**SOFTWARE ENGINEERING**

**UNIT III**

**1. Design Concepts: 2. The Design Process – 3. Design Concepts – 4. The Design Model – 5. Architectural Design: 6. Software Architecture – 7. Architectural Genres – 8. Architectural Styles – 9. Architectural Design – 10. Assessing Alternative Architectural Designs – 11. Architectural Mapping Using Data Flow.**

**1. 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 you in 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.

**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 QUICK LOOK 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, componentlevel, 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.

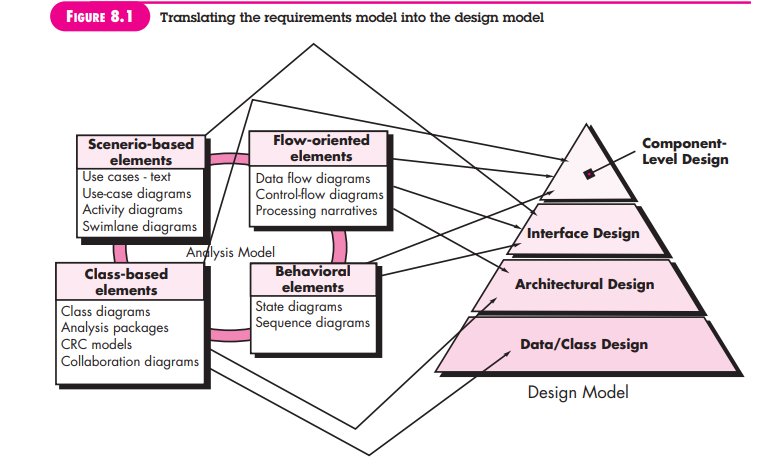
Design is pivotal to successful software engineering. In the early 1990s Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in Dr. Dobbs Journal.

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

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 5 through 7). 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.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, I 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 9 through 13 I’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.



**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.

**1. Software Quality Guidelines and Attributes**

**a. Quality Guidelines**

**b. Quality Attributes**

**2. The Evolution of Software Design**

**1. Software Quality Guidelines and Attributes**

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

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 15. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

• The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

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

**a. Quality Guidelines**

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

1. A design should exhibit an architecture that

(1) Has been created using recognizable architectural styles or patterns,

(2) Is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and

(3) Can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

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.

**b. Quality Attributes**

Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability.

The FURPS quality attributes represent a target for all software design:

• **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

• **Usability** is assessed by considering human factors (Chapter 11), overall aesthetics, consistency, and documentation.

• **Reliability** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

• **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

• **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 22), the ease with which a system can be installed, and the ease with which problems can be localized.

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

**2. The Evolution of Software Design**

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

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapters 6 and 7, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics:

(1) a mechanism for the translation of the requirements model into a design representation,

(2) a notation for representing functional components and their interfaces,

(3) heuristics for refinement and partitioning, and

(4) guidelines for quality assessment.

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

**3. Design Concepts**

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

• What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

M. A. Jackson [Jac75] once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.” Fundamental software design concepts provide the necessary framework for “getting it right.”

In the sections that follow, I present a brief overview of important software design concepts that span both traditional and object-oriented software development.

**1. Abstraction**

**2. Architecture**

**a. Structural properties**

**b. Extra-functional properties**

**c. Families of related systems**

**3. Patterns**

**4. Separation of Concerns**

**5. Modularity**

**6. Information Hiding**

**7. Functional Independence**

**8. Refinement**

**9. Aspects**

**10. Refactoring**

**11. Object-Oriented Design Concepts**

**12. Design Classes**

**a. Complete and sufficient**

**b. Primitiveness**

**c. High cohesion**

**d. Low coupling**

**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.

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

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

Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design:

**a. Structural properties**

This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

**b. Extra-functional properties**

The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

**c. Families of related systems**

The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. Structural models represent architecture as an organized collection of program components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

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

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

**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 particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine

(1)Whether the pattern is applicable to the current work,

(2) Whether the pattern can be reused (hence, saving design time), and

(3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern. Design patterns are discussed in detail in Chapter 12.

**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.

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

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

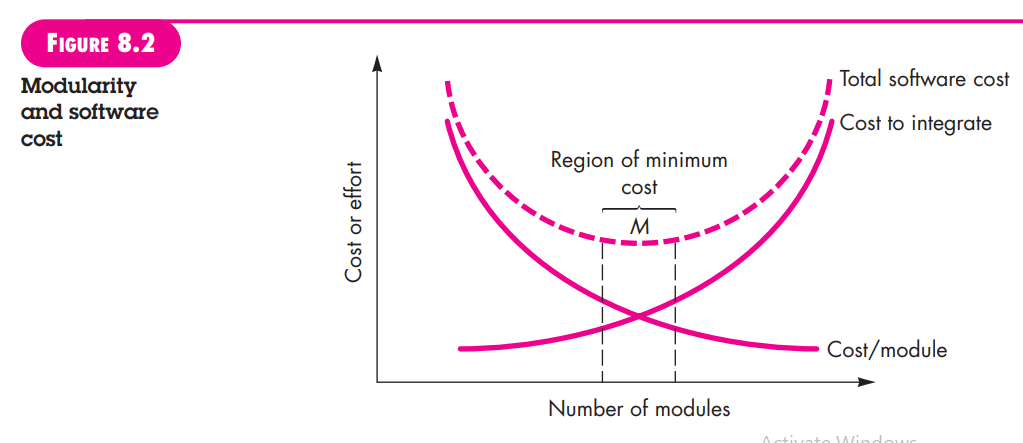
Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow

**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 my discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.

****

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

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

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

**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 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.

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

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

To summarize, functional independence is a key to good design, and design is the key to software quality. Independence is assessed using two qualitative criteria: cohesion and coupling. ***Cohesion is an indication of the relative functional strength of a module***. ***Coupling is an indication of the relative interdependence among modules***.

Cohesion is a natural extension of the information-hiding concept described in Section 8.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, “schizophrenic” components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

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

**8. Refinement**

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

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

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

**9. Aspects**

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

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

For example, consider two requirements for the SafeHomeAssured.com **WebApp**. Requirement A is described via the ACS-DCV use case discussed in Chapter 6. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement B is a generic security requirement that states that a registered user must be validated prior to using **SafeHomeAssured.com**. This requirement is applicable for all functions that are available to registered **SafeHome** users. As design refinement occurs, A\* is a design representation for requirement A and B\* is a design representation for requirement B. Therefore, A\* and B\* are representations of concerns, and B\* crosscuts A\*.

An aspect is a representation of a crosscutting concern. Therefore, the design representation, B\*, of the requirement a registered user must be validated prior to using SafeHomeAssured.com, is an aspect of the **SafeHome WebApp**. It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components [Ban06]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

**10. Refactoring**

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

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

The result will be software that is easier to integrate, easier to test, and easier to maintain.

**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,

Polymorphism

**12. Design Classes**

The requirements model defines a set of analysis classes (Chapter 6). Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high. As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]:

• User interface classes define all abstractions that are necessary for humancomputer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

• **Business domain classes** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

• **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.

• **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.

• **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation. Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is “well-formed.”

They define four characteristics of a well-formed design class:

**a. 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.

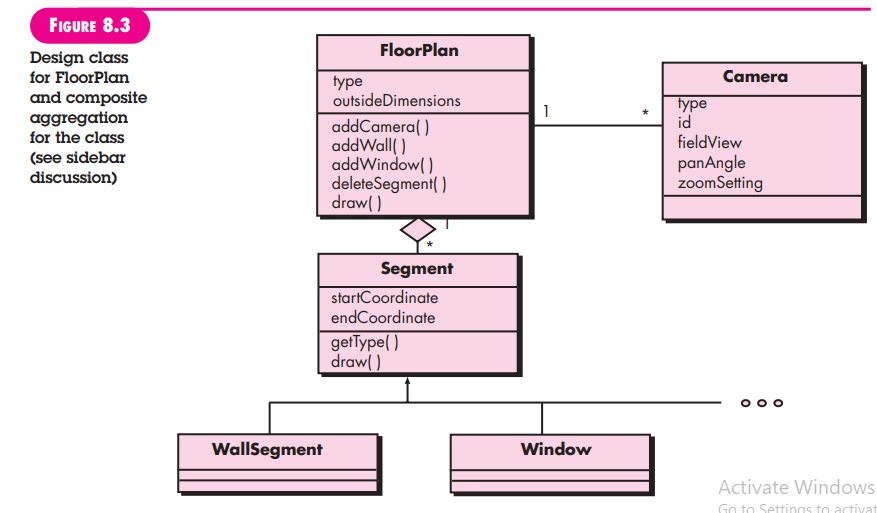
**b. Primitiveness**

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.

**c. 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 **Video Clip** 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.

**d. Low Coupling**

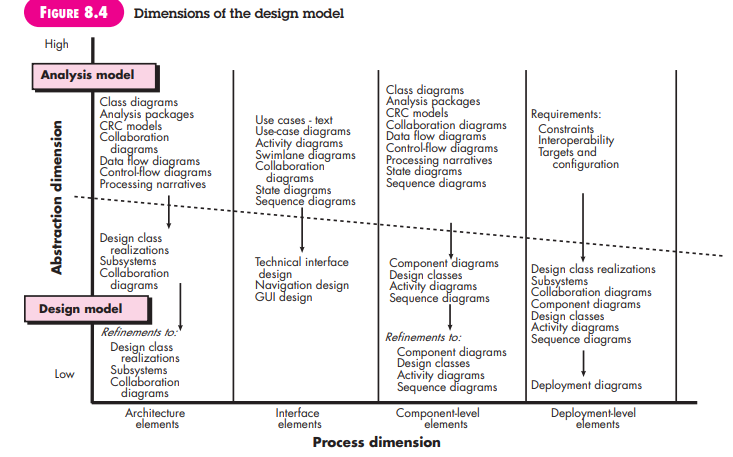
Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test,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

****

**4. The Design Model**

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

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

****

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed. You can apply design patterns (Chapter 12) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

**1. Data Design Elements**

**2. Architectural Design Elements**

**3. Interface Design Elements**

**4. Component-Level Design Elements**

**5. Deployment-Level Design Elements**

**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 9.

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

The architectural model [Sha96] is derived from three sources:

(1) Information about the application domain for the software to be built;

(2) Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and

(3) The availability of architectural styles (Chapter 9) and patterns (Chapter 12).

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have it’s own architecture (e.g., a graphical user interface might be structured according to a pre existing architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 9.

**3. Interface Design Elements**

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan. They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor’s presence, and how a security system is to be installed. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:

(1) The user interface (UI);

(2) External interfaces to other systems, devices, networks, or other producers or consumers of information; and

(3) Internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called usability design) is a major software engineering action and is considered in detail in Chapter 11. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

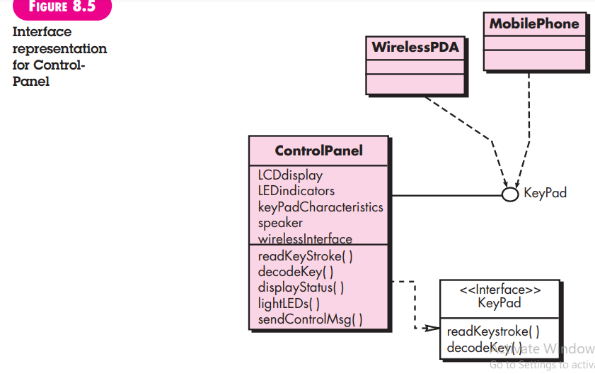
The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 5) and verified once the interface design commences.8 The design of external interfaces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

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

In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner [OMG03a]: “An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure.” Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

For example, the SafeHome security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone.

The ControlPanel class (Figure 8.5) provides the behavior associated with a keypad, and therefore, it must implement the operations readKeyStroke () and decodeKey (). If these operations are to be provided to other classes (in this case, WirelessPDA and MobilePhone), it is useful to define an interface as shown in the figure. The interface, named KeyPad, is shown as an <> 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.



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

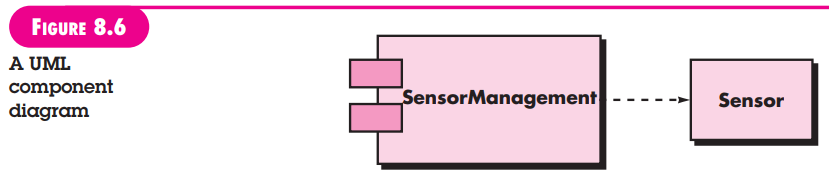
**4. Component-Level Design Elements**

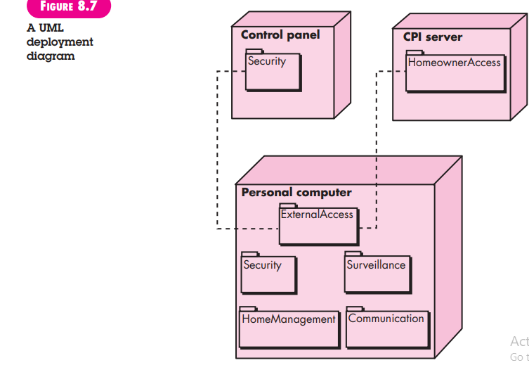
The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets,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

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6. In this figure, a component named SensorManagement (part of the SafeHome security function) is represented. A dashed arrow connects the component to a class named Sensor that is assigned to it. The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them. Further discussion of component diagrams is presented in Chapter 10.

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

**5. Deployment-Level Design Elements**





Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the SafeHome product are configured to operate within three primary computing environments—a home-based PC, the SafeHome control pane

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

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

**5. Architectural Design:**

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. This description is extended by Freeman [Fre80]:

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

As I noted in Chapter 8, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

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

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

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

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

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

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

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

**6. Software Architecture**

In their landmark book on the subject, Shaw and Garlan [Sha96] discuss software architecture in the following manner:

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

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

**1. What Is Architecture?**

**2. Why Is Architecture Important?**

**3. Architectural Descriptions**

**4. Architectural Decisions**

**1. What Is Architecture?**

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity.

It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal “living environment.” It is small details— the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art. But architecture is also something else. It is “thousands of decisions, both big and small” [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

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

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

This definition emphasizes the role of “software components” in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and “middleware” that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what I call “architectural design”) and the actions that are applied to derive the software design. As one reviewer of this edition noted:

There is a distinct difference between the terms architecture and design. A design is an instance of an architecture similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix “architecture” and “design” with each other.

Although I agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design that evolves from them. Design begins with a consideration of architecture.

In this book the design of software architecture considers two levels of the design pyramid (Figure 8.1)—data design and architectural design. In the context of the preceding discussion, data design enables you to represent the data component of the architecture in conventional systems and class definitions (encompassing attributes and operations) in object-oriented systems. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

**2. Why Is Architecture Important?**

In a book dedicated to software architecture, Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

• Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

• Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together” [Bas03].

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 9.2 through 9.4) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable

**3. Architectural Descriptions**

Each of us has a mental image of what the word architecture means. In reality, however, it means different things to different people. The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

For example, the architect of a major office building must work with a variety of different stakeholders. The primary concern of the owner of the building (one stakeholder) is to ensure that it is aesthetically pleasing and that it provides sufficient office space and infrastructure to ensure its profitability. Therefore, the architect must develop a description using views of the building that address the owner’s concerns. The viewpoints used are a three-dimensional drawings of the building (to illustrate the aesthetic view) and a set of two-dimensional floor plans to address this stakeholder’s concern for office space and infrastructure.

But the office building has many other stakeholders, including the structural steel fabricator who will provide steel for the building skeleton. The structural steel fabricator needs detailed architectural information about the structural steel that will support the building, including types of I-beams, their dimensions, connectivity, materials, and many other details. These concerns are addressed by different work products that represent different views of the architecture. Specialized drawings (another viewpoint) of the structural steel skeleton of the building focus on only one of many of the fabricator’s concerns.

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building. Tyree and Akerman [Tyr05] note this when they write: “Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture’s key aspects.” Each of these “wants” is reflected in a different view represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, [IEE00], with the following objectives: (1) to establish a conceptual framework and vocabulary for use during the design of software architecture, (2) to provide detailed guidelines for representing an architectural description, and (3) to encourage sound architectural design practices.

The IEEE standard defines an architectural description (AD) as “a collection of products to document an architecture.” The description itself is represented using multiple views, where each view is “a representation of a whole system from the perpective of a related set of [stakeholder] concerns.” A view is created according to rules and conventions defined in a viewpoint—“a specification of the conventions for constructing and using a view” [IEE00]. A number of different work products that are used to de

**4. Architectural Decisions**

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture.

The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns. As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish an historical record that can be useful when design modifications must be made.

**7. Architectural Genres**

Although the underlying principles of architectural design apply to all types of architecture, the architectural genre will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, genre implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on. Within each general style, more specific styles might apply (Section 9.3). Each style would have a structure that can be described using a set of predictable patterns.

In his evolving Handbook of Software Architecture [Boo08], Grady Booch suggests the following architectural genres for software-based systems:

• Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes. • Commercial and nonprofit—Systems that are fundamental to the operation of a business enterprise. • Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure. • Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts.

• Devices—Systems that interact with the physical world to provide some point service for an individual.

• Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience.

• Financial—Systems that provide the infrastructure for transferring and managing money and other securities.

• Games—Systems that provide an entertainment experience for individuals or groups.

• Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.

• Industrial—Systems that simulate or control physical processes.

• Legal—Systems that support the legal industry.

• Medical—Systems that diagnose or heal or that contribute to medical research.

• Military—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.

• Operating systems—Systems that sit just above hardware to provide basic software services.

• Platforms—Systems that sit just above operating systems to provide advanced services.

• Scientific—Systems that are used for scientific research and applications.

• Tools—Systems that are used to develop other systems.

• Transportation—Systems that control water, ground, air, or space vehicles.

• Utilities—Systems that interact with other software to provide some point service.

From the standpoint of architectural design, each genre represents a unique challenge. As an example, consider the software architecture for a game system. Game systems, sometimes called immersive interactive applications, require the computation of intensive algorithms, sophisticated computer graphics, streaming multimedia data sources, real-time interactivity via conventional and unconventional inputs, and a variety of other specialized concerns.

Alexandre Francois [Fra03] suggests a software architecture for Immersipresence1 that can be applied for a gaming environment. He describes the architecture in the following manner:

SAI (Software Architecture for Immersipresence) is a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. The goal of SAI is to provide a universal framework for the distributed implementation of algorithms and their easy integration into complex systems. . . . The underlying extensible data model and hybrid (shared repository and message-passing) distributed asynchronous parallel processing model allow natural and efficient manipulation of generic data streams, using existing libraries or native code alike. The modularity of the style facilitates distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution.

A detailed discussion of SAI is beyond the scope of this book. However, it is important to recognize that the gaming system genre can be addressed with an architectural style (Section 9.3) that has been specifically designed to address gaming system concerns. If you have further interest, see [Fra03]

**8. Architectural Styles**

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

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

(1) a set of components (e.g., a database, computational modules) that perform a function required by a system;

(2) a set of connectors that enable “communication, coordination and cooperation” among components;

(3) constraints that define how components can be integrated to form the system; and

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

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

However, a pattern differs from a style in a number of fundamental ways:

(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00]; (3) architectural patterns (Section 9.4) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. In Section 9.3.1, I consider commonly used architectural styles and patterns for software.

**Canonical Architectural Structures**

In essence, software architecture represents a structure in which some collection of entities (often called components) is connected by a set of defined relationships (often called connectors). Both components and connectors are associated with a set of properties that allow the designer to differentiate the types of components and connectors that can be used. But what kinds of structures (components, connectors, and properties) can be used to describe an architecture? Bass and Kazman [Bas03] suggest five canonical or foundation architectural structures:

**1. Functional structure**. Components represent function or processing entities. Connectors represent interfaces that provide the ability to “use” or “pass data to” a component. Properties describe the nature of the components and the organization of the interfaces.

**2. Implementation structure.** “Components can be packages, classes, objects, procedures, functions, methods, etc., all of which are vehicles for packaging functionality at various levels of abstraction” [Bas03]. Connectors include the ability to pass data and control, share data, “use”, and “is-an-instance-of.” Properties focus on quality characteristics (e.g., maintainability, reusability) that result when the structure is implemented.

**3.Concurrency structure.** Components represent “units of concurrency” that are organized as parallel tasks or threads. “Relations [connectors] include synchronizes-with, is-higher-priority-than, sends-data-to, can’t-run-without, and can’t-run-with. Properties relevant to this structure include priority, preemptability, and execution time” [Bas03]. **4. Physical structure.** This structure is similar to the deployment model developed as part of design. The components are the physical hardware on which software resides. Connectors are the interfaces between hardware components, and properties address capacity, bandwidth, performance, and other attributes. **5. Developmental structure.** This structure defines the components, work products, and other information sources that are required as software engineering proceeds. Connectors represent the relationships among work products, and properties identify the characteristics of each item. Each of these structures presents a different view of software architecture, exposing information that is useful to the software team as modeling and construction proceed.

**1. A Brief Taxonomy of Architectural Styles**

**a. Data - Centered architectures.**

**b. Data - flow architectures.**

**c. Call and return architectures.**

**d. Object - oriented architectures.**

**e. Layered architectures.**

**2. Architectural Patterns**

**3. Organization and Refinement**

**a. Control**

**b. Data**

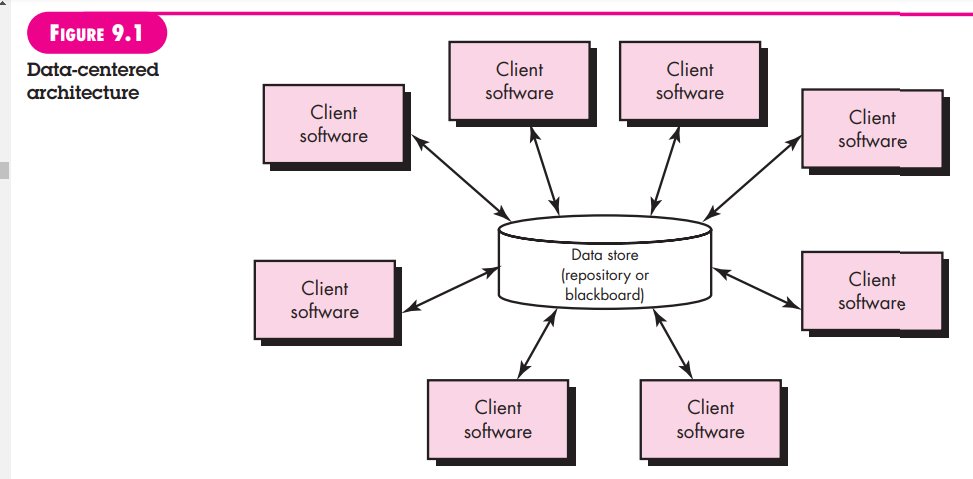
**1. A Brief Taxonomy of Architectural Styles**

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

**a. Data - Centered architectures.**

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.

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

****

**b. Data - flow architectures.**

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

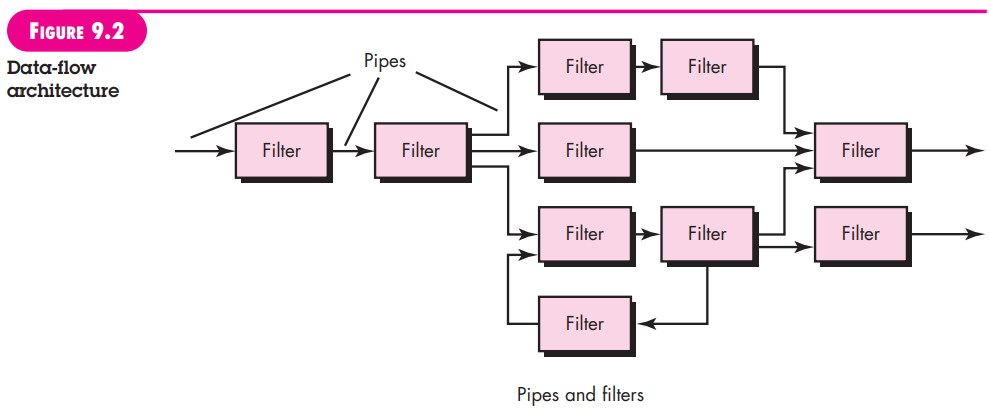
If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

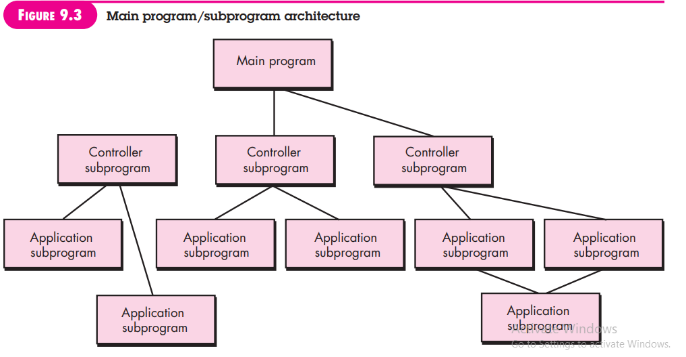
**c. Call and return architectures.**

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

**• Main program/subprogram architectures.** This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. Figure 9.3 illustrates an architecture of this type.

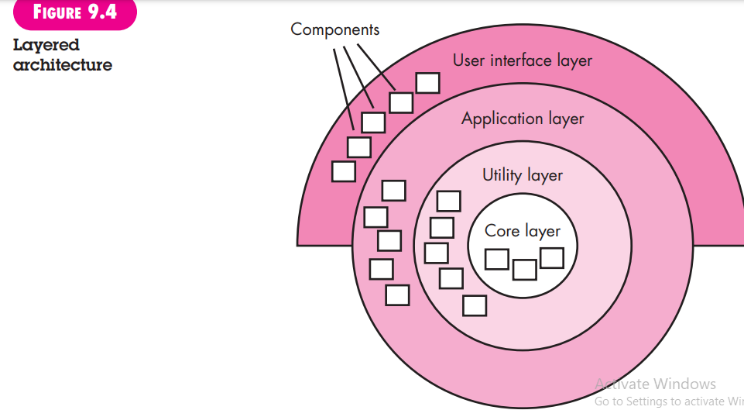
**• Remote procedure call architectures**. The components of a main program/subprogram architecture are distributed across multiple computers on a network.



****

**d. Object - oriented architectures.**

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

****

**e. Layered architectures.**

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

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

**2. Architectural Patterns**

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

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

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

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

**3. Organization and Refinement**

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

**a. Control**

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

**b. Data**

How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

**9. Architectural Design**

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived. In the sections that follow we examine each of these architectural design tasks in a bit more detail.

**1. Representing the System in Context**

**2. Defining Archetypes**

**3. Refining the Architecture into Components**

**4. Describing Instantiations of the System**

**1. Representing the System in Context**

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 9.5. Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

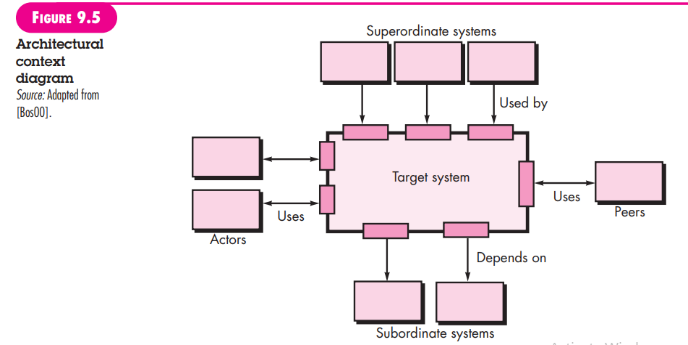
**• Superordinate systems—**those systems that use the target system as part of some higher-level processing scheme.

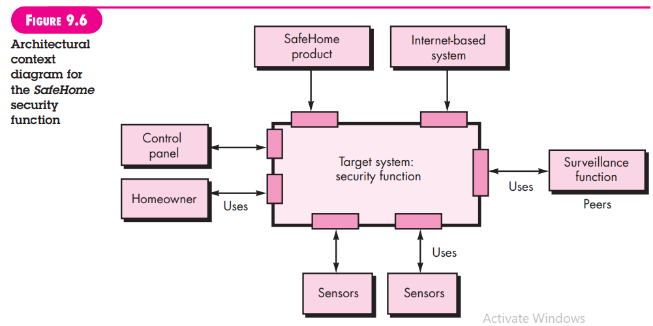
**• Subordinate systems—**those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

**• Peer-level systems—**those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

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

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

****

****

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

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

**2. Defining Archetypes**

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

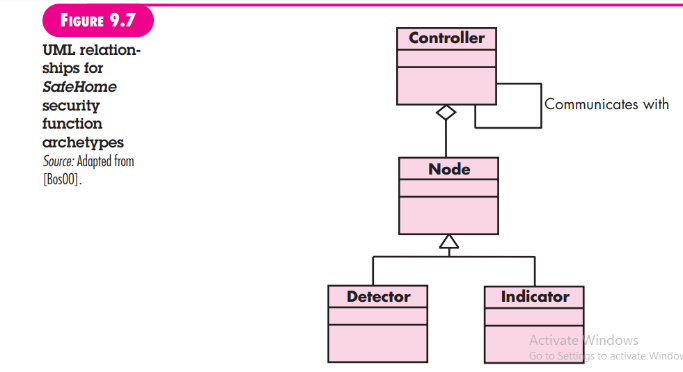
In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the SafeHome home security function, you might define the following archetypes:

**• Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.

**• Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

**• Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

**• Controller**. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

****

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

**3. Refining the Architecture into Components**

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model.4 These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

The interfaces depicted in the architecture context diagram (Section 9.4.1) imply one or more specialized components that process the data that flows across the interface.

In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

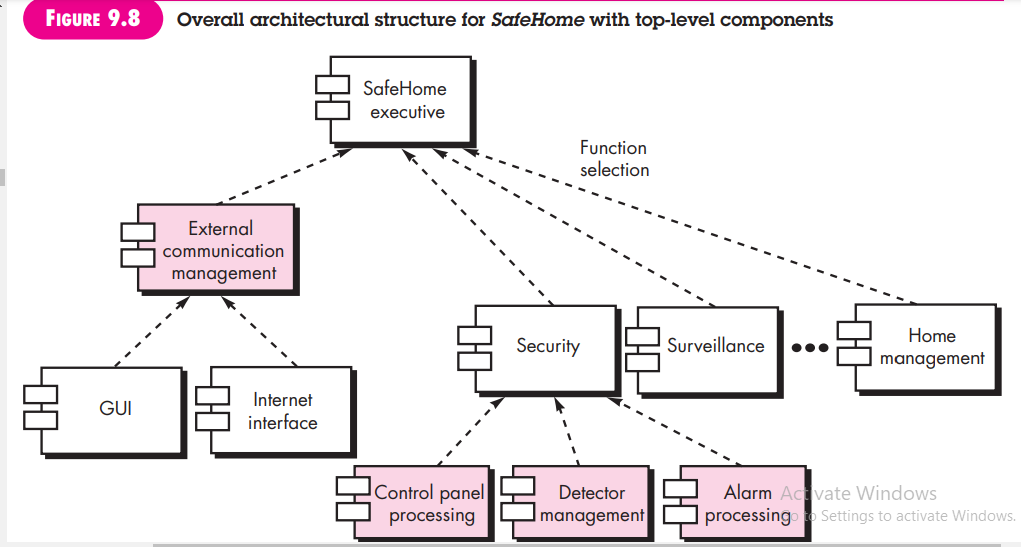
Continuing the SafeHome home security function example, you might define the set of top-level components that address the following functionality:

• External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

• Control panel processing—manages all control panel functionality.

• Detector management—coordinates access to all detectors attached to the system.

• Alarm processing—verifies and acts on all alarm conditions.

****

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

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 9.8. Transactions are acquired by external communication management as they move in from components that process the SafeHome GUI and the

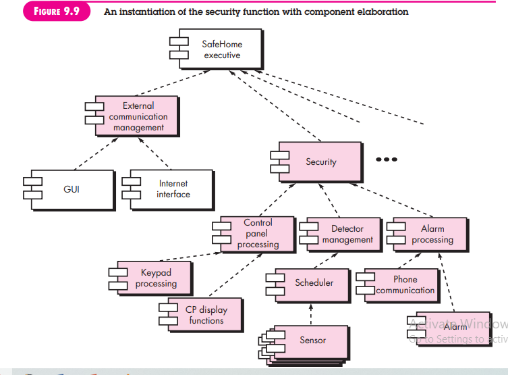
Internet interface. This information is managed by a SafeHome executive component that selects the appropriate product function (in this case security). The control panel processing component interacts with the homeowner to arm/disarm the security function. The detector management component polls sensors to detect an alarm condition, and the alarm processing component produces output when an alarm is detected.

**4. Describing Instantiations of the System**

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

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

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



**10. Assessing Alternative Architectural Designs**

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state:

To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn’t it be nice to know in advance if you’ve placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you’re buying a system or paying for its development, wouldn’t you like to have some assurance that it’s started off down the right path? If you’re the architect yourself, wouldn’t you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Indeed, answers to these questions would have value. Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, I present two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

**1. An Architecture Trade-Off Analysis Method**

**2. Architectural Complexity**

**3. Architectural Description Languages**

**1. An Architecture Trade-Off Analysis Method**

The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) [Kaz98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively: 1. Collect scenarios. A set of use cases (Chapters 5 and 6) is developed to represent the system from the user’s point of view.

2. Elicit requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.

3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements. The architectural style(s) should be described using one of the following architectural views:

• Module view for analysis of work assignments with components and the degree to which information hiding has been achieved.

• Process view for analysis of system performance.

• Data flow view for analysis of the degree to which the architecture meets functional requirements.

4. Evaluate quality attributes by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.

6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5. The SEI describes this approach in the following manner [Kaz98]:

Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.6

**2. Architectural Complexity**

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao [Zha98] suggests three types of dependencies:

**Sharing dependencies** represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.

**Flow dependencies** represent dependence relationships between producers and consumers of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.

**Constrained dependencies** represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling discussed in Chapter 8. Coupling is an important design concept that is applicable at the architectural level and at the component level. Simple metrics for evaluating coupling are discussed in Chapter 23.

**3. Architectural Description Languages**

The architect of a house has a set of standardized tools and notation that allow the design to be represented in an unambiguous, understandable fashion. Although the software architect can draw on UML notation, other diagrammatic forms, and a few related tools, there is a need for a more formal approach to the specification of an architectural design.

**Architectural description language (ADL)** provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [Hof01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, languagebased techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

**11. Architectural Mapping Using Data Flow.**

The architectural styles discussed in Section 9.3.1 represent radically different architectures. So it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques discussed in Section 9.4.

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures discussed earlier in this chapter. For example, the architecture of one or more components of a client-server architecture might be call and return. A mapping technique, called structured design [You79], is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram (Chapter 7) to software architecture.7 The transition from information flow (represented as a DFD) to program structure is accomplished as part of a sixstep process:

(1) the type of information flow is established,

(2) flow boundaries are indicated, (3) the DFD is mapped into the program structure,

(4) control hierarchy is defined,

(5) the resultant structure is refined using design measures and heuristics, and

(6) the architectural description is refined and elaborated.

As a brief example of data flow mapping, I present a step-by-step “transform” mapping for a small part of the SafeHome security function.8 In order to perform the mapping, the type of information flow must be determined. One type of ***information flow is called transform flow*** and exhibits a linear quality. Data flows into the system along an incoming flow path where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a transform center. Finally, it flows out of the system along an outgoing flow path that transforms the data into external world form.9

**1. Transform Mapping**

**Step 1. Review the fundamental system model**

**Step 2. Review and refine data flow diagrams for the software**

**Step 3. Determine whether the DFD has transform or transaction flow11 characteristics**

**Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.**

**Step 5. Perform “first-level factoring.”**

**Step 6. Perform “second-level factoring.”**

**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.**

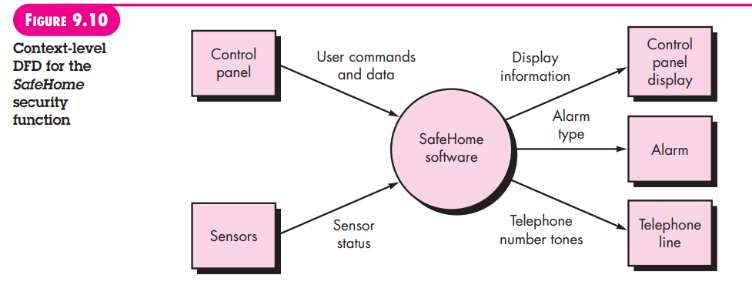
**2. Refining the Architectural Design**

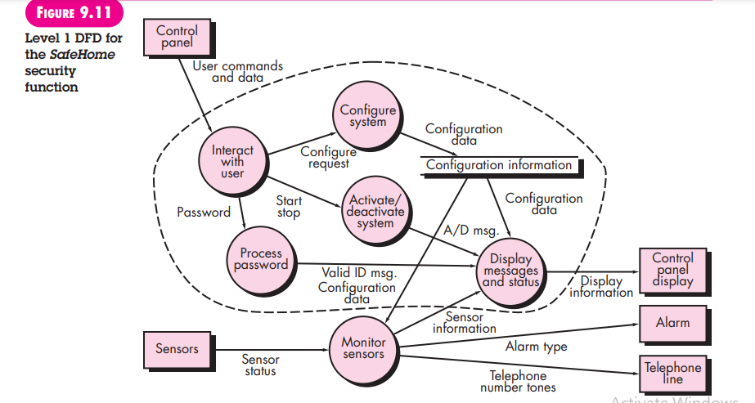
**1. Transform Mapping**

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the SafeHome security function.10 One element of the analysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

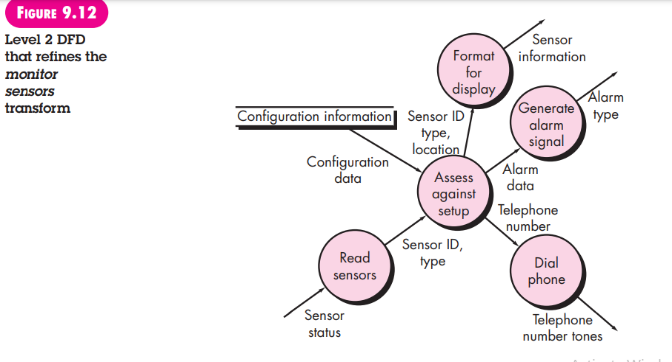
**Step 1. Review the fundamental system model**

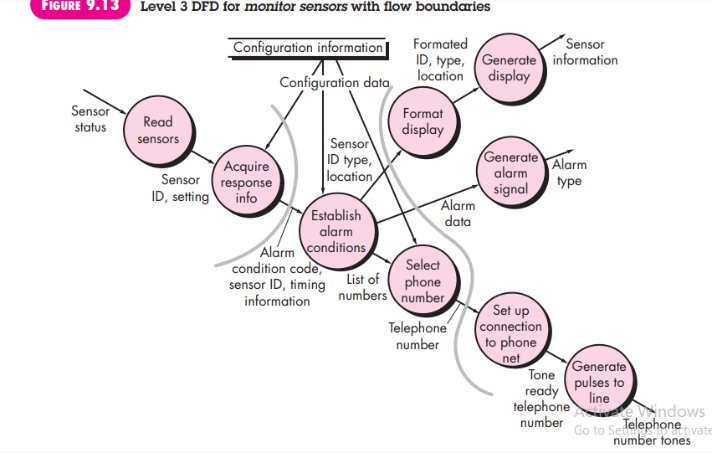
The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

****

****

**Step 2. Review and refine data flow diagrams for the software**

****

****

Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. At level 3, each transform in the data flow diagram exhibits relatively high cohesion (Chapter 8). That is, the process implied by a transform performs a single, distinct function that can be implemented as a component in the SafeHome software. Therefore, the DFD in Figure 9.13 contains sufficient detail for a “first cut” at the design of architecture for the monitor sensors subsystem, and we proceed without further refinement

**Step 3. Determine whether the DFD has transform or transaction flow11 characteristics**

Evaluating the DFD (Figure 9.13), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

**Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.**

Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g., an incoming flow boundary separating read sensors and acquire response info could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

**Step 5. Perform “first-level factoring.”**

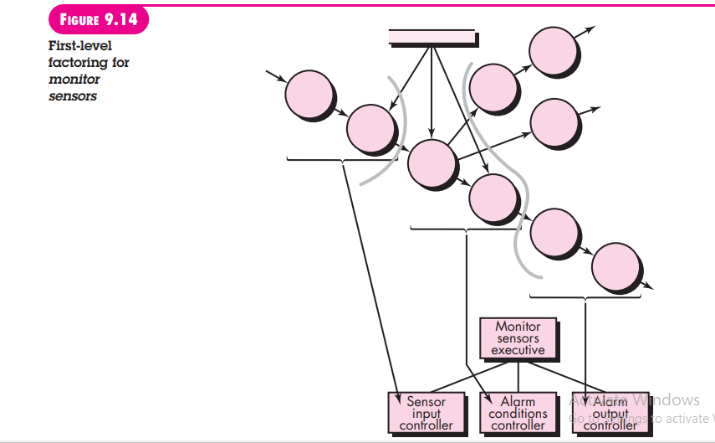
The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and lowlevel components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the monitor sensors subsystem is illustrated in Figure 9.14. A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:

• An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.

• A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

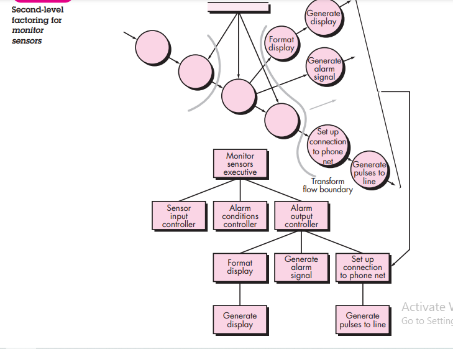
• An outgoing information processing controller, called alarm output controller, coordinates production of output information.



Although a three-pronged structure is implied by Figure 9.14, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics

**Step 6. Perform “second-level factoring.”**

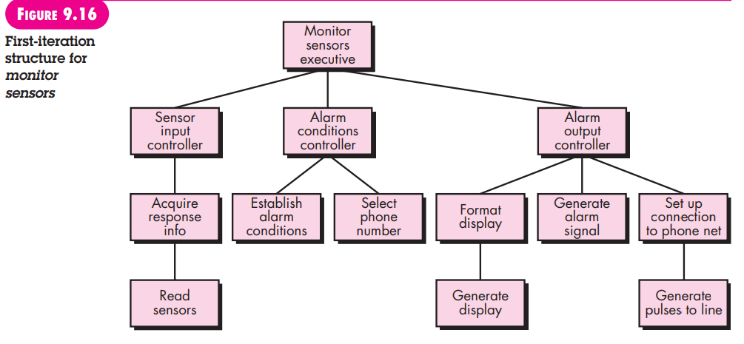
” Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to secondlevel factoring is illustrated in Figure 9.15.

****

Although Figure 9.15 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measuresof design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a “first-iteration” design.

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure 9.16.

The components mapped in the preceding manner and shown in Figure 9.16 represent an initial design of software architecture. Although components are named in a manner that implies function, a brief processing narrative (adapted from the process specification developed for a data transformation created during requirements modeling) should be written for each. The narrative describes the component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features (e.g., file input-output, hardwaredependent characteristics, special timing requirements).

****

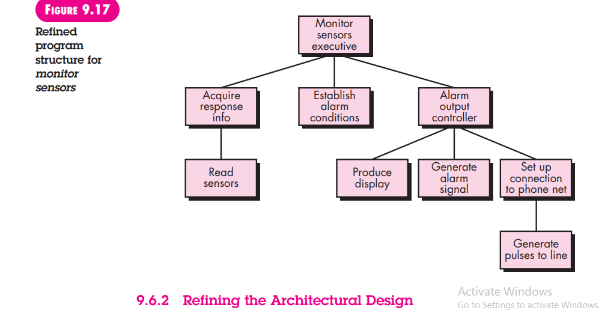
**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.**

A first-iteration architecture can always be refined by applying concepts of functional independence (Chapter 8). Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refinements are dictated by the analysis and assessment methods described briefly in Section 9.5, as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a component that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

You should pause for a moment and consider the difference between the design approach described and the process of “writing programs.” If code is the only representation of software, you and your colleagues will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty “seeing the forest for the trees.”

****

**2. Refining the Architectural Design**

Any discussion of design refinement should be prefaced with the following comment: “Remember that an ‘optimal design’ that doesn’t work has questionable merit.” You should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

Refinement of software architecture during early stages of design is to be encouraged. As I discussed earlier in this chapter, alternative architectural styles may be derived, refined, and evaluated for the “best” approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements