# OBJECT ORIENTED SYSTEMS DEVELOPMENT

**Ali Bahrami**
Boeing Applied Research & Technology

Irwin McGraw-Hill

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid Mexico City
Milan New Delhi Seoul Singapore Sydney Taipei Toronto

# CONTENTS

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



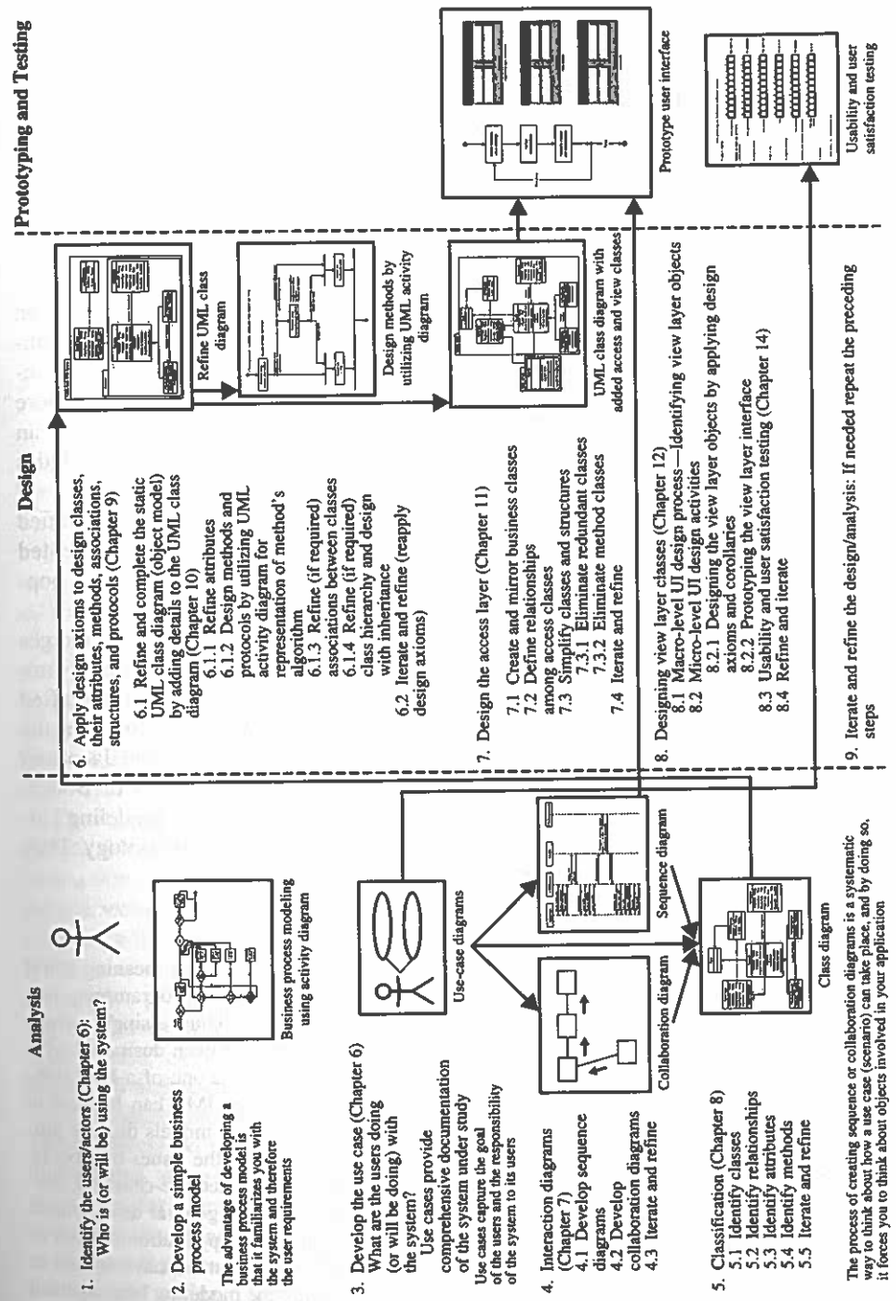



**FIGURE 1–1**
The unified approach road map.

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.



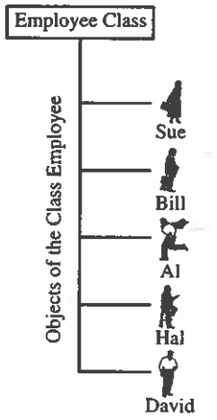


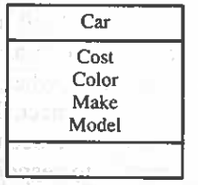

**FIGURE 2–1**
Sue, Bill, Al, Hal, and David are instances or objects of the class Employee.

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 meaningless 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.



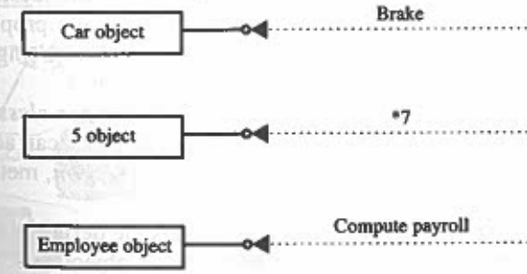
***Messages*** essentially are nonspecific function calls: We would send a *draw* message to a chart when we want the chart to draw itself. A message is different from a subroutine call, since different objects can respond to the same message in different ways. For example, cars, motorcycles, and bicycles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

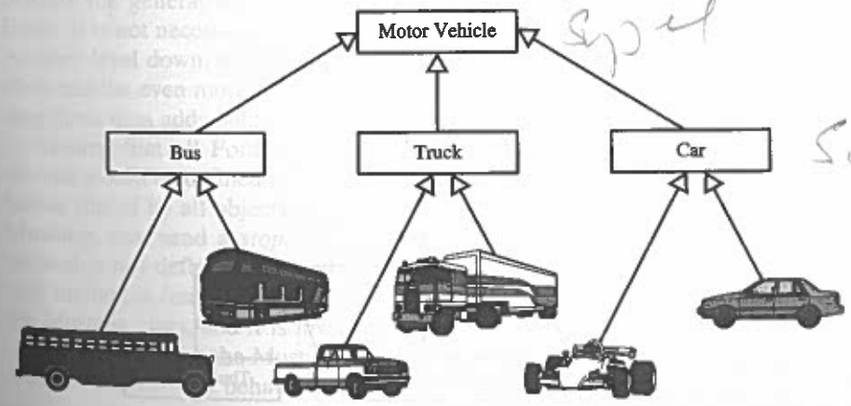
A car engine is an example of encapsulation. Although engines may differ in implementation, the interface between the driver and the car is through a common protocol: Step on the gas to increase power and let up on the gas to decrease power. Since all drivers know this protocol, all drivers can use this 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.



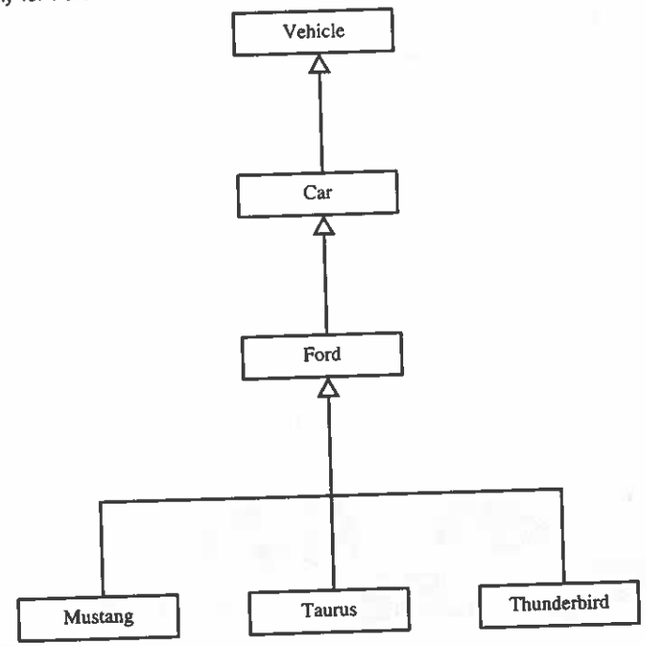
only have one occupant, the driver. In this manner, subclasses modify the attribute (number of passengers) of its superclass, Car.

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-

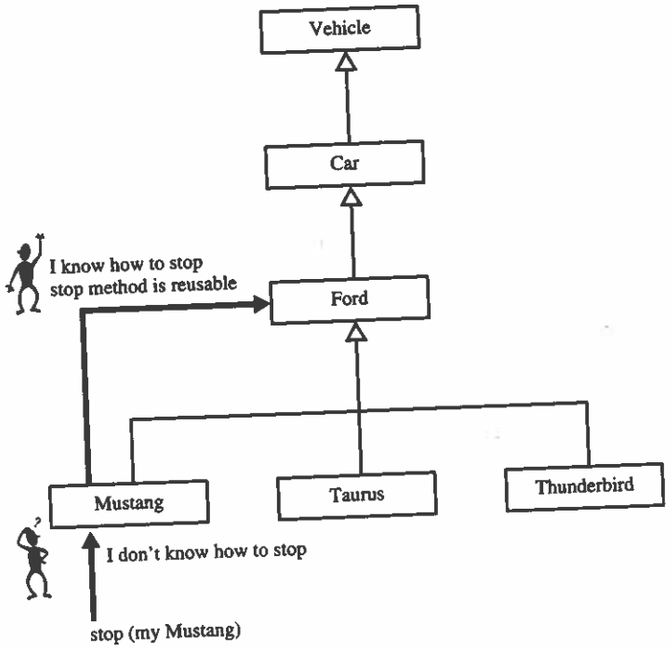



**FIGURE 2–6**
Inheritance allows reusability.

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.



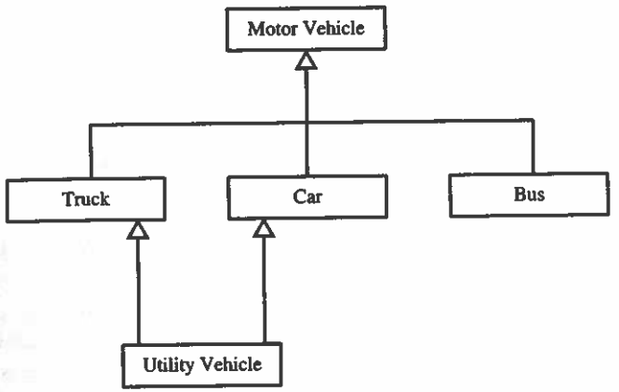



**FIGURE 2–7**
Utility vehicle inherits from both the Car and Truck classes.

### 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.

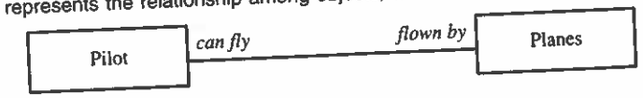
Associations are bidirectional; that means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. For example, *can fly* connects a pilot to certain airplanes. The inverse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





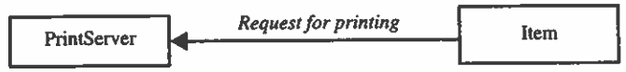



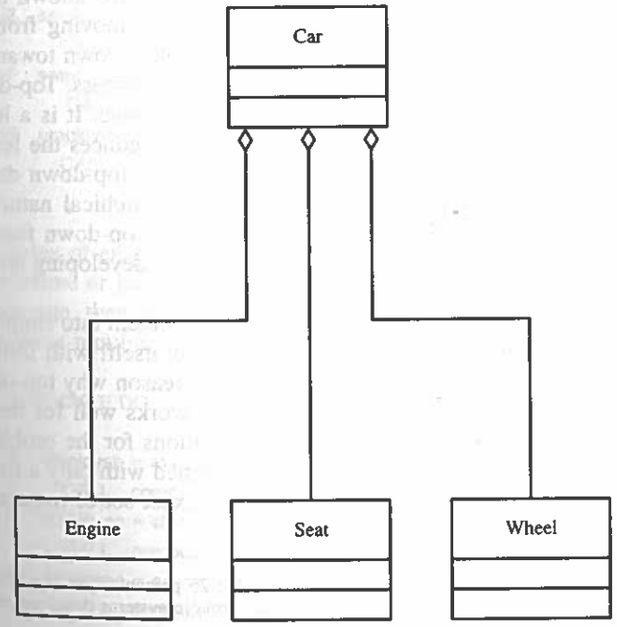

**FIGURE 2–9**
The consumer/producer association.

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.



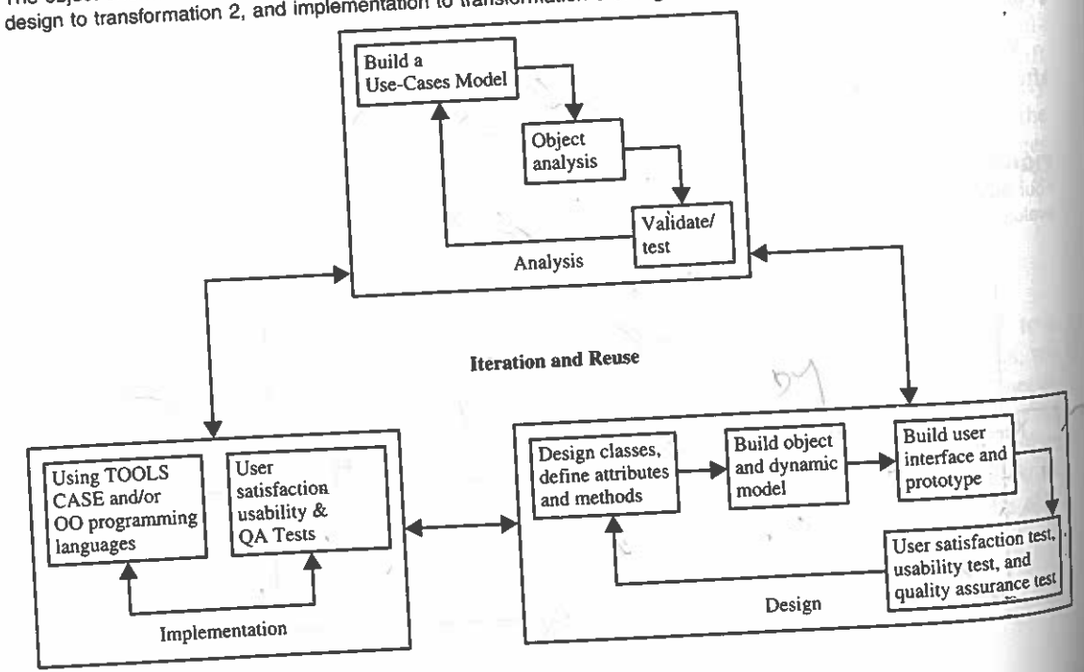
of each other. It is possible to have a product that corresponds to the specification, but if the specification proves to be incorrect, we do not have the right product; for example, say a necessary report is missing from the delivered product, since it was not included in the original specification. 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.





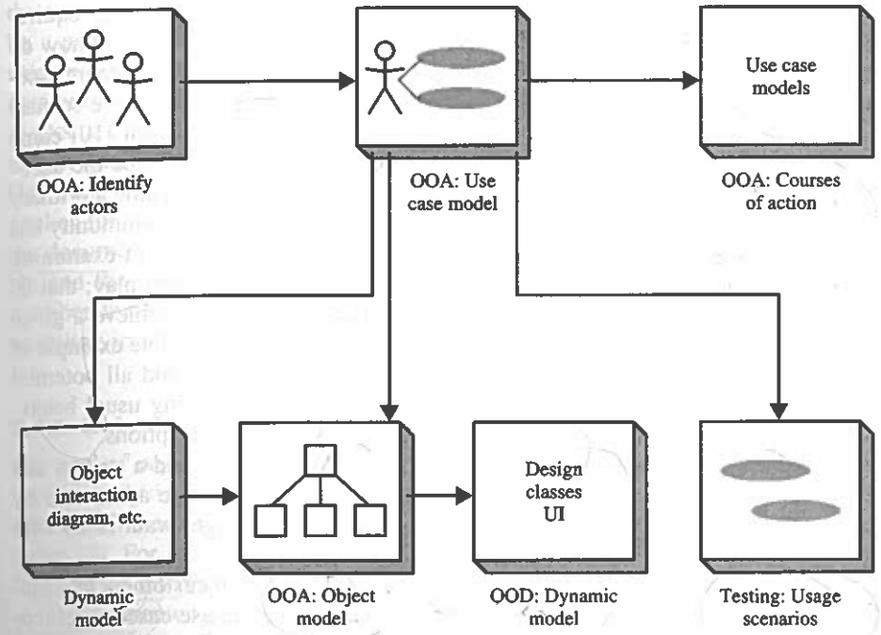




**FIGURE 3–5**
By following the life cycle model of Jacobson et al., we produce designs that are traceable across requirements, analysis, implementation, and testing.

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



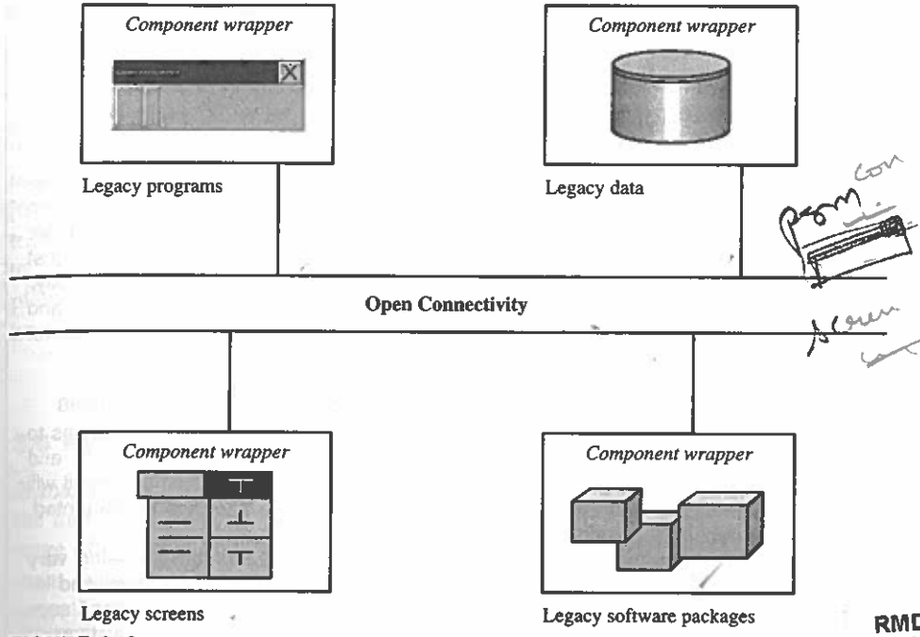



**FIGURE 3–6**
Reusing legacy system via component wrapping technology.



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).

To get a feel for object-oriented methodologies, let us look at some of the methods developed in the 1980s and 1990s. This list by no means is complete [14].

- 1986. Booch [6] developed the object-oriented design concept, the Booch method.
- 1987. Sally Shlaer and Steve Mellor [21] created the concept of the recursive design approach.
- 1989. Beck and Cunningham produced class-responsibility-collaboration cards.
- 1990. Wirfs-Brock, Wilkerson, and Wiener [23] came up with responsibility-driven design.
- 1991. Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT) [19].
- 1991. Peter Coad and Ed Yourdon [11] developed the Coad lightweight and prototype-oriented approach to methods.
- 1994. Ivar Jacobson [16] introduced the concept of the use case and object-oriented software engineering (OOSE).

These methodologies and many other forms of notational language provided system designers and architects many choices but created a very split, competitive, and confusing environment. Most of the methods were very similar but contained a number of often annoying minor differences, and each had a group of practitioners that liked its ideas. The same basic concepts appeared in very different notations, which caused confusion among the users [14].

The trend in object-oriented methodologies, sometimes called *second-generation object-oriented methods*, has been toward combining the best aspects of the most popular methods instead of coming out with new methodologies, which was the tendency in first-generation object-oriented methods. In the next section, to give you a taste of object-oriented methodologies, we will look at some of the most popular ones.

## 4.2 SURVEY OF SOME OF THE OBJECT-ORIENTED METHODOLOGIES

Many methodologies are available to choose from for system development. Each methodology is based on modeling the business problem and implementing the application in an object-oriented fashion; the differences lie primarily in the documentation of information and modeling notations and language. An application can be implemented in many ways to meet the same requirements and provide the same functionality. The largest noticeable differences will be in the trade-offs and detailed design decisions made. Two people using the same methodology may produce application designs that look radically different. This does not necessarily mean that one is right and one is wrong, just that they are different. In the following sections, we look at the methodologies and their modeling notations developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML).

Each method has its strengths. The Rumbaugh et al. method is well-suited for describing the object model or the static structure of the system. The Jacobson et al. method is good for producing user-driven analysis models. The Booch method produces detailed object-oriented design models.



## 4.3 RUMBAUGH ET AL.'S OBJECT MODELING TECHNIQUE

The object modeling technique (OMT) presented by Jim Rumbaugh and his coworkers describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. Details such as class attributes, method, inheritance, and association also can be expressed easily. The dynamic behavior of objects within a system can be described using the OMT dynamic model. This model lets you specify detailed state transitions and their descriptions within a system. Finally, a process description and consumer-producer relationships can be expressed using OMT's functional model. OMT consists of four phases, which can be performed iteratively:

1. *Analysis*. The results are objects and dynamic and functional models.
2. *System design*. The results are a structure of the basic architecture of the system along with high-level strategy decisions.
3. *Object design*. This phase produces a design document, consisting of detailed objects static, dynamic, and functional models.
4. *Implementation*. This activity produces reusable, extendible, and robust code.

OMT separates modeling into three different parts:

1. An *object model*, presented by the object model and the data dictionary.
2. A *dynamic model*, presented by the state diagrams and event flow diagrams.
3. A *functional model*, presented by data flow and constraints.

### 4.3.1 The Object Model

The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations. The object model is represented graphically with an object diagram (see Figure 4–1). The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among the classes. Each association line represents a set of links from the objects of one class to the objects of another class.

### 4.3.2 The OMT Dynamic Model

OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions. The OMT state transition diagram is a network of states and events (see Figure 4–2). Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.

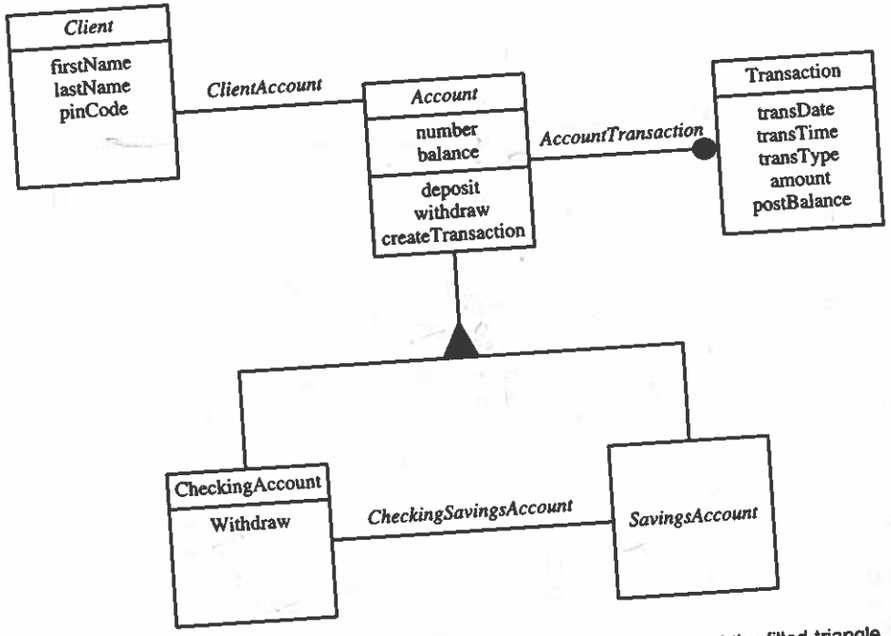



**FIGURE 4–1**
The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one too many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship betwen Client and Account classes is one to one: A client can have only one account and account can belong to only one person (in this model joint accounts are not allowed).

### 4.3.3 The OMT Functional Model

The OMT data flow diagram (DFD) shows the flow of data between different processes in a business. An OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems [3].

Data flow diagrams use four primary symbols:

1. The *process* is any function being performed; for example, verify Password or PIN in the ATM system (see Figure 4–3).
2. The *data flow* shows the direction of data element movement; for example, PIN code.
3. The *data store* is a location where data are stored; for example, account is a data store in the ATM example.
4. An *external entity* is a source or destination of a data element; for example, the ATM card reader.

Overall, the Rumbaugh et al. OMT methodology provides one of the strongest tool sets for the analysis and design of object-oriented systems.



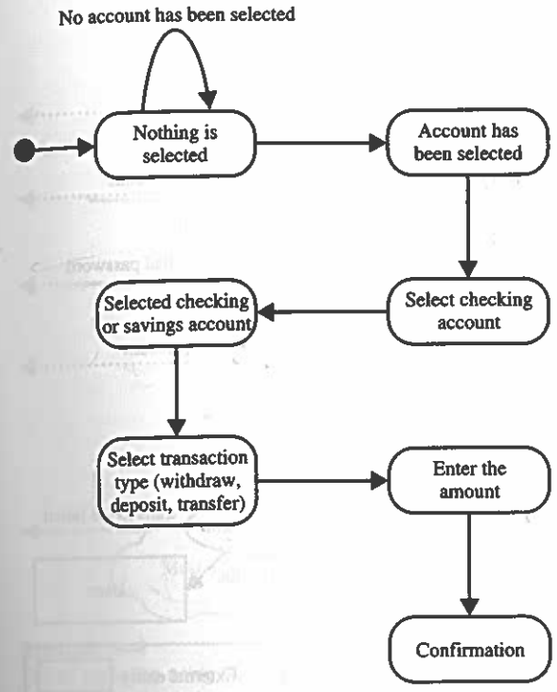



**FIGURE 4–2**
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

## 4.4 THE BOOCH METHODOLOGY

The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm. It covers the analysis and design phases of an object-oriented system. Booch sometimes is criticized for his large set of symbols. Even though Booch defines a lot of symbols to document almost every design decision, if you work with his method, you will notice that you never use all these symbols and diagrams. You start with class and object diagrams (see Figures 4–4 and 4–5) in the analysis phase and refine these diagrams in various steps. Only when you are ready to generate code, do you add design symbols—and this is where the Booch method shines, you can document your object-oriented code. The Booch method consists of the following diagrams:

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams

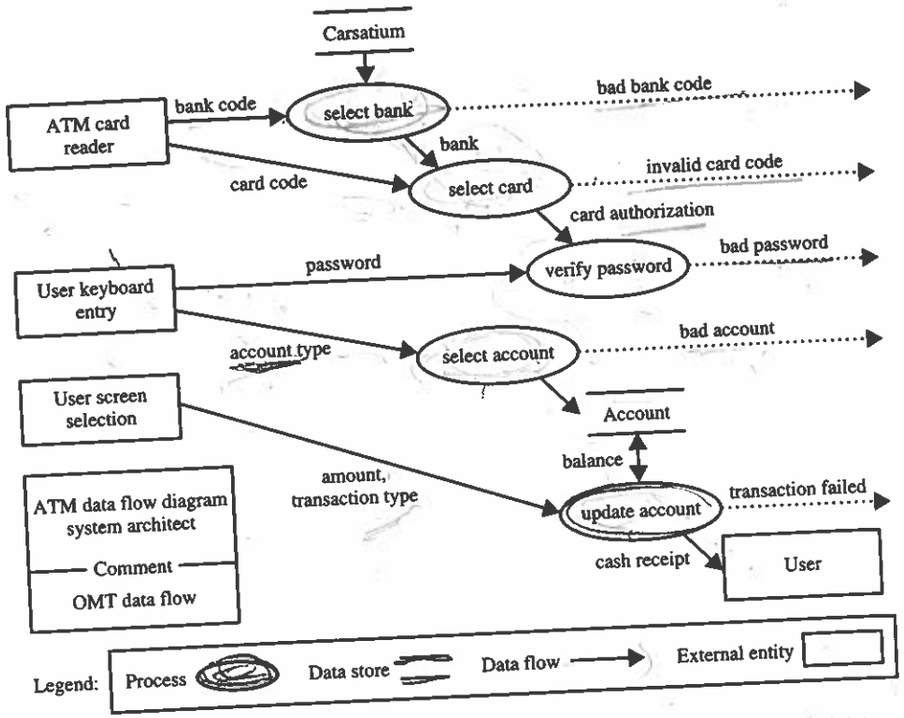



**FIGURE 4–3**
OMT DFD of the ATM system. The data flow lines include arrows to show the direction of data element movement. The circles represent processes. The boxes represent external entities. A data store reveals the storage of data.

The Booch methodology prescribes a macro development process and a micro development process.

### 4.4.1 The Macro Development Process

The macro process serves as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system. Such management is interested less in the actual object-oriented design than in how well the project corresponds to the requirements set for it and whether it is produced on time. In the macro process, the traditional phases of analysis and design to a large extent are preserved [4].

The macro development process consists of the following steps:

1. *Conceptualization.* During conceptualization, you establish the core requirements of the system. You establish a set of goals and develop a prototype to prove the concept.
2. *Analysis and development of the model.* In this step, you use the class diagram to describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system. Then, you use the object diagram to describe the desired behavior of the system in terms of scenarios or, alternatively, use the interaction diagram to describe behavior of the system in terms of scenarios.
3. *Design or create the system architecture.* In the design phase, you use the class diagram to decide what classes exist and how they relate to each other. Next, you use the object diagram to decide what mechanisms are used to regulate how objects collaborate. Then, you use the module diagram to map out where each class and object should be declared. Finally, you use the process diagram to determine to which processor to allocate a process. Also, determine the schedules for multiple processes on each relevant processor.
4. *Evolution or implementation.* Successively refine the system through many iterations. Produce a stream of software implementations (or executable releases), each of which is a refinement of the prior one.
5. *Maintenance.* Make localized changes to the system to add new requirements and eliminate bugs.



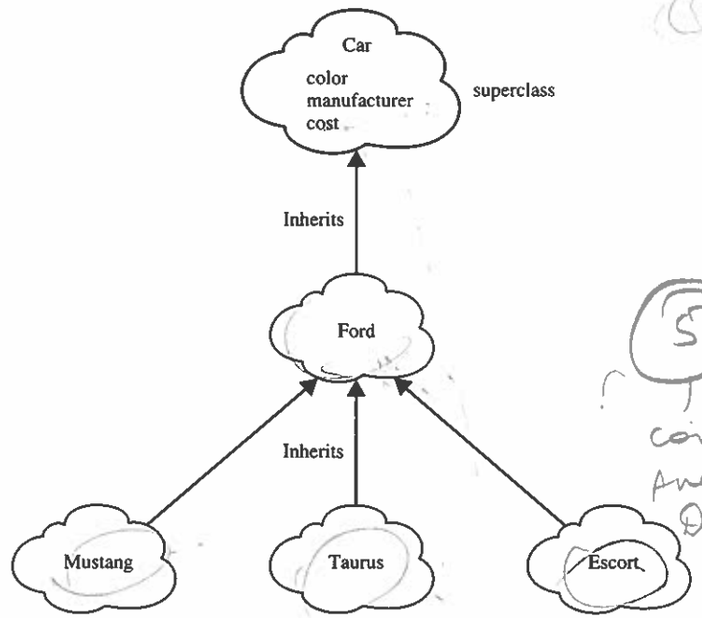



**FIGURE 4–4**
Object modeling using Booch notation. The arrows represent specialization; for example, the class Taurus is subclass of the class Ford.

### 4.4.2 The Micro Development Process

Each macro development process has its own micro development processes. The micro process is a description of the day-to-day activities by a single or small group of software developers, which could look blurry to an outside viewer, since the analysis and design phases are not clearly defined.

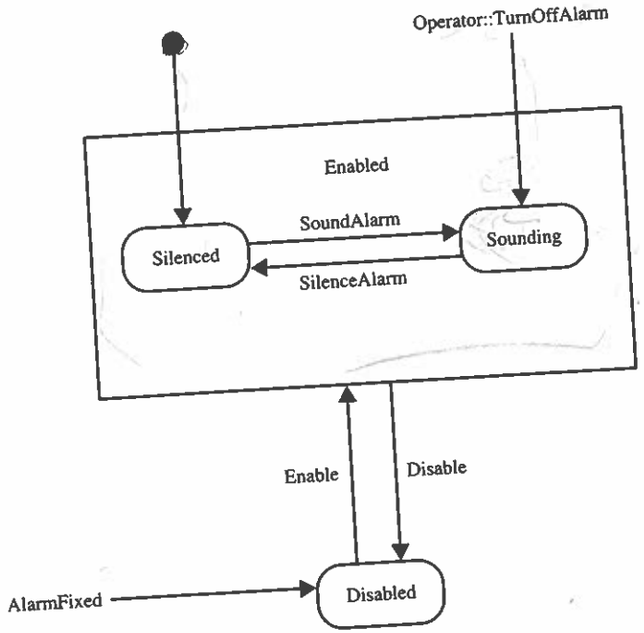



**FIGURE 4–5**
An alarm class state transition diagram with Booch notation. This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

The micro development process consists of the following steps:

1. *Identify classes and objects.*
2. *Identify class and object semantics.*
3. *Identify class and object relationships.*
4. *Identify class and object interfaces and implementation.*

## 4.5 THE JACOBSON ET AL. METHODOLOGIES

The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire life-cycle and stress traceability between the different phases, both forward and backward. This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code. At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

### 4.5.1 Use Cases

Use cases are scenarios for understanding system requirements. A use case is an interaction between users and a system. The use-case model captures the goal of



the user and the responsibility of the system to its users (see Figure 4–6). In the requirements analysis, the use cases are described as one of the following [4]:

- Nonformal text with no clear flow of events.
- Text, easy to read but with a clear flow of events to follow (this is a recommended style).
- Formal style using pseudo code.

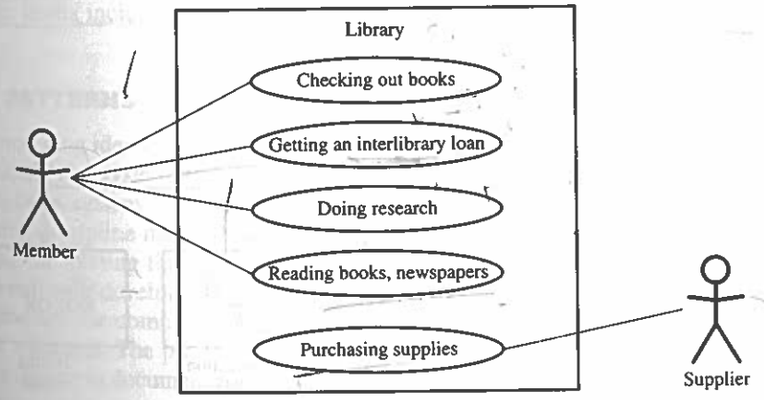
The use case description must contain

- *How* and *when* the use case begins and ends.
- The interaction between the use case and its actors, including *when* the interaction occurs and *what* is exchanged.
- *How* and *when* the use case will need data stored in the system or will store data in the system.
- *Exceptions* to the flow of events.
- *How* and *when* concepts of the problem domain are handled.

Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use-case model employs extends and uses relationships. The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it extends the functionality of the original use case (like a subclass). The uses relationship reuses common behavior in different use cases.

Use cases could be viewed as concrete or abstract. An *abstract use case* is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extends relationships.

**FIGURE 4–6**
Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.



### 4.5.2 Object-Oriented Software Engineering: Objectory

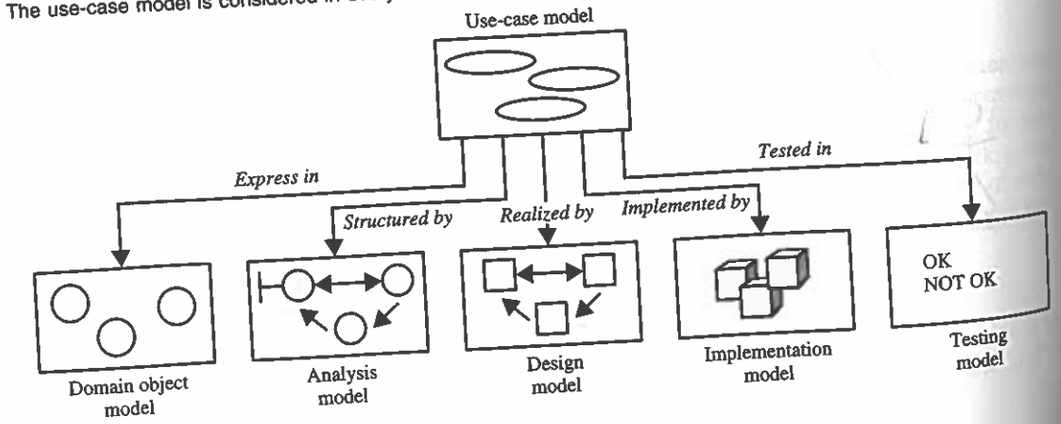
Object-oriented software engineering (OOSE), also called *Objectory*, is a method of object-oriented development with the specific aim to fit the development of large, real-time systems. The development process, called *use-case driven development*, stresses that use cases are involved in several phases of the development (see Figure 4–7), including analysis, design, validation, and testing. The use-case scenario begins with a user of the system initiating a sequence of interrelated events.

The system development method based on OOSE, Objectory, is a disciplined process for the industrialized development of software, based on a use-case driven design. It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used. By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both more usable and more robust, adapting more easily to changing usage. Jacobson et al.'s Objectory has been developed and applied to numerous application areas and embodied in the CASE tool systems.

Objectory is built around several different models:

- *Use case-model*. The use-case model defines the outside (actors) and inside (use case) of the system's behavior.
- *Domain object model*. The objects of the "real" world are mapped into the domain object model.
- *Analysis object model*. The analysis object model presents how the source code (implementation) should be carried out and written.
- *Implementation model*. The implementation model represents the implementation of the system.
- *Test model*. The test model constitutes the test plans, specifications, and reports.

**FIGURE 4–7**
The use-case model is considered in every model and phase.






The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service.

### 4.5.3 Object-Oriented Business Engineering

Object-oriented business engineering (OOBE) is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes.

- *Analysis phase*. The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model, and the analysis model. The analysis process should not take into account the actual implementation environment. This reduces complexity and promotes maintainability over the life of the system, since the description of the system will be independent of hardware and software requirements. Jacobson [16] does not dwell on the development of the problem-domain object model, but refers the developer to Coad and Yourdon's [11] or Booch's [6] discussion of the topic, who suggest that the customer draw a picture of his view of the system to promote discussions. In their view, a full development of the domain model will not localize changes and therefore will not result in the most "robust and extensible structure." This model should be developed just enough to form a base of understanding for the requirements model. The analysis process is iterative but the requirements and analysis models should be stable before moving on to subsequent models. Jacobson et al. suggest that prototyping with a tool might be useful during this phase to help specify user interfaces.
- *Design and implementation phases*. The implementation environment must be identified for the design model. This includes factors such as Database Management System (DBMS), distribution of process, constraints due to the programming language, available component libraries, and incorporation of graphical user interface tools. It may be possible to identify the implementation environment concurrently with analysis. The analysis objects are translated into design objects that fit the current implementation environment.
- *Testing phase*. Finally, Jacobson describes several testing levels and techniques. The levels include unit testing, integration testing, and system testing.

## 4.6 PATTERNS

An emerging idea in systems development is that the process can be improved significantly if a system can be analyzed, designed, and built from prefabricated and predefined system components. One of the first things that any science or engineering discipline must have is a vocabulary for expressing its concepts and a language for relating them to each other. Therefore, we need a body of literature to help software developers resolve commonly encountered, difficult problems and a vocabulary for communicating insight and experience about these problems and their solutions. The primary focus here is not so much on technology as on creating a culture to document and support sound engineering architecture and design [5].

Even though they are related in this manner, it is important to recognize that frameworks and design patterns are two distinctly separate beasts: A framework is executable software, whereas design patterns represent knowledge and experience about software. In this respect, frameworks are of a physical nature, while patterns are of a logical nature: Frameworks are the physical realization of one or more software pattern solutions; patterns are the instructions for how to implement those solutions [5].

Gamma et al. describe the major differences between design patterns and frameworks as follows [15]:

- *Design patterns are more abstract than frameworks.* Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, design patterns have to be implemented each time they are used. Design patterns also explain the intent, trade-offs, and consequences of a design.
- *Design patterns are smaller architectural elements than frameworks.* A typical framework contains several design patterns but the reverse is never true.
- *Design patterns are less specialized than frameworks.* Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these would not dictate an application architecture.

## 4.8 THE UNIFIED APPROACH

The approach promoted in this book is based on the best practices that have proven successful in system development and, more specifically, the work done by Booch, Rumbaugh, and Jacobson in their attempt to unify their modeling efforts. The unified approach (UA) (see Figure 1-1) establishes a unifying and unitary framework around their works by utilizing the unified modeling language (UML) to describe, model, and document the software development process. The idea behind the UA is not to introduce yet another methodology. The main motivation here is to combine the best practices, processes, methodologies, and *guidelines* along with UML notations and diagrams for better understanding object-oriented concepts and system development.

The unified approach to software development revolves around (but is not limited to) the following processes and concepts (see Figure 4–8). The processes are:

Use-case driven development
Object-oriented analysis
Object-oriented design
Incremental development and prototyping
Continuous testing



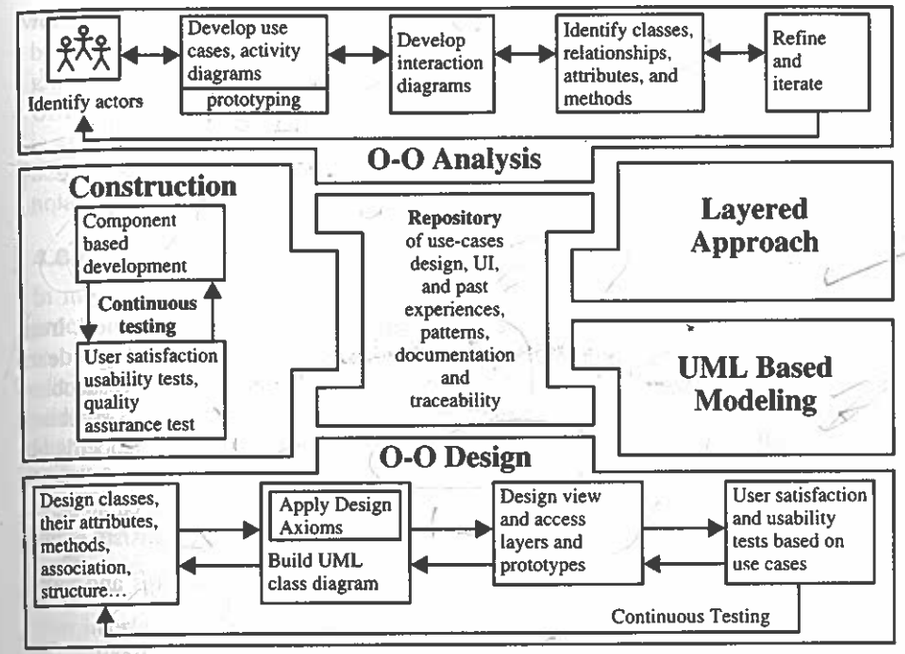



**FIGURE 4–8**
The processes and components of the unified approach.

The methods and technology employed include

Unified modeling language used for modeling.
Layered approach.
Repository for object-oriented system development patterns and frameworks.
Component-based development (Although, UA promote component-based development, the treatment of the subject is beyond the scope of the book.)

The UA allows iterative development by allowing you to go back and forth between the design and the modeling or analysis phases. It makes backtracking very easy and departs from the linear waterfall process, which allows no form of backtracking.

### 4.8.1 Object-Oriented Analysis

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. 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. 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 user's

perspective rather than that of the machine. OOA Process consists of the following Steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

### 4.8.2 Object-Oriented Design

Booch [9] provides the most comprehensive object-oriented design method. Ironically, since it is so comprehensive, the method can be somewhat imposing to learn and especially tricky to figure out where to start. Rumbaugh et al.'s and Jacobson et al.'s high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson et al.'s analysis and interaction diagrams, Booch's object diagrams, and Rumbaugh et al.'s domain models. Furthermore, by following Jacobson et al.'s life cycle model, we can produce designs that are traceable across requirements, analysis, design, coding, and testing. OOD Process consists of:

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User interface
- User Satisfaction and Usability Tests based on the Usage/Use Cases
- Iterate and refine the design

### 4.8.3 Iterative Development and Continuous Testing

You must iterate and reiterate until, eventually, you are satisfied with the system. Since testing often uncovers design weaknesses or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving on to reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results. During this iterative process, your prototypes will be incrementally transformed into the actual application. The UA encourages the integration of testing plans from day 1 of the project. Usage scenarios can become test scenarios; therefore, use cases will drive the usability testing. Usability testing is the process in which the functionality of software is measured. Chapter 13 will cover usability testing.

### 4.8.4 Modeling Based on the Unified Modeling Language

The unified modeling language was developed by the joint efforts of the leading object technologists Grady Booch, Ivar Jacobson, and James Rumbaugh with contributions from many others. The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch's methodology, Jacobson et al.'s use case, and Rumbaugh et al.'s object modeling technique. The UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways. The UML has become the standard notation for object-oriented modeling systems. It is an evolving notation that still is under development. The UA uses the UML to describe and model the analysis and design phases of system development (UML notations will be covered in Chapter 5).



### 4.8.5 The UA Proposed Repository

In modern businesses, best practice sharing is a way to ensure that solutions to process and organization problems in one part of the business are communicated to other parts where similar problems occur. Best practice sharing eliminates duplication of problem solving. For many companies, best practice sharing is institutionalized as part of their constant goal of quality improvement. Best practice sharing must be applied to application development if quality and productivity are to be added to component reuse benefits. Such sharing extends the idea of software reusability to include all phases of software development such as analysis, design, and testing [22].

The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format. As we saw previously, central to the discussion on developing this best practice sharing is the concept of a pattern. Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository. The advantage of repositories is that, if your organization has done projects in the past, objects in the repositories from those projects might be useful. You can select any piece from a repository—from the definition of one data element, to a diagram, all its symbols, and all their dependent definitions, to entries—for reuse.

The UA's underlying assumption is that, if we design and develop applications based on previous experience, creating additional applications will require no more than assembling components from the library. Additionally, applying lessons learned from past developmental mistakes to future projects will increase the quality of the product and reduce the cost and development time. Some basic capability is available in most object-oriented environments, such as Microsoft repository, VisualAge, PowerBuilder, Visual C++, and Delphi. These repositories contain all objects that have been previously defined and can be reused for putting together a new software system for a new application. If a new requirement surfaces, new objects will be designed and stored in the main repository for future use.

The same arguments can be made about patterns and frameworks. Specifications of the software components, describing the behavior of the component and how it should be used, are registered in the repository for future reuse by teams of developers.

The repository should be accessible to many people. Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods,

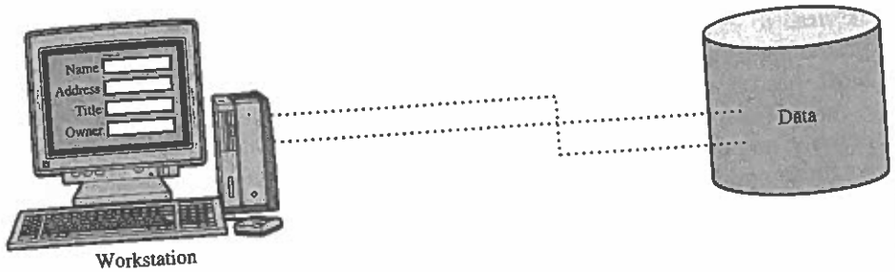



**FIGURE 4–9**
Two-layered architecture: interface and data.

or other characteristics. For example, application developers could select prebuilt components from the central component repository that match their business needs and assemble these components into a single application, customizing where needed.

Tools to fully support a comprehensive repository are not accessible yet, but this will change quickly and, in the near future, we will see more readily available tools to capture all phases of software development into a repository for use and reuse.

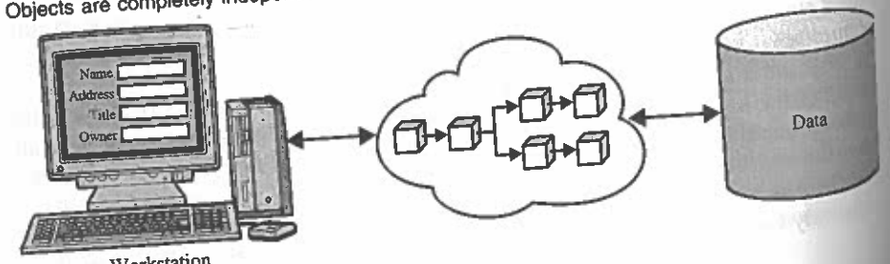
### 4.8.6 The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known as *two-layered architecture*: interface and data (see Figure 4–9).

In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens; for example, a routine that executes when you click on a button. With every interface you create, you must re-create the business logic needed to run the screen. The routines required to access the data must exist within every screen. Any change to the business logic must be accomplished in every screen that deals with that portion of the business. This approach results in objects that are very specialized and cannot be reused easily in other projects.

A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business. This approach also isolates the business from the details of the data access (see Figure 4–10). Using the three-

**FIGURE 4–10**
Objects are completely independent of how they are represented or stored.





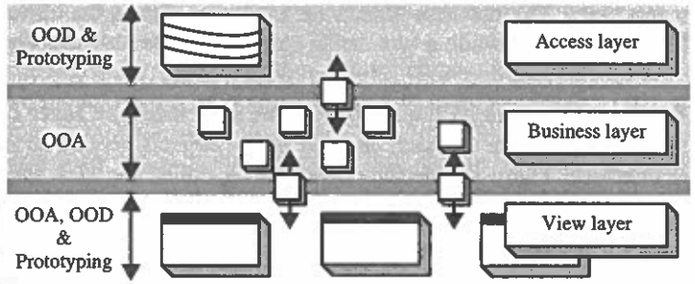




**FIGURE 4–11**
Business objects represent tangible elements of the application. They should be completely independent of how they are represented to the user or how they are physically stored.

layered approach, you are able to create objects that represent tangible elements of your business yet are completely independent of how they are represented to the user (through an interface) or how they are physically stored (in a database). The three-layered approach consists of a view or user interface layer, a business layer, and an access layer (see Figure 4–11).

**4.8.6.1 The Business Layer** The business layer contains all the objects that represent the business (both data and behavior). This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist. Most modern object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.

The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes. When creating the business layer, however, it is important to keep in mind a couple of things. These objects should not be responsible for the following:

- *Displaying details*. Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.
- *Data access details*. Business objects also should have no special knowledge of "where they come from." It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object-oriented analysis.

A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability. An individual business object can appear in different business models. Business models also incorporate control objects that

direct their processes. The business objects are identified during the object-oriented analysis. Use cases can provide a wonderful tool to capture business objects.

**4.8.6.2 The User Interface (View) Layer** The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The user interface layer also is called the *view layer*.

This layer typically is responsible for two major aspects of the applications:

- *Responding to user interaction*. The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process; remember, the business logic does not exist here, just the knowledge of which message to send to which business object.
- *Displaying business objects*. This layer must paint the best possible picture of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.

The user interface layer's objects are identified during the object-oriented design phase. However, the requirement for a user interface or how a user will use the system is the responsibility of object-oriented analysis. Use cases can provide a very useful tool for understanding user interface requirements.

**4.8.6.3 The Access Layer** The access layer contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database, mainframe, Internet, or file. Regardless of where the data actually reside, the access layer has two major responsibilities:

- *Translate request*. The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. (For example, if Customer number 55552 needs to be retrieved, the access layer must be able to create the correct SQL statement and execute it.)
- *Translate results*. The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back up into the business layer.

Access objects are identified during object-oriented design.

## 4.9 SUMMARY

In this chapter, we looked at current trends in object-oriented methodologies, sometimes known as *second-generation object-oriented methods*, which have been toward combining the best aspects of today's most popular methods.

Each method has its strengths. Rumbaugh et al. have a strong method for producing object models (sometimes known as *domain object models*). Jacobson et al. have a strong method for producing user-driven requirement and object-oriented



analysis models. Booch has a strong method for producing detailed object-oriented design models.

Each method has a weakness, too. While Rumbaugh et al.'s OMT has strong methods for modeling the problem domain, OMT models cannot fully express the requirements. Jacobson et al. deemphasize object modeling and, although they cover a fairly wide range of the life cycle, they do not treat object-oriented design to the same level as Booch, who focuses almost entirely on design, not analysis.

Booch and Rumbaugh et al. are object centered in their approaches and focus more on figuring out what are the objects of a system, how are they related, and how do they collaborate with each other. Jacobson et al. are more user centered, in that everything in their approach derives from use cases or usage scenarios.

The main idea behind a pattern is the documentation to help categorize, communicate about, and locate solutions to recurring problems. Frameworks are a way of delivering application development patterns to support best practice sharing during application development. A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns. Writing good patterns is very difficult, since it should not only provide facts but also tell a story that captures the experience the pattern is trying to convey.

The UA is an attempt to combine the best practices, processes, and guidelines along with UML notations and diagrams for better understanding object-oriented concepts and object-oriented system development. The UA consists of the following processes:

- Use-case driven development
- Object-oriented analysis
- Object-oriented design
- Incremental development and prototyping
- Continuous testing

Futhermore, it utilizes the methods and technologies such as, unified modeling language, layered approach and promotes repository for all phases of software development.

## KEY TERMS

Abstract use case (p. 69)
Framework (p. 77)
Pattern (p. 72)
Pattern mining (p. 76)
Pattern thumbnail (p. 76)
Proto-pattern (p. 73)

## REVIEW QUESTIONS

1. What is a method?
2. What is a methodology?
3. What is process?

Efraim Turban [9] describes a model as a simplified representation of reality. A model is simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem we are trying to describe or solve. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus, modeling enables us to cope with the complexity of a system.

Most modeling techniques used for analysis and design involve graphic languages. These graphic languages are sets of symbols. The symbols are used according to certain rules of the methodology for communicating the complex relationships of information more clearly than descriptive text. The main goal of most CASE tools is to aid us in using these graphic languages, along with their associated methodologies.

Modeling frequently is used during many of the phases of the software life cycle, such as analysis, design, and implementation. For example, Objectory is built around several different models:

- *Use-case model*. The use-case model defines the outside (actors) and inside (use case) of the system's behavior.
- *Domain object model*. Objects of the "real" world are mapped into the domain object model.
- *Analysis object model*. The analysis object model presents how the source code (i.e., the implementation) should be carried out and written.
- *Implementation model*. The implementation model represents the implementation of the system.
- *Test model*. The test model constitutes the test plans, specifications, and reports.

Modeling, like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same.

In this chapter, we look at unified modeling language (UML) notations and diagrams. The main idea here is to gain exposure to the UML syntax, semantics, and modeling constructs. Many new concepts will be introduced here from a modeling standpoint. We apply these concepts in system analysis and design contexts in later chapters.

## 5.2 STATIC AND DYNAMIC MODELS

Models can represent static or dynamic situations. Each representation has different implications for how the knowledge about the model might be organized and represented [7].

### 5.2.1 Static Model

A ***static model*** can be viewed as a snapshot of a system's parameters at rest or at a specific point in time. Static models are needed to represent the structural or



static aspect of a system. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Static models assume stability and an absence of change in data over time. The unified modeling language class diagram is an example of a static model.

### 5.2.2 Dynamic Model

A ***dynamic model***, in contrast to a static model, can be viewed as a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability.

A system can be described by first developing its static model, which is the structure of its objects and their relationships to each other frozen in time, a baseline. Then, we can examine changes to the objects and their relationships over time. Dynamic modeling is most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of UML dynamic models.

## 5.3 WHY MODELING?

Building a model for a software system prior to its construction is as essential as having a blueprint for building a large building. Good models are essential for communication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Many other factors add to a project's success, but having a rigorous modeling language is essential. A modeling language must include [2]

- Model elements—fundamental modeling concepts and semantics.
- Notation—visual rendering of model elements.
- Guidelines—expression of usage within the trade.

In the face of increasingly complex systems, visualization and modeling become essential, since we cannot comprehend any such system in its entirety. The use of visual notation to represent or model a problem can provide us several benefits relating to clarity, familiarity, maintenance, and simplification.

- *Clarity*. We are much better at picking out errors and omissions from a graphical or visual representation than from listings of code or tables of numbers. We very easily can understand the system being modeled because visual examination of the whole is possible.
- *Familiarity*. The representation form for the model may turn out to be similar to the way in which the information actually is represented and used by the employees currently working in the problem domain. We, too, may find it more comfortable to work with this type of representation.
- *Maintenance*. Visual notation can improve the maintainability of a system. The visual identification of locations to be changed and the visual confirmation of

those changes will reduce errors. Thus, you can make changes faster, and fewer errors are likely to be introduced in the process of making those changes.

- *Simplification.* Use of a higher level representation generally results in the use of fewer but more general constructs, contributing to simplicity and conceptual understanding.

Turban cites the following advantages of modeling [9]:

1. Models make it easier to express complex ideas. For example, an architect builds a model to communicate ideas more easily to clients.
2. The main reason for modeling is the reduction of complexity. Models reduce complexity by separating those aspects that are unimportant from those that are important. Therefore, it makes complex situations easier to understand.
3. Models enhance and reinforce learning and training.
4. The cost of the modeling analysis is much lower than the cost of similar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulating a real system.

To summarize, here are a few key ideas regarding modeling:

- A model is rarely correct on the first try.
- Always seek the advice and criticism of others. You can improve a model by reconciling different perspectives.
- Avoid excess model revisions, as they can distort the essence of your model. Let simplicity and elegance guide you through the process.

## 5.4 INTRODUCTION TO THE UNIFIED MODELING LANGUAGE

The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as *object constraint language* (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

The goals of the unification efforts were to keep it simple; to cast away elements of existing Booch, OMT, and OOSE methods that did not work in practice; to add elements from other methods that were more effective; and to invent new methods only when an existing solution was unavailable. Because the UML authors, in effect, were designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and overengineering (having a symbol or figure for every conceivable modeling element). To that end, they were very careful about adding new things: They did not want to make the UML unnecessarily complex. A similar situation exists with the problem of UML not supporting other diagrams. Booch et al. explain that other diagrams, such as the data flow diagram (DFD), were not included in the UML because they do not fit as cleanly into a consistent object-oriented paradigm. For example, activity diagrams accomplish much of what people want from DFDs and then some; activity diagrams also are useful for modeling work flow. The authors of the UML clearly are promoting the UML diagrams over all others for object-oriented projects but do not condemn all other diagrams. Along the way, however, some things were found that were advantageous to add because they had proven useful in other modeling practice.



The primary goals in the design of the UML were as follows [2, p. 3]:

1. Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

This section of the chapter is based on the *The Unified Modeling Language, Notation Guide Version 1.1* written by Grady Booch, Ivar Jacobson, and James Rumbaugh [2].

## 5.5 UML DIAGRAMS

Every complex system is best approached through a small set of nearly independent views of a model; no single view is sufficient. Every model may be expressed at different levels of fidelity. The best models are connected to reality. The UML defines nine graphical diagrams:

1. Class diagram (static)
2. Use-case diagram
3. Behavior diagram (dynamic):
   - 3.1. Interaction diagram:
     - 3.1.1. Sequence diagram
     - 3.1.2. Collaboration diagram
   - 3.2. Statechart diagram
   - 3.3. Activity diagram
4. Implementation diagram:
   - 4.1. Component diagram
   - 4.2. Deployment diagram

The choice of what models and diagrams one creates has a great influence on how a problem is encountered and how a corresponding solution is shaped. We will study applications of different diagrams throughout the book. However, in this chapter we concentrate on the UML notations and its semantics.

## 5.6 UML CLASS DIAGRAM

The UML ***class diagram***, also referred to as *object modeling*, is the main static analysis diagram. These diagrams show the static structure of the model. A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents; for example, the things that exist (such as classes), their internal structures, and their relationships to other classes. Class diagrams do not show temporal information, which is required in dynamic modeling.

Object modeling is the process by which the logical objects in the real world (problem space) are represented (mapped) by the actual objects in the program (logical or a mini world). This visual representation of the objects, their relationships, and their structures is for ease of understanding. To effectively develop a model of the real world and to determine the objects required in the system, you first must ask what objects are needed to model the system. Answering the following questions will help you to stay focused on the problem at hand and determine what is inside the problem domain and what is outside it:

- What are the goals of the system?
- What must the system accomplish?

You need to know what objects will form the system because, in the object-oriented viewpoint, objects are the primary abstraction. The main task of object modeling is to graphically show what each object will do in the problem domain, describe the structure (such as class hierarchy or part-whole) and the relationships among objects (such as associations) by visual notation, and determine what behaviors fall within and outside the problem domain.

### 5.6.1 Class Notation: Static Structure

A class is drawn as a rectangle with three components separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes, are in the middle compartment, and the bottom compartment holds a list of operations (see Figure 5–1).

Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it. The class name and other properties should be displayed in up to three sections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

### 5.6.2 Object Diagram

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.



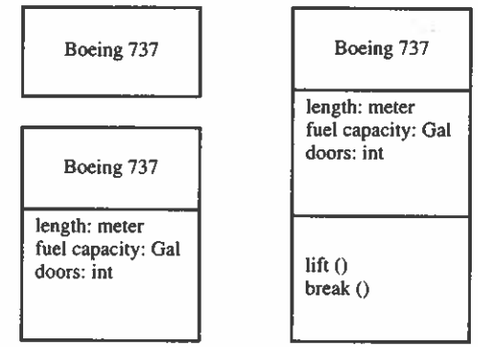



**FIGURE 5–1**
In class notation, either or both the attributes and operation compartments may be suppressed.

### 5.6.3 Class Interface Notation

Class interface notation is used to describe the externally visible behavior of a class; for example, an operation with public visibility. Identifying class interfaces is a design activity of object-oriented system development. The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations. For example, a Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Figure 5–2 with UML class interface notation.

### 5.6.4 Binary Association Notation

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. Furthermore, the association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name. The end of an association, where it connects to a class, is called the *association role* (see Figure 5–3).

### 5.6.5 Association Role

A simple association—the technical term for it is *binary association*—is drawn as a solid line connecting two class symbols. The end of an association, where it connects to a class, shows the association role. The role is part of the association, not

**FIGURE 5–2**
Interface notation of a class.

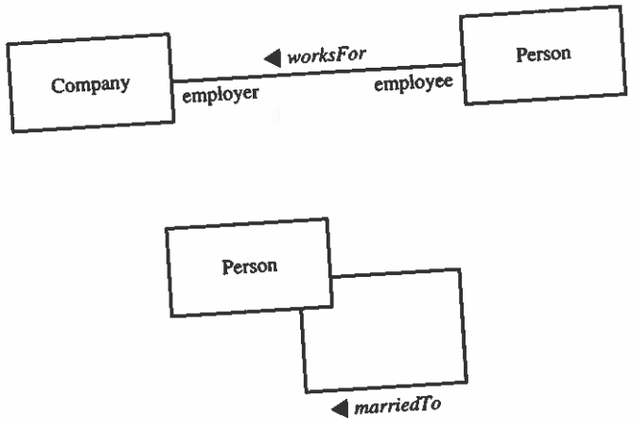



**FIGURE 5–3**
Association notation.

part of the class. Each association has two or more roles to which it is connected. In Figure 5–3, the association worksFor connects two roles, employee and employer. A Person is an employee of a Company and a Company is an employer of a Person.

The UML uses the term *association navigation* or *navigability* to specify a role affiliated with each end of an association relationship. An arrow may be attached to the end of the path to indicate that navigation is supported in the direction of the class pointed to. An arrow may be attached to neither, one, or both ends of the path. In particular, arrows could be shown whenever navigation is supported in a given direction. In the UML, association is represented by an open arrow, as represented in Figure 5–4. Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

In Figure 5–4, the association is navigable in only one direction, from the BankAccount to Person, but not the reverse. This might indicate a design decision, but it also might indicate an analysis decision, that the Person class is frozen and cannot be extended to know about the BankAccount class, but the BankAccount class can know about the Person class.

### 5.6.6 Qualifier

A ***qualifier*** is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the account#. The account# is the qualifier of this association (see Figure 5–5).

**FIGURE 5–4**
Association notation.



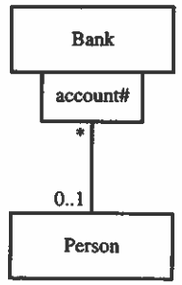




**FIGURE 5–5**
The figure depicts association qualifier and its multiplicity.

A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle (see Figure 5–5).

### 5.6.7 Multiplicity

***Multiplicity*** specifies the range of allowable associated classes. It is given for roles within associations, parts within compositions, repetitions, and other purposes. A multiplicity specification is shown as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in this format (see Figure 5–5):

lower bound .. upper bound.

The terms *lower bound* and *upper bound* are integer values, specifying the range of integers including the lower bound to the upper bound. The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer range contains the single values. For example,

0..1
0..*
1..3, 7..10, 15, 19..*

### 5.6.8 OR Association

An ***OR association*** indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string {or} labeling the dashed line (see Figure 5–6). In other words, any instance of the class may participate in, at most, one of the associations at one time.

### 5.6.9 Association Class

An ***association class*** is an association that also has class properties. An association class is shown as a class symbol attached by a dashed line to an association

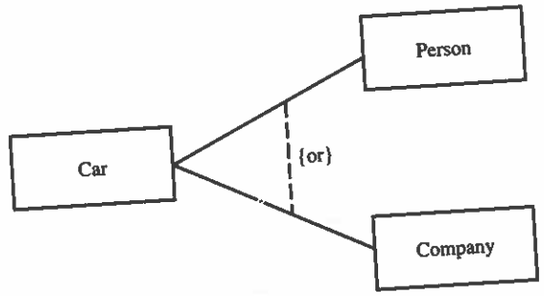


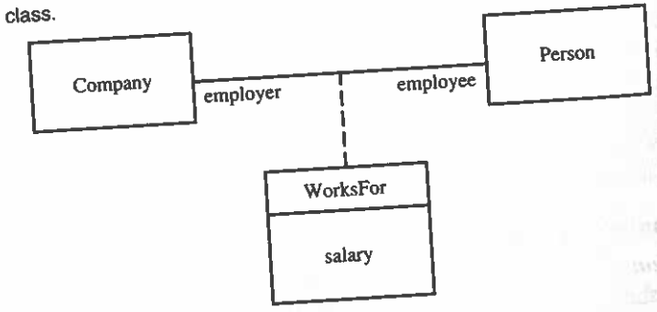

**FIGURE 5–6**
An OR association notation. A car may associate with a person or a company.

path. The name in the class symbol and the name string attached to the association path are the same (see Figure 5–7). The name can be shown on the path or the class symbol or both. If an association class has attributes but no operations or other associations, then the name may be displayed on the association path and omitted from the association class to emphasize its "association nature." If it has operations and attributes, then the name may be omitted from the path and placed in the class rectangle to emphasize its "class nature."

### 5.6.10 N-Ary Association

An ***n-ary association*** is an association among more than two classes. Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association. However, here, for the sake of completeness, we cover the notation of n-ary association. An n-ary association is shown as a large diamond with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. The role attachment may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted. An association class symbol may be attached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation, or associations. The example depicted in Figure 5–8 shows the grade book of a class in each semester.

**FIGURE 5–7**
Association class.





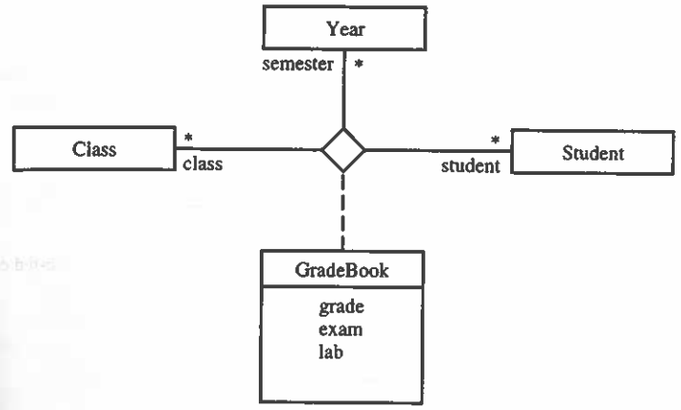




**FIGURE 5–8**
An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab.

### 5.6.11 Aggregation and Composition (a-part-of)

Aggregation is a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, and it need not be presented at all (see Figure 5–9).

Composition, also known as the *a-part-of*, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a *part-whole relationship*. The UML notation for composition is a solid diamond at the end of a path. Alternatively, the UML provides a graphically nested form that, in many cases, is more convenient for showing composition (see Figure 5–10).

Parts with multiplicity greater than one may be created after the aggregate itself but, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

### 5.6.12 Generalization

*Generalization* is the relationship between a more general class and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowhead

**FIGURE 5–9**
Association path.

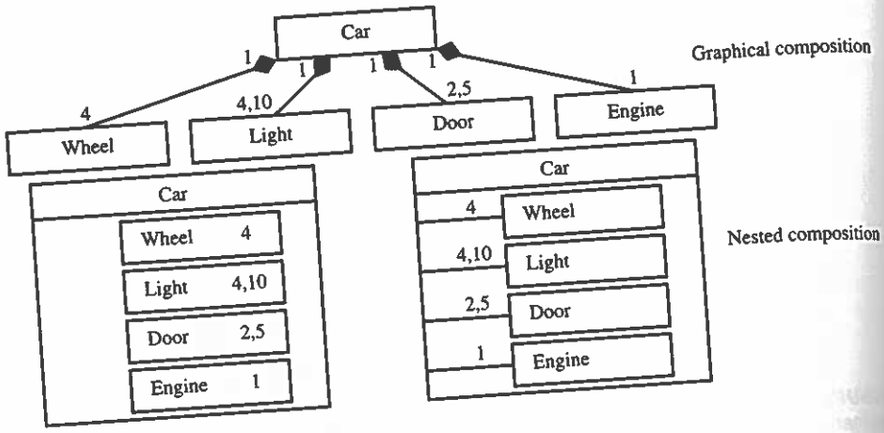
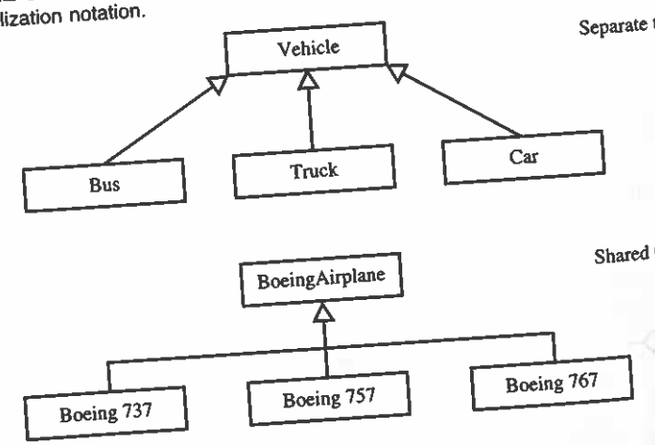



**FIGURE 5–10**
Different ways to show composition.

at the superclass end (see Figure 5–11). The UML allows a *discriminator* label to be attached to a generalization of the superclass. For example, the class BoeingAirplane has instances of the classes Boeing 737, Boeing 747, Boeing 757, and Boeing 767, which are subclasses of the class BoeingAirplane. Ellipses (...) indicate that the generalization is incomplete and more subclasses exist that are not shown (see Figure 5–12). The constructor complete indicates that the generalization is complete and no more subclasses are needed.

If a text label is placed on the hollow triangle shared by several generalization paths to subclasses, the label applies to all of the paths. In other words, all subclasses share the given properties.

**FIGURE 5–11**
Generalization notation.





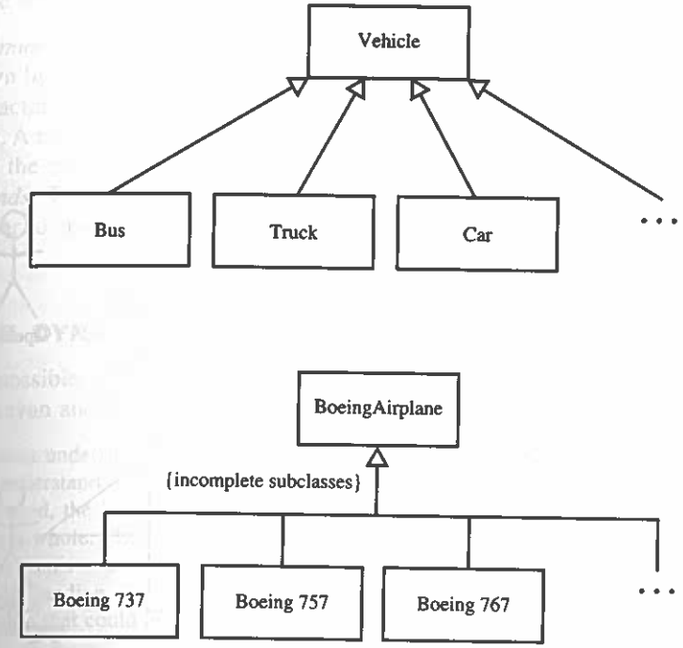




**FIGURE 5–12**
Ellipses (. . .) indicate that additional classes exist and are not shown.

## 5.7 USE-CASE DIAGRAM

The use-case concept was introduced by Ivar Jacobson in the object-oriented software engineering (OOSE) method [5]. The functionality of a system is described in a number of different use cases, each of which represents a specific flow of events in the system.

A use case corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors) and engages internal objects to interact with one another and with the system's surroundings.

The description of a use case defines what happens in the system when the use case is performed. In essence, the use-case model defines the outside (actors) and inside (use case) of the system's behavior. Use cases represent specific flows of events in the system. The use cases are initiated by actors and describe the flow of events that these actors set off. An actor is anything that interacts with a use case: It could be a human user, external hardware, or another system. An actor represents a category of user rather than a physical user. Several physical users can play the same role. For example, in terms of a Member actor, many people can be members of a library, which can be represented by one actor called *Member*.

A ***use-case diagram*** is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.

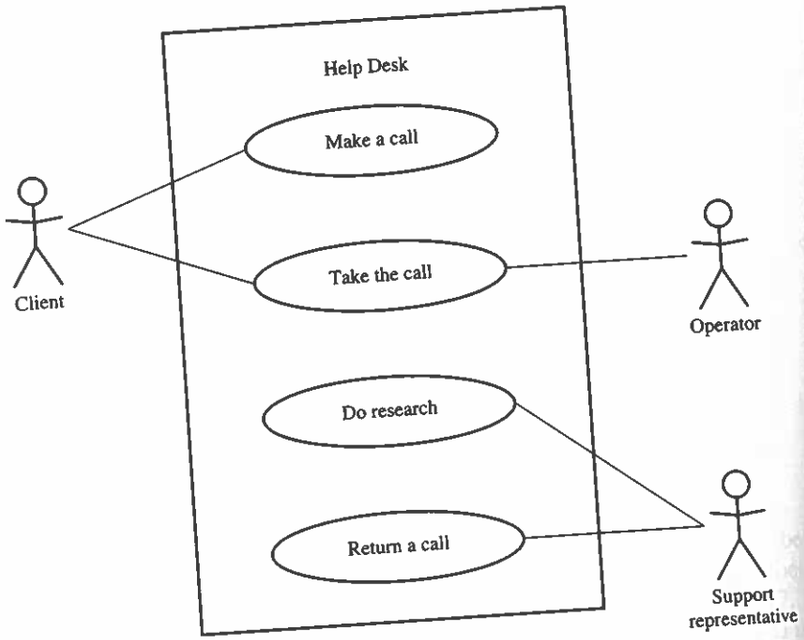


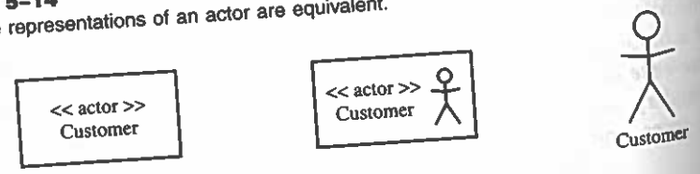

**FIGURE 5–13**
A use-case diagram shows the relationship among actors and use cases within a system.

Figure 5–13 diagrams use cases for a Help Desk. A use-case diagram shows the relationship among the actors and use cases within a system. A client makes a call that is taken by an operator, who determines the nature of the problem. Some calls can be answered immediately; other calls require research and a return call.

A use case is shown as an ellipse containing the name of the use case. The name of the use case can be placed below or inside the ellipse. Actors' names and use case names should follow the capitalization and punctuation guidelines of the model.

An actor is shown as a class rectangle with the label <<actor>>, or the label and a stick figure, or just the stick figure with the name of the actor below the figure (see Figure 5–14).

**FIGURE 5–14**
The three representations of an actor are equivalent.






These relationships are shown in a use-case diagram:

1. *Communication*. The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid path. The actor is said to "communicate" with the use case.
2. *Uses*. A uses relationship between use cases is shown by a generalization arrow from the use case.
3. *Extends*. The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it is like a subclass.

## 5.8 UML DYNAMIC MODELING (BEHAVIOR DIAGRAMS)

It is impossible to capture all details of a complex system in just one model or view. Kleyan and Gingrich explain:

> One must understand both the structure and the function of the objects involved. One must understand the taxonomic structure of class objects, the inheritance and mechanisms used, the individual behaviors of objects, and the dynamic behavior of the system as a whole. The problem is somewhat analogous to that of viewing a sports event such as tennis or a football game. Many different camera angles are required to provide an understanding of the action taking place. Each camera reveals particular aspects of the action that could not be conveyed by one camera alone. [6]

The diagrams we have looked at so far largely are static. However, events happen dynamically in all systems: Objects are created and destroyed, objects send messages to one another in an orderly fashion, and in some systems, external events trigger operations on certain objects. Furthermore, objects have states. The state of an object would be difficult to capture in a static model.

> The state of an object is the result of its behavior. Booch provides us an excellent example: "When a telephone is first installed, it is in idle state, meaning that no previous behavior is of great interest and that the phone is ready to initiate and receive calls. When someone picks up the handset, we say that the phone is now off-hook and in the dialing state; in this state, we do not expect the phone to ring: we expect to be able to initiate a conversation with a party or parties on another telephone. When the phone is on-hook, if it rings and then we pick up the handset, the phone is now in the receiving state, and we expect to be able to converse with the party that initiated the conversation [1]."

Booch explains that describing a systematic event in a static medium such as on a sheet of paper is difficult, but the problem confronts almost every discipline. In object-oriented development, you can express the dynamic semantics of a problem with the following diagrams:

Behavior diagrams (dynamic):
- Interaction diagrams:
  - Sequence diagrams
  - Collaboration diagrams
- Statechart diagrams
- Activity diagrams

- Corollary 2. *Single purpose*. Each class must have a single, clearly defined purpose. When you document, you should be able to easily describe the purpose of a class in a few sentences.
- Corollary 3. *Large number of simple classes*. Keeping the classes simple allows reusability.
- Corollary 4. *Strong mapping*. There must be a strong association between the physical system (analysis's object) and logical design (design's object).
- Corollary 5. *Standardization*. Promote standardization by designing interchangeable components and reusing existing classes or components.
- Corollary 6. *Design with inheritance*. Common behavior (methods) must be moved to superclasses. The superclass-subclass structure must make logical sense.

### 9.4.1 Corollary 1. Uncoupled Design with Less Information Content

The main goal here is to maximize objects cohesiveness among objects and software components in order to improve coupling because only a minimal amount of essential information need be passed between components.

**9.4.1.1 Coupling** *Coupling* is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship: A is coupled with B. Coupling is important when evaluating a design because it helps us focus on an important issue in design. For example, a change to one component of a system should have a minimal impact on other components [3]. Strong coupling among objects complicates a system, since the class is harder to understand or highly interrelated with other classes. The degree of coupling is a function of

1. How complicated the connection is.
2. Whether the connection refers to the object itself or something inside it.
3. What is being sent or received.

The degree, or strength, of coupling between two components is measured by the amount and complexity of information transmitted between them. Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface. Coupling decreases (becomes lower) when the connection is to the component interface rather than to an internal component. Coupling also is lower for data connections than for control connections. Object-oriented design has two types of coupling: interaction coupling and inheritance coupling [3].

Interaction coupling involves the amount and complexity of messages between components. It is desirable to have little interaction. Coupling also applies to the complexity of the message. The general guideline is to keep the messages as simple and infrequent as possible. In general, if a message connection involves more than three parameters (e.g., in Method (X, Y, Z), the X, Y, and Z are parameters), examine it to see if it can be simplified. It has been documented that objects connected to many very complex messages are tightly coupled, meaning any change to one invariability leads to a ripple effect of changes in others (see Figure 9–3).



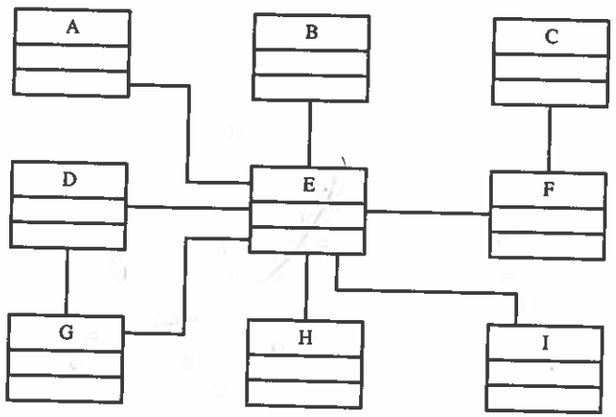



**FIGURE 9–3**
E is a tightly coupled object.

In addition to minimizing the complexity of message connections, also reduce the number of messages sent and received by an object [3]. Table 9–1 contains different types of interaction couplings.

Inheritance is a form of coupling between super- and subclasses. A subclass is coupled to its superclass in terms of attributes and methods. Unlike interaction coupling, high inheritance coupling is desirable. However, to achieve high inheritance

**TABLE 9–1**

**TYPES OF COUPLING AMONG OBJECTS OR COMPONENTS (shown from highest to lowest)**

| Degree of coupling | Name | Description |
|---|---|---|
| Very high | Content coupling | The connection involves direct reference to attributes or methods of another object. |
| High | Common coupling | The connection involves two objects accessing a "global data space," for both to read and write. |
| Medium | Control coupling | The connection involves explicit control of the processing logic of one object by another. |
| Low | Stamp coupling | The connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure. |
| Very low | Data coupling | The connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. This should be the goal of an architectural design. |

coupling in a system, each specialization class should not inherit lots of unrelated and unneeded methods and attributes. For example, if the subclass is overwriting most of the methods or not using them, this is an indication inheritance coupling is *low* and the designer should look for an alternative generalization-specialization structure (see Corollary 6).

**9.4.1.2 Cohesion** Coupling deals with interactions between objects or software components. We also need to consider interactions within a single object or software component, called ***cohesion***. Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes (see Corollaries 2 and 3).

Method cohesion, like function cohesion, means that a method should carry only one function. A method that carries multiple functions is undesirable. Class cohesion means that all the class's methods and attributes must be highly cohesive, meaning to be used by internal methods or derived classes' methods. Inheritance cohesion is concerned with the following questions [3]:

- How interrelated are the classes?
- Does specialization really portray specialization or is it just something arbitrary?

See Corollary 6, which also addresses these questions.

### 9.4.2 Corollary 2. Single Purpose

Each class must have a purpose, as was explained in Chapter 7. Every class should be clearly defined and necessary in the context of achieving the system's goals. When you document a class, you should be able to easily explain its purpose in a sentence or two. If you cannot, then rethink the class and try to subdivide it into more independent pieces. In summary, keep it simple; to be more precise, each method must provide only one service. Each method should be of moderate size, no more than a page; half a page is better.

### 9.4.3 Corollary 3. Large Number of Simpler Classes, Reusability

A great benefit results from having a large number of simpler classes. You cannot possibly foresee all the future scenarios in which the classes you create will be reused. The less specialized the classes are, the more likely future problems can be solved by a recombination of existing classes, adding a minimal number of subclasses. A class that easily can be understood and reused (or inherited) contributes to the overall system, while a complex, poorly designed class is just so much dead weight and usually cannot be reused. Keep the following guideline in mind:

> The smaller are your classes, the better are your chances of reusing them in other projects. Large and complex classes are too specialized to be reused.

Object-oriented design offers a path for producing libraries of reusable parts [2]. The emphasis object-oriented design places on encapsulation, modularization, and



polymorphism suggests reuse rather than building anew. Cox's description of a software IC library implies a similarity between object-oriented development and building hardware from a standard set of chips [5]. The software IC library is realized with the introduction of design patterns, discussed later in this chapter.

Coad and Yourdon argue that software reusability rarely is practiced effectively. But the organizations that will survive in the 21st century will be those that have achieved high levels of reusability—anywhere from 70–80 percent or more [3]. Griss [6] argues that, although reuse is widely desired and often the benefit of utilizing object technology, many object-oriented reuse efforts fail because of too narrow a focus on technology and not on the policies set forth by an organization. He recommended an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. These assets consistently are used and maintained to obtain high levels of reuse, thereby optimizing the organization's ability to produce high-quality software products rapidly and effectively [6].

Coad and Yourdon [3] describe four reasons why people are not utilizing this concept:

1. Software engineering textbooks teach new practitioners to build systems from "first principles"; reusability is not promoted or even discussed.
2. The "not invented here" syndrome and the intellectual challenge of solving an interesting software problem in one's own unique way mitigates against reusing someone else's software component.
3. Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.
4. Most organizations provide no reward for reusability; sometimes productivity is measured in terms of new lines of code written plus a discounted credit (e.g., 50 percent less credit) for reused lines of code.

The primary benefit of software reusability is higher productivity. Roughly speaking, the software development team that achieves 80 percent reusability is four times as productive as the team that achieves only 20 percent reusability. Another form of reusability is using a design pattern, which will be explained in the next section.

### 9.4.4 Corollary 4. Strong Mapping

Object-oriented analysis and object-oriented design are based on the same model. As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same. For example, during analysis we might identify a class Employee. During the design phase, we need to design this class—design its methods, its association with other objects, and its view and access classes. A strong mapping links classes identified during analysis and classes designed during the design phase (e.g., view and access classes). Martin and Odell describe this important issue very elegantly:

about the world around us. As new facts are acquired, we relate them to existing structures in our environment (model). After enough new facts are acquired about a certain area, we create new structures to accommodate the greater level of detail in our knowledge.

The single most important activity in designing an application is coming up with a set of classes that work together to provide the functionality you desire. A given problem always has many solutions. However, at this stage, you must translate the attributes and operations into system implementation. You need to decide where in the class tree your new classes will go. Many object-oriented programming languages and development environments, such as Smalltalk, C++, or PowerBuilder, come with several built-in class libraries. Your goal in using these systems should be to reuse rather than create anew. Similarly, if you design your classes with reusability in mind, you will gain a lot in productivity and reduce the time for developing new applications.

The first step in building an application, therefore, should be to design a set of classes, each of which has a specific expertise and all of which can work together in useful ways. Think of an object-oriented system as an organic system, one that evolves as you create each new application. Applying design axioms (see Chapter 9) and carefully designed classes can have a synergistic effect, not only on the current system but on its future evolution. If you exercise some discipline as you proceed, you will begin to see some extraordinary gains in your productivity compared to a conventional approach.

## 10.3 UML OBJECT CONSTRAINT LANGUAGE

In Chapter 5, we learned that the UML is a graphical language with a set of rules and semantics. The rules and semantics of the UML are expressed in English, in a form known as *object constraint language*. ***Object constraint language*** (OCL) is a specification language that uses simple logic for specifying the properties of a system.

Many UML modeling constructs require expression; for example, there are expressions for types, Boolean values, and numbers. Expressions are stated as strings in object constraint language. The syntax for some common navigational expressions is shown here. These forms can be chained together. The leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable.

- *Item.selector*. The selector is the name of an attribute in the item. The result is the value of the attribute; for example, John.age (the age is an attribute of the object John, and John.age represents the value of the attribute).
- *Item.selector [qualifier-value]*. The selector indicates a qualified association that qualifies the item. The result is the related object selected by the qualifier; for example, array indexing as a form of qualification; for example, John.Phone[2], assuming John has several phones.
- *Set –> select (boolean-expression)*. The Boolean expression is written in terms of objects within the set. The result is the subset of objects in the set for which the Boolean expression is true; for example, company.employee –> salary > 30000. This represents employees with salaries over $30,000.



Other expressions will be covered as we study their appropriate UML notations. However, for more details and syntax, see UML OCL documents.

## 10.4 DESIGNING CLASSES: THE PROCESS

In Chapter 9, we looked at the object-oriented design process. In this section, we concentrate on step 1 of the process, which consists of the followings activities:

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols.
   - 1.1. Refine and complete the static UML class diagram by adding details to that diagram.
     - 1.1.1. Refine attributes.
     - 1.1.2. Design methods and the protocols by utilizing a UML activity diagram to represent the method's algorithm.
     - 1.1.3. Refine the associations between classes (if required).
     - 1.1.4. Refine the class hierarchy and design with inheritance (if required).
   - 1.2. Iterate and refine.

Object-oriented design is an iterative process. After all, design is as much about discovery as construction. Do not be afraid to change your class design as you gain experience, and do not be afraid to change it a second, third, or fourth time. At each iteration, you can improve the design. However, the trick is to correct the design flaws as early as possible; redesigning late in the development cycle always is problematic and may be impossible.

## 10.5 CLASS VISIBILITY: DESIGNING WELL-DEFINED PUBLIC, PRIVATE, AND PROTECTED PROTOCOLS

In designing methods or attributes for classes, you are confronted with two problems. One is the ***protocol***, or interface to the class operations and its visibility; and the other is how it is implemented. Often the two have very little to do with each other. For example, you might have a class Bag for collecting various objects that counts multiple occurrences of its elements. One implementation decision might be that the Bag class uses another class, say, Dictionary (assuming that we have a class Dictionary), to actually hold its elements. Bags and dictionaries have very little in common, so this may seem curious to the outside world. Implementation, by definition, is hidden and off limits to other objects. The class's protocol, or the messages that a class understands, on the other hand, can be hidden from other objects (private protocol) or made available to other objects (public protocol). Public protocols define the functionality and external messages of an object; private protocols define the implementation of an object (see Figure 10–1).

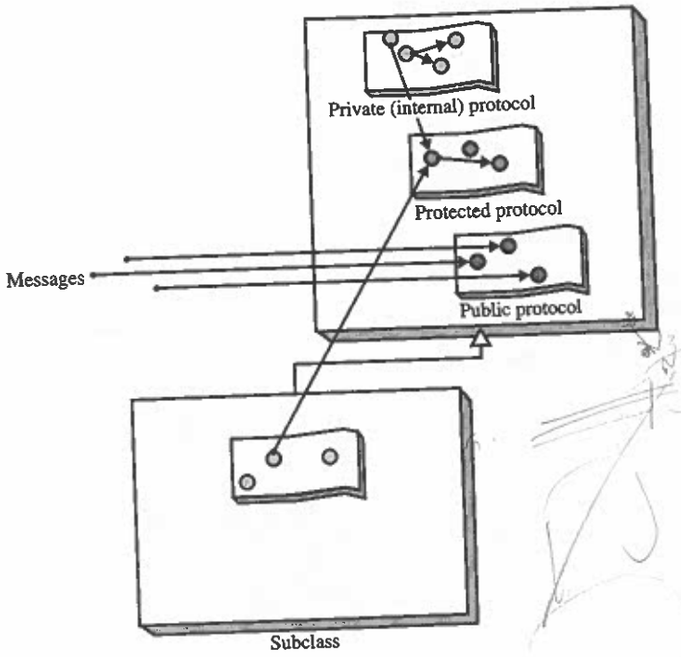



**FIGURE 10–1**
Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object.

It is important in object-oriented design to define the public protocol between the associated classes in the application. This is a set of messages that a class of a certain generic type must understand, although the interpretation and implementation of each message is up to the individual class.

A class also might have a set of methods that it uses only internally, messages to itself. This, the ***private protocol (visibility)*** of the class, includes messages that normally should not be sent from other objects; it is accessible only to operations of that class. In private protocol, only the class itself can use the method. The ***public protocol (visibility)*** defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes. If the methods or attributes can be used by the class itself or its subclasses, a protected protocol can be used. In a ***protected protocol (visibility)***, subclasses the can use the method in addition to the class itself.

Lack of a well-designed protocol can manifest itself as encapsulation leakage. The problem of ***encapsulation leakage*** occurs when details about a class's internal implementation are disclosed through the interface. As more internal details become visible, the flexibility to make changes in the future decreases. If an implementation is completely open, almost no flexibility is retained for future changes. It is fine to reveal implementation when that is intentional, necessary, and carefully controlled. However, do not make such a decision lightly because that could impact the flexibility and therefore the quality of the design.



For example, public or protected methods that can access private attributes can reveal an important aspect of your implementation. If anyone uses these functions and you change their location, the type of attribute, or the protocol of the method, this could make the client application inoperable.

Design the interface between a superclass and its subclasses just as carefully as the class's interface to clients; this is the contract between the super- and subclasses. If this interface is not designed properly, it can lead to violating the encapsulation of the superclass. The protected portion of the class interface can be accessed only by subclasses. This feature is helpful but cannot express the totality of the relationship between a class and its subclasses. Other important factors include which functions might or might not be overridden and how they must behave. It also is crucial to consider the relationship among methods. Some methods might need to be overridden in groups to preserve the class's semantics. The bottom line is this: Design your interface to subclasses so that a subclass that uses every supported aspect of that interface does not compromise the integrity of the public interface. The following paragraphs summarize the differences between these layers.

### 10.5.1 Private and Protected Protocol Layers: Internal

Items in these layers define the implementation of the object. Apply the design axioms and corollaries, especially Corollary 1 (uncoupled design with less information content, see Chapter 9) to decide what should be private: what attributes (instance variables)? What methods? Remember, highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

### 10.5.2 Public Protocol Layer: External

Items in this layer define the functionality of the object. Here are some things to keep in mind when designing class protocols:

- Good design allows for polymorphism.
- Not all protocol should be public; again apply design axioms and corollaries.

The following key questions must be answered:

- What are the class interfaces and protocols?
- What public (external) protocol will be used or what external messages must the system understand?
- What private or protected (internal) protocol will be used or what internal messages or messages from a subclass must the system understand?

## 10.6 DESIGNING CLASSES: REFINING ATTRIBUTES

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute was sufficient. However, in the design phase, detailed information must be added to the

11. ONTOS, Inc. "Object/Relational Integration: How to Use Objects to Enhance Your Relational Data." White paper, 1998.
12. Rob, Peter; and Coronel, Carlos. *Database Systems—Design, Implementation, and Management*, 2d ed. Belmont, CA: Wadsworth Publishing Company, 1997.
13. Robertson-Dunn, Bernard. comp.client-server FAQ, 1996.
14. Taylor, Lloyd. comp.client-server FAQ, 1996.
15. White, Set; Cattell, Rick; and Finkelstein, Shel. "Enterprise Java Platform Data Access." *Proceedings of ACM SIGMOD International Conference on Management of Data* 27, no. 2 (June 1998).

CHAPTER 12

# View Layer: Designing Interface Objects

### Chapter Objectives

You should be able to define and understand

- Identifying view classes.
- Designing interface objects.

## 12.1 INTRODUCTION

Once the analysis is complete (and sometimes concurrently), we can start designing the user interfaces for the objects and determining how these objects are to be presented. The main goal of a user interface (UI) is to display and obtain needed information in an accessible, efficient manner. The design of the software's interface, more than anything else, affects how a user interacts and therefore experiences an application [5]. It is important for a design to provide users the information they need and clearly tell them how to successfully complete a task. A well-designed UI has visual appeal that motivates users to use your application. In addition, it should use the limited screen space efficiently.

In this chapter, we learn how to design the view layer by mapping the UI objects to the view layer objects, we look at UI design rules based on the design corollaries, and finally, we look at the guidelines for developing a graphical user interface. A ***graphical user interface*** (GUI) uses icons to represent objects, a pointing device to select operations, and graphic imagery to represent relationships. See Appendix B for a review of Windows and graphical user interface basics and treatments.

## 12.2 USER INTERFACE DESIGN AS A CREATIVE PROCESS

Creative thinking is not confined to a particular field or a few individuals but is possessed in varying degrees by people in many occupations: The artist sketches, the journalist promotes an idea, the teacher encourages student development, the

BOX 12.1

## Real-World Issues on Agenda

### TOWARD AN OBJECT-ORIENTED USER INTERFACE

In the mid-1980s, mainstream PC software developers started making the move from character-based user interfaces such as DOS to graphical user interfaces (GUIs). We now face the next major shift in UI design, from GUI to OOUI (object-oriented user interface).[1] Like the last software design transition, the move to OOUI requires some rethinking about how to design software, not only from the development side but also from the human computer interface side.

Why objects? Tandy Trower, director of the Advanced User Interface group at Microsoft explains that using objects to express an interface is a natural choice because we interact with our environment largely through the manipulation of objects. Objects also allow the definition of a simple, common set of interactive conventions that can be applied consistently across the interface. For example, an object has properties, characteristics, or attributes that define its appearance or state, such as its color or size. Because objects, as large as a file or as small as a single character, can have properties, viewing and editing those properties can be generalized across the interface [5].

An ***object-oriented user interface*** focuses on objects, the "things" people use to accomplish their work. Users see and manipulate object representations of their information. Each different kind of object supports actions appropriate for the information it represents. Typical users need not be aware of computer programs and underlying computer technology [2].

While many of the concepts are similar, object-oriented programming (OOP) and object-oriented user interfaces are not the same thing. Simply using an object-oriented language does not guarantee an OOUI; as a matter of fact, you need not use an object-oriented language to create an OOUI, but it helps. Because the concepts involved are similar, the two disciplines can be used in a complementary relationship. The primary distinction to keep in mind is that OOUI design concentrates on the objects perceived by users, and object-oriented programming focuses on implementation details, which often need to be hidden from the user.

An OOUI allows a user to focus on objects and work with them directly, which more closely reflects the user's view of doing work. This is in contrast to the traditional application-oriented or current graphical user interfaces, where users must find a program appropriate for both the task they want to perform and the type of information they want to use, start the program, then use some mechanism provided by the program, such as an Open dialog, to locate their information and use it.

#### OOUI UNDER THE MICROSCOPE

An object-oriented user interface allows organizing objects in the computer environment similarly to how we organize objects in the real world. We can keep objects used in many tasks in a common, convenient place and objects used for specific tasks in more specific places.

UI objects typically are represented on a user's screen as icons. Icons are small graphic images that help a user identify an object. They typically consist of a picture that conveys the object's class and a text title that identifies the specific object. Icons are intended to provide a concise, easy-to-manipulate representation of an object regardless of how much additional information the object may contain. If desired, we can "open" an icon to see another view with this additional information. We can perform actions on icons using various techniques, such as point selecting, choosing an action from a menu, or dragging and dropping. Icons help depict the class of an object by providing a pictorial representation. For example, consider Windows 98 or its predecessor Windows 95, where you can click the right mouse button while selecting any object (icon) on the desktop, which will result in a menu popping up that gives access to the icon's properties and the operations possible on the icon.

Although we create and manipulate objects, many people never need to be consciously aware of the class to which an object belongs. For example, a person approaching a recliner need not stop and think, "This is a sofa, which belongs to the class chair. Therefore, I can sit on it." Likewise, a user can work with charts and come to expect that all charts will behave in the same way without caring that the charts are a subclass of the data object class.

UI classes also are very useful to you when designing an interface, because they force us to think about making clear distinctions among the classes of objects that should be provided the user. Classes must be carefully defined with respect to tasks and distinctions that users currently understand and that are useful. When the UI classes are carefully defined, these distinctions make it easy for users to learn the role of an object in performing their tasks and to predict how an object will behave.

In Chapter 2, we saw that most 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 such objects into the objects from which they are composed is decomposition. The depth to which object decomposition should be supported in the interface depends entirely on what a user finds useful in performing a particular task. A user writing a report, for example, probably would not be interested in dealing with objects smaller than characters, so in this task characters would be elemental objects. However, a user creating or editing a character font might need to manipulate individual pixels or strokes. In this task, characters would be composed of pixels or strokes, and therefore a character would not be an elemental object.

#### WHY OOUI?

An OOUI lessens the need for users to be aware of the programming providing the functions they employ. Instead, they can concentrate on locating the objects needed to accomplish their task and on performing actions on those objects. The aspects of starting and running programs are hidden to all but those users who want to be aware of them. A user should need to know only which objects are required to complete the task and how to use those objects to achieve the desired result [2]. The learning process is further simplified because the user has to deal with only one process, viewing an object, as opposed to starting an application, then finding and opening or creating a file. Although this is the main objective of OOUI, we are a few years away from completely achieving the goal. However, a computer is a tool, and as with any other tool, it has to be learned to be used effectively. Therefore, when you can help a user by simplifying the process of learning to use a tool, you should do so.

[1] However, currently we are in a transition phase between GUI and OOUI.



BOX 12.1 (CONTINUED)

scientist develops a theory, the manager implements a new strategy, and the programmer develops a new software system or improves an existing system to create a better one.

Creativity implies newness, but often it is concerned with the improvement of old products as much as with the creation of a new one. For example, newly created software must be useful, it should be of benefit to people, yet should not be so much of an innovation that others will not use it. A "how to make something better" attitude, tempered with good judgment, is an essential characteristic of an effective, creative process.

By bringing together, in the mind, various combinations of known objects or situations, we are using inventive imagination to develop new products, systems, or designs. It is not necessary to visualize absolutely new objects or to go beyond the bounds of our own experience. Inventive imagination can take place simply by putting together known materials (objects) in a new way. Therefore, a developer might conceive new software by using inventive imagination to combine objects already in his or her mind to satisfy user needs and requirements. As an example of this, see the Real-World Issues on Agenda "Toward an Object-Oriented User Interface."

Is creative ability born in an individual or can someone develop this ability? Both parts of this question can be answered in the affirmative. Certainly, some people are born with more creativity than others, just as certain people are born with better skills (athletes, artists, etc.) in some areas, than others. Just as it is possible to develop mental and physical skills through study and practice, it is possible to develop and improve one's creative ability.

To view user interface design as a creative process, it is necessary to understand what the creative process really involves. The creative process, in part, is a combination of the following:

1. A curious and imaginative mind.
2. A broad background and fundamental knowledge of existing tools and methods.
3. An enthusiastic desire to do a complete and thorough job of discovering solutions once a problem has been defined.
4. Being able to deal with uncertainty and ambiguity and to defer premature closure.

One aid to development or restoration of curiosity is to train yourself to be observant. You must be observant of any software that you are using. You must ask how or from what objects or components the user interface is made, how satisfied the users are with the UI, why it was designed using particular controls, why and how it was developed as it was, and how much it costs. These observations lead the creative thinker to see ways in which software can be improved or to devise a better component to take its place.

## 12.3 DESIGNING VIEW LAYER CLASSES

An implicit benefit of three-layer architecture and separation of the view layer from the business and access layers is that, when you design the UI objects, you have to think more explicitly about distinctions between objects that are useful to users. A distinguishing characteristic of view layer objects or interface objects is that they are the only exposed objects of an application with which users can interact. After all, view layer classes or interface objects are objects that represent the set of operations in the business that users must perform to complete their tasks, ideally in a way they find natural, easy to remember, and useful. Any objects that have direct contact with the outside world are visible in interface objects, whereas business or access objects are more independent of their environment.

As explained in Chapter 4, the view layer objects are responsible for two major aspects of the applications:

1. *Input—responding to user interaction.* The user interface must be designed to translate an action by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process. Remember, the business logic does not exist here, just the knowledge of which message to send to which business object.
2. *Output—displaying or printing business objects.* This layer must paint the best picture possible of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.



The process of designing view layer classes is divided into four major activities:

1. *The macro level UI design process—identifying view layer objects.* This activity, for the most part, takes place during the analysis phase of system development. The main objective of the macro process is to identify classes that interact with human actors by analyzing the use cases developed in the analysis phase. As described in previous chapters, each use case involves actors and the task they want the system to do. These use cases should capture a complete, unambiguous, and consistent picture of the interface requirements of the system. After all, use cases concentrate on describing what the system does rather than how it does it by separating the behavior of a system from the way it is implemented, which requires viewing the system from the user's perspective rather than that of the machine. However, in this phase, we also need to address the issue of how the interface must be implemented. Sequence or collaboration diagrams can help by allowing us to zoom in on the actor-system interaction and extrapolate interface classes that interact with human actors; thus, assisting us in identifying and gathering the requirements for the view layer objects and designing them.
2. *Micro level UI design activities:*
   - 2.1 *Designing the view layer objects by applying design axioms and corollaries.* In designing view layer objects, decide how to use and extend the components so they best support application-specific functions and provide the most usable interface.
   - 2.2 *Prototyping the view layer interface.* After defining a design model, prepare a prototype of some of the basic aspects of the design. Prototyping is particularly useful early in the design process.
3. *Testing usability and user satisfaction.* "We must test the application to make sure it meets the audience requirements. To ensure user satisfaction, we must measure user satisfaction and its usability along the way as the UI design takes form. Usability experts agree that usability evaluation should be part of the development process rather than a post-mortem or forensic activity. Despite the importance of usability and user satisfaction, many system developers still fail to pay adequate attention to usability, focusing primarily on functionality" [4, pp. 61–62]. In too many cases, usability still is not given adequate consideration. Adoption of usability in the later stages of the life cycle will not produce sufficient improvement of overall quality. We will study how to develop user satisfaction and usability in Chapter 14.
4. *Refining and iterating the design.*

## 12.4 MACRO-LEVEL PROCESS: IDENTIFYING VIEW CLASSES BY ANALYZING USE CASES

The interface object handles all communication with the actor but processes no business rules or object storage activities. In essence, the interface object will

operate as a buffer between the user and the rest of the business objects [3]. The interface object is responsible for behavior related directly to the tasks involving contact with actors. Interface objects are unlike business objects, which lie inside the business layer and involve no interaction with actors. For example, computing employee overtime is an example of a business object service. However, the data entry for the employee overtime is an interface object.

Jacobson, Ericsson, and Jacobson explain that an interface object can participate in several use cases. Often, the interface object has a coordinating responsibility in the process, at least responsibility for those tasks that come into direct contact with the user. As explained in earlier chapters, the first step here is to begin with the use cases, which help us to understand the users' objectives and tasks. Different users have different needs; for example, advanced, or "power," users want efficiency whereas other users may want ease of use. Similarly, users with disabilities or in an international market have still different requirements. The challenge is to provide efficiency for advanced users without introducing complexity for less-experienced users. However, developing use cases for advanced as well as less-experienced users might lead you to solutions such as shortcuts to support more advanced users.

The view layer macro process consists of two steps:

1. For every class identified (see Figure 12–1), *determine if the class interacts with a human actor*. If so, perform the following; otherwise, move to the next class.
   - 1.1 *Identify the view (interface) objects for the class*. Zoom in on the view objects by utilizing sequence or collaboration diagrams to identify the interface objects, their responsibilities, and the requirements for this class.
   - 1.2 *Define the relationships among the view (interface) objects*. The interface objects, like access classes, for the most part, are associated with the business classes. Therefore, you can let business classes guide you in defining the relationships among the view classes. Furthermore, the same rule as applies in identifying relationships among business class objects also applies among interface objects (see Chapter 8).
2. Iterate and refine.

The advantage of utilizing use cases in identifying and designing view layer objects is that the focus centers on the user, and including users as part of the planning and design is the best way to ensure accommodating them. Once the interface objects have been identified, we must identify the basic components or objects used in the user tasks and the behavior and the characteristics that differentiate each kind of object, including the relationships of interface objects to each other and to the user. Also identify the actions performed, the objects to which they apply, and the state information or attributes that each object in the task must preserve, display, and allow to be edited. Figure 12–2 shows the relationships among business, access, and view layer objects. The relationships among view class and business class objects is opposite of that among business class and access class objects. After all, the interface object handles all communication with the user but does not process any business rules; that will be done by the business objects.



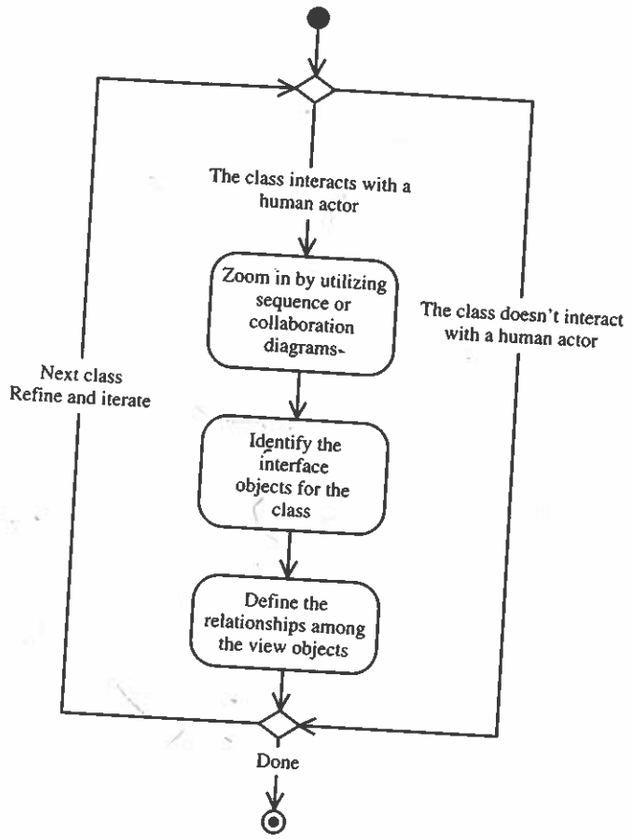



**FIGURE 12–1**
The macro-level design process.

Effective interface design is more than just following a set of rules. It also involves early planning of the interface and continued work through the software development process. The process of designing the user interface involves clarifying the specific needs of the application, identifying the use cases and interface objects, and then devising a design that best meets users' needs. The remainder of this chapter describes the micro-level UI design process and the issues involved.

## 12.5 MICRO-LEVEL PROCESS

To be successful, the design of the view layer objects must be user driven or user centered. A ***user-centered interface*** replicates the user's view of doing things by providing the outcomes users expect for any action. For example, if the goal of an

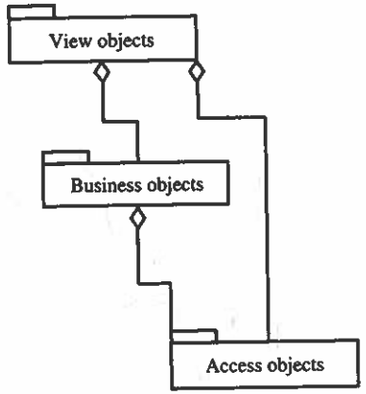



**FIGURE 12–2**
The relationships among business, access, and view objects. In some situations the view class can become a direct aggregate of the access object, as when designing a Web interface that must communicate with an application/Web server through access objects. See also Figure 11–18.

application is to automate what was a paper process, then the tool should be simple and natural. Design your application so it allows users to apply their previous real-world knowledge of the paper process to the application interface. Your design then can support this work environment and goal. After all, the main goal of view layer design is to address users' needs.

The following is the process of designing view (interface) objects:

1. For every interface object identified in the macro UI design process (see Figure 12–3), *apply micro-level UI design rules and corollaries to develop the UI.* Apply design rules and GUI guidelines to design the UI for the interface objects identified.
2. Iterate and refine.

In the following sections, we look at the three UI design rules based on the design axioms and corollaries of Chapter 9.

### 12.5.1 UI Design Rule 1. Making the Interface Simple (Application of Corollary 2)

First and foremost, your user interface should be so simple that users are unaware of the tools and mechanisms that make the application work. As applications become more complicated, users must have an even simpler interface, so they can learn new applications more easily. Today's car engines are so complex that they have onboard computers and sophisticated electronics. However, the driver interface remains simple: The driver needs only a steering wheel and the gas and brake pedals to operate a car. Drivers do not have to understand what is under the hood or even be aware of it to drive a car, because the driver interface remains simple. The UI should provide the same simplicity for users.



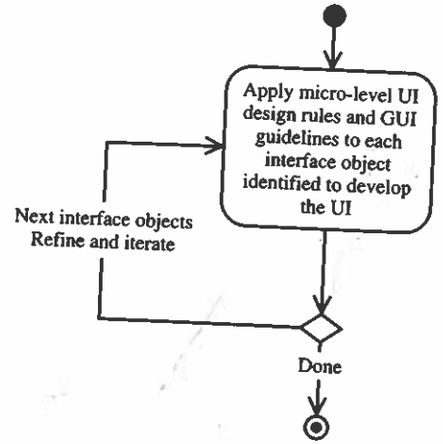



**FIGURE 12–3**
The micro-level design process.

This rule is an application of Corollary 2 (single purpose, see Chapter 9) in UI design. Here, it means that each UI class must have a single, clearly defined purpose. Similarly, when you document, you should be able easily to describe the purpose of the UI class with a few sentences. Furthermore, we have all heard the acronym KISS (Keep It Simple, Stupid). Maria Capucciati, an expert in user interface design and standards, has a better acronym for KISS—Keep It Simple and Straightforward. She says that, once you know what fields or choices to include in your application, ask yourself if they really are necessary. Labels, static text, check boxes, group boxes, and option buttons often clutter the interface and take up twice the room mandated by the actual data. If a user cannot sit before one of your screens and figure out what to do without asking a multitude of questions, your interface is not simple enough; ideally, in the final product, all the problems will have been solved.

A number of additional factors may affect the design of your application. For example, deadlines may require you to deliver a product to market with a minimal design process, or comparative evaluations may force you to consider additional features. Remember that additional features and shortcuts can affect the product. There is no simple equation to determine when a design trade-off is appropriate. So, in evaluating the impact, consider the following:

- Every additional feature potentially affects the performance, complexity, stability, maintenance, and support costs of an application.
- It is harder to fix a design problem after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.
- Simplicity is different from being simplistic. Making something simple to use often requires a good deal of work and code.
- Features implemented by a small extension in the application code do not necessarily have a proportional effect in a user interface. For example, if the primary

task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carry out. Designing a UI based on its purpose will be explained in the next section.

### 12.5.2 UI Design Rule 2. Making the Interface Transparent and Natural (Application of Corollary 4)

The user interface should be so intuitive and natural that users can anticipate what to do next by applying their previous knowledge of doing tasks without a computer. An application, therefore, should reflect a real-world model of the users' goals and the tasks necessary to reach those goals.

The second UI rule is an application of Corollary 4 (strong mapping) in UI design. Here, this corollary implies that there should be strong mapping between the user's view of doing things and UI classes. A ***metaphor***, or analogy, relates two otherwise unrelated things by using one to denote the other (such as a question mark to label a Help button). For example, writers use metaphors to help readers understand a conceptual image or model of the subject. This principle also applies to UI design. Using metaphors is a way to develop the users' conceptual model of an application. Familiar metaphors can assist the users to transfer their previous knowledge from their work environment to the application interface and create a strong mapping between the users' view and the UI objects. You must be careful in choosing a metaphor to make sure it meets the expectations users have because of their real-world experience. Often an application design is based on a single metaphor. For example, billing, insurance, inventory, and banking applications can represent forms that are visually equivalent to the paper forms users are accustomed to seeing.

The UI should not make users focus on the mechanics of an application. A good user interface does not bother the user with mechanics. Computers should be viewed as a tool for completing tasks, as a car is a tool for getting from one place to another. Users should not have to know how an application works to get a task done, as they should not have to know how a car engine works to get from one place to another. A goal of user interface design is to make the user interaction with the computer as simple and natural as possible.

### 12.5.3 UI Design Rule 3. Allowing Users to Be in Control of the Software (Application of Corollary 1)

The third UI design rule states that the users always should feel in control of the software, rather than feeling controlled by the software. This concept has a number of implications. The first implication is the operational assumption that actions are started by the user rather than the computer or software, that the user plays an active rather than reactive role. Task automation and constraints still are possible, but you should implement them in a balanced way that allows the user freedom of choice.

The second implication is that users, because of their widely varying skills and preferences, must be able to customize aspects of the interface. The system software provides user access to many of these aspects. The software should reflect user settings for different system properties such as color, fonts, or other options.



The final implication is that the software should be as interactive and responsive as possible. Avoid modes whenever possible. A ***mode*** is a state that excludes general interaction or otherwise limits the user to specific interactions. Users are in control when they are able to switch from one activity to another, change their minds easily, and stop activities they no longer want to continue. Users should be able to cancel or suspend any time-consuming activity without causing disastrous results. There are situations in which modes are useful; for example, selecting a file name before opening it. The dialog that gets me the file name must be modal (more on this later in the section).

This rule is a subtle but important application of Corollary 1 (uncoupled design with less information content) in UI design. It implies that the UI object should represent, at most, one business object, perhaps just some services of that business object. The main idea here is to avoid creating a single UI class for several business objects, since it makes the UI less flexible and forces the user to perform tasks in a monolithic way. Some of the ways to put users in control are these:

- Make the interface forgiving.
- Make the interface visual.
- Provide immediate feedback.
- Avoid modes.
- Make the interface consistent.

**12.5.3.1 Make the Interface Forgiving** The users' actions should be easily reversed. When users are in control, they should be able to explore without fear of causing an irreversible mistake. Users like to explore an interface and often learn by trial and error. They should be able to back up or undo previous actions. An effective interface allows for interactive discovery. Actions that are destructive and may cause the unexpected loss of data should require a confirmation or, better, should be reversible or recoverable. Even within the best designed interface, users can make mistakes. These mistakes can be both physical (accidentally pointing to the wrong command or data) and mental (making a wrong decision about which command or data to select). An effective design avoids situations that are likely to result in errors. It also accommodates potential user errors and makes it easy for the user to recover. Users feel more comfortable with a system when their mistakes do not cause serious or irreversible results.

**12.5.3.2 Make the Interface Visual** Design the interface so users can see, rather than recall, how to proceed. Whenever possible, provide users a list of items from which to choose, instead of making them remember valid choices.

**12.5.3.3 Provide Immediate Feedback** Users should never press a key or select an action without receiving immediate visual or audible feedback or both. When the cursor is on a choice, for example, the color, emphasis, and selection indicators show users they can select that choice. After users select a choice, the color, emphasis, and selection indicators change to show users their choice is selected.

**12.5.3.4 Avoid Modes** Users are in a mode whenever they must cancel what they are doing before they can do something else or when the same action has different results in different situations. Modes force users to focus on the way an application works, instead of on the task they want to complete. Modes, therefore, interfere with users' ability to use their conceptual model of how the application should work. It is not always possible to design a modeless application; however, you should make modes an exception and limit them to the smallest possible scope. Whenever users are in a mode, you should make it obvious by providing good visual cues. The method for ending the mode should be easy to learn and remember. These are some of the modes that can be used in the user interface:

- *Modal dialog*. Sometimes an application needs information to continue, such as the name of a file into which users want to save something. When an error occurs, users may be required to perform an action before they continue their task. The visual cue for modal dialog is a color boundary for the dialog box that contains the modal dialog.
- *Spring-loaded modes*. Users are in a ***spring-loaded mode*** when they continually must take some action to remain in that mode; for example, dragging the mouse with a mouse button pressed to highlight a portion of text. In this case, the visual cue for the mode is the highlighting, and the text should stay highlighted for other operations such as Cut and Paste.
- *Tool-driven modes*. If you are in a drawing application, you may be able to choose a tool, such as a pencil or a paintbrush, for drawing. After you select the tool, the mouse pointer shape changes to match the selected tool. You are in a mode, but you are not likely to be confused because the changed mouse pointer is a constant reminder you are in a mode.

**12.5.3.5 Make the Interface Consistent** Consistency is one way to develop and reinforce the user's conceptual model of applications and give the user the feeling that he or she is in control, since the user can predict the behavior of the system. User interfaces should be consistent throughout the applications; for example, using a consistent user interface for the inventory application.

## 12.6 THE PURPOSE OF A VIEW LAYER INTERFACE

Your user interface can employ one or more windows. Each window should serve a clear, specific purpose. Windows commonly are used for the following purposes:

- *Forms and data entry windows*. ***Data entry windows*** provide access to data that users can retrieve, display, and change in the application.
- *Dialog boxes*. Dialog boxes display status information or ask users to supply information or make a decision before continuing with a task. A typical feature of a dialog box is the OK button that a user clicks with a mouse to process the selected choices.
- *Application windows (main windows)*. An ***application window*** is a container of application objects or icons. In other words, it contains an entire application with which users can interact.



You should be able to explain the purpose of a window in the application in a single sentence. If a window serves multiple purposes, consider creating a separate one for each.

### 12.6.1 Guidelines for Designing Forms and Data Entry Windows

When designing a data entry window or forms (or Web forms), identify the information you want to display or change. Consider the following issues:

- In general, what kind of information will users work with and why? For example, a user might want to change inventory information, enter orders, or maintain prices for stock items.
- Do users need access to all the information in a table or just some information? When working with a portion of the information in a table, use a query that selects the rows and columns users want.
- In what order do users want rows to appear? For example, users might want to change inventory information stored alphabetically, chronologically, or by inventory number. You have to provide a mechanism for the user so that the order can be modified.

Next, identify the tasks that users need to work with data on the form or data entry window. Typical data entry tasks include the following:

- Navigating rows in a table, such as moving forward and backward, and going to the first and last record.
- Adding and deleting rows.
- Changing data in rows.
- Saving and abandoning changes.

You can provide menus, push buttons, and speed bar buttons that users choose to initiate tasks. You can put controls anywhere on a window. However, the layout you choose determines how successfully users can enter data using the form. Here are some guidelines to consider:

- You can use an existing paper form, such as a printed invoice, as the starting point for your design.
- If the printed form contains too much information to fit on a screen, consider using a main window with optional smaller windows that users can display on demand or using a window with multiple pages (see Figure 12–4). Users typically are more productive when a screen is not cluttered.
- Users scan a screen in the same way they read a page of a book, from left to right and top to bottom. In general, put required or frequently entered information toward the top and left side of the form, entering optional or seldom-entered information toward the bottom and right side. For example, on a window for entering inventory data, the inventory number and item name might best be placed in the upper-left corner, while the signature could appear lower and to the right (see Figure 12–5).
- When information is positioned vertically, align fields at their left edges (in Western countries). This usually makes it easier for the user to scan the information.

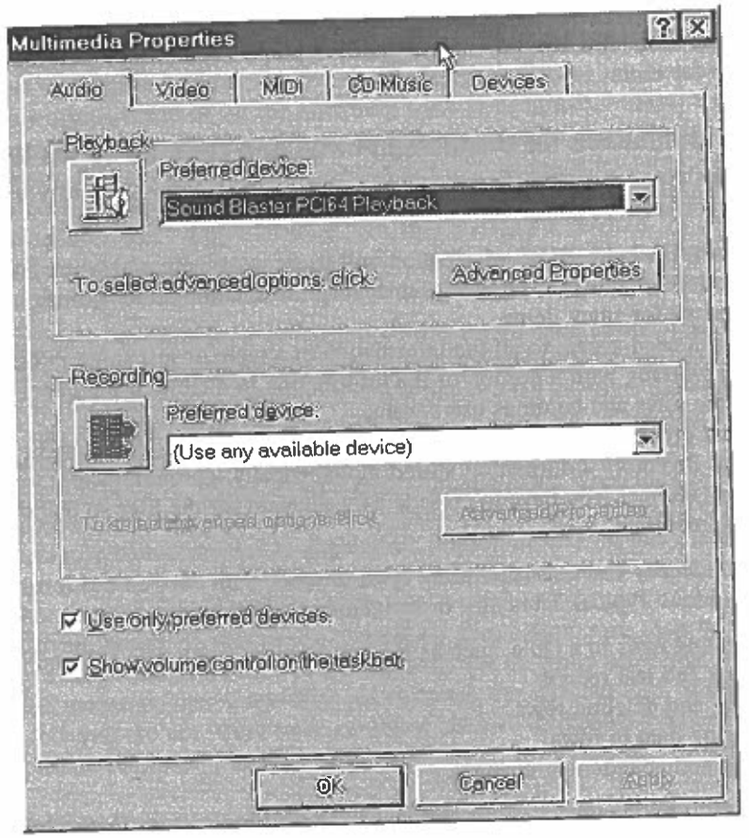



**FIGURE 12–4**
An example of a dialog box with multiple pages in the Microsoft multimedia setup.

Text labels usually are left aligned and placed above or to the left of the areas to which they apply. When placing text labels to the left of text box controls, align the height of the text with text displayed in the text box (see Figure 12–6).

- When entering data, users expect to type information from left to right and top to bottom, as if they were using a typewriter (usually the Tab key moves the focus from one control to another). Arrange controls in the sequence users expect to enter data. However, you may want the users to be able to jump from one group of controls to the beginning of another group, skipping over individual controls. For example, when entering address information, users expect to enter the Address, City, State, and Zip Code (see Figure 12–7).
- Put similar or related information together, and use visual effects to emphasize the grouping. For example, you might want to put a company's billing and shipping address information in separate groups. To emphasize a group, you can enclose its controls in a distinct visual area using a rectangle, lines, alignment, or colors (see Figure 12–4).



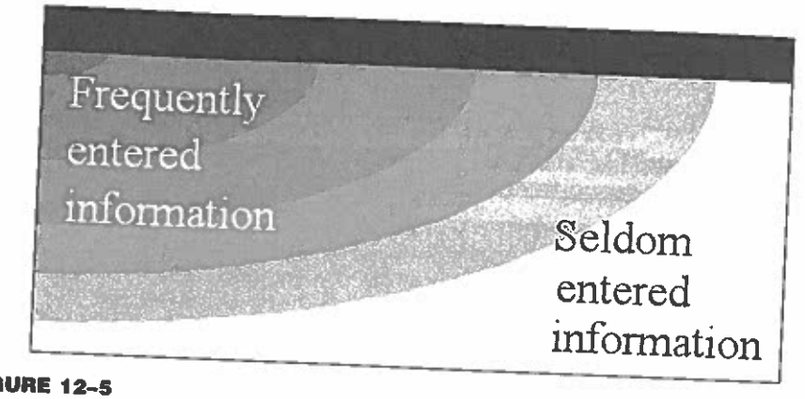
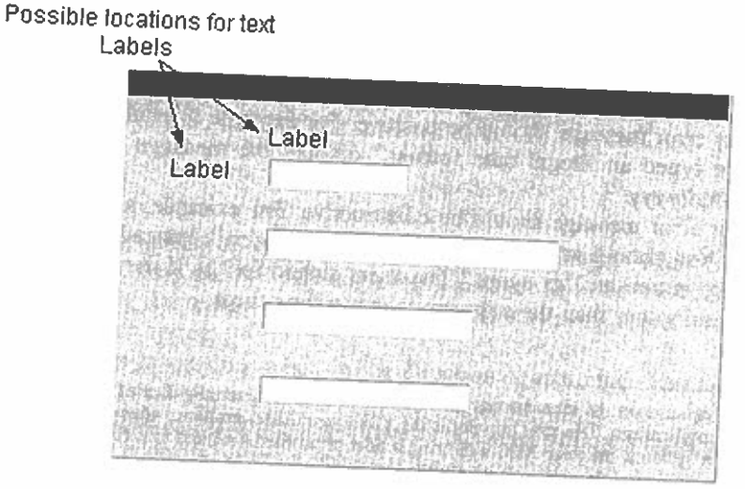



**FIGURE 12–5**
Required information should be put toward the top and left side of the form, entering optional or seldom entered information toward the bottom.

**FIGURE 12–6**
Place text labels to the left of text box controls, align the height of the text with text displayed in the text box.



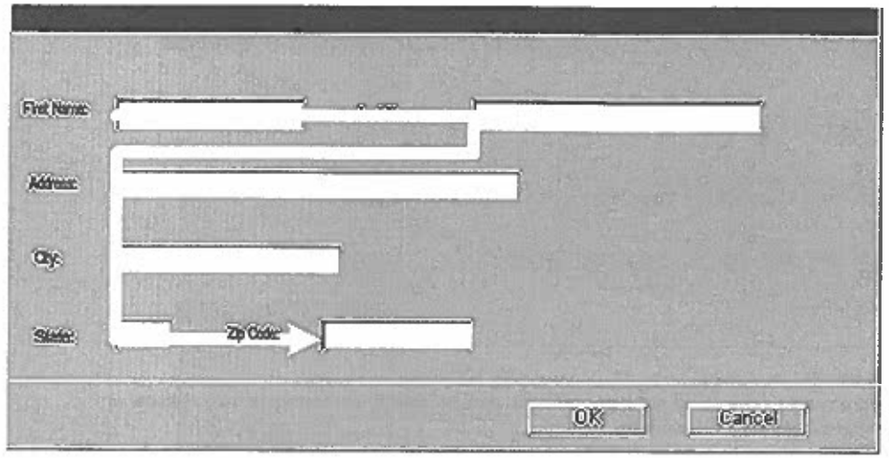



**FIGURE 12–7**
Arrange controls left to right and top to bottom.

The Real-World Issues on Agenda "Future of the GUI Landscape" examines window presentations for the future.

### 12.6.2 Guidelines for Designing Dialog Boxes and Error Messages

A dialog box provides an exchange of information or a dialog between the user and the application. Because dialog boxes generally appear after a particular menu item (including pop-up or cascading menu items) or a command button, define the title text to be the name of the associated command from the menu item or command button. However, do not include ellipses and avoid including the command's menu title unless necessary to compose a reasonable title for the dialog box. For example, for a Print command on the File Menu, define the dialog box window's title text as Print, Not Print . . ., or File Print.

If the dialog box is for an error message, use the following guidelines:

- Your error message should be positive. For example instead of displaying "You have typed an illegal date format," display the message "Enter date format mm/dd/yyyy."[2]
- Your error message should be constructive. For example, avoid messages such as "You should know better! Use the OK button"; instead display "Press the Undo button and try again." The users should feel as if they are controlling the system rather than the software is controlling them.

[2] Note: Sometimes, an innocent design decision (such as representing date as mm/dd/yy) can have immense implications. The case in point is the Y2K (year 2000) problem, where for many computer and software systems, the year 2000 will bring a host of problems related to software programs that were designed to record the year using only the last two digits.



BOX 12.2

## Real-World Issues on Agenda

### FUTURE OF THE GUI LANDSCAPE: 3-D OR FLATLAND?

*Stephanie Wilkinson*

Not so long ago, using icons, windows and drop-down menus to navigate applications was a radical idea for corporate systems builders. Today, that GUI is all but ubiquitous. But what will the GUI of tomorrow look like?

If the visionaries had their way, corporate PC users would interact with their computers in a wholly naturalistic way. They'd never need help screens to explain an icon or find a file or launch an application. Everything would appear "virtually real." Everything would be three-dimensional.

"The GUI interface of today is a vast two-dimensional flatland," says John Latta, president of 4th Wave Inc., an Alexandria, Va., research firm. "3-D is a portal to the next generation."

3-D isn't just for games anymore. Corporate IT departments are awakening to the power of data visualization, next-generation GUIs and of course, the lure of the Web. Here's the business justification for going 3-D:

*Because 3-D environments are more like real life,* workers can perform tasks more easily and with less training. 3-D interfaces trade cognitive effort for simple perception: instead of having to mull over how to attach a document to a memo using commands or icons, the user chooses a stapler on the desktop.

*More information can be presented—and understood*—in a 3-D format than in 2-D. Example: An 80-page organizational chart can be represented in 3-D on a single screen. The hierarchical and lateral relationships between departments and employees are also instantly apparent.

*3-D ratchets up the power of data mining a full notch.* By using 3-D, commonplace data matrix—national widget sales in seven regions over the last three quarters, for instance—can be transformed into a full-color, animated map that allows hidden trends to emerge.

"The average office worker has to deal with vast amounts of information, most of which is not well-organized," says Robertson. A 3-D interface not only allows users to see more on screen at once, "they also see the structure of that information," he notes. For instance, the contents of a user's hard drive could appear in 3-D space, allowing the user to locate files and launch applications by zooming in on—or "foregrounding"—a particular part of the scene.

Of course, analysts such as Latta say there is no guarantee that what comes from Microsoft will become the next GUI standard. Xerox PARC itself is working on a 3-D interface technology that will eventually result in a commercial version called WebForager. And Intel Corp., which has a vision of how the task of graphics processing should take place inside the box, is readying its own set of APIs.

So there's no need to worry quite yet about choosing the next corporate GUI and making the transition to 3-D on every desktop. Says Latta: "That's probably still a few years away."

By Stephanie Wilkinson, *PC Week*, September 23, 1996, Vol. 13, Number 38.

- Your error message should be brief and meaningful. For example, "*ERROR*: type check *Offending Command* . . ." Although this message might be useful for the programmer during the testing and debugging phase, it is not a useful message for the user of your system.
- Orient the controls in the dialog box in the direction people read. This usually means left to right and top to bottom. Locate the primary field with which the user interacts as close to the upper-left corner as possible. Follow similar guidelines for orienting controls within a group in the dialog box.

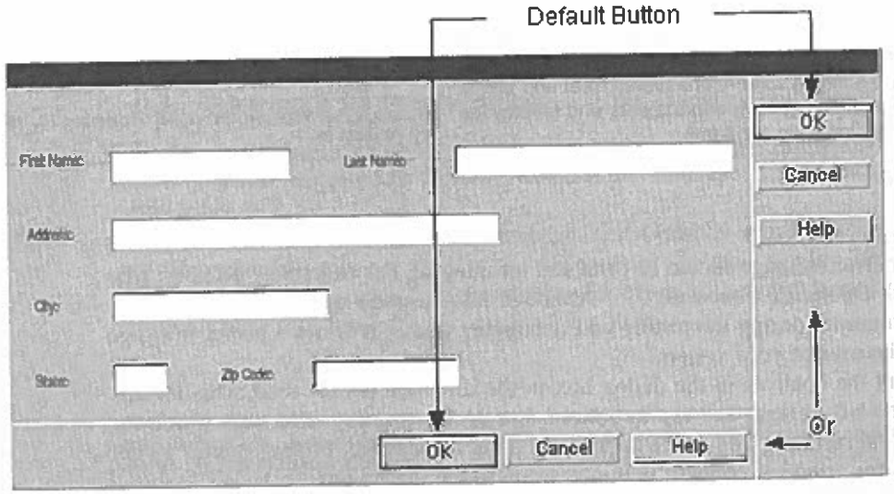
### 12.6.3 Guidelines for the Command Buttons Layout

Lay out the major command buttons either stacked along the upper-right border of the dialog box or lined up across the bottom of the dialog box (see Figure 12–8). Positioning buttons on the left border is very popular in Web interfaces (see Figure 12–9). Position the most important button, typically the default command, as the first button in the set. If you use the OK and Cancel buttons, group them together. If you include a Help command button, make it the last button in the set.

You can use other arrangements if there is a compelling reason, such as a natural mapping relationship. For example, it makes sense to place buttons labeled North, South, East, and West in a compasslike layout. Similarly, a command button that modifies or provides direct support for another control may be grouped or placed next to that control. However, avoid making this button the default button because the user will expect the default button to be in the conventional location. Once again, let consistency guide you through the design.

For easy readability, make buttons a consistent length. Consistent visual and operational styles will allow users to transfer their knowledge and skills more easily. However, if maintaining this consistency greatly expands the space required by a set of buttons, it may be reasonable to have one button larger than the rest. Placement of command buttons (or other controls) within a tabbed page implies the application of only the transactions on that page. If command buttons are placed within the window but not on the tabbed page, they apply to the entire window (see Figure 12–4).

**FIGURE 12–8**
Arrange the command buttons either along the upper-right border of the form or dialog box or lined up across the bottom.





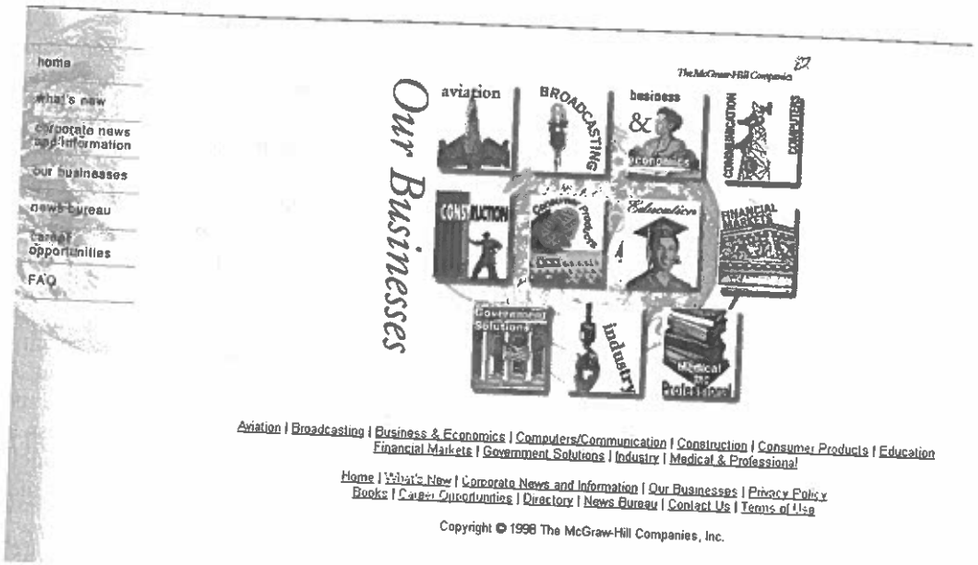




**FIGURE 12–9**
Positioning buttons on the left is popular in Web interfaces.

### 12.6.4 Guidelines for Designing Application Windows

A typical application window consists of a frame (or border) that defines its extent and a title bar that identifies what is being viewed in the window. If the viewable content of the window exceeds the current size of the window, scroll bars are used. The window also can include other components like menu bars, toolbars, and status bars.

An application window usually contains common drop-down menus. While a command drop-down menu is not required for all applications, apply these guidelines when including such menus in your software's interface:

- *The File menu.* The File menu provides an interface for the primary operations that apply to a file. Your application should include commands such as Open, Save, Save As . . ., and Print. Place the Exit command at the bottom of the File menu preceded by a menu separator. When the user chooses the Exit command, close any open windows and files and stop any further processing. If the object remains active even when its window is closed, such as a folder or printer, then include the Close command instead of Exit.
- *The Edit menu.* Include general purpose editing commands on the Edit menu. These commands include the Cut, Copy, and Paste commands. Depending on your application, you might include the Undo, Find, and Delete commands.

- *The View menu and other command menus*. Commands on the View menu should change the user's view of data in the window. On this menu, include commands that affect the view and not the data itself; for example, Zoom or Outline. Also include commands for controlling the display of particular interface elements in the view; for example, Show Ruler. These commands should be placed on the pop-up menu of the window.
- *The Window menu*. Use the Window menu in multiple document, interface-style applications for managing the windows within the main workspace.
- *The Help menu*. The Help menu contains commands that provide access to Help information. Include a Help Topics command. This command provides access to the Help Topics browser, which displays topics included in the application's Help file. Alternatively, provide individual commands that access specific pages of the Help Topics browser, such as Contents, Index, and Find Topic. Also include other user assistance commands or wizards that can guide the users and show them how to use the system. It is conventional to provide access to copyright and version information for the application, which should be included in the About Application name command on this menu. Other command menus can be added, depending on your application's needs.
- *Toolbars and status bars*. Like menu bars, toolbars and status bars are special interface constructs for managing sets of controls. A toolbar is a panel that contains a set of controls, as shown in Figure 12–10, designed to provide quick access to specific commands or options. Some specialized toolbars are called *ribbons*, *toolboxes*, and *palettes*. A status bar, shown in Figure 12–11, is a special area within a window, typically at the bottom, that displays information such as the current state of what is being viewed in the window or any other contextual information, such as keyboard state. You also can use the status bar to provide descriptive messages about a selected menu or toolbar button, and it provides excellent feedback to the users. Like a toolbar, a status bar can contain controls; however, typically, it includes read-only or noninteractive information.

### 12.6.5 Guidelines for Using Colors

For all objects on a window, you can use colors to add visual appeal to the form. However, consider the hardware. Your Windows-based application may end up being run on just about any sort of monitor. Do not choose colors exclusive to a particular configuration, unless you know your application will be run on that specific hardware. In fact, do not dismiss the possibility that a user will run your application with no color support at all.

Figure out a color scheme. If you use multiple colors, do not mix them indiscriminately. Nothing looks worse than a circus interface [1]. Do you have a good color sense? If you cannot make everyday color decisions, ask an artist or a designer to review your color scheme. Use color as a highlight to get attention. If there is one field you want the user to fill first, color it in such a way that it will stand out from the other fields.



**FIGURE 12–10**
Toolbar.


**FIGURE 12–11**
Status bar.

How long will users be sitting in front of your application? If it is eight hours a day, this is not the place for screaming red text on a sunny yellow background. Use common sense and consideration. Go for soothing, cool, and neutral colors such as blues or other neutral colors. Text must be readable at all times; black is the standard color, but blue and dark gray also can work.

Associate meanings to the colors of your interface. For example, use blue for all the uneditable fields, green to indicate fields that will update dynamically, and red to indicate error conditions. If you choose to do this, ensure color consistency from screen to screen and make sure the users know what these various colors indicate. Do not use light gray for any text except to indicate an unavailable condition.

Remember that a certain percentage of the population is color blind. Do not let color be your only visual cue. Use an animated button, a sound package, or a message box. Finally, color will not hide poor functionality.

The following guidelines can help you use colors in the most effective manner:

- You can use identical or similar colors to indicate related information. For example, savings account fields might appear in one color. Use different or contrasting colors to distinguish groups of information from each other. For example, checking and savings accounts could appear in different colors.
- For an object background, use a contrasting but complementary color. For example, in an entry field, make sure that the background color contrasts with the data color so that the user can easily read data in the field.
- You can use bright colors to call attention to certain elements on the screen, and you can use dim colors to make other elements less noticeable. For example, you might want to display the required field in a brighter color than optional fields.
- Use colors consistently within each window and among all windows in your application. For example, the colors for push buttons should be the same throughout.
- Using too many colors can be visually distracting and will make your application less interesting.
- Allow the user to modify the color configuration of your application.

### 12.6.6 Guidelines for Using Fonts

Consistency is the key to an effective use of fonts and color in your interface. Most commercial applications use 12-point System font for menus and 10-point System font in dialog boxes. These are fairly safe choices for most purposes. If System is too boring for you, any other sans serif font is easy to read (such as Arial or Helvetica). The most practical serif font is Times New Roman.

Avoid Courier unless you deliberately want something to look like it came from a typewriter. Other fonts may be appropriate for word processing or desktop publishing purposes but do not really belong on Windows-based application screens. Avoid using all uppercase text in labels or any other text on your screens: It is harder to read and feels like you are shouting at the users. The only exception is the OK command button. Also avoid mixing more than two fonts, point sizes, or styles, so your screens have a cohesive look. The following guidelines can help you use fonts to best convey information:

- Use commonly installed fonts, not specialized fonts that users might not have on their machines.
- Use bold for control labels, so they will remain legible when the object is dimmed.
- Use fonts consistently within each form and among all forms in your application. For example, the fonts for check box controls should be the same throughout. Consistency is reassuring to users, and psychologically makes users feel in control.
- Using too many font styles, sizes, and colors can be visually distracting and should be avoided. Too many font styles are confusing and make users feel less in control.
- To emphasize text, increase its font size relative to other words on the form or use a contrasting color. Avoid underlines; they can be confusing and difficult to read on the screen.

## 12.7 PROTOTYPING THE USER INTERFACE

Rapid prototyping encourages the incremental development approach, "grow, don't build." Prototyping involves a number of iterations. Through each iteration, we add a little more to the application, and as we understand the problem a little better, we can make more improvements. This, in turn, makes the debugging task easier.

It is highly desirable to prepare a prototype of the user interface during the analysis to better understand the system requirements. This can be done with most CASE tools,[3] operational software using visual prototyping, or normal development tools. Visual and rapid prototyping is a valuable asset in many ways. First, it provides an effective tool for communicating the design. Second, it can help you define task flow and better visualize the design. Finally, it provides a low-cost vehicle for getting user input on a design. This is particularly useful early in the design process.

[3] System Architect Screen Painter can be used to prototype Windows screens and menus.



Creating a user interface generally consists of three steps (see Figure 12–12):

1. Create the user interface objects (such as buttons, data entry fields).
2. Link or assign the appropriate behaviors or actions to these user interface objects and their events.
3. Test, debug, then add more by going back to step 1.

**FIGURE 12–12**
Prototyping user interface consists of three steps.