? What characteristics are common to all design methods?

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

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

TASK SET

Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   Be certain that each subsystem is functionally cohesive.
   Design subsystem interfaces.
   Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   Translate analysis class description into a design class.
   Check each design class against design criteria; consider inheritance issues.
   Define methods and messages associated with each design class.
   Evaluate and select design patterns for a design class or a subsystem.
   Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   Review results of task analysis.
   Specify action sequence based on user scenarios.
   Create behavioral model of the interface.
   Define interface objects, control mechanisms.
   Review the interface design and revise as required.
7. Conduct component-level design.
   Specify all algorithms at a relatively low level of abstraction.
   Refine the interface of each component.
   Define component-level data structures.
   Review each component and correct all errors uncovered.
8. Develop a deployment model.

12.3 Design Concepts

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you define criteria that can be used to partition software into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

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

Quote:

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

Grady Booch

12.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

ADVICE

As a designer, work hard to derive both procedural and data abstractions that serve the problem at hand. If they can serve an entire domain of problems, that's even better.

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

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

WebRef

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

12.3.2 Architecture

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

5 It should be noted, however, that one set of operations can be replaced with another, as long as the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *open* would change dramatically if the door were automatic and attached to a sensor.

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

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

uote:

"A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule, and cost."

Len Bass et al.

Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design. *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics. *Families of related systems* "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

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

uote:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

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

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

12.3.3 Patterns

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

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

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

12.3.4 Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

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

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

12.3.5 Modularity

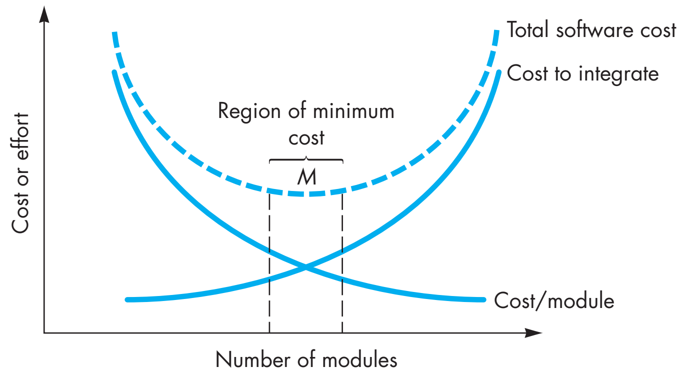
Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules,* that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 12.2, the effort (cost) to develop an individual software module does decrease as the total number of

FIGURE 12.2

Modularity and software cost



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

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

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

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

12.3.6 Information Hiding

KEY POINT

The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

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.

12.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design Wirth [Wir71] and Parnas [Par72] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

? Why should you strive to create independent modules?

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.

KEY POINT

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

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

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

KEY POINT

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

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.

12.3.8 Refinement

ADVICE

There is a tendency to move immediately to full detail, skipping refinement steps. This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement.

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

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

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

12.3.9 Aspects

Quote:

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

Bruce Tognazzini

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

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

KEY POINT

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

For example, consider two requirements for the **www.safehomeassured.com** WebApp. Requirement *A* is described via the ACS-DCV use case discussed in Chapter 9. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **www.safehomeassured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and *B* crosscuts A**.

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

12.3.10 Refactoring

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

An important design activity suggested for many agile methods (Chapter 5), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

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

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

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. As a consequence, refactoring tools [Soa10] are used to analyze changes automatically and to "generate a test suite suitable for detecting behavioral changes."

12.3.11 Object-Oriented Design Concepts

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

SAFEHOME

Design Concepts

The scene: Vinod's cubicle, as design modeling begins.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

The conversation:

[All four team members have just returned from a morning seminar entitled "Applying Basic Design Concepts," offered by a local computer science professor.]

Vinod: Did you get anything out of the seminar?

Ed: Knew most of the stuff, but it's not a bad idea to hear it again, I suppose.

Jamie: When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

Vinod: Because . . . bottom line . . . it's a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

Shakira: I wasn't a CS grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don't see why this stuff is so important.

Jamie: I've seen your work, Shak, and you know what, you do a lot of this stuff naturally . . . that's why your designs and code work.

Shakira (smiling): Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . that sort of thing.

Ed: Modularity, functional independence, hiding, patterns . . . see.

Jamie: I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

Vinod: Same thing can be applied to design, you know.

Jamie: The only concepts I hadn't heard of before were "aspects" and "refactoring."

Shakira: That's used in Extreme Programming, I think she said.

Ed: Yep. It's not a whole lot different than refinement, only you do it after the design or code is completed. Kind of an optimization pass through the software, if you ask me.

Jamie: Let's get back to *SafeHome* design. I think we should put these concepts on our review checklist as we develop the design model for *SafeHome*.

Vinod: I agree. But as important, let's all commit to think about them as we develop the design.

? What types of classes does the designer create?

12.3.12 Design Classes

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

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]. *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor. *Business domain classes* identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes. *Process classes* implement lower-level business abstractions required to fully manage the business domain classes. *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software. *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

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

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

? What is a "well-formed" design class?

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

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

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

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.[6]

SAFEHOME

Refining an Analysis Class into a Design Class

The scene: Ed's cubicle, as design modeling begins.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

[Ed is working on the **FloorPlan** class (see sidebar discussion in Section 10.3 and Figure 10.2) and has refined it for the design model.]

Ed: So you remember the **FloorPlan** class, right? It's used as part of the surveillance and home management functions.

Vinod (nodding): Yeah, I seem to recall that we used it as part of our CRC discussions for home management.

Ed: We did. Anyway, I'm refining it for design. Want to show how we'll actually implement the **FloorPlan** class. My idea is to implement it as a set of linked lists [a specific data structure]. So . . . I had to refine the analysis class **FloorPlan** (Figure 10.2) and actually, sort of simplify it.

Vinod: The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan,** and obviously, there can be many cameras in the floor plan.

Vinod: Phew, let's see a picture of this new **FloorPlan** design class.

[Ed shows Vinod the drawing shown in Figure 12.3.]

Vinod: Okay, I see what you're trying to do. This allows you to modify the floor plan easily because new items can be added to or deleted from the list—the aggregation—without any problems.

Ed (nodding): Yeah, I think it'll work.

Vinod: So do I.

12.3.13 Dependency Inversion

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

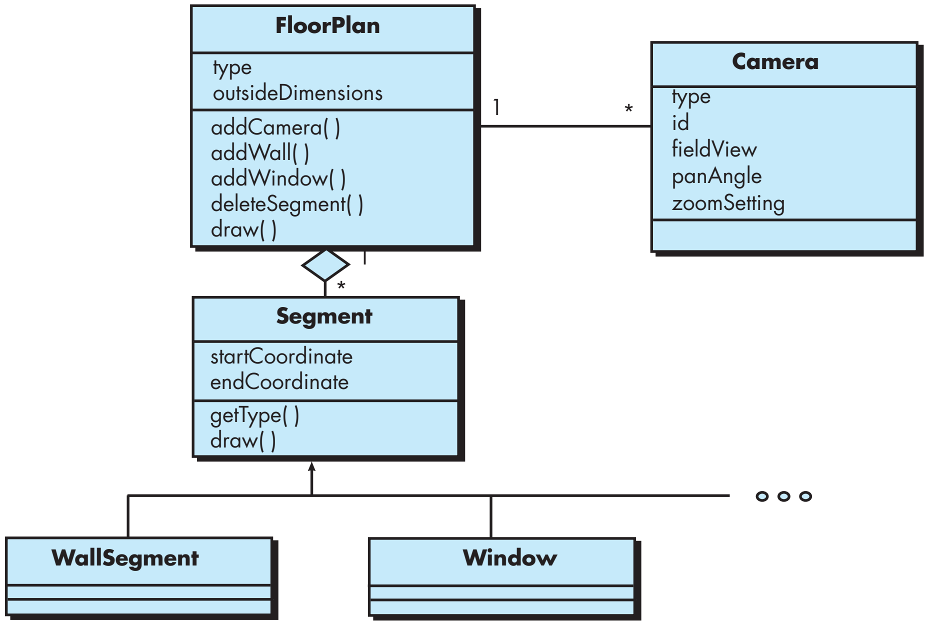
? What is the "dependency inversion principle"?

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

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

FIGURE 12.3

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



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

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

12.3.14 Design for Test

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

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

Quote:

"Test fast, fail fast, adjust fast."

Tom Peters

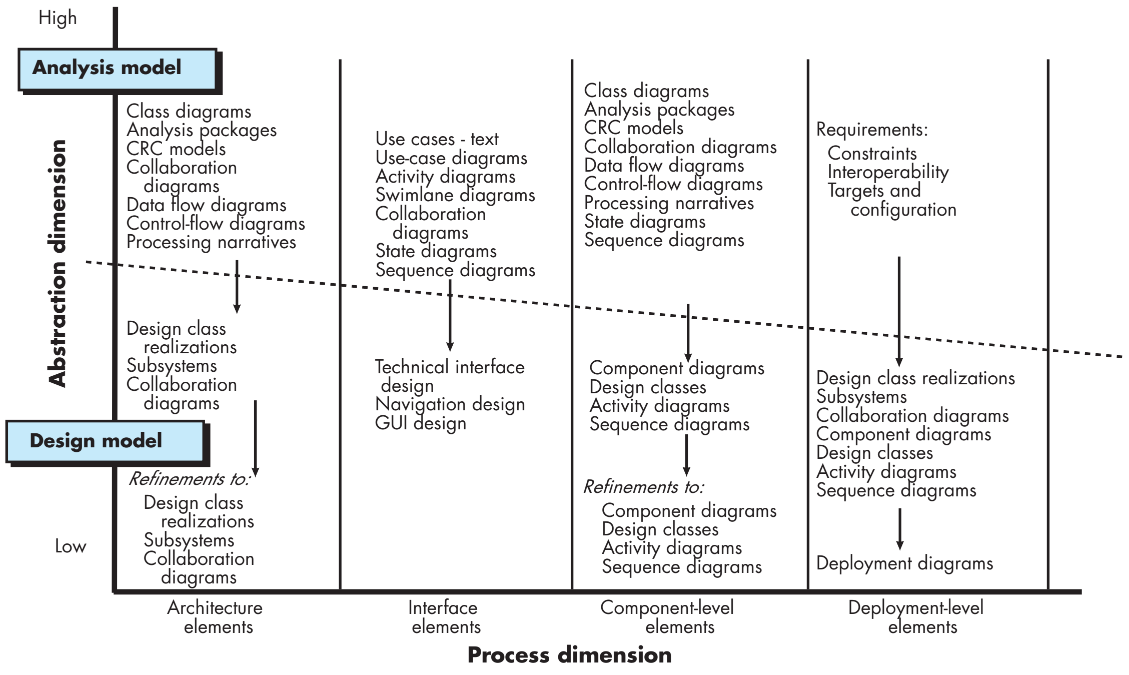
But if design comes first, then the design (and code) must be developed with *seams*—locations in the detailed design where you can "insert test code that probes the state of your running software" and/or "isolate code under test from its production environment so that you can exercise it in a controlled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

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

12.4 The Design Model

The design model can be viewed in two different dimensions as illustrated in Figure 12.4. The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to the figure, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design

Figure 12.4 Dimensions of the design model


High
Analysis model
Abstraction dimension
Design model
Low
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
Design class realizations
Subsystems
Collaboration diagrams
Refinements to:
Design class realizations
Subsystems
Collaboration diagrams
Use cases - text
Use-case diagrams
Activity diagrams
Swimlane diagrams
Collaboration diagrams
State diagrams
Sequence diagrams
Technical interface design
Navigation design
GUI design
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
State diagrams
Sequence diagrams
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Refinements to:
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Requirements:
Constraints
Interoperability
Targets and configuration
Design class realizations
Subsystems
Collaboration diagrams
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Deployment diagrams
Architecture elements
Interface elements
Component-level elements
Deployment-level elements
Process dimension