CHAPTER

11 COMPONENT-LEVEL DESIGN

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. Component-level design bridges the gap between architectural design and coding.

KEY CONCEPTS

QUICK LOOK

What is it? A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it? A software engineer performs component-level design.

Why is it important? You need to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic.

What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct.

Component-level design will reduce the number of errors introduced during coding. As you translate the design model into source code, you should follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

11.1 What Is a Component?

A *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG03a] defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

As we discussed in Chapter 10, components populate the software architecture and play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

The true meaning of the term *component* will differ depending on the point of view of the software engineer who uses it. In the sections that follow, we examine three important views of what a component is and how it is used as design modeling proceeds.

11.1.1 An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes.[1] Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the analysis model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component). To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived.

The attributes and operations defined during analysis are noted at the top of Figure 11.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation[2] shown in the middle right of the figure. Note that **PrintJob** has two interfaces, *computeJob,*

1 In some cases, a component may contain a single class.

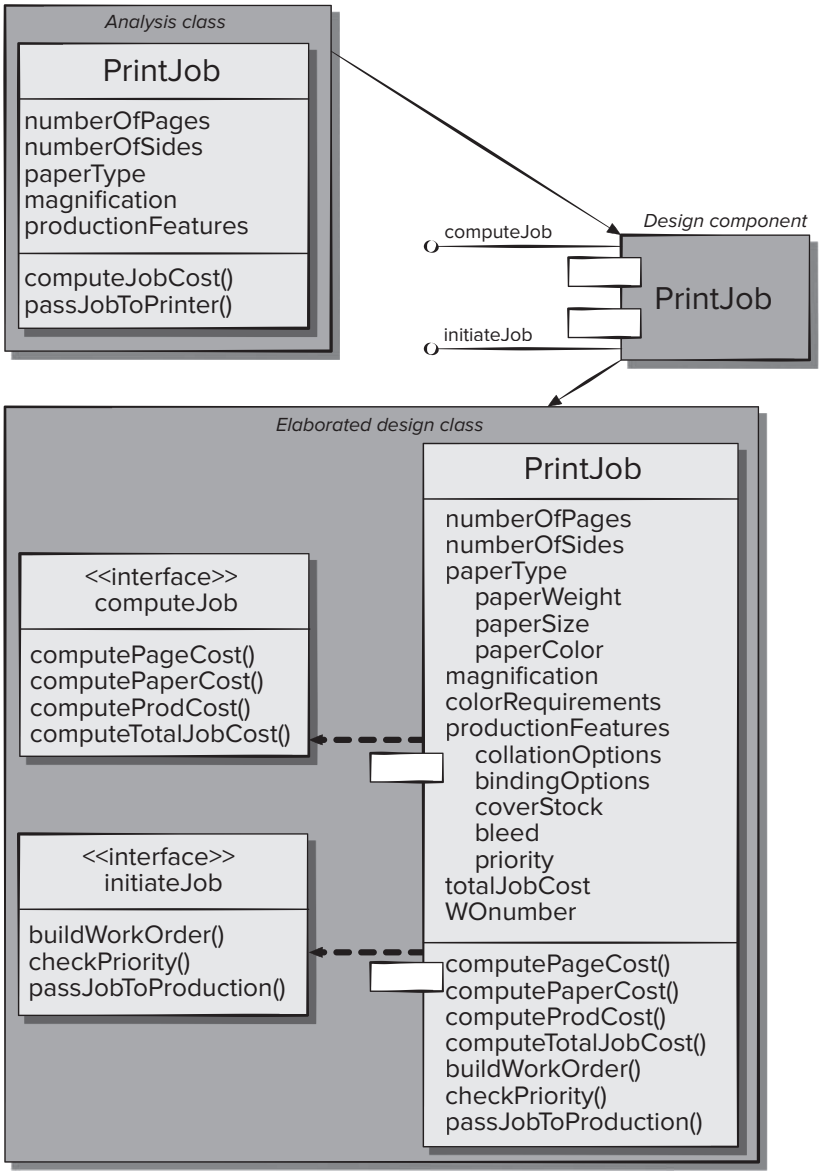
2 Readers who are unfamiliar with UML notation should refer to Appendix 1.

which provides job costing capability, and *initiateJob,* which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob.** Referring to the lower right portion of Figure 11.1, the elaborated design class **PrintJob** contains more

FIGURE 11.1

Elaboration of a design component



detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

This elaboration activity is applied to every component defined as part of the architectural design. Once it is completed, further elaboration is applied to each attribute, operation, and interface. The data structures appropriate for each attribute must be specified. In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed. This procedural design activity is discussed later in this chapter. Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

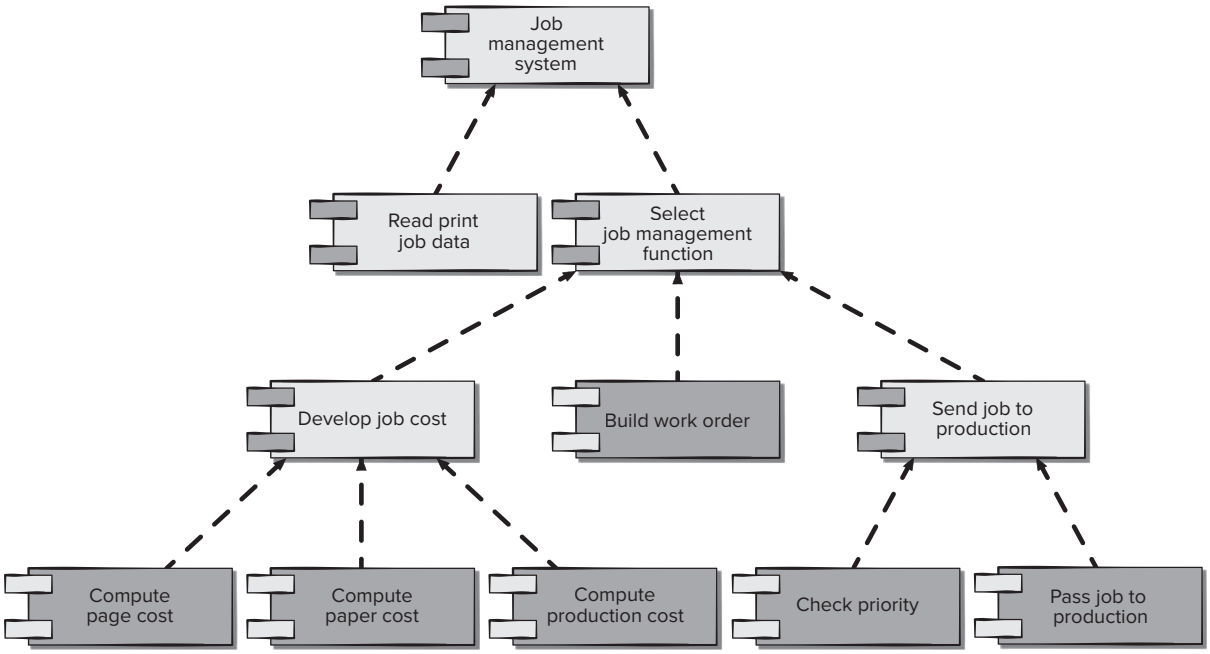
11.1.2 The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a *module,* resides within the software architecture and serves one of three important roles: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. In this case, however, the component elaboration element of the analysis model serves as the basis for the derivation. Each component representing the component hierarchy is mapped (Section 10.6) into a module hierarchy. Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence (Chapter 9) are applied as components are elaborated.

To illustrate this process of design elaboration for traditional components, again consider software to be built for the print shop noted earlier. A hierarchical architecture is derived and shown in Figure 11.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 11.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

FIGURE 11.2 Structure chart for a traditional system

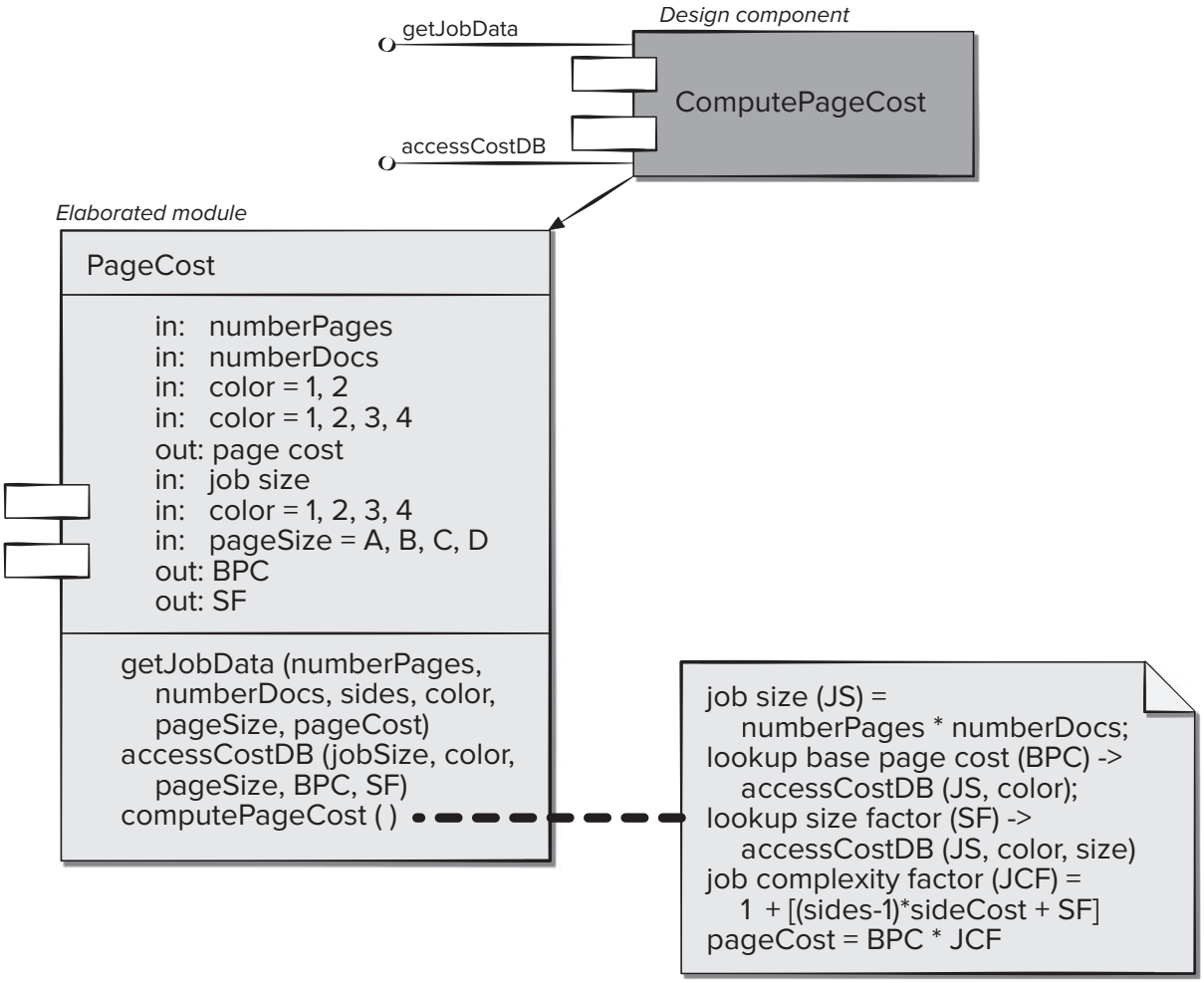


During component-level design, each module in Figure 11.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 9. The behavior of the module is sometimes represented using a state diagram.

To illustrate this process, consider the module *ComputePageCost.* The intent of this module is to compute the printing cost per page based on specifications provided by the customer. Data required to perform this function are: number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements, and size requirements. These data are passed to *ComputePageCost* via the module's interface. *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. Figure 11.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is

FIGURE 11.3 Component-level design for *ComputePageCost*



elaborated to provide algorithm detail and interface detail (Figure 11.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

11.1.3 A Process-Related View

The object-oriented and traditional views of component-level design presented in Sections 11.1.1 and 11.1.2 assume that the component is being designed from scratch. That is, you always create a new component based on specifications derived from the requirements model. There is, of course, another approach.

Over the past four decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. To do this, a catalog of proven design or code-level components needs to be made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you. We will save a discussion on the pros and cons of component-based software engineering (CBSE) for Section 11.4.4.

11.2 DESIGNING CLASS-BASED COMPONENTS

As we have already noted, component-level design draws on information developed as part of the requirements model (Chapter 8) and represented as part of the architectural model (Chapter 10). When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model. The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

11.2.1 Basic Design Principles

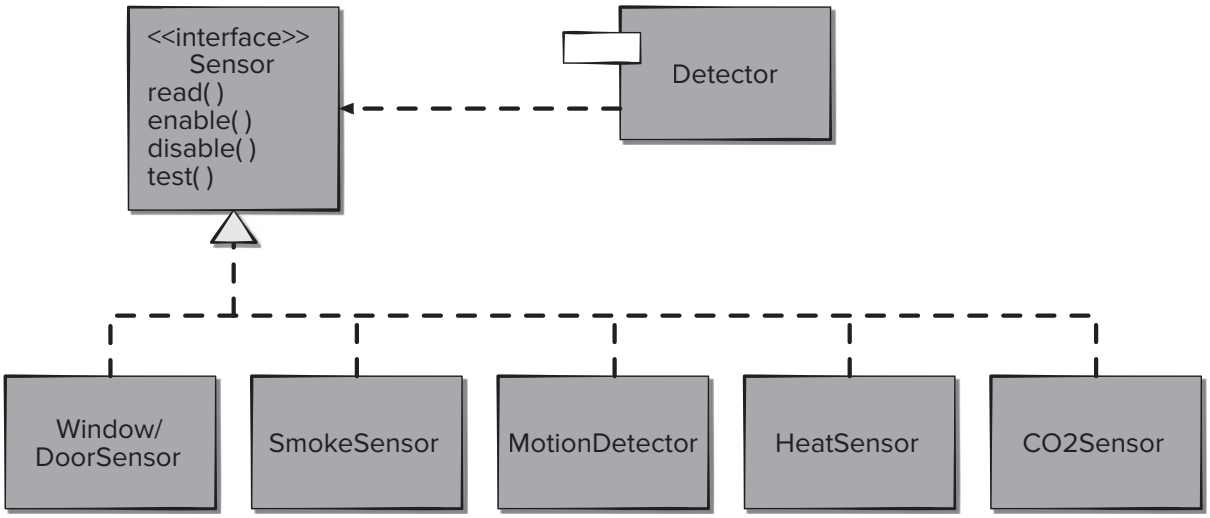
Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur. You can use these principles as a guide as each software component is developed.

The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification"* [Mar00]. This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 11.4. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added, no change is required for the **Detector** class (component). The OCP is preserved.

FIGURE 11.4 Following the OCP



SAFEHOME

The OCP in Action

The scene: Vinod's cubicle.

The players: Vinod and Shakira, members of the *SafeHome* software engineering team.

The conversation:

Vinod: I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.

Shakira (smirking): Not again, jeez!

Vinod: Yeah . . . and you're not going to believe what these guys have come up with.

Shakira: Amaze me.

Vinod (laughing): They call it a doggie angst sensor.

Shakira: Say what?

Vinod: It's for people who leave their pets home in apartments or condos or houses that are close to one another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.

Shakira: You're kidding me, right?

Vinod: Nope. Doug wants to know how much time it's going to take to add it to the security function.

Shakira (thinking a moment): Not much . . . look. [She shows Vinod Figure 11.4.] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an appropriate component . . . uh, class, for it. No change to the **Detector** component at all.

Vinod: So I'll tell Doug it's no big deal.

Shakira: Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.

Vinod: That's not a bad thing, but can you implement now if he wants you to?

Shakira: Yeah, the way we designed the interface lets me do it with no hassle.

Vinod (thinking a moment): Have you ever heard of the open-closed principle?

Shakira (shrugging): Never heard of it.

Vinod (smiling): Not a problem.

The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes"* [Mar00]. This design principle, originally proposed by Barbara Liskov [Lis88], suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a *precondition* that must be true before the component uses a base class and a *postcondition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and post-conditions.

The Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions"* [Mar00]. As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend. Just remember that code is the ultimate concretion. If you dispense with design and hack out code, you're violating DIP.

The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface"* [Mar00]. There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to an individual client category should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

As an example, consider the **FloorPlan** class that is used for the *SafeHome* security and surveillance functions (Chapter 10). For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()* to place, show, group, and remove sensors from the floor plan. The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID().* Hence, the ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them. The interface for security would encompass only the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice().* The interface for surveillance would incorporate the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice(),* along with *showFOV()* and *showDeviceID().*

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design proceeds? Martin [Mar00] suggests additional packaging principles that are applicable to component-level design. These principles follow.

The Reuse/Release Equivalency Principle (REP). *"The granule of reuse is the granule of release"* [Mar00]. When classes or components are designed for reuse,

an implicit contract is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve. Designing components for reuse requires more than good technical design. It also requires effective configuration control mechanisms (Chapter 22).

The Common Closure Principle (CCP). *"Classes that change together belong together"* [Mar00]. Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together"* [Mar00]. When one or more classes with a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operated without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

11.2.2 Component-Level Design Guidelines

In addition to the principles discussed in Section 11.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler [Amb02b] suggests the following guidelines:

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class name **FloorPlan** is meaningful to everyone reading it regardless of technical background. On the other hand, infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning. If a linked list is to be managed as part of the **FloorPlan** implementation, the operation *manageList()* is appropriate, even if a nontechnical person might misinterpret it.[3]

You can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, `<<infrastructure>>` might be used to identify an

3 It is unlikely that someone from marketing or the customer organization (a nontechnical type) would examine detailed design information.

infrastructure component, <<database>> could be used to identify a database that services one or more design classes or the entire system, and <<table>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler [Amb02c] recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex, (2) for consistency, interfaces should flow from the left-hand side of the component box, (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available. These recommendations are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, components' interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency. Following the philosophy of the OCP, this will help to make the system more maintainable.

11.2.3 Cohesion

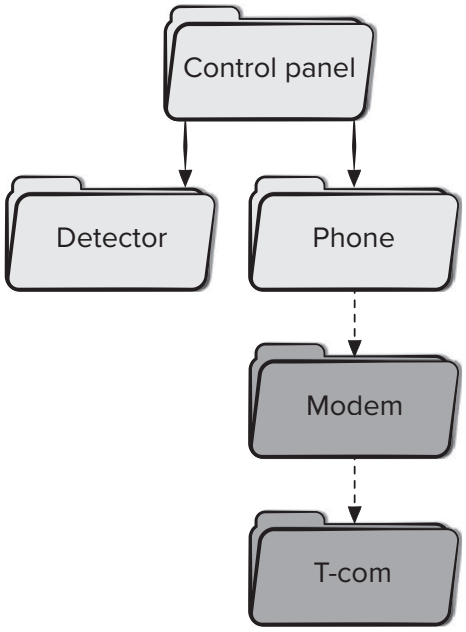
In Chapter 9, we described cohesion as the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to each other and to the class or component itself. Lethbridge and Laganiére [Let04] define several different types of cohesion (listed in order of the level of the cohesion):[4]

> **Functional.** Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns a result.
>
> **Layer.** Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider, for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages, as shown in Figure 11.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.
>
> **Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

4 In general, the higher the level of cohesion, the easier the component is to implement, test, and maintain.

FIGURE 11.5

Layer cohesion



Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. You should strive to achieve these levels of cohesion whenever possible. It is important to note, however, that pragmatic design and implementation issues sometimes force you to opt for lower levels of cohesion.

SAFEHOME

Cohesion in Action

The scene: Jamie's cubicle.

The players: Jamie and Ed, members of the *SafeHome* software engineering team who are working on the surveillance function.

The conversation:

Ed: I have a first-cut design of the **camera** component.

Jamie: Wanna do a quick review?

Ed: I guess . . . but really, I'd like your input on something.

(Jamie gestures for him to continue.)

Ed: We originally defined five operations for **camera**. Look . . .

determineType() tells me the type of camera.

translateLocation() allows me to move the camera around the floor plan.

displayID() gets the camera ID and displays it near the camera icon.

displayView() shows me the field of view of the camera graphically.

displayZoom() shows me the magnification of the camera graphically.

Ed: I've designed each separately, and they're pretty simple operations. So I thought it might be a good idea to combine all of the display operations into just one that's called *displayCamera()*—it'll show the ID, the view, and the zoom. Whaddaya think?

Jamie (grimacing): Not sure that's such a good idea.

Ed (frowning): Why? All of these little ops can cause headaches.

Jamie: The problem with combining them is we lose cohesion, you know, the *displayCamera()* op won't be single-minded.

Ed (mildly exasperated): So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.

Jamie: And what if marketing decides to change the way that we represent the view field?

Ed: I just jump into the *displayCamera()* op and make the mod.

Jamie: What about side effects?

Ed: Whaddaya mean?

Jamie: Well, say you make the change but inadvertently create a problem with the ID display.

Ed: I wouldn't be that sloppy.

Jamie: Maybe not, but what if some support person 2 years from now has to make the mod? He might not understand the op as well as you do, and, who knows, he might be sloppy.

Ed: So you're against it?

Jamie: You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.

Ed (thinking a moment): Maybe we'll go with separate display ops.

Jamie: Good decision.

11.2.4 Coupling

In earlier discussions of analysis and design, we noted that communication and collaboration are essential elements of any object-oriented system. There is, however, a darker side to this important (and necessary) characteristic. As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as possible.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére [Let04] define a spectrum of coupling categories. For example, *content coupling* occurs when one component "surreptitiously modifies data that is internal to another component" [Let04]. This violates information hiding—a basic design concept. *Control coupling* occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result. *External coupling* occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

SAFEHOME

Coupling in Action

The scene: Shakira's cubicle.

The players: Vinod and Shakira, members of the *SafeHome* software team who are working on the security function.

The conversation:

Shakira: I had what I thought was a great idea . . . then I thought about it a little, and it seemed like a not-so-great idea. I finally rejected it, but I just thought I'd run it by you.

Vinod: Sure. What's the idea?

Shakira: Well, each of the sensors recognizes an alarm condition of some kind, right?

Vinod (smiling): That's why we call them sensors, Shakira.

Shakira (exasperated): Sarcasm, Vinod, you've got to work on your interpersonal skills.

Vinod: You were saying?

Shakira: Okay, anyway, I figured . . . why not create an operation within each sensor object called *makeCall()* that would collaborate directly with the **OutgoingCall** component, well, with an interface to the **OutgoingCall** component.

Vinod (pensive): You mean rather than having that collaboration occur out of a component like **ControlPanel** or something?

Shakira: Yeah . . . but then, I said to myself, that means that every sensor object will be connected to the **OutgoingCall** component, and that means that it's indirectly coupled to the outside world and . . . well, I just thought it made things complicated.

Vinod: I agree. In this case, it's a better idea to let the sensor interface pass info to the **ControlPanel** and let it initiate the outgoing call. Besides, different sensors might result in different phone numbers. You don't want the sensor to store that information because if it changes . . .

Shakira: It just didn't feel right.

Vinod: Design heuristics for coupling tell us it's not right.

Shakira: Whatever . . .

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, a designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

11.3 CONDUCTING COMPONENT-LEVEL DESIGN

Earlier in this chapter we noted that component-level design is elaborative in nature. You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated as described in Section 11.1.1.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we

have noted earlier, classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

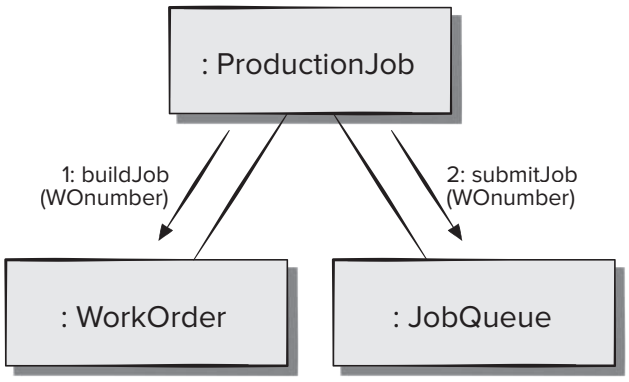
Figure 11.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner [Ben10a]:

```
[guard condition] sequence expression (return value) :=
message name (argument list)
```

where a `[guard condition]` is written in Object Constraint Language (OCL)[5] and specifies any set of conditions that must be met before the message can be sent; `sequence expression` is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; `(return value)` is the name of the information that is returned by the operation invoked by the message; `message name` identifies the operation that is to be invoked; and `(argument list)` is the list of attributes that are passed to the operation.

FIGURE 11.6

Collaboration diagram with messaging



5 OCL is discussed briefly in Appendix 1.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations . . ." [Ben10a]. Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. The elaboration of interfaces is illustrated in Figure 11.1. The operations defined for the design class are categorized into one or more abstract classes. Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

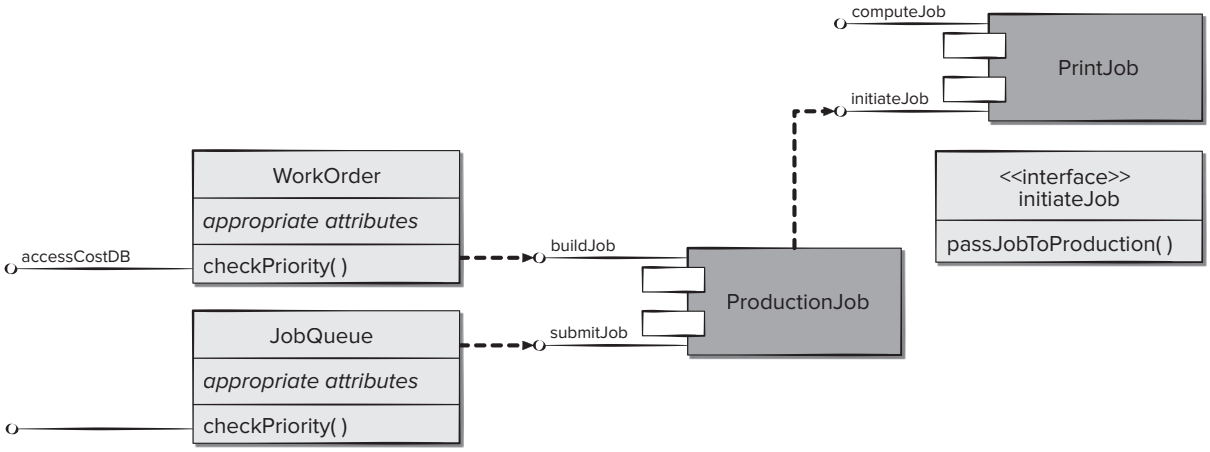
Referring to Figure 11.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. It performs three different subfunctions—building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 11.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly single-minded.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

```
name : type-expression = initial-value {property-string}
```

where `name` is the attribute name, `type expression` is the data type, `initial value` is the value that the attribute takes when an object is created, and `property-string` defines a property or characteristic of the attribute.

FIGURE 11.7 Refactoring interfaces and class definitions for PrintJob



During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 11.1, the attribute list for **PrintJob** lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defined using the UML attribute format noted. For example, `paperType-weight` is defined in the following manner:

```
paperType-weight: string = "A" {contains 1 of 4 values – A, B, C, or D}
```

which defines `paperType-weight` as a string variable initialized to the value A that can take on one of four values from the set {A, B, C, D}.

If an attribute appears repeatedly across several design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through several iterations that apply the stepwise refinement concept (Chapter 9).

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. For example, the operation *computePaperCost()* noted in Figure 11.1 can be expanded in the following manner:

```
computePaperCost (weight, size, color): numeric
```

This indicates that *computePaperCost()* requires the attributes `weight`, `size`, and `color` as input and returns a value that is numeric (actually a dollar value) as output.

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 11.8 depicts a UML activity diagram for *computePaperCost()*. When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code.

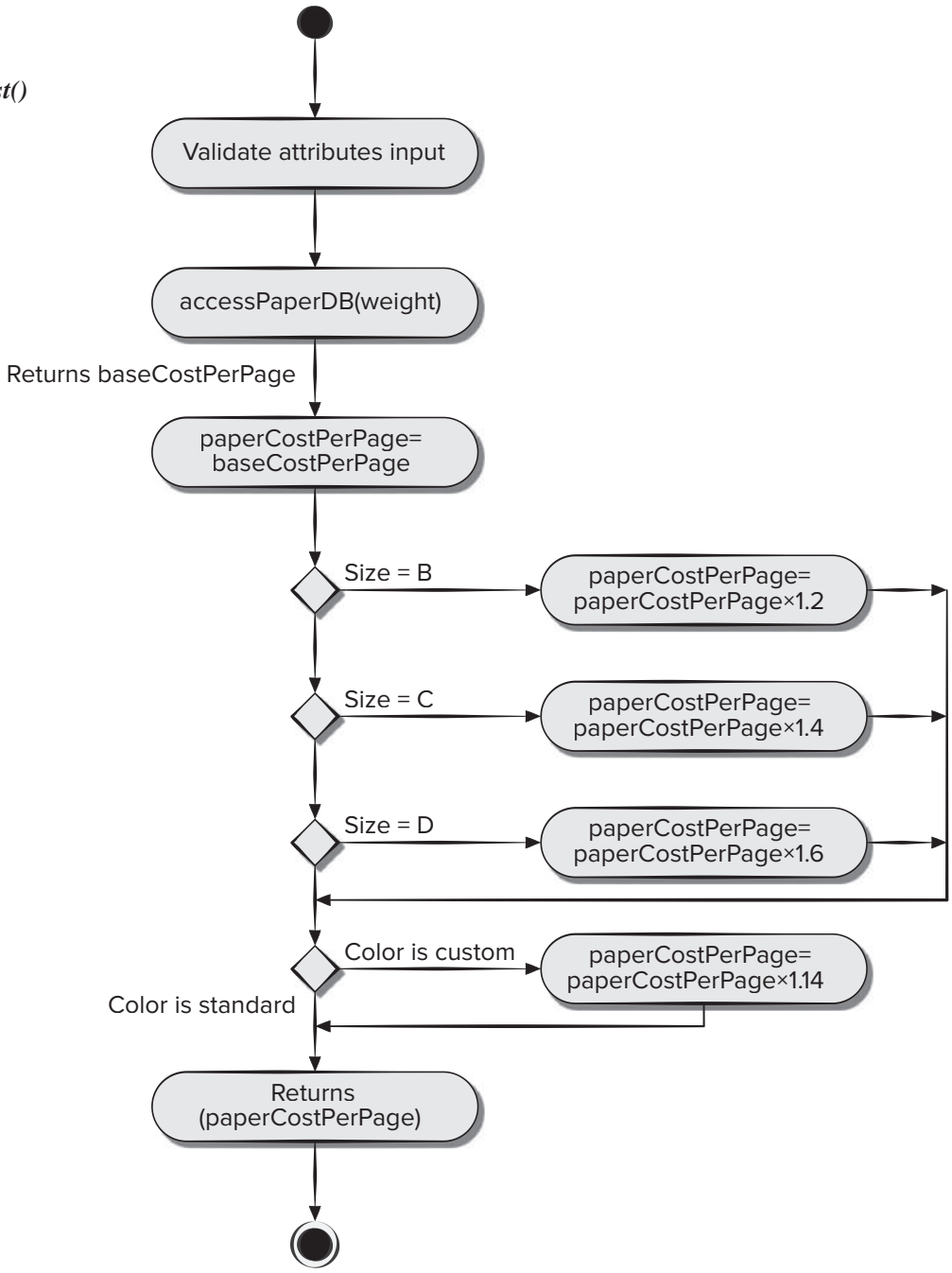
Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, you

FIGURE 11.8

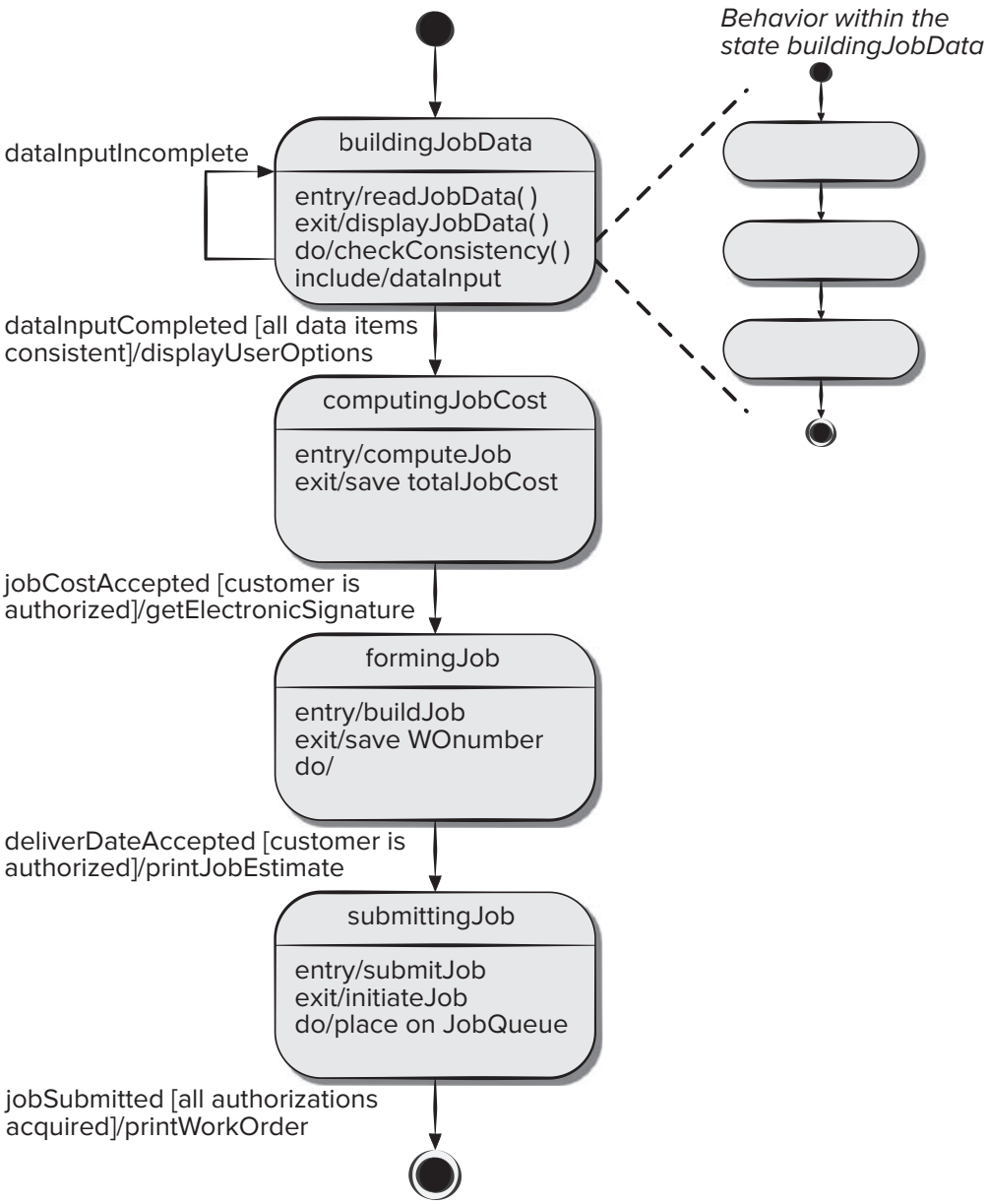
UML activity diagram for *computePaperCost()*



should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) is represented using a UML statechart [Ben10a], as illustrated in Figure 11.9.

FIGURE 11.9

Statechart fragment for PrintJob class



The transition from one state (represented by a rectangle with rounded corners) to another occurs following an event that takes the form of:

```
Event-name (parameter-list) [guard-condition] / action expression
```

where `event-name` identifies the event, `parameter-list` incorporates data that are associated with the event, `guard-condition` is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and `action expression` defines an action that occurs as the transition takes place.

Referring to Figure 11.9, each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that occur while in the state, and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 11.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve), the print job cannot be submitted because there is no way to reach the *submittingJob* state.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams (Chapter 9) are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented (often as subsystems) within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always consider alternatives. Throughout this book, we emphasize that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

In addition, you should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before settling on the final design model. Develop alternatives and consider each carefully, using the design principles and concepts presented in Chapter 9 and in this chapter.

11.4 SPECIALIZED COMPONENT-LEVEL DESIGN

There are many programming languages and many ways to create the components required to implement a software architectural design. The principles described in this chapter provide general advice for designing components. Many software products require the use of specialized program development environments to allow their deployment on targeted end user devices such as cell phones or digital assistants. In this section, we present overviews of some specialized component design techniques.

11.4.1 Component-Level Design for WebApps

The boundary between content and function is often blurred when Web-based systems and applications (WebApps) are considered. Therefore, it is reasonable to ask: What is a WebApp component?

In the context of this chapter, a WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content design at the component level focuses on content objects and the ways they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelationships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation.[6] In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

A good example of a component that might be part of an e-commerce WebApp is the "shopping cart." A shopping cart provides a convenient way for e-commerce customers to store and review their selected items prior to checking out. They can then pay for their selections with a single transaction at the end of their e-commerce session. A carefully designed shopping cart can be reused in several Web store applications by simply editing its content model.

WebApp functionality can be delivered as a series of components developed in parallel with the information architecture to ensure consistency. The shopping cart component described previously contains both content and algorithmic elements. You begin by considering both the requirements model and the initial information architecture. Next, you examine how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A *functional architecture* is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

11.4.2 Component-Level Design for Mobile Apps

Mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. If you are building a mobile app as a thin Web-based client, the only components residing on a mobile device are those required to implement the user interface. Some mobile apps may incorporate the components required to implement the business and/or data layers on the mobile device, subjecting these layers to the limitations of the physical characteristics of the device.

---

6 Content components can also be reused in other WebApps.

Considering the user interface layer first, it is important to recognize that a small display area requires the designer to be more selective in choosing the content (text and graphics) to be displayed. It may be helpful to tailor the content to a specific user group(s) and display only what each group needs. The business and data layers are often implemented by composing Web or cloud service components. If the components providing business and data services reside entirely on the mobile device, connectivity issues are not a significant concern. Intermittent (or missing) Internet connectivity must be considered when designing components that require access to current application data that reside on a networked server.

If a desktop application is being ported to a mobile device, the business-layer components may need to be reviewed to see if they meet nonfunctional requirements (e.g., security, performance, accessibility) required by the new platform. The target mobile device may lack the necessary processor speed, memory, or display real estate. The design of mobile applications is considered in greater detail in Chapter 13.

An example of a component in a mobile application might be the single-window full-screen user interface (UI) typically designed for phones and tablets. With careful design it may be possible to allow the mobile app to sense the display characteristics of the mobile device and adapt its appearance to ensure that text, graphics, and UI controls function correctly on many different screen types. This allows the mobile app to function in similar ways on all platforms, without having to be reprogrammed.

11.4.3 Designing Traditional Components

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra ([Dij65], [Dij76b]) and others (e.g., [Boh66]). A traditional software component implements an element of processing that addresses a function or subfunction in the problem domain or some capability in the infrastructure domain. Often these traditional components are called functions, modules, procedures, or subroutines. Traditional components do not encapsulate data in the same way that object-oriented components do. Most programmers make frequent use of function libraries and data structure templates when developing new software products.

In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures. Complexity metrics (Chapter 23) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which

you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties.

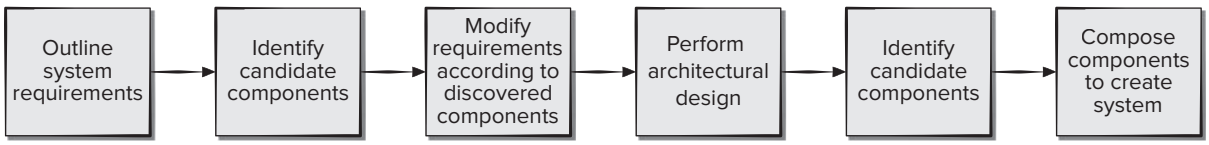
11.4.4 Component-Based Development

In software engineering, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer-based systems must be built in very short time periods and demand a more organized approach to reuse.

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software components (Figure 11.10). Considering this description, many questions arise. Is it possible to construct complex systems by assembling them from a catalog of reusable software components? Can this be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is management willing to incur the added expense associated with creating reusable software components? Can a library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it? Can existing components be found by those who need them? Increasingly, the answer to each of these questions is yes.

Figure 11.10 shows the principle steps in CBSE. You start with the system requirements and refine them to the point that needed components can be identified. The developers would then search the repository to see if any of the components already exist. Each component has its own postconditions and preconditions. Components whose postconditions match a system requirement are identified, and the preconditions of each component are checked. If the preconditions are satisfied, the component is selected for inclusion in the current build. When no components can be selected, the developers must decide whether to modify the requirements or modify a component that most closely matches the original requirements. This is often an iterative process that continues until the architecture design can be implemented, using a combination of existing or newly created components.

FIGURE 11.10 **Component-based software design**



Consider the task of developing autonomous vehicles, either in real life or a video game. The software for these complex systems is typically created by combing several reusable components, where the components provide distinct modular services. Typically they would include many software components: a component that manages obstacle detection, a planning or navigation component, an artificial intelligence component to manage decision making, and a component of some type controlling vehicle movement or braking. Because these types of software modules have the potential to be used in many different vehicles, it would be desirable to be able to house them in a library of components.

Because CBSE makes use of existing components, it can shorten development time and increase quality. Practitioners [Vit03] often attribute the following advantages to CBSE:

- **Reduced lead time.** It is faster to build complete applications from a pool of existing components.
- **Greater return on investment (ROI).** Sometimes savings can be realized by purchasing components rather than redeveloping the same functionality in-house.
- **Leveraged costs of developing components.** Reusing components in multiple applications allows the costs to be spread over multiple projects.
- **Enhanced quality.** Components are reused and tested in many different applications.
- **Maintenance of component-based applications.** With careful engineering, it can be relatively easy to replace obsolete components with new or enhanced components.

Use of components in CBSE is not without risks. Several of these include the following [Kau11]:

- **Component selection risks.** It is difficult to predict component behavior for black-box components, or there may be poor mapping of user requirements to the component architectural design.
- **Component integration risks.** There is a lack of interoperability standards between components; this often requires the creation of "wrapper code" to interface components.
- **Quality risks.** Unknown design assumptions made for the components makes testing more difficult, and this can affect system safety, performance, and reliability.
- **Security risks.** A system can be used in unintended ways, and system vulnerabilities can be caused by integrating components in untested combinations.
- **System evolution risks.** Updated components may be incompatible with user requirements or contain additional undocumented features.

One of the challenges facing widespread component reuse is *architectural mismatch* [Gar09a]—incompatibilities between assumptions made about components and their operating environments.[7] These assumptions often focus on the component

7 This can be a result of several forms of coupling that should be avoided whenever possible.

control model, the nature of the component connections (interfaces), the architectural infrastructure itself, and the nature of the construction process.

Early detection of architectural mismatch can occur if stakeholder assumptions are explicitly documented. In addition, the use of a risk-driven process model emphasizes the definition of early architectural prototypes and points to areas of mismatch. Repairing architectural mismatch is often very difficult without making use of mechanisms like wrappers or adapters.[8] Sometimes it is necessary to completely redesign a component interface or the component itself to remove coupling issues.

11.5 COMPONENT REFACTORING

Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, and software quality assurance (SQA) all contribute to the creation of software components that will be easier to refactor. Most developers would agree that refactoring components to improve quality is a good practice. It is often hard to convince management that it is important to expend resources fixing components that are working correctly instead of adding new functionality to them.

In this book, we focus on the incremental design and delivery of system components. Although there is no quantifiable relationship describing the effects of code changes on architectural quality, most software engineers agree that over time large numbers of changes to a system can lead to the creation of problematic structures in the code base. Failing to address these problems increases the amount of technical debt (Chapter 9) associated with the software system. Reducing this technical debt often involves architectural refactoring, which is generally perceived by developers as both costly and risky. You cannot simply break up large components into smaller components and expect to see an automatic increase in cohesion and a reduction in coupling that will reduce technical debt.

Large software systems may have thousands of components. Making use of data-mining techniques to identify refactoring opportunities can be very beneficial to this work. Automated tools can analyze the source code of system components and make refactoring recommendations to developers, based on generic design rules known to be associated with architectural problems. But it is still up to the developers and their managers to decide which changes to accept and which to ignore [Lin16].

It turns out that many of the error-prone components in a software system are architecturally connected to one another. These flawed architectural connections tend to propagate defects among themselves and accumulate high maintenance costs. If it was possible to automatically identify the technical debt present in the system and the associated maintenance costs, it would be easier to convince developers and managers to spend time refactoring these components. Accomplishing this type of

8 An *adapter* is a software device that allows a client with an incompatible interface to access a component by translating a request for service into a form that can access the original interface.

work requires examining the change histories of the system components [Xia16]. For example, if two or three components are always checked out of the code repository for modification at the same time, it may suggest the components share a common design defect.

11.6 SUMMARY

The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Three different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The traditional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

Several important principles and concepts guide the designer as classes are elaborated. Ideas encompassed in the open-closed principle and the dependency inversion principle, along with concepts such as coupling and cohesion, guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated by specifying messaging details, identifying appropriate interfaces, elaborating attributes, and defining data structures to implement them, describing processing flow within each operation, and representing behavior at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Traditional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of several design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Component-level design for WebApps considers both content and functionality as a Web-based system will deliver it. Content design at the component level focuses on content objects and the ways they may be packaged for presentation to a WebApp end user. Functional design for WebApps focuses on processing functions that manipulate content, perform computations, process database queries, and establish interfaces with other systems. All component-level design principles and guidelines apply.

Component-level design for mobile apps makes use of a multilayered architecture that includes a user interface layer, a business layer, and a data layer. If the mobile app requires the design of components that implement the business and/or data layers on the mobile device, the limitations of the physical characteristics of the device become important constraints on the design.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Component-based software engineering identifies, constructs, catalogs, and disseminates a set of software components for an application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

PROBLEMS AND POINTS TO PONDER

11.1. The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for object-oriented and traditional software. Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

11.2. Why are control components necessary in traditional software and generally not required in object-oriented software?

11.3. Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?

11.4. Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?

11.5. Select three components that you have developed recently, and assess the types of cohesion that each exhibits. If you had to define the primary benefit of high cohesion, what would it be?

11.6. Select three components that you have developed recently, and assess the types of coupling that each exhibits. If you had to define the primary benefit of low coupling, what would it be?

11.7. Develop (1) an elaborated design class, (2) interface descriptions, (3) an activity diagram for one of the operations within the class, and (4) a detailed statechart diagram for one of the *SafeHome* classes that we have discussed in earlier chapters.

11.8. What is a WebApp component?

11.9. Select the code from a small software component and represent it using an activity diagram.

11.10. Why is "chunking" important during the component-level design review process?