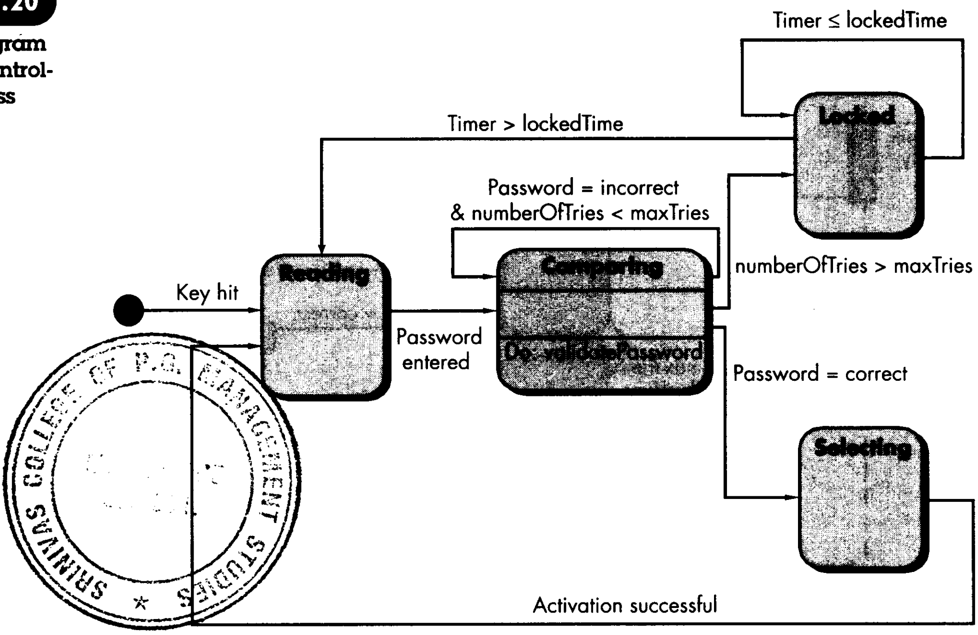
FIGURE 8.20

State diagram for the Control-Panel class



transition to occur. For example, the guard for the transition from the "reading" state to the "comparing state" in Figure 8.20 can be determined by examining the use-case:

if (password input = 4 digits) then compare to stored password

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

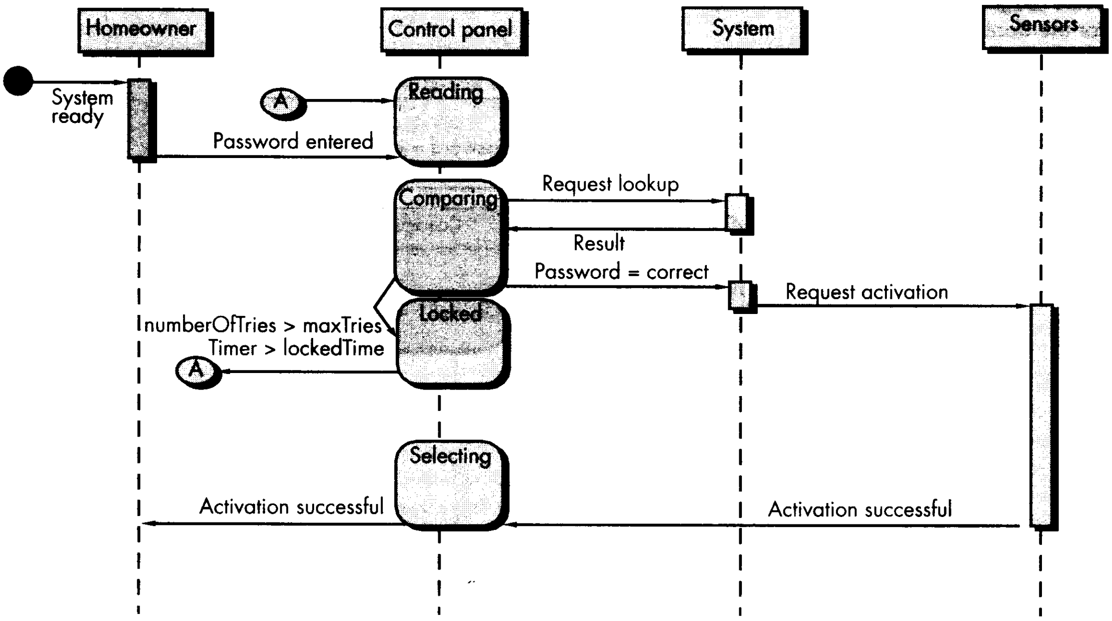
KEY POINT

Unlike a state diagram that represents behavior without noting the classes involved, a sequence diagram represents behavior by describing how classes move from state to state.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 8.20) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

Sequence diagrams. The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use-case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to

FIGURE 8.21 Sequence diagram (partial) for the *SafeHome* security function



another as a function of time. In essence, the sequence diagram is a shorthand version of the use-case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 8.21 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use-case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical timeline.

The first event, *system ready,* is derived from the external environment and channels behavior to a **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System** which activates sensors with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

SOFTWARE TOOLS

Generalized Analysis Modeling in UML

Objective: Analysis modeling tools provide the capability to develop scenario-based models, class-based models, and behavioral models using UML notation.

Mechanics: Tools in this category support the full range of UML diagrams required to build an analysis model (these tools also support design modeling). In addition to diagramming, tools in this category (1) perform consistency and correctness checks for all UML diagrams; (2) provide links for design and code generation; (3) build a database that enables the management and assessment of large UML models required for complex systems.

Representative Tools[27]

The following tools support a full range of UML diagrams required for analysis modeling:

ArgoUML, an open source tool (argouml.tigris.org).

Control Center, developed by TogetherSoft (www.togethersoft.com).

Enterprise Architect, developed by Sparx Systems (www.sparxsystems.com.au).

Object Technology Workbench (OTW), developed by OTW Software (www.otwsoftware.com).

PowerDesigner, developed by Sybase (www.sybase.com).

Rational Rose, developed by Rational Corporation (www.rational.com).

System Architect, developed by Popkin Software (www.popkin.com).

UML Studio, developed by Pragsoft Corporation (www.pragsoft.com).

Visio, developed by Microsoft (www.microsoft.com).

Visual UML, developed by Visal Object Modelers (www.visualuml.com).

8.9 SUMMARY

The objective of analysis modeling is to create a variety of representations that depict software requirements for information, function, and behavior. To accomplish this, two different (but potentially complementary) modeling philosophies can be applied: structured analysis and object-oriented analysis. Structured analysis views software as an information transformer. It assists the software engineer in identifying data objects, their relationships, and the manner in which those data objects are transformed as they flow through software processing functions. Object-oriented analysis examines a problem domain defined as a set of use-cases in an effort to extract classes that define the problem. Each class has a set of attributes and operations. Classes are related to one another in a variety of different ways and are modeled using UML diagrams. The analysis model is composed of four modeling elements: scenario-based models, flow models, class-based models, and behavioral models.

Scenario-based models depict software requirements from the user's point of view. The use-case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirement elicitation, the use-case defines the key steps for a specific function or interaction. The

[27] Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8.8. What is an analysis package, and how might it be used?

8.9. Develop CSPECs and PSPECs for the system you selected in Problem 8.6. Try to make your model as complete as possible.

8.10. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a Web site and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on a crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material, and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes the citizen's name, address, phone number, type of damage, dollar amount of damage. PHTRS is a Web-based system; all queries are to be made interactively.

Using structured analysis notation, develop an analysis model for PHTRS.

8.11. Describe the object-oriented terms *encapsulation* and *inheritance.*

8.12. Using the context-level DFD developed in Problem 8.7, develop level 1 and level 2 data flow diagrams. Use a "grammatical parse" on the context-level processing narrative to get yourself started. Remember to specify all information flow by labeling all arrows between bubbles. Use meaningful names for each transform.

8.13. How does a state diagram for analysis classes differ from the state diagrams presented for the complete system?

8.14. Develop a class model for the PHTRS system introduced in Problem 8.10.

8.15. Develop a complete set of CRC model index cards for the PHTRS system introduced in Problem 8.10.

8.16. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

8.17. Describe the difference between an association and a dependency for an analysis class.

8.18. Draw a UML use-case diagram for the PHTRS system introduced in Problem 8.10. You'll have to make a number of assumptions about the manner in which a user interacts with this system.

8.19. Write a template-based use-case for the *SafeHome* home management system described informally in the sidebar following Section 8.7.4.

Further Readings and Information Sources

Dozens of books have been published on structured analysis. Most cover the subject adequately, but only a few do a truly excellent job. DeMarco and Plauger (*Structured Analysis and System Specification,* Pearson, 1985) is a classic that remains a good introduction to the basic notation. Books by Kendall and Kendall (*Systems Analysis and Design,* fifth edition, Prentice-Hall, 2002) and Hoffer et al. (*Modern Systems Analysis and Design,* Addison-Wesley, third edition., 2001) are worthwhile references. Yourdon's book (*Modern Structured Analysis,* Yourdon-Press, 1989) on the subject remains among the most comprehensive coverage published to date.

Allen (*Data Modeling for Everyone,* Wrox Press, 2002), Simpson and Witt (*Data Modeling Essentials,* second edition, Coriolis Group, 2000) Reingruber and Gregory (*Data Modeling Handbook,* Wiley, 1995) present detailed tutorials for creating industry-quality data models. An interesting book by Hay (*Data Modeling Patterns,* Dorset House, 1995) presents typical data model patterns that are encountered in many different businesses. A detailed treatment of behavioral modeling can be found in Kowal (*Behavior Models: Specifying User's Expectations,* Prentice-Hall, 1992).

Use-cases form the foundation of object-oriented analysis. Books by Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [COC01], Armour and Miller (*Advanced Use-Case Modeling: Software Systems,* Addison-Wesley, 2000), and Rosenberg and Scott (*Use-Case Driven Object Modeling with UML: A Practical Approach,* Addison-Wesley, 1999) provide worthwhile guidance in the creation and use of this important requirements elicitation and representation mechanism.

Worthwhile discussions of UML have been written by Arlow and Neustadt [ARL02], Schmuller [SCH02], Fowler and Scott (*UML Distilled,* second edition, Addison-Wesley, 1999), Booch and his colleagues (*The UML User Guide,* Addison-Wesley, 1998), and Rumbaugh and his colleagues (*The Unified Modeling Language Reference Manual,* Addison-Wesley, 1998).

The underlying analysis and design methods that support the Unified Process are discussed by Larman (*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process,* second edition, Prentice-Hall, 2001), Dennis and his colleagues (*System Analysis and Design: An Object-Oriented Approach with UML,* Wiley, 2001), and Rosenberg and Scott (*Use-Case Driven Object Modeling with UML,* Addison-Wesley, 1999). Balcer and Mellor (*Executable UML: A Foundation for Model Driven Architecture,* Addison-Wesley, 2002) discuss the overall semantics of UML, the models that can be created, and a way to consider UML as an executable language. Starr (*Executable UML: How to Build Class Models,* Prentice-Hall, 2001) provides useful guidelines and detailed suggestions for creating effective analysis and design classes.

A wide variety of information sources on analysis modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

9 Design Engineering

Key Concepts

abstraction
architecture
design classes
model elements
quality
functional independence
information hiding
modularity
patterns
refactoring
refinement

Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles (discussed in Chapter 5) establish an overriding philosophy that guides the designer in the work that is performed. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design engineering is not a commonly used phrase in the software engineering context. And yet, it should be. Design is a core engineering activity. In the early 1990s Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:

> What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. . .
>
> The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software. *Firmness:* A program should not have any bugs that inhibit its function. *Commodity:* A program should be suitable for the purposes for which it was intended. *Delight:* The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software.

Quick Look

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the analysis model (that focuses on describing required data, function, and behavior), the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks.

Why is it important? Design allows a software engineer to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end-users become involved in large numbers. Design is the place where software quality is established.

What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end-users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of [illegible] ferent design action, but all [illegible] of basic design concepts that guide all [illegible] design work.

What is the work product? A design [illegible] that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design.

[illegible] that I've done it right? [illegible] assessed by the software [illegible] in an effort to determine whether it contains [illegible], inconsistencies, or omissions; [illegible] alternatives exist, and whether the design can be implemented within the constraints, schedule, and cost that have been established.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Belady [BEL81] states that "diversification is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind." Once this diverse set of information is assembled, the designer must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering (Chapter 7) and the analysis model (Chapter 8). As this occurs, alternatives are considered and rejected, and the design engineer converges on "one particular configuration of components, and thus the creation of the final product" [BEL81].

Diversification and convergence demand intuition and judgment. These qualities are based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve. Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied. In this chapter, we explore the fundamental concepts and principles that are applicable to all software design, the elements of the design model, and the impact of patterns on the design process. In Chapters 10, 11, and 12 we examine a variety of software design methods as they are applied to architectural, interface, and component-level design.

9.1 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).

> "The most common miracle of software engineering is the transition from analysis to design and design to code."
> Richard Due

ADVICE

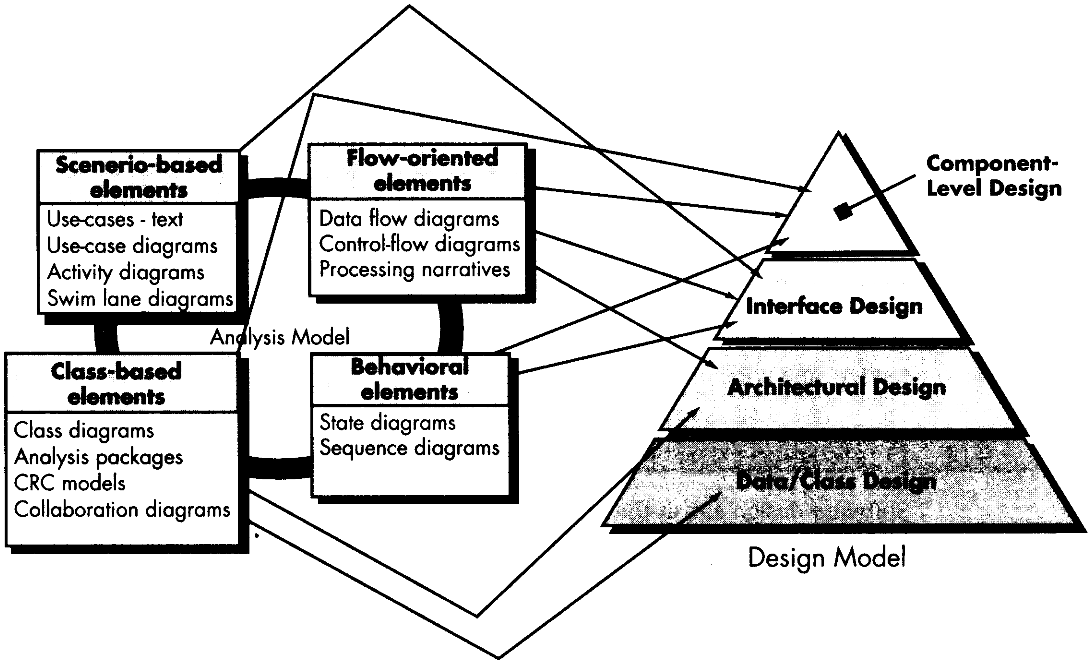
Design engineering should always begin with a consideration of data—the foundation for all other elements of the design. After the foundation is laid, the architecture must be derived. Only then, should you perform other design tasks.

Each of the elements of the analysis model (Chapter 8) 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 9.1. The analysis 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 analysis-class models (Chapter 8) into design class realizations and the requisite data structures required to implement the software. The classes and relationships defined by CRC index cards and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity. 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

FIGURE 9.1 Translating the analysis model into the design model



the way in which architectural can be implemented [SHA96]. The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis 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.

> "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are obviously no deficiencies. The first method is far more difficult."
>
> C.A.R. Hoare

During design we 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 us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer'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, we 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.

9.2 Design Process and Design Quality

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.

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in Chapter 26. McGlaughlin [MCG91] 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 analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- 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?

> "[W]riting a clever piece of code that works is one thing; designing something that can support a long-lasting business is quite another."
>
> C. Ferguson

Quality guidelines. In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this chapter, we discuss design concepts that also serve as software quality criteria. For the time being, we present the following guidelines:

What are the characteristics of a good design?

1. A design should exhibit an architecture that (a) has been created using recognizable architectural styles or patterns, (b) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (c) can be implemented in an evolutionary fashion,[1] 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.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

[1] For smaller systems, design can sometimes be developed linearly.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

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

INFO

Assessing Design Quality—The Formal Technical Review

Design is important because it allows a software team to assess the quality[2] of the software before it is implemented—at a time when errors, omissions, or inconsistencies are easy and inexpensive to correct. But how do we assess quality during design? The software can't be tested because there is no executable software to test. What to do?

During design, quality is assessed by conducting a series of formal technical reviews (FTRs). FTRs are discussed in detail in Chapter 26,[3] but it's worth providing a summary of the technique at this point. An FTR is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed. Each person plays a role: the *review leader* plans the meeting, sets an agenda, and then runs the meeting; the *recorder* takes notes so that nothing is missed; the *producer* is the person whose work product (e.g., the design of a software component) is being reviewed. Prior to the meeting, each person on the review team is given a copy of the design work product and is asked to read it, looking for errors, omissions, or ambiguity. When the meeting commences, the intent is to note all problems with the work product so that they can be corrected before implementation begins. The FTR typically lasts between 90 minutes and two hours. At the conclusion of the FTR, the review team determines whether further actions are required on the part of the producer before the design work product can be approved as part of the final design model.

> **"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree."**
>
> **Robert Pirsig**

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.

[2] The quality factors discussed in Chapter 15 can assist the review team as it assesses quality.

[3] You might consider reviewing Section 26.4 at this time. FTRs are a critical part of the design process and are an important mechanism for achieving design quality.

ADVICE

Software designers tend to focus on the problem to be solved. Just don't forget that the FURPS attributes are always part of the problem. They must be considered.

- *Usability* is assessed by considering human factors (Chapter 12), 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 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*—in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, (Chapter 27), 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.

TASK SET

Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture.
   Be certain that each subsystem is functionally cohesive.
   Design subsystem interfaces.
   Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components.
   Translate each analysis class description into a design class.
   Check each design class against design criteria; consider inheritance issues.
   Define methods and messages associated with each design class.
   Evaluate and select design patterns for a design class or a subsystem.
   Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface.
   Review results of task analysis.
   Specify action sequence based on user scenarios.
   Create behavioral model of the interface.
   Define interface objects, control mechanisms.
   Review the interface design and revise as required.
7. Conduct component-level design.
   Specify all algorithms at a relatively low level of abstraction.
   Refine the interface of each component.
   Define component-level data structures.
   Review each component and correct all errors uncovered.
8. Develop a deployment model.

9.3 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.

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."

9.3.1 Abstraction

When we 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.

> **"Abstraction is one of the fundamental ways that we as humans cope with complexity."**
>
> **Grady Booch**

ADVICE

As a designer, work hard to derive both procedural and data abstractions that serve the problem at hand. If they can serve an entire domain of problems, that's even better.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of 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.).[4]

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.**

9.3.2 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

[4] It should be noted, however, that one set of operations can be replaced with another, as long as the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *open* would change dramatically if the door were automatic and attached to a sensor.

(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.

> "A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule, and cost."
>
> **Len Bass et al.**

WebRef

An in depth discussion of software architecture can be found at **www.sei.cmu.edu/ata/ata_init.html.**

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 enable a software engineer to reuse design-level concepts.

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. Architectural design is discussed in Chapter 10.

ADVICE

Don't just let architecture happen. If you do, you'll spend the rest of the project trying to force fit the design. Design architecture explicitly.

9.3.3 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" [APP98]. 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.

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
>
> **Christopher Alexander**

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 more detail in Section 9.5.

9.3.4 Modularity

Software architecture and design patterns embody *modularity;* that is, 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. To illustrate this point, consider the following argument based on observations of human problem solving.

Consider two problems, p_1 and p_2. If the perceived complexity of p_1 is greater than the perceived complexity of p_2, it follows that the effort required to solve p_1 is greater than the effort required to solve p_2. 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 modularity and software. It is, in fact, an argument for modularity.

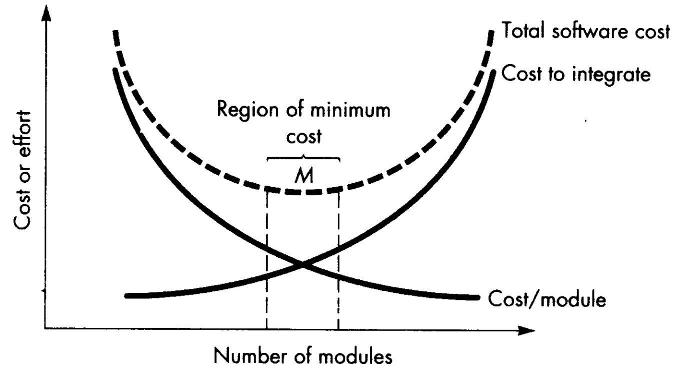
ADVICE

Don't overmodularize. The simplicity of each small module will be overshadowed by the complexity of integration.

It is possible to conclude that, if we 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 9.2, 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

FIGURE 9.2

Modularity and software cost



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.

The curves shown in Figure 9.2 do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of *M.* Undermodularity or overmodularity should be avoided. But how do we know the vicinity of *M*? How modular should we make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

We 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.

9.3.5 Information Hiding

The concept of modularity leads every software designer to a fundamental question: How do we 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].

KEY POINT
The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

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 procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

9.3.6 Functional Independence

The concept of *functional independence* is a direct outgrowth of 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 "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction 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 may 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.

KEY POINT

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

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 9.3.5. 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.

KEY POINT

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

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, we strive for 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.

9.3.7 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.

ADVICE

There is a tendency to move immediately to full detail, skipping refinement steps. This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement.

Refinement is actually a process of *elaboration*. We begin with a statement of function (or description of data) 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 data. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

> "I have not failed. I've just found 10,000 ways that won't work."
>
> Thomas Edison

9.3.8 Refactoring

WebRef
Excellent resources for refactoring can be found at **www.refactoring.com.**

An important design activity suggested for many agile methods (Chapter 4), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [FOW99] 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). The designer 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.

SAFEHOME

Design Concepts

[illegible] Vinod's cubicle, as [illegible] begins.

[illegible] Jamie, and Ed—members of the [illegible] engineering team . Also, Shakira, a [illegible] team.

[illegible] just returned from a [illegible] entitled "Applying Basic Design [illegible] by a local computer science [illegible]

[illegible] anything out of the seminar?

[illegible] the stuff, but it's not a bad idea to hear [illegible]

[illegible] as an undergrad CS major, I never [illegible] why information hiding was as [illegible] it is.

[illegible] bottom line . . . it's a technique for [illegible] concerns in a program. Actually, [illegible] also accomplishes the same

Shakira: I wasn't a CS grad, so a lot of the stuff the instructor mentioned is new to me. I can generate [illegible] code and fast. I don't see why this stuff is so [illegible]

Jamie: I've seen your work, Shak, and [illegible] you do a lot of this stuff naturally . . . that's why your designs and code work.

Shakira (smiling): Well, I always do try to [illegible] the code, keep it focused on one thing, [illegible] simple and constrained, reuse code [illegible] that sort of thing.

Ed: Modularity, functional independence, hiding, patterns . . . see.

Jamie: I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

Vinod: Same thing can be applied to design, [illegible]

Ed: The only concept I hadn't heard of [illegible] "refactoring."

Shakira: That's used in Extreme Programming, [illegible] she said.

Ed: Yep. It's not a whole lot different than refinement, only you do it after the design or code is completed. Kind of an optimization pass through the software, if you ask me.

Jamie: Let's get back to *SafeHome* design. I think we should put these concepts on our review checklist as we develop the design model for *SafeHome*.

Vinod: I agree. But as important, let's all [illegible] about them as we develop the design.

9.3.9 Design Classes

In Chapter 8, we noted that the analysis model defines a complete set of analysis classes. Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user or customer visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, the software team must define a set of *design classes* that (1) refine the analysis classes by providing design detail that will enable the classes to be implemented, and (2) create a new set of design classes that implement a software infrastructure to support the business solution. Five different types of design classes, each representing a different layer of the design architecture are suggested [AMB01]:

What types of classes does the designer create?

- *User interface classes* define all abstractions that are necessary for human-computer 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 design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent 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:

What is 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** of the 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** of the video editing software 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 classes in other subsystems. This restriction, called the *Law of Demeter* [LIE03], suggests that a method should only send messages to methods in neighboring classes.[5]

[5] A less formal way of stating the Law of Demeter is "Each unit should only talk to its friends; don't talk to strangers."

SAFEHOME

Refining an Analysis Class into a Design Class

The scene: Ed's cubicle, as design modeling continues.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

(Ed is working on the **FloorPlan** class [see sidebar discussion in Section 8.7.4 and Figure 8.14] and has refined it for the design model.)

Ed: So you remember the **FloorPlan** class, right? It's used as part of the surveillance and home management functions.

Vinod (nodding): Yeah, I seem to recall that we used it as part of our CRC discussions for home management.

Ed: We did. Anyway, I'm refining it for design. Want to show how we'll actually implement the **FloorPlan** class. My idea is to implement it as a set of linked lists [a specific data structure]. So . . . I had to refine the analysis class **FloorPlan** (Figure 8.14) and, actually, sort of simplify it.

Vinod: The analysis class showed only things in the problem domain well, actually on the computer screen, that were visible to the end-user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan**, and obviously, there can be many cameras in the floor plan.

Vinod: Phew, let's see a picture of this new **FloorPlan** design class.

(Ed shows Vinod the drawing shown in Figure 9.3.)

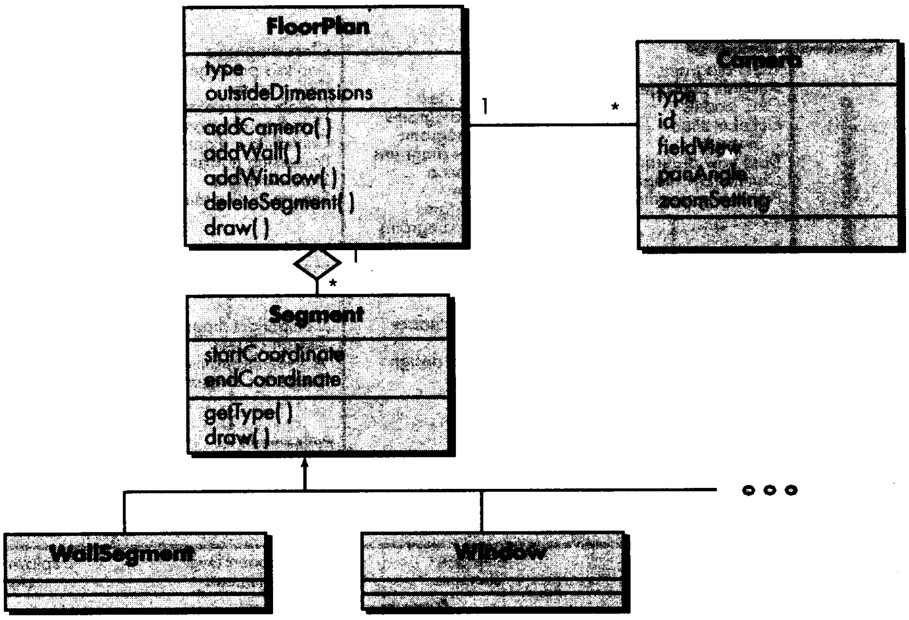
Vinod: Okay, I see what you're trying to do. This allows you to modify the floor plan easily because new items can be added or deleted to the list—the aggregation—without any problems.

Ed (nodding): Yeah, I think it'll work.

Vinod: So do I.

FIGURE 9.3

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



9.4 The Design Model

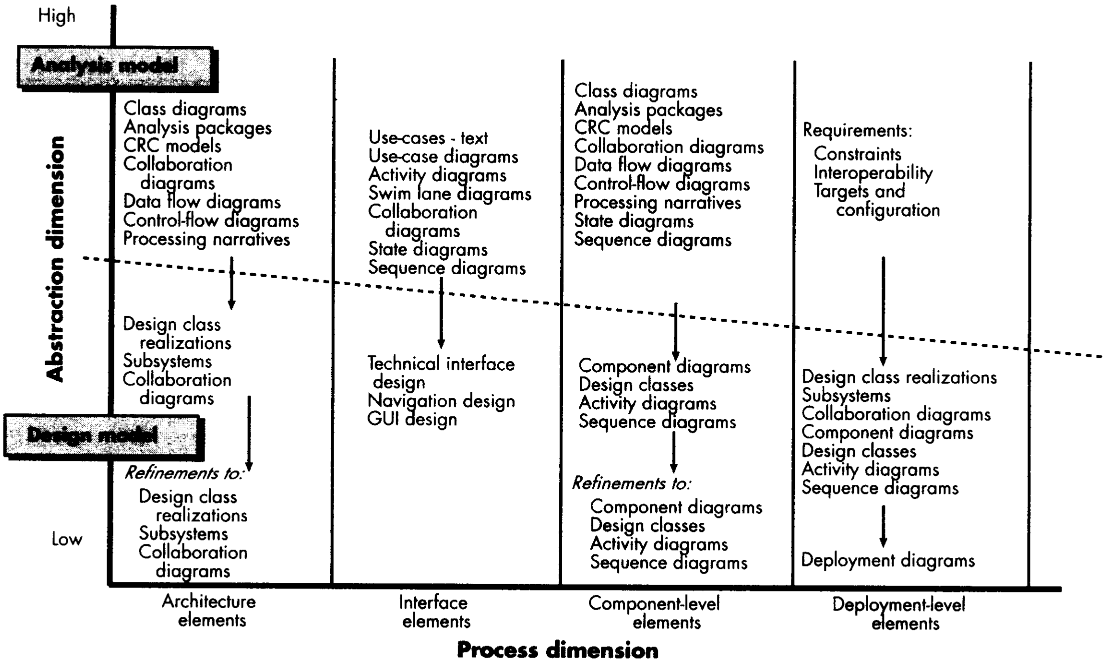
The *design model* can be viewed in two different dimensions as illustrated in Figure 9.4. The *process* dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction* dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to the figure, 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.

KEY POINT
The design model has four major elements: data, architecture, components, and interface.

The elements of the design model use many of the same UML diagrams 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.

> **"Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all."**
>
> **Douglas Martin**

FIGURE 9.4 **Dimensions of the design model**



It is important to mention however, that model elements noted 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.

9.4.1 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.

POINT

At the architectural (application) level, data design focuses on files or data bases; at the component level, data design considers the data structures that are required to implement local data objects.

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 10.

9.4.2 Architectural Design Elements

POINT

The architectural model is derived from the application domain, the analysis model, and available styles and patterns.

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.

> "You can use an eraser on the drafting table or a sledge hammer on the construction site."
>
> Frank Lloyd Wright

The architectural model [SHA96] is derived from three sources: (1) information about the application domain for the software to be built; (2) specific analysis 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 patterns (Section 9.5) and styles (Chapter 10).

9.4.3 Interface Design Elements

The *interface design* for software is the equivalent 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 utilities 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 door bell 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 tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

> **"The public is more familiar with bad design than good design. It is, in effect, conditioned to prefer bad design, because that is what it lives with. The new becomes threatening, the old reassuring."**
>
> **Paul Rand**

KEY POINT

There are three parts to the interface design element: the user interface; interfaces to systems external to the application, and interfaces to components within the application.

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 is a major software engineering action and is considered in detail in Chapter 12. The design of a UI 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.

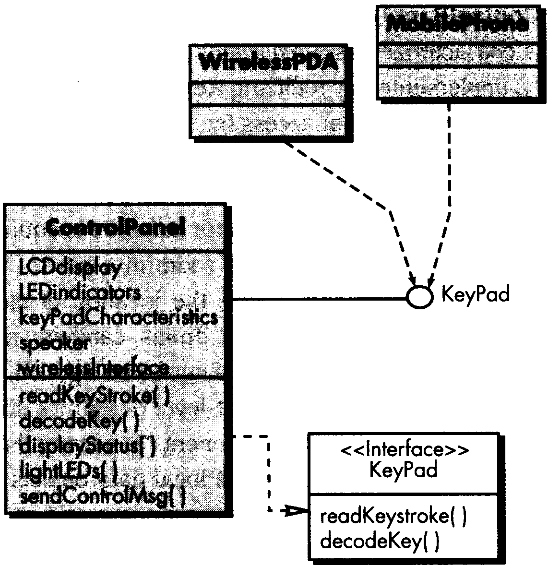
The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 7) and verified once the interface design commences.[6] The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

The design of internal interfaces is closely aligned with component-level design (Chapter 11). 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 accom-

[6] It is not uncommon for interface characteristics to change with time. Therefore, a designer should ensure that the specification for the interface is kept up-to-date.

FIGURE 9.5

UML interface representation for Control-Panel



modate the requisite information transfer and the specific functional requirements of the operation that has been requested.

WebRef
Extremely valuable information on UI design can be found at www.useit.com.

In some cases, an interface is modeled in much the same way as a class. UML defines an *interface* in the following manner [OMG01]: "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 those 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 a mobile phone.

The **ControlPanel** class (Figure 9.5) provides the behavior associated with a keypad, and, therefore, it must implement 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.

> "A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools."
>
> Douglas Adams

The dashed line with an open triangle at its end (Figure 9.5) 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.

9.4.4 Component-Level Design Elements

The component-level design for software is 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 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).

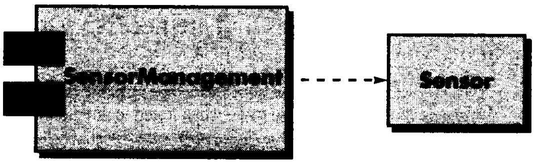
> "The details are not the details. They make the design."
>
> Charles Eames

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 9.6. 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 is presented in Chapter 11.

The design details of a component can be modeled at many different levels of abstraction. An 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 11) or some diagrammatic form (e.g., an activity diagram or flowchart).

FIGURE 9.6

UML component diagram for SensorManagement



9.4.5 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).

During design, a UML deployment diagram is developed and then refined as shown in Figure 9.7. 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.

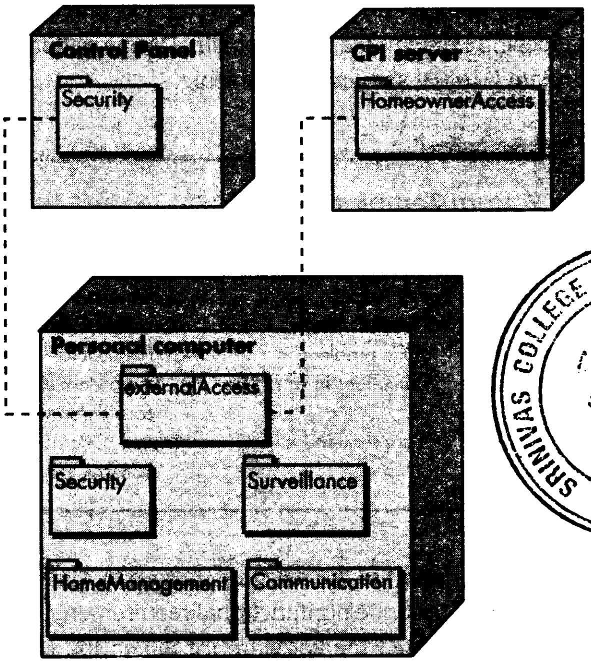
KEY POINT

Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used, and elements of the configuration are explicitly described.

The diagram shown in Figure 9.7 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 "Wintel" PC or a Macintosh, a Sun workstation or a Linux-box. These

FIGURE 9.7

UML deployment diagram for *SafeHome*



details are provided when the deployment diagram is revisited in *instance form* during latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

> **"Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen."**
>
> **Leonardo DaVinci**

9.5 Pattern-Based Software Design

WebRef
If you need to find a design (or other) patterns, visit **www.patterndepot.com/pages/.**

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. Throughout the design process, a software engineer should look for every opportunity to reuse existing design patterns (when they meet the needs of the design) rather than creating new ones.

9.5.1 Describing a Design Pattern

Mature engineering disciplines make use of thousands of design patterns. For example, a mechanical engineer uses a two-step, keyed shaft as a design pattern. Inherent in the pattern are attributes (the diameters of the shaft, the dimensions of the keyway, etc.) and operations (e.g., shaft rotation, shaft connection). An electrical engineer uses an integrated circuit (an extremely complex design pattern) to solve a specific element of a new problem. Design patterns may be described using the template [MAI03] shown in the sidebar.

INFO

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name.

Intent—describes the pattern and what it does.

Also-known-as—lists any synonyms for the pattern.

Motivation—provides an example of the problem.

Applicability—notes specific design situations in which the pattern is applicable.

Structure—describes the classes that are required to implement the pattern.

Participants—describes the responsibilities of the classes that are required to implement the pattern.

Collaborations—describes how the participants collaborate to carry out their responsibilities.

Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented.

Related patterns—cross-references related design patterns.

A description of the design pattern may also consider a set of design forces. *Design forces* describe nonfunctional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied. In addition forces define the constraints that may restrict the manner in which the design is to be implemented. In essence, design forces describe the environment and conditions that must exist to make the design pattern applicable. The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems [GAM95]. These attributes represent characteristics of the design that can be searched (e.g., via a database) so that an appropriate pattern can be found. Finally, guidance associated with the use of a design pattern provides an indication of the ramifications of design decisions.

KEY POINT

Design forces are those characteristics of the problem and attributes of the solution that constrain the way in which the design can be developed.

> **"Patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment."**
>
> **Martin Fowler**

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

9.5.2 Using Patterns in Design

Design patterns can be used throughout software design. Once the analysis model (Chapter 8) has been developed, the designer can examine a detailed representation of the problem to be solved and the constraints that are imposed by the problem. The problem description is examined at various levels of abstraction to determine if it is amenable to one or more of the following types of design patterns:

What types of design patterns are available for the software engineer?

Architectural patterns. These patterns define the overall structure of the software, indicate the relationships among subsystems and software components, and define the rules for specifying relationships among the elements (classes, packages, components, subsystems) of the architecture.

Design patterns. These patterns address a specific element of the design such as an aggregation of components to solve some design problem, relationships among components, or the mechanisms for effecting component-to-component communication.

Idioms. Sometimes called *coding patterns,* these language-specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components.

Each of these pattern types differs in the level of abstraction with which it is represented and the degree to which it provides direct guidance for the construction activity (in this case, coding) of the software process.

9.5.3 Frameworks

In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework,* for design work. That is, the designer may select a "*reusable mini-architecture* 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" [APP98].

POINT
A framework is a code skeleton that can be fleshed out with specific classes or functionality that have been designed to address the problem at hand.

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 a designer to integrate problem specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.

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.

9.6 SUMMARY

Design engineering commences as the first iteration of requirements engineering comes to a conclusion. The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it. Design engineers must sift through many design alternatives and converge on a solution that best suits the needs of project stakeholders.

The design process moves from a "big picture" view of software to a more narrow view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified; and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Design concepts have evolved over the first half-century of software engineering work. They describe attributes of computer software that should be present regardless of the software engineering process that is chosen, the design methods that are applied, or the programming languages that are used.

The design model encompasses four different elements. As each of these elements is developed, a more complete view of the design evolves. The architectural element uses information derived from the application domain, the analysis model, and available catalogs for patterns and styles to derive a complete structural representation of the software, its subsystems and components. Interface design elements model external and internal interfaces and the user interface. Component-level elements define each of the modules (components) that populate the architecture. Finally, deployment-level design elements allocate the architecture, its components, and the interfaces to the physical configuration that will house the software.

Pattern-based design is a technique that reuses design elements that have proven successful in the past. Each architectural pattern, design pattern, or idiom is cataloged, thoroughly documented, and carefully considered as it is assessed for inclusion in a specific application. Frameworks, an extension of patterns, provide an architectural skeleton for the design of complete subsystems within a specific application domain.

References

[AMB01] Ambler, S., *The Object Primer,* Cambridge Univ. Press, 2nd ed., 2001.

[APP98] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," downloadable at http://www.enteract.com/~bradapp/docs/patterns-intro.html.

[ARL02] Arlow, J., and I. Neustadt, *UML and the Unified Process,* Addison-Wesley, 2002.

[BEL81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L.J. Peters, author), Yourdon Press, 1981.

[FOW00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000.

[GAM95] Gamma, E., et al., *Design Patterns,* Addison-Wesley, 1995.

[GAR95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.

[GRA87] Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program,* Prentice-Hall, 1987.

[JAC75] Jackson, M. A., *Principles of Program Design,* Academic Press, 1975.

[LIE03] Lieberherr, K., "Demeter: Aspect-Oriented Programming," May 2003, available at: http://www.ccs.neu.edu/home/lieber/LoD.html.

[MAI03] Maioriello, J., "What Are Design Patterns and Do I Need Them?," developer.com, 2003, available at http://www.developer.com/design/article.php/ 1474561.

[MCG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes,* vol. 16, no. 4, October 1991, pp. 53–54.

[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand,1978.

[OMG01] Object Management Group, *OMG Unified Modeling Language Specification,* version 1.4, September 2001.

[PAR72] Parnas, D. L., "On Criteria to be used in Decomposing Systems into Modules," *CACM,* vol. 14, no. 1, April 1972, pp. 221–227.

[ROS75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer,* vol. 8, no. 5, May 1975.

[SCH02] Schmuller, J., *Teach Yourself UML,* SAMS Publishing, 2002.

[SHA96] Shaw, M., and D. Garlan, *Software Architecture,* Prentice-Hall, 1996.

[STA02] "Metaphor," *The Stanford HCI Learning Space,* 2002, http://hci.stanford.edu/ hcils/ concepts/metaphor.html.

[STE74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, no. 2, 1974, pp. 115–139.

[WIR71] Wirth, N., "Program Development by Stepwise Refinement," *CACM,* vol. 14, no. 4, 1971, pp. 221–227.

Problems and Points to Ponder

9.1. If a software design is not a program (and it isn't), then what is it?

9.2. Do you design software when you "write" a program? What makes software design different from coding?

9.3. Describe software architecture in your own words.

9.4. Visit a design patterns repository (on the Web) and spend a few minutes browsing through the patterns. Pick one and present it to your class.

9.5. Provide examples of three data abstractions and the procedural abstractions that can be used to manipulate them.

9.6. Apply a "stepwise refinement approach" to develop three different levels of procedural abstraction for one or more of the following programs: (a) Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check; (b) Iteratively solve for the roots of a transcendental equation; (c) Develop a simple task scheduling algorithm for an operating system.

9.7. When should a modular design be implemented as monolithic software? How can this be accomplished? Is performance the only justification for implementation of monolithic software?

9.8. Suggest a design pattern that you encounter in a category of everyday things (e.g., consumer electronics, automobiles, appliances). Fully document the pattern using the template provided in Section 9.5.

9.9. Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.

9.10. Is there a case when complex problems require less effort to solve? How might such a case affect the argument for modularity?

9.11. How are the concepts of coupling and software portability related? Provide examples to support your discussion.

9.12. Examine the task set presented for design. Where is quality assessed within the task set? How is this accomplished?

9.13. Do a bit of research on Extreme Programming and write a brief paper on the use of refactoring for that agile software development process.

9.14. How do we assess the quality of a software design?

Further Readings and Information Sources

Donald Norman has written two books (*The Design of Everyday Things,* Doubleday, 1990, and *The Psychology of Everyday Things,* HarperCollins, 1988) that have become classics in the design literature and "must" reading for anyone who designs anything that humans use. Adams (*Conceptual Blockbusting,* third edition, Addison-Wesley, 1986) has written a book that is essential reading for designers who want to broaden their way of thinking. Finally, a classic text by Polya (*How to Solve It,* Princeton University Press, second edition, 1988) provides a generic problem-solving process that can help software designers when they are faced with complex problems.

Following in the same tradition, Winograd et al. (*Bringing Design to Software,* Addison-Wesley, 1996) discusses software designs that work, those that don't, and why. A fascinating book edited by Wixon and Ramsey (*Field Methods Casebook for Software Design,* Wiley, 1996) suggests field research methods (much like those used by anthropologists) to understand how end-users do the work they do and then provides guidance for designing software that meets their needs. Beyer and Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs,* Academic Press, 1997) offer another view of software design that integrates the customer/user into every aspect of the software design process.

McConnell (*Code Complete,* Microsoft Press, 1993) presents an excellent discussion of the practical aspects of designing high-quality computer software. Robertson (*Simple Program Design,* third edition, Boyd and Fraser Publishing, 1999) offers an introductory discussion of software design that is useful for those beginning their study of the subject. Fowler and his colleagues (*Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999) discuss techniques for the incremental optimization of software designs.

Over the past decade, many books on pattern-based design have been written for software engineers. Gamma and his colleagues [GAM95] have written the seminal book on the subject. Other books by Douglass (*Real-Time Design Patterns,* Addison-Wesley, 2002), Metsker (*Design Patterns Java Workbook,* Addison-Wesley, 2002), Juric et al. (*J2EE Design Patterns Applied,* Wrox Press, 2002), Marinescu and Roman (*EJB Design Patterns,* Wiley, 2002), and Shalloway and Trott (*Design Patterns Explained,* Addison-Wesley, 2001) discuss design patterns in specific application and language environments. In addition, classic books by the architect Christopher Alexander (*Notes on the Synthesis of Form,* Harvard University Press, 1964 and *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, 1977) are must reading for a software designer who intends to fully understand design patterns.

A wide variety of information sources on design engineering are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software design and design engineering can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER 10 CREATING AN ARCHITECTURAL DESIGN

KEY CONCEPTS

ADL

archetypes

architectural

- **assessment**
- **complexity**
- **context diagram**
- **definition**
- **design**
- **mapping**
- **patterns**

ATAM

Components

data design

factoring

styles

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. This description is extended by Freeman [FRE80]:

> [D]esign is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels. . . .

As we have noted in Chapter 9, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The informational, functional, and behavioral domains serve as a guide for the creation of the software design.

Methods required to create "coherent, well planned representations" of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

QUICK LOOK

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it?. Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style for the requirements derived during system engineering and software requirements analysis.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

10.1 Software Architecture

In their landmark book on the subject, Shaw and Garlan [SHA96] discuss software architecture in the following manner:

> Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

> "The architecture of a system is a comprehensive framework that describes its form and structure—its components and how they fit together."
>
> Jerrold Grochow

10.1.1 What Is Architecture?

When we discuss the architecture of a building, many different attributes come to mind. At the most simplistic level, we consider the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art.

KEY POINT

Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

But what about *software architecture?* Bass, Clements, and Kazman [BAS03] define this elusive term in the following way:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

> **"Marry your architecture in haste, repent at your leisure."**
>
> **Barry Boehm**

WebRef

Useful pointers to many software architecture sites can be obtained at **www2.umassd.edu/SECenter/SAResources.html.**

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers.

In this book the design of software architecture considers two levels of the design pyramid (Figure 9.1)—*data design* and *architectural design.* In the context of the preceding discussion, data design enables us to represent the data component of the architecture in conventional systems and class definitions (encapsulating attributes and operations) in object-oriented systems. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

10.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [BAS03] identify three key reasons that software architecture is important:

KEY POINT

The architectural model provides a Gestalt view of the system, allowing the software engineer to examine it as a whole.

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [BAS03].

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture styles and patterns (Section 10.3.1) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

10.2 Data Design

The *data design* action translates data objects defined as part of the analysis model (Chapter 8) into data structures at the software component level and, when necessary, a database architecture at the application level. In some situations, a database must be designed and built specifically for a new system. In others, however, one or more existing databases are used.

10.2.1 Data Design at the Architectural Level

Today, businesses large and small are awash in data. It is not unusual for even a moderately sized business to have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The challenge is to extract useful information from this data environment, particularly when the information desired is cross-functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

> "Data quality is the difference between a data warehouse and a data garbage dump."
>
> Jarrett Rosenberg

WebRef
Information of data warehouse technologies can be obtained at **www.datawarehouse.com.**

To solve this challenge, the business IT community has developed *data mining* techniques, also called *knowledge discovery in databases* (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment. An alternative solution, called a *data warehouse,* adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business [MAT96]. In a sense, a data warehouse is a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business.

A detailed discussion of the design of data structures, databases, and the data warehouse is best left to books dedicated to these subjects (e.g., [DAT00], [PRE98], [KIM98]). The interested reader should see the *Further Readings and Information Sources* section of this chapter for additional references.

SOFTWARE TOOLS

Data Mining/Warehousing

Objective: Data mining tools assist in the identification of significant relationships among attributes that describe a specific data object or set of data objects. Tools for data warehousing assist in the design of data models for a data warehouse.

Mechanics: Tool mechanics vary. In general, mining tools accept large data sets as input and allow the user to query the data in an effort to better understand relationships among various data items. Warehousing tools that are used for design provide entity relationship or other modeling capabilities.

Representative Tools[1]

Data Mining:

Business Objects, developed by Business Objects, SA (www.business objects.com), is a data design tool set hat supports "data integration, query, reporting, analysis, and analytics."

SPSS, developed by SPSS, Inc. (www.spss.com), provides a wide array of statistical functions to allow the analysis of large data sets.

Data Warehousing:

Industry Warehouse Studio, developed by Sybase (www.sybase.com), provides a packaged data warehouse infrastructure that "jumpstarts" data warehouse design.

IFW Business Intelligence Suite, developed by Modelware (www.modelwarepl.com), is a set of models, software tools, and database designs that "provide a fast path to data warehouse and datamart design and implementation."

A comprehensive list of data mining/warehousing tools and resources can be found at the Data Warehousing Information Center (www.dwinfocenter.org).

10.2.2 Data Design at the Component Level

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman [WAS80] has proposed a set of principles that may be used to specify and design such data structures. In actuality, the design of data begins during the creation of the analysis model. Recalling that requirements analysis and design often overlap, we consider the following set of principles (adapted from [WAS80]) for data specification:

? What principles are applicable to data design?

1. *The systematic analysis principles applied to function and behavior should also be applied to data.* Representations of data flow and content should also be developed and reviewed, data objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated.
2. *All data structures and the operations to be performed on each should be identified.* The design of an efficient data structure must take the operations to be performed on the data structure into account. The attributes and operations encapsulated within a class satisfy this principle.
3. *A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it.* Class diagrams

[1] Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

(Chapter 8) define the data items (attributes) contained within a class and the processing (operations) that are applied to these data items.

4. *Low-level data design decisions should be deferred until late in the design process.* A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component-level design.
5. *The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure.* The concept of information hiding and the related concept of coupling (Chapter 9) provide important insight into the quality of a software design.
6. *A library of useful data structures and the operations that may be applied to them should be developed.* A class library achieves this.
7. *A software design and programming language should support the specification and realization of abstract data types.* The implementation of a sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language chosen for implementation.

These principles form a basis for a component-level data design approach that can be integrated into both the analysis and design activities.

10.3 Architectural Styles and Patterns

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 importantly, 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.

> **"There is at the back of every artist's mind, a pattern or type of architecture."**
>
> **G. K. Chesterton**

? What is an architectural style?

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].

WebRef
Attribute-based architectural styles (ABAS) can be used as building blocks for software architectures. Information can be obtained at **www.sei.cmu.edu/ata/abas.html.**

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 31), 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 tend to address specific behavioral issues within the context of the architectural, e.g., how a real-time application handles synchronization or interrupts. Patterns can be used in conjunction with an architectural style to establish the shape the overall structure of a system. In the section that follows, we consider commonly used architectural styles and patterns for software.

10.3.1 A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 50 years, the vast majority can be categorized (see [SHA96], [BUS96], [BAS03]) into one of a relatively small number of architectural styles:

Data-centered architecture. 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 10.1 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.

A data-centered architecture promotes *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 architecture. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into

FIGURE 10.1

Data-centered architecture

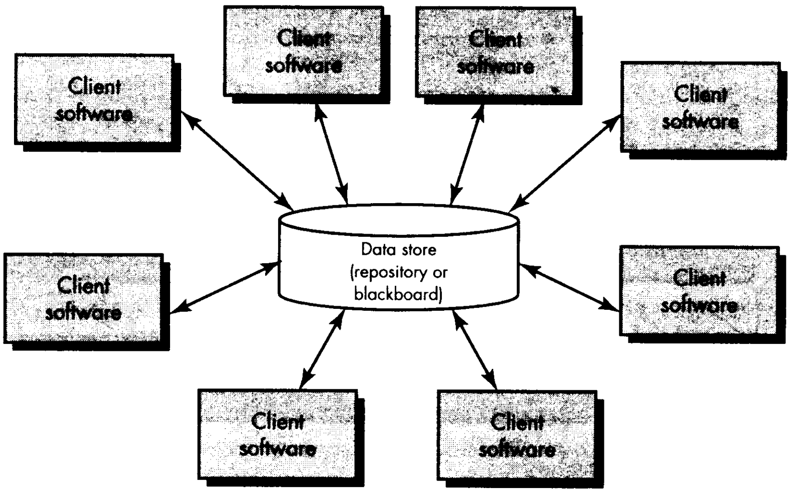
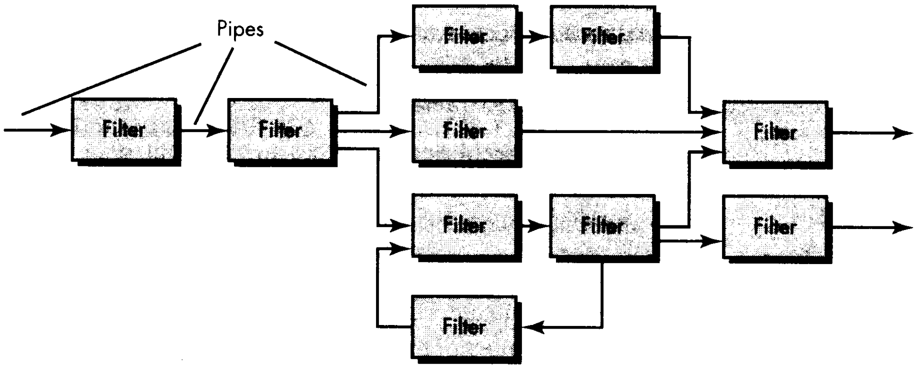


FIGURE 10.2

Data-flow architecture



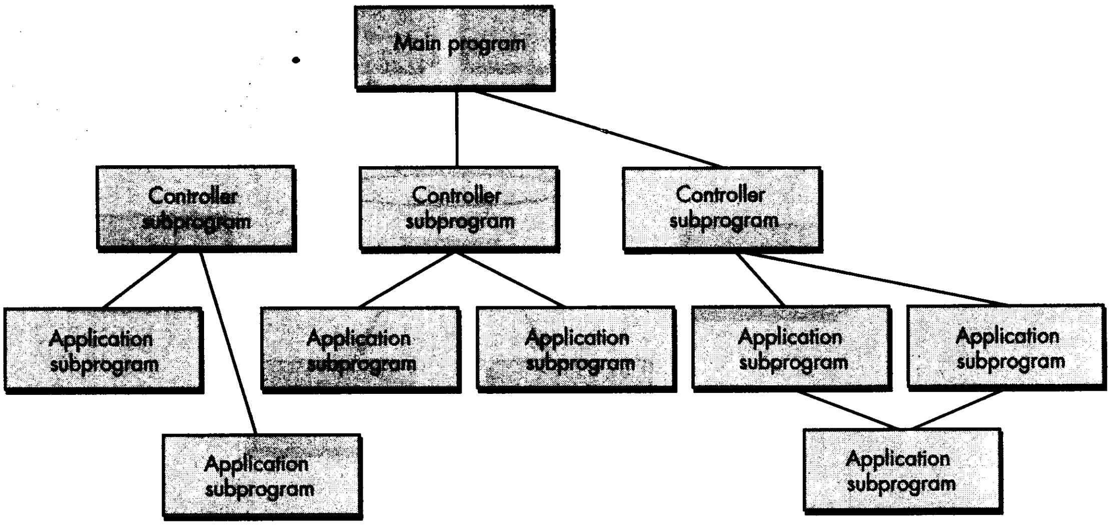
Pipes and filters

output data. A pipe and filter structure (Figure 10.2) 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.

"The use of patterns and styles of design is pervasive in engineering disciplines."

Mary Shaw and David Garlan

FIGURE 10.3 Main program/subprogram 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.

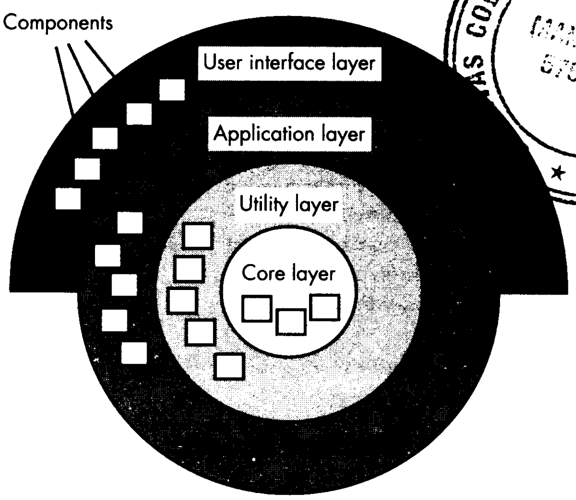
Call and return architecture. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. Two substyles [BAS03] exist within this category:

- *Main program/subprogram architecture.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 10.3 illustrates an architecture of this type.
- *Remote procedure call architecture.* The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

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

Layered architecture. The basic structure of a layered architecture is illustrated in Figure 10.4. 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.

FIGURE 10.4
Layered architecture



These architectural styles are only a small subset of those available to the software designer.[2] Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style or combination of styles that best fits those characteristics and constraints can be chosen. In many cases, more than one style might be appropriate, and alternatives might 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.

SAFEHOME

Choosing an Architectural Style

The scene: Jamie's cubicle, as design modeling continues.

The players: Jamie and Ed—members of the *SafeHome* software engineering team.

The conversation:

Ed (frowning): We've been modeling the security function using UML . . . you know classes, relationships, that sort of stuff. So I guess the object-oriented architecture[3] is the right way to go.

Jamie: But. . . ?

Ed: But . . . I have trouble visualizing what an object-oriented architecture is. I get the call and return architecture, sort of a conventional process hierarchy, but OO . . I don't know. It seems sort of amorphous.

[2] See [BOS00], [HOF00], [BAS03], [SHA97], [BUS96], and [SHA96] for a detailed discussion of architectural styles and patterns.

[3] It can be argued that the *SafeHome* architecture should be considered at a higher level than the architecture noted. *SafeHome* has a variety of subsystems—home monitoring functionality, the company's monitoring site, and the subsystem running in the owner's PC. Within subsystems, concurrent processes (e.g. those monitoring sensors) and event handling are prevalent. Some architectural decisions at this level are made during system or product engineering (Chapter 6), but architectural design within software engineering may very well have to consider these issues.

[illegible], huh?

[illegible] I can't visualize a real [illegible] floating in space.

[illegible] There are class hierarchies [illegible] hierarchy (aggregation) we did for the [illegible] (Figure 9.3). An OO architecture is a [illegible] and the interconnections—[illegible] between the classes. We can [illegible] describing the attributes and operations, [illegible] that goes on, and the structure of the [illegible]

Ed: I'm going to spend an hour [illegible] return architecture, then I'll go [illegible] architecture.

Jamie: Doug'll have no problem with [illegible] we should consider architectural [illegible] there's absolutely no reason why [illegible] couldn't be used in combination with [illegible]

Ed: Good. I'm on it.

10.3.2 Architectural Patterns

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed on the design.[4]

Architectural patterns are a bit different.[5] For example, every house (and every architectural style for houses) employs a *kitchen* pattern. The *kitchen* pattern defines the need for placement of basic kitchen appliances, the need for a sink, the need for cabinets, and possibly, rules for placement of these things relative to workflow in the room. In addition, the pattern might specify the need for counter tops, lighting, wall switches, a central island, flooring, and so on. Obviously, there is more than a single design for a kitchen, but every design can be conceived within the context of the "solution" suggested by the *kitchen* pattern.

KEY POINT
A software architecture may have a number of architectural patterns that address issues such as concurrency, persistence, and distribution.

As we have already noted, architectural patterns for software define a specific approach for handling some behavioral characteristic of the system. Bosch [BOS00] defines a number of architectural pattern domains. Representative examples are provided in the paragraphs that follow.

Concurrency. Many applications must handle multiple tasks in a manner that simulates parallelism (i.e., this occurs whenever multiple "parallel" tasks or components are managed by a single processor). There are a number of different ways in which an

[4] This implies that there will be a central foyer and hallway, that rooms will be placed to the left and right of the foyer, that the house will have two (or more) stories, that the bedrooms of the house will be upstairs, and so on. These "rules" are imposed once the decision is made to use the *center-hall colonial* style.

[5] It is important to note that there is not universal agreement on this terminology. Some people (e.g., [BUS96]) use the terms *styles* and *patterns* synonymously, while others make the subtle distinction suggested in this section.

application can handle concurrency, and each can be presented by a different architectural pattern. For example, one approach is to use an *operating system process management* pattern that provides built-in OS features that allow components to execute concurrently. The pattern also incorporates OS functionality that manages communication between processes, scheduling, and other capabilities required to achieve concurrency. Another approach might be to define a task scheduler at the application level. A *task scheduler* pattern contains a set of active objects that each contains a *tick()* operation [BOS00]. The scheduler periodically invokes *tick()* for each object, which then performs the functions it must perform before returning control back to the scheduler, which then invokes the *tick()* operation for the next concurrent object.

Persistence. Data persists if it survives past the execution of the process that created it. Persistent data are stored in a database or file and may be read or modified by other processes at a later time. In object-oriented environments, the idea of a persistent object extends the persistence concept a bit further. The values of all of the object's attributes, the general state of the object, and other supplementary information are stored for future retrieval and use. In general, two architectural patterns are used to achieve persistence—a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture or an *application level persistence* pattern that builds persistence features into the application architecture (e.g., word processing software that manages its own document structure).

Distribution. The distribution problem addresses the manner in which systems or components within systems communicate with one another in a distributed environment. There are two elements to this problem: (1) the way in which entities connect to one another, and (2) the nature of the communication that occurs. The most common architectural pattern established to address the distribution problem is the *broker* pattern. A broker acts as a "middle-man" between the client component and a server component. The client sends a message to the broker (containing all appropriate information for the communication to be effected), and the broker completes the connection. CORBA (Chapter 30) is an example of a broker architecture.

Before any one of the architectural patterns noted in the preceding paragraphs can be chosen, it must be assessed for its appropriateness for the application and the overall architectural style, as well as its maintainability, reliability, security, and performance.

10.3.3 Organization and Refinement

Because the design process often leaves a software engineer 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. The following questions [BAS03] provide insight into the architectural style that has been derived.

How do I assess an architectural style that has been derived?

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.

10.4 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 analysis model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, the designer specifies the structure of the system by defining and refining software components that implement the architecture. This process continues iteratively until a complete architectural structure has been derived.

> "A doctor can bury his mistakes, but an architect can only advise his client to plant vines."
>
> Frank Lloyd Wright

10.4.1 Representing the System in Context

KEY POINT

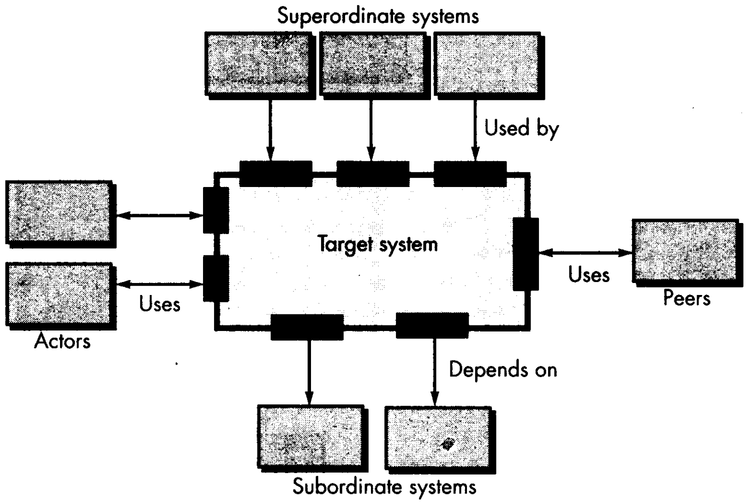
Architectural context represents how the software interacts with entities external to its boundaries.

In Chapter 6, we noted that a system engineer must model context. A system context diagram (Figure 6.4) accomplishes this requirement by representing the flow of information into and out of the system, the user interface, and relevant support processing. 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 10.5.

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:

FIGURE 10.5

Architectural context diagram (adapted from [BOS00])



How do systems interoperate with one another?

- *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*—those entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

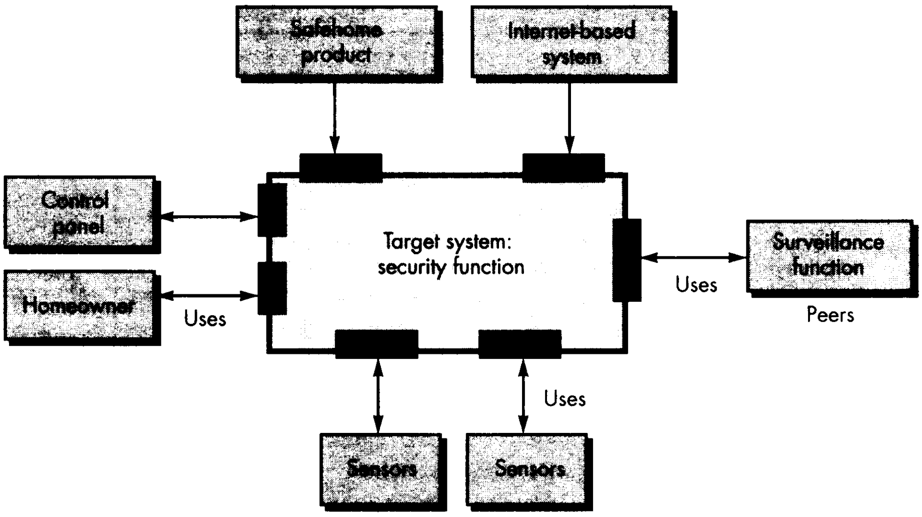
Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD we again 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 10.6. 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 10.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

FIGURE 10.6

Architectural context diagram for the *SafeHome* security function



10.4.2 Defining Archetypes

KEY POINT

Archetypes are the abstract building blocks of an architectural design.

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 in 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 analysis model. Continuing our discussion of the *SafeHome* home security function, we 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.

Each of these archetypes is depicted using UML notation as shown in Figure 10.7.