PART FOUR

OBJECT-ORIENTED DESIGN

During the design phase, we must elevate the model into actual objects that can perform the required task. There is a shift in emphasis from the application domain to implementation. The classes identified during analysis provide us a framework for the design phase. In this part, we discuss business, view, and access layers classes. The part consists of Chapters 9, 10, 11, and 12.

CHAPTER 9

The Object-Oriented Design Process and Design Axioms

Chapter Objectives

You should be able to define and understand

- The object-oriented design process.
- Object-oriented design axioms and corollaries.
- Design patterns.

9.1 INTRODUCTION

It was explained in previous chapters that the main focus of the analysis phase of software development is on "what needs to be done." The objects discovered during analysis can serve as the framework for design [9]. The class's attributes, methods, and associations identified during analysis must be designed for implementation as a data type expressed in the implementation language. New classes must be introduced to store intermediate results during program execution. Emphasis shifts from the application domain to implementation and computer concepts such as user interfaces or view layer and access layer (see Figures 1–11 and 4–11).

During the analysis, we look at the physical entities or business objects in the system; that is, who the players are and how they cooperate to do the work of the application. These objects represent tangible elements of the business. As we saw in Chapter 7, these objects could be individuals, organizations, machines, or whatever else makes sense in the context of the real-world system. During the design phase, we elevate the model into logical entities, some of which might relate more to the computer domain (such as user interfaces or the access layer) than the real-world or the physical domain (such as people or employees). This is where we begin thinking about how to actually implement the problem in a program. The goal here is to design the classes that we need to implement the system. Fortunately,

the design model does not look terribly different from the analysis model. The difference is that, at this level, we focus on the view and access classes, such as how to maintain information or the best way to interact with a user or present information. It also is useful, at this stage, to have a good understanding of the classes in a development environment that we are using to enforce reusability.

In software development, it is tempting not to be concerned with design. After all, you (the designer) are so involved with the system that it might be difficult to stop and think about the consequences of each design choice. However, the time spent on design has a great impact on the overall success of the software development project. A large payoff is associated with creating a good design "up front," before writing a single line of code. While this is true of all programming, classes and objects underscore the approach even more. Good design usually simplifies the implementation and maintenance of a project.

In this chapter, we look at the object-oriented design process and *axioms*. The basic goal of the axiomatic approach is to formalize the design process and assist in establishing a scientific foundation for the object-oriented design process, to provide a fundamental basis for the creation of systems. Without scientific principles, the design field never will be systematized and so will remain a subject difficult to comprehend, codify, teach, and practice [10].

9.2 THE OBJECT-ORIENTED DESIGN PROCESS

During the design phase the classes identified in object-oriented analysis must be revisited with a shift in focus to their implementation. New classes or attributes and methods must be added for implementation purposes and user interfaces.

The object-oriented design process consists of the following activities (see Figure 9–1):

1. Apply design axioms to design classes, their attributes, methods, associations, structures, and protocols (Chapter 10).
   - **1.1.** Refine and complete the static UML class diagram by adding details to the UML class diagram. This step consists of the following activities:
     - **1.1.1.** Refine attributes.
     - **1.1.2.** Design methods and protocols by utilizing a UML activity diagram to represent the method's algorithm.
     - **1.1.3.** Refine associations between classes (if required).
     - **1.1.4.** Refine class hierarchy and design with inheritance (if required).
   - **1.2.** Iterate and refine again.
2. Design the access layer (Chapter 11).
   - **2.1.** *Create mirror classes.* For every business class identified and created, create one access class. For example, if there are three business classes (Class1, Class2, and Class3), create three access layer classes (Class1DB, Class2DB, and Class3DB).
   - **2.2.** *Identify access layer class relationships.*

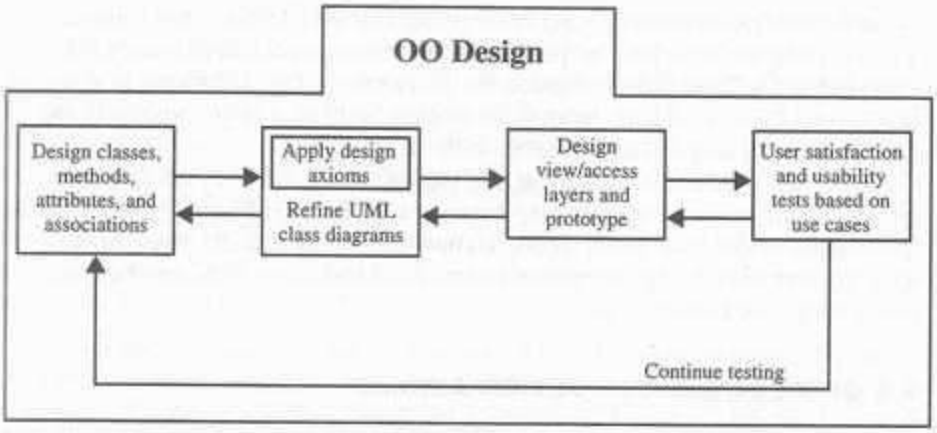


FIGURE 9–1
The object-oriented design process in the unified approach.

2.3. *Simplify classes and their relationships*. The main goal here is to eliminate redundant classes and structures.
   2.3.1. Redundant classes: Do not keep two classes that perform similar *translate request* and *translate results* activities. Simply select one and eliminate the other.
   2.3.2. Method classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
2.4. Iterate and refine again.

3. Design the view layer classes (Chapter 12).
   3.1. Design the macro level user interface, identifying view layer objects.
   3.2. Design the micro level user interface, which includes these activities:
      3.2.1. Design the view layer objects by applying the design axioms and corollaries.
      3.2.2. Build a prototype of the view layer interface.
   3.3. Test usability and user satisfaction (Chapters 13 and 14).
   3.4. Iterate and refine.
4. Iterate and refine the whole design. Reapply the design axioms and, if needed, repeat the preceding steps.

Utilizing an incremental approach such as the UA, all stages of software development (analysis, modeling, designing, and implementation or programming) can be performed incrementally. Therefore, all the right decisions need not be made up front.

From the UML class diagram, you can begin to extrapolate which classes you will have to build and which existing classes you can reuse. As you do this, also begin thinking about the inheritance structure. If you have several classes that seem related but have specific differences, you probably will want to make them

common subclasses of an existing class or one that you define. Often, superclasses are generated while coding, as you realize that common characteristics can be factored out or in. Good object-oriented design is very iterative. As long as you think in terms of class of objects, learn what already is there, and are willing to experiment, you soon will feel comfortable with the process.

Design also must be traceable across requirements, analysis, design, code, and testing. There must be a clear step-by-step approach to the design from the requirements model. All the designed components must directly trace back to the user requirements. Usage scenarios can serve as test cases to be used during system testing (see Figure 1–1).

9.3 OBJECT-ORIENTED DESIGN AXIOMS

By definition, an ***axiom*** is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. Suh explains that axioms may be hypothesized from a large number of observations by noting the common phenomena shared by all cases; they cannot be proven or derived, but they can be invalidated by counterexamples or exceptions. A ***theorem*** is a proposition that may not be self-evident but can be proven from accepted axioms. It, therefore, is equivalent to a law or principle. Consequently, a theorem is valid if its referent axioms and deductive steps are valid. A ***corollary*** is a proposition that follows from an axiom or another proposition that has been proven. Again, a corollary is shown to be valid or not valid in the same manner as a theorem [10].

The author has applied Suh's design axioms to object-oriented design. Axiom 1 deals with relationships between system components (such as classes, requirements, and software components), and Axiom 2 deals with the complexity of design.

- Axiom 1. *The independence axiom*. Maintain the independence of components.
- Axiom 2. *The information axiom*. Minimize the information content of the design.

Axiom 1 states that, during the design process, as we go from requirement and use case to a system component, each component must satisfy that requirement without affecting other requirements. To make this point clear, let's take a look at an example offered by Suh [10]. You been asked to design a refrigerator door, and there are two requirements: The door should provide access to food, and the energy lost should be minimal when the door is opened and closed. In other words, opening the door should be independent of losing energy. Is the vertically hung door a good design? We see that vertically hung door violates Axiom 1, because the two specific requirements (i.e., access to the food and minimal energy loss) are coupled and are not independent in the proposed design. When, for example, the door is opened to take out milk, cold air in the refrigerator escapes and warm air from the outside enters. What is an uncoupled design that somehow does not combine these two requirements? Once such uncoupled design of the refrigerator door is a horizontally hinged door, such as used in chest-type freezers. When the door is opened to take out milk, the cold air (since it is heavier than warm air) will sit at the bot-

tom and not escape. Therefore, opening the door provides access to the food and is independent of energy loss. This type of design satisfies the first axiom.

Axiom 2 is concerned with simplicity. Scientific theoreticians often rely on a general rule known as *Occam's razor*, after William of Occam, a 14th century scholastic philosopher. Briefly put, Occam's razor says that, "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness."

Occam's razor has a very useful implication in approaching the design of an object-oriented application. Let us restate Occam's razor rule of simplicity in object-oriented terms:

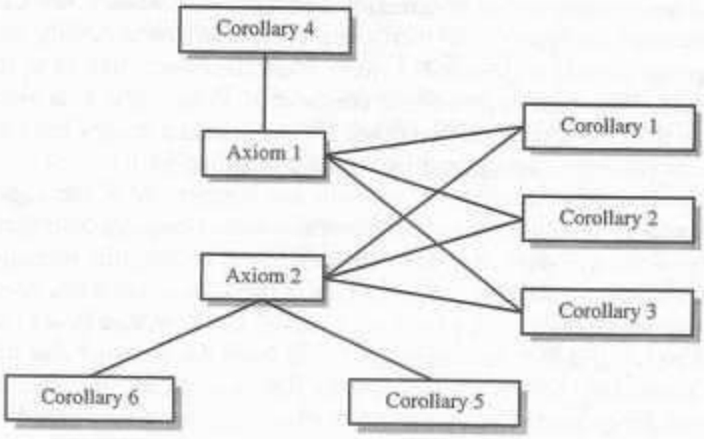
> The best designs usually involve the least complex code but not necessarily the fewest number of classes or methods. Minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application. In an object-oriented system, the best way to minimize complexity is to use inheritance and the system's built-in classes and to add as little as possible to what already is there.

9.4 COROLLARIES

From the two design axioms, many corollaries may be derived as a direct consequence of the axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called *design rules*, and all are derived from the two basic axioms [10] (see Figure 9–2):

- Corollary 1. *Uncoupled design with less information content.* Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

FIGURE 9–2
The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.



- 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).

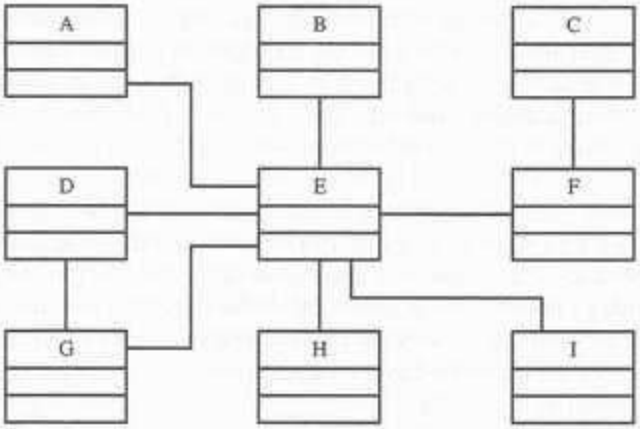


FIGURE 9–3
E is a tightly coupled object.

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).

within a single object or s/w component

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:

> With OO techniques, the same paradigm is used for analysis, design, and implementation. The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects. The designer adds detail to this model perhaps designing screens, user interaction, and client-server interaction. The thought process flows so naturally from analyst to design that it may be difficult to tell where analysis ends and design begins. [8, p. 100]

9.4.5 Corollary 5. Standardization

To reuse classes, you must have a good understanding of the classes in the object-oriented programming environment you are using. Most object-oriented systems, such as Smalltalk, Java, C++, or PowerBuilder, come with several built-in class libraries. Similarly, object-oriented systems are like organic systems, meaning that they grow as you create new applications. The knowledge of existing classes will help you determine what new classes are needed to accomplish the tasks and where you might inherit useful behavior rather than reinvent the wheel. However, class libraries are not always well documented or, worse yet, they are documented but not up to date. Furthermore, class libraries must be easily searched, based on users' criteria. For example, users should be able to search the class repository with commands like "show me all Facet classes." The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

9.4.6 Corollary 6. Designing with Inheritance

When you implement a class, you have to determine its ancestor, what attributes it will have, and what messages it will understand. Then, you have to construct its methods and protocols. Ideally, you will choose inheritance to minimize the amount of program instructions. Satisfying these constraints sometimes means that a class inherits from a superclass that may not be obvious at first glance.

For example, say, you are developing an application for the government that manages the licensing procedure for a variety of regulated entities. To simplify the example, focus on just two types of entities: motor vehicles and restaurants. Therefore, identifying classes is straightforward. All goes well as you begin to model these two portions of class hierarchy. Assuming that the system has no existing classes similar to a restaurant or a motor vehicle, you develop two classes, MotorVehicle and Restaurant.

Subclasses of the MotorVehicle class are PrivateVehicle and CommercialVehicle. These are further subdivided into whatever level of specificity seems appropriate (see Figure 9–4). Subclasses of Restaurant are designed to reflect their own licensing procedures. This is a simple, easy to understand design, although somewhat limited in the reusability of the classes. For example, if in another project you must build a system that models a vehicle assembly plant, the classes from the licensing application are not appropriate, since these classes have instructions and data that deal with the legal requirements of motor vehicle license acquisition and renewal.

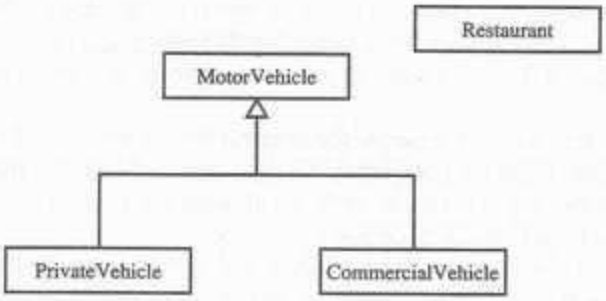
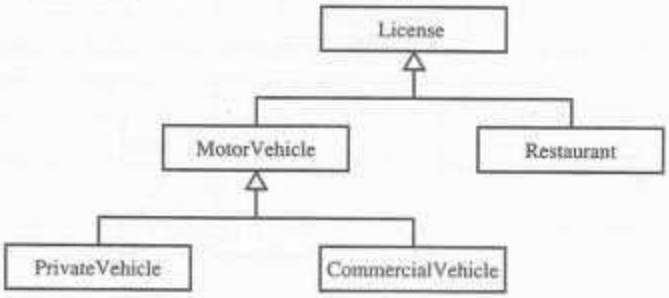


FIGURE 9–4
The initial single inheritance design.

In any case, the design is approved, implementation is accomplished, and the system goes into production. Now, here comes the event that every designer both knows well and dreads—when the nature of the real-world problem exceeds the bounds of the system, so far an elegant design. Say, six months later, while discussing some enhancements to the system with the right people (we learned how to identify right people in Chapter 6), one of them says, "What about coffee wagons, food trucks, and ice cream vendors? We're planning on licensing them as both restaurants and motor vehicles."

You know you need to redesign the application—but redesign how? The answer depends greatly on the inheritance mechanisms supported by the system's target language. If the language supports single inheritance exclusively, the choices are somewhat limited. You can choose to define a formal super class to both MotorVehicle and Restaurant, License, and move common methods and attributes from both classes into this License class (see Figure 9–5). However, the MotorVehicle and Restaurant classes have little in common, and for the most part, their attributes and methods are inappropriate for each other. For example, of what use is the gross weight of a diner or the address of a truck? This necessi-

FIGURE 9–5
The single inheritance design modified to allow licensing food trucks.



tates a very weak formal class (License) or numerous blocking behaviors in both MotorVehicle and Restaurant. This particular decision results in the least reusable classes and potentially extra code in several locations. So, let us try another approach.

Alternatively, you could preserve the original formal classes, MotorVehicle and Restaurant. Next, define a FoodTruck class to descend from CommercialVehicle and copy enough behavior into it from the Restaurant class to support the application's requirements (see Figure 9–6).

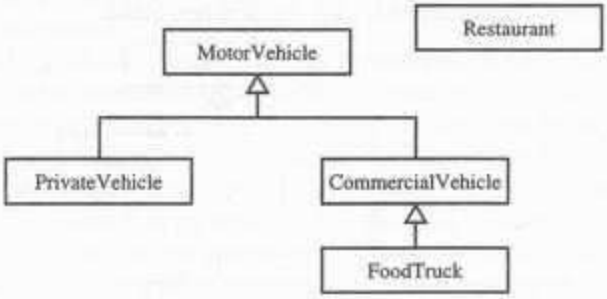
You can give FoodTruck copies of data and instructions from the Restaurant class that allow it to report on food type, health code categories, number of chefs and support staff, and the like. The class is not very reusable (Coad and Yourdon call it *cut-and-paste* reusability), but at least its extra code is localized, allowing simpler debugging and enhancement. Coad and Yourdon describe cut-and-paste type of reusability as follows [4, p. 138]:

> This is better than no reuse at all, but is the most primitive form of reuse. The clerical cost of transcribing the code has largely disappeared with today's cut-and-paste text editors; nevertheless, the software engineer runs the risk of introducing errors during the copying (and modifications) of the original code. Worse is the configuration management problem: it is almost impossible for the manager to keep track of the multiple mutated uses of the original "chunk" of code.

If, on the other hand, the intended language supports multiple inheritance, another route can be taken, one that more closely models the real-world situation. In this case, you design a specialized class, FoodTruck, and specify dual ancestry. Our new class alternative seems to preserve the integrity and code bulk of both ancestors and does nothing that appears to affect their reusability.

In actuality, since we never anticipated this problem in the original design, there probably are instance variables and methods in both ancestors that share the same names. Most languages that support multiple inheritance handle these "hits" by giving precedence to the first ancestor defined. Using this mechanism, reworking will be required in the FoodTruck descendant and, quite possibly, in both ancestors (see Figure 9–7). It easily can become difficult to determine which method,

FIGURE 9–6
Alternatively, you can modify the single inheritance design to allow licensing food trucks.



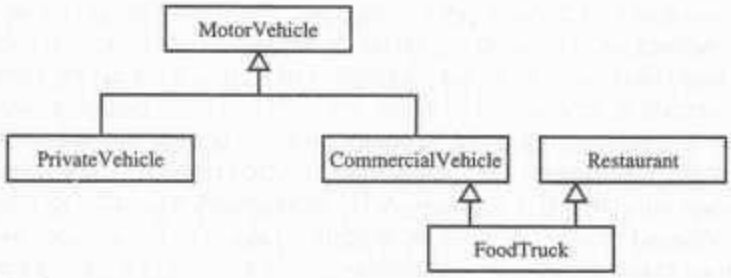


FIGURE 9–7
Multiple inheritance design of the system structure.

in which class, affected an erroneously updated variable in an instance of a new descendant. The difficulties in maintaining such a design increase geometrically with the number of ancestors assigned to a given class.

9.4.6.1 Achieving Multiple Inheritance in a Single Inheritance System *Single inheritance* means that each class has only a single superclass. This technique is used in Smalltalk and several other object-oriented systems. One result of using a single inheritance hierarchy is the absence of ambiguity as to how an object will respond to a given method; you simply trace up the class tree beginning with the object's class, looking for a method of the same name. However, languages like LISP or C++ have a multiple inheritance scheme whereby objects can inherit behavior from unrelated areas of the class tree. This could be desirable when you want a new class to behave similar to more than one existing class. However, multiple inheritance brings with it some complications, such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system.

One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and add an object of another class as an attribute or aggregation. Therefore, as class designer, you have two ways to borrow existing functionality in a class. One is to inherit it, and the other is to use the instance of the class (object) as an attribute. This approach is described in the next section.

9.4.6.2 Avoiding Inheriting Inappropriate Behaviors Beginners in an object-oriented system frequently err by designing subclasses that inherit from inappropriate superclasses. Before a class inherits, ask the following questions:

- Is the subclass fundamentally similar to its superclass (high inheritance coupling)?
- Is it an entirely new thing that simply wants to borrow some expertise from its superclass (low inheritance coupling)?

Often you will find that the latter is true, and if so, you should add an attribute that incorporates the proposed superclass's behavior rather than an inheritance from the superclass. This is because inheritors of a class must be intimate with all its implementation details, and if some implementation is inappropriate, the inheritor's proper functioning could be compromised. For example, if FoodTruck inherits from both Restaurant and CommercialVehicle classes, it might inherit a few inappropriate attributes and methods. A better approach would be to inherit only from CommercialVehicle and have an attribute of the type Restaurant (an instance of Restaurant class). In other words, Restaurant class becomes a-part-of FoodTruck class (see Figure 9–8).

9.5 DESIGN PATTERNS

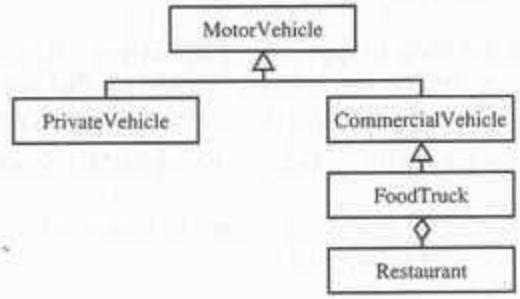
In Chapter 4, we looked at the concept of patterns. A design pattern provides a scheme for refining the subsystems or components of a software system or the relationships among them [1]. In other words, ***design patterns*** are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context. For example, in programming, we have encountered many problems that occurred before and will occur again. The question we must ask ourselves is how we are going to solve it this time [7].

In Chapter 4, we learned that documenting patterns is one way that allows reuse and possibly sharing information learned about how it is best to solve a specific program design problem.

Essays usually are written by following a fairly well-defined form, and so is documenting design patterns (see Chapter 4 for the general form for documenting a pattern). Let us take a look at a design pattern example created by Kurotsuchi [7].

- *Pattern Name*: Facade
- *Rationale and Motivation*: The facade pattern can make the task of accessing a large number of modules much simpler by providing an additional interface layer. When designing good programs, programmers usually attempt to avoid excess coupling between modules/classes. Using this pattern helps to simplify

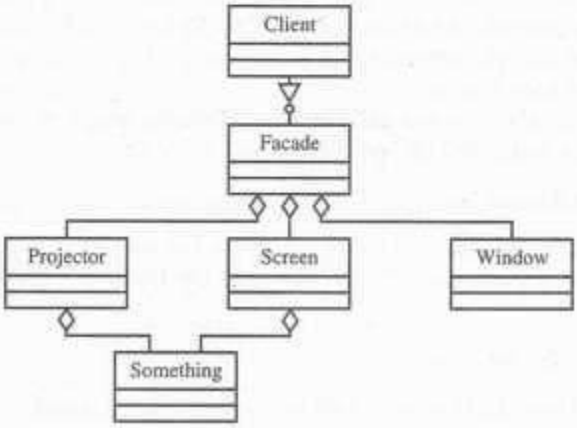
FIGURE 9–8
The FoodTruck class inherits from CommercialVehicle and has an attribute of the type Restaurant. The relationship between FoodTruck and Restaurant is a-part-of.



much of the interfacing that makes large amounts of coupling complex to use and difficult to understand. In a nutshell, this is accomplished by creating a small collection of classes that have a single class that is used to access them, the facade.

- *Classes*: There can be any number of classes involved in this "facade" system, but at least four or more classes are required: One client, the facade, and the classes underneath the facade. In a typical situation, the facade would have a limited amount of actual code, making calls to lower layers most of the time.
- *Advantages/Disadvantages*: As stated before, the primary advantage to using the facade is to make the interfacing between many modules or classes more manageable. One possible disadvantage to this pattern is that you may lose some functionality contained in the lower level of classes, but this depends on how the facade was designed.
- *Examples*: Imagine that you need to write a program that needs to represent a building as rooms that can be manipulated—manipulated as in interacting with objects in the room to change their state. The client that ordered this program has determined that there will be a need for only a finite number of objects (e.g., windows, screens, projectors, etc.) possible in each room and a finite number of operations that can be performed on each of them. You, as the program architect, have decided that the facade pattern will be an excellent way to keep the amount of interfacing low, considering the number of possible objects in each room, and the actions that the client has specified. A sample action for a room is to "prepare it for a presentation." You have decided that this will be part of your facade interface since it deals with a large number of classes but does not really need to bother the programmer with interacting with each of them when a room needs to be prepared. Here is how that facade might be organized (see Figure 9–9). Consider the sheer simplicity from the client's side of the problem.

FIGURE 9–9
Using a design pattern facade eliminates the need for the Client class to deal with a large number of classes.



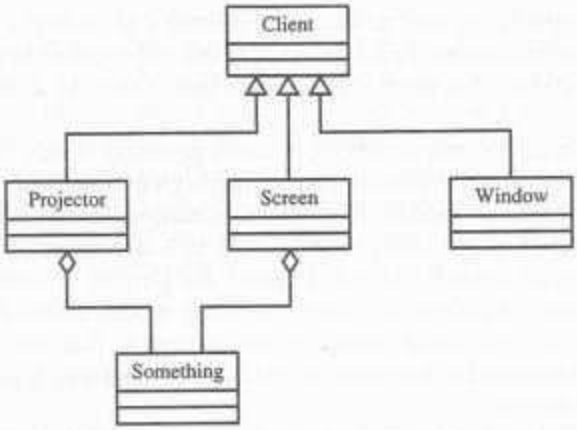


FIGURE 9–10
Not utilizing the design pattern facade, the Client class needs to deal with a large number of classes.

A less thought-out design may have looked like this, making lots of interaction by the client necessary (see Figure 9–10).

9.6 SUMMARY

In this chapter, we looked at the object-oriented design process and design axioms. Integrating design axioms and corollaries with incremental and evolutionary styles of software development will provide you a powerful way for designing systems.

During design, emphasis shifts from the application domain concept toward implementation, such as view (user interface) and access classes. The objects discovered during analysis serve as the framework for design.

The object-oriented design process consists of

- Designing classes (their attributes, methods, associations, structures, and protocols) and applying design axioms. If needed, this step is repeated.
- Designing the access layer.
- Designing the user interface.
- Testing user satisfaction and usability, based on the usage and use cases.
- Iterating and refining the design.

The two design axioms are

- Axiom 1. *The independence axiom*. Maintain the independence of components.
- Axiom 2. *The information axiom*. Minimize the information content of the design.

The six design corollaries are

- Corollary 1. Uncoupled design with less information content.

- Corollary 2. Single purpose.
- Corollary 3. Large number of simple classes.
- Corollary 4. Strong mapping.
- Corollary 5. Standardization.
- Corollary 6. Design with inheritance.

Finally, we looked at the concept of design patterns, which allow systems to share knowledge about their design. These describe commonly recurring problems. Rather than keep asking how to solve the problem this time, we could apply the design pattern (solution) in a previous problem.

KEY TERMS

Axiom (p. 202)
Cohesion (p. 206)
Corollary (p. 202)
Coupling (p. 204)
Design pattern (p. 212)
Theorem (p. 202)

REVIEW QUESTIONS

1. What is the task of design? Why do we need analysis?
2. What is the significance of Occam's razor?
3. How does Occam's razor relate to object-oriented design?
4. How would you differentiate good design from bad design?
5. What is the basic activity in designing an application?
6. Why is a large number of simple classes better than a small number of complex classes?
7. What is the significance of being able to describe in a few sentences what a class does?
8. What clues would you use to identify whether a class is in need of revision?
9. What is the common occurrence in the first attempt of designing classes with inheritance? How would you know? What should you do to fix it?
10. How can an object-oriented system be thought of as an organic system?
11. How can encapsulation, modularization, and polymorphism improve reusability? (Hint: Review Chapter 2.)
12. Why are people not utilizing reusability? List some reasons.
13. Why is it important to know about the classes in the object-oriented programming system you use?
14. How would you decide on subdividing your classes into a hierarchy of super- and subclasses?
15. What are the challenges in designing with inheritance?
16. Describe single and multiple inheritance.
17. What are the risks of a cut-and-paste type of reusability?
18. How can you achieve multiple inheritance in a single inheritance system?
19. How can you avoid a subclass inheriting inappropriate behavior?
20. List the object-oriented design axioms and corollaries.

21. What is the relationship between coupling and cohesion?
22. How would you further refine your design?

PROBLEMS

1. Consult the World Wide Web or the library to obtain an article on the Booch design method. Write a paper based on your findings.
2. Research the Web and write a report on the tools that support patterns-based design and development.
3. Revisit the classes that you identified in the object-oriented analysis for the Grandma's Soups application. What are some of the new classes or attributes and methods that must be added for implementation?
4. The compilers used every day to process computer code are a prime example of the facade pattern in action. What other examples are there?

REFERENCES

1. Appleton, Brad. "Patterns and Software: Essential Concepts and Terminology." http://www.enteract.com/~bradapp/docs/pattern-intro.html,1997.
2. Blum, Bruce I. *Software Engineering, a Holistic View*. New York: Oxford University Press, 1992.
3. Coad, P.; and Yourdon, E. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Yourdon Press, 1991.
4. Coad, P.; and Yourdon, E. *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press, 1991.
5. Cox, B. J. *Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1986.
6. Griss, M. L. "Software Reuse: Objects and Frameworks Are Not Enough." *Object Magazine* 4, no. 9 (February 1995).
7. Kurotsuchi, Brian T. "Design Patterns." http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/.
8. Martin, James; and Odell, James. *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
9. Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; and Lorenson, William. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
10. Suh, Nam. *The Principle of Design*. New York: Oxford University Press, 1990.

Designing Classes

Chapter Objectives

You should be able to define and understand

- Designing classes.
- Designing protocols and class visibility.
- The UML object constraint language (OCL).
- Designing methods.

10.1 INTRODUCTION

Object-oriented design requires taking the objects identified during object-oriented analysis and designing classes to represent them. As a class designer, you have to know the specifics of the class you are designing and be aware of how that class interacts with other classes. Once you have identified your classes and their interactions, you are ready to design classes.

Underlying the functionality of any application is the quality of its design. In this chapter, we look at guidelines and approaches to use in designing classes and their methods. Although the design concepts to be discussed in this chapter are general, we will concentrate on designing the business classes (see Chapter 6). The access and view layer classes will be described in the subsequent chapters. However, the same concepts will apply to designing access and view layer classes.

10.2 THE OBJECT-ORIENTED DESIGN PHILOSOPHY

Object-oriented development requires that you think in terms of classes. A great benefit of the object-oriented approach is that classes organize related properties into units that stand on their own. We go through a similar process as we learn

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).

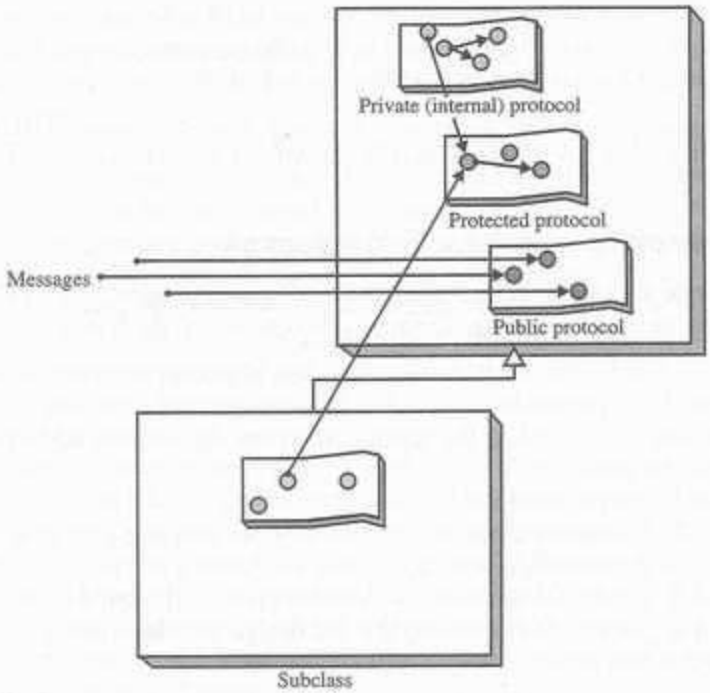


FIGURE 10–1
Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object.

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

model (especially, that defining the class attributes and operations). The main goal of this activity is to refine existing attributes (identified in analysis) or add attributes that can elevate the system into implementation.

10.6.1 Attribute Types

The three basic types of attributes are

1. Single-value attributes.
2. Multiplicity or multivalue attributes.
3. Reference to another object, or instance connection.

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes. The single-value attribute is the most common attribute type. It has only one value or state. For example, attributes such as name, address, or salary are of the single-value type.

The multiplicity or multivalue attribute is the opposite of the single-value attribute since, as its name implies, it can have a collection of many values at any point in time [2]. For example, if we want to keep track of the names of people who have called a customer support line for help, we must use the multivalues attributes.

Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities, in other words, instance connection model association. For example, a person might have one or more bank accounts. A person has zero to many instance connections to Account(s). Similarly, an Account can be assigned to one or more persons (i.e., joint account). Therefore, an Account also has zero to many instance connections to Person(s).

10.6.2 UML Attribute Presentation

As discussed in Chapter 5, OCL can be used during the design phase to define the class attributes. The following is the attribute presentation suggested by UML:

visibility name : type-expression=initial-value

Where *visibility* is one of the following:

+ public visibility (accessibility to all classes).
# protected visibility (accessibility to subclasses and operations of the class).
− private visibility (accessibility only to operations of the class).

Type-expression is a language-dependent specification of the implementation type of an attribute.

Initial-value is a language-dependent expression for the initial value of a newly created object. The initial value is optional. For example, +size: length = 100

The UML style guidelines recommend beginning attribute names with a lowercase letter.

In the absence of a multiplicity indicator (array), an attribute holds exactly one value. Multiplicity may be indicated by placing a multiplicity indicator in brackets after attribute name; for example,

names[10]: String
points[2..*]: Point

The multiplicity of 0..1 provides the possibility of null values: the absence of a value, as opposed to a particular value from the range. For example, the following declaration permits a distinction between the null value and an empty string: name[0..1]: String

10.7 REFINING ATTRIBUTES FOR THE VIANET BANK OBJECTS

In this section, we go through the ViaNet bank ATM system classes and refine the attributes identified during object-oriented analysis (see Chapter 8).

10.7.1 Refining Attributes for the BankClient Class

During object-oriented analysis, we identified the following attributes (see Chapter 8):

firstName
lastName
pinNumber
cardNumber

At this stage, we need to add more information to these attributes, such as visibility and implementation type. Furthermore, additional attributes can be identified during this phase to enable implementation of the class:

#firstName: String
#lastName: String
#pinNumber: String
#cardNumber: String
#account: Account (instance connection)

In Chapter 8, we identified an association between the BankClient and the Account classes (see Figure 8–9). To design this association, we need to add an *account* attribute of type Account, since the BankClient needs to know about his or her account and this attribute can provide such information for the BankClient class. This is an example of instance connection, where it represents the association between the BankClient and the Account objects. All the attributes have been given *protected visibility*.

10.7.2 Refining Attributes for the Account Class

Here is the refined list of attributes for the Account class:

#number: String
#balance: float

#transaction: Transaction (This attribute is needed for implementing the association between the Account and Transaction classes.)

#bankClient: BankClient (This attribute is needed for implementing the association between the Account and BankClient classes.)

At this point we must make the Account class very general, so that it can be reused by the checking and savings accounts.

10.7.3 Refining Attributes for the Transaction Class

The attributes for the Transaction class are these:

#transID: String
#transDate: Date
#transTime: Time
#transType: String
#amount: float
#postBalance: float

Problem 10.1

Why do we not need the account attribute for the Transaction class? Hint: Do transaction objects need to know about account objects?

10.7.4 Refining Attributes for the ATMMachine Class

The ATMMachine class could have the following attributes:

#address: String
#state: String

10.7.5 Refining Attributes for the CheckingAccount Class

Add the *savings* attribute to the class. The purpose of this attribute is to implement the association between the CheckingAccount and SavingsAccount classes.

10.7.6 Refining Attributes for the SavingsAccount Class

Add the *checking* attribute to the class. The purpose of this attribute is to implement the association between the SavingsAccount and CheckingAccount classes.

Figure 10–2 (see Chapter 8) shows a more complete UML class diagram for the bank system. At this stage, we also need to add a very short description of each attribute or certain attribute constraints. For example,

Class ATMMachine

#address: String (The address for this ATM machine.)

#state: String (The state of operation for this ATM machine, such as running, off, idle, out of money, security alarm.)

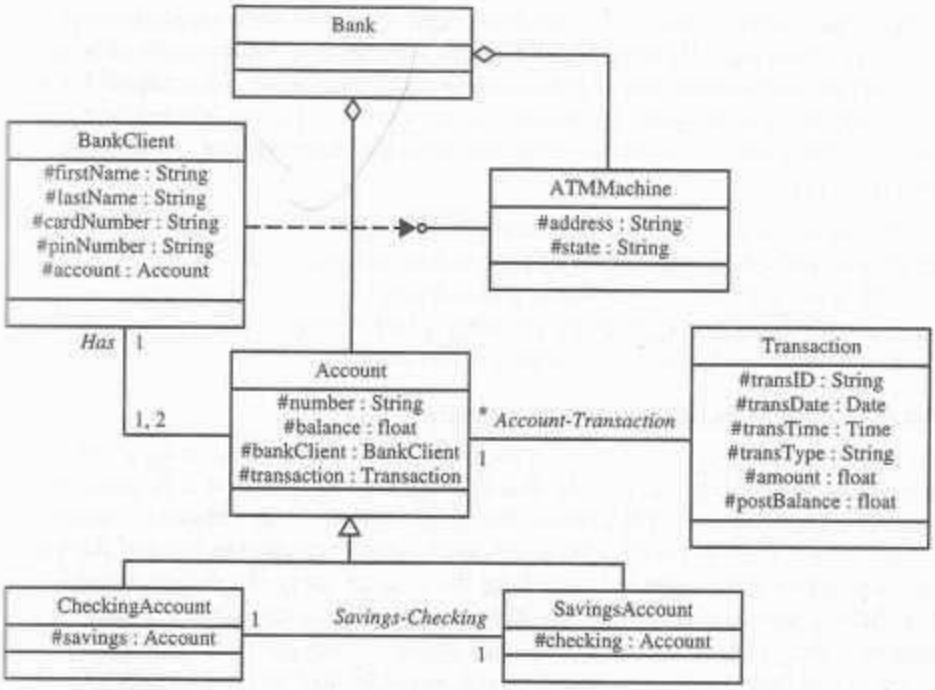


FIGURE 10-2
A more complete UML class diagram for the ViaNet bank system.

10.8 DESIGNING METHODS AND PROTOCOLS

The main goal of this activity is to specify the algorithm for methods identified so far. Once you have designed your methods in some formal structure such as UML activity diagrams with an OCL description, they can be converted to programming language manually or in automated fashion (i.e., using CASE tools). A class can provide several types of methods [3]:

- *Constructor*. Method that creates instances (objects) of the class.
- *Destructor*. The method that destroys instances.
- *Conversion method*. The method that converts a value from one unit of measure to another.
- *Copy method*. The method that copies the contents of one instance to another instance.
- *Attribute set*. The method that sets the values of one or more attributes.
- *Attribute get*. The method that returns the values of one or more attributes.
- *I/O methods*. The methods that provide or receive data to or from a device.
- *Domain specific*. The method specific to the application.

Recall from Chapter 9, Corollary 1, that in designing methods and protocols you must minimize the complexity of message connections and keep as low as possible the number of messages sent and received by an object. Your goal should be

to maximize cohesiveness among objects and software components to improve coupling, because only a minimal amount of essential information should be passed between components. Abstraction leads to simplicity and straightforwardness and, at the same time, increases class versatility. The requirement of simplification, while retaining functionality, seems to lead to increased utility. Here are five rules [1]:

1. If it looks messy, then it's probably a bad design.
2. If it is too complex, then it's probably a bad design.
3. If it is too big, then it's probably a bad design.
4. If people don't like it, then it's probably a bad design.
5. If it doesn't work, then it's probably a bad design.

10.8.1 Design Issues: Avoiding Design Pitfalls

As described in Chapter 9, it is important to apply design axioms to avoid common design problems and pitfalls. For example, we learned that it is much better to have a large set of simple classes than a few large, complex classes. A common occurrence is that, in your first attempt, your class might be too big and therefore more complex than it needs to be. Take the time to apply the design axioms and corollaries, then critique what you have proposed. You may find you can gather common pieces of expertise from several classes, which in itself becomes another "peer" class that the others consult; or you might be able to create a superclass for several classes that gathers in a single place very similar code. Your goal should be maximum reuse of what you have to avoid creating new classes as much as possible. Take the time to think in this way—good news, this gets easier over time.

Lost object focus is another problem with class definitions. A meaningful class definition starts out simple and clean but, as time goes on and changes are made, becomes larger and larger, with the class identity becoming harder to state concisely (Corollary 2). This happens when you keep making incremental changes to an existing class. If the class does not quite handle a situation, someone adds a tweak to its description. When the next problem comes up, another tweak is added. Or, when a new feature is requested, another tweak is added, and so on. Apply the design axioms and corollaries, such as Corollary 2 (which states that each class must have a single, clearly defined purpose). When you document, you easily should be able to describe the purpose of a class in a few sentences.

These problems can be detected early on. Here are some of the warning signs that something is going amiss. There are bugs because the internal state of an object is too hard to track and solutions consist of adding patches. Patches are characterized by code that looks like this: "If this is the case, then force that to be true" or "Do this just in case we need to" or "Do this before calling that function, because it expects this."

Some possible actions to solve this problem are these:

- Keep a careful eye on the class design and make sure that an object's role remains well defined. If an object loses focus, you need to modify the design. Apply Corollary 2 (single purpose).

- Move some functions into new classes that the object would use. Apply Corollary 1 (uncoupled design with less information content).
- Break up the class into two or more classes. Apply Corollary 3 (large number of simple classes).
- Rethink the class definition based on experience gained.

10.8.2 UML Operation Presentation

The following operation presentation has been suggested by the UML. The operation syntax is this:

visibility name: (parameter-list): return-type-expression

Where *visibility* is one of:

+ public visibility (accessibility to all classes).
# protected visibility (accessibility to subclasses and operations of the class).
− private visibility (accessibility only to operations of the class).

Here, *name* is the name of the operation.

Parameter-list: is a list of parameters, separated by commas, each specified by *name: type-expression = default value* (where name is the name of the parameter, type-expression is the language-dependent specification of an implementation type, and default-value is an optional value).

Return-type-expression: is a language-dependent specification of the implementation of the value returned by the method. If return-type is omitted, the operation does not return a value; for example,

+getName(): aName

+getAccountNumber (account: type): account Number

The UML guidelines recommend beginning operation names with a lowercase letter.

10.9 DESIGNING METHODS FOR THE VIANET BANK OBJECTS

At this point, the design of the bank business model is conceptually complete. You have identified the objects that make up your business layer, as well as what services they provide. All that remains is to design methods, the user interface, database access, and implement the methods using any object-oriented programming language. To keep the book language independent, we represent the methods' algorithms with UML activity diagrams, which very easily can be translated into any language. In essence, this phase prepares the system for the implementation. The actual coding and implementation (although they are beyond the scope of this book) should be relatively easy and, for the most part, can be automated by using CASE tools. This is because we know what we want to code. It is always difficult to code when we have no clear understanding of what we want to do.

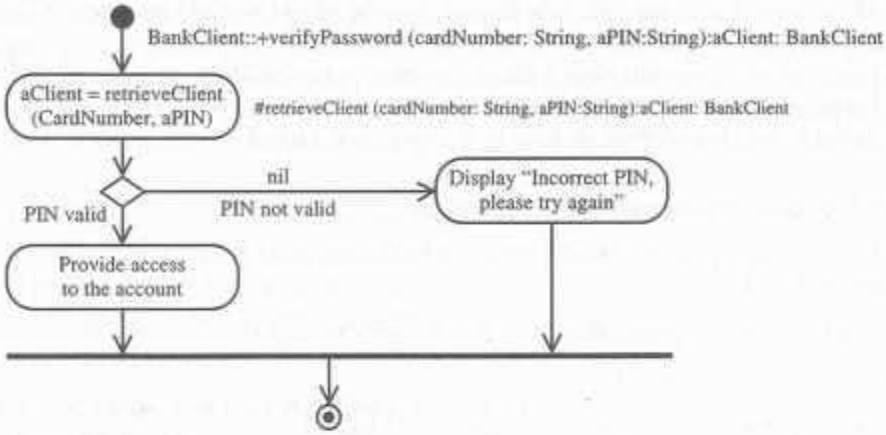


FIGURE 10–3
An activity diagram for the BankClient class verifyPassword method, using OCL to describe the diagram. The syntax for describing a class's method is Class name::methodName. We postpone design of the retrieveClient to Chapter 11, Section 11.10, Designing Access Layer Classes.

10.9.1 BankClient Class VerifyPassword Method

The following describes the verifyPassword service in greater detail. A client PIN code is sent from the ATMMachine object and used as an argument in the verifyPassword method. The verifyPassword method retrieves the client record and checks the entered PIN number against the client's PIN number. If they match, it allows the user to proceed. Otherwise, a message sent to the ATMMachine displays "Incorrect PIN, please try again" (see Figure 10–3).

The verifyPassword methods performs first creates a bank client object and attempts to retrieve the client data based on the supplied card and PIN numbers. At this stage, we realize that we need to have another method, retrieveClient. The retrieveClient method takes two arguments, the card number and a PIN number, and returns the client object or "nil" if the password is not valid. We postpone design of the retrieveClient method to Chapter 11 (Section 11.10, designing the access layer classes).

10.9.2 Account Class Deposit Method

The following describes the deposit service in greater detail. An amount to be deposited is sent to an account object and used as an argument to the deposit service. The account adjusts its balance to its current balance plus the deposit amount. The account object records the deposit by creating a transaction object containing the date and time, posted balance, and transaction type and amount (see Figure 10–4).

Once again we have discovered another method, updateClient. This method, as the name suggests, updates client data. We postpone design of the updateClient method to the Chapter 11 (designing the access layer classes).

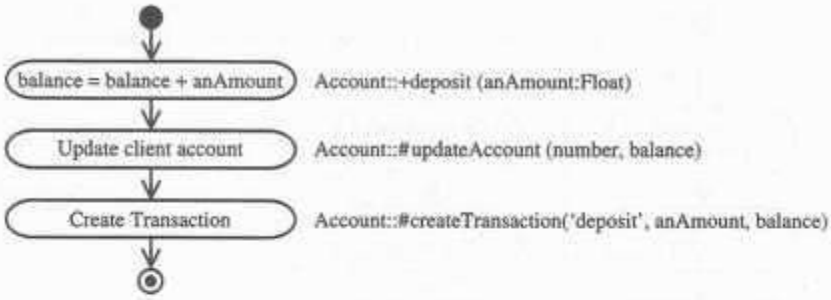


FIGURE 10–4
An activity diagram for the Account class deposit method.

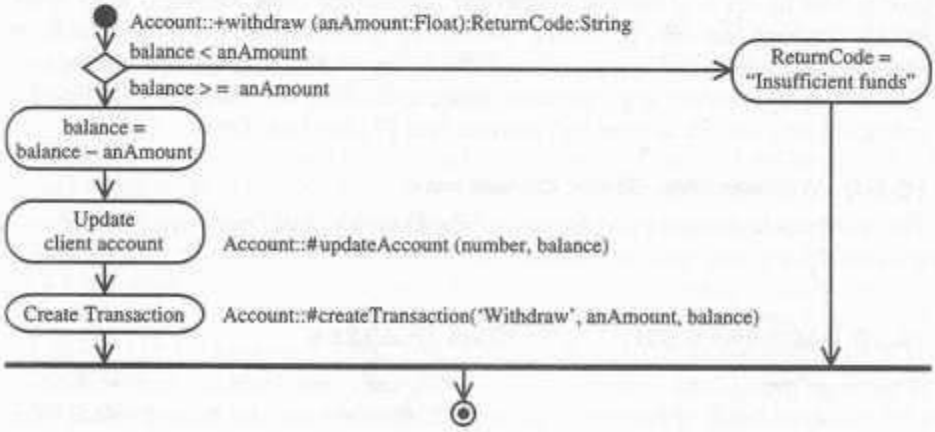
10.9.3 Account Class Withdraw Method

This is the generic withdrawal method that simply withdraws funds if they are available. It is designed to be inherited by the CheckingAccount and SavingsAccount classes to implement automatic funds transfer. The following describes the withdraw method. An amount to be withdrawn is sent to an account object and used as the argument to the withdraw service. The account checks its balance for sufficient funds. If enough funds are available, the account makes the withdrawal and updates its balance; otherwise, it returns an error, saying "insufficient funds." If successful, the account records the withdrawal by creating a transaction object containing date and time, posted balance, and transaction type and amount (see Figure 10–5).

10.9.4 Account Class CreateTransaction Method

The createTransaction method generates a record of each transaction performed against it. The description is as follows. Each time a successful transaction is

FIGURE 10–5
An activity diagram for the Account class withdraw method.



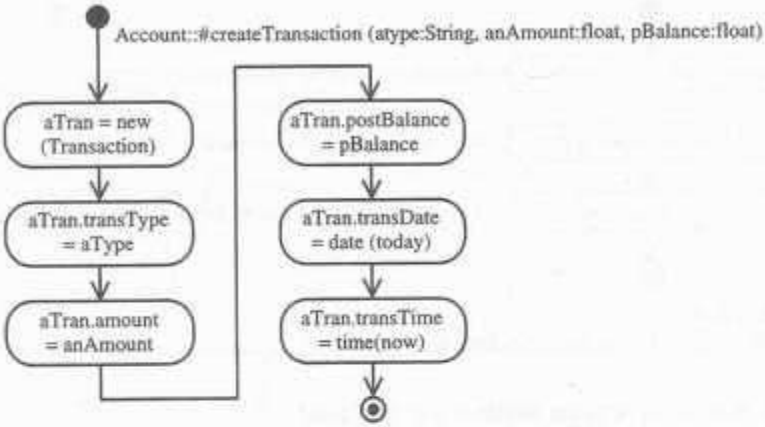


FIGURE 10–6
An activity diagram for the Account class createTransaction method.

performed against an account, the account object creates a transaction object to record it. Arguments into this service include transaction type (withdrawal or deposit), the transaction amount, and the balance after the transaction. The account creates a new transaction object and sets its attributes to the desired information. Add this description to the *createTransaction's* description field (see Figure 10–6).

10.9.5 Checking Account Class Withdraw Method

This is the checking account-specific version of the withdrawal service. It takes into consideration the possibility of withdrawing excess funds from a companion savings account. The description is as follows. An amount to be withdrawn is sent to a checking account and used as the argument to the withdrawal service. If the account has insufficient funds to cover the amount but has a companion savings account, it tries to withdraw the excess from there. If the companion account has insufficient funds, this method returns the appropriate error message. If the companion account has enough funds, the excess is withdrawn from there, and the checking account balance goes to zero (0). If successful, the account records the withdrawal by creating a transaction object containing the date and time, posted balance, and transaction type and amount (see Figure 10–7).

10.9.6 ATMMachine Class Operations

The ATMMachine class provides an interface (view) to the bank system. We postpone designing this class to Chapter 12.

10.10 PACKAGES AND MANAGING CLASSES

A package groups and manages the modeling elements, such as classes, their associations, and their structures. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model ele-

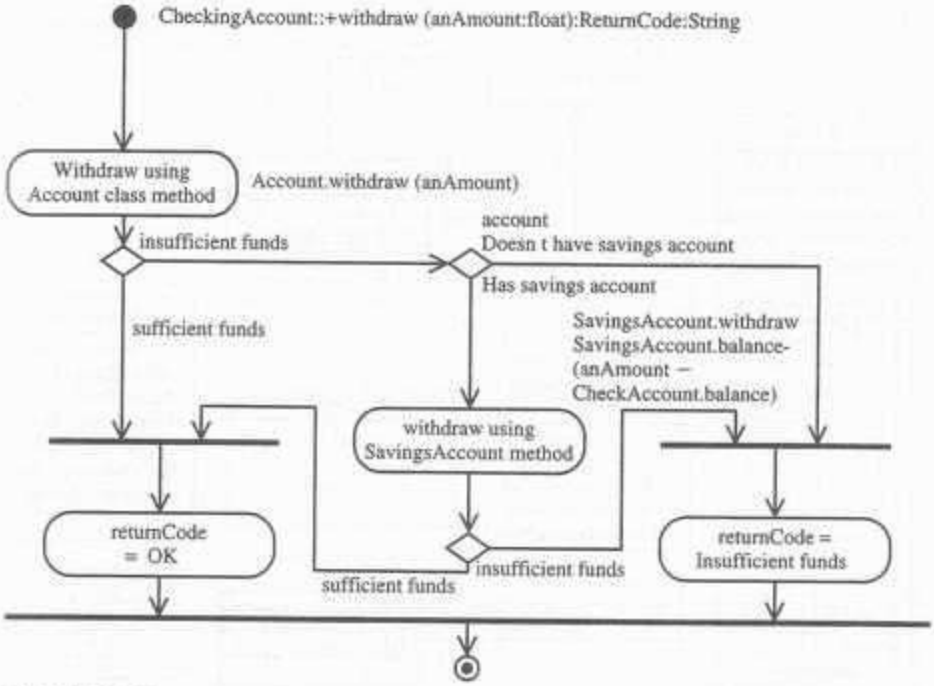


FIGURE 10–7
An activity diagram for the CheckingAccount class withdrawal method.

ments. The entire system description can be thought of as a single high-level subsystem package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages. For example, some packages may contain groups of classes and their relationships, subsystems, or models. A package provides a hierarchy of different system components and can reference other packages. For example, the bank system can be viewed as a package that contains other packages, such as Account package, Client package, and so on. Classes can be packaged based on the services they provide or grouped into the business classes, access classes, and view classes (see Figure 10–8). Furthermore, since packages own model elements and model fragments, they can be used by CASE tools as the basic storage and access control.

In Chapter 5, we learned that a package is shown as a large rectangle with a small rectangular tab. If the contents of the package are shown, then the name of the package may be placed within the tab. A keyword string may be placed above the package name. The keywords *subsystem* and *model* indicate that the package is a meta-model subsystem or model. The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol (+ for public, − for private, # for protected). If the element is in an inner package, its visibility as exported by the outer package is obtained by combining the visibility of an element within the package with the visibility of the package itself: The most restrictive visibility prevails.

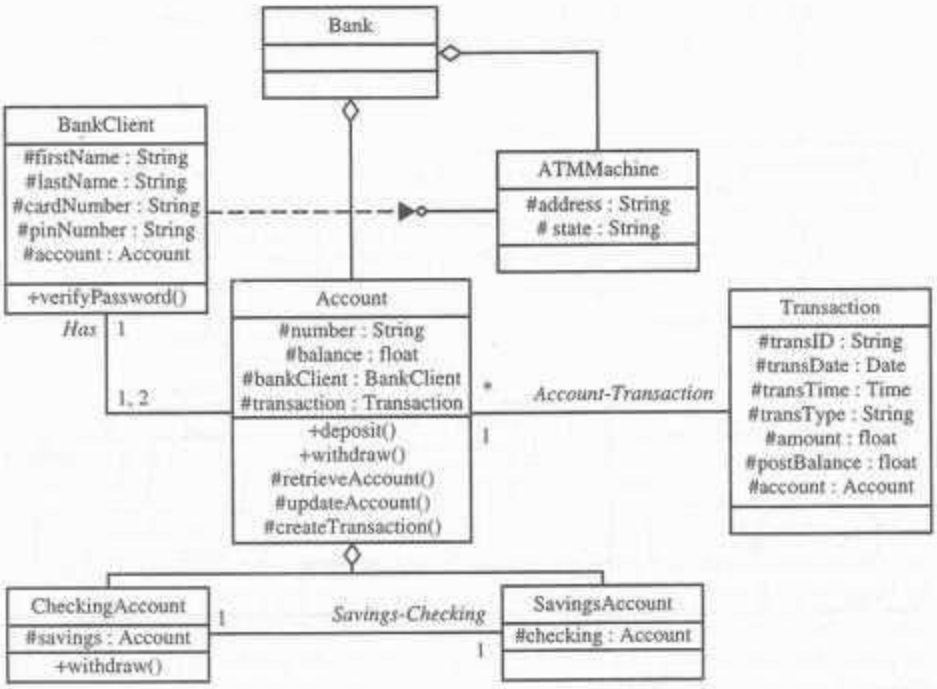


FIGURE 10–8
More complete UML class diagram for the ViaNet bank ATM system. Note that the method parameter list is not shown.

Relationships may be drawn between package symbols to show relationships between at least some of the elements in the packages. In particular, dependency between packages implies one or more dependencies among the elements. Figure 10–9 depicts an even more complete class diagram the ViaNet bank ATM system.

10.11 SUMMARY

The single most important activity in designing an application is coming up with a set of classes that work together to provide the needed functionality. After all, underlying the functionality of any application is the quality of its design.

This chapter concentrated on the first step of the object-oriented design process, which consists of applying the design axioms and corollaries to design classes, their attributes, methods, associations, structures, and protocols; then, iterating and refining.

During the analysis phase, the name of the attribute should be sufficient. However, during the design phase, detailed information must be added to the model (especially, definitions of the class attributes and operations). The UML provides a language to do just that. The rules and semantics of the UML can be 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.

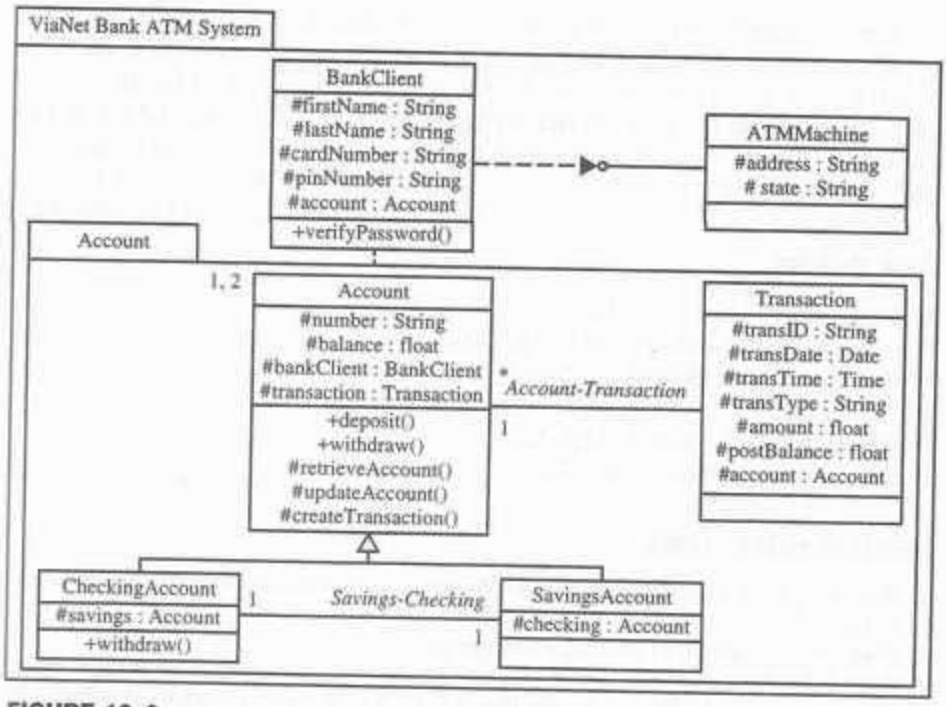


FIGURE 10–9
The ViaNet bank ATM system package and its subsystems.

Lack of a well-designed protocol can manifest itself as encapsulation leakage. This problem occurs when details about a class's internal implementation are disclosed through the interface. As more internal details become visible, the flexibility for making changes in the future is decreased. If an implementation is completely open, almost no flexibility is retained for future changes. Decide what attributes and methods should be private, protected, or public. Use private and protected protocols to define the implementation of the object; use public protocols to define the functionality of the object.

Remember five rules to avoid bad design:

1. If it looks messy, then it's probably a bad design.
2. If it is too complex, then it's probably a bad design.
3. If it is too big, then it's probably a bad design.
4. If people don't like it, then it's probably a bad design.
5. If it doesn't work, then it's probably a bad design.

The UML package is a grouping of model elements. It can organize the modeling elements including classes. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model elements. The entire system description can be thought of as a single, high-level subsystem package with everything else in it.

Object-oriented design is an iterative process. Designing is as much about discovery as construction. Do not be afraid to change a class design, based on experience gained, 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 fix the design as early as possible; redesigning late in the development cycle is problematic and may be impossible.

KEY TERMS

Encapsulation leakage (p. 220)
Object constraint language (OCL) (p. 218)
Private protocol (visibility) (p. 220)
Protocol (p. 219)
Protected protocol (visibility) (p. 220)
Public protocol (visibility) (p. 220)

REVIEW QUESTIONS

1. What are public and private protocols? What is the significance of separating these two protocols?
2. What are some characteristics of a *bad* design?
3. One of the most important skills you can develop is questioning your design, which causes you to think, "Wait a minute, this is starting to get messy." What are some other warning signs that things are about to go amiss?
4. How do design axioms help avoid design pitfalls?
5. Name some problems that come from the lack of a well-designed protocol; for example, giving every method and attribute public visibility.
6. We learned that, to design association, we need to add an instance connection attribute to a class. In a client-server association, does the server need to know about the client? In other words, must we add instance connection attributes of the client in the server class?

PROBLEMS

1. Which corollary (or corollaries) would you apply to design well-defined public, private, and protected protocols?
2. To solve some of the design pitfalls, we could apply the following corollaries. Please apply each corollary and explain how the design axioms and corollaries can help in avoiding design axioms:
   - Keep a careful eye on the class design and make sure that an object's role remains well defined. If an object loses focus, you need to modify the design. Apply Corollary 2 (single purpose).
   - Move some functions into new classes that the object would use. Apply Corollary 1 (uncoupled design with less information content).
   - Break up the class into two or more classes. Apply Corollary 3 (large number of simple classes).
3. Design the queue, order queue, and inventory queue classes in the Grandma's Soups application (see Chapter 6).

REFERENCES

1. Gause, Donald G.; and Weinberg, G. M. *Exploring Requirements: Quality Before Design*. New York: Dorset House, 1989.
2. Norman, Ronald. *Object-Oriented Systems Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
3. Texel, Putnam; and Williams, Charles B. *Use Cases Combined with Booch OMT UML*. Englewood Cliffs, NJ: Prentice-Hall, 1997.

CHAPTER 11

Access Layer: Object Storage and Object Interoperability

Chapter Objectives

You should be able to define and understand

- Object storage and persistence.
- Database management systems and their technology.
- Client-server computing.
- Distributed databases.
- Distributed object computing.
- Object-oriented database management systems.
- Object-relational systems.
- Designing access layer objects.

11.1 INTRODUCTION

A ***database management system*** (DBMS) is a set of programs that enables the creation and maintenance of a collection of related data. A DBMS and associated programs access, manipulate, protect, and manage the data. The fundamental purpose of a DBMS is to provide a reliable, persistent data storage facility and the mechanisms for efficient, convenient data access and retrieval. A database is supposed to represent a real-world situation as completely and accurately as possible. The data model incorporated into a database system defines a framework of concepts that can be used to express an application [5].

Persistence refers to the ability of some objects to outlive the programs that created them. Object lifetimes can be short, as for local objects (these objects are transient), or long, as for objects stored indefinitely in a database (these objects are persistent). Most object-oriented languages do not support serialization or object

persistence, which is the process of writing or reading an object to and from a persistent storage medium, such as a disk file. Even though a reliable, persistent storage facility is the most important aspect of a database, there are many other aspects as well. Persistent object stores do not support query or interactive user interface facilities, as found in fully supported object-oriented database management systems. Furthermore, controlling concurrent access by users, providing ad-hoc query capability, and allowing independent control over the physical location of data are examples of features that differentiate a full database from simply a persistent store. This chapter introduces you to the issues regarding object storage, relational and object-oriented database management systems, object interoperability, and other technologies. We then look at current trends to combine object and relational systems to provide a very practical solution to object storage. We conclude the chapter with a discussion on how to design the access layer objects.

11.2 OBJECT STORE AND PERSISTENCE: AN OVERVIEW

A program will create a large amount of data throughout its execution. Each item of data will have a different lifetime. Atkinson et al. [1] describe six broad categories for the lifetime of data:

1. Transient results to the evaluation of expressions.
2. Variables involved in procedure activation (parameters and variables with a localized scope).
3. Global variables and variables that are dynamically allocated.
4. Data that exist between the executions of a program.
5. Data that exist between the versions of a program.
6. Data that outlive a program.

The first three categories are *transient data*, data that cease to exist beyond the lifetime of the creating process. The other three are nontransient, or *persistent*, data.

Typically, programming languages provide excellent, integrated support for the first three categories of transient data. The other three categories can be supported by a DBMS, or a file system.

The same issues also apply to objects; after all, objects have a lifetime, too. They are created explicitly and can exist for a period of time (during the application session). However, an object can persist beyond application session boundaries, during which the object is stored in a file or a database. A file or a database can provide a longer life for objects—longer than the duration of the process in which they were created. From a language perspective, this characteristic is called *persistence*. Essential elements in providing a persistent store are [4]:

- Identification of persistent objects or reachability (object ID).
- Properties of objects and their interconnections. The store must be able to coherently manage nonpointer and pointer data (i.e., interobject references).
- Scale of the object store. The object store should provide a conceptually infinite store.

- Stability. The system should be able to recover from unexpected failures and return the system to a recent self-consistent state. This is similar to the reliability requirements of a DBMS, object-oriented or not.

Having separate methods of manipulating the data presents many problems. Atkinson et al. [1] claim that typical programs devote significant amounts of code to transferring data to and from the file system or DBMS. Additionally, the use of these external storage mechanisms leads to a variety of technical issues, which will be examined in the following sections.

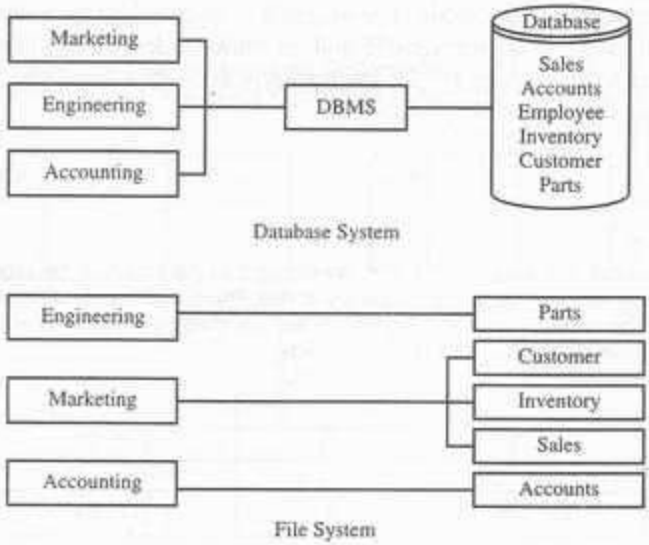
11.3 DATABASE MANAGEMENT SYSTEMS

Databases usually are large bodies of data seen as critical resources to a company. As mentioned earlier, a DBMS is a set of programs that enable the creation and maintenance of a collection of related data. DBMSs have a number of properties that distinguish them from the file-based data management approach. In traditional file processing, each application defines and implements the files it requires. Using a database approach, a single repository of data is maintained, which can be defined once and subsequently accessed by various users (see Figure 11–1).

A fundamental characteristic of the database approach is that the DBMS contains not only the data but a complete definition of the data formats it manages. This description is known as the ***schema***, or ***meta-data***, and contains a complete definition of the data formats, such as the data structures, types, and constraints.

In traditional file processing applications, such meta-data usually are encapsulated in the application programs themselves. In DBMS, the format of the meta-data is independent of any particular application data structure; therefore, it will

FIGURE 11–1
Database system vs. file system.



provide a generic storage management mechanism. Another advantage of the database approach is program-data independence. By moving the meta-data into an external DBMS, a layer of insulation is created between the applications and the stored data structures. This allows any number of applications to access the data in a simplified and uniform manner.

11.3.1 Database Views

The DBMS provides the database users with a conceptual representation that is independent of the low-level details (physical view) of how the data are stored. The database can provide an abstract data model that uses logical concepts such as field, records, and tables and their interrelationships. Such a model is understood more easily by the user than the low-level storage concepts.

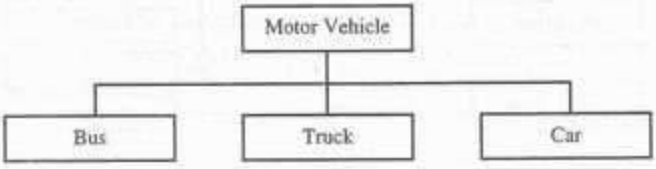
This abstract data model also can facilitate multiple views of the same underlying data. Many applications will use the same shared information but each will be interested in only a subset of the data. The DBMS can provide multiple virtual views of the data that are tailored to individual applications. This allows the convenience of a private data representation with the advantage of globally managed information.

11.3.2 Database Models

A database model is a collection of logical constructs used to represent the data structure and data relationships within the database. Basically, database models may be grouped into two categories: conceptual models and implementation models. The conceptual model focuses on the logical nature of that data presentation. Therefore, the conceptual model is concerned with *what* is represented in the database and the implementation model is concerned with *how* it is represented [12].

11.3.2.1 Hierarchical Model The hierarchical model represents data as a single-rooted tree. Each node in the tree represents a data object and the connections represent a parent-child relationship. For example, a node might be a record containing information about Motor vehicle and its child nodes could contain a record about Bus parts (see Figure 11–2). Interestingly enough, a hierarchical model resembles super-sub relationship of objects.

FIGURE 11–2
A hierarchical model. The top layer, the root, is perceived as the parent of the segment directly below it. In this case motor vehicle is the parent of Bus, Truck, and Car. A segment also is called a node. The segments below another node are the children of the node above them. Bus, Truck, and Car are the children of Motor Vehicle.



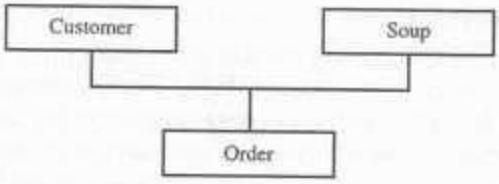
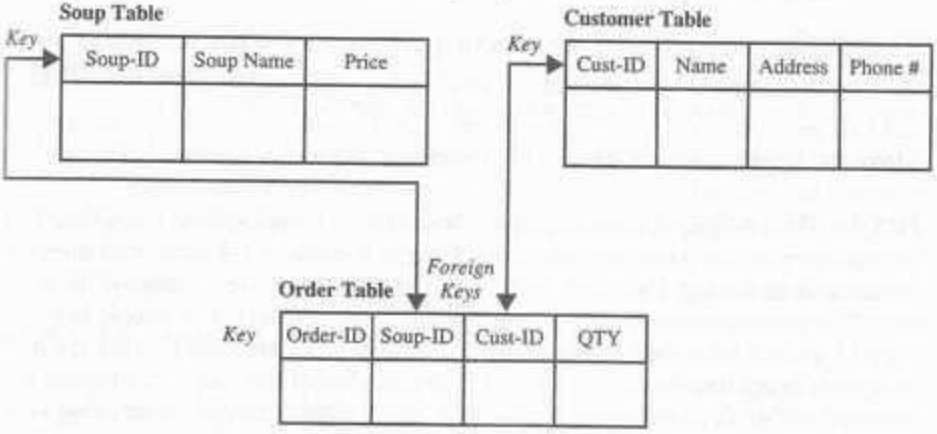


FIGURE 11–3
Network model. An Order contains data from both Customer and Soup.

11.3.2.2 Network Model A network database model is similar to a hierarchical database, with one distinction. Unlike the hierarchical model, a network model's record can have more than one parent. For example, in Figure 11–3, an Order contains data from the Soup and Customer nodes.

11.3.2.3 Relational Model Of all the database models, the relational model has the simplest, most uniform structure and is the most commercially widespread. The primary concept in this database model is the relation, which can be thought of as a table. The columns of each table are attributes that define the data or value domain for entries in that column. The rows of each table are ***tuples*** representing individual data objects being stored. A relational table should have only one primary key. A ***primary key*** is a combination of one or more attributes whose *value* unambiguously locates each row in the table. In Figure 11–4, Soup-ID, Cust-ID, and Order-ID are primary keys in Soup, Customer, and Order tables. A ***foreign key*** is a primary key of one table that is embedded in another table to link the tables. In Figure 11–4, Soup-ID and Cust-ID are foreign keys in the Order table.

FIGURE 11–4
The figure depicts primary and foreign keys in a relation database. Soup-ID is a primary key of the Soup table, Cust-ID is a primary key of the Customer table, and Order-ID is a primary key of the Order table. Soup-ID and Cust-ID are foreign keys in the Order table.



11.3.3 Database Interface

The interface on a database must include a data definition language (DDL), a query, and data manipulation language (DML). These languages must be designed to fully reflect the flexibility and constraints inherent in the data model. Database systems have adopted two approaches for interfaces with the system. One is to embed a database language, such as structured query language (SQL), in the host programming language. This approach is a very popular way of defining and designing a database and its schema, especially with the popularity of languages such as SQL, which has become an industry standard for defining databases. The problem with this approach is that application programmers have to learn and use two different languages. Furthermore, the application programmers have to negotiate the differences in the data models and data structures allowed in both languages [8].

Another approach is to extend the host programming language with database-related constructs. This is the major approach, since application programmers need to learn only a new construct of the same language rather than a completely new language. Many of the currently operational databases and object-oriented database systems have adopted this approach; a good example is GemStone from Servio Logic, which has extended the Smalltalk object-oriented programming.

11.3.3.1 Database Schema and Data Definition Language To represent information in a database, a mechanism must exist to describe or specify to the database the entities of interest. A ***data definition language*** (DDL) is the language used to describe the structure of and relationships between objects stored in a database. This structure of information is termed the *database schema*. In traditional databases, the schema of a database is the collection of record types and set types or the collection of relationships, templates, and table records used to store information about entities of interest to the application.

For example, to create logical structure or schema, the following SQL command can be used:

```
CREATE SCHEMA AUTHORIZATION (creator)
CREATE DATABASE (database name)
```

For example,

```
CREATE TABLE INVENTORY (Inventory_Number CHAR(10)NOT NULL
DESCRIPTION CHAR(25) NOT NULL PRICE DECIMAL (9, 2));
```

where the boldface words are SQL keywords.

11.3.3.2 Data Manipulation Language and Query Capabilities Any time data are collected on virtually any topic, someone will want to ask questions about it. Someone will want the answers to simple questions like "How many of them are there?" or more intricate questions like "What is the percentage of people between ages 21 and 45 who have been employed for five years and like playing tennis?"

Asking questions—more formally, making queries of the data—is a typical and common use of a database. A query usually is expressed through a query language. A ***data manipulation language*** (DML) is the language that allows users to access

and manipulate (such as, create, save, or destroy) data organization. The ***structured query language*** (SQL) is the standard DML for relational DBMSs. SQL is widely used for its query capabilities. The query usually specifies

- The domain of the discourse over which to ask the query.
- The elements of general interest.
- The conditions or constraints that apply.
- The ordering, sorting, or grouping of elements and the constraints that apply to the ordering or grouping.

Query processes generally have sophisticated "engines" that determine the best way to approach the database and execute the query over it. They may use information in the database or knowledge of the whereabouts of particular data in the network to optimize the retrieval of a query.

Traditionally, DML are either procedural or nonprocedural. A procedural DML requires users to specify what data are desired and how to get the data. A nonprocedural DML, like most databases' fourth generation programming language (4GLs), requires users to specify what data are needed but not how to get the data. Object-oriented query and data manipulation languages, such as Object SQL, provide object management capabilities to the data manipulation language.

In a relational DBMS, the DML is independent of the host programming language. A host language such as C or COBOL would be used to write the body of the application. Typically, SQL statements then are embedded in C or COBOL applications to manipulate data. Once SQL is used to request and retrieve database data, the results of the SQL retrieval must be transformed into the data structures of the programming language. A disadvantage of this approach is that programmers code in two languages, SQL and the host language. Another is that the structural transformation is required in both database access directions, to and from the database.

For example, to check the table content, the `SELECT` command is used, followed by the desired attributes. Or, if you want to see all the attributes listed, use the (*) to indicate all the attributes: `SELECT DESCRIPTION, PRICE FROM INVENTORY;` where inventory is the name of a table.

11.4 LOGICAL AND PHYSICAL DATABASE ORGANIZATION AND ACCESS CONTROL

Logical database organization refers to the conceptual view of database structure and the relationships within the database. For example, object-oriented systems represent databases composed of objects, and many allow multiple databases to share information by defining the same object. Physical database organization refers to how the logical components of the database are represented in a physical form by operating system constructs (i.e., objects may be represented as files).

11.4.1 Shareability and Transactions

Data and objects in the database often need to be accessed and shared by different applications. With multiple applications having access to the object concurrently, it is likely that conflicts over object access will arise. The database then

must detect and mediate these conflicts and promote the greatest amount of sharing possible without sacrificing the integrity of data. This mediation process is managed through concurrency control policies, implemented, in part, by transactions.

A ***transaction*** is a unit of change in which many individual modifications are aggregated into a single modification that occurs in its entirety or not at all. Thus, either all changes to objects within a given transaction are applied to the database or none of the changes. A transaction is said to ***commit*** if all changes can be made successfully to the database and to ***abort*** if canceled because all changes to the database cannot be made successfully. This ability of transactions ensures **atomicity** of change that maintain the database in a consistent state.

Many transaction systems are designed primarily for short transactions (lasting on the order of seconds or minutes). They are less suitable for long transactions, lasting hours or longer. Object databases typically are designed to support both short and long transactions. A concurrence control policy dictates what happens when conflicts arise between transactions that attempt access to the same object and how these conflicts are to be resolved.

11.4.2 Concurrency Policy

As you might expect, when several users (or applications) attempt to read and write the same object simultaneously, they create a contention for object. The concurrency control mechanism is established to mediate such conflicts by making policies that dictate how they will be handled.

A basic goal of the transaction is to provide each user with a consistent view of the database. This means that transactions must occur in serial order. In other words, a given user must see the database as it exists either before a given transaction occurs or after that transaction.

The most conservative way to enforce serialization is to allow a user to lock all objects or records when they are accessed and to release the locks only after a transaction commits. This approach, traditionally known as a *conservative* or *pessimistic policy*, provides exclusive access to the object, despite what is done to it. The policy is very conservative because no other user can view the data until the object is released. However, by distinguishing between querying (reading or getting data from) the object and writing to it (which is achieved by qualifying the type of lock placed in the object-read lock or -write lock), somewhat greater concurrency can be achieved. This policy allows many readers of an object but only one writer.

Under an optimistic policy, two conflicting transactions are compared in their entirety and then their serial ordering is determined. As long as the database is able to serialize them so that all the objects viewed by each transaction are from a consistent state of the database, both can continue even though they have read and write locks on a shared object. Thus, a process can be allowed to obtain a read lock on an object already write locked if its entire transaction can be serialized as if it occurred either entirely before or entirely after the conflicting transaction. The reverse also is true: A process may be allowed to obtain a write lock on an object that has a read lock if its entire transaction can be serialized as if it occurred after the conflicting transaction. In such cases, the optimistic policy allows more processes to operate concurrently than the conservative policy.

11.5 DISTRIBUTED DATABASES AND CLIENT-SERVER COMPUTING

Many modern databases are ***distributed databases***, which implies that portions of the database reside on different nodes (computers) and disk drives in the network. Usually, each portion of the database is managed by a server, a process responsible for controlling access and retrieval of data from the database portion. The server dispenses information to client applications and makes queries or data requests to these client applications or other servers. Clients generally reside on nodes in the network other than those on which the servers execute. However, both can reside on the same node, too.

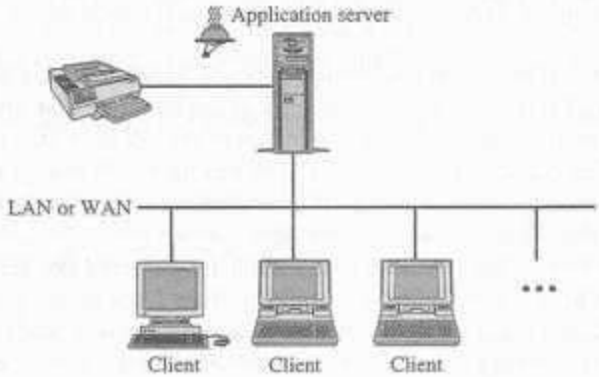
11.5.1 What Is Client-Server Computing?

Client-server computing is the logical extension of modular programming. The fundamental assumption of modular programming is that separation of a large piece of software into its constituent parts ("modules") creates the possibility for easier development and better maintainability.

Client-server computing extends this theory a step further by recognizing that all those modules need not be executed within the same memory space or even on the same machine. With this architecture, the calling module becomes the "client" (that which requests a service) and the called module becomes the "server" (that which provides the service; see Figure 11–5). Another important component of client-server computing is connectivity, which allows applications to communicate transparently with other programs or processes, regardless of their locations. The key element of connectivity is the network operating system (NOS), also known as *middleware*. The NOS provides services such as routing, distribution, messages, filing and printing, and network management [6].

The client is a process (program) that sends a message to a server process (program) requesting that the server perform a task (service). Client programs usually manage the user interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic. The business layer contains all the objects that represent the business (real objects),

FIGURE 11–5
Two-tier client-server system.



such as Order, Customer, Lineitem, Inventory. The client-based process is the front-end of the application, which the user sees and interacts with. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. It also manages the local resources with which the user interacts, such as the monitor, keyboard, workstation, CPU, and peripherals. A key component of a client workstation is the graphical user interface (GUI), which normally is a part of the operating system (i.e., the Windows manager). It is responsible for detecting user actions, managing the Windows on the display, and displaying the data in the Windows.

A server process (program) fulfills the client request by performing the task requested. Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity, and dispatch responses to client requests. Sometimes, server programs execute common or complex business logic. The server-based process "may" run on another machine on the network. This server could be the host operating system or network file server; the server then is provided both file system services and application services. In some cases, another desktop machine provides the application services. The server process acts as a software engine that manages shared resources such as databases, printers, communication links, or high-powered processors. The server process performs the back-end tasks that are common to similar applications.

The server can take different forms. The simplest form of server is a file server. With a file server, the client passes requests for files or file records over a network to the file server. This form of data service requires large bandwidth (the range of data that can be sent over a given medium simultaneously) and can considerably slow down a network with many users. Traditional LAN computing allows users to share resources, such as data files and peripheral devices [6].

More advanced forms of servers are database servers, transaction servers, application servers, and more recently object servers. With database servers, clients pass SQL requests as messages to the server and the results of the query are returned over the network. Both the code that processes the SQL request and the data reside on the server, allowing it to use its own processing power to find the requested data. This is in contrast to the file server, which requires passing all the records back to the client and then letting the client find its own data.

With transaction servers, clients invoke remote procedures that reside on servers, which also contain an SQL database engine. The server has procedural statements to execute a group of SQL statements (transactions), which either all succeed or fail as a unit.

The applications based on transaction servers, handled by on-line transaction processing (OLTP), tend to be mission-critical applications that always require a 1–3 second response time and tight control over the security and the integrity of the database. The communication overhead in this approach is kept to a minimum, since the exchange typically consists of a single request and reply (as opposed to multiple SQL statements in database servers).

Application servers are not necessarily database centered but are used to serve user needs, such as downloading capabilities from Dow Jones or regulating an electronic mail process. Basing resources on a server allows users to share data, while security and management services, also based on the server, ensure data in-

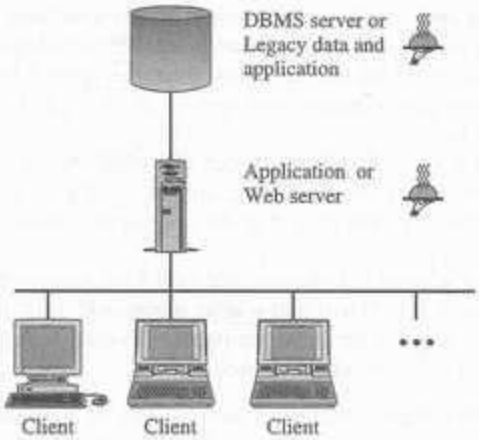


FIGURE 11–6
Three-tiered architecture.

tegrity and security [6]. The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions. For example, database management system servers should run on platforms specially designed and configured to perform queries, and file servers should run on platforms with special elements for managing files.

In a *two-tier* architecture, a client talks directly to a server, with no intervening server. This type of architecture typically is used in small environments with less than 50[1] users (see Figure 11–5). A common error in client-server development is to prepare a prototype of an application in a small, two-tier environment then scale up by simply adding more users to the server. This approach usually will result in an ineffective system, as the server becomes overwhelmed. To properly scale up to hundreds or thousands of users, it usually is necessary to move to a three-tier architecture [14].

A *three-tier* architecture introduces a server (application or Web server) between the client and the server. The role of the application or Web server is manifold. It can provide translation services (as in adapting a legacy application on a mainframe to a client-server environment), metering services (as in acting as a transaction monitor to limit the number of simultaneous requests to a given server), or intelligent agent services (as in mapping a request to a number of different servers, collating the results, and returning a single response to the client) [14] (see Figure 11–6).

Ravi Kalakota describes the basic characteristics of client-server architectures as follows [6]:

1. A combination of a client or front-end portion that interacts with the user and a server or backend portion that interacts with the shared resource. The client process contains solution-specific logic and provides the interface between the user and the rest

[1]Please note that this number depends on many other factors, such as number of transactions per second, as well as the size of the server, the capacity of the network, and so forth.

of the application system. The server process acts as a software engine that manages shared resources such as databases, printers, modems, or high-powered processors.
2. The front-end task and back-end task have fundamentally different requirements for computing resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.
3. The environment is typically heterogeneous and multivendor. The hardware platform and operating system of client and server are not usually the same. Client and server processes communicate through a well-defined set of standard application program interfaces (APIs) . . .
4. An important characteristic of client-server systems is scalability. They can be scaled horizontally or vertically. Horizontal scaling means adding or removing client workstations with only a slight performance impact. Vertical scaling means migrating to a larger and faster server machine or multiservers.

Client-server and distributed computing have arisen because of a change in business needs. Unfortunately, most businesses have existing systems, based on older technology, that must be incorporated into the new, integrated environment; that is, mainframes with a great deal of legacy (older application) software.

Robertson-Dunn [13] answers the question "why build client-server applications?" by pointing out that "business demands the increased benefits." The distinguishing characteristic of a client-server application is the high degree of interaction among various application components [3]. These are the interactions between the client's requests and the server's reactions to those requests. To understand these interactions, we look at the client-server application's components. A typical client-server application consists of the following components:

1. *User interface*. This major component of the client-server application interacts with users, screens, Windows, Windows management, keyboard, and mouse handling.
2. *Business processing*. This part of the application uses the user interface data to perform business tasks. In this book, we look at how to develop this component by utilizing an object-oriented technology.
3. *Database processing*.This part of the application code manipulates data within the application. The data are managed by a database management system, object oriented or not. Data manipulation is done using a data manipulation language, such as SQL or a dialect of SQL (perhaps, an object-oriented query language). Ideally, the DBMS processing is transparent to the business processing layer of the application.

The development and implementation of client-server computing is more complex, more difficult, and more expensive than traditional, single process applications. However, utilizing an object-oriented methodology, we can manage the complexity of client-server applications.

11.5.2 Distributed and Cooperative Processing

The distributed processing means distribution of applications and business logic across multiple processing platforms. Distributed processing implies that processing will occur on more than one processor in order for a transaction to be com-

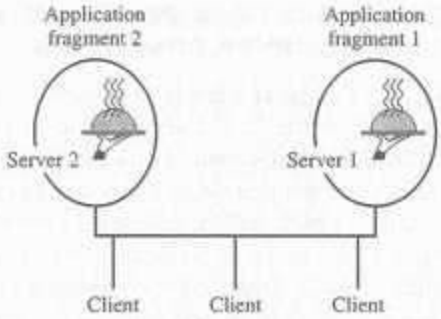
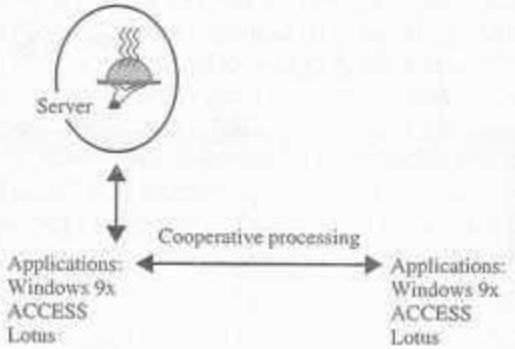


FIGURE 11–7
Distributed processing.

pleted. In other words, processing is distributed across two or more machines, where each process performs part of an application in a sequence. These processes may not run at the same time (see Figure 11–7). For example, in processing an order from a client, the client information may process at one machine and the account information then may process on a different machine. Often, the object used in a distributed processing environment also is distributed across platforms [6].

Cooperative processing is computing that requires two or more distinct processors to complete a single transaction. Cooperative processing is related to both distributed and client-server processing. Cooperative processing is a form of distributed computing in which two or more distinct processes are required to complete a single business transaction. Usually, these programs interact and execute concurrently on different processors (see Figure 11–8). Cooperative processing also can be considered to be a style of distributed processing, if communication between processors is performed through a message-passing architecture [6].

FIGURE 11–8
Cooperative processing.



11.6 DISTRIBUTED OBJECTS COMPUTING: THE NEXT GENERATION OF CLIENT-SERVER COMPUTING

In the preceding section, we looked at what is now considered the first generation of client-server computing. Eventually, the server code in your client-server system will give way to collections of distributed objects. Since all of them will need to talk to each other, the second generation of client-server computing is based on distributed object computing, which will be covered in the next section.

Software technology is in the midst of a major computational shift toward distributed object computing (DOC). Distributed computing is poised for a second client-server revolution, a transition from first generation client-server era to a next generation client-server era. In this new client-server model, servers are plentiful instead of scarce (because every client can be a server) and proximity no longer matters. This immensely expanded client-server model is made possible by the recent exponential network growth and the progress in network-aware multithreaded desktop operating systems.

In the first generation client-server era, which still is very much in progress, SQL databases, transaction processing (TP) monitors, and groupware have begun to displace file servers as client-server application models. In the new client-server era, distributed object technology is expected to dominate other client-server application models.

Distributed object computing promises the most flexible client-server systems, because it utilizes reusable software components that can roam anywhere on networks, run on different platforms, communicate with legacy applications by means of object wrappers,[2] and manage themselves and the resources they control. Objects can help break monolithic applications into more manageable components that coexist on the expanded bus.

Distributed objects are reusable software components that can be distributed and accessed by users across the network. These objects can be assembled into distributed applications [9]. Distributed object computing introduces a higher level of abstraction into the world of distributed applications. Applications no longer consist of clients and servers but users, objects, and methods. The user no longer needs to know which server process performs a given function. All information about the function is hidden inside the encapsulated object. A message requesting an operation is sent to the object, and the appropriate method is invoked.

Distributed object computing will be the key part of tomorrow's information systems. DOC resulted from the need to integrate mission-critical applications and data residing on systems that are geographically remote, sometimes from users and often from each other, and running on many different hardware platforms. Furthermore, the information systems must link applications developed in different languages, use data from object and relational databases and from mainframe systems, and be optimized for use across the Internet and through departmental intranets. Historically, businesses have had to integrate applications and data by writing custom interfaces between systems, forcing developers to spend their time

[2] Conceptually, an object wrapper is very similar to an access layer, discussed later in this chapter.

building and maintaining an infrastructure rather than adding new business functionality.

Distributed object technology has been tied to standards from the early stage. Since 1989, the ***Object Management Group*** (OMG), with over 500 member companies, has been specifying the architecture for an open software bus on which object components written by different vendors can operate across networks and operating systems. The OMG and the object bus are well on their way to becoming the universal client-server middleware.

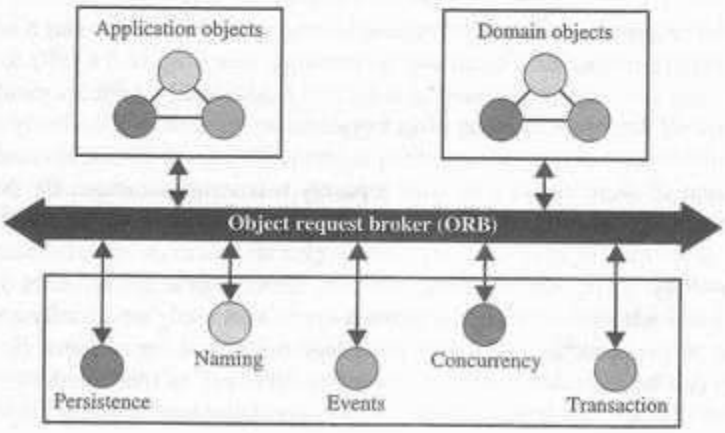
Currently, there are several competing DOC standards, including the Object Management Group's CORBA, OpenDoc, and Microsoft's ActiveX/DCOM. Although DOC technology offers unprecedented computing power, few organizations have been able to harness it as yet. The main reasons commonly cited for slow adoption of DOC include closed legacy architecture, incompatible protocols, inadequate network bandwidths, and security issues. In the next subsections, we look at Microsoft's DCOM and OMG's CORBA.

11.6.1 Common Object Request Broker Architecture

Many organizations are now adopting the Object Management Group's ***common object request broker architecture*** (CORBA), a standard proposed as a means to integrate distributed, heterogeneous business applications and data. The CORBA interface definition language (IDL) allows developers to specify language-neutral, object-oriented interfaces for application and system components. IDL definitions are stored in an interface repository, a sort of phone book that offers object interfaces and services. For distributed enterprise computing, the interface repository is central to communication among objects located on different systems.

CORBA ***object request brokers*** (ORBs) implement a communication channel through which applications can access object interfaces and request data and services (see Figure 11–9). The CORBA common object environment (COE) provides system-

FIGURE 11–9
The Common Object Request Broker Architecture (CORBA).



level services such as life cycle management for objects accessed through CORBA, event notification between objects, and transaction and concurrency control.

11.6.2 Microsoft's ActiveX/DCOM

Microsoft's component object model (COM) and its successor the distributed component object model (DCOM) are Microsoft's alternatives to OMG's distributed object architecture CORBA. Microsoft and the OMG are competitors, and few can say for sure which technology will win the challenge. Although CORBA benefits from wide industry support, DCOM is supported mostly by one enterprise, Microsoft. However, Microsoft is no small business concern and holds firmly a huge part of the microcomputer population, so DCOM has appeared a very serious competitor to CORBA. DCOM was bundled with Windows NT 4.0 and there is a good chance to see DCOM in all forthcoming Microsoft products.

The ***distributed component object model***, Microsoft's alternative to OMG's CORBA, is an Internet and component strategy where ActiveX (formerly known as object linking and embedding, or OLE) plays the role of DCOM object. DCOM also is backed by a very efficient Web browser, the Microsoft Internet Explorer.

11.7 OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS: THE PURE WORLD

Database management systems have progressed from indexed files to network and hierarchical database systems to relational systems. The requirements of traditional business data processing applications are well met in functionality and performance by relational database systems focused on the needs of business data processing applications. However, as many researchers observed, they are inadequate for a broader class of applications with unconventional and complex data type requirements. These requirements along with the popularity of object-oriented programming have resulted in great demand for an object-oriented DBMS (OODBMS). Therefore, the interest in OODBMS initially stemmed from the data storage requirements of design support applications (e.g., CAD, CASE, office information systems).

The ***object-oriented database management system*** is a marriage of object-oriented programming and database technology (see Figure 11–10) to provide what we now call *object-oriented databases*. Additionally, object-oriented databases allow all the benefits of an object orientation as well as the ability to have a strong equivalence with object-oriented programs, an equivalence that would be lost if an alternative were chosen, as with a purely relational database. By combining object-oriented programming with database technology, we have an integrated application development system, a significant characteristic of object-oriented database technology. Many advantages accrue from including the definition of operations with the definition of data. First, the defined operations apply universally and are not dependent on the particular database application running at the moment. Second, the data types can be extended to support complex data such as multimedia by defining new object classes that have operations to support the new kinds of information.

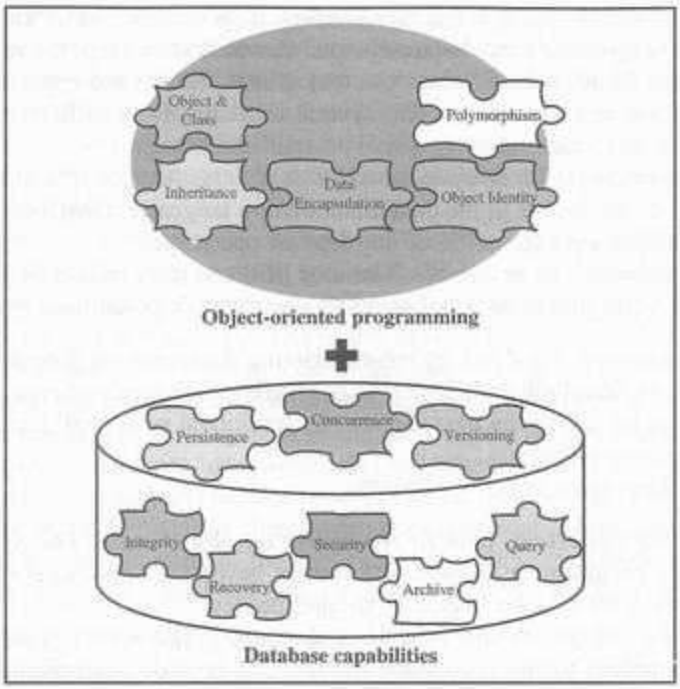


FIGURE 11-10
The object-oriented database management system is a marriage of object-oriented programming and database technology.

The "Object-Oriented Database System Manifesto" by Malcom Atkinson et al. [2] described the necessary characteristics that a system must satisfy to be considered an object-oriented database. These categories can be broadly divided into *object-oriented* language properties and *database* requirements.

First, the rules that make it an object-oriented system are as follows:

1. *The system must support complex objects.* A system must provide simple ***atomic types of objects*** (integers, characters, etc.) from which complex objects can be built by applying constructors to atomic objects or other complex objects or both.
2. *Object identity must be supported.* A data object must have an identity and existence independent of its values.
3. *Objects must be encapsulated.* An object must encapsulate both a program and its data. Encapsulation embodies the separation of interface and implementation and the need for modularity.
4. *The system must support types or classes.* The system must support either the type concept (embodied by C++) or the class concept (embodied by Smalltalk).
5. *The system must support inheritance.* Classes and types can participate in a class hierarchy. The primary advantage of inheritance is that it factors out shared code and interfaces.

6. *The system must avoid premature binding*. This feature also is known as *late binding* or *dynamic binding* (see Chapter 2, which shows that the same method name can be used in different classes). Since classes and types support encapsulation and inheritance, the system must resolve conflicts in operation names at run time.
7. *The system must be computationally complete*. Any computable function should be expressible in the data manipulation language (DML) of the system, thereby allowing expression of any type of operation.
8. *The system must be extensible*. The user of the system should be able to create new types that have equal status to the system's predefined types.

These requirements are met by most modern object-oriented programming languages such as Smalltalk and C++. Also, clearly, these requirements are *not* met *directly* (more on this in the next section) by traditional relational, hierarchical, or network database systems.

Second, these rules make it a DBMS:

9. *It must be persistent, able to remember an object state*. The system must allow the programmer to have data survive beyond the execution of the creating process for it to be reused in another process.
10. *It must be able to manage very large databases*. The system must efficiently manage access to the secondary storage and provide performance features, such as indexing, clustering, buffering, and query optimization.
11. *It must accept concurrent users*. The system must allow multiple concurrent users and support the notions of atomic, serializable transactions.
12. *It must be able to recover from hardware and software failures*. The system must be able to recover from software and hardware failures and return to a coherent state.
13. *Data query must be simple*. The system must provide some high-level mechanism for ad-hoc browsing of the contents of the database. A graphical browser might fulfill this requirement sufficiently.

These database requirements are met by the majority of existing database systems. From these two sets of definitions it can be argued that an OODBMS is a DBMS with an underlying object-oriented model.

11.7.1 Object-Oriented Databases versus Traditional Databases

The scope of the responsibility of an OODBMS includes definition of the object structures, object manipulation, and recovery, which is the ability to maintain data integrity regardless of system, network, or media failure. Furthermore, OODBMSs like DBMSs must allow for sharing; secure, concurrent multiuser access; and efficient, reliable system performance.

One obvious difference between the traditional and object-oriented databases is derived from the object's ability to interact with other objects and with itself. The objects are an "active" component in an object-oriented database, in contrast to conventional database systems, where records play a passive role. Yet another distinguishing feature of object-oriented database is inheritance. Relational database

systems do not explicitly provide inheritance of attributes and methods. Object-oriented databases, on the other hand, represent relationships explicitly, supporting both navigational and associative access to information. As the complexity of interrelationships between information within the database increases, so do the advantages of representing relationships explicitly. Another benefit of using explicit relationships is the improvement in data access performance over relational value-based relationships.

Object-oriented databases also differ from the more traditional relational databases in that they allow representation and storage of data in the form of objects. Each object has its own identity, or *object-ID* (as opposed to the purely value-oriented approach of traditional databases). The object identity is independent of the state of the object. For example, if one has a car object and we remodel the car and change its appearance, the engine, the transmission, and the tires so that it looks entirely different, it would still be recognized as the same object we had originally. Within an object-oriented database, one always can ask whether this is the same object I had previously, assuming one remembers the object's identity. Object identity allows objects to be related as well as shared within a distributed computing network.

All these advantages point to the application of object-oriented databases to information management problems that are characterized by the need to manage

- A large number of different data types.
- A large number of relationships between the objects.
- Objects with complex behaviors.

Application areas where this kind of complexity exists include engineering, manufacturing, simulations, office automation, and large information systems ("No More Fishing for Data" is a real-world example of this).

11.8 OBJECT-RELATIONAL SYSTEMS: THE PRACTICAL WORLD

In practice, even though many applications increasingly are developed in an object-oriented programming technology, chances are good that the data those applications need to access live in a very different universe—a relational database. In such an environment, the introduction of object-oriented development creates a fundamental mismatch between the programming model (objects) and the way in which existing data are stored (relational tables) [9].

To resolve the mismatch, a mapping tool between the application objects and the relational data must be established. Creating an object model from an existing relational database layout (schema) often is referred to as ***reverse engineering***. Conversely, creating a relational schema from an existing object model often is referred to as ***forward engineering***. In practice, over the life cycle of an application, forward and reverse engineering need to be combined in an iterative process to maintain the relationship between the object and relational data representations.

Tools that can be used to establish the object-relational mapping processes have begun to emerge. The main process in relational and object integration is defining the relationships between the table structures (represented as schemata) in the relational database with classes (representing classes) in the object model. Sun's Java

BOX 11.1

Real-World Issues on the Agenda

NO MORE FISHING FOR DATA

Client/Server: With the help of a three-tier decision support system, a Canadian department baits salmon spawning

Esther Shein

Tracking the spawning habits of salmon using high technology may sound like fishy business, but it's more important than you'd think—especially when you're up against the whims of Mother Nature.

The huge amount of data that needed to be tracked was daunting, according to Ian Williams, a senior biologist and head of the Fresh Water Habitats Science Group for the Department of Fisheries (DOF), in Nanaimo, British Columbia. To make matters worse, different groups within the DFO had been creating independent databases focusing on their area of interest. It was time for some serious streamlining.

THREE-TIER TO RESCUE

The consolidation came in the form of the decision support system, dubbed The Integrated Fraser Salmon model, which was built using Facet Decision Systems Inc.'s development environment. Facet's tool comprises middleware for links to third-party databases; an object-oriented spreadsheet-like development environment; and 3-D visualization tools. Facet's object-oriented capabilities and capacity to accommodate ever-changing business rules make it applicable for any industry—for example, finance—that needs to construct and analyze large data models.

The Fraser Salmon model was built in three layers: one for data access, one for data integration and one to parlay the biologist's rules which produces the technical results. DFO officials wanted all the miscellaneous databases linked so employees would have access to the same information—for example, the number of fish caught in oceans and rivers over a particular period, the estimated space still available for spawning and where forest fires occur. The top layer of the system contains policy analysis, which are tools to create and compare scenarios "to see technical impacts and translate them into the information you need to make decisions," explains Scott Akenhead, vice president of Business Development at Facet.

The model, which Akenhead likens to a spreadsheet, has cells that are object-oriented in nature, in the form of graphics, maps or the links to the Oracle data and rules written by a biologist.

The model differs from the typical data warehouse, because of the use of advanced object-oriented technology, which allows Facet to build a much larger model. "We didn't just assemble the data and drop it in their laps. The data was analyzed by the Facet system using rules the biologist provided," he explains.

"We found a way to make new object-oriented technology available to people who are not programmers," Akenhead says.

Today, using map as the user interface, the DFO has moved from raw digital map data (a representation of a paper map on-screen) to 3-D maps that can be analyzed to compare policy suggestions. "We created river networks and drainage surfaces from raw data, which are more useful to the biologist" because they do things the raw maps couldn't do, such as simulate the fish swimming up the streams, Akenhead says.

By Esther Shein, PC Week, September 23, 1996, Vol. 13, Number 38.

Blend is an example of such a tool. Java Blend allows the developer access to relational data as Java objects, thus avoiding the mismatch between the relational and object data models. Java Blend also has mapping capabilities to define Java classes from relational tables or relational tables from the Java classes [15].

11.8.1 Object-Relation Mapping

In a relational database, the schema is made up of tables, consisting of rows and columns, where each column has a name and a simple data type. In an object

model, the counterpart to a table is a class (or classes), which has a set of attributes (properties or data members). Object classes describe behavior with methods.

A tuple (row) of a table contains data for a single entity that correlates to an object (instance of a class) in an object-oriented system. In addition, a stored procedure in a relational database may correlate to a method in an object-oriented architecture. A ***stored procedure*** is a module of precompiled SQL code maintained within the database that executes on the server to enforce rules the business has set about the data. Therefore, the mappings essential to object and relational integration are between a table and a class, between columns and attributes, between a row and an object, and between a stored procedure and a method.

Table row column stored procedure

Class object attribute method

For a tool to be able to define how relational data maps to and from application objects, it must have at least the following mapping capabilities (note all these are two-way mappings, meaning they map from the relational system to the object and from the object back to the relational system):

- Table-class mapping.
- Table-multiple classes mapping.
- Table-inherited classes mapping.
- Tables-inherited classes mapping.

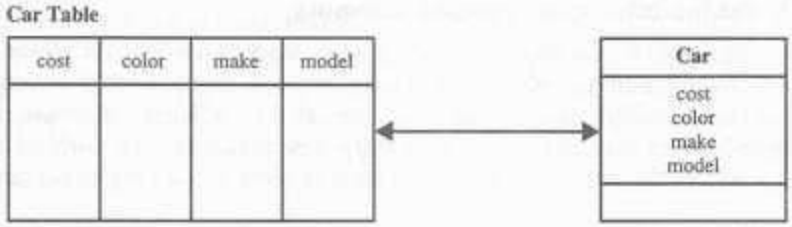
Furthermore, in addition to mapping column values, the tool must be capable of interpretation of relational foreign keys. The tool must describe both how the foreign key can be used to navigate among classes and instances in the mapped object model and how referential integrity is maintained. ***Referential integrity*** means making sure that a dependent table's foreign key contains a value that refers to an existing valid tuple in another relation.

11.8.2 Table-Class Mapping

Table-class mapping is a simple one-to-one mapping of a table to a class and the mapping of columns in a table to properties in a class. In this mapping, a single table is mapped to a single class, as shown in Figure 11–11.

In such mapping, it is common to map all the columns to properties. However, this is not required, and it may be more efficient to map only those columns for which an object model is required by the application(s). With the table-class ap-

FIGURE 11–11
Table-class mapping. Each row in the table represents an object instance and each column in the table corresponds to an object attribute.



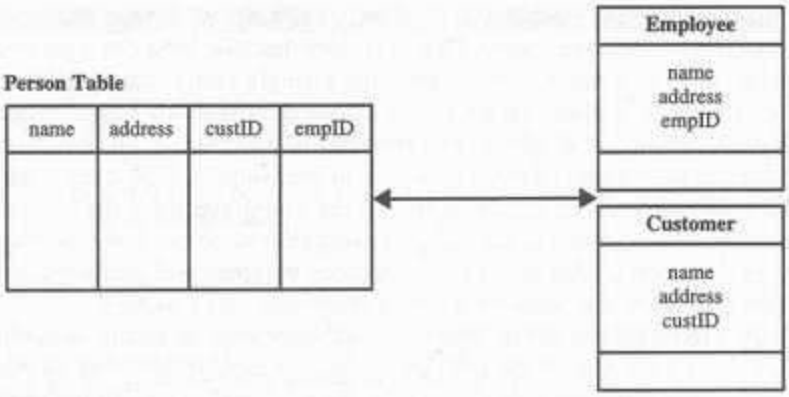


FIGURE 11–12
Table-multiple classes mapping. The custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

proach, each row in the table represents an object instance and each column in the table corresponds to an object attribute. This one-to-one mapping of the table-class approach provides a literal translation between a relational data representation and an application object. It is appealing in its simplicity but offers little flexibility.

11.8.3 Table-Multiple Classes Mapping

In the table-multiple classes mapping, a single table maps to multiple noninheriting classes. Two or more distinct, noninheriting classes have properties that are mapped to columns in a single table. At run time, a mapped table row is accessed as an instance of one of the classes, based on a column value in the table [11].

In Figure 11–12, the custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

11.8.4 Table-Inherited Classes Mapping

In table-inherited classes mapping, a single table maps to many classes that have a common superclass. This mapping allows the user to specify the columns to be shared among the related classes. The superclass may be either abstract or instantiated. In Figure 11–13, instances of salariedEmployee can be created for any row in the Person table that has a non null value for the Salary column. If Salary is null, the row is represented by an hourlyEmployee instance.

11.8.5 Tables-Inherited Classes Mapping

Another approach here is tables-inherited classes mapping, which allows the translation of *is-a* relationships that exist among tables in the relational schema into class inheritance relationships in the object model. In a relational database, an is-a relationship often is modeled by a primary key that acts as a foreign key to another table. In the object model, *is-a* is another term for an inheritance relation-

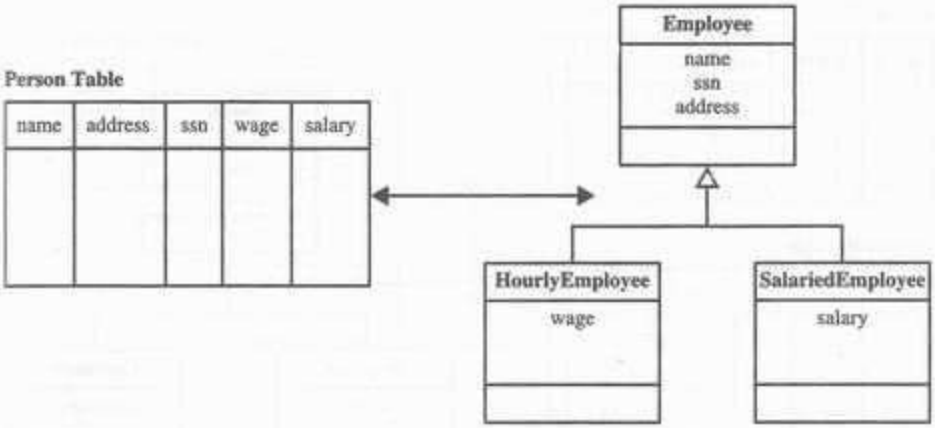


FIGURE 11–13
Table-inherited classes mapping. Instances of SalariedEmployee can be created for any row in the Person table that has a non null value for the salary column. If salary is null, the row is represented by an HourlyEmployee instance.

ship. By using the inheritance relationship in the object model, the mapping can express a richer and clearer definition of the relationships than is possible in the relational schema.

Figure 11–14 shows an example that maps a Person table to class Person and then maps a related Employee table to class Employee, which is a subclass of class Person. In this example, instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joining of the Employee and Person tables on the SSN key). Furthermore, SSN is used both as a primary key on the Person table for activating instances of Person and as a foreign key on the Person table and a primary key on the Employee table for activating instances of type Employee.

11.8.6 Keys for Instance Navigation

In mapping columns to properties, the simplest approach is to translate a column's value into the corresponding class property value. There are two interpretations of this mapping: Either the column is a data value or it defines a navigable relationship between instances (i.e., a foreign key). The mapping also should specify how to convert each data value into a property value on an instance.

In addition to simple data conversion, mapping of column values defines the interpretation of relational foreign keys. The mapping describes both how the foreign key can be used to navigate among classes and instances in the mapped object model and how referential integrity is maintained. A foreign key defines a relationship between tables in a relational database. In an object model, this association is where objects can have references to other objects that enable instance-to-instance navigation.

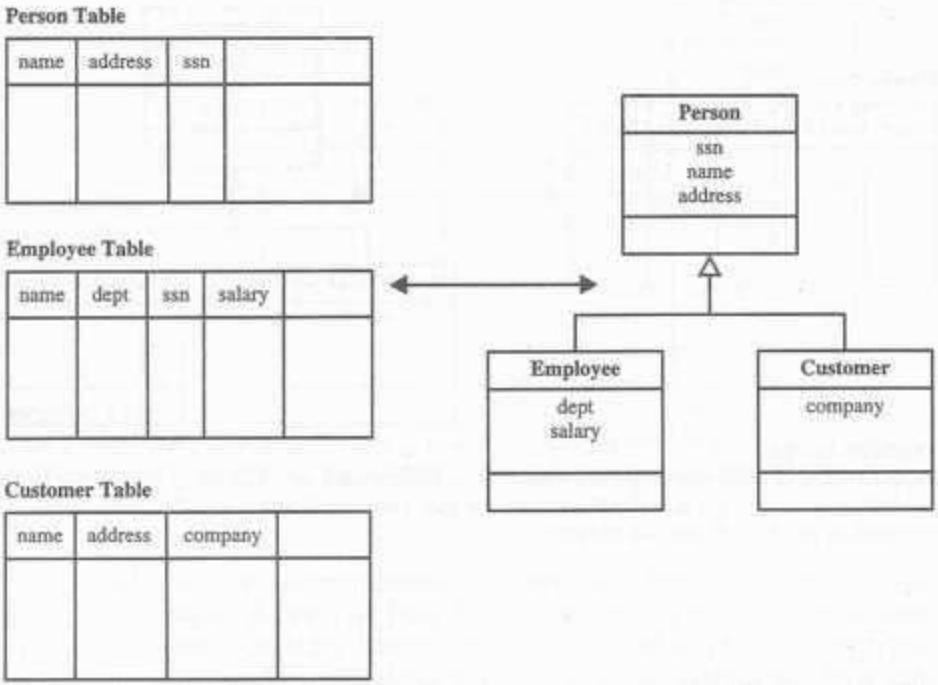


FIGURE 11–14
Tables-inherited classes mapping. Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joining of the Employee and Person tables on the ssn key). The ssn is used both as a primary key on the Person table and as a foreign key on the Person table and a primary key on the Employee table for activating instances of type Employee.

In Figure 11–15, the departmentID property of Employee uses the foreign key in column Employee.departmentID. Each Employee instance has a direct reference of class Department (association) to the department object to which it belongs.

A popular mechanism in relational databases is the use of stored procedures. As mentioned earlier, stored procedures are modules of precompiled SQL code stored in the database that execute on the server to enforce rules the business has set about the data. Mapping should support the use of stored procedures by allowing mapping of existing stored procedures to object methods.

11.9 MULTIDATABASE SYSTEMS

A different approach for integrating object-oriented applications with relational data environments is multidatabase systems or heterogeneous database systems, which facilitate the integration of heterogeneous databases and other information sources.

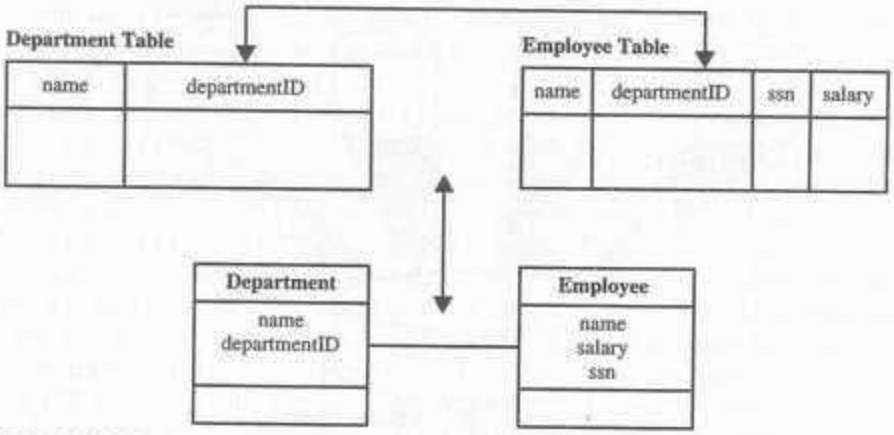


FIGURE 11-15
Class instance relationship.

Heterogeneous information systems facilitate the integration of heterogeneous information sources, where they can be structured (having regular schema), semi-structured, and sometimes even unstructured. Some heterogeneous information systems are constructed on a global schema over several databases. This way users can have the benefits of a database with a schema (i.e., uniform interfaces, such as an SQL-style interface) to access data stored in different databases and cross-database functionality. Such heterogeneous information systems are referred to as *federated multidatabase systems* [9].

Federated multidatabase systems, as a general solution to the problem of interoperating heterogeneous data systems, provide uniform access to data stored in multiple databases that involve several different data models. A ***multidatabase system*** (MDBS) is a database system that resides unobtrusively on top of, say, existing relational and object databases and file systems (called *local database systems*) and presents a single database illusion to its users (see Figure 11–16). In particular, an MDBS maintains a single global database schema against which its users will issue queries and updates; an MDBS maintains only the global schema, and the local database systems actually maintain all user data. The global schema is constructed by consolidating (integrating) the schemata of the local databases; the schematic differences (conflicts) among them are handled by ***neutralization*** (***homogenization***), the process of consolidating the local schemata.

The MDBS translates the global queries and updates for dispatch to the appropriate local database system for actual processing, merges the results from them, and generates the final result for the user. Further, the MDBS coordinates the committing and aborting of global transactions by the local database systems that processed them to maintain the consistency of the data within the local databases. An MDBS actually controls multiple gateways (or drivers). It manages local databases through the gateways, one gateway for each local database.

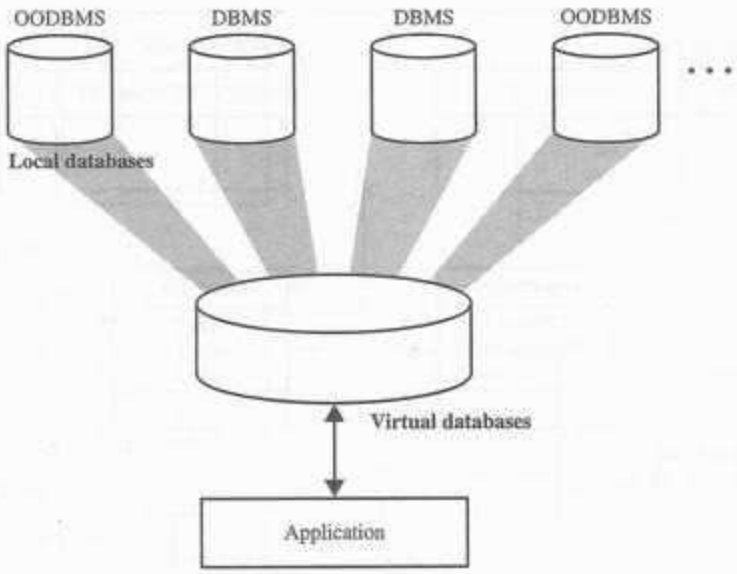


FIGURE 11–16
A multidatabase system (MDBS) is a database system that resides on top of, say existing relational and object databases and file systems (called local database systems) and presents a single database illusion to its users. In other words, users are under an impression that they are working with a single database.

To summarize the distinctive characteristics of multidatabase systems,

- Automatic generation of a unified global database schema from local databases, in addition to schema capturing and mapping for local databases.
- Provision of cross-database functionality (global queries, updates, and transactions) by using unified schemata.
- Integration of heterogeneous database systems with multiple databases.
- Integration of data types other than relational data through the use of such tools as driver generators.
- Provision of a uniform but diverse set of interfaces (e.g., an SQL-style interface, browsing tools, and C++) to access and manipulate data stored in local databases [9].

11.9.1 Open Database Connectivity: Multidatabase Application Programming Interfaces

The benefits of being able to port database applications by writing to an application programming interface (API) for a virtual DBMS are so appealing to software developers that the computer industry recently introduced several multidatabase APIs. Developers use these call-level interfaces for applications that access multiple databases using a single set of function calls, minimizing differences in application source code [10]. Open database connectivity (ODBC) is an application programming interface that provides solutions to the multidatabase programming

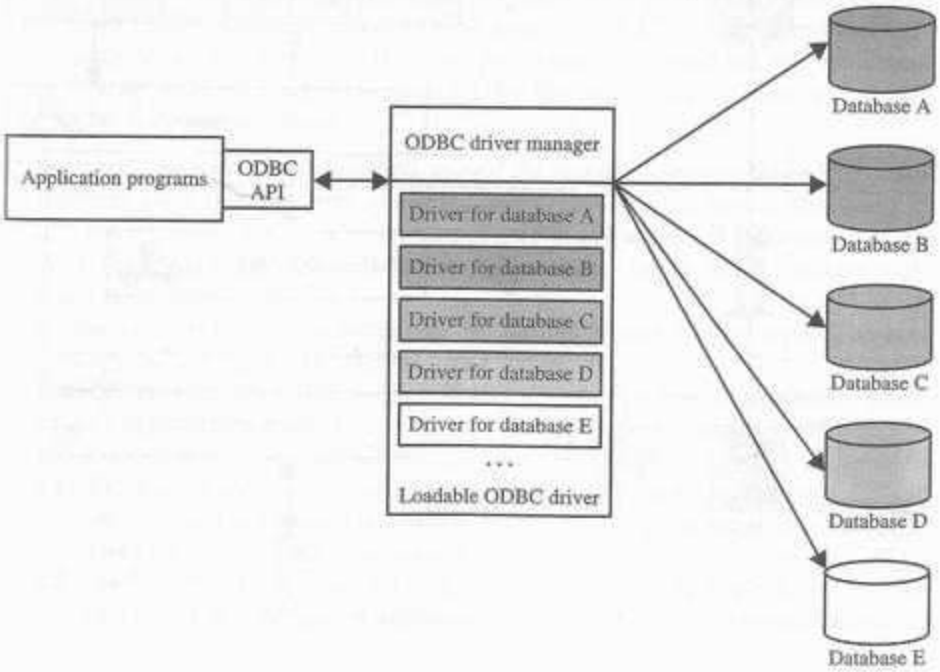
problem. Initially proposed by Microsoft, ODBC provides a vendor-neutral mechanism for independently accessing multiple database hosts.

ODBC and the other APIs provide standard database access through a common client-side interface. It thus allows software developers to write desktop applications without the burden of learning multiple database APIs. Another ODBC advantage is the ability to store data for various applications or data from different sources in any database and transparently access or combine the data on an as-needed basis. Details of the back-end data structure are hidden from the user.

As a standard, ODBC has strong industry support. Currently, a majority of software and hardware vendors, including both Microsoft and Apple, have endorsed ODBC as the database interoperability standard. In addition, most database vendors either provide or will soon provide ODBC-compliant interfaces.

ODBC is conceptually similar to the Windows print model, where the application developer writes to a generic printer interface and a loadable driver maps that logic to hardware-specific commands. This approach virtualizes the target printer or DBMS because the person with the specialized knowledge to make the application logic work with the printer or database is the driver developer and not the application programmer. The application interacts with the ODBC driver manager, which sends the application calls (such as SQL statements) to the database. The driver manager loads and unloads drivers, performs status checks, and manages multiple connections between applications and data sources (see Figure 11–17).

FIGURE 11–17
Open database connectivity (ODBC) provides a mechanism for creating a virtual DBMS.



11.10 DESIGNING ACCESS LAYER CLASSES

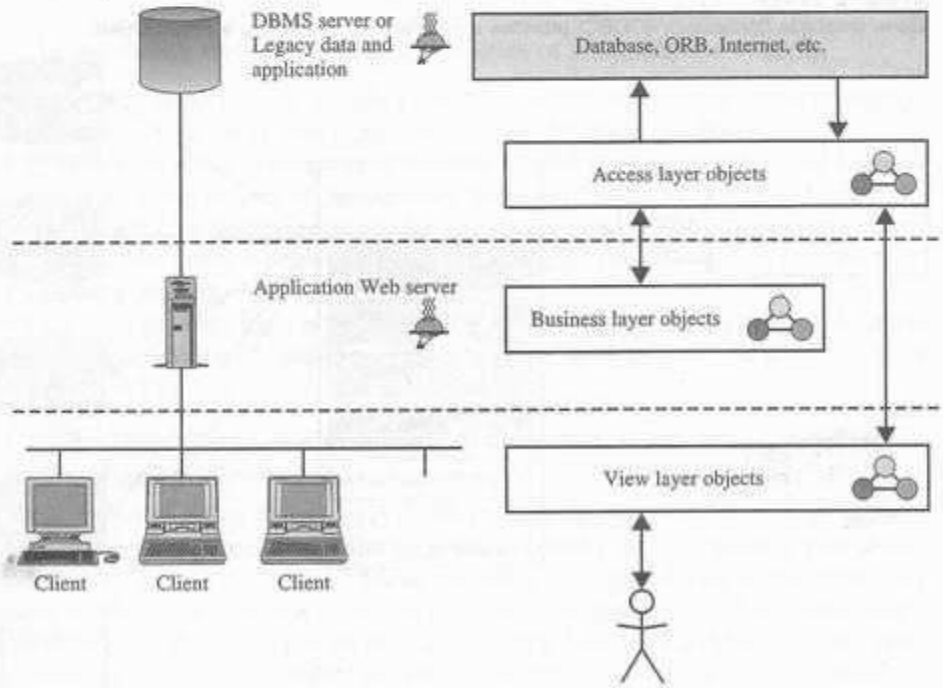
Now that we studied DBMS, client-server, distributed objects, OODBMS relational-object systems, multidatabases, and other related technologies, we have a better appreciation for why we need an access layer.

The main idea behind creating an access layer is to create a set of classes that know how to communicate with the place(s) where the data actually reside. Regardless of where the data actually reside, whether it be a file, relational database, mainframe, Internet, DCOM, or via ORB, the access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. Furthermore, these classes also must be able to translate the data retrieved back into the appropriate business objects. The access layer's main responsibility is to provide a link between business or view objects and data storage. Three-layer architecture, in essence, is similar to three-tier architecture. For example, the view layer corresponds to the client tier, the business layer to the application server tier, and the access layer to the database tier of three-tier architecture (see Figure 11–18).

The access layer performs two major tasks:

1. *Translate the request.* The access layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data

FIGURE 11–18
The business layer objects and view layer objects should not directly access the database. Instead, they should consult with the access layer for all external system connectivity.



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.)

2. *Translate the results.* The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back into the business layer.

The main advantage of this approach is that the design is not tied to any database engine or distributed object technology, such as CORBA or DCOM. With this approach, we very easily can switch from one database to another with no major changes to the user interface or business layer objects. All we need to change are the access classes' methods. Other benefits of access layer classes are these:

- Access layer classes provide easy migration to emerging distributed object technology, such as CORBA and DCOM.
- These classes 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.

Designing the access layer object is the same as for business layer objects and the same guidelines apply to access layer classes, so we do not repeat them here (see Chapter 10). However, we need to deal with the following fundamental questions:

- How do we decide what access layer objects to include?
- How do access layer objects fit with business layer (or view layer) objects? Or, what is the relationship between a business class and its associated access class?

11.10.1 The Process

The access layer design process consists of the following activities (see Figures 11–19 and 11–20). If a class interacts with a *nonhuman actor*, such as another system, database, or the Web, then the class automatically should become an access class. The process of creating an access class for the business classes we identified so far follows:

1. For every business class identified, *mirror the business class package*. For every business class that has been identified and created, create one access class in the access layer package. For example, if there are three business classes (Class1, Class2, and Class3), create three access layer classes (Class1DB, Class2DB, and Class3DB).
2. *Define relationships*. The same rule as applies among business class objects also applies among access classes (see Chapter 8).
3. *Simplify classes and relationships*. The main goal here is to eliminate redundant or unnecessary classes or structures. In most cases, you can combine simple access classes and simplify the super- and subclass structures.
   - **3.1.** *Redundant classes*. If you have more than one class that provides similar services (e.g., similar *Translate request* and *Translate results*), simply select one and eliminate the other(s).
   - **3.2.** *Method classes*. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes. If

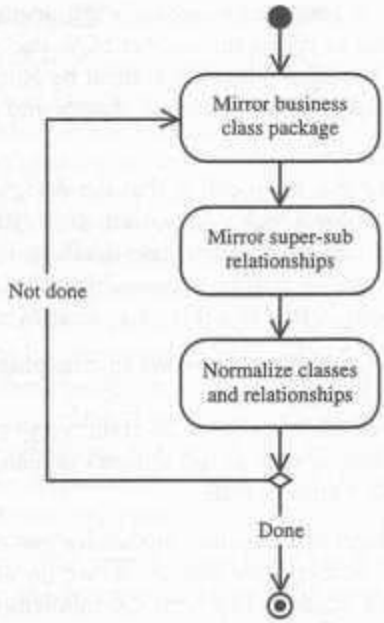


FIGURE 11-19
The process of creating access layer classes.

you can find no class from the access layer package, select its associated class from the business package and add the method(s) as a private method(s) to it. In this case, we have created an *access method.*

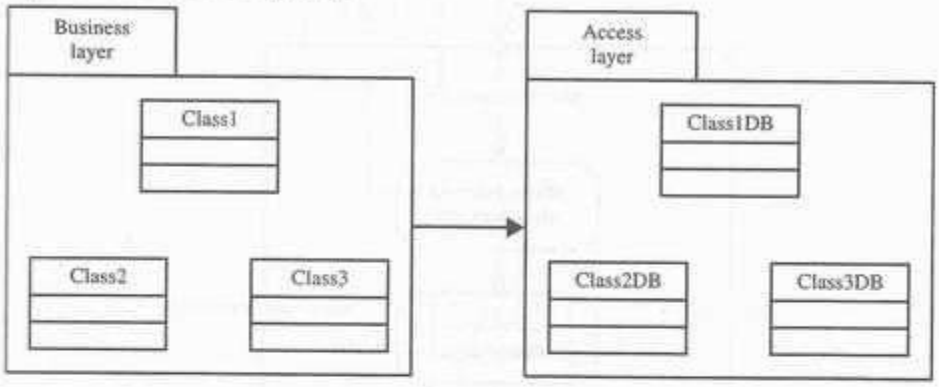
4. Iterate and refine.

In this process, the access layer classes are assumed to store not only the attributes but also the methods. This can done by utilizing an OODBMS or a relational database (as described in section 11.8.1).

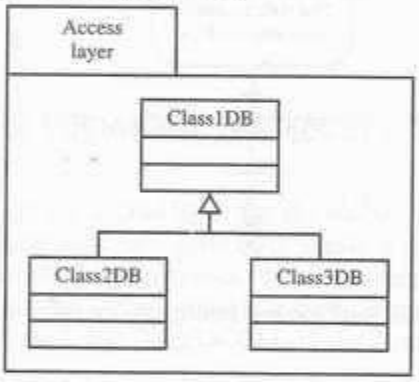
Another approach is to let the methods be stored in a program (e.g., a compiled C++ program stored on a file) and store only the *persistent attributes*. Here is the modified process:

1. For every business class identified (see Figure 11–21), *determine if the class has persistent data*. An attribute can be either transient or persistent (nontransient). An attribute is *transient* if the following condition exists: Temporary storage for an expression evaluation or its value can be dynamically allocated. An attribute is *persistent* if the following condition exists: Data must exist between executions of a program or outlive the program. If the method has any persistent attributes, go to the next step (mirror the business class package); otherwise, the class needs no associated access layer class.
2. *Mirror the business class package*. For every business class identified and created, create one access class in the access layer package. For example, if there are three business classes (Class1, Class2, and Class3), create three access layer classes (Class1DB, Class2DB, and Class3DB).

Step 1. Mirror business class package



Step 2. Define relationships among access layer class



Step 3. Simplify classes and relationships

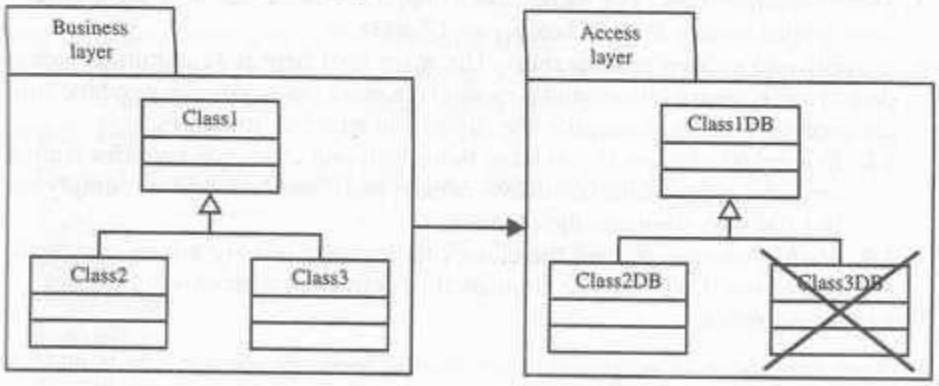


FIGURE 11-20
The process of creating access layer classes.

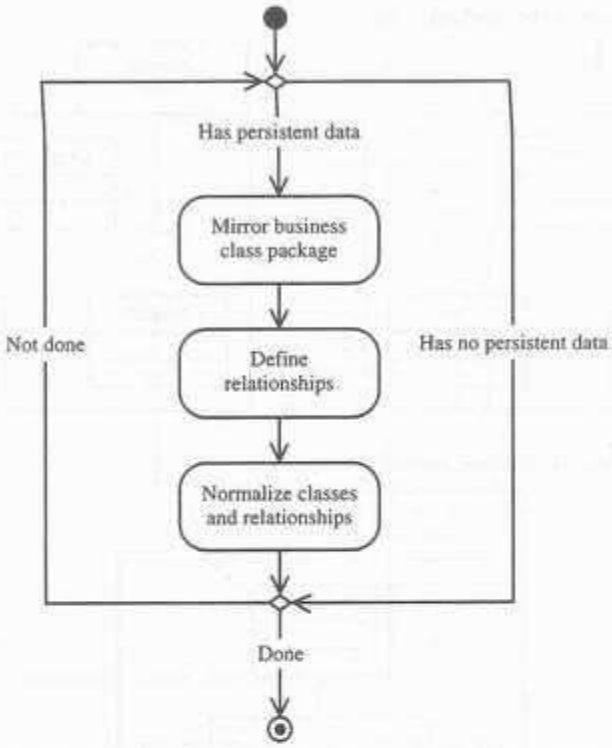


FIGURE 11–21
The process of creating access layer classes. Storing only the persistent attributes.

3. *Define relationships*. The same rule as applies among business class objects also applies among access classes (see Chapter 8).
4. *Simplify classes and relationships*. The main goal here is to eliminate redundant or unnecessary classes and structures. In most cases, you can combine simple access classes and simplify the super- and subclass structures.
   - 4.1. *Redundant classes*. If you have more than one class that provides similar services (e.g., similar *Translate request* and *Translate results*), simply select one and eliminate the other(s).
   - 4.2. *Method classes*. Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.
5. Iterate and refine.

In either case, once an access class has been defined, all you need do is make it a-part-of its business class (see Figure 11–22).

Next, we apply this process to design the access layer classes for our bank system application. To make the problem more interesting, we use a relational database for storing the objects and the second approach in designing its access layer classes, assuming the methods will be stored in the program.

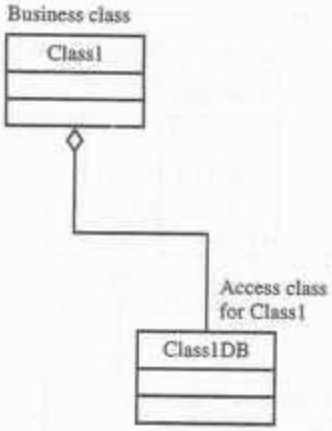


FIGURE 11-22
The relation between a business class and its associated access class.

11.11 CASE STUDY: DESIGNING THE ACCESS LAYER FOR THE VIANET BANK ATM

We are ready to develop the access layer for the ViaNet bank ATM. Remember that the main idea behind an access layer is to create a set of classes that know how to communicate with the data source. They are simply *mediators* between business or view classes and storage places or they communicate with other objects over a network through the ORB/DCOM, in the case of distributed objects.

11.11.1 Creating an Access Class for the BankClient Class

Here, we apply the access layer design process to identify the access classes.

Step 1. *Determine if a class has persistent data.*
Step 2. *Mirror the business class package.* Since the BankClient has persistent attributes, we need to create an access class for it.
Step 3. *Define relationships.*

TheBankClient class has the following attributes (see Chapter 10):

firstName
lastName
cardNumber
pinNumber
account

The firstName, lastName, cardNumber, and pinNumber are persistent attributes, and account is used to link (or implement the association among) the BankClient and Account classes. To link the BankClient table to the Account table we need to use the card number (cardNumber) as a primary key in both tables (see Figure 11-23).

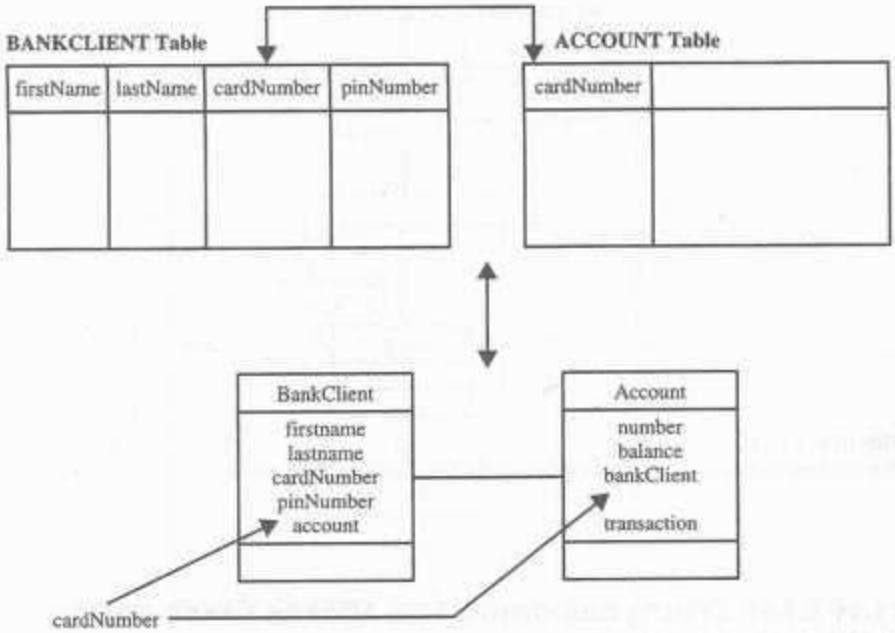


FIGURE 11-23
The cardNumber column facilitates the link between the BANKCLIENT and ACCOUNT tables. It also implements the association among the BankClient and Account classes. Note that the cardNumber is a primary key for the ACCOUNT and BANKCLIENT tables.

Here we decided to create an access class instead of creating access method within the BankClient class. Let us call our access class BankDB. The purpose of this class is to save the state of the BankClient objects. In other words, it must update and retrieve the BankClient attributes by translating any data-related requests from the BankClient class into the appropriate protocol for data access.

Notice that *retrieveClient* method of BankClient object simply sends a message to the BankDB object to get the client information:

Listing 1.

```
BankClient::+retrieveClient (aCardNumber, aPIN): BankClient
     aBankDB : BankDB
     aBankDB.retrieveClient (aCardNumber, aPIN)
```

In here, all we need to do is to create an instance of the access class BankDB and then send a message to it to get information on the client object. The *retrieveClient* of the BankDB class will do the actual work of getting the information from the database. Let us assume our database is relational and we are using SQL:

Listing 2.

```
BankDB::+retrieveClient (aCardNumber, aPIN): BankClient
SELECT firstName, lastName
    FROM BANKCLIENT
        WHERE cardNumber = aCardnumber and pinNumber = aPin)
```

The *retrieveClient* return type is defined as BankClient to return the attributes of the BankClient. Access class methods (as you might guess) are highly language dependent. Remember that in actuality, during design, you have to select your implementation language. Since implementation is beyond the scope of this book and to keep the description language independent, the implementation details have been skipped for the most part. A return of null means that the supplied PIN number is not valid.

The *updateClient* method updates or changes attributes such as pinNumber, firstName, or lastName. Here again, just like the *retrieveClient* method, the BankClient::updateClient sends a message to the access class BankDB::updateClient to update client information:

Listing 3.

```
BankDB::+updateClient (aClient: BankClient, aCardNumber: String)
UPDATE BANKCLIENT
    SET firstName = aClient.firstName
    SET lastName = aClient.lastName
    SET pinNumber = aClient.pinNumber
            WHERE cardNumber = aCardnumber)
```

The Account class has the following attributes (see Chapter 10):

number

balance

bankClient

transaction

Attributes such as number and balance are persistent. The bankClient attribute is transient and is used for implementing the association between the Account and BankClient classes. We already have taken care of this link by adding *cardNumber* to the Account table. However, to link the Account table to the Transaction table, we need to add the *transID* as a foreign key to the Account table (see Figure 11–24).

Figure 11–25 shows how generalization relationships among the Account, CheckingAccount, and SavingsAccount classes have been represented in our relational database. Here, since we are using a relational database that provides no inheritance or super-sub generalization, we added four columns to the Account table: one for the savings account number (*sNumber*), one for the checking account number (*cNumber*), one for the savings balance (*sBalance*), and finally one for the checking balance (*cBalance*).

According to step 2, we need to add three more access classes: one for the Account class (AccountDB), one for the CheckingAccount class (CheckingAccountDB), and one for the SavingsAccount class (SavingsAccountDB). However, at this point, we realize that we need an access class with only four methods, two

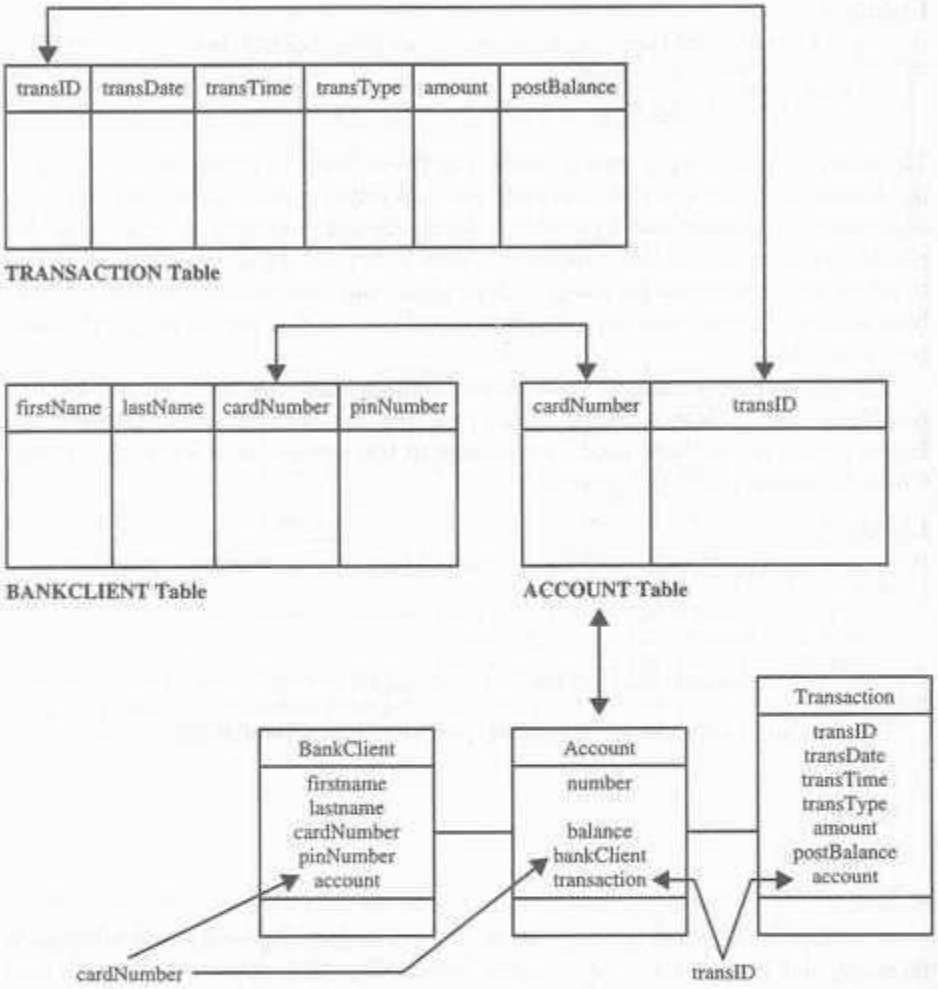


FIGURE 11-24
To represent the association between the ACCOUNT and TRANSACTION tables, we need to add transID to the ACCOUNT table as a foreign key.

methods for SavingsAccount (update and retrieve) and two methods for CheckingAccount. The Account class is an abstract class (has no instances since the accounts are either savings or checking accounts) and therefore needs no additional methods. The methods are

updateSavingsAccount
retrieveSavingsAccount
updateCheckingAccount
retrieveCheckingAccount

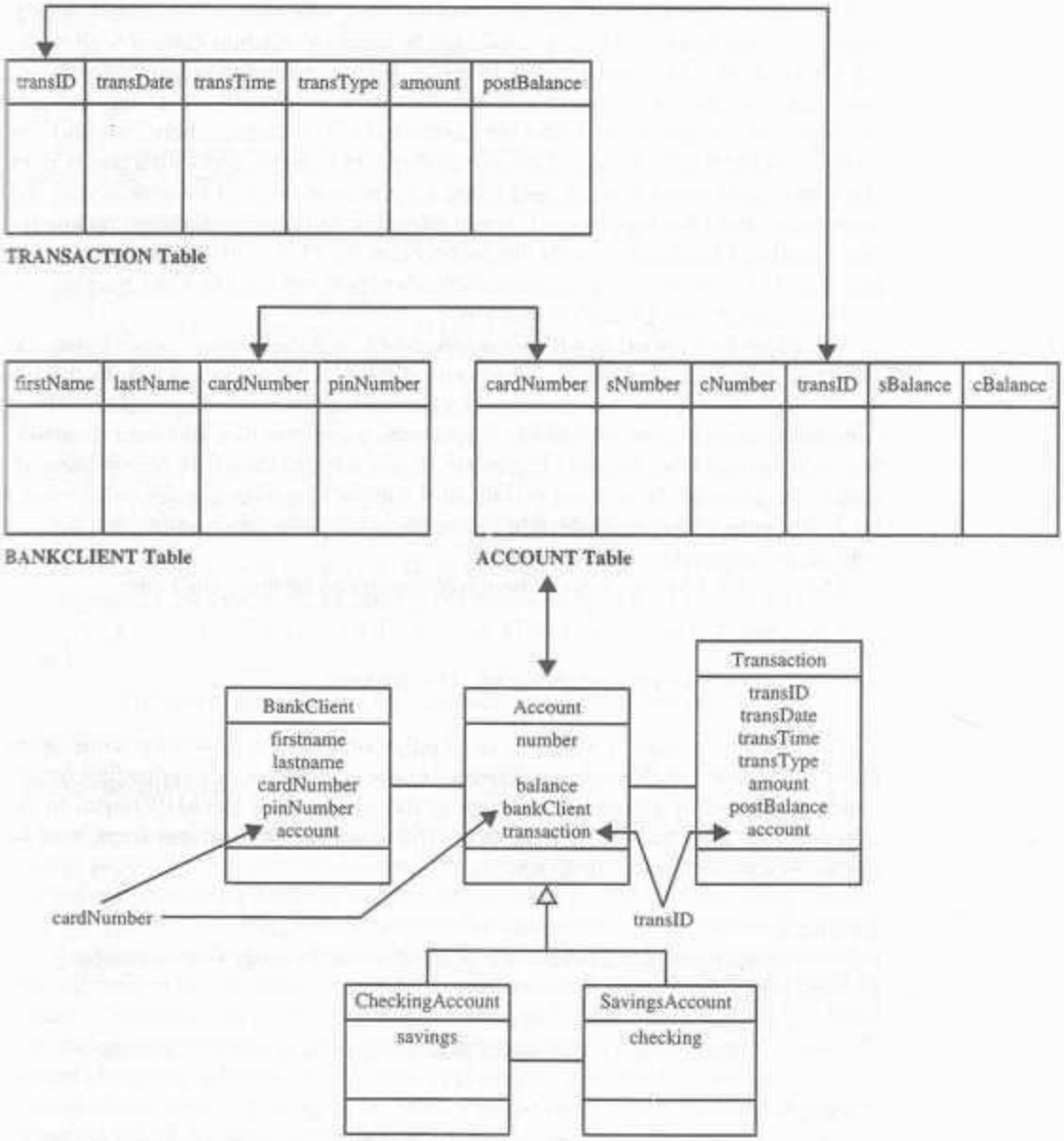


FIGURE 11-25
Four columns are added to the ACCOUNT table: one for the savings account number *(sNumber)*, one for the checking account number *(cNumber)*, one for the savings balance *(sBalance)*, and one for the checking balance *(cBalance)*. Instances of SavingsAccount and CheckingAccount can be created for any row in the ACCOUNT table that has a non null value for *sNumber* or *cNumber*.

The previous section explained that *method classes* are classes that consist of only one or two methods, and it is a good idea to combine method classes with existing access classes if it makes sense to do so. If you can find no class from the access layer package, select its associated class from the business package and create a private method(s) and add the methods to it. However, here, we add the methods to the BankDB class. After all, an Account is associated with a BankClient class and most times account and client information need to be accessed at the same time. Here *SavingsAccount::retrieveAccount* and *SavingsAccount::updateAccount* methods send messages to the access class *BankDB::retrieveSavingsAccount* and *BankDB::updateSavingsAccount* methods to perform the job. The CheckingAccount is similar (see Listings 6 and 10).

You might be wondering why *SavingsAccount::retrieveAccount* needs to use the *BankDB::retrieveSavingsAccount* to perform the job. Why does it not do the job itself? Well, *SavingsAccount::retrieveAccount* can perform this operation without calling the access class. However, if you need to switch to a different database, you must modify the method. Therefore, if you want to create an access method, make sure to make its protocol private so the impact on other classes will be minimal. For most cases, it is possible to use the access class instead of creating private access methods.

The following listing depicts the implementations of these methods:

Listing 4.

```
SavingsAccount::-retrieveAccount (): Account
bankDB.retrieveSavingsAccount (bankClient.cardNumber, number)
```

The *retrieveAccount* is an example of polymorphism (in which the same operation may behave differently on different classes), where it is overloading its superclass method by sending a message to the access class BankDB object to retrieve the savings account information. The same mechanism has been used to invoke other access class methods:

Listing 5.

```
BankDB::+retrieveSavingsAccount (aCardNumber: String, savingsNumber:
String): Account
SELECT sBalance, transID,
     FROM ACCOUNT
          WHERE cardNumber = aCardNumber and sNumber = savingsNumber)
```

Listing 6.

```
CheckingAccount::-retrieveAccount (): Account
bankDB.retrieveCheckingAccount (bankClient.cardNumber, number)
```

Listing 7.

```
BankDB::+retrieveCheckingAccount (aCardNumber: String, checkingNumber:
String): Account
SELECT cBalance, transID,
     FROM ACCOUNT
          WHERE cardNumber = aCardNumber and cNumber = checkingNumber)
```

Listing 8.

```
SavingsAccount::-updateAccount():Account
bankDB.updateSavingsAccount (bankClient.cardNumber, number, balance)
```

Listing 9.

```
BankDB::+updateSavingsAccount (aCardPinNumber: String, aNumber: String
newBalance: float)
UPDATE ACCOUNT
    Set sBalance = newBalance
        WHERE cardNumber = aCardNumber and sNumber = aNumber)
```

Listing 10.

```
CheckingAccount::-updateAccount():Account
bankDB.updateCheckingAccount (bankClient.cardNumber, number, balance)
```

Listing 11.

```
BankDB::+updateCheckingAccount (aCardPinNumber: String, aNumber:
String newBalance: float)
UPDATE ACCOUNT
    Set cBalance = newBalance
        WHERE cardNumber = aCardNumber and cNumber = aNumber)
```

Figure 11–26 depicts the relationships among the classes we have designed so far, especially the relationships among the access class and other business classes. Designing an access class for the Transaction class is left as an exercise; see problem 1.

11.12 SUMMARY

A database management system (DBMS) is a collection of related data and associated programs that access, manipulate, protect, and manage data. The fundamental purpose of a DBMS is to provide a reliable persistent data storage facility and the mechanisms for efficient, convenient data access and retrieval.

Many modern databases are distributed databases. This implies that portions of the database reside on different nodes (computers) and disk drives in the network. Usually, each portion of the database is managed by a server, a process responsible for controlling access and retrieval of data from the database portion. The server dispenses information to client applications and makes queries or data requests to the servers. Clients generally reside on nodes in the network other than those on which the servers execute.

Client-server computing is the logical extension of modular programming. The fundamental assumption of modular programming is that separation of a large piece of software into its constituent parts ("modules") creates the possibility for easier development and better maintainability.

Distributed computing is poised for a second client-server revolution, a transition to an immensely expanded client-server era. In this new client-server model, servers are plentiful instead of scarce (because every client can be a server) and

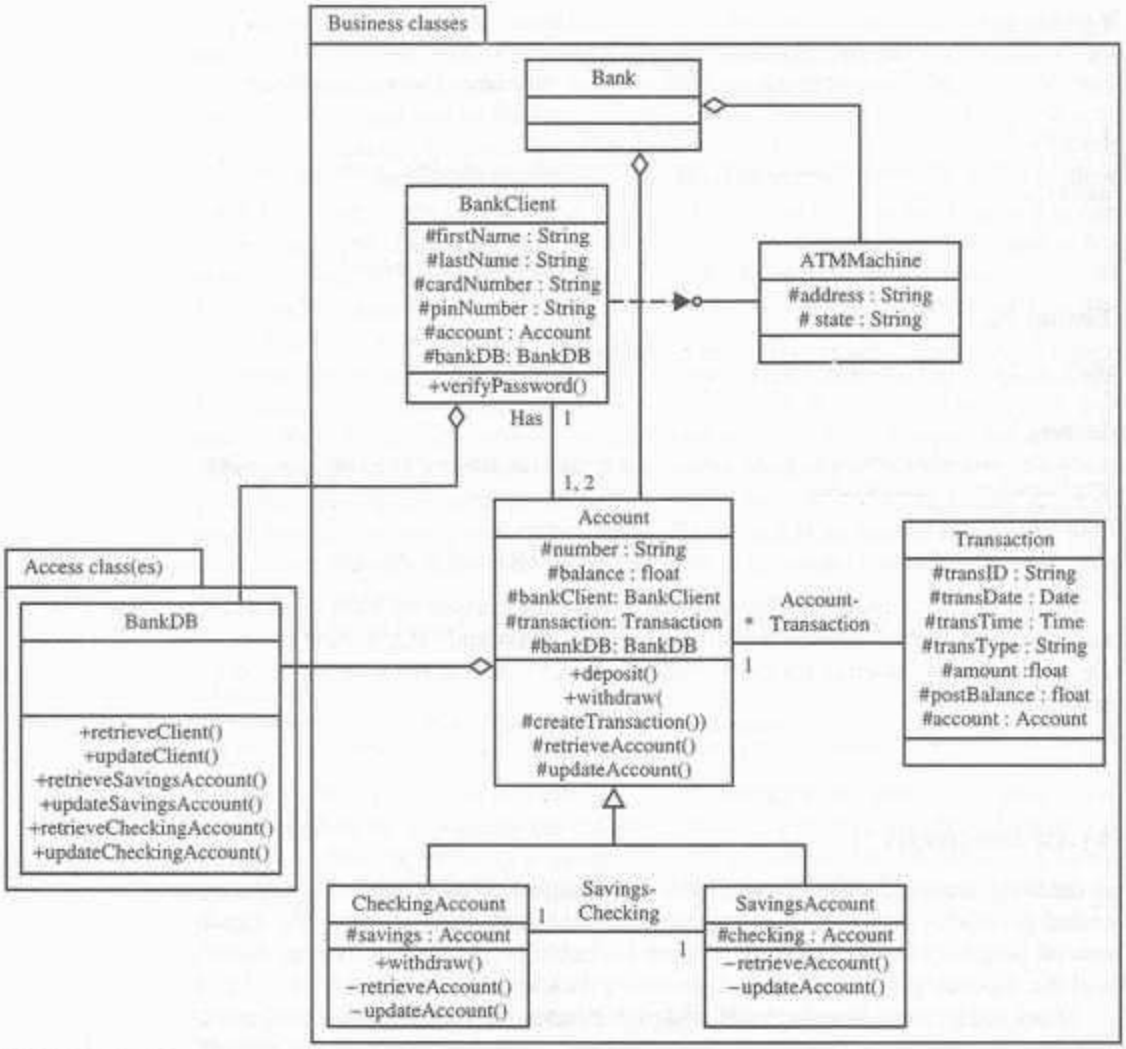


FIGURE 11-26
A still more complete UML class diagram of the ViaNet Bank ATM system. It shows the relationship of the new access class (BankDB) with the Account and BankClient business classes. Note the addition of the bankDB attributes to the Account and BankClient classes and addition of four new private methods to checkingAccount and SavingsAccount classes.

proximity no longer matters. The new generation of the client-server model is made possible by the recent exponential network growth and the progress in network-aware multithreaded desktop operating systems.

The object-oriented database technology is a marriage of object-oriented programming and database technology. The programming and database come together to provide what we call *object-oriented databases*. By combining object-oriented

programming with database technology, we have an integrated application development system, a significant characteristic of object-oriented database technology. "The Object-Oriented Database System Manifesto" by Malcom Atkinson et al. describes the necessary characteristics a system must satisfy to be considered an object-oriented database. These categories can be broadly divided into object-oriented language properties and database requirements.

In practice, even though many applications increasingly are developed using object-oriented programming technology, chances are good that the data those applications need to access live in a very different universe—a relational database. To resolve such a mismatch, the application objects and the relational data must be mapped. Tools that can be used to establish the object-relational mapping processes have begun to emerge. The main process in relational-object integration is defining the relationships between the table structures (represented as schemata) in the relational database with classes (representing classes) in the object model.

A different approach for integrating object-oriented applications with relational data environments involves multidatabase systems or heterogeneous database systems, which facilitate the integration of heterogeneous databases and other information sources.

The main idea behind an access layer is to create a set of classes that know how to communicate with a data source. Regardless of whether the data actually are in a file, relational database, mainframe, or Internet, the access classes must be able to translate 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.

KEY TERMS

Abort (p. 244)
Atomic type of objects (p. 253)
Atomicity (p. 244)
Commit (p. 244)
Common object request broker architecture (CORBA) (p. 251)
Data definition language (DDL) (p. 242)
Data manipulation language (DML) (p. 242)
Database management system (DBMS) (p. 237)
Distributed component object model (DCOM) (p. 252)
Distributed database (p. 245)
Distributed object computing (DOC) (p. 250)
Foreign key (p. 241)
Forward engineering (p. 255)
Homogenization (p. 261)
Meta-data (p. 239)

Multidatabase system (MDBS) (p. 261)
Neutralization (p. 261)
Object management group (OMG) (p. 251)
Object-oriented database management system (OODBMS) (p. 252)
Object request broker (ORB) (p. 251)
Persistence (p. 237)
Primary key (p. 241)
Referential integrity (p. 257)
Reverse engineering (p. 255)
Schema (p. 239)
Stored procedure (p. 257)
Structured query language (SQL) (p. 243)
Transaction (p. 244)
Tuples (p. 241)

REVIEW QUESTIONS

1. How do you distinguish transient data from persistent data?
2. What is a DBMS?
3. Is a persistent object the same as a DBMS? What are the differences?
4. What is a relational database? Explain tuple, primary key, and foreign key.
5. What is a database schema? What is the difference between a schema and meta-data?
6. What is a DDL?
7. What a distributed database?
8. What is concurrency control?
9. What is shareability?
10. What is a transaction?
11. What is a concurrency policy?
12. What is a query?
13. Describe client-server computing.
14. What are different types of servers? Briefly describe each.
15. Why do you think DOC is so important in the computing world?
16. Describe CORBA, ORB, and DCOM.
17. What is an OODBMS? Describe the differences between an OODBMS and object-oriented programming.
18. Describe the necessary characteristics that a system must satisfy to be considered an object-oriented database.
19. Describe reverse and forward engineering.
20. Describe a federated multidatabase system.
21. Describe the process of creating the access layer classes.

PROBLEMS

1. Design an access class for the Transaction class of the bank system. Try both alternatives and write a paragraph pro or con for each design. (Design it once as an access class, the second time as access methods. Compare these approaches and report on their similarities and differences.)

2. Consult the WWW to obtain information on DOC, especially comparing CORBA with DCOM. Write a research paper based on your findings.
3. Consult the WWW to obtain information on the object-relational systems and tools. Select one of the development tools, and write on your rationale for selecting that tool.
4. Consult the WWW to obtain information on OODBMS vendors. Select one of the development tools and write on your rationale for selecting that tool.
5. Consult the WWW or the library to obtain an article on objected-oriented DML and query languages. Write a paper based on your findings.
6. Consult the WWW or the library to obtain an article on the Web objects. Write a paper based on your findings.
7. Consult the WWW or the library to obtain an article on ActiveX. Write a paper based on your findings.
8. Some developments in SQL technology involve the integration of object-oriented features into mainstream commercial databases. Despite the growing number of object databases available today, there is no commercial SQL standard to create and access objects stored in such databases. OQL (object query language) and SQL3 are two separate but overlapping efforts to merge object database technology with standardized, next-generation query languages. Do research to find out more about SQL3 and OQL.
9. The ViaNet bank system wants to go on-line and create on-line banking, where customers can be connected electronically to the bank through the Internet and should be able to conduct almost the same banking transactions as they would with a regular ATM machine. The only variation from previous requirements is that they cannot withdraw cash from the ATM machine, but instead they can write electronic checks to a payee. Design the architecture for the on-line banking of the ViaNet bank.

REFERENCES

1. Atkinson, M. P.; Bailey, P. J.; Chrisholm, K. J.; Cockshott, W. P.; and Morrison, R. "An Approach to Persistent Programming." *Computer Journal* 26, no. 4 (1983), pp. 360–65.
2. Atkinson, M.; Bancilhon, F.; DeWilt, D; Dittrich, K; Maier, D.; and Zdonik. "The Object-Oriented Database System Manifesto." In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989, pp. 223–40.
3. Berson, Alex. *Client/Server Architecture*. New York: McGraw-Hill, 1992.
4. Brown, A. L.; and Morrison, R. "A Generic Persistent Object Store." *Software Engineering Journal* 7, no. 2 (1992), pp. 161–68.
5. Dittrich, Klaus R. "Object-Oriented Database Systems: The Notion and Issues." Proceedings of the 1986 IEEE International Workshop on Object-Oriented Database Systems.
6. Kalakota, Ravi. comp.client-server FAQ, 1996.
7. Kalakota, Ravi; and Whinston, Andrew. *The Frontiers of Electronic Commerce*. Reading, MA: Addison-Wesley, 1995.
8. Kim, Won. *Object-Oriented Database*. Cambridge, MA: Massachusetts Institute of Technology Press, 1990.
9. Lee, Juhnyoung; and Forslund, David. "Coexistence of Relations and Objects in Distributed Object Computing." White paper, Sunrise (July 26, 1995).
10. North, Ken. "Understanding Multidatabase APIs and ODBC." *DBMS* (June 1994).

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).

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

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.

BOX 12.1 (CONTINUED)

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.

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.

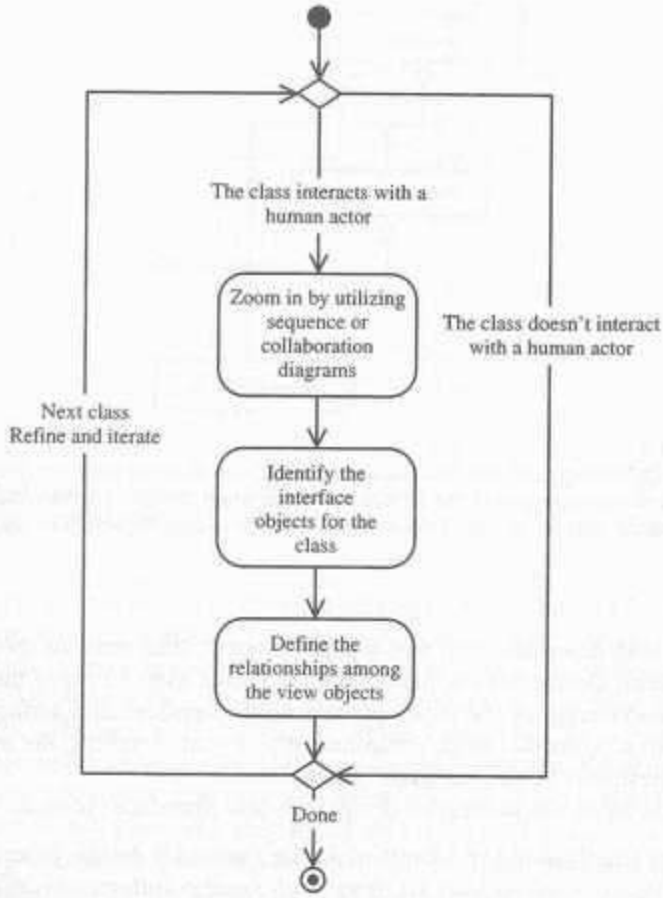


FIGURE 12-1
The macro-level design process.

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

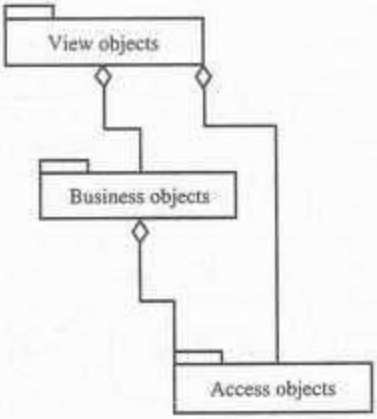


FIGURE 12–2
The relationships among business, access, and view objects. In some situations the view class can become a direct aggregate of the access object, as when designing a Web interface that must communicate with an application/Web server through access objects. See also Figure 11–18.

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.

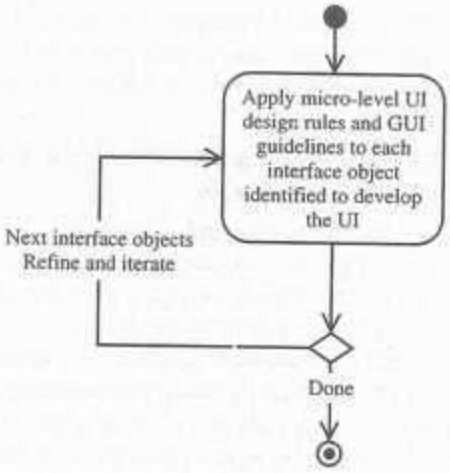


FIGURE 12-3
The micro-level design process.

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 re-

flect 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.

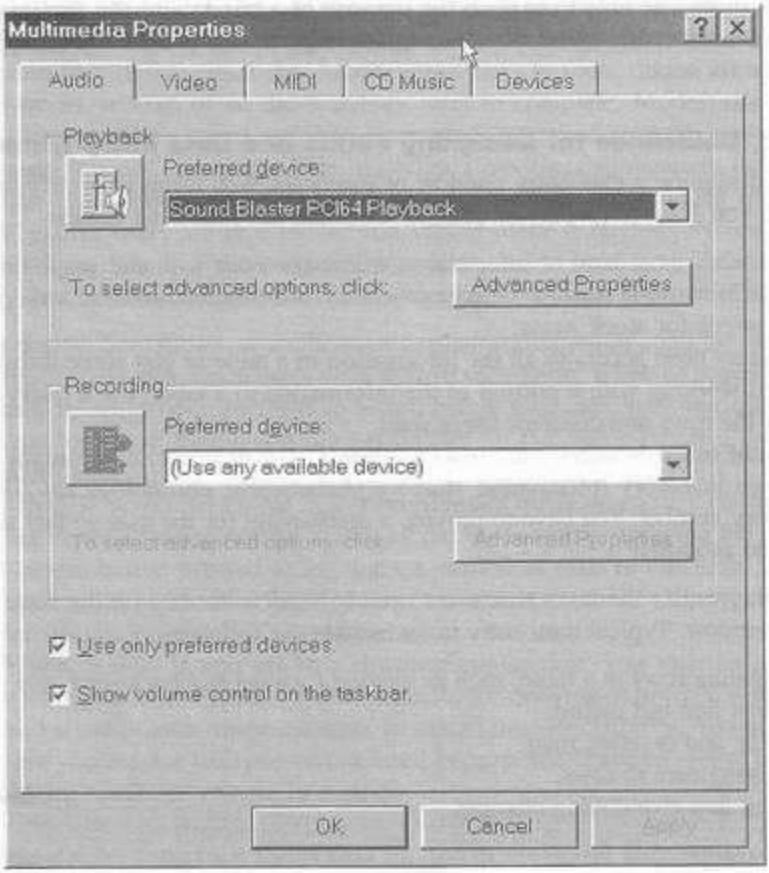


FIGURE 12–4
An example of a dialog box with multiple pages in the Microsoft multimedia setup.

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).

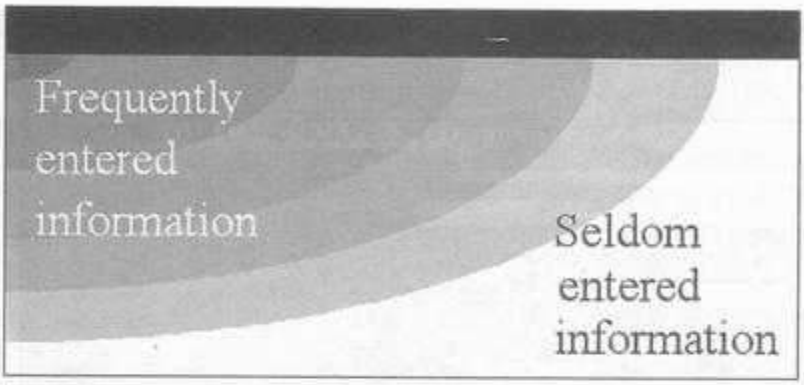
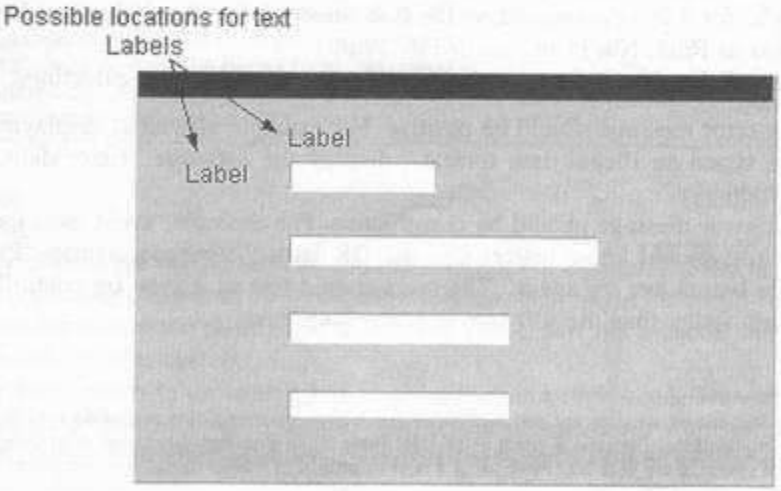


FIGURE 12–5
Required information should be put toward the top and left side of the form, entering optional or seldom entered information toward the bottom.

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.



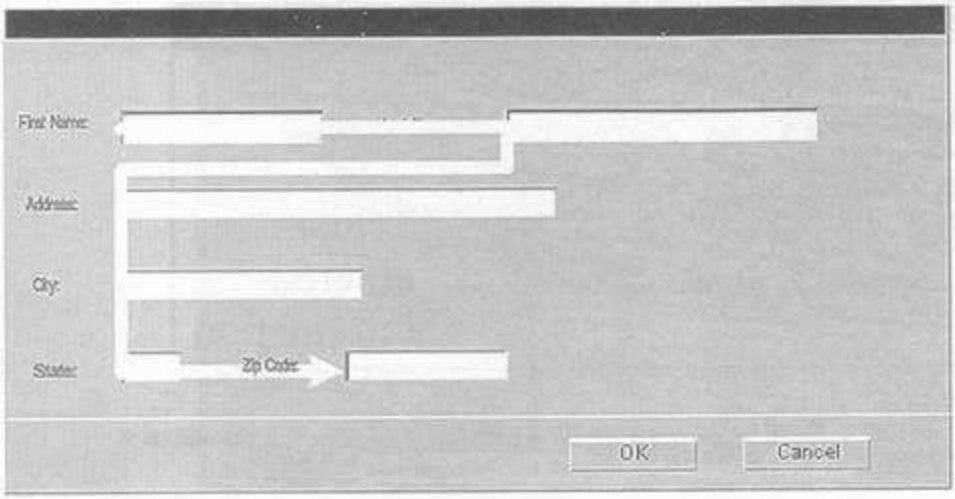


FIGURE 12-7
Arrange controls left to right and top to bottom.

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.

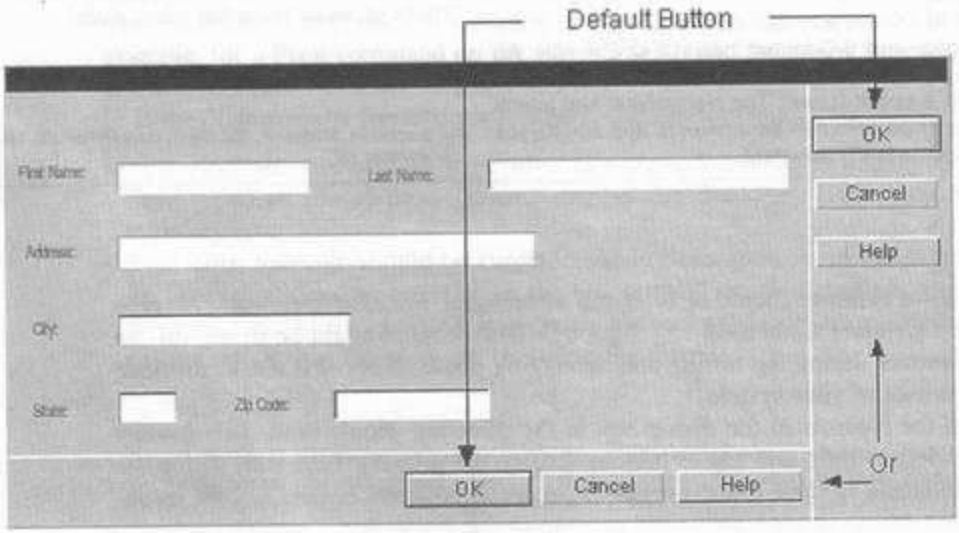
12.6.3 Guidelines for the Command Buttons Layout

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.



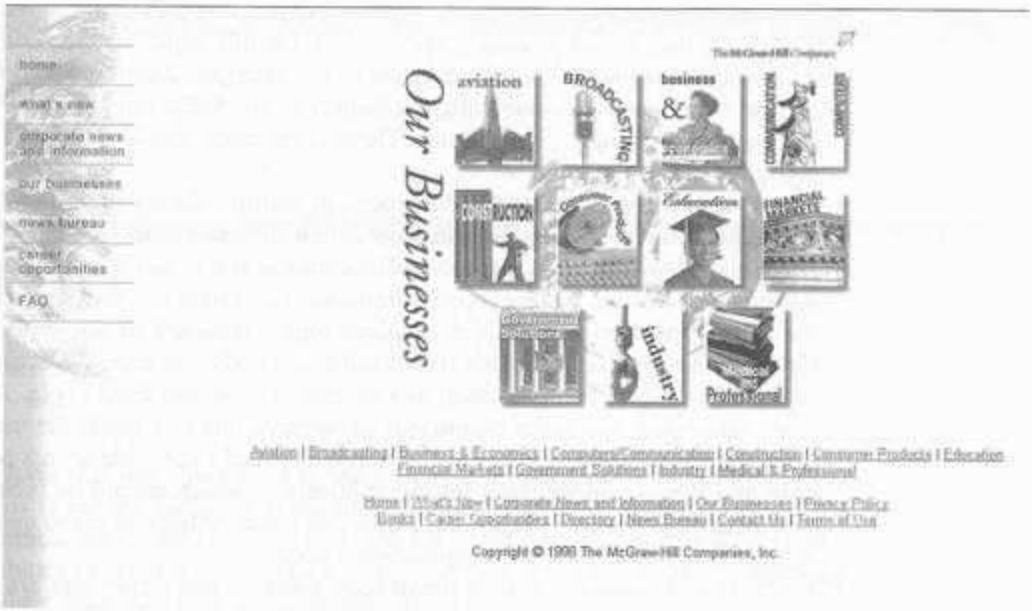


FIGURE 12–9
Positioning buttons on the left is popular in Web interfaces.

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 ve-

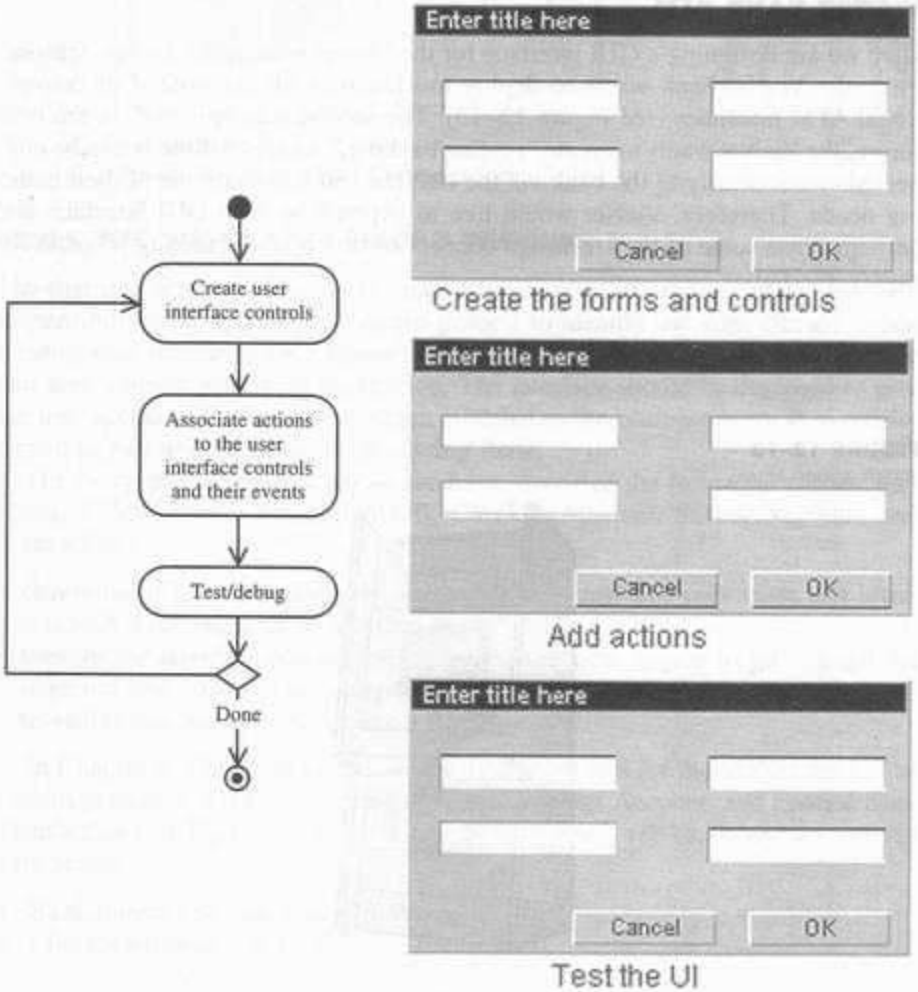
[3] System Architect Screen Painter can be used to prototype Windows screens and menus.

hicle for getting user input on a design. This is particularly useful early in the design process.

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.



When you complete the first prototype, meet with the user for an exchange of ideas about how the system would go together. When approaching the user, describe your prototype briefly. The main purpose should be to spark ideas. The user should not feel that you are imposing or even suggesting this design. You should be very positive about the user's system and wishes. Instead of using leading phrases like "we could do this . . ." or "It would be easier if we . . .," choose phrases that give the user the feeling that he or she is in charge. Some example phrases are [5]:

"Do you think that if we did . . . it would make it easier for the users?"

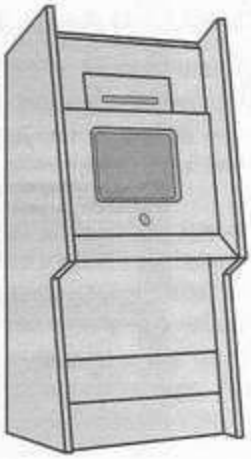
"Do users ever complain about . . .? We could add . . . to make it easier."

This cooperative approach usually results in more of your ideas being used in the end.

12.8 CASE STUDY: DESIGNING USER INTERFACE FOR THE VIANET BANK ATM

Here we are designing a GUI interface for the ViaNet bank ATM for two reasons. First, the ViaNet bank wants to deploy touch-screen kiosks instead of conventional ATM machines (see Figure 12–13). The second reason is that, in the near future, the ViaNet wants to create "on-line banking," where customers can be connected electronically to the bank via the Internet and conduct most of their banking needs. Therefore, ViaNet would like to experiment with GUI interface and perhaps reuse some of the UI design concept for the on-line banking project (see Figure 12–14).

FIGURE 12–13
Touch screen kiosk.



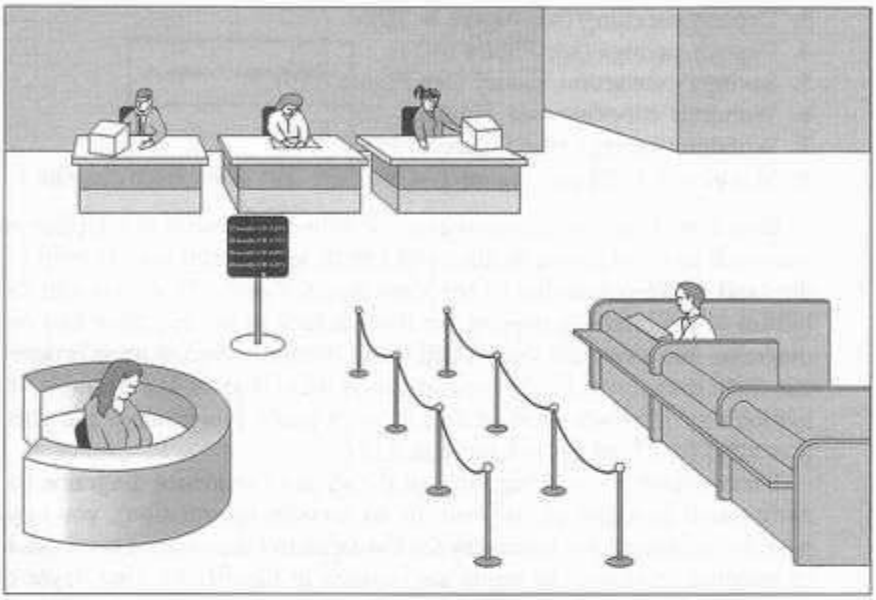


FIGURE 12–14
An example of an on-line banking project.

12.8.1 THE VIEW LAYER MACRO PROCESS

The first step here is to identify the interface objects, their requirements, and their responsibilities by applying the macro process to identify the view classes. When creating user interfaces for a business model, it is important to remember the role that view objects play in an application. The interface should be designed to give the user access to the business process modeled in the business layer. It is not designed to perform the business processing itself.

For every class identified (so far we have identified the following classes: Account, ATMMachine, Bank, BankDB, CheckingAccount, SavingsAccount, and Transaction),

- *Determine if the class interacts with a human actor.* The only class that interacts with a human actor is ATMMachine.
- *Identify the interface objects for the class.* The next step is to go through the sequence and collaboration diagrams to identify the interface objects, their responsibilities, and the requirements for this class.

In Chapter 6, we identified the scenarios or use cases for the ViaNet bank. The various scenarios involve Checking Account, Savings Account, and general bank Transaction (see Figures 6–9, 6–10, and 6–11). These use cases interact directly with actors:

1. Bank transaction (see Figure 6–9).
2. Checking transaction history (see Figure 6–9).

3. Deposit checking (see Figure 6–10).
4. Deposit savings (see Figure 6–11).
5. Savings transaction history (see Figure 6–9).
6. Withdraw checking (see Figure 6–10).
7. Withdraw savings (see Figure 6–11).
8. Valid/invalid PIN (see Figure 7–4, we have only a sequence diagram for this one).

Based on these use cases, we have identified eight view or interface objects. The sequence and collaboration diagrams can be very useful here to help us better understand the responsibility of the view layers objects. To understand the responsibilities of the interface objects, we need to look at the sequence and collaboration diagrams and study the events that these interface objects must process or generate. Such events will tell us the makeup of these objects. For example, the PIN validation user interface must be able to get a user's PIN number and check whether it is valid (see Figures 7–4 through 7–8).

Furthermore, by walking through the steps of sequence diagrams for each scenario (such as withdraw, deposit, or an account information), you can determine what view objects are necessary for the steps to take place. Therefore, the process of creating sequence diagrams also assists in identifying view layer classes and even understanding their relationships.

- *Define relationships among the view (interface) objects.* Next, we need to identify the relationships among these view objects and their associated business classes.

  So far, we have identified eight view classes:

  AccountTransactionUI(for a bank transaction)
  CheckingTransactionHistoryUI
  SavingsTransactionHistoryUI
  BankClientAccessUI (for validating a PIN code)
  DepositCheckingUI
  DepositSavingsUI
  WithdrawCheckingUI
  WithdrawSavingsUI

The three transaction view objects—AccountTransactionUI, CheckingTransactionHistoryUI, and SavingsTransactionHistoryUI—basically do the same thing, display the transaction history on either a checking or savings account. (To refresh your memory, look at Figures 11–23 through 11–25 to see how we implemented this for object storage and the access class). Therefore, we need only one view class for displaying transaction history, and let us call it *AccountTransactionUI*.

The AccountTransactionUI view class is the account transaction interface that displays the transaction history for both savings and checking accounts. Figure 12–15 depicts the relation among the AccountTransactionUI and the account class. The relationship between the view class and business object is opposite of that between business class and access class. As said earlier, the interface object handles all communication with the user but processes no business rules and lets that work

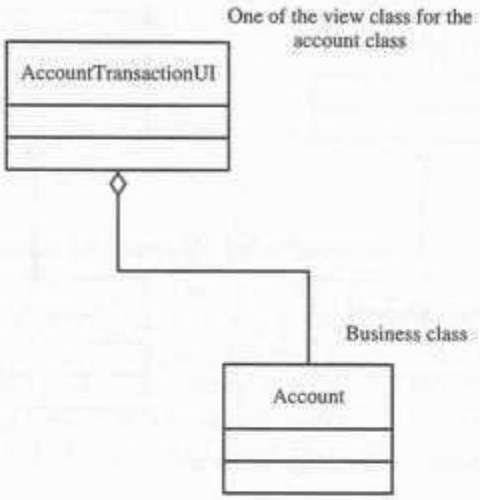


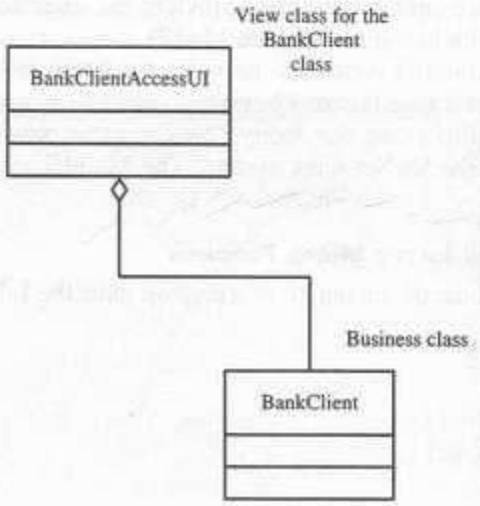
FIGURE 12–15
Relation between the view class AccountTransactionUI and its associated business class (Account).

be done by the business objects themselves. In this case, the account class provides the information to AccountTransactionUI for display to the users.

The BankClientAccessUI view class provides access control and PIN code validation for a bank client (see Figure 12–16).

The four remaining view objects are the DepositCheckingUI view class (interface for deposit to checking accounts), DepositSavingsUI view class (interface for deposit to savings accounts), WithdrawSavingsUI view class (interface for with-

FIGURE 12–16
Relation between the view class (BankClientAccessUI) and its associated business class (BankClient).



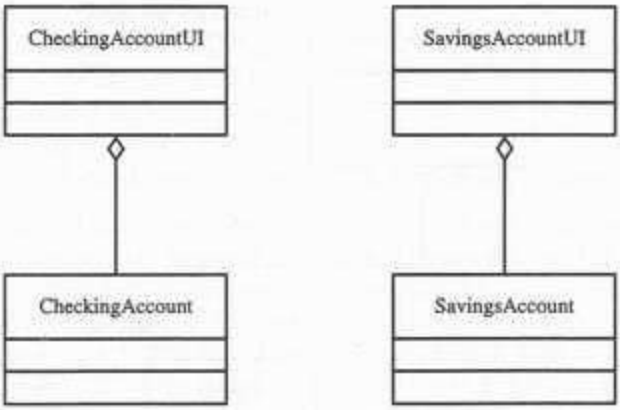


FIGURE 12–17
The view classes for checking and savings accounts.

drawal from savings accounts), and WithdrawCheckingUI view class (interface for withdrawal from savings accounts).

- *Iterate and refine.* This is the final step. Through the iteration and refinement process, we notice that the four classes DepositCheckingUI, DepositSavingsUI, WithdrawSavingsUI, and WithdrawCheckingUI basically provide a single service, which is getting the amount of the transaction (whether user wants to withdraw or deposit) and sending appropriate messages to SavingsAccount or CheckingAccount business classes. Therefore, they are good candidates to be combined into two view classes, one for CheckingAccount and one for SavingsAccount (by following UI rule 3). The CheckingAccountUI and SavingsAccountUI allow users to deposit money to or withdraw money from checking and savings accounts.

The CheckingAccountUI view class provides the interface for a checking account deposit or withdrawal (see Figure 12–17).

The SavingsAccountUI view class provides the interface for a savings account deposit or withdrawal (see Figure 12–17).

Finally, we need to create one more view class that provides the main control or the main UI to the ViaNet bank system. The MainUI view class provides the main control interface for the ViaNet bank system.

12.8.2 The View Layer Micro Process

Based on the outcome of the macro process, we have the following view classes:

BankClientAccessUI
MainUI
AccountTransactionUI
CheckingAccountUI
SavingsAccountUI

For every interface object identified in the macro UI design process,

- *Apply micro-level UI design rules and corollaries to develop the UI.* We need to go through each identified interface object and apply design rules (such as making the UI simple, transparent, and controlled by the user) and GUI guidelines to design them.
- *Iterate and refine.*

12.8.3 The BankClientAccessUI Interface Object

The BankClientAccessUI provides clients access to the system by allowing them to enter their PIN code for validation. The BankClientAccessUI is designed to work with a card reader device, where the user can insert the card and the card number should be displayed automatically in the card number field. In a situation where there is no card reader, such as on-line banking (e.g., user wants to log onto the system from home), the user must enter his or her card number (see Figure 12–18).

12.8.4 The MainUI Interface Object

The MainUI provides the main control to the ATM services. Users can select to deposit money to savings or checking, withdraw money from savings or checking, inquire as to a balance or transaction history, or quit the session (see Figure 12–19).

12.8.5 The AccountTransactionUI Interface Object

The AccountTransactionUI interface object will display the transaction history of either a savings or checking account. The user must select the account type by pressing the radio buttons. Figure 12–20 displays the account balance inquiry and transaction history interface.

FIGURE 12–18
The BankClientAccessUI interface. The buttons are enlarged to make it easier for touch screen users. The numeric keypad on the right side of the dialog box is for data entry and is not a component of the BankClientAccessUI.

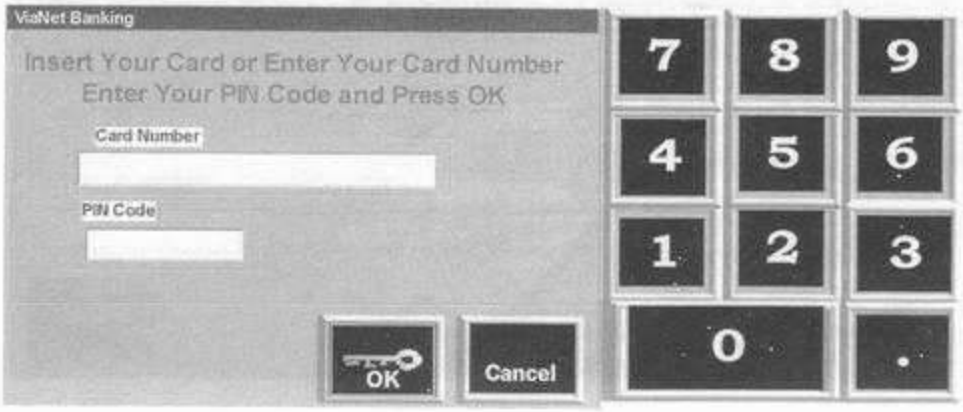
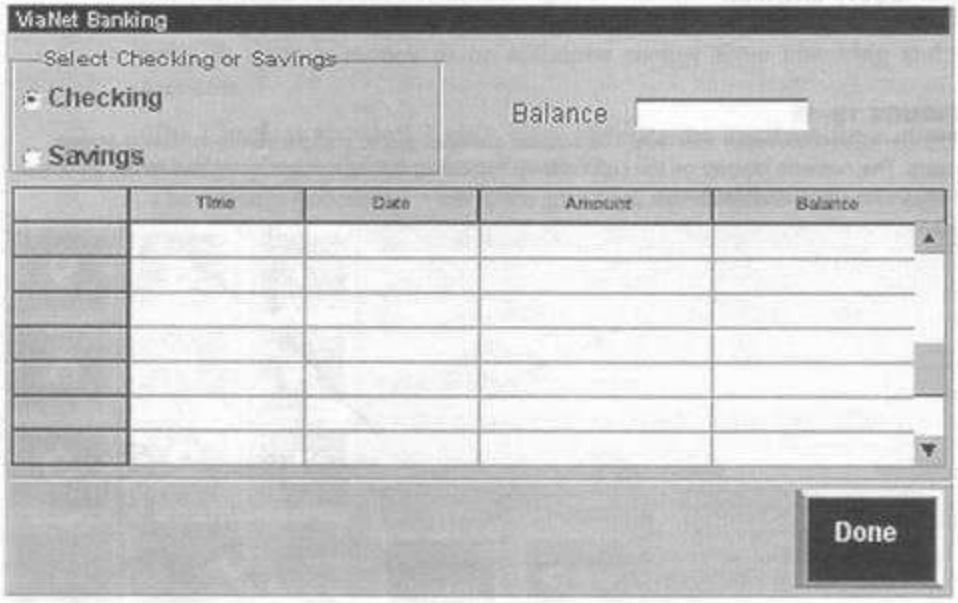


FIGURE 12–19
The MainUI interface.

FIGURE 12–20
The AccountTransactionUI interface.



12.8.6 The CheckingAccountUI and SavingsAccountUI Interface Objects

The CheckingAccountUI and SavingsAccountUI interface objects allow users to deposit to and withdraw from checking and savings accounts. These two interfaces are designed with two tabs, one for deposit and one for withdrawal. When users press one of the MainUI's buttons, say, Deposit Savings, the SavingsAccountUI will be activated and the system should go automatically to the Deposit Savings window. Figure 12–21 displays the SavingsAccountUI and CheckingAccountUI interfaces.

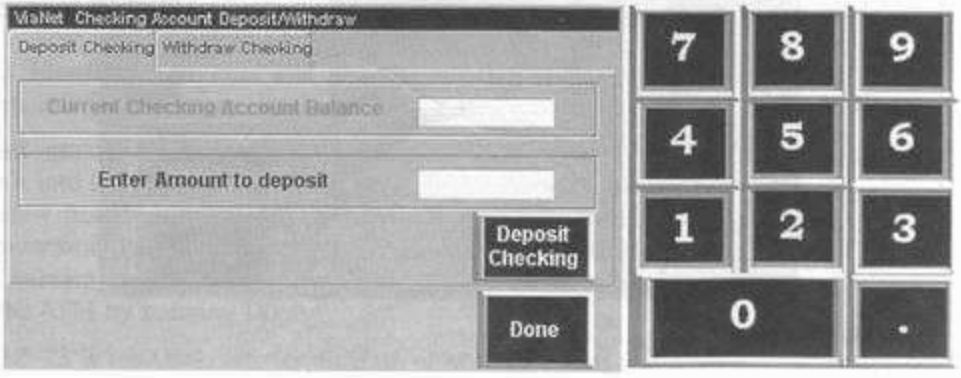
See problem 8 for an alternative design for SavingsAccountUI and CheckingAccountUI classes. It always is a good idea to create an alternative design and select the one that best satisfies the requirements.

12.8.7 Defining the Interface Behavior

The role of a view layer object is to allow the users to manipulate the business model. The actions a user takes on a screen (for example, pressing the Done button) should be translated into a request to the business object for some kind of processing. When the processing is completed, the interface can update itself by displaying new information, opening a new window, or the like.

Defining behavior for an interface consists of identifying the *events* to which you want the system to respond and the *actions* to be taken when the event occurs. Both GUI and business objects can generate events when something happens to them (for example, a button is pushed or a client's name changes). In response to these events, you define actions to take. An action is a combination of an object and a message sent to it.

FIGURE 12–21
The CheckingAccountUI and SavingsAccountUI interface objects.



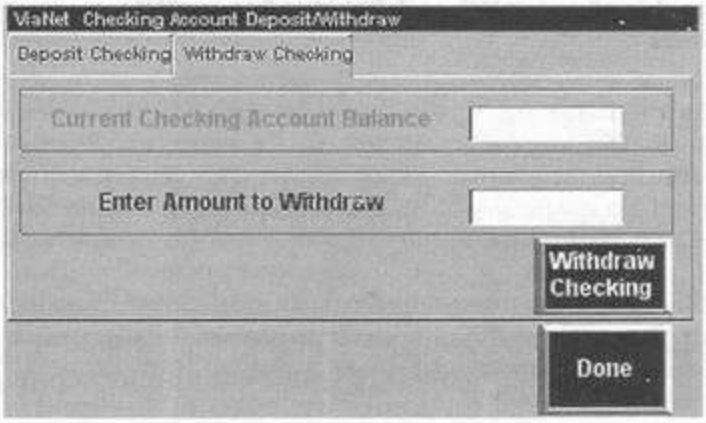


FIGURE 12-21
Continued.

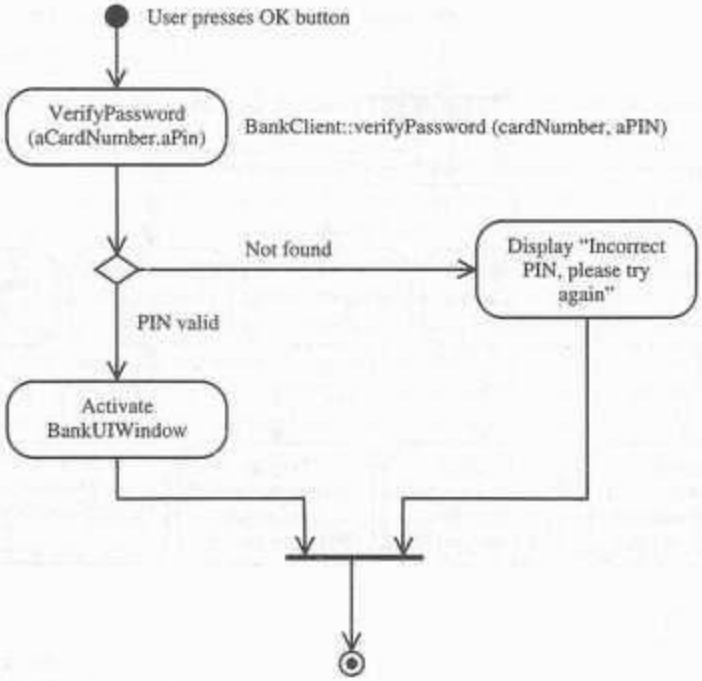


FIGURE 12–22
An activity diagram for the BankClientAccessUI.

12.8.7.1 Identifying Events and Actions for the BankClientAccessUI Interface Object When the user inserts his or her card, types in a PIN code, and presses the OK button, the interface should send the message **BankClient::verifyPassword** (see Chapter 10) to the object to identify the client. If the password is found correct, the MainUI should be displayed and provide users with the ATM services; otherwise, an error message should be displayed. Figure 12–22 is the UML activity diagram of BankClientAccessUI events and actions.

12.8.7.2 Identifying Events and Actions for the MainUI Interface Object From this interface, the user should be able to do the following:

- Deposit into the checking account by pressing the Deposit Checking button.
- Deposit into the savings account by pressing the Deposit Savings button.
- Withdraw from the savings account by pressing the Withdraw Savings button.
- Withdraw from the checking account by pressing the Withdraw Checking button.
- View balance and transaction history by pressing the Balance Inquiry button.
- Exit the ATM by pressing Done.

Figure 12–23 is the UML activity diagram of MainUI events and actions.

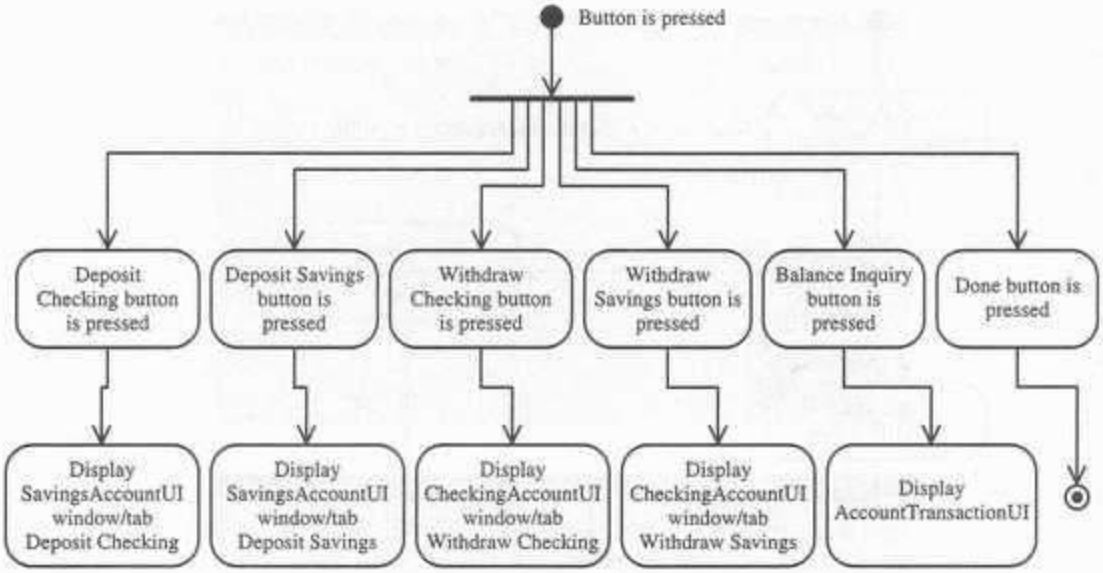
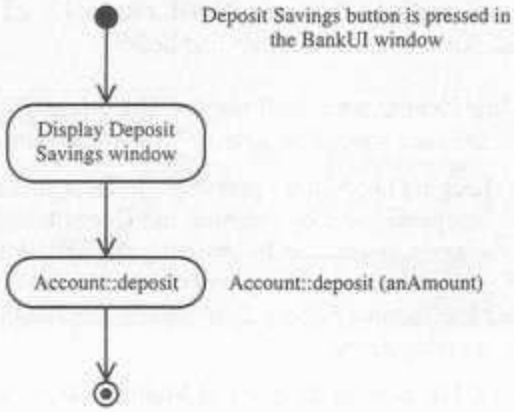


FIGURE 12-23
An activity diagram for the MainUI.

12.8.7.3 Identifying Events and Actions for the SavingsAccountUI Interface Object The SavingsAccountUI has two tabs. First, the SavingsAccountUI opens the appropriate tab. For example, if the user selects the Deposit Savings from the MainUI, the SavingsAccountUI will display the Deposit Savings tab. Figure 12–24 shows the activity diagram for the Deposit Savings. A withdrawal is similar to Deposit Savings and has been left as an exercise; see problem 6.

FIGURE 12-24
Activity diagram for processing a deposit to a savings account.

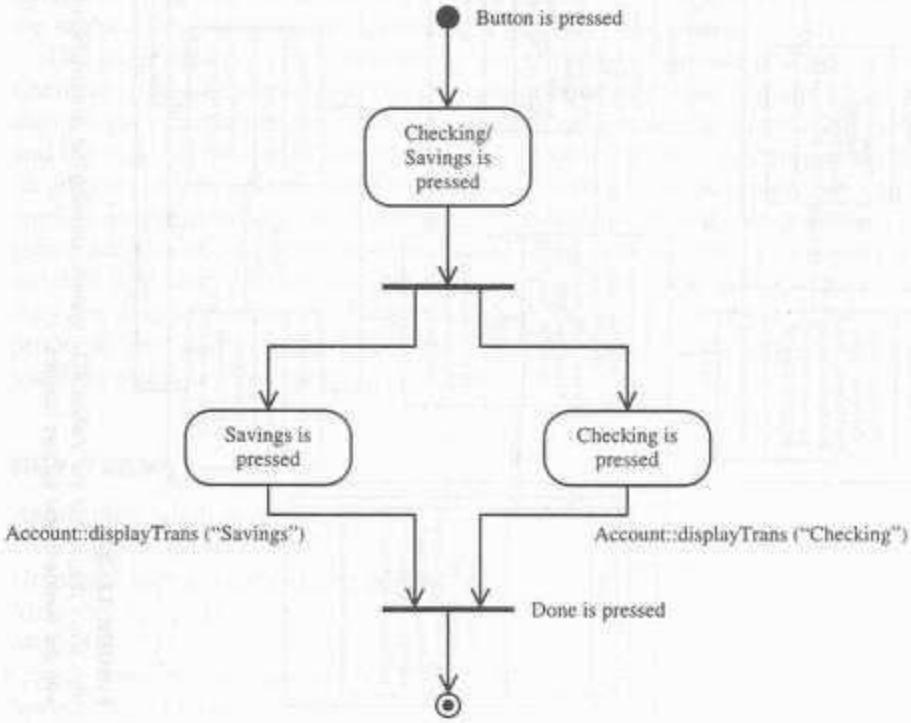


Identifying events and actions for the CheckingAccountUI interface object is left as an exercise; see problem 7.

12.8.7.4 Identifying Events and Actions for the AccountTransactionUI Interface Object A user can select either savings or checking account by pressing on the Savings or Checking radio button. The system then will display the balance and transaction history for the selected account type. The default is the checking account, so when the AccountTransactionUI window is opened for the first time, it will show the checking account history. Pressing on the savings account radio button will cause it to display the savings account balance and history. To close the display and get back to MainUI, the user presses the Done button (see Figure 12–25). Notice that here we assume that the account has a method called displayTrans, which takes a string parameter for type of account (Savings or Checking) and retrieves the appropriate transaction. Since we did not identify or design it, we need to develop it here. This occurs quite often during software development, which is why the process is iterative.

Figure 12–26 shows the relationships among the classes we have designed so far, especially the relationship among the view classes and other business classes.

FIGURE 12–25
Activity diagram for displaying the account transaction.



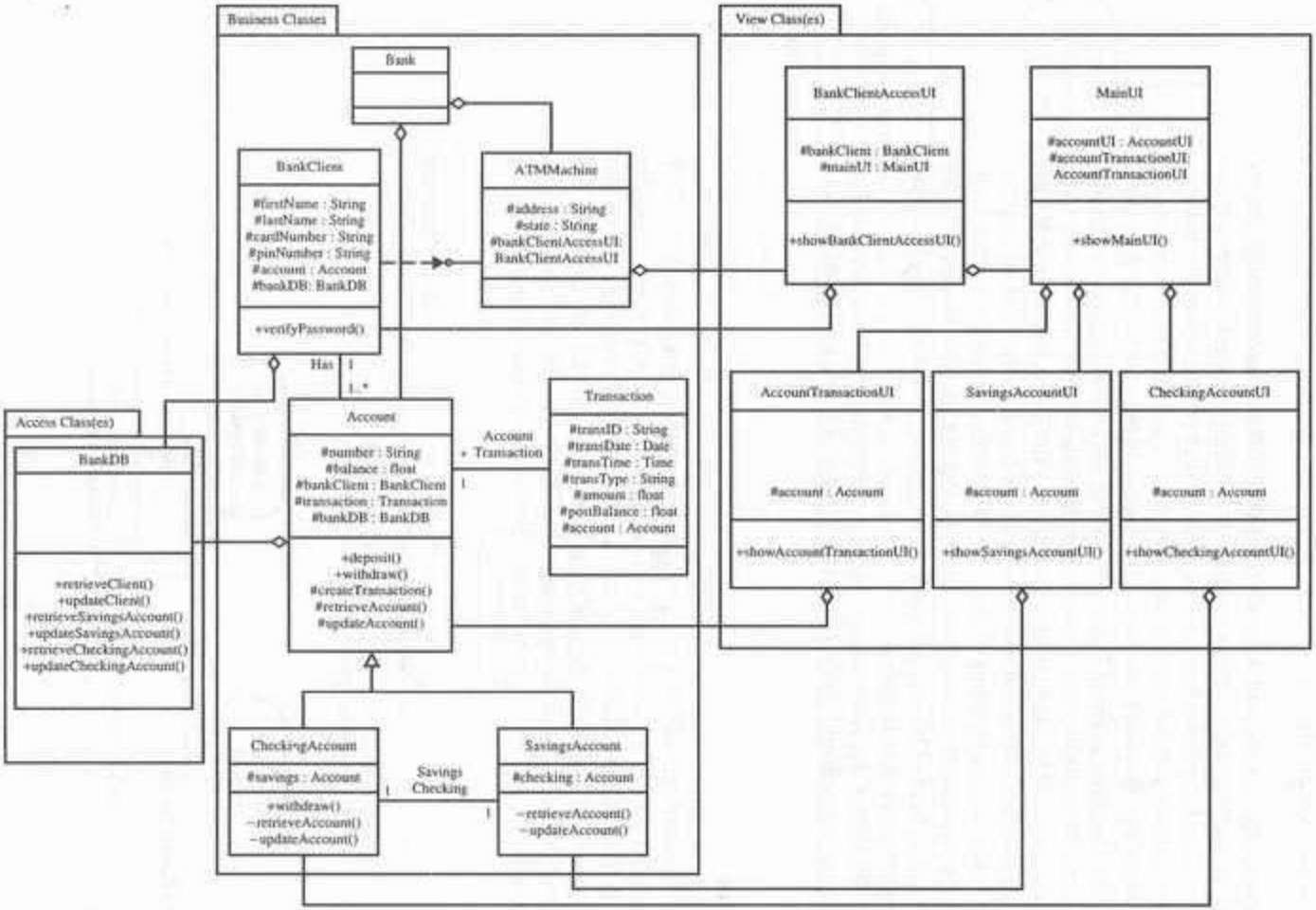


FIGURE 12–26
UML class diagram of the ViaNet ATM system, showing the relationship of the new view classes with the business and access classes.

12.9 SUMMARY

The main goal of a user interface is to display and obtain information needed in an accessible, efficient manner. The design of a software's interface, more than anything else, affects how the user interacts and, therefore, experiences the application. It is important for the design to provide users the information they need and clearly tell them how to complete a task successfully. A well-designed UI has visual appeal that motivates users to use the application. In addition, it should use the limited screen space efficiently.

In this chapter, we learned that the process of designing view layer classes consists of the following steps:

1. Macro-level UI design process: identify view layer objects.
2. Micro-level UI design activities:
   2.1. Design the view layer objects by applying the design axioms and corollaries.
   2.2. Prepare a prototype of the view layer interface.
3. Test usability and user satisfaction.
4. Refine and iterate.

The first step of the process concerns identifying the view classes and their responsibilities by utilizing the view layer macro-level process. The second step is to design these classes by utilizing view layer micro-level processes. User satisfaction and usability testing will be studied in the next chapter. Furthermore, we looked at UI design rules, which are based on the design corollaries; and finally we studied the guidelines for developing a graphical user interface (GUI).

The guidelines are not a substitution for effective evaluation and iterative refinement within a design. However, they can provide helpful advice during the design process. Guidelines emphasize the need to understand the intended audience and the tasks to be carried out, the need to adopt an iterative design process and identify use cases, and the need to consider carefully how the guidelines can be applied in specific situations. Nevertheless, the benefits gained from following design guidelines should not be underestimated. They provide valuable reference material to help with difficult decisions that crop up during the design process, and they are a springboard for ideas and a checklist for omissions. Used with the proper respect and in context, they are a valuable adjunct to relying on designer intuition alone to solve interface problems.

KEY TERMS

Application window (p. 292)
Data entry window (p. 292)
Graphical user interface (GUI) (p. 281)
Metaphor (p. 290)
Mode (p. 291)
Object-oriented user interface (OOUI) (p. 282)
Spring-loaded mode (p. 292)
User-centered interface (p. 287)

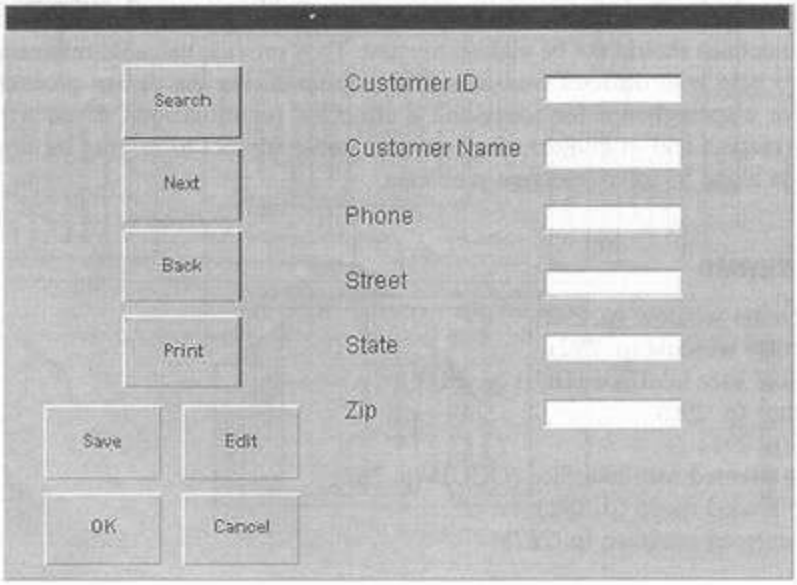
REVIEW QUESTIONS

1. Why is user interface one of the most important components of any software?
2. How can we develop or improve our creativity?
3. Perform a research on GUI and OOUI and write a short paper comparing them.
4. Why do users find OOUI easier to use?
5. How can use cases help us design the view layer objects?
6. Describe the macro and micro processes of view layer design.
7. How can metaphors be used in the design of a user interface?
8. Under what circumstances can you use modes in your user interface?
9. Describe the UI design rules.
10. What is KISS?
11. How would you achieve consistency in your user interface?
12. How can you make your UI forgiving?
13. Describe some of the ways that you can provide the user feedback.

PROBLEMS

1. A touch screen is one way to interact with the ViaNet kiosk. What are some other ways to interact with ViaNet kiosk? Use your imagination to design an interface. Also, design it for people with disability challenges.
2. Research the WWW or your local library on OOUI tools on the market and write a paper of your findings.
3. Please describe problems with the design of the window in Figure 12–27.
4. How can you improve the design of the interface in Figure 12–28?
5. The window in Figure 12–29 suffers from an overkill of radio buttons. Improve the interface by redesigning it.

FIGURE 12–27
Problem 3.



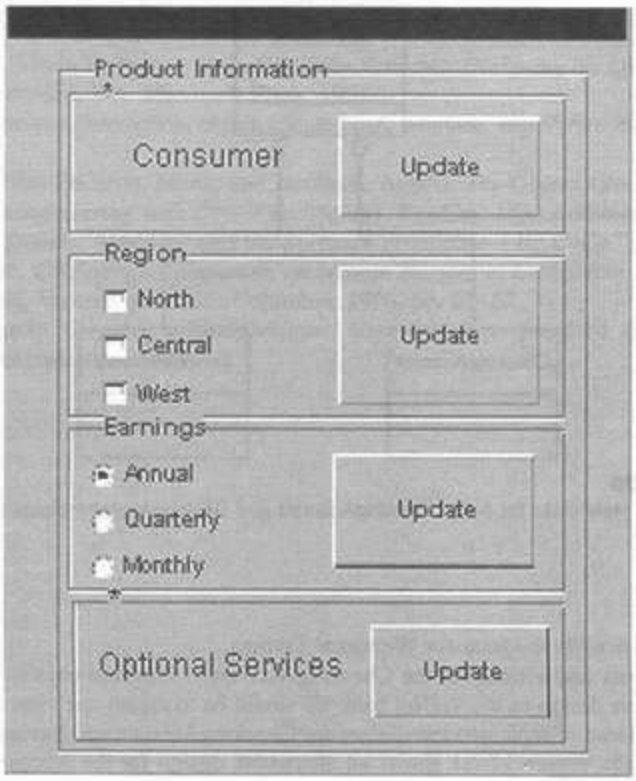
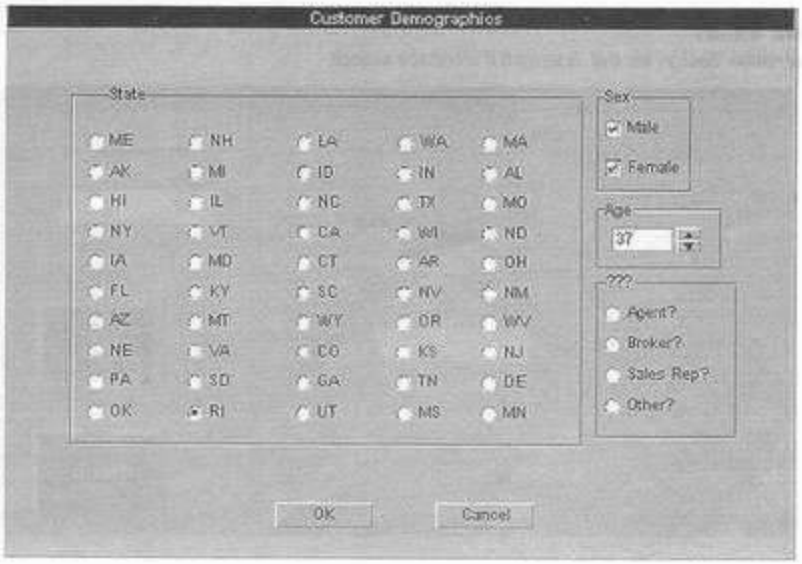


FIGURE 12–28
Problem 4.

FIGURE 12–29
Problem 5.



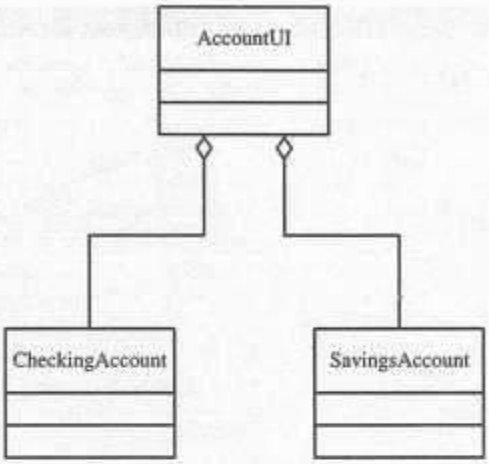
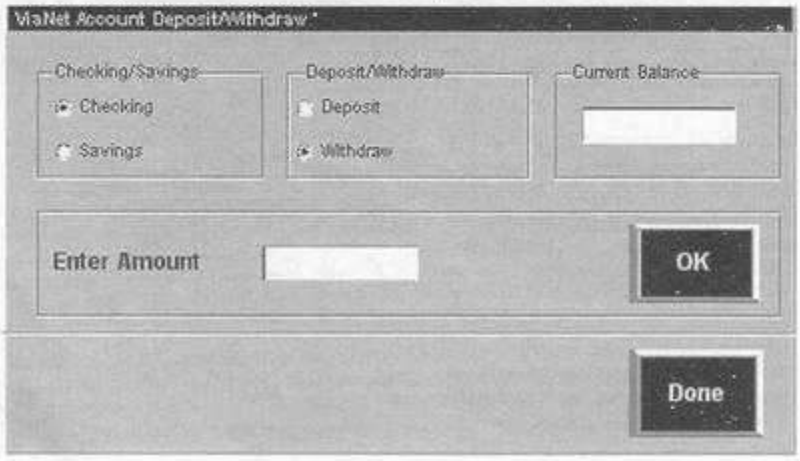


FIGURE 12–30
The AccountUI view class for both CheckingAccount and SavingsAccount classes.

6. Develop an activity diagram for Withdraw Savings.
7. Identify events and actions for the CheckingAccountUI interface object.
8. An alternative design to the ViaNet bank UI would be to create one view class (say, AccountUI) instead of separate view classes for CheckingAccount and SavingsAccount (see Figure 12–30). Figure 12–31 shows an alternative design for the AccountUI. Compare this design to the one in the text and point out advantages and disadvantages to each design.

FIGURE 12–31
An alternative design for the AccountUI interface object.



REFERENCES

1. Capucciati, Maria R. *Putting Your Best Face Forward: Designing an Effective User Interface*. Redmond, WA: Microsoft Press, 1991.
2. IBM. Human-user interaction, object-oriented user interface, http://www.ibm.com/ibm/hci, 1997.
3. Jacobson, Ivar; Ericsson, Maria; and Jacobson, Agneta. *The Object Advantage Business Process, Reengineering with Object Technology*. Reading, MA: Addison-Wesley, 1995.
4. Sulaiman, Suziah. "Usability and the Software Production Life Cycle." *Proceedings of the CHI '96, Conference Companion on Human Factors in Computing Systems: Common Ground*, Vancouver, British Columbia, 1996, pp. 61–62.
5. Trower, Tandy. *Creating a Well-Designed User Interface*. Stanford, CA: University Video Communication, 1994.