James Wood

Object-Oriented Programming with ABAP™ Objects
Contents at a Glance

PART I Basics
1 Introduction to Object-Oriented Programming ......................... 23
2 Working with Objects .............................................................. 41

PART II Core Concepts
3 Encapsulation and Implementation Hiding ................................. 89
4 Object Initialization and Cleanup ............................................. 107
5 Inheritance ............................................................................. 127
6 Polymorphism ......................................................................... 155
7 Component-Based Design Concepts ........................................... 183
8 Error Handling with Exceptions ................................................ 201
9 Unit Testing with ABAP Unit ...................................................... 233

PART III Case Studies
10 Working with the SAP List Viewer ............................................ 257
11 ABAP Object Services ........................................................... 279
12 Working with XML ................................................................. 303
13 Where to Go From Here ......................................................... 329

APPENDIX
A Debugging Objects ............................................................... 333
B The Author ............................................................................ 343
Contents

Introduction ..................................................................................................... 15

PART I: Basics

1 Introduction to Object-Oriented Programming ................................. 23
  1.1 The Need for a Better Abstraction ............................................... 23
  1.2 Classes and Objects ................................................................. 24
  1.3 Establishing Boundaries ............................................................ 27
  1.4 Reuse ..................................................................................... 29
    1.4.1 Composition ................................................................. 29
    1.4.2 Inheritance ................................................................. 30
    1.4.3 Polymorphism .............................................................. 31
  1.5 Object Management ............................................................... 33
  1.6 UML Tutorial: Class Diagram Basics ............................................ 33
    1.6.1 Classes ......................................................................... 34
    1.6.2 Attributes ................................................................. 36
    1.6.3 Operations ................................................................. 37
    1.6.4 Associations .............................................................. 38
    1.6.5 Notes .......................................................................... 39
  1.7 Summary ................................................................................. 39

2 Working with Objects ............................................................................ 41
  2.1 Syntax Overview .......................................................................... 41
    2.1.1 Defining Classes .......................................................... 42
    2.1.2 Declaring Components ............................................... 42
    2.1.3 Implementing Methods ............................................... 51
  2.2 Creating and Using Objects ......................................................... 52
    2.2.1 Object References ....................................................... 52
    2.2.2 Creating Objects .......................................................... 53
    2.2.3 Object Reference Assignments ..................................... 53
    2.2.4 Working with Instance Components .............................. 54
    2.2.5 Working with Class Components .................................... 57
    2.2.6 Creating Complex Expressions Using Functional
         Methods ........................................................................... 61
Contents

2.3 Building Your First Object-Oriented Program .................................. 64
2.4 Getting Started with the Class Builder ............................................. 71
  2.4.1 Class Pools ......................................................................... 71
  2.4.2 Accessing the Class Builder ................................................ 71
  2.4.3 Creating Classes ............................................................... 72
  2.4.4 Defining Class Components ............................................... 74
  2.4.5 Editing the Class Definition Section Directly ....................... 82
2.5 Case Study: Working with Regular Expressions ............................... 82
2.6 UML Tutorial: Object Diagrams ...................................................... 84
2.7 Summary ........................................................................................ 86

PART II: Core Concepts

3 Encapsulation and Implementation Hiding ............................... 89
  3.1 Lessons Learned from the Procedural Approach ......................... 89
    3.1.1 Decomposing Functional Decomposition ......................... 90
    3.1.2 Case Study: A Procedural Code Library in ABAP .............. 91
    3.1.3 Moving Toward Objects ................................................ 94
  3.2 Data Abstraction with Classes .................................................. 94
  3.3 Defining Component Visibilities ............................................... 95
    3.3.1 Visibility Sections ....................................................... 96
    3.3.2 Friends ....................................................................... 99
  3.4 Hiding the Implementation ...................................................... 101
  3.5 Designing by Contract .......................................................... 102
  3.6 UML Tutorial: Sequence Diagrams ............................................. 103
  3.7 Summary ................................................................................. 105

4 Object Initialization and Cleanup ............................................. 107
  4.1 Creating Objects ..................................................................... 107
  4.2 Controlling Object Initialization with Constructors .................. 111
  4.3 Taking Control of the Instantiation Process .............................. 117
  4.4 Garbage Collection .................................................................. 121
  4.5 Tuning Performance .............................................................. 122
    4.5.1 Design Considerations .................................................. 123
    4.5.2 Lazy Initialization ........................................................ 123
    4.5.3 Reusing Objects ......................................................... 124
    4.5.4 Using Class Attributes ............................................... 124
4.6 UML Tutorial: State Machine Diagrams ................................................. 125
4.7 Summary ........................................................................................ 126

5 Inheritance ....................................................................................... 127
  5.1 Generalization and Specialization ..................................................... 128
  5.2 Inheriting Components .................................................................. 133
    5.2.1 Designing the Inheritance Interface .................................... 133
    5.2.2 Visibility of Instance Components in Subclasses ................. 135
    5.2.3 Visibility of Class Components in Subclasses ....................... 136
    5.2.4 Redefining Methods ........................................................... 136
    5.2.5 Instance Constructors ......................................................... 138
    5.2.6 Class Constructors .............................................................. 140
  5.3 The Abstract and Final Keywords .................................................. 140
    5.3.1 Abstract Classes and Methods ............................................ 140
    5.3.2 Final Classes ....................................................................... 143
    5.3.3 Final Methods .................................................................... 144
  5.4 Inheritance Versus Composition ...................................................... 145
  5.5 Using the Refactoring Assistant ....................................................... 148
  5.6 UML Tutorial: Advanced Class Diagrams Part I ............................... 150
    5.6.1 Generalization .................................................................... 150
    5.6.2 Dependencies and Composition ......................................... 151
    5.6.3 Abstract Classes and Methods ............................................ 152
  5.7 Summary ........................................................................................ 153

6 Polymorphism ..................................................................................... 155
  6.1 Object Reference Assignments Revisited ....................................... 155
    6.1.1 Static and Dynamic Types .................................................. 156
    6.1.2 Casting ............................................................................... 158
  6.2 Dynamic Method Call Binding ......................................................... 160
  6.3 Interfaces ....................................................................................... 163
    6.3.1 Interface Inheritance Versus Implementation Inheritance ... 164
    6.3.2 Defining Interfaces ............................................................. 165
    6.3.3 Implementing Interfaces ...................................................... 167
    6.3.4 Working with Interfaces ..................................................... 170
    6.3.5 Nesting Interfaces .............................................................. 177
  6.4 UML Tutorial: Advanced Class Diagrams Part II .............................. 180
    6.4.1 Interfaces ........................................................................... 180
9 Unit Testing with ABAP Unit ................................................. 233
  9.1 ABAP Unit Overview ...................................................... 234
    9.1.1 The Need for Unit Testing Frameworks ....................... 234
    9.1.2 Unit Testing Terminology ........................................... 235
    9.1.3 Understanding How ABAP Unit Works ......................... 236
  9.2 Creating Unit Test Classes .............................................. 237
    9.2.1 Test Attributes ...................................................... 238
    9.2.2 Test Methods ......................................................... 239
    9.2.3 Managing Fixtures ................................................... 239
    9.2.4 Generating Test Classes for Global Classes .................. 240
  9.3 Case Study: Creating a Unit Test in ABAP Unit ................. 241
  9.4 Executing Unit Tests .................................................... 244
    9.4.1 Integration with the ABAP Workbench ......................... 245
    9.4.2 Integration with the Code Inspector ........................... 245
  9.5 Evaluating Unit Test Results .......................................... 246
  9.6 Moving Toward Test-Driven Development ......................... 247
  9.7 UML Tutorial: Use Case Diagrams .................................... 248
    9.7.1 Use Case Terminology .............................................. 249
    9.7.2 An Example Use Case .............................................. 249
    9.7.3 The Use Case Diagram ............................................. 251
    9.7.4 Use Cases for Requirements Verification .................... 252
    9.7.5 Use Cases and Testing ............................................ 252
  9.8 Summary ........................................................................ 253

PART III: Case Studies

10 Working with the SAP List Viewer ....................................... 257
  10.1 Overview of the SAP Control Framework ......................... 257
    10.1.1 Control Framework Architecture ............................... 258
    10.1.2 Survey of Available Controls ................................... 259
  10.2 Overview of the ALV Object Model .................................. 260
  10.3 Getting Started with the Flight Query Report .................... 261
    10.3.1 Understanding the Report Requirements ..................... 261
    10.3.2 Report Design Using the MVC Design Pattern ............... 261
    10.3.3 Developing the Flight Model Class ............................. 262
    10.3.4 Developing the Report Controller Class ..................... 264
    10.3.5 Implementing the Report View .................................. 267
## Contents

10.4 Event Handling with the ALV Object Model ................................................. 271
10.4.1 Integrating Event Handler Methods into the Controller ....... 271
10.4.2 Registering Event Handler Methods .............................................. 272
10.4.3 Responding to Events .............................................................. 273
10.4.4 Triggering Events on the Frontend ..................................... 274
10.4.5 Timing of Event Handling ....................................................... 275
10.5 UML Tutorial: Communication Diagrams ............................................. 275
10.6 Summary ........................................................................................ 277

11 ABAP Object Services ............................................................................. 279

11.1 Object-Relational Mapping Concepts ................................................... 280
11.2 Persistence Service Overview ............................................................ 280
11.2.1 Managed Objects ........................................................................ 282
11.2.2 Mapping Concepts ................................................................. 284
11.2.3 Understanding the Class Agent API ..................................... 285
11.3 Building Persistent Classes ................................................................. 286
11.3.1 Creating a Persistent Class in the Class Builder ................. 287
11.3.2 Defining Persistent Attributes with the Mapping Assistant ..................................................... 288
11.3.3 Working with Object References ........................................... 292
11.4 Working with Persistent Objects ......................................................... 293
11.4.1 Creating New Persistent Objects ........................................ 294
11.4.2 Reading Persistent Objects Using the Query Service ........... 295
11.4.3 Updating Persistent Objects ............................................... 297
11.4.4 Deleting Persistent Objects ................................................ 298
11.5 UML Tutorial: Advanced Sequence Diagrams .............................. 298
11.5.1 Creating and Deleting Objects ........................................... 298
11.5.2 Depicting Control Logic with Interaction Frames ............ 300
11.6 Summary ........................................................................................ 301

12 Working with XML .................................................................................... 303

12.1 XML Overview ................................................................................... 303
12.1.1 Why Do We Need XML? .................................................... 304
12.1.2 Understanding XML Syntax ................................................ 305
12.1.3 Defining XML Semantics .................................................... 307
12.2 XML Processing Concepts ................................................................. 309
12.2.1 Processing XML Using a Parser ........................................ 309
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.2 Modeling XML with the DOM</td>
<td>310</td>
</tr>
<tr>
<td>12.2.3 Features of the iXML Library</td>
<td>310</td>
</tr>
<tr>
<td>12.3 Case Study: Developing a Reading List ADT</td>
<td>311</td>
</tr>
<tr>
<td>12.4 Case Study: Building an XML Document</td>
<td>314</td>
</tr>
<tr>
<td>12.5 Case Study: Reading an XML Document</td>
<td>320</td>
</tr>
<tr>
<td>12.6 UML Tutorial: Advanced Activity Diagrams</td>
<td>325</td>
</tr>
<tr>
<td>12.7 Summary</td>
<td>327</td>
</tr>
<tr>
<td>13 Where to Go From Here</td>
<td>329</td>
</tr>
</tbody>
</table>

**APPENDIX**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Debugging Objects</td>
<td>333</td>
</tr>
<tr>
<td>A.1 Debugging Objects Using the Classic ABAP Debugger</td>
<td>333</td>
</tr>
<tr>
<td>A.1.1 Displaying and Editing Attributes</td>
<td>333</td>
</tr>
<tr>
<td>A.1.2 Tracing Through Methods</td>
<td>336</td>
</tr>
<tr>
<td>A.1.3 Displaying Events and Event Handler Methods</td>
<td>336</td>
</tr>
<tr>
<td>A.1.4 Viewing Reference Assignments for an Object</td>
<td>337</td>
</tr>
<tr>
<td>A.1.5 Troubleshooting Class-Based Exceptions</td>
<td>338</td>
</tr>
<tr>
<td>A.2 Debugging Objects Using the New ABAP Debugger</td>
<td>340</td>
</tr>
<tr>
<td>B The Author</td>
<td>343</td>
</tr>
</tbody>
</table>

Index                                                                 | 345  |
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-
Introduction

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
Introduction

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:

```abap
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
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.
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.
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.
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
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.
  ENDCCLASS.

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

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. 

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

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.
Inheriting Components

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 Re defin ing 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 Re defining Methods in Global Classes](image)
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.

```abap
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 `lcl_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`.

```plaintext
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. Never-
theless, 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.
Inheritance

* 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](image)

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](image)

### 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.

```plaintext
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).

![Image](image_url)

**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.

```plaintext
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).
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.
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,
```

Figure 5.10 Inheritance Tree for Order Types
get_delivery_date RETURNING re_date
TYPE sydatum.

PRIVATE SECTION.
  DATA: delivery_date TYPE sydatum.
ENDCLASS.

CLASS lcl_delivery IMPLEMENTATION.
  METHOD constructor.
    delivery_date = sy-datum.
  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)
Using the Refactoring Assistant

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).

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.”

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.
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.

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.
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.
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](image)

**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.
# 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
  - Template (cont.)
  - 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
# Index

## C

### C++
- 23

### Casting
- \(?=\) 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
- (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
Index

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_IO 226
  CX_SY_MOVE_CAST_ERROR 205
  CX_SY_ZERO_DIVIDE 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
<table>
<thead>
<tr>
<th>Functional method (cont.)</th>
<th>Inheritance (cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical expression 64</td>
<td>Class component in subclass 136</td>
</tr>
<tr>
<td>LOOP statement 64</td>
<td>Class component scope 136</td>
</tr>
<tr>
<td>MODIFY statement 64</td>
<td>Class constructor behavior 140</td>
</tr>
<tr>
<td>MOVE statement 63</td>
<td>Comparison with composition 145</td>
</tr>
<tr>
<td>Operand 61</td>
<td>Component namespace 136</td>
</tr>
<tr>
<td>Syntax example 63</td>
<td>Copy-and-paste approach 131</td>
</tr>
<tr>
<td>Usage example 61</td>
<td>Defining relationship in global class 131</td>
</tr>
</tbody>
</table>

**Index**

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
- 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_SERIALIZEABLE_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_UTIL_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_STREAM 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
Index

N
Narrowing cast 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 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 55
Assignment operator (=) 53
Illustrated example 53
MOVE statement 53

Mark up language 304
   Element 304
Tag 304
Meta-mark up language 304
   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 46
Model-View-Controller → MVC 261
   ABAP reports 266
   Controller 262
   Coupling 270
   Data binding 264
   Model 262
   Report 262
   View 262
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
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 ZCLVECTOR 171
  Usage example 176
Index

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