CONTENTS

Preface xv

PART ONE

Introduction

1. AN OVERVIEW OF OBJECT-ORIENTED SYSTEMS DEVELOPMENT 3

1.1 Introduction 3

1.2 Two Orthogonal Views of the Software 4

1.3 Object-Oriented Systems Development Methodology 4

1.4 Why an Object Orientation? 5

1.5 Overview of the Unified Approach 6

1.6 Organization of This Book 6

1.7 Summary 11

2. OBJECT BASICS 13

2.1 Introduction 13

2.2 An Object-Oriented Philosophy 14

2.3 Objects 15

2.4 Objects Are Grouped in Classes 16

2.5 Attributes: Object State and Properties 17

2.6 Object Behavior and Methods 18

2.7 Objects Respond to Messages 18

2.8 Encapsulation and Information Hiding 20

2.9 Class Hierarchy 21

2.9.1 Inheritance 23

2.9.2 Multiple Inheritance 25

2.10 Polymorphism 25

2.11 Object Relationships and Associations 26

2.11.1 Consumer-Producer Association 26

2.12 Aggregations and Object Containment 27

2.13 Case Study: A Payroll Program 28

2.13.1 Structured Approach 28

2.13.2 The Object-Oriented Approach 30

2.14 Advanced Topics 32
2.14.1 Object and Identity 32
2.14.2 Static and Dynamic Binding 34
2.14.3 Object Persistence 34
2.14.4 Meta-Classes 34
2.15 Summary 35

3. OBJECT-ORIENTED SYSTEMS DEVELOPMENT LIFE CYCLE 39
3.1 Introduction 39
3.2 The Software Development Process 40
3.3 Building High-Quality Software 42
3.4 Object-Oriented Systems Development: A Use-Case Driven Approach 44
3.4.1 Object-Oriented Analysis—Use-Case Driven 45
3.4.2 Object-Oriented Design 47
3.4.3 Prototyping 47
3.4.4 Implementation: Component-Based Development 49
3.4.5 Incremental Testing 53
3.5 Reusability 53
3.6 Summary 54

PART TWO

Methodology, Modeling, and Unified Modeling Language

4. OBJECT-ORIENTED METHODOLOGIES 61
4.1 Introduction: Toward Unification—Too Many Methodologies 61
4.2 Survey of Some of the Object-Oriented Methodologies 62
4.3 Rumbaugh et al.'s Object Modeling Technique 63
4.3.1 The Object Model 63
4.3.2 The OMT Dynamic Model 63
4.3.3 The OMT Functional Model 64
4.4 The Booch Methodology 65
4.4.1 The Macro Development Process 66
4.4.2 The Micro Development Process 67
4.5 The Jacobson et al. Methodologies 68
4.5.1 Use Cases 68
4.5.2 Object-Oriented Software Engineering: Objectory 70
4.5.3 Object-Oriented Business Engineering 71
4.6 Patterns 71
4.6.1 Generative and Nongenerative Patterns 73
4.6.2 Patterns Template 74
4.6.3 Antipatterns 76
4.6.4 Capturing Patterns 76
4.7 Frameworks 77
4.8 The Unified Approach 78
4.8.1 Object-Oriented Analysis 79
4.8.2 Object-Oriented Design 80
4.8.3 Iterative Development and Continuous Testing 80
4.8.4 Modeling Based on the Unified Modeling Language 80
4.8.5 The UA Proposed Repository 81
4.8.6 The Layered Approach to Software Development 82
4.8.6.1 The Business Layer 83
4.8.6.2 The User Interface (View) Layer 84
4.8.6.3 The Access Layer 84
4.9 Summary 84

5. UNIFIED MODELING LANGUAGE 89
5.1 Introduction 89
5.2 Static and Dynamic Models 90
5.2.1 Static Model 90
5.2.2 Dynamic Model 91
5.3 Why Modeling? 91
5.4 Introduction to the Unified Modeling Language 92
5.5 UML Diagrams 93
5.6 UML Class Diagram 94
5.6.1 Class Notation: Static Structure 94
5.6.2 Object Diagram 94
5.6.3 Class Interface Notation 95
5.6.4 Binary Association Notation 95
5.6.5 Association Role 95
5.6.6 Qualifier 96
5.6.7 Multiplicity 97
5.6.8 OR Association 97
5.6.9 Association Class 97
5.6.10 N-Ary Association 98
5.6.11 Aggregation and Composition 99
5.6.12 Generalization 99
5.7 Use-Case Diagram 101
5.8 UML Dynamic Modeling 103
5.8.1 UML Interaction Diagrams 104
5.8.1.1 UML Sequence Diagram 104
5.8.1.2 UML Collaboration Diagram 105
5.8.2 UML Statechart Diagram 106
5.8.3 UML Activity Diagram 109
5.8.4 Implementation Diagrams 111
5.8.4.1 Component Diagram 112
5.8.4.2 Deployment Diagram 112
5.9 Model Management: Packages and Model Organization 114
5.10 UML Extensibility 115
5.10.1 Model Constraints and Comments 116
5.10.2 Note 117
5.10.3 Stereotype 117
5.11 UML Meta-Model 117
5.12 Summary 118

PART THREE

Object-Oriented Analysis: Use-Case Driven

6. OBJECT-ORIENTED ANALYSIS PROCESS: IDENTIFYING USE CASES 125
6.1 Introduction 125
6.2 Why Analysis Is a Difficult Activity 126
6.3 Business Object Analysis: Understanding the Business Layer 127
6.4 Use-Case Driven Object-Oriented Analysis: The Unified Approach 128
6.5 Business Process Modeling 129
6.6 Use-Case Model 129
6.6.1 Use Cases under the Microscope 131
6.6.2 Uses and Extends Associations 133
6.6.3 Identifying the Actors 134
6.6.4 Guidelines for Finding Use Cases 136
6.6.5 How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue 136
6.6.6 Dividing Use Cases into Packages 137
6.6.7 Naming a Use Case 137

6.7 Developing Effective Documentation 138
6.7.1 Organizing Conventions for Documentation 139
6.7.2 Guidelines for Developing Effective Documentation 139
6.8 Case Study: Analyzing the ViaNet Bank ATM—The Use-Case Driven Process 140
6.8.1 Background 140
6.8.2 Identifying Actors and Use Cases for the ViaNet Bank ATM System 141
6.8.3 The ViaNet Bank ATM Systems' Packages 146
6.9 Summary 146

7. OBJECT ANALYSIS: CLASSIFICATION 151
7.1 Introduction 151
7.2 Classifications Theory 152
7.3 Approaches for Identifying Classes 154
7.4 Noun Phrase Approach 154
7.4.1 Identifying Tentative Classes 154
7.4.2 Selecting Classes from the Relevant and Fuzzy Categories 155
7.4.3 The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach 156
7.4.4 Initial List of Noun Phrases: Candidate Classes 156
7.4.5 Reviewing the Redundant Classes and Building a Common Vocabulary 158
7.4.6 Reviewing the Classes Containing Adjectives 159
7.4.7 Reviewing the Possible Attributes 160
7.4.8 Reviewing the Class Purpose 161
7.5 Common Class Patterns Approach 162
7.5.1 The ViaNet Bank ATM System: Identifying Classes by Using Common Class Patterns 163
7.6 Use-Case Driven Approach: Identifying Classes and Their Behaviors through Sequence/ Collaboration Modeling 164
7.6.1 Implementation of Scenarios 164
7.6.2 The ViaNet Bank ATM System: Decomposing a Use-Case Scenario with a Sequence Diagram: Object Behavior Analysis 165
7.7 Classes, Responsibilities, and Collaborators 169
7.7.1 Classes, Responsibilities, and Collaborators Process 170
7.7.2 The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators 171
7.8 Naming Classes 172
7.9 Summary 174

8. IDENTIFYING OBJECT RELATIONSHIPS, ATTRIBUTES, AND METHODS 177
8.1 Introduction 177
8.2 Associations 178
8.2.1 Identifying Associations 179
8.2.2 Guidelines for Identifying Associations 179
8.2.3 Common Association Patterns 179

8.2.4 Eliminate Unnecessary Associations 180
8.3 Super-Sub Class Relationships 181
8.3.1 Guidelines for Identifying Super-Sub Relationship, a Generalization 181
8.4 A-Part-of Relationships—Aggregation 182
8.4.1 A-Part-of Relationship Patterns 183
8.5 Case Study: Relationship Analysis for the ViaNet Bank ATM System 184
8.5.1 Identifying Classes' Relationships 184
8.5.2 Developing a UML Class Diagram Based on the Use-Case Analysis 184
8.5.3 Defining Association Relationships 185
8.5.4 Defining Super-Sub Relationships 186
8.5.5 Identifying the Aggregation/a-Part-of Relationship 187
8.6 Class Responsibility: Identifying Attributes and Methods 188
8.7 Class Responsibility: Defining Attributes by Analyzing Use Cases and Other UML Diagrams 189
8.7.1 Guidelines for Defining Attributes 189
8.8 Defining Attributes for ViaNet Bank Objects 190
8.8.1 Defining Attributes for the BankClient Class 190
8.8.2 Defining Attributes for the Account Class 190
8.8.3 Defining Attributes for the Transaction Class 191
8.8.4 Defining Attributes for the ATMMachine Class 191
8.9 Object Responsibility: Methods and Messages 191
8.9.1 Defining Methods by Analyzing UML Diagrams and Use Cases 192
8.10 Defining Methods for ViaNet Bank Objects 192
8.10.1 Defining Account Class Operations 192
8.10.2 Defining BankClient Class Operations 193
8.10.3 Defining CheckingAccount Class Operations 193
8.11 Summary 194

PART FOUR

Object-Oriented Design

9. THE OBJECT-ORIENTED DESIGN PROCESS AND DESIGN AXIOMS 199
9.1 Introduction 199
9.2 The Object-Oriented Design Process 200
9.3 Object-Oriented Design Axioms 202
9.4 Corollaries 203
9.4.1 Corollary 1: Uncoupled Design with Less Information Content 204
9.4.1.1 Coupling 204
9.4.1.2 Cohesion 206
9.4.2 Corollary 2: Single Purpose 206
9.4.3 Corollary 3: Large Number of Simpler Classes, Reusability 206
9.4.4 Corollary 4: Strong Mapping 207
9.4.5 Corollary 5: Standardization 208

9.4.6 Corollary 6. Designing with Inheritance 208
9.4.6.1 Achieving Multiple Inheritance in a Single Inheritance System 211
9.4.6.2 Avoiding Inheriting Inappropriate Behaviors 211
9.5 Design Patterns 212
9.6 Summary 214

10. DESIGNING CLASSES 217
10.1 Introduction 217
10.2 The Object-Oriented Design Philosophy 217
10.3 UML Object Constraint Language 218
10.4 Designing Classes: The Process 219
10.5 Class Visibility: Designing Well-Defined Public, Private, and Protected Protocols 219
10.5.1 Private and Protected Protocol Layers: Internal 221
10.5.2 Public Protocol Layer: External 221
10.6 Designing Classes: Refining Attributes 221
10.6.1 Attribute Types 222
10.6.2 UML Attribute Presentation 222
10.7 Refining Attributes for the ViaNet Bank Objects 223
10.7.1 Refining Attributes for the BankClient Class 223
10.7.2 Refining Attributes for the Account Class 223
10.7.3 Refining Attributes for the Transaction Class 224
Problem 10.1 224
10.7.4 Refining Attributes for the ATMMachine Class 224
10.7.5 Refining Attributes for the CheckingAccount Class 224
10.7.6 Refining Attributes for the SavingsAccount Class 224
10.8 Designing Methods and Protocols 225
10.8.1 Design Issues: Avoiding Design Pitfalls 226
10.8.2 UML Operation Presentation 227
10.9 Designing Methods for the ViaNet Bank Objects 227
10.9.1 BankClient Class VerifyPassword Method 228
10.9.2 Account Class Deposit Method 228
10.9.3 Account Class Withdraw Method 229
10.9.4 Account Class CreateTransaction Method 229
10.9.5 Checking Account Class Withdraw Method 230
10.9.6 ATMMachine Class Operations 230
10.10 Packages and Managing Classes 230
10.11 Summary 232

11. ACCESS LAYER: OBJECT STORAGE AND OBJECT INTEROPERABILITY 237
11.1 Introduction 237
11.2 Object Store and Persistence: An Overview 238

11.3 Database Management Systems 239
11.3.1 Database Views 240
11.3.2 Database Models 240
11.3.2.1 Hierarchical Model 240
11.3.2.2 Network Model 241
11.3.2.3 Relational Model 241
11.3.3 Database Interface 242
11.3.3.1 Database Schema and Data Definition Language 242
11.3.3.2 Data Manipulation Language and Query Capabilities 242
11.4 Logical and Physical Database Organization and Access Control 243
11.4.1 Shareability and Transactions 243
11.4.2 Concurrency Policy 244
11.5 Distributed Databases and Client-Server Computing 245
11.5.1 What Is Client-Server Computing? 245
11.5.2 Distributed and Cooperative Processing 248
11.6 Distributed Objects Computing: The Next Generation of Client-Server Computing 250
11.6.1 Common Object Request Broker Architecture 251
11.6.2 Microsoft's ActiveX/DCOM 252
11.7 Object-Oriented Database Management Systems: The Pure World 252
11.7.1 Object-Oriented Databases versus Traditional Databases 254
11.8 Object-Relational Systems: The Practical World 255
11.8.1 Object-Relation Mapping 256
11.8.2 Table-Class Mapping 257
11.8.3 Table-Multiple Classes Mapping 258
11.8.4 Table-Inherited Classes Mapping 258
11.8.5 Tables-Inherited Classes Mapping 258
11.8.6 Keys for Instance Navigation 259
11.9 Multidatabase Systems 260
11.9.1 Open Database Connectivity: Multidatabase Application Programming Interfaces 262
11.10 Designing Access Layer Classes 264
11.10.1 The Process 265
11.11 Case Study: Designing the Access Layer for the ViaNet Bank ATM 269
11.11.1 Creating an Access Class for the BankClient Class 269
11.12 Summary 275
12. VIEW LAYER: DESIGNING INTERFACE OBJECTS 281
12.1 Introduction 281
12.2 User Interface Design as a Creative Process 281
12.3 Designing View Layer Classes 284
12.4 Macro-Level Process: Identifying View Classes by Analyzing Use Cases 285
12.5 Micro-Level Process 287
12.5.1 UI Design Rule 1. Making the Interface Simple 286

12.5.2 UI Design Rule 2. Making the Interface Transparent and Natural 290
12.5.3 UI Design Rule 3. Allowing Users to Be in Control of the Software 290
12.5.3.1 Make the Interface Forgiving 291
12.5.3.2 Make the Interface Visual 291
12.5.3.3 Provide Immediate Feedback 291
12.5.3.4 Avoid Modes 292
12.5.3.5 Make the Interface Consistent 292
12.6 The Purpose of a View Layer Interface 292
12.6.1 Guidelines for Designing Forms and Data Entry Windows 293
12.6.2 Guidelines for Designing Dialog Boxes and Error Messages 296
12.6.3 Guidelines for the Command Buttons Layout 298
12.6.4 Guidelines for Designing Application Windows 299
12.6.5 Guidelines for Using Colors 300
12.6.6 Guidelines for Using Fonts 302
12.7 Prototyping the User Interface 302
12.8 Case Study: Designing User Interface for the ViaNet Bank ATM 304
12.8.1 The View Layer Macro Process 305
12.8.2 The View Layer Micro Process 308
12.8.3 The BankClientAccessUI Interface Object 309
12.8.4 The MainUI Object Interface 309
12.8.5 The AccountTransactionUI Interface Object 309
12.8.6 The CheckingAccountUI and SavingsAccountUI Interface Objects 311
12.8.7 Defining the Interface Behavior 311
12.8.7.1 Identifying Events and Actions for the BankClientAccessUI Interface Object 313
12.8.7.2 Identifying Events and Actions for the MainUI Interface Object 313
12.8.7.3 Identifying Events and Actions for the SavingsAccountUI Interface Object 314
12.8.7.4 Identifying Events and Actions for the AccountTransactionUI Interface Object 315
12.9 Summary 317

PART FIVE

Software Quality

13. SOFTWARE QUALITY ASSURANCE 325
13.1 Introduction 325
13.2 Quality Assurance Tests 326
13.3 Testing Strategies 328
13.3.1 Black Box Testing 328
13.3.2 White Box Testing 329
13.3.3 Top-Down Testing 329
13.3.4 Bottom-Up Testing 330
13.4 Impact of Object Orientation on Testing 330
13.4.1 Impact of Inheritance in Testing 331

PART ONE

INTRODUCTION

The objective of Part I is to provide an overview of object-oriented systems development and why we should study it. In this part, we also look at object basics and the systems development life cycle. Part I consists of Chapters 1, 2, and 3.

CHAPTER 1

An Overview of Object-Oriented Systems Development

Chapter Objectives

You should be able to define and understand

- The object-oriented philosophy and why we need to study it.
- The unified approach.

1.1 INTRODUCTION

Software development is dynamic and always undergoing major change. The methods we will use in the future no doubt will differ significantly from those currently in practice. We can anticipate which methods and tools are going to succeed, but we cannot predict the future. Factors other than just technical superiority will likely determine which concepts prevail.

Today a vast number of tools and methodologies are available for systems development. *Systems development* refers to all activities that go into producing an information systems solution. Systems development activities consist of systems analysis, modeling, design, implementation, testing, and maintenance. A ***software development methodology*** is a series of processes that, if followed, can lead to the development of an application. The software processes describe how the work is to be carried out to achieve the original goal based on the system requirements. Furthermore, each process consists of a number of steps and rules that should be performed during development. The software development process will continue to exist as long as the development system is in operation.

This chapter provides an overview of object-oriented systems development and discusses why we should study it. Furthermore, we study the unified approach, which is the methodology used in this book for learning about object-oriented systems development.

1.2 TWO ORTHOGONAL VIEWS OF THE SOFTWARE

Object-oriented systems development methods differ from traditional development techniques in that the traditional techniques view software as a collection of programs (or functions) and isolated data. What is a program? Niklaus Wirth [8], the inventor of Pascal, sums it up eloquently in his book entitled, interestingly enough, *Algorithms + Data Structures = Programs*: "A software system is a set of mechanisms for performing certain action on certain data."

This means that there are two different, yet complementary ways to view software construction: We can focus primarily on the functions or primarily on the data. The heart of the distinction between traditional system development methodologies and newer object-oriented methodologies lies in their primary focus, where the traditional approach focuses on the functions of the system—What is it doing?—object-oriented systems development centers on the object, which combines data and functionality. As we will see, this seemingly simple shift in focus radically changes the process of software development.

1.3 OBJECT-ORIENTED SYSTEMS DEVELOPMENT METHODOLOGY

Object-oriented development offers a different model from the traditional software development approach, which is based on functions and procedures. In simplified terms, object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified, and reused. Furthermore, it encourages a view of the world as a system of cooperative and collaborating objects. In an object-oriented environment, software is a collection of discrete objects that encapsulate their data as well as the functionality to model real-world "objects." An object orientation yields important benefits to the practice of software construction. Each object has attributes (data) and methods (functions). Objects are grouped into classes; in object-oriented terms, we discover and describe the classes involved in the problem domain.

In an object-oriented system, everything is an object and each object is responsible for itself. For example, every Windows application needs Windows objects that can open themselves on screen and either display something or accept input. A Windows object is responsible for things like opening, sizing, and closing itself. Frequently, when a window displays something, that something also is an object (a chart, for example). A chart object is responsible for things like maintaining its data and labels and even for drawing itself.

The object-oriented environment emphasizes its cooperative philosophy by allocating tasks among the objects of the applications. In other words, rather than writing a lot of code to do all the things that have to be done, you tend to create a lot of helpers that take on an active role, a spirit, and that form a community whose interactions become the application. Instead of saying, "System, compute the payroll of this employee," you tell the employee object, "compute your payroll." This has a powerful effect on the way we approach software development.

1.4 WHY AN OBJECT ORIENTATION?

Object-oriented methods enable us to create sets of objects that work together synergistically to produce software that better model their problem domains than similar systems produced by traditional techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater design and code reuse. Object-oriented development allows us to create modules of functionality. Once objects are defined, it can be taken for granted that they will perform their desired functions and you can seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. Here are some reasons why object orientation works [3–7]:

- *Higher level of abstraction.* The top-down approach supports abstraction at the function level. The object-oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction. The development can proceed at the object level and ignore the rest of the system for as long as necessary. This makes designing, coding, testing, and maintaining the system much simpler.
- *Seamless transition among different phases of software development.* The traditional approach to software development requires different styles and methodologies for each step of the process. Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds. This transition not only can slow the development process but also increases the size of the project and the chance for errors introduced in moving from one language to another. The object-oriented approach, on the other hand, essentially uses the same language to talk about analysis, design, programming, and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.
- *Encouragement of good programming techniques.* A class in an object-oriented system carefully delineates between its interface (specifications of *what* the class can do) and the implementation of that interface (*how* the class does what it does). The routines and attributes within a class are held together tightly. In a properly designed system, the classes will be grouped into subsystems but remain independent; therefore, changing one class has no impact on other classes, and so, the impact is minimized. However, the object-oriented approach is not a panacea; nothing is magical here that will promote perfect design or perfect code. But, by raising the level of abstraction from the function level to the object level and by focusing on the real-world aspects of the system, the object-oriented method tends to promote clearer designs, which are easier to implement, and provides for better overall communication. Using object-oriented language is not strictly necessary to achieve the benefits of an object orientation. However, an object-oriented language such as C++, Smalltalk, or Java adds support for object-oriented design and makes it easier to produce more modular and reusable code via the concept of class and inheritance [5].
- *Promotion of reusability.* Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers (other objects). Within this framework, the class does not

concern itself with the rest of the system or how it is going to be used within a particular system. This means that classes are designed generically, with reuse as a constant background goal. Furthermore, the object orientation adds inheritance, which is a powerful technique that allows classes to be built from each other, and therefore, only differences and enhancements between the classes need to be designed and coded. All the previous functionality remains and can be reused without change.

1.5 OVERVIEW OF THE UNIFIED APPROACH

This book is organized around the unified approach for a better understanding of object-oriented concepts and system development. The ***unified approach*** (UA) is a methodology for software development that is proposed by the author, and used in this book. The UA, based on methodologies by Booch, Rumbaugh, and Jacobson, tries to combine the best practices, processes, and guidelines along with the Object Management Group's unified modeling language. The ***unified modeling language*** (UML) is a set of notations and conventions used to describe and model an application. However, the UML does not specify a methodology or what steps to follow to develop an application; that would be the task of the UA. Figure 1–1 depicts the essence of the unified approach. The heart of the UA is Jacobson's use case. The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs. In its simplest usage, you capture a use case by talking to typical users and discussing the various ways they might want to use the system. The use cases are entered into all other activities of the UA.

The main advantage of an object-oriented system is that the class tree is dynamic and can grow. Your function as a developer in an object-oriented environment is to foster the growth of the class tree by defining new, more specialized classes to perform the tasks your applications require. After your first few projects, you will accumulate a repository or class library of your own, one that performs the operations your applications most often require. At that point, creating additional applications will require no more than assembling classes from the class library. Additionally, applying lessons learned from past developmental efforts to future projects will improve the quality of the product and reduce the cost and development time.

This book uses a layered architecture to develop applications. ***Layered architecture*** is an approach to software development that allows us to create objects that represent tangible elements of the business independent of how they are represented to the user through an interface or physically stored in a database. The layered approach consists of view or user interface, business, and access layers. This approach reduces the interdependence of the user interface, database access, and business control; therefore, it allows for a more robust and flexible system.

1.6 ORGANIZATION OF THIS BOOK

Chapter 2 introduces the basics of the object-oriented approach and why we should study it. Furthermore, we learn that the main thrust of the object-oriented approach

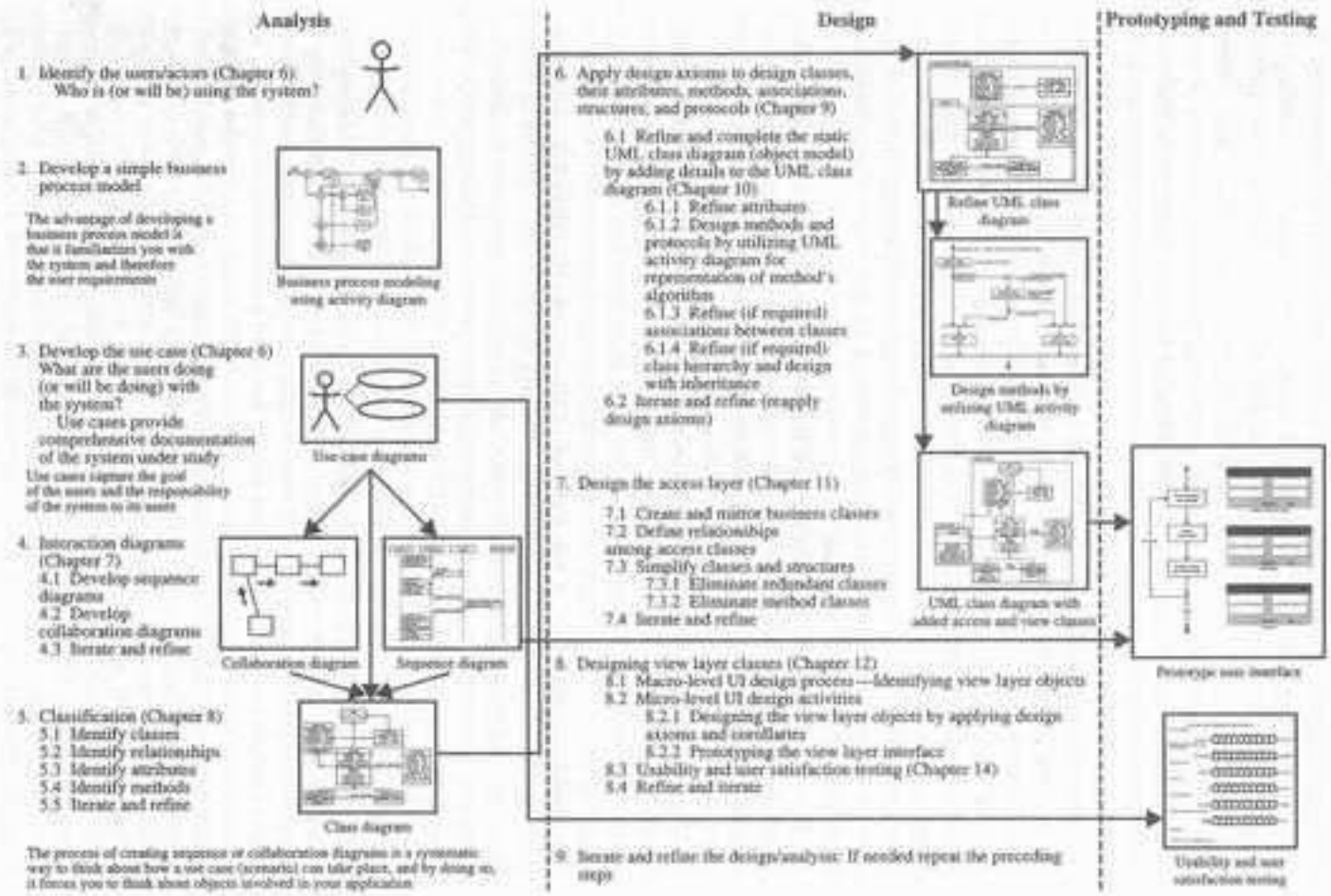


FIGURE 1–1
The unified approach road map.

is to provide a set of objects that closely reflects the underlying application. For example, the user who needs to develop a financial application could develop it in a financial language with considerably less difficulty. An object-oriented approach allows the base concepts of the language to be extended to include ideas closer to those of its application. You can define a new data type (object) in terms of an existing data type until it appears that the language directly supports the primitives of the application. The real advantage of using an object-oriented approach is that you can build on what you already have.

Chapter 3 explains the object-oriented system development life cycle (SDLC). The essence of the software process is the transformation of users' needs in the application domain into a software solution that is executed in the implementation domain. The concept of use case or set of scenarios can be a valuable tool for understanding the users' needs. We will learn that an object-oriented approach requires a more rigorous process up front to do things right. We need to spend more time gathering requirements, developing a requirements model, developing an analysis model, then turning that into the design model. This chapter concludes Part I (Introduction) of the book.

Chapter 4 is the first chapter of Part II (Methodology, Modeling, and Unified Modeling Language). Chapter 4 looks at the current trend in object-oriented methodologies, which is toward combining the best aspects of today's most popular methods. We also take a closer look at the unified approach.

Chapter 5 describes the unified modeling language in detail. The UML merges the best of the notations developed by the so-called three amigos—Booch, Rumbaugh, and Jacobson—in their attempt to unify their modeling efforts. The unified modeling language originally was called the *unified method* (UM). However, the methodologies that were an integral part of the Booch, Rumbaugh, and Jacobson methods were separated from the notation; and the unification efforts of Booch, Jacobson, and Rumbaugh eventually focused more on the graphical modeling language and its semantics and less on the underlying process and methodology. They sum up the reason for the name as follows:

> The UML is intended to be a universal language for modeling systems, meaning that it can express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways. Thus, a single universal process for all styles of development did not seem possible or even desirable: what works for a shrink-wrapped software project is probably wrong for a one-of-a-kind globally distributed, human-critical family of systems. However, the UML can be used to express the artifacts of all of these different processes, namely, the models that are produced. Our move to the UML does not mean that we are ignoring the issues of process. Indeed, the UML assumes a process that is use case driven, architecture-centered, iterative and incremental. It is our observation that the details of this general development process must be adapted to the particular development culture or application domain of a specific organization. We are also working on process issues, but we have chosen to separate the modeling language from the process. By making the modeling language and its process nearly independent, we therefore give users and other methodologists considerable degrees of freedom to craft a specific process yet still use a common language

> of expression. This is not unlike blueprints for buildings: there is a commonly understood language for blueprints, but there are a number of different ways to build, depending upon the nature of what is being built and who is doing the building. This is why we say that the UML is essentially the language of blueprints for software. [2, p. 5]

The UML has become the standard notation for object-oriented modeling systems. It is an evolving notation that is still under development. Chapter 5 concludes this part of the book.

Chapter 6 is the first chapter of Part III (Object-Oriented Analysis: Use-Case Driven). The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. The main task of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the users' perspective rather than that of the machine. This analysis is focused on the domain of the problem and concerned with externally visible behavior [1]. Other activities of object-oriented analysis are to identify the objects that make up the system, their behaviors, and their relationships. Chapter 6 explains the object-oriented analysis process and provides a detailed discussion of use-case driven object-oriented analysis. The use case is a typical interaction between a user and a computer system utilized to capture users' goals and needs. The use-case model represents the users' view of the system or the users' needs. In its simplest usage, you capture a use case by talking to typical users and discussing the various things they might want to do with the system. The heart of the UA is the Jacobson's use case. The use cases are a part of all other activities of the UA (see Figure 1–1).

The main activities of the object-oriented analysis are to identify classes in the system. Finding classes is one of the hardest activities in the analysis. There is no such a thing as the *perfect* class structure or the *right* set of objects. Nevertheless, several techniques, such as the use-case driven approach or the noun phrase and other classification methods, can offer us guidelines and general rules for identifying the classes in the given problem domain. Furthermore, identifying classes is an iterative process; and as you gain more experience, you will get better at identifying classes. In Chapter 7, we study four approaches to identifying classes: the noun phrase, class categorization, use-case driven, and class responsibilities collaboration approaches.

In an object-oriented environment, objects take on an active role in a system. These objects do not exist in isolation but interact with each other. Indeed, their interactions and relationships become the application. Chapter 8 describes the guidelines for identifying object relationships, attributes, and methods. Chapter 8 concludes the object-oriented analysis part of the book.

Chapter 9 is the first chapter of Part IV (Object-Oriented Design). In this part of the book, we learn how to elevate the analysis model into actual objects that can perform the required task. Emphasis is shifted from the application domain to implementation. The classes identified during analysis provide a framework for the

design phase. Object-oriented design and object-oriented analysis are distinct disciplines, but they are intertwined as well. Object-oriented development is highly incremental; in other words, you start with object-oriented analysis, model it, create an object-oriented design, then some more of each, again and again, gradually refining and completing the models of the system. Part IV describes object-oriented design. Other activities of object-oriented design are the user interface design and prototype and the design of the database access. Chapter 9 explains the object-oriented design process and design axioms. The main objective of the axiomatic approach is to formalize the design process and assist in establishing a scientific foundation for the object-oriented design process, so as to provide a fundamental basis of the creation of systems. These guidelines, with incremental and evolutionary styles of software development, will provide you a powerful way for designing systems.

In Chapter 10, we look at guidelines and approaches that you can use to design objects and their methods. Although the design concepts to be discussed in this chapter are general, we concentrate on designing the business objects. Chapter 10 describes the first step of the object-oriented design process, which consists of applying design axioms to design objects and their attributes, methods, associations, structures, and protocols.

Chapter 11 introduces issues regarding object storage, relational and object-oriented database management systems, and object interoperability. We then look at current trends to combine object and relational systems to provide a very practical solution to the problem of object storage. We conclude the chapter with how to design the access layer objects. The main idea behind the access layer is to create a set of classes that know how to communicate with the data source, regardless of their format, whether it is a file, relational database, mainframe, or Internet. The access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. Access layer classes provide easy migration to emerging distributed object technology, such as CORBA and DCOM. Furthermore, they should be able to address the (relatively) modest needs of two-tier client-server architectures as well as the difficult demands of fine-grained, peer-to-peer distributed-object architectures.

The main goals of view layer objects are to display and obtain the information needed in an accessible, efficient manner. The design of your user interface and view layer objects, more than anything else, affects how a user interacts and therefore experiences the application. A well-designed user interface has visual appeal that motivates users to use the application. In Chapter 12, we learn how to design the view layer by mapping the user interface objects to the view layer objects; we look at user interface design rules, which are based on several design axioms, and finally at the guidelines for developing a graphical user interface. This chapter concludes the object-oriented design part of the book.

Chapter 13 is the first chapter of Part V (Software Quality), which discusses different aspects of software quality and testing. In Chapter 13, we look at testing strategies, the impact of object orientation on software quality, and guidelines for developing comprehensive test cases and plans that can detect and identify potential problems before delivering the software to the users.

Usability testing is different from quality assurance testing in that, rather than finding programming defects, you assess how well the interface or the software fits users' needs and expectations. Furthermore, to ensure usability of the system, we must measure user satisfaction throughout the system development. Chapter 14 describes usability and user satisfaction tests. We study how to develop user satisfaction and usability tests based on the use cases identified during the analysis phase.

Appendix A contains a template for documenting a system requirement. The template in this appendix is not to replace the documentation capability of a CASE tool but to be used as an example for issues or modeling elements that are needed for creating an effective system document. Finally, Appendix B provides a review of Windows and graphical user interface basics.

1.7 SUMMARY

In an object-oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real-world "objects." Once objects are defined, you can take it for granted that they will perform their desired functions and so seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. The object-oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.

An object orientation produces systems that are easier to evolve, more flexible, more robust, and more reusable than a top-down structure approach. An object orientation

- Allows working at a higher level of abstraction.
- Provides a seamless transition among different phases of software development.
- Encourages good development practices.
- Promotes reusability.

The unified approach (UA) is the methodology for software development proposed and used in this book. Based on the Booch, Rumbaugh, and Jacobson methodologies, the UA consists of the following concepts:

- Use-case driven development.
- Utilizing the unified modeling language for modeling.
- Object-oriented analysis (utilizing use cases and object modeling).
- Object-oriented design.
- Repositories of reusable classes and maximum reuse.
- The layered approach.
- Incremental development and prototyping.
- Continuous testing.

KEY TERMS

Layered architecture (p. 6)
Software development methodology (p. 3)

Unified approach (UA) (p. 6)
Unified modeling language (UML) (p. 6)

REVIEW QUESTIONS

1. What is system development methodology?
2. What are orthogonal views of software?
3. What is the object-oriented systems development methodology?
4. How does the object-oriented approach differ from the traditional top-down approach?
5. What are the advantages of object-oriented development?
6. Describe the components of the unified approach.

PROBLEMS

1. Object-oriented development already is big in industry and will grow bigger in the years to come. More and more companies will use an object-oriented approach to build their complex (multimedia, workflow, database, artificial intelligence, real-time, and client-server) systems. Research the library or WWW to obtain an article about a major company that has used object-oriented technology to build its future information system.
2. Consult the WWW or library to obtain an article on a real-world application that has incorporated object-oriented tools. Write a summary report of your findings.
3. Consult the WWW or library to obtain an article on an object-oriented methodology. Write a summary of your findings.
4. Consult the WWW or library to obtain an article that describes a large software system that was behind schedule, over budget, and failed to achieve the expected functionality. What factors were blamed, and how could the failure have been avoided?
5. Consult the WWW or library to obtain an article on visual and object-oriented programming. Write a paper based on your findings.

REFERENCES

1. Anderson, Michael; and Bergstrand, John. "Formalizing Use Cases with Message Sequence Charts." Master's thesis, Department of Communication Systems at Lund Institute of Technology, 1995.
2. Booch, Grady; Jacobson, Ivar; and Rumbaugh, James. *The Unified Modeling Language, Notation Guide Version 1.1*. http://www.rational.com/uml/html/notation (September 1997).
3. Edwards, John. "Lessons Learned in Practical Application of the OO Paradigm." Object-Oriented Systems Symposium, Washington, DC, January 1990.
4. Graham, Ian. *Object Oriented Methods*, 2d ed. Reading MA: Addison-Wesley Publishing Company, 1994.
5. King, Gary Warren. "Object-Oriented Really Is Better Than Structured." http://www.oz.net/~gking/whyoop.htm (September 20, 1995).
6. Lassesen, Kenneth M. "Leveraging the Mainframe in Business Solutions with Microsoft Access and Visual Basic." *TechEd* (1995).
7. Burnett, Margaret; Goldberg, Adele; and Lewis, Ted, eds. *Visual Object-Oriented Programming: Concepts and Environments*. Englewood Cliffs, NJ: Prentice-Hall/Manning Publications, 1995.
8. Wirth, Niklaus. *Algorithms + Data Structure = Programs*. Englewood Cliffs, NJ: Prentice-Hall,1975.

CHAPTER 2

Object Basics

Chapter Objectives

You should be able to define and understand

- Why we need to study object-oriented concepts.
- Objects and classes—and their differences.
- Class attributes and methods.
- The concept of messages.
- Class hierarchy inheritance and multiple inheritance.
- Object relationships and associations.
- Encapsulation and information hiding.
- Polymorphism.
- Advantage of the object-oriented approach.
- Aggregations.
- Static and dynamic binding.
- Object persistence.
- Meta-classes.

2.1 INTRODUCTION

If there is a single motivating factor behind object-oriented system development, it is the desire to make software development easier and more natural by raising the level of abstraction to the point where applications can be implemented in the same terms in which they are described by users. Indeed, the name *object* was chosen because "everyone knows what an object is." The real question, then, is not so much "What is an object?" but "What do objects have to do with system development?"

Let us develop the notion of an object through an example. A car is an *object:* a real-world entity, identifiably separate from its surroundings. A car has a well-defined set of attributes in relation to other objects—such as color, manufacturer, cost, and owner—and a well-defined set of things you normally do with it—drive it, lock it, tow it, and carry passengers in it. In an object model, we call the former *properties* or *attributes* and the latter *procedures* or *methods*. ***Properties*** (or ***attributes***) describe the state (data) of an object. ***Methods*** (procedures) define its behavior. Stocks and bonds might be objects for a financial investment application. Parts and assemblies might be objects of a bill of materials application. Therefore, we can conclude that an object is whatever an application wants to "talk" about.

2.2 AN OBJECT-ORIENTED PHILOSOPHY

Most programming languages provide programmers with a way of describing processes. Although most programming languages are computationally equivalent (a process describable in one is describable in another), the ease of description, reusability, extensibility, readability, computational efficiency, and ability to maintain the description can vary widely depending on the language used. It has been said that, "One should speak English for business, French for seduction, German for engineering, and Persian for poetry." A similar quip could be made about programming languages.

A language, natural or programming, provides its users a base set of constructs. Many programming languages derive their base ideas from the underlying machine. The machine may "understand" or recognize data types such as integers, floating point numbers, and characters; and the programming language will represent precisely these types as structures. The machine may understand indirect addressing modes or base plus offset addressing; and the programming language correspondingly will represent the concepts of pointers and vectors. Nothing is terribly wrong with this, but these concepts are pretty far removed from those of a typical application. In practical terms, it means that a user or programmer is implementing, say, a financial investment (risk, returns, growth, and the various investment instruments) into the much lower-level primitives of the programming language, like vectors or integers.

It would be marvelous if we could build a machine whose underlying primitives were precisely those of an application. The user who needs to develop a financial application could develop a financial investment machine directly in financial investment machine language with no mental translation at all. Clearly, it is too expensive to design new hardware on a per-application basis. But, it really is not necessary to go this far, because programming languages can bridge the semantic gap between the concepts of the application and those of the underlying machine.

A fundamental characteristic of ***object-oriented programming*** is that it allows the base concepts of the language to be extended to include ideas and terms closer to those of its applications. New data types can be defined in terms of existing data types until it appears that the language directly supports the primitives of the application. In our financial investment example, a bond (data type) may be defined that has the same understanding within the language as a character data type. A

buy operation on a bond can be defined that has the same understanding as the familiar plus (+) operation on a number. Using this data abstraction mechanism, it is possible to create new, higher-level, and more specialized data abstractions. You can work directly in the language, manipulating the kinds of "objects" required by you or your application, without having to constantly struggle to bridge the gap between how to conceive of these objects and how to write the code to represent them.

The fundamental difference between the object-oriented systems and their traditional counterparts is the way in which you approach problems. Most traditional development methodologies are either algorithm centric or data centric. In an ***algorithm-centric methodology,*** you think of an algorithm that can accomplish the task, then build data structures for that algorithm to use. In a ***data-centric methodology,*** you think how to structure the data, then build the algorithm around that structure.

In an object-oriented system, however, the algorithm and the data structures are packaged together as an object, which has a set of attributes or properties. The state of these attributes is reflected in the values stored in its data structures. In addition, the object has a collection of procedures or methods—things it can do—as reflected in its package of methods. The attributes and methods are equal and inseparable parts of the object; one cannot ignore one for the sake of the other. For example, a car has certain attributes, such as *color, year, model,* and *price,* and can perform a number of operations, such as *go, stop, turn left,* and *turn right.*

The traditional approach to software development tends toward writing a lot of code to do all the things that have to be done. The code is the plans, bricks, and mortar that you use to build structures. You are the only active entity; the code, basically, is just a lot of building materials. The object-oriented approach is more like employing a lot of helpers that take on an active role and form a community whose interactions become the application. Instead of saying, "System, write the value of this number to the screen," we tell the number object, "Write yourself." This has a powerful effect on the way we approach software development.

In summary, object-oriented programming languages bridge the semantic gap between the ideas of the application and those of the underlying machine, and objects represent the application data in a way that is not forced by hardware architecture.

2.3 OBJECTS

The term *object* was first formally utilized in the Simula language, and objects typically existed in Simula programs to simulate some aspect of reality [5]. The term ***object*** means a combination of data and logic that represents some real-world entity. For example, consider a Saab automobile. The Saab can be represented in a computer program as an object. The "data" part of this object would be the car's name, color, number of doors, price, and so forth. The "logic" part of the object could be a collection of programs (show mileage, change mileage, stop, go).

In an object-oriented system, everything is an object: A spreadsheet, a cell in a spreadsheet, a bar chart, a title in a bar chart, a report, a number or telephone number, a file, a folder, a printer, a word or sentence, even a single character all are examples of an object. Each of us deals with objects daily. Some objects, such as a telephone, are so common that we find them in many places. Other objects, like the folders in a file cabinet or the tools we use for home repair, may be located in a certain place [7].

When developing an object-oriented application, two basic questions always arise:

- What objects does the application need?
- What functionality should those objects have?

For example, every Windows application needs Windows objects that can either display something or accept input. Frequently, when a window displays something, that something is an object as well.

Conceptually, each object is responsible for itself. For example, a Windows object is responsible for things like opening, sizing, and closing itself. A chart object is responsible for maintaining its data and labels and even for drawing itself.

Programming in an object-oriented system consists of adding new kinds of objects to the system and defining how they behave. Frequently, these new object classes can be built from the objects supplied by the object-oriented system.

2.4 OBJECTS ARE GROUPED IN CLASSES

Many of us find it fairly natural to partition the world into objects, properties (states), and procedures (behavior). This is a common and useful partitioning or classification. Also, we routinely divide the world along a second dimension: We distinguish classes from instances. When an eagle flies over us, we have no trouble identifying it as an eagle and not an airplane. What is occurring here? Even though we might never have seen this particular bird before, we can immediately identify it as an eagle. Clearly, we have some general idea of what eagles look like, sound like, what they do, and what they are good for—a generic notion of eagles, or what we call the *class* eagle.

Classes are used to distinguish one type of object from another. In the context of object-oriented systems, a ***class*** is a set of objects that share a common structure and a common behavior; a single object is simply an ***instance*** of a class [3]. A class is a specification of structure (instance variables), behavior (methods), and inheritance for objects. (Inheritance is discussed later in this chapter.)

Classes are an important mechanism for classifying objects. The chief role of a class is to define the properties and procedures (the state and behavior) and applicability of its instances. The class car, for example, defines the *property* color. Each individual car (formally, each instance of the class car) will have a value for this property, such as maroon, yellow, or white.

In an object-oriented system, a *method* or behavior of an object is defined by its class. Each object is an instance of a class. There may be many different classes.

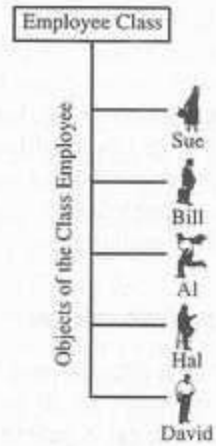


FIGURE 2–1
Sue, Bill, Al, Hal, and David are instances or objects of the class Employee.

Think of a class as an object template (see Figure 2–1). Every object of a given class has the same data format and responds to the same instructions. For example, employees, such as Sue, Bill, Al, Hal, and David all are instances of the class Employee. You can create unlimited instances of a given class. The instructions responded to by each of those instances of employee are managed by the class. The data associated with a particular object is managed by the object itself. For example, you might have two employee objects, one called Al and the other Bill. Each employee object is responsible for its own data, such as social security number, address, and salary. In short, the objects you use in your programs are instances of classes. You can use any of the predefined classes that are part of an object-oriented system or you can create your own.

2.5 ATTRIBUTES: OBJECT STATE AND PROPERTIES

Properties represent the state of an object. Often, we want to refer to the description of these properties rather than how they are represented in a particular programming language. In our example, the properties of a car, such as color, manufacturer, and cost, are abstract descriptions (see Figure 2–2). We could represent

FIGURE 2–2
The attributes of a car object.

each property in several ways in a programming language. For color, we could choose to use a sequence of characters such as *red,* or the (stock) number for red paint, or a reference to a full-color video image that paints a red swatch on the screen when displayed. Manufacturer could be denoted by a name, a reference to a manufacturer object, or a corporate tax identification number. Cost could be a floating point number, a fixed point number, or an integer in units of pennies or even lira. The importance of this distinction is that an object's abstract state can be independent of its physical representation.

2.6 OBJECT BEHAVIOR AND METHODS

When we talk about an elephant or a car, we usually can describe the set of things we normally do with it or that it can do on its own. We can drive a car, we can ride an elephant, or the elephant can eat a peanut. Each of these statements is a description of the object's behavior. In the object model, object behavior is described in methods or procedures. A method implements the behavior of an object. Basically, a method is a function or procedure that is defined for a class and typically can access the internal state of an object of that class to perform some operation. *Behavior* denotes the collection of methods that abstractly describes what an object is capable of doing. Each procedure defines and describes a particular behavior of an object. The object, called the *receiver,* is that on which the method operates. Methods encapsulate the behavior of the object, provide interfaces to the object, and hide any of the internal structures and states maintained by the object. Consequently, procedures provide us the means to communicate with an object and access its properties. The use of methods to exclusively access or update properties is considered good programming style, since it limits the impact of any later changes to the representation of the properties.

Objects take responsibility for their own behavior. In an object-oriented system, one does not have to write complicated code or utilize extensive conditional checks through the use of case statements for deciding what function to call based on a data type or class. For example, an employee object knows how to compute its salary. Therefore, to compute an employee salary, all that is required is to send the *computePayroll* "message" to the employee object. This simplification of code simplifies application development and maintenance.

2.7 OBJECTS RESPOND TO MESSAGES

An object's capabilities are determined by the methods defined for it. Methods conceptually are equivalent to the function definitions used in procedural languages. For example, a draw method would tell a chart how to draw itself. However, to do an operation, a message is sent to an object. Objects perform operations in response to messages. For example, when you press on the brake pedal of a car, you send a *stop* message to the car object. The car object knows how to respond to the *stop* message, since brakes have been designed with specialized parts such as brake pads and drums precisely to respond to that message. Sending the same *stop* message to a different object, such as a tree, however, would be mean-

ingless and could result in an unanticipated (if any) response. Following a set of conventions, or protocols, protects the developer or user from unauthorized data manipulation.

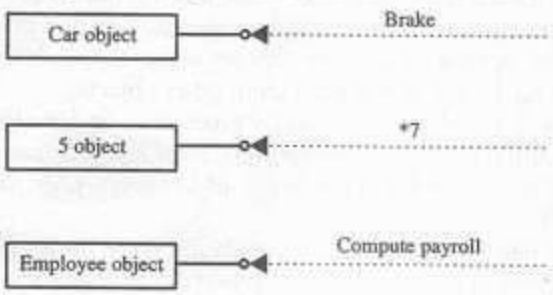
Messages essentially are nonspecific function calls: We would send a *draw* message to a chart when we want the chart to draw itself. A message is different from a subroutine call, since different objects can respond to the same message in different ways. For example, cars, motorcycles, and bicycles will all respond to a *stop* message, but the actual operations performed are object specific.

In the top example, depicted in Figure 2–3, we send a *Brake* message to the *Car* object. In the middle example, we send a *multiplication* message to *5* object followed by the number by which we want to multiply 5. In the bottom example, a *Compute Payroll* message is sent to the *Employee* object, where the employee object knows how to respond to the *Payroll* message. Note that the message makes no assumptions about the class of the receiver or the arguments; they are simply objects. It is the receiver's responsibility to respond to a message in an appropriate manner. This gives you a great deal of flexibility, since different objects can respond to the same message in different ways. This is known as *polymorphism* (more on polymorphism later in this chapter), meaning "many shapes (behaviors)." Polymorphism is the main difference between a message and a subroutine call.

Methods are similar to functions, procedures, or subroutines in more traditional programming languages, such as COBOL, Basic, or C. The area where methods and functions differ, however, is in how they are invoked. In a Basic program, you call the subroutine (e.g., GOSUB 1000); in a C program, you call the function by name (e.g., draw chart). In an object-oriented system, you invoke a method of an object by sending an object a message. A message is much more general than a function call. To draw a chart, you would send a *draw* message to the chart object. Notice that *draw* is a more general instruction than, say, *draw a chart*. That is because the *draw* message can be sent to many other objects, such as a line or circle, and each object could act differently.

It is important to understand the difference between methods and messages. Say you want to tell someone to make you French onion soup. Your instruction is the

FIGURE 2–3
Objects respond to messages according to methods defined in its class.



message, the way the French onion soup is prepared is the method, and the French onion soup is the object. In other words, the message is the instruction and the method is the implementation. An object or an instance of a class understands messages. A message has a name, just like a method, such as cost, set cost, cooking time. An object understands a message when it can match the message to a method that has a same name as the message. To match up the message, an object first searches the methods defined by its class. If found, that method is called up. If not found, the object searches the superclass of its class. If it is found in a superclass, then that method is called up. Otherwise, it continues the search upward. An error occurs only if none of the superclasses contains the method.

A message differs from a function in that a function says how to do something and a message says what to do. Because a message is so general, it can be used over and over again in many different contexts. The result is a system more resilient to change and more reusable, both within an application and from one application to another.

2.8 ENCAPSULATION AND INFORMATION HIDING

Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings. As in conventional programming, some languages permit arbitrary access to objects and allow methods to be defined outside of a class. For example, Simula provides no protection, or information hiding, for objects, meaning that an object's data, or ***instance variables***, may be accessed wherever visible. However, most object-oriented languages provide a well-defined interface to their objects through classes. For example, C++ has a very general ***encapsulation*** protection mechanism with public, private, and protected members. Public members (member data and member functions) may be accessed from anywhere. For instance, the *computePayroll* method of an employee object will be public. Private members are accessible only from within a class. An object data representation, such as a list or an array, usually will be private. Protected members can be accessed only from subclasses.

Often, an object is said to *encapsulate* the data and a program. This means that the user cannot see the inside of the object "capsule," but can use the object by calling the object's methods [8]. Encapsulation or information hiding is a design goal of an object-oriented system. Rather than allowing an object direct access to another object's data, a message is sent to the target object requesting information. This ensures not only that instructions are operating on the proper data but also that no object can operate directly on another object's data. Using this technique, an object's internal format is insulated from other objects.

Another issue is per-object or per-class protection. In ***per-class protection***, the most common form (e.g., Ada, C++, Eiffel), class methods can access any object of that class and not just the receiver. In ***per-object protection***, methods can access only the receiver.

An important factor in achieving encapsulation is the design of different classes of objects that operate using a common ***protocol***, or object's user interface. This

means that many objects will respond to the same message, but each will perform the message using operations tailored to its class. In this way, a program can send a generic message and leave the implementation up to the receiving object, which reduces interdependencies and increases the amount of interchangeable and reusable code.

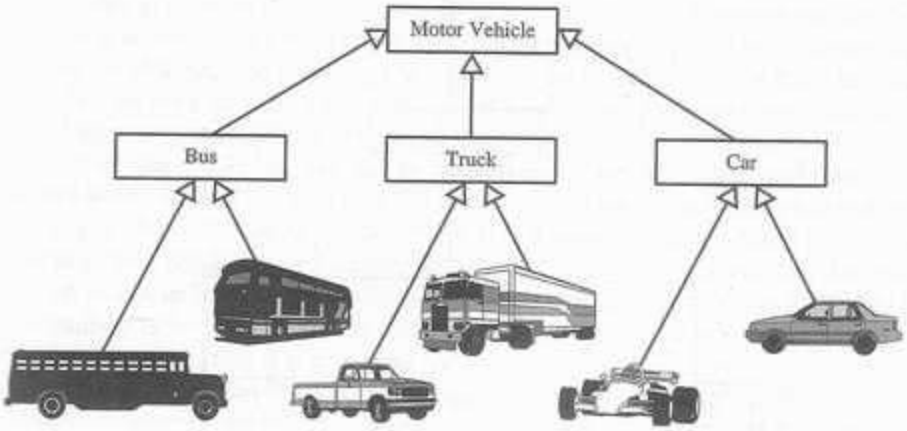
A car engine is an example of encapsulation. Although engines may differ in implementation, the interface between the driver and the car is through a common protocol: Step on the gas to increase power and let up on the gas to decrease power. Since all drivers know this protocol, all drivers can use this method in all cars, no matter what engine is in the car. That detail is insulated from the rest of the car and from the driver. This simplifies the manipulation of car objects and the maintenance of code.

Data abstraction is a benefit of the object-oriented concept that incorporates encapsulation and polymorphism. Data are abstracted when they are shielded by a full set of methods and only those methods can access the data portion of an object.

2.9 CLASS HIERARCHY

An object-oriented system organizes classes into a subclass-superclass hierarchy. Different properties and behaviors are used as the basis for making distinctions between classes and subclasses. At the top of the ***class hierarchy*** are the most general classes and at the bottom are the most specific. The family car in Figure 2–4 is a subclass of car. A ***subclass*** inherits all of the properties and methods (procedures) defined in its ***superclass***. In this case, we can drive a family car just as we can drive any car or, indeed, almost any motor vehicle. Subclasses generally add new methods and properties specific to that class. Subclasses may refine or constrain the state and behavior inherited from its superclass. In our example, race cars

FIGURE 2–4
Superclass/subclass hierarchy.



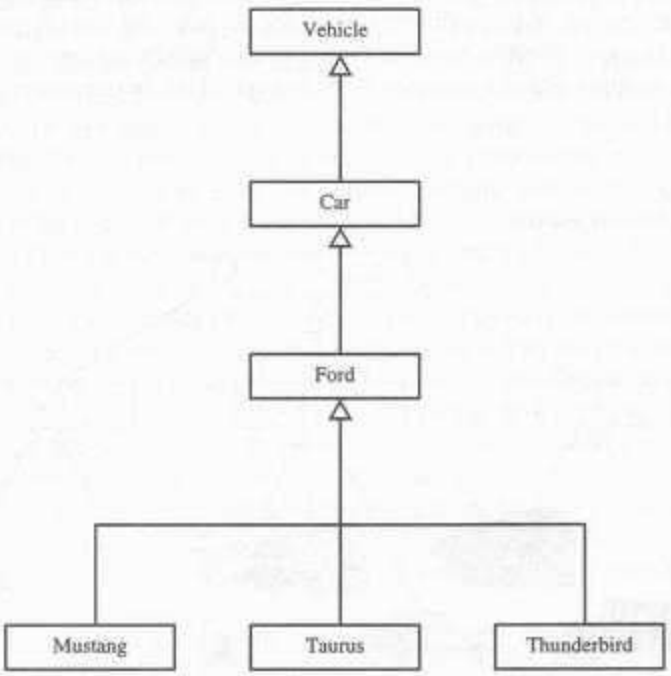
only have one occupant, the driver. In this manner, subclasses modify the attribute (number of passengers) of its superclass, Car.

By contrast, superclasses generalize behavior. It follows that a more general state and behavior is modeled as one moves up the superclass-subclass hierarchy (or simply class hierarchy) and a more specific state is modeled as one moves down.

It is evident from our example that the notion of subclasses and superclasses is relative. A class may simultaneously be the subclass to some class and a superclass to another class(es). Truck is a subclass of a motor vehicle and a superclass of both 18-wheeler and pickup. For example, Ford is a class that defines Ford car objects (see Figure 2–5). However, more specific classes of Ford car objects are Mustang, Taurus, Escort, and Thunderbird. These classes define Fords in a much more specialized manner than the Ford car class itself. Since the Taurus, Escort, Mustang, and Thunderbird classes are more specific classes of Ford cars, they are considered subclasses of class Ford and the Ford class is their superclass. However, the Ford class may not be the most general in our hierarchy. For instance, the Ford class is the subclass of the Car class, which is the subclass of the Vehicle class. Object-oriented notation will be covered in Chapter 5, the chapter on object-oriented modeling.

The car class defines how a car behaves. The Ford class defines the behavior of Ford cars (in addition to cars in general), and the Mustang class defines the behavior of Mustangs (in addition to Ford cars in general). Of course, if all you

FIGURE 2–5
Class hierarchy for Ford class.



wanted was a Ford Mustang object, you would write only one class, Mustang. The class would define exactly how a Ford Mustang car operates. This methodology is limiting because, if you decide later to create a Ford Taurus object, you will have to duplicate most of the code that describes not only how a vehicle behaves but also how a car, and specifically a Ford, behaves.

This duplication occurs when using a procedural language, since there is no concept of hierarchy and inheriting behavior. An object-oriented system eliminates duplicated effort by allowing classes to share and reuse behaviors.

You might find it strange to define a Car class. After all, what is an instance of the Car class? There is no such thing as a generic car. All cars must be of some make and model. In the same way, there are no instances of Ford class. All Fords must belong to one of the subclasses: Mustang, Escort, Taurus, or Thunderbird. The Car class is a formal class, also called an abstract class. ***Formal*** or ***abstract classes*** have no instances but define the common behaviors that can be inherited by more specific classes.

In some object-oriented languages, the terms *superclass* and *subclass* are used instead of ***base*** and ***derived***. In this book, the terms *superclass* and *subclass* are used consistently.

2.9.1 Inheritance

Inheritance is the property of object-oriented systems that allows objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The parent class also is known as the *base class* or *superclass*. Inheritance provides programming by extension as opposed to programming by reinvention [10]. The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods, a class also inherits the behaviors and attributes of all of its superclasses.

For example, the Car class defines the general behavior of cars. The Ford class inherits the general behavior from the Car class and adds behavior specific to Fords. It is not necessary to redefine the behavior of the car class; this is inherited. Another level down, the Mustang class inherits the behavior of cars from the Car class and the even more specific behavior of Fords from the Ford class. The Mustang class then adds behavior unique to Mustangs.

Assume that all Fords use the same braking system. In that case, the *stop* method would be defined in class Ford (and not in Mustang class), since it is a behavior shared by all objects of class Ford. When you step on the brake pedal of a Mustang, you send a *stop* message to the Mustang object. However, the *stop* method is not defined in the Mustang class, so the hierarchy is searched until a *stop* method is found. The *stop* method is found in the Ford class, a superclass of the Mustang class, and it is invoked (see Figure 2–6).

In a similar way, the Mustang class can inherit behaviors from the Car and the Vehicle classes. The behaviors of any given class really are behaviors of its su-

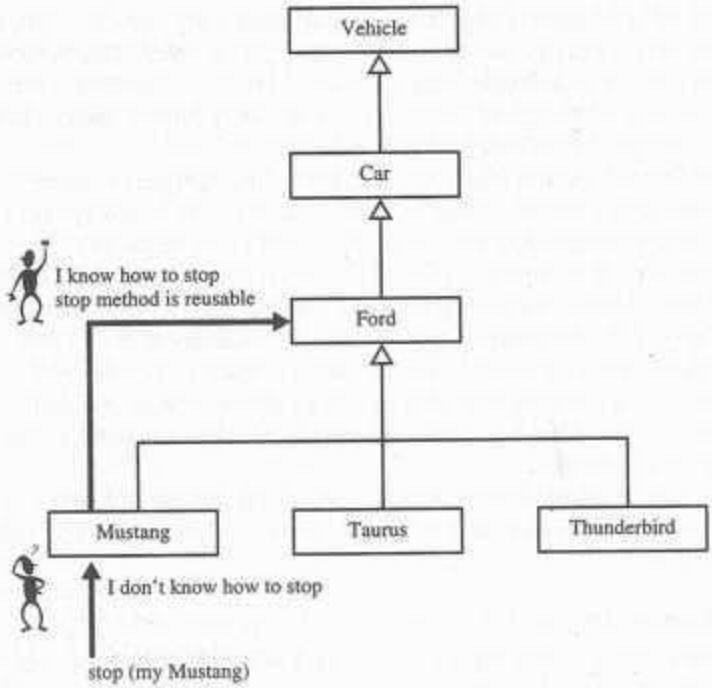


FIGURE 2-6
Inheritance allows reusability.

perclass or a collection of classes. This straightforward process of inheritance prevents you from having to redefine every behavior into every level or reinvent the wheel, or brakes, for that matter.

Suppose that most Ford cars use the same braking system, but the Thunderbird has its own antilock braking system. In this case, the Thunderbird class would redefine the *stop* method. Therefore, the *stop* method of the Ford class would never be invoked by a Thunderbird object. However, its existence higher up in the class hierarchy causes no conflict, and other Ford cars will continue to use the standard braking system.

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. A previous example was a Windows object changing into an icon and then back again, which involves changing a base class between a Windows class and an Icon class. More specifically, *dynamic inheritance* refers to the ability to add, delete, or change parents from objects (or classes) at run time.

In object-oriented programming languages, variables can be declared to hold or reference objects of a particular class. For example, a variable declared to reference a motor vehicle is capable of referencing a car or a truck or any subclass of motor vehicle.

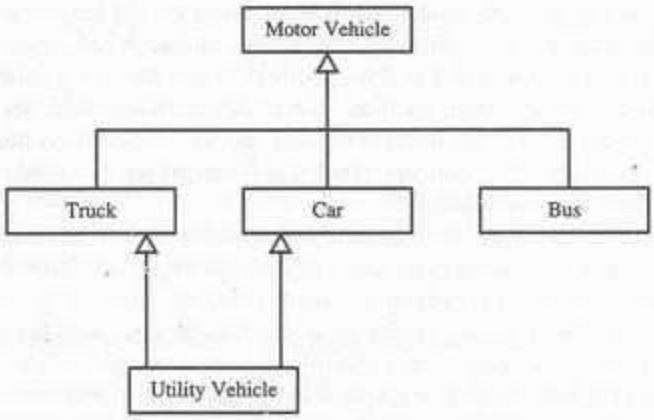


FIGURE 2–7
Utility vehicle inherits from both the Car and Truck classes.

2.9.2 Multiple Inheritance

Some object-oriented systems permit a class to inherit its state (attributes) and behaviors from more than one superclass. This kind of inheritance is referred to as ***multiple inheritance***. For example, a utility vehicle inherits attributes from both the Car and Truck classes (see Figure 2–7).

Multiple inheritance can pose some difficulties. For example, several distinct parent classes can declare a member within a multiple inheritance hierarchy. This then can become an issue of choice, particularly when several superclasses define the same method. It also is more difficult to understand programs written in multiple inheritance systems.

One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and then add an object of another class as an attribute.

2.10 POLYMORPHISM

Poly means "many" and *morph* means "form." In the context of object-oriented systems, it means objects that can take on or assume many different forms. ***Polymorphism*** means that the same operation may behave differently on different classes [11]. Booch [1–3] defines *polymorphism* as the relationship of objects of many different classes by some common superclass; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way. For example, consider how driving an automobile with a manual transmission is different from driving a car with an automatic transmission. The manual transmission requires you to operate the clutch and the shift, so in addition to all other mechanical controls, you also need information on when to shift gears. Therefore, although driving is a behavior we perform with all cars (and all motor vehicles), the specific behavior can be different, depending on the kind of car we

are driving. A car with an automatic transmission might implement its *drive* method to use information such as current speed, engine RPM, and current gear. Another car might implement the *drive* method to use the same information but require additional information, such as "the clutch is depressed." The method is the same for both cars, but the implementation invoked depends on the type of car (or the class of object). This concept, termed *polymorphism*, is a fundamental concept of any object-oriented system.

Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation details to the objects involved. Since no assumption is made about the class of an object that receives a message, fewer dependencies are needed in the code and, therefore, maintenance is easier. For example, in a payroll system, manager, office worker, and production worker objects all will respond to the *compute payroll* message, but the actual operations performed are object specific.

2.11 OBJECT RELATIONSHIPS AND ASSOCIATIONS

Association represents the relationships between objects and classes. For example, in the statement "a pilot *can fly* planes" (see Figure 2–8), the italicized term is an association.

Associations are bidirectional; that means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. For example, *can fly* connects a pilot to certain airplanes. The inverse of *can fly* could be called *is flown by*.

An important issue in association is ***cardinality***, which specifies how many instances of one class may relate to a single instance of an associated class [12]. Cardinality constrains the number of related objects and often is described as being "one" or "many." Generally, the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of cylinders in an engine is four, six, or eight. Consider a client-account relationship where one client can have one or more accounts and vice versa (in case of joint accounts); here the cardinality of the client-account association is many to many.

2.11.1 Consumer-Producer Association

A special form of association is a consumer-producer relationship, also known as a ***client-server association*** or a ***use relationship***. The ***consumer-producer relationship*** can be viewed as one-way interaction: One object requests the service of another object. The object that makes the request is the consumer or client, and the object that receives the request and provides the service is the producer or

FIGURE 2–8
Association represents the relationship among objects, which is bidirectional.

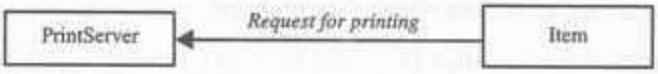


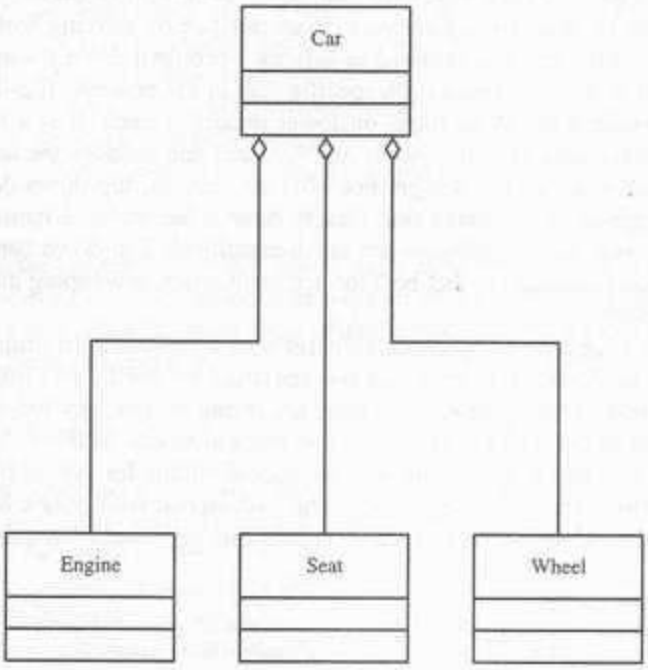
FIGURE 2–9
The consumer/producer association.

server. For example, we have a print object that prints the consumer object. The print producer provides the ability to print other objects. Figure 2–9 depicts the consumer-producer association.

2.12 AGGREGATIONS AND OBJECT CONTAINMENT

All objects, except the most basic ones, are composed of and may contain other objects. For example, a spreadsheet is an object composed of cells, and cells are objects that may contain text, mathematical formulas, video, and so forth. Breaking down objects into the objects from which they are composed is decomposition. This is possible because an object's attributes need not be simple data fields; attributes can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as ***aggregation***, where an attribute can be an object itself. For instance, a car object is an aggregation of engine, seat, wheels, and other objects (see Figure 2–10).

FIGURE 2–10
A Car object is an aggregation of other objects such as engine, seat, and wheel objects.



2.13 CASE STUDY: A PAYROLL PROGRAM

Consider a payroll program that processes employee records at a small manufacturing firm. The company has several classes of employees with particular payroll requirements and rules for processing each. This company has three types of employees:

1. *Managers* receive a regular salary.
2. *Office Workers* receive an hourly wage and are eligible for overtime after 40 hours.
3. *Production Workers* are paid according to a piece rate.

We will walk through traditional and object-oriented system development approaches to highlight their similarities and differences with an eye on object-oriented concepts.[1] The main focus of this exercise is to better understand the object-oriented approach. To keep the discussion simple, many issues (such as data flow diagrams, entity-relationship diagrams, feasibility analysis, and documentation) will not be addressed here.

2.13.1. Structured Approach

The traditional structured analysis/structured design (SA/SD) approach relies on modeling the processes that manipulate the given input data to produce the desired output. The first few steps in SA/SD involve creation of preliminary data flow diagrams and data modeling. Data modeling is a systems development methodology concerned with the system's entities, their associations, and their activities. Data modeling is accomplished through the use of entity-relationship diagrams. The SA/SD approach encourages the top-down design (also known as *top-down decomposition* or *stepwise refinement),* characterized by moving from a general statement about the process involved in solving a problem down toward more and more detailed statements about each specific task in the process. Top-down design works well because it lets us focus on fewer details at once. It is a logical technique that encourages orderly system development and reduces the level of complexity at each stage of the design. For obvious reasons, top-down design works best when applied to problems that clearly have a hierarchical nature. Unfortunately, many real-world problems are not hierarchical. Top-down function-based design has other limitations that become apparent when developing and maintaining large systems.

Top-down design works by continually refining a problem into simpler and simpler chunks. Each chunk is analyzed and specified by itself, with little regard (if any) for the rest of the system. This, after all, is one reason why top-down design is so effective at analyzing a problem. The method works well for the initial design of a system and helps ensure that the specifications for the problem are met and solved. However, each program element is designed with only a limited set of requirements in mind. Since it is unlikely that this exact set of requirements will

[1]Structured approach is a valid approach. It is estimated that 25 percent of firms use structured development approaches. Therefore, it will continue to play a role in systems development.

return in the next problem, the program's design and code are not general and reusable. Top-down design does not preclude the creation of general routines that are shared among many programs, but it does not encourage it. Indeed, the idea of combining reusable programs into a system is a bottom-up approach, quite the opposite of the top-down style [9].

Once the system modeling and analysis have been completed, we can proceed to design. During the design phase, many issues must be studied. These include user interface design (input and output), hardware and software issues such as system platform and operating systems, and data or database management issues. In addition, people and procedural issues, such as training and documentation, must be addressed.

Finally, we proceed to implementing the system using a procedural language. Most current programming languages, such as FORTRAN, COBOL, and C, are based on procedural programming. That is, the programmer tells the computer exactly how to process each piece of data, presents selections from which the user can choose, and codes an appropriate response for each choice. Today's applications are much more sophisticated and developed with more demanding requirements than in the past, which makes systems development using these tools much more difficult.

In a procedural approach such as C or COBOL, the payroll program would include conditional logic to check the employee code and compute the payroll accordingly:

```
FOR EVERY EMPLOYEE DO
    BEGIN
        IF  employee = manager      THEN
            CALL  computeManagerSalary
        IF  employee = office worker  THEN
            CALL  computeOfficeWorkerSalary
        IF  employee = production worker  THEN
            CALL  computeProductionWorkerSalary
    END
```

If new classes of employees are added, such as temporary office workers ineligible for overtime or junior production workers who receive an hourly wage plus a lower piece rate, then the main logic of the application must be modified to accommodate these requirements:

```
FOR EVERY EMPLOYEE DO
    BEGIN
        IF  employee = manager  THEN
            CALL  computeManagerSalary
        IF  employee = office worker  THEN
            CALL  computeOfficeWorkerSalary
        IF  employee = production worker  THEN
            CALL  computeProductionWorkerSalary
```

```
        IF  employee = temporary office worker  THEN
              CALL  computeTemporaryOfficeWorkerSalary
        IF  employee = junior production worker  THEN
              CALL  computeJuniorProductionWorkerSalary
END
```

Similarly, other areas of the program that pertain to data entry or reporting might have to be modified to take into account new kinds of employees. A procedural language does not lend itself easily to writing code in a generic and reusable way. Therefore you must provide detailed processing steps for each type (class) of employee. The main problem with this traditional way of programming is the following:

> The introduction of new classes of data with different needs requires changing the main logic of the program. It also may require the addition of new code in many different areas of the application.

This problem limits a programmer's ability to reuse code, since each function or procedure is tied to the data on which it operates. On the other hand, object-oriented programming allows the programmer to solve problems by creating logical objects that incorporate both data and functionality in a unit. You can create fully tested and debugged classes of objects that can be reused in other programs and this, in turn, can reduce the amount of code that must be written for successive applications.

2.13.2 The Object-Oriented Approach

Object-oriented systems development consists of

- Object-oriented analysis.
- Object-oriented information modeling.
- Object-oriented design.
- Prototyping and implementation.
- Testing, iteration, and documentation.

Object-oriented software development encourages you to view the problem as a system of cooperative objects. Object-oriented analysis shares certain aspects with the structured approach, such as determining the system requirements. However, one major difference is that we do not think of data and procedures separately, because objects incorporate both. When developing an object-oriented application, two basic questions always arise:

- What objects does the application need?
- What functionality should those objects have?

Each object is entirely responsible for itself. For example, an employee object is responsible for things (operations) like computing payrolls, printing paychecks, and storing data about itself, such as its name, address, and social security num-

ber. The first task in object-oriented analysis is to find the class of objects that will compose the system. At the first level of analysis, we can look at the physical entities in the system. That is, who are the players and how do they cooperate to do the work of the system? These entities (objects) could be individuals, organizations, machines, units of information, molecules, pictures, or whatever else makes sense in the context of the real-world system. This usually is a very good starting point for deciding what *classes* to design. At this level, you must look at the physical entities in the system. In the process of developing the model, the objects that emerge can help us establish a workable system. However, Coad and Yourdon [4] have listed the following clues for finding the candidate classes and objects:

- ***Persons.*** What role does a person play in the system? For example, customers, employees of which the system needs to keep track.
- ***Places.*** These are physical locations, buildings, stores, sites or offices about which the system keeps information.
- ***Things or events.*** These are events, points in time that must be recorded. For example, the system might need to remember when a customer makes an order; therefore, an order is an object. Associated with things remembered are attributes (after all, things to remember are objects) such as who, what, when, where, how, or why. For example, some of the data or attributes of *order object* are customer-ID (who), date-of-order (when), soup-ID (what), and so on.

Next, we need to identify the hierarchical relation between superclasses and subclasses. Another task in object-oriented analysis is to identify the attributes (properties) of objects, such as color, cost, and manufacturer. Identifying behavior (methods) is next. The main question to ask is What services must a class provide? The answer to the question allows us to identify the methods a class must contain. Once you identify the overall system's responsibilities and the information it needs to remember, you can assign each responsibility to the class to which it logically belongs.

As in the structured approach, we need to model the system's objects and their relationships. For example, the objects in our payroll system are Employee, Manager, Production Worker, and Temporary Office Worker.

Other objects (things) are part of the system, such as the paycheck or the product being made and the process being used to make the product. However, here we focus on the class of Employees. The payroll program would have different classes of Employees corresponding to the different types of employees in the company (see Figure 2–11). Every employee object would know how to calculate its payroll according to its own requirements.

The goal of object-oriented analysis is to identify objects and classes that support the problem domain and system's requirements. Object-oriented design identifies and defines additional objects and classes that support an implementation of the requirements [4]. For example, during design you might need to add objects for the user interface for the systems; that is, data entry windows, browse windows, and the like.

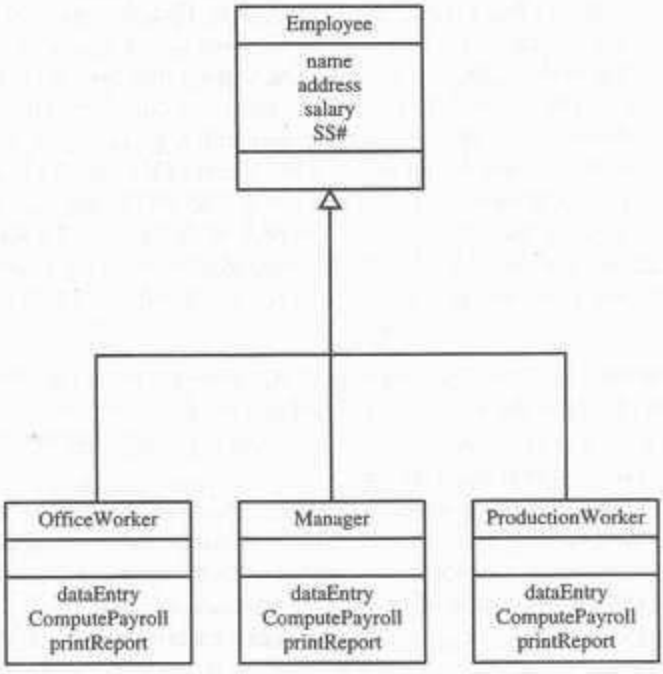


FIGURE 2–11
Class hierarchy for the payroll application.

The main program would be written in a general way that looped through all of the employees and sent a message to each employee to calculate its payroll:

```
FOR EVERY EMPLOYEE DO
    BEGIN
        employee computePayroll
    END
```

If a new class of employee were added, a class for that type of employee would have to be created (see Figure 2–12). This class would know how to calculate its payroll. Unlike the procedural approach, the main program and other related parts of the program would not have to be modified; changes would be limited to the addition of a new class.

2.14 ADVANCED TOPICS

2.14.1 Object and Identity

A special feature of object-oriented systems is that every object has its own unique and immutable identity. An object's identity comes into being when the object is created and continues to represent that object from then on. This identity never is confused with another object, even if the original object has been deleted. The identity name never changes even if all the properties of the object change—it is

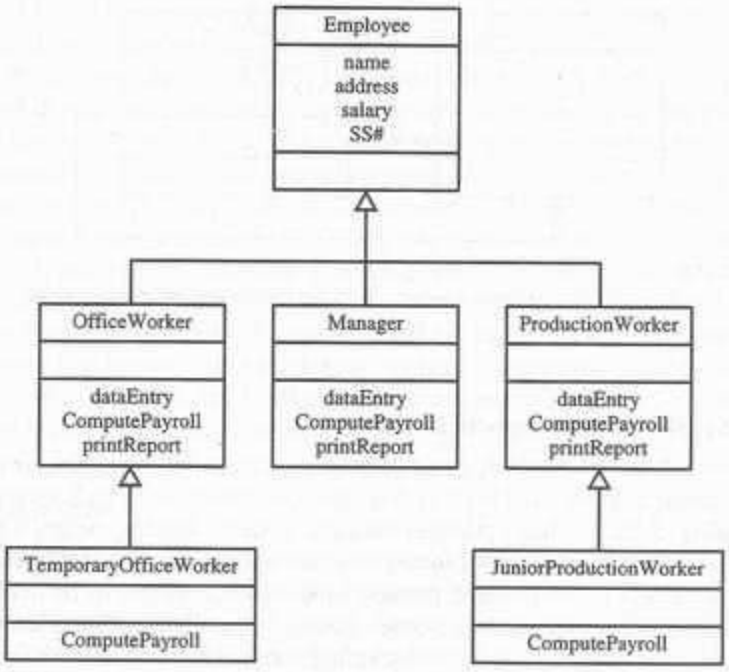


FIGURE 2–12
The hierarchy for the payroll application.

independent of the object's state. In particular, the identity does not depend on the object's name, or its key, or its location. All these can change with no effect on being able to recognize the object as the "same one."

In an object system, object identity often is implemented through some kind of ***object identifier*** (OID) or unique identifier (UID). An OID is dispensed by a part of the object-oriented programming system that is responsible for guaranteeing the uniqueness of every identifier. OIDs are never reused.

As another example (see Figure 2–13), cars may have an "owner" property of class Person. A particular instance of a car, a black Saab 900S, is owned by person instance named Hal. In an object system, the relationship between the car and Hal can be implemented and maintained by a reference. A reference in an object-oriented system is similar to a pointer in other programming languages. Pointers refer directly to the address of the thing they point to in the physical memory. ***Object references*** directly denote the object to which they refer. References often are implemented by using the UID of the object as the reference, since the UID guarantees object identity over time. Fortunately, we need not be concerned with or manage the UID. Most object-oriented systems will perform that transparently.

In Figure 2–13, the owner property of a car contains a reference to the person instance named Hal. In addition, an object may refer back to an object that refers to it. In this example, a person has an *owns* property that contains a reference to the car instance. References may or may not carry class information with them.

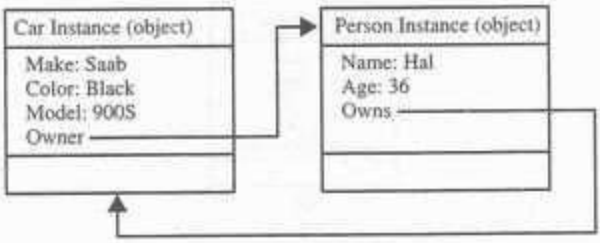


FIGURE 2-13
The owner property of a car contains a reference to the person instance named Hal.

2.14.2 Static and Dynamic Binding

The process of determining (dynamically) at run time which function to invoke is termed ***dynamic binding***. Making this determination earlier, at compile time, is called ***static binding***. Static binding optimizes the calls; dynamic binding occurs when polymorphic calls are issued. Not all function invocations require dynamic binding.

Dynamic binding allows some method invocation decisions to be deferred until the information is known. A run-time selection of methods often is desired, and even required, in many applications, including databases and user interaction (e.g., GUIs). For example, a cut operation in an Edit submenu may pass the cut operation (along with parameters) to any object on the Desktop, each of which handles the message in its own way. If an (application) object can cut many kinds of objects, such as text and graphic objects, many overloaded cut methods, one per type of object to be cut, are available in the receiving object; the particular method being selected is based on the actual type of object being cut (which in the GUI case is not available until run time) [6].

2.14.3 Object Persistence

Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process in which they were created. A file or a database can provide support for objects having a longer lifeline—longer than the duration of the process for which they were created. From a language perspective, this characteristic is called ***object persistence***. An object can persist beyond application session boundaries, during which the object is stored in a file or a database, in some file or database form. The object can be retrieved in another application session and will have the same state and relationship to other objects as at the time it was saved. The lifetime of an object can be explicitly terminated. After an object is deleted, its state is inaccessible and its persistent storage is reclaimed. Its identity, however, is never reused, not even after the object is deleted. Object storage and its access from the database will be covered in Chapter 11.

2.14.4 Meta-Classes

Earlier, we said that, in an object-oriented system, everything is an object: numbers, arrays, records, fields, files, forms, and so forth. How about a class? Is a class

an object? Yes, a class is an object. So, if it is an object, it must belong to a class (in classical object-oriented systems, anyway). Indeed, such a class belongs to a class called a ***meta-class***, or a class of classes. For example, classes must be implemented in some way; perhaps with dictionaries for methods, instances, and parents and methods to perform all the work of being a class. This can be declared in a class named *meta-class*. The meta-class also can provide services to application programs, such as returning a set of all methods, instances, or parents for review (or even modification). Therefore, we can say that all objects are instances of a class and all classes are instances of a meta-class. The meta-class is a class and therefore an instance of itself.

Generally speaking, meta-classes are used by the compiler. For example, the meta-classes handle messages to classes, such as constructors, "new," and "class variables" (a term from Smalltalk), which are variables shared between all instances of a class (static member data in C++).

2.15 SUMMARY

The goal of object-oriented programming is to make development easier, quicker, and more natural by raising the level of abstraction to the point where applications can be implemented in the same terms in which they are described by the application domain. The main thrust of object-oriented programming is to provide the user with a set of objects that closely reflects the underlying application. The user who needs to develop a financial application could develop it in a financial language with considerably less difficulty. Object-oriented programming allows the base concepts of the language to be extended to include ideas closer to those of its application. You can define a new data type (object) in terms of an existing data type until it appears that the language directly supports the primitives of the application. The real advantage of using the object-oriented approach is that you can build on what you already have.

Object-oriented software development is a significant departure from the traditional structured approach. The main advantage of the object-oriented approach is the ability to reuse code and develop more maintainable systems in a shorter amount of time. Additionally, object-oriented systems are better designed, more resilient to change, and more reliable, since they are built from completely tested and debugged classes.

Rather than treat data and procedures separately, object-oriented systems link both closely into objects. Events occur when objects respond to messages. The objects themselves determine the response to the messages, allowing the same message to be sent to many objects.

Each object is an instance of a class. Classes are organized hierarchically in a class tree, and subclasses inherit the behavior of their superclasses. Good object-oriented programming uses encapsulation and polymorphism, which, when used in the definition of classes, result in completely reusable abstract data classes. Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process for which they were created. A file or a database can provide support for objects having a longer lifeline—longer than the duration of the process for which they were created.

KEY TERMS

Abstract classes (p. 23)
Aggregation (p. 27)
Algorithm-centric methodology (p. 15)
Association (p. 26)
Attribute (p. 14)
Base classes (p. 23)
Cardinality (p. 26)
Class hierarchy (p. 21)
Class (p. 16)
Client-server association (p. 26)
Consumer-producer relationship (p. 26)
Data-centric methodology (p. 15)
Derived classes (p. 23)
Dynamic binding (p. 34)
Dynamic inheritance (p. 24)
Encapsulation (p. 20)
Formal classes (p. 23)
Inheritance (p. 23)
Instance (p. 16)
Instance variable (p. 20)
Message (p. 19)
Meta-class (p. 35)
Method (p. 14)
Multiple inheritance (p. 25)
Object (p. 15)
Object identifier (OID) (p. 33)
Object-oriented programming (p. 14)
Object persistence (p. 34)
Object reference (p. 33)
Per-class protection (p. 20)
Per-object protection (p. 20)
Polymorphism (p. 25)
Property (p. 14)
Protocol (p. 20)
Static binding (p. 34)
Subclass (p. 21)
Superclass (p. 21)
Use relationship (p. 26)

REVIEW QUESTIONS

1. What is an object?
2. What is the main advantage of object-oriented development?
3. What is polymorphism?

4. What is the difference between an object's methods and an object's attributes?
5. How are classes organized in an object-oriented environment?
6. How does object-oriented development eliminate duplication?
7. What is inheritance?
8. What is data abstraction?
9. Why is encapsulation important?
10. What is a protocol?
11. What is the difference between a method and a message?
12. Why is polymorphism useful?
13. How are objects identified in an object-oriented system?
14. What is the lifetime of an object and how can you extend the lifetime of an object?
15. What is association?
16. What is a consumer-producer relationship?
17. What is a formal class?
18. What is an instance?

For questions 19–25 match the term to the definition:

a. object
b. class
c. method
d. class hierarchy
e. inheritance
f. message
g. polymorphism

19. _____ An object template.
20. _____ A unit of functionality.
21. _____ The scheme for representing the relationships between classes.
22. _____ The generic message-sending scheme that allows flexibility in design.
23. _____ The scheme for sharing operations and data between related classes.
24. _____ A high-level instruction to perform an operation on an object.
25. _____ The implementation of a high-level operation for a specific class of objects.

PROBLEMS

1. Identify the attributes and methods of a dishwasher object.
2. Identify all the attributes and methods of the checkbook object. Write a short description of services that each method will provide.
3. If you are in market to buy a car, which attributes or services are relevant to you.
4. Identify objects in a payroll system.
5. Create a class hierarchy to organize the following drink classes: alcoholic, nonalcoholic, grape juice, mineral water, lemonade, beer, and wine. (Hint: At the top of the hierarchy are the most general classes and at the bottom are the most specific. Classes should be related to one another in superclass-subclass hierarchies.)
6. Assume the drinks in problem 6 have the following characteristics:

   - Alcoholic drinks are not for drivers or children.
   - Nonalcoholic drinks are thirst quenching.
   - Wine is made of grapes and for adults only.
   - Grape juice is made from grapes and has the taste of a fruit.

- Mineral water is bubbling and does not taste like fruit.
- Lemonade is bubbling and tastes like a fruit.

How would you define the class hierarchy? (Hint: Utilize the inheritance capability of an object-oriented system.)

REFERENCES

1. Booch, Grady. *Software Engineering with Ada*, 2d ed. Menlo Park, CA: Benjamin-Cummings, 1987.
2. Booch, Grady. *Software Components with Ada, Structures, Tools, and Subsystems.* Menlo Park, CA: Benjamin-Cummings, 1987.
3. Booch, Grady. *Object-Oriented Design with Applications*, 2d ed. Menlo Park, CA: Benjamin-Cummings, 1994.
4. Coad, P.; and Yourdon, E. *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press Computing Series, 1991.
5. Dahl, O. J.; and Nygaard, K. "SIMULA—An Agol Based Simulation Language." *Communications of the ACM* 9, no. 9 (1966), pp. 671–78.
6. Garfinkel, Simson L.; and Michael K. Mahoney. *NeXTSTEP Programming Step One: Object-Oriented Applications*. New York: Springer-Verlag, 1993.
7. IBM. "Human-User Interaction, Object-Oriented User Interface." http://www.ibm.com/ibm/hci, 1997.
8. Kim, Won. *Introduction to Object-Oriented Databases*. Cambridge, MA: The MIT Press, 1990.
9. King, Gary Warren. "Object-Oriented Really Is Better Than Structured." 1996, http://www.oz.net/~gking/whyoop.htm.
10. LaLonde, Wilf R.; and Pugh, John R. *Inside Smalltalk*, vol.1. Englewood Cliffs, NJ: Prentice-Hall Engineering, Science, and Math, 1990.
11. Rumbaugh, James; Blaha, Michael; Permerlani, William; Eddy, Frederick; and Lorensen, William. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

CHAPTER 3

Object-Oriented Systems Development Life Cycle

He who does not lay his foundations beforehand may by great abilities do so afterwards, although with great trouble to the architect and danger to the building.

—Niccolo Machiavelli
The Prince

Chapter Objectives

You should be able to define and understand

- The software development process.
- Building high-quality software.
- Object-oriented systems development.
- Use-case driven systems development.
- Prototyping.
- Component-based development.
- Rapid application development.

3.1 INTRODUCTION

The essence of the ***software development process*** that consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfies those needs. However, some people view the software development process as interesting but feel it has little importance in developing software. It is tempting to ignore the process and plunge into the implementation and programming phases of software development, much like the builder who would bypass the architect. Some programmers have been able to ignore the counsel of systems development in building a system; but, in general, the dynamics of software development provide little room for such shortcuts, and bypasses have been less than successful. Furthermore, the object-oriented approach requires a more rigorous process to do things right. This way, you need not see code until after about 25 percent of the development time, because you need to spend more time gathering requirements, developing a requirements model and an analysis

model, then turning them into the design model. Now, you can develop code quickly—you have a recipe for doing it. However, you should construct a prototype of some of the key system components shortly after the products are selected, to understand how easy or difficult it will be to implement some of the features of the system. The prototype also can give users a chance to comment on the usability and usefulness of the design and let you assess the fit between the software tools selected, the functional specification, and the users' needs.

This chapter introduces you to the systems development life cycle in general and, more specifically, to an object-oriented approach to software development. The main point of this chapter is the idea of building software by placing emphasis on the analysis and design aspects of the software life cycle. The emphasis is intended to promote the building of high-quality software (meeting specifications and being adaptable for change). The software industry previously suffered from the lack of focus on the early stages of the life cycle [5].

3.2 THE SOFTWARE DEVELOPMENT PROCESS

System development can be viewed as a process. Furthermore, the development itself, in essence, is a process of change, refinement, transformation, or addition to the existing product. Within the process, it is possible to replace one subprocess with a new one, as long as the new subprocess has the same interface as the old one, to allow it to fit into the process as a whole. With this method of change, it is possible to adapt the new process. For example, the object-oriented approach provides us a set of rules for describing inheritance and specialization in a consistent way when a subprocess changes the behavior of its parent process.

The process can be divided into small, interacting phases—subprocesses. The subprocesses must be defined in such a way that they are clearly spelled out, to allow each activity to be performed as independently of other subprocesses as possible. Each subprocess must have the following [1]:

- A description in terms of how it works
- Specification of the input required for the process
- Specification of the output to be produced

The software development process also can be divided into smaller, interacting subprocesses. Generally, the software development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation (see Figure 3–1):

- *Transformation 1 (analysis)* translates the users' needs into system requirements and responsibilities. The way they use the system can provide insight into the users' requirements. For example, one use of the system might be analyzing an incentive payroll system, which will tell us that this capacity must be included in the system requirements.
- *Transformation 2 (design)* begins with a problem statement and ends with a detailed design that can be transformed into an operational system. This transformation includes the bulk of the software development activity, including the

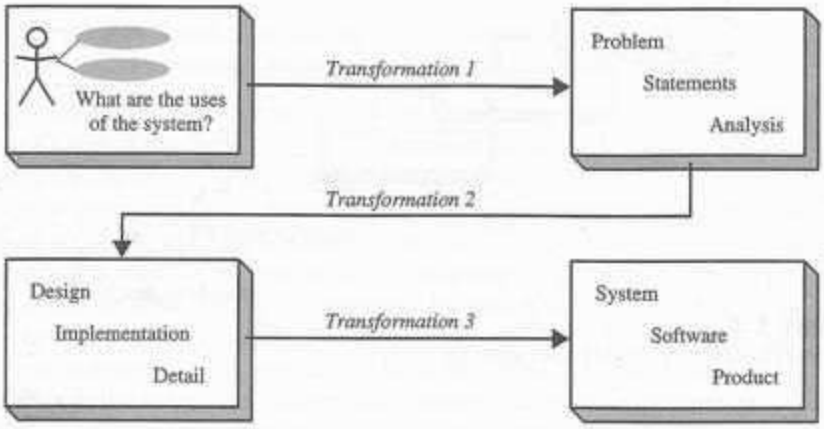


FIGURE 3–1
Software process reflecting transformation from needs to a software product that satisfies those needs.

definition of how to build the software, its development, and its testing. It also includes the design descriptions, the program, and the testing materials.

- *Transformation 3 (implementation)* refines the detailed design into the system deployment that will satisfy the users' needs. This takes into account the equipment, procedures, people, and the like. It represents embedding the software product within its operational environment. For example, the new compensation method is programmed, new forms are put to use, and new reports now can be printed. Here, we try to answer the following question: What procedures and resources are needed to compensate the employees under the new accounting system?

An example of the software development process is the ***waterfall approach***,[1] which starts with deciding *what* is to be done (what is the problem). Once the requirements have been determined, we next must decide *how* to accomplish them. This is followed by a step in which we *do it*, whatever "it" has required us to do. We then must *test* the result to see if we have satisfied the users' requirements. Finally, we *use* what we have done (see Figure 3–2).

In the real world, the problems are not always well-defined and that is why the waterfall model has limited utility. For example, if a company has experience in building accounting systems, then building another such product based on the existing design is best managed with the waterfall model, as it has been described. Where there is uncertainty regarding what is required or how it can be built, the waterfall model fails. This model assumes that the requirements are known before the design begins, but one may need experience with the product before the re-

[1]Many in the software development community feel that the waterfall model has been discarded, whereas others believe it offers a high-level representation of the software process.

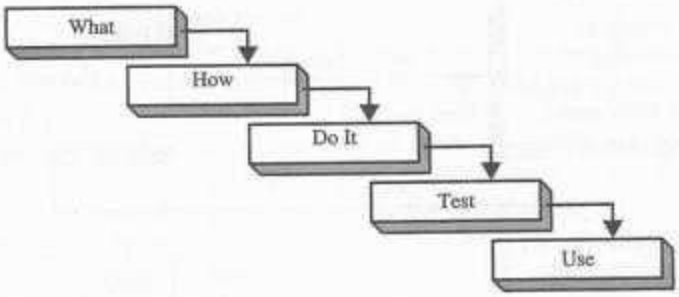


FIGURE 3–2
The waterfall software development process.

quirements can be fully understood. It also assumes that the requirements will remain static over the development cycle and that a product delivered months after it was specified will meet the delivery-time needs.

Finally, even when there is a clear specification, it assumes that sufficient design knowledge will be available to build the product. The waterfall model is the best way to manage a project with a well-understood product, especially very large projects. Clearly, it is based on well-established engineering principles. However, its failures can be traced to its inability to accommodate software's special properties and its inappropriateness for resolving partially understood issues; furthermore, it neither emphasizes nor encourages software reusability.

After the system is installed in the real world, the environment frequently changes, altering the accuracy of the original problem statement and, consequently, generating revised software requirements. This can complicate the software development process even more. For example, a new class of employees or another shift of workers may be added or the standard workweek or the piece rate changed. By definition, any such changes also change the environment, requiring changes in the programs. As each such request is processed, system and programming changes make the process increasingly complex, since each request must be considered in regard to the original statement of needs as modified by other requests.

3.3 BUILDING HIGH-QUALITY SOFTWARE

The software process transforms the users' needs via the application domain to a software solution that satisfies those needs. Once the system (programs) exists, we must test it to see if it is free of bugs. High-quality products must meet users' needs and expectations. Furthermore, the products should attain this with minimal or no defects, the focus being on improving products (or services) prior to delivery rather than correcting them after delivery. The ultimate goal of building high-quality software is user satisfaction. To achieve high quality in software we need to be able to answer the following questions:

- How do we determine when the system is ready for delivery?
- Is it now an operational system that satisfies users' needs?

- Is it correct and operating as we thought it should?
- Does it pass an evaluation process?

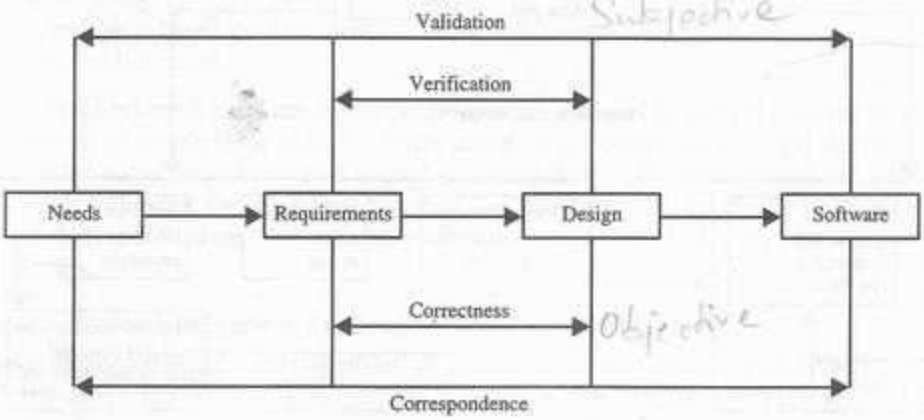
There are two basic approaches to systems testing. We can test a system according to how it has been built or, alternatively, what it should do. Blum [3] describes a means of system evaluation in terms of four quality measures: correspondence, correctness, verification, and validation. ***Correspondence*** measures how well the delivered system matches the needs of the operational environment, as described in the original requirements statement. ***Validation*** is the task of predicting correspondence. True correspondence cannot be determined until the system is in place (see Figure 3–3). ***Correctness*** measures the consistency of the product requirements with respect to the design specification. Blum argues that ***verification*** is the exercise of determining correctness. However, correctness always is objective. Given a specification and a product, it should be possible to determine if the product precisely satisfies the requirements of the specification. For example, does the payroll system accurately compute the amount of compensation? Does it report productivity accurately and to the satisfaction of the workers, and does it handle information as originally planned?

Validation, however, is always subjective, and it addresses a different issue—the appropriateness of the specification. This may be considered 20/20 hindsight: Did we uncover the true users' needs and therefore establish the proper design? If the evaluation criteria could be detailed, they would have been included in the specification. Boehm [4] observes that these quality measures, verification and validation, answer the following questions:

- *Verification.* Am I building the product right?
- *Validation.* Am I building the right product?

Validation begins as soon as the project starts, but verification can begin only after a specification has been accepted. Verification and validation are independent

FIGURE 3–3
Four quality measures (correspondence, correctness, validation, and verification) for software evaluation.



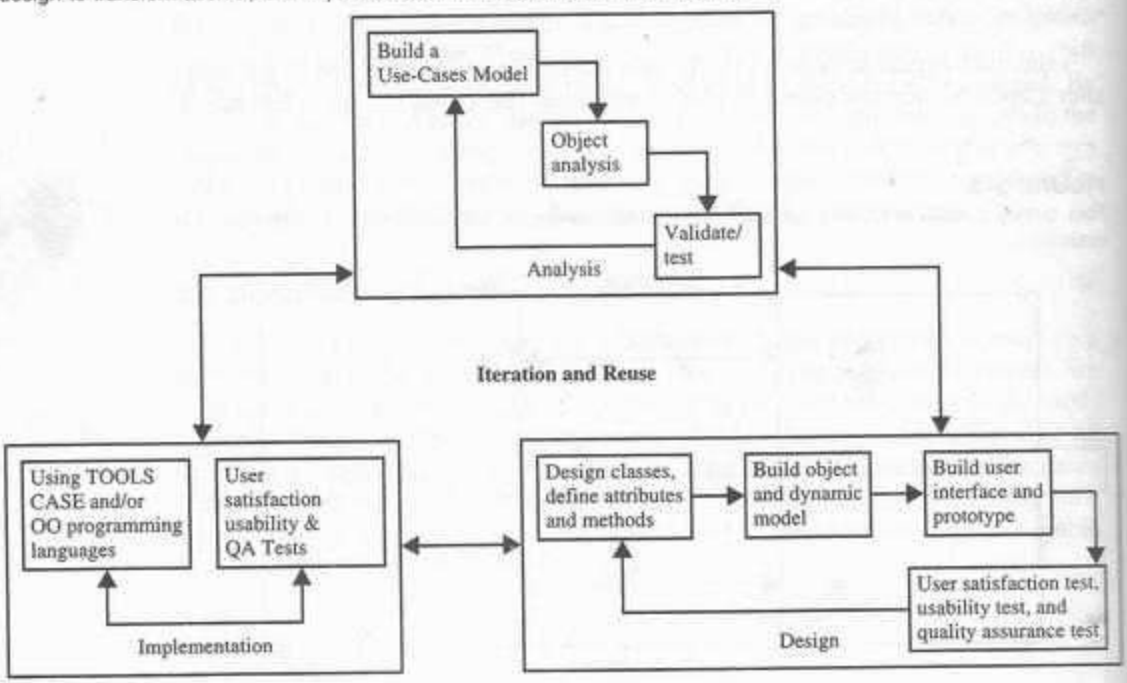
of each other. It is possible to have a product that corresponds to the specification, but if the specification proves to be incorrect, we do not have the right product; for example, say a necessary report is missing from the delivered product, since it was not included in the original specification. A product also may be correct but not correspond to the users' needs; for example, after years of waiting, a system is delivered that satisfies the initial design statement but no longer reflects current operating practices. Blum argues that, when the specification is informal, it is difficult to separate verification from validation. Chapter 13 looks at the issue of software validation and correspondence by proposing a way to measure user satisfaction and software usability. The next section looks at an object-oriented software development approach that eliminates many of the shortcomings of traditional software development, such as the waterfall approach.

3.4 OBJECT-ORIENTED SYSTEMS DEVELOPMENT: A USE-CASE DRIVEN APPROACH

The object-oriented ***software development life cycle*** (SDLC) consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation (see Figure 3–4).

The use-case model can be employed throughout most activities of software development. Furthermore, by following the life cycle model of Jacobson, Ericsson,

FIGURE 3–4
The object-oriented systems development approach. Object-oriented analysis corresponds to transformation 1; design to transformation 2, and implementation to transformation 3 of Figure 3–1.



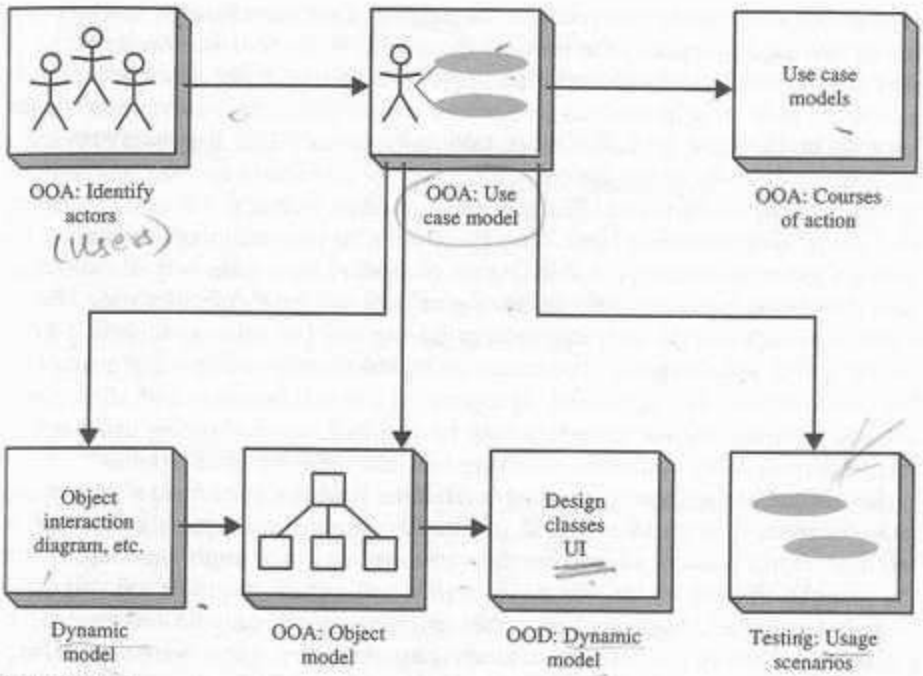


FIGURE 3–5
By following the life cycle model of Jacobson et al., we produce designs that are traceable across requirements, analysis, implementation, and testing.

and Jacobson [11], one can produce designs that are traceable across requirements, analysis, design, implementation, and testing (as shown in Figure 3–5). The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can become test scenarios.

Object-oriented system development includes these activities:

- Object-oriented analysis—use case driven
- Object-oriented design
- Prototyping
- Component-based development
- Incremental testing

Object-oriented software development encourages you to view the problem as a system of cooperative objects. Furthermore, it advocates incremental development. Although object-oriented software development skills come only with practice, by following the guidelines listed in this book you will be on the right track for building sound applications. We look at these activities in detail in subsequent chapters.

3.4.1 Object-Oriented Analysis—Use-Case Driven

The object-oriented analysis phase of software development is concerned with determining the system requirements and identifying classes and their relation-

ship to other classes in the problem domain. To understand the system requirements, we need to identify the users or the actors. Who are the actors and how do they use the system? In object-oriented as well as traditional development, scenarios are used to help analysts understand requirements. However, these scenarios may be treated informally or not fully documented. Ivar Jacobson [10] came up with the concept of the ***use case***, his name for a scenario to describe the user–computer system interaction. The concept worked so well that it became a primary element in system development. The object-oriented programming community has adopted use cases to a remarkable degree. Scenarios are a great way of examining who does what in the interactions among objects and what ***role*** they play; that is, their interrelationships. This intersection among objects' roles to achieve a given goal is called ***collaboration***. The scenarios represent only one possible example of the collaboration. To understand all aspects of the collaboration and all potential actions, several different scenarios may be required, some showing usual behaviors, others showing situations involving unusual behavior or exceptions.

In essence, a use case is a typical interaction between a user and a system that captures users' goals and needs. In its simplest usage, you capture a use case by talking to typical users, discussing the various things they might want to do with the system.

Expressing these high-level processes and interactions with customers in a scenario and analyzing it is referred to as *use-case modeling*. The use-case model represents the users' view of the system or users' needs. For example, consider a word processor, where a user may want to be able to replace a word with its synonym or create a hyperlink. These are some uses of the system, or a system responsibility.

This process of developing uses cases, like other object-oriented activities, is iterative—once your use-case model is better understood and developed you should start to identify classes and create their relationships.

Looking at the physical objects in the system also provides us important information on objects in the systems. The objects could be individuals, organizations, machines, units of information, pictures, or whatever else makes up the application and makes sense in the context of the real-world system. While developing the model, objects emerge that help us establish a workable system. It is necessary to work iteratively between use-case and object models. For example, the objects in the incentive payroll system might include the following examples:

The employee, worker, supervisor, office administrator.

The paycheck.

The product being made.

The process used to make the product.

Of course, some problems have no basis in the real world. In this case, it can be useful to pose the problem in terms of analogous physical objects, kind of a mental simulation. It always is possible to think of a problem in terms of some kinds of objects, although in some cases, the objects may be synthetic or esoteric, with no direct physical counterparts. The objects need to have meaning only within

the context of the application's domain. For example, the application domain might be a payroll system; and the tangible objects might be the paycheck, employee, worker, supervisor, office administrator; and the intangible objects might be tables, data entry screen, data structures, and so forth.

Documentation is another important activity, which does not end with object-oriented analysis but should be carried out throughout the system development. However, make the documentation as short as possible. The 80–20 rule generally applies for documentation: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know. Remember that documentation and modeling are not separate activities, and good modeling implies good documentation.

3.4.2 Object-Oriented Design

The goal of ***object-oriented design*** (OOD) is to design the classes identified during the analysis phase and the user interface. During this phase, we identify and define additional objects and classes that support implementation of the requirements [8]. For example, during the design phase, you might need to add objects for the user interface to the system (e.g., data entry windows, browse windows).

Object-oriented design and object-oriented analysis are distinct disciplines, but they can be intertwined. Object-oriented development is highly incremental; in other words, you start with object-oriented analysis, model it, create an object-oriented design, then do some more of each, again and again, gradually refining and completing models of the system. The activities and focus of object-oriented analysis and object-oriented design are intertwined—grown, not built (see Figure 3–4).

First, build the object model based on objects and their relationships, then iterate and refine the model:

- Design and refine classes.
- Design and refine attributes.
- Design and refine methods.
- Design and refine structures.
- Design and refine associations.

Here are a few guidelines to use in your object-oriented design:

- Reuse, rather than build, a new class. Know the existing classes.
- Design a large number of simple classes, rather than a small number of complex classes.
- Design methods.
- Critique what you have proposed. If possible, go back and refine the classes.

3.4.3 Prototyping

Although the object-oriented analysis and design describe the system features, it is important to construct a prototype of some of the key system components shortly

after the products are selected. It has been said "a picture may be worth a thousand words, but a prototype is worth a thousand pictures"[author unknown]. Not only is this true, it is an understatement of the value of software prototyping. Essentially, a prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. A prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system. It also can give users a chance to comment on the usability and usefulness of the user interface design and lets you assess the fit between the software tools selected, the functional specification, and the user needs. Additionally, prototyping can further define the use cases, and it actually makes use-case modeling much easier. Building a prototype that the users are happy with, along with documentation of what you did, can define the basic courses of action for those use cases covered by the prototype. The main idea here is to build a prototype with uses-case modeling to design systems that users like and need.

Traditionally, prototyping was used as a "quick and dirty" way to test the design, user interface, and so forth, something to be thrown away when the "industrial strength" version was developed. However, the new trend, such as using rapid application development, is to refine the prototype into the final product. Prototyping provides the developer a means to test and refine the user interface and increase the usability of the system. As the underlying prototype design begins to become more consistent with the application requirements, more details can be added to the application, again with further testing, evaluation, and rebuilding, until all the application components work properly within the prototype framework.

Prototypes have been categorized in various ways. The following categories are some of the commonly accepted prototypes and represent very distinct ways of viewing a prototype, each having its own strengths:

- A ***horizontal prototype*** is a simulation of the interface (that is, it has the entire user interface that will be in the full-featured system) but contains no functionality. This has the advantages of being very quick to implement, providing a good overall feel of the system, and allowing users to evaluate the interface on the basis of their normal, expected perception of the system.
- A ***vertical prototype*** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth. In practice, prototypes are a hybrid between horizontal and vertical: The major portions of the interface are established so the user can get the feel of the system, and features having a high degree of risk are prototyped with much more functionality [7].
- An ***analysis prototype*** is an aid for exploring the problem domain. This class of prototype is used to inform the user and demonstrate the proof of a concept. It is not used as the basis of development, however, and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code.
- A ***domain prototype*** is an aid for the incremental development of the ultimate

software solution. It often is used as a tool for the staged delivery of subsystems to the users or other members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product [9].

The typical time required to produce a prototype is anywhere from a few days to several weeks, depending on the type and function of prototype. Prototyping should involve representation from all user groups that will be affected by the project, especially the end users and management members to ascertain that the general structure of the prototype meets the requirements established for the overall design. The purpose of this review is threefold:

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.
2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.
3. To give management and everyone connected with the project the first (or it could be second or third . . .) glimpse of what the technology can provide.

The evaluation can be performed easily if the necessary supporting data is readily available. Testing considerations must be incorporated into the design and subsequent implementation of the system.

Prototyping is a useful exercise at almost any stage of the development. In fact, prototyping should be done in parallel with the preparation of the functional specification. As key features are specified, prototyping those features usually results in modifications to the specification and even can reveal additional features or problems that were not obvious until the prototype was built.

3.4.4 Implementation: Component-Based Development

Manufacturers long ago learned the benefits of moving from custom development to assembly from prefabricated components. Component-based manufacturing makes many products available to the marketplace that otherwise would be prohibitively expensive. If products, from automobiles to plumbing fittings to PCs, were custom-designed and built for each customer, the way business applications are, then large markets for these products would not exist. Low-cost, high-quality products would not be available. Modern manufacturing has evolved to exploit two crucial factors underlying today's market requirements: reduce cost and time to market by building from prebuilt, ready-tested components, but add value and differentiation by rapid customization to targeted customers [13].

Today, software components are built and tested in-house, using a wide range of technologies. For example, computer-aided software engineering (CASE) tools allow their users to rapidly develop information systems. The main goal of CASE technology is the automation of the entire information system's development life cycle process using a set of integrated software tools, such as modeling, methodology, and automatic code generation. However, most often, the code generated by

CASE tools is only the skeleton of an application and a lot needs to be filled in by programming by hand. A new generation of CASE tools is beginning to support component-based development.

Component-based development (CBD) is an industrialized approach to the software development process. Application development moves from custom development to assembly of prebuilt, pretested, reusable software components that operate with each other. Two basic ideas underlie component-based development. First, the application development can be improved significantly if applications can be assembled quickly from prefabricated software components. Second, an increasingly large collection of interpretable software components could be made available to developers in both general and specialist catalogs. Put together, these two ideas move application development from a craft activity to an industrial process fit to meet the needs of modern, highly dynamic, competitive, global businesses. The industrialization of application development is akin to similar transformations that occurred in other human endeavors.

A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of prebuilt components or old-fashioned code written in a language such as C, assembler, or COBOL. Visual tools or actual code can be used to "glue" together components. Although it is practical to do simple applications using only "visual glue" (e.g., by "wiring" components together as in Digitalk's Smalltalk PARTS, or IBM's VisualAge), putting together a practical application still poses some challenges. Of course, all these are "under the hood" and should be invisible to end users. The impact to users will come from faster product development cycles, increased flexibility, and improved customization features. CBD will allow independently developed applications to work together and do so more efficiently and with less development effort [13].

Existing (legacy) applications support critical services within an organization and therefore cannot be thrown away. Massive rewriting from scratch is not a viable option, as most legacy applications are complex, massive, and often poorly documented. The CBD approach to legacy integration involves application wrapping, in particular component wrapping, technology. An application wrapper surrounds a complete system, both code and data. This wrapper then provides an interface that can interact with both the legacy and the new software systems (see Figure 3–6). Off-the-shelf application wrappers are not widely available. At present, most application wrappers are homegrown within organizations. However, with component-based development technology emerging rapidly, component wrapper technology will be used more widely.

The ***software components*** are the functional units of a program, building blocks offering a collection of reusable services. A software component can request a service from another component or deliver its own services on request. The delivery of services is independent, which means that components work together to accomplish a task. Of course, components may depend on one another without interfering with each other. Each component is unaware of the context or inner workings of the other components. In short, the object-oriented concept addresses analysis, design, and programming, whereas component-based development is

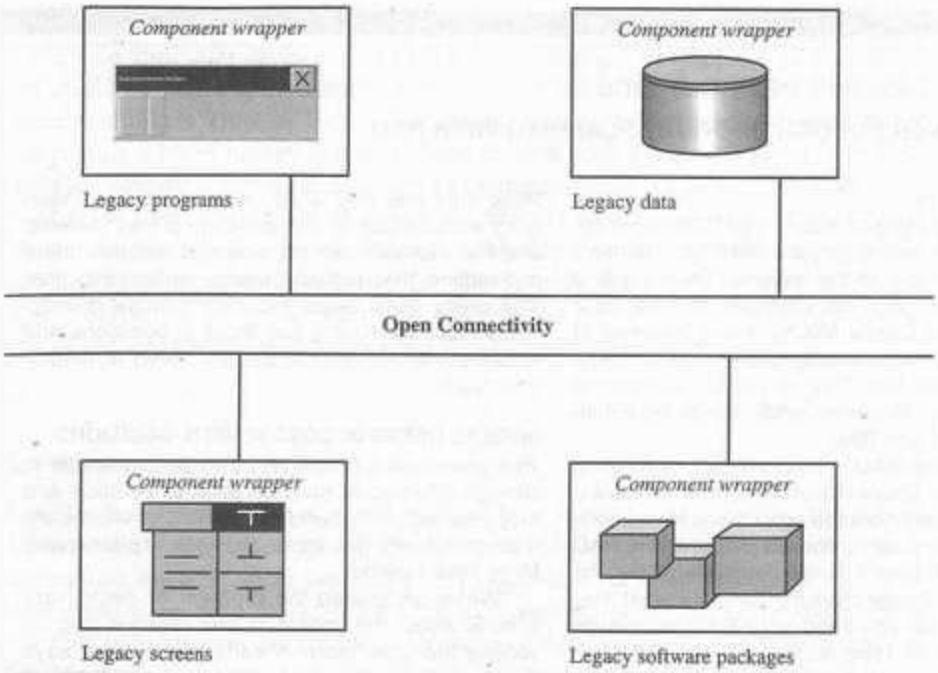


FIGURE 3-6
Reusing legacy system via component wrapping technology.

concerned with the implementation and system integration aspects of software development.

Rapid application development (RAD) is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods. The term often is used in conjuction with software prototyping. It is widely held that, to achieve RAD, the developer sacrifices the quality of the product for a quicker delivery. This is not necessarily the case. RAD is concerned primarily with reducing the "time to market," not exclusively the software development time. In fact, one successful RAD application achieved a substantial reduction in time to market but realized no significant reduction in the individual software cycles [12].

RAD does not replace the system development life cycle (see the Real-World case) but complements it, since it focuses more on process description and can be combined perfectly with the object-oriented approach. The task of RAD is to build the application quickly and incrementally implement the design and user requirements, through tools such as Delphi, VisualAge, Visual Basic, or PowerBuilder.

After the overall design for an application has been completed, RAD begins. The main objective of RAD is to build a version of an application rapidly to see whether we actually have understood the problem (analysis). Further, it determines whether the system does what it is supposed to do (design). RAD involves a number of iterations. Through each iteration we might understand the problem a little

BOX 3.1

Real-World Issues on the Agenda

THERE'S NEVER ENOUGH UP-FRONT PLANNING WITH RAD

By Clair Tristram

What's the single largest reason that RAD [rapid application development] projects fail? Poor up-front planning, according to the experts. "Planning is a bad word these days, but I happen to think it's a good idea," says Carma McClure, vice president of research at Extended Intelligence Inc., a Chicago-based consulting firm. "You've got to have control over the process." Runaway requirements are a dangerous problem with RAD.

If you choose RAD methodologies to develop your application, you're vulnerable. You won't have a hefty set of requirements to protect you from users. Instead, the same users who are critical to the RAD equation are the very same people who can be counted on to change their minds about what they want. So how do you keep your RAD project on track and on time? Here are some suggestions.

WRITE THINGS DOWN

Sure, you've gotten rid of the onerous task of creating a hefty requirement document by choosing RAD over the waterfall approach. But don't make the mistake of neglecting to write down your business objective at the beginning of the project, and make sure your clients agree on what the core requirements should be.

"A lot of people get stuck in what we call 'proto cycling'," says Richard Hunter, research director at Gartner Group Inc., in Stamford, CT. "They don't know what the business problem is that they're trying to solve, and in that case it can take a long time to find out what you're doing."

AVOID "SHALLOW" PROTOTYPING

RAD tools make great demos, but can you deliver? Make sure that your team understands the underlying architecture of the prototypes they develop and that they can develop prototype features under a deadline that actually works, rather than just look pretty. "RAD helps you build a model quickly," notes McClure. "Users can make suggestions and virtually see the results. But you need to control your team."

INVOLVE USERS IN COST-BENEFIT DECISIONS

Your users see a prototype interface that seems to change effortlessly from iteration to iteration and they may not understand the amount of effort it will take to actually get those changes implemented. Make sure they do.

"We've eliminated the problem by being very specific about the impact of any changes and involving the user team in setting priorities," says Rick Irving, director of worldwide sales systems at American Express Stored Value Group, in Salt Lake City.

DON'T DEVELOP APPLICATIONS IN ISOLATION

"RAD makes it easy to come up quickly with something good for a single group, but that doesn't satisfy the needs of additional groups," McClure says. "Will you throw it away and start over?"

To avoid clusters of applications with limited utility, McClure recommends honing an understanding of how your RAD project fits into your company's strategic system plan before you begin. That, and always build with reuse in mind.

Source: Clair Tristram, "There's never enough up-front planning with RAD," *PC Week* 13, no. 12 (March 25, 1995).

better and make an improvement. RAD encourages the incremental development approach of "grow, do not build" software.

Prototyping and RAD do not replace the object-oriented software development model. Instead, in a RAD application, you go through those stages in rapid (or incomplete) fashion, completing a little more in the next iteration of the prototype. One thing that you should remember is that RAD tools make great demos. However, make sure that you can develop prototype features within a deadline that actually works, rather than just looks good.

3.4.5 Incremental Testing

If you wait until after development to test an application for bugs and performance, you could be wasting thousands of dollars and hours of time. That's what happened at Bankers Trust in 1992: "Our testing was very complete and good, but it was costing a lot of money and would add months onto a project," says Glenn Shimamoto, vice president of technology and strategic planning at the New York bank [6]. In one case, testing added nearly six months to the development of a funds transfer application. The problem was that developers would turn over applications to a quality assurance (QA) group for testing only after development was completed. Since the QA group wasn't included in the initial plan, it had no clear picture of the system characteristics until it came time to test.

3.5 REUSABILITY

A major benefit of object-oriented system development is reusability, and this is the most difficult promise to deliver on. For an object to be really reusable, much more effort must be spent designing it. To deliver a reusable object, the development team must have the up-front time to design reusability into the object. The potential benefits of reuse are clear: increased reliability, reduced time and cost for development, and improved consistency. You must effectively evaluate existing software components for reuse by asking the following questions as they apply to the intended applications [2]:

- Has my problem already been solved?
- Has my problem been partially solved?
- What has been done before to solve a problem similar to this one?

To answer these questions, we need detailed summary information about existing software components. In addition to the availability of the information, we need some kind of search mechanism that allows us to define the candidate object simply and then generate broadly or narrowly defined queries. Thus, the ideal system for reuse would function like a skilled reference librarian. If you have a question about a subject area, all potential sources could be identified and the subject area could be narrowed by prompting. Some form of browsing with the capability to provide detailed information would be required, one where specific subjects could be looked up directly.

The reuse strategy can be based on the following:

- Information hiding (encapsulation).
- Conformance to naming standards.
- Creation and administration of an object repository.
- Encouragement by strategic management of reuse as opposed to constant redevelopment.
- Establishing targets for a percentage of the objects in the project to be reused (i.e., 50 percent reuse of objects).

3.6 SUMMARY

This chapter introduces the system development life cycle (SDLC) in general and object-oriented and use-case driven SDLC specifically. The essence of the software process is the transformation of users' needs through the application domain into a software solution that is executed in the implementation domain. The concept of the use case, or a set of scenarios, can be a valuable tool for understanding the users' needs. The emphasis on the analysis and design aspects of the software life cycle is intended to promote building high-quality software (meeting the specifications and being adaptable for change).

High-quality software provides users with an application that meets their needs and expectations. Four quality measures have been described: correspondence, correctness, verification, and validation. Correspondence measures how well the delivered system corresponds to the needs of the problem. Correctness determines whether or not the system correctly computes the results based on the rules created during the system analysis and design, measuring the consistency of product requirements with respect to the design specification. Verification is the task of determining correctness (am I building the *product right?*). Validation is the task of predicting correspondence (am I building the *right product?*).

Object-oriented design requires more rigor up front to do things right. You need to spend more time gathering requirements, developing a requirements model and an analysis model, then turning them into the design model. Now, you can develop code quickly—you have a recipe for doing it. Object-oriented systems development consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation. Object-oriented analysis requires building a use-case model and interaction diagrams to identify users' needs and the system's classes and their responsibility, then validating and testing the model, documenting each step along the way. Object-oriented design centers on establishing design classes and their protocol; building class diagrams, user interfaces, and prototypes; testing user satisfaction and usability based on usage and use cases. The use-case concept can be employed through most of the activities of software development. Furthermore, by following Jacobson's life cycle model, one can produce designs that are traceable across requirements, analysis, design, implementation, and testing.

Component-based development (CBD) is an industrialized approach to software development. Software components are functional units, or building blocks offering a collection of reusable services. A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of prebuilt components or old-fashioned code written in a language such as C, assembler, or COBOL. The object-oriented concept addresses analysis, design, and programming; whereas component-based development is concerned with the implementation and system integration aspects of software development.

The rapid application development (RAD) approach to systems development rapidly develops software to quickly and incrementally implement the design by using tools such as CASE.

Reusability is a major benefit of object-oriented system development. It is also the most difficult promise to deliver. To develop reusable objects, you must spend time up front to design reusability in the objects.

KEY TERMS

Analysis prototype (p. 48)
Collaboration (p. 46)
Component-based development (CBD) (p. 50)
Correctness (p. 43)
Correspondence (p. 43)
Domain prototype (p. 48)
Horizontal prototype (p. 48)
Object-oriented analysis (p. 45)
Object-oriented design (OOD) (p. 47)
Rapid application development (RAD) (p. 51)
Role (p. 46)
Software components (p. 50)
Software development life cycle (SDLC) (p. 44)
Software development process (p. 39)
Use case (p. 46)
Validation (p. 43)
Vertical prototype (p. 48)
Verification (p. 43)
Waterfall approach (p. 41)

REVIEW QUESTIONS

1. What is the waterfall SDLC?
2. What are some of the advantages and disadvantages of the waterfall process?
3. What is the software development process?
4. What is software correspondence?
5. What is software correctness?
6. What is software validation?
7. What is software verification?
8. How is software verification different from validation?
9. What is prototyping and why is it useful?
10. What are some of the advantages and disadvantages of prototyping?
11. If you have to choose between prototyping and the waterfall approach, which one would you select and why?
12. Describe the macro processes of the object-oriented system development approach.
13. What is the object-oriented SDLC? Compare it with traditional approaches.
14. What are some of the object-oriented analysis processes?
15. What are the object-oriented design processes?
16. What is use-case modeling?
17. What is object modeling?

18. Why can users get involved more easily in prototyping than the traditional software development process such as the waterfall approach?
19. What is RAD?
20. Why is CBD important?
21. Why is reusability important? How does object-oriented software development promote reusability?

PROBLEMS

1. Take a look at the Web site of Popkin software (www.popkin.com). Find out if SA (System Architect) has repository of objects. Write a report on your findings.
2. What are some of the classes of your university system? (Do not worry if you are not sure how to identify objects yet, it will be covered in later chapters. For example, a possible class might be Professor. Think about other objects based on your own personal experience.)
3. You have been hired as a system analyst for the Matrix Corporation. Your first assignment is to propose a new system for communication among employees. Assuming that you would like to apply the waterfall approach, what would you do at the "what" phase? How would you accomplish it? Should you develop several alternatives or just one, and why? Come up with several alternatives.
4. Most of the Comnet Bank data processing systems were developed in the late 1960s. Although the systems are working properly at this time and they meet management's information needs, an increasing percentage of the systems' development efforts are spent on maintaining existing programs. In addition, in the past several years, the internal audit department has hired two EDP auditors who specialize in auditing computer code in these programs. They have found it almost impossible to follow the code's logic, since most of the coding is not documented, is poorly designed and coded, and has been modified many times (spaghetti code). As a result, they have abandoned direct review of program code as an audit technique. Do you think that Comnet Bank has a problem? If so, what is the nature and cause of the problem? What do you recommend to correct it? Should the bank abandon the old system and start from scratch or should it reuse the legacy system by applying component-based development. Do some research on CBD and write a report on your findings.

REFERENCES

1. Anderson, Michael; and Bergstrand, John. "Formalizing Use Cases with Message Sequence Charts." Masters thesis, Department of Communication Systems at Lund Institute of Technology, 1995.
2. Binder, Robert. "Software Process Improvement: A Case Study." *Software Development* 2, no. 1 (January 1994).
3. Blum, Bruce I. *Software Engineering, a Holistic View*. New York: Oxford University Press, 1992.
4. Boehm, Barry W. "Verifying and Validating Software Requirements and Design Specifications." *Software* (January 1984), pp. 75–88.
5. Booch, Grady. *Object Oriented Design with Applications*. Menlo Park, CA: Benjamin-Cummings, 1991.
6. Callaway, Erin. "Continuous Testing Cures the Last-Minute Crunch." *PC Week* 13, no. 12 (March 25, 1995).

7. Card, D. "The RAD Fad: Is Timing Really Everything?" *IEEE Software Engineering* 12, no. 5 (September 1995), pp. 19–22.
8. Coad, Peter; and Yourdon, Edward. *Object-Oriented Analysis*, 2d ed. Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall, 1991.
9. Dollas, A. "Reducing the Time to Market Through Rapid Prototyping." *IEEE Computer* 28, no. 2 (February 1995), pp. 14–15.
10. Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, Object Technology Series, 1994.
11. Jacobson, Ivar; Ericsson, Maria; and Jacobson, Agneta. *The Object Advantage Business Process Reengineering with Object Technology*. Reading, MA: Addison-Wesley Publishing Company, 1995.
12. Rafii, F.; and Perkins, S. "Internationalizing Software with Concurrent Engineering." *IEEE Software Engineering* 12, no. 5 (September 1995), pp. 39–46.
13. Short, Keith. *Component Based Development and Object Modeling*. Texas Instruments Software, February 1997.