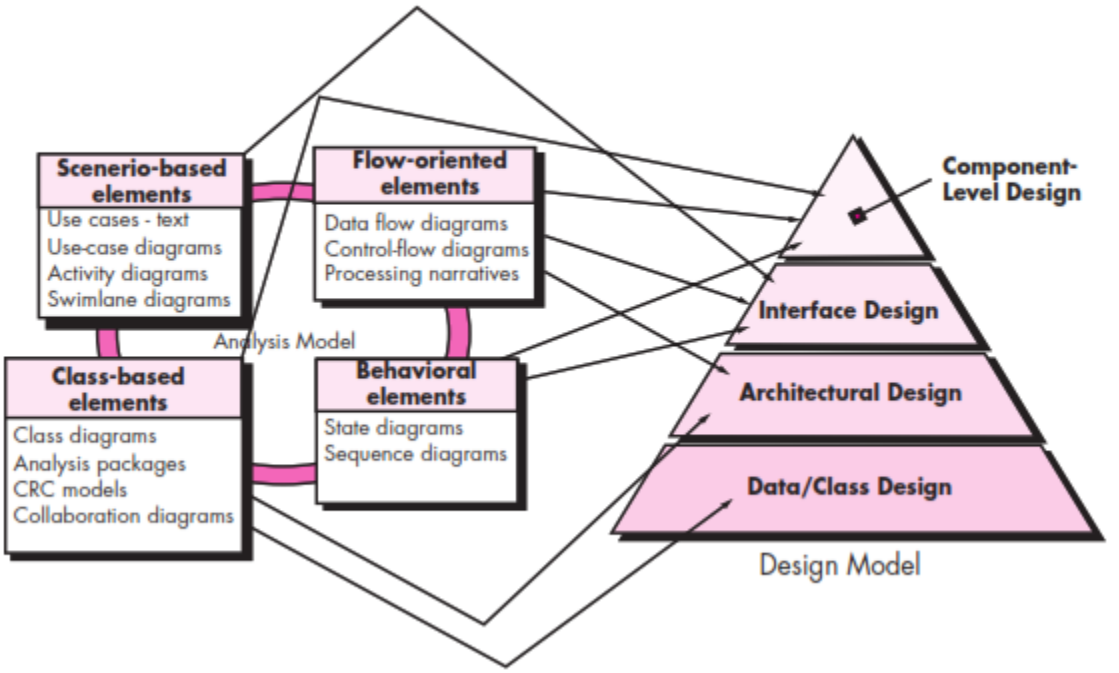
8.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

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)

FIGURE 8.1 Translating the requirements model into the design model



Each of the elements of the requirements model 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 8.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

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

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

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

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

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

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

8.2 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.

8.2.1 Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 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?

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
   i. has been created using recognizable architectural styles or patterns,
   ii. is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and
   iii. 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.
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.

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.

8.i.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 and methods for refining software structures in a topdown manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure, into a design definition. Newer design approaches, proposed an 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.

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.

8.3 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.

8.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create both procedural and data abstractions.

1. A procedural abstraction refers to a sequence of instructions that have a specific and limited function.
2. A data abstraction is a named collection of data that describes a data object.

8.3.2 Architecture

Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

8.3.3 Patterns

Brad Appleton defines a design pattern in the following manner: “A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns” [App00].

8.3.4 Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

8.3.5 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.

8.3.6 Information Hiding

The principle of information hiding [Par72] suggests that modules be “characterized by design decisions that (each) hides from all others.”

8.3.7 Functional Independence

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

8.3.8 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. Refinement is actually a process of elaboration.

8.3.9 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”. An aspect is a representation of a crosscutting concern.

8.3.10 Refactoring

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