CHAPTER

9 DESIGN CONCEPTS

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. *Design principles* establish an overriding philosophy that guides the design work you must perform. *Design concepts* must be understood before the mechanics of design practice are applied, and *design practice* leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

KEY CONCEPTS

QUICK LOOK

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where requirements and technical considerations come together in the formulation of a product or system. Design creates a representation or model of the software and provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks while continuing communication with the stakeholders.

Why is it important? During the design phase, you model the system or product that needs to be built. The design model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

What are the steps? Design makes use of several different representations of the software. First, the architecture of the system or product must be modeled. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are represented. Finally, the software components that are used to construct the system are designed.

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

How do I ensure that I've done it right? The design model is assessed by the software team (including relevant stakeholders) in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

Design is pivotal to successful software engineering. Some developers are tempted to begin programming once the use cases have been created, without regard to how the software components needed to implement the use cases relate to one another. It is possible to do analysis, design, and implementation iteratively by creating several software increments. It is a bad idea to ignore the design considerations needed to create an appropriate architecture for the evolving software product. *Technical debt* is a concept in software development that refers to costs associated with rework caused by choosing a "quick and dirty" solution right now instead of using a better approach that would take more time. It is impossible to avoid creating technical debt when building a software product incrementally. However, a good development team must try to pay down this technical debt by refactoring (Section 9.3.9) the software on a regular basis. Just like taking out a loan, you can wait until the loan is due and pay a lot of interest or you can pay the loan off a little at a time and pay less interest overall.

One strategy to keep technical debt in check without delaying coding is to make use of the design practices of diversification and convergence. *Diversification* is the practice of identifying possible design alternatives suggested by the elements of the requirements model. *Convergence* is the process of evaluating and rejecting design alternatives that do not meet the constraints imposed by the nonfunctional requirements defined for the software solution. Diversification and convergence combine (1) intuition and judgment based on experience in building similar entities, (2) a set of principles and/or heuristics that guide the way in which the model evolves, (3) a set of criteria that enables quality to be judged, and (4) a process of iteration that ultimately leads to a final design representation. Once a viable design alternative is identified this way, the developers are in a good position to create a software increment that is not likely to be a throwaway prototype.

Software design changes continually as new methods, better analysis, and broader understanding evolve.[1] 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 through 14 we'll present a variety of software design methods as they are applied to architectural, interface, and component-level design as well as pattern-based, mobile, and user experience design approaches.

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).

1 Those readers with further interest in the philosophy of software design might have interest in Philippe Kruchen's intriguing discussion of "postmodern" design [Kru05].

Each of the elements of the requirements model (Chapter 8) provide 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 requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

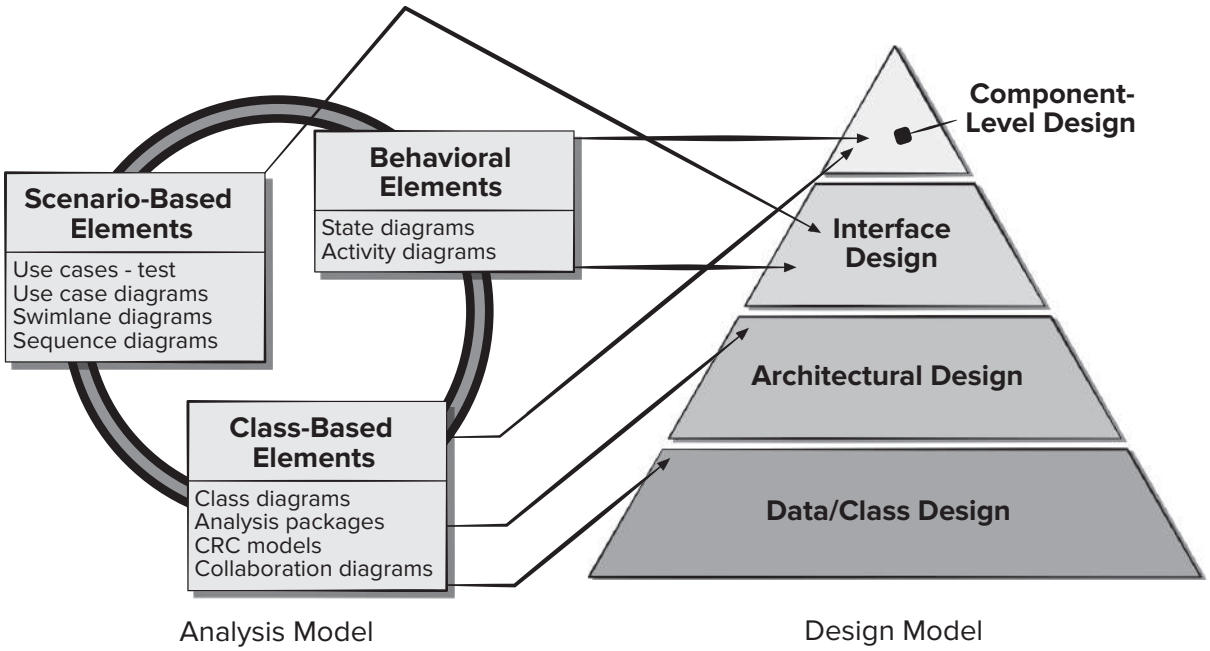
The data/class design transforms class models (Chapter 8) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC model and the detailed data content depicted by class attributes and other notation provide the 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 style, and patterns (Chapter 14) that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha15]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

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

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

FIGURE 9.1 **Translating the requirements model into the design model**



During design you make decisions that will ultimately affect the success of software construction and, just 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. It provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholders' requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process. Late in the project is when time is short, and many budgeted dollars have already been spent.

SAFEHOME

Design Versus Coding

The scene: Jamie's cubicle, as the team prepares to translate requirements into design.

The players: Jamie, Vinod, and Ed, all members of the *SafeHome* software engineering team.

The conversation:

Jamie: You know, Doug [the team manager] is obsessed with design. I gotta be honest, what I really love doing is coding. Give me C++ or Java, and I'm happy.

Ed: Nah . . . you like to design.

Jamie: You're not listening—coding is where it's at.

Vinod: I think what Ed means is that you don't really like coding; you like to design and express it in code. Code is the language you use to represent the design.

Jamie: And what's wrong with that?

Vinod: Level of abstraction.

Jamie: Huh?

Ed: A programming language is good for representing details like data structures and algorithms, but it's not so good for representing architecture or component-to-component collaboration . . . stuff like that.

Vinod: And a screwed-up architecture can ruin even the best code.

Jamie (thinking for a minute): So, you're saying that I can't represent architecture in code . . . that's not true.

Vinod: You can certainly imply architecture in code, but in most programming languages, it's difficult to get a quick, big-picture read on architecture by examining the code.

Ed: And that's what we want before we begin coding.

Jamie: Okay, maybe design and coding are different, but I still like coding better.

9.2 THE DESIGN PROCESS

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

refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connections may not be obvious at these lower levels of abstraction.

9.2.1 Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 16. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design should implement all explicit requirements contained in the requirements model, and it must accommodate all the implicit requirements desired by stakeholders.
- The design should 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 a goal of the design process. But how is each of these goals achieved?

Quality Guidelines. To evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 9.3, we discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that (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,[2] thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
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.
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.

2 For smaller systems, design can sometimes be developed linearly.

Chance alone will not achieve these design guidelines. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

INFO

Assessing Design Quality—The Technical Review

Design is important because it allows a software team to assess the quality[3] 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 technical reviews (TRs). TRs are discussed in detail in Chapter 16,[4] but it's worth providing a summary of the technique at this point. A technical review 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 runs the meeting; the *recorder* takes notes so that nothing is missed; and 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 TR typically lasts between 60 to 90 minutes. After the TR concludes, the review team determines whether further actions are required from the producer before the design work product can be approved as part of the final design model.

9.2.2 The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs [Den73] and methods for refining software structures in a top-down "structured" manner ([Wir71], [Dah72], [Mil72]). Newer design approaches (e.g., [Jac92], [Gam95]) proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture [Kru06] and the design patterns that can be used to implement software architectures and lower levels of design abstractions (e.g., [Hol06], [Sha05]). There is a growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04], which focus on techniques for achieving more effective modularity and architectural structure in the designs that are created.

In the past 10 years, Search-Based Software Engineering (SBSE) techniques have been applied to all phases of the software engineering life cycle, including design [Har12]. SBSE attempts to solve software engineering problems using automated search techniques augmented by operations research and machine learning algorithms to provide design recommendations to software developers. Many modern software systems must accommodate a high degree of variability, both in their deployment environments and

3 The quality factors discussed in Chapter 23 can assist the review team as it assesses quality.
4 You might consider looking ahead to Chapter 16 at this time. Technical reviews are a critical part of the design process and are an important mechanism for achieving design quality.

the number of usage scenarios they expected to satisfy. Design of *variability-intensive systems*[5] requires developers to anticipate future changes in the features to be modified in future versions of the product being designed today [Gal16]. A detailed discussion of the design of variability-intensive systems is beyond the scope of this book.

Several design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapter 8, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, each of these methods has common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

TASK SET

Generic Task Set for Design

Please note: These tasks are often performed iteratively and in parallel. They are rarely completed sequentially and in isolation from one another unless you are following the waterfall process model.

1. Examine the information 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 an 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 a behavioral model of the interface.
   - Define interface objects and 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.

5 Variability-intensive systems refers to systems that may be required to be self-modifying based on changes in the run-time environment or families of software products resulting from product line engineering practices for building specialized product variants out of existing software products.

9.3 DESIGN CONCEPTS

Several fundamental software design concepts have evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you define criteria that can be used to partition software into individual components, separate out data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design. These concepts help developers design the software actually needed, rather than simply focusing on creating any old working program.

9.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment (e.g., a user story). At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology to state a solution (e.g., use case). Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented (e.g., pseudocode).

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *use* for a camera in the *SafeHome* system. *Use* implies a long sequence of procedural steps (e.g., activate the *SafeHome* system on a mobile device, log on to the *SafeHome* system, select a camera to preview, locate the camera controls on mobile app user interface, etc.).[6]

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 **camera.** Like any data object, the data abstraction for **camera** would encompass a set of attributes that describe the camera (e.g., camera ID, location, field view, pan angle, zoom). It follows that the procedural abstraction *use* would make use of information contained in the attributes of the data abstraction **camera.**

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" [Sha15]. In its simplest form, architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are

6 It should be noted, however, that one set of operations can be replaced with another, if the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *use* would change dramatically if the camera were automatic and attached to a sensor that automatically triggered an alert on your mobile device.

used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garlan [Sha15] describe a set of properties that should be specified as part of an architectural design. *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics (e.g., nonfunctional system requirements)." *Families of related systems* "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."[7]

Given the specification of these properties, the architectural design can be represented using one or more of several 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 (patterns) 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.

Several different *architectural description languages* (ADLs) have been developed to represent these models [Sha15]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the ways in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The ways in which software architecture is characterized and its role in design are discussed in Chapter 10.

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" [App00]. Stated in another way, a design pattern describes a design structure that solves a well-defined 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.

---

7 These families of related software products sharing common features are called *software product lines*.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different, pattern. Design patterns are discussed in detail in Chapter 14.

9.3.4 Separation of Concerns

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

It 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 for software modularity.

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

9.3.5 Modularity

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

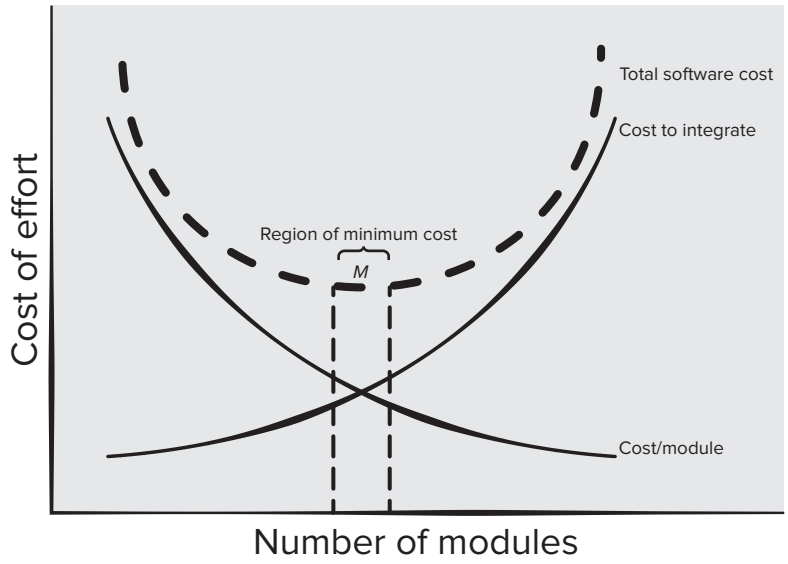
It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 9.2, the effort (cost) to develop an individual software module tends to decrease as the total number of modules increases.

Given the same set of requirements, the more modules used in your program means smaller individual sizes. However, as the number of modules grows, the effort (cost) associated with integrating modules with each other grows. These characteristics lead to a total cost or effort curve, shown in Figure 9.2. 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 qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. Using too few modules or too many modules should be avoided.

FIGURE 9.2
Modularity and software cost



But how do you know the vicinity of *M*? How modular should you make software? The answers to these questions require an understanding of other design concepts considered in Sections 9.3.6 through 9.3.9.

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

9.3.6 Information Hiding

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

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

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

9.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] each 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, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality. Evaluation of your CRC card model (Chapter 8) can help you spot problems with functional independence. User stories that contain many instances of words such as *and* or *except* are not likely to encourage you to design modules that are "single-minded" system functions.

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 9.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnections among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less likely to propagate errors found in one module to other system modules.

9.3.8 Stepwise Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. Successively refining levels of procedural detail is a good way to develop an application. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is a process of *elaboration.* You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the

statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

9.3.9 Refactoring

An important design activity suggested for many agile methods (Chapter 3), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a large component that exhibits low cohesion (i.e., it performs three functions that have only a limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each of which exhibits high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. Refactoring tools [Soa10] are sometimes used to analyze code changes automatically and to "generate a test suite suitable for detecting behavioral changes."

SAFEHOME

Design Concepts

The scene: Vinod's cubicle, as design modeling begins.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

The conversation:

(All four team members have just returned from a morning seminar entitled "Applying Basic Design Concepts," offered by a local computer science professor.)

Vinod: Did you get anything out of the seminar?

Ed: Knew most of the stuff, but it's not a bad idea to hear it again, I suppose.

Jamie: When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

Vinod: Because . . . bottom line . . . it's a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

Shakira: I wasn't an SE grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don't see why this stuff is so important.

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

Shakira (smiling): Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . 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, you know.

Jamie: The only concepts I hadn't heard of before were "design classes" and "refactoring."

Shakira: Refactoring is used in Extreme Programming, I think 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 like an optimization pass through the software, if you ask me.

Jamie: Let's get back to the *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 commit to think about them as we develop the design.

9.3.10 Design Classes

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

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented and to create a software infrastructure that supports the business solution.

As the software architecture forms, the level of abstraction is reduced as each analysis class (Chapter 8) is transformed into a design representation. That is, analysis classes represent data objects and associated services that are applied to them. Design classes present significantly more technical detail as a guide for implementation.

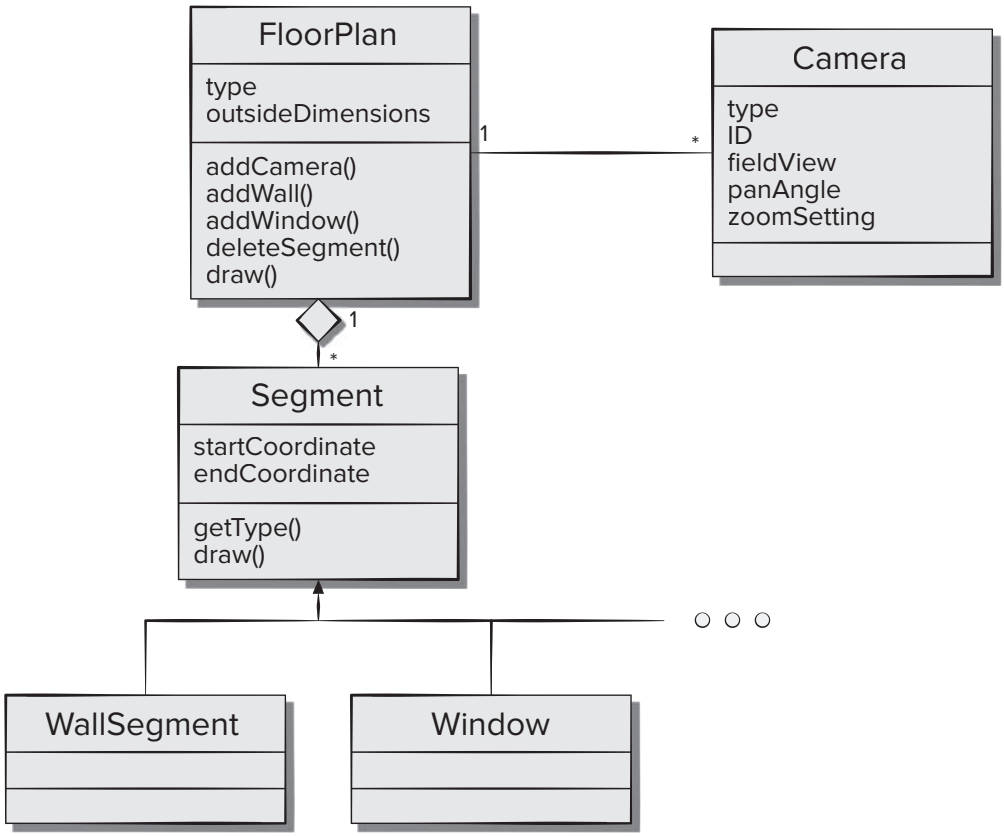
Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **FloorPlan** (Figure 9.3) defined for the *SafeHome* room layout software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a floor plan. 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 **Segment** (Figure 9.3) for use by the room layout

FIGURE 9.3

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



software might have attributes `startCoordinate` and `endCoordinate` to indicate the start and end points of the segment to be drawn. The method *setCoordinates()* provides the only means for establishing start and end points for the segment.

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 **FloorPlan** (Figure 9.3) might contain a set of methods for editing the house floor plan. As long as each method focuses solely on attributes associated with the floor plan, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.[8]

8 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 begins.

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 8.3.3 and Figure 8.4) 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. I 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 8.4) 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 to or deleted from the list—the aggregation—without any problems.

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

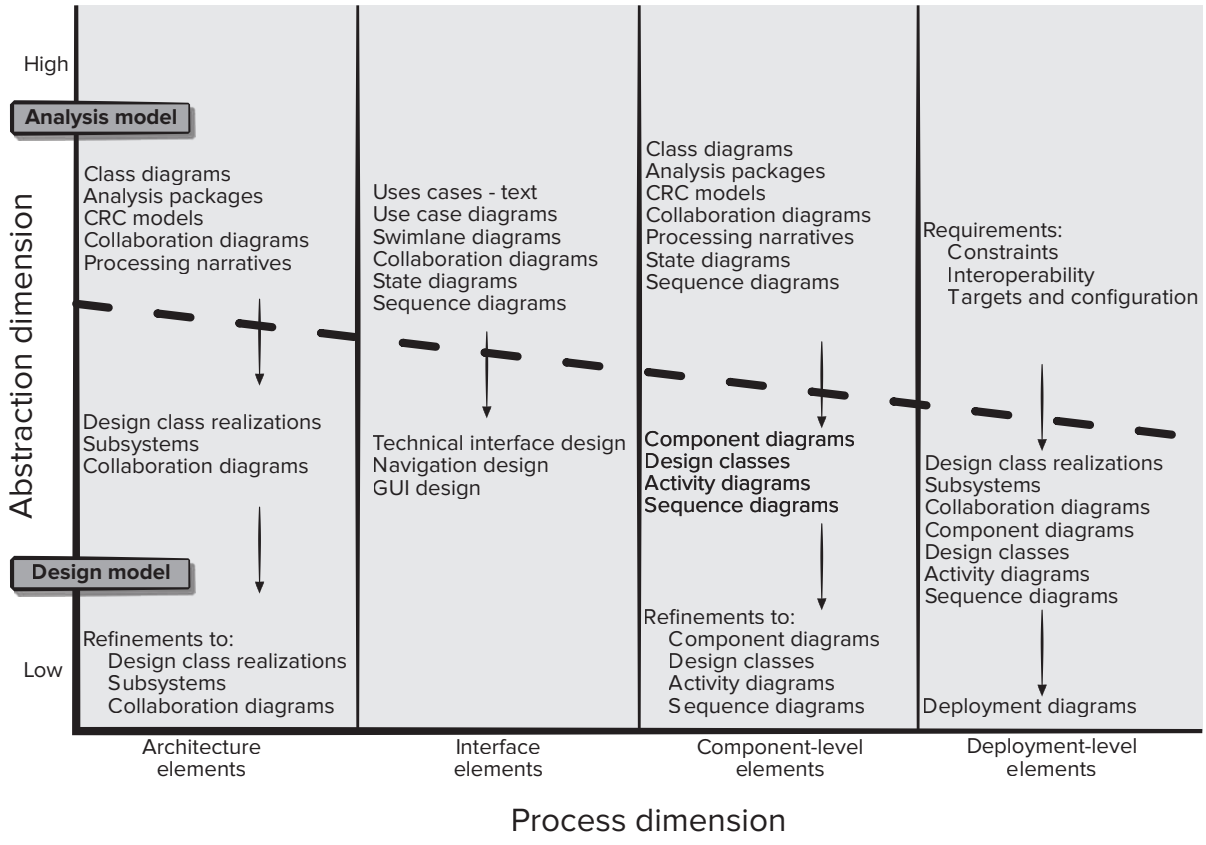
Vinod: So do I.

9.4 THE DESIGN MODEL

The software design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

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.

FIGURE 9.4 Dimensions of the design model



The elements of the design model use many of the same UML diagrams[9] that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

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

You can apply design patterns (Chapter 14) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

9 Appendix 1 provides a tutorial on basic UML concepts and notation.

9.4.1 Design Modeling Principles

There is no shortage of methods for deriving the various elements of a software design model. Some methods are data driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern driven, using information about the problem domain (the requirements model) to develop architectural styles and processing patterns. Still others are object oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied, regardless of the method that is used:

Principle 1. ***Design should be traceable to the requirements model.*** The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

Principle 2. ***Always consider the architecture of the system to be built.*** Software architecture (Chapter 10) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. ***Design of data is as important as design of processing functions.*** Data design is an essential element of architectural design. The ways in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4. ***Interfaces (both internal and external) must be designed with care.*** The ways in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. ***User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use.*** The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. ***Component-level design should be functionally independent.*** Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function.

Principle 7. ***Components should be loosely coupled to one another and to the external environment.*** Coupling is achieved in many ways—via a component interface, by messaging, and through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle 8. ***Design representations (models) should be easily understandable.*** The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. ***The design should be developed iteratively.*** With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

Principle 10. ***Creation of a design model does not preclude an agile approach.*** Some proponents of agile software development (Chapter 3) insist that the code is the only design documentation that is needed. Yet the purpose of a design model is to help others who must maintain and evolve the system. It is extremely difficult to understand either the higher-level purpose of a code fragment or its interactions with other modules in a modern multithreaded run-time environment.

Agile design documentation should be kept in sync with the design and development, so that at the end of the project the design is documented at a level that allows the code to be understood and maintained. The design model provides benefit because it is created at a level of abstraction that is stripped of unnecessary technical detail and is closely coupled to the application concepts and requirements. Complementary design information can incorporate a design rationale, including the descriptions of rejected architectural design alternatives.

9.4.2 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 or user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 10.

9.4.3 Architectural Design Elements

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

The architectural model [Sha15] is derived from three sources: (1) information about the application domain for the software to be built, (2) specific requirements model elements such as use cases or analysis classes, their relationships, and collaborations for the problem at hand, and (3) the availability of architectural styles (Chapter 10) and patterns (Chapter 14).

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 10.

9.4.4 Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. The detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

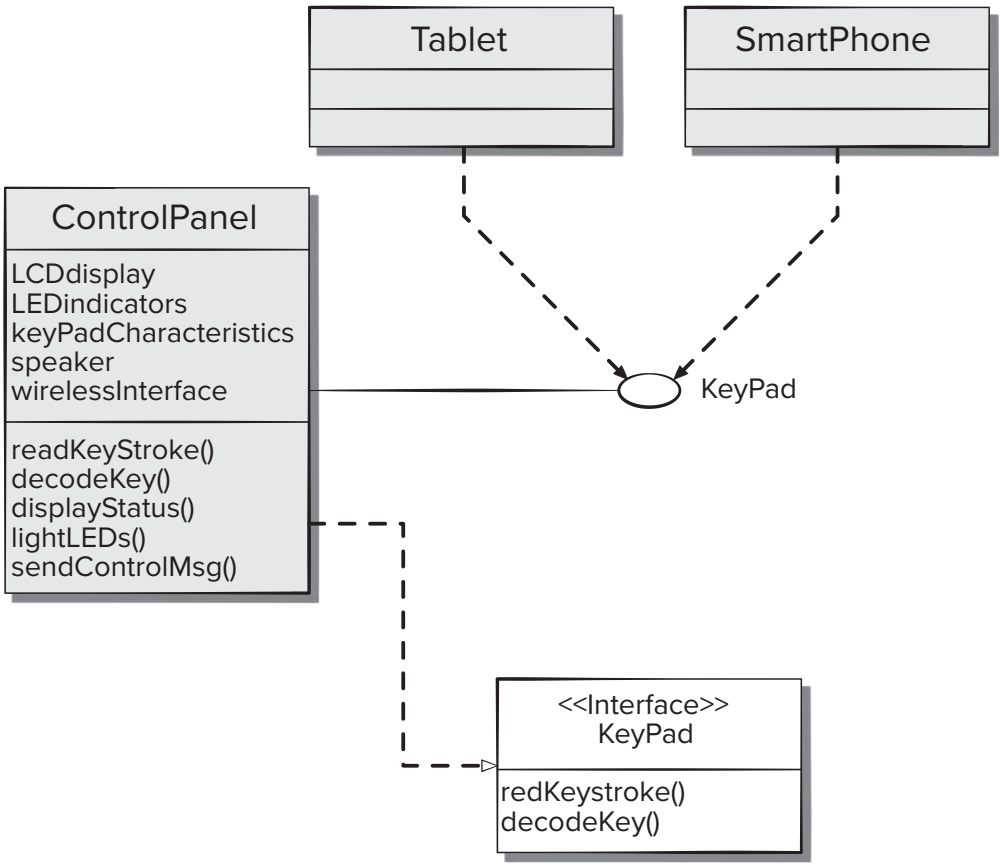
There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces; to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called *UX* or *user experience design*) is a major software engineering action and is considered in detail in Chapter 12. UX design focuses on ensuring the usability of the UI design. A usable design incorporates carefully chosen aesthetic elements (e.g., layout, color, graphics, information layout), ergonomic elements (e.g., interaction mechanisms, information placement, metaphors, UI navigation), and technical elements (e.g., UX patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture designed to provide the end user with a satisfying user experience.

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.[10] The design of external interfaces should incorporate error checking and appropriate security features.

10 Interface characteristics can change with time. Therefore, a designer should ensure that the specification for the interface is accurate and complete.

FIGURE 9.5
Interface representation for ControlPanel



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

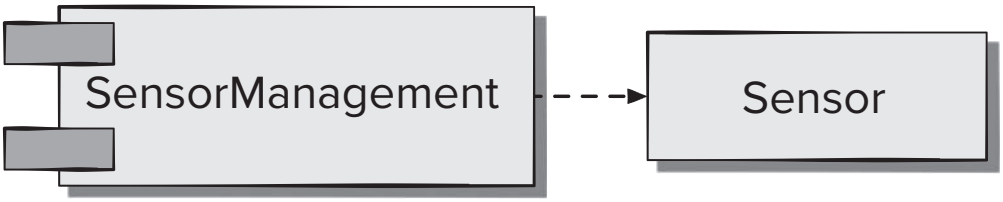
In some cases, an interface is modeled in much the same way as a class. An interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain features of the security function. In an advanced version of the system, control panel functions may be implemented via a mobile platform (e.g., smartphone or tablet) and are represented in Figure 9.5.

9.4.5 Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches,

FIGURE 9.6
A UML component diagram



faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

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

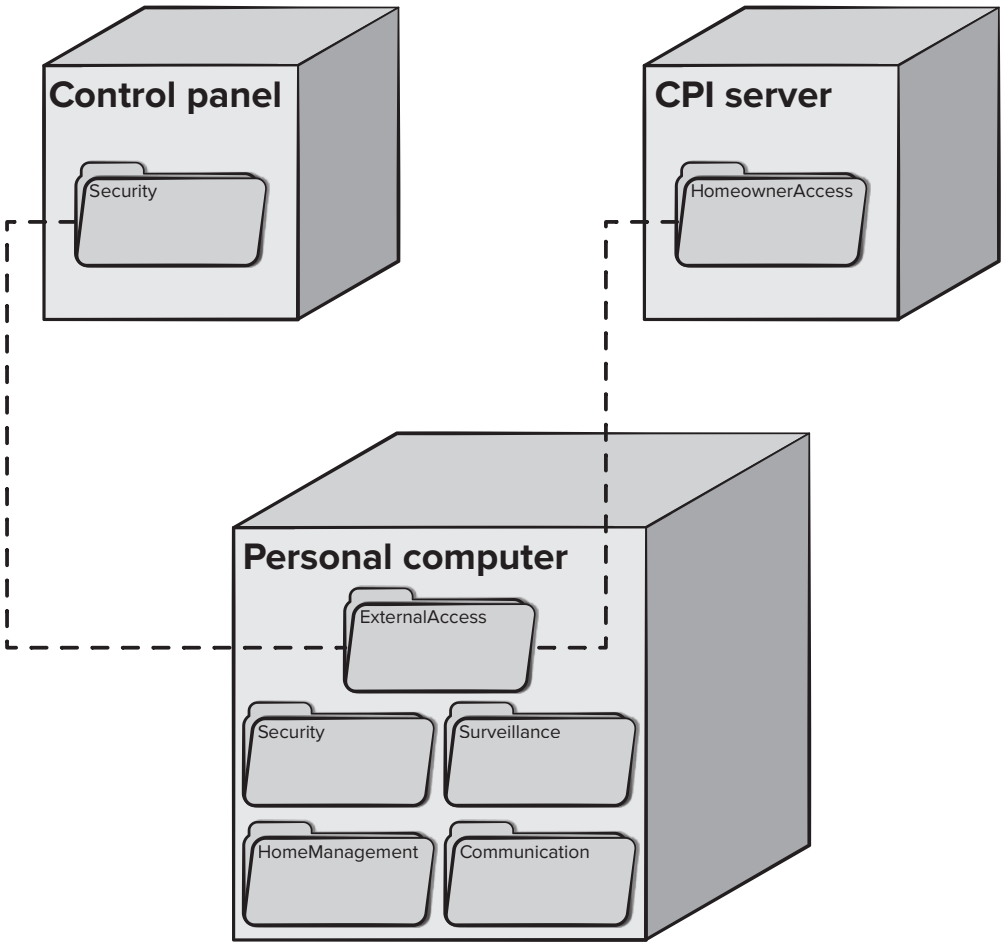
The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Algorithmic structure details for a component can be represented using either pseudocode (a programming languagelike representation described in Chapter 11) or some other diagrammatic form (e.g., flowchart). Data structure details are usually modeled using pseudocode or the programming language to be used for implementation.

9.4.6 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 mobile device—in this case a 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 the full design there would be more details included: sensors, cameras, and the functionality delivered by mobile platforms). 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.

FIGURE 9.7
A UML deployment diagram



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 Mac, a Windows-based PC, a Linux box, or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

9.5 SUMMARY

Software design commences as the first iteration of requirements engineering concludes. 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. Software designers 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 narrower 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 at the same time.

Design concepts have evolved over the first 60 years 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. In essence, design concepts emphasize the need for abstraction as a mechanism for creating reusable software components; the importance of architecture as a way to better understand the overall structure of a system; the benefits of pattern-based engineering as a technique for designing software with proven capabilities; the value of separation of concerns and effective modularity as a way to make software more understandable, more testable, and more maintainable; the consequences of information hiding as a mechanism for reducing the propagation of side effects when errors do occur; the impact of functional independence as a criterion for building effective modules; the use of refinement as a design mechanism; the application of refactoring for optimizing the design that is derived; the importance of object-oriented classes and the characteristics that are related to them; the need to use abstraction to reduce coupling between components; and the importance of design for testing.

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 requirements 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.

PROBLEMS AND POINTS TO PONDER

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

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

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

9.4. Describe software architecture in your own words.

9.5. Describe separation of concerns in your own words. Is there a case when a "divide and conquer" strategy may not be appropriate? How might such a case affect the argument for modularity?

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

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

9.8. Apply a "stepwise refinement approach" to develop three different levels of procedural abstractions for one or more of the following programs: (1) Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check. (2) Iteratively solve for the roots of a transcendental equation. (3) Develop a simple task-scheduling algorithm for an operating system.

9.9. Does *refactoring* mean that you modify the entire design iteratively? If not, what does it mean?

9.10. Briefly describe each of the four elements of the design model.