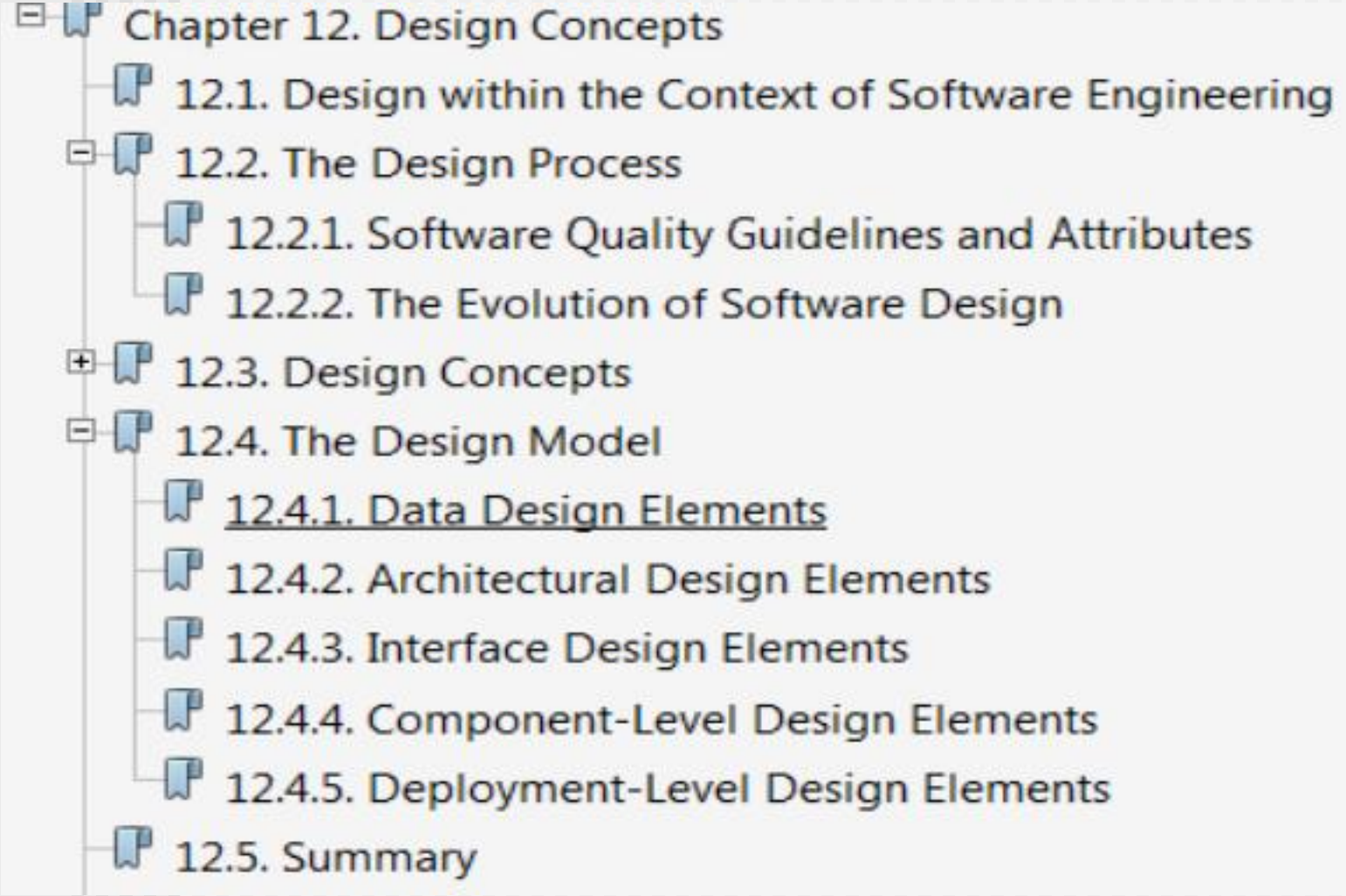
Chapter 12 DESIGN CONCEPTS



Chapter 12 DESIGN CONCEPTS

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

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（system or software architecture), data structures(including data warehouse,database ,data structure), interfaces(external and internal interface), and components (class design)that are necessary to implement the system.

Chapter 12 DESIGN CONCEPTS

This design 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.

Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then,data design and 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.

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

Chapter 12 DESIGN CONCEPTS

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.

In this chapter, we explore the fundamental concepts and principles that are applicable to all software design, the elements of the design model, and the impact of patterns on the design process. In Chapters 12 to 14 we'll present a variety of software design methods as they are applied to architectural,data,interface, and component-level design as well as pattern-based and Web-oriented design approaches.

Chapter 12 DESIGN CONCEPTS

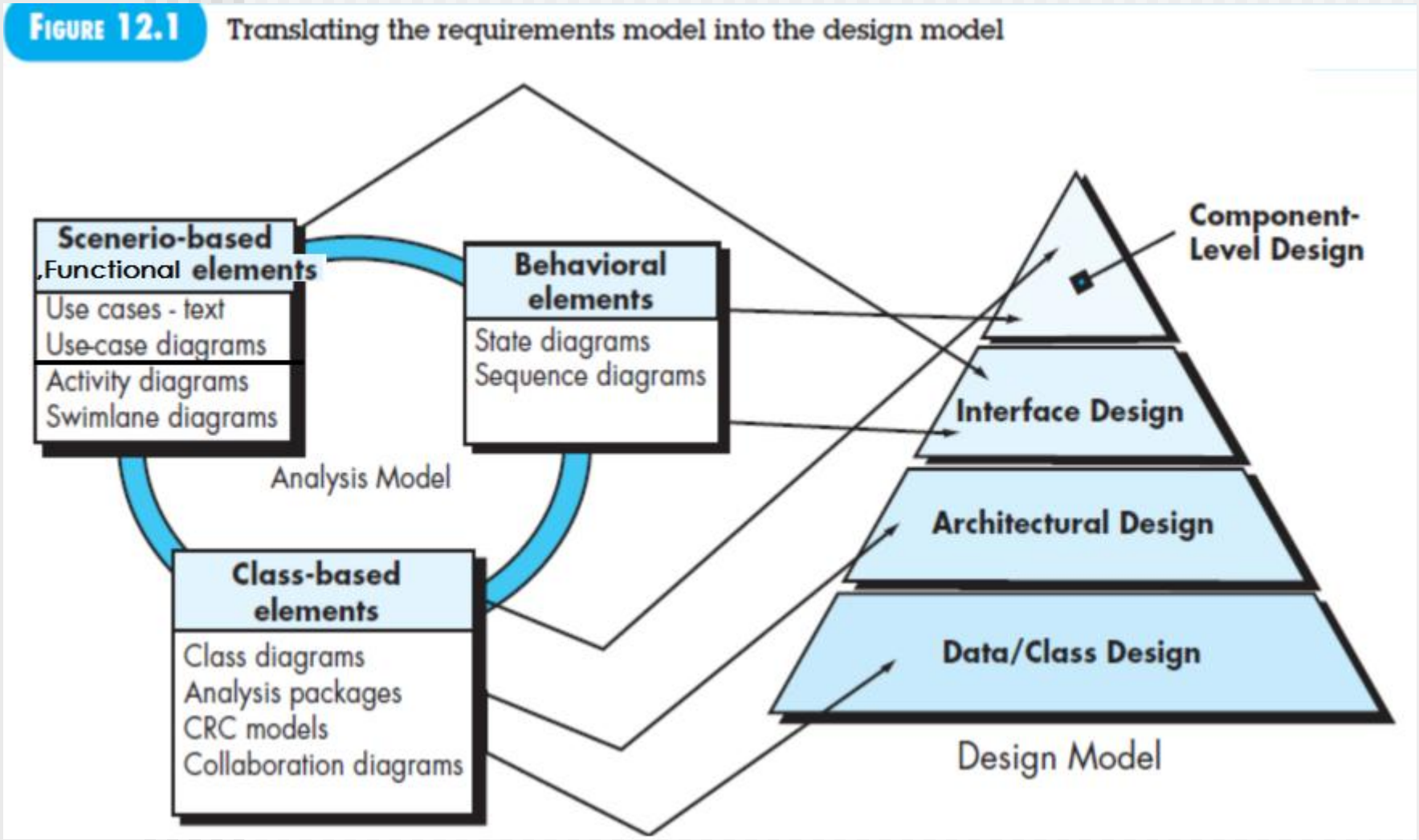
12.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

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

Each of the elements of the requirements model (Chapters 9–11) provides information that is necessary to create the four design models required for a complete specification of design.The flow of information during software design is illustrated in Figure 12.1 . The requirements model, manifested by scenario-based, class-based, functionaland behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design

Chapter 12 DESIGN CONCEPTS

FIGURE 12.1 Translating the requirements model into the design model



Chapter 12 DESIGN CONCEPTS

produces a data/class design, an architectural design, an interface design, and a component design.

The data/class design transforms class models (Chapter 10) into design class realizations(Here design class is different from component design) and the requisite data structures(including database design) required to implement the software.The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns (Chapter 13 that can be used to achieve the requirements defined for the system, and the constraints that affect the way in

which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

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

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

Chapter 12 DESIGN CONCEPTS

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

12.2 THE DESIGN PROCESS

Chapter 12 DESIGN CONCEPTS

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

12.2.1 Software Quality Guidelines and Attributes

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

Chapter 12 DESIGN CONCEPTS

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

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

Chapter 12 DESIGN CONCEPTS

Quality Guidelines. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 12.3, we discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

① **A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.**

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

Chapter 12 DESIGN CONCEPTS

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.

Chapter 12 DESIGN CONCEPTS

- **A design should be represented using a notation that effectively communicates its meaning.**

These design guidelines are not achieved by chance.

Quality Attributes. Hewlett-Packard 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.**

Chapter 12 DESIGN CONCEPTS

- ***Usability*** **is assessed by considering human factors (Chapters 6 and 15),overall aesthetics, consistency, and documentation.**
- ***Reliability*** **is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.**
- ***Performance*** **is measured using processing speed, response time, resource consumption, throughput, and efficiency.**
- ***Supportability*** **combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, maintainability —and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 29), the ease with which a system can be installed, and the ease with which problems can be localized.**

Chapter 12 DESIGN CONCEPTS

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

12.2.2 The Evolution of Software Design(read by yourselves)

The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down "structured" manner. Newer design approaches proposed an

Chapter 12 DESIGN CONCEPTS

object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods,model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

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.

Chapter 12 DESIGN CONCEPTS

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.

12.3 DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time.

In the sections that follow, we present an overview of fundamental software design concepts that provide the necessary framework for "getting it right."

12.3.1 Abstraction

Chapter 12 DESIGN CONCEPTS

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.

Give an example for the abstraction.

12.3.2 Architecture

Software architecture **alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". 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.**

Chapter 12 DESIGN CONCEPTS

A set of architectural patterns enables a software engineer to reuse design-level concepts.

12.3.3 Patterns

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.

12.3.4 Separation of Concerns

Separation of concerns is a design concept 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.

Chapter 12 DESIGN CONCEPTS

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

12.3.5 Modularity

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

Let C(x) be a function that defines the perceived complexity of a problem x, and E(x) be a function that defines the effort (in time) required to solve a problem x. For two problems, p 1 and p 2 , if

C(p 1) > C(p 2) it follows that E(p 1) > E(p 2) .

Chapter 12 DESIGN CONCEPTS

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,C(p 1 + p 2) > C(p 1) + C(p 2) , it follows that, E(p 1 + p 2) > E(p 1) + E(p 2) .

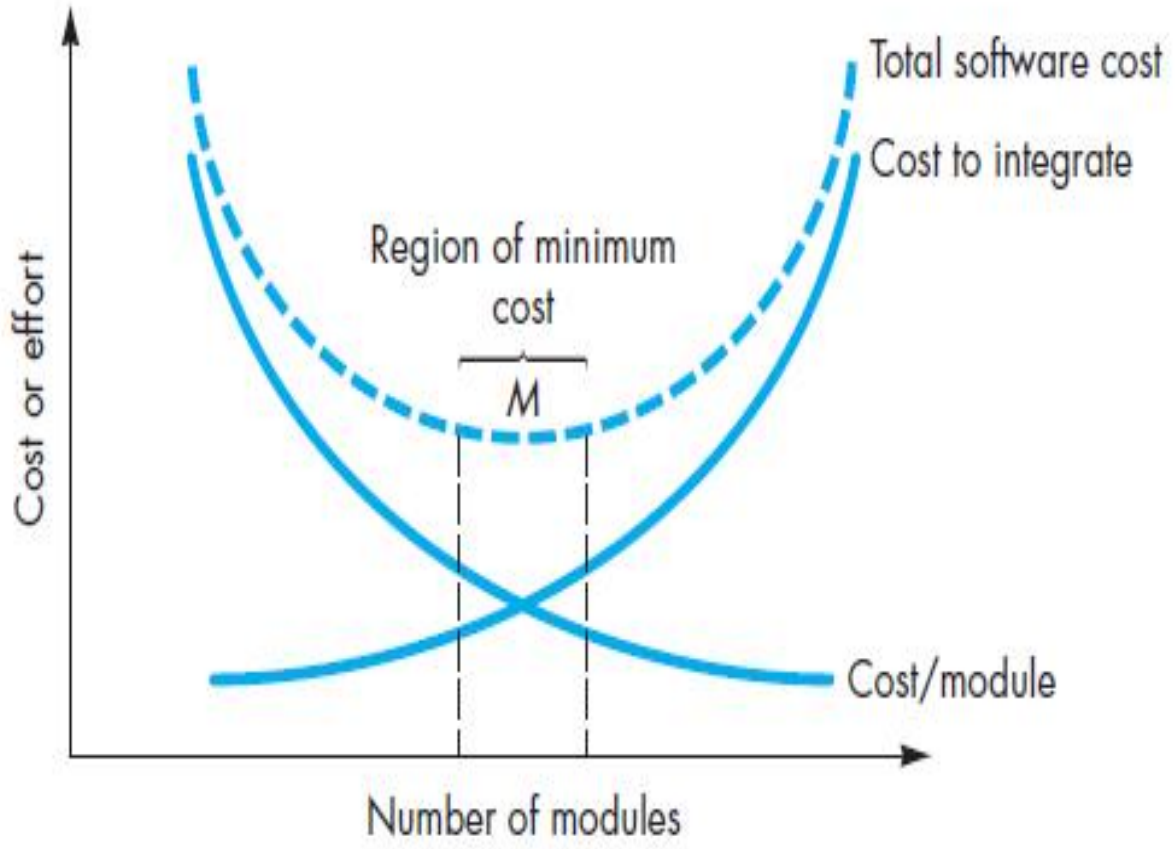
Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort(cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. In the figure12.2,there is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

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.

Chapter 12 DESIGN CONCEPTS

FIGURE 12.2

Modularity and software cost



Chapter 12 DESIGN CONCEPTS

12.3.6 Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding 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.

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.

Chapter 12 DESIGN CONCEPTS

12.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

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.

high cohesion,

low coupling.

12.3.8 Refinement

Refinement is actually a process of elaboration.You begin with a statement of function (or description of information) that is defined

Chapter 12 DESIGN CONCEPTS

at a high level of abstraction. You then elaborate on the original statement,providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts.

12.3.9 (Design) aspects(read by yourselves)

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”.

An aspect is a representation of a crosscutting concern. an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout

many components. 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.（for example:interface class may be as an aspect）

12.3.10 Refactoring

An important design activity suggested for many agile methods (Chapter 5), refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.

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.

Chapter 12 DESIGN CONCEPTS

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.

12.3.11 Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. we have learned these comcepts in other courses.

12.3.12 Design Classes

The analysis model defines a set of analysis classes (Chapter 10).

Chapter 12 DESIGN CONCEPTS

Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed:

Business domain classes identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.

Chapter 12 DESIGN CONCEPTS

User interface classes **define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor.**

Process classes (interface or control class) **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(如，Java的主类，有Main()方法) **implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.**

As the architecture forms, the level of abstraction is reduced as each analysis class (Chapter 10) is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon

of the business domain. Design classes present significantly more technical detail as a guide for implementation.

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

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class.

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.

High cohesion. A cohesive design class has a small, focused set of

responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.

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, suggests that a method should only send messages to methods in neighboring classes.

Chapter 12 DESIGN CONCEPTS

12.3.13 Dependency Inversion(see appendix 2,read by yourselves)

The structure of many older software architectures is hierarchical. At the top of the architecture, “control” components rely on lower level “worker” components to perform various cohesive tasks.

In object-oriented software engineering, abstractions are implemented as abstract classes,This approach reduces coupling and improves the testability and maintainability of a design.

12.3.14 Design for Test(read by yourselves)

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

Chapter 12 DESIGN CONCEPTS

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

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

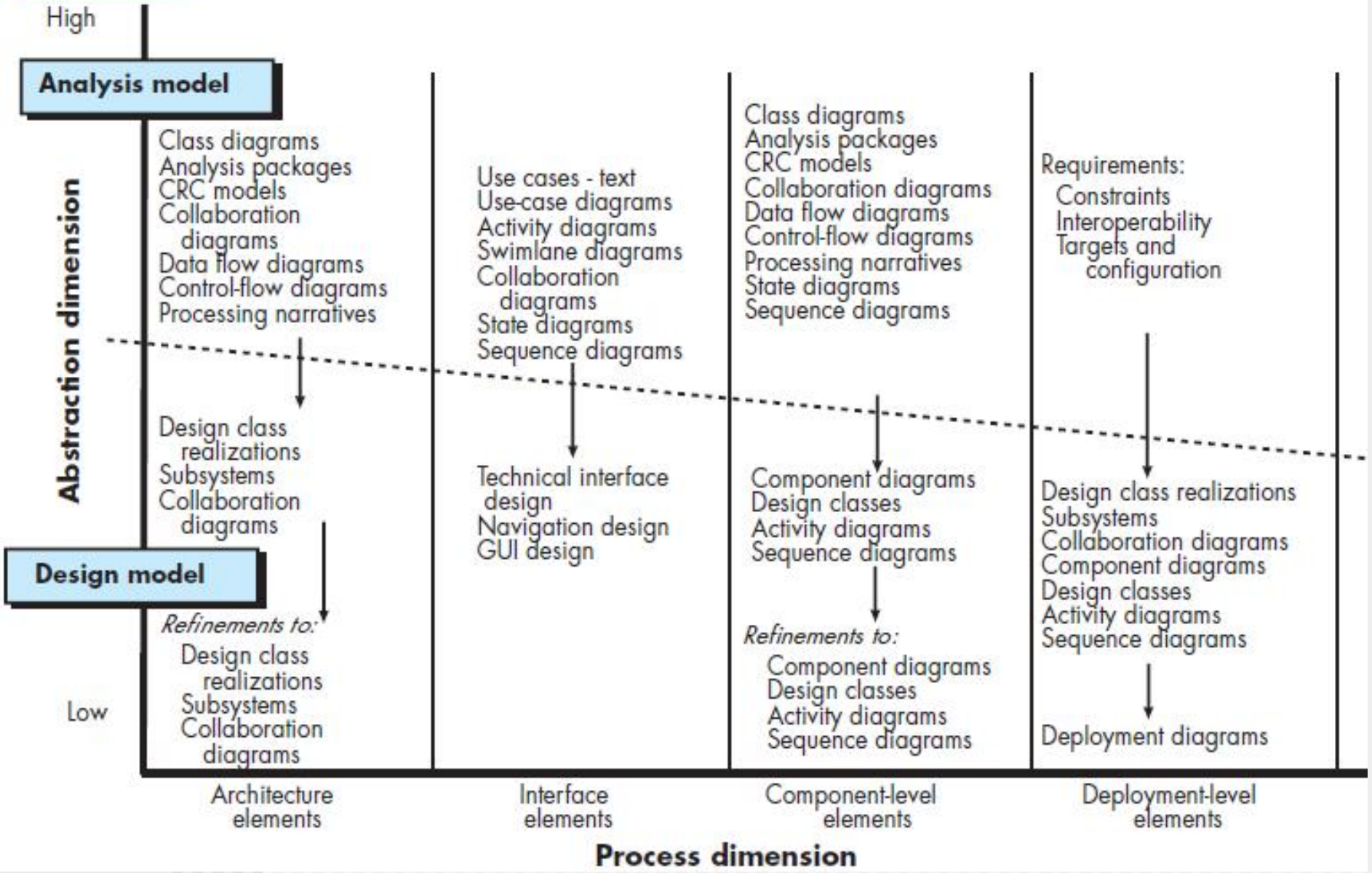
12.4 THE DESIGN MODEL(read by yourselves)

The design model can be viewed in two different dimensions as illustrated in Figure 12.4 . The process dimension indicates the

Chapter 12 DESIGN CONCEPTS

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

FIGURE 12.4 Dimensions of the design model



Chapter 12 DESIGN CONCEPTS

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

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

You can apply design patterns (Chapter 16) at any point during

design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

12.4.1 Data Design Elements(database and data structure,(read by yourselves))

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.

Chapter 12 DESIGN CONCEPTS

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 13.

Chapter 12 DESIGN CONCEPTS

12.4.2 Architectural Design Elements(read by yourselves)

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

Chapter 12 DESIGN CONCEPTS

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 13.

12.4.3 Interface Design Elements(read by yourselves)

The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design: (1) the user interface (UI), (2) external interfaces to other systems, devices, networks, or other producers or consumers of information, and (3) internal interfaces between various design components.

Chapter 12 DESIGN CONCEPTS

UI design (increasingly called usability design) is a major software engineering action.

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 ControlPanel class (Figure 12.5) provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke ()* and *decodeKey ()* . If these operations are to be provided to other classes (in this case, Tablet and SmartPhone), it is useful to define an interface as shown in the

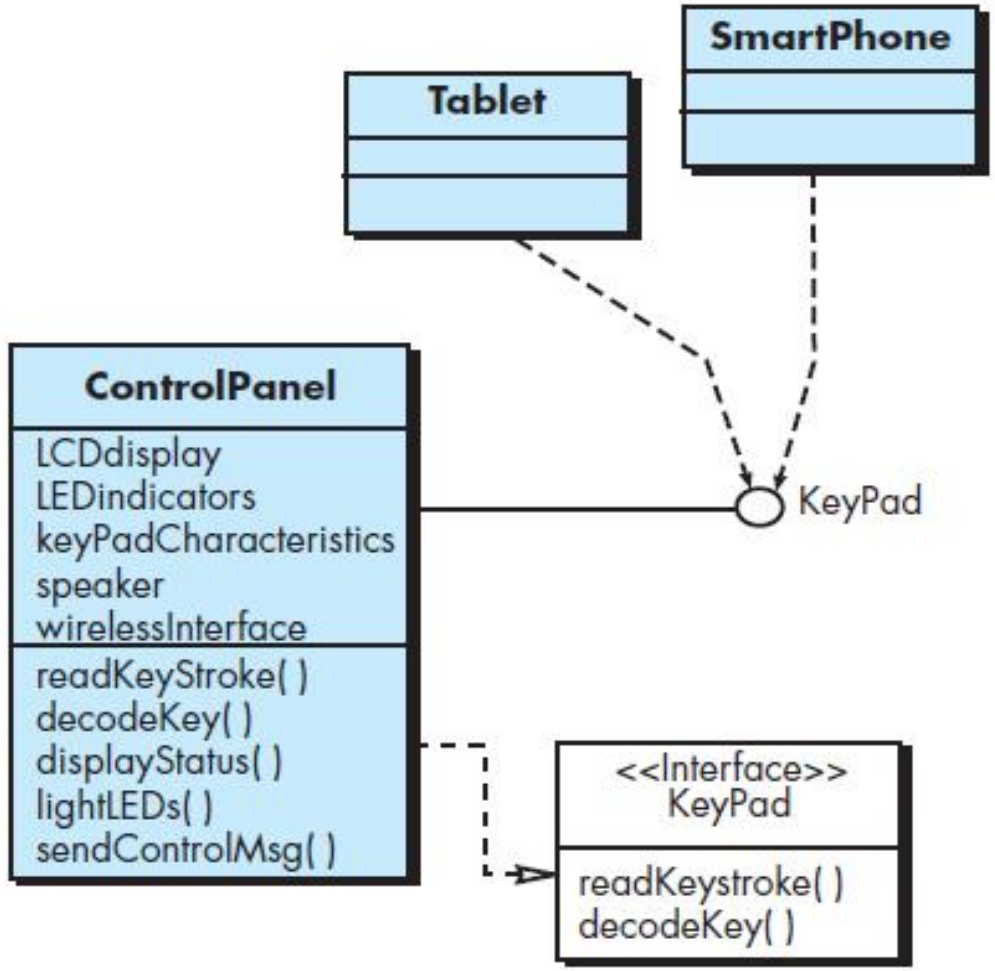
figure. The interface, named KeyPad , is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

The dashed line with an open triangle at its end (Figure 12.5) indicates that the ControlPanel class provides KeyPad operations as part of its behavior. In UML, this is characterized as a realization. That is, part of the behavior of ControlPanel will be implemented by realizing KeyPad operations. These operations will be provided to other classes that access the interface.

Chapter 12 DESIGN CONCEPTS

FIGURE 12.5

Interface representation for ControlPanel



Chapter 12 DESIGN CONCEPTS

12.4.4 Component-Level Design Elements(read by yourselves)

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.

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

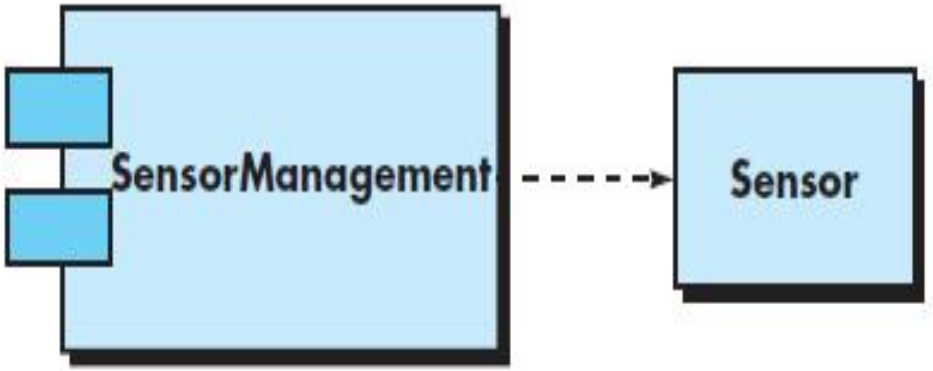
Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 12.6 . In this figure, a component named SensorManagement (part of the SafeHome security function) is

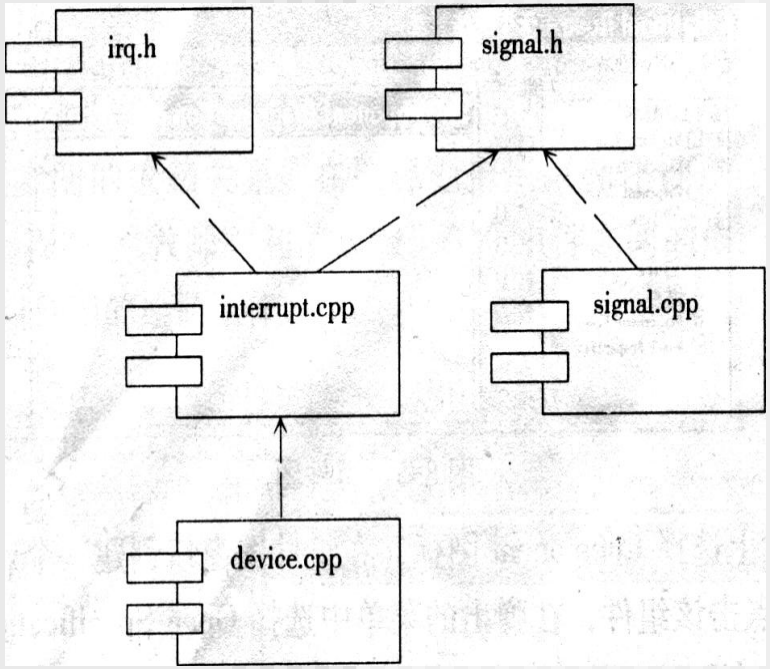
represented. A dashed arrow connects the component to a class named Sensor that is assigned to it. The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them. Further discussion of component diagrams is presented in Chapter 14.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming language like representation described in Chapter 14). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

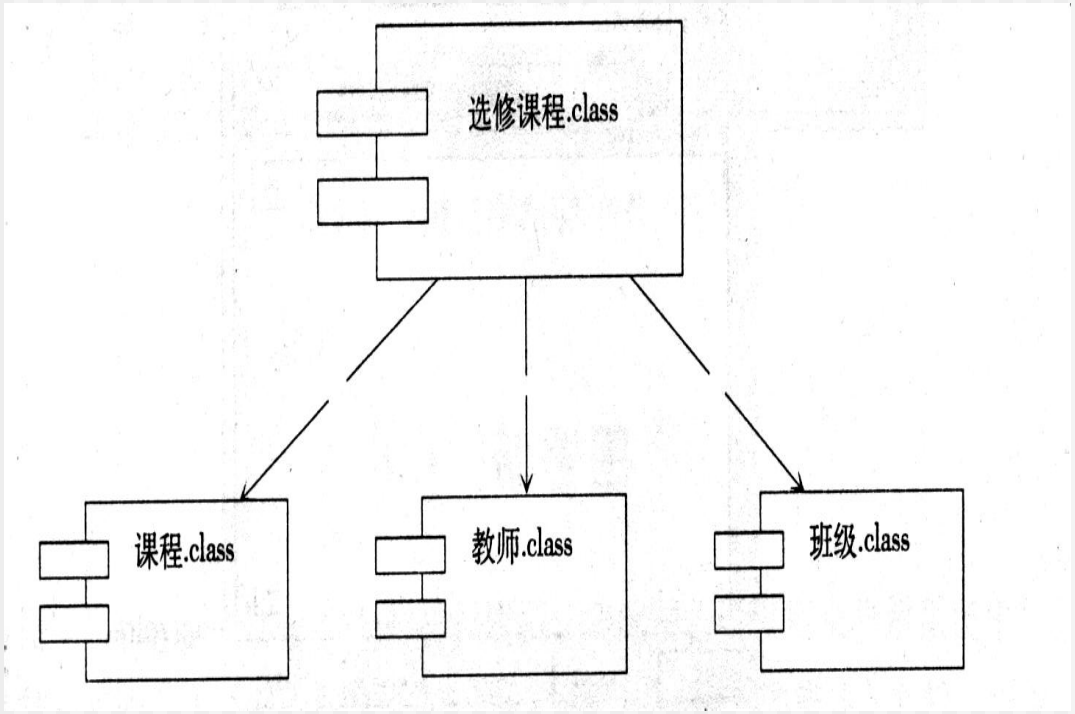
FIGURE 12.6

A UML component diagram





组件图用于对源代码建模



组件图用于对运行系统建模

Chapter 12 DESIGN CONCEPTS

12.4.5 Deployment-Level Design Elements(read by yourselves)

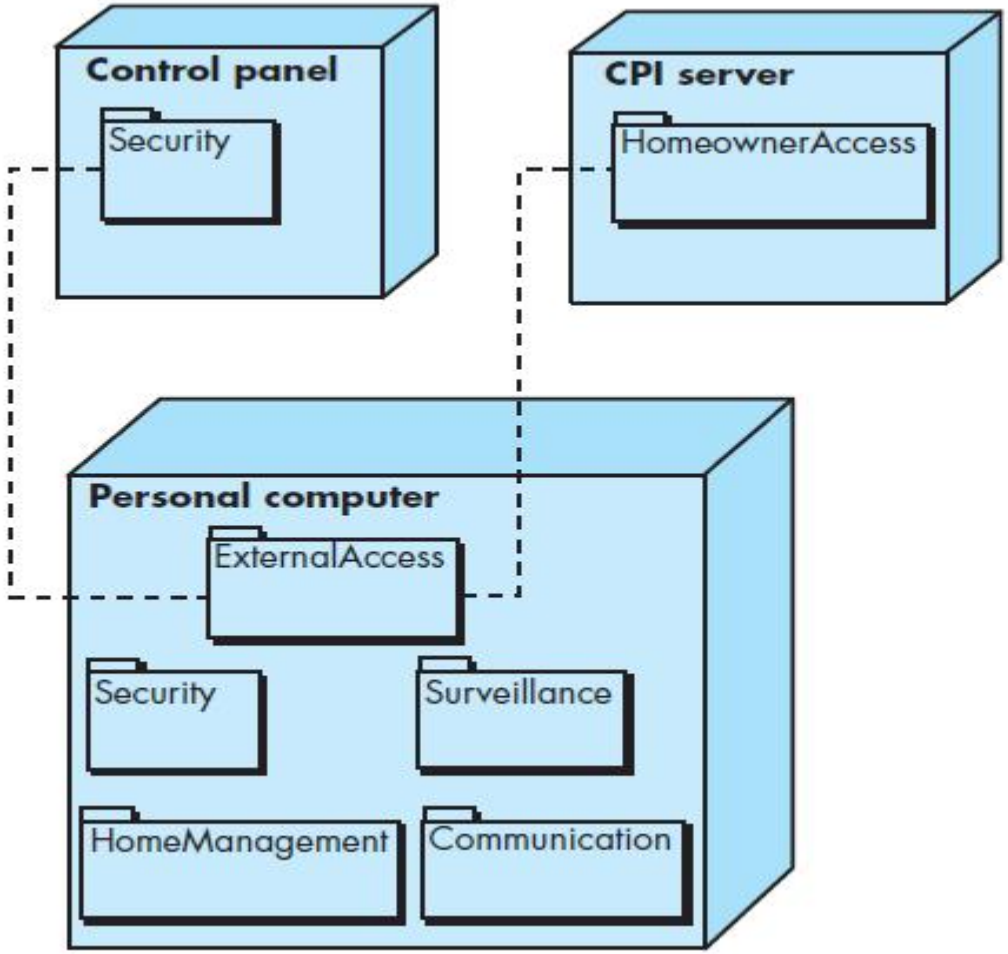
Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the SafeHome product are configured to operate within three primary computing environments—a homebased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

During design, a UML deployment diagram is developed and then refined as shown in Figure 12.7 . In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated.

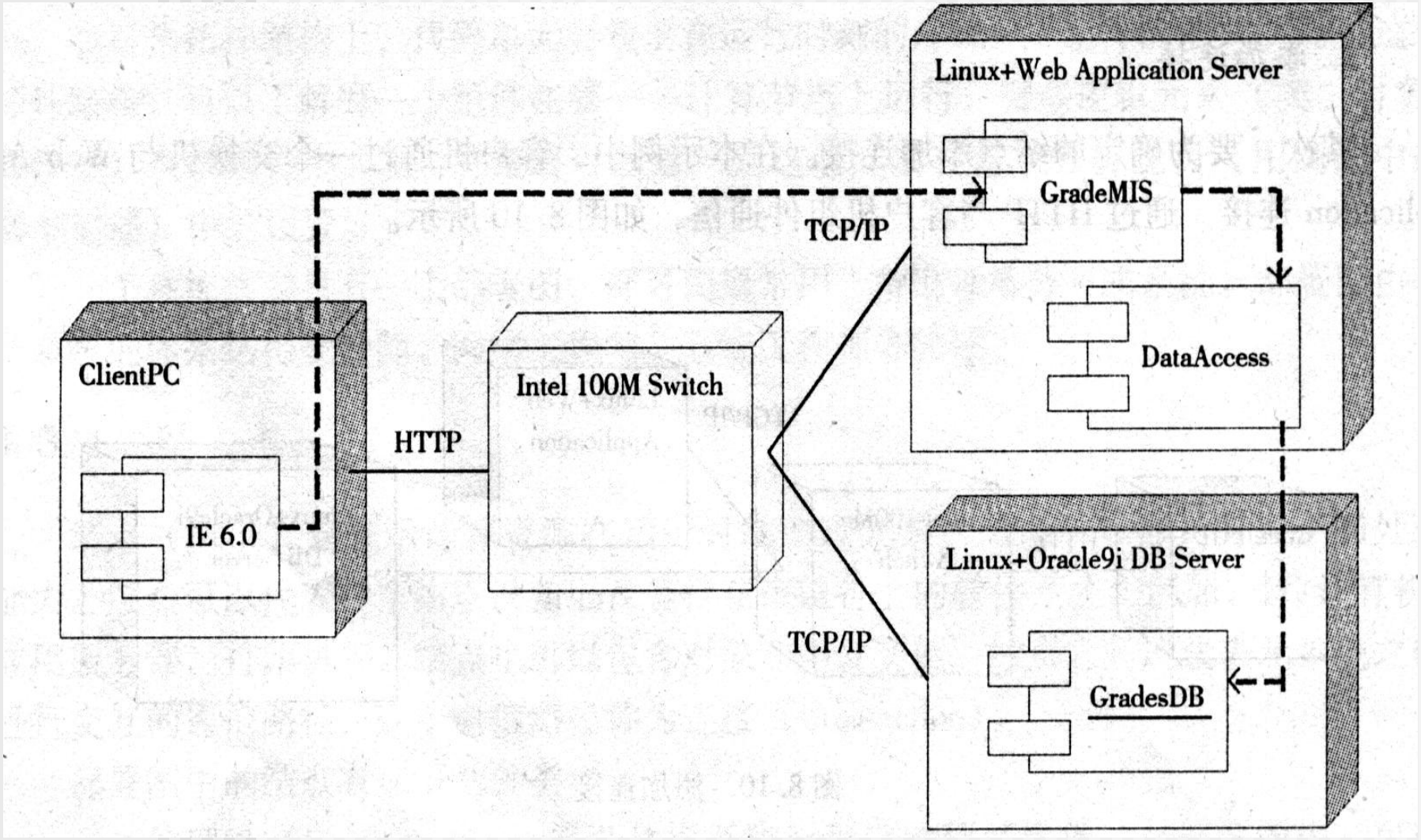
Chapter 12 DESIGN CONCEPTS

FIGURE 12.7

A UML deployment diagram



Chapter 12 DESIGN CONCEPTS



Chapter 12 DESIGN CONCEPTS

The diagram shown in Figure 12.7 is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac, a Windows-based PC, a Linux-box or a mobile platform with its associated operating system.

12.5 SUMMARY

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

Chapter 12 DESIGN CONCEPTS

alternatives and converge on a solution that best suits the needs of project stakeholders.