to perform assignments between object reference variables that do not share the same static type. For example, because instances of a superclass and its subclasses are interchangeable, it should be possible to perform an assignment between object reference variables that have these different static types. The code snippet in Listing 6.2 shows an example of this kind of assignment with the `lr_parent = lr_child` statement.

```
CLASS lcl_parent DEFINITION.  
  PUBLIC SECTION.  
    METHODS: a.  
             b.  
ENDCLASS.  
  
CLASS lcl_parent IMPLEMENTATION.  
  METHOD a.  
    WRITE: / 'In method a.'.  
  ENDMETHOD.  
  
  METHOD b.  
    WRITE: / 'In method b.'.  
  ENDMETHOD.  
ENDCLASS.  
  
CLASS lcl_child DEFINITION  
  INHERITING FROM lcl_parent.  
  PUBLIC SECTION.  
    METHODS: c.  
ENDCLASS.  
  
CLASS lcl_child IMPLEMENTATION.  
  METHOD c.  
    WRITE: / 'In method c.'.  
  ENDMETHOD.  
ENDCLASS.  
  
DATA: lr_parent TYPE REF TO lcl_parent.  
      lr_child  TYPE REF TO lcl_child.  
  
CREATE OBJECT lr_parent.  
CREATE OBJECT lr_child.  
lr_parent = lr_child.
```

Listing 6.2 Performing a Cast with an Object Reference Assignment

To understand how this kind of assignment works behind the scenes, let's take a step back and think about what is happening from a logical perspective. In Chapter 2, *Working with Objects*, we considered the relationship between a remote control and a TV as a metaphor for describing the link between an object reference variable and the object it points to. When you purchase a new TV, the pack-

age normally comes with a remote control that is able to interact with the TV out of the box. In other words, the *static type* of that remote control is defined in terms of the TV.

Now, imagine that you decide to purchase a universal remote to replace the default remote that came with the TV. In this case, even though the *static type* of the universal remote is more generic than the one provided by the manufacturer, it is still *compatible* with the public interface provided by the TV (i.e., it has common operations such as Turn On, Adjust Volume, etc.). However, before you can use the universal remote with the TV, it must first be *reprogrammed* with information about the actual TV model it is interfacing with. Similarly, object reference variables that are reassigned to point to objects that do not have the same static type must be reprogrammed with *dynamic type* information at runtime.

The dynamic type of an object reference variable refers to the class type of the object pointed to by the reference variable. In the example shown in Listing 6.2, the statement `CREATE OBJECT lr_parent` instantiates an object of type `lcl_parent` and assigns a pointer to that object to the `lr_parent` object reference variable. At this point, the static and dynamic type of the `lr_parent` reference is the same. However, when you perform the assignment statement `lr_parent = lr_child`, the dynamic type of the `lr_parent` reference is changed by the ABAP runtime environment to refer to the `lcl_child` class type. This information is crucial for the ABAP runtime environment to interact with the compatible components of the `lcl_child` object that is now being pointed to by the `lr_parent` reference variable.

It is worth mentioning that you cannot arbitrarily set the dynamic type of an object reference to an incompatible type. In other words, these kinds of assignments don't make sense without some kind of an inheritance relationship between the source and target object reference variables. Section 6.1.2, Casting, explores the rules that govern how these assignments work.

6.1.2 Casting

If the static type of the source and target object reference variables is not the same in an assignment operation, a special operation called a *cast* must occur for the assignment to work. A cast operation is allowed whenever the static type of the target object reference is the same as or more general than the dynamic type of

the source object reference. There are two different types of cast operations: a *narrowing cast* and a *widening cast*.

Narrowing Casts

A narrowing cast occurs in an object reference assignment statement whenever the static type of a target object reference variable is more generic than the static type of the source object reference variable. The assignment statement in Listing 6.3 shows an example of a narrowing cast between the reference variables `lr_parent` and `lr_child`. This type of assignment is called a narrowing cast because the class type `lcl_parent` is more general than `lcl_child`, effectively *narrowing* the scope of the components that can be accessed in the `lcl_child` object to those defined in the `lcl_parent` superclass.

```
DATA: lr_parent TYPE REF TO lcl_parent.
      lr_child  TYPE REF TO lcl_child.
CREATE OBJECT lr_parent.
CREATE OBJECT lr_child.
lr_parent = lr_child.
* CALL METHOD lr_parent->c.  "Syntax Error!
```

Listing 6.3 Attempting to Call a Method That Is Out of Scope

This reduction in scope prevents the target object reference variable (i.e., `lr_parent`) from accessing components that are not defined in its static type definition. For example, the method call that is commented out in Listing 6.3 would cause a syntax error because method `c` is not defined for type `lcl_parent`. Of course, this reduction in scope does not imply that the object itself is changed or truncated in some way; the `lcl_child` object in Listing 6.3 is still a full-fledged object of type `lcl_child`, for example.

It is also possible to use the `TYPE` addition of the `CREATE OBJECT` statement to perform an implicit narrowing cast during the instantiation process. For example, the `CREATE OBJECT` statement in Listing 6.4 creates an object of type `lcl_child` and assigns a pointer to this object to the `lr_parent` object reference, implicitly performing a narrowing cast operation along the way.

```
DATA: lr_parent TYPE REF TO lcl_parent.
CREATE OBJECT lr_parent TYPE lcl_child.
```

Listing 6.4 Performing a Narrowing Cast During Object Creation

Widening Casts

In cases where the static type of the target object reference is more specific than the static type of the source object reference, a *widening cast* has to be applied for an assignment statement to pass muster with the ABAP compiler. Widening casts allow you to take control of the assignment process by telling the compiler that you know what you are doing when you are performing your assignment.

Of course, this delegation does not mean that a validity check never takes place; it just means that it is deferred until runtime when the dynamic type of the source object reference is known. Here, as we stated before, the static type of the target object reference must be the same as or more general than the dynamic type of the source object reference. Otherwise, an exception will occur. In Chapter 8, Error Handling with Exceptions, we will look at how to recover from these types of exceptions in your programs. Still, it is recommended that you use widening casts carefully because they can be somewhat confusing (and dangerous).

Widening casts require you to use a special assignment operator called the casting operator (?=). You can also perform widening casts using the ?TO option of the MOVE statement. This syntax effectively tells the compiler to bypass the static syntax check on the assignment statement. Listing 6.5 shows how to perform a widening cast using both types of syntax.

```
DATA: lr_parent TYPE REF TO lcl_parent,
      lr_child  TYPE REF TO lcl_child,
CREATE OBJECT lr_parent TYPE lcl_child.
CREATE OBJECT lr_child.
lr_child ?= lr_parent.
MOVE lr_parent ?TO lr_child.
```

Listing 6.5 Performing Widening Casts

6.2 Dynamic Method Call Binding

Now that you understand how to use casts to perform assignments between related types, you are ready to start implementing designs using polymorphism. The report program ZPOLYTEST in Listing 6.6 contains an abstract class `lcl_animal`, two subclasses (`lcl_cat` and `lcl_dog`), and a test driver class called `lcl_see_and_say`. The `lcl_see_and_say` class is modeled loosely after the See-n-Say® educational toys manufactured by Mattel, Inc.

```
REPORT zpolytest.

CLASS lcl_animal DEFINITION ABSTRACT.
  PUBLIC SECTION.
    METHODS: get_type ABSTRACT,
             speak ABSTRACT.
ENDCLASS.

CLASS lcl_cat DEFINITION
  INHERITING FROM lcl_animal.
  PUBLIC SECTION.
    METHODS: get_type REDEFINITION,
             speak REDEFINITION.
ENDCLASS.

CLASS lcl_cat IMPLEMENTATION.
  METHOD get_type.
    WRITE: 'Cat'.
  ENDMETHOD.

  METHOD speak.
    WRITE: 'Meow'.
  ENDMETHOD.
ENDCLASS.

CLASS lcl_dog DEFINITION
  INHERITING FROM lcl_animal.
  PUBLIC SECTION.
    METHODS: get_type REDEFINITION,
             speak REDEFINITION.
ENDCLASS.

CLASS lcl_dog IMPLEMENTATION.
  METHOD get_type.
    WRITE: 'Dog'.
  ENDMETHOD.

  METHOD speak.
    WRITE: 'Bark'.
  ENDMETHOD.
ENDCLASS.
```

```

CLASS lcl_see_and_say DEFINITION.
  PUBLIC SECTION.
  CLASS-METHODS:
    play IMPORTING im_animal
          TYPE REF TO lcl_animal.
ENDCLASS.

CLASS lcl_see_and_say IMPLEMENTATION.
  METHOD play.
    WRITE: 'The'.
    CALL METHOD im_animal->get_type.
    WRITE: 'says'.
    CALL METHOD im_animal->speak.
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  DATA: lr_cat TYPE REF TO lcl_cat.
        lr_dog TYPE REF TO lcl_dog.

  CREATE OBJECT lr_cat.
  CREATE OBJECT lr_dog.

  CALL METHOD lcl_see_and_say=>play
    EXPORTING
      im_animal = lr_cat.
  NEW-LINE.
  CALL METHOD lcl_see_and_say=>play
    EXPORTING
      im_animal = lr_dog.

```

Listing 6.6 Dynamic Binding with Method Calls

In this implementation, the class method `play` allows you to play the sound made by various types of animals. If you look closely at the signature of method `play`, you will observe that it receives an importing parameter of type `lcl_animal`. However, in the `START-OF-SELECTION` event of program `ZPOLYTEST`, you will notice that this method is called at runtime with objects of type `lcl_cat` and `lcl_dog`. In this case, the ABAP runtime environment performs an implicit narrowing cast during the assignment of the importing parameter `im_animal`. This subtle feature makes it possible for the code inside the `play` method to be implemented generically.

The dynamic type information associated with an object reference variable allows the ABAP runtime environment to dynamically bind a method call with the implementation defined in the object pointed to by the object reference variable. For example, the importing parameter `im_animal` for method `play` in the `lcl_see_and_say` class from Listing 6.6 refers to an abstract type that could never be instantiated on its own. However, whenever the method is called with a concrete subclass implementation such as `lcl_cat` or `lcl_dog`, the dynamic type of the `im_animal` reference parameter is bound to one of these concrete types. Therefore, the calls to methods `get_type` and `speak` refer to the implementations provided in the `lcl_cat` or `lcl_dog` subclasses rather than the undefined abstract implementations provided in class `lcl_animal`.

Dynamic binding provides for tremendous flexibility in designs. The simple example from Listing 6.6 only implemented for a cat and a dog. In the future, developers may decide to implement subclasses for various other types of animals such as a horse, a cow, a pig, and so on. However, because the `lcl_see_and_say` class works with the generic `lcl_animal` type, they can integrate these new types into the “See-n-Say device” seamlessly. Such designs are said to be *extensible* in the sense that we can easily introduce new functionality by simply creating a new subclass and plugging it into the design.

6.3 Interfaces

Throughout the course of this book, we have used the term *interface* to describe various interaction points between classes and their clients. For example, a method's signature defines an interface that is used by clients wanting to call that method. From an object-oriented perspective, you can think of an interface as a type of *protocol* that defines rules for communicating with objects.

This analogy should be familiar because we interact with many types of protocols every day. For instance, the *Hypertext Transfer Protocol* (HTTP) defines the rules that clients (i.e., web browsers such as Microsoft Internet Explorer) and web servers must adhere to in order to reliably publish and retrieve content on the World Wide Web. These rules make it possible for your web browser to request web pages from many different types of web server implementations (e.g., Microsoft, Apache, SAP, etc.) without having to worry about how these servers are implemented. Similarly, you have seen how polymorphism can be used to dynamically bind many different types of implementations to a single interface.

Sometimes, it can be advantageous to define an interface independently of any particular class. Such interfaces do not have an implementation associated with them and consequently cannot be instantiated on their own. In this section, we will look at how interfaces can be used expand a class's scope into multiple dimensions.

6.3.1 Interface Inheritance Versus Implementation Inheritance

Some object-oriented languages support a *multiple inheritance* model, allowing you to define several inheritance relationships within a given class. As you may have guessed by now, ABAP Objects only supports a *single inheritance* model. This is a design decision that has been employed by many modern object-oriented languages in an effort to avoid some of the ambiguity that can arise with complex inheritance hierarchies.

To illustrate some of the potential problems associated with a multiple inheritance model, let's consider an example. The class diagram in Figure 6.1 depicts a diamond-shaped class hierarchy. In this case, let's imagine that classes B and C have both redefined method `someMethod` from class A. If class D does not redefine method `someMethod`, from which implementation does it inherit: B or C? This problem is known as the *diamond problem*.

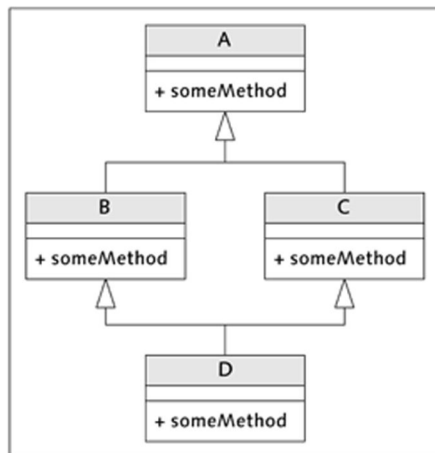


Figure 6.1 Class Diagram for Diamond-Shaped Inheritance Hierarchy

A single-inheritance model avoids these kinds of vagaries because a subclass always inherits from a single superclass. However, interfaces can enhance this model by providing a way to extend the *type* of a class without having to bring along all of the implementation baggage associated with multiple inheritance. In Section 6.3.4, Working with Interfaces, you will see how the implementation of an interface allows a class to be used polymorphically wherever a reference of that interface type is used.

6.3.2 Defining Interfaces

The syntax required to define an interface is very similar to the syntax that is used in the declaration part of a class definition. Listing 6.7 shows an example of how to define a local interface called `lif_iface`. Notice that none of the interface components have been defined within a visibility section. This is because all components of an interface are implicitly defined within the public visibility section. If you think about it, this makes sense because the purpose of an interface is to expand the public interface of implementing classes.

```
INTERFACE lif_iface.
  DATA: a TYPE string.
  METHODS: m.
  EVENTS: e.
ENDINTERFACE.
```

Listing 6.7 Syntax for Defining a Local Interface

Most of the time, you will just use interfaces to add additional methods to the public interface of a class. However, you can technically define all of the same types of components that you can define for classes in an interface (see Chapter 2, Working with Objects, for more details on the types of components that you can define for classes).

Interfaces can also be defined as global Repository objects in the Class Builder tool. To illustrate how this works, let's create a new global interface called `ZIF_COMPARABLE` that can be used to specify an ordering for implementing classes. You will see an example of how to implement and use this interface in Sections 6.3.3, Implementing Interfaces, and 6.3.4, Working with Interfaces, respectively.

1. To create a global interface in the Class Builder, enter the name in the interface in the OBJECT TYPE field, and click on the CREATE button (see Figure 6.2).

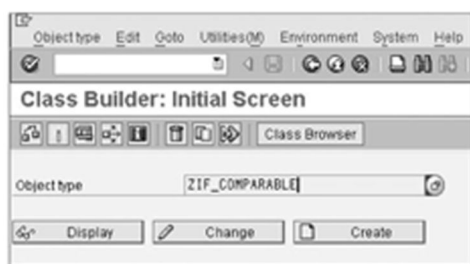


Figure 6.2 Creating a Global Interface – Part I

- The use of the `IF_` prefix convention causes the Class Builder to bring up the CREATE INTERFACE dialog box where you can confirm the name of the interface along with a short description of its purpose. Click the SAVE button to save your changes to the ABAP Repository (see Figure 6.3).

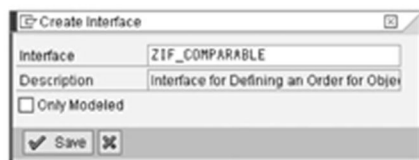


Figure 6.3 Creating a Global Interface – Part II

- After confirming the Repository details in the CREATE OBJECT DIRECTORY ENTRY screen, you are navigated to the Class Editor screen where you can edit the interface's components in the same way that you edit the components of a class (see Figure 6.4).

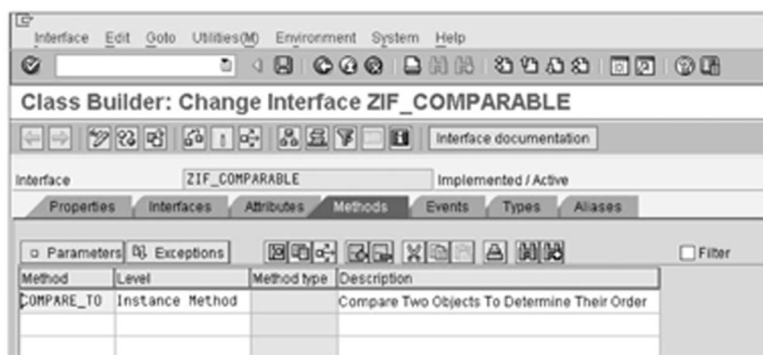


Figure 6.4 Editing a Global Interface in the Class Editor

4. The `ZIF_COMPARABLE` interface contains a single instance method called `COMPARE_TO` that can be used to compare two objects to determine if the source object is greater than, less than, or equal to the target object. Here, define the `IM_OBJECT` parameter with the generic `OBJECT` type (see Figure 6.5). Because every ABAP Objects class is implicitly derived from this class, you can implement this interface for any class type.

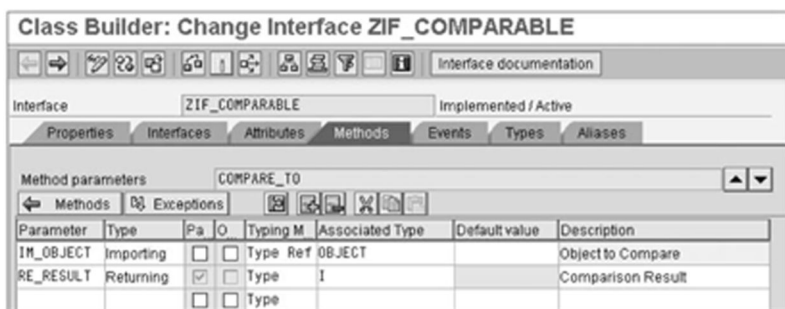


Figure 6.5 Defining an Instance Method in an Interface

6.3.3 Implementing Interfaces

Interfaces are not all that interesting until you start implementing them in classes. You can implement an interface in a local class using the `INTERFACES` keyword. Listing 6.8 shows how a local class called `lcl_implementer` implements the `lif_iface` interface.

```
INTERFACE lif_iface.
  METHODS: m1,
           m2.
ENDINTERFACE.

CLASS lcl_implementer DEFINITION.
  PUBLIC SECTION.
    INTERFACES: lif_iface.
ENDCLASS.

CLASS lcl_implementer IMPLEMENTATION.
  METHOD lif_iface~m1.
    WRITE: 'In method lif_iface~m1'.
  ENDMETHOD.
```

```

METHOD lif_iface~m2.
    WRITE: 'In method lif_iface~m2'.
ENDMETHOD.
ENDCLASS.

```

Listing 6.8 Implementing an Interface in a Local Class

You can declare the implementation of interfaces in a global class by entering the interface name in the **INTERFACE** column on the **INTERFACES** tab of the Class Editor screen (see Figure 6.6).

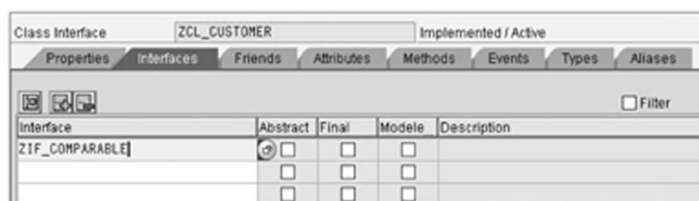


Figure 6.6 Implementing an Interface in a Global Class

After you implement an interface in a class, you will see the components of that interface show up as components for the implementing class. In Figure 6.7, class **ZCL_CUSTOMER** has been defined that implements the **ZIF_COMPARABLE** interface. On the **METHODS** tab of the Class Editor, you can see that the **ZIF_COMPARABLE~COMPARE_TO** method has been added to the list of methods defined for class **ZCL_CUSTOMER** (see Figure 6.7).

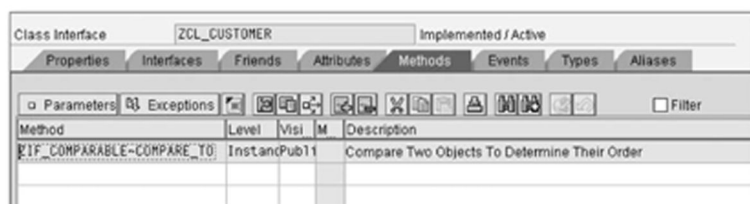


Figure 6.7 Inheriting Interface Components in a Global Class

The implementation of the **ZIF_COMPARABLE** interface extends the scope of customer objects, providing a means for sorting customers by their internal customer ID number. Of course, this extension cannot be realized until an implementation for the **COMPARE_TO** method is provided. The code in Listing 6.9

provides an implementation for the `COMPARE_TO` method for the `ZCL_CUSTOMER` class. This method must be addressed using the fully qualified interface component name `ZIF_COMPARABLE~COMPARE_TO` rather than just simply `COMPARE_TO`. The tilde (`~`) between the interface name and the interface component name is called the *interface component selector operator*.

```
METHOD zif_comparable~compare_to.
* Method-Local Data Declarations:
  DATA: lr_object TYPE REF TO zcl_customer.

* Perform a widening cast on the comparison object:
  lr_object ?= im_object.

* Compare the two customers based on their ID number:
  IF me->id GT lr_object->id.
    re_result = 1.
  ELSEIF me->id LT lr_object->id.
    re_result = -1.
  ELSE.
    re_result = 0.
  ENDIF.
ENDMETHOD.
```

Listing 6.9 Implementing an Interface Method in a Global Class

The comparison for class `ZCL_CUSTOMER` is based on the internal customer `id` attribute (see Figure 6.8). Here, notice that a widening cast on the comparison object parameter `IM_OBJECT` is required to be able to access the `id` attribute on the customer comparison object because it is undefined for objects of type `OBJECT`.

Attribute	Level	Visibility	Re	Typing	Associated Type	Description	Initial value
NEXT_ID	Static Attribute	Private	<input type="checkbox"/>	Type	I	Next Available Customer Number	
ID	Instance Attribute	Private	<input type="checkbox"/>	Type	I	Customer ID Number	
NAME	Instance Attribute	Private	<input type="checkbox"/>	Type	STRING	Customer Name	
			<input type="checkbox"/>	Type			
			<input type="checkbox"/>	Type			
			<input type="checkbox"/>	Type			

Figure 6.8 Attribute Definitions for Class `ZCL_CUSTOMER`

The customer ID number is initialized in the `CONSTRUCTOR` method for class `ZCL_CUSTOMER`. The definition and implementation of this method is shown in Figure 6.9 and Listing 6.10, respectively.

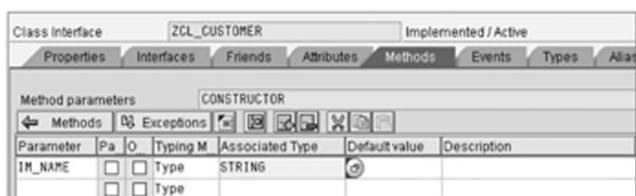


Figure 6.9 Defining the Signature of Method `CONSTRUCTOR`

```
METHOD constructor.
  next_id = next_id + 1.
  id = next_id.
  name = im_name.
ENDMETHOD.
```

Listing 6.10 Implementing Method `CONSTRUCTOR`

6.3.4 Working with Interfaces

At this point, you might be wondering why you would ever want to go to all of the trouble of defining an interface and implementing it in a class. To demonstrate the value of interfaces, let's look at how you might use the `ZIF_COMPARABLE` interface defined in Section 6.3.2, Defining Interfaces, practically in a real-world scenario.

ABAP Objects allows you to define the line type of an internal table using reference types. For example, we can define an internal table to hold customer objects of type `ZCL_CUSTOMER` using the syntax shown in Listing 6.11.

```
DATA: lt_customers TYPE STANDARD TABLE
      OF REF TO zcl_customer.
```

Listing 6.11 Defining Internal Tables Using Reference Types

Internal tables such as the one shown in Listing 6.11 are convenient for storing and looping through object references. However, one thing that you can't easily do with this kind of table is sort. After all, how do you sort a set of objects of type `ZCL_CUSTOMER`, and so on? This is where interface `ZIF_COMPARABLE` comes in.

Classes that implement the `COMPARE_TO` method of interface `ZIF_COMPARABLE` have a mechanism for determining a sort order. The only thing missing is the logic to actually perform the sort operation. Of course, if you go to the trouble of writing the sort operation, you want to be able to reuse it again for other classes. Therefore, it makes sense to wrap this logic inside of a generic container that can store objects, sort them, and so on. In many object-oriented language implementations, this type of container is called a *vector*.

The global class `ZCL_VECTOR` shown in Figure 6.10 provides an example of a basic vector implementation in ABAP Objects. For now, the implementation simply provides some methods to add and remove elements from the vector, sort the elements, and so on.

Method	Level	Visibility	M	Description
CONSTRUCTOR	Instance Method	Public		CONSTRUCTOR
ADD	Instance Method	Public		Add an Element to the Vector
REMOVE	Instance Method	Public		Remove an Element from the Vector
SORT	Instance Method	Public		Sort the Elements in the Vector
ITERATOR	Instance Method	Public		Return an Iterator to Read the Elements in the Vector

Figure 6.10 Utility Class `ZCL_VECTOR`

The actual vector elements are stored in a private attribute called `ELEMENTS` of type `SWF_UTL_OBJECT_TAB` (see Figure 6.11).

Attribute	Level	Visibility	Read-Only	Typing	Associated Type	Description
ELEMENTS	Instance Attribute	Private	<input type="checkbox"/>	Type	SWF_UTL_OBJECT_TAB	Table Type for Runtime Objects

Figure 6.11 Defining Attribute `ELEMENTS` in Class `ZCL_VECTOR`

The table type `SWF_UTL_OBJECT_TAB` has a reference line type pointing to objects of type `OBJECT` (see Figure 6.12). Because every ABAP Objects class implicitly inherits from type `OBJECT`, we can store any type of object inside of our vector container.

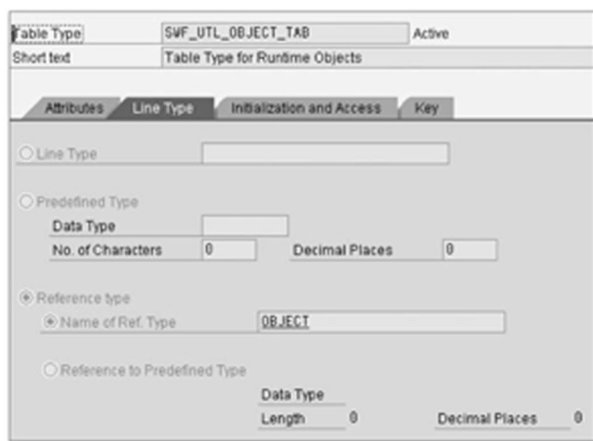


Figure 6.12 Standard Table Type SWF_UTL_OBJECT_TAB

To add an object to the vector, we use the `ADD` method shown in Figure 6.13 and Listing 6.12. This method simply appends the `IM_OBJECT` parameter to the private `ELEMENTS` attribute. Note that the type of the `IM_OBJECT` parameter is the generic `OBJECT` type. This allows callers to store any type of object inside the vector.

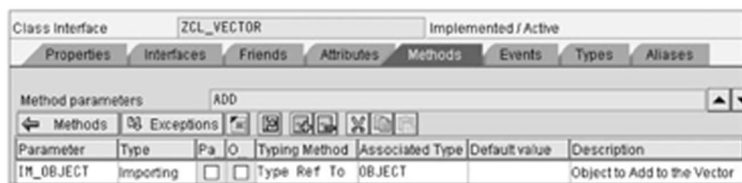


Figure 6.13 Defining the Signature of Method ADD

```
METHOD add.
* Add an object to the vector:
  APPEND im_object TO elements.
ENDMETHOD.
```

Listing 6.12 Implementing Method ADD

Elements can be removed from the vector using the `REMOVE` method shown in Figure 6.14 and Listing 6.13. Here, we simply need to verify that the provided index is within the bounds of the internal table. If it is, then the element can be removed, and the vector remains in sorted order (provided it was sorted in the first place).

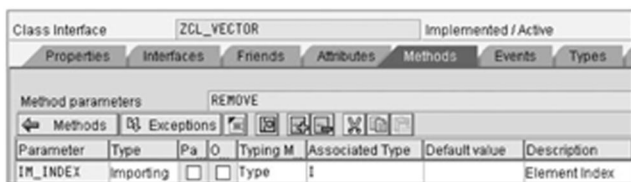


Figure 6.14 Defining the Signature of Method REMOVE

```

METHOD remove.
* Remove the element at the provided index.
* assuming it is in bounds:
  IF im_index GE 1 AND im_index LE lines( elements ).
    DELETE elements INDEX im_index.
  ENDIF.
ENDMETHOD.

```

Listing 6.13 Implementing Method REMOVE

After we have populated the vector, we can sort its elements using the `SORT` method shown in Listing 6.14. For demonstration purposes, the simple (but inefficient) *Insertion Sort* algorithm is used to perform the sort operation. This algorithm operates in a similar fashion to the way you might sort a hand of playing cards. For example, after the dealer deals you your cards, they are laying face down on the table. As you pick up each card, you insert it into the proper position in your hand (i.e., based on the value of the card). In the case of method `SORT`, the ordering of the elements in the vector is determined by a class's implementation of the `COMPARE_TO` method defined in interface `ZIF_COMPARABLE`. In other words, this method assumes that the elements of the vector implement the `ZIF_COMPARABLE` interface.

```

METHOD sort.
* Method-Local Data Declarations:
  DATA: lr_key      TYPE REF TO object,
         lr_element  TYPE REF TO object,
         lr_compare  TYPE REF TO zif_comparable,
         lr_temp     TYPE REF TO object,
         lv_i        TYPE i,
         lv_j        TYPE i VALUE 2,
         lv_index    TYPE i.

```



```

* Sort the vector elements using the Insertion Sort
* algorithm:
LOOP AT elements INTO lr_key FROM 2.
    lv_i = lv_j - 1.
    READ TABLE elements INDEX lv_i INTO lr_element.
    lr_compare ?= lr_element.

    WHILE lv_i GT 0 AND
        lr_compare->compare_to( lr_key ) EQ 1.
        READ TABLE elements INDEX lv_i INTO lr_temp.
        lv_index = lv_i + 1.
        MODIFY elements FROM lr_temp INDEX lv_index.

        lv_i = lv_i - 1.
        READ TABLE elements INDEX lv_i INTO lr_element.
        lr_compare ?= lr_element.
    ENDWHILE.

    lv_index = lv_i + 1.
    MODIFY elements FROM lr_key INDEX lv_index.
    lv_j = lv_j + 1.
ENDLOOP.
ENDMETHOD.

```

Listing 6.14 Implementing Method SORT

To use the functionality provided by the `ZIF_COMPARABLE` interface, we must perform a widening cast on the elements we are comparing because each element in the vector has the generic type `OBJECT`. However, we don't want to perform the cast in terms of each of the various classes because this would make our `SORT` method too specific. Instead, we perform a widening cast using the `lr_compare` reference variable. Here, notice that the static type of the `lr_compare` reference variable is the interface type `ZIF_COMPARABLE`. Reference variables such as `lr_compare` are referred to as *interface references*. Interface reference variables can be assigned to point to objects of classes that implement the interface type (i.e., static type) of the interface reference.

Therefore, the assignment statement `lr_compare ?= lr_element` is valid because we assume that the dynamic type of the object pointed to by `lr_element` provides an implementation of the `ZIF_COMPARABLE` interface. Interface reference variables are allowed to directly access the components defined by the interface. Consequently, as you can see in Listing 6.14, the call to method `COMPARE_TO` does

not require the use of the interface component selector operator (e.g., `ZIF_COMPARABLE~COMPARE_TO`, etc.) that is required whenever we address interface components using a regular object reference variable.

The `ZCL_VECTOR` class uses an *iterator* to provide access to its elements. You can think of an iterator as a type of cursor that lets you traverse through the elements of a collection. The method `ITERATOR` shown in Figure 6.15 and Listing 6.15 returns an iterator object of type `CL_SWF_UTL_ITERATOR`. Class `CL_SWF_UTL_ITERATOR` provides a series of methods that enable you to read elements from the vector without having direct access to the private `elements` attribute hidden inside class `ZCL_VECTOR`.

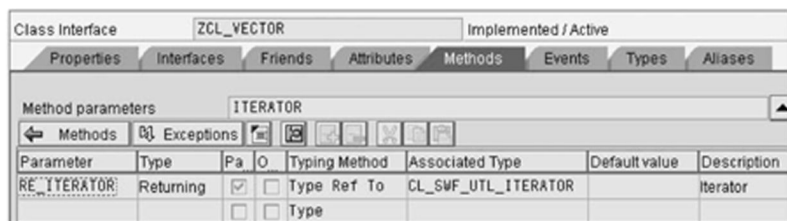


Figure 6.15 Defining the Signature of Method ITERATOR

```
METHOD iterator.
* Create an iterator object to provide access to the
* vector elements:
  CREATE OBJECT re_iterator
  EXPORTING
    im_object_list = elements.
ENDMETHOD.
```

Listing 6.15 Implementing Method ITERATOR

As stated before, the implementation of an interface expands the scope of a class. For instance, the class `ZCL_CUSTOMER` is a customer, but it is also *comparable*. Thus, just because you can only inherit from a single superclass does not mean that you can't expand the public interface of your class into additional dimensions using interfaces. In fact, you are free to implement as many interfaces in a class as you want.

Often, you can use a design technique known as *forwarding* to quickly implement an interface using composition techniques. In this case, the methods of the inter-

face are implemented in a class using functionality provided by composed objects. The simple report program ZVECTORTEST shown in Listing 6.16 demonstrates how to use the generic vector class developed in this section.

```
REPORT zvectortest.

DATA: gr_customer1 TYPE REF TO zcl_customer,
      gr_customer2 TYPE REF TO zcl_customer,
      gr_customer3 TYPE REF TO zcl_customer,
      gr_vector     TYPE REF TO zcl_vector.

START-OF-SELECTION.
* Create three sample customers:
CREATE OBJECT gr_customer1
  EXPORTING
    im_name = 'Andrea'.

CREATE OBJECT gr_customer2
  EXPORTING
    im_name = 'Andersen'.

CREATE OBJECT gr_customer3
  EXPORTING
    im_name = 'Paige'.

* Add the customers to the vector container in
* random order:
CREATE OBJECT gr_vector.

CALL METHOD gr_vector->add
  EXPORTING
    im_object = gr_customer2.

CALL METHOD gr_vector->add
  EXPORTING
    im_object = gr_customer3.

CALL METHOD gr_vector->add
  EXPORTING
    im_object = gr_customer1.

* Show the customers before the sort operation:
```

```

    PERFORM show_customers USING gr_vector.

* Now, sort the customers:
CALL METHOD gr_vector->sort( ).

* Show the customers after the sort operation:
PERFORM show_customers USING gr_vector.

FORM show_customers USING im_vector TYPE REF TO zcl_vector.
* Local Data Declarations:
DATA: lr_iterator TYPE REF TO cl_swf_utl_iterator,
      lv_count    TYPE i,
      lr_object   TYPE REF TO object,
      lr_customer TYPE REF TO zcl_customer.

lr_iterator = im_vector->iterator( ).
lv_count = lr_iterator->get_count( ).
DO lv_count TIMES.
    lr_object = lr_iterator->get_current( ).
    lr_customer ?= lr_object.
    lr_customer->display( ).

    lr_iterator->get_next( ).
ENDDO.
ENDFORM.

```

Listing 6.16 Example Program Demonstrating Class ZCL_VECTOR

It is recommended that you make liberal use of interfaces in your designs because they are extremely flexible to work with. In the book *Java Programming Language* (Addison-Wesley, 2006), James Gosling, the inventor of the Java programming language, suggests that "... every major class in an application should be an implementation of some interface that captures the contract of that class." One can argue as to whether or not position is too extreme for practical purposes, but there is value to be gained in developing logic based on interfaces because you can integrate any class that implements that interface seamlessly into your design.

6.3.5 Nesting Interfaces

So far, we have only considered simple, elementary interfaces. However, it is possible to *nest* interfaces inside of a compound or *nested interface*. Interfaces embedded inside of a nested interface are called *component interfaces*. Listing 6.17 shows

an example of the syntax used to nest the component interface `lif_component` inside of the nested interface `lif_nested`. As you can see, interfaces are nested using the `INTERFACES` statement.

```
INTERFACE lif_component.  
    METHODS: c1.  
            c2.  
ENDINTERFACE.  
  
INTERFACE lif_nested.  
    INTERFACES: lif_component.  
    METHODS: n1.  
            n2.  
ENDINTERFACE.
```

Listing 6.17 Nesting Interfaces Example

You can assign component interfaces to a global nested interface by entering the component interface name in the `INCLUDES` column on the `INTERFACES` tab of the Class Editor (see Figure 6.16).

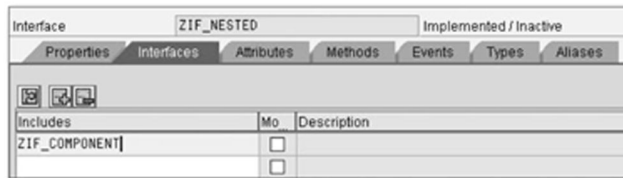


Figure 6.16 Nesting Interfaces in Global Interfaces

All of the components in a nested interface exist at the same level. If a given component interface happens to be nested more than one time, there will only be a single instance of the components defined in that component interface inside the nested interface.

Components of component interfaces are not directly visible in the nested interface. However, you can make these components visible in a nested interface by defining *alias names* for the components. Listing 6.18 shows how the `ALIASES` statement was used to define aliases for the `c1` and `c2` methods of component interface `lif_component` in nested interface `lif_nested`. The class `lcl_`

`nested_impl` is now able to implement the nested methods defined in the `lif_component` component interface.

```

INTERFACE lif_component.
    METHODS: c1,
             c2.
ENDINTERFACE.

INTERFACE lif_nested.
    INTERFACES: lif_component.
    ALIASES: c1 FOR lif_component~c1,
            c2 FOR lif_component~c2.
    METHODS: n1,
             n2.
ENDINTERFACE.

CLASS lcl_nested_impl DEFINITION.
    PUBLIC SECTION.
        INTERFACES: lif_nested.
ENDCLASS.

CLASS lcl_nested_impl IMPLEMENTATION.
    METHOD lif_nested~n1.
    ENDMETHOD.

    METHOD lif_nested~n2.
    ENDMETHOD.

    METHOD lif_component~c1.
    ENDMETHOD.

    METHOD lif_component~c2.
    ENDMETHOD.
ENDCLASS.

```

Listing 6.18 Working with Alias Names

Much like you saw in Section 6.1.2, Casting, you can also perform casts in assignments between interface reference variables. For example, Listing 6.19 shows an example of how you can perform a narrowing cast between interface reference variables defined using the static types `lif_component` and `lif_nested` from Listing 6.18. In this case, the narrowing cast is allowed because `lif_component` is a component interface of `lif_nested`.

```

DATA: lr_component TYPE REF TO lif_component.
      lr_nested    TYPE REF TO lif_nested.
CREATE OBJECT lr_nested TYPE lcl_nested_impl.
lr_component = lr_nested.
CALL METHOD lr_component->c1.

```

Listing 6.19 Performing Narrowing Casts Using Interface References

6.4 UML Tutorial: Advanced Class Diagrams Part II

In this section, we will complete our discussion of the UML class diagram by introducing the notation for working with interfaces and their components.

6.4.1 Interfaces

The notation for defining interfaces in a UML class diagram is almost identical to the one used to define classes. The only difference is the addition of the `<< interface >>` tag in the top name section of the interface notation (see Figure 6.17).

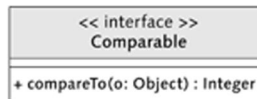


Figure 6.17 Notation for Defining Interfaces

The relationship between a nested interface and its component interfaces is shown using the same generalization notation used to depict inheritance relationships. For example, in Figure 6.18, the `Nested` interface is inheriting the components from interface `Component`.

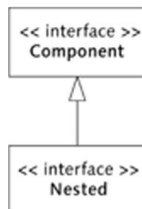


Figure 6.18 Notation for Defining Nested Interfaces

6.4.2 Providing and Required Relationships with Interfaces

Figure 6.19 shows two kinds of relationships that a class can have with an interface. The dashed line between class `Customer` and interface `Comparable` indicates that class `Customer` *provides* (or implements) the `Comparable` interface. Notice how the notation for this relationship is similar to the one you have seen for generalization relationships. In this case, the interface `Comparable` represents one kind of generalization for class `Customer`. Implicitly, this tells us that we can substitute instances of class `Customer` in places where the interface `Comparable` is used. The dashed arrow between class `Vector` and interface `Comparable` represents a dependency, indicating that class `Vector` *requires* the `Comparable` interface in some way. As you saw in Section 6.3.4, Working with Interfaces, this dependency exists in method `sort`, which performs comparisons between vector elements using the `compareTo` method defined in interface `Comparable`.

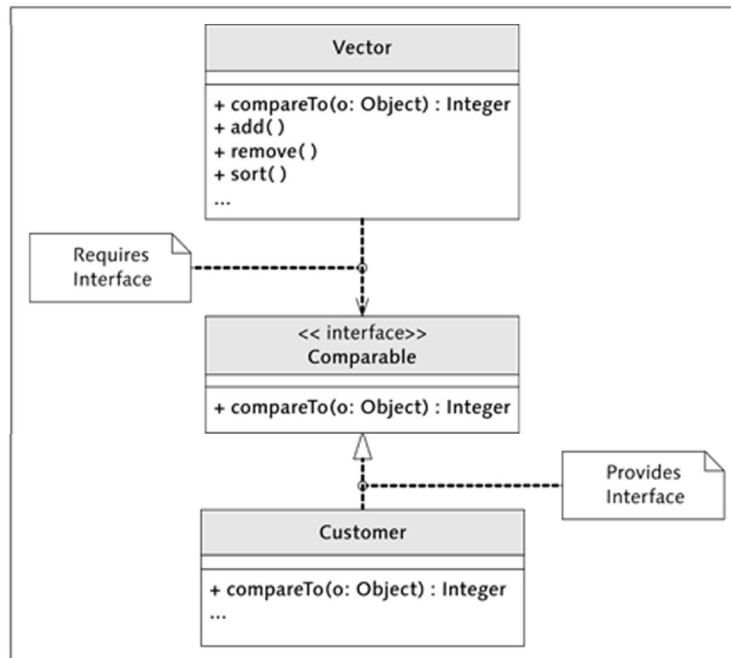


Figure 6.19 Defining Providing and Required Relationships

6.4.3 Static Attributes and Methods

You can define static (class) attributes and methods by simply underlining them in a class or interface icon. In Figure 6.20, class `Point` has four static attributes `QUADRANT1`, `QUADRANT2`, `QUADRANT3`, and `QUADRANT4` as well as a static method called `getDistance()`.

Point
<u>+QUADRANT1 : Integer</u>
<u>+QUADRANT2 : Integer</u>
<u>+QUADRANT3 : Integer</u>
<u>+QUADRANT4 : Integer</u>
<u>+getDistance()</u>
...

Figure 6.20 Defining Static Attributes and Methods

6.5 Summary

This chapter concludes our basic introduction to object-oriented programming. In many ways, the powerful designs that you can implement with polymorphism represent part of the big payoff for all of the hard work that goes into designing families of abstract data types. In Section 6.3, Interfaces, we expanded our view of abstract data types by introducing the concept of an interface. Interfaces are a welcome addition to any developer's tool bag, providing tremendous flexibility for defining complex type hierarchies.

In the next chapter, we will look at ways to organize and partition our class libraries into high-level software components using the SAP package concept.

A component-based approach to software engineering breaks a system down into a series of logical components that communicate using well-defined interfaces. When designed properly, these components become reusable software assets that can be used elsewhere in other projects or systems. In this chapter, we will look at ways to implement component-based designs in ABAP Objects.

7 Component-Based Design Concepts

Now that you have learned the basic principles of object-oriented software development in ABAP Objects, we can begin to broaden our focus by looking at ways to organize class libraries and their related resources into reusable software components. This process begins with the assignment of an ABAP development object to a modular software unit called a *package*. Packages bring structure to ABAP development objects, transforming fine-grained code libraries into coarse-grained development components.

In this chapter, you will learn how to create and work with packages. You will also see how packages fit into the overall SAP component-based software logistics model. These concepts will help you keep your software catalog organized as your class libraries evolve over time.

7.1 Understanding the SAP Component Model

To understand how to effectively implement component-based software designs in an ABAP development environment, it is helpful to be familiar with the component model that SAP uses to manage its own internal software logistics. As you can see in Figure 7.1, SAP assembles its software products as aggregates using high-level software units called *software components*. Software components are composed of a series of *packages* that organize the development objects that provide the actual implementation part of the system (e.g., classes, function groups,

ABAP Dictionary objects, etc.). You will learn more about packages in Section 7.2, The Package Concept.

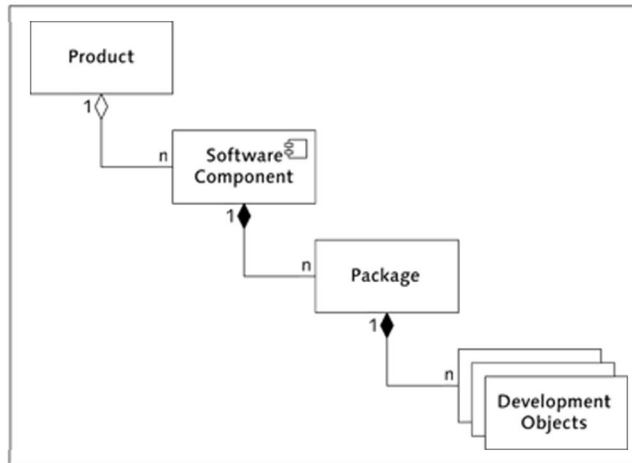


Figure 7.1 The SAP Component Model for ABAP Software Logistics

All of the objects shown in Figure 7.1 exist in versions. For example, many customers have release 6.0 of the SAP ERP product installed in their landscape. This product is assembled using particular versions¹ of the software components `SAP_HR`, `SAP_APPL`, and so on. Each software component version is composed of a series of packages that contain development objects whose versions are managed inside the ABAP Repository.

You can see a list of the installed software components in your SAP NetWeaver Application Server system by selecting `SYSTEM • STATUS` from the menu bar and clicking on the `COMPONENT INFORMATION` button (see Figure 7.2). In addition to the components shown in Figure 7.2, each ABAP system also contains two other software components called `HOME` and `LOCAL`, which are used for customer and local developments, respectively.

¹ Software components actually have a release version as well as a support package level. For example, notice that the software component `SAP_ABA` shown in Figure 7.2 has a release version 700 (7.0) and support package level 15 (SP15).

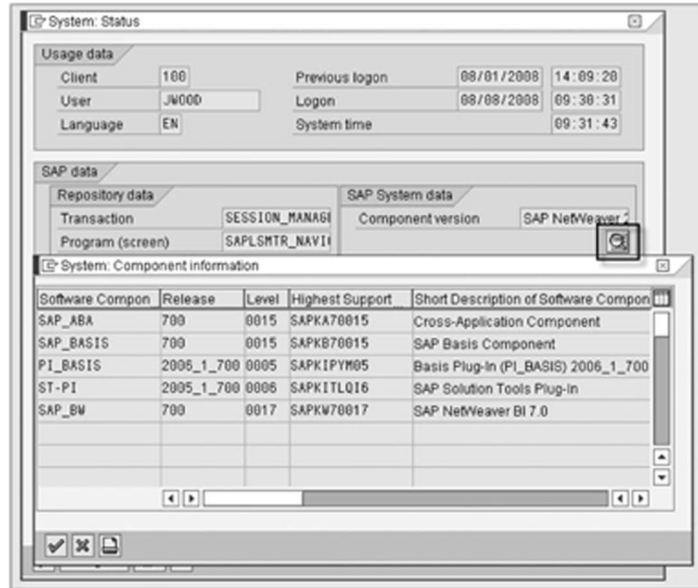


Figure 7.2 Showing the Installed Software Components in a System

Adopting a layered approach to system design has many advantages. First of all, it helps organize the software into logical pieces that are easier to work with. This is particularly important for a large organization such as SAP because it has developers working on its software around the world. Secondly, it promotes reuse of common components in other systems. For example, SAP reuses the `SAP_BASIS` and `SAP_ABA` components in other SAP Business Suite solutions such as SAP Customer Relationship Management (SAP CRM), SAP Supply Chain Management (SAP SCM), and so on. Finally, it makes the software more extensible because defined dependencies between components make it easier to determine how to integrate new or revised components into the system.

7.2 The Package Concept

Prior to release 6.10 of the SAP Web Application Server (today named SAP NetWeaver Application Server), all development objects within the ABAP Repository were grouped together into logical containers called *development classes*. The

development class concept provided a simple way for categorizing related development objects by functional area. In version 6.10 of the SAP Web Application Server, SAP replaced the development class concept with the *package concept*. At the time, the release of the package concept was not met with much fanfare because many developers assumed that the term “package” was simply a new name for a development class. However, as you will see in this section, the package concept represents a significant upgrade to the logistical capabilities provided for developers interested in improving the organization of their ABAP-based software development projects.

7.2.1 What Is a Package?

Packages are used to *encapsulate* related development objects together inside of a logical container. We emphasize the term encapsulate here because packages do more than arbitrarily assign a category to a development object; they also allow you to define boundaries that control how development objects are used outside of their enclosing package. These boundaries extend the encapsulation concepts you learned about for classes in Chapter 3, Encapsulation and Implementation Hiding, to larger abstract development components that can be more easily reused and integrated into other systems.

There are three types of packages you can use to organize your ABAP development objects: *structure packages*, *main packages*, and *sub-packages*. These package types are organized into the hierarchical structure shown in Figure 7.3. In the following subsections, you will learn about the function and purpose of each of these three package types.

Structure Packages

Structure packages, as their name suggests, are used to provide *structure* around the lower-level packages that define the various modules used to implement functionality in the system. As such, structure packages are not *extendable* in the sense that you cannot add development objects directly underneath them. Instead, structure packages can only embed other packages. Because structure packages sit at the top of the package hierarchy, they tend to be very general. For example, SAP organizes all of its Basis-related development objects underneath the structure package `BASIS`.

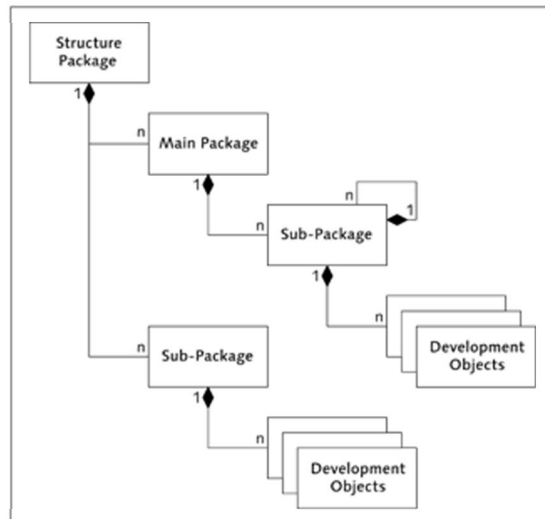


Figure 7.3 Structure of the ABAP Package Hierarchy

In the customer environment, structure packages tend to be used to organize all of the development within a given project. Although this is a reasonable approach, it is recommended that you avoid taking this relationship too literally when you actually begin to define your structure packages. For example, many projects tend to have strategic names based on catchy acronyms, release schedules, and so on. Here, it is preferable to name the structure package according to the functionality implemented by that project because it will have more semantic meaning in the long term than the fleeting project name.

Main Packages

Underneath a structure package, you can organize your development into high-level modules called main packages. Main packages are typically used to group development objects by *function*. Development objects embedded inside a main package are logically related in some way. Often, a main package is used to group together modules related to the development of a complex application, and so on. However, main packages, just like structure packages, cannot have development objects embedded directly beneath them (refer to Figure 7.3).

Sub-Packages

ABAP development objects can only be directly assigned to sub-packages. This assignment occurs whenever you are prompted with the CREATE OBJECT DIRECTORY ENTRY dialog box shown in Figure 7.4. Here, we are assigning the class ZCL_SOME_CLASS to a sub-package called ZSUBPKG.

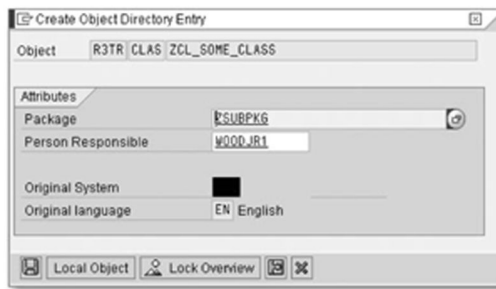


Figure 7.4 Assigning ABAP Repository Objects to Packages

You use sub-packages to organize closely related development objects. In some cases, you might group an entire application such as a simple ALV report inside a sub-package that is nested directly underneath a structure package.

However, most applications are much more complex. For example, let's imagine that you are creating a new Web Dynpro ABAP (WDA) application. WDA uses the Model-View-Controller (MVC) design pattern for separating the user interface from the underlying business/data model that is being manipulated. Here, it is conceivable that you might have a separate sub-package for the WDA application, the class library that implements the business model, and the ABAP Dictionary elements that provide the persistence layer for the application. Partitioning the elements of the application this way helps keep things organized. It also makes it easier for multiple developers working on a single application to work within a defined logistics area.

7.2.2 Creating and Organizing Packages Using the Package Builder

You can maintain packages using the Package Builder transaction (Transaction SE21). Figure 7.5 shows the initial screen of the Package Builder.

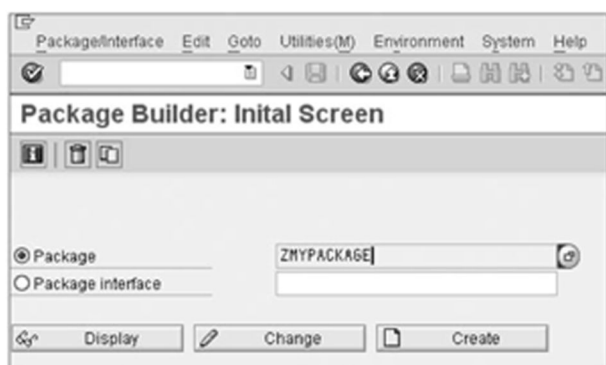


Figure 7.5 Package Builder Transaction: Initial Screen

The Package Builder application is also integrated into the Object Navigator (Transaction SE80) as you can see in Figure 7.6. Most of the time, you will want to work with packages inside the Object Navigator because its context-sensitive features can be much easier to work with.



Figure 7.6 Integration of the Package Builder in the Object Navigator

1. To create a new package,² navigate to Transaction SE21.
2. Enter a package name, and click on the CREATE button (refer to Figure 7.5) to open the CREATE PACKAGE dialog box shown in Figure 7.7.

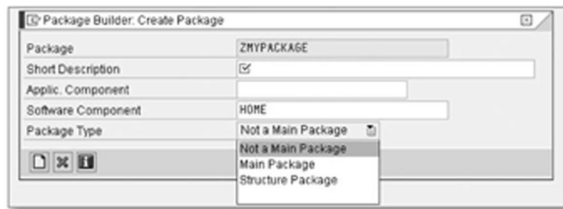


Figure 7.7 Package Builder: Create Package Dialog Box

3. In the CREATE PACKAGE dialog box, provide a SHORT DESCRIPTION of the package, the PACKAGE TYPE, and the SOFTWARE COMPONENT to which it belongs to (typically HOME for customer development).

In addition, notice that packages can be assigned to an application component within the SAP Application Hierarchy. The SAP Application Hierarchy (Transaction SE81) is used to organize SAP software from a logical or business perspective.

In Figure 7.8, you can see that the BASIS structure package has been assigned to the BC (i.e., Basis Components) application node of the SAP Application Hierarchy.

It is highly recommended that you align your packages closely with the SAP Application Hierarchy because this makes it much easier for new team members to locate custom development objects. For example, let's imagine that a new team member is looking for a class library that works with purchase orders. Without proper organization (and oftentimes documentation), a developer is forced to do a ZCL* lookup to scan through all of the custom classes defined within the ABAP Repository. However, if the purchase order class library has been added to a package that is assigned to the proper node in the SAP Application Hierarchy, a devel-

² Packages are normally maintained by Basis administrators, so even if you have the proper authorizations to access these transactions, you will want to check with them before experimenting with the creation of a package. In particular, you will want to work with them to come up with a strategy for integrating your packages with the Change and Transport System (e.g., via the TRANSPORT LAYER setting).

oper can narrow his search to classes defined in custom packages underneath the MM-PUR node in the SAP Application Hierarchy.

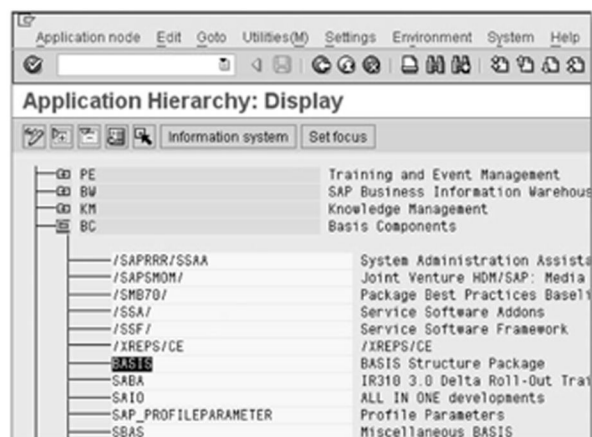


Figure 7.8 Integrating Packages into the SAP Application Hierarchy

Another advantage of aligning your packages with the SAP Application Hierarchy is that you force yourself to organize your development objects according to the same criteria used by SAP. Often, a by-product of this categorization process is the discovery of pre-existing development objects provided by SAP that may already satisfy your requirements. Or, even if you don't find an exact match for your requirements, you might stumble onto functionality that can minimize your development effort (e.g., via composition).

7.2.3 Embedding Packages

To build the kind of package hierarchy shown earlier in Figure 7.3, you must be able to embed packages inside other packages as shown here:

1. Click on the PACKAGES INCLUDED tab of the Package Builder (see Figure 7.9).
2. Click the ADD button to create a new package underneath the current package, or click on the ADD EXISTING PACKAGE button to add an existing package underneath the current package.
3. In either case, click the SAVE button in the Package Builder to save the changes to the embedding package.

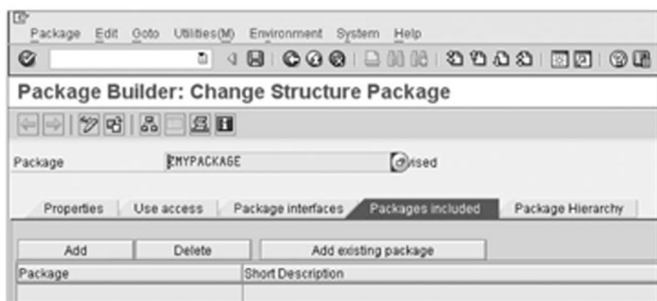


Figure 7.9 Embedding Packages Inside of Another Package

7.2.4 Defining Package Interfaces

Typically, you want to build components (i.e., packages) like “black boxes.” In other words, you want to be able to restrict access to the underlying development objects inside a package so that only certain development objects of your choosing are visible. These visible development objects make up the package’s *interface*. You can create a package interface by following these steps:

1. Select the PACKAGE INTERFACES tab in the Package Builder, and click on the ADD button.
2. In the CREATE PACKAGE INTERFACE dialog box shown in Figure 7.10, enter the name of the package interface along with a short description, and click on the APPLY VALUES button to add the package interface to the package (remember to also save the package by clicking on the SAVE button in the Package Builder).

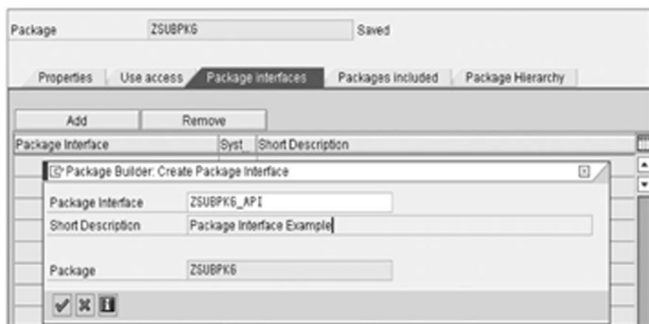


Figure 7.10 Creating a New Package Interface

- After the package interface is created, double-click it to open the CHANGE PACKAGE INTERFACE screen of the Package Builder (see Figure 7.11). Click the ADD button to add development objects from the package into the package interface.



Figure 7.11 Editing the Package Interface

- Normally, however, you will want to add elements by dragging and dropping development objects from the object list on the left-hand side of the Object Navigator onto the folder nodes entitled FROM THIS PACKAGE and FROM EMBEDDED PACKAGES³ (see Figure 7.12).

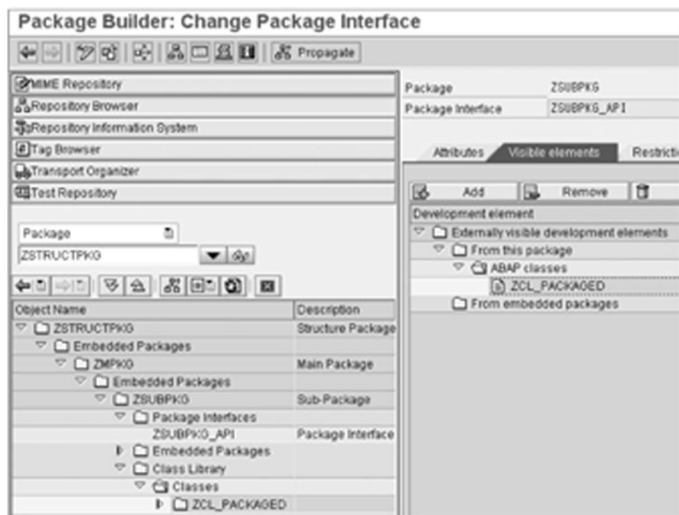


Figure 7.12 Adding Elements to a Package Interface

- Note that development objects from embedded packages must first be added to a package interface of their enclosing package before they can be embedded in a parent package. You can also add the entire package interface of a sub-package using the ADD PACKAGE INTERFACE button shown in Figure 7.11.

You should design your package interfaces using the *least privilege* principle described in Chapter 5, Inheritance. In other words, only add the development objects that outside packages will need to carry out their work. This way, you can potentially use the *use access* functionality described in Section 7.2.5, Creating Use Accesses, to ensure that no one can use development objects hidden within a package. This gives you the flexibility to change or remove development objects within a package without having to worry about what kind of problems it might cause to other programs.

7.2.5 Creating Use Accesses

Frequently, development objects in one package depend on development objects defined in another package. Prior to the release of the package concept, dependencies between development objects could be created at whim by developers without any restrictions whatsoever. From a logistics perspective, this was highly problematic because it was next to impossible to prevent developers from using development objects that they shouldn't be using for one reason or another. Here, for example, developers might start using a "helper" class or function that was not intended to be used publicly. Once this dependency exists, it is difficult to modify (and remove) this helper module without causing widespread problems in the system.

This logistics problem can be solved by creating explicit *use accesses* between packages. Technically, these dependency checks are only enforced whenever certain system settings are turned on (more on this in Section 7.2.6, Performing Package Checks). A use access formally declares a client (or user) package's intentions to use development objects defined in the package interface of a server (or provider) package. To define a use access, follow these steps:

1. Open the client package in the Package Builder, and click on the USE ACCESS tab.
2. Click on the CREATE button to open the CREATE USE ACCESS dialog box shown in Figure 7.13. Choose the desired package interface defined in the provider package. In all but the most extraordinary situations, you should always select the NO RESPONSE option in the ERROR SEVERITY listbox (see the SAP online help documentation for more information about the other options). This option makes sure that messages do not show up in the extended program check tool (Transaction SLIN) whenever a *true permission* exists. The other options allow you to tune the severity of the messages as appropriate.

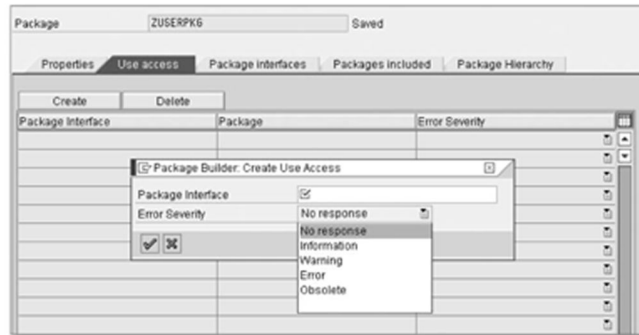


Figure 7.13 Creating Use Accesses in Packages

3. Click the **APPLY VALUES** button to create the use access. Also remember to click the **SAVE** button in the Package Builder to save the changes to the client package.

Even if your development organization is not ready for component-based software development, it is still a good idea to create explicit use accesses so that you can keep track of the dependencies that exist between various packages. To fully appreciate the value of this practice, let's consider an example where knowledge of these dependencies might come in handy.

Let's imagine that an organization is looking to expand its current SAP footprint to include other products in the SAP Business Suite such as SAP CRM or SAP SRM. Because all of these products run on the SAP NetWeaver Application Server, it is quite possible that there might be common development objects that you want to share across the landscape. However, without defined use accesses, it may be very difficult to transport these common objects because it is next to impossible to physically trace all of the possible dependencies that may exist between the individual objects. In this case, just a little bit of preparation up-front can make all the difference in avoiding many unpleasant logistical problems down the road.

7.2.6 Performing Package Checks

The creation of use accesses between packages requires some additional discipline in the software development process. After all, it is much easier to simply start using a development object without having to register its use beforehand. To

save developers from themselves, SAP has provided a *package check* tool that is integrated in the extended syntax check feature of the ABAP Workbench.

The package check tool can be used to assist you in finding places in your code where you are using development objects in other packages without first declaring a use access. By default, the package check tool is switched off in customer systems, so you will need to work with your local Basis administration staff to turn this setting on (see SAP Note 648898 for details on how to configure this setting in your system). The SAP online help documentation walks through several scenarios that demonstrate how the package check works based on various system setting configurations.

7.2.7 Package Design Concepts

There are no hard-and-fast rules for designing packages and package interfaces. For example, there is no law that states you must use structure or main packages to organize your development objects; you can simply organize your development objects into sub-packages. In fact, many projects simply create one big sub-package to store everything. Conversely, it is also possible to overdesign package hierarchies to the point that they are too granular to be of much use.

In his book *UML Distilled* (Addison-Wesley, 2004), Martin Fowler describes three basic principles that you can use to help design your package architectures⁴:

- ▶ The *Common Closure Principle* says that development objects within the same package should be changed for the same reasons.
- ▶ The *Common Reuse Principle* suggests that development objects within a package should all be reused together.
- ▶ The *Static Dependencies Principle* advises you to consider how *stable* your package is if there are many dependencies flowing into it.

For example, if 10 packages are dependent on a single package, it is important for the interface of that package to be stable to avoid widespread rippling affects whenever a change occurs. Here, it is often useful to define the interface of that package in terms of interfaces and abstract classes because they provide the flexibility that is needed to adapt to change.

⁴ Mr. Fowler credits Robert Cecil Martin's *The Principles, Patterns, and Practices of Agile Software Development* (Prentice-Hall, 2003) when describing these principles.

The application of these principles to your package designs should help keep you on track. Also, keep in mind that you are not locked into a particular design if you eventually find that it is not working for your project. Package relationships, just like classes, sometimes require refactoring. Fortunately, the ABAP Workbench makes it easy for you to reassign development objects to other packages.

7.3 UML Tutorial: Package Diagrams

The component design process can become quite involved, being heavily influenced by the subjective whims of developers that often have conflicting design goals. Typically, this process evolves over several iterations that gradually reshape the model to reflect the system that is being implemented. The UML supports the documentation of this design process with the *package diagram*. A package diagram allows you to group related classes and interfaces (and other development objects) into higher-level units called *packages*.

Note that the overlap between the term "package" in the UML and the ABAP packages you learned about in Section 7.2, The Package Concept, is purely coincidental. A UML package is a logical concept that could be implemented in many different ways by various programming languages. However, as you will see, the ABAP package concept aligns very closely with the UML package construct.

Figure 7.14 shows an example of a package diagram for a simple online travel reservation application built using the Web Dynpro ABAP framework. Each of the folder-shaped icons in the diagram are packages. The dotted lines between packages depict *dependencies*. The direction of the line indicates the direction of the dependency. For example, the `Customer UI` and `Travel Agent UI` packages both depend on the `WDA Framework` and `Travel Reservation Model` packages. Similarly, objects within the `Travel Reservation Model` package depend on ABAP Dictionary objects defined within the `Travel Reservation Dictionary` package.

The formal ABAP package names could have been used in this diagram, but as you can see in Figure 7.14, this is optional in the UML. Like many diagrams in the UML, there are not a lot of restrictions in terms of the notation for a package diagram. For example, the package diagram in Figure 7.15 shows some of the classes inside packages `P1` and `P2`. The familiar plus (+) and minus (-) visibility tokens indicate whether or not the classes belong to the public or private interface of the package. In this case, class `B` has been added to the package interface of

package P1. The dependency line drawn between package P2 and P1 could be used to represent a use access (i.e., class C is using class B, etc.).

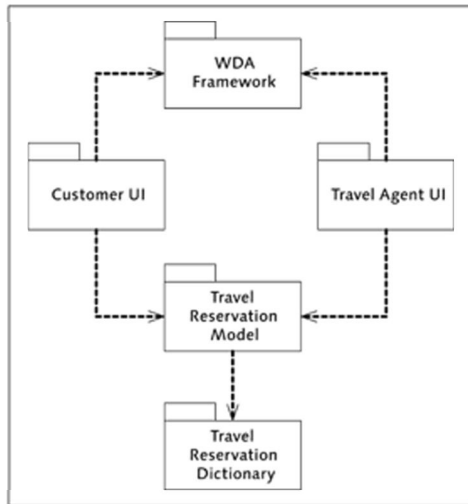


Figure 7.14 Package Diagram for Web Dynpro Application

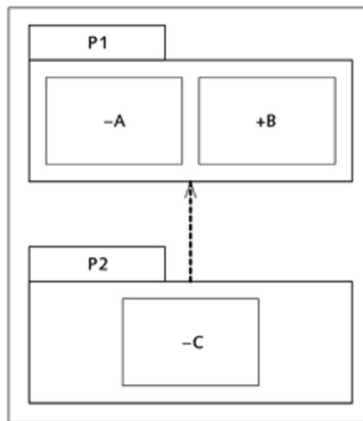


Figure 7.15 Including Objects in a Package Diagram

Package diagrams are very useful in illustrating a system design in terms of its constituent components. If you find that the package diagram for your system looks like a plate of spaghetti, it is likely that your packages are not well encapsulated. Consequently, updating your package diagrams periodically is a good way to gauge the effectiveness of your component designs over time.

7.4 Summary

In this chapter, we looked at how to perform component-based software development in ABAP. As you learned, component-based software development in ABAP is predicated upon the package concept, which goes beyond the visibility concepts of classes to encapsulate development objects together inside of a logical software unit.

In the next chapter, we will explore the ABAP class-based exception handling concept, which enables you to cleanly implement logic to gracefully recover from exception situations that can occur within programs.

Software programs operate in an environment based on rules. However, as the saying goes, there's an exception to every rule. In this chapter, we will explore ways to deal with exception situations that can arise in ABAP Objects programs.

8 Error Handling with Exceptions

No matter how hard we may try to improve the quality of our code, there is simply no way to avoid every type of error that could occur during the execution of an application. In fact, some errors are accidentally introduced by programmers trying too hard to make their applications *error-proof*. Here, for instance, the error-handling logic obscures the main purpose of the program flow, making the code harder to understand and maintain, and thus, more susceptible to errors.

Error-handling logic is a *cross-cutting* concern that becomes tangled within the normal flow of the core application logic. Ideally, you want to de-tangle error-handling logic from the main program flow so that these two orthogonal concerns can be managed separately. In this chapter, you will learn how to apply the ABAP *class-based exception-handling concept* to achieve this kind of separation of concerns.

8.1 Lessons Learned from Prior Approaches

Prior to Release 6.10 of the SAP Web Application Server, there was no comprehensive strategy for dealing with exceptions within the ABAP Objects language. Consequently, developers were forced to improvise, weaving custom exception-handling code into their normal program flow. The code snippet in Listing 8.1 shows an example of some error-handling logic that has been added to a program that is calling a series of subroutines to perform various tasks.

```
DATA: lv_retcode TYPE sy-subrc.  
PERFORM sub_routine1 CHANGING lv_retcode.
```

```

IF lv_retcode NE 0.
    "Error Handling Logic
ENDIF.
PERFORM sub_routine2 CHANGING lv_retcode.
CASE lv_retcode.
    WHEN 0.
        ...
    WHEN 1.
        ...
    WHEN 2.
        ...
    WHEN OTHERS.
        ...
ENDIF.
PERFORM sub_routine3 CHANGING lv_retcode.
IF lv_retcode NE 0.
    "Redundant Error Handling Logic???"
ENDIF.

```

Listing 8.1 Example of a Manual Exception-Handling Approach

In a contrived example such as this, it is not hard to follow what the program is doing. Still, notice that percentage-wise, many more lines of code are devoted to dealing with exceptions than the actual program logic. In larger production programs, this problem becomes even more pronounced.

For the sake of brevity, the details of the exception-handling logic are omitted from the example code shown in Listing 8.1. Nevertheless, common sense tells us that there needs to be logic in here to correct the problem, identify a workaround, or forward the error on to another exception handler or user (e.g., via some kind of message). To do this, we need to have detailed information about the nature of the error. Frequently, a return code such as the one shown in Listing 8.1 (i.e., `lv_retcode`) doesn't tell us everything we need to know. In this case, we may need to enhance the interfaces of our subroutines, and so on, to include additional data objects that collect more details about the errors that may occur within the module.

However, this makes the process all the more cumbersome, especially if those modules call other modules that don't share the same interface. For example, most BAPI function modules return an error message table parameter that has the line type `BAPIRET2`. Internally, these BAPIs frequently call other standard function modules or subroutines that do not maintain message table parameters of this

type. Consequently, additional code has to be written in the BAPI function to translate between the various message table types. In Section 8.5, *Creating Exception Classes*, you will see how to develop exception classes that encapsulate these details more efficiently.

Another problem with ad-hoc exception handling strategies is the fact that it can be very difficult to identify the types of errors that can occur within a given module without having to first dig into the code. For instance, in Listing 8.1, how do you determine the potential problems that you need to account for when you call `sub_routine1`, and so on? From a design perspective, you want the interface of your modules to be more explicit about the types of errors that can occur within them.

After all, exceptions are part of the API contract for a module, too. To some degree, certain previous concepts provide support for this requirement. For example, you can create named exceptions for function modules using the `EXCEPTIONS` addition. However, these exceptions are essentially static error codes that have been assigned some semantic meaning inside the function module. The meaning of these exceptions tends to become obscured outside of the scope of the function module, especially when new exceptions are added into the mix.

Recognizing this, SAP decided to implement a new class-based concept for dealing with exceptions that could be used consistently in all ABAP contexts (i.e., programs, processing blocks, etc.). You will learn more about this concept in Section 8.2, *The Class-Based Exception Handling Concept*.

8.2 The Class-Based Exception Handling Concept

As the name suggests, the class-based exception handling concept uses special classes called *exception classes* to encapsulate exception situations that can occur within a program. These classes are integrated into a framework that makes it easier for you to separate the exception-handling aspects of a program from the core functional aspects of the program. This framework is orchestrated by the `TRY` control structure whose form is shown in Listing 8.2.

```
TRY.
    *Application coding block
CATCH cx_exception_type
```

```

    "Exception handler block
CATCH cx_...
    "Exception handler block
CLEANUP.
    "Cleanup block
ENDTRY.

```

Listing 8.2 Basic Form of the TRY...ENDTRY Control Structure

The TRY statement separates the normal application flow from the exception-handling flow(s) by creating separate execution blocks. The TRY block contains the normal application code that may trigger various types of exceptions. These exceptions are handled by special exception handler blocks called CATCH blocks that contain code that is used to recover from a particular exception situation in some application-specific kind of way. You also have the option of adding a special CLEANUP block to do any sort of cleanup work that might need to be done before the TRY statement returns control to the normal program flow. This basic flow of a TRY statement is depicted in the diagram shown in Figure 8.1.

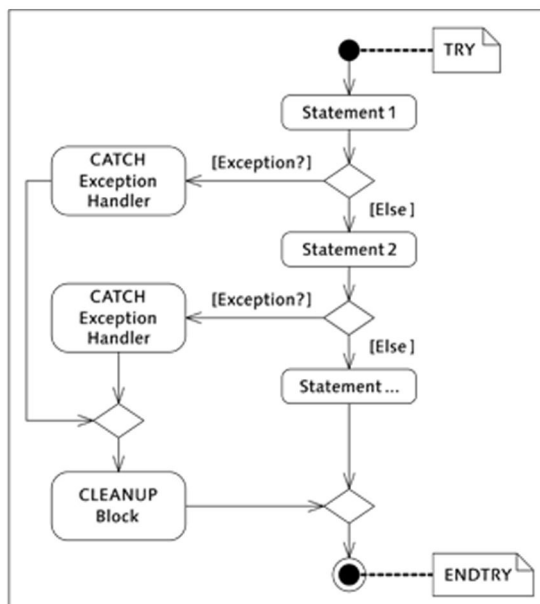


Figure 8.1 Activity Diagram for TRY...ENDTRY Control Statement

Sometimes, developers see the *class-based* part of all this and assume that these features can only be used in ABAP Objects classes. However, class-based exceptions were designed to be used in all ABAP contexts and should be taken advantage of in procedural subroutines, event blocks, and so on.

8.3 Dealing with Exceptions

Two different types of exceptions can occur during the course of an ABAP program: those that are raised programmatically using the `RAISE EXCEPTION` statement, and those that are raised automatically by the ABAP runtime environment. Exceptions that are raised by the ABAP runtime environment cannot always be handled within the context of a running program. In this case, a runtime error (i.e., a *short dump*) occurs, and the program abends (has an abnormal end). In fact, all unhandled exceptions result in a runtime error because the problem condition was never formally addressed in the program. In this section, we will look at how to use the `CATCH` and `CLEANUP` statements to recover from exception situations that can be handled programmatically.

8.3.1 Handling Exceptions

As you have seen, exceptions are dealt with inside of a `CATCH` block. `CATCH` blocks are designed to handle exceptions of a particular type (or, as you will see, a family of related types). The exception type is defined in terms of an exception class that is part of an inheritance hierarchy based on the common `CX_ROOT` superclass. Many predefined exception classes are available out of the box in any AS ABAP installation. It is also possible to define your own custom exception types (see Section 8.5, *Creating Exception Classes*, for more details).

Whenever an exception of a particular type is raised, the system will search for a corresponding `CATCH` block to handle that exception. To demonstrate how this works, let's consider some example code that handles a class cast exception.

The `TRY` block in Listing 8.3 contains some code that performs an illegal widening cast between the object reference variables `lr_parent` and `lr_child`. At runtime, the ABAP runtime environment will detect the invalid cast and raise an exception of type `CX_SY_MOVE_CAST_ERROR`, which is handled in the corresponding `CATCH` block. The ABAP keyword documentation describes the types of exceptions that

may occur whenever an ABAP statement is executed. To figure out which exception class to use, a search was performed on the keyword `MOVE`, which showed that an exception of type `CX_SY_MOVE_CAST_ERROR` can occur in an assignment statement using object reference variables. The relevant logic needed to recover from the error can be implemented inside the `CATCH` block. The code in Listing 8.3 generates a simple error report on the screen (see Figure 8.2).

```

CLASS lcl_parent DEFINITION.
  PUBLIC SECTION.
    METHODS: a, b.
ENDCLASS.

CLASS lcl_parent IMPLEMENTATION.
  METHOD a.
    ENDMETHOD.

    METHOD b.
    ENDMETHOD.
ENDCLASS.

CLASS lcl_child DEFINITION
  INHERITING FROM lcl_parent.
  PUBLIC SECTION.
    METHODS: c.
ENDCLASS.

CLASS lcl_child IMPLEMENTATION.
  METHOD c.
    ENDMETHOD.
ENDCLASS.

DATA: lr_parent   TYPE REF TO lcl_parent.
      lr_child    TYPE REF TO lcl_child.
      lr_ex       TYPE REF TO cx_sy_move_cast_error.
      lv_progname TYPE syrepid.
      lv_inclname TYPE syrepid.
      lv_line     TYPE i.
      lv_text     TYPE string.
      lv_longtext TYPE string.

```

```

* Attempt a widening cast where the dynamic type of the
* source object reference is not compatible with
* the static type of the target object reference:
TRY.
  CREATE OBJECT lr_parent.
  CREATE OBJECT lr_child.
  MOVE lr_parent ?TO lr_child.
CATCH cx_sy_move_cast_error INTO lr_ex.
* Read information about the exception:
  CALL METHOD lr_ex->get_source_position
    IMPORTING
      program_name = lv_progname
      include_name = lv_inclname
      source_line = lv_line.

  lv_text = lr_ex->get_text( ).
  lv_longtext = lr_ex->get_longtext( ).
  CONDENSE lv_longtext.

* Output an error report about the exception:
WRITE: / 'Error Report'.
ULINE.
WRITE: / 'Program Name:', lv_progname.
WRITE: / 'Include Name:', lv_inclname.
WRITE: / 'Line Number:', lv_line.
WRITE: / 'Short Text:', lv_text.
WRITE: / 'Long Text:', lv_longtext.
ENDTRY.

```

Listing 8.3 Handling a Class Cast Exception Using the TRY Statement



Figure 8.2 Generating an Exception Report Using Exception Objects

The details of the error report generated in the `CATCH` block in Listing 8.3 were obtained via method calls made using the `lr_ex` object reference variable. The `lr_ex` object reference variable is initialized using the optional `INTO` addition of the `CATCH` statement¹. Here, notice that the static type of the `lr_ex` object reference is compatible with the exception type `CX_SY_MOVE_CAST_ERROR`. As you can see, an exception object can be used to obtain quite a bit of useful information about the exception, including a short and long text description of the problem, the line number in the program that triggered the exception, and so on.

You can include as many `CATCH` blocks as you want inside of a `TRY` statement. For example, there might be certain types of exceptions that are raised from methods of class `lc1_parent` or `lc1_child`. Whenever an exception is raised in a `TRY` statement with multiple `CATCH` blocks, the system searches through the `CATCH` blocks to find a suitable handler that can handle an exception of a given type. Sometimes, it may be preferable to use a generic exception type in an exception handler block to handle families of related exceptions. For example, rather than setting up a specific exception handler block to handle exception types such as division by zero and arithmetic overflow (which are defined in exception classes `CX_SY_ZERODIVIDE` and `CX_SY_ARITHMETIC_OVERFLOW`, respectively), you could define a `CATCH` block using the `CX_SY_ARITHMETIC_ERROR` superclass. However, if you use generic exception types in your `CATCH` blocks, they must be declared *after* any `CATCH` blocks that define exception handlers for subordinate classes. If you think about it, this makes sense as the more specific exception handlers would never be reached because the system would first find a matching exception handler for the superordinate class. If all of this seems confusing, don't worry, the compiler will tell you where you've gone wrong.

A tendency of some developers new to the class-based exception handling concept is to create extremely large `TRY` statements that surround their entire program logic. This is a very poor design practice that minimizes the effectiveness of the exception handlers. A good rule of thumb is to create `TRY` statements that encapsulate a single logical unit of work. Therefore, if you find that you are mixing and matching many different types of exception classes in `CATCH` blocks, it is likely that your `TRY` statement is too large. Smaller `TRY` statements are much easier to follow and trace.

¹ If the `INTO` addition of the `CATCH` statement is not used, an exception object will not be generated to conserve system resources.

Similarly, it is also important to avoid cutting corners when defining `CATCH` blocks by making the exception handlers too generic. For example, as you will see in Section 8.5, Creating Exception Classes, the root of the exception type hierarchy is class `CX_ROOT`. You could create an all-encompassing `CATCH` block using this exception type that would handle all of the possible exceptions that could be handled in an ABAP program (see Listing 8.4). The problem with this approach is that it makes it very difficult to implement custom exception handling logic for specific types of errors. Therefore, it is worth the additional effort to go ahead and configure specific exception handlers to avoid the headaches associated with refactoring the exception handling logic down the road.

```
TRY.
    Statements...
CATCH cx_root INTO lr_ex.
    ...
ENDTRY.
```

Listing 8.4 Example of a Lazy Design Using Generic Exception Types

8.3.2 Cleaning up the Mess

After we have recovered from an exception situation in a `CATCH` block, it is likely that we may need to perform some additional cleanup tasks before we hand control back over to the normal program flow. For example, let's imagine that you are writing some code to output some data to a file. Inside the `TRY` block, you open up a file and start writing records to it. However, at some point, an exception occurs, and processing halts in the `TRY` block before you get a chance to close the file. In this case, you can use the optional `CLEANUP` block to close the file because this block is guaranteed to be called by the ABAP runtime environment before the `TRY` statement is exited. A simplified example of this scenario is shown in Listing 8.5.

```
TRY.
* Open the extract file for output:
  OPEN DATASET lv_file FOR OUTPUT IN TEXT MODE
    ENCODING DEFAULT.

* Transfer the extract records to the file:
  LOOP AT lt_extract INTO ls_record.
    PERFORM sub_format_record CHANGING ls_record.
```

```

        TRANSFER ls_record TO lv_file.
    ENDOLOOP.

    * Close the output file:
    CLOSE DATASET lv_file.
    CATCH cx_sy_file_access_error INTO lr_file_ex.
    * Process I/O errors...
    CATCH lcx_format_error INTO lr_format_ex.
    * Process custom formatting errors...
    CLEANUP.
    * Clean up any used external resources:
    CLOSE DATASET lv_file.
    ENDRY.

    FORM sub_format_record CHANGING ps_record TYPE ...
        RAISING lcx_format_error.
    ...
    RAISE EXCEPTION TYPE lcx_format_error...
    ENDFORM.

```

Listing 8.5 Releasing External Resources Using the CLEANUP Block

Note that the `CLEANUP` block in a `TRY` statement is guaranteed to be called whenever an exception occurs regardless of whether or not the system can actually locate a suitable exception handler in that `TRY` statement. As you will see in Section 8.4, Raising and Forwarding Exceptions, if an exception handler is not found, the exception will be propagated up the call stack as the system continues to search for a valid exception handler. Prior to exiting, the `CLEANUP` block is executed to clean up any local resources used within the context of the current `TRY` statement.

You should only use the `CLEANUP` block for its intended purpose because it does not support statements that are used to alter the control flow of a program such as `RETURN`, `STOP`, and so on.

8.4 Raising and Forwarding Exceptions

Exceptions can be raised by statements in various parts of any program. Sometimes, it is best to deal with these exceptions close to the point where they are triggered; other times, it makes sense to forward these exceptions on to another part of the program that is better suited to deal with the problem. In the follow-

ing sections, we will look closely at how exceptions are triggered and consider options for forwarding these exceptions up the call stack.

8.4.1 System-Driven Exceptions

As you saw in the example from Listing 8.3, it is possible for regular ABAP statements to trigger an exception at runtime. These exceptions are triggered automatically by the ABAP runtime environment. The types of exceptions that can be triggered are predefined in the system, and their names begin with the `CX_SY_` prefix. To find out the possible exceptions that might be triggered for a particular ABAP statement, consult the ABAP keyword documentation.

Most of the time, exceptions raised by the runtime environment are an indication of some kind of error in your program logic. For example, an exception of type `CX_SY_ZERODIVIDE` probably reveals a place in your code where you failed to check the value of a divisor before performing a division operation. Consequently, runtime exceptions such as these are said to be *unchecked exceptions*. Here, it may or may not make sense to implement a `CATCH` block to handle these errors because they should never occur in a valid program. You will learn more about checked and unchecked exception types in Section 8.5.1, Understanding Exception Class Types.

8.4.2 The RAISE EXCEPTION Statement

Most of the time, you shouldn't have to worry too much about system-driven exceptions after your code has been thoroughly tested. However, this doesn't mean that exception situations will not occur in your programs. For example, imagine that you are tasked with writing a utility method to look up a customer's credit rating. The input to this method is the customer's ID number; the output is the customer's credit score. A simple scaffolding of this method is provided in Listing 8.6.

```
CLASS lcl_customer DEFINITION.
  PUBLIC SECTION.
    METHODS:
      get_credit_rating IMPORTING im_customer_id
                       TYPE bu_partner
                       RETURNING VALUE(re_rating)
                       TYPE i.
```

```

ENDCLASS.

CLASS lcl_customer IMPLEMENTATION.
    METHOD get_credit_rating.
*       Read the customer master record from the database:
        SELECT SINGLE *
            INTO ...
            FROM but000
            WHERE partner EQ im_customer_id.

        IF sy-subrc NE 0.
            re_rating = -1.
            RETURN.
        ENDIF.
        ...
    ENDMETHOD.
ENDCLASS.

```

Listing 8.6 Handling Errors in the Application Logic – Part 1

The first step in this method is to look up the customer master record using the customer number provided in the importing parameter `im_customer_id`. However, look at what happens whenever the provided customer number is invalid. Because it is not possible for the method logic to continue without a valid customer number, a dummy credit rating (i.e., -1) is returned to the caller. Here, it is assumed that the caller of the method will check the value of the credit rating to determine if it was valid.

The exception-handling technique shown in Listing 8.6 is not very intuitive. For example, let's assume that the customer's credit score is calculated based on the popular FICO® credit score used within the United States. Based on this assumption, users of method `get_credit_rating` should expect to receive valid credit scores in the range of 300 to 850. Passing back undefined values such as -1 can be dangerous because unwitting developers may accidentally use this value incorrectly. From a design perspective, it is better to throw up a red flag so that you can bring this exception situation to the attention of the caller. This can be achieved using the `RAISE EXCEPTION` statement.

The `RAISE EXCEPTION` statement is used to explicitly trigger an exception of a particular type. The basic syntax for the `RAISE EXCEPTION` statement is given in Listing 8.7.

```

PUBLIC SECTION.
METHODS:
    get_credit_rating IMPORTING im_customer_id
                      TYPE bu_partner
                      RETURNING VALUE(re_rating)
                      TYPE i.
ENDCLASS.

CLASS lcl_customer IMPLEMENTATION.
METHOD get_credit_rating.
*   Read the customer master record from the database:
    SELECT SINGLE *
      INTO ...
      FROM but000
      WHERE partner EQ im_customer_id.

    IF sy-subrc NE 0.
      RAISE EXCEPTION TYPE lcx_customer_not_found.
    ENDIF.
    ...
ENDMETHOD.
ENDCLASS.

```

Listing 8.9 Handling Errors in the Application Logic — Part 2

If you try to compile the code in Listing 8.9, you will receive a warning during the syntax check that indicates that the exception type `lcx_customer_not_found` is not caught or declared in the `RAISING` clause of the `get_credit_rating` method. You have two options for dealing with exceptions that are triggered using the `RAISE EXCEPTION` statement:

- ▶ You can deal with it directly via a `TRY` statement.
- ▶ If you are in a procedure (i.e., a method, a function module, or a subroutine), you can forward the exception on to the caller of that procedure².

You have already seen how to handle exceptions in a `TRY` statement, and you will learn how to forward exceptions in Section 8.4.3, Propagating Exceptions.

² The exceptions to this rule are class constructors and event handler methods. It is not reasonable to forward exceptions from these methods because they are called implicitly by the ABAP runtime environment.

8.4.3 Propagating Exceptions

As you have seen, there are times when it simply doesn't make sense for a procedure to deal with an exception directly. For example, the logic in the method `get_credit_rating` from Listing 8.9 is predicated on the fact that a valid customer is provided in the importing `im_customer_id` parameter. Therefore, if an invalid customer ID is passed into the method, it makes no sense for the method to continue because it does not have the necessary information to do its job.

Rather than trying to force the issue, it is better for the method to explicitly trigger an exception so that the caller of the method can deal with the problem. Of course, a caller can only respond to exceptions that it knows about. Such information is revealed in the signature of a procedure using the `RAISING` addition.

Listing 8.10 shows how the `RAISING` addition was added to the signature of method `get_credit_rating`. This addition ensures that callers of this method know that they need to implement logic to account for exceptions of type `lcx_customer_not_found`.

```

CLASS lcl_customer DEFINITION.
  PUBLIC SECTION.
    METHODS:
      get_credit_rating IMPORTING im_customer_id
                        TYPE bu_partner
                        RETURNING VALUE(re_rating)
                        TYPE i
                        RAISING lcx_customer_not_found.
ENDCLASS.

```

Listing 8.10 Defining a Method Using the `RAISING` Addition

The basic form of the `RAISING` addition for methods is given in Listing 8.11. The same syntax pattern also applies to subroutines and function modules.

```

METHOD some_method RAISING cx_... cx_...
ENDMETHOD.

```

Listing 8.11 Basic Syntax of the `RAISING` Addition

It is important to be careful not to abuse the use of the `RAISING` addition by cluttering a method signature with every possible exception type that might occur within a method. For example, let's imagine that the `get_credit_rating` method

was calling a Web Service to obtain the customer's credit rating. In this case, there could be many types of exceptions that could occur during the course of the lookup process (e.g., network connectivity problems, etc.). However, these exception types represent internal implementation details that should be hidden from users. One way to deal with these exceptions is to wrap them inside of another more generic exception object. This can be achieved using the `PREVIOUS` importing parameter of an exception type's constructor.

In Listing 8.12, the `get_credit_rating` method has been redefined to raise an exception of type `lcx_lookup_failed`. Internally, the credit score lookup is now being driven by a Web Service call brokered through SAP NetWeaver Process Integration (SAP NetWeaver PI), the open integration and application platform product offered by SAP. The details of this Web Service are omitted here for the sake of brevity. Still, as you can see, this Web Service call can raise exceptions of type `CX_AI_SYSTEM_FAULT` or `CX_AI_APPLICATION_FAULT`, respectively. Rather than declare these technical exception types explicitly within the method signature, the method implementation wraps the exception object inside of a more user-friendly exception object of type `lcx_lookup_failed` at runtime using the `PREVIOUS` parameter. This technique shields users from the various internal exceptions that might be triggered in the lookup process. Of course, users still have the option of digging into the details of these triggering exception types via the public attribute `PREVIOUS`, which is defined using the generic type `CX_ROOT`.

```

CLASS lcx_lookup_failed DEFINITION
    INHERITING FROM cx_static_check.
ENDCLASS.

CLASS lcl_customer DEFINITION.
    PUBLIC SECTION.
    METHODS:
        get_credit_rating IMPORTING im_customer_id
                           TYPE bu_partner
                           RETURNING VALUE(re_rating)
                           TYPE i
                           RAISING lcx_lookup_failed.
ENDCLASS.

CLASS lcl_customer IMPLEMENTATION.
    METHOD get_credit_rating.

```

```

*   Method-Local Data Declarations:
    DATA: lr_sys_ex TYPE REF TO cx_ai_system_fault,
          lr_app_ex TYPE REF TO cx_ai_application_fault.

    TRY.
      CALL ...
    CATCH cx_ai_system_fault INTO lr_sys_ex.
      RAISE EXCEPTION TYPE lcx_lookup_failed
      EXPORTING
        previous = lr_sys_ex.
    CATCH cx_ai_application_fault INTO lr_app_ex.
      RAISE EXCEPTION TYPE lcx_lookup_failed
      EXPORTING
        previous = lr_app_ex.
    ENDTRY.
  ENDMETHOD.
ENDCLASS.

```

Listing 8.12 Raising an Exception with an Existing Exception Object

You can also declare nonclass-based exceptions using the `EXCEPTIONS` addition. For example, we could have defined method `get_credit_rating` using the syntax shown in Listing 8.13.

```

CLASS lcl_customer DEFINITION.
  PUBLIC SECTION.
    METHODS:
      get_credit_rating IMPORTING im_customer_id
                       TYPE bu_partner
                       RETURNING VALUE(re_rating)
                       TYPE i
                       EXCEPTIONS customer_not_found.
ENDCLASS.

```

Listing 8.13 Declaring Nonclass-Based Exceptions for Methods

However, as you saw in Section 8.1, *Lessons Learned from Prior Approaches*, exceptions such as `customer_not_found` are nothing more than named error codes. Therefore, it is highly recommended that you avoid using nonclass-based exceptions when defining your methods.

To specify specific exception types that are raised in methods of global classes, you must explicitly select an indicator that declares the use of class-based exceptions as opposed to the default nonclass-based exceptions.

1. To declare the exceptions that are raised in methods for global classes, open up the class in the Class Builder, place your cursor on the name of the method you want to edit, and click the EXCEPTIONS button (see Figure 8.3).

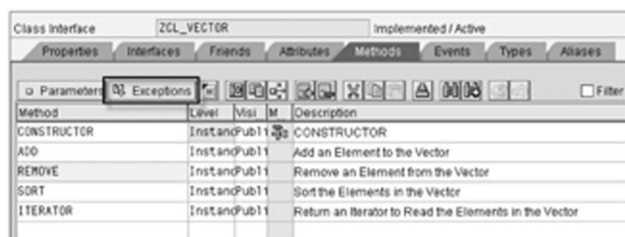


Figure 8.3 Opening up the Method Exceptions Editor

2. In the METHOD EXCEPTIONS editor screen, click on the EXCEPTION CLASSES check-box so that you can change the mode of the editor to support the configuration of class-based exceptions (see Figure 8.4).



Figure 8.4 Turning on Class-Based Exceptions for a Global Method

3. Enter the name of the exception class in the EXCEPTION column of the exceptions table for the method (see Figure 8.5).

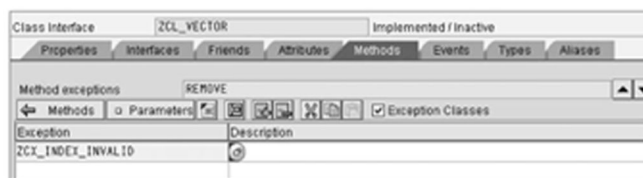


Figure 8.5 Adding Exceptions to Global Methods

In Section 8.5.1, Understanding Exception Class Types, you will see that you do not always have to declare all of the exception types that are triggered in a proce-

ture using the `RAISING` addition. Of course, if the triggered exceptions are not handled somewhere, a runtime error will occur. Consequently, exceptions cannot be propagated from processing blocks that do not have a local data area such as a subroutine, function module, or method.

For example, if you were to raise an exception inside the `START-OF-SELECTION` event of an executable report program, there would be no way to implement an exception handler to deal with that exception. This rule is enforced by the compiler to protect you from implementing code that is guaranteed to generate a runtime error. The same rule also applies to class constructors and event handler methods that are called implicitly by the ABAP runtime environment.

8.5 Creating Exception Classes

Whenever an exception situation occurs, you want to be able to collect as much information as you can so that the designated exception handler has all of the data it needs to properly react and (hopefully) recover from the error gracefully. In the class-based exception handling concept, this information is captured by an object that is an instance of a class that inherits from `CX_STATIC_CHECK`, `CX_DYNAMIC_CHECK`, or `CX_NO_CHECK`. Each of these three abstract classes are subclasses of the root exception class `CX_ROOT` (see Figure 8.6).

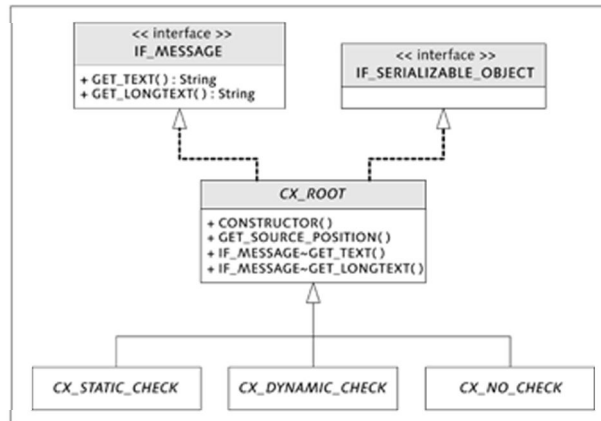


Figure 8.6 Class Diagram for `CX_ROOT` Inheritance Tree

As you can see in Figure 8.6, class `CX_ROOT` also implements the `IF_SERIALIZABLE_OBJECT` and `IF_MESSAGE` interfaces. The `IF_SERIALIZABLE_OBJECT` interface makes it possible to *serialize* (or compress) an object onto a file or network stream, and so on. The `IF_MESSAGE` interface defines methods for extracting a short and long text description of the error message. Section 8.5.4, Defining Exception Texts, includes more about these methods.

8.5.1 Understanding Exception Class Types

Each of the three subclass types shown in Figure 8.6 are treated differently by the class-based exception handling framework. When building a custom exception class, it is important to understand the differences between these base types because it impacts how they are used in your programs. These differences are outlined in Table 8.1.

Exception Class	Usage Type
<code>CX_STATIC_CHECK</code>	Exceptions of this type are used to represent checked error conditions that may occur within the logic of an application program. Such exceptions must either be explicitly declared in a procedure's interface using the <code>RAISING</code> addition or handled locally within a <code>TRY</code> statement. If an exception of this type is not properly handled, the compiler will issue a warning during the syntax check.
<code>CX_DYNAMIC_CHECK</code>	Exceptions of this type are used to represent unchecked error conditions that likely stem from errors in the program logic. For example, the standard exception class <code>CX_SY_ZERODIVIDE</code> is used to represent a situation where a division operation was attempted with a divisor whose value is 0. Realistically, this kind of error should not happen, and if it does, it might not be possible to recover from it gracefully. Moreover, because a mathematics-intensive program could produce this kind of error in almost every statement, it is not practical to handle all of the possible exceptions that might occur. Therefore, exceptions of this type do not have to be explicitly handled and are not subject to static syntax checks at compilation time. Of course, failure to properly handle such an exception will ultimately result in a runtime error.

Table 8.1 Differences Between Base-Level Exception Types

Exception Class	Usage Type
CX_NO_CHECK	Exceptions of this type are similar to ones deriving from CX_DYNAMIC_CHECK. The primary difference is that these kinds of exceptions are automatically forwarded if they are not explicitly handled locally in a TRY statement. In other words, the RAISING clause of a method, subroutine, and so on implicitly contains the CX_NO_CHECK addition in its signature, so it is not possible to add additional subordinate classes of this type to the signature of a procedure.

Table 8.1 Differences Between Base-Level Exception Types (cont.)

Most of the time, you will want to define custom exception types based on the CX_STATIC_CHECK superclass. By using this base type, you are enlisting the aid of the compiler in ensuring that exceptions are being dealt with correctly by users. This practice also makes sure that potential exception situations are adequately documented in the procedure's interface.

8.5.2 Local Exception Classes

Specific exceptions unique to a particular application can be defined locally in the same way that we have defined various other local classes throughout the course of this book. Listing 8.14 demonstrates the definition of a local class called lcx_local_exception.

```
CLASS lcx_local_exception DEFINITION
  INHERITING FROM cx_static_check.
ENDCLASS.
```

Listing 8.14 Defining a Local Exception Class

Local exception classes can include the definition of a constructor and various attributes. However, SAP recommends that you do not define additional methods or redefine inherited methods in local exception classes.

8.5.3 Global Exception Classes

Most of the time, whenever you define an exception class, you will prefer to create it globally in the ABAP Repository so that it can be reused in other programs/contexts. Global exception classes, like other global classes, are defined using the

Class Builder tool, which adjusts to the EXCEPTION BUILDER perspective whenever you are editing an exception class.

As you can see in Figure 8.7, the CREATE CLASS dialog box looks a bit different whenever the EXCEPTION CLASS type is selected. Here, you enter a name for the exception class, the superclass (which must be one of the three base exception classes CX_STATIC_CHECK, CX_DYNAMIC_CHECK, CX_NO_CHECK, or a subclass of those types), as well as some familiar fields that have been used to define other global classes. The WITH MESSAGE CLASS checkbox is used to include support for the integration of messages defined within a message class (i.e., in Transaction SE91). We will discuss this option more in Section 8.5.5, Mapping Exception Texts to Message IDs.



Figure 8.7 Creating a Global Exception Class in the Exception Builder

Exception classes must be named using the prefix CX_; the compiler will complain if you try to name it otherwise. Of course, as you can see in Figure 8.7, exception classes in the customer namespace can be defined as ZCX_, and so on. For the most part, the Exception Builder perspective looks almost identical to the normal Class Builder perspective.

For example, in Figure 8.8, you can see that a custom class called ZCX_CUSTOMER_NOT_FOUND contains three methods inherited from the base exception class CX_ROOT. In addition, a default constructor method has also been created. However, if you attempt to edit the constructor, you will notice that the Exception Builder will not allow it. This is by design because the generated constructor

method contains the proper interface and implementation to ensure that exceptions are always initialized correctly.

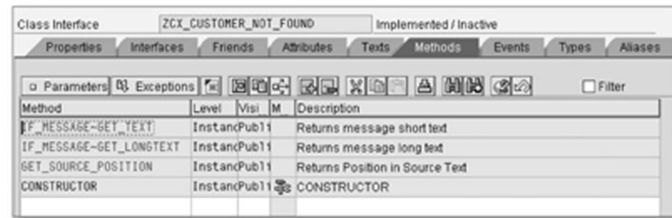


Figure 8.8 Initial View of the Exception Builder

In particular, this initialization process makes sure that text descriptions and previous exceptions are stored in context. This information is stored in the instance attributes `TEXTID` and `PREVIOUS`, respectively, which are inherited from `CX_ROOT`. In the following Section 8.5.4, Defining Exception Texts, you will see how to define the exception texts that are stored in the `TEXTID` attribute.

8.5.4 Defining Exception Texts

Ideally, whenever an exception occurs, we want to recover from it gracefully using logic defined within an exception handler block. Unfortunately, this is not always possible for every type of exception. Unexpected exception situations typically require human intervention of some kind. Sometimes, this intervention comes in the form of an error message displayed on a screen; other times, a message is written to an error log. In either case, we need to be able to produce a meaningful descriptive text for someone to be able to investigate the problem. The Exception Builder tool supports you in this endeavor by allowing you to configure *exception texts* for global classes.

Exception texts are defined on the `TEXTS` tab of the Exception Builder (see Figure 8.9). However, the actual text is stored in the *Online Text Repository (OTR)*. The OTR is a central storage repository for texts that are defined within the AS ABAP, providing support for internationalization, and so on.

Within the Exception Builder, each exception text is defined using a unique exception text ID (i.e., `READ_ERROR` in Figure 8.10). The exception text ID correlates to a constant attribute with the same name that has the data type `SOTR_CONC`.

Exception ID	Text
CX_ROOT	An exception occurred
CX_SY_FILE_ACCESS_ERROR	Error accessing the file '&FILENAME&'
CX_SY_FILE_IO	Error with input/output to file '&FILENAME&'. Operating system error: &ERRORCODE& (&ERRORTXT&).
READ_ERROR	Could not read the file '&FILENAME&'; operating system error: &ERRORCODE& (&ERRORTXT&)
WRITE_ERROR	The file '&FILENAME&' could not be written. Operating system error: &ERRORCODE& (&ERRORTXT&).

Figure 8.9 Defining Exception Texts in the Exception Builder

These constant attributes belong to the same namespace as normal attributes, so it is a good idea to use the standard naming convention for constants (i.e., the `CO_` prefix) when defining exception text IDs in the Exception Builder.

Attribute	Level	Visi	Re	Typing	Associated Type	Description	Initial value
CX_ROOT	Constan	Public	<input type="checkbox"/>	Type	SOTR_CONC	Exception ID: Value for Attr	'16A9A3937A9BB56E10000000A11447B'
TEXTID	Instance	Public	<input checked="" type="checkbox"/>	Type	SOTR_CONC	Key for Access to Message	
PREVIOUS	Instance	Public	<input checked="" type="checkbox"/>	Type Ref	CX_ROOT	Exception Mapped to the C	
KERNEL_ERRID	Instance	Public	<input checked="" type="checkbox"/>	Type	S388ERRID	Internal Name of Exception	
CX_SY_FILE_ACCESS_ERF	Constan	Public	<input type="checkbox"/>	Type	SOTR_CONC	Exception ID: Value for Attr	'4182174D0303006300000000A1551B1'
FILENAME	Instance	Public	<input checked="" type="checkbox"/>	Type	STRING	Name of File that Caused	
CX_SY_FILE_IO	Constan	Public	<input type="checkbox"/>	Type	SOTR_CONC	Exception ID: Value for Attr	'D50BB8396F051547E10000000A11447B'
READ_ERROR	Constan	Public	<input type="checkbox"/>	Type	SOTR_CONC	Exception ID: Value for Attr	'E30BB8396F051547E10000000A11447B'
WRITE_ERROR	Constan	Public	<input type="checkbox"/>	Type	SOTR_CONC	Exception ID: Value for Attr	'F30BB8396F051547E10000000A11447B'
ERRORCODE	Instance	Public	<input checked="" type="checkbox"/>	Type	I	Error Number of Operating	
ERRORTXT	Instance	Public	<input checked="" type="checkbox"/>	Type	STRING	Error Text of Operating Sys	

Figure 8.10 Reading the OTR Key in the Exception Builder

If you look carefully, you will notice that each constant attribute defined in relation to an exception text ID is initialized with a hexadecimal string value. This value is the globally unique key of the corresponding text object in the OTR. For example, in Figure 8.10, the OTR key for the `READ_ERROR` exception text is highlighted.

You can see the corresponding OTR text by opening up Transaction `SOTR_EDIT`. Here, you can plug in the OTR key in the `CONCEPT` field, select the desired LAN-

GUAGE, and click the DISPLAY button (see Figure 8.11). Figure 8.12 shows the CHANGE CONCEPT screen of the OTR for the selected OTR key.

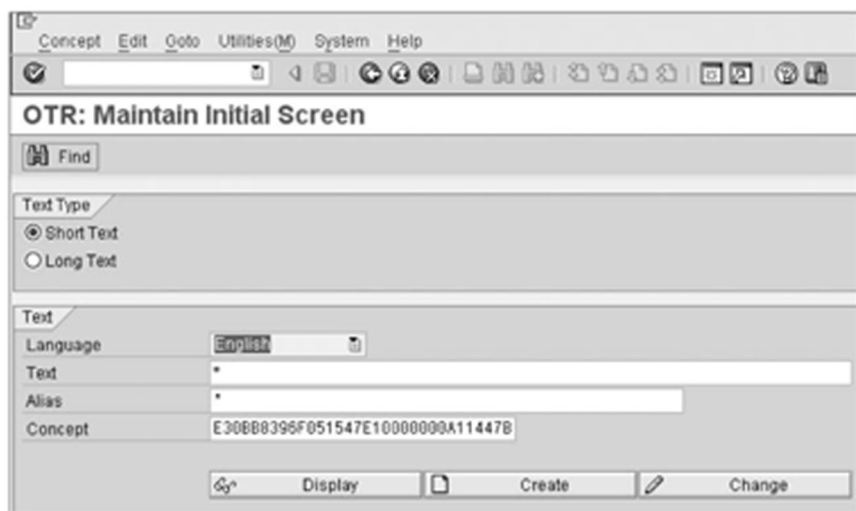


Figure 8.11 Looking up Texts in the OTR – Part 1

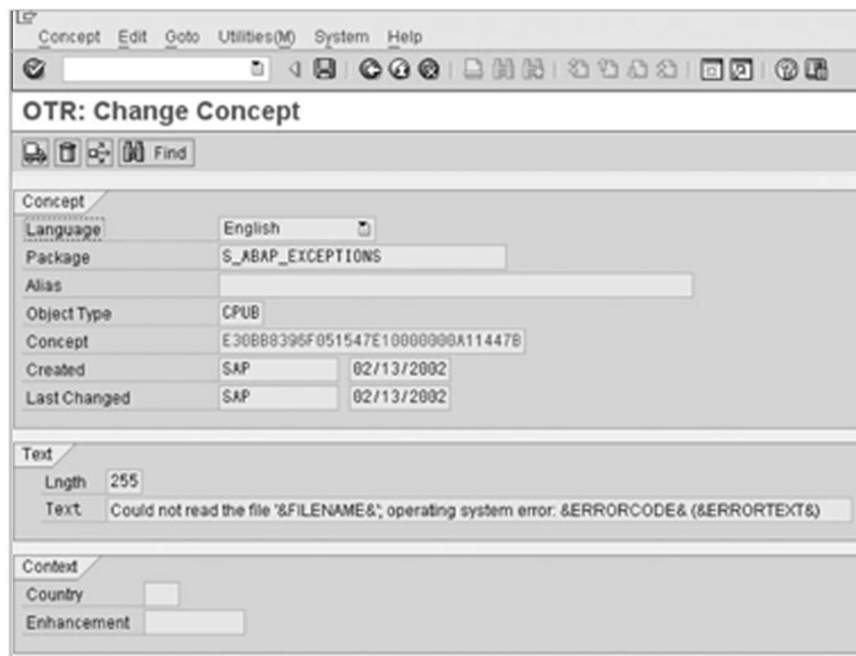


Figure 8.12 Looking up Texts in the OTR – Part 2

You can define text parameters in your exception texts by surrounding elementary attribute names between ampersands. For example, the exception text `READ_ERROR` from exception class `CX_SY_FILE_IO` shown in Figure 8.9 contains three text parameters: `&FILENAME&`, `&ERRORCODE&`, and `&ERRORTXT&`. At runtime, method `GET_TEXT`, which is defined in interface `IF_MESSAGE` and whose implementation is inherited through class `CX_ROOT`, will replace these text parameter tags with the values of the corresponding instance attributes defined in class `CX_SY_FILE_IO` to produce a message text that is more meaningful to the end user.

For exception classes to be able to support message parameters, there must be a way to pass data to the exception object. However, if you recall from Section 8.5.3, Global Exception Classes, you are not allowed to edit the constructor of a global class. In spite of this, it is possible to supplement the interface of the constructor by defining public instance attributes. The Class Builder will add importing parameters using the same name and typing information used to define the attribute. In addition, it will also augment the implementation of the constructor method to include assignment statements to store the importing parameter within the corresponding attribute. This allows you to create an exception object with all of the relevant parameters needed to accurately define the exception situation.

For example, the code in Listing 8.15 shows how an exception of type `CX_SY_FILE_IO` can be raised whenever a read error occurs. In this case, the optional `FILENAME` parameter was used to indicate the name of the file that could not be read.

```
DATA: lr_ex TYPE REF TO cx_sy_file_io,
      lv_msg TYPE string.
TRY.
  ...
  RAISE EXCPETION TYPE cx_sy_file_io
  EXPORTING
    textid = cx_sy_file_io=>read_error
    filename = 'somefile.txt'.
CATCH cx_sy_file_io INTO lr_ex.
  lv_msg = lr_ex->get_text( ).
  MESSAGE lv_msg TYPE 'I'.
ENDTRY.
```

Listing 8.15 Retrieving Explanatory Text from Exception Instances

8.5.5 Mapping Exception Texts to Message IDs

Beginning with the release 6.40 of the SAP NetWeaver Application Server, you can now map exception texts to message IDs within a message class. You can enable this functionality in your exception classes by choosing the **WITH MESSAGE CLASS** option in the **CREATE CLASS** dialog box (as you've already seen in Figure 8.7).

To demonstrate how to map exception texts to message IDs, let's consider an example of an exception class called `ZCX_USER_CREATE_FAILED` that is used to handle exceptions related to the creation of users in a custom user management API.

1. To provide information about the user in question, a public instance attribute called `USER_NAME` has been provided that will be added to the parameter interface of the `CONSTRUCTOR` method (see Figure 8.13).

Attribute	Level	Visi	Re	Typing	Associated Type	Description	Initial value
IF_T100_MESSAGE~DEFAULT	Constan	Public	<input type="checkbox"/>				
IF_T100_MESSAGE~T100	Instance	Public	<input type="checkbox"/>	Type	SCX_T100KEY	T100 Key with Parameters	
CX_ROOT	Constan	Public	<input type="checkbox"/>	Type	S0TR_CONC	Exception ID: Value for Attri	16AA9A3937
TEXTID	Instance	Public	<input checked="" type="checkbox"/>	Type	S0TR_CONC	Key for Access to Message	
PREVIOUS	Instance	Public	<input checked="" type="checkbox"/>	Type Ref	CX_ROOT	Exception Mapped to the C	
KERNEL_ERRID	Instance	Public	<input checked="" type="checkbox"/>	Type	S380ERRID	Internal Name of Exceptio	
CO_DUPLICATE_USER	Constan	Public	<input type="checkbox"/>	Type	S0TR_CONC		
USER_NAME	Instance	Public	<input type="checkbox"/>	Type	SYUNAME	User Name	
			<input type="checkbox"/>	Type			

Figure 8.13 Defining Parameters in `ZCX_USER_CREATE_FAILED`

2. If you look carefully on the **INTERFACES** tab of the Exception Builder, you will notice that this exception class is implementing interface `IF_T100_MESSAGE`. `IF_T100_MESSAGE` is a nested interface that embeds the `IF_MESSAGE` interface that you have normally seen used with exception classes. The `IF_T100_MESSAGE` interface also defines some attributes that support the mapping of exception texts to message IDs. These details are encapsulated inside the implementations of methods `IF_MESSAGE~GET_TEXT` and `IF_MESSAGE~GET_LONGTEXT` in the root exception class `CX_ROOT`.
3. To map an exception text to a message ID, you can simply create a new exception text and click on the **MESSAGE TEXT** button in the toolbar above the exception text input table. This will bring up the **ASSIGN ATTRIBUTES OF AN EXCEPTION**

CLASS TO A MESSAGE dialog box shown in Figure 8.14. Here, we are assigning message ID 088 from message class 01 to the exception text CO_DUPLICATE_USER.

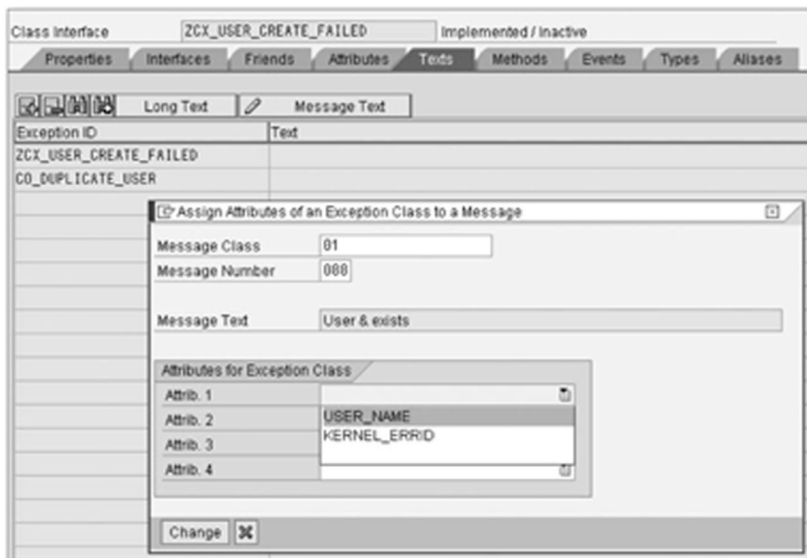


Figure 8.14 Mapping a Message ID to an Exception Text

4. Figure 8.15 shows this message as defined in message maintenance Transaction SE91. As you can see, you also have the option to map elementary attributes from the ZCX_USER_CREATE_FAILED class to the message parameters.

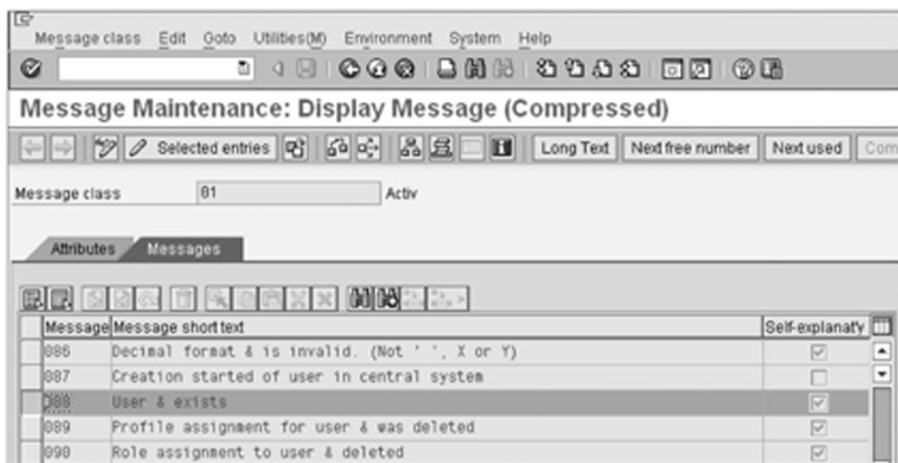


Figure 8.15 Displaying a Message in Message Maintenance

The primary benefit for using the message mapping option with your exception classes is that you can leverage a pre-existing message base that is being maintained across the development landscape. Such messages can be maintained with long text and translated into other languages using the familiar tools provided in Transaction SE91.

Another advantage of using the message mapping option is the fact that the MESSAGE statement in ABAP now provides direct support for exception classes that implement the IF_T100_MESSAGE interface; this functionality was introduced with the SAP NetWeaver 2004 release. The code snippet in Listing 8.16 shows how the MESSAGE statement can be used to output an exception text to the screen. In the past, we would have had to extract the message text into a string variable using the GET_TEXT method before we could display it using the MESSAGE statement.

```
TRY.
  ...
CATCH cx_some_exception INTO lr_ex.
  MESSAGE lr_ex TYPE 'E'.
ENDTRY.
```

Listing 8.16 Displaying Exception Texts with the MESSAGE Statement

8.6 UML Tutorial: Activity Diagrams

The UML activity diagram is a behavioral diagram used to depict the high level flow within a module or program. Activity diagrams are quite similar in nature to flowcharts. However, as you will see, there are certain things you can do with an activity diagram that you cannot do with a flowchart.

Figure 8.16 shows an example of an activity diagram that is being used to depict the flow of a simple ABAP extract program. The flow begins at the *initial node* action and proceeds to the very first action called Receive Query Parameters. Notice that the action names used here are fairly generic. For example, in the ABAP extract program, the Receive Query Parameters action would encompass the generation of a selection screen and the entry of selection parameters by a user. You can trace the control flow of an activity diagram by following the directed edges between actions. Eventually, the program flow will proceed all the way down to the Activity Final action.

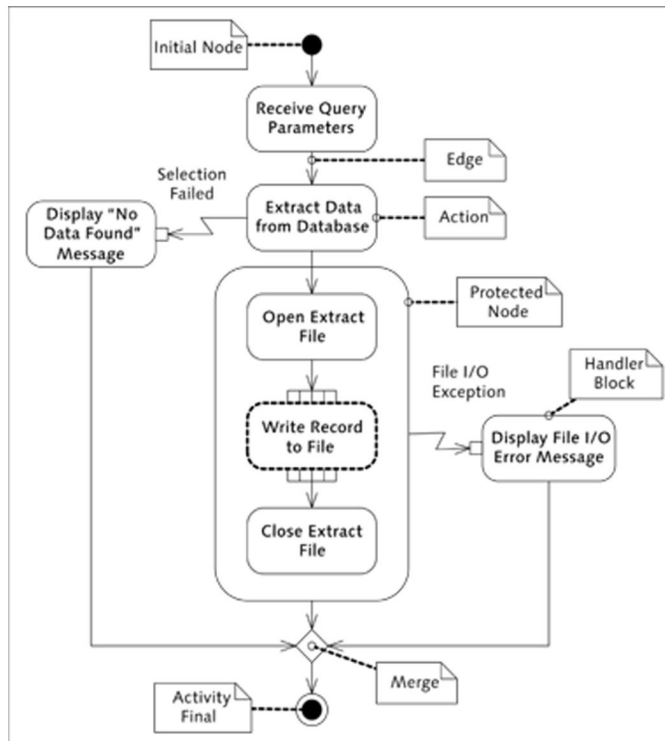


Figure 8.16 Example of a UML Activity Diagram

One significant addition to activity diagrams in the UML 2.0 standard was the specification of *protected nodes* and *handler blocks*. As you can see in Figure 8.16, the action *Extract Data from Database* is a protected node that might trigger an exception (i.e., *Selection Failed*). If no data is found in the database for the given selection criteria, control is transferred to the *Display "No Data Found" Message* handler block. It is also possible to group together multiple actions inside of a protected node. For example, all of the file I/O actions in Figure 8.16 were grouped together in a protected node that reacts to file I/O exceptions.

In the diagram shown in Figure 8.16, notice that flow from each handler block leads into a diamond-shaped node called a *merge*. Merges provide a convenient way of channeling multiple input flows into a common output flow.

If you look carefully at the Write Record to File action in Figure 8.16, you will notice that its boundary is depicted using a dotted line in lieu of the solid line used with all of the other normal actions. This dotted line marks an *expansion region* in the activity diagram. In the flow from Figure 8.16, the Write Record to File expansion region (along with the inputting tokens shown as small pins along the top of the action) represents a loop that takes the extract records from the database lookup and iteratively writes them to the extract file. This kind of notation is much more elegant than the typical use of conditionals in flowcharts to determine if there are more records to process, and so on.

One of the beauties of activity diagrams is that they are extremely easy to read, often requiring little to no translation for nontechnical members of the team. Consequently, they are an excellent communication tool for describing and refining a program flow with functional team members. Typically, after the process flow within an activity diagram is agreed upon, the design can be put in more technical terms in the form of interaction diagrams such as a sequence diagram, and so on. We will discuss some more advanced features of activity diagrams in Chapter 12, Working with XML.

8.7 Summary

The class-based exception handling concept greatly simplifies the process of dealing with errors that can occur within an application. The definition of a common framework for dealing with exceptions is essential for the development of reusable components because it provides a consistent model for propagating exceptions to users. In particular, the `RAISING` addition described in Section 8.4.3, Propagating Exceptions, helps complete the signature of methods by fully specifying a method's API contract.

In the next chapter, we will look at ways to simplify the testing of these methods to ensure that their implementation satisfies the conditions outlined in their API contract.

Unit tests measure the correctness of individual software modules, helping developers identify bugs earlier in the software development lifecycle. Unit testing frameworks such as ABAP Unit assist developers in writing quality unit tests. In this chapter, you will learn how to develop and execute unit tests using the ABAP Unit testing framework.

9 Unit Testing with ABAP Unit

During the implementation phase of an SAP project, ABAP developers find themselves in a fairly consistent routine of coding, compiling, activating, running, and testing their individual development objects. At this stage, the testing process is fairly informal, providing developers with a quick checkpoint to make sure that they are progressing on the right track. Of course, if errors are found, the code is changed on the spot and tested again. After the code reaches a stable state, it is ready for a more formalized *unit test*.

Unit tests verify the correctness of an individual software unit. In terms of ABAP development, this generally refers to a single method in a class, a subroutine, a function module, and so on. If you have ever worked with unit tests before, this view of unit tests might seem a bit unorthodox because the individual unit tests are so narrow in scope. This point of confusion likely stems from the IEEE definition, which broadly describes unit testing as “testing of individual hardware or software units or groups of related units.”¹ As you will see, ABAP Unit supports both views of unit testing by allowing you to flexibly group individual unit tests together during test runs.

One of the main goals of every unit test is to confirm that your individual modules are fulfilling the terms of their API contracts. Verifying this behavior early on helps to eliminate the tedious module-level bugs that prohibit integration and

¹ This explanation of unit testing scope was detailed in the book *JUnit in Action* (Manning, 2004). The actual definition was taken from the *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries* (IEEE, 1990).

functional tests from running smoothly. In this chapter, you will learn how the ABAP Unit test tool can support you in the process of developing and executing unit tests.

9.1 ABAP Unit Overview

ABAP Unit is a testing framework that allows you to build automated tests of individual units of software (e.g., function modules, class methods, etc.). In this section, we will introduce the basic concepts of ABAP Unit in particular and unit testing frameworks in general.

9.1.1 The Need for Unit Testing Frameworks

Frequently, unit tests are conducted *by feel*. For example, module pool applications are often tested by clicking buttons on the screen and seeing what happens. If the screen doesn't blow up, and a record is written to the database (which you can verify in Transaction SE16), then perhaps everything is working correctly. However, more careful inspection might show that certain screen progressions or inputs cause the database record to be written incorrectly, and so on.

In this case, developers often take the test to the next level by enlisting the aid of the ABAP Debugger. Here, they can step through the code line-by-line to verify that inputs are processed correctly, and so on. Occasionally, certain modules are so difficult to test by hand that they require a special testing program to be written. These makeshift programs usually contain logic to automate certain test tasks that are too difficult to perform manually.

Having considered some of the various options for conducting unit tests, let's take a step back and consider some of the problems associated with these kinds of informal testing methods:

- ▶ "Feel tests" are basically just functional tests in disguise. Clearly, unit tests and functional tests have a similar objective: to produce a working product or module.

However, the approach is very different for each type of test. Functional tests are "black box" tests designed by functional analysts that typically have little to

no understanding of the code used to implement the functionality under test. Unit tests, on the other hand, should be "white box" tests designed by developers with intimate knowledge of the code used to implement the module in question. This inside knowledge provides broader test coverage, testing hidden pieces of logic that are often difficult to analyze in a functional test.

- ▶ Each of the ad hoc techniques described earlier requires a fairly significant amount of manual effort to set up the test and execute it. These efforts represent wasted time for developers that could be moving on to new development tasks, projects, and so on.
- ▶ It is difficult to document and interpret the test results. After all, how do you document a debugging session? Often, the documentation effort boils down to the tedious process of capturing screenshots that must be accompanied with step-by-step narrative text.
- ▶ Ad hoc tests are difficult to reproduce because it is too easy to neglect to perform some basic setup task or skip over a step in the overall test sequence.

Recognizing these types of problems, Kent Beck² decided to develop a unit testing framework that could be used to provide some of the common elements necessary for building and running automated unit tests. The first incarnation of this framework created for the Smalltalk language was called SUnit. Since that time, this same testing model (colloquially known as xUnit these days) has been adapted to create testing frameworks for other languages such as Java (JUnit), .NET (NUnit), and with the SAP NetWeaver 2004 release, ABAP Objects. The SAP implementation of the xUnit test framework is called ABAP Unit.

9.1.2 Unit Testing Terminology

To understand how to use ABAP Unit, it is important to define some of the basic terms that are used throughout the framework (see Table 9.1). These terms (and concepts they represent) are largely based on concepts outlined in the core xUnit framework.

² Kent Beck is the creator of *Extreme Programming*, a software engineering methodology that, among other things, advocates test-driven development using automated unit tests.

Term	Description
Test Class	A test class defines an environment for running multiple related unit tests (implemented as test methods).
Test Method	Test methods are special instance methods of a test class that can be invoked to produce test results. In the xUnit framework, a test method represents a single unit test.
Fixture	A fixture defines an environment for running unit tests in the proper context. Fixtures are configured in special callback methods defined within a test class. You can insert code in these methods to obtain and clean up the resources (e.g., file handles, connections, etc.) that are used within the unit test methods.
Test Task	A test task groups test classes together, allowing their methods to be executed together in a single test run.
Test Run	A test run controls the execution of a test task. Test runs produce test results that can be viewed in the ABAP Unit result display.
Assertion	Inside a test method, individual logical tests are made to assess the correctness of a particular piece of functionality. Whenever these tests are completed, the actual results of the test need to be compared with expected results to determine whether or not the test was successful. The ABAP Unit framework supports you in this comparison process by providing a common utility class called <code>CL_AUNIT_ASSERT</code> , which has many useful methods for affirming the correctness of a given logical test.

Table 9.1 Basic ABAP Unit Terminology

9.1.3 Understanding How ABAP Unit Works

The ABAP Unit test framework is tightly integrated into the ABAP Workbench, making it very easy to set up and execute tests for a given ABAP program. From a development perspective, ABAP Unit tests are nothing more than local classes defined within an ABAP program (i.e., an executable [or report] program, module pools, class pools, function groups, etc.). These classes contain special parameterless instance methods that perform the actual tests, verifying their results with utility methods from class `CL_AUNIT_ASSERT`.

Tests can be run individually via various menu options in the ABAP Workbench, or en masse via integration with the Code Inspector tool (Transaction SCI). In

either case, the actual tests are grouped together inside of a test task that is executed in a test run. During the test run, the individual test methods of the test classes are executed separately by a test driver in the ABAP runtime environment.

However, before the test method is called, the runtime environment will search for a special method called `setup` in the test class. If the `setup` method exists, the runtime environment will call it *before* it calls the test method to ensure that the test is set up properly. Similarly, after the test method completes, the runtime environment will execute the `teardown` method if it exists. This important feature of the framework makes sure that each test runs independently, helping you to identify subtle bugs related to unforeseen dependencies between methods, and so on.

The outcomes of the various tests are shown in the ABAP Unit results display. The details shown in the ABAP Unit results display provide information about what went wrong, and where. The details are context-sensitive, allowing you to navigate within the ABAP Workbench to the source of the problem.

9.2 Creating Unit Test Classes

For the most part, you define and implement test classes in the same way that you would build a regular ABAP Objects class. However, test classes and test methods must be defined using the `FOR TESTING` addition. Listing 9.1 shows the basic form of an ABAP Objects test class definition.

```
CLASS lcl_my_test_class DEFINITION   *#AU Risk_Level Harmless
    FOR TESTING.                   *#AU Duration Short
    [...]
ENDCLASS.
```

Listing 9.1 Basic Form of ABAP Unit Test Classes

As we stated previously, you define test classes as local classes in your ABAP program³. At first glance, this might not seem like a very safe practice because you run the risk of transporting some potentially dangerous test code into a produc-

³ As of Release 7.0 of the SAP NetWeaver Application Server, you can also create global test classes. Global test classes must be defined as abstract, are not allowed to have fixtures, and can only be used in local test classes. Given these limitations, we will not investigate the use of global test classes in this book.

tive environment. However, the `FOR TESTING` addition effectively divides the program into two separate parts: testing code and production code. Test code is typically not even generated in production systems (controlled by the AS ABAP profile parameter `abap/test_generation`). And, even if it is, the test framework will not execute an ABAP Unit test run in a production client. So, developers can rest easy knowing that their test code will not wreak havoc in a productive environment.

9.2.1 Test Attributes

When defining a test class, you must specify a couple of attributes that are used by the ABAP runtime environment during the execution of a test. In local classes, these attributes are not traditional attributes in the sense that they are defined using the `DATA` statement. Instead, they are defined as special *pseudo comments* added after the `CLASS` statement.

In Listing 9.1, you can see both pseudo comments added to the definition of local test class `lcl_my_test_class`. The `Risk_Level` attribute describes the effects that the test could have on the system. For example, it is possible that test methods might invoke functionality in the programs under test that could result in changes to the database, and so on. Test classes that introduce this kind of risk can be restricted for execution based on client customizing settings (defined in the Implementation Guide [IMG] or in Transaction `SAUNIT_CLIENT_SETUP`). This way, for example, you can protect a *golden client* from test side effects that could impact other project efforts. The possible values for the `Risk_Level` attribute are shown in Table 9.2.

Risk Level	Potential Side Effects
Critical	Could alter system settings, customizing, and so on.
Dangerous	Could change records in the database.
Harmless	No side effects; the test is innocuous.

Table 9.2 Risk Level Attribute Values

The `Duration` attribute specifies the expected execution duration of a test class. This attribute helps the ABAP runtime environment to know when a test has run too long (perhaps due to an error in the test code such as an infinite loop). The

possible values of this attribute are `Short`, `Medium`, and `Long`, and they have default values of 1 minute, 5 minutes, and an hour, respectively. These default values can also be adjusted in Transaction `SAUNIT_CLIENT_SETUP`.

9.2.2 Test Methods

Test methods are defined as parameterless instance methods in a test class. The signature of these methods also requires the `FOR TESTING` addition (see Listing 9.2).

```
CLASS lcl_my_test_class DEFINITION  *#AU Risk_Level Harmless
                                FOR TESTING. *#AU Duration Short
PRIVATE SECTION.
METHODS:
    test_method1 FOR TESTING.
    test_method2 FOR TESTING.
ENDCLASS.
```

Listing 9.2 Defining Test Methods

Each test method in a test class corresponds to a single unit test. Consequently, a test method should concentrate on testing a single software unit (e.g., a method, a function module, etc.) rather than testing an entire application. It is important to keep unit tests granular so that we can focus in closely on potential bugs that might creep into various parts of the program. Most of the time, the implementation of a test method will simply consist of a single call to a module of the program under test followed by a status check using utility methods defined in class `CL_AUNIT_ASSERT`.

If you look closely at the code in Listing 9.2, you will notice that the test methods have been defined in the private section of the test class. This is by design because test classes implicitly share a friendship relationship with the test driver of the ABAP runtime environment. Consequently, you should prefer to define your test methods in the private or protected sections of your test class.

9.2.3 Managing Fixtures

Test classes group related test methods (i.e., unit tests) together into a logical unit. In these classes, you can also define special fixture methods that help set up and teardown unit tests. These methods have predefined names that are automatically

recognized by the ABAP runtime environment. Each method is defined in the private section of the class and has no parameters. Table 9.3 describes the various types of fixture methods that are supported by the ABAP Unit framework.

Method Name	Usage Type
setup	This instance method is called prior to the invocation of every test method in the test class.
teardown	This instance method is called after every invocation of a test method in the test class.
class_setup	This class method is called once before any test methods are called in the test class.
class_teardown	This class method is invoked after all of the test methods in the test class have been called.

Table 9.3 Fixture Methods and Their Usages

Fixture methods are an excellent place for defining common initialization code that is relevant for all of the test methods in a test class. In particular, the instance methods `setup` and `teardown` provide a useful hook for implementing code that ensures that each test is executed independently using the proper runtime configuration.

9.2.4 Generating Test Classes for Global Classes

You can generate test classes for global classes using the test class generation tool provided with the Class Builder. To access this generation tool, select the menu path `UTILITIES • TEST CLASS GENERATION` in the Class Builder. This brings up the dialog box shown in Figure 9.1. Here, you are presented with options for specifying the unit test class attributes, generating test methods, and so on. You can access the generated test class by selecting `GOTO • LOCAL TEST CLASSES` in the menu path of the Class Editor screen of the Class Builder.

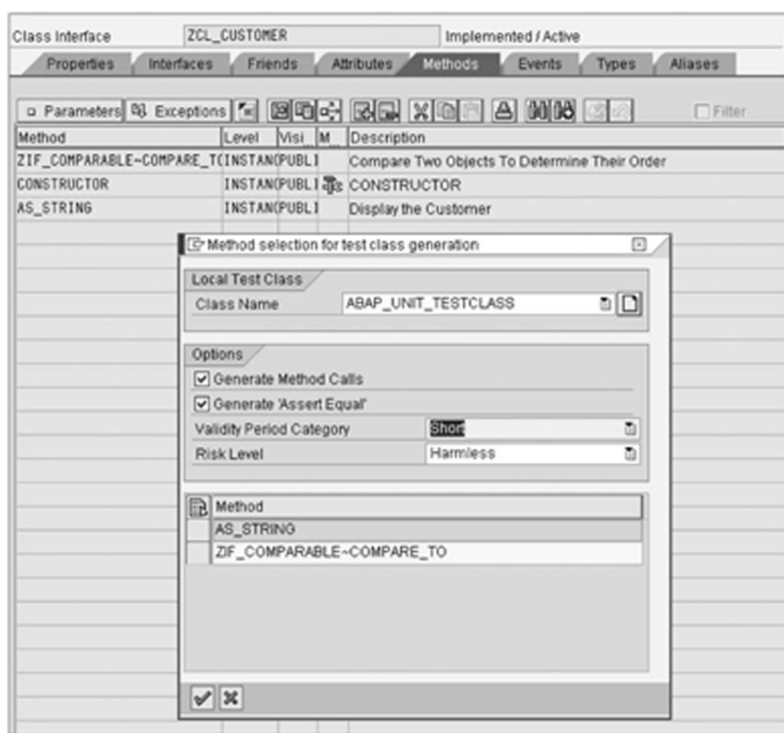


Figure 9.1 Generating a Test Class in the Class Builder

9.3 Case Study: Creating a Unit Test in ABAP Unit

Now that you have a basic understanding of how to create unit test classes, let's consider an example. The program ZUNITTEST in Listing 9.3 defines a simple class called `lcl_account` that is used to represent a bank account. It also defines a testing class called `lcl_account_test` that contains a single test method called `test_transfer` that is used to test the behavior of the `transfer` method defined in class `lcl_account`. The environment for this test method is established in the fixture method `setup`, which instantiates and initializes two account objects called `lr_checking` and `lr_savings`, respectively. Inside the test method, we attempt to transfer \$1,000 from the checking account into the savings account using the `transfer` method. Because this method can throw an exception of type `lcx_insufficient_funds`, this method call is wrapped inside a `TRY` statement. If this type of exception does occur, it would be because the fixture was set up

incorrectly in the `setup` method. In this event, there is no point in proceeding with the test, so you use the `FAIL` method of class `CL_AUNIT_ASSERT` to terminate the test with an error.

```
REPORT zunittest.

CLASS lcx_insufficient_funds DEFINITION
  INHERITING FROM cx_static_check.
ENDCLASS.

CLASS lcl_account DEFINITION.
  PUBLIC SECTION.
  DATA: account_id TYPE numc5 READ-ONLY,
        balance     TYPE bapicurr_d READ-ONLY.
  METHODS:
    constructor IMPORTING im_acct_id TYPE numc5,
    deposit     IMPORTING im_amount TYPE bapicurr_d,
    withdrawal  IMPORTING im_amount TYPE bapicurr_d
                RAISING lcx_insufficient_funds,
    transfer    IMPORTING im_amount TYPE bapicurr_d
                im_to_account TYPE REF
                TO lcl_account
                RAISING lcx_insufficient_funds.
ENDCLASS.

CLASS lcl_account IMPLEMENTATION.
  METHOD constructor.
    account_id = im_acct_id.
  ENDMETHOD.

  METHOD deposit.
    balance = balance + im_amount.
  ENDMETHOD.

  METHOD withdrawal.
    IF balance LT im_amount.
      RAISE EXCEPTION TYPE lcx_insufficient_funds.
    ENDIF.

    balance = balance - im_amount.
  ENDMETHOD.

  METHOD transfer.
```

```

    IF balance LT im_amount.
      RAISE EXCEPTION TYPE lcx_insufficient_funds.
    ENDIF.

    im_to_account->deposit( im_amount ).
  ENDMETHOD.
ENDCLASS.

CLASS lcl_account_test DEFINITION   *#AU Risk_Level Harmless
  FOR TESTING. *#AU Duration Short
  PRIVATE SECTION.
    DATA: lr_checking TYPE REF TO lcl_account,
          lr_savings  TYPE REF TO lcl_account.

    METHODS:
      setup,
      test_transfer FOR TESTING,
      teardown.
ENDCLASS.

CLASS lcl_account_test IMPLEMENTATION.
  METHOD setup.
    CREATE OBJECT lr_checking
      EXPORTING
        im_acct_id = '10000'.

    CREATE OBJECT lr_savings
      EXPORTING
        im_acct_id = '20000'.

    lr_checking->deposit( '1500.00' ).
    lr_savings->deposit( '500.00' ).
  ENDMETHOD.

  METHOD test_transfer.
    TRY.
      CALL METHOD lr_checking->transfer
        EXPORTING
          im_amount      = '1000.00'
          im_to_account = lr_savings.
    CATCH lcx_insufficient_funds.
      CALL METHOD cl_aunit_assert=>fail

```

```

        EXPORTING
            msg = 'Insufficient funds...'.
    ENDMETHOD.

    CALL METHOD cl_aunit_assert=>assert_equals
    EXPORTING
        act = lr_savings->balance
        exp = '1500.00'
        msg = 'Savings account not credited correctly.'.

    CALL METHOD cl_aunit_assert=>assert_equals
    EXPORTING
        act = lr_checking->balance
        exp = '500.00'
        msg = 'Checking account not debited correctly.'.
    ENDMETHOD.

    METHOD teardown.
        CLEAR: lr_checking, lr_savings.
    ENDMETHOD.
ENDCLASS.

```

Listing 9.3 A Simple Unit Test Example

Assuming the call to method `transfer` does not raise an exception, you check the results using the `ASSERT_EQUALS` utility method provided with class `CL_AUNIT_ASSERT`. This method compares the actual value of the `balance` attribute of each account object with the value you would expect each account to have after a successful transfer operation. In the following Section 9.4, *Executing Unit Tests*, you will see that this particular test discovers an error in the logic of the `transfer` method. Also, notice that an implementation for the `teardown` fixture method is provided in class `lcl_account_test`. Here, the fixtures used in the `test_transfer` method are being cleaned up. Now, if you decide to add additional test methods, you can be sure that the attributes `lr_checking` and `lr_savings` are properly initialized before they are used in a unit test.

9.4 Executing Unit Tests

After you have created your unit tests in ABAP Unit, you can run them in several different ways. In the following subsections, we will look at options for perform-

ing unit tests individually using the ABAP Workbench and in batch via the Code Inspector tool.

9.4.1 Integration with the ABAP Workbench

As we stated previously, the ABAP Unit test tool is tightly integrated into the ABAP Workbench. Therefore, it is easy to start test runs using standard menu options. For example, to initiate a test run for the `ZUNITTEST` program defined in Listing 9.3, select `PROGRAM • TEST • UNIT TEST` from the menu bar (see Figure 9.2). Similar menu options exist for the Function Builder and Class Builder tools.

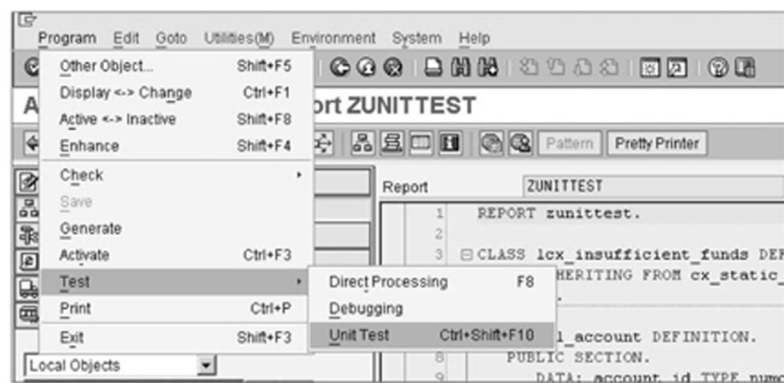


Figure 9.2 Executing a Unit Test from the ABAP Workbench

Test runs initiated in the ABAP Workbench group together the test classes defined in the program underneath a test task that is automatically generated. There is no predefined sequence in which the test methods are executed; after all, they are meant to be run independently. If the test(s) succeed, then a success message will appear in the status bar at the bottom of the screen. However, if there are errors in the unit test, then the ABAP Unit interface will be displayed. We will look at the results of a test run with errors in Section 9.5, Evaluating Unit Test Results.

9.4.2 Integration with the Code Inspector

You can also integrate ABAP Unit tests inside the Code Inspector tool (Transaction SCI). This tool is used to perform additional static checks for ABAP Repository

objects. Examples of these checks include the verification of naming conventions for variables, the proper use of ABAP statements, and so on. Although the configuration and use of this tool is outside of the scope of this book, it is a very useful tool for implementing additional quality assurance steps in the development cycle. The integration of ABAP Unit inside the Code Inspector allows developers to automate the creation of the deliverables typically required by formal code reviews (i.e., proof of adherence to project coding standards and positive unit test results), speeding up the overall development process.

9.5 Evaluating Unit Test Results

If you execute the example test case defined in Listing 9.3, you will discover an error in the `transfer` method defined for class `lcl_account`. In Figure 9.3, you can see the results of this test run in the ABAP UNIT: RESULT DISPLAY screen.

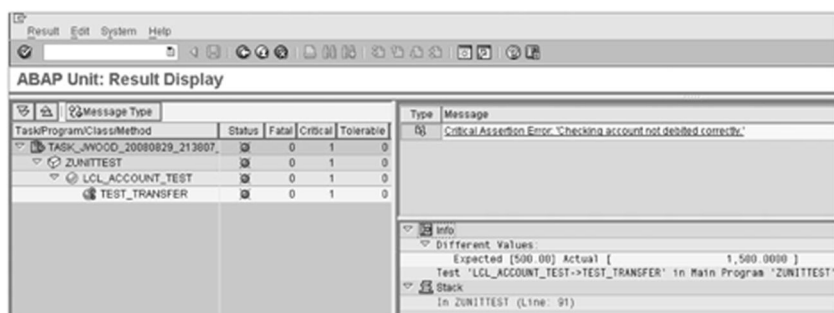


Figure 9.3 Viewing Test Results in the ABAP Unit Result Display

Here, on the left-hand side of the screen, there is a tree structure that shows the auto-generated test task, along with the associated program(s) and test classes/methods. In Figure 9.3, you can see that there was a critical error detected in the `test_transfer` method. If you double-click the method in the tree structure, the top-right pane of the result display will contain the message associated with the error. In this case, you will notice that the message *Checking account not debited correctly* is the same one used in the call to method `ASSERT_EQUALS` to check the balance of the checking account after the transfer operation.

In the bottom-right pane, you can see that we expected the balance of the checking account after the funds transfer to be \$500 in lieu of the actual balance of \$1,500. This tells us that we forgot to debit the checking account before we credited the savings account in method `transfer` of the `lcl_account` class. If the test task had been larger, there could have been more errors generated from other test methods. Therefore, the stack information provided in the bottom-right pane can be very useful in determining where a particular assertion failed.

In the example test in Listing 9.3, the default error severity defined in the `ASSERT_EQUALS` method of class `CL_AUNIT_ASSERT` has been accepted. However, the importing parameter `level` does support other severity levels such as `Fatal` or `Tolerable`. Within your individual test methods, you will have to decide how critical a given error truly is. You should spend some time browsing through the documentation for class `CL_AUNIT_ASSERT` so that you can get comfortable with the various assertion methods and their parameters because this can increase the usefulness of your unit tests.

9.6 Moving Toward Test-Driven Development

In his book *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 1999), Kent Beck asserts that "Any program feature without an automated unit test simply doesn't exist." This is an outlook shared by many developers who have embraced a new software design technique called *test-driven development* (TDD). TDD places a much higher emphasis on testing, requiring that unit tests be written *before* development begins. After the unit tests are written in an automated testing framework such as ABAP Unit, developers can begin to iteratively implement the functionality described by those unit tests while receiving continuous feedback from the testing runs.

Although the full-blown use of TDD (and indeed the extreme programming methodology from which it came) may be too controversial for your particular development team, the value of quality automated unit tests cannot be underestimated. It is imperative that you define your ABAP Unit tests as quickly as possible so that you can incorporate them into your normal development process. These tests will help keep you on target by providing immediate feedback whenever your individual modules begin to deviate from the terms of their API contracts. Unit tests will also shed light on areas of your design that need some work. For

example, if you find that a given module is difficult to test, there is likely something wrong with it.

Finally, unit tests should inspire you with the confidence to take “risks” in your development. For instance, in Chapter 5, Inheritance, we discussed the concept of refactoring to improve the design of some existing code. Without unit testing, you might be hesitant to perform certain refactorings for fear of breaking some unforeseen dependent code. Similarly, you might also be cautious about implementing enhancements for the same reasons. However, with unit tests, you can apply the changes and know immediately whether or not you broke something in the system without having to conduct a full-scale regression test.

9.7 UML Tutorial: Use Case Diagrams

Even if you haven't spent much time working with the UML before, it is likely that you may have heard the term *use case* used in various contexts at one time or another. Use cases are an important part of the UML standard, although ironically, the UML specification has very little to say about how to actually define one. Instead, it focuses on the use case diagram, which only tells a very small part of the story.

In his book *Writing Effective Use Cases* (Addison-Wesley, 2001), Alistair Cockburn defines a use case as something that “... captures a contract between the stakeholders of a system about its behavior.” In other words, you can think of a use case as a method for capturing the functional requirements of a system or module. A use case is fairly succinct, describing a single interaction scenario between a requesting user or system (referred to as an *actor*) and the system under discussion. Each use case defines a *main success scenario* that defines how an actor can achieve its goal. At each step within the main success scenario, it is highly possible that something might occur to cause the flow of the use case to deviate. These deviation scenarios are referred to as *extensions*. Separating these extension scenarios from the main success scenario makes the use case much easier to read.

Use case development is a collaborative process that requires a lot of communication within a project team. Most of the time, this process is driven heavily by business analysts that may not be familiar with the UML. Therefore, use case scenarios are often best represented in text form. You will see an example of this form in Section 9.7.2, An Example Use Case.

9.7.1 Use Case Terminology

Before we proceed with the development of an example use case, it is important to understand some basic terminology. Table 9.4 provides a description of some of the most common terms used in use case parlance.

Term	Description
Actor	A user or system that interacts with the system under discussion. From the perspective of the system under discussion, an actor is defined in terms of the role(s) it plays in the system.
Primary Actor	The primary actor is the actor that initiates the use case scenario.
Scope	The scope describes the system under discussion.
Precondition	Preconditions describe what must be true before the use case can begin. For example, a precondition of a web application might be that the user has been properly authenticated. In this case, the precondition simplifies the prose in the use case scenario because you don't have to include steps to verify that a user is authenticated before executing a given step, and so on.
Guarantee	A guarantee describes the invariants maintained by the system throughout a use case scenario. For example, a use case scenario describing a transfer of funds between two accounts in a banking system would have guarantees ensuring that both the source and target account are debited/credited correctly, and so on.
Main Success Scenario	The primary scenario of the use case that describes how an actor will reach its goal. You can think of this scenario as the "sunny day" scenario for the use case.
Extension Scenario	Extension scenarios are scenarios that describe alternative behavior within the main success scenario.

Table 9.4 Some Basic Use Case Terms

9.7.2 An Example Use Case

As stated previously, there are no hard-and-fast rules for defining use cases. The use case example shown in Figure 9.4 highlights some of the more common elements used when defining use case documents.

Use Case: Student Registering for a Training Class Online	
Primary Actor	Student
Scope	Online Course Registration website
Preconditions	Student has logged onto course website.
Main Success Scenario	
1)	Student <u>browses the course catalog and selects the course he wants to attend.</u>
2)	Student clicks a button to register for the class.
3)	Student fills in basic contact information (i.e., name, email, etc.).
4)	Student fills in payment information (e.g., credit card, etc.).
5)	Student submits the registration request.
6)	System verifies that seats are available.
7)	System <u>verifies payment information</u> , authorizing the purchase.
8)	System displays success confirmation on the screen.
9)	System sends a follow-up email confirming the registration.
Extensions	
6a)	No seats are available. <ul style="list-style-type: none"> • 1) System displays message indicating class is full. • 2) Returns to main success scenario at step 1.
7a)	Payment information is invalid. <ul style="list-style-type: none"> • 1) Student can select another form of payment or cancel the process.

Figure 9.4 An Example Use Case Document

Ideally, if done correctly, the use case in Figure 9.4 should be very easy to read. This is a use case for registering for a training class online. Initially, we define the primary actor, the system under discussion, and some basic preconditions for executing the use case. Next, we proceed into the main success scenario, which is defined as a sequence of numbered steps. As you can see, each step is described using action words that are direct and to the point.

To keep things succinct, you can reference other use cases by simply underlining a particular bit of action text. For example, in Step 1 of the main success scenario, the phrase *browses the course catalog and selects the course he wants to attend* has been underlined to indicate that the search functionality of the course catalog is described in another use case. The use case in Figure 9.4 also contains a couple of

extension scenarios. These scenarios describe what happens whenever the class is full or if the provided payment details are invalid.

Keep in mind that the example shown in Figure 9.4 is just one way of documenting a use case. A use case is good as long as it accurately describes an interaction with the system. When you read a use case document, you should be able to quickly ascertain the *who*, *what*, *when*, *where*, and *why* of a particular interaction within the system. When it comes to use case documentation, often less is more.

9.7.3 The Use Case Diagram

Figure 9.5 shows an example of a use case diagram for the use case outlined in earlier in Figure 9.4. As you can see, the graphical notation for use cases in the UML is fairly simple, basically showing the relationships between actors and use cases. The use cases are drawn within a rectangular box that represents the boundaries of the system. Internally, use cases can define *include* relationships to depict their dependencies on other use cases.

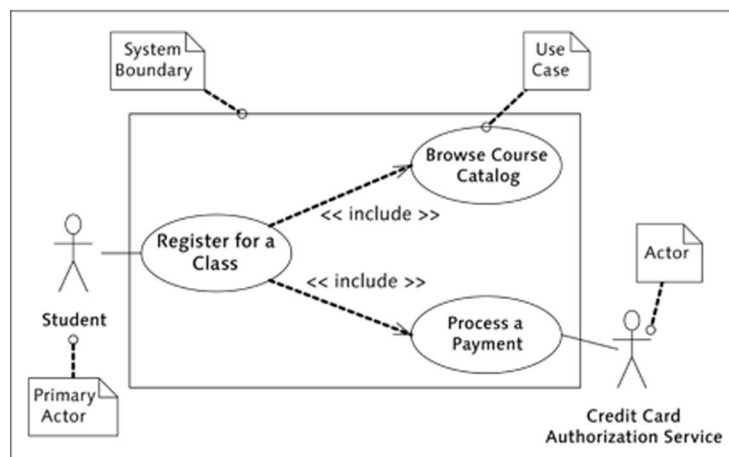


Figure 9.5 A Use Case Diagram Example

In his book *UML Distilled* (Addison-Wesley, 2004), Martin Fowler suggests that one way to look at use case diagrams is as a type of graphical table of contents for a set of use case documents. For example, the use case diagram in Figure 9.5

shows a high-level overview of the course registration system, its use cases, and the relevant actors interacting with those use cases. For more information about any particular use case, you must consult detailed use case documentation such as that shown in Figure 9.4.

9.7.4 Use Cases for Requirements Verification

Use cases are an excellent method for capturing functional requirements. Unfortunately, they are not widely used in SAP projects. Consequently, as an ABAP developer, you might be asking yourself why you should care about use cases. After all, most of the time, the documentation of functional requirements falls under the purview of the business analysts working on a project.

Typically, in most of the waterfall methodologies employed on SAP projects, developers do not enter into the software development process until a functional specification is written. Here, developers are often expected to simply read through the functional specification and start the design process. However, before proceeding too far down this path, the smart developer will check back with the business analysts to make sure that his interpretation of the requirements is consistent with the vision of the business analysts so that nothing is lost in the translation of the functional requirements. Use cases can be a very effective tool for documenting such interpretations.

Moreover, by spending just a little bit of extra time documenting use cases, developers can make life much easier for themselves and others by distilling the requirements into a form that is straightforward and easy to interpret. This documentation becomes a vital part of a technical design document, saving future developers from having to try to interpret a complex functional design from scratch.

9.7.5 Use Cases and Testing

Use cases can also come in handy when you are ready to start developing unit or functional tests. Each action step in a main success or extension scenario probably represents a unit of work that should be tested independently. At the very least, it should give you an excellent start for narrowing down your test scenarios. When compared with the alternative of trying to comb through a large functional specification document in search of test scenarios, you can really see where the effort of documenting use cases is justified.

9.8 Summary

Unit tests are the last quality assurance checkpoint a development object must pass through before it is turned over to the wider project community. Consequently, it is important that you get them right so you can deliver quality development objects. The design of automated unit tests using the ABAP Unit testing framework simplifies this endeavor by facilitating the creation of robust test cases that produce repeatable results.

In the next chapter, we will shift gears and begin looking at some of the more typical places where ABAP Objects classes are used in common development efforts within an SAP project.



PART III
Case Studies

The SAP Control Framework makes it possible to manage custom desktop UI controls from an ABAP program running remotely on the SAP NetWeaver Application Server. These controls greatly enhance the user experience of working with the classic Dynpro UI by leveraging complex controls that are already installed on a user's workstation. In this chapter, you will learn how to develop interactive reports using the SAP List Viewer control and the ALV Object Model.

10 Working with the SAP List Viewer

This chapter begins the first of several case studies that demonstrate how ABAP Objects classes are used in some common ABAP development tasks. The SAP List Viewer (commonly known as the *ABAP List Viewer* or *ALV*) is a flexible UI control that can be used to display structured datasets in various formats. In the past, if you wanted to work with the ALV toolset, you had one of two options:

- ▶ Use function modules from the Reuse Library (e.g., `REUSE_ALV_GRID_DISPLAY`).
- ▶ Interact with the grid control directly using the SAP Control Framework (e.g., via proxy classes `CL_GUI_ALV_GRID`, etc.).

However, with SAP NetWeaver 2004, SAP has provided a unified API called the *ALV Object Model* that is based purely on ABAP Objects classes.

Throughout the course of this chapter, you will see how to design and implement a simple flight query report using ALV Object Model. Along the way, we will revisit the concept of ABAP Objects events to see how to respond to events triggered by users interacting with the ALV control on their frontend workstations.

10.1 Overview of the SAP Control Framework

In the early days of screen programming in the SAP R/3 environment, the list of available screen elements that could be added to a classic Dynpro screen was limited to simple labels, input fields, buttons, and so on. These elements could be

used to develop screens that were *functional*, but it was very difficult to create high fidelity screens with a look-and-feel that many users were accustomed to working with in other popular desktop applications. Recognizing these shortcomings, SAP introduced the SAP Control Framework in Release 4.5 of the Basis kernel (which was the predecessor to the SAP Web AS).

At a high level, the SAP Control Framework makes it possible to manage custom UI controls on a user's desktop client from an ABAP application running remotely on the SAP Web AS. These custom controls are implemented using either Microsoft's ActiveX® or Sun's JavaBeans™ component technologies depending on the version of the SAP GUI client that is being used. Support for common industry component models such as ActiveX or JavaBeans makes it possible to integrate many types of complex controls into the classic Web Dynpro UI.

10.1.1 Control Framework Architecture

As you can see in Figure 10.1, the SAP Control Framework architecture exists in two distributed parts:

- ▶ The server part of the framework is based on a series of ABAP Objects classes collectively referred to as the *ABAP Objects Control Framework*. This framework contains some core classes that define the low-level interfaces between the application server and the frontend, as well as a series of proxy classes that encapsulate the functionality of custom controls behind an ABAP Objects interface. These proxy classes inherit from the base control class `CL_GUI_CONTROL`.
- ▶ The client side of this framework is managed by a special component integrated into the SAP GUI client called the *Automation Controller*. The Automation Controller manages all of the custom control instances being used on the SAP GUI frontend. It is also responsible for managing communication between the custom controls and the ABAP Objects Control Framework. This RFC-based communication includes the transfer of data between the ABAP program and the custom controls as well as the propagation of events triggered on the frontend to event handler methods registered on the backend. To maximize performance, all communication is buffered inside of an *Automation Queue* on both the client and the server.

One of the main advantages of working with the control framework is the fact that much of the functionality in an application can be delegated to the frontend client. For example, the ALV grid control is able to sort and filter the data it dis-

plays without requiring an interaction with the server. This reduces the load on the application server and also makes users happy because they get their results much sooner than they would if they had to wait on a dialog step.

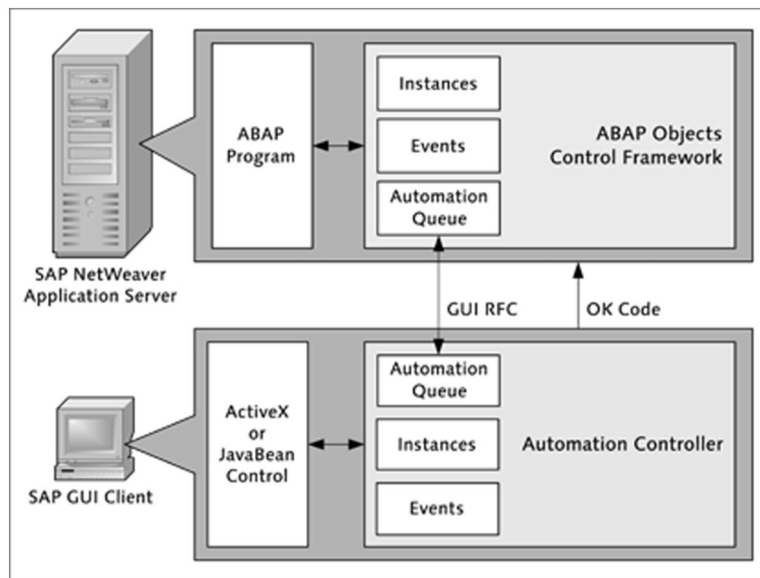


Figure 10.1 SAP Control Framework Architecture

As you can see in Figure 10.1, the SAP Control Framework also keeps track of the events that are triggered on the frontend client. These events are dispatched by the framework to event handler methods defined in a custom ABAP Objects class. The signature of these event handler methods provides information about the event as well as a reference to the *sender* object (i.e., the proxy object for the custom control). This information allows you to react to the event in some application-specific way. You will see an example of this in Section 10.4, Event Handling with the ALV Object Model.

10.1.2 Survey of Available Controls

Besides the ALV grid control that we have discussed already, there are many other useful custom controls in the SAP NetWeaver AS that are available out of the box. Examples of these controls include special container types, a calendar

control, a control for embedding a web browser on a screen, and many others. It is also possible to use the SAP Control Framework to design your own custom controls based on ActiveX components, JavaBeans, or even .NET controls.

10.2 Overview of the ALV Object Model

Before we get started with our example report program, it will help to become familiar with some of the more common elements of the ALV Object Model that you will be working with. The example in this chapter will be using class `CL_SALV_TABLE` to represent the flight query results. This class is used to render a simple two-dimensional table in a classical ABAP list, a GUI container on a Web Dynpro screen, or as a full screen table display. To keep things simple, we will accept the default full screen table display type as we design our report.

If you look closely at the properties of class `CL_SALV_TABLE`, you will notice that it is configured with the private instantiation type. As you will recall from Chapter 4, Object Initialization and Cleanup, private instantiation implies that we cannot create instances of class `CL_SALV_TABLE` directly using the `CREATE OBJECT` statement. Instead, SAP has provided a special class method called `FACTORY` that can be used to obtain an instance of the ALV tool. *Factory methods* are provided to simplify the creation of objects. In the case of class `CL_SALV_TABLE`, the `FACTORY` method takes care of initializing the surrounding GUI environment, populating data in the grid, and, perhaps most importantly, dynamically deriving the metadata the ALV tool needs to render the report. In the past, this metadata had to be built by hand in a special internal table variable called a *field catalog*.

The ALV table object returned by the `FACTORY` method is fully functional and can be displayed on the screen using the `DISPLAY` method. However, you will usually want to tweak various aspects of the table before you hand it over to the users. In the ALV Object Model, these settings have been encapsulated into distinct ABAP Objects classes. These classes mostly contain getter and setter methods that you can use to configure various settings that affect how the table is displayed or interacted with. The UML class diagram shown in Figure 10.2 shows some of the common getter methods provided in class `CL_SALV_TABLE` for accessing instances of these property classes. For more information about the details of an individual property class, consult the SAP online help documentation (<http://help.sap.com>).

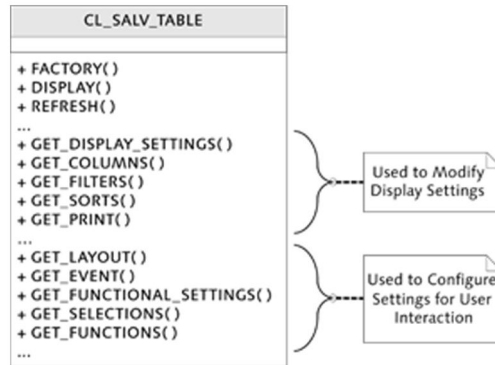


Figure 10.2 UML Class Diagram for Class CL_SALV_TABLE

10.3 Getting Started with the Flight Query Report

Now that you have a basic understanding of the ALV Object Model, we can proceed with the development of our flight query report. The following subsections show you how to implement various aspects of the report program in detail.

10.3.1 Understanding the Report Requirements

The general requirements for the flight query report are to allow users to display a set of upcoming airline flights according to some basic selection criteria. The data for this report will be driven out of the familiar *Flight Data Model* used in many of the standard examples provided by SAP. The search results of the flight query will be displayed in an ALV grid control. To demonstrate ABAP Objects eventing, users will be able to double-click on a particular flight in the search results to display a pop-up window showing the number of seats available on the selected flight.

10.3.2 Report Design Using the MVC Design Pattern

To maximize reusability, you can isolate the various aspects of the report into separate layers. One common way of organizing GUI applications is to employ the popular Model-View-Controller (MVC) design pattern. The MVC design pattern

separates the user interface layer (or *view*) from the underlying *business model* layer so that the two can vary independently. The “glue” component between these two layers is referred to as a *controller*. Controllers act as traffic cops for the application, managing the user experience by responding to user events, querying and updating the business model, and selecting the appropriate view.

In keeping with this best practice for UI design, the flight query report will be separated into the three layers prescribed by the MVC design pattern. However, for the sake of brevity, we will bend the rules a little bit here and there so that we can focus in on the aspects of the design that more closely pertain to the topics at hand in this chapter. Where appropriate, these shortcuts will be identified so that you will understand the areas of the design that need some additional work for a report that is being used in a real-world setting.

10.3.3 Developing the Flight Model Class

Because the foundation of any application is based on the underlying business model, it seems appropriate to begin the development of our example report by first creating a model class to encapsulate our interactions with the SAP flight data model. Normally, you will prefer to develop your model classes globally so that they can be more easily reused. However, to keep things simple (and also to practice our ABAP Objects syntax), we will build a local model class called `lcl_flight_model` inside of an include program called `ZALVFLIGHT_CLASSES` (see Listing 10.1). The `lcl_flight_model` class defines a single public instance method called `query_flights` that uses the standard BAPI function `BAPI_FLIGHT_GETLIST` to query the flight database according to some basic search criteria.

```
*&-----*
*& Include ZALVFLIGHT_CLASSES *
*&-----*
TYPES: ty_flight_list TYPE STANDARD TABLE
        OF bapisfldat,
        ty_date_range TYPE STANDARD TABLE
        OF bapisfldra.

CLASS lcl_flight_model DEFINITION.
  PUBLIC SECTION.
    METHODS:
      query_flights
```



```

        IMPORTING im_from      TYPE s_airport
                im_depart_date TYPE s_date
                im_to         TYPE s_airport
                im_return_date TYPE s_date
        EXPORTING ex_results   TYPE ty_flight_list.
ENDCLASS.

CLASS lcl_flight_model IMPLEMENTATION.
    METHOD query_flights.
    *   Method-Local Data Declarations:
        DATA: ls_from_dest TYPE bapisfldst,
              ls_to_dest   TYPE bapisfldst,
              lt_date_range TYPE ty_date_range.
        FIELD-SYMBOLS:
            <lfs_date_range> LIKE LINE OF lt_date_range.

    *   Populate the from/to destination structures:
        ls_from_dest-airportid = im_from.
        ls_to_dest-airportid = im_to.

    *   Build the selection date range - as necessary:
        IF NOT im_depart_date IS INITIAL.
            APPEND INITIAL LINE TO lt_date_range
                ASSIGNING <lfs_date_range>.
            <lfs_date_range>-low = im_depart_date.

            IF NOT im_return_date IS INITIAL.
                <lfs_date_range>-sign = 'I'.
                <lfs_date_range>-option = 'BT'.
                <lfs_date_range>-high = im_return_date.
            ELSE.
                <lfs_date_range>-sign = 'I'.
                <lfs_date_range>-option = 'EQ'.
            ENDIF.
        ENDIF.

    *   Use the standard BAPI to perform the flight query:
        CALL FUNCTION 'BAPI_FLIGHT_GETLIST'
            EXPORTING
                destination_from = ls_from_dest
                destination_to   = ls_to_dest
            TABLES

```

```

        date_range      = lt_date_range
        flight_list     = ex_results.
    ENDMETHOD.
ENDCLASS.

```

Listing 10.1 The Flight Data Model Class

Normally, you would want to store the search results inside the model instance so that you can define additional business methods to work on this data, and so on. However, class `CL_SALV_TABLE` requires direct access to the internal table that it displays so that it can sort the records in place. This requirement is difficult because we prefer not to expose data attributes in the public interface of class `lcl_flight_model`. If there was more time, a framework could be developed in the controller layer that seamlessly *bound* this data using getter and setter methods. This technique is used in the MVC framework associated with Business Server Pages (BSPs) technology, for example. For now, to keep it simple, the search results are exported in the `ex_results` parameter of method `query_flights` so they can be cached locally in the controller class.

10.3.4 Developing the Report Controller Class

Initially, the controller class for our report should be very straightforward. The only *event* that it needs to respond to is a request to display the report. Listing 10.2 shows our first pass at the controller class definition for our flight query report. The class `lcl_query_ctrl`, which is also part of the include program `ZALVFLIGHT_CLASSES`, provides a public instance method called `do_display_report` to handle requests to render the report on the screen.

```

*&-----*
*& Include ZALVFLIGHT_CLASSES *
*&-----*
CLASS lcl_flight_model IMPLEMENTATION.
...
ENDCLASS.

```

```

CLASS lcl_query_ctrl DEFINITION.
PUBLIC SECTION.
METHODS:
    constructor,
    do_display_report

```

```

        IMPORTING im_from      TYPE s_airport
                im_depart_date TYPE s_date
                im_to         TYPE s_airport
                im_return_date TYPE s_date.
PRIVATE SECTION.
DATA: flight_model TYPE REF TO lcl_flight_model,
      flight_list  TYPE ty_flight_list,
      grid         TYPE REF TO cl_salv_table.

METHODS:
      show_grid.
ENDCLASS.

CLASS lcl_query_ctrl IMPLEMENTATION.
METHOD constructor.
*   Initialize the flight model:
    CREATE OBJECT flight_model.
ENDMETHOD.

METHOD do_display_report.
*   Use the model method "query_flights" to query the
*   flight database:
    REFRESH flight_list.
    CALL METHOD flight_model->query_flights
        EXPORTING
            im_from      = im_from
            im_depart_date = im_depart_date
            im_to         = im_to
            im_return_date = im_return_date
        IMPORTING
            ex_results   = flight_list.

*   Display the result set in an ALV grid:
    show_grid( ).
ENDMETHOD.

METHOD show_grid.
*   Implementation deferred for now...
ENDMETHOD.
ENDCLASS.
"lcl_query_ctrl

```

Listing 10.2 Implementing the Flight Query Controller Class

Normally, whenever an input event is triggered in the view, the controller responds to that event by notifying the model and selecting the next appropriate view for the user. In a typical ABAP report program, the initial view shown to a user is a *selection screen*. In the flight query report, we will provide users with a selection screen containing parameters that allow them to filter the flight query results by destination and date range. After users enter this data, they can execute the report by clicking on the EXECUTE button (see Figure 10.3).

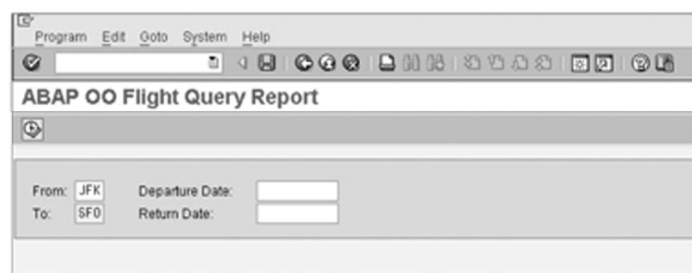


Figure 10.3 Defining Selection Criteria for the Flight Query Report

Whenever a user executes the report, the `START-OF-SELECTION` event block is triggered. In Listing 10.3, you can see that the `ZALVFLIGHT` report program is delegating this event to an instance of our custom `lcl_query_ctrl` controller class by calling instance method `do_display_report`.

```
*&-----*
*& Report ZALVFLIGHT *
*&-----*
REPORT zalvflight.

INCLUDE zalvflight_classes.

DATA: lr_controller TYPE REF TO lcl_query_ctrl.

SELECTION-SCREEN BEGIN OF BLOCK blk_main WITH FRAME.
  SELECTION-SCREEN BEGIN OF LINE.
    SELECTION-SCREEN COMMENT 1(5) text-t02.
    PARAMETERS p_from TYPE s_airport OBLIGATORY.
    SELECTION-SCREEN COMMENT 15(15) text-t03.
    PARAMETERS p_dptdat TYPE s_date.
```

```

SELECTION-SCREEN END OF LINE.

SELECTION-SCREEN BEGIN OF LINE.
  SELECTION-SCREEN COMMENT 1(5) text-t04.
  PARAMETERS p_to TYPE s_airport OBLIGATORY.
  SELECTION-SCREEN COMMENT 15(15) text-t05.
  PARAMETERS p_retdat TYPE s_date.
SELECTION-SCREEN END OF LINE.
SELECTION-SCREEN END OF BLOCK blk_main.

LOAD-OF-PROGRAM.
  CREATE OBJECT lr_controller.

START-OF-SELECTION.
  CALL METHOD lr_controller->do_display_report
    EXPORTING
      im_from      = p_from
      im_depart_date = p_dptdat
      im_to        = p_to
      im_return_date = p_retdat.

```

Listing 10.3 Integrating the Controller into an Executable Report

Inside method `do_display_report` (refer to Listing 10.2), the provided selection screen parameters are used to query the flight model. Here, as you saw in Listing 10.1, the `query_flights` method is calling the standard BAPI function `BAPI_FLIGHT_GETLIST` to execute the flight query. The results are cached in the private `flight_list` instance attribute defined in class `lcl_query_ctrl`.

After the flight model has been queried, the next task for the controller is to provide users with a view of the flight query list results. In this case, the view is encapsulated inside of class `CL_SALV_TABLE`. The private helper method `show_grid` of class `lcl_query_ctrl` is used to select this view for the users. We will look at how to implement method `show_grid` in the following section.

10.3.5 Implementing the Report View

The simplicity of the ALV Object Model makes the view selection task for controller class `lcl_query_ctrl` very straightforward. Listing 10.4 shows the implementation of the `show_grid` helper method that is used to display the flight query results view.

```

CLASS lcl_query_ctrl IMPLEMENTATION.
...
METHOD show_grid.
*   Method-Local Data Declarations:
    DATA: lr_grid_ex TYPE REF TO cx_salv_msg,
           lv_message TYPE string,
           lr_functions TYPE REF TO cl_salv_functions.

*   Use the factory method to create an instance of the
*   ALV grid:
    TRY.
        CALL METHOD cl_salv_table=>factory
            IMPORTING
                r_salv_table = grid
            CHANGING
                t_table      = flight_list.
        CATCH cx_salv_msg INTO lr_grid_ex.
            lv_message = lr_grid_ex->get_text( ).
            MESSAGE lv_message TYPE 'I'.
            RETURN.
        ENDTRY.


*   Enable the standard ALV toolbar:
    lr_functions = grid->get_functions( ).
    lr_functions->set_all( abap_true ).

*   Display the ALV grid:
    grid->display( ).
ENDMETHOD.
ENDCLASS.

```

Listing 10.4 Displaying the Flight Query Results View

As you can see in Listing 10.4, the majority of the work is handled by the aforementioned `FACTORY` method of class `CL_SALV_TABLE`. To simplify the task of calling this method, do the following:

1. Click on the `PATTERN` button in the ABAP Editor to call up the Insert Statement wizard shown in Figure 10.4.
2. Select the `ABAP OBJECTS PATTERNS` button, and press .

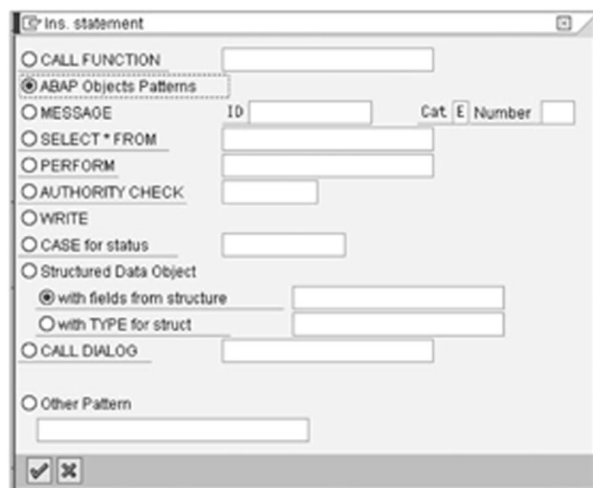


Figure 10.4 Working with the ABAP Objects Pattern Generator – Part 1

- This brings you to the OO STATEMENT PATTERN dialog box shown in Figure 10.5. This screen provides various options for generating method calls, raising events, and so on¹. Use the CALL METHOD pattern to build the syntax needed to call method FACTORY of class CL_SALV_TABLE.



Figure 10.5 Working with the ABAP Objects Pattern Generator – Part 2

¹ Note that the ABAP Objects pattern generator tool can only work with global classes defined in the ABAP Repository.

After the grid object is instantiated, you can enable the standard ALV toolbar by invoking method `SET_ALL` of class `CL_SALV_FUNCTIONS`. Figure 10.6 shows an example of the report display generated by method `show_grid`.

Air	Airline	Flig	Flight Date	De	Depart city	Ta	Arrival city	Departure	Arrival	Arrival date	Airfare	Curre	IS
AA	American Airlines	17	10/10/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	10/10/2007	422.9400	USD	USC
AA	American Airlines	17	11/07/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	11/07/2007	422.9400	USD	USC
AA	American Airlines	17	12/05/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	12/05/2007	422.9400	USD	USC
AA	American Airlines	17	01/02/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	01/02/2008	422.9400	USD	USC
AA	American Airlines	17	01/30/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	01/30/2008	422.9400	USD	USC
AA	American Airlines	17	02/27/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	02/27/2008	422.9400	USD	USC
AA	American Airlines	17	03/26/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	03/26/2008	422.9400	USD	USC
AA	American Airlines	17	04/23/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	04/23/2008	422.9400	USD	USC
AA	American Airlines	17	05/21/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	05/21/2008	422.9400	USD	USC
AA	American Airlines	17	06/18/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	06/18/2008	422.9400	USD	USC
AA	American Airlines	17	07/16/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	07/16/2008	422.9400	USD	USC
AA	American Airlines	17	08/13/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	08/13/2008	422.9400	USD	USC
AA	American Airlines	17	09/10/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	09/10/2008	422.9400	USD	USC
AA	American Airlines	17	10/08/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	10/08/2008	422.9400	USD	USC
AA	American Airlines	17	11/05/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00	11/05/2008	422.9400	USD	USC
DL	Delta Airlines	1699	11/05/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	11/05/2007	422.9400	USD	USC
DL	Delta Airlines	1699	12/03/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	12/03/2007	422.9400	USD	USC
DL	Delta Airlines	1699	12/31/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	12/31/2007	422.9400	USD	USC
DL	Delta Airlines	1699	01/28/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	01/28/2008	422.9400	USD	USC
DL	Delta Airlines	1699	02/25/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	02/25/2008	422.9400	USD	USC
DL	Delta Airlines	1699	03/24/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	03/24/2008	422.9400	USD	USC
DL	Delta Airlines	1699	04/21/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	04/21/2008	422.9400	USD	USC
DL	Delta Airlines	1699	05/19/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	05/19/2008	422.9400	USD	USC
DL	Delta Airlines	1699	06/16/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	06/16/2008	422.9400	USD	USC
DL	Delta Airlines	1699	07/14/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	07/14/2008	422.9400	USD	USC
DL	Delta Airlines	1699	08/11/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	17:15:00	20:37:00	08/11/2008	422.9400	USD	USC

Figure 10.6 Displaying the Flight Query Results in an ALV Table

As you can see in Listing 10.4, shown earlier, there are few dependencies between the `CL_SALV_TABLE` view class and the underlying data model. In the past, the coupling between the data model and the grid control was much tighter because developers were forced to provide view metadata (i.e., the ALV field catalog). Fortunately, with the encapsulation techniques employed in the design of the ALV Object Model, SAP has eliminated these dependencies to make report development much more flexible.

10.4 Event Handling with the ALV Object Model

We made a lot of progress with the flight query report in Section 10.3, Getting Started with the Flight Query Report. However, one requirement that we still have not implemented is the event handling logic needed to display the number of available seats whenever the user double-clicks on a flight in the report list. In this section, we will rework the flight query report to use ABAP Objects eventing functionality to satisfy this requirement.

10.4.1 Integrating Event Handler Methods into the Controller

If you recall from Chapter 2, Working with Objects, ABAP Objects events are handled by event handler methods. These methods are defined in terms of the interface specified for the triggering event and can be created in any class that has visibility to the class that defines the event. In our report example, the natural location for an event handler method is in the controller class.

Listing 10.5 shows that an event handler method called `on_double_click` has been added to class `lcl_query_ctrl`. This event handler method will respond to the `double_click` event defined in class `CL_SALV_EVENTS_TABLE`. You will see how to handle this event later in Section 10.4.3, Responding to Events.

```

CLASS lcl_query_ctrl DEFINITION.
  PUBLIC SECTION.
    METHODS:
      constructor,
      do_display_report
        IMPORTING im_from      TYPE s_airport
                  im_depart_date TYPE s_date
                  im_to        TYPE s_airport
                  im_return_date TYPE s_date.
      on_double_click FOR EVENT double_click
                      OF cl_salv_events_table
                      IMPORTING row
                               column
                               sender.
  PRIVATE SECTION.
    DATA: flight_model TYPE REF TO lcl_flight_model,
          flight_list  TYPE ty_flight_list,
          grid          TYPE REF TO cl_salv_table.

```

```

METHODS:
    show_grid.
ENDCLASS.

```

Listing 10.5 Defining an Event Handler Method for the Report

10.4.2 Registering Event Handler Methods

Simply defining an event handler method as in Listing 10.5 does not *register* our controller class as a listener for the `DOUBLE_CLICK` event of class `CL_SALV_EVENTS_TABLE`. Instead, this registration must take place at runtime using the `SET HANDLER` statement. Listing 10.6 shows how the controller method `on_double_click` has been registered to listen for events triggered in class `CL_SALV_EVENTS_TABLE` using the `SET HANDLER` statement.

```

METHOD show_grid.
* Method-Local Data Declarations:
  DATA: lr_grid_ex TYPE REF TO cx_salv_msg,
        lv_message TYPE string,
        lr_functions TYPE REF TO cl_salv_functions,
        lr_events TYPE REF TO cl_salv_events_table.

* Use the factory method to create an instance of the
* ALV grid:
  TRY.
    CALL METHOD cl_salv_table=>factory
      IMPORTING
        r_salv_table = grid
      CHANGING
        t_table = flight_list.
  CATCH cx_salv_msg INTO lr_grid_ex.
    lv_message = lr_grid_ex->get_text( ).
    MESSAGE lv_message TYPE 'I'.
  RETURN.
  ENDTRY.

* Enable the standard ALV toolbar:
  lr_functions = grid->get_functions( ).
  lr_functions->set_all( abap_true ).

* Register the event handler method "on_double_click" so
* that we can respond to double-click events on the grid:

```

```

lr_events = grid->get_event( ).
SET HANDLER me->on_double_click FOR lr_events.

* Display the ALV grid:
grid->display( ).
ENDMETHOD.

```

Listing 10.6 Registering the Controller as an Event Handler

10.4.3 Responding to Events

Now that we have registered our `on_double_click` event handler method, we need to implement the logic necessary for displaying the number of available seats on the flight selected by the user. Listing 10.7 shows an example of this logic. Here, we use the importing `row` parameter provided by the event to determine which flight the user has selected in the flight query results. After we have this information, we can query table `SFLIGHT` to obtain additional information about the flight. Here, we use the `SEATSMAX` and `SEATSOCC` fields to calculate the number of available seats on the flight and then display the results in a pop-up using the `MESSAGE` statement.

```

METHOD on_double_click.
* Method-Local Data Declarations:
DATA: ls_selected_flight TYPE bapisfldat,
      ls_flight_info     TYPE sflight,
      lv_open_seats     TYPE numc5,
      lv_message        TYPE string.

* Read the selected flight record from the model
* using the "row" index provided by the event:
READ TABLE flight_list
INDEX row
INTO ls_selected_flight.

* Use the key of the selected flight record to extract
* additional flight information from table SFLIGHT:
SELECT SINGLE *
INTO ls_flight_info
FROM sflight
WHERE carrid EQ ls_selected_flight-airlineid
AND connid EQ ls_selected_flight-connectid
AND fldate EQ ls_selected_flight-flightdate.

```

```

* Calculate the total number of available seats left
* on the flight:
lv_open_seats =
    ls_flight_info-seatsmax - ls_flight_info-seatsocc.
SHIFT lv_open_seats LEFT DELETING LEADING '0'.
SHIFT ls_flight_info-connid LEFT DELETING LEADING '0'.

* Display the results in a pop-up message:
CONCATENATE 'There are'(001)
            lv_open_seats
            'seats available on Flight'(002)
            ls_flight_info-connid
            INTO lv_message
            SEPARATED BY space.

MESSAGE lv_message TYPE 'I'.
ENDMETHOD.

```

Listing 10.7 Implementing the Event Handler Method

10.4.4 Triggering Events on the Frontend

After we have activated all of the changes outlined in this section, we can execute the report again to test the `double_click` event. Figure 10.7 shows the pop-up message generated by double-clicking on a flight record in the search result list.

Behind the scenes, the SAP Control Framework is propagating the `double_click` event on the frontend ALV grid control back to the ABAP Objects Control Framework. Internally, the event is captured inside an instance of the `CL_SALV_EVENTS_TABLE` and forwarded to our event handler method via method `RAISE_DOUBLE_CLICK`. Here, method `RAISE_DOUBLE_CLICK` uses the `RAISE EVENT` statement to trigger the instance event. The syntax for the `RAISE EVENT` statement is shown in Listing 10.8.

```

RAISE EVENT evt
  [EXPORTING
    e1 = f1
    e2 = f2
    ...].

```

Listing 10.8 Syntax Diagram for the RAISE EVENT Statement

The screenshot shows a window titled "ABAP OO Flight Query Report" with a menu bar (List, Edit, Goto, Settings, System, Help) and a toolbar. Below the toolbar is a table with the following columns: Air, Airline, Flig, Flight Date, De, Depart city, Ta, Arrival city, Departure, and Arrival. The table contains several rows of flight data for American Airlines. An information popup is overlaid on the table, displaying the message: "There are 14 seats available on Flight 17".

Air	Airline	Flig	Flight Date	De	Depart city	Ta	Arrival city	Departure	Arrival
AA	American Airlines	17	10/10/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines	17	11/07/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines	17	12/05/2007	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines	17	01/02/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines	17	01/30/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines							00:00:00	21:00:00
AA	American Airlines	17	09/10/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00
AA	American Airlines	17	10/08/2008	JFK	NEW YORK	SFO	SAN FRANCISCO	00:00:00	21:00:00

Figure 10.7 Triggering the Double-Click Event in the Report

Note that the `RAISE EVENT` requires actual parameters to be provided for each non-optional formal parameter defined in the event interface. The implicit sender parameter is mapped to the `me` self-reference variable described in Chapter 5, Inheritance.

10.4.5 Timing of Event Handling

Events are processed synchronously. In other words, whenever an event is triggered using the `RAISE EVENT` statement, all of the registered event handler methods will be processed before the next statement is processed. The event handler methods are processed in the order they were registered with the system. Therefore, you should not develop any dependencies between event handler methods as you cannot guarantee that one event handler method will run before the other, and so on.

10.5 UML Tutorial: Communication Diagrams

One of the most difficult stages of the Object-Oriented Analysis and Design (OOAD) process is the point in which we begin to try to assign roles and responsibilities to the classes identified during the structural analysis phase. At this point

in the process, all that we have to work with are high-level behavioral diagrams (e.g., activity diagrams, use cases, etc.) as well as some class and object diagrams that describe the classes we have modeled. Certainly, associations in class diagrams help us understand the relationships between these classes, but they aren't very useful in describing the behavior of a system in terms of these classes.

Frequently, this kind of detailed behavior is captured in a sequence diagram as you saw in Section 3.6, UML Tutorial: Sequence Diagrams. Sequence diagrams are an example of an *interaction diagram*. Interaction diagrams emphasize the flow of data and control between objects interacting in a system. In this section, we will look at another type of interaction diagram in the UML called the *communication diagram*.

Communication diagrams (formerly known as *collaboration diagrams* in UML 1.x) blend elements from class, object, sequence, and use case diagrams together in the graph notation shown in Figure 10.8. This communication diagram depicts the same *Withdraw Cash* interaction that we considered in Section 3.6, UML Tutorial: Sequence Diagrams, when we looked at sequence diagrams. Here, as you will recall, we are depicting the interaction between objects collaborating in an ATM cash withdrawal transaction.

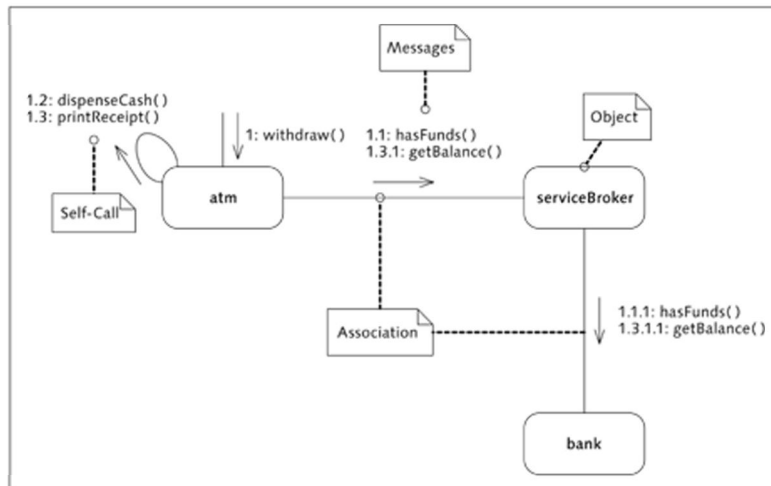


Figure 10.8 Example UML Communication Diagram

As you can see, there are a lot of similarities between both of these diagrams. In fact, whether you use one notation or the other is mainly a matter of preference. However, many developers like to use communication diagrams to whiteboard their ideas because they are easier to sketch than sequence diagrams due to the fact that you do not need to organize your objects into a two-dimensional graph. In fact, one way to develop communication diagrams is to begin overlaying an object diagram with messages.

One challenge with working with communication diagrams is the nested decimal numbering scheme shown in Figure 10.8. For this reason, it is important that you keep a communication diagram small so that the message numbers don't become too nested and hard to read.

Perhaps the most valuable aspect of a communication diagram is the fact that it keeps static associations in focus as you begin to develop the interactions between classes. This visualization is important because it helps you keep your architectural vision intact as you begin to connect the dots between your classes at runtime.

10.6 Summary

In this chapter, you saw how standard and customized ABAP Objects classes can be used together to implement an interactive report program. For the most part, we were able to minimize our use of report-specific event blocks by encapsulating the various aspects of our report into layers as outlined in the MVC design pattern. These layers help us to isolate specific changes so that maintenance and enhancement efforts are greatly simplified.

We also saw how ABAP Objects eventing can be used to respond to user events in a very flexible manner. Of course, the use of such events is not limited to GUI development; ABAP Objects events can be used in any application that needs to implement a distributed event handling system.

In the next chapter, we will look at ways to store objects persistently using ABAP Object Services.

By default, instances of ABAP Objects classes are transient in nature. In other words, whenever a program ends, these objects disappear. In this chapter, we will look at how the ABAP Object Services framework can be used to create persistent objects whose state is maintained in a persistent data store long after a program ends.

11 ABAP Object Services

Throughout the course of this book, we have considered some of the various types of relationships that you can establish between classes and interfaces. In Chapter 6, Polymorphism, you learned how to use polymorphism to take advantage of these relationships by substituting related objects interchangeably at runtime. The flexibility afforded by polymorphism makes it possible to develop sophisticated *software frameworks*.

Software frameworks provide a foundation for building solutions that solve a particular type of problem. Here, the framework provides the majority of the solution infrastructure; you only need to integrate a few strategic custom classes into the framework to implement some desired customized behavior. One such example of a software framework in the ABAP world is the *ABAP Object Services* framework.

ABAP Object Services provide various services that allow you to create and work with *persistent objects*. Persistent objects provide an abstraction on top of a persistent data store, allowing you to work with a pure object-oriented data model in your programs as opposed to a relational data model. In this chapter, you will learn how to use the Class Builder tool to create *persistent classes*. You will also see some examples that demonstrate how to use the generated persistent class API methods to perform basic CRUD operations (Create, Remove, Update, and Display) on persistent objects without having to know how they are being persisted behind the scenes.

11.1 Object-Relational Mapping Concepts

Before delving into the details of the ABAP Object Services framework, it is important to understand the concepts from which it was derived. The ABAP Object Services framework is an ABAP-based implementation¹ of an *object-relational mapping* (ORM) tool. ORM tools are used to encapsulate persistence details inside persistent classes by *mapping* a persistence data model onto an object-oriented data model. Conceptually, there's nothing magical about persistent classes; behind the scenes, SQL statements still have to be issued to interact with a database, and so on. However, the difference here is that the ORM tool takes care of these details so that you don't have to.

There are several benefits to be gained by using ORM tools:

- ▶ First and foremost, they reduce the amount of program code you have to write to implement persistent classes.
- ▶ Secondly, you work with persistent objects in the exact same way that you use transient objects. This transparency frees you from having to worry about persistence issues, allowing you to focus your design around a pure object model.
- ▶ Finally, the encapsulation of persistence details inside of a framework provides the opportunity to improve performance through caching techniques, lazy initialization techniques, and so on.

11.2 Persistence Service Overview

As you learned in Chapter 4, Object Initialization and Cleanup, the lifecycle of an object begins when it is created with the `CREATE OBJECT` statement and ends when it is destroyed by the garbage collector. Along the way, data can be stored in attributes, but the data inside these attributes is *transient* in nature. In other words, after an object is destroyed, so also is the data stored inside its internal attributes. Often, you will want to preserve this data after the object is destroyed so that it can be retrieved again later. Normally, this implies that you store (or *persist*) the data inside a database using SQL statements. You can then recreate the object later by issuing SQL queries to extract the data (perhaps in the constructor method, for instance).

¹ ABAP Object Services were made available with release 6.10 of the SAP Web AS.

If you are working with small entity objects, the effort involved in manually building a persistence layer into your classes is probably not that big of a deal. On the other hand, as the data model gets larger and more complex, the mapping process becomes much more tedious, and many ABAP developers are tempted to cut corners.

For example, consider an object model that is designed to encapsulate a transactional document such as a sales order that has header and line item details stored in ABAP Dictionary tables. Ideally, you would want to use composition techniques to create an object model that has a *sales order header* object that maintains a collection of *sales order line item* objects. However, after implementing all of the code to map the sales order header table onto the header class, a developer may decide to forego this same development effort at the line item level. Here, the developer may prefer to store the line item details inside an internal table attribute whose line type is already directly compatible with a line item table queried with the `SELECT` statement. In this case, the effectiveness of the object model begins to break down as more and more procedural elements creep into the design.

To bridge the gap between object and persistence data models, SAP introduced the *Persistence Service* as part of the ABAP Object Services framework. The Persistence Service is a software layer designed to make it easier to work with persistent objects. As you can see in Figure 11.1, the Persistence Service provides a layer of abstraction between a persistent object and the underlying data store.

There is nothing fundamentally different between a persistent class and a normal ABAP Objects class. In other words, instances of persistent classes are still transient in nature. However, when managed by the Persistence Service, these transient objects *behave* like persistent objects within an ABAP program. For example, as you can see in Figure 11.1, the Persistence Service is extracting data from the underlying AS ABAP database and initializing the persistent object on behalf of an ABAP program. Similarly, the Persistence Service is brokering the persistence of changes made to those objects from within the program. You will learn more about the relationship between a persistent object and the Persistence Service in the upcoming sections.

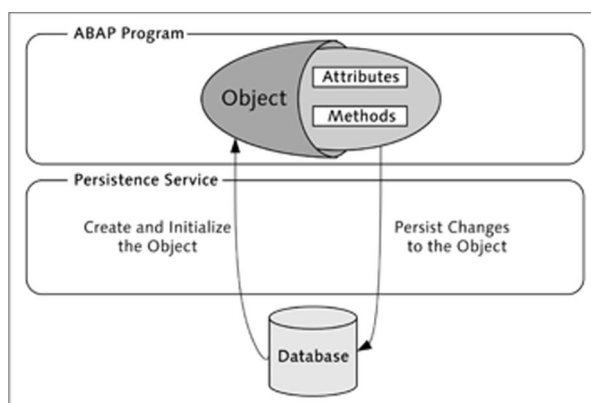


Figure 11.1 Basic Positioning of the Persistence Service

11.2.1 Managed Objects

The Persistence Service is designed to work with instances of *persistent classes*. Persistent classes are created and maintained in the Class Builder tool. In addition to the actual persistent class, the Class Builder also generates a couple of additional *agent classes* that manage all of the low-level interaction details between objects of the persistent class and the Persistence Service. In the UML class diagram shown in Figure 11.2, these agent classes are called `CA_PERSISTENT` and `CB_PERSISTENT`, respectively. The methods of the agent classes are used to *manage* the objects of persistent classes such as `CL_PERSISTENT`. Consequently, objects of persistent classes are referred to as *managed objects*.

As the name implies, the lifecycle of a managed object is controlled by a separate object. In the case of the Persistence Service, this separate object is an instance of the agent class, which is patterned as a *singleton*. The term *singleton* refers to a design pattern² that is used to restrict the instantiation of a class to a single object. In the class diagram shown in Figure 11.2, you can see that the singleton object for class `CA_PERSISTENT` is stored in a class attribute called `agent`, which is instantiated within the `CLASS_CONSTRUCTOR` method whenever the `CA_PERSISTENT` agent class is first accessed within an ABAP program.

² This design pattern was originally introduced in the classic "Gang of Four" design patterns text entitled *Design Patterns: Elements of Reusable Software* (Addison-Wesley, 1994).

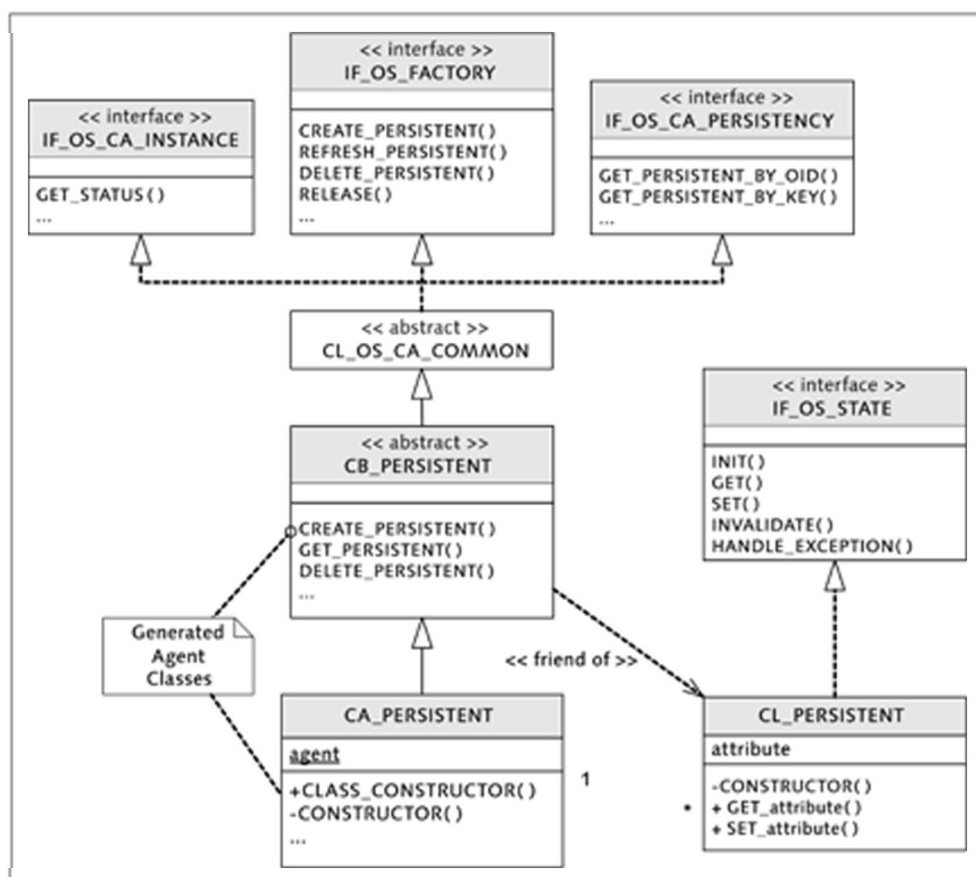


Figure 11.2 Class Diagram for a Persistent Class

At this point, you might be wondering why the Class Builder generated two agent classes instead of one. The short answer here is that SAP wanted to apply a layered approach for integrating custom persistent classes into the Persistence Service to separate concerns.

As you can see in Figure 11.2, the agent class `CB_PERSISTENT` is defined as an abstract class that inherits from the common abstract framework class `CL_OS_CA_COMMON`. Class `CB_PERSISTENT` is tightly integrated into the Persistence Service and defines the low-level persistence mapping functionality for the persistent class. This functionality is inherited by the agent class `CA_PERSISTENT`. However, because the low-level details are defined in the `PROTECTED SECTION` of class `CB_PERSISTENT`, you have limited access to them within class `CA_PERSISTENT`. This is important because it is possible to redefine and extend the `CA_PERSISTENT` agent class to enhance its capabilities. In this case, the layered approach prevents

developers from revealing too many of the low-level details through the class agent API.

If all this seems confusing, rest assured that the actual usage of a class agent is not all that complex. For the most part, developers always work with the `CA_...` agent class. Again, this is an example of a highly sophisticated framework being exposed via a simple and intuitive public interface. You will see how to interact with this interface in Section 11.4, Working with Persistent Objects.

11.2.2 Mapping Concepts

One of the prerequisites for defining a persistent class is to create a mapping between the object model and the underlying persistence layer. Table 11.1 shows the three different types of mapping strategies that you can employ in your persistent classes. These mapping types provide you with the flexibility to tap into pre-existing relational data models or generate new data models from scratch.

Mapping Type	Description
By Business Key	Can be used to map an existing table in the ABAP Dictionary that has a semantic primary key. For example, the business key for standard Table BUT000 is the <code>Partner</code> field.
By Instance-GUID	Used to map tables that have a primary key that consists of a single field of type <code>OS_GUID</code> . Here, the term GUID refers to a system-generated <i>Globally Unique Identifier</i> .
By Instance-GUID and Business Key	This mapping type combines both techniques. In this case, the target table has a semantic primary key as well as a nonkey field of type <code>OS_GUID</code> that is defined as part of a unique secondary index. The combination of these keys makes it possible to access a persistent object by a business key or an instance-GUID.

Table 11.1 Persistence Mapping Types

However, keep in mind that you must be able to represent these models using ABAP Dictionary objects. These ABAP Dictionary objects must exist *before* you try to create a mapping in the Class Builder; the ORM tools provided by SAP will not generate these objects automatically.

Normally, your persistence map will be based on one or more relational database tables. Keep in mind that the Persistence Service also supports other storage

media such as files. Irrespective of the underlying storage medium, you must use an ABAP Dictionary object (i.e., a table, view, or structure) as the basis for your mapping. The following list describes how various ABAP Dictionary objects can be used to help you create your persistence maps:

► **Single-table mapping**

Most of the time, you will map the attributes of your persistent class to a single ABAP Dictionary table. Here, you must map all of the fields from the table to attributes in the persistent class. Of course, sometimes you may not want to map all of the fields of a given table to your persistent class. For example, in a large table with many fields, you might only be interested in a subset of fields that have certain relevance to a particular usage scenario. In this case, you can create a view that contains a subset of fields that you want to map and then use the view to build your persistence mapping.

► **Multiple-table mapping**

It is also possible to map multiple tables onto a single persistent class. The only requirement here is that each of the tables shares the exact same primary key. At runtime, the Persistence Service is smart enough to connect the relevant attributes used in the mapping with their associated tables so that the object data is distributed across each of the tables correctly.

► **Structure mappings**

For more complex mappings, you can also use structure types. Structure types are typically used to implement persistence mapping to files, and so on. However, they can also be used to map persistent classes that have a *one-to-many* relationship to another persistent class type (e.g., a sales order and its line items). Of course, because structure types do not refer to an actual database table, the ORM tool will not be able to generate the code to persist the data. Instead, you must implement your own logic for storing the persistent data in persistent classes mapped from a structure.

11.2.3 Understanding the Class Agent API

After you finalize the persistence mapping for your class, the next step is to activate your changes in the Class Builder. Here, the Class Builder tool will use the mapping details to construct the persistence class along with its associated agent classes. The UML class diagram shown earlier in Figure 11.2 shows the relationships between these classes. It also shows some of the more useful methods that you can use to obtain and work with objects of your persistent class.

For the most part, the names of the methods are pretty self-explanatory (e.g., you use method `CREATE_PERSISTENT` to create a persistent object, etc.). However, the generation of some of these methods varies depending upon the type of mapping you chose to implement. For example, the base agent class `CB_...` will not have methods `GET_PERSISTENT` and `DELETE_PERSISTENT` if the persistent objects are not managed by business keys. In this case, you must use methods implemented from interfaces `IF_OS_FACTORY` and `IF_OS_CA_PERSISTENCY` instead.

You can find more information about these methods in the SAP online help documentation for ABAP Object Services (<http://help.sap.com>). You will also see some concrete examples of these methods in Section 11.4, Working with Persistent Objects.

11.3 Building Persistent Classes

At this point, you are probably eager to get started with some real-world examples. In the next several subsections, we will iteratively design a couple of persistent classes to model a portion of a data model used to build an online course registration system. In particular, we will create two separate persistent classes called `ZCL_OS_PERSON` and `ZCL_OS_ADDRESS`, where instances of class `ZCL_OS_PERSON` can optionally have an instance of class `ZCL_OS_ADDRESS` associated with them. The database tables upon which these persistent classes will be based are shown in Figure 11.3 and Figure 11.4, respectively.

Field	Key	Init	Data element	Data Ty	Length	Decim	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		Client
GUID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	OS_GUID	RAW	16		Globally Unique Identifier
NAME_FIRST	<input type="checkbox"/>	<input type="checkbox"/>	AD_NAMEFIR	CHAR	40		First name
NAME_LAST	<input type="checkbox"/>	<input type="checkbox"/>	AD_NAMELAS	CHAR	40		Last name
ADDR_CLASS	<input type="checkbox"/>	<input type="checkbox"/>	OS_GUID	RAW	16		Globally Unique Identifier
ADDR_REF	<input type="checkbox"/>	<input type="checkbox"/>	OS_GUID	RAW	16		Globally Unique Identifier

Figure 11.3 ABAP Dictionary Table Definition for Person Entity

Field	Key	Initi.	Data element	Data Ty	Length	Decim	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		0 Client
GUID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	OS_GUID	RAW	16		0 Globally Unique Identifier
STREET1	<input type="checkbox"/>	<input type="checkbox"/>	AD_STREET	CHAR	60		0 Street
STREET2	<input type="checkbox"/>	<input type="checkbox"/>	AD_STRSPP1	CHAR	40		0 Street 2
CITY	<input type="checkbox"/>	<input type="checkbox"/>	AD_CITY1	CHAR	40		0 City
REGION	<input type="checkbox"/>	<input type="checkbox"/>	REGIO	CHAR	3		0 Region (State, Province, County)
COUNTRY	<input type="checkbox"/>	<input type="checkbox"/>	LAND1	CHAR	3		0 Country Key
POSTAL_CODE	<input type="checkbox"/>	<input type="checkbox"/>	AD_PSICD1	CHAR	10		0 City postal code
PHONE_NUMBER	<input type="checkbox"/>	<input type="checkbox"/>	AD_TLNMBR	CHAR	30		0 Telephone no.: dialling code+number

Figure 11.4 ABAP Dictionary Table Definition for Address Entity

For the purposes of this contrived example, we will minimize the overall number of fields used to represent a particular entity. For instance, the Table ZCA_PERSON only contains a GUID-based key, fields to represent the first and last name of the person, and two reference fields used to associate a person with an address. Similarly, Table ZCA_ADDRESS only contains a GUID-based key and some basic address fields.

11.3.1 Creating a Persistent Class in the Class Builder

For the most part, you create persistent classes in the exact same way that you create other global classes in the Class Builder. The only difference is in the class type that you choose.

1. Select PERSISTENT CLASS as the CLASS TYPE (see Figure 11.5).
2. Click SAVE. The Class Builder takes you to the Class Editor screen where you can begin editing the persistent class.

Figure 11.6 shows class ZCL_OS_ADDRESS after it is initially created. Here, because you have not yet mapped the persistent fields for this class, the class only contains implementations for the methods defined in interface IF_OS_STATE.



Figure 11.5 Creating Persistent Class ZCL_OS_ADDRESS

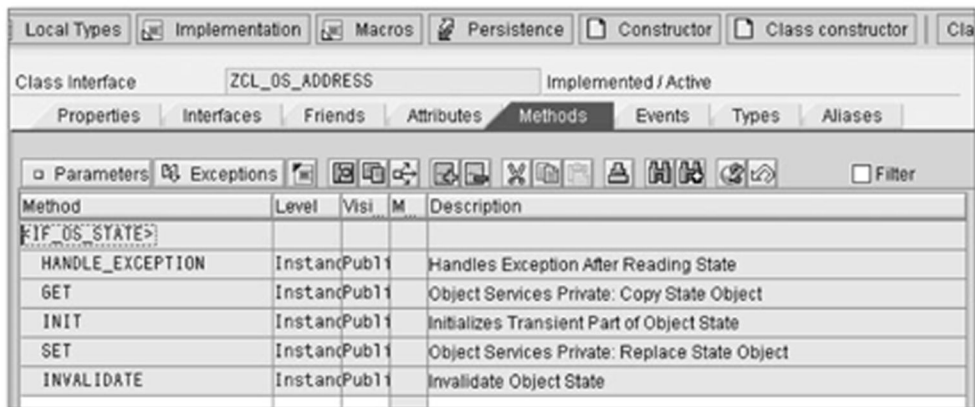


Figure 11.6 Editing a Persistent Class

11.3.2 Defining Persistent Attributes with the Mapping Assistant

After you have created a persistent class, you can define its persistence representation:

1. Click on the PERSISTENCE button in the toolbar of the Class Editor screen (see Figure 11.6). This takes you to the Mapping Assistant tool shown in Figure 11.7.
2. Initially, you are prompted to enter a table, view, or structure that will be used as the basis for the persistence representation. For example, in Figure 11.7, we

are defining the persistence representation for class `ZCL_OS_ADDRESS` using Table `ZCA_ADDRESS` (whose fields were shown earlier in Figure 11.4).

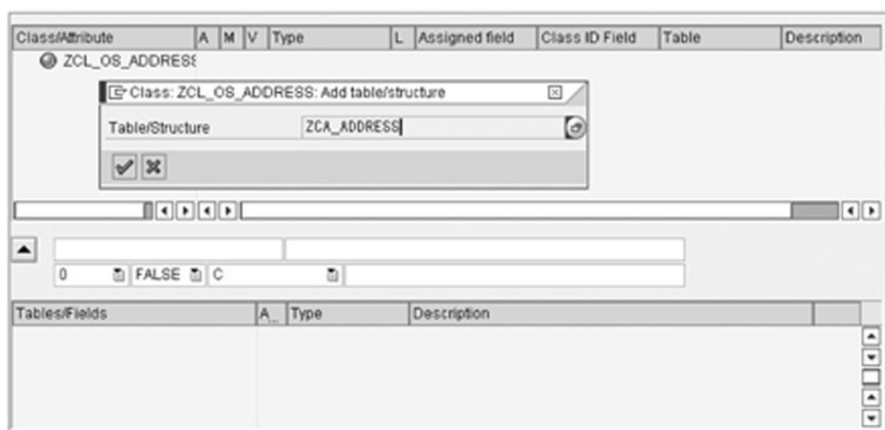


Figure 11.7 Defining Persistence with the Mapping Assistant

3. After you have selected the relevant ABAP Dictionary object, begin editing the persistent attributes of the class by double-clicking on any of the fields in the TABLES/FIELDS display in the bottom panel of the Mapping Assistant screen.
4. This loads the field into the editing area in the middle of the screen (see Figure 11.8). Edit the attribute name (which doesn't have to be the same as the table/structure field), description, visibility, accessibility, and assignment type.

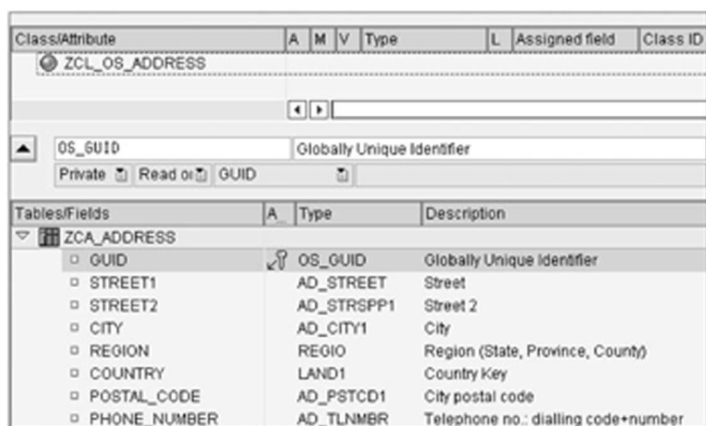


Figure 11.8 Mapping Persistent Class Attributes

5. In most cases, the default properties defined for a given field by the Mapping Assistant will be correct. Of course, you may decide to restrict access to a particular field by customizing the visibility and accessibility properties. Moreover, you may need to modify the assignment type for certain fields. Table 11.2 provides a description of the assignment types that you can configure for a given attribute.

Assignment Type	Meaning
Business Key	Derived by the Mapping Assistant for primary key fields of an ABAP Dictionary table that has a semantic primary key. You cannot change this particular assignment type.
GUID	Derived by the Mapping Assistant for the primary key field of an ABAP Dictionary table that has a GUID-based primary key. You cannot change this particular assignment type.
Value Attribute	Used to define nonkey attributes of a given ABAP Dictionary object.
Class Identifier	Used in conjunction with another table/structure field to uniquely identify an object reference. The table/structure field must be of type OS_GUID.
Object Reference	Used in conjunction with another table/structure field to uniquely identify an object reference. The table/structure field must be of type OS_GUID.

Table 11.2 Persistent Attribute Assignment Types

6. Figure 11.9 shows the completed persistence representation for class ZCL_OS_ADDRESS. After you have finished mapping the attributes, save your changes, and return to the Class Editor screen to activate the changes to your persistence class.

Class/Attribute	A	M	V	Type	L	Assigned field	Class ID Field	Table	Description
⊖ ZCL_OS_ADDRESS									
⊕ OS_GUID						GUID		ZCA_ADDRESS	Globally Unique Identifier
⊖ STREET1						STREET1		ZCA_ADDRESS	Street
⊖ STREET2						STREET2		ZCA_ADDRESS	Street 2
⊖ CITY						CITY		ZCA_ADDRESS	City
⊖ REGION						REGION		ZCA_ADDRESS	Region (State, Province, County)
⊖ COUNTRY						COUNTRY		ZCA_ADDRESS	Country Key
⊖ POSTAL_CODE						POSTAL_CODE		ZCA_ADDRESS	City postal code
⊖ PHONE_NUMBER						PHONE_NUMB.		ZCA_ADDRESS	Telephone no.: dialling code+number

Figure 11.9 Persistent Attribute Details for Class ZCL_OS_ADDRESS

7. If this is the first time you have activated your changes, you are asked whether or not you want to also activate the class actor (see Figure 11.10). Select the YES button to generate the class actors for class ZCL_OS_ADDRESS (i.e., classes ZCA_OS_ADDRESS and ZCB_OS_ADDRESS).

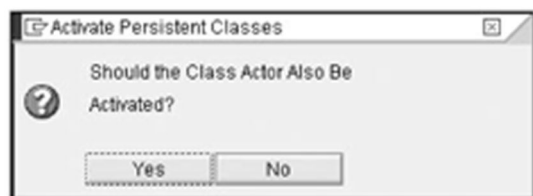


Figure 11.10 Activating the Class Actor for a Persistent Class

Figure 11.11 shows the generated getter and setter methods for class ZCL_OS_ADDRESS after it has been activated.

Class Interface		ZCL_OS_ADDRESS		Implemented / Active	
Properties Interfaces Friends Attributes Methods Events Types Aliases					
Parameters Exceptions Filter					
Method	Level	Visi	M	Description	
<ZIF_OS_STATE>					
HANDLE_EXCEPTION	Instan	Publ		Handles Exception After Reading State	
GET	Instan	Publ		Object Services Private: Copy State Object	
INIT	Instan	Publ		Initializes Transient Part of Object State	
SET	Instan	Publ		Object Services Private: Replace State Object	
INVALIDATE	Instan	Publ		Invalidate Object State	
SET_STREET2	Instan	Publ		Sets Attribute STREET2	
SET_STREET1	Instan	Publ		Sets Attribute STREET1	
SET_REGION	Instan	Publ		Sets Attribute REGION	
SET_POSTAL_CODE	Instan	Publ		Sets Attribute POSTAL_CODE	
SET_PHONE_NUMBER	Instan	Publ		Sets Attribute PHONE_NUMBER	
SET_COUNTRY	Instan	Publ		Sets Attribute COUNTRY	
SET_CITY	Instan	Publ		Sets Attribute CITY	
GET_STREET2	Instan	Publ		Reads Attribute STREET2	
GET_STREET1	Instan	Publ		Reads Attribute STREET1	
GET_REGION	Instan	Publ		Reads Attribute REGION	
GET_POSTAL_CODE	Instan	Publ		Reads Attribute POSTAL_CODE	
GET_PHONE_NUMBER	Instan	Publ		Reads Attribute PHONE_NUMBER	
GET_COUNTRY	Instan	Publ		Reads Attribute COUNTRY	
GET_CITY	Instan	Publ		Reads Attribute CITY	

Figure 11.11 GET and SET Methods for Class ZCL_OS_ADDRESS

11.3.3 Working with Object References

All of the attributes defined for class `ZCL_OS_ADDRESS` were based on simple, elementary table fields. However, there is a twist to class `ZCL_OS_PERSON`. Here, as you will recall, you want to provide the ability to associate an instance of class `ZCL_OS_ADDRESS` with the `ZCL_OS_PERSON` instance. To achieve this kind of binding, you need to map two foreign key fields of type `OS_GUID` to the `ZCL_OS_ADDRESS` type.

1. Create a persistent class called `ZCL_OS_PERSON` using the steps described in Section 11.3.1, *Creating a Persistent Class in the Class Builder*.
2. Map the fields of Table `ZCA_PERSON` just as you did in Section 11.3.2, *Defining Persistent Attributes with the Mapping Assistant*. However, be careful in the way that you map the `ADDR_CLASS` and `ADDR_REF` database fields. These two fields will be used together to uniquely define an object reference attribute called `ADDRESS`.
3. In Figure 11.12, notice that the `CLASS IDENTIFIER` assignment type has been used to define the source field in Table `ZCA_PERSON` that will provide the GUID of the persistent class pointed to by the object reference. This class GUID is quietly assigned to every global class that is defined in the Class Builder. As you will see, this low-level detail is transparent to users of the class agent API. Nevertheless, you still have to provide a mapping to this attribute for the object reference to be valid.

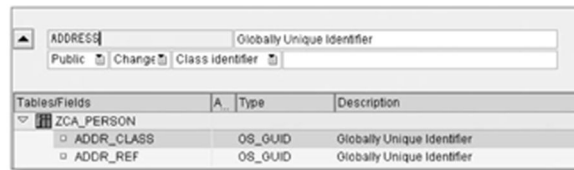


Figure 11.12 Setting the Class Identifier for an Object Reference

4. Finally, map the source field in Table `ZCA_PERSON` that will store the instance GUID of an actual persistent object of type `ZCL_OS_ADDRESS`. Here, notice that the object reference assignment is qualified with the `ZCL_OS_ADDRESS` class created earlier. This information is needed for the Class Builder to properly generate `SET_` and `GET_` methods for this object reference attribute. Figure 11.13 shows how this field was configured for class `ZCL_OS_PERSON`.

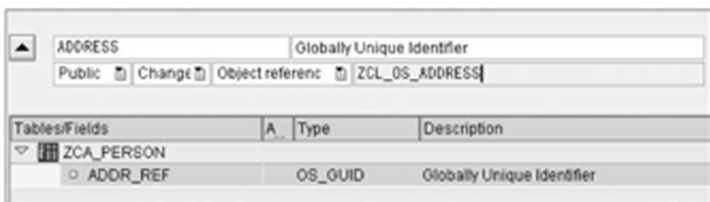


Figure 11.13 Defining the Object Reference

Figure 11.14 shows the completed attribute details for the ADDRESS object reference. You can also see the signature of the generated SET_ADDRESS and GET_ADDRESS methods in Figure 11.15 and Figure 11.16, respectively.

Class/Attribute	A	M	V	Type	L	Assigned field	Class ID Field	Table	Description
ZCL_OS_PERSON									
OS_GUID				GUID				ZCA_PERSON	Globally Unique Identifier
NAME_FIRST				AD_NAMEFIR		NAME_FIRST		ZCA_PERSON	First name
NAME_LAST				AD_NAMELAS		NAME_LAST		ZCA_PERSON	Last name
ADDRESS				ZCL_OS_ADD...		ADDR_REF	ADDR_CLASS	ZCA_PERSON	Globally Unique Identifier

Figure 11.14 Attribute Details for the ADDRESS Object Reference

Method parameters SET_ADDRESS							
Parameter	Type	Pass by Value	Optional	Typing Method	Associated Type	Default value	Description
_ADDRESS	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type Ref To	ZCL_OS_ADDRESS		Attribute Value
		<input type="checkbox"/>	<input type="checkbox"/>	Type			

Figure 11.15 Signature for Method SET_ADDRESS

Method parameters GET_ADDRESS							
Parameter	Type	Pass by Value	Optional	Typing Method	Associated Type	Default value	Description
RESULT	Returning	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Type Ref To	ZCL_OS_ADDRESS		Attribute Value
		<input type="checkbox"/>	<input type="checkbox"/>	Type			

Figure 11.16 Signature for Method GET_ADDRESS

11.4 Working with Persistent Objects

Now that we have defined a couple of persistent classes, let's look at ways to interact with persistent objects of these classes using the generated API methods of the associated class agents.

11.4.1 Creating New Persistent Objects

The code required to create a new persistent object is remarkably straightforward. Listing 11.1 shows how to create persistent objects of type `ZCL_OS_ADDRESS` and `ZCL_OS_PERSON`.

```

DATA: lr_address_agent TYPE REF TO zca_os_address,
      lr_address       TYPE REF TO zcl_os_address,
      lr_person_agent  TYPE REF TO zca_os_person,
      lr_person        TYPE REF TO zcl_os_person.

TRY.
* First, create an address instance:
lr_address_agent = zca_os_address=>agent.
CALL METHOD lr_address_agent->create_persistent
  EXPORTING
    i_city      = 'Mos Eisley'
    i_country   = 'TAT'
    i_phone_number = '999-555-5555'
    i_postal_code = '99999-9999'
    i_region    = 'SKY'
    i_street1   = '123 Millennium Falcon Way'
  RECEIVING
    result      = lr_address.

* Next, create a person instance:
lr_person_agent = zca_os_person=>agent.
CALL METHOD lr_person_agent->create_persistent
  EXPORTING
    i_address = lr_address
    i_name_first = 'Andersen'
    i_name_last = 'Wood'
  RECEIVING
    result      = lr_person.

* Must execute COMMIT WORK to persist the objects:
COMMIT WORK.
CATCH cx_os_object_existing.
ENDTRY.

```

Listing 11.1 Creating Persistent Objects

In both cases, a reference to the associated agent class had to be obtained via the `AGENT` class attribute defined for the respective agent classes (i.e., classes

ZCA_OS_ADDRESS and ZCA_OS_PERSON). These references were then used to call the CREATE_PERSISTENT method generated in the agent classes. Here, notice that the interface of the method has been enhanced to allow you to initialize value and reference attributes upon creation.

To actually store the objects in the database, the COMMIT WORK statement had to be issued. Note that the Persistence Service will not check to see if a persistent object that is mapped using business keys already exists before issuing an INSERT statement behind the scenes. Consequently, it is possible that the COMMIT WORK statement might generate an exception of type CX_OS_OBJECT_EXISTING.

However, because this exception is triggered in an update module, this exception type would never be caught in a CATCH block such as the one shown in Listing 11.1. Therefore, it is a good idea to get into the habit of checking to see if persistent objects of this type already exist using the GET_PERSISTENT method before trying to create a new persistent object with a particular business key.

11.4.2 Reading Persistent Objects Using the Query Service

If a persistence class is mapped using business keys, the Class Builder generates a method called GET_PERSISTENT in the agent class. The signature of the importing parameters for this method will match the semantic primary key fields defined in the target database table. Therefore, you can call this method to retrieve a particular persistent object using a business key.

Frequently, however, you might not have the full primary key at your disposal when you need to look up a persistent object. Moreover, if you have mapped your persistent classes by an instance-GUID, you need another way to access the persistent objects because it is unlikely that you will happen to have a hexadecimal key field just lying around. You can achieve this advanced kind of lookup behavior by using the *Query Service*.

As part of the ABAP Object Services framework, the Query Service allows you to search for and retrieve one or more persistent objects via queries based on *logical expressions*. These logical expressions are similar in nature to the ones that you are probably accustomed to using with the SQL SELECT statement. However, in terms of the Query Service, these expressions are encapsulated inside an object that implements the IF_OS_QUERY interface.

Queries are created via a *Query Manager* object that can be retrieved via a call to a class method called `GET_QUERY_MANAGER` in class `CL_OS_SYSTEM`. The Query Manager object contains a single instance method called `CREATE_QUERY` that you use to create the query instance. The `CREATE_QUERY` method provides parameters for defining *filters* and an *ordering* of the results. For more information about the various types of filter and sort conditions that you can use in your queries, consult the SAP online help documentation for the Query Service (<http://help.sap.com>).

The code snippet in Listing 11.2 shows how the Query Service can be used to look up persistent objects of type `ZCL_OS_PERSON`. This simple query contains a filter on the `NAME_FIRST` attribute of class `ZCL_OS_PERSON`. This query is passed into the Persistence Service via a call to the `GET_PERSISTENT_BY_QUERY` method (defined in interface `IF_OS_CA_PERSISTENCY`). This method returns an internal table of type `OSREFTAB`. The line type of this internal table is a reference to the generic type `OBJECT`. Consequently, a widening cast had to be performed to interface with the resultant persistent objects.

```
DATA: lr_agent      TYPE REF TO zca_os_person,
      lt_people     TYPE osreftab,
      lr_oref       TYPE REF TO object,
      lr_person     TYPE REF TO zcl_os_person,
      lr_query_mgr  TYPE REF TO if_os_query_manager,
      lr_query      TYPE REF TO if_os_query,
      lv_first_name TYPE string.

TRY.
* Create a new query based on the first name attribute of
* the "Person" persistent object:
lr_agent = zca_os_person=>agent.
lr_query_mgr = cl_os_system=>get_query_manager( ).
lr_query = lr_query_mgr->create_query(
    i_filter = 'NAME_FIRST = PA1' ).

* Retrieve the set of "Person" objects matching the filter
* condition:
lt_people =
    lr_agent->if_os_ca_persistency~get_persistent_by_query(
        i_query = lr_query
        i_par1 = 'Paige' ).
```

```

* Display the results:
LOOP AT lt_people INTO lr_oref.
  lr_person ?= lr_oref.
  lv_first_name = lr_person->get_name_first( ).
  lv_last_name = lr_person->get_name_last( ).
  WRITE: / 'Name is: ', lv_first_name, lv_last_name.
ENDLOOP.
CATCH cx_os_object_not_found.
CATCH cx_os_query_error.
ENDTRY.

```

Listing 11.2 Looking Up Person Objects Using the Query Service

11.4.3 Updating Persistent Objects

After you have obtained a reference to a persistent object, you can begin making changes to its persistent attributes by calling the generated setter methods. Listing 11.3 shows the updating of the CITY attribute of the ZCL_OS_ADDRESS persistent object associated with a person object retrieved via some kind of query.

```

DATA: lr_agent TYPE REF TO zca_os_person,
      lr_person TYPE REF TO zcl_os_person,
      lr_address TYPE REF to zcl_os_address.

TRY.
* Execute a query to retrieve a person object:
...

* Retrieve the address associated with that person:
lr_address = lr_person->get_address( ).

* Update the "CITY" attribute on the address object:
lr_address->set_city( 'Far Far Away' ).
COMMIT WORK.
CATCH cx_os_object_not_found.
CATCH cx_os_query_error.
ENDTRY.

```

Listing 11.3 Updating a Persistent Object

Here, as was the case when the persistent object was created originally, the COMMIT WORK statement must be executed to actually trigger the Persistence Service to propagate the changes to the database.

11.4.4 Deleting Persistent Objects

You can delete a persistent object using the `DELETE_PERSISTENT` method defined in the implemented interface `IF_OS_FACTORY`. Listing 11.4 shows how to call this method to delete a persistent object of type `ZCL_OS_PERSON`. Here, once again, the `COMMIT WORK` statement had to be executed to trigger the deletion within the Persistence Service.

```
DATA: lr_agent TYPE REF TO zca_os_person,
      lr_person TYPE REF TO zcl_os_person.

TRY.
* Execute a query to retrieve a person object:
...

* Delete the person object:
lr_agent->if_os_factory~delete_persistent( lr_person ).
COMMIT WORK.
CATCH cx_os_object_not_found.
CATCH cx_os_query_error.
ENDTRY.
```

Listing 11.4 Deleting a Persistent Object

11.5 UML Tutorial: Advanced Sequence Diagrams

In this section, we will look at some advanced features of the UML sequence diagram. As a frame of reference for this discussion, the sequence diagram example from Section 3.6, UML Tutorial: Sequence Diagrams, has been revised in Figure 11.17 to include some of the more advanced features that will be discussed in the upcoming subsections.

11.5.1 Creating and Deleting Objects

Within a given activation, it is not uncommon for a method to need to dynamically create another object to carry out a particular task. As you can see in Figure 11.17, the creation of an object is initiated by a message. The message name is optional, but the general convention is to name the message `new`. Here, notice that the object box for the `receipt` object is aligned with the creation message.

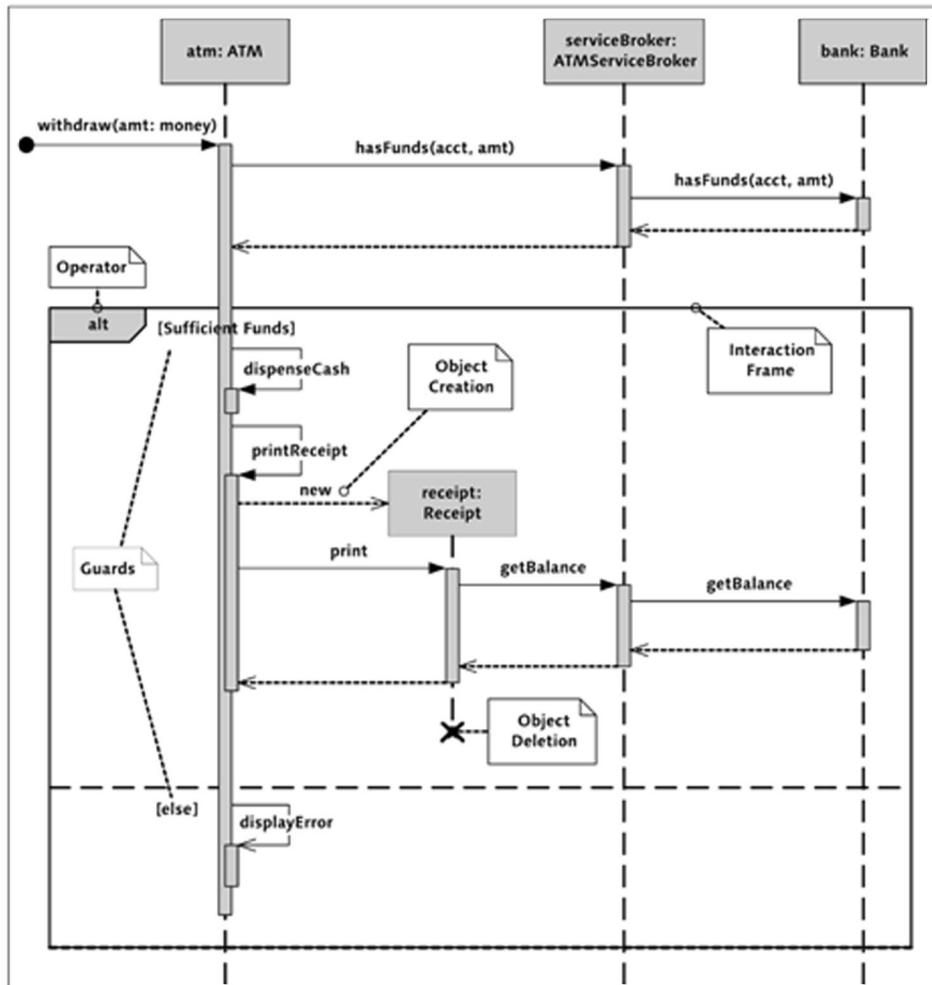


Figure 11.17 Sequence Diagram for Withdrawing Cash from an ATM

This notation helps to clarify the fact that the object did not exist whenever the interaction began. After an object is created, you can send messages to it just like any of the other objects in the sequence diagram.

If the created object is a temporary object (e.g., a local variable inside a method, etc.), then you can depict the deletion of the object by terminating the object life-line with an X (see Figure 11.17). It is also possible for one object to explicitly

delete another object by mapping a message from the requesting object to an X on the target object's lifeline.

11.5.2 Depicting Control Logic with Interaction Frames

As a rule, you typically do not want to depict a lot of control logic in a sequence diagram. However, it is sometimes helpful to include some high-level logic so that the interaction between the objects is clear. In UML 2.0, this control flow is depicted using *interaction frames*. An interaction frame partitions off a portion of the sequence diagram inside of a rectangular boundary. The functionality depicted in an interaction frame is described by an *operator* that is listed in the upper-left corner of the frame.

For example, the sequence diagram in Figure 11.17 shows an interaction frame that is using the `alt` operator. The `alt` operator is used to depict conditional logic such as an `IF...ELSE` or `CASE` statement. The branches of this conditional logic are divided by a horizontal dashed line. Furthermore, each branch of the logic also contains a conditional expression called a *guard*. As you would expect, guards control whether or not the control flows to a particular branch of the conditional logic. For instance, in the sequence diagram shown in Figure 11.17, the `atm` object only dispenses cash if there are sufficient funds in the account. Otherwise, an error message is displayed on the console.

Table 11.3 describes some of the basic operators that can be used with interaction frames. Again, note that interaction frames should be used very sparingly. If you need to depict more complex logic, consider using an activity diagram or even some basic pseudocode.

Operator	Usage Type
<code>alt</code>	Used to depict conditional logic such as an <code>IF...ELSE</code> or <code>CASE</code> statement.
<code>opt</code>	Used to depict an optional piece of logic such as a basic <code>IF</code> statement.
<code>par</code>	Used to depict parallel behavior. In this case, each fragment in the interaction frame runs in parallel.
<code>loop</code>	Used to depict various types of looping structures (i.e., <code>LOOP</code> , <code>DO</code> , etc.).

Table 11.3 Interaction Frame Operators

Operator	Usage Type
ref	Used to reference an interaction defined on another sequence diagram.
sd	Used to surround an embedded sequence diagram within the current sequence diagram.

Table 11.3 Interaction Frame Operators (cont.)

11.6 Summary

In this chapter, you learned how the ABAP Object Services framework could be used to develop a robust persistence layer based solely upon ABAP Object classes. Here, once again, you saw the power of object-oriented programming with the seamless integration of custom persistent classes into the Persistence Service framework.

In many respects, we barely scratched the surface with regards to the functionality provided by the ABAP Object Services framework. In particular, we did not have time to consider the features of the Transaction Service that makes it possible to integrate your persistent objects with the SAP transaction concept. As you play around with some persistent object examples, read through the online help documentation to learn more about some of these advanced features.

In the next chapter, you will learn how to work with XML documents using ABAP Objects classes.

XML is a meta-markup language used to define structured documents. In its short history, XML has quickly become the “lingua franca” for information exchange in the IT industry. In this chapter, you will learn how to work with XML using the ABAP Objects-based iXML Library.

12 Working with XML

The *extensible markup language* (XML) is a meta-markup language used to define structured documents. Endorsed by the *World Wide Web Consortium* (<http://www.w3.org>), XML evolved as a subset of the *Standard Generalized Markup Language* (SGML) in the late 1990s to provide a standard for building structured documents that can be exchanged over the Internet. Since that time, XML use has proliferated across many diverse application domains. These days, XML is everywhere.

In this chapter, you will learn how to work with XML using the object-oriented iXML library provided with the SAP NetWeaver Application Server. If you have never worked with XML before, don't worry; we will spend some time exploring some basic XML concepts before diving into the XML programming model. To demonstrate these concepts, we will develop a couple of examples that show you how to create and consume XML documents within an ABAP program.

12.1 XML Overview

When you take away all of the hype, XML is basically just a standard that can be used to define the format of text-based documents. Initially, this may seem disappointing as XML doesn't seem to *do anything* in and of itself. Nevertheless, XML does serve a very important purpose in the Information Age that we live in today: to structure and organize various types of data. This simple capability is part of the foundation upon which many new and exciting applications are being developed.

12.1.1 Why Do We Need XML?

Before we proceed too much further, it is important to clarify the positioning of the XML so that you can understand what it is used for. As mentioned before, XML is a *meta-markup language*. This definition is a mouthful that requires some explanation. A *markup language* is used to *mark up* a document with instructions that define how its content is organized, formatted, and so on. Typically, markup languages use special annotations called *tags* for this purpose. Tags are used to label and categorize information within a document.

If you have ever worked with HTML (*Hypertext Markup Language*) before, you have already encountered a markup language that uses tags to format its content. HTML is used to define the structure and format of Web pages. Figure 12.1 shows an example of some HTML markup. Here, you can see that tags (or *elements*) are escaped using the less-than and greater-than characters (< and >). For example, the tags <head> and </head> define the *heading* section of the HTML document.

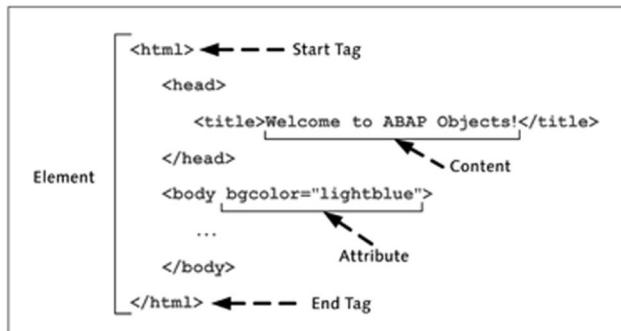


Figure 12.1 An HTML Document Markup Example

As you will see in Section 12.1.2, Understanding XML Syntax, the syntax shown in the HTML markup from Figure 12.1 is very similar to the syntax used to create XML markup. However, as stated earlier, XML is a *meta-markup language* as opposed to just a regular markup language. In essence, this implies that XML does not define any particular markup (i.e., defined tags such as <head>, <body>, etc.). Rather, it is used to define *other* markup languages such as XHTML.

To achieve this lofty goal, the XML standard was designed to be extremely flexible. This flexibility supports the creation of a wide variety of document formats,

allowing you to customize the format so that *form follows function*. In other words, because the XML standard has very little to say about the elements and content of a document, you don't have to try and bend an application data model to fit within the confines of some ill-fitting standard. Instead, you can define the document format using domain-specific terms that help to make the document *self-describing*. This characteristic of XML makes XML documents much easier to read and interpret for humans and computers alike.

Much of the beauty of XML lies in its simplicity. Rather than defining a complex or proprietary binary file format, the designers of the XML standard defined an open, text-based format that provides support for multiple languages with Unicode. The openness of the XML standard simplifies the process of exchanging information between heterogeneous systems. This is perhaps best evidenced by the recent explosion of Web Service technologies that use XML to define protocols for message exchange, and so on.

12.1.2 Understanding XML Syntax

XML syntax is designed to be relatively straightforward and easy to learn. XML documents are organized into a series of *elements*. The basic syntax for defining an element in XML is shown in Listing 12.1.

```
<element_name [attribute_name="attribute_value"...]>
  <!-- Element Content -->
</element>
```

Listing 12.1 Basic Syntax for Defining an XML Element

You can also define empty elements (i.e., elements without any content) using the syntax shown in Listing 12.2. As you can see, because these elements do not have any content, there is no need for a closing tag. Instead, the element tag is marked as complete with the `/>` characters.

```
<element_name [attribute_name="attribute_value"...] />
```

Listing 12.2 Syntax for Defining Empty Elements in XML

Elements can be *nested*. In other words, the content of an element can in turn contain additional *child* elements. Consequently, XML documents form a tree-like structure beginning with a single *root* element. As you would expect, root elements do not have a parent element.

To demonstrate this concept, let's consider an XML document that is used to represent a reading list. The document in Listing 12.3 contains a reading list for developers interested in learning more about ABAP programming.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is an XML comment. -->
<ReadingList>
  <Topic>ABAP Programming</Topic>
  <RecommendedBooks>
    <Book isbn="978-1-59229-039-0">
      <Title>The Official ABAP Reference</Title>
      <Author>Horst Keller</Author>
      <Publisher>SAP PRESS</Publisher>
    </Book>

    <Book isbn="978-1-59229-139-7">
      <Title>Next Generation ABAP Development</Title>
      <Author>Rich Heilman</Author>
      <Author>Thomas Jung</Author>
      <Publisher>SAP PRESS</Publisher>
    </Book>
  </RecommendedBooks>
</ReadingList>
```

Listing 12.3 A Sample XML Document

The XML document shown in Listing 12.3 begins with the optional XML *declaration statement*. The XML declaration statement is used to provide some basic information about the version of the XML standard being used as well as the character encoding of the actual content to assist XML processors in interpreting the data. The syntax for an XML declaration statement is `<?xml version="1.0" encoding="..."?>`. The next statement demonstrates how comments can be included in an XML document. The basic syntax for creating comments in XML is `<!-- Comment Text -->`.

The root element of the sample reading list document is appropriately called `<ReadingList>`. The `<ReadingList>` element contains two child elements called `<Topic>` and `<RecommendedBooks>`. Similarly, the `<RecommendedBooks>` element can contain multiple `<Book>` elements (which are also complex elements in their own right). Within a given element, you can see that plain text content or additional child elements are defined.

Another aspect of XML syntax shown in Listing 12.3 is the definition of an *attribute*. An attribute is a name-value pair that describes a property of an XML element. For example, in Listing 12.3, an attribute called `isbn` qualifies a `<Book>` element with an ISBN (*International Standard Book Number*). The ISBN number could have been captured inside of a child element just as easily. However, it is often handy to have attribute details defined inline within the element itself rather than having to traverse further down the document tree to find the information.

Even in a basic XML document such as the one shown in Listing 12.3, it is easy to see how deep and complex an XML document can become. Fortunately, computers are quite good at processing tree-like data structures such as the ones defined in XML documents. You will see an example of this in Section 12.5, Case Study: Reading an XML Document. Figure 12.2 shows the XML tree for the sample document from Listing 12.3.

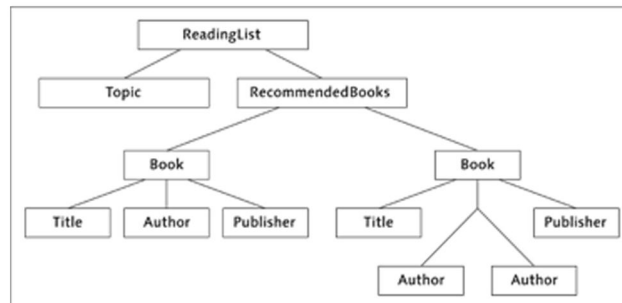


Figure 12.2 XML Tree for Example Reading List Document

XML markup is case sensitive. For example, the element names `<ReadingList>`, `<READINGLIST>`, and `<readinglist>` are all distinct within an XML document. The typical convention is to define XML markup using *camel case*. With camel case, you write compound words or phrases by capitalizing the first character in every word (e.g., `ReadingList`).

12.1.3 Defining XML Semantics

XML documents that follow the syntax rules described in Section 12.1.2, Understanding XML Syntax, are considered to be *well-formed*. Strict enforcement of

XML syntax rules makes the job of interpreting documents much easier for an XML processor because it eliminates a lot of guesswork. Still, to effectively exchange documents encoded in XML, you need a way to describe the document so that all parties involved in the message exchange know what format to expect. Two of the most common methods for describing XML documents are *Document Type Definitions (DTDs)* and *XML Schema*.

DTDs and XML Schema are examples of languages that are used to specify an XML schema. Here, the term *schema* refers to the format or outline of the content within an XML document. Schema languages are used to place *constraints* on a document to make sure that the document is *valid* according to an agreed-upon standard. For example, many industry sectors are using XML Schema languages to define standard formats for various common document types (e.g., invoices, sales orders, etc.).

Due to its more advanced capabilities, the XML Schema language has surpassed DTDs as the standard for defining XML Schemas. Although the description of the syntax of XML Schema documents is outside the scope of this book, Listing 12.4 provides an example of an XML Schema document for the reading list example that you saw in Listing 12.3. This schema document places constraints on the reading list, specifying the data types of particular elements, their cardinalities, and so on. In a simple example such as this, few constraints are placed on the contents of each element. For instance, only `<Author>` elements have been specified to contain content of type `xsd:string`. In a real-world scenario, you might place limitations on the length of the name, and so on.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.sap-press.com"
  targetNamespace="http://www.sap-press.com">

  <!-- Definition of "ReadingList" root element -->
  <xsd:element name="ReadingList">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Topic" type="xsd:string"
          minOccurs="1" />
        <xsd:element name="RecommendedBooks"
          type="tns:RecommendedBooks" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

    </xsd:complexType>
</xsd:element>

<xsd:complexType name="RecommendedBooks">
  <xsd:sequence>
    <xsd:element name="Book" type="tns:Book"
      minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Book">
  <xsd:attribute name="isbn" type="xsd:string"
    use="required" />
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"
      minOccurs="1" />
    <xsd:element name="Author" type="xsd:string"
      minOccurs="1" maxOccurs="unbounded" />
    <xsd:element name="Publisher" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Listing 12.4 XML Schema for Reading List Example Document

12.2 XML Processing Concepts

One of the many benefits of developing a standard for document markup is the fact that independent software vendors have the opportunity to develop powerful document processing tools that can be used to process any valid XML document. In this section, we will look at how to process XML using a *parser*. Along the way, you will also learn about the XML processing tools available in the AS ABAP.

12.2.1 Processing XML Using a Parser

Whenever you work with an external data source such as a file, you must implement code to convert the raw content of the file into data variables and vice versa. Oftentimes, this conversion logic is hard coded into the program and therefore cannot be easily reused. For example, to properly interpret a file with fixed-

length data records, the code must be smart enough to understand which positions in a file record correspond with particular variables, and so on. In technical terms, this conversion process is referred to as *parsing*. A parser receives input from a data stream, verifies that the data has the correct syntax, and then copies that data into a more user-friendly data structure.

The self-describing nature of XML makes it possible to define generic parsers that can be used to parse any well-formed XML document. Two of the more popular processing models for XML parsers are the *Simple API for XML (SAX)* and *Document Object Model (DOM)* APIs. We will look at the DOM processing model in the next section. For more information about SAX, read *Learning XML* (O'Reilly, 2003).

12.2.2 Modeling XML with the DOM

The DOM represents an XML document as a tree-like data structure. Each element in the XML document is represented as a *node*, beginning with the root node. From the root node, the DOM API provides methods for traversing the tree, reading child elements and their attributes, and so on. One of the nice features of DOM is that the entire document is read into memory, allowing you to manipulate the document as a whole. Of course, this can also be a detriment if you are working with large documents. In this case, you may prefer to work with the SAX API or another processing model.

12.2.3 Features of the iXML Library

The iXML library is an ABAP Objects-based API that provides ABAP developers with access to an XML 1.0-compliant parser. This parser supports the DOM-based processing model, allowing you to both consume and construct XML documents using the DOM API. In addition, the iXML library also provides XML rendering services that assist you in rendering (or converting) XML content into various formats, encodings, and so on.

The iXML library was made available with the 4.6C release of the Basis kernel (which was the predecessor of the SAP Web/SAP NetWeaver Application Server). In the next three sections, we will use the iXML library to build and manipulate some sample XML documents.

12.3 Case Study: Developing a Reading List ADT

Now that you have some understanding of what XML is all about, we can begin to develop some examples that use the iXML library. In this section, we will begin by developing an abstract data type (ADT) class to represent the reading list example described in Section 12.1.2, Understanding XML Syntax. Here, we want to stay consistent with our object-oriented design principles by encapsulating the XML details inside the class.

Throughout the course of this book, we have typically represented the internal state of the classes using basic ABAP data types. However, while this approach is often convenient, it is not a hard requirement. For example, in the case of our `Reading List` class, we will prefer to store the list in a DOM-based data structure. Of course, good object-oriented design principles will ensure that these details are hidden from the end user.

The UML class diagram shown in Figure 12.3 shows the core iXML classes and interfaces that we will be using to develop our `Reading List` class. As you can see, the iXML library makes heavy use of interfaces. In fact, the iXML design is a classic example of the principle of *programming to an interface*. The basic idea here is to provide an API based on generic interfaces rather than concrete implementations. This gives SAP the flexibility to change the way the iXML library is implemented (perhaps by plugging in a different XML parser, etc.) without affecting users of the API.

The starting point for working with the iXML library is the core `IF_IXML` interface. You can obtain a reference to an object that implements this interface by calling the class method `CREATE` of factory class `CL_IXML`. As you can see in Figure 12.3, the `IF_IXML` interface defines a series of factory methods that can be used to obtain references to various objects and services within the iXML library.

After you have a reference to the `IF_IXML` factory object, you have two options for creating a DOM-based XML document:

- ▶ If you want to load an XML document from an external data source (e.g., a file), you must first read that document into an *input stream* and then parse it using the XML parser provided with the iXML library. You can obtain references to the `IF_IXML_ISTREAM` and `IF_IXML_PARSER` objects using the instance methods `CREATE_STREAM_FACTORY` and `CREATE_PARSER` of interface `IF_IXML`.

- ▶ If you want to create a new XML document from scratch, you can call instance method `CREATE_DOCUMENT` of interface `IF_IXML`.

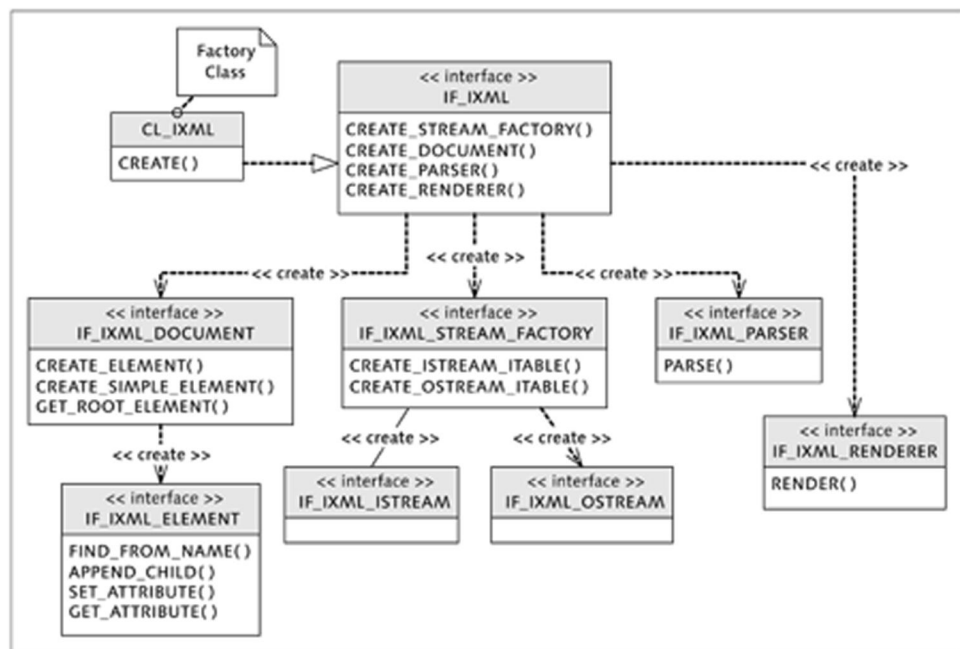


Figure 12.3 UML Class Diagram for Core iXML Classes and Interfaces

Regardless of the approach, the result of either of these creation operations is a DOM-based XML document instance that implements the `IF_IXML_DOCUMENT` interface. From here, we can use the methods defined in this interface to create and manipulate the individual elements of the document. Elements in the DOM-based document are represented as objects that implement the `IF_IXML_ELEMENT` interface.

At this point, we should have enough of a foundation to develop our `Reading List` class. The code in Listing 12.5 defines a local class called `lcl_reading_list` that we will use to represent our `Reading List` ADT. One of the first things to point out here is that the instantiation context for this class has been defined using the `CREATE PRIVATE` addition. This restriction forces users to create a new reading list using the creational class methods `create_new_list` and `create_from_file`. Method `create_new_list` can be used to generate a new list without any recommended books. Method `create_from_file` can be used to load a pre-existing XML-based reading list into context.

```

CLASS lcl_reading_list DEFINITION CREATE PRIVATE.
PUBLIC SECTION.
CLASS-METHODS:
    class_constructor,
    create_new_list IMPORTING im_topic TYPE string
                   RETURNING VALUE(re_list)
                   TYPE REF TO lcl_reading_list,
    create_from_file RETURNING VALUE(re_list)
                   TYPE REF TO lcl_reading_list
                   RAISING cx_ixml_parse_error.

METHODS:
    add_book IMPORTING im_isbn    TYPE string
              im_title    TYPE string
              im_authors  TYPE string_table
              im_publisher TYPE string,
    display,
    serialize RAISING cx_sy_file_io.

PRIVATE SECTION.
TYPE-POOLS: ixml.
TYPES: ty_xml_line(1024) TYPE x.

CONSTANTS: CO_OPEN_DIALOG TYPE i VALUE 1,
            CO_SAVE_DIALOG TYPE i VALUE 2.

CLASS-DATA:
    factory          TYPE REF TO if_ixml,
    stream_factory  TYPE REF TO if_ixml_stream_factory.

DATA: xml_document TYPE REF TO if_ixml_document,
      list_node    TYPE REF TO if_ixml_element,
      books_node   TYPE REF TO if_ixml_element.

METHODS:
    constructor,
    upload_list_file RAISING cx_ixml_parse_error,
    get_filename    IMPORTING im_dialog_type TYPE i
                   RETURNING VALUE(re_filename)
                   TYPE string.
ENDCLASS.

```

Listing 12.5 Defining Local Class LCL_READING_LIST

In addition to the creational methods, class `lcl_reading_list` also defines other methods to add books to the reading list, display the reading list in a browser, and store a revised reading list to an XML file. We will investigate these methods in detail in the following sections.

Finally, you can see that a series of class and instance attributes have been defined based on some of the iXML interface types shown in Figure 12.3. The creation and usage of these interface reference attributes is also demonstrated in the following sections.

12.4 Case Study: Building an XML Document

In this section, we will look at how to create a new XML document using the iXML library and the `lcl_reading_list` class defined in Listing 12.5. The process is relatively straightforward:

1. Obtain a reference to the iXML factory object.
2. Use that factory object to create a new DOM-based XML document instance.
3. Manipulate that XML document by creating various elements.

This basic logic is demonstrated in the implementation part of class `lcl_reading_list` shown in Listing 12.6.

```

CLASS lcl_reading_list IMPLEMENTATION.
  METHOD class_constructor.
    * Obtain a reference to the iXML factory object:
    factory = cl_ixml=>create( ).

    * Obtain a reference to the iXML stream factory:
    stream_factory =
      factory->create_stream_factory( ).
  ENDMETHOD.

  METHOD constructor.
    * Create an instance of the reading list document:
    xml_document = factory->create_document( ).
  ENDMETHOD.

  METHOD create_new_list.

```

```

*   Use the private constructor to create an instance
*   of a DOM-based XML document:
    CREATE OBJECT re_list.

*   Build the "ReadingList" root element:
    re_list->list_node =
        re_list->xml_document->create_simple_element(
            name = 'ReadingList'
            parent = re_list->xml_document ).

*   Build the "Topic" element:
    re_list->xml_document->create_simple_element(
        name = 'Topic'
        value = im_topic
        parent = re_list->list_node ).

*   Build the "RecommendedBooks" element:
    re_list->books_node =
        re_list->xml_document->create_simple_element(
            name = 'RecommendedBooks'
            parent = re_list->list_node ).
    ENDMETHOD.
    ...
ENDCLASS.

```

Listing 12.6 Creating a New Reading List Document

In the implementation code shown in Listing 12.6, notice that a reference to the `iXML` `factory` attribute is obtained inside the `class_constructor` method as opposed to the instance constructor method. Here, it made sense to share the `factory` attribute across all object instances. Similarly, a reference to the `iXML` stream factory is obtained in the `stream_factory` class attribute. You will see how I/O streams are used a little bit later on.

Users can create a new reading list by calling the class method `create_new_list`. This method defers the creation of the `IF_IXML_DOCUMENT` instance (i.e., attribute `xml_document`) to the private `constructor` method. After the DOM-based document is created, you can create the `<ReadingList>` root element using the instance method `CREATE_SIMPLE_ELEMENT` of interface `IF_IXML_DOCUMENT`. This method returns a reference to an `IF_IXML_ELEMENT` object that is stored in the `list_node` instance attribute. From here, you can recursively call the

`CREATE_SIMPLE_ELEMENT` method to create the child elements `<Topic>` and `<RecommendedBooks>` for the `<ReadingList>` root node. The topic of the reading list is provided via the `im_topic` importing parameter of method `create_new_list`.

The method `create_new_list` returns an instance of type `lcl_reading_list`. Initially, this reading list is empty. To add books to the reading list, users must call the instance method `add_book` whose implementation is shown in Listing 12.7.

```

CLASS lcl_reading_list IMPLEMENTATION.
...
METHOD add_book.
*   Method-Local Data Declarations:
    DATA: lr_book_node   TYPE REF TO if_ixml_element,
           lv_author      TYPE string.

*   Build the new "Book" element:
    lr_book_node =
        xml_document->create_element( name = 'Book' ).

    lr_book_node->set_attribute(
        name = 'isbn' value = im_isbn ).

    xml_document->create_simple_element(
        name = 'Title'
        value = im_title
        parent = lr_book_node ).

    xml_document->create_simple_element(
        name = 'Publisher'
        value = im_publisher
        parent = lr_book_node ).

    LOOP AT im_authors INTO lv_author.
        xml_document->create_simple_element(
            name = 'Author'
            value = lv_author
            parent = lr_book_node ).
    ENDLLOOP.

*   Add the "Book" element to the book collection:
    books_node->append_child( lr_book_node ).

```

```

    ENDMETHOD.
    ...
ENDCLASS.

```

Listing 12.7 Adding Books to the Reading List

This method starts off by generating a new <Book> element using the `CREATE_ELEMENT` method of interface `IF_IXML_DOCUMENT`. At this point, this is a standalone element that is not associated with the reading list. However, before appending the <Book> element to the list, the relevant attribute and child elements need to be generated for the book. Here, the `isbn` attribute is set using the instance method `SET_ATTRIBUTE` defined in interface `IF_IXML_ELEMENT`. Next, the <Title>, <Publisher>, and <Author> elements are set using method `CREATE_SIMPLE_ELEMENT`. Finally, after the <Book> element is constructed, it is appended to the <RecommendedBooks> node by calling instance method `APPEND_CHILD` on the `books_node` instance attribute.

After the reading list has been generated, we want to be able to save the list as an XML file on the user's workstation. Listing 12.8 shows how a public instance method called `serialize` has been defined for this purpose.

```

CLASS lcl_reading_list IMPLEMENTATION.
    ...
    METHOD serialize.
*      Method-Local Data Declarations:
      DATA: lr_ostream TYPE REF TO if_ixml_ostream,
             lt_xml     TYPE TABLE OF ty_xml_line,
             lr_renderer TYPE REF TO if_ixml_renderer,
             lv_retcode TYPE i,
             lv_filesize TYPE i,
             lv_filename TYPE string.

*      Prompt the user for the name of the output file:
      lv_filename = get_filename( CO_SAVE_DIALOG ).

*      Create an XML output stream:
      lr_ostream =
        stream_factory->create_ostream_itable(
          table = lt_xml ).

*      Render the XML document into the XML output stream:

```

```

lr_renderer =
  factory->create_renderer(
    ostream = lr_ostream
    document = xml_document ).
lv_retcode = lr_renderer->render( ).
IF lv_retcode NE 0.
  RAISE EXCEPTION TYPE cx_sy_file_io
  EXPORTING
    textid = cx_sy_file_io=>write_error
    filename = lv_filename.
ENDIF.

* Download the XML file to the user's workstation:
lv_filesize = lr_ostream->get_num_written_raw( ).

CALL METHOD cl_gui_frontend_services=>gui_download
  EXPORTING
    bin_filesize = lv_filesize
    filename = lv_filename
    filetype = 'BIN'
  CHANGING
    data_tab = lt_xml
  EXCEPTIONS
    others = 24.

IF sy-subrc NE 0.
  RAISE EXCEPTION TYPE cx_sy_file_io
  EXPORTING
    textid =
      cx_sy_file_io=>cx_sy_file_access_error
    filename = lv_filename.
ENDIF.
ENDMETHOD.

METHOD get_filename.
* Method-Local Data Declarations:
DATA: lt_files TYPE filetable,
      lv_retcode TYPE i,
      lv_path TYPE string,
      lv_fullpath TYPE string.

* Let the user select a file from his workstation:

```



```

CASE im_dialog_type.
  WHEN CO_OPEN_DIALOG.
    CALL METHOD
      cl_gui_frontend_services=>file_open_dialog
    CHANGING
      file_table = lt_files
      rc         = lv_retcode
    EXCEPTIONS
      others     = 5.

    READ TABLE lt_files INDEX 1 into re_filename.
  WHEN CO_SAVE_DIALOG.
    CALL METHOD
      cl_gui_frontend_services=>file_save_dialog
    CHANGING
      filename = re_filename
      path     = lv_path
      fullpath = lv_fullpath
    EXCEPTIONS
      others   = 4.
ENDCASE.
ENDMETHOD.
...
ENDCLASS.

```

Listing 12.8 Serializing the Reading List to an XML File

The first task of method `serialize` is to create an *output stream* instance. The term *stream* refers to a sequence of bytes. Therefore, an iXML output stream refers to a sequence of bytes that are being rendered as an ABAP data object. In method `serialize`, the raw XML content is bound into an internal table variable called `lt_xml`. After the output stream is created, the next step is to *render* (or convert) the DOM-based XML document into that output stream object. This is achieved via an XML renderer object that implements the `IF_IXML_RENDERER` interface. This interface defines an instance method called `RENDER` that is used to serialize the XML document into the `lt_xml` internal table variable. Finally, the standard `CL_GUI_FRONTEND_SERVICES` class features are used to allow users to save the XML file to their local workstations.

The report program `ZIXMLWRITER` shown in Listing 12.9 demonstrates how to use the `lcl_reading_list` class to create and serialize a reading list document. It begins by creating a new ABAP Programming reading list using the `create_`

`new_list` creational method. Next, the title ABAP Basics is added to the list by calling method `add_book`. Finally, the reading list is saved to a file by calling method `serialize`. In the next section, you will learn how to parse this XML file using the parser provided with the iXML library.

```
REPORT zixmlwriter.

INCLUDE: zxmllib.                "Reading List ADT
DATA: gr_reading_list TYPE REF TO lcl_reading_list,
      lt_authors      TYPE string_table.

START-OF-SELECTION.
* Create a new "ABAP Programming" reading list:
gr_reading_list =
  lcl_reading_list=>create_new_list(
    'ABAP Programming' ).

* Add a book to the list:
APPEND 'Guenther Faerber' TO lt_authors.
APPEND 'Julia Kirchner' TO lt_authors.

CALL METHOD gr_reading_list->add_book
  EXPORTING
    im_isbn      = '978-1-59229-153-3'
    im_title     = 'ABAP Basics'
    im_authors   = lt_authors
    im_publisher = 'SAP PRESS'.

* Serialize the generated XML file:
gr_reading_list->serialize( ).
```

Listing 12.9 Sample Program to Test the Creation of a Reading List

12.5 Case Study: Reading an XML Document

In the previous section, you learned how to interact with the DOM API to generate a brand new XML document. You also saw how to serialize the XML document into an output file. Frequently, you may want to read and/or modify that file again later. To read the XML file back into context, you must enlist the aid of the XML parser. Listing 12.10 shows how to implement this logic inside of the

creational class method `create_from_file`, which delegates much of the heavy lifting to the private helper method `upload_list_file`.

```

CLASS lcl_reading_list IMPLEMENTATION.
...
METHOD create_from_file.
*   Create an instance of the reading list:
    CREATE OBJECT re_list.

*   Upload a pre-existing reading list file into context:
    re_list->upload_list_file( ).

*   Obtain references to the index nodes:
    re_list->list_node =
        re_list->xml_document->get_root_element( ).

    re_list->books_node =
        re_list->list_node->find_from_name(
            name = 'RecommendedBooks' ).
ENDMETHOD.

METHOD upload_list_file.
*   Method-Local Data Declarations:
    DATA: lt_files      TYPE filetable,
           lv_retcode   TYPE i,
           lv_filename  TYPE string,
           lv_filesize  TYPE i,
           lt_xml       TYPE TABLE
                       OF ty_xml_line,
           lr_istream   TYPE REF TO if_ixml_istream,
           lr_parser    TYPE REF TO if_ixml_parser,
           lr_parse_error TYPE REF TO if_ixml_parse_error,
           lv_error_code TYPE i,
           lv_reason     TYPE string,
           lv_line       TYPE i,
           lv_column    TYPE i.

*   Let the user select a file on his workstation:
    lv_filename = get_filename( CO_OPEN_DIALOG ).

*   Upload the file from the client's workstation:
    CALL METHOD cl_gui_frontend_services=>gui_upload

```

```

EXPORTING
    filename = lv_filename
    filetype = 'BIN'
IMPORTING
    filelength = lv_filesize
CHANGING
    data_tab = lt_xml
EXCEPTIONS
    others = 19.

* Convert the raw XML content into an XML input stream:
lr_istream =
    stream_factory->create_istream_itable(
        table = lt_xml
        size = lv_filesize ).
IF lr_istream IS INITIAL.
    RAISE EXCEPTION TYPE cx_ixml_parse_error
    EXPORTING
        textid =
            cx_ixml_parse_error=>cx_ixml_parse_error
        code = sy-subrc
        reason = 'Could not read the XML list file!'.
ENDIF.

* Parse the XML file into a DOM-based XML document:
lr_parser =
    factory->create_parser(
        stream_factory = stream_factory
        istream = lr_istream
        document = xml_document ).

* Check the results:
lv_retcode = lr_parser->parse( ).
IF lv_retcode NE 0.

* Propagate the parse exception to the caller:
lr_parse_error = lr_parser->get_error( 0 ).
lv_error_code = lr_parse_error->get_number( ).
lv_reason = lr_parse_error->get_reason( ).
lv_line = lr_parse_error->get_line( ).
lv_column = lr_parse_error->get_column( ).

```

```

RAISE EXCEPTION TYPE cx_ixml_parse_error
EXPORTING
    textid =
        cx_ixml_parse_error=>cx_ixml_parse_error
    code   = lv_error_code
    reason = lv_reason
    line   = lv_line
    column = lv_column.
ENDIF.
ENDMETHOD.
...
ENDCLASS.

```

Listing 12.10 Parsing an XML-Based Reading List File

Listing 12.10 again uses the standard class `CL_GUI_FRONTEND_SERVICES` to read the XML file into an internal table variable called `lt_xml`. Next, an *input stream* object is created that will be used to feed the raw input from the XML file into the XML parser. After that, a reference to an XML parser instance is obtained that implements the `IF_IXML_PARSER` interface. Finally, the `PARSE` method defined in the `IF_IXML_PARSER` interface is used to parse the XML document. This method produces a return code that indicates whether or not the parser was able to successfully parse the document. In the event that there are parsing errors (due to malformed XML content, for instance), you can call the `GET_ERROR` method to obtain a reference to an object that implements the `IF_IXML_PARSE_ERROR` interface. This object provides various methods that provide details about the error such as the line number/column where the error occurred, a text-based description of the error, and so on.

If there are no errors during the parsing process, the generated DOM-based XML document will be stored in the `xml_document` instance attribute. For convenience, the `create_from_file` method also looks up the `<ReadingList>` and `<Recommended Books>` elements and stores references to them inside the instance attributes `list_node` and `books_node`, respectively. Caching this information makes it easier to modify the XML document later via public instance methods, and so on.

To verify that the XML document is parsing correctly, a `display` method has been implemented that can be used to display the derived XML document in a browser window. This display functionality is driven by the standard function module `SDIXML_DOM_TO_SCREEN` (see Listing 12.11).

```

CLASS lcl_reading_list IMPLEMENTATION.
  ...
  METHOD display.
*   Display the XML document in a browser:
    CALL FUNCTION 'SDIXML_DOM_TO_SCREEN'
      EXPORTING
        document    = xml_document
      EXCEPTIONS
        no_document = 1
        others      = 2.
  ENDMETHOD.
  ...
ENDCLASS.

```

Listing 12.11 Displaying the Reading List File in a Browser

The report program `ZIXMLREADER` shown in Listing 12.12 shows how the updated `lcl_reading_list` class can be used to read a pre-existing XML file into memory. After the DOM-based XML document is loaded, the public `add_book` method can be called to add additional books to the list. The updated document is displayed in a browser window via the call to method `display` (see Figure 12.4).

```

REPORT zixmlreader.

INCLUDE: zxmllib.                "Reading List ADT
DATA: gr_reading_list TYPE REF TO lcl_reading_list,
      lt_authors      TYPE string_table.

START-OF-SELECTION.
* Read an existing reading list document into context:
  gr_reading_list =
    lcl_reading_list=>create_from_file( ).

* Add another book to the list:
  APPEND 'Horst Keller' TO lt_authors.

  CALL METHOD gr_reading_list->add_book
    EXPORTING
      im_isbn    = '978-1-59229-039-0'
      im_title   = 'The Official ABAP Reference'
      im_authors = lt_authors

```

```
im_publisher = 'SAP PRESS'.
```

- * Display the updated document in a browser:
gr_reading_list->display().

Listing 12.12 Sample Program to Read an Existing List

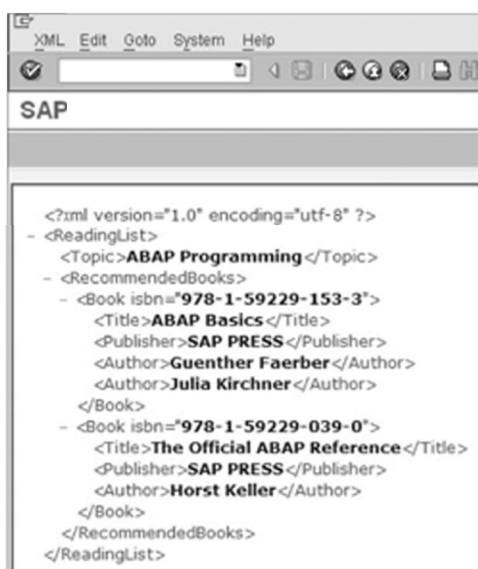


Figure 12.4 Displaying the Revised XML Document in a Browser

12.6 UML Tutorial: Advanced Activity Diagrams

In Section 8.6, UML Tutorial: Activity Diagrams, you learned how to model basic process flows using activity diagrams. In this section, we will expand upon these basic capabilities and look at some of the more advanced flow control elements provided for activity diagrams in the UML 2 standard.

The activity diagram shown in Figure 12.5 depicts an employee leave request workflow process. To effectively trace this process across all of the relevant parties/systems, the diagram is split into rectangular swim lanes called *partitions*. Partitions can be labeled to depict a class, a person/role, a system, an organization, and so on. The basic idea here is to show who does what in the process flow.

In this case, an Employee initiates the workflow process by creating a leave request. However, in this action (i.e., *Create Leave Request*), you will notice that there is a little “fork-like” icon in the left-hand side of the action icon. This notation indicates that the *Create Leave Request* action is actually a sub-activity whose details are described in another activity diagram.

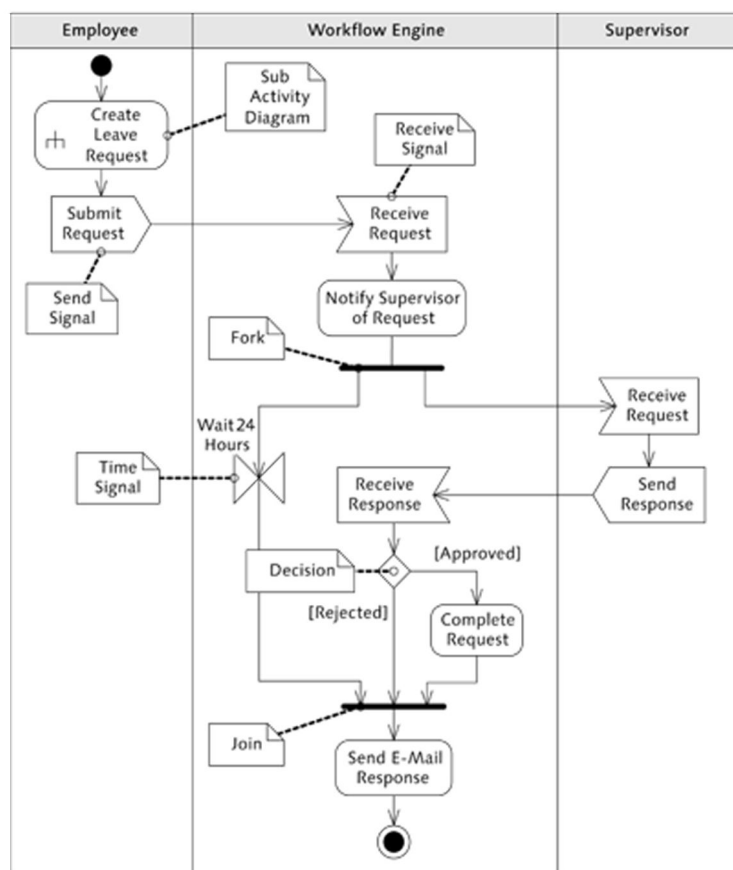


Figure 12.5 An Advanced UML Activity Diagram

Tracing the flow in Figure 12.5, you will notice that after the leave request is created, the next action is to submit the request to a workflow engine. This submission process is depicted using special action types called *signals*. The use of signals

here helps to signify the fact that the workflow process receives the request from an external process. Signals can also be used to depict other complex synchronous and asynchronous messaging scenarios.

After the leave request is received by the workflow engine, it is forwarded to the employee's supervisor (via email, for instance). At this point in time, the workflow process is in a holding pattern as it waits on a couple of potential outcomes. This holding pattern is depicted using a *fork* element. Here, one of two things can happen.

- ▶ Ideally, the supervisor will receive the request (again via a signal action), process it, and send a response back to the workflow engine.
- ▶ However, if the supervisor hasn't responded within 24 hours, the process should terminate. This 24-hour watch period is depicted using a *time signal* (i.e., the "bow-tie" icon shown in Figure 12.5).

In either case, the process comes together again in a *join* element. From here, an email response message (favorable or otherwise) is forwarded back to the initiating employee, and the process terminates as per usual.

Another element of the activity diagram that we have not yet considered is the diamond-shaped *decision* node shown in Figure 12.5. This node looks just like the *merge* node described in Section 8.6, UML Tutorial: Activity Diagrams. However, in this case, there is one input and multiple outputs as opposed to multiple inputs and a single output. Each of the outputs of a decision node is marked with a special guard text (e.g., [Approved] or [Rejected]) that describes the condition(s) in which that particular output path is selected. As you might expect, decision nodes are very good for depicting an IF/ELSE or CASE statement in a flow.

In many ways, even the advanced elements described in this section barely scratch the surface with regards to the types of things you can model using activity diagrams. To learn more about activity diagrams, see Martin Fowler's *UML Distilled* (Addison-Wesley, 2004).

12.7 Summary

In this chapter, you applied the object-oriented skills that you have gained throughout the course of this book to learn how to work with the object-oriented iXML library. The design techniques used in this chapter have been consistent

with one of the main themes of this book: to encapsulate complexity behind the boundaries of a class. In this case, you wrapped XML processing details inside of an interface that is more closely aligned with the concept of a *reading list*. Users of class `lcl_reading_list` can take advantage of this abstraction without understanding how the iXML library is being used behind the scenes.

This brief introduction to XML provided the foundation necessary for you to begin branching out and looking at other XML-based technologies such as XSLT, Web Services, and so on. If you are interested in learning more about some of these concepts, read *XML Data Exchange Using ABAP* (SAP PRESS Essentials, 2006).

13 Where to Go From Here

By now, you should feel equipped to begin applying object-oriented programming techniques in your next project assignment. Often, this transition period can be difficult in an environment where the majority of your peers operate in a procedural mindset. Here, it is more important than ever to stick to the basics and develop well-encapsulated classes with high cohesion.

After all, objects should be easy to use even if they weren't so easy to implement. Such objects are excellent teaching aids that typically spark a lot of interest within the project as more and more developers begin interfacing with your classes. In other words, *if you build it, they will come*.

In many respects, this book only reveals a very small part of the world of object-oriented programming. Clearly, understanding object-oriented syntax and basic theoretical concepts is important. However, to maximize your effectiveness as an object-oriented programmer, you need to train your mind to think in object-oriented terms. This process takes time and experience.

One way to accelerate the learning curve is to spend time looking at quality object-oriented designs created by seasoned object-oriented developers. A good place to start is to look at the catalog of designs outlined in *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994). Also, for an ABAP-centric perspective of some of these design patterns, see *Design Patterns in Object-Oriented ABAP* (SAP PRESS, 2006).

I hope that you feel that your investment in this book has been worthwhile. Having a good foundation in object-oriented skills should position you well as SAP continues to migrate away from procedural ABAP. For a glimpse into the future, I highly recommend the book *Next Generation ABAP Development* (SAP PRESS, 2007).

Appendix

A	Debugging Objects	333
B	The Author	343

A Debugging Objects

As you start working with objects in your programs, it is useful to be able to interactively step through a program to see the values assigned to various attributes and trace through the program flow logic. In this appendix, we will look at how to use the ABAP Debugger tool to debug ABAP Objects classes. The material in this appendix is presented assuming that you have some familiarity for debugging programs using the ABAP Debugger tool. If you have not used this tool before, you should read through the SAP online help documentation (<http://help.sap.com>) because basic concepts are not covered in this section.

Prior to Release 6.40 of the SAP Web AS, there was only one debugger tool available to developers. However, over time, certain limitations in this tool prompted SAP to implement the New ABAP Debugger tool using a different and more flexible architecture. Although the details of the differences between these two debugger types are beyond the scope of this book, the discussion of debugging objects has been separated into two sections so that you will understand how to use both tools to debug objects.


A.1 Debugging Objects Using the Classic ABAP Debugger

For the most part, you will find that dealing with objects in a debugger session is quite similar to working with normal data objects, procedures, and so on. Nevertheless, there are elements of the debugging process that are unique to objects. This section highlights some of these particular concepts.

A.1.1 Displaying and Editing Attributes

You can display an object in ABAP Objects in a debugging session by performing the following steps:

1. If you are not already in the `FIELDS` display mode, select this display mode by clicking on the `FIELDS` button underneath the application toolbar.

2. Select the object reference variable that points to the object you want to inspect by double-clicking the reference variable name in the ABAP program code display. Alternatively, you can enter the reference variable name in the FIELD NAMES section and press the  key (see Figure A.1).

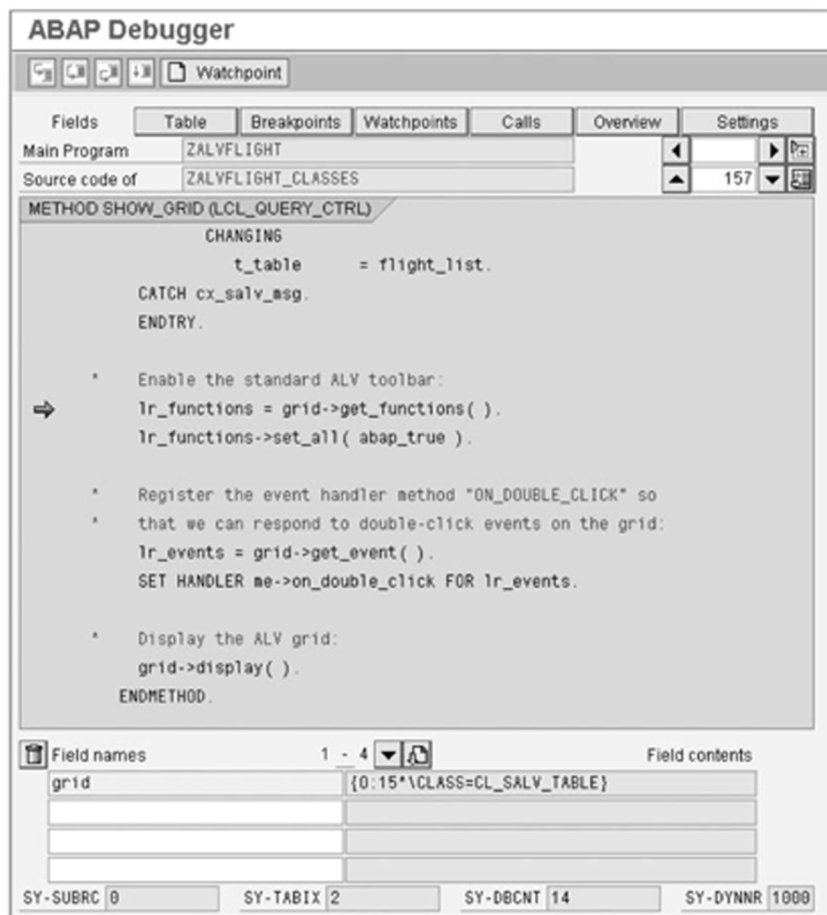


Figure A.1 Selecting an Object Reference Variable for Display

3. As you can see in Figure A.1, the FIELD CONTENTS display for an object reference variable only shows the internal object ID of the object pointed to by the object reference variable. To view the values of the attributes in this object, you must double-click on the object ID. This opens up the OBJECT display mode view shown in Figure A.2.

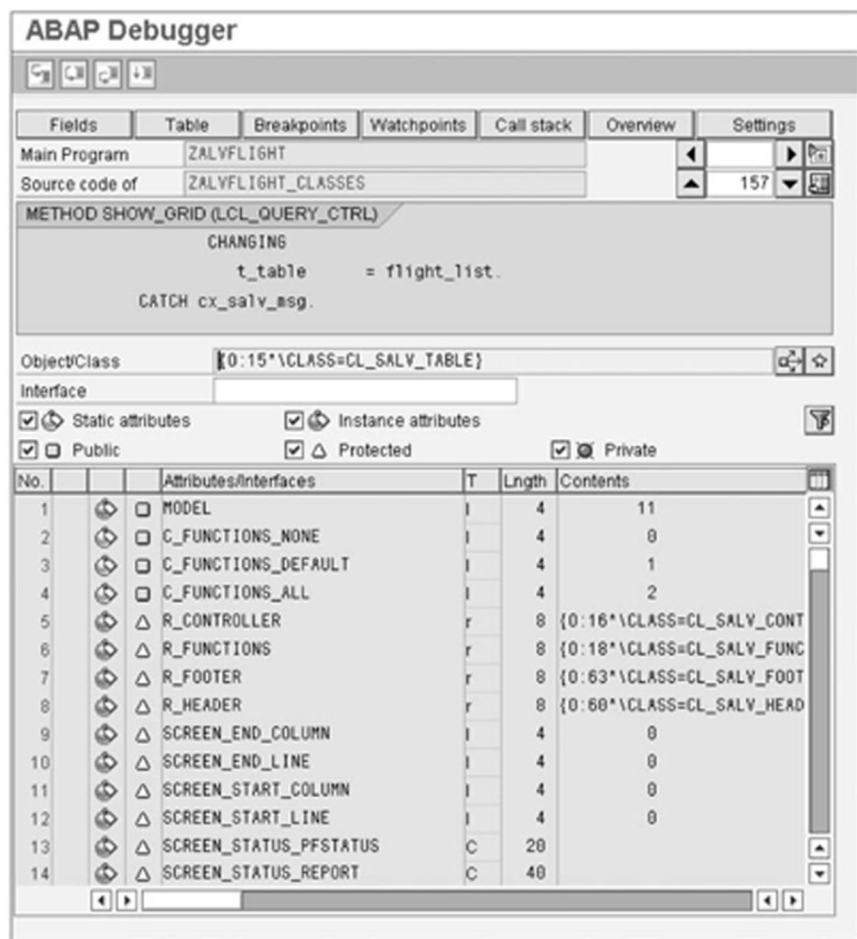


Figure A.2 Displaying the Attributes of an Object/Class

- In the OBJECT display mode, you can edit individual primitive attributes by double-clicking their names in the ATTRIBUTES/INTERFACES column. Similarly, you can edit complex types by drilling into structures, internal tables, and embedded objects.
- You also can filter the display to specific types of attributes. For example, in Figure A.2, notice that both the static and instance attributes are displayed for an object of type CL_SALV_TABLE. You can also filter attributes based on visibility section assignment (e.g., public, private, or protected).
- Finally, if the class of the object in question implements an interface, you can filter the attribute list to just those fields defined within the interface.

A.1.2 Tracing Through Methods

The process of tracing method logic is no different from tracing through a subroutine or function module. Prior to a method invocation, you can select the **SINGLE STEP** button to debug the method's implementation code. Inside the method, you can continue to step into or step over individual lines of code as usual.

Similarly, if you want to exit from the method and resume debugging after the method call, you can select the **RETURN** button. Of course, if you want to step over the method implementation entirely, you can select the **EXECUTE** button to execute the method.

One thing to note is that constructor methods do not behave in the same way as normal methods in the debugger. If you step into the **CREATE OBJECT** statement, the debugger will begin tracing through the constructor method. This is not the case, however, with class constructors. Here, you must explicitly set a breakpoint to debug the class constructor logic.

A.1.3 Displaying Events and Event Handler Methods

To display the registered events for an object/class:

1. Open the **OBJECT** display mode in the debugger.
2. Click on the **EVENTS** button to switch from object mode to events mode (see Figure A.3).



Figure A.3 Displaying the Registered Events for an Object — Part 1

- This view shows you all of the events defined for the object/class as well as any registered handling objects for the events. You can navigate to the objects shown in the HANDLING OBJECT column to set breakpoints in the event handler methods to debug event handling scenarios. This can be sometimes useful in frameworks when you are not real clear about where event handlers are registered, and so on.

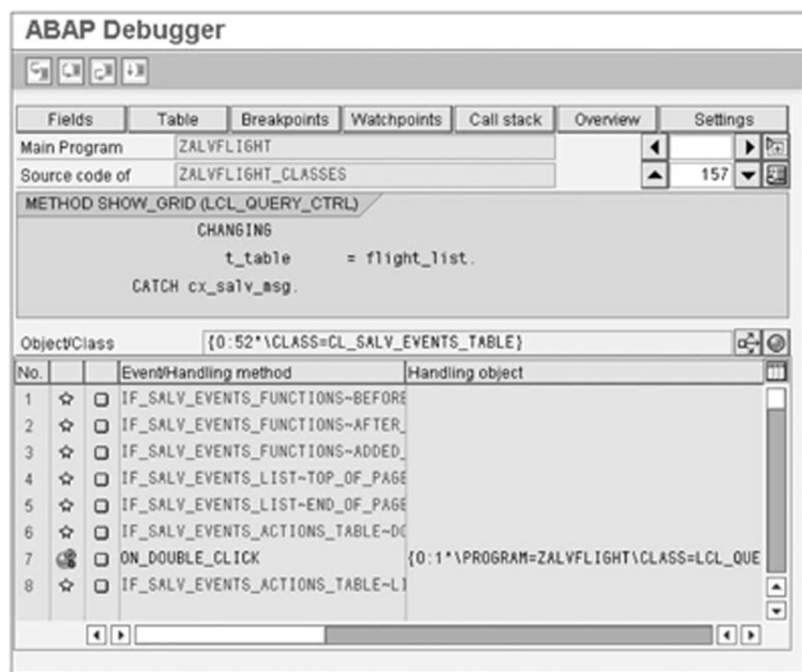


Figure A.4 Displaying the Registered Events for an Object — Part 2

A.1.4 Viewing Reference Assignments for an Object

You may encounter situations where an object is manipulated in ways that you didn't expect. Here, it is possible that more than one reference to the object exists. To view reference assignments for an object:

- You can identify the set of references to an object by selecting **GOTO • SYSTEM • FIND REFERENCE** in the menu area.
- This menu option opens a dialog box showing you all of the references to the object in question in the system (see Figure A.5). For instance, you can see that

the object of type `CL_SALV_TABLE` has four references, including the attribute `grid` from class `lcl_query_ctrl` as well as the kernel reference.

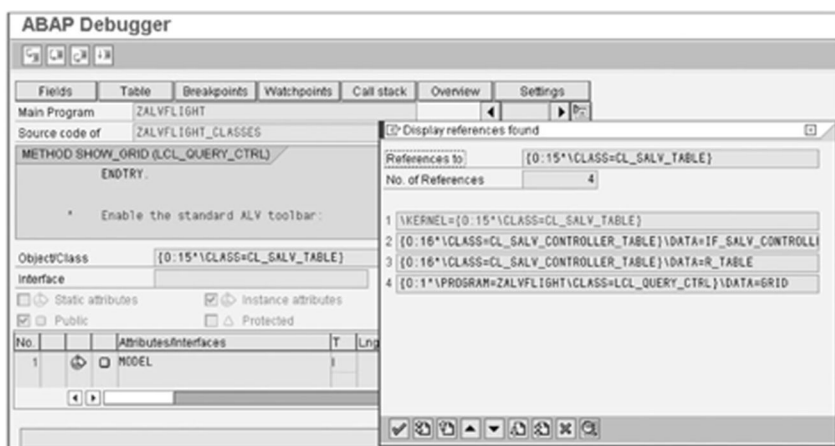


Figure A.5 Showing the Reference Assignments for an Object

A.1.5 Troubleshooting Class-Based Exceptions

Whenever a statement triggers an exception during a debugging session, the ABAP debugger traces the exception propagation process back to the exception handler block that captures the exception — assuming there was one. Often, you will encounter debugging scenarios where a developer chose not to capture the exception situation in an exception object. This information can be crucial for debugging complex exception situations. Therefore, you can modify the debugger session settings to dynamically generate an exception object that you can use to troubleshoot an error. To configure this setting, follow these steps:

1. Select the **SETTINGS** display mode, and click on the **ALWAYS CREATE EXCEPTION OBJECT** checkbox (see Figure A.6).
2. After the exception is triggered, the cursor is placed at the beginning of the **CATCH** block defined to handle the exception. If this **CATCH** block is not defined using the **INTO** addition, then you can display the exception object by clicking on the **DISPLAY EXCEPTION OBJECT** button (see Figure A.7). Otherwise, you can double-click on the exception object reference just as you would for any normal object reference variable.

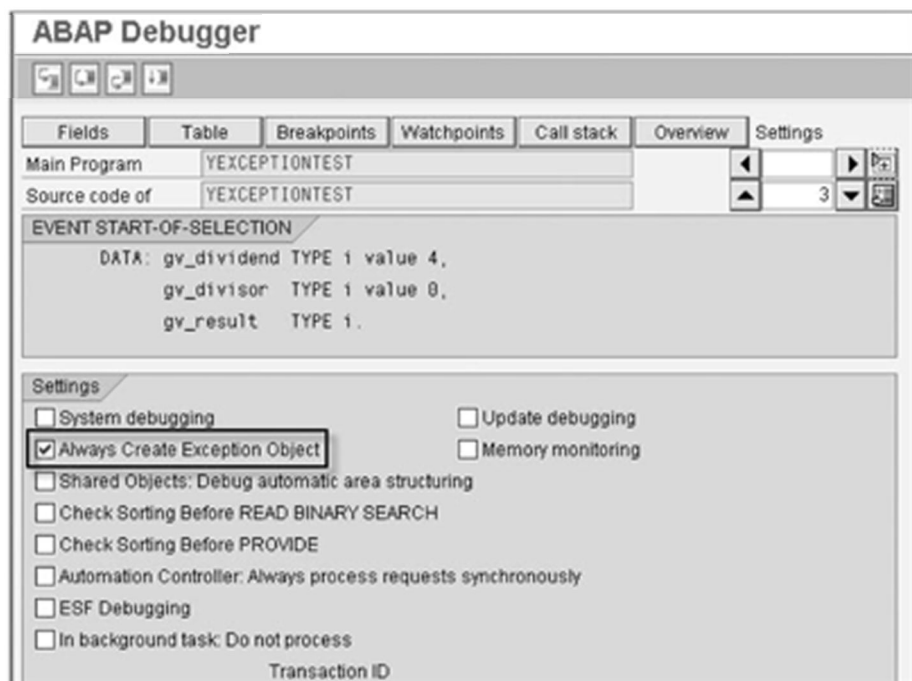


Figure A.6 Turning on the Automatic Creation of Exception Objects

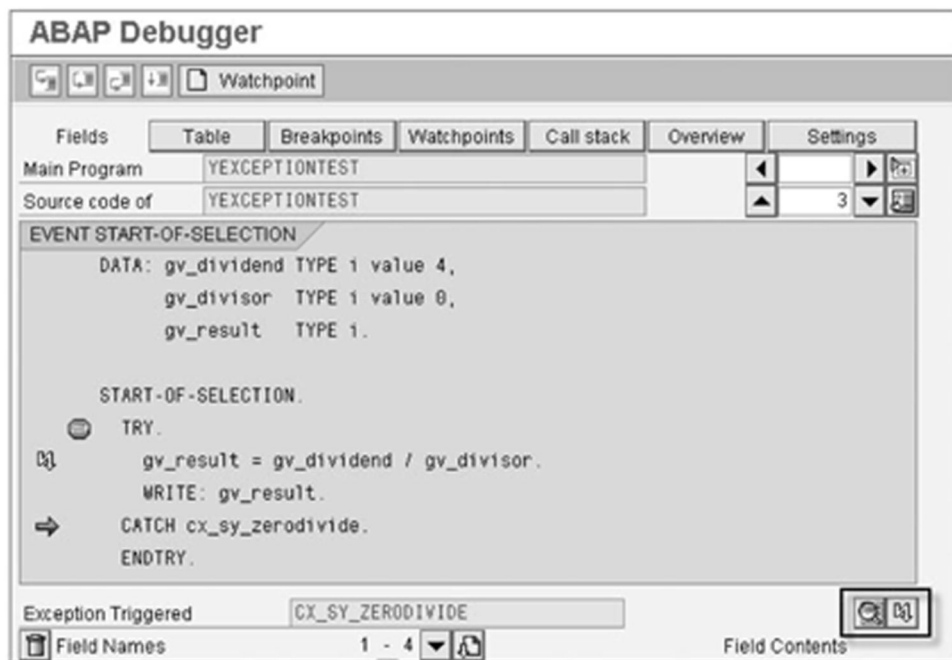


Figure A.7 Displaying an Exception Object – Part 1

- The exception object is displayed in the OBJECT display mode just like any other object type. Here, you can trace the exception chain via the PREVIOUS attribute in addition to any custom attributes that further define the exception situation (see Figure A.8).

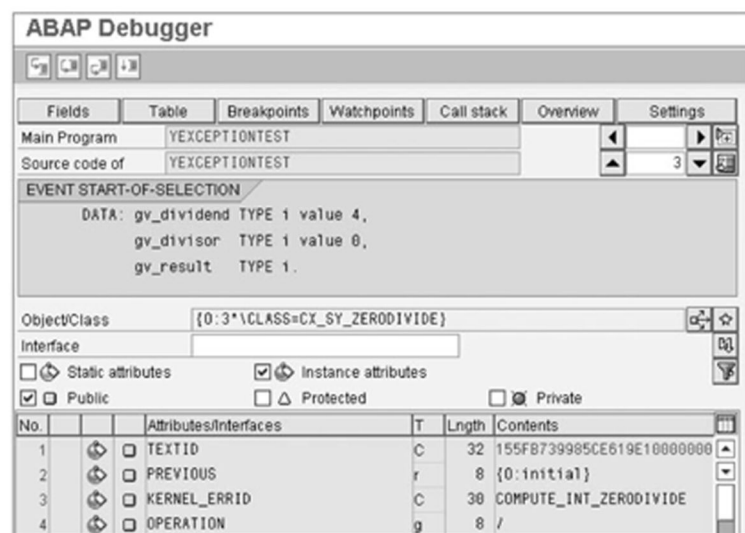


Figure A.8 Displaying an Exception Object — Part 2

A.2 Debugging Objects Using the New ABAP Debugger

In most respects, the process of debugging objects in the New ABAP Debugger is quite similar to the process of debugging objects in the Classic ABAP Debugger. The primary difference from a functional perspective is in the layout of the OBJECT display area. Context-sensitive access to this view is provided in all of the various variable display sections within the debugger window. For example, by double-clicking the object ID in the VALUE column shown in Figure A.9, you are taken to the OBJECT display shown in Figure A.10.

(1) - ABAP Debugger Controls Session 1 (Exclusive)

The screenshot displays the ABAP Debugger interface. The main window shows the source code of a method named `show_grid`. The code includes comments and several lines of ABAP code, such as `MESSAGE iv_message TYPE 'I'.`, `ENDMETHOD.`, and `grid->display().`. The cursor is positioned at line 163. To the right, the Variables window is open, showing a table with columns for Variable, Val, and Hexadecimal Value. The variable `GRID` is listed with the value `(0:15*1CLASS+CL_SALV_TABLE)`.

```

143
144 MESSAGE iv_message TYPE 'I'.
145 ENDMETHOD.
146
147 METHOD show_grid.
148 * Method-Local Data Declarations:
149 DATA: lr_functions TYPE REF TO cl_salv_functions,
150       lr_events     TYPE REF TO cl_salv_events_tabl.
151
152 * Use the Factory method to create an instance of the
153 * class.
154 CALL METHOD cl_salv_table->factory
155 IMPORTING
156   r_salv_table = grid
157 CHANGING
158   r_table      = flight_list.
159
160 *
161 ENDMETHOD.
162
163 * Enable the standard ALV toolbar:
164 lr_functions = grid->get_functions( ).
165 lr_functions->set_all( abap_true ).
166
167 * Register the event handler method "ON_DOUBLE_CLICK"
168 * that we can respond to double-click events on the
169 * table.
170 lr_events = grid->get_events( ).
171 SET HANDLER me->on_double_click FOR lr_events.
172
173 * Display the ALV grid:
174 grid->display( ).
175 ENDMETHOD.

```

Variable	Val	Hexadecimal Value
GRID	(0:15*1CLASS+CL_SALV_TABLE)	

Figure A.9 Basic Layout of the New ABAP Debugger

(1) - ABAP Debugger Controls Session 1 (Exclusive)

The screenshot displays the ABAP Debugger interface with the Objects window open. The Object field shows `(0:15*1CLASS+CL_SALV_TABLE)`. The Objects window displays a tree view of the object's attributes. The attributes are listed in a table with columns for Attribute, Value, and Hexadecimal Value.

Attribute	Value	Hexadecimal Value
OBJECT		
CL_SALV_MODEL		
MODEL	11	00000000
R_CONTROLLER	(0:16*1CLASS+CL_SALV_CONTROLLER_TABLE)	
CL_SALV_MODEL_BASE		
C_FUNCTIONS_NONE	0	00000000
C_FUNCTIONS_DEFAULT	1	01000000
C_FUNCTIONS_ALL	2	02000000
R_FUNCTIONS	(0:18*1CLASS+CL_SALV_FUNCTIONS_LIST)	
R_FOOTER	(0:03*1CLASS+CL_SALV_FOOTER)	
R_HEADER	(0:60*1CLASS+CL_SALV_HEADER)	
SCREEN_END_COLUMN	0	00000000
SCREEN_END_LINE	0	00000000

As you can see in Figure A.10, the OBJECT display in the New ABAP Debugger is much more streamlined. Here, by default, you can see the attributes of the object/class in question arranged hierarchically according to the inheritance hierarchy. You can turn this feature off by selecting the SUPERCLASSES ON/OFF button directly above the attribute display. Events and their registered event handler methods are displayed using the EVENTS button. References to the object in question can be viewed using the DISPLAY REFERENCES button.

In addition to all of the standard features carried over from the Classic ABAP Debugger, the New ABAP Debugger also provides a function to display the inheritance hierarchy of a given object at runtime. This function can be accessed by clicking on the DISPLAY INHERITANCE HIERARCHY button. Figure A.11 shows the INHERITANCE RELATIONSHIP view for an object of type CL_SALV_TABLE.

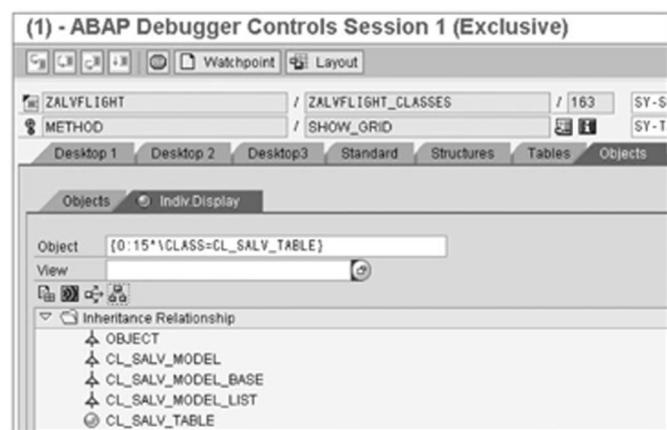


Figure A.11 Showing the Inheritance Hierarchy of an Object

B The Author



James Wood is the founder and principal consultant of Bowdark Consulting, Inc., an SAP NetWeaver consulting and training organization. With more than seven years of experience as a software engineer, James specializes in custom development in the areas of ABAP Objects, Java/J2EE, SAP NetWeaver Process Integration, and the SAP NetWeaver Portal.

Before starting Bowdark Consulting, Inc. in 2006, James was an SAP NetWeaver consultant for SAP America, Inc. and IBM Corporation where he was involved in multiple SAP implementations. He holds a master's degree in software engineering from Texas Tech University. To learn more about James and the book, please check out his website at <http://www.bowdarkconsulting.com>.

Index

A

ABAP Debugger tool 333
ABAP Editor
 Insert Statement wizard 268
ABAP List Viewer → SAP List Viewer
ABAP Object Services 279
 Persistence Service 281
 Query Service 295
ABAP Objects 23
 Obsolete statement 52
ABAP Objects class
 Declaration part 41
 Implementation part 41
 Syntax overview 41
ABAP Unit 233
 ABAP Workbench 236, 245
 Automating test run 245
 Code Inspector 245
 Controlling test generation 238
 Creating test class 237
 Error severity 247
 Evaluating unit test result 246
 Example 241
 Executing unit test 244
 Fixture 239
 Generating test class for global class 240
 Overview 234
 Profile parameter abap/test_generation 238
 Release version 235
 Result display screen layout 246
 SAUNIT_CLIENT_SETUP 238
ABAP visibility section 42, 96
 Global class assignment 96
 PRIVATE SECTION 42, 96
 PROTECTED SECTION 42, 96, 133
 PUBLIC SECTION 42, 96
Abstract class 140
 Defining in global class 142
 Defining in local class 141
 Delegation 140
 Problem with dummy method 140

Abstract class (cont.)
 Template 140
 Usage scenario 141
Abstract data type 25
 Class example 101
 Function group example 94
 lcl_date 67
 Reading list ADT 311
Abstract method 140
 in global class 142
 in local class 141
Addition
 RAISING 215
ALV Object Model 257, 260
 CL_SALV_EVENTS_TABLE 271
 CL_SALV_TABLE 260
 Event handling 271
 Example program 261
 Overview 260
 Release 257
ALV → SAP List Viewer
Assertion
 CL_AUNIT_ASSERT 236
Attribute 43
 CLASS-DATA keyword 43
 Constant 43
 CONSTANTS keyword 43
 DATA keyword 43
 Defining class attribute 43
 Defining instance attribute 43
 Definition example 43
 Naming 44
 READ-ONLY keyword 98
Automation Controller
 Eventing 258
 Function 258

B

BAPI_FLIGHT_GETLIST function 262
Business Address Services 50

C

-
- C++ 23
 - Casting
 - ?TO option* 160
 - Basic rule* 158
 - Casting operator (?=)* 160
 - Defined* 158
 - Dynamic type* 158
 - Example* 156
 - Implicit narrowing cast example* 159
 - Incompatible type* 158
 - Narrowing cast* 159
 - Static type* 156
 - Widening cast* 160
 - CATCH statement 204
 - Best practices* 208
 - INTO addition* 208
 - CL_SALV_TABLE
 - Instantiation context* 260
 - CL_SALV_TABLE Class
 - UML class diagram* 260
 - CLASS
 - Keyword* 25
 - Class 25
 - Attribute* 24
 - Comparison to function group* 91
 - Extending* 129
 - Inheriting component* 129
 - Method* 24
 - OBJECT* 129
 - Private interface* 96
 - Public interface* 96
 - Syntax overview* 41
 - Visibility section* 27
 - Class agent API
 - Agent class relationship* 283
 - Architecture* 282
 - CREATE_PERSISTENT method* 295
 - GET_PERSISTENT method* 295
 - IF_OS_CA_PERSISTENCY interface* 286, 296
 - IF_OS_FACTORY interface* 286
 - Useful method* 285
 - Class Builder 71
 - ABAP Workbench* 71
 - Class Editor screen* 74
 - Class Builder (cont.)
 - Creating class* 72
 - Declaring and using types* 80
 - Defining attribute* 75
 - Defining class component* 74
 - Defining event* 79
 - Defining method* 77
 - Exception Builder view* 222
 - Mapping Assistant* 288
 - Object Navigator* 71
 - Class component 42, 57
 - Context* 57
 - Defined* 42
 - Selector* 59
 - Selector operator* 59
 - Static component* 42
 - Usage example* 57
 - CLASS DEFINITION statement 42
 - ABSTRACT addition* 141
 - CREATE addition* 118
 - DEFERRED addition* 100
 - FINAL addition* 143
 - FOR TESTING addition* 237
 - INHERITING FROM addition* 131
 - CLASS IMPLEMENTATION statement 51
 - Class pool
 - Defined* 71
 - Class-based exception handling concept 203
 - CATCH block* 205
 - CLEANUP block* 209
 - Exception class* 203
 - Propagating exception* 215
 - Propagation rule* 219
 - RAISE EXCEPTION statement* 211
 - Scope* 205
 - System-driven exception* 211
 - TRY control structure* 203
 - CLASS-DATA keyword 43
 - Classic ABAP Debugger tool 333
 - Debugging object* 333
 - Fields display mode* 333
 - Filtering the object display* 335
 - CLEANUP statement 204
 - Call sequence* 210
 - Forbidden control statement* 210
 - Usage example* 209

- Cohesion 27
 - Component 41
 - Class component 42
 - Declaring 42
 - Defining 42
 - Instance component 42
 - Namespace 43
 - Visibility 95
 - Component-based design 183
 - Composition 29, 145
 - Defined 146
 - Example 146
 - Favoring over inheritance 148
 - Forwarding 175
 - Has-a relationship 146
 - Constant
 - Naming convention 43
 - Readability 44
 - Usage example 59
 - CONSTANTS keyword 43
 - Constructor 111
 - Call sequence 112
 - Class constructor 115
 - Guaranteed initialization 112
 - Initializing class attribute with class constructor 115
 - Instance constructor 111
 - Coupling
 - Loose coupling 28
 - CREATE OBJECT statement 53
 - TYPE addition 159
 - Creating object 107
 - CREATE OBJECT statement 53, 108
 - Memory management concept 108
- D**
-
- DATA keyword 43
 - Data object
 - Analogy to remote control 157
 - Conversion 156
 - Dynamic data object 110
 - Dynamic type 158
 - Static type 156
 - DATA statement
 - TYPE REF TO addition 53
 - Data type
 - Abstract data type 25, 67, 94, 101, 311
 - Debugging object 333
 - Create Exception Object option 338
 - CREATE OBJECT statement 336
 - Displaying and editing attribute 333
 - Displaying event and registered handler method 336
 - New ABAP Debugger 340
 - Tracing through method 336
 - Troubleshooting class-based exception 338
 - Viewing reference assignment 337
 - Design by contract 28, 102
 - Invariant 103
 - Postcondition 103
 - Precondition 103
 - Development class 186
 - Diamond problem 164
 - Document Object Model → DOM
 - DOM 310
 - Format 310
 - Dynamic object allocation
 - Performance costs 107
 - Dynamic type 156
- E**
-
- Encapsulation 27, 89
 - Combining data and behavior 94
 - Least privilege concept 134
 - Purpose 95
 - Event 49
 - Class event declaration syntax 49
 - CLASS-EVENTS keyword 49
 - Debugging 336
 - Event handler method 49
 - Event handling example 259, 271
 - EVENTS Keyword 49
 - EXPORTING parameter 49
 - Implementing event handler method 271
 - Instance EVENT Declaration syntax 49
 - Processing behavior 275
 - RAISE EVENT statement 50, 274
 - Registering handler method 272
 - Sending object 49
 - SET HANDLER statement 272

Index

- Event (cont.)
 - Triggering 274
 - Event handler method
 - Declaration scope 49
 - Declaration syntax 49
 - Example 273
 - Implementing 273
 - Importing parameter interface 50
 - Exception 201
 - CX_ROOT 219
 - Debugging 338
 - Defining exception class 219
 - Defining exception in the global class method 218
 - Defining exception text 223
 - Defining global exception class 221
 - Defining local exception class 221
 - Exception Builder tool 222
 - Exception class type 220
 - Exception class with message class 222
 - Non-class-based exception 203
 - RAISE EXCEPTION statement 212
 - Runtime error 205
 - Showing exception details example 206
 - Unchecked exception 211
 - Exception class
 - Class hierarchy 219
 - Customer namespace 222
 - CX_AI_APPLICATION_FAULT 216
 - CX_AI_SYSTEM_FAULT 216
 - CX_DYNAMIC_CHECK 219, 220
 - CX_NO_CHECK 219, 221
 - CX_STATIC_CHECK 219, 220
 - CX_SY_ARITHMETIC_ERROR 208
 - CX_SY_ARITHMETIC_OVERFLOW 208
 - CX_SY_FILE_JO 226
 - CX_SY_MOVE_CAST_ERROR 205
 - CX_SY_ZERODIVIDE 208
 - Global exception class 226
 - Local exception class 221
 - PREVIOUS attribute 216
 - Standard exception type 206, 211
 - Superclass CX_ROOT 205
 - With Message Class option 227
 - Exception handling
 - Ad-Hoc exception handling example 201
 - Exception handling (cont.)
 - Cross-cutting concern 201
 - Message table parameter 202
 - Return code parameter 202
 - Strategy 202
 - Exception text 223
 - As constants 223
 - Mapping to message ID 227
 - MESSAGE statement support 229
 - Text parameter 226
 - Viewing in the OTR 224
 - Extensible Markup Language → XML
- ## F
-
- Factory pattern 260
 - Final class 143
 - Completing an inheritance tree 143
 - Marking global class as final 144
 - Marking local class as final 143
 - Final method 144
 - Defining final method in global class 144
 - Defining final method in local class 144
 - Friend 99
 - Argument against 100
 - Defining relationship 99
 - Example 99
 - Relationship behavior 100
 - Syntax 99
 - Function group 91
 - Example 91
 - Function module 91
 - Global data 91
 - Limitation 91
 - Stateless function module 91
 - Function module
 - SDIXML_DOM_TO_SCREEN 323
 - Functional method 47
 - CALL METHOD statement 61
 - CASE statement 64
 - COMPUTE statement 64
 - Creating expression 61
 - Declaration syntax 47
 - DELETE statement 64
 - Expression 63
 - IMPORTING parameter syntax 63

Functional method (cont.)

- Logical expression* 64
- LOOP statement* 64
- MODIFY statement* 64
- MOVE statement* 63
- Operand* 61
- Syntax example* 63
- Usage example* 61

G

- Garbage collection 121
 - ABAP runtime environment* 121
 - CLEAR statement* 121
 - Destructor method* 122
- Global class 41
 - Class pool* 41
 - Defining read-only attribute* 98
 - Definition section* 82
 - Direct type entry* 81
 - Local inner class definition* 81
 - Method implementation* 78
 - Naming convention* 72
 - Only Modeled option* 74

H

- HTML 304

I

- IF statement
 - IS SUPPLIED option* 119
- IF_MESSAGE interface 220
- IF_SERIALIZABLE_OBJECT interface 220
- IF_T100_MESSAGE interface 227
- Implementation hiding 27, 89
 - Approach* 95
 - Example* 101
 - Getter method* 98
 - Hiding data* 97
 - Responding to change* 102
 - Setter method* 97
- Inheritance 30, 127
 - Abstract class* 140
 - Basic example* 129

Inheritance (cont.)

- Class component in subclass* 136
- Class component scope* 136
- Class constructor behavior* 140
- Comparison with composition* 145
- Component namespace* 136
- Copy-and-paste approach* 131
- Defining relationship in global class* 131
- Defining relationship in local class* 129
- Defining the inheritance interface* 133
- Diamond problem* 164
- Eliminating redundant code* 127
- Final class* 140
- Generalization and specialization* 128
- Implementation inheritance* 155
- Instance component in subclass* 135
- Instance constructor behavior* 138
- Interface example* 133
- Interface inheritance* 155
- Interface inheritance versus implementation inheritance* 164
- Is-a relationship* 31
- Is-a versus has-a relationship* 145
- Multiple inheritance* 164
- Public interface inheritance* 155
- Redefining method* 136
- Relationship behavior* 131
- Single inheritance* 164
- Subclass* 30, 129
- Super pseudo reference* 135
- Superclass* 30, 129
- Superclass method implementation* 138
- Tree example* 129
- Insert Statement wizard 268
 - ABAP Objects pattern* 268
 - Statement pattern* 269
- Instance component 42, 54
 - Defined* 42
 - Object component selector operator* 55
- Instantiation context 117
 - CREATE addition* 118
 - Creational class method* 119
 - Defining* 118
 - Defining for global class* 118
 - Option* 118
 - Pattern example* 119

Index

- Interaction frame 300
 - alt* Operator 300
 - common operator* 300
 - Example* 300
 - Guard* 300
 - loop Operator* 300
 - Notation* 300
 - Operator* 300
 - opt Operator* 300
 - par Operator* 300
 - ref Operator* 301
 - sd Operator* 301
 - Interface 26, 163
 - Contract of a class* 177
 - Defining a local interface* 165
 - Defining alias name* 178
 - Defining component* 165
 - Defining global interface* 165
 - Defining in ABAP Objects* 165
 - Elementary interface* 165
 - Generic basic definition* 163
 - IF_MESSAGE* 220
 - IF_SERIALIZABLE_OBJECT* 220
 - IF_T100_MESSAGE* 227
 - Implementing in a global class* 168
 - Implementing in a local class* 167
 - Implementing in class* 167
 - Interface component selector operator* 169
 - INTERFACES* keyword 167
 - Nesting interface* 177
 - Public visibility section* 165
 - Scope* 165
 - Syntax* 165
 - Using interface* 170
 - Interface reference variable 174
 - Casting example* 174
 - Using references polymorphically* 174
 - Is-a relationship 31
 - Iterator 175
 - CL_SWF_UTL_ITERATOR* 175
 - iXML library 303
 - Architecture of API* 311
 - CL_IXML_factory class* 311
 - DOM-based XML document* 311
 - Feature* 310
 - IF_IXML interface* 311
 - iXML library (cont.)
 - IF_IXML_DOCUMENT interface* 312
 - IF_IXML_ELEMENT interface* 312
 - IF_IXML_ISTREAM interface* 311
 - IF_IXML_PARSE_ERROR interface* 323
 - IF_IXML_PARSER interface* 311, 323
 - IF_IXML_RENDERER interface* 319
 - Interface* 311
 - Release* 310
 - Usage example* 311
-
- ## J
-
- Java 23
-
- ## K
-
- Keyword
 - CLASS* 25
 - CLASS-DATA* 43
 - CONSTANTS* 43
 - DATA* 43
-
- ## L
-
- Local class 41
 - INCLUDE program* 70
 - Including in program* 64
 - Visibility* 70
- Local variable
 - Hiding attribute* 52
 - Naming* 52
 - Usage* 52
- Loose coupling 28
-
- ## M
-
- Mapping Assistant 288
 - Assignment type* 290
 - Business Key assignment type* 290
 - Class Identifier assignment type* 290
 - GUID assignment type* 290
 - Object Reference assignment type* 290
 - Tables/Fields display* 289
 - Value Attribute assignment type* 290

Markup language 304
 Element 304
 Tag 304
 Meta-markup language
 Definition 304
 Method 45
 ABSTRACT addition 141
 Boolean 62
 CALL METHOD statement 56
 CHANGING parameter 45
 Class method declaration syntax 48
 CLASS-METHODS keyword 48
 Declaration example 47
 DEFAULT addition 48
 Defining parameter 45
 EXCEPTIONS addition 203
 EXPORTING parameter 45
 FOR TESTING addition 239
 Functional method 47
 Implementing 51
 IMPORTING parameter 45
 Instance method declaration syntax 45
 Method signature 46
 METHOD...ENDMETHOD statement 51
 METHODS Keyword 45
 Naming convention 48
 OPTIONAL addition 47
 Parameter 45
 RAISING addition 215
 Redefining 136
 Redefining method in global class 137
 Redefining method in local class 136
 REDEFINITION addition 136
 RETURNING parameter 47
 Using local variable 52
 Method signature
 Defined 46
 Model-View-Controller → MVC
 MVC 261
 ABAP reports 266
 Controller 262
 Coupling 270
 Data binding 264
 Model 262
 Report 262
 View 262

N

Narrowing cast
 Definition 159
 Implicit cast for importing parameter 162
 Nested interface 177
 Alias name 178
 ALIASES statement 178
 Component interface 177
 Component scope 178
 Defining alias in local interface 178
 Defining component interface in global interface 178
 Defining component interface in local interface 178
 INTERFACES statement 178
 New ABAP Debugger tool 333
 Displaying inheritance hierarchy 342
 Layout 340
 Release 333
 Nonclass-based exception
 Example 217

O

Object 24, 41
 Autonomy 94
 CREATE OBJECT statement 53
 Debugging 333
 Dynamic allocation 107
 Header data 110
 Identity 94
 Initialization and cleanup 107
 Object instance 26
 Object lifecycle 107
 Performance tuning 122
 Self-reference (me) 56
 SWF_UTL_OBJECT_TAB table type 171
 OBJECT class 129
 Object component selector 55
 Usage example 55
 Object instance 26
 Object reference assignment
 Assignment operator (=) 53
 Illustrated example 53
 MOVE statement 53

Index

- Object reference variable 26, 52
 - Advanced assignment* 155
 - Assignment* 53
 - Assignment between families of related types* 156
 - Compatible type* 156
 - Declaration context* 53
 - Declaration syntax* 53
 - Defined* 52
 - Pointing to object* 110
 - Relationship to object* 54
 - Self-reference (me)* 56
 - Super pseudo reference variable* 135
 - Object-Oriented Analysis and Design → OOAD
 - Object-oriented programming → OOP
 - Object-relational mapping 280
 - Benefit* 280
 - Concept* 280
 - Tool* 280
 - Online Text Repository → OTR
 - OOAD 33, 275
 - Defining class relationship* 128
 - Delegating responsibilities to objects* 107
 - Domain modeling* 128
 - OOP 23
 - Emphasis on data and behavior* 94
 - OSREFTAB table type 296
 - OTR 223
- ## P
-
- Package
 - Assignment to SAP Application Hierarchy* 190
 - Assignment to software component* 190
 - Common closure principle* 196
 - Common reuse principle* 196
 - Local Object* 66
 - Package type* 190
 - Static dependency principle* 196
 - Package Builder
 - Adding package* 191
 - Defining a package interface* 192
 - Defining use access* 194
 - Editing a package interface* 193
 - Package Builder (cont.)
 - Object Navigator* 189
 - Package Interfaces tab* 192
 - Packages Included tab* 191
 - Transaction* 188
 - Package check tool 196
 - ABAP Workbench* 196
 - Package concept 185
 - Creating use access* 194
 - Defining package interface* 192
 - Definition* 186
 - Design tip* 196
 - Embedding package* 191
 - Main package* 186, 187
 - Package Builder* 188
 - Package check* 195
 - Package versus development class* 185
 - Release version* 185
 - SAP Application Hierarchy* 190
 - Structure package* 186
 - Sub-package* 186, 188
 - Parameter 45
 - Actual parameter* 46
 - CHANGING* 45
 - Default behavior* 46
 - EXPORTING* 45
 - Formal parameter* 46
 - IMPORTING* 45
 - Pass-by-reference* 46
 - Pass-by-value* 46
 - RETURNING* 47
 - Syntax* 45
 - Performance tuning 122
 - Class attribute usage* 124
 - Design consideration* 123
 - Lazy initialization* 123
 - Reusing object* 124
 - Persistence mapping 284
 - By business key* 284
 - By instance-GUID* 284
 - Example* 288
 - Mapping Assistant* 288
 - Multiple-table mapping* 285
 - Single-table mapping* 285
 - Strategy* 284
 - Structure mapping* 285

- Persistence Service
 - Architecture 281
 - CX_OS_OBJECT_EXISTING* 295
 - Deleting persistent object 298
 - Layered agent class approach 283
 - Managed object 282
 - Managing persistent object 281
 - Mapping concept 284
 - Mapping strategy 284
 - Overview 280
 - Persistent class 282
 - Updating persistent object 297
 - Persistent class 279
 - Agent class 282
 - Architecture 282
 - Building a persistence layer 281
 - Class agent API 285
 - Class Builder 287, 292
 - Defining 286
 - IF_OS_STATE* interface 287
 - Mapping to a persistence model 280
 - Object reference 292
 - Persistent object 279
 - COMMIT WORK* statement 295
 - Creating new instance 294
 - Deletion example 298
 - Managed object 282
 - Transient object 280
 - Updating example 297
 - Working with 293
 - Pointer 110
 - Defined 110
 - Polymorphism 31, 155
 - Dynamic method call binding 160
 - Example 160
 - Extensibility 163
 - Flexibility 163
 - Procedural programming 89
 - ABAP example 91
 - Comparison to OOP 94
 - Data support 94
 - Functional decomposition 90
 - Limitation 90
 - Procedure 90
 - Reuse 127
 - Step-wise refinement 90
 - Procedural programming (cont.)
 - Structured programming 90
 - Tight coupling with main program 90
- ## Q
-
- Query Service 295
 - CL_OS_SYSTEM* 296
 - Filter 296
 - IF_OS_QUERY* interface 295
 - Logical expression query 295
 - Query Manager 296
 - Results ordering 296
 - Usage example 296
- ## R
-
- RAISE EVENT* statement 50, 274
 - Parameter usage 275
 - Syntax 274
 - RAISE EXCEPTION* statement 212
 - Behavior 213
 - Exception object 213
 - Syntax 212
 - Usage example 213
 - RAISING* addition
 - Syntax 215
 - Usage example 215
 - Reading list ADT 312
 - Basic structure 314
 - CL_GUI_FRONTEND_SERVICES* 319
 - Class constructor method 315
 - Example program *ZIXMLREADER* 324
 - Example program *ZIXMLWRITER* 319
 - Method *add_book* 316
 - Method *create_from_file* 321
 - Method *create_new_list* 314
 - Method *display* 323
 - Method *serialize* 317
 - New XML document 314
 - Private instantiation context 312
 - XML document 320
 - Refactoring 148
 - Example 148
 - Move method example 149

Index

- Refactoring assistant 149
 - Accessing* 149
 - Example* 149
 - Expansion* 149
- Reference parameter
 - Defined* 46
 - Example* 46
- Reference variable 26
- Regular expression 82
 - CL_ABAP_MATCHER* 82
 - CL_ABAP_REGEX* 82
 - Defined* 82
 - Example* 83
 - Literal character* 82
 - Metacharacter* 82
 - Search pattern* 82
 - Selection screen input* 83
 - Telephone number example* 83
- Reuse 29
- Reuse Library 257
 - REUSE_ALV_GRID_DISPLAY* 257
- S**

- SAP component model 183
 - Hierarchy* 183
 - Package* 183
 - Product* 183
 - Software component* 183
- SAP Control Framework
 - ABAP Objects control framework* 258
 - ALV grid control* 259
 - Architecture* 258
 - Automation controller* 258
 - Automation queue* 258
 - Basis release* 258
 - Calendar control* 260
 - CL_GUI_CONTROL* 258
 - Custom container* 259
 - Custom control* 258
 - Distributed processing* 258
 - HTML viewer control* 260
 - Microsoft ActiveX control* 258
 - Overview* 257
 - Proxy class* 258
- SAP Control Framework (cont.)
 - Sun JavaBeans control* 258
 - User-defined control* 260
- SAP flight data model 262
- SAP List Viewer 257
 - Field catalog* 260
- SAP NetWeaver Application Server
 - ABAP instance* 108
 - ABAP runtime environment* 107
 - Extended memory* 110
 - Internal session* 109
 - Main session* 109
 - Memory architecture* 108
 - Multiplexing work process* 109
 - Performance optimization* 111
 - Program call stack* 109
 - Roll buffer* 108
 - Shared memory* 108
 - User session* 108
 - Work process* 108
- SAX 310
- SDIXML_DOM_TO_SCREEN function module 323
- Self-reference variable 56
- SET HANDLER statement 272
- SGML 303
- Simple API for XML → SAX
- Singleton design pattern 282
 - Implemented in persistent class* 282
- Software component
 - HOME* 184
 - Installed component* 184
 - LOCAL* 184
 - Version* 184
- Software framework 279
 - ABAP Object Services* 279
- Sorting
 - Insertion sort algorithm* 173
- Standard Generalized Markup Language → SGML
- Statement
 - CATCH* 204, 208
 - CLASS DEFINITION* 42, 100, 118, 131, 141, 143, 237
 - CLASS IMPLEMENTATION* 51

Statement (cont.)

CLEANUP 204, 209, 210
CREATE OBJECT 53, 159
DATA 53
IF 119
RAISE EVENT 50, 274, 275
RAISE EXCEPTION 212, 213
SET HANDLER 272
TRY 203, 204, 205, 208
TYPE-POOLS 50

Static type 156

T

Test class

Attribute 238
Basic form 237
CL_AUNIT_ASSERT 239
Creating 237
Defining fixture method 239
Defining test method 239
Duration attribute 238
Example 241
Risk_Level attribute 238

Test-driven development 247

Transaction

SAUNIT_CLIENT_SETUP 238
SE24 71

TRY statement

CATCH block 204
CATCH block organization 208
CATCH statement selection process 205
CLEANUP block 204
Control flow 204
Defined 204
Syntax 203

Type 50

Naming convention 50
Scope 50
TYPES statement 50
Usage 50
Usage example 50

TYPE-POOLS statement 50

U

UML 34

UML activity diagram 229

Action 229
Activity final node 229
Decision node 327
Decision node guard 327
Expansion region 231
Fork 327
Handler block 230
Initial node 229
Join 327
Merge node 230
Notation 229
Partition 325
Protected node 230
Signal 326
Sub-activity 326
Time signal 327

UML class diagram 34

Abstract class and method notation 152
Association 38
Attribute notation 36
Class notation 35
Composition notation 151
Dependency notation 151
Depicting nested and component interface 180
Generalization notation 150
Generalization notation for interface 180
Interface notation 180
Non-normative notation for abstract class 153
Notation for relationship with interface 181
Notation for static attribute and method 182
Operation notation 37

UML communication diagram 275

Collaboration diagram 276
Ease of use 277
Interaction diagram 276
Notation 276
Numbering scheme 277

UML communication diagrams

Object diagram 277

Index

- UML diagram 34
 - Behavioral diagram* 103
 - Cardinality notation* 39
 - Interaction diagram* 105
 - UML diagram (cont.)
 - Note* 39
 - UML object diagram 84
 - Defined* 84
 - Notation* 86
 - Object box notation* 85
 - UML package diagram 197
 - Defining visibility of component* 197
 - Dependency notation* 197
 - Notation* 197
 - Package* 197
 - Relaxed notation* 197
 - UML sequence diagram 103
 - Deleting an object lifeline* 299
 - Found message* 104
 - Interaction frame* 300
 - Message* 104, 105
 - New message* 298
 - Notation* 104
 - Object activation bar* 105
 - Object lifeline* 105
 - Return message* 105
 - Self call* 105
 - UML state machine diagram 125
 - Final state* 126
 - Initial pseudostate* 125
 - Notation* 125
 - State* 125
 - Transition* 125
 - Transition label syntax* 125
 - UML use case diagram 248
 - Notation* 251
 - Usage* 251
 - Unified Modeling Language → UML
 - Unit testing 233
 - ABAP Unit* 233
 - Assertion* 236
 - Black box test* 234
 - Fixture* 236
 - IEEE definition* 233
 - Unit testing (cont.)
 - Informal testing process* 234
 - Problem with ad hoc test* 234
 - Scope* 233
 - Terminology* 235
 - Test class* 236
 - Test method* 236
 - Test run* 236
 - Test task* 236
 - Test-driven development* 247
 - Unit testing framework* 234
 - Validating API contract* 233
 - White box test* 235
 - Unit testing framework 234
 - ABAP Unit* 235
 - Automated testing* 235
 - JUnit* 235
 - NUnit* 235
 - SUnit* 235
 - xUnit* 235
 - Use case 248
 - Actor* 248, 249
 - Defined* 248
 - Extension* 248
 - Extension scenario* 249
 - Guarantee* 249
 - Identifying test case* 252
 - Main success scenario* 248, 249
 - Precondition* 249
 - Primary actor* 249
 - Requirements verification* 252
 - Scope* 249
 - Terminology* 249
-
- ## V
- Value parameter
 - Defined* 46
 - Example* 46
 - Vector 171
 - Defining an iterator* 175
 - Global class ZCL_VECTOR* 171
 - Usage example* 176

W

- W3C 303
- Web Dynpro
 - Screen programming* 257
- Widening cast
 - Compiler check* 160
 - Danger of using* 160
 - Definition* 160
- World Wide Web Consortium → W3C

X

- XML 303
 - Attribute syntax* 307
 - Case sensitivity* 307
 - Data modeling* 305
 - Element syntax* 305
 - Empty element syntax* 305
 - Format* 305
 - Markup convention* 307
 - Meta-markup language* 304
 - Openness* 305
 - Overview* 303
 - Parser* 309
 - Processing concept* 309
 - Purpose* 304
 - Self-describing document* 305

- XML (cont.)
 - Semantic* 307
 - Syntax overview* 305
 - Web service* 328
 - XSLT* 328
- XML attribute 307
- XML document
 - Comment syntax* 306
 - Declaration statement syntax* 306
 - DTD* 308
 - Root element* 305
 - Tree structure* 307
 - Validity* 308
 - Well-formed* 307
 - XML Schema* 308
- XML element 305
 - Nesting* 305
- XML parser 309
 - DOM processing model* 310
 - Function* 310
 - SAX processing model* 310
- XML Schema 308
 - Constraint* 308
 - Standard* 308

Z

- ZIF_COMPARABLE Interface 165