From

Intro. to

Software Engineering

Design

Christopher Fox

Addison-Wesley, 2007

9 Architectural Design

Chapter Objectives

This is the first of two chapters surveying architectural design. This chapter introduces the topic and discusses architectural design activities and notations. The place of architectural design in the software engineering design resolution process is shown in Figure 9-O-1.

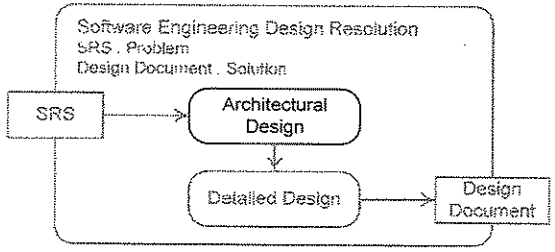

Figure 9-O-1 Software Engineering Design Resolution

By the end of this chapter you will be able to

- State the role of software architecture in the engineering design process and explain the architectural design process;
- List and explain the contents of a software architecture design template;
- Explain what quality attributes are and why they are important in architectural design;
- Specify module interface syntax, semantics, and pragmatics;
- Read and write architectural models using box-and-line diagrams;
- Read and write UML diagrams that include notes, constraints, properties, stereotypes, and dependency relations;
- Read and write design models using UML package and component diagrams; and
- Read and write UML deployment diagrams depicting physical architecture.

Chapter Contents

9.1 Introduction to Architectural Design

The Context of Architectural Design

Engineering design resolution has two phases: architectural design and detailed design. Architectural design is a problem-solving activity whose input is the product description in an SRS and whose output is the abstract specification of a program realizing the desired product. Architectural design thus sits between software product design and detailed design in the software design process.

But in fact, the architectural design activity is not as cleanly separated from product design and detailed design as the previous paragraph suggests. Some architectural design occurs during product design for the following reasons:

- Product designers must judge the feasibility of their designs, which may be difficult without some initial engineering design work.
- Stakeholders must be convinced that their needs will be met, which may be difficult without demonstrating how the engineers plan to build the product.
- Designers and stakeholders must trade off requirements to create a feasible product that can be built on schedule and within budget. Trade-offs may not be clear without exploring alternative software architectures.
- Project planners must have some idea about what software must be built to create schedules and allocate resources.

Consequently, engineering design work often begins during product design, proceeds in parallel with it, and influences product design decisions.

The boundary between architectural and detailed design is even less clear. In Chapter 8, we noted that a software architecture specifies a program's major constituents, their responsibilities and properties, and the relationships and interactions among them. Detailed design refines the architecture by specifying the internal details of the major program constituents and fleshing out the details of their properties, relationships, and interactions. This picture is vague, and, in particular, we may wonder:

- What comprises a "major" constituent?
- How abstract should architectural specifications be?

There are no definitive answers to either of these questions. Some architects lean toward quite abstract specifications of a few top-level constituents, while others insist on detailed constituent specifications through several layers of abstraction.

Problem Context

The circumstances of the design problem also dictate different amounts of detail in a software architecture. In a very small program consisting of only a handful of classes or modules interacting in simple ways, the software architecture is hardly distinguishable from the detailed design, so it is appropriate for the architecture to be simple and quite abstract. However, a very large and complicated system with hundreds or thousands of classes, distributed over many machines, interacting with many peripheral devices, and with demanding non-functional requirements, demands a detailed high-level specification that is carefully worked out and analyzed. Experience and engineering judgment must dictate the level of detail provided in an architectural specification.

Organizational Context

The organization in which architectural design takes place influences the architectural design process. An organization has investments, such as code libraries, standards and guidelines, software tools, and people with particular skills, that software architects are expected to make the most of in their designs. Organizations also have structures that may influence designers. For example, an organization may have groups that specialize in developing certain kinds of software, such as user interfaces, database systems, middleware, and networks. Architects may have to design programs whose constituents can be farmed out to existing development groups. Finally, the skills, experience, and preferences of architects themselves obviously influence the designs they produce.

Incidentally, software architectures have a reciprocating influence on the organizations that create and use them. An organization may make investments in tools, technologies, methods, and people needed to build a program according to a software architecture. Groups may be formed to implement an architecture's major parts or sub-systems. Architects learn and grow as they solve new problems and learn what works and what does not.

The Architectural Design Process

The architectural design process is a straightforward application of the generic design process to the problem of architectural design. The activity diagram in Figure 9-1-1 reproduces and slightly elaborates part of the engineering design process activity diagram from Chapter 2.

As indicated in the diagram, the input to this process is a software requirements specification (SRS), and its output is a software architecture document. A **software architecture document (SAD)** is simply a document that specifies the architecture of a software system. We consider the contents of the SAD next.

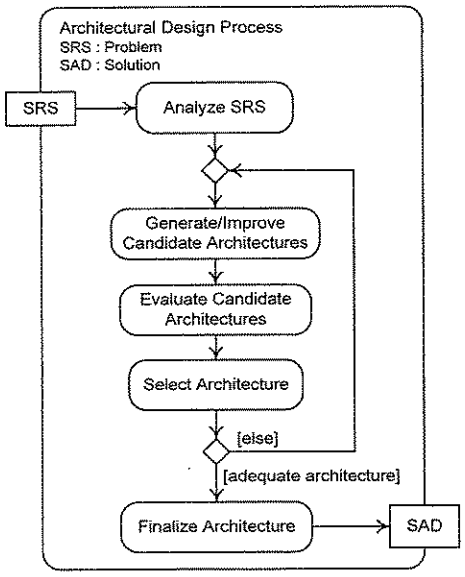

Figure 9-1-1 Architectural Design Process

SAD Contents

The contents of a SAD vary depending on the program being designed and the needs of the development team. A small system with a simple architecture and a good SRS may need only a few architectural design models and minimal supporting material, while a complex system may require much more. The template shown in Figure 9-1-2 is appropriate for documenting the software architectures of small- to medium-sized systems.

1. Product Overview
2. Architectural Models
3. Mapping Between Models
4. Architectural Design Rationale

Figure 9-1-2 Software Architecture Document Template

The template has the following sections:

Product Overview—This section either summarizes the product vision, stakeholders, target market, assumptions, constraints, and business requirements or refers readers to the project mission statement. It is present because readers would have difficulty understanding the software architecture without knowing anything about the target product.

Architectural Models—This section presents the architecture, using a variety of models to represent different aspects or views. It typically uses the design notations discussed later in this chapter.

Mapping Between Models—Sometimes it is difficult to connect different architectural models. This section uses tables and textual explanations to help readers see these connections.

Architectural Design Rationale—This section explains the main design decisions made in arriving at the architecture. Architects have the time and energy to discuss only a few of the many decisions made during architectural design in the SAD. In deciding which decisions to discuss, architects should select those that took a lot of time and effort to make, are crucial for fulfilling important requirements, are puzzling at first glance, or will be hard to change later. Architects should explain the factors affecting each decision, the design alternatives considered, evaluations of the design alternatives, and the reasoning behind the final choice.

The section on architectural models must contain enough information that the models can be understood and used as a basis of detailed design and eventual implementation. In particular, it is important to include appropriate DeSCRIPTR information, as discussed in Chapter 8. A graphic, table, or short textual description generally does not supply all the needed information and must be supplemented by other specifications, usually written in English. Additional information may help readers understand the design. In particular, it may help to include a design rationale listing design alternatives and explaining why the architects chose the alternative they did.

Appendix B contains the AquaLush SAD. It uses the template shown in Figure 9-1-2 and illustrates the information contained in each section.

Quality Attributes

Software architecture is crucial not only for satisfying a product's functional requirements, but also for satisfying its non-functional requirements. Recall from Chapter 5 that non-functional requirements specify properties or characteristics that a software product must have. These are called quality attributes.

> A **quality attribute** is a characteristic or property of a software product independent of its function that is important in satisfying stakeholder needs and desires.

Quality attributes fall into two categories: *development attributes* and *operational attributes*. Development attributes include properties important to development organization stakeholders, such as the following characteristics:

Maintainability—A product's **maintainability** is the ease with which it can be corrected, improved, or ported. Sometimes more specific kinds of maintainability attributes are used, such as modifiability or portability.

Reusability—A product's **reusability** is the degree to which its parts can be used in another software product. A product designed for reuse will have higher reusability.

Operational quality attributes include the following properties:

Performance—A program's **performance** is its ability to accomplish its function within limits of time or computational resources. Programs often must respond to external events within a certain time or must do their jobs using small amounts of memory or processor time.

Availability—A program's **availability** is its readiness for use. A Web server, for example, may need to be available for all but a few minutes a day.

Reliability—A program's **reliability** is its ability to behave in accord with its requirements under normal operating conditions. Any program that handles money or can endanger humans must have high reliability.

Security—A program's **security** is its ability to resist being harmed or causing harm by hostile acts or influences.

Programs can have a wide variety of architectural structures and still satisfy a product's functional requirements, but various architectural structures make it easier or harder to satisfy non-functional requirements. Furthermore, architectures that increase the ability of a program to satisfy some non-functional requirements may decrease its ability to satisfy others. Software architects must consider alternative structures that enable a program to satisfy its functional requirements and select those that allow it to best satisfy its non-functional requirements.

To illustrate, consider a program responsible for matching fingerprints read from scanners against a database to allow people into and out of a secure facility. Besides its functional requirements, this program has some obvious non-functional requirements. For example, it must respond quickly, it must be available the entire time people are entering or leaving the facility, it must match fingerprints fairly reliably, and it must resist attackers.

Many software architectures can meet the functional requirements of such a program, but some will be better able to meet the non-functional requirements. Higher availability can be achieved by having redundant program units, such as a backup database, but this is more complicated, which may decrease reliability. On the other hand, redundant databases may speed processing so this architectural alternative may improve performance. Fingerprint matching can be made more or less reliable depending on the algorithm used, but the more reliable algorithms may take more time, which affects performance. The program may be made more secure by adding units for data encryption and decryption, but this makes it more complicated (decreasing reliability) and slower (decreasing performance).

Architectural Design Challenges

Software architects must record their designs somehow to assist their thinking, share their ideas with others, evaluate their designs, and document them. Architectural constituents are large and abstract, and several kinds of models and notations are needed to represent software architectures fully. We consider how to specify software architectures in the next section of this chapter.

Architectural design, like all design, makes demands on the creativity of designers. Where do ideas for architectures come from? How can architectures be improved? We consider some answers to these questions in Chapter 10.

Software architects must somehow assess the ability of a software architecture to meet its requirements, even though the architecture is an abstract specification and there is no software to run. This is one of the most challenging aspects of architectural design, but there are techniques for evaluating architectures. We survey them in Chapter 10 as well.

A product's architecture should be validated before moving on to detailed design. The last section of Chapter 10 discusses how to validate software architectures with reviews and inspections.

Section Summary

- Architectural design may begin during product design, and it can provide information to product designers and stakeholders that is important for making product design decisions.
- There is no clear boundary between architectural and detailed design.
- There are no accepted standards for the level of abstraction of architectural specifications.
- A **quality attribute** is a software product property independent of a program's function that is important for satisfying stakeholder needs and desires.
- Quality attributes are specified in non-functional requirements.
- Software architects must specify architectural structures that meet both functional and non-functional requirements, paying special attention to the way that alternative structures affect quality attributes.

Review Quiz 9.1

1. How do development organizations affect architectural design, and how do software architectures affect development organizations?
2. What information should be included in a software architecture document?
3. What is the difference between operational and development quality attributes?
4. Name and describe three quality attributes.

9.2 Specifying Software Architectures

Notational Variety

Software architectures are specified using notations that describe static program structure and dynamic program behavior. Both static and dynamic design models are needed in architectural design.

A wide variety of notations can be used to represent software architectures, including several described elsewhere in this book. In particular

- UML activity diagrams, discussed in Chapter 2, can be used to describe the processes that units follow as they interact.
- UML use case diagrams and use case descriptions, presented in Chapter 6, can be used to describe the interactions between architectural constituents.
- UML class diagrams can describe the static architectural structure of small programs. These are discussed in Chapters 7 and 11.
- UML interaction diagrams can describe the communication between architectural constituents. These are discussed in Chapter 12.
- UML state diagrams can describe the states and state changes of major program units. State diagrams are covered in Chapter 13.

In this section we discuss various textual notations for architectural specifications. We also present box-and-line diagrams, a design notation especially useful for describing software architectures. In the next section we discuss notations that can be used in any UML diagram, along with UML package, component, and deployment diagrams, which are useful for both architectural and detailed design modeling.

Table 9-2-1 catalogs how the various notations mentioned previously are used for DeSCRIPTR aspects of architectural specification.

Type of Specification	Useful Notations
Decomposition	Box-and-line diagrams, class diagrams, package diagrams, component diagrams, deployment diagrams
States	State diagrams
Collaborations	Sequence and communication diagrams, activity diagrams, box-and-line diagrams, use case models
Responsibilities	Text, box-and-line diagrams, class diagrams
Interfaces	Text, class diagrams
Properties	Text
Transitions	State diagrams
Relationships	Box-and-line diagrams, component diagrams, class diagrams, deployment diagrams, text

Table 9-2-1 Notations for Architectural Specifications

Textual Specifications

Text is particularly important for specifying responsibilities, interfaces, properties, and relationships. We consider each in turn.

Specifying Responsibilities and Relationships

A unit's responsibilities are usually indicated in part by its name and in part by the symbols used to represent it in various diagrams. Similarly, unit relationships are often indicated by various connectors and by the names of the relationships. Names and symbols provide only part of the information about what a unit is supposed to do and how it is related to other units, though; supplementary information is usually needed.

There is no standard format for specifying responsibilities or relationships. As in all technical writing, responsibility and relationship specifications should be spelled out in simple, clear, and precise sentences. Grammar and spelling should be correct, and the material should be formatted to aid the reader. The AquaLush SAD in Appendix B contains many examples of textual responsibility specifications.

Specifying Interfaces

As mentioned in Chapter 8, an **interface** is a communication boundary between entities, and an **interface specification** describes the mechanism that an entity uses to communicate with its environment. Interface specifications, as descriptions of a means of communication, include specification of three characteristics:

Syntax—The **syntax** of a communications medium specifies the elements of the medium and the ways they may be combined to form legitimate messages. For example, the syntax of a programming language describes the lexical elements in the language (keywords, operators, and so forth), along with rules dictating how these may be combined to form legitimate programs.

Semantics—The **semantics** of a communications medium specify the meanings of messages. For example, the semantics of Java specify that the meaning of the statement `x = x+4` is that the variable `x` has its value increased by four.

Pragmatics—The **pragmatics** of a communications medium specify how messages are used in context to accomplish certain tasks. For example, Java `Collection` objects can store only reference values, not values of primitive types. Thus, for example, `int` values cannot be stored in an `ArrayList`. However, values of primitive types can be wrapped in objects and then stored in `Collections`. For example, an `int` value can be placed in an `Integer` object that can be stored in an `ArrayList`. This is part of the pragmatics of Java `Collections`.

An interface specification should cover the syntax, semantics, and pragmatics of the communications between a module and its environment. Both the messages that can be sent to the module and the messages that the module sends need to be specified.

Sometimes the pragmatics of several messages are best explained together because the module expects a certain **protocol**—that is, a certain ordering

or timing of the messages sent to it. Even when a module does not have a protocol, examples of how to use it are helpful additions to an interface specification. Finally, an interface specification can include an explanation of why the interface is designed as it is.

These considerations lead to the template shown in Figure 9-2-2 for specifying module interfaces.

1. Services Provided
   For each service provided specify its
   a) Syntax
   b) Semantics
   c) Pragmatics
2. Services Required
   Specify each required service by name.
   A service description may be included.
3. Usage Guide
4. Design Rationale

Figure 9-2-2 Interface Specification Template

There are many ways to specify syntax, semantics, and pragmatics. Syntax may be specified very abstractly by simply stating the names, inputs, and outputs of messages. UML class diagram operation specifications provide a more detailed notation. Programming languages provide a very detailed notation for syntactic specification.

One popular method for specifying semantics and pragmatics is to use preconditions and postconditions. A **precondition** is an assertion that must be true at the initiation of an activity or operation, while a **postcondition** must be true at the completion of an activity or operation. Together, pre- and postconditions show how the state of a computation changes when an operation executes, which explains the operation's semantics. Preconditions also state the context in which an operation can be used, which is part of its pragmatics. Postconditions should also state what happens when an operation is used even though its preconditions are violated. This is part of the operation's semantics.

AquaLush Interface Specification Example

Let us consider part of the AquaLush SAD to illustrate textual interface specification. AquaLush has a major constituent called the Device Interface layer. This constituent contains virtual devices that implement interfaces to physical devices (or simulations of them), such as valves, sensors, the parts of the control panel, and so forth. Every virtual device has an interface to which all device drivers of that sort must conform. For example, the virtual valve device has an interface to which all valve device drivers must conform. The advantages of having a device interface layer are discussed in Chapter 10. For now, we focus on one virtual device and its interface: the clock device.

The AquaLush virtual ClockDevice must keep track of the day of the week and the time of the day. A component, such as a master clock, that needs the current day or time of day can query the ClockDevice to obtain this information. The day of the week and the time of the day can also be set. A component that wants to be notified by the ClockDevice that time has passed can register itself as the device's TickListener. The ClockDevice sends messages to its TickListener every minute to notify it that time has passed. The ClockDevice needs to be told that time has passed by another entity that really keeps track of the time: a real (or simulated) clock of some sort. A real clock object must notify the ClockDevice that time has passed at least every minute. It also needs to be able to determine the current actual day and time, but we assume that this feature is supported by the programming language so we do not include it as an interface requirement.

This rather vague English description is made more precise in the interface specification for a ClockDevice shown in Figure 9-2-3 on page 264.

Specifying Properties

Non-functional requirements specify properties or quality attributes for a program as a whole that are then propagated to its parts. It is often difficult to specify quality attributes precisely. For example, maintainability is very hard to specify precisely—stating that a program or its parts must be "easy to change" says almost nothing. Specifications can be made more precise by introducing a target metric. For example, an SRS might specify that new editor features can be added with no more than one person-month of effort on average. This is much better, but how can such a program property be propagated to individual parts during design?

One way to make property specification more precise and easier to work with is to characterize them with a collection of scenarios. In Chapter 6, a **scenario** was defined as a specific interaction between a product and particular individuals that instantiates a use case. Use cases are episodes of interaction between a program and its actors. Scenarios for quality attributes are specific interactions between a program and any entity, including its developers and maintainers. Collections of scenarios typify the interactions relevant to a particular quality attribute. Studying these scenarios helps specify properties and can be the basis for architectural evaluation, which is discussed in the next chapter.

Returning to the maintenance example, the requirements that new editor features can be added with no more than one person-month of effort can be elaborated by creating a set of scenarios (typically three to five) of specific editor features that stakeholders believe are likely to be requested. Architects can consider how each scenario would be implemented to generate sets of modification scenarios for architectural constituents. These sets of scenarios can then be used to specify constituent properties. We will consider scenarios in greater detail in Chapter 10.

Services Provided

All operations with preconditions on parameters throw IllegalArgumentExceptions if the precondition is violated.

Set the time of the day	*Syntax:*	setTime(milTime : int)
	Pre:	milTime is a legitimate military time specification.
	Post:	The clock device is reset to milTime.
Get the time of the day	*Syntax:*	getTime() : int
	Pre:	None.
	Post:	The current time is returned, accurate to the minute, in military time format.
Set the day of the week	*Syntax:*	setDay(d : Day)
	Pre:	None.
	Post:	The clock device is reset to day d.
Get the day of the week	*Syntax:*	getDay() : Day
	Pre:	None.
	Post:	The current day of the week is returned.
Set the ClockDevice listener	*Syntax:*	setListener(l : TickListener)
	Pre:	None.
	Post:	TickListener l will start to receive notifications of the passage of time every minute. Any previous TickListener is replaced.

Services Required

This layer requires a Day enumeration type. The ClockDevice requires some sort of real or simulated device that notifies it when one minute has passed.

Usage Guide

The Clock, not the ClockDevice, is the main time and time notification service provider. The Clock uses the ClockDevice for time notification. To use a ClockDevice:

1. Create a new ClockDevice object. It should be unique.
2. Register the ClockDevice with the simulated hardware or software entity that supplies it with time notifications.

The day of the week and the time of the day may be adjusted at any time.

Design Rationale

AquaLush requirements are satisfied by a clock that keeps track of the day of the week and the time of the day, accurate to one minute. Hence the ClockDevice has only these features.

The real clock time increment does not matter as long as it is no greater than one minute.

Figure 9-2-3 AquaLush ClockDevice Interface

Box-and-Line Diagrams

Perhaps the most widely used architectural design notation has no precise specification and indeed hardly qualifies as a notation at all. It consists of various symbols or icons, called *boxes*, usually connected with various sorts of *lines*. Such graphics are called **box-and-line diagrams**. Figure 9-2-4 is an example.

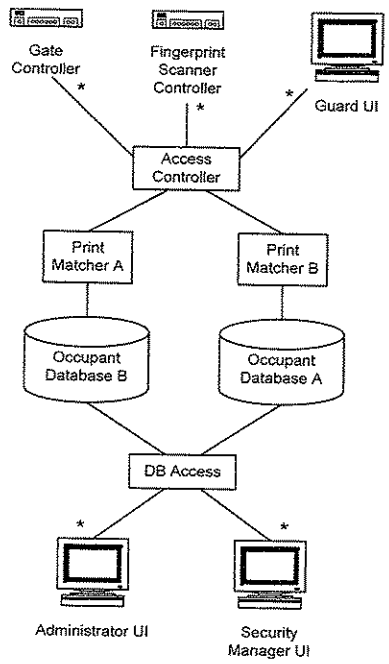

Figure 9-2-4 A Box-and-Line Diagram

This diagram is a static model of the Fingerprint Access System (FAS), a program that helps maintain physical security by controlling and monitoring access to a building using fingerprint scans. The program compares fingerprint scans to those recorded in a database and opens access gates for people whose fingerprints match. A guard can also control the gates. Administrators maintain the database, and security managers can query the database to determine who is in the building. The program's data store is replicated to increase availability and performance.

The boxes in this diagram all refer to software or data store components. The lines indicate interaction relationships between components. The asterisks at the ends of some lines indicate that several instances of such

components may be involved in the interaction. For example, several Gate Controllers may interact with the Access Controller. The box shapes classify the components as user interfaces, device controllers, internal components, or data stores.

Because box-and-line diagrams have no conventions about the meanings of symbols, it is a very good idea to include a legend with every box-and-line diagram. A legend for the FAS diagram in Figure 9-2-4 appears in Figure 9-2-5.

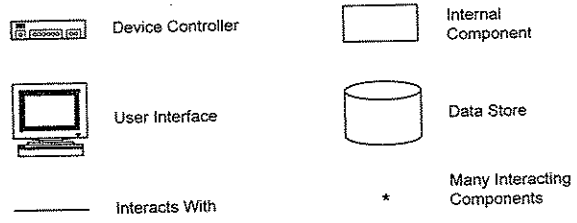

Figure 9-2-5 A Box-and-Line Diagram Legend

Even if a diagram uses a standard notation, marking the notation type on the diagram in place of the legend helps avoid confusion.

Sometimes physical relationships between boxes represent relationships in a box-and-line diagram. For example, it is common to represent layered modules using adjacency. The diagram in Figure 9-2-6 illustrates this case.

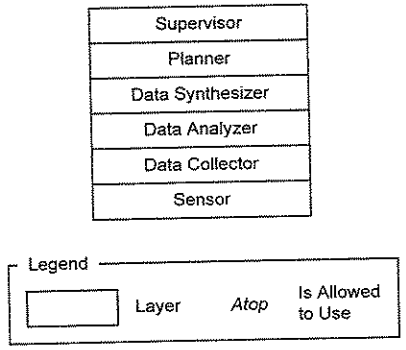

Figure 9-2-6 A Box-and-Line Diagram Showing Layers

Layers are allowed to interact only with the layers below them, so this diagram models the program quite clearly.

Box-and-Line Diagram Uses

As box-and-line diagrams are so loosely specified, they can be used for both static and dynamic modeling, and they can represent any aspect of a program that the architect desires. In practice, they tend to be used mostly for static models that show decomposition into major constituents or subsystems, along with interaction relationships.

Box-and-Line Diagram Heuristics

We have already mentioned the advisability of including a legend explaining the symbols in a box-and-line diagram. In addition, the following heuristics can help make better box-and-line diagrams:

Make box-and-line diagrams only when no standard notation can meet modeling needs. There is no need to invent notations when good standard notations already exist. On the other hand, few standard notations are intended for architectural modeling, though they can be used for this purpose. A notation not meant to model architectures but used for that purpose is often not as good as a box-and-line drawing. For example, UML packages (discussed later in this chapter) can be used to represent architectural layers, but this notation takes longer to draw and is not as easy to read as the stacked boxes in a diagram such as Figure 9-2-6. Consequently, architectural layers are most often, and appropriately, shown using box-and-line diagrams.

Keep the boxes and lines simple. People sometimes get carried away with fancy graphics, but often there is no need to use many fancy symbols in box-and-line diagram. Keeping things simple usually makes the diagram easier to create, read, change, and reproduce, often with no loss of descriptive power.

Make symbols for different things look different. This fundamental rule of technical communication is especially relevant for box-and-line drawings because the architect is responsible for the notation as well as the design. Diagrams without distinct symbols are harder to read and more likely to be misinterpreted. For example, suppose two different kinds of relationships (such as an interaction relationship and a decomposition relationship) are represented by a solid line and a slightly thicker solid line. Readers might not notice the difference between the lines, and they might have a hard time recognizing which is which even if they do. It would be better to use very different line styles (such as a solid line and a dashed line), to add special symbols at the ends of the lines (such as arrowheads at the ends of interaction lines only), or to follow UML practice and put stereotypes on symbols to distinguish them.

Use symbols consistently in different diagrams. Even though each box-and-line diagram can have its own legend explaining its symbols, both the modelers and the model readers will have a hard time keeping symbols straight in different diagrams if they vary too much. By the same token, it is good practice to adopt notational conventions from standard diagrams for box-and-line diagrams. For example, an architect might always use solid lines with labels for relationships on static models, following UML practice for associations.

Use grammatical conventions to name elements. Noun phrases name things, and verb phrases name actions or activities. Diagram elements that represent things should be named with noun phrases, and elements that represent actions, relationships, or interactions should be named with verb phrases.

Don't mix static and dynamic elements. A common mistake when making box-and-line diagrams is to add a few dynamic elements (such as data or control flows) to a static model. This usually messes up the model. Decide before making the model whether it will be a static or dynamic model, and add only elements in accord with this decision.

Heuristics Summary

Figures 9-2-7 and 9-2-8 summarize the heuristics discussed in this section.

- Write good technical prose when specifying architectures.
- Use a template to specify interfaces.
- Specify the syntax, semantics, and pragmatics of interfaces.
- Use preconditions and postconditions to specify semantics and pragmatics.
- Elaborate quality attributes with scenarios.

Figure 9-2-7 Architectural Specification Heuristics

- Make box-and-line diagrams only when no standard notation can meet modeling needs.
- Keep the boxes and lines simple.
- Make symbols for different things look different.
- Use symbols consistently in different diagrams.
- Include a legend explaining the symbols in the diagram.
- Use grammatical conventions to name elements.
- Don't mix static and dynamic elements.

Figure 9-2-8 Box-and-Line Diagram Heuristics

Section Summary

- Several different kinds of notations are used to specify software architectures.
- Text is often used to specify responsibilities and relationships, interfaces, and properties.
- **Interface specifications** should state the **syntax**, **semantics**, and **pragmatics** of services provided and required by a unit.
- **Scenarios** can be used to help articulate component properties.
- **Box-and-line diagrams** use symbols or icons (*boxes*) connected by various kinds of *lines* to model software.

Review Quiz 9.2

1. What diagrams can be used to model collaborations between program parts?
2. Give an example from ordinary English to illustrate the difference between syntax, semantics, and pragmatics.
3. Give an example from everyday life of a protocol governing communication between two people.
4. What are preconditions and postconditions?
5. What sorts of symbols can appear in box-and-line diagrams?

9.3 UML Package and Component Diagrams

UML Common Notations

Several UML notations can be used in any UML diagram. We discuss them here because they are often found in UML models of software architectures and because some of them are needed to discuss UML package and component diagrams.

Notes

A **note** is a dog-eared box connected to any model element by a dashed line. A note may contain arbitrary text. Notes usually contain comments, but they may also contain constraints.

Extension Mechanisms

UML provides several mechanisms for extending the notation. These extension mechanisms allow symbols to take on special meanings or give UML the ability to model things that its basic notation does not support. The main ways the extend UML are the following mechanisms:

Constraints—A **constraint** is a statement that must be true of the entities designated by one or more model elements; in other words, it is a condition or restriction on the target of the model. Constraints are written inside curly brackets in a format not specified by UML. Constraints may appear near a name or model element. Constraints that apply to two model elements may be placed next to a dashed line connecting the elements; constraints applying to more than two elements may be placed in a note connected by dashed lines to all the elements involved.

Properties—A **property** is a characteristic of the entity designated by a model element. Properties are specified in comma-separated lists of tagged values enclosed in curly brackets. A *tagged value* is a name-value pair connected by an equals sign. For example, the property {version=2.2, synchronized=true} associates the value 2.2 with the tag version and the value true with the tag synchronized. If a tag has the Boolean value true then the value and the equals sign can be omitted. For example, the previous property can be abbreviated as {version=2.2, synchronized}. Properties can appear next to the elements that they describe or be connected to them with dashed lines.

Stereotypes—A **stereotype** is a UML model element given more specific meaning. Special icons, colors, or other graphic features can represent stereotyped elements, but placing a stereotype keyword before or above the model element name is the usual way to make one. *Stereotype keywords* are words placed between balanced guillemots, which look like double angle brackets but are distinct symbols. For example, «uses» is a stereotype keyword.

Figure 9-3-1 contains an example of a note, a constraint, a property, and a stereotype.

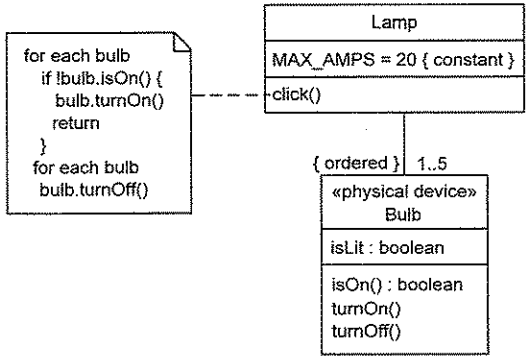

Figure 9-3-1 Stereotypes, Notes, Properties, and Constraints

The Bulb class is stereotyped as a «physical device», which means (let us suppose) that it is a special kind of class that controls a physical device. The MAX_AMPS attribute in the Lamp class has the property of being a constant. The { ordered } constraint specifies that the collection of Bulb instances held by a Lamp instance must be ordered. The note contains pseudocode that specifies the click() operation's behavior.

Dependencies

A dependency is a kind of binary relation that holds between two things, defined as follows.

> A **dependency relation** holds between two entities *D* and *I* when a change in *I* (the *independent entity*) may affect *D* (the *dependent entity*).

For example, suppose class *D* calls one or more operations of class *I*. A change to *I* may affect *D*, so *D* depends on *I*. There are many kinds of dependency relations. The following instances are common examples:

- Module *D* *uses* module *I* when a correct version of *I* must be present for *D* to work correctly.

- Module *D* *depends for compilation* on module *I* (in other words, *D* cannot be compiled without *I*).
- Class *D* *imports* elements from package *I*.

UML represents dependency relations with a *dependency arrow*. The dependency arrow points from the dependent to the independent entity. Dependency arrows may be stereotyped to indicate the precise nature of the dependency relation. The diagram in Figure 9-3-2 shows several dependencies between classes.

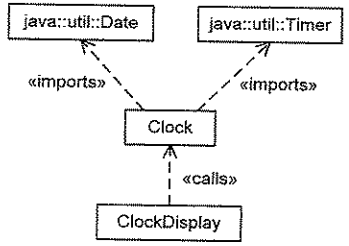

Figure 9-3-2 Some Class Dependencies

Dependencies represent links between individual model elements, not relations between sets of instances, so they do not have association adornments.

Packages

A UML **package** is simply a collection of model elements, called package *members*. The UML package symbol is a tabbed rectangle, or a file folder symbol. Figure 9-3-3 illustrates the package symbol.

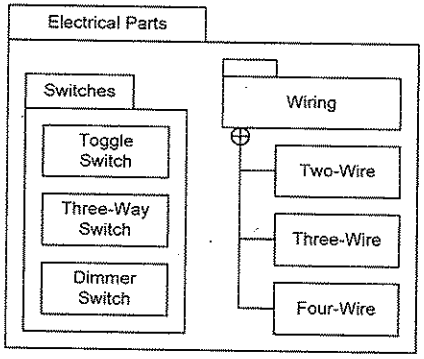

Figure 9-3-3 A Package Example

The members of a package can be shown either inside the rectangle or using a special containment symbol (a circled plus sign) attached to the containing package, with lines running to the members. The package name appears in the tab if the rectangle is occupied or in the rectangle if it is empty. Package symbols can be connected by dependency arrows to show that one package imports or exports members to another.

Package Diagrams

A UML **package diagram** is one whose primary symbols are package symbols. A package diagram may show groupings of use cases, classes, components, or nodes (discussed later in this chapter). A package diagram may also consist of only package symbols.

Modeling Architectures with Package Diagrams

UML package diagrams are useful for making static models of modules, their parts, and their relationships. UML packages are thus useful for architectural modeling. For example, Figure 9-3-4 shows the layers from Figure 9-2-6.

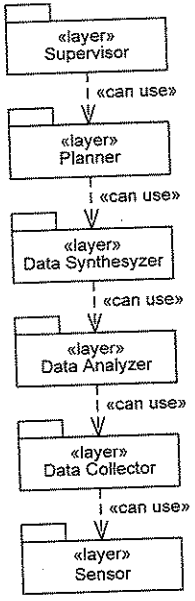

Figure 9-3-4 Layers Represented by Packages

The «layer» stereotype emphasizes that each package is an architectural layer. The *can use* dependency relation indicates that each layer is allowed to use the services of the layer beneath it.

UML packages are not as good as box-and-line diagrams for simply showing architectural layers, but they are very handy when the contents of the layer are shown as well.

Software Components

Reusable parts that can be bought from suppliers and included in software products are an important software development resource. For example, the Java class libraries are collections of such reusable parts, and database management systems, XML parsers, and transaction managers are examples of larger reusable parts. Another advantage of using such parts is that they can be replaced by comparable parts with different properties as product needs change. Developers may design programs with replaceable parts to increase the changeability of their designs.

Such reusable and replaceable parts are called **software components.** Products can be designed with them in mind and built in whole or in part with commercially available or custom-built software components, an approach called **component-based development**. UML reflects the importance of software components by including symbols and diagrams for modeling them.

UML Components and Component Diagrams

A UML **component** is a modular, replaceable unit with well-defined interfaces. Components, like packages, may include classes, but unlike packages they hide their internals to make them as replaceable and reusable as possible.

Components are represented by *component symbols* that resemble class symbols, except that they must either be stereotyped «component» or have a special component icon in their upper right-hand corners. Component names appear inside the component symbol. Figure 9-3-5 illustrates component symbols.

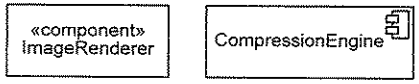

Figure 9-3-5 Component Symbols

A UML **component diagram** shows components, their relationships to their environment, and possibly their internal structure.

UML Interfaces

Interfaces are very important in component diagrams because they define the relationship between a component and its environment. In UML an **interface** is a named collection of public attributes and abstract operations. An **abstract operation** is an unimplemented operation. An abstract operation has a signature indicating its name, parameters, and return values, but no specification of the computation that it carries out when called.

Interfaces are like classes in that they have operations and attributes, but they are unlike classes in that they cannot be instantiated; instead, they must be realized by classes or components. A class or component **realizes** an interface when it includes the interface's attributes and implements its operations.

UML has two kinds of symbols for representing interfaces: One uses a rectangle with compartments for specifying interface details, and the other is an abbreviated form that shows only the interface name. We present the abbreviated form here and cover the other form in Chapter 11.

UML distinguishes two kinds of interfaces depending on an interface's relationship to a class or component:

Provided Interfaces—Interfaces realized by a class or component are **provided interfaces**.

Required Interfaces—Interfaces on which a class or component depends are **required interfaces**.

Each kind of interface is indicated by its own symbol in UML. Provided interfaces are represented by an unfilled circle connected to the class or component providing the interface by a solid line. This is a *ball* or *lollipop* symbol. Required interfaces are represented by half circles connected to the requiring class or component by a solid line. This is the *socket* symbol. Interface names are written beside the circle or half-circle. To illustrate, consider Figure 9-3-6.

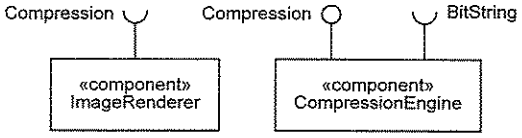

Figure 9-3-6 Provided and Required Interfaces

In this example, the ImageRenderer component displays images on a screen. The CompressionEngine converts between bit strings and various compression formats, such as ZIP, GIF, or JPG. The CompressionEngine provides a Compression interface containing operations such as compressToJPG() and decompressJPG(). The ImageRenderer requires this interface to handle compressed images. Furthermore, the CompressionEngine requires a BitString interface with operations for manipulating bit strings.

The CompressionEngine provides an interface the ImageRenderer requires. A designer can "wire" these components together in a design, specifying that one component uses the services of another, by combining the ball and

socket symbols to form an *assembly connector*. This is illustrated in Figure 9-3-7.

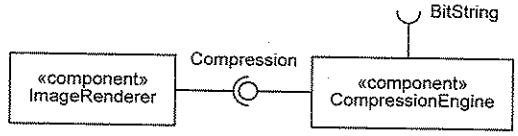

Figure 9-3-7 An Assembly Connector

It should now be clear why the interface symbols are called ball and socket symbols.

Component required and provided interfaces can also be listed inside the component symbol in a compartment below the component name. The interfaces can be listed under the stereotypes «required interfaces» and «provided interfaces». Other compartments can be added as well to document additional information about a component.

Component Internal Structure

So far, we have concentrated on the UML notation for representing components and their relationships to their environment. However, components typically encapsulate classes and perhaps other components, and this structure may need to be designed as well. UML provides the means for modeling the constituents of components and their relationships to one another.

Besides the component name, the main compartment of a component symbol can contain class and component diagram symbols. In addition, special symbols are provided to connect the nested symbols with the interfaces that the component uses to interact with its environment.

A *delegation connector* ties a component interface to one or more internal classes or components that realize or use the interface. Delegation connectors are solid arrows stereotyped «delegate». A delegation connector may extend from an external interface lollipop to an internal class, component, or lollipop to indicate the constituent that realizes the provided interface. A delegation connector may also extend from an internal class, component, or interface socket to an external interface socket. For example, the component diagram shown in Figure 9-3-8 on page 276 shows the internal structure of the CompressionEngine component.

This diagram shows that the CompressionEngine has three internal parts handling different kinds of compression. Services provided by the Compression interface are actually performed by the ZIPcompression component or one of the compression classes. The GIF and JPG compression classes require the BitString interface.

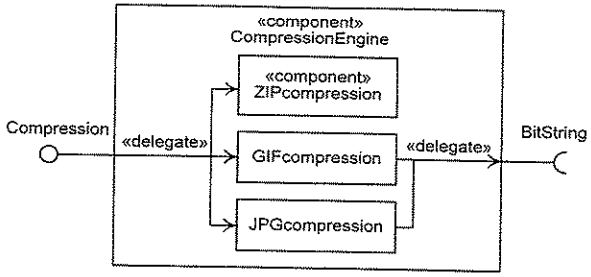

Figure 9-3-8 Component Internals

External interface and delegation connectors may meet at a *port*, which is an interaction point between a component and its environment. Ports are drawn as rectangles on component borders. Their use is optional, however, so we have left them out of the component diagram in Figure 9-3-8.

Modeling Architectures with Components

Architects may decide that certain portions of a program should be reusable and replaceable sub-systems that are either purchased from software component vendors, obtained from previous projects, or developed as part of the product. In any case, such reusable and replaceable architectural constituents can be modeled in UML as components. It may be that an entire program is composed of components; in this case, architects can use UML component diagrams to show decomposition and component assembly relationships.

When designers must create a new component, they can use component diagrams to model the static internal structure of the component. The internal structures of components may be at too low a level of abstraction to be of interest during architectural design. If so, these diagrams can be used for component detailed design.

Heuristics Summary

Figure 9-3-9 summarizes the heuristics discussed in this section.

- Use notes, constraints, properties, and stereotypes to add information to UML models.
- Use stereotypes to name dependencies.
- Use packages to group elements in static models.
- Use package diagrams to model architectural modules, their parts, and their relationships.
- Use components to model reusable and replaceable program parts.
- Use component diagrams to represent the assembly relationships between components and their internal static structure.

Figure 9-3-9 UML Diagramming Heuristics

Section Summary

- Comments can be added to UML diagrams as **notes**.
- UML can be extended using **constraints**, **properties**, and **stereotypes**.
- A **dependency relation** can be shown between UML elements using a dependency arrow.
- UML **packages** are collections of items.
- UML **package diagrams** have packages as their primary symbols.
- In UML, a **component** is a modular, replaceable unit with well-defined interfaces.
- A UML **component diagram** shows components, their relationships to their environment, and possibly their internal structures.
- A UML **interface** is a named collection of public features and obligations.
- UML distinguishes between **provided interfaces** realized by a component or class and **required interfaces** used by a component or class.
- Components are defined by their provided and required interfaces.

Review Quiz 9.3

1. What can notes be attached to in UML?
2. Give two examples, different from those in the text, of dependency relations.
3. How are the contents of packages represented in UML?
4. Can UML components be nested?
5. How are required and provided interfaces represented?
6. What are assembly and delegation connectors?

9.4 UML Deployment Diagrams

Logical and Physical Architecture

So far we have been concerned with notations for modeling what might be called a product's **logical architecture**, which is the configuration of a product's major constituents and their relationships to one another in abstraction from the product's implementation as code and its execution on actual machines. Also important, especially for products deployed on several computers, is **physical architecture**, which is the realization of a product as code and data files residing and executing on computational resources.

Physical architecture can be modeled with box-and-line diagrams. UML also provides a notation for physical architecture modeling, called deployment diagrams.

Artifacts and Nodes

A UML **artifact** is any physical representation of data used or produced during software development or software product operation. Examples of artifacts are files, documents, messages, database tables, program code, and diagrams.

Artifacts have types and instances. For example, the source code file prog.java may exist in multiple copies on various storage media. Each of these copies is an instance of the single abstract file type prog.java.

Artifacts are represented in UML by rectangles containing the artifacts' names and marked with either the stereotype «artifact» or a special artifact icon in the upper right-hand corners. Artifact types have non-underlined names, while artifact instances have underlined names. The diagram in Figure 9-4-1 illustrates artifact symbols.

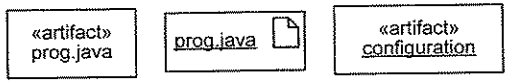

Figure 9-4-1 Artifact Symbols

The symbol on the left represents the artifact type prog.java, while the symbol in the middle represents an instance of this type. The dog-eared page is the artifact icon. The symbol on the right is an instance of an artifact called configuration.

Artifacts are the physical manifestations of the abstract software entities in our models. UML uses a dependency arrow stereotyped «manifest» to indicate the relationship between an abstract model element and the artifacts that realize it as a physical object.

A **node** is a computational resource. There are two kinds of nodes:

Device—A device is a physical processing unit, such as a computer, telephone, refrigerator controller, and so forth.

Execution Environment—An execution environment is a software system that implements a virtual machine. For example, an operating system, a database system, and the Java Virtual Machine are all execution environments.

A node is a real or virtual machine. Nodes are represented in UML by box or slab icons. Devices are stereotyped with «device» and execution environments either with the stereotype «execution environment» or, more often, with a stereotype describing the virtual machine, such as «OS» or «JVM».

Like artifacts, nodes have types and instances. For example, WebServer may be a type of machine, and server1, server2, and server3 may be instances of that type. Node types are labeled with their names. Node instances are labeled with underlined identifiers of the form *name : type*. The *name* may be left off, indicating an unnamed instance of the type, or the *: type* may be left off, indicating a named instance with an unspecified type.

The diagram in Figure 9-4-2 illustrates node symbols.

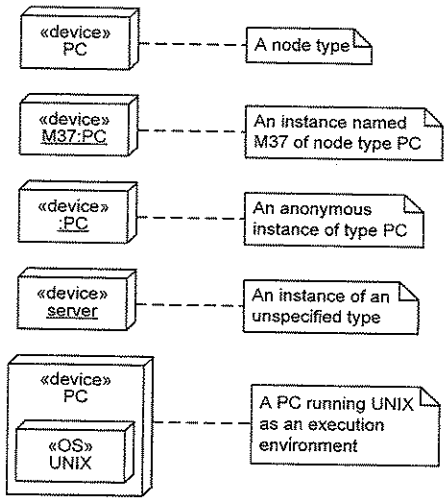

Figure 9-4-2 Node Symbols

Nodes may be nested, as indicated by the last example. Usually, the outer node is a device and the inner nodes are various kinds of execution environments.

Deployment Diagrams

A **deployment diagram** models computational resources, the communication paths between them, and the artifacts that reside and execute on them. Deployment diagrams represent computational resources with node type or instance symbols. Communication connections between nodes are shown with *communication paths*, which are solid lines. Like an association line, a communication path can be labeled with the name of the communication link, and it can have multiplicities and role names at its ends.

Artifacts are deployed to nodes where they reside and may execute. The deployment relationships between artifacts and nodes can be shown in three ways: artifact symbols can be placed inside node symbols, artifacts names can be listed inside node symbols, or artifact symbols can be connected to node symbols with dependency arrows stereotyped «deploy». The deployment diagram in Figure 9-4-3 on page 280 illustrates these conventions.

This diagram depicts the physical architecture of a system for playing games over the Internet. A game player runs a GameClient program on a ClientPC that communicates with a ServerPC using TCP/IP. The ServerPC runs a GameServer program that directs play and mediates communication among the players. The GameServer consults a GameData database to

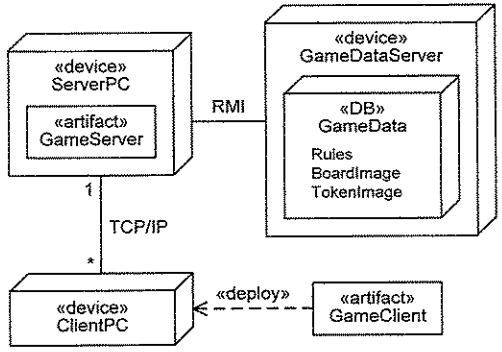

Figure 9-4-3 A Deployment Diagram

obtain the information it needs to oversee a game. The GameData database contains Rules, BoardImage, and TokenImage artifacts. The database is an execution environment that runs on its own GameDataServer computer. The ServerPC and GameDataServer communicate via Remote Method Invocation (RMI).

Deployment Diagram Uses

Deployment diagrams show the real and virtual machines used in a system, the paths over which they communicate, the program and data files realizing the system, and where programs run and data reside. Deployment diagrams thus provide a rich notation for modeling physical architecture.

Deployment diagrams are useful during architectural design, particularly for distributed systems. They can also be helpful in modeling physical deployment during detailed design.

Section Summary

- **Artifacts** are physical representations—mainly files of some sort—of data used or produced in software development or operation.
- A **node** is a computational resource.
- UML **deployment diagrams** show nodes, communication paths between them, and the artifacts that reside and execute on them.
- Deployment diagrams are useful for representing **physical architecture**, which is the realization of a product as code and data files residing and executing on computational resources.

Review Quiz 9.4

1. What is the difference between artifact types and instances?
2. How does the UML notation for representing artifact instances differ from the notation for representing node instances?
3. How can the deployment of an artifact to a node be shown in UML deployment diagrams?

Chapter 9 Further Reading

Section 9.1 Different views about the amount of detail appropriate for architectural design are evident in various software architecture texts, such as [Bass et al. 2003], [Bosch 2000], [Clements et al. 2003], and [Shaw and Garlan 1996]. Bass et al. [2003] discuss extensively the reciprocal influences between architecture and organizations. Bass et al. [2003] and Bosch [2000] discuss the architectural design process in detail. Clements et al. [2003] provide a much more complete SAD template appropriate for larger systems.

Section 9.2 The best book on architectural design notations and documentation is [Clements et al. 2003], which goes far beyond our brief survey in this section. Box-and-line diagrams are discussed in [Shaw and Garlan 1996].

Section 9.3-4 UML extension mechanisms, dependency relations, and package, component, and deployment diagrams are discussed in most UML books, including [Bennett et al. 2001], [Booch 2005], [OMG 2003], [OMG 2004], and [Rumbaugh 2004].

Chapter 9 Exercises

Section 9.1

1. *Fill in the blanks*: Architectural design must take account of both _____ and _____ requirements. Any number of architectural structures may allow a program to satisfy its functional requirements, but only _____ of these will allow it to also satisfy its non-functional requirements. Software architects must consider _____ to find those specifying a program that can satisfy both its functional and non-functional requirements.
2. Suppose you are writing software for a radio station that manages its playlists. The program will generate candidate playlists from a record library automatically and station personnel can then check and modify them. Disc jockeys must also be able to change playlists when they are used because what is actually played is often different from what is planned. The playlists are then used to generate reports for paying royalties. You must decide what sort of data structure to use to store playlists. Make a choice and write a design rationale. Your rationale should explain the factors that went into your decision, the design alternatives you considered, your evaluation of design decisions, and the reasoning for your final choice.
3. Classify the following specifications by quality attribute. Choose from the attributes maintainability, reusability, performance, availability, reliability, and security.
   (a) The program must never lose data that has been saved to persistent store.
   (b) Developers must be able to incorporate at least 60% of the StereoColor product code in the StereoColor Deluxe product.
   (c) The program must be able to respond to queries between the hours of 10 A.M. and 7 P.M. Eastern Standard Time.

(d) The program must be able to execute 9,000 transactions per minute and two million transactions per day.
(e) The program must require passwords from all users.
(f) When a transcript is requested, the program must scan the student record to detect tampering before producing the transcript.

Section 9.2

4. Classify the following specifications as syntactic, semantic, or pragmatic:
   (a) A Java interface may contain only static final variable declarations and public method header declarations.
   (b) A Java synchronized method can execute only when it has a lock on the object in which it resides.
   (c) Only one class name can follow the `extends` keyword.
   (d) In Java, a static method cannot access non-static attributes.
   (e) Null values can be added to a Java `Vector`.
   (f) The `length()` method of an array should be used to control loops that process each element of the array.
   (g) Java has no pointers, so linked structures must be implemented using references.
   (h) In Java, all `RuntimeExceptions` are unchecked.
5. Write an interface for a stack module using the interface specification template in Figure 9-2-2.
6. Write an interface for a hash table module using the interface specification template in Figure 9-2-2.
7. Compilers have a standard architecture consisting of a Tokenizer that converts incoming program text to tokens (or meaningful language units), a Parser that analyzes the syntax of the program and produces a syntax tree, a Code Generator that examines the syntax tree and produces object code, and an Optimizer that processes the object code to make it more efficient. Make a box-and-line diagram modeling this architecture.
8. A common way to design many applications that manipulate a database is to have a module for the user interface, a module for data processing, and a module to access the database. This is called a *three-tier architecture*. Draw a box-and-line diagram of a three-tier architecture.

Section 9.3

9. Make a UML package diagram showing a three-tier architecture (see the last exercise).
10. Make a UML class diagram with package symbols to model the java.util.regex package. You need only show class name compartments.
11. Draw a UML class diagram in which you have an attribute with a {constant} property, an operation with a {synchronized} property, and two associations with an {xor} (exclusive or) constraint.

12. Compare and contrast module interfaces, UML interfaces, and Java interfaces.
13. A software component vendor sells a product that provides text indexing and searching. The product provides a Boolean search interface and an SQL interface, and it requires file operations to store its index. Draw a component diagram showing this product and its provided and required interfaces.
14. *Find the errors*: Find at least four errors in the component diagram in Figure 9-E-1.

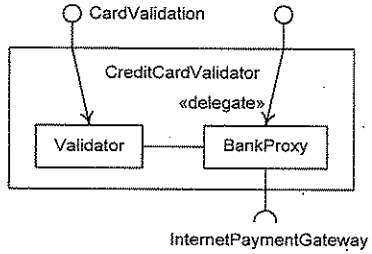

Figure 9-E-1 An Erroneous Component Diagram for Exercise 14

Section 9.4

15. An egg timer program is implemented in `EggTimer.java` and `Pulse.java` files. These files are compiled into class files. A `manifest` file is used by the `jar` program to create an executable `EggTimer.jar` file. Make a diagram to illustrate the artifacts involved in this process.
16. Make a deployment diagram showing the computers (as nodes) and programs (as artifacts) involved when you use your computer to access http://java.sun.com. You may assume that an instance of the Apache program is running on a Sun Web server computer.
17. Consider two architectures for an instant messaging system. Both architectures have a messaging server running somewhere on a TCP/IP network, and both have messaging clients running on computers somewhere on the network. In one architecture, the clients obtain connections to other clients through the server and also send all messages through the server. In the other architecture, the clients obtain connections through the server, but all messages are sent directly between the clients. Draw deployment diagrams depicting these alternative architectures.
18. *Find the errors*: Find at least four errors in the deployment diagram in Figure 9-E-2.

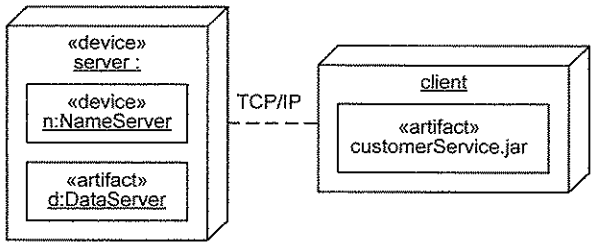

Figure 9-E-2 An Erroneous Deployment Diagram for Exercise 18

Research Project

19. Consult software engineering and software architecture texts to write a glossary of quality attributes. Each entry should include a definition of the attribute and an example illustrating it.

Chapter 9 Review Quiz Answers

Review Quiz 9.1

1. Development organizations influence architectural design in the following ways: the structure of the organization may be reflected in the structure of the architecture; strengths and expertise of the organization may be reflected in the architecture; the assets, standards, and practices of the organization may influence the architecture; and the experience, knowledge, and skills of the organization's architects influence the architecture.
   A program's architecture may influence a development organization because groups may be formed to implement and support architectural constituents; people may be hired or trained to implement and maintain the architecture; assets may be created, practices altered, and standards created or modified to accommodate the architecture; and architects learn and change based on their design experiences.
2. A software architecture document should have (or refer to) a product overview, present architectural models, provide mappings between the models, explain the design rationale, and include a glossary.
3. Development quality attributes are program properties of interest to development stakeholders (developers and their managers), such as maintainability and portability. Operational quality attributes are program properties of interest to non-development stakeholders (clients, purchasers, users, and so forth), such as performance, reliability, and security.
4. Availability is the readiness of a program for use. Maintainability is the ease with which a product can be changed. Performance is the ability of a program to do its job within resource limits. Reliability is the ability of a program to function according to its specification when used normally. Reusability is the degree to which artifacts created during product development can be used in developing other products. Security is the ability of a program to resist attack.

Review Quiz 9.2

1. Collaborations between program parts can be modeled using UML sequence, communication, and activity diagrams; use case models; data flow diagrams; and box-and-line diagrams.

2. English syntax specifies that words must appear in a certain order. For example, "Snake a Peter is" is not syntactically correct, but "Peter is a snake" is correct. Semantics specify the meanings of syntactically correct expressions, so the sentence "Peter is a snake" means what it does by virtue of the semantics of English. Pragmatics specify how expressions can be used to accomplish tasks. For example, saying that Peter is a snake when discussing someone's pet conveys the species of the pet. Saying that Peter is a snake when discussing someone's personality states a metaphor.
3. A simple example of a protocol is the exchange that occurs at the start and end of a phone call. When people answer the phone, they say something to indicate that they are present, often giving their name or organization as well. The caller then proceeds with business. At the end of the conversation, one person says goodbye, the other person acknowledges with another goodbye, and both parties hang up.
4. A precondition is an assertion that must be true before an activity or operation begins. A postcondition is an assertion that must be true after an activity or operation finishes.
5. Box-and-line diagrams are composed of boxes (icons or symbols) and lines of various sorts.

Review Quiz 9.3

1. In UML, notes can be attached to any model element.
2. Dependency relations that often show up in UML models include the *call* relation (when one entity invokes an operation of another), the *instantiate* relation (when an object is an instance of a class), the *manifest* relation (when code realizes a model element), the *deploy* relation (when a file is stored or executed on a computational resource), and the *extend* relation (when one entity augments the behavior of another).
3. UML package contents are represented in two ways: Either the contents are placed within the main rectangle of the package symbol, or they are connected by a line to a special symbol (a circled cross) attached to the package symbol.
4. UML components can be nested. Component symbols can contain components, classes, interfaces, and associations between them, as well assembly connectors.
5. Required interfaces are represented by socket symbols, which are half circles connected to the requiring component or class by a solid line. Provided interfaces are represented by ball or lollipop symbols, which are unfilled circles attached to the providing class or component by a solid line.
6. An assembly connector is formed when an interface ball symbol is fitted into an interface socket symbol. This connector shows how the interface needs of one component or class, expressed by a required interface, are met by another component or class that provides the needed interface. A delegation connector is used inside component symbols to attach an external provided interface symbol to an internal entity that supplies it for the component, or to attach an internal component or class requiring an interface to an external required interface symbol. This connector shows how a component's provided or required interfaces are related to the components or classes it encapsulates.

Review Quiz 9.4

1. An artifact instance is a physical entity with a unique spatiotemporal location; for example, the copy of the Linux operating system presently residing on the disk drive of my computer. There is another copy of this operating system residing in the main memory (and the swap space) that is actually being executed. These two instances of Linux are not identical because they are in different places, but they are the same in some sense because they are both instances of the same artifact type. An artifact type is thus a kind of physical entity.
2. Artifact instances and types have the same names, but artifact instance names are underlined and artifact type names are not. For example, Key.java is an artifact type and Key.java is an artifact instance (of that type). Node type and instance names are different. Node types are labeled with the type name, with no underlining. Node instances are labeled with an identifier of the form *name : type*, where *name* may be omitted (indicating an unnamed instance of the type) or *: type* may be omitted (indicating a named instance of an unspecified type). For example, junior : SparcStation is a node called junior that is an instance of a SparcStation, magilla is the name of a node of unspecified type, and : IntelPC is an unnamed instance of type IntelPC.
3. The deployment of an artifact on a node can be shown in three ways in a deployment diagram: the artifact symbol can be placed within the node symbol, the artifact symbol can appear outside the node symbol but be attached to it by a dependency arrow from the artifact to the node stereotyped «deploy», and the artifact name can be listed inside the node symbol.

10 Architectural Design Resolution

Chapter Objectives

This chapter continues our survey of architectural design, focusing specifically on the steps in the architectural design resolution process, as illustrated in Figure 10-O-1.

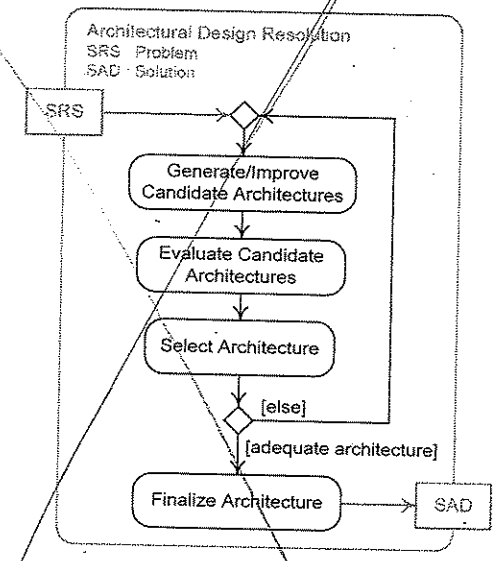

Figure 10-O-1 Architectural Design Resolution

The process is illustrated using the AquaLush architecture.

By the end of this chapter you will be able to

- List and explain several techniques for generating and improving a software architecture;
- Generate architectural alternatives by determining functional components or determining components based on quality attributes;
- Improve architectural designs by combining aspects of different design alternatives;
- Evaluate architectural design alternatives using scenarios and prototypes;