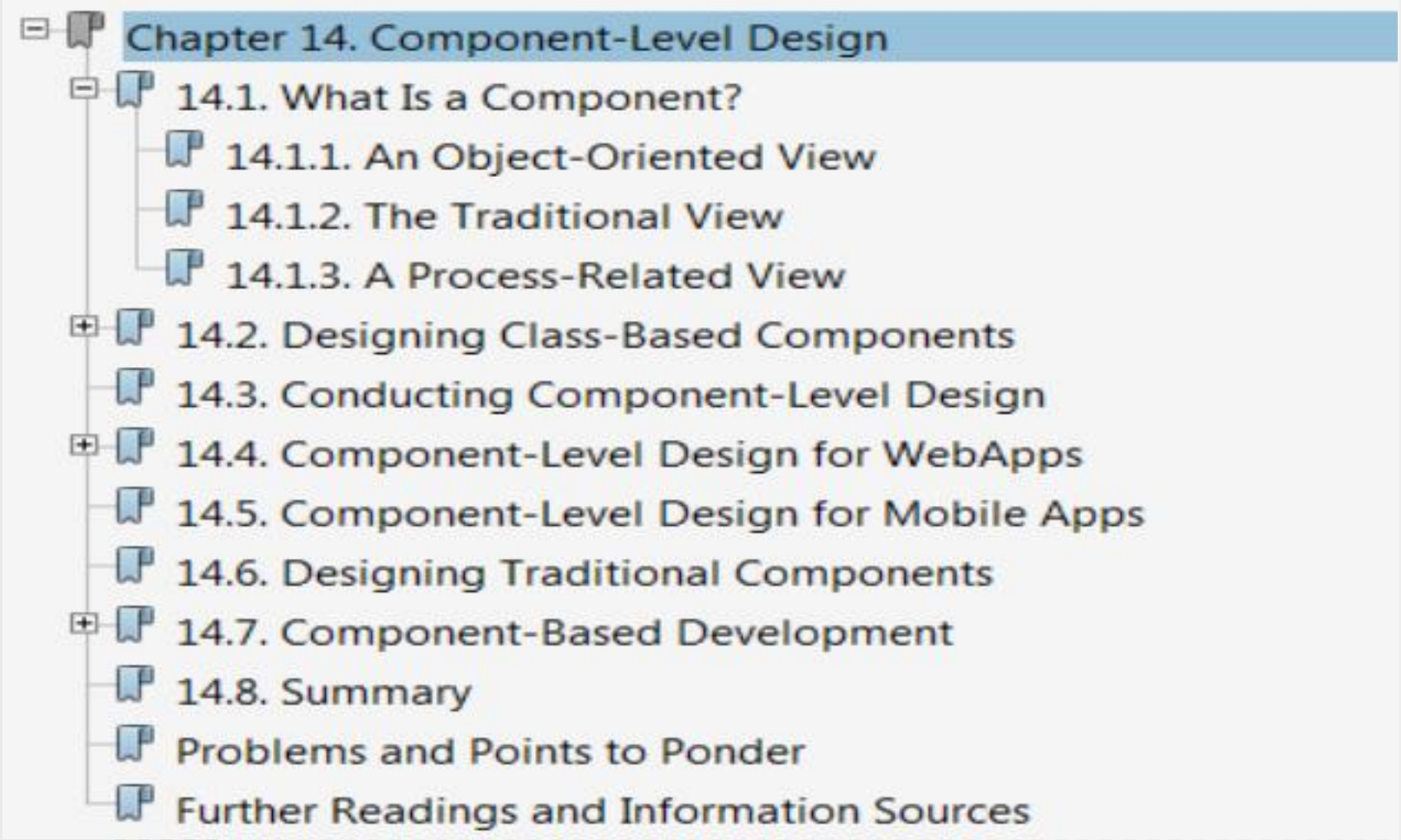
Chapter 14 COMPONENT-LEVEL DESIGN



Chapter 14 COMPONENT-LEVEL DESIGN

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.

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

A design review is conducted. The design is examined to determine whether data structures,interfaces, processing sequences, and logical conditions are correct.

Chapter 14 COMPONENT-LEVEL DESIGN

14.1 WHAT IS A COMPONENT?

A component is a modular building block for computer software,a component is seen as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

14.1.1 An Object-Oriented View

In the context of object-oriented software engineering, a component not only contains itself but also involves in a set of collaborating classes. 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

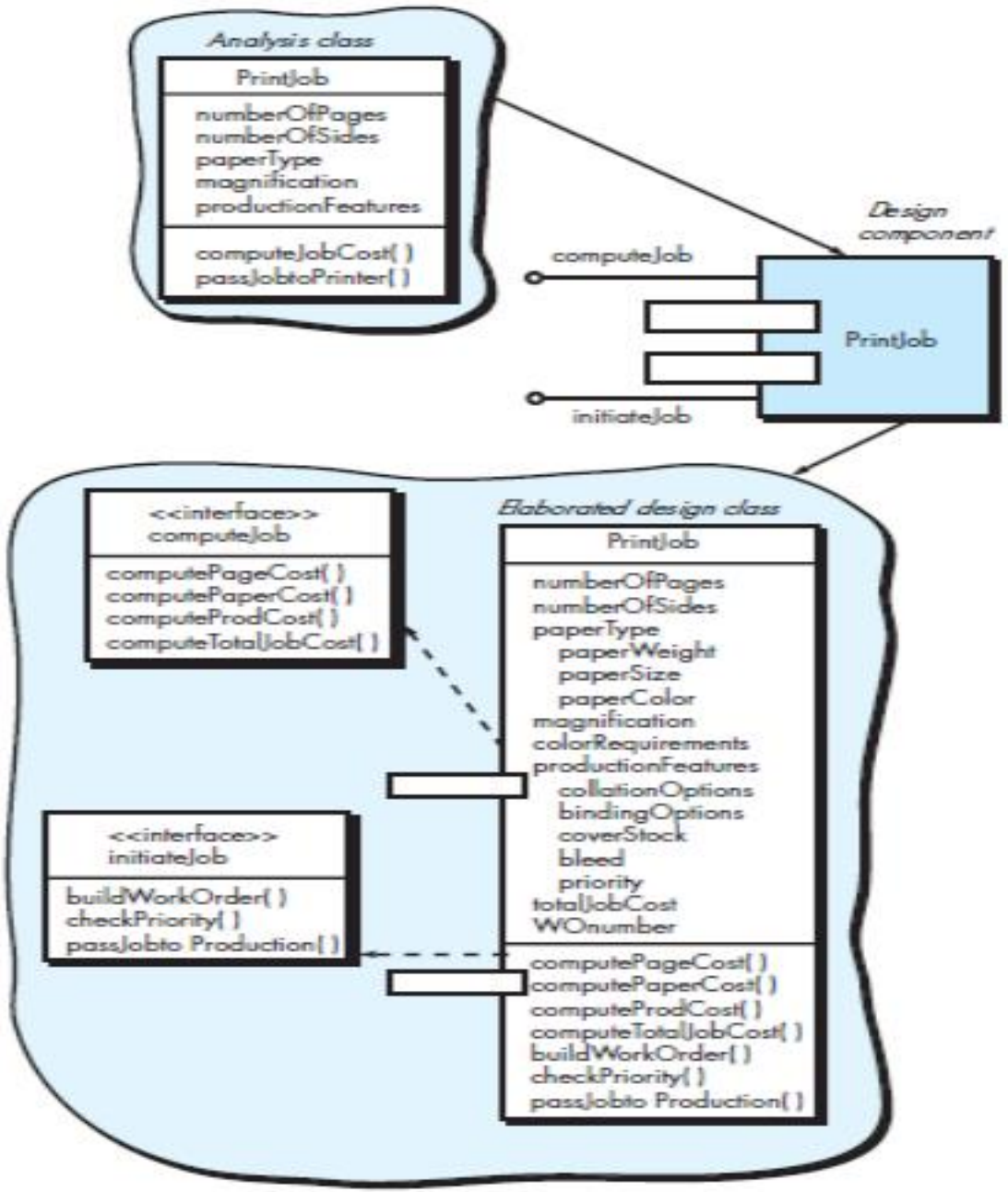
components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop, figure 14.1.

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.

FIGURE 14.1

Elaboration of a design component



Chapter 14 COMPONENT-LEVEL DESIGN

14.1.2 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 represented the component hierarchy is mapped (Section 13.6) 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 12) are applied as components are elaborated.

To illustrate this process of design elaboration for traditional components, again consider software to be built for a sophisticated print shop,figure 14.2.

FIGURE 14.2 Structure chart for a traditional system

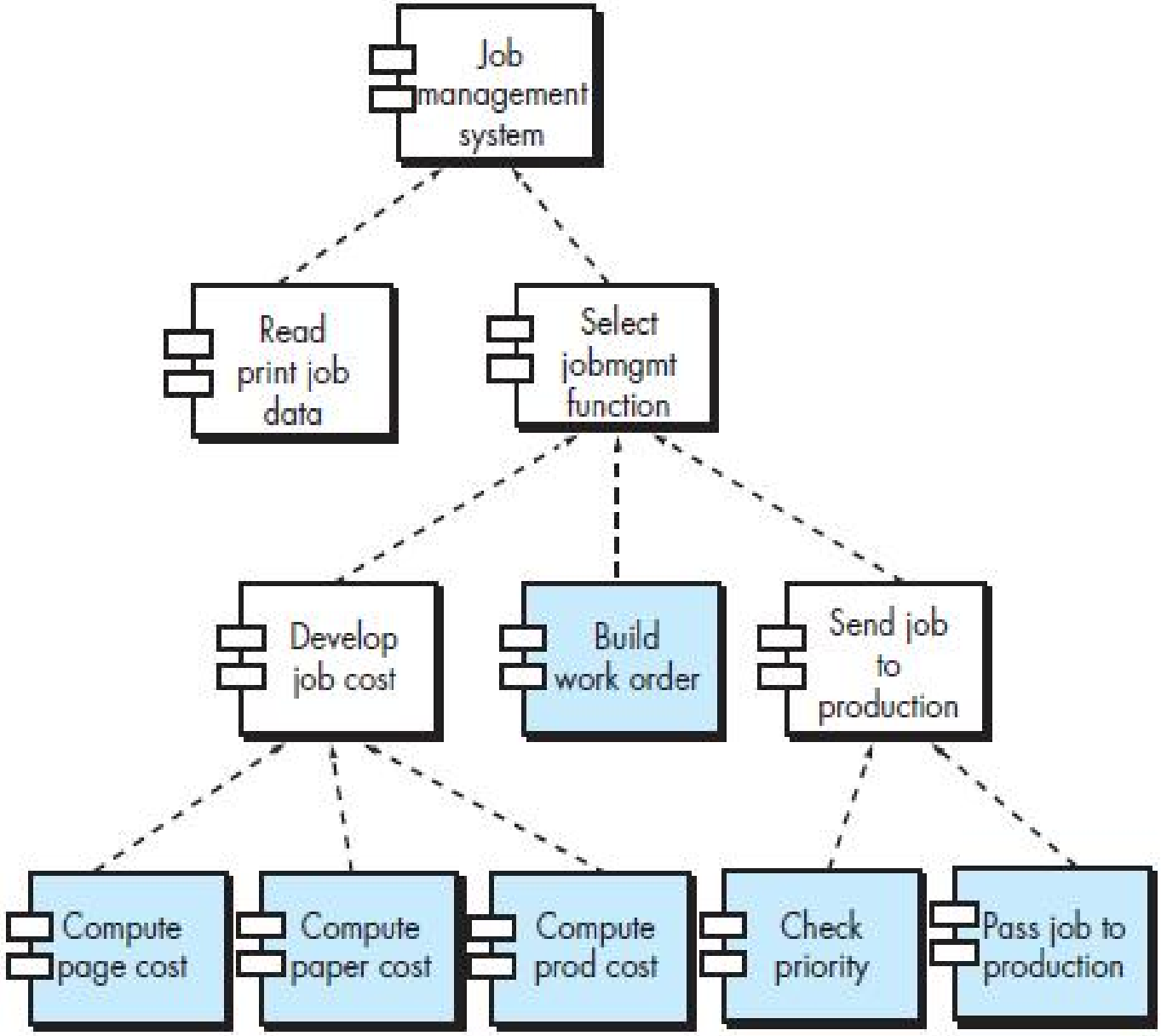
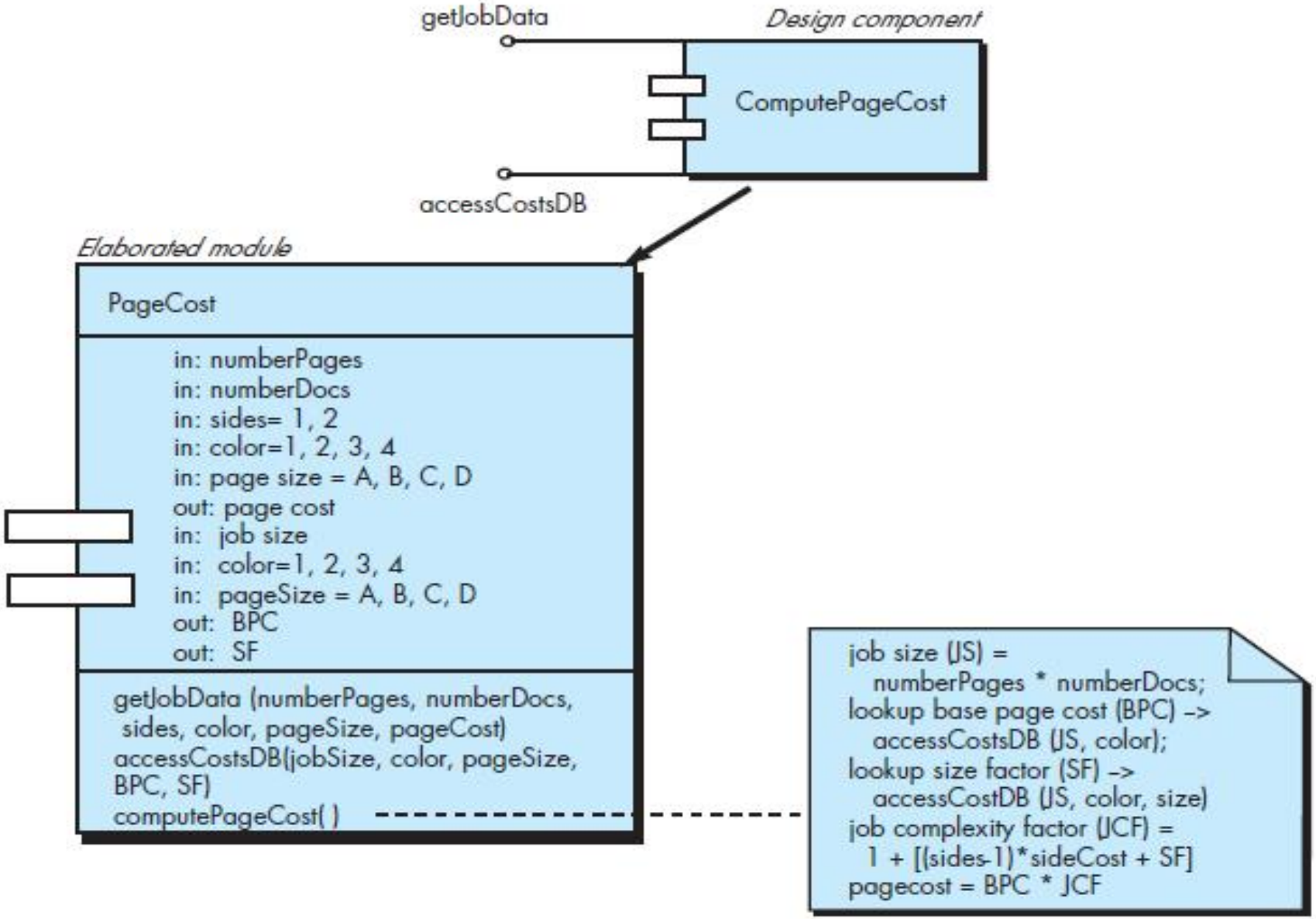


FIGURE 14.3 Component-level design for *ComputePageCost*



Chapter 14 COMPONENT-LEVEL DESIGN

14.1.3 A Process-Related View（read by yourselves）

Over the past three decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is 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 discuss some of the important aspects of component-based software engineering (CBSE) later in Section 14.6.

Chapter 14 COMPONENT-LEVEL DESIGN

INFO

Component-Based Standards and Frameworks

One of the key elements that lead to the success or failure of CBSE is the availability of component-based standards, sometimes called middleware. *Middleware* is a collection of infrastructure components that enable problem domain components to communicate with one another across a network or within a complex system. Software engineers who want to use component-based development as their software process can choose from among the following standards:

OMG CORBA– www.corba.org/

Microsoft COM– http://www.microsoft.com/com/default.mspx

Microsoft .NET– http://msdn.microsoft.com/en-us/netframework/default.aspx

Sun JavaBeans– http://www.oracle.com/technetwork/java/javaee/ejb/index.html

The websites noted present a wide array of tutorials, white papers, tools, and general resources on these important middleware standards.

Chapter 14 COMPONENT-LEVEL DESIGN

14.2 DESIGNING CLASS-BASED COMPONENTS(read by yourselves)

14.2.1 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*****. 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.

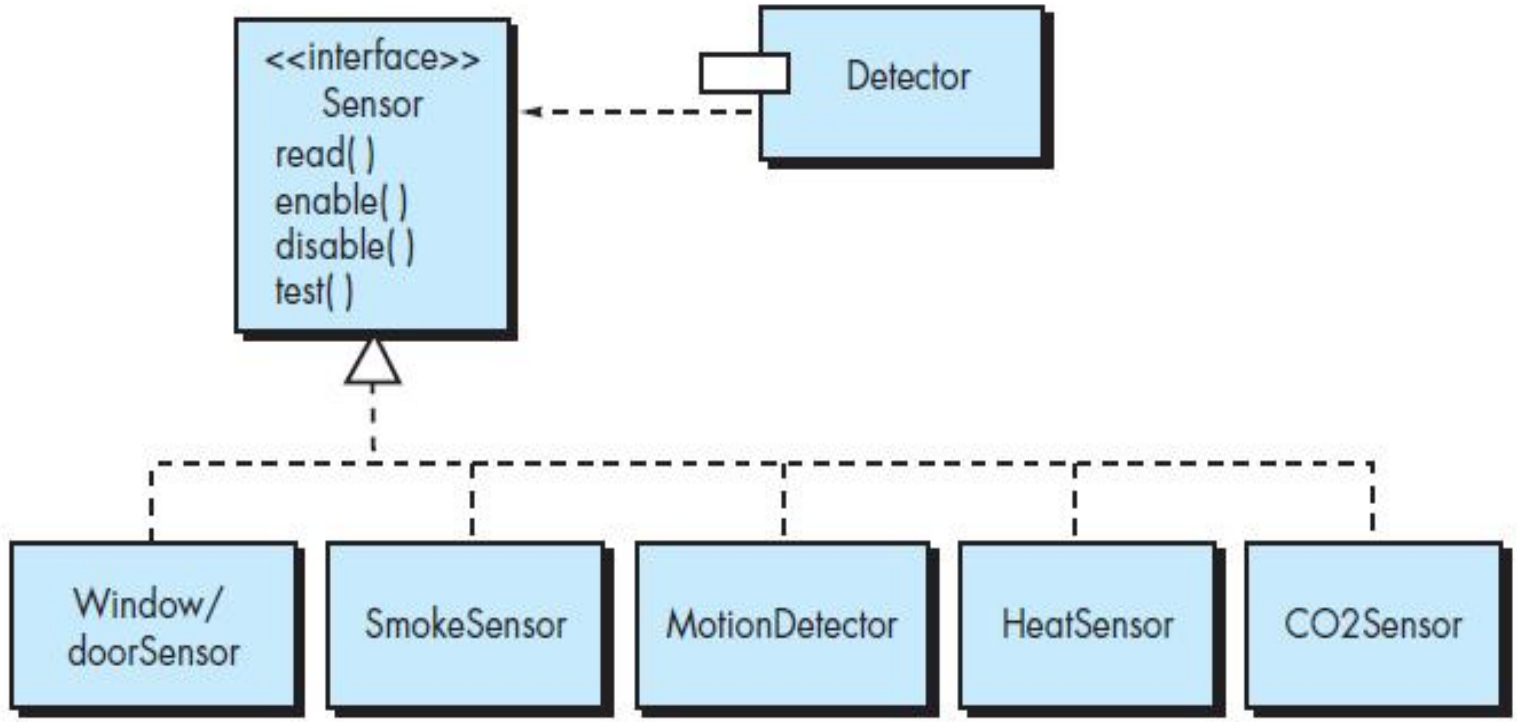
For example, assume that the SafeHome security function makes use of a Detector class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else 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 14.4. 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.

Chapter 14 COMPONENT-LEVEL DESIGN

FIGURE 14.4

Following the OCP



Chapter 14 COMPONENT-LEVEL DESIGN

The Liskov Substitution Principle (LSP).

Dependency Inversion Principle (DIP).

The Interface Segregation Principle (ISP).“Many client-specific interfaces are better than one general purpose interface”.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 a particular category of clients 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()*.

The Release Reuse Equivalency Principle (REP)(read by yourselves).“ The granule of reuse is the granule of release” . 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.

The Common Closure Principle (CCP). " Classes that change together belong together."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.

Chapter 14 COMPONENT-LEVEL DESIGN

The Common Reuse Principle (CRP).(read by yourself) " Classes that aren't reused together should not be grouped together" . When one or more classes with a package changes, the release number(or version 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.

14.2.2 Component-Level Design Guidelines(read by yourselves)

In addition to the principles discussed in Section 14.2.1, 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.

Chapter 14 COMPONENT-LEVEL DESIGN

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.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler recommends that (1)

Chapter 14 COMPONENT-LEVEL DESIGN

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. 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.

Chapter 14 COMPONENT-LEVEL DESIGN

14.2.3 Cohesion(read by yourselves)

In Chapter 12, 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 one another and to the class or component itself.

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. 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.（access database related class）

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.

Chapter 14 COMPONENT-LEVEL DESIGN

14.2.4 Coupling(read by yourselves)

In earlier discussions of analysis and design, we noted that communication and collaboration are essential elements of any object-oriented system. 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 is possible.

Class coupling can manifest itself in a variety of ways:

Chapter 14 COMPONENT-LEVEL DESIGN

Content coupling occurs when one component "surreptitiously modifies data that is internal to another component" .

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. 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.

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

Chapter 14 COMPONENT-LEVEL DESIGN

14.3 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 14.1.1.

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（control）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.

Figure 14.6 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:

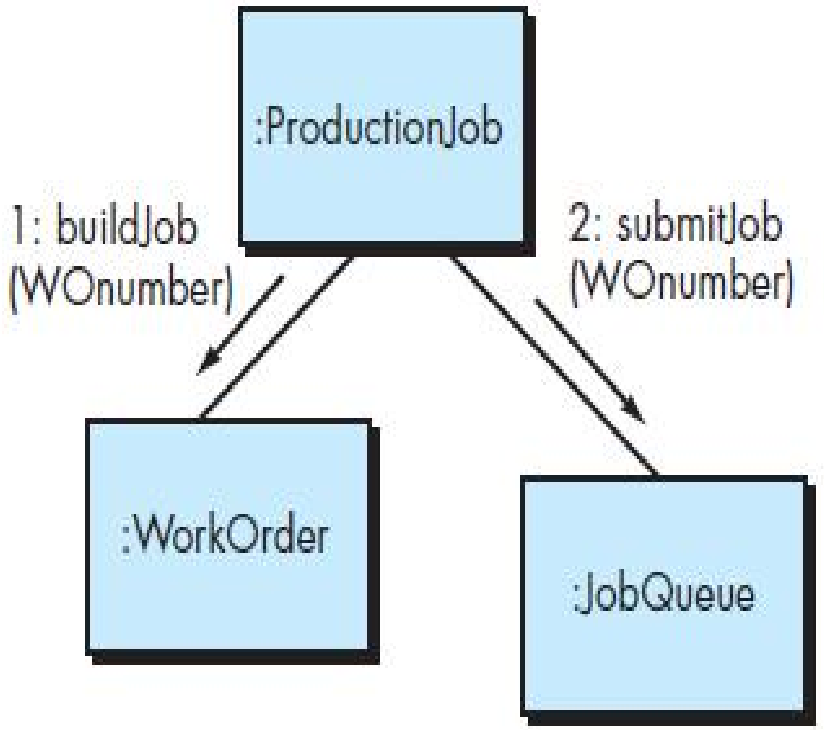
[guard condition] sequence expression (return value) :=

message name (argument list)

Chapter 14 COMPONENT-LEVEL DESIGN

FIGURE 14.6

Collaboration diagram with messaging



where a [guard condition] is written in Object Constraint Language (OCL) 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.

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. . ." . Stated more formally, an interface is the equivalent of an abstract

Chapter 14 COMPONENT-LEVEL DESIGN

class that provides a controlled connection between design classes.

The elaboration of interfaces is illustrated in Figure 14.1 . In essence, 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.

Chapter 14 COMPONENT-LEVEL DESIGN

Step 3c. Elaborate attributes and define data types and data structures required to implement them. 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.

During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 14.1 , the attribute list for PrintJob lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defi ned 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 a number of 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 a number of iterations that apply the stepwise refinement concept (Chapter 12).

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 14.1 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.

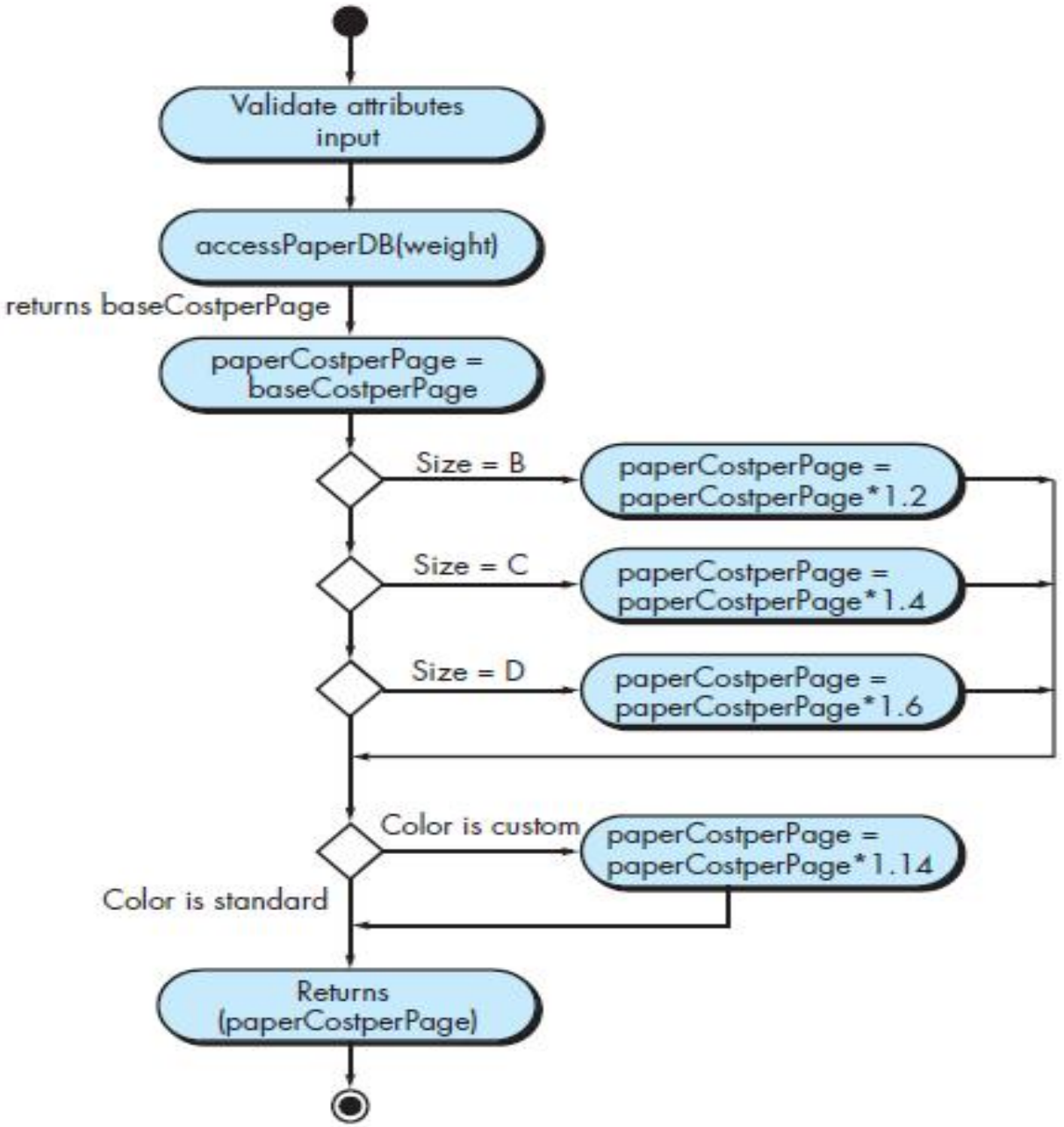
If the algorithm required to implement computePaperCost() is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration

is required at this stage. Figure 14.8 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. An alternative approach—the use of pseudocode for design specifi cation—is discussed in Section 14.5.3.

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.

FIGURE 14.8

UML activity diagram for *computePaperCost()*



Chapter 14 COMPONENT-LEVEL DESIGN

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.

Chapter 14 COMPONENT-LEVEL DESIGN

Step 6. Elaborate deployment diagrams to provide additional implementation detail. **Deployment diagrams (Chapter 12) are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented 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.**

Chapter 14 COMPONENT-LEVEL DESIGN

Step 7. Refactor every component-level design representation and always consider alternatives. **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 nth iteration you apply to the model. It is essential to refactor as design work is conducted.**

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 12 and in this chapter.

Chapter 14 COMPONENT-LEVEL DESIGN

14.4 COMPONENT-LEVEL DESIGN FOR WEBAPPS(read by yourselves)

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.*

14.4.1 Content Design at the Component Level

Content (text and graphics)design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user.

Chapter 14 COMPONENT-LEVEL DESIGN

The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built.

14.4.2 Functional Design at the Component Level

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.

14.5 COMPONENT-LEVEL DESIGN FOR MOBILE APPS(read by yourselves)

In Chapter 13 we noted that 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

Chapter 14 COMPONENT-LEVEL DESIGN

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.

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

14.6 DESIGNING TRADITIONAL COMPONENTS(read by yourselves)

In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed.

Chapter 14 COMPONENT-LEVEL DESIGN

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.

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.

14.7 COMPONENT-BASED DEVELOPMENT(read by yourselves)