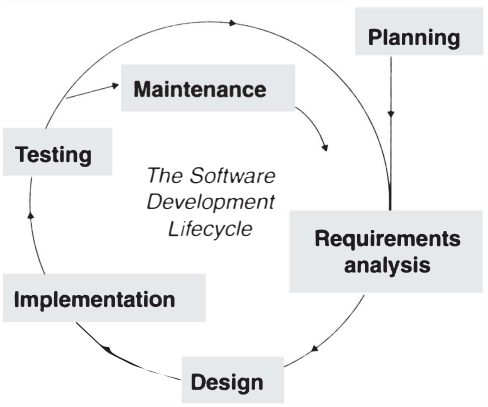
15

Principles of Software Design



- What are the goals of software design?
- How do you use various design models for a single application?
- What are use case model, class models, data flow models, and state models?
- How are frameworks used in design?
- What are the IEEE standards for expressing designs?
- How does a team prepare for design in practice?

Figure 15.1 The context and learning goals for this chapter

A "software design" is a representation, or model, of the software to be built. Its purpose is to enable programmers to implement the requirements by designating the projected parts of the implementation. It is a set of documents containing text and diagrams to serve as the base on which an application can be fully programmed. A complete software design should be so explicit that a programmer could code the application from it without the need for any other documents. Software designs are like the blueprints of a building that are sufficient for a contractor to build the required building. They can be understood in two parts: high-level design, often referred to as "software architecture," which is generally indispensable, and all other design, referred to as "detailed design." It can be beneficial to make designs very detailed, short of being actual code. This is because engineers can examine a detailed design for defects and improvements prior to the creation of code rather than examining

only the code. The benefits of a fully detailed design are balanced against the time required to document and maintain detailed designs. For large efforts, levels in between high level and detailed design may be identified.

This chapter introduces the concepts, needs, and terminology of software design. It sets the stage for the remaining chapters in this part of the book, which include various concrete examples.

15.1 THE GOALS OF SOFTWARE DESIGN

The first goal of a software design is to be *sufficient* for satisfying the requirements. Usually, software designs must also anticipate changes in the requirements, and so a second goal is *flexibility*. Another goal of software design is *robustness*: the ability of the product to anticipate a broad variety of input. These and other goals are summarized in Figure 15.2.

These goals sometimes oppose one another. For example, to make a design efficient it may be necessary to combine modules in ways that limit flexibility. In fact, we trade off goals against each other in ways that depend on the project's priorities.

A software design is *sufficient* if it provides the components for an implementation that satisfies the requirements. To assess such sufficiency, one needs to be able to understand it. This fact is obvious, but it has profound consequences. It can be difficult to create an understandable design for applications due to the large number of options that are typically available. OpenOffice, for example, is a very complex application when viewed in complete detail. Yet OpenOffice is simple when viewed at a high level, as consisting of a few subapplications: word processing, spreadsheet, presentations, and database.

Modularity is thus a key to understandability. Software is modular when it is divided into separately named and addressable components. Modular software is much easier to understand than monolithic software, and parts can be replaced without affecting other parts. It is easier to plan, develop, modify, document, and test. When software is modular you can more easily assign different people to work on different parts.

A design is a form of communication. In its most elementary form, it documents the result of a designer's thought process, and is used to communicate back to himself thereafter when he needs to know what he designed. This is fine if the designer is to be the only person who has this need, but a project usually involves several people throughout its lifetime. If a design is not *understandable* for them, it is of limited value, and the project's health is at risk. Design simplification, in particular, frequently results in a better design. Understandability is usually achieved by organizing the design as a progression from a high level with a manageable number of parts, then increasing the detail on the parts.

A good software architect and designer forms a clear mental model of how the application will work at an overall level, then develops a decomposition to match this mental model. She first asks the key modularity

- ***Sufficiency:*** handles the requirements
- ***Understandability:*** can be understood by intended audience
- ***Modularity:*** divided into well-defined parts
- ***Cohesion:*** organized so like-minded elements are grouped together
- ***Coupling:*** organized to minimize dependence between elements
- ***Robustness:*** can deal with wide variety of input
- ***Flexibility:*** can be readily modified to handle changes in requirements
- ***Reusability:*** can use parts of the design and implementation in other applications
- ***Information hiding:*** module internals are hidden from others
- ***Efficiency:*** executes within acceptable time and space limits
- ***Reliability:*** executes with acceptable failure rate

Figure 15.2 Principal goals of software design

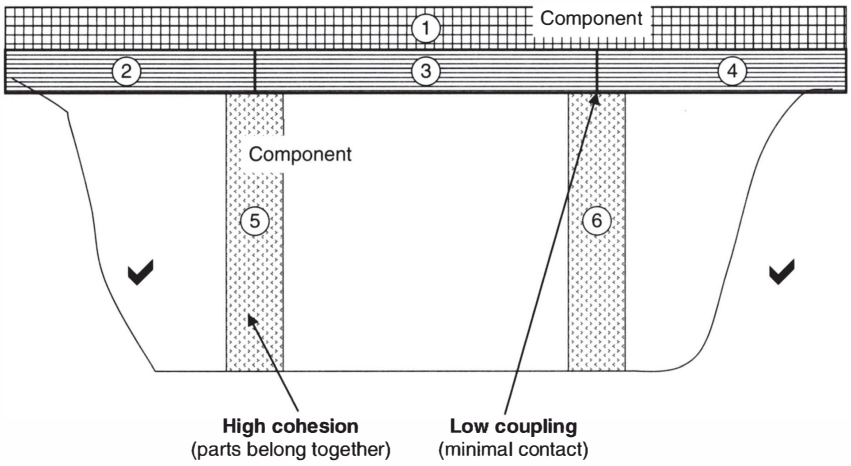


Figure 15.3 High cohesion and low coupling—bridge example

question such as: What five or six modules should we use to decompose a personal finance application? What four or five modules neatly encompass a word processing application? After deciding this, she turns to decomposing the components, and so on. This process is sometimes called "recursive design" because it repeats the design process on design components at successively fine scales. Software decomposition itself involves consideration of *cohesion* and *coupling*.

Cohesion within a module is the degree to which the module's elements belong together. In other words, it is a measure of how focused a module is. The idea is not just to divide software into arbitrary parts (i.e., modularity), but to keep related issues in the same part. Coupling describes the degree to which modules communicate with other modules. The higher the degree of coupling, the harder it is to understand and change the system. To modularize effectively, we *maximize cohesion* and *minimize coupling*. This principle helps to decompose complex tasks into simpler ones.

Software engineering uses Unified Modeling Language (UML) as a principal means of explaining design. Understanding software design concepts by means of analogous physical artifacts is helpful to some, and we will employ this means on occasion. Figure 15.3, for example, suggests coupling/cohesion goals by showing an architecture for a bridge, in which each of the six components has a great deal of cohesion and where the coupling between them is low. The parts of each bridge component belong together (e.g., the concrete and the embedded metal reinforcing it)—this is high cohesion. On the other hand, each component depends on just a few other components—two or three, in fact. This is low coupling.

The "Steel truss" in Figure 15.4, on the other hand, shows many components depending on each other at one place. We would question this high degree of coupling.

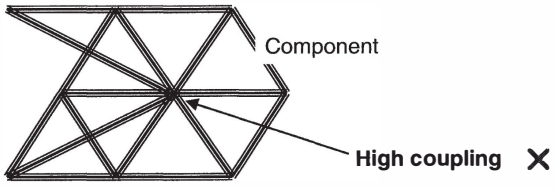


Figure 15.4 A questionable architecture—high coupling in a truss

- *Obtaining more or less of what's already present*
  Example: handle more kinds of accounts without needing to change the existing design or code
- *Adding new kinds of functionality*
  Example: add *withdraw* to existing *deposit* function
- *Changing functionality*
  Example: allow withdrawals to create an overdraft

Figure 15.5 Aspects of flexibility: ways in which an application may be required to change

Low coupling and high cohesion are particularly important for software design because we typically need to modify applications on an ongoing basis. Compare the life cycle of a typical software application with that of the bridge in Figure 15.3: the likelihood that the software will require modification is many times greater. Low coupled/high cohesion architectures are far easier to modify, since they tend to minimize the effects of changes.

The number of top-level packages in an architecture should be small so that people can comprehend the result. A range of "7 ± 2" is a useful guideline, although specific projects can vary greatly from this range for special reasons. As an example, OpenOffice would be hard to understand if we decomposed it into 100 parts instead of four as follows: "word processing, spreadsheet, presentations, and database." The mind has much trouble thinking of 100 separate things at more or less the same time. The difference between small- and large-scale projects (such as OpenOffice) is the amount of nesting of modules or packages. Projects typically decompose each top-level package into subpackages, these into subsubpackages, and so on. The "7 ± 2" guideline applies to each of these decompositions.

A design or implementation is *robust* if it is able to handle miscellaneous and unusual conditions. These include bad data, user error, programmer error, and environmental conditions. A design for the video store application is robust if it deals with attempts to enter DVDs with wrong or inconsistent information, or customers who don't exist or who have unusually long names, or if it handles late rental situations of all kinds. Robustness is an important consideration for applications that must handle communication.

The requirements of an application can change in many ways, and as a result a design must be *flexible* to accommodate these changes. Figure 15.5 illustrates ways in which an application may change.

A set of previously used *design patterns* is a useful resource for flexible designs. Design patterns are discussed in Chapter 17. We design so that parts of our own applications can be *reused* by others and ourselves. Figure 15.6 lists the types of artifacts that often can be reused.

We can reuse . . . .

- **Object code** (or equivalent)
  Example: sharing dlls between word processor and spreadsheet
- **Classes**—in source code form
  Example: Customer class used by several applications
  Thus, we write generic code whenever possible
- **Assemblies of related classes**
  Example: the java.awt package
- **Patterns** of class assemblies

Figure 15.6 Types of reuse

As an example of class reuse, consider the class *DVDRental* that associates the DVD and the customer renting it. Such a class is reusable only in another application dealing with the rental of DVDs, which is limited. If, however, we design a *Rental* class dealing with the rental of an *Item* to a *Customer*, and if *DVDRental* inherits from *Rental*, then we would be able to reuse the *Rental* portion for other applications due to its generality. Design patterns facilitate the reuse of assemblies of related classes rather than individual ones.

Information hiding is a design principle in which the internals of a module are deliberately not usable by code that does not need to know the details. This is supported in object-oriented languages by declaring a public interface through which user code accesses the objects of a class. Private methods and attributes are only accessible to the objects of the class itself. Information hiding allows the internals of a module to be modified without the users of the module having to change. It also reduces complexity because the module interface is fixed and well defined; using code need not be concerned with internal details.

Efficiency refers to the use of available machine cycles and memory. We create designs and implementations that are as fast as required, and that make use of no more than the required amount of RAM and disk. Efficient designs are often achieved in stages as follows. First, a design is conceived without regard to efficiency; efficiency bottlenecks are then identified, and finally the original design is modified. As an example, consider the use of maps on the Web. When the user browses a map and wants to move to an adjacent area, early algorithms required a separate page fetch. Algorithms were introduced, however, to automatically fetch appropriate adjacent maps after the initial fetch, improving efficiency and, with it, the user's experience. It is ideal if efficiency is considered at the time of initial design, however.

It is unrealistic to expect that a real-world application be 100 percent defect-free. An application is *reliable* if it falls within a predetermined standard of fault occurrence. Metrics make this precise, such as the average time between failures. Reliability is related to but different from robustness. Robustness is mostly a design issue because one must specifically accommodate erroneous data in advance—accommodating it does not happen by accident or experience. On the other hand, reliability is mostly a process issue, requiring thorough inspection and testing of artifacts. Design affects reliability in that clean designs make it easier for developers to produce error-free applications. However, a wide range of other factors make for reliability, because an application can fail for a wide variety of reasons.

15.2 INTEGRATING DESIGN MODELS

The architectural drawings of an office building comprise the front elevation, the side elevation, the electrical plan, the plumbing plan, and so on. In other words, several different views are required to express a building's architecture. Similarly, several different views are required to express a software design. They are called *models*.

Four important models are shown in Figure 15.7, and they are explained next. Several ideas here are taken from the Unified Software Development method of Booch, Jacobson, and Rumbaugh. Figure 15.7 includes an example from the Encounter video game case study to illustrate the four models and how they fit together.

The subsequent sections of this chapter elaborate on these models so that they can be contrasted and combined with each other. Subsequent chapters in this part of the book provide detailed examples of each.

15.2.1 Use Case Model

This section describes several levels of use cases and related concepts. The *use case model* consists of the following four parts:

1. The *business use cases*: a narrative form, suitable for developer-to-customer communication, describing the required basic sequences of actions

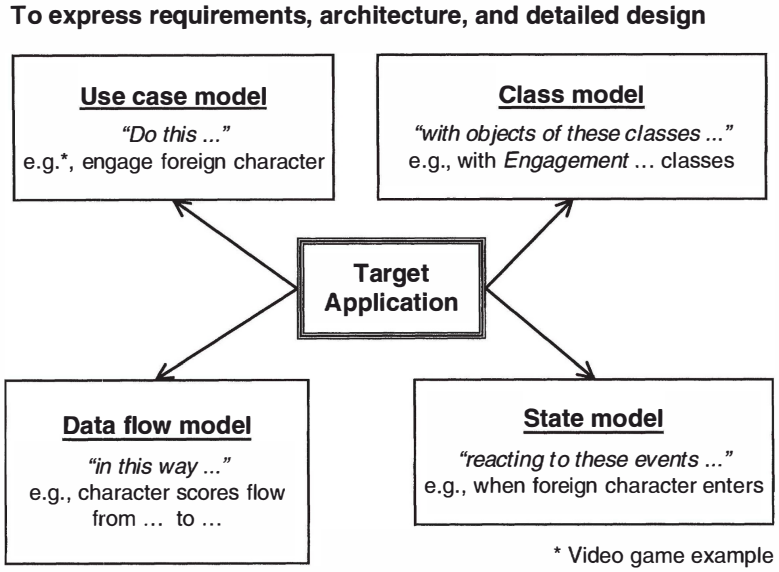


Figure 15.7 Models to express designs (and parts of requirements)

2. Their refinement into regular *use cases*, described in Chapter 11
3. Their transformation into *sequence diagrams* (covered in Chapter 16), which in turn can be in two successive stages of refinement:
   (a) With informal functionality descriptions, at first lacking all details
   (b) With specific function names and showing all details
4. *Scenarios*: instances of use cases that contain specifics, and that can be used for testing. For example, a scenario of the use case step

   Customer chooses account

   would be something like

   John Q. Smith chooses checking account 12345.

The use case model expresses what the application is supposed to do, as suggested in Figure 15.8.

15.2.2 Class Models

Classes are the building blocks—more precisely, the types of building blocks—of designs. Class models consist of *packages* (a grouping of classes) that decompose into smaller packages, and so on, which decompose into classes, and these, in turn, decompose primarily into methods. This is shown in Figure 15.9. We cover classes in more detail in Chapter 16.

15.2.3 Data Flow Models

The class model describes the *kinds* of objects involved; it does not show actual objects. The *data flow model*, on the other hand, shows specific objects and the types of data flowing between them. It is related to the class

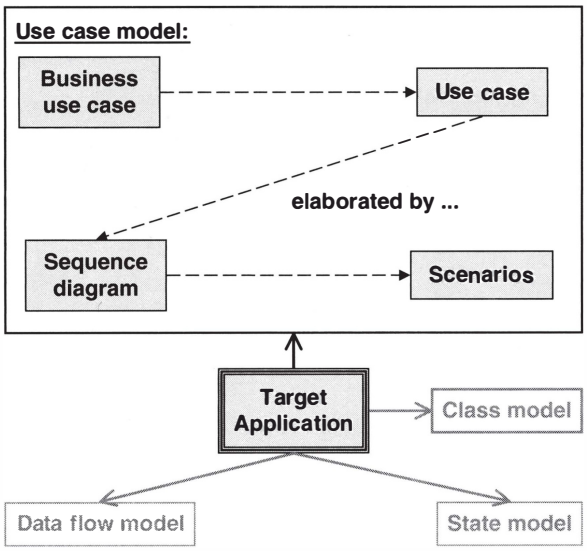


Figure 15.8 The role of use case models in design

model, because the objects involved must belong to the classes in the class model. We discuss data flow diagrams in Chapter 16. Figure 15.10 shows the parts of a data flow model.

15.2.4 State Models

State models reflect reactions to events. Events include mouse actions and changes in variable values. Events are not described in class models or component models, but they do occur in the state model. We introduced states in Chapter 11 as one way to describe the requirements of an application. The states in that context describe the external behavior of the application. The states we are referring to here reflect the state of

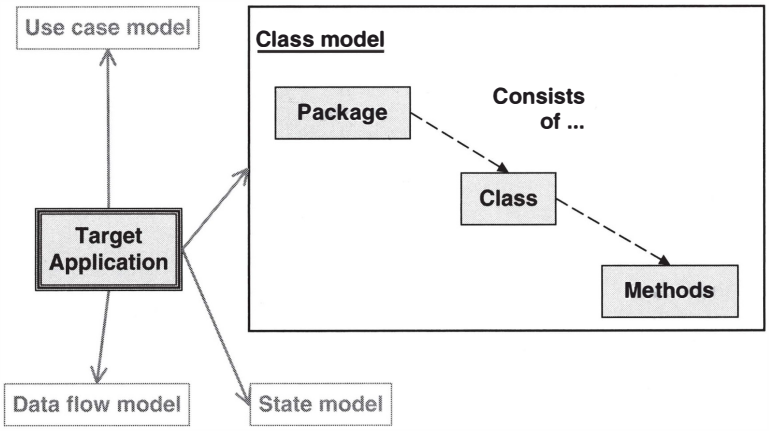


Figure 15.9 The role of class models in design

Source: Jacobson, Ivar, "Object Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley, 1992.

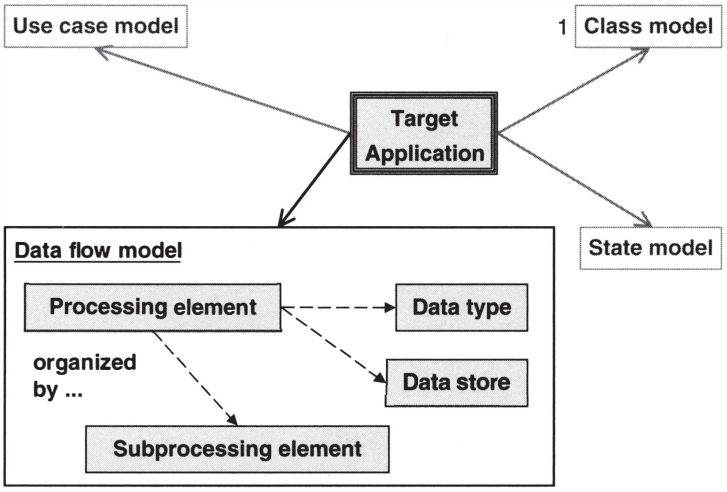


Figure 15.10 The role of component models in design

Source: Jacobson, Ivar, "Object Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley, 1992.

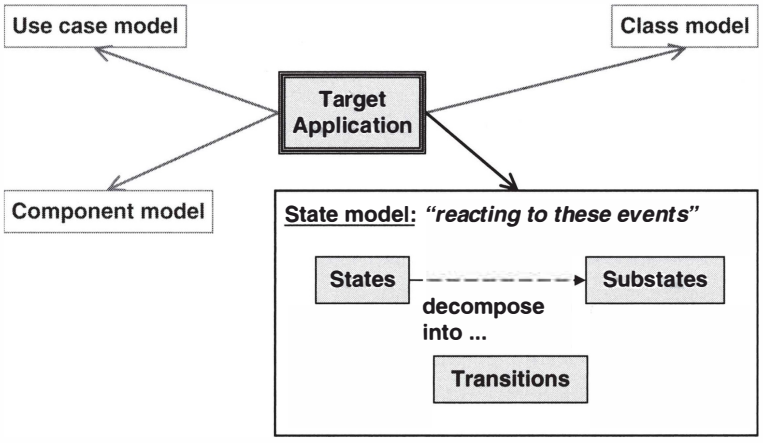


Figure 15.11 The role of state models in design

Source: Jacobson, Ivar, "Object Oriented Software Engineering: A Use Case Driven Approach," Addison-Wesley, 1992.

elements in a software design. The role of state models is shown in Figure 15.11. They are described in more detail in Chapter 16.

15.3 FRAMEWORKS

As we have seen, the *reuse* of components is a major goal in software development. If an organization can't leverage its investments in the skill of its designers and programmers by using their work several times over, competitors who do so will be faster to market with superior products. The parts of an application that are particular to it and are not reused are often called its *business logic*. Business classes are essentially the domain classes discussed in Chapter 12.

Where do we keep the classes slated for reuse? How do we organize them? Should we build in relationships among these classes? Do they control the application or does the application control them? The computing

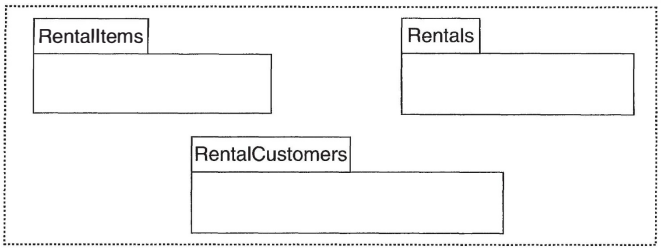


Figure 15.12 A framework for rental applications

community has learned from experience that merely making a list of available functionality does not necessarily result in reuse. We have learned, however, that arrangements like the Java API (coherent sets of classes) do indeed lend themselves to highly successful reuse. The Java APIs (*3D*, *2D*, *Swing*, etc.) are *frameworks*.

A *framework*, sometimes called a *library*, is a collection of software artifacts usable by several different applications. These artifacts are typically implemented as classes, together with the software required to utilize them. A framework is a kind of common denominator for a family of applications. Progressive development organizations designate selected classes as belonging to their framework. Typically, a framework begins to emerge by the time a development organization develops its second to fourth application. As an example, consider the *Rental* framework shown in Figure 15.12 that our video store application could use. This architecture divides the elements into the items that can be rented (books, DVDs, etc.), the people who can rent them, and the rental records that associate members from the first two parts.

The individual classes in the framework will be supplied later. Frameworks like these are usually obtained in a combined bottom-up and top-down manner: bottom-up by examining the structure of actual application architectures such as the video store application, seeking more general forms; and top-down by conceptualizing about the needs of a particular family of applications such as rentals.

Classes within a framework may be related. They may be abstract or concrete. Applications may use them by means of inheritance, aggregation, or dependency. Alternatively, as we will see below, a framework may feel like a generic application that we customize by inserting our own parts.

Figure 15.13 shows the relationship between framework classes, domain classes, and the remaining design classes. The design for an application consists of (1) the domain classes (that are special to the

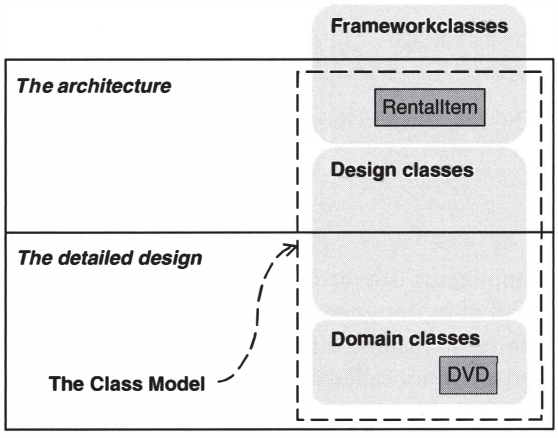


Figure 15.13 Class model vs. architecture and detailed design

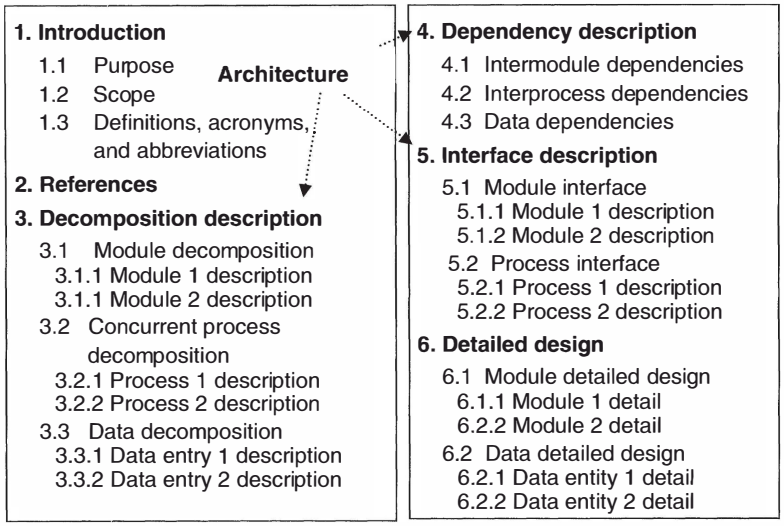


Figure 15.14 IEEE 1016-1998 SDD example table of contents

application), (2) some of the framework classes (generally speaking, not all are needed), and (3) the remaining classes needed to complete the design, which we are calling simply "design" classes. The design classes consist of those required for the architecture and those that are not. The latter are effectively the detail design classes, required to complete the design. These three constitute the class model for the application. All of the domain classes are in the detailed design because they are very specific. The framework classes used are part of the application's architecture.

The framework classes used in the design are part of an application's architecture, as shown in the figure. The domain classes are usually part of the detailed design since they are specific to the application and are not architectural in nature.

15.4 IEEE STANDARDS FOR EXPRESSING DESIGNS

The IEEE Software Design Document (SDD) standard 1016-1998 provides guidelines for the documentation of design. The table of contents is shown in Figure 15.14. IEEE guidelines explain how the SDD could be organized for various architectural styles, most of which are described above. The case study uses the IEEE standard, with a few modifications, to account for an emphasis on the object-oriented perspective. As shown in Figure 15.14, Sections 1 through 5 can be considered software architecture, and Section 6 can be considered the detailed design, to be covered in the next chapter.

15.5 SUMMARY

Software design is a model of the intended software application, as specified by its requirements. A software design is analogous to the blueprints of a house.

Good software designs should exhibit the following characteristics: sufficiency, understandability, modularity, high cohesion, low coupling, robustness, flexibility, reusability, information hiding, efficiency, and reliability.

When expressing a software design, it is helpful to use several different connected views, or models. A use case model describes what the application is intended to do from the point of view of the user. Sequence

diagrams are derived from use cases and describe objects and the sequence of methods calls between them. Class models are a static representation of the classes of the design and the relationship between them. A data flow model shows specific objects and the types of data flowing between them. A state model shows design elements and their reaction to events.

Frameworks are collections of reusable software that implements a general solution to a general problem. For example, GUIs are often designed via frameworks so that new applications do not have to rewrite code to implement a GUI. They can instead leverage the existing GUI code and just add their own customization.

15.6 EXERCISES

1. Write a paragraph describing what a "software design" is, and why it is important.
2. In your own words, define the goals of software design and explain why each goal is important.
3. In your own words, define the following terms: *modularity*, *cohesion*, and *coupling*. Why is each a desirable property of software designs?
4. Can a design be cohesive and exhibit a high degree of coupling? Explain your answer and provide an example.
5. How might coupling and reusability be related? How might cohesion and reusability be related? Explain your answer and provide one example for each.
6. In your own words, explain what is meant by *robustness*. Below is code for a method *divide()*. Make the method more robust in at least two ways.

```
public double divide( Double aNumerator, Double aDenominator )
{
  return aNumerator.doubleValue() / aDenominator.doubleValue();
}
```

7. Using the Internet, research one of the many Java API frameworks (e.g., Swing, JMF, 2D, 3D). In a few paragraphs, describe the design of the framework and how it accommodates reuse.
8. Provide a modularization for an application that advises clients on stock picks, and enables them to transfer funds among various stocks and savings accounts. Explain your solution. *Hint:* One reasonable solution employs four packages.