

KEY
CONCEPTS

abstraction	232
architecture	232
aspects	237
cohesion	236
data design	244
design process	228
functional	
independence	236
good design	228
information hiding	235
modularity	234
object-oriented	
design	238
patterns	233
quality attributes	230

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 itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design is pivotal to successful software engineering. In the early 1990s Mitch Kapur, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal*. He wrote:

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 . . .

QUICK
LOOK

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder 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 requirements model (that focuses on describing required data, function, and behavior), the design model 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.

Why is it important? Design allows you 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 these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

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 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.

quality guidelines	228
refactoring	238
separation of concerns	234
software design	230
stepwise refinement	237

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.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you 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, you must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model (Chapters 8 to 11). As this occurs, alternatives are considered and rejected, and you converge on “one particular configuration of components, and thus the creation of the final product” [Bel81].

Diversification and convergence combine intuition and judgment 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.

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 12 to 18 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 and Web-oriented design approaches.

note:

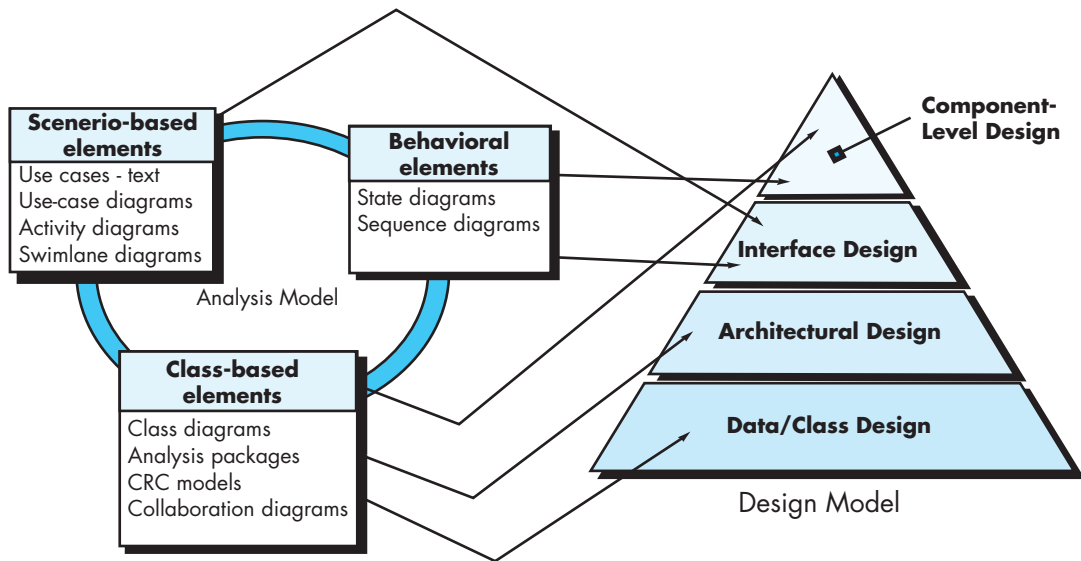
“The most common miracle of software engineering is the transition from analysis to design and design to code.”

Richard Due'

12.1 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

¹ Those readers with further interest in the philosophy of software design might have interest in Philippe Kruchten's intriguing discussion of “post-modern” design [Kru05].

FIGURE 12.1 Translating the requirements model into the design model

software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

Each of the elements of the requirements model (Chapters 9–11) 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 12.1. 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.



Software design 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.

The data/class design transforms class models (Chapter 10) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design 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 patterns (Chapter 13 that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

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

note:

"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 no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

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.

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

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

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 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 pretty 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.

12.2 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 connection is more subtle.

12.2.1 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 20. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

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

Quality Guidelines. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 12.3, 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 (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,² thereby facilitating implementation and testing.

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

note:

“[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

? What are the characteristics of a good design?

note:

"Design is not just what it looks like and feels like. Design is how it works."

Steve Jobs

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.

These design guidelines are not achieved by chance. 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 quality³ 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 20,⁴ 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; 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. At the conclusion of the TR, 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.

³ The quality factors discussed in Chapter 30 can assist the review team as it assesses quality.

⁴ You might consider looking ahead to Chapter 20 at this time. Technical reviews are a critical part of the design process and are an importance mechanism for achieving design quality.

Note:

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

Robert Pirsig



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.

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

- **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- **Usability** is assessed by considering human factors (Chapters 6 and 15), 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 using processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 29), 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.

12.2.2 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]). Growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04] emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

Note:

"A designer knows that he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

Antoine de St-Exupéry

? What characteristics are common to all design methods?

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapters 9 to 11, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

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

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 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.
5. 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.

12.3 DESIGN CONCEPTS

A set of fundamental software design concepts has 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 or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

M. A. Jackson [Jac75] once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.” In the sections that follow, we present an overview of fundamental software design concepts that provide the necessary framework for “getting it right.”

Note:

“Abstraction is one of the fundamental ways that we as humans cope with complexity.”

Grady Booch

12.3.1 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. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).⁵

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**.

WebRef

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

12.3.2 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 (modules), the manner in which these components interact, and the

⁵ 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.

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

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

Shaw and Garlan [Sha95a] 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. *Families of related systems* “draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.”

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (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.

A number of different *architectural description languages* (ADLs) have been developed to represent these models [Sha95b]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The manner in which software architecture is characterized and its role in design are discussed in Chapter 13.

12.3.3 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].

note:

“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.

note:

“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

Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern. Design patterns are discussed in detail in Chapter 16.

12.3.4 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 with regard to software modularity.

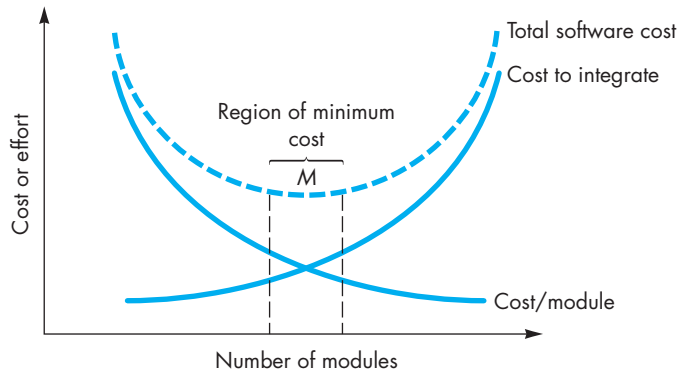
Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

12.3.5 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, as a consequence, 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 12.2, the effort (cost) to develop an individual software module does decrease as the total number of

FIGURE 12.2**Modularity and software cost**

modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

? What is the right number of modules for a given system?

The curves shown in Figure 12.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M . Undermodularity or overmodularity should be avoided. But how do you know the vicinity of M ? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

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

12.3.6 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 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

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.

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.

12.3.7 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] 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.

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

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across

? Why should you strive to create independent modules?

KEY POINT

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

the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect” [Ste74], caused when errors occur at one location and propagate throughout a system.

12.3.8 Refinement



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.

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

Refinement is actually a process of *elaboration*. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

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

12.3.9 Aspects



“It’s hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn’t a book on software design.”

Bruce Tognazzini

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

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

For example, consider two requirements for the **www.safehomeassured.com** WebApp. Requirement *A* is described via the ACS-DCV use case discussed in Chapter 9. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using www.safehomeassured.com*. This requirement



A crosscutting concern is some characteristic of the system that applies across many different requirements.

is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, A^* is a design representation for requirement A and B^* is a design representation for requirement B . Therefore, A^* and B^* are representations of concerns, and B^* *crosscuts* A^* .

An *aspect* is a representation of a crosscutting concern. Therefore, the design representation, B^* , of the requirement *a registered user must be validated prior to using **www.safehomeassured.com***, is an aspect of the *SafeHome* WebApp. It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components [Ban06a]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

12.3.10 Refactoring

WebRef

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

An important design activity suggested for many agile methods (Chapter 5), *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.”

WebRef

A variety of refactoring patterns can be found at <http://c2.com/cgi/wiki?RefactoringPatterns>.

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

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. As a consequence, refactoring tools [Soa10] are used to analyze changes automatically and to “generate a test suite suitable for detecting behavioral changes.”

12.3.11 Object-Oriented Design Concepts

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

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 a CS 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 “aspects” and “refactoring.”

Shakira: That's 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 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 commit to think about them as we develop the design.

? What types of classes does the designer create?

12.3.12 Design Classes

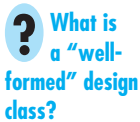
The analysis model defines a set of analysis classes (Chapter 10). Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

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 implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]. *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor. *Business domain classes* identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or

more analysis classes. *Process classes* implement lower-level business abstractions required to fully manage the business domain classes. *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software. *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class (Chapter 10) is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

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



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

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

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to

implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.⁶

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 10.3 and Figure 10.2) 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 10.2) 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 12.3.]

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.

12.3.13 Dependency Inversion

The structure of many older software architectures is hierarchical. At the top of the architecture, “control” components rely on lower-level “worker” components to perform various cohesive tasks. Consider a simple program with three components. The intent of the program is to read keyboard strokes and then print the result to a printer. A control module, *C*, coordinates two other modules—a keystroke reader module, *R*, and a module that writes to a printer, *W*.

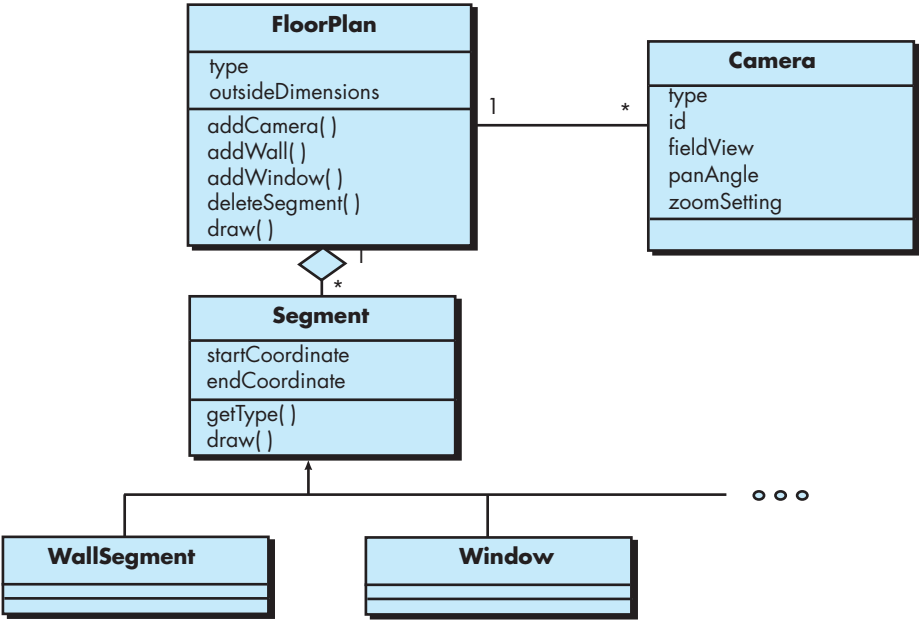
The design of the program is coupled because *C* is highly dependent on *R* and *W*. To remove the level of dependence that exists, the “worker” modules *R* and *W* should be invoked from the control module *S* using abstractions. In

? What is the “dependency inversion principle”?

⁶ A less formal way of stating the Law of Demeter is “Each unit should only talk to its friends; Don't talk to strangers.”

FIGURE 12.3

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



object-oriented software engineering, abstractions are implemented as abstract classes, **R*** and **W***. These abstract classes could then be used to invoke worker classes that perform any read and write function. Therefore a **copy** class, **C**, invokes abstract classes, **R*** and **W***, and the abstract class points to the appropriate worker-class (e.g., the **R*** class might point to a *read()* operation within a **keyboard** class in one context and a *read()* operation within a **sensor** class in another. This approach reduces coupling and improves the testability of a design.

The example discussed in the preceding paragraph can be generalized with the *dependency inversion principle* [Obj10], which states: *High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

12.3.14 Design for Test

There is an ongoing chicken-and-egg debate about whether software design or test case design should come first. Rebecca Wirfs-Brock [Wir09] writes:

Advocates of test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters’ credo, “Test fast, fail fast, adjust fast.” Testing guides their design as they implement in short, rapid-fire “write test code—fail the test—write enough code to pass—then pass the test” cycles.

note:

“Test fast, fail fast, adjust fast.”

Tom Peters

Copyright © 2014, McGraw-Hill Higher Education. All rights reserved.

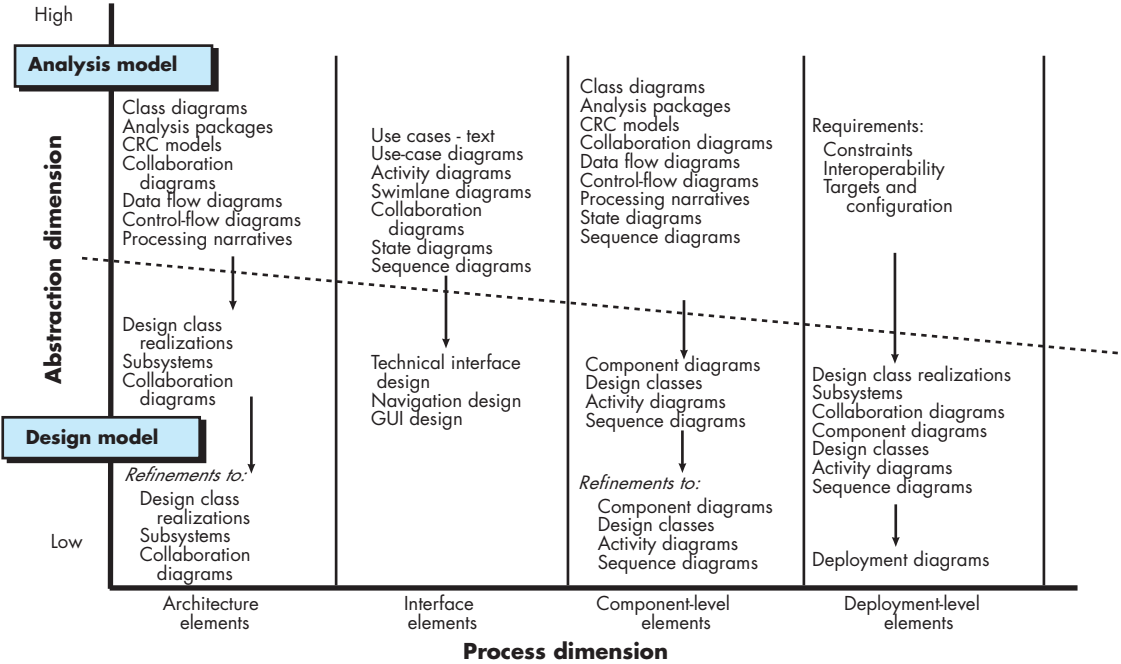
But if design comes first, then the design (and code) must be developed with *seams*—locations in the detailed design where you can “insert test code that probes the state of your running software” and/or “isolate code under test from its production environment so that you can exercise it in a controlled testing context” [Wir09].

Sometimes referred to as “test hooks,” seams must be consciously designed at the component level. To accomplish this, a designer must give thought to the tests that will be conducted to exercise the component. As Wirfs-Brock states: “In short, you need to provide appropriate test affordances—factoring your design in a way that lets test code interrogate and control the running system.”

12.4 THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 12.4. 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

FIGURE 12.4 Dimensions of the design model



KEY POINT

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

note:

“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

KEY POINT

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

models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

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

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

You can apply design patterns (Chapter 16) 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.

12.4.1 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.

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 13.

12.4.2 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

⁷ Appendix 1 provides a tutorial on basic UML concepts and notation.

note:

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

Frank Lloyd Wright

and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model [Sha96] is derived from three sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles (Chapter 13) and patterns (Chapter 16).

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 13.

12.4.3 Interface Design Elements

note:

"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

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. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of 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 *usability design*) is a major software engineering action and is considered in detail in Chapter 15. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

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

KEY POINT

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

⁸ Interface characteristics can change with time. Therefore, a designer should ensure that the specification for the interface is accurate and complete.

note:

“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

WebRef

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

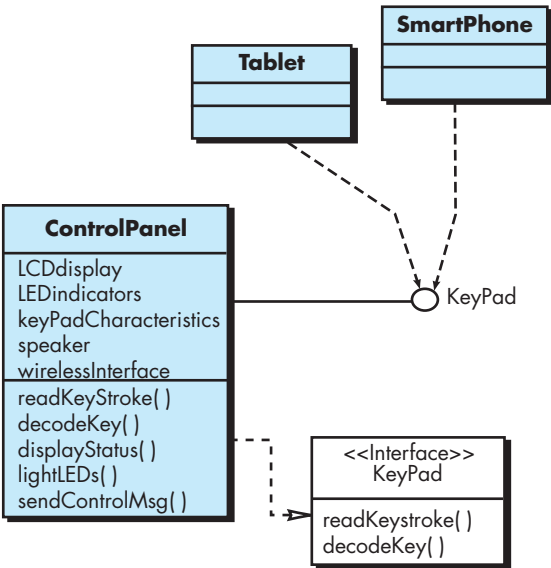
The design of internal interfaces is closely aligned with component-level design (Chapter 14). Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed 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. In UML, an interface is defined in the following manner [OMG03a]: “An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure.” Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

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

The **ControlPanel** class (Figure 12.5) provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke()* and *decodeKey()*. If these operations are to be provided to other classes (in this case, **Tablet** and **SmartPhone**), it is useful to define an interface as shown in the figure. The interface, named **KeyPad**, is shown as an `<<interface>>` stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

FIGURE 12.5
Interface representation for ControlPanel



note:

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

Douglas Adams

The dashed line with an open triangle at its end (Figure 12.5) 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.

12.4.4 Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

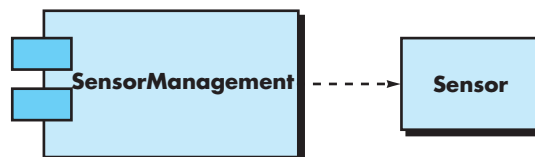
The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows 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 12.6. 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 14.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming language-like representation described in Chapter 14) or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

FIGURE 12.6

A UML component diagram




12.4.5 Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and sub-systems 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). In addition, limited functionality may be provided with mobile platforms.

During design, a UML deployment diagram is developed and then refined as shown in Figure 12.7. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and 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.

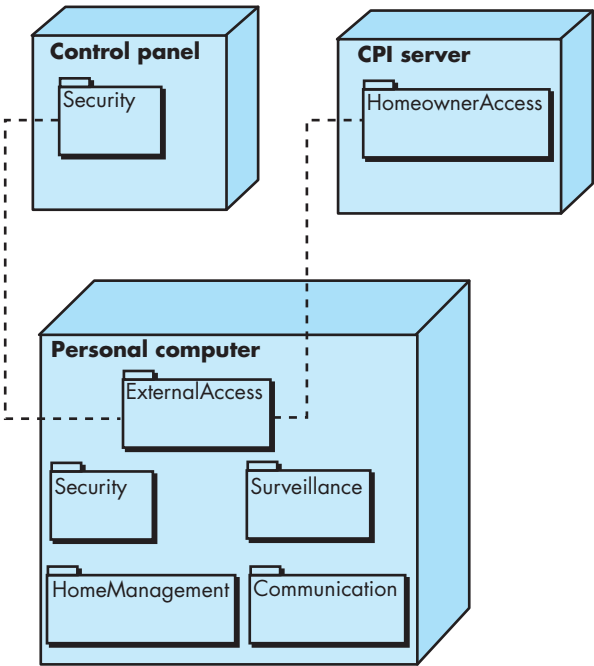
The diagram shown in Figure 12.7 is in *descriptor form*. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the “personal computer” is not further



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.

FIGURE 12.7
A UML deployment diagram



Copyright © 2014, McGraw-Hill Higher Education. All rights reserved.

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.

12.5 SUMMARY

Software design 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. 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 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 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; a consideration of aspects that crosscut system requirements; 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

- 12.1. Do you design software when you “write” a program? What makes software design different from coding?
- 12.2. If a software design is not a program (and it isn't), then what is it?
- 12.3. How do we assess the quality of a software design?
- 12.4. Examine the task set presented for design. Where is quality assessed within the task set? How is this accomplished? How are the quality attributes discussed in Section 12.2.1 achieved?
- 12.5. Provide examples of three data abstractions and the procedural abstractions that can be used to manipulate them.
- 12.6. Describe software architecture in your own words.
- 12.7. Suggest a design pattern that you encounter in a category of everyday things (e.g., consumer electronics, automobiles, appliances). Briefly describe the pattern.
- 12.8. 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?
- 12.9. 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?
- 12.10. Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.
- 12.11. How are the concepts of coupling and software portability related? Provide examples to support your discussion.
- 12.12. 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.
- 12.13. Consider the software required to implement a full navigation capability (using GPS) in a mobile, handheld communication device. Describe two or three crosscutting concerns that would be present. Discuss how you would represent one of these concerns as an aspect.
- 12.14. Does “refactoring” mean that you modify the entire design iteratively? If not, what does it mean?
- 12.15. Discuss what the dependency inversion principle is in your own words.
- 12.16. Why is design for testing so important?
- 12.17. Briefly describe each of the four elements of the design model.

FURTHER READINGS AND INFORMATION SOURCES

Donald Norman has written three books (*Emotional Design: We Love (or Hate) Everyday Things*, Basic Books, 2005), (*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*, 4th ed., Addison-Wesley, 2001) 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*, 2nd ed., Princeton University Press, 1988) provides a generic problem-solving process that can help software designers when they are faced with complex problems.

Books by Hanington and Martin (*Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*, Rockport, 2012) and Hanington and Martin (*Universal Principles of Design: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, 2nd ed., Rockport, 2010) discuss design principles in general.

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 design software that meets their needs. Holtzblatt (*Rapid Contextual Design: A How-to Guide to Key Techniques for User-Centered Design*, Morgan Kaufman, 2004) and 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. Bain (*Emergent Design*, Addison-Wesley, 2008) couples patterns, refactoring, and test-driven development into an effective design approach.

Comprehensive treatment of design in the context of software engineering is presented by Otero (*Software Engineering Design: Theory and Practice*, Auerbach, 2012), Venit and Drake (*Prelude to Programming: Concepts and Design*, 5th ed., Addison-Wesley, 2010), Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006), and Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2nd ed., Microsoft Press, 2004) presents an excellent discussion of the practical aspects of designing high-quality computer software. Robertson (*Simple Program Design*, 5th ed., Course Technology, 2006) presents an introductory discussion of software design that is useful for those beginning their study of the subject. Budgen (*Software Design*, 2nd ed., Addison-Wesley, 2004) introduces a variety of popular design methods, comparing and contrasting each. Fowler and his colleagues (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discuss techniques for the incremental optimization of software designs. Rosenberg and Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) discuss the development of object-oriented designs using use cases as a foundation.

A worthwhile historical survey of software design is contained in an anthology edited by Freeman and Wasserman (*Software Design Techniques*, 4th ed., IEEE, 1983). This tutorial reprints many of the classic papers that have formed the basis for current trends in software design. Measures of design quality, presented from both the technical and management perspectives, are considered by Card and Glass (*Measuring Software Design Quality*, Prentice Hall, 1990).

A wide variety of information sources on software design 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 website: www.mhhe.com/pressman.