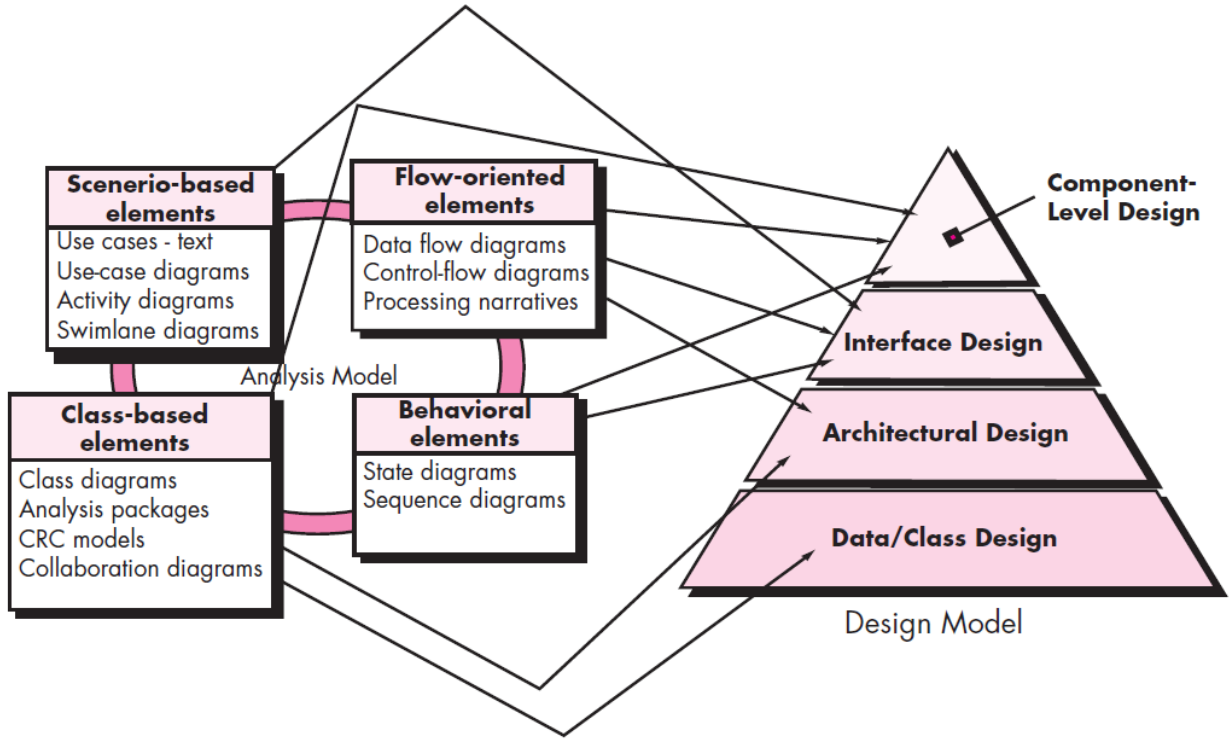
DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

FIGURE 8.1 Translating the requirements model into the design model



Each of the elements of the requirements model (Chapters 6 and 7) provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 8.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design. The data/class design transforms class models (Chapter 6) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

a. THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle

➢ Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

Quality Guidelines. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 8.3, I discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
4. A design should lead to components that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
6. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
7. A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Quality Attributes. Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors (Chapter 11), overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 22), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security.
Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

➢ The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs [Den73] and methods for refining software structures in a topdown manner [Wir71]. Procedural aspects of design definition evolved into a philosophy called *structured programming* [Dah72], [Mil72]. Later work proposed methods for the translation of data flow [Ste74] or data structure (e.g., [Jac75], [War74]) into a design definition. Newer design approaches (e.g., [Jac92], [Gam95]) proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture [Kru06] and the design patterns that can be used to implement software architectures and lower levels of design abstractions (e.g., [Hol06] [Sha05]). Growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04] emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapters 6 and 7, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.
Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

b. DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

M. A. Jackson [Jac75] once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."
In the sections that follow, I present a brief overview of important software design concepts that span both traditional and object-oriented software development.

➢ Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.). A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

➢ Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.One goal of software design is to derive an architectural rendering of a system.
This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems. Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design:
Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks. Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system. A number of different *architectural description languages* (ADLs) have been developed to represent these models [Sha95b]. Although many different ADLs

have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The manner in which software architecture is characterized and its role in design are discussed in Chapter 9.

➢ Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00]. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern. Design patterns are discussed in detail in Chapter 12.

➢ Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

For two problems, $p1$ and $p2$, if the perceived complexity of $p1$ is greater than the perceived complexity of $p2$, it follows that the effort required to solve $p1$ is greater than the effort required to solve $p2$. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

➢ Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.

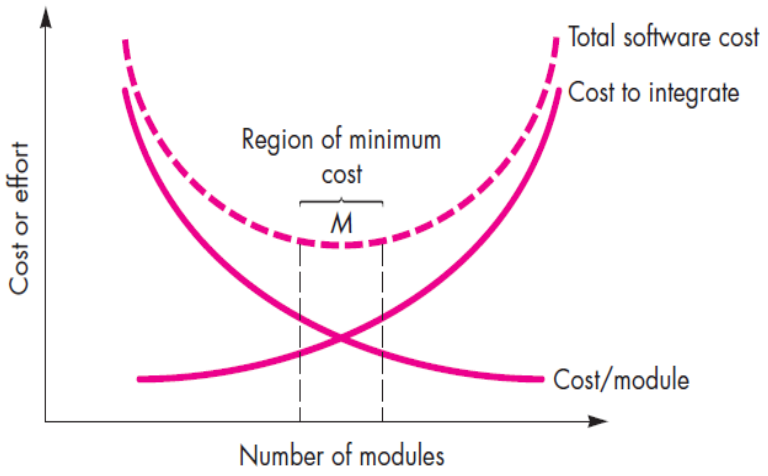
It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling my discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

FIGURE 8.2

Modularity and software cost



The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of *M*. Undermodularity or overmodularity should be avoided. But how do you know the vicinity of *M*? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

➢ Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding [Par72] suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

➢ Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important.

Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or

code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information-hiding concept described in Section 8.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [Ste74], caused when errors occur at one location and propagate throughout a system.

➢ Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. Refinement is actually a process of *elaboration.* You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

➢ Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts" [AOS07]. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, *A* and *B.* Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account" [Ros04].

For example, consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the **ACS-DCV** use case discussed in Chapter 6. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B.* Therefore, *A** and *B** are representations of concerns, and *B* crosscuts A**. An *aspect* is a representation of a crosscutting concern. Therefore, the design representation, *B**, of the requirement *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp. It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components [Ban06]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

➢ Refactoring

An important design activity suggested for many agile methods (Chapter 3), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure." When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

➢ Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. Appendix 2 has been provided for those readers who may be unfamiliar with OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

➢ Design Classes

The requirements model defines a set of analysis classes (Chapter 6). Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high. As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]:

- *User interface classes* define all abstractions that are necessary for humancomputer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

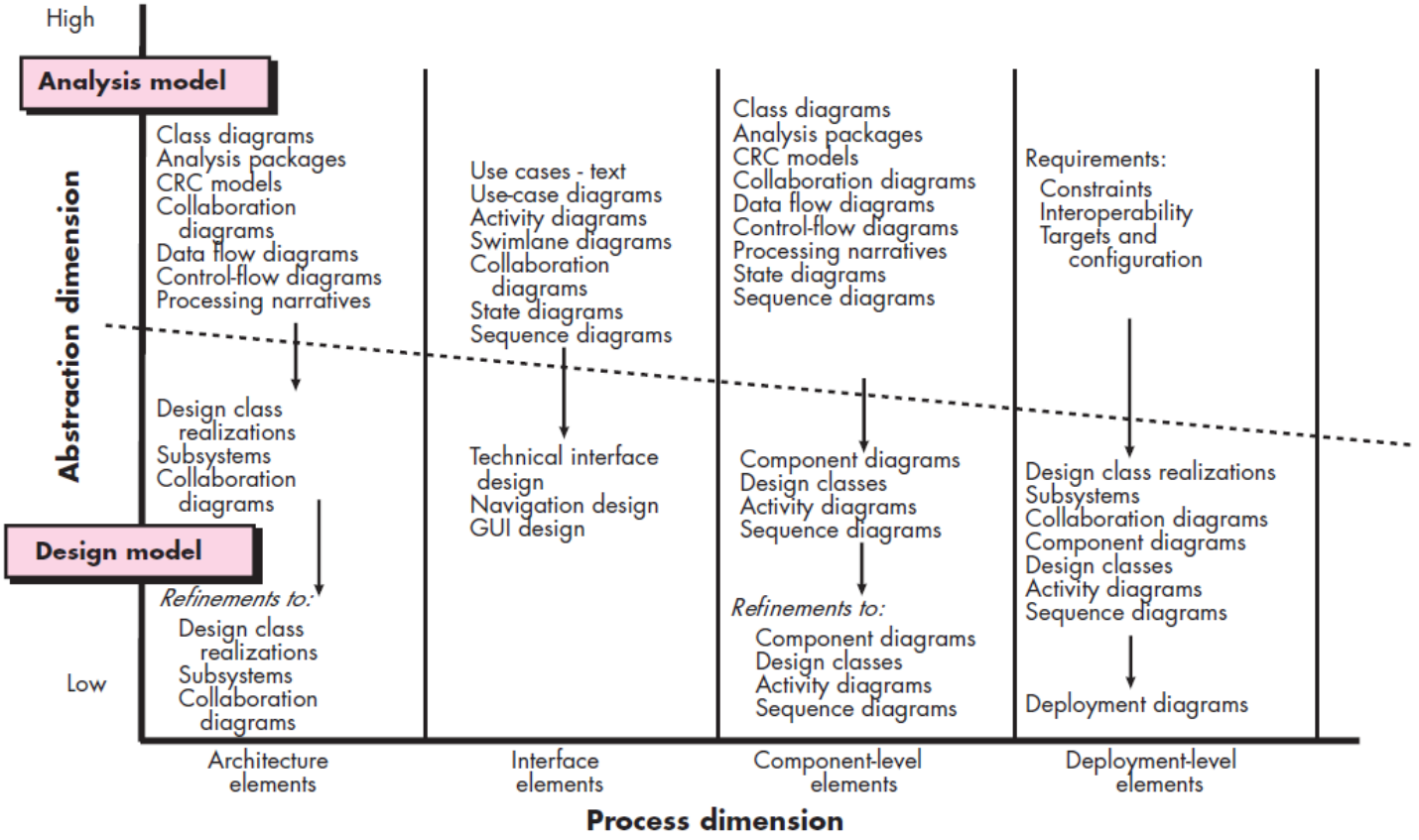
Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip** for video-editing software might have attributes start-point and end-point to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, *setStartPoint()* and *setEndPoint()*, provide the only means for establishing start and end points for the clip.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained. **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.6

c. THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 8.4. The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to Figure 8.4, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

FIGURE 8.4 Dimensions of the design model



The elements of the design model use many of the same UML diagrams7 that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

You can apply design patterns (Chapter 12) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

➢ Data Design Elements

Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 9.

➢ Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model [Sha96] is derived from three sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles (Chapter 9) and patterns (Chapter 12). The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have it's own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 9.

➢ Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.

They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture. UI design (increasingly called *usability design*) is a major software engineering action and is considered in detail in Chapter 11. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

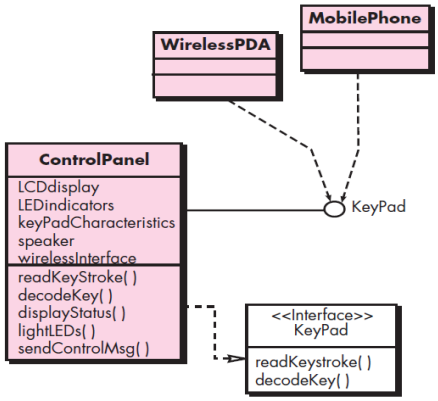
The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 5) and verified once the interface design commences.8 The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

The design of internal interfaces is closely aligned with component-level design (Chapter 10). Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested. If the classic input-process-output approach to design is chosen, the interface of each software component is designed based on data flow representations and the functionality described in a processing narrative.

In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner [OMG03a]: "An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure." Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations. For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone.

The **ControlPanel** class (Figure 8.5) provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke ()* and *decodeKey ()*. If these operations are to be provided to other classes (in this case, **WirelessPDA** and **MobilePhone**), it is useful to define an interface as shown in the figure. The interface, named **KeyPad**, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

FIGURE 8.5
Interface representation for ControlPanel



The dashed line with an open triangle at its end (Figure 8.5) indicates that the **ControlPanel** class provides **KeyPad** operations as part of its behavior. In UML, this is characterized as a *realization.* That is, part of the behavior of **ControlPanel** will be implemented by realizing **KeyPad** operations. These operations will be provided to other classes that access the interface.

- **Component-Level Design Elements**

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6. In this figure, a component named **SensorManagement** (part of the *SafeHome* security function) is represented. A dashed arrow connects the component to a class named **Sensor** that is assigned to it. The **SensorManagement** component performs all functions associated with *SafeHome* sensors including monitoring and configuring them. Further discussion of component diagrams.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming language-like representation described in Chapter 10) or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

➢ Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a home-based PC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system).

FIGURE 8.6

A UML component diagram

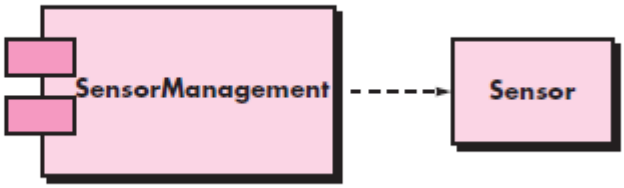
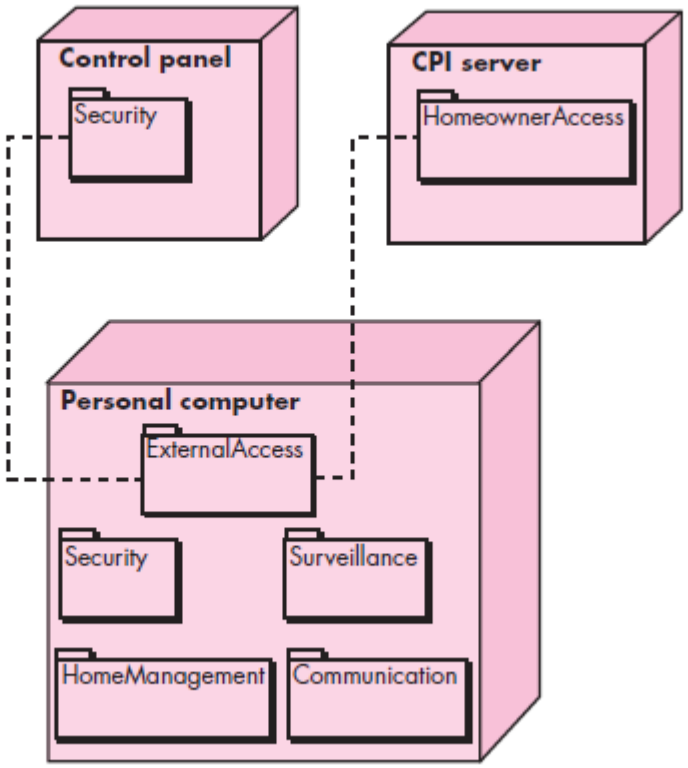


FIGURE 8.7
A UML deployment diagram



During design, a UML deployment diagram is developed and then refined as shown in Figure 8.7. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and others). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

The diagram shown in Figure 8.7 is in *descriptor form.* This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac or a Windows-based PC, a Sun workstation, or a Linux-box. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

ARCHITECTURAL DESIGN

d. ARCHITECTURAL STYLES

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system; (2) a set of connectors that enable "communication, coordination and cooperation" among components; (3) constraints that define how components can be integrated to form the

system; and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [Bas03].

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered (Chapter 29), the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components [Bos00].

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety; (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00]; (3) architectural patterns (Section 9.4) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. In Section 9.3.1, I consider commonly used architectural styles and patterns for software.

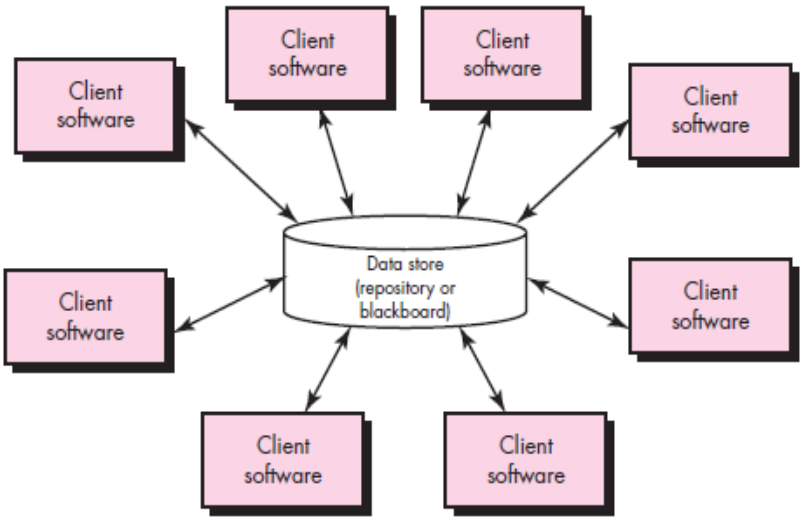
➢ A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository.

In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

FIGURE 9.1
Data-centered architecture

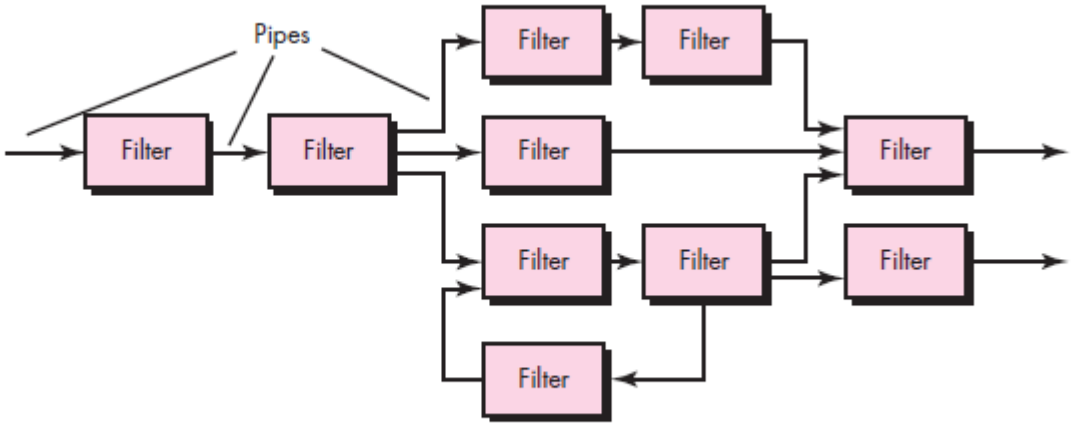


Data-centered architectures promote *integrability* [Bas03]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a

certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

FIGURE 9.2
Data-flow architecture

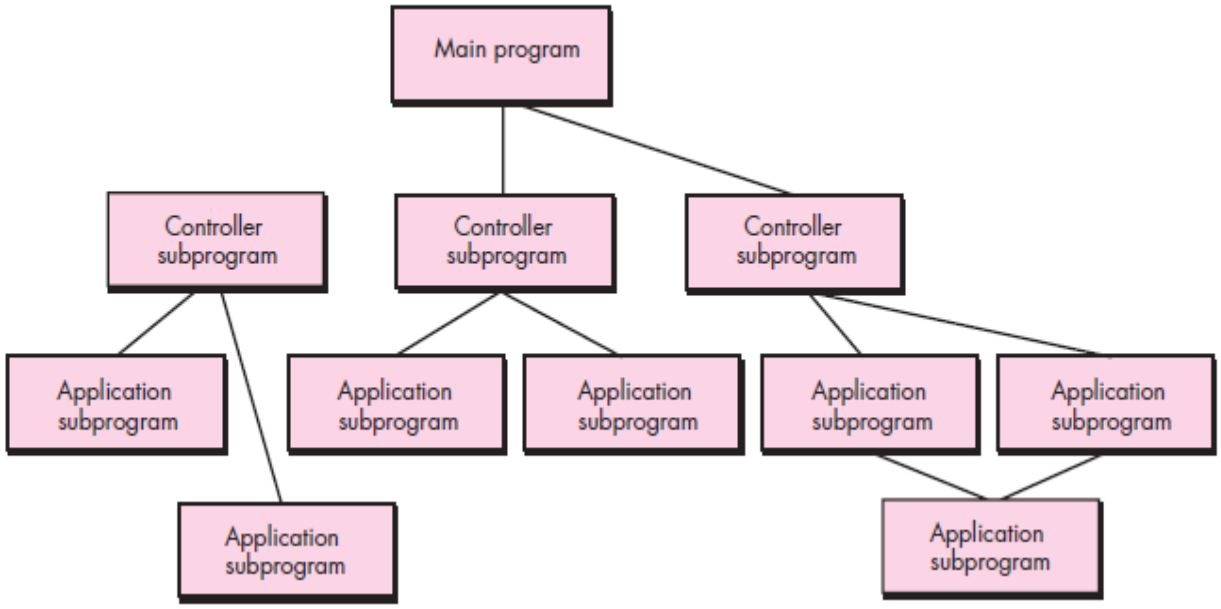


If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles [Bas03] exist within this category:

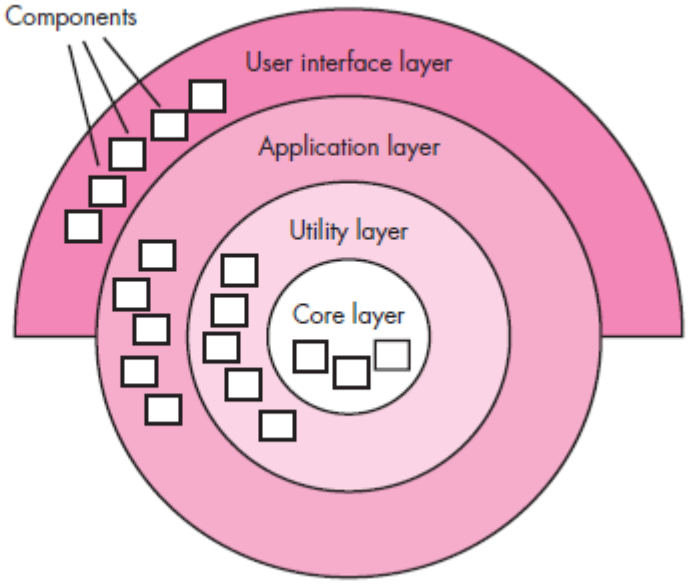
- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other
- components. Figure 9.3 illustrates an architecture of this type. *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

FIGURE 9.3 Main program/subprogram architecture



Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

FIGURE 9.4
Layered architecture



Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available.2 Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

➢ Architectural Patterns

As the requirements model is developed, you'll notice that the software must address a number of broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online?*

The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way in which you address the problem to be solved.

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Earlier in this chapter, I noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call-and return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns are presented in Chapter 12.

➢ Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas03] provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system? These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

e. ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived. In the sections that follow we examine each of these architectural design tasks in a bit more detail.

➢ Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 9.5.

Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the *SafeHome* product. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 9.6. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that are both producers and consumers of information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

As part of the architectural design, the details of each interface shown in Figure 9.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

FIGURE 9.5
Architectural context diagram
Source: Adapted from [Bos00].

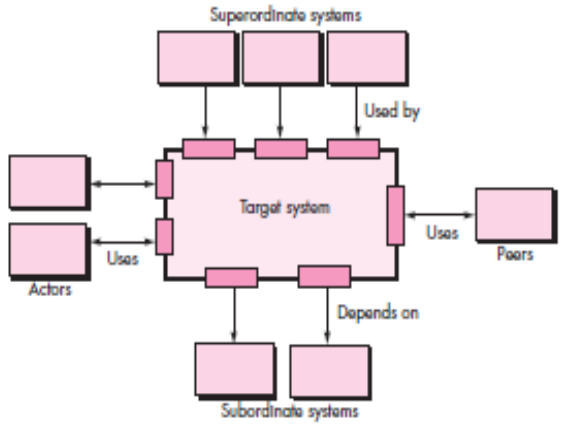
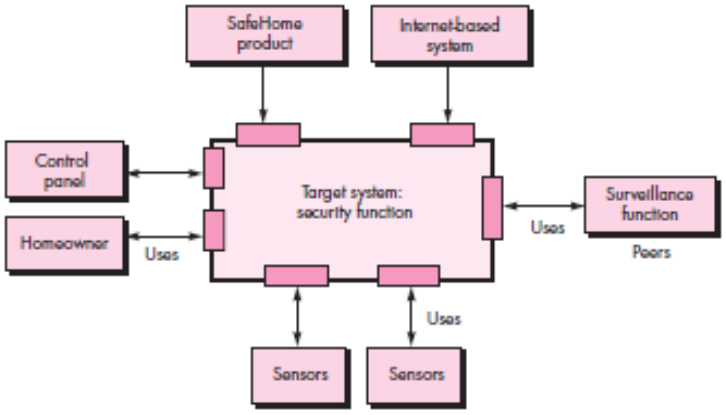


FIGURE 9.6
Architectural context diagram for the *SafeHome* security function



➢ Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.
In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

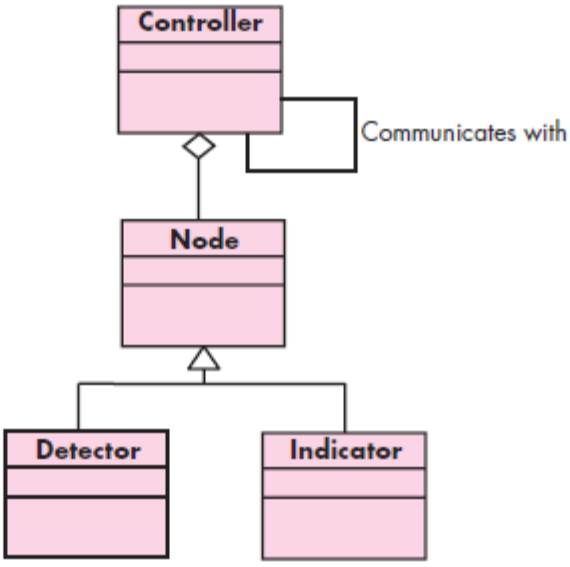


FIGURE 9.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

Each of these archetypes is depicted using UML notation as shown in Figure 9.7. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

➢ Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

The interfaces depicted in the architecture context diagram (Section 9.4.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

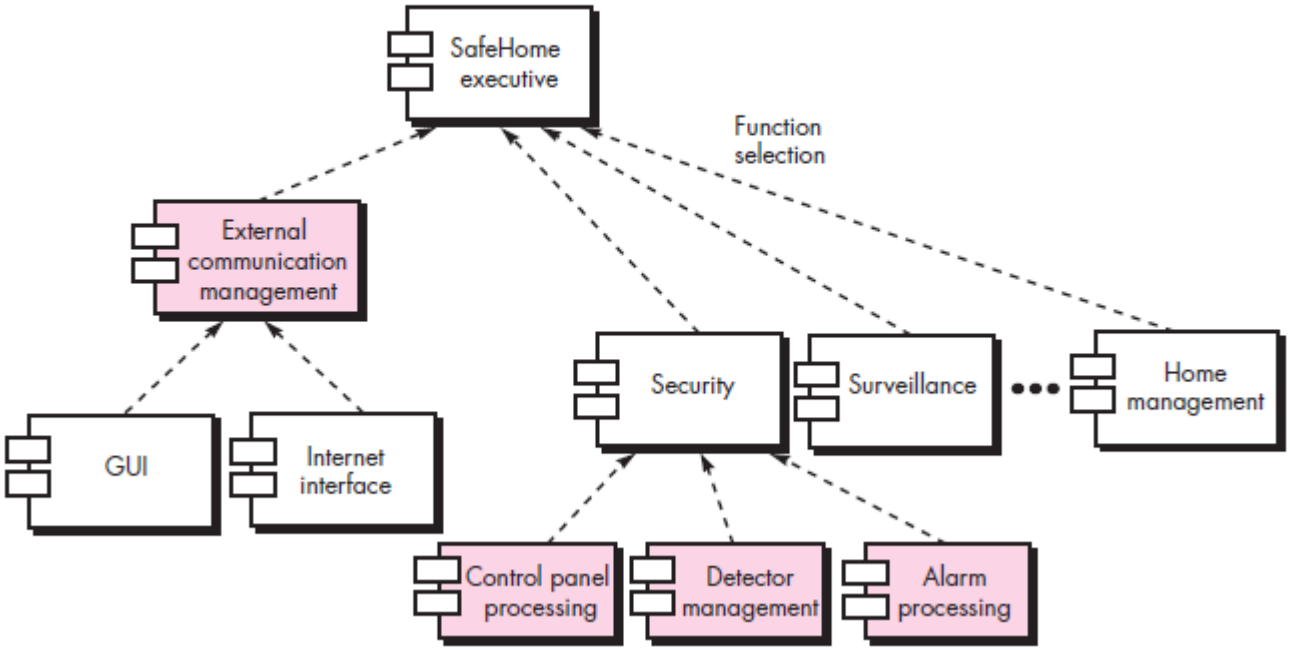
- *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design (Chapter 10).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 9.8. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management*

component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

FIGURE 9.8 Overall architectural structure for *SafeHome* with top-level components



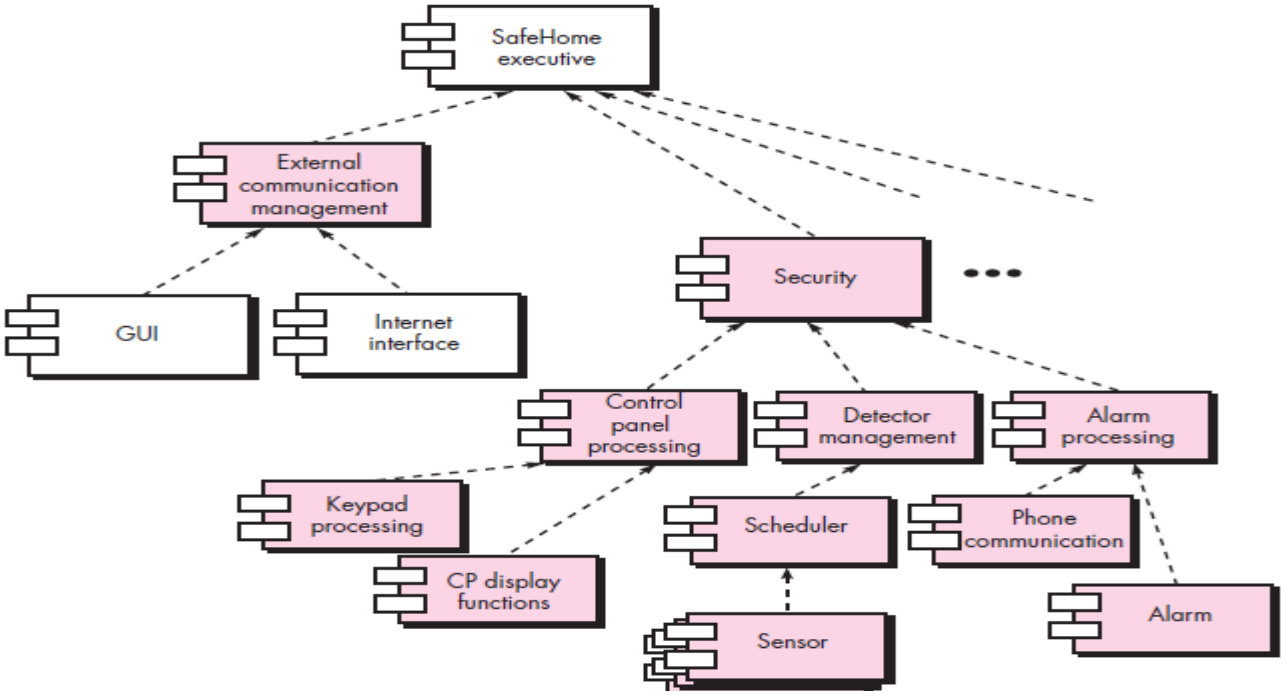
➢ Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure 9.9 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 9.8 are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 9.8.

FIGURE 9.9 An instantiation of the security function with component elaboration



USER INTERFACE DESIGN

f. THE GOLDEN RULES

In his book on interface design, Theo Mandel [Man97] coins three *golden rules:*

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

➢ Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. "What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that."

My first reaction was to shake my head and smile, but I paused for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her. Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom? As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel [Man97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

➢ Reduce the User's Memory Load

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

➢ Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3)

mechanisms for navigating from task to task are consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion. The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the interface design process itself.

g. USER INTERFACE ANALYSIS AND DESIGN

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

➢ Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model,* the software engineer creates a *design model,* the end user develops a mental image that is often called the user's *mental model* or the *system perception*, and the implementers of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. In his introductory column on "user-centric design," Jeff Patton [Pat07] notes: The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Selfsubstitution isn't user centric—it's self-centric.

To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [Shn04]. In addition, users can be categorized as:

Novices. No syntactic knowledge1 of the system and little semantic knowledge2 of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.

The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The *implementation model* combines the outward manifestation of the computer based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident,

users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

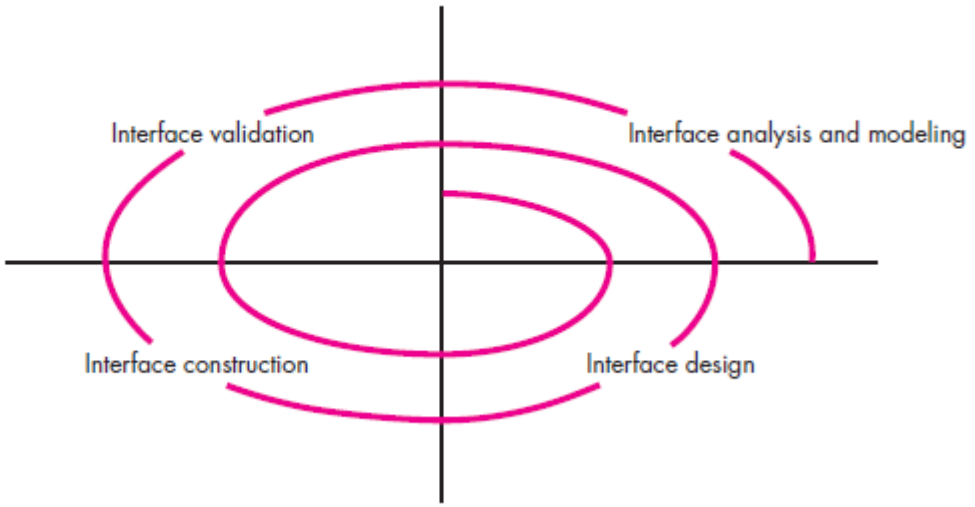
The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive system" [Mon84]. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

➢ The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 2. Referring to Figure 11.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities [Man97]: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 11.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

FIGURE 11.1

The user interface design process



Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception (Section 11.2.1) for each class of users.

Once general requirements have been defined, a more detailed *task analysis* is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 11.3. Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences. The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. Interface design is discussed in more detail in Section 11.4.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 11.5) may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.

As I have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

h. INTERFACE ANALYSIS

A key tenet of all software engineering process models is this: *understand the problem before you attempt to design a solution.* In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. In the sections that follow, I examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

➢ User Analysis

The phrase "user interface" is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier I noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions (adapted from [Hac98]) will help you to better understand the users of a system:

- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

➢ Task Analysis and Modeling

The goal of task analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

To answer these questions, you must draw upon techniques that I have discussed earlier in this book, but in this instance, these techniques are applied to the user interface.

Use cases. In earlier chapters you learned that the use case describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first-person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function. When asked: "How do you decide where to put furniture in a room?" an interior designer writes the following informal use case:

I begin by sketching the floor plan of the room, the dimensions and the location of windows and doors. I'm very concerned about light as it enters the room, about the view out of the windows (if it's beautiful, I want to draw attention to it), about the running length of an unobstructed wall, about the flow of movement through the room. I then look at the list of furniture my customer and I have chosen—tables, chairs, sofa, cabinets, the list of accents—lamps, rugs, paintings, sculpture, plants, smaller pieces, and my notes on any desires my customer has for placement. I then draw each item from my lists using a template that is scaled to the floor plan. I label each item I draw and use pencil because I always move things. I consider a number of alternative placements and decide on the one I like best. Then, I draw a rendering (a 3-D picture) of the room to give my customer a feel for what it'll look like.

This use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction. In addition, other features of the system that would please the interior designer might also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through each window.

Task elaboration. In Chapter 8, I discussed stepwise elaboration (also called functional decomposition or stepwise refinement) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Task analysis can be applied in two ways. As I have already noted, an interactive, computer-based system is often used to replace a manual or semi manual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, you must first define and classify tasks. I have already noted that one approach is stepwise elaboration. For example, let's reconsider the computer-aided design system for interior designers discussed earlier. By observing an interior designer at work, you notice that interior design comprises a number of major activities: furniture layout (note the use case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use case,

furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions, (2) place windows and doors at appropriate locations, (3a) use furniture templates to draw scaled furniture outlines on the floor plan, (3b) use accents templates to draw scaled accents on the floor plan, (4) move furniture outlines and accent outlines to get the best placement, (5) label all furniture and accent outlines, (6) draw dimensions to show location, and (7) draw a perspective-rendering view for the customer. A similar approach could be used for each of the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.4 The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

Object elaboration. Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include size, shape, location, and others. The interior designer would *select* the object from the **Furniture** class, *move* it to a position on the floor plan (another object in this context), *draw* the furniture outline, and so forth. The tasks *select*, *move*, and *draw* are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

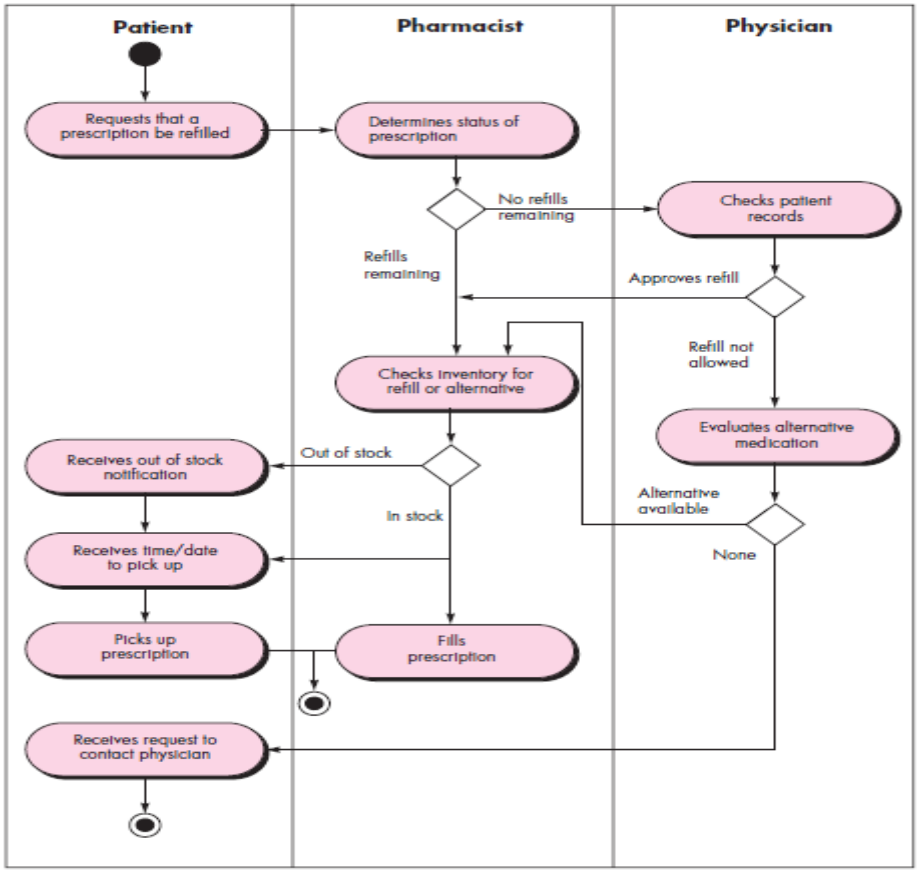
Workflow analysis. When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis.* This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process5 will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).

We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 11.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted earlier. This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., *Fills prescription* could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.

FIGURE 11.2 Swimlane diagram for prescription refill function



User task: ***Requests that a prescription be refilled***

- *Provide identifying information.*
- *Specify name.*
- *Specify userid.*
- *Specify PIN and password.*
- *Specify prescription number.*
- *Specify date refill is required.*

To complete the task, three subtasks are defined. One of these subtasks, *provide identifying information*, is further elaborated in three additional sub-subtasks.

➢ Analysis of Display Content

The user tasks identified in Section 11.3.2 lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques discussed in Chapters 6 and 7 identify the output data objects that are produced by an application. These data objects may be (1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question. During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?

- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?
- The answers to these (and other) questions will help you to establish requirements for content presentation.

➢ Analysis of the Work Environment

Hackos and Redish [Hac98] discuss the importance of work environment analysis when they state: People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use. In some applications the user interface for a computer-based system is placed in a "user-friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

i. INTERFACE DESIGN STEPS

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step. Although many different user interface design models (e.g., [Nor86], [Nie00]) have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis (Section 11.3), define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

In some cases, you can begin with sketches of each interface state (i.e., what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, you should (1) always follow the golden rules discussed in Section 11.1, (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools) that will be used.

➢ Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed in much the same way as described in Chapter 6. That is, a use case is written.Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication of this action is to create a hard-copy report. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted. If a real-world metaphor is appropriate

for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

To provide a brief illustration of the design steps noted previously, consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary use case (written by the homeowner) for the interface follows:

Preliminary use case: I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my notebook computer (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming *SafeHome*. I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this use case, the following homeowner tasks, objects, and data items are identified:

- *accesses* the *SafeHome* system
- *enters* an **ID** and **password** to allow remote access
- *checks* **system status**
- *arms* or *disarms SafeHome* system
- *displays* **floor plan** and **sensor locations**
- *displays* **zones** on floor plan
- *changes* **zones** on floor plan
- *displays* **video camera locations** on floor plan
- *selects* **video camera** for viewing
- *views* **video images** (four frames per second)
- *pans* or *zooms* the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

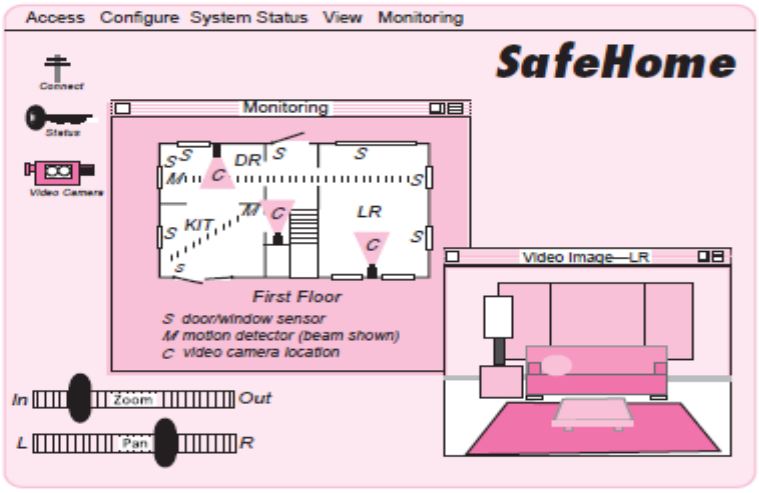
A preliminary sketch of the screen layout for video monitoring is created (Figure 11.3).6 To invoke the video image, a video camera location icon, *C*, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

➢ User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As I noted earlier in this book, a design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

FIGURE 11.3
Preliminary screen layout



As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

A vast array of interface design patterns has been proposed over the past decade. A more detailed discussion of user interface design patterns is presented in Chapter 12. In addition, Erickson [Eri08] provides pointers to many Web-based collections.

➢ Design Issues

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. *Variability* refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.

Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of "user manuals" may be the only option. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues [Rub88] must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function key, or a HELP command.

- How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
- How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error handling. Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. There are few computer users who have not encountered an error of the form: *"Application XXX has been forced to quit because an error of type 1023 has been encountered."* Somewhere, an explanation for error 1023 must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in jargon that the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).
- The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."
- The message should be "nonjudgmental." That is, the wording should never place blame on the user.

Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

Menu and command labeling. The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

As I noted earlier in this chapter, conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

Application accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines (e.g., [W3C03])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [App08], [Mic08]) provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. *Localization* features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM03]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

PATTERN-BASED DESIGN

j. DESIGN PATTERNS

A *design pattern* can be characterized as "a three-part rule which expresses a relation between a certain context, a problem, and a solution" [Ale79]. For software design, *context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment. A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how the problem can be interpreted within its context and how the solution can be effectively applied.

To better understand these concepts, consider a situation2 in which a person must travel between New York and Los Angeles. In this context, travel will occur within an industrialized country (the United States), using an existing transportation infrastructure (e.g., roads, airlines, railways). The system of forces that will affect the way in which the travel problem is solved will include: how quickly the person wants to get from New York to LA, whether the trip will include site-seeing or stopovers, how much money the person can spend, whether the trip is intended to accomplish a specific purpose, and the personal vehicles the person has at her disposal. Given these forces, the problem (traveling from New York to LA) can be better defined. For example, investigation (requirements gathering) indicates that the person has very little money, owns only a bicycle (and is an avid cyclist), wants to make the trip to raise money for her favorite charity, and has plenty of time to spare. The solution to the problem, given the context and the system of forces, might be a cross-country bike trip. If the forces were different (e.g., travel time must be minimized and the purpose of the trip is a business meeting), another solution might be more appropriate.

It is reasonable to argue that most problems have multiple solutions, but that a solution is effective only if it is appropriate within the context of the existing problem. It is the system of forces that causes a designer to choose a specific solution. The intent is to provide a solution that best satisfies the system of forces, even when these forces are contradictory. Finally, every solution has consequences that may have an impact on other aspects of the software and may themselves become part of the system of forces for other problems to be solved within the larger system. Coplien [Cop05] characterizes an effective design pattern in the following way:

- *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
- *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Stated even more pragmatically, a good design pattern captures hard-earned, pragmatic design knowledge in a way that enables others to reuse that knowledge "a million times over without ever doing it the same way twice." A design pattern saves you from "reinventing the wheel," or worse, inventing a "new wheel" that is

slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer.

➢ Kinds of Patterns

One of the reasons that software engineers are interested in (and intrigued by) design patterns is that human beings are inherently good at pattern recognition. If we weren't, we'd be frozen in space and time—unable to learn from past experience, unwilling to venture forward because of our inability to recognize situations that might lead to high risk, unhinged by a world that seems to have no regularity or logical consistency. Luckily, none of this occurs because we do recognize patterns in virtually every aspect of our lives.

In the real world, the patterns we recognize are learned over a lifetime of experience. We recognize them instantly and inherently understand what they mean and how they might be used. Some of these patterns provide us with insight into recurring phenomenon. For example, you're on your way home from work on the interstate when your navigation system (or car radio) informs you that a serious accident has occurred on the interstate in the opposing direction. You're 4 miles from the accident, but already you begin to see traffic slowing, recognizing a pattern that we'll call **RubberNecking.** People in the travel lanes moving in your direction are slowing at the sight of the accident to get a better view of what happened on the opposite side of the highway. The **RubberNecking** pattern yields remarkably predictable results (a traffic jam), but it does nothing more than describe a phenomenon.

In patterns jargon, it might be called a *nongenerative* pattern because it describes a context and a problem but it does not provide any clear-cut solution. When software design patterns are considered, we strive to identify and document *generative* patterns. That is, we identify a pattern that describes an important and repeatable aspect of a system and that provides us with a way to build that aspect within a system of forces that are unique to a given context. In an ideal setting, a collection of generative design patterns could be used to "generate" an application or computer-based system whose architecture enables it to adapt to change. Sometimes called *generativity*, "the successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution which emerges indirectly as a result of the smaller solutions" [App00].

Design patterns span a broad spectrum of abstraction and application. *Architectural patterns* describe broad-based design problems that are solved using a structural approach. *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture. *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. At a lower level of abstraction, *idioms* describe how to implement all or part of a specific algorithm or data structure for a software component within the context of a specific programming language.

In their seminal book on design patterns, Gamma and his colleagues3 [Gam95] focus on three types of patterns that are particularly relevant to object-oriented design: creational patterns, structural patterns, and behavioral patterns. *Creational patterns* focus on the "creation, composition, and representation" of objects. Gamma and his colleagues [Gam95] note that creational patterns "encapsulate knowledge about which concrete classes the system uses" but at the same time "hide how instances of these classes are created and put together."

Creational patterns provide mechanisms that make the instantiation of objects easier within a system and nforce "constraints on the type and number of objects that can be created within a system" [Maa07].

Structural patterns focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. In essence, they help to establish relationships between entities within a system. For example, structural patterns that focus on class-oriented issues might provide inheritance mechanisms that lead to more effective program interfaces. Structural patterns that focus on objects suggest techniques for combining objects within other objects or integrating objects into a larger structure.

Behavioral patterns address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects.

➢ Frameworks

Patterns themselves may not be sufficient to develop a complete design. In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework*, for design work. That is, you can select a "*reusable miniarchitecture* that provides the generic structure and behavior for a family of software abstractions, along with a context . . . which specifies their collaboration and use within a given domain" [Amb98]. A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain. The plug points enable you to integrate problem-specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.

Gamma and his colleagues [Gam95] describe the differences between design patterns and frameworks in the following manner:

1. *Design patterns are more abstract than frameworks*. Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. . . .
2. *Design patterns are smaller architectural elements than frameworks*. A typical framework contains several design patterns but the reverse is never true.
3. *Design patterns are less specialized than frameworks*. Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture.

In essence, the designer of a framework will argue that one reusable mini architecture is applicable to all software to be developed within a limited domain of application. To be most effective, frameworks are applied with no changes. Additional design elements may be added, but only via the plug points that allow the designer to flesh out the framework skeleton.

➢ Describing a Pattern

Pattern-based design begins with the recognition of patterns within the application you intend to build, continues with a search to determine whether others have addressed the pattern, and concludes with the application of an appropriate pattern to the problem at hand. The second of these three tasks is often the most difficult. How do you find patterns that fit your needs? An answer to this question must rely on effective communication of the problem the pattern addresses, the context in which the pattern resides, the system of forces that mold the context, and the solution that is proposed. To communicate this information unambiguously, a standard form or template for pattern descriptions is required. Although a number of different pattern templates have been proposed, almost all contain a major subset of the content suggested by Gamma and his colleagues [Gam95]. A simplified pattern template is shown in the sidebar.

The names of design patterns should be chosen with care. One of the key technical problems in pattern-based design is the inability to find existing patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name.

A pattern template provides a standardized means for describing a design pattern. Each of the template entries represents characteristics of the design pattern that can be searched (e.g., via a database) so that the appropriate pattern can be found.

➢ Pattern Languages and Repositories

When we use the term *language*, the first thing that comes to mind is either a natural language (e.g., English, Spanish, Chinese) or a programming language (e.g., C__, Java). In both cases the language has a syntax and semantics that are used to communicate ideas or procedural instructions in an effective manner.

When the term *language* is used in the context of design patterns, it takes on a slightly different meaning. A *pattern language* encompasses a collection of patterns, each described using a standardized template (Section 12.1.3) and interrelated to show how these patterns collaborate to solve problems across an application domain.4

In a natural language, words are organized into sentences that impart meaning. The structure of sentences is described by the language's syntax. In a pattern language, design patterns are organized in a way that provides a "structured method of describing good design practices within a particular domain."5

In a way, a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain. The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction. In a software context, broad design problems tend to be architectural in nature and address the overall structure of the application and the data or content that serve it. Architectural problems are refined to lower levels of abstraction, leading to design patterns that solve subproblems and collaborate with one another at the component (or class) level. Rather than a sequential list of patterns, a pattern language represents an interconnected collection in which the user can begin with a broad design problem and "burrow down" to uncover specific problems and their solutions.

Dozens of pattern languages have been proposed for software design [Hil08]. In most cases, the design patterns that are part of pattern language are stored in a Webaccessible patterns repository (e.g., [Boo08], [Cha03], [HPR02]). The repository provides an index of all design patterns and contains hypermedia links that enable the user to understand the collaborations between patterns.

k. PATTERN-BASED SOFTWARE DESIGN

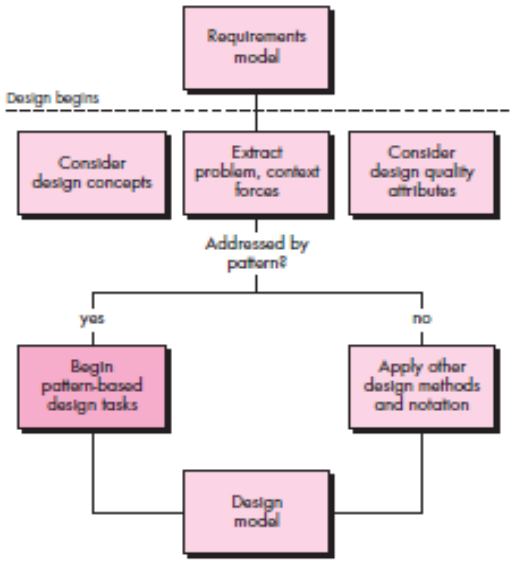
The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution. The software developers at Microsoft [Mic04] discuss this when they write: While pattern-based design is relatively new in the field of software development, industrial technology has used pattern-based design for decades, perhaps even centuries.

Catalogs of mechanisms and standard configurations provide design elements that are used to engineer automobiles, aircraft, machine tools, and robots. Applying pattern based design to software development promises the same benefits to software as it does to industrial technology: predictability, risk mitigation, and increased productivity. Throughout the design process, you should look for every opportunity to apply existing design patterns (when they meet the needs of the design) rather than creating new ones.

➢ Pattern-Based Design in Context

Pattern-based design is not used in a vacuum. The concepts and techniques discussed for architectural, component-level, and user interface design (Chapters 9 through 11) are all used in conjunction with a pattern-based approach. In Chapter 8, I noted that a set of quality guidelines and attributes serve as the basis for all software design decisions. The decisions themselves are influenced by a set of fundamental design concepts (e.g., separation of concerns, stepwise refinement, functional independence) that are achieved using heuristics that have evolved over many decades, and best practices (e.g., techniques, modeling notation) that have been proposed to make design easier to perform and more effective as a basis for construction.

FIGURE 12.1
Pattern-based design in context



The role of pattern-based design in all of this is illustrated in Figure 12.1. A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system. The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway. It may imply the design in an abstract manner, but the requirements model does little to represent the design explicitly.

As you begin your work as a designer, it's always important to keep quality attributes in mind. These attributes (e.g., a design must implement all explicit requirements addressed in the requirements model) establish a way to assess software quality but do little to help you actually achieve it. The design you create should exhibit the fundamental design concepts discussed in Chapter 8. Therefore, you should apply proven techniques for translating the abstractions contained in the requirements model into a more concrete form that is the software design. To accomplish this, you'll use the methods and modeling tools available for architectural, component level, and interface design. But only when you're faced with a problem, context, and system of forces that have not been solved before. If a solution already exists, use it! And that means applying a pattern-based design approach.

➢ Thinking in Patterns

In an excellent book on pattern-based design, Shalloway and Trott [Sha05] comment on a "new way of thinking" when one uses patterns as part of the design activity: I had to open my mind to a new way of thinking. And when I did so, I heard [Christopher] Alexander say that "good software design cannot be achieved simply by adding together performing parts."

Good design begins by considering context—the big picture. As context is evaluated, you extract a hierarchy of problems that must be solved. Some of these problems will be global in nature, while others will address specific features and functions of the software. All will be affected by a system of forces that will influence the nature of the solution that is proposed.

Shalloway and Trott [Sha05] suggest the following approach6 that enables a designer to think in patterns:

1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with "big picture" patterns that establish a context or skeleton for further design work.
4. "Work inward from the context" [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
5. Repeat steps 1 to 4 until the complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.

It's important to note that patterns are not independent entities. Design patterns that are present at a high level of abstraction will invariably influence the manner in which other patterns are applied at lower levels of abstraction. In addition, patterns often collaborate with one another. The implication—when you select an architectural pattern, it may very well influence the component-level design patterns you choose. Likewise, when you select a specific interface design pattern, you are sometimes forced to use other patterns that collaborate with it.

To illustrate, consider the **SafeHomeAssured.com** WebApp. If you consider the big picture, the WebApp must address a number of fundamental problems such as:

- How to provide information about *SafeHome* products and services
- How to sell *SafeHome* products and services to customers
- How to establish Internet-based monitoring and control of an installed security system

Each of these fundamental problems can be further refined into a set of subproblems. For example *How to sell via the Internet* implies an **E-commerce** pattern that itself implies a large number of patterns at lower levels of abstraction. The **E-commerce** pattern (likely, an architectural pattern) implies mechanisms for setting up a customer account, displaying the products to be sold, selecting products for purchase, and so forth. Hence, if you think in patterns, it is important to determine whether a pattern for setting up an account exists. If **SetUpAccount** is available as a viable pattern for the problem context, it may collaborate with other patterns

such as **BuildInputForm, ManageFormsInput,** and **Validate- FormsEntry.** Each of these patterns delineates problems to be solved and solutions that may be applied.

➢ Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:

1. **Examine the requirements model and develop a problem hierarchy.** Describe each problem and subproblem by isolating the problem, the context, and the system of forces that apply. Work from broad problems (high level of abstraction) to smaller subproblems (at lower levels of abstraction).
2. **Determine if a reliable pattern language has been developed for the problem domain.** As I noted in Section 12.1.4, a pattern language addresses problems associated with a specific application domain. The *SafeHome* software team would look for a pattern language developed specifically for home security products. If that level of pattern language specificity could not be found, the team would partition the *SafeHome* software problem into a series of generic problem domains (e.g., digital device monitoring problems, user interface problems, digital video management problems) and search for appropriate pattern languages.
3. **Beginning with a broad problem, determine whether one or more architectural patterns is available for it.** If an architectural pattern is available, be certain to examine all collaborating patterns. If the pattern is appropriate, adapt the design solution proposed and build a design model element that adequately represents it. As I noted in Section 12.2.2, a broad problem for the **SafeHomeAssured.com** WebApp is addressed with an **E-commerce** pattern. This pattern will suggest a specific architecture for addressing e-commerce requirements.
4. **Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems and search for appropriate patterns to address them.** It may be necessary to search through other pattern repositories as well as the list of patterns that corresponds to the architectural solution. If an appropriate pattern is found, adapt the design solution proposed and build a design model element that adequately represents it. Be certain to apply step 7.
5. **Repeat steps 2 through 5 until all broad problems have been addressed.** The implication is to begin with the big picture and elaborate to solve problems at increasingly more detailed levels.
6. **If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.** Proceed in a manner similar to steps 3, 4, and 5.
7. **Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.** Be certain to examine context and forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.
8. **Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.** Although this design approach is top-down, real-life design solutions are sometimes more complex. Gillis [Gil06] comments on this when he writes:

Design patterns in software engineering are meant to be used in a deductive, rationalistic fashion. So you have this general problem or requirement, X, design pattern Y solves X, therefore use Y. Now, when I reflect on my own process—and I've got reason to believe that I'm not alone here—I find that it's more organic than that, more inductive than deductive, more bottom-up than top-down. Obviously, there's a balance to be achieved. When a project is in the initial bootstrap phase and I'm trying to make the jump from abstract requirements to a concrete design solution, I'll often perform a sort of breadth-first search . . . I've found design patterns to be helpful, allowing me to quickly frame up the design problem in concrete terms.

In addition, the pattern-based approach must be used in conjunction with other software design concepts and techniques.

➢ Building a Pattern-Organizing Table

As pattern-based design proceeds, you may encounter trouble organizing and categorizing candidate patterns from multiple pattern languages and repositories. To help organize your evaluation of candidate patterns,

Microsoft [Mic04] suggests the creation of a *pattern-organizing table* that takes the general form shown in Figure 12.2.
A pattern-organizing table can be implemented as a spreadsheet model using the form shown in the figure. An abbreviated list of problem statements, organized by data/content, architecture, component-level, and user interface issues, is presented in the left-hand (shaded) column. Four pattern types—database, application, implementation, and infrastructure—are listed across the top row. The names of candidate patterns are noted in the cells of the table.

FIGURE 12.2
A pattern-organizing table
Source: Adapted from [Mic04].

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

To provide entries for the organizing table, you'll search through pattern languages and repositories for patterns that address a particular problem statement. When one or more candidate patterns is found, it is entered in the row corresponding to the problem statement and the column that corresponds to the pattern type. The name of the pattern is entered as a hyperlink to the URL of the Web address that contains a complete description of the pattern.

➢ Common Design Mistakes

Pattern-based design can make you a better software designer, but it is *not* a panacea. Like all design methods, you must begin with first principles, emphasizing software quality fundamentals and ensuring that the design does, in fact, address the needs expressed by the requirements model.
A number of common mistakes occur when pattern-based design is used. In some cases, not enough time has been spent to understand the underlying problem and its context and forces, and as a consequence, you select a pattern that looks right but is inappropriate for the solution required. Once the wrong pattern is selected, you refuse to see your error and force-fit the pattern. In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit. Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.
Can these mistakes be avoided? In most cases the answer is "yes." Every good designer looks for a second opinion and welcomes review of her work. The review techniques discussed in Chapter 15 can help to ensure that the pattern-based design you've developed will result in a high-quality solution for the software problem to be solved.

l. COMPONENT-LEVEL DESIGN PATTERNS

Component-level design patterns provide you with proven solutions that address one or more subproblems extracted from the requirements model. In many cases, design patterns of this type focus on some functional element of a system. For example, the **SafeHomeAssured.com** application must address the following design subproblem:
How can we get product specifications and related information for any SafeHome device? Having enunciated the subproblem that must be solved, you should now consider context and the system of forces that affect the

solution. Examining the appropriate requirements model use case, you learn that the consumer uses the specification for a *SafeHome* device (e.g., a security sensor or camera) for informational purposes.
However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected. The solution to the subproblem involves a search. Since searching is a very common problem, it should come as no surprise that there are many search-related patterns. Looking through a number of patterns repositories, you find the following patterns, along with the problem that each solves:
AdvancedSearch. Users must find a specific item in a large collection of items.
HelpWizard. Users need help on a certain topic related to the website or when they need to find a specific page within the site.
SearchArea. Users must find a page.
SearchTips. Users need to know how to control the search engine.
SearchResults. Users have to process a list of search results.
SearchBox. Users have to find an item or specific information.
For **SafeHomeAssured.com** the number of products is not particularly large,
and each has a relatively simple categorization, so **AdvancedSearch** and **HelpWizard** are probably not necessary. Similarly, the search is simple enough not to require **SearchTips**. The description of **SearchBox**, however, is given (in part) as:
Search Box
(Adapted from www.welie.com/patterns/showPattern.php?patternID=search.)
Problem: The users need to find an item or specific information. **Motivation:** Any situation in which a keyword search is applied across a collection of content objects organized as Web pages.
Context: Rather than using navigation to acquire information or content, the user wants to do a direct search through content contained on multiple Web pages. Any website that already has primary navigation. User may want to search for an item in a category. User might want to further specify a query.
Forces: The website already has primary navigation. Users may want to search for an item in a category. Users might want to further specify a query using simple Boolean operators.
Solution: Offer search functionality consisting of a search label, a keyword field, a filter if applicable, and a "go" button. Pressing the return key has the same function as selecting the go button. Also provide Search Tips and examples in a separate page. A link to that page is placed next to the search functionality. The edit box for the search term is large enough to accommodate three typical user queries (typically around 20 characters). If the number of filters is more than 2, use a combo box for filters selection, otherwise a radio button.
The search results are presented on a new page with a clear label containing at least "Search results" or similar. The search function is repeated in the top part of the page with the entered keywords, so that the users know what the keywords were. *The pattern description continues with other entries as described in Section 12.1.3.* The pattern goes on to describe how the search results are accessed, presented, matched, and so on. Based on this, the **SafeHomeAssured.com** team can design the components required to implement the search or (more likely) acquire existing reusable components.

m. USER INTERFACE DESIGN PATTERNS

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of the following 10 categories of patterns (discussed with a representative example8) as described by Tidwell [Tid02] and vanWelie [Wel01]:
Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.
Pattern: TopLevelNavigation

> **Brief description:** Used when a site or application implements a number of major functions. Provides a top-level menu, often coupled with a logo or major functions.
> **Details:** Major functions (generally limited to between four and seven function names) are listed across the top of the display (vertical column formats are also possible) in a horizontal line of text. Each name provides a link to the appropriate function or information source. Often used with the
> **BreadCrumbs** pattern discussed later.
> **Navigation elements:** Each function/content name represents a link to the appropriate function or content.

Page layout. Address the general organization of pages (for websites) or distinct screen displays (for interactive applications).

Pattern: CardStack

Brief description: Used when a number of specific subfunctions or content categories related to a feature or function must be selected in random order. Provides the appearance of a stack of tabbed cards, each selectable with a mouse click and each representing specific subfunctions or content categories.

Details: Tabbed cards are a well-understood metaphor and are easy for the user to manipulate. Each tabbed card (divider) may have a slightly different format. Some may require input and have buttons or other navigation mechanisms; others may be informational. May be combined with other patterns such as **DropDownList**, **Fill-in-the-Blanks,** and others.

Navigation elements: A mouse click on a tab causes the appropriate card to appear. Navigation features within the card may also be present, but in general, these should initiate a function that is related to card data, not cause
an actual link to some other display.

Forms and input. Consider a variety of design techniques for completing formlevel input.

Pattern: Fill-in-the-Blanks

Brief description: Allow alphanumeric data to be entered in a "text box."

Details: Data may be entered within a text box. In general, the data are validated and processed after some text or graphic indicator (e.g., a button containing "go," "submit," "next") is picked. In many cases this pattern can be combined with drop-down list or other patterns (e.g., SEARCH <*drop down list*> FOR <*fill-in-the-blanks* text box>).

Navigation elements: A text or graphic indicator that initiates validation and processing.

Tables. Provide design guidance for creating and manipulating tabular data of all kinds.

Pattern: SortableTable

Brief description: Display a long list of records that can be sorted by selecting a toggle mechanism for any column label.

Details: Each row in the table represents a complete record. Each column represents one field in the record. Each column header is actually a selectable button that can be toggled to initiate an ascending or descending sort on the field associated with the column for all records displayed. The table is generally resizable and may have a scrolling mechanism if the number of records is larger than available window space.

Navigation elements: Each column header initiates a sort on all records. No other navigation is provided, although in some cases, each record may itself contain navigation links to other content or functionality.

Direct data manipulation. Address data editing, modification, and transformation.

Pattern: BreadCrumbs

Brief description: Provides a full navigation path when the user is working with a complex hierarchy of pages or display screens.

Details: Each page or display screen is given a unique identifier. The navigation path to the current location is specified in a predefined location for every display. The path takes the form: **home>major topic page>subtopic page> specific page>current page.**

Navigation elements: Any of the entries within the bread crumbs display can be used as a pointer to link back to a higher level of the hierarchy.

Navigation. Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.

Pattern: EditInPlace

Brief description: Provide simple text editing capability for certain types of content in the location that it is displayed. No need for the user to enter a text editing function or mode explicitly.

Details: The user sees content on the display that must be changed. A mouse double click on the content indicates to the system that editing is desired. The content is highlighted to signify that editing mode is available
and the user makes appropriate changes.

Navigation elements: None.

Searching. Enable content-specific searches through information maintained within a website or contained by persistent data stores that are accessible via an interactive application.

Pattern: SimpleSearch

Brief description: Provides the ability to search a website or persistent data source for a simple data item described by an alphanumeric string.

Details: Provides the ability to search either locally (one page or one file) or globally (entire site or complete database) for the search string. Generates a list of "hits" in order of their probability of meeting the user's needs. Does not provide multiple item searches or special Boolean operations (see *advanced search pattern*).

Navigation elements: Each entry in the list of hits represents a navigation link to the data referenced by the entry.

Page elements. Implement specific elements of a Web page or display screen.

Pattern: Wizard

Brief description: Takes the user through a complex task one step at a time, providing guidance for the completion of the task through a series of simple window displays.

Details: Classic example is a registration process that contains four steps. The wizard pattern generates a window for each step, requesting specific information from the user one step at a time.

Navigation elements: Forward and back navigation allows the user to revisit each step in the wizard process.

E-commerce. Specific to websites, these patterns implement recurring elements of e-commerce applications.

Pattern: ShoppingCart

Brief description: Provides a list of items selected for purchase.

Details: Lists item, quantity, product code, availability (in stock, out of stock), price, delivery information, shipping costs, and other relevant purchase information. Also provides ability to edit (e.g., remove, change quantity).

Navigation elements: Contains ability to proceed with shopping or go to checkout.

Miscellaneous. Patterns that do not easily fit into one of the preceding categories. In some cases, these patterns are domain dependent or occur only for specific classes of users.

Pattern: ProgressIndicator

Brief description: Provides an indication of progress when an operation takes longer than *n* seconds.

Details: Represented as an animated icon or a message box that contains some visual indication (e.g., a rotating "barber pole," a slider with a percent complete indicator) that processing is under way. May also contain a text content indication of the status of processing.

Navigation elements: Often contains a button that allows the user to pause or cancel processing.

Each of the preceding example patterns (and all patterns within each category) would also have a complete component-level design, including design classes, attributes, operations, and interfaces.

A comprehensive discussion of user interface patterns is beyond the scope of this book. If you have further interest, see [Duy02], [Bor01], [Tid02], and [Wel01] for further information.

n. WEBAPP DESIGN PATTERNS

Throughout this chapter you've learned that there are different types of patterns and many different ways they can be categorized. When you consider the design problems that must be solved when a WebApp is to be built, it's worth considering pattern categories by focusing on two dimensions: the design focus of the pattern and its level of granularity. *Design focus* identifies which aspect of the design model is relevant (e.g., information architecture, navigation, interaction). *Granularity* identifies the level of abstraction that is being considered (e.g., does the pattern apply to the entire WebApp, to a single Web page, to a subsystem, or an individual WebApp component?).

- **Design Focus**

In earlier chapters I emphasized a design progression that begins by considering architecture, component-level issues, and user interface representations. At each step, the problems you consider and the solutions you propose begin at a high level of abstraction and slowly become more detailed and specific. Stated another way, design focus becomes "narrower" as you move further into design. The problems (and solutions) you will encounter when designing an information architecture for a WebApp are different from the problems (and solutions) that are encountered when performing interface design. Therefore, it should come as no surprise that patterns for WebApp design can be developed for different levels of design focus, so that you can address the unique problems (and related solutions) that are encountered at each level. WebApp patterns can be categorized using the following levels of design focus:

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action, how a user expands content based on usage context and user desires, how to best describe the destination that is implied by a link, how to inform the user about the status of an ongoing interaction, and interface-related issues.
- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability, how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communication, and other algorithmic elements within a WebApp.

In most cases, it would be fruitless to explore the collection of information architecture patterns when a problem in interaction design is encountered. You would examine interaction patterns, because that is the design focus that is relevant to the work being performed.

➢ Design Granularity

When a problem involves "big picture" issues, you should attempt to develop solutions (and use relevant patterns) that focus on the big picture. Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly. In terms of the level of granularity, patterns can be described at the following levels:

- **Architectural patterns.** This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.
- **Design patterns.** These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component-tocomponent communication. An example might be the **Broadsheet** pattern for the layout of a WebApp home page.
- **Component patterns.** This level of abstraction relates to individual smallscale elements of a WebApp. Examples include individual interaction elements (e.g., radio buttons), navigation items (e.g., how might you format links?) or functional elements (e.g., specific algorithms).

It is also possible to define the relevance of different patterns to different classes of applications or domains. For example, a collection of patterns (at different levels of design focus and granularity) might be particularly relevant to e-business.

AN INTRODUCTION TO UML

The *Unified Modeling Language* (UML) is "a standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system". In otherwords, just as building architects create blueprints to be used by a construction company, software architects create UML diagrams to help software developers build the software. If you understand the vocabulary of UML (the

diagrams' pictorial elements and their meanings), you can much more easily understand and specify a system and explain the design of that system to others.

Grady Booch, Jim Rumbaugh, and Ivar Jacobson developed UML in the mid- 1990s with much feedback from the software development community. UML merged a number of competing modeling notations that were in use by the software industry at the time. In 1997, UML 1.0 was submitted to the Object Management Group, a nonprofit consortium involved in maintaining specifications for use by the computer industry. UML 1.0 was revised to UML 1.1 and adopted later that year. The current standard is UML 2.0 and is now an ISO standard. Because this standard is so new, many older references, such as [Gam95] do not use UML notation.

UML 2.0 provides 13 different diagrams for use in software modeling. In this appendix, I will discuss only *class, deployment, use case, sequence, communication, activity,* and *state* diagrams. These diagrams are used in this edition of *Software Engineering: A Practitioner's Approach.* You should note that there are many optional features in UML diagrams. The UML language provides these (sometimes arcane) options so that you can express all the important aspects of a system. At the same time, you have the flexibility to suppress those parts of the diagram that are not relevant to the aspect being modeled in order to avoid cluttering the diagram with irrelevant details. Therefore, the omission of a particular feature does not mean that the feature is absent; it may mean that the feature was suppressed. In this appendix, exhaustive coverage of all the features of the UML diagrams is *not* presented. Instead, I will focus on the standard options, especially those options that have been used in this book.

➢ CLASS DIAGRAMS

To model classes, including their attributes, operations, and their relationships and associations with other classes,2 UML provides a *class diagram.* A class diagram provides a static or structural view of a system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram.

The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. An *attribute* refers to something that an object of that class knows or can provide all the time. Attributes are usually implemented as fields of the class, but they need not be.

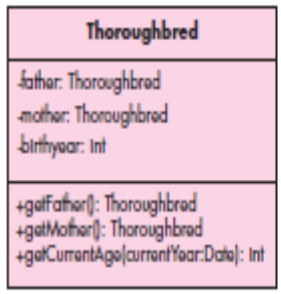
They could be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed. For example, an object might always know the current time and be able to return it to you whenever you ask. Therefore, it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested.

The third section of the class diagram contains the operations or behaviors of the class. An *operation* refers to what objects of the class can do. It is usually implemented as a method of the class.

Figure A1.1 presents a simple example of a **Thoroughbred** class that models thoroughbred horses. It has three attributes displayed—mother, father, and birthyear.

The diagram also shows three operations: *getCurrentAge(), getFather(),* and *getMother()*. There may be other suppressed attributes and operations not shown in the diagram.

FIGURE A1.1
A class diagram for a Thoroughbred class



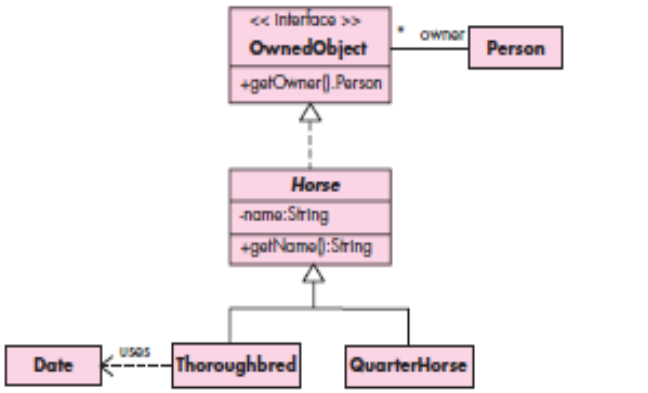
Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding –, #, ~, or +, indicating, respectively, *private, protected, package,* or *public* visibility. In Figure A1.1, all attributes have

private visibility, as indicated by the leading minus sign (–). You can also specify that an attribute is a static or class attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type.

An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the **Horse** class in Figure A1.2 for an example. An interface is indicated by adding the phrase "«interface»" (called a *stereotype*) above the name. See the **OwnedObject** interface in Figure A1.2. An interface can also be represented graphically by a hollow circle. It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This section is particularly useful when transitioning from CRC cards (Chapter 6) to class diagrams in that the responsibilities listed on the CRC cards can be added to this fourth section in the class box in the UML diagram before creating the attributes and operations that carry out these responsibilities.

This fourth section is not shown in any of the figures in this appendix. Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a *generalization*. For example, in Figure A1.2, the **Thoroughbred** and **QuarterHorse** classes are shown to be subclasses of the **Horse** abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a *realization*. For example, in Figure A1.2, the **Horse** class implements or realizes the **OwnedObject** interface.

FIGURE A1.2
A class diagram regarding horses



An *association* between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A1.2, there is an association between **OwnedObject** and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class.

An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the first class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A1.2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A1.2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The

attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type. A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them. However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A1.2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses." The *multiplicity* of one end of an association means the number of objects of thatclass associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a **Person** class and a **SocialSecurityNumber** class in a class diagram. A multiplicity specified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*" means zero or more. An * was used as the multiplicity on the **OwnedObject** end of the association with class **Person** in Figure A1.2 because a **Person** can own zero or more objects.

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An *aggregation* is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered a "part" of the class at the diamond end of the association.

A *composition* is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner. See Figure A1.3 for examples of aggregationand composition.

A **College** has an aggregation of **Building** objects, which represent the buildings making up the campus. The college also has a collection of courses. If the college were to fold, the buildings would still exist (assuming the college wasn't physically destroyed) and could be used for other things, but a **Course** object has no use outside of the college at which it is being offered. If the college were to cease to exist as a business entity, the **Course** object would no longer be useful and so it would also cease to exist.

Another common element of a class diagram is a *note*, which is represented by a box with a dog-eared corner and is connected to other icons by a dashed line. It can have arbitrary content (text and graphics) and is similar to comments in programming languages. It might contain comments about the role of a class or constraints that all objects of that class must satisfy. If the contents are a constraint, braces surround the contents. Note the constraint attached to the **Course** class in Figure A1.3.

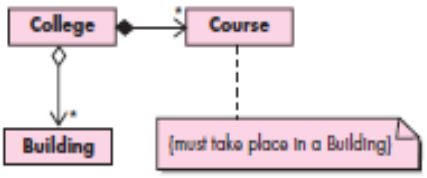


FIGURE A1.3 The relationship between Colleges, Courses, and Buildings

➢ DEPLOYMENT DIAGRAMS

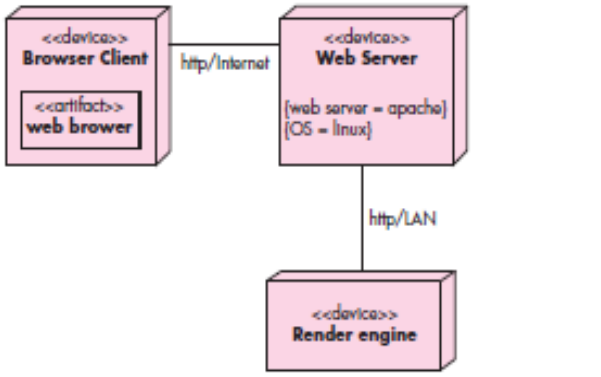
A UML *deployment diagram* focuses on the structure of a software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments. Suppose, for example, you are developing a Web-based graphics-rendering package. Users of your package will use their Web browser to go to your website and enter rendering information. Your website would render a graphical

image according to the user's specification and send it back to the user. Because graphics rendering can be computationally expensive, you decide to move the rendering itself off the Web server and onto a separate platform. Therefore, there will be three hardware devices involved in your system: the Web client (the users' computer running a browser), the computer hosting the Web server, and the computer hosting the rendering engine.

Figure A1.4 shows the deployment diagram for such a package. In such a diagram, hardware components are drawn in boxes labeled with "«device»". Communication paths between hardware components are drawn with lines with optional labels. In Figure A1.4, the paths are labeled with the communication protocol and the type of network used to connect the devices.

Each node in a deployment diagram can also be annotated with details about the device. For example, in Figure A1.4, the browser client is depicted to show that it contains an artifact consisting of the Web browser software. An artifact is typically a file containing software running on a device. You can also specify tagged values, as is shown in Figure A1.4 in the Web server node. These values define the vendor of the Web server and the operating system used by the server. Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label "«execution environment»". These nodes represent systems, such as operating systems, that can host other software.

FIGURE A1.4
A deployment diagram



➢ USE-CASE DIAGRAMS

Use cases (Chapters 5 and 6) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, I'll create some for a software application for managing digital music files, similar to Apple's iTunes software. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

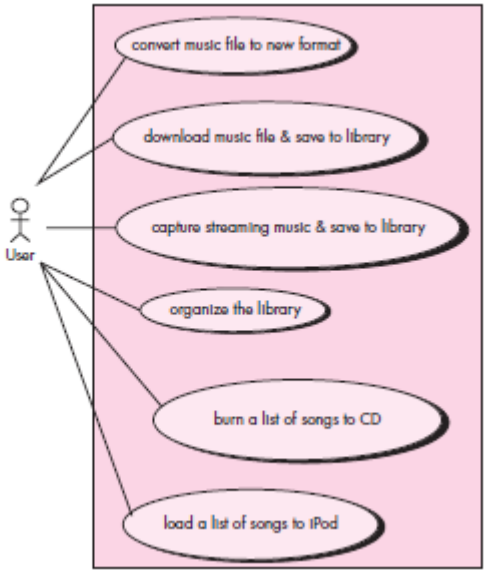
This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD). Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

In this diagram, the stick figure represents an *actor* (Chapter 5) that is associated with one category of user (or other interaction element). *C*omplex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

FIGURE A1.5
A use-case diagram for the music system

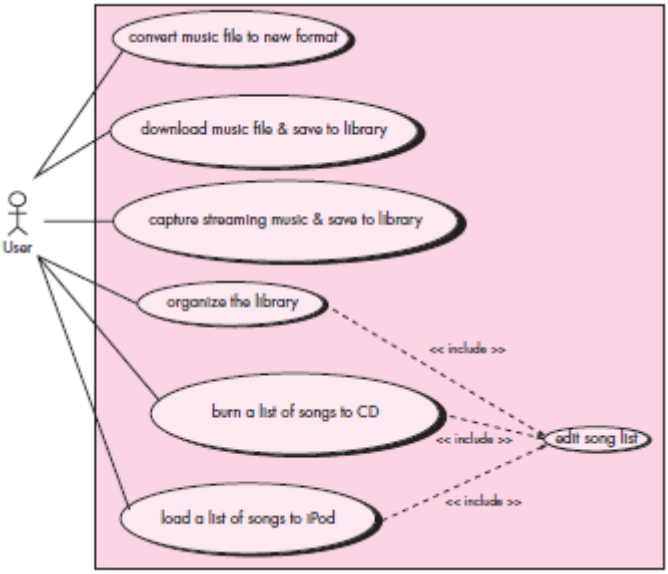


In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the actors are outside the system.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps. Such inclusion is indicated in use-case diagrams, as in Figure A1.6, by means of a dashed arrow labeled «include» connecting a use case with an included use case.

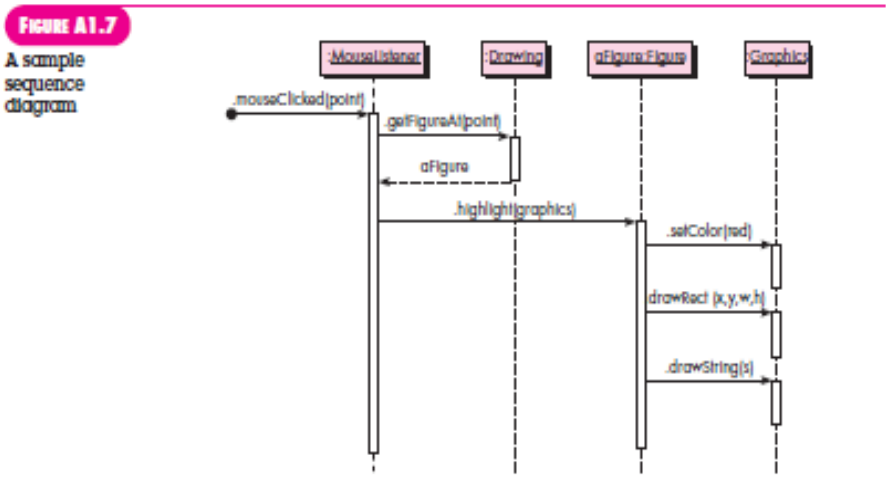
A use-case diagram, because it displays all use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In the digital music organizer, you would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most valuable contribution of use cases to the software development process is the textual description of each use case, not the overall use-case diagram. [Fow04b]. It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

FIGURE A1.6
A use-case diagram with included use cases

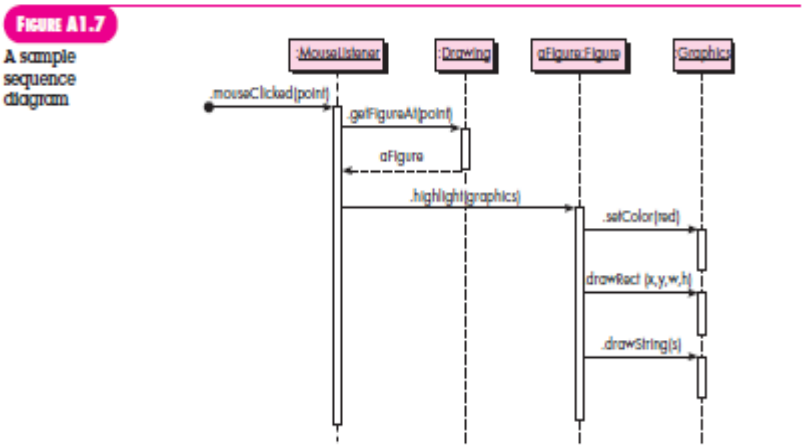


➢ SEQUENCE DIAGRAMS

In contrast to class diagrams and deployment diagrams, which show the static structure of a software component, a *sequence diagram* is used to show the dynamic communications between objects during execution of a task. It shows the temporal order in which messages are sent between the objects to accomplish that task. One might use a sequence diagram to show the interactions in one use case or in one scenario of a software system.



In Figure A1.7, you see a sequence diagram for a drawing program. The diagram shows the steps involved in highlighting a figure in a drawing when it has been clicked. Each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes. If the box represents an object (as is the case in all our examples), then inside the box you can optionally state the type of the object preceded by the colon. You can also precede the colon and type by a name for the object, as shown in the third box in Figure A1.7. Below each box there is a dashed line called the *lifeline* of the object.
The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.
A sequence diagram shows method calls using horizontal arrows from the *caller* to the *callee*, labeled with the method name and optionally including its parameters, their types, and the return type. For example, in Figure A1.7, the **MouseListener** calls the **Drawing**'s *getFigureAt()* method. When an object is executing a method (that is, when it has an activation frame on the stack), you can optionally display a white bar, called an *activation bar*, down the object's lifeline. In Figure A1.7, activation bars are drawn for all method calls. The diagram can also optionally show the return from a method call with a dashed arrow and an optional label. In Figure A1.7, the *getFigureAt()* method call's return is shown labeled with the name of the object that was returned. A common practice, as we have done in Figure A1.7, is to leave off the return arrow when a void method has been called, since it clutters up the diagram while providing little information of importance. A black circle with an arrow coming from it indicates a *found message* whose source is unknown or irrelevant.



You should now be able to understand the task that Figure A1.7 is displaying. An unknown source calls the *mouseClicked()* method of a **MouseListener**, passing in the point where the click occurred as the argument. The **MouseListener** in turn calls the *getFigureAt()* method of a **Drawing**, which returns a **Figure.** The **MouseListener** then calls the highlight method of **Figure**, passing in a **Graphics** object as an argument. In response, **Figure** calls three methods of the **Graphics** object to draw the figure in red.

The diagram in Figure A1.7 is very straightforward and contains no conditionals or loops. If logical control structures are required, it is probably best to draw a separate sequence diagram for each case. That is, if the message flow can take two different paths depending on a condition, then draw two separate sequence diagrams, one for each possibility.

If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use *interaction frames*, which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent. Figure A1.8 illustrates this, showing the process involved in highlighting all figures inside a given rectangle. The **MouseListener** is sent the rectDragged message.

The **MouseListener** then tells the drawing to highlight all figures in the rectangle by called the method *highlightFigures()*, passing the rectangle as the argument. The method loops through all **Figure** objects in the **Drawing** object and, if the **Figure** intersects the rectangle, the **Figure** is asked to highlight itself. The phrases in square brackets are called *guards*, which are Boolean conditions that must be true if the action inside the interaction frame is to continue.

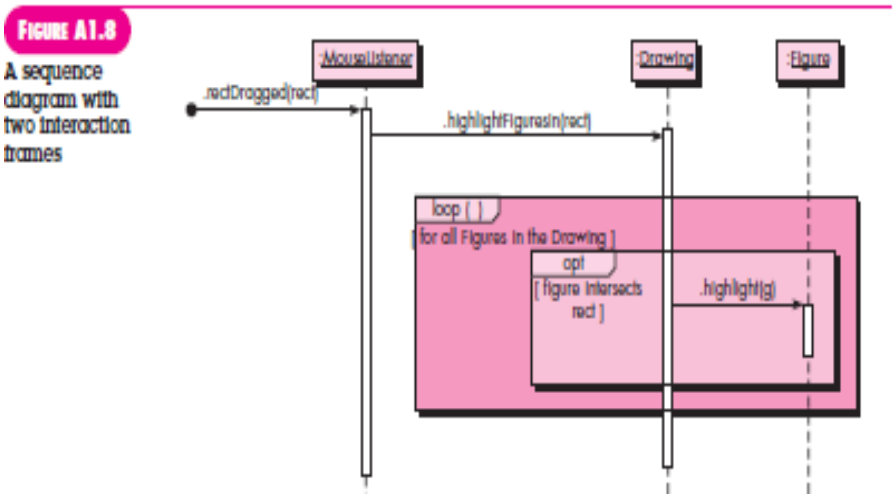


FIGURE A1.8
A sequence diagram with two interaction frames

There are many other special features that can be included in a sequence diagram. For example:

1. You can distinguish between synchronous and asynchronous messages Synchronous messages are shown with solid arrowheads while asynchronous messages are shown with stick arrowheads.
2. You can show an object sending itself a message with an arrow going out from the object, turning downward, and then pointing back to the same object.
3. You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label) to an object's box. In this case, the box will appear lower in the diagram than the boxes corresponding to objects already in existence when the action begins.
4. You can show object destruction by a big X at the end of the object's lifeline. Other objects can destroy an object, in which case an arrow points from the other object to the X. An X is also useful for indicating that an object is no longer usable and so is ready for garbage collection.

The last three features are all shown in the sequence diagram in Figure A1.9.

- **COMMUNICATION DIAGRAMS**

The UML *communication diagram* (called a "collaboration diagram" in UML 1.X) provides another indication of the temporal order of the communications but emphasizes the relationships among the objects and classes instead of the temporal order. A communication diagram, illustrated in Figure A1.10, displays the same actions shown in the sequence diagram in Figure A1.7.

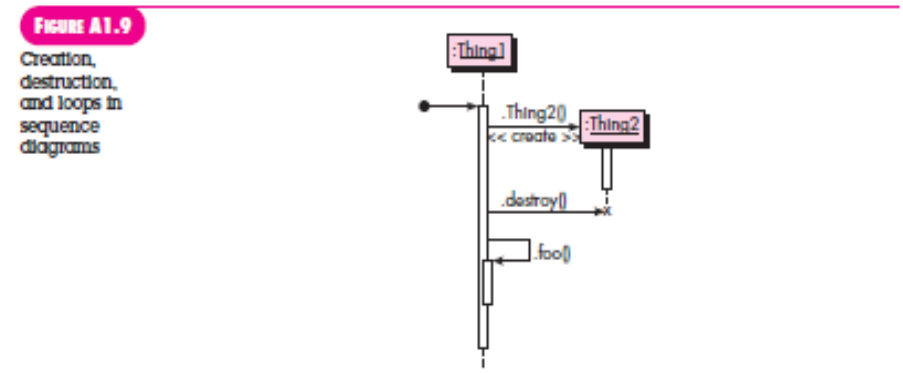


FIGURE A1.9
Creation, destruction, and loops in sequence diagrams

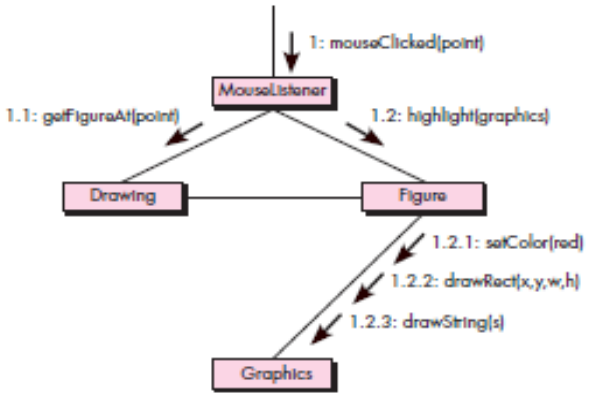


FIGURE A1.10
A UML communication diagram

In a communication diagram the interacting objects are represented by rectangles. Associations between objects are represented by lines connecting the rectangles. There is typically an incoming arrow to one object in the diagram that starts the sequence of message passing. That arrow is labeled with a number and a message name. If the incoming message is labeled with the number 1 and if it causes the receiving object to invoke other messages on other objects, then those messages are represented by arrows from the sender to the receiver along an association line and are given numbers 1.1, 1.2, and so forth, in the order they are called. If those messages in turn invoke other messages, another decimal point and number are added to the number labeling these messages, to indicate further nesting of the message passing.

In Figure A1.10, you see that the **mouseClicked** message invokes the methods *getFigureAt()* and then *highlight().* The *highlight()* message invokes three other messages: *setColor(), drawRect(),* and *drawstring().* The numbering in each label shows the nesting as well as the sequential nature of each message.

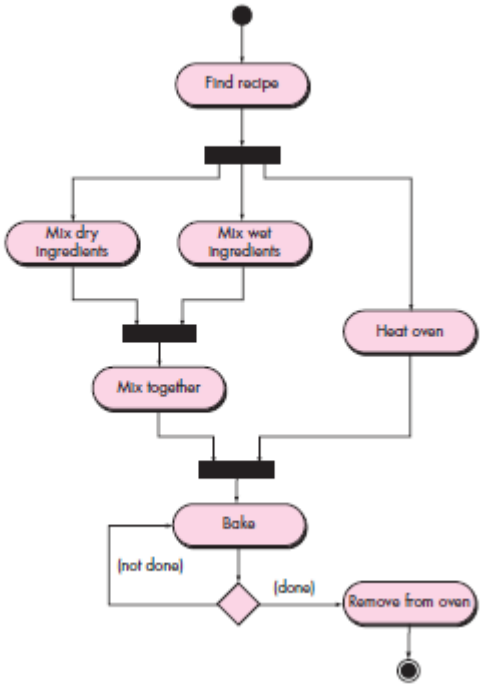
There are many optional features that can be added to the arrow labels. For example, you can precede the number with a letter. An incoming arrow could be labeled A1: mouseClicked(point). indicating an execution thread, A. If other messages are executed in other threads, their label would be preceded by a different letter. For example, if the *mouseClicked()* method is executed in thread A but it creates a new thread B and invokes *highlight()* in that thread, then the arrow from **MouseListener** to **Figure** would be labeled 1.B2: highlight(graphics). If you are interested in showing the relationships among the objects in addition to the messages being sent between them, the communication diagram is probably a better option than the sequence diagram. If you are more interested in the temporal order of the message passing, then a sequence diagram is probably better.

- **ACTIVITY DIAGRAMS**

A UML *activity diagram* depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows. The main component of an activity diagram is an *action* node, represented by a rounded rectangle, which corresponds to a task performed by the software system. Arrows from one action node to another indicate the flow of control. That is, an arrow between two action nodes means that after the first action is complete the second action begins. A solid black dot forms the *initial node* that indicates the starting point of the activity. A black dot surrounded by a black circle is the *final node* indicating the end of the activity.

A *fork* represents the separation of activities into two or more concurrent activities. It is drawn as a horizontal black bar with one arrow pointing to it and two or more arrows pointing out from it. Each outgoing arrow represents a flow of control that can be executed concurrently with the flows corresponding to the other outgoing arrows. These concurrent activities can be performed on a computer using different threads or even using different computers. Figure A1.11 shows a sample activity diagram involving baking a cake. The first step is finding the recipe. Once the recipe has been found, the dry ingredients and wet ingredients can be measured and mixed and the oven can be preheated. The mixing of the dry ingredients can be done in parallel with the mixing of the wet ingredients and the preheating of the oven.

FIGURE A1.11
A UML activity diagram showing how to bake a cake



A *join* is a way of synchronizing concurrent flows of control. It is represented by a horizontal black bar with two or more incoming arrows and one outgoing arrow. The flow of control represented by the outgoing arrow cannot begin execution until all flows represented by incoming arrows have been completed. In Figure A1.11, we have a join before the action of mixing together the wet and dry ingredients. This join indicates that all dry ingredients must be mixed and all wet ingredients must be mixed before the two mixtures can be combined. The second join in the figure indicates that, before the baking of the cake can begin, all ingredients must be mixed together and the oven must be at the right temperature.

A *decision* node corresponds to a branch in the flow of control based on a condition. Such a node is displayed as a white triangle with an incoming arrow and two or more outgoing arrows. Each outgoing arrow is labeled with a guard (a condition inside square brackets). The flow of control follows the outgoing arrow whose guard is true. It is advisable to make sure that the conditions cover all possibilities so that exactly one of them is true every time a decision node is reached. Figure A1.11 shows a decision node following the baking of the cake. If the cake is done, then it is removed from the oven. Otherwise, it is baked for a while longer.

One of the things the activity diagram in Figure A1.11 does not tell you is who or what does each of the actions. Often, the exact division of labor does not matter. But if you do want to indicate how the actions are divided among the participants, you can decorate the activity diagram with swimlanes, as shown in Figure A1.12. *Swimlanes*, as the name implies, are formed by dividing the diagram into strips or "lanes," each of which corresponds to one of the participants. All actions in one lane are done by the corresponding participant. In Figure A1.12, Evan is responsible for mixing the dry ingredients and then mixing the dry and wet ingredients together, Helen is responsible for heating the oven and taking the cake out, and Mary is responsible for everything else.

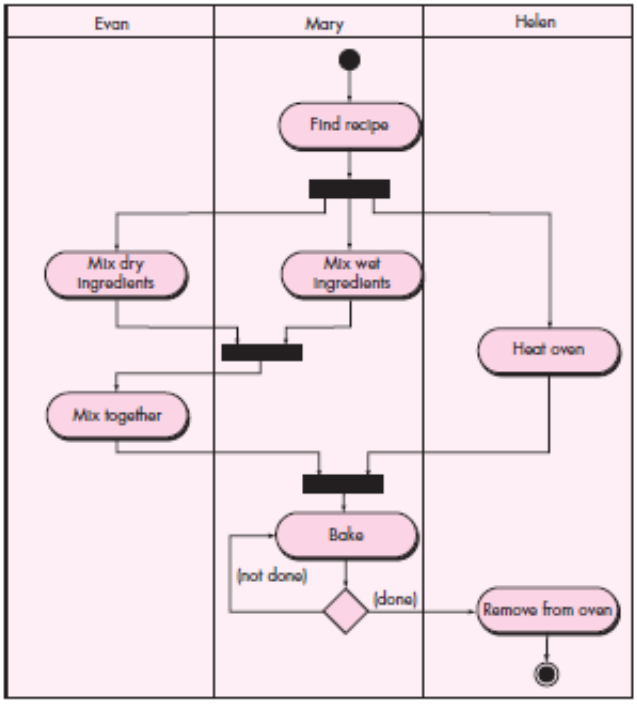


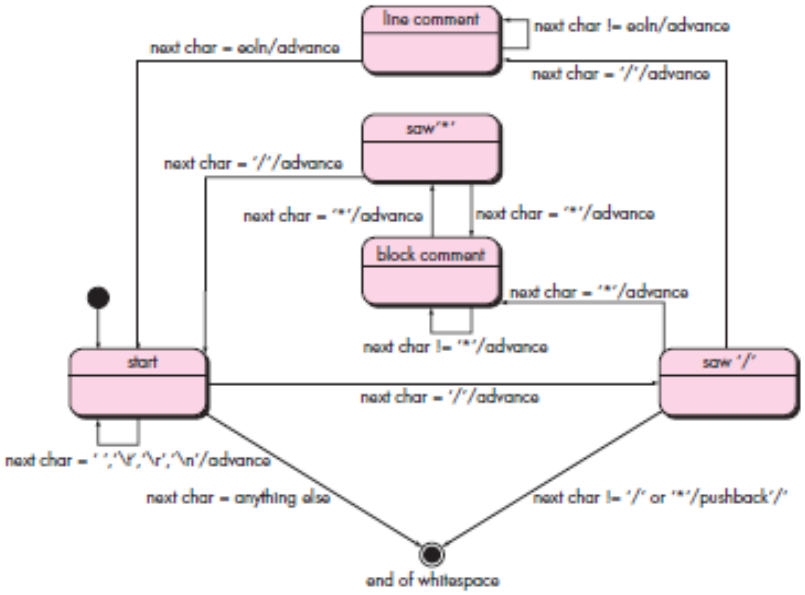
FIGURE A1.12 The cake-baking activity diagram with swimlanes added

➢ STATE DIAGRAMS

The behavior of an object at a particular point in time often depends on the state of the object, that is, the values of its variables at that time. As a trivial example, consider an object with a Boolean instance variable. When asked to perform an operation, the object might do one thing if that variable is *true* and do something else if it is *false.* A UML *state diagram* models an object's states, the actions that are performed depending on those states, and the transitions between the states of the object.

As an example, consider the state diagram for a part of a Java compiler. The input to the compiler is a text file, which can be thought of as a long string of characters. The compiler reads characters one at a time and from them determines the structure of the program. One small part of this process of reading the characters involves ignoring "white-space" characters (e.g., the *space, tab, newline,* and *return* characters) and characters inside a comment.

FIGURE A1.13 A state diagram for advancing past white space and comments in Java



Suppose that the compiler delegates to a **WhiteSpaceAndCommentEliminator** the job of advancing over white-space characters and characters in comments. That is, this object's job is to read input characters until all white-space and comment characters have been read, at which point it returns control to the compiler to read and process non-white-space and noncomment characters. Think about how the **WhiteSpaceAndCommentEliminator** object reads in characters and determines whether the next character is white space or part of a comment. The object can check for white space by testing the next character against " ", "\t", "\n", and "\r". But how does it determine whether the next character is part of a comment? For example, when it sees a"/"for the first time, it doesn't yet know whether that character represents a division operator, part of the /= operator, or the beginning of a line or block comment. To make this determination, **WhiteSpaceAndCommentEliminator** needs to make a note of the fact that it saw a "/" and then move on to the next character. If the character following the "/" is another "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that it is now reading a comment and can advance to the end of the comment without processing or saving any characters. If the character following the first "/" is anything other than a "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that the "/" represents the division operator or part of the /= operator and so it stops advancing over characters. In summary, as **WhiteSpaceAndCommentEliminator** reads in characters, it needs to keep track of several things, including whether the current character is white space, whether the previous character it read was a "/", whether it is currently reading characters in a comment, whether it has reached the end of comment, and so forth. These all correspond to different states of the **WhiteSpaceAndCommentEliminator** object. In each of these states, **WhiteSpaceAndCommentEliminator** behaves differently with regard to the next character read in.

To help you visualize all the states of this object and how it changes state, you can use a UML state diagram as shown in Figure A1.13. A state diagram displays states using rounded rectangles, each of which has a name in its upper half. There is also a black circle called the "initial pseudostate," which isn't really a state and instead just points to the initial state. In Figure A1.13, the **start** state is the initial state. Arrows from one state to another state indicate transitions or changes in the state of the object. Each transition is labeled with a trigger event, a slash (/), and an activity. All parts of the transition labels are optional in state diagrams. If the object is in one state and the trigger event for one of its transitions occurs, then that transition's activity is performed and the object takes on the new state indicated by the transition. For example, in Figure A1.13, if the **WhiteSpaceAndCommentEliminator** object is in the **start** state and the next character is "/", then **WhiteSpaceAndCommentEliminator** advances past that character and changes to the **saw '/'** state. If the character after the "/" is another "/", then the object advances to the **line comment** state and stays there until it reads an end-of-line character. If instead the next character after the "/" is a "*", then the object advances to the **block comment** state and stays there until it sees another "*" followed by a "/", which indicates the end of the block comment. Study the diagram to make sure you understand it. Note that, after advancing past white space or a comment, **WhiteSpaceAndCommentEliminator** goes back to the **start** state and starts over. That behavior is necessary since there might be several successive comments or white-space characters before any other characters in the Java source code. An object may transition to a final state, indicated by a black circle with a white circle around it, which indicates there are no more transitions. In Figure A1.13, the **WhiteSpaceAndCommentEliminator** object is finished when the next character is not white space or part of a comment. Note that all transitions except the two transitions leading to the final state have activities consisting of advancing to the next character. The two transitions to the final state do not advance over the next character because the next character is part of a word or symbol of interest to the compiler. Note that if the object is in the **saw '/'** state but the next character is not "/" or "*", then the "/" is a division operator or part of the /= operator and so we don't want to advance. In fact, we want to back up one character to make the "/" into the next character, so that the "/" can be used by the compiler. In Figure A1.13, this activity of backing up is labeled as pushback '/'. A state diagram will help you to uncover missed or unexpected situations. That is, with a state diagram, it is relatively easy to ensure that all possible trigger events for all possible states have been accounted for. For example, in Figure A1.13, you can easily verify that every state has included transitions for all possible characters.

UML state diagrams can contain many other features not included in Figure A1.13. For example, when an object is in a state, it usually does nothing but sit and wait for a trigger event to occur. However, there is a special kind of state, called an *activity state*, in which the object performs some activity, called a *do-activity*,

while it is in that state. To indicate that a state is an activity state in the state diagram, you include in the bottom half of the state's rounded rectangle the phrase "do/" followed by the activity that is to be done while in that state. The do-activity may finish before any state transitions occur, after which the activity state behaves like a normal waiting state. If a transition out of the activity state occurs before the do-activity is finished, then the do-activity is interrupted. Because a trigger event is optional when a transition occurs, it is possible that no trigger event may be listed as part of a transition's label. In such cases for normal waiting states, the object will immediately transition from that state to the new state. For activity states, such a transition is taken as soon as the do-activity finishes. Figure A1.14 illustrates this situation using the states for a business telephone.

When a caller is placed on hold, the call goes into the **on hold with music** state (soothing music is played for 10 seconds). After 10 seconds, the do-activity of the state is completed and the state behaves like a normal nonactivity state. If the caller pushes the # key when the call is in the **on hold with music** state, the call transitions to the **canceled** state and then transitions immediately to the **dial tone** state. If the # key is pushed before the 10 seconds of soothing music has completed, the do-activity is interrupted and the music stops immediately.

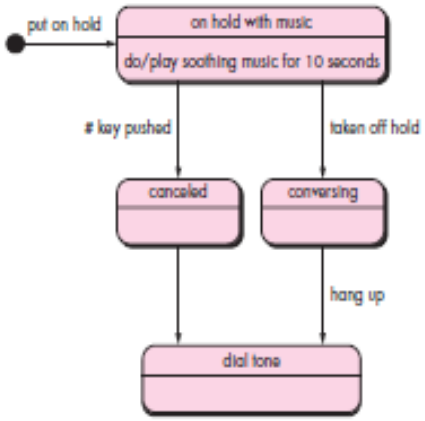


FIGURE A1.14
A state diagram with an activity state and a triggerless transition