A Pragmatic Introduction to UML

Learning

UML 2.0

O'REILLY®

Russ Miles & Kim Hamilton

CHAPTER 12

Managing and Reusing Your System's Parts: Component Diagrams

When designing a software system, it's rare to jump directly from requirements to defining the classes in your system. With all but the most trivial systems, it's helpful to plan out the high-level pieces of your system to establish the architecture and manage complexity and dependencies among the parts. Components are used to organize a system into manageable, reusable, and swappable pieces of software.

UML component diagrams model the components in your system and as such form part of the development view, as shown in Figure 12-1. The development view describes how your system's parts are organized into modules and components and is great at helping you manage layers within your system's architecture.

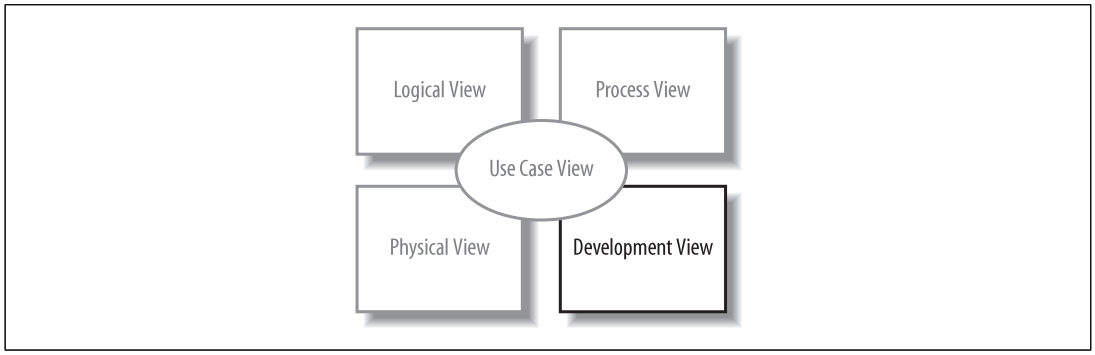


Figure 12-1. The Development View of your model describes how your system's parts are organized into modules and components

What Is a Component?

A *component* is an encapsulated, reusable, and replaceable part of your software. You can think of components as building blocks: you combine them to fit together (possibly building successively larger components) to form your software. Because of this, components can range in size from relatively small, about the size of a class, up to a large subsystem.

Good candidates for components are items that perform a key functionality and will be used frequently throughout your system. Software, such as loggers, XML parsers, or online shopping carts, are components you may already be using. These happen to be examples of common third-party components, but the same principles apply to components you create yourself.

In your own system, you might create a component that provides services or access to data. For example, in a CMS you could have a conversion management component that converts blogs to different formats, such as RSS feeds. RSS feeds are commonly used to provide XML-formatted updates to online content (such as blogs).

In UML, a component can do the same things a class can do: generalize and associate with other classes and components, implement interfaces, have operations, and so on. Furthermore, as with composite structures (see Chapter 11), they can have ports and show internal structure. The main difference between a class and a component is that a component generally has bigger responsibilities than a class. For example, you might create a user information *class* that contains a user's contact information (her name and email address) and a user management *component* that allows user accounts to be created and checked for authenticity. Furthermore, it's common for a component to contain and use other classes or components to do its job.

Since components are major players in your software design, it's important that they are loosely coupled so that changes to a component do not affect the rest of your system. To promote loose coupling and encapsulation, components are accessed through interfaces. Recall from Chapter 5 that interfaces separate a behavior from its implementation. By allowing components to access each other through interfaces, you can reduce the chance that a change in one component will cause a ripple of breaks throughout your system. Refer back to Chapter 5 for a review of interfaces.

A Basic Component in UML

A component is drawn as a rectangle with the `<<component>>` stereotype and an optional tabbed rectangle icon in the upper righthand corner. Figure 12-2 shows a `ConversionManagement` component used in the CMS that converts blogs to different formats and provides feeds such as RSS feeds.

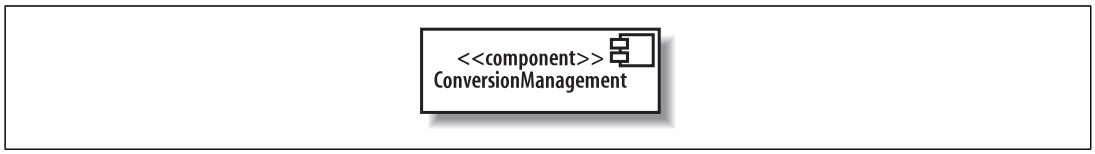


Figure 12-2. The basic component symbol showing a ConversionManagement component

In earlier versions of UML, the component symbol was a larger version of the tabbed rectangle icon, so don't be surprised if your UML tool still shows that symbol.

You can show that a component is actually a subsystem of a very large system by replacing `<<component>>` with `<<subsystem>>`, as shown in Figure 12-3. A *subsystem* is a secondary or subordinate system that's part of a larger system. UML considers a subsystem a special kind of component and is flexible about how you use this stereotype, but it's best to reserve it for the largest pieces in your overall system, such as a legacy system that provides data or a workflow engine in the CMS.

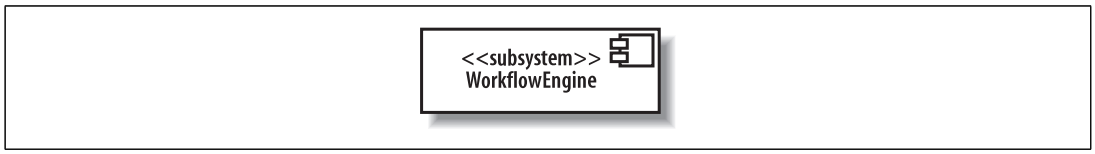


Figure 12-3. You can substitute the <<subsystem>> stereotype to show the largest pieces of your system

Provided and Required Interfaces of a Component

Components need to be loosely coupled so that they can be changed without forcing changes on other parts of the system—this is where interfaces come in. Components interact with each other through provided and required interfaces to control dependencies between components and to make components swappable.

A *provided interface* of a component is an interface that the component realizes. Other components and classes interact with a component through its provided interfaces. A component's provided interface describes the services provided by the component.

A *required interface* of a component is an interface that the component needs to function. More precisely, the component needs another class or component that realizes that interface to function. But to stick with the goal of loose coupling, it accesses the class or component through the required interface. A required interface declares the services a component will need.

There are three standard ways to show provided and required interfaces in UML: ball and socket symbols, stereotype notation, and text listings.

Ball and Socket Notation for Interfaces

You can show a provided interface of a component using the ball symbol introduced in Chapter 5. A required interface is shown using the counterpart of the ball—the socket symbol—drawn as a semicircle extending from a line. Write the name of the interface near the symbols.

Figure 12-4 shows that the `ConversionManagement` component provides the `FeedProvider` and `DisplayConverter` interfaces and requires the `DataSource` interface.

The ball and socket notation is the most common way to show a component's interfaces, compared with the following techniques.

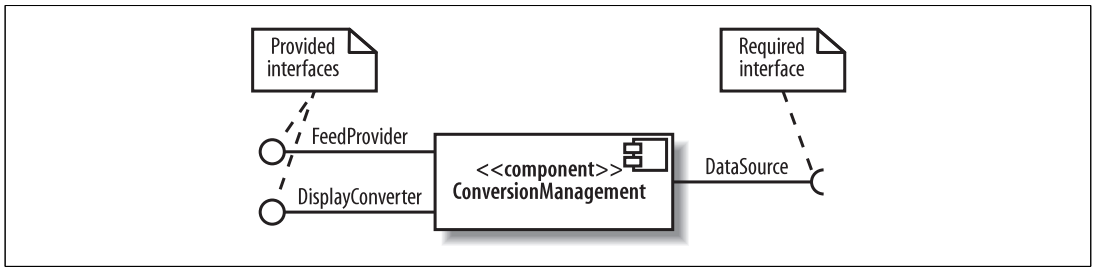


Figure 12-4. The ball and socket notation for showing a component's provided and required interfaces

Stereotype Notation for Interfaces

You can also show a component's required and provided interfaces by drawing the interfaces with the stereotyped class notation (introduced in Chapter 5). If a component realizes an interface, draw a realization arrow from the component to the interface. If a component requires an interface, draw a dependency arrow from the component to the interface, as shown in Figure 12-5.

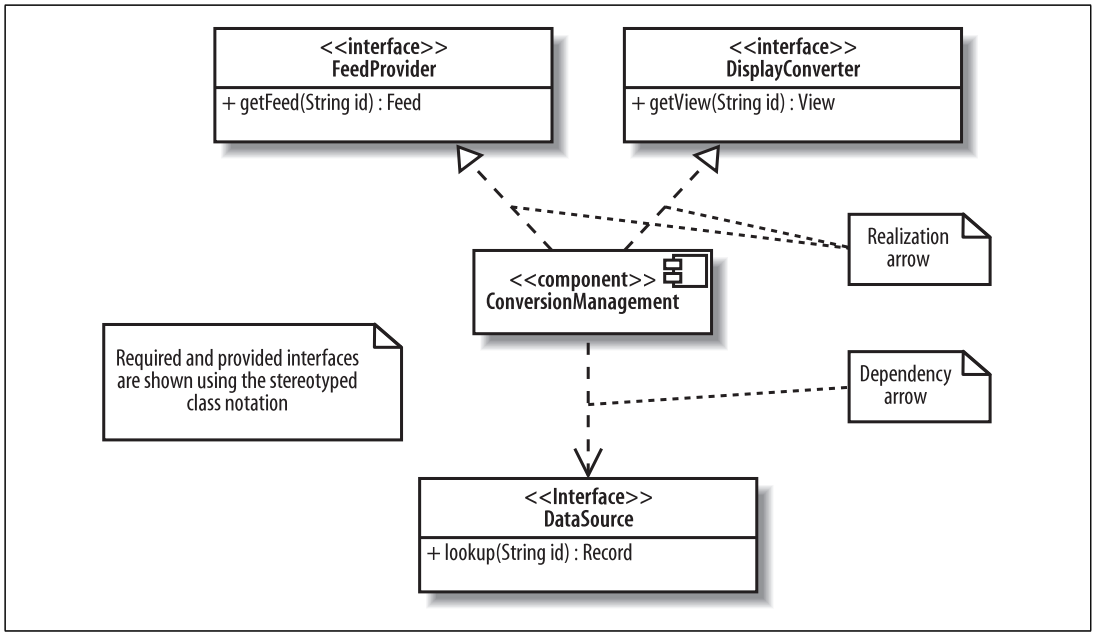


Figure 12-5. The stereotyped class notation, showing operations of the required and provided interfaces

This notation is helpful if you want to show the operations of interfaces. If not, it's best to use the ball and socket notation, since it shows the same information more compactly.

Listing Component Interfaces

The most compact way of showing required and provided interfaces is to list them inside the component. Provided and required interfaces are listed separately, as shown in Figure 12-6.

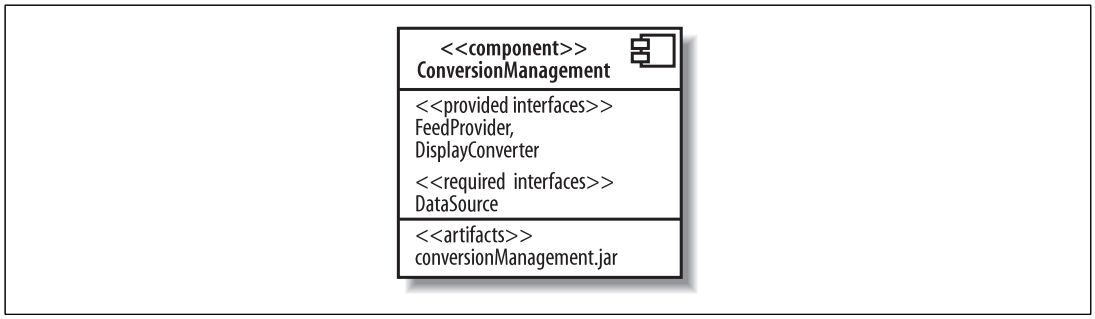


Figure 12-6. Listing required and provided interfaces within the component is the most compact representation

This notation additionally has an `<<artifacts>>` section listing the artifacts, or physical files, manifesting this component. Since artifacts are concerned with how your system is deployed, they are discussed in deployment diagrams (see Chapter 15). Listing the artifacts within the component is an alternative to the techniques shown in Chapter 15 for showing that artifacts manifest components.

Deciding when to use which notation for required and provided interfaces depends on what you're trying to communicate. This question can be answered more fully when examining components working together.

Showing Components Working Together

If a component has a required interface, then it needs another class or component in the system to provide it. To show that a component with a required interface depends on another component that provides it, draw a dependency arrow from the dependent component's socket symbol to the providing component's ball symbol, as shown in Figure 12-7.

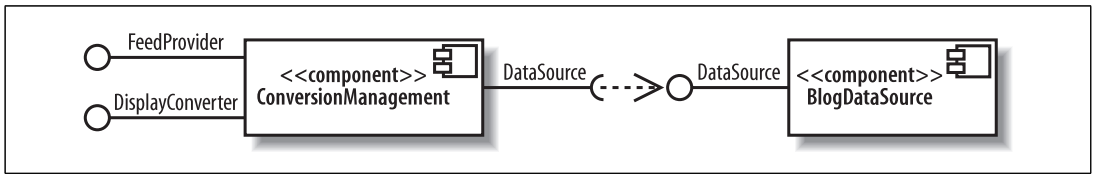


Figure 12-7. The ConversionManagement component requires the DataSource interface, and the BlogDataSource component provides that interface

As a presentation option for Figure 12-7, your UML tool may let you get away with snapping the ball and socket together (omitting the dependency arrow), as shown in

Figure 12-8. This is actually the assembly connector notation, which is introduced later in this chapter.

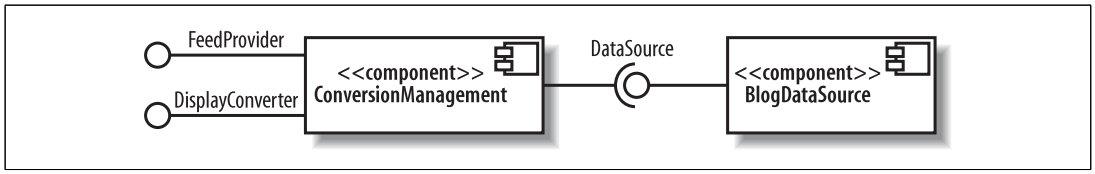


Figure 12-8. Presentation option that snaps the ball and socket together

You can also omit the interface and draw the dependency relationship directly between the components, as shown in Figure 12-9.

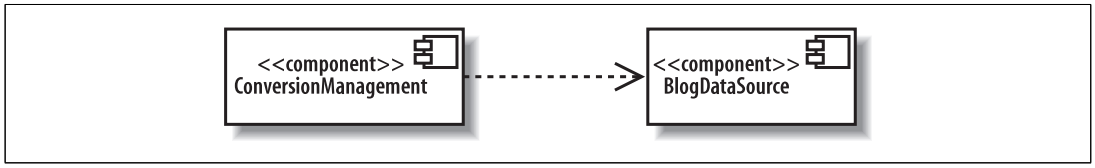


Figure 12-9. You can draw dependency arrows directly between components to show a higher level view

The second notation (omitting the interface, shown in Figure 12-9) is simpler than the first (including the interface, shown in Figure 12-7), so you may be tempted to use that as a shorthand, but keep in mind a few factors when choosing how to draw component dependencies.

Remember that interfaces help components stay loosely coupled, so they are an important factor in your component architecture. Showing the key components in your system and their interconnections through interfaces is a great way to describe the architecture of your system, and this is what the first notation is good at, as shown in Figure 12-10.

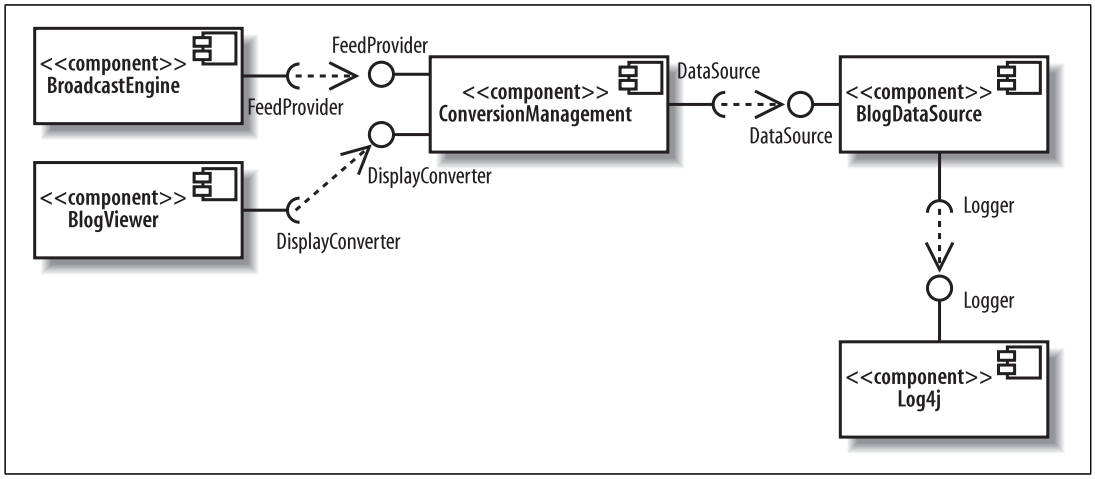


Figure 12-10. Focusing on the key components and interfaces in your system

The second notation is good at showing simplified higher level views of component dependencies. This can be useful for understanding a system's configuration management or deployment concerns because emphasizing component dependencies and listing the manifesting artifacts allows you to clearly see which components and related files are required during deployment, as shown in Figure 12-11.

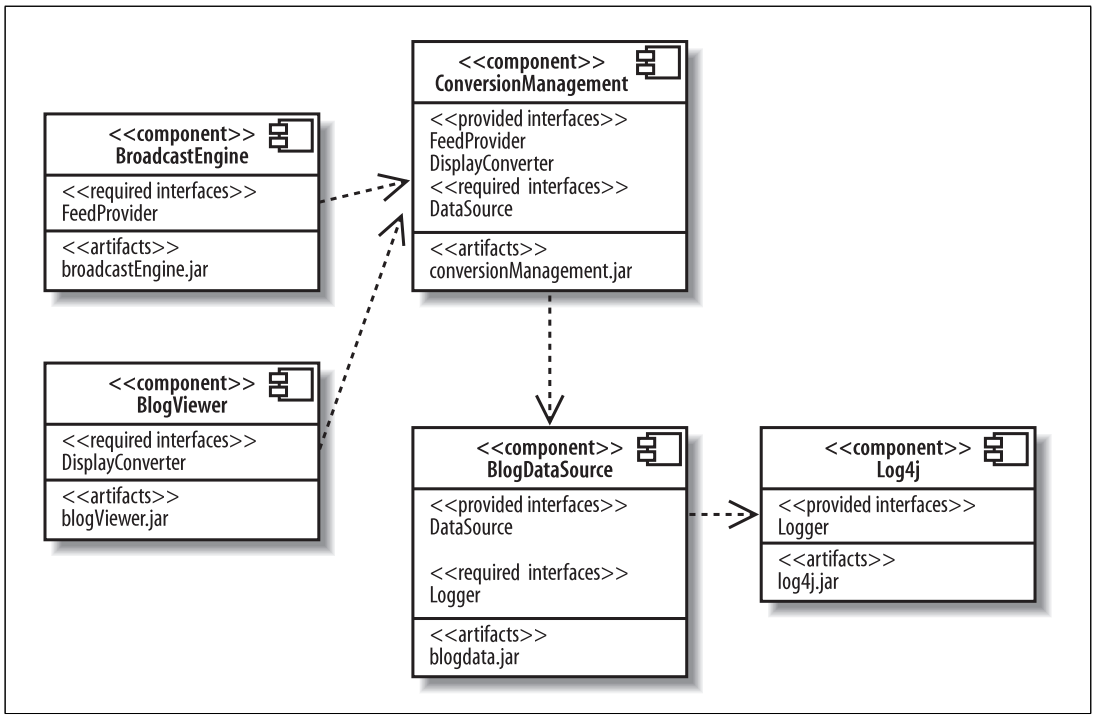


Figure 12-11. Focusing on component dependencies and the manifesting artifacts is useful when you are trying control the configuration or deployment of your system

Classes That Realize a Component

A component often contains and uses other classes to implement its functionality. Such classes are said to *realize* a component—they help the component do its job.

You can show realizing classes by drawing them (and their relationships) inside the component. Figure 12-12 shows that the `BlogDataSource` component contains the `Blog` and `Entry` classes. It also shows the aggregation relationship between the two classes.

You can also show a component's realizing classes by drawing them outside the component with a dependency arrow from the realizing class to the component, as shown in Figure 12-13.

The final way to show realizing classes is to list them in a `<<realizations>>` compartment inside the component, as shown in Figure 12-14.

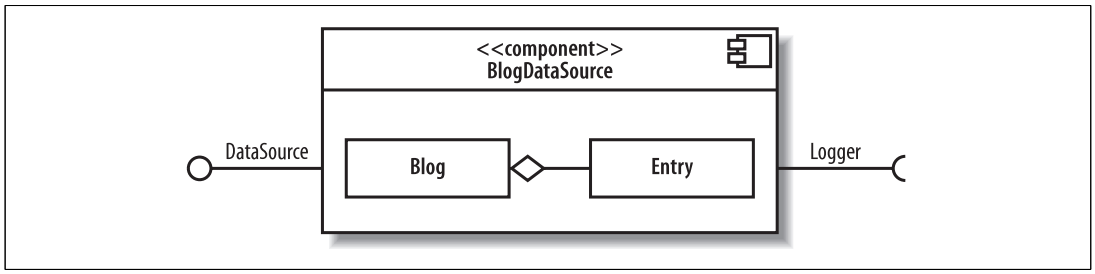


Figure 12-12. The Blog and Entry classes realize the BlogDataSource component

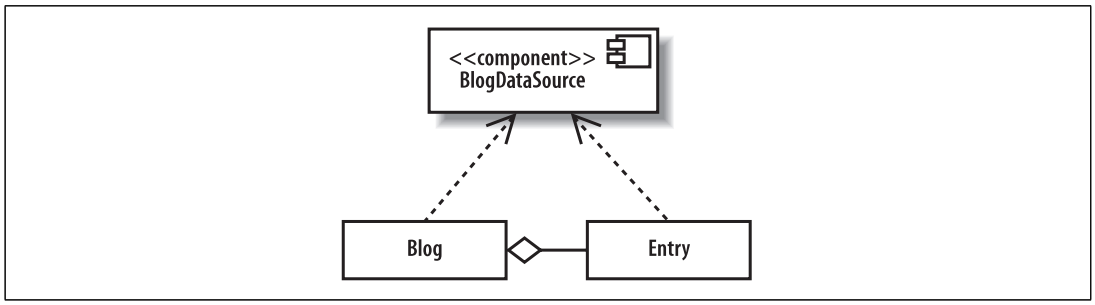


Figure 12-13. Alternate view, showing the realizing classes outside with the dependency relationship

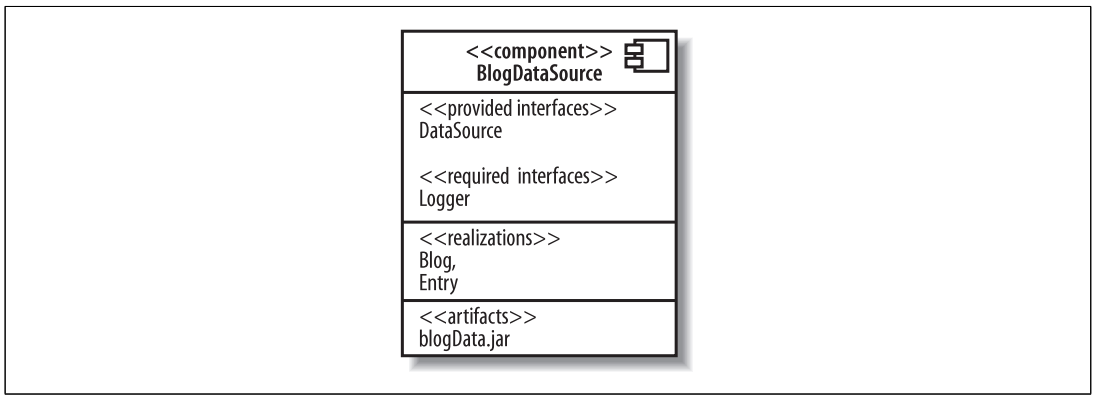


Figure 12-14. You can also list the realizing classes inside the component

How do you decide which notation to use to show the classes that realize a component? You may be limited by your UML tool, but if you have the choice, many modelers prefer the first notation (drawing the realizing classes inside) rather than drawing them outside since drawing them inside visually emphasizes that the classes make up a component to achieve its functionality. Listing the realizing classes may be helpful if you want something compact, but keep in mind that it can't show relationships between the realizing classes, whereas the first two notations can.

Ports and Internal Structure

Chapter 11 introduced ports and internal structure of a class. Components can also have ports and internal structure.You can use ports to model distinct ways that a component can be used with related interfaces attached to the port. In Figure 12-15, the `ConversionManagement` component has a `Formatting` and a `Data` port, each with their associated interfaces.

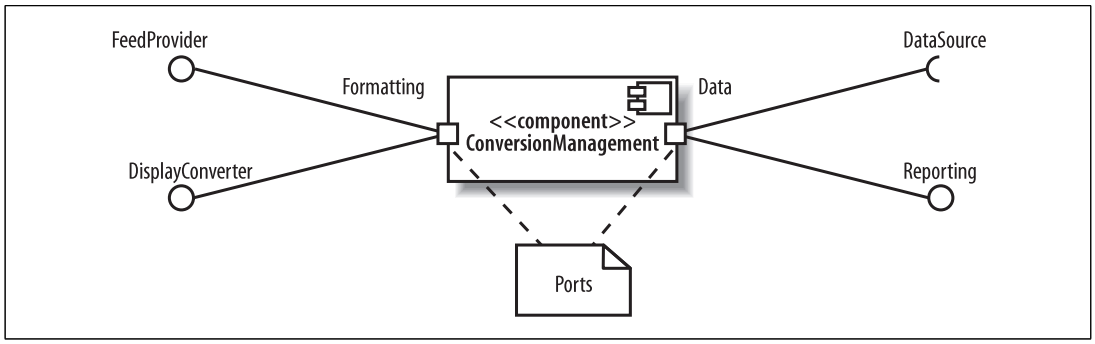


Figure 12-15. Ports show unique uses of a component and group "like" interfaces

You can show the internal structure of a component to model its parts, properties, and connectors (see Chapter 11 for a review of internal structure). Figure 12-16 shows the internal structure of a `BlogDataSource` component.

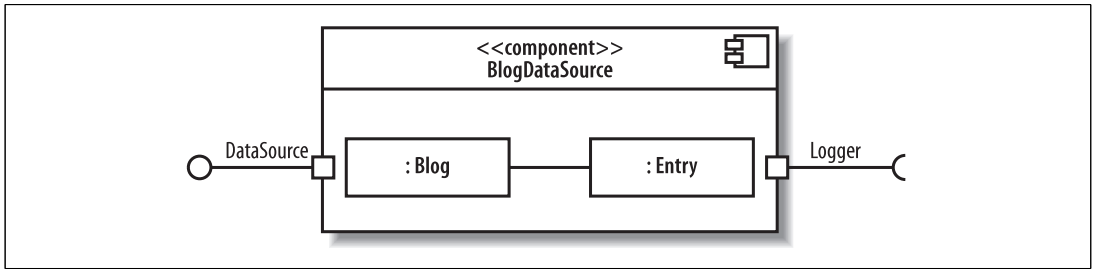


Figure 12-16. Showing the internal structure of a component

Components have their own unique constructs when showing ports and internal structure—called delegation connectors and assembly connectors. These are used to show how a component's interfaces match up with its internal parts and how the internal parts work together.

Delegation Connectors

A component's provided interface can be realized by one of its internal parts. Similarly, a component's required interface can be required by one of its parts. In these cases, you can use *delegation connectors* to show that internal parts realize or use the component's interfaces.

Delegation connectors are drawn with arrows pointing in the "direction of traffic," connecting the port attached to the interface with the internal part. If the part realizes a provided interface, then the arrow points from the port to the internal part.

If the part uses a required interface, then the arrow points from the internal part to the port. Figure 12-17 shows the use of delegation connectors to connect interfaces with internal parts.

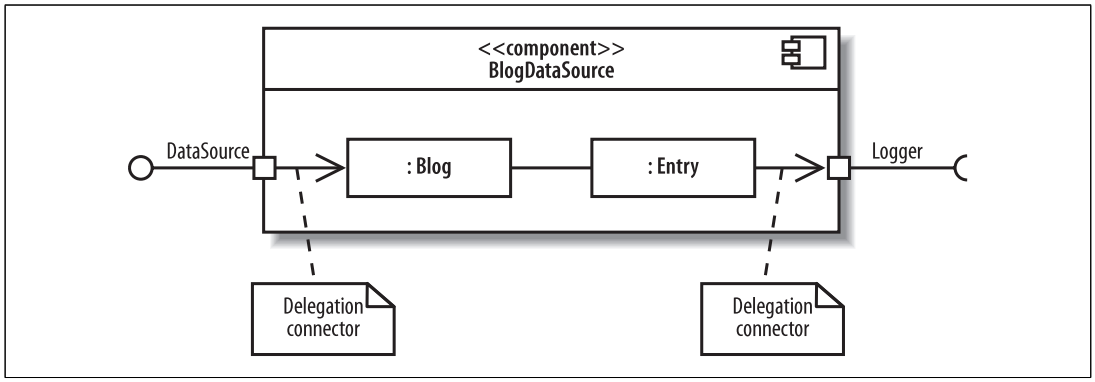


Figure 12-17. Delegation connectors show how interfaces correspond to internal parts: the Blog class realizes the DataSource interface and the Entry class requires the Logger interface

You can think of the delegation connectors as follows: the port represents an opening into a component through which communications pass, and delegation connectors point in the direction of communication. So, a delegation connector pointing from a port to an internal part represents messages being passed to the part that will handle it.

If you're showing the interfaces of the internal parts, you can connect delegation connectors to the interface instead of directly to the part. This is commonly used when showing a component that contains other components. Figure 12-19 demonstrates this notation. The `ConversionManagement` component has a `Controller` and a `BlogParser` component. The `ConversionManagement` component provides the `FeedProvider` interface, but this is actually realized internally by the `Controller` part.

Assembly Connectors

Assembly connectors show that a component requires an interface that another component provides. Assembly connectors snap together the ball and socket symbols that represent required and provided interfaces.

Figure 12-19 shows the assembly connector notation connecting the `Controller` component to the `BlogParser` component.

Assembly connectors are special kinds of connectors that are defined for use when showing composite structure of components. Notice that `Controller` and `BlogParser`

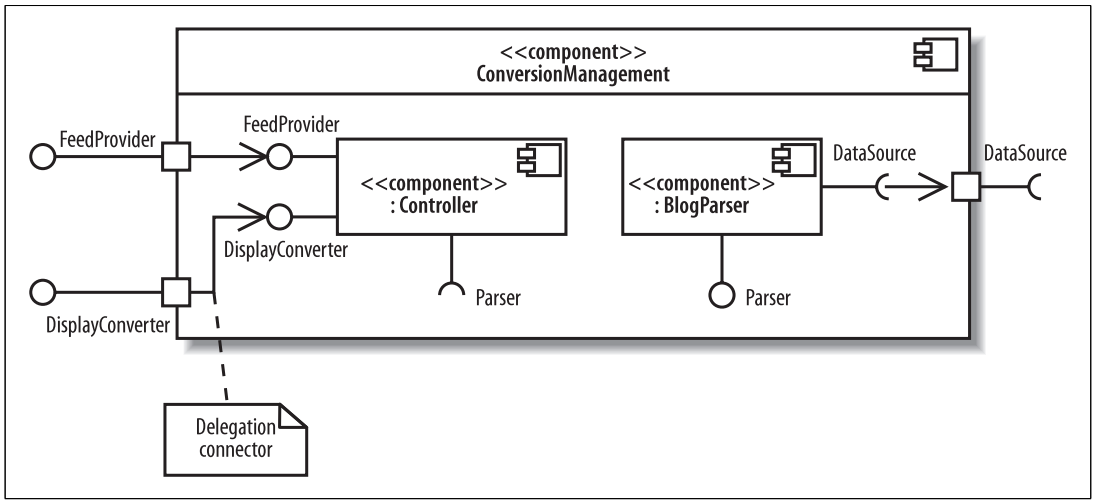


Figure 12-18. Delegation connectors can also connect interfaces of internal parts with ports

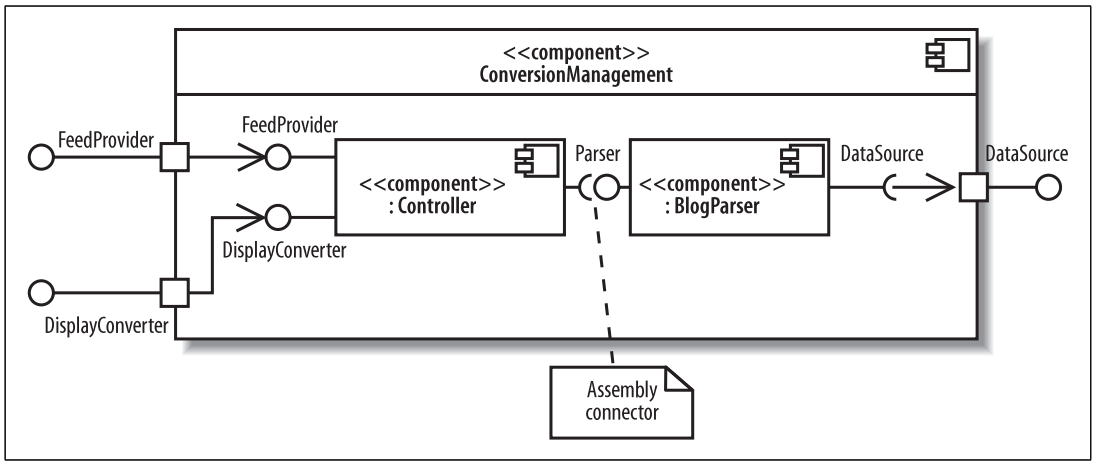


Figure 12-19. Assembly connectors show components working together through interfaces

use the `roleName:className` notation introduced in composite structures and help form the internal structure of `ConversionManagement`. But assembly connectors are also sometimes used as a presentation option for component dependency through interfaces in general, as shown earlier in Figure 12-8.

Black-Box and White-Box Component Views

There are two views of components in UML: a black-box view and a white-box view. The *black-box view* shows how a component looks from the outside, including its required interfaces, its provided interfaces, and how it relates to other components. A black-box view specifies nothing about the internal implementation of a compo-

nent. The *white-box view*, on the other hand, shows which classes, interfaces, and other components help a component achieve its functionality.

In this chapter, you've seen both black-box and white-box views. So, what's the difference in practical terms? A white-box view is one that shows parts inside a component, whereas a black-box view doesn't, as shown in Figure 12-20.

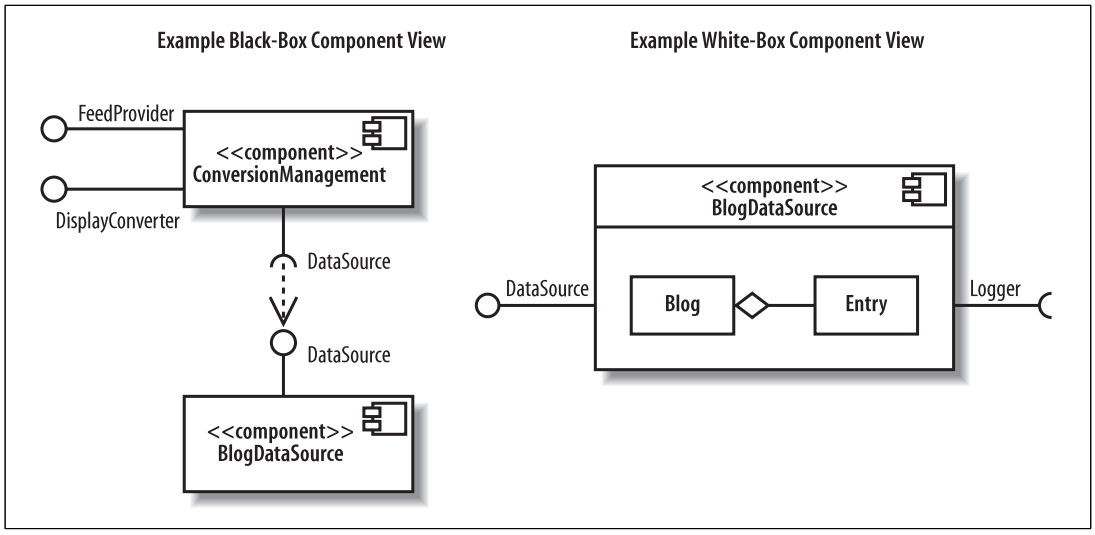


Figure 12-20. Black-box component views are useful for showing the big picture of the components in your system, whereas white-box views focus on the inner workings of a component

When modeling your system, it's best to use black-box views to focus on large-scale architectural concerns. Black-box views are good at showing the key components in your system and how they're connected. White-box views, on the other hand, are useful for showing how a component achieves its functionality through the classes it uses.

Black-box views usually contain more than one component, whereas in a white-box view, it's common to focus on the contents of one component.

What's Next?

Now that you know how to model the components in your system, you may want to look at how your components are deployed to hardware in deployment diagrams. Deployment diagrams are covered in Chapter 15.

There is heavy overlap between certain topics in component diagrams and composite structures. The ability to have ports and internal structure is defined for classes in composite structures. Components inherit this capability and introduce some of their own features, such as delegation and assembly connectors. Refer back to Chapter 11 to refresh your memory about a class's internal structure and ports.

CHAPTER 15

Modeling Your Deployed System: Deployment Diagrams

If you've been applying the UML techniques shown in earlier chapters of this book, then you've seen all but one view of your system. That missing piece is the *physical* view. The physical view is concerned with the physical elements of your system, such as executable software files and the hardware they run on.

UML *deployment diagrams* show the physical view of your system, bringing your software into the real world by showing how software gets assigned to hardware and how the pieces communicate (see Figure 15-1).

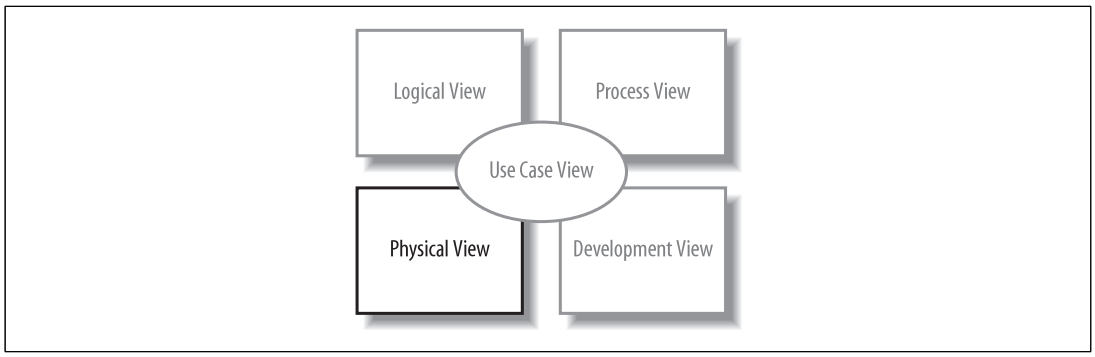
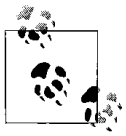


Figure 15-1. Deployment diagrams focus on the Physical View of your system



The word *system* can mean different things to different people; in the context of deployment diagrams, it means the software you create and the hardware and software that allow your software to run.

Deploying a Simple System

Let's start by showing a deployment diagram of a very simple system. In this simplest of cases, your software will be delivered as a single executable file that will reside on one computer.

To show computer hardware, you use a *node*, as shown in Figure 15-2.

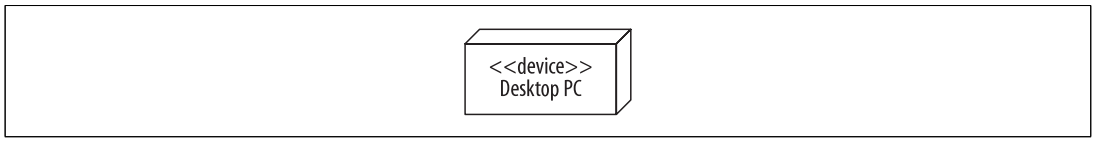


Figure 15-2. Use nodes to represent hardware in your system

This system contains a single piece of hardware—a Desktop PC. It's labeled with the stereotype `<<device>>` to specify that this is a hardware node.

One More Time...Model Levels

It must be about time to bring up modeling at the right level again. In Figure 15-2, the hardware node is specified as a Desktop PC. It's entirely up to you how much detail you want to give node names. You could be very precise with a name such as "64-bit Processor Intel Workstation," or very general with a name such as "Generic PC."

If you have specific hardware requirements for your system, you're likely to give your nodes very precise names. If your hardware requirements are undefined or insignificant, you might have vague node names. As with all other aspects of UML, it is important to make sure that you are modeling at the right level for *your* system.

Now, you need to model the software that runs on the hardware. Figure 15-3 shows a simple software artifact (see "Deployed Software: Artifacts," next), which in this case is just a JAR file named *3dpacman.jar*, containing a 3D-Pacman application.

Figure 15-3. A physical software file such as a jar file is modeled with an artifact

Finally, you need to put these two pieces together to complete the deployment diagram of your system. Draw the artifact inside the node to show that a software artifact is deployed to a hardware node. Figure 15-4 shows that *3dpacman.jar* runs on a Desktop PC.

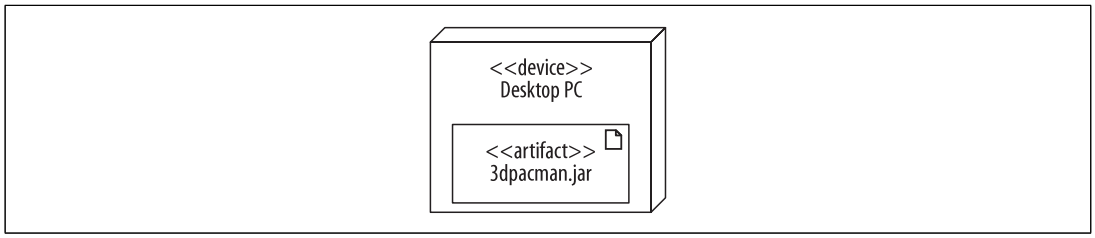


Figure 15-4. Drawing an artifact inside a node shows that the artifact is deployed to the node

But is it really complete? Don't you need to model the Java Virtual Machine (JVM) because without it, your code wouldn't execute? What about the operating system; isn't that important? The answer, unfortunately, is possibly.

Your deployment diagrams should contain details about your system that are important to your audience. If it is important to show the hardware, firmware, operating system, runtime environments, or even device drivers of your system, then you should include these in your deployment diagram. As the rest of this chapter will show, deployment diagram notation can be used to model all of these types of things. If there's a feature of your system that's not important, then it's not worth adding it to your diagram since it could easily clutter up or distract from those features of your design that *are* important.

Deployed Software: Artifacts

The previous section showed a sneak preview of some of the notation that can be used to show the software and hardware in a deployed system. The *3dpacman.jar* software was deployed to a single hardware node. In UML, that JAR file is called an artifact.

Artifacts are physical files that execute or are used by your software. Common artifacts you'll encounter include:

- Executable files, such as *.exe* or *.jar* files
- Library files, such as *.dll*s (or support *.jar* files)
- Source files, such as *.java* or *.cpp* files
- Configuration files that are used by your software at runtime, commonly in formats such as *.xml*, *.properties*, or *.txt*

An artifact is shown as a rectangle with the stereotype `<<artifact>>`, or the document icon in the upper right hand corner, or both, as shown in Figure 15-5. For the rest of the book, an artifact will be shown with both the stereotype `<<artifact>>` and the document icon.

Figure 15-5. Equivalent representations of a 3dpacman.jar artifact

Deploying an Artifact to a Node

An artifact is *deployed* to a node, which means that the artifact resides on (or is installed on) the node. Figure 15-6 shows the *3dpacman.jar* artifact from the previous example deployed to a `Desktop PC` hardware node by drawing the artifact symbol inside the node.

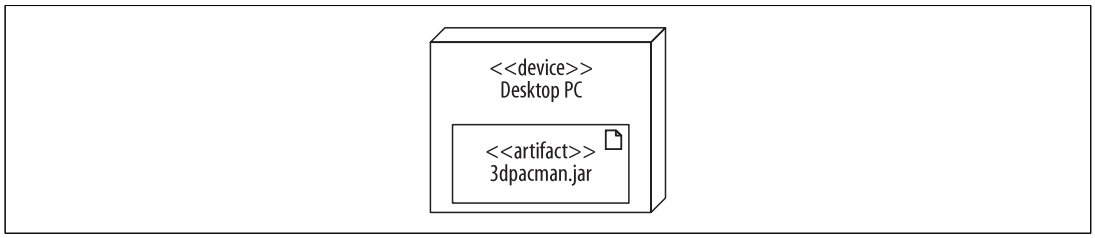


Figure 15-6. The 3dpacman.jar artifact deployed to a Desktop PC node

You can model that an artifact is deployed to a node in two other ways. You can also draw a dependency arrow from the artifact to the target node with the stereotype `<<deploy>>`, as shown in Figure 15-7.

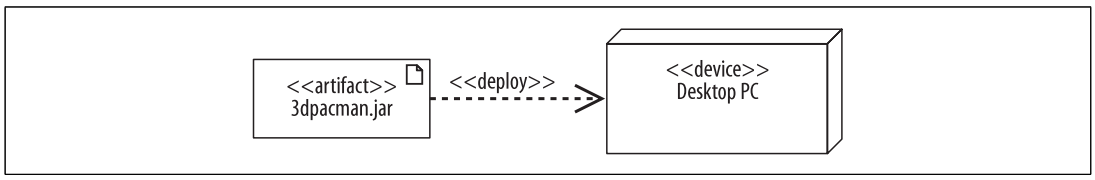


Figure 15-7. An alternate way to model the relationship deployment

When you're pressed for space, you might want to represent the deployment by simply listing the artifact's name inside the target node, as shown in Figure 15-8.

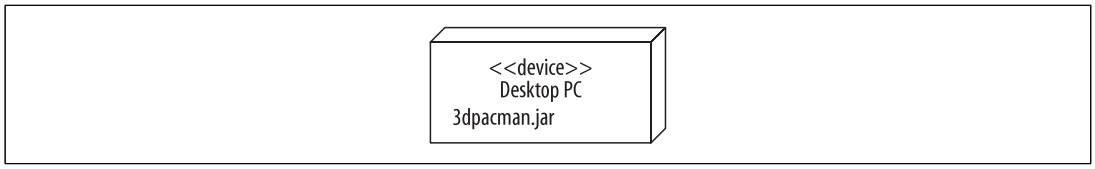


Figure 15-8. A compact way to show deployment is to write the name of the artifact inside the node

All of these methods show the same deployment relationship, so here are some guidelines for picking a notation.

Listing the artifacts (without the artifact symbol) can really save space if you have a lot of artifacts, as in Figure 15-9. Imagine how big the diagram would get if you drew the artifact symbol for each artifact.

But be careful; by listing your artifacts, you cannot show dependencies between artifacts. If you want to show that an artifact uses another artifact, you have to draw the artifact symbols and a dependency arrow connecting the artifacts, as shown in Figure 15-10.

Tying Software to Artifacts

When designing software, you break it up into cohesive groups of functionality, such as components or packages, which eventually get compiled into one or more files—or artifacts. In UML-speak, if an artifact is the physical actualization of a

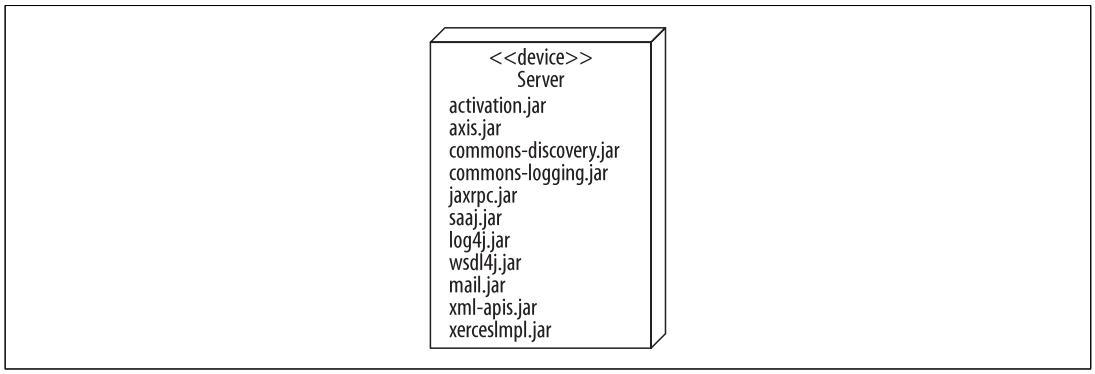


Figure 15-9. Listing artifact names inside a node saves a lot of space compared to drawing an artifact symbol for each artifact

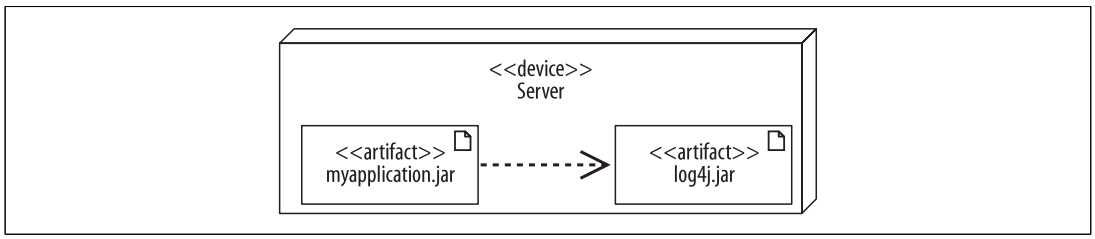


Figure 15-10. A deployment notation that uses artifact symbols (instead of listing artifact names) allows you to show artifact dependencies

component, then the artifact *manifests* that component. An artifact can manifest not just components but any packageable element, such as packages and classes.

The manifest relationship is shown with a dependency arrow from the artifact to the component with the stereotype `<<manifest>>`, as shown in Figure 15-11.

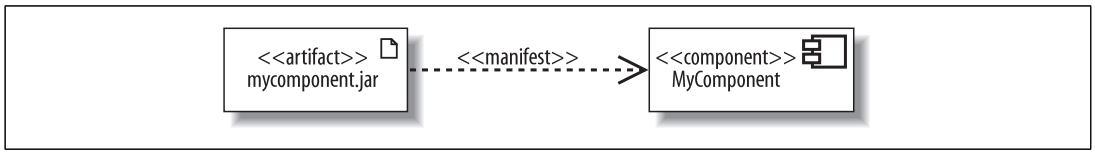


Figure 15-11. The artifact mycomponent.jar manifests the component MyComponent

Since artifacts can then be assigned to nodes, the manifest relationship provides the missing link in modeling how your software components are mapped to hardware. However, linking a component to an artifact to a node can result in a cluttered diagram, so it's common to show the manifest relationships separate from the deployment relationships, even if they're on the same deployment diagram.

You can also show the manifest relationship in component diagrams by listing the artifacts manifesting a component within the component symbol, as discussed in Chapter 12.

If you're familiar with earlier versions of UML, you may be tempted to model a component running on hardware by drawing the component symbol inside the node. As of UML 2.0, artifacts have nudged components toward a more conceptual interpretation, and now artifacts represent physical files.

However, many UML tools aren't fully up to date with the UML 2.0 standard, so your tool may still use the earlier notation.

What Is a Node?

You've already seen that you can use nodes to show hardware in your deployment diagram, but nodes don't have to be hardware. Certain types of software—software that provides an environment within which other software components can be executed—are nodes as well.

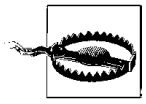
A *node* is a hardware or software resource that can host software or related files. You can think of a software node as an application context; generally not part of the software you developed, but a third-party environment that provides services to your software.

The following items are reasonably common examples of hardware nodes:

- Server
- Desktop PC
- Disk drives

The following items are examples of execution environment nodes:

- Operating system
- J2EE container
- Web server
- Application server



Software items such as library files, property files, and executable files that cannot host software are *not* nodes—they are artifacts (see "Deployed Software: Artifacts," earlier in the chapter).

Hardware and Execution Environment Nodes

A node is drawn as a cube with its type written inside, as shown in Figure 15-12. The stereotype `<<device>>` emphasizes that it's a hardware node.

Figure 15-13 shows an Application Server node. Those familiar with enterprise software development will recognize this as a type of execution environment since it's a software environment that provides services to your application. The stereotype `<<executionEnvironment>>` emphasizes that this node is an execution environment.

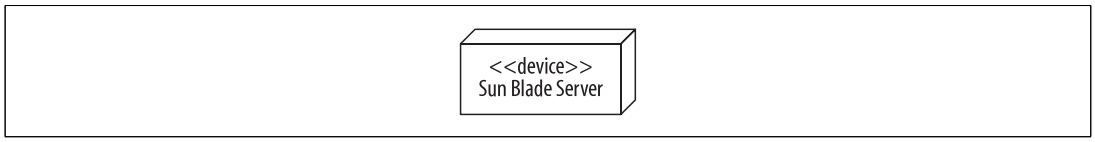


Figure 15-12. A Sun Blade Server hardware node marked with the stereotype <<device>>

Figure 15-13. An Application Server node marked with the stereotype <<executionEnvironment>>

Execution environments do not exist on their own—they run on hardware. For example, an operating system needs computer hardware to run on. You show that an execution environment resides on a particular device by placing the nodes inside one another, nesting them as shown in Figure 15-14.

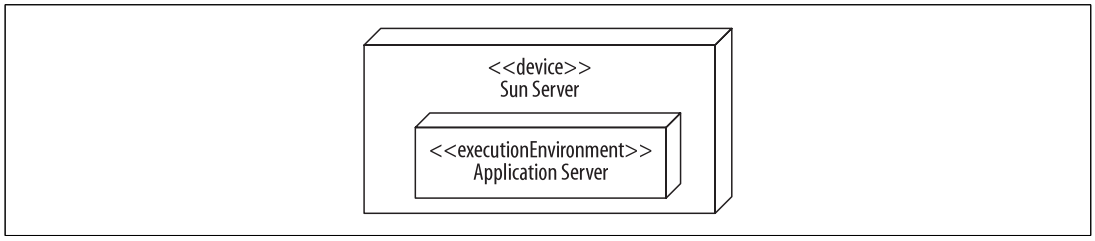


Figure 15-14. An Application Server node is shown nested in a Sun Server node, meaning that the Application Server runs on Sun Server hardware.

It's not strictly necessary in UML 2.0 to distinguish device nodes from execution environment nodes, but it's a good habit to get into because it can clarify your model.

Want more variety? If you're using a profile (discussed in Appendix B), you can apply node stereotypes that are more relevant to your domain, such as `<<J2EE Container>>`. These new node types can be specified in your profile as a special kind of execution environment.

Showing Node Instances

There are times when your diagram includes two nodes of the same type, but you want to draw attention to the fact that they are actually different instances. You can show an instance of a node by using the `name : type` notation as shown in Figure 15-15.

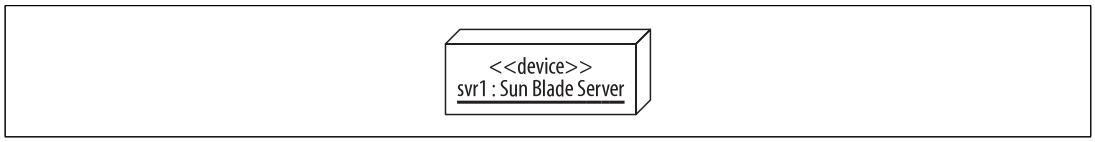


Figure 15-15. Showing the name and type of a node; an instance of a Sun Blade Server named svr1

Figure 15-16 shows how two nodes of the same type can be modeled. The nodes in this example, `svr1` and `svr2`, are assigned different types of traffic from a load balancer (a common situation in enterprise systems).

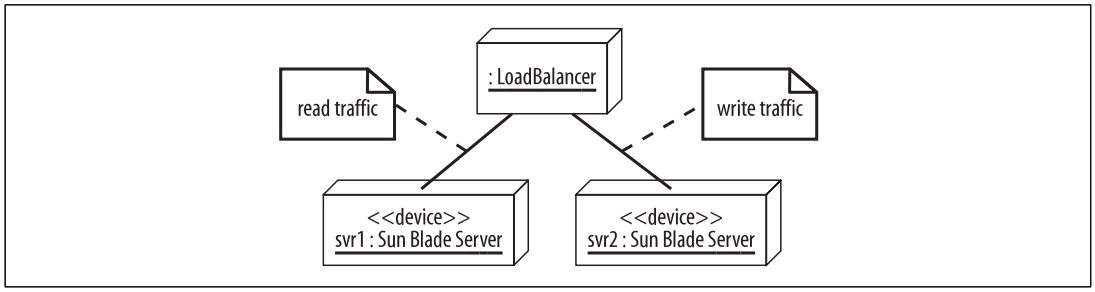


Figure 15-16. One node gets read traffic and the other gets write traffic

Communication Between Nodes

To get its job done, a node may need to communicate with other nodes. For example, a client application running on a desktop PC may retrieve data from a server using TCP/IP.

Communication paths are used to show that nodes communicate with each other at runtime. A communication path is drawn as a solid line connecting two nodes. The type of communication is shown by adding a stereotype to the path. Figure 15-17 shows two nodes—a desktop PC and a server—that communicate using TCP/IP.

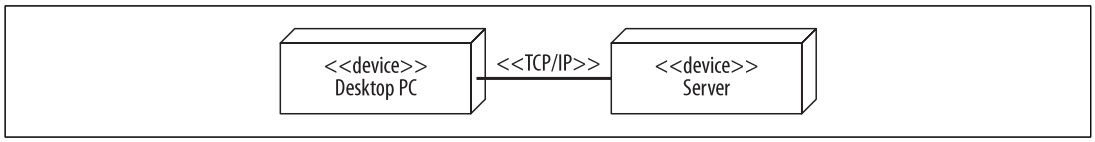


Figure 15-17. A Desktop PC and Server communicate via TCP/IP

You can also show communication paths between execution environment nodes. For example, you could model a web server communicating with an EJB container through RMI, as shown in Figure 15-18. This is more precise than showing an RMI communication path at the device node level because the execution environment nodes "speak" RMI. However, some modelers draw the communication paths at the outermost node level because it can make the diagram less cluttered.

Assigning a stereotype to a communication path can sometimes be tricky. RMI is layered using a TCP/IP transport layer. So, should you assign an `<<RMI>>` or a `<<TCP/IP>>` stereotype? As a rule of thumb, your communication stereotype should be as high-level as possible because it communicates more about your system. In this case, `<<RMI>>` is the right choice; it is higher level, and it tells the reader that you're using

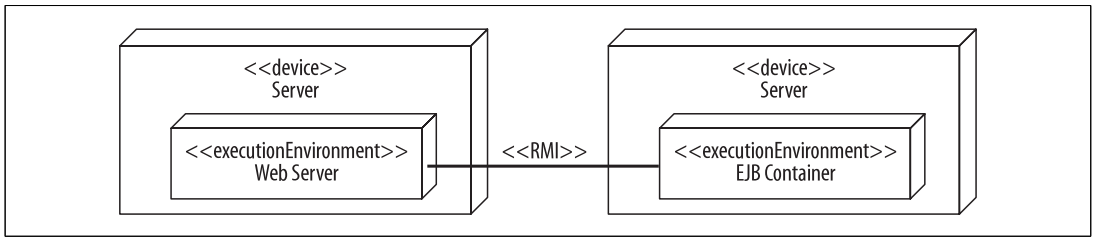


Figure 15-18. You can also show communication paths between execution environment nodes

a Java implementation. However, as with all UML modeling, you should tailor the diagram to your audience.

Communication paths show that the nodes are capable of communicating with each other and are not intended to show individual messages, such as messages in a sequence diagram.

As of UML 2.0, stereotypes are supposed to be specified in a profile, so in theory, you should use only the stereotypes that your profile provides. However, even if you're not using a profile, your UML tool may allow you to make up any stereotype. Since stereotypes are a good way to show the types of communication in a system, feel free to make your own if necessary and if your tool allows. But if you do, try to keep them consistent. For example, don't create two stereotypes `<<RMI>>` and `<<Remote Method Invocation>>`, which are the same type of communication.

Deployment Specifications

Installing software is rarely as easy as dropping a file on a machine; often you have to specify configuration parameters before your software can execute. A *deployment specification* is a special artifact specifying how another artifact is deployed to a node. It provides information that allows another artifact to run successfully in its environment.

Deployment specifications are drawn as a rectangle with the stereotype `<<deployment spec>>`. There are two ways to tie a deployment specification to the deployment it describes:

- Draw a dependency arrow from the deployment specification to the artifact, nesting both of these in the target node.
- Attach the deployment specification to the deployment arrow, as shown in Figure 15-19.

The *deploy.wsdd* file, shown in Figure 15-19, is the standard deployment descriptor file that specifies how a web service is deployed to the Axis web service engine. This file states which class executes the web service and which methods on the class can be called. You can list these properties in the deployment specification using the `name`

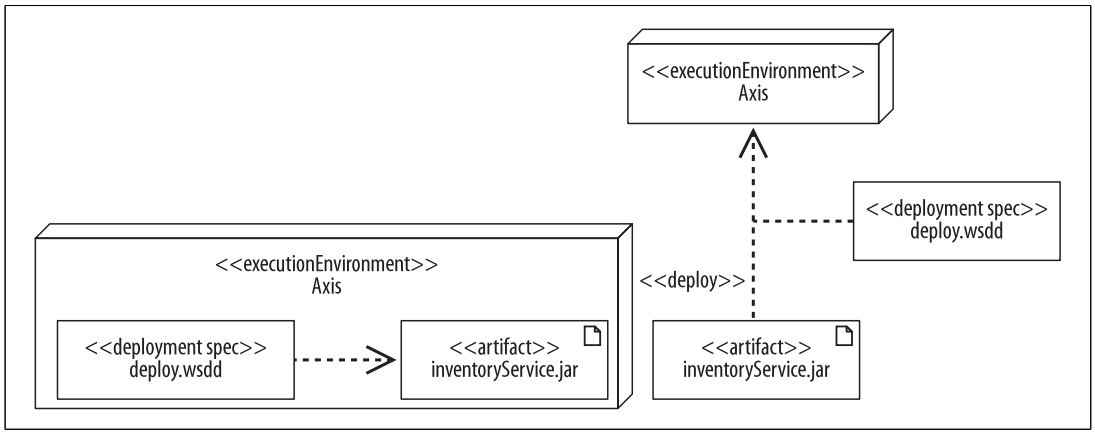


Figure 15-19. Equivalent ways of tying a deployment specification to the deployment it describes

`: type` notation. Figure 15-20 shows the *deploy.wsdd* deployment specification with the properties `className` and `allowedMethods`.

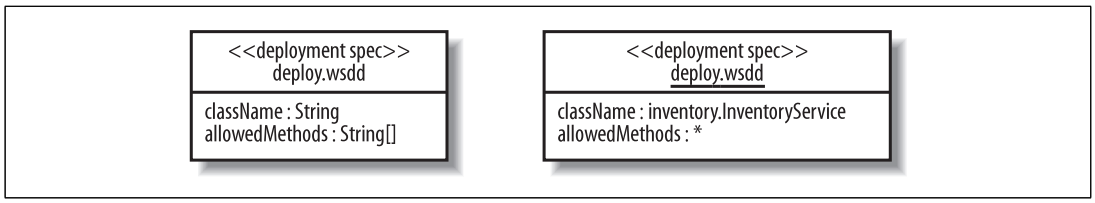


Figure 15-20. Showing the properties of a deployment specification: the notation on the right shows an instance populated with values

The symbol on the right shows an *instance* of a deployment specification populated with values. Use this notation if you want to show the actual property values instead of just the types.

This chapter has only briefly mentioned instances of elements in deployment diagrams, but you can model instances of nodes, artifacts, and deployment specifications. In deployment diagrams, many modelers don't bother to specify that an element is an instance if the intent is clear. However, if you want to specify property values of a deployment specification (as on the right side of Figure 15-20), then this is a rare situation where a UML tool may force you to use the instance notation.

Currently, many UML tools don't support the deployment specification symbol. If yours is one of them, you can attach a note containing similar information.

You don't need to list every property in a deployment specification—only properties you consider important to the deployment. For example, *deploy.wsdd* may contain other properties such as allowed roles, but if you're not using that property or it's insignificant (i.e., it's the same for all your web services), then leave it out.

When to Use a Deployment Diagram

Deployment diagrams are useful at all stages of the design process. When you begin designing a system, you probably know only basic information about the physical layout. For example, if you're building a web application, you may not have decided which hardware to use and probably don't know what your software artifacts are called. But you want to communicate important characteristics of your system, such as the following:

- Your architecture includes a web server, application server, and database.
- Clients can access your application through a browser or through a richer GUI interface.
- The web server is protected with a firewall.

Even at this early stage you can use deployment diagrams to model these characteristics. Figure 15-21 shows a rough sketch of your system. The node names don't have to be precise, and you don't have to specify the communication protocols.

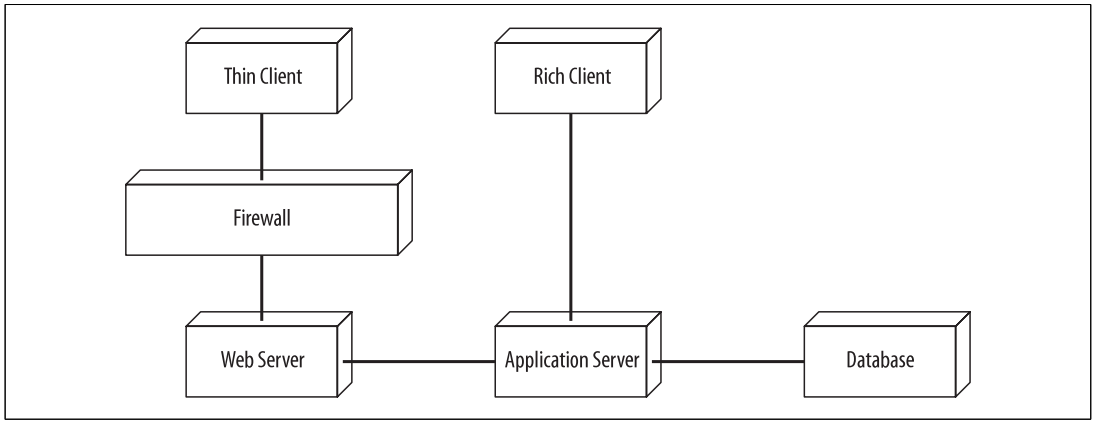


Figure 15-21. A rough sketch of your web application

Deployment diagrams are also useful in later stages of software development. Figure 15-22 shows a detailed deployment diagram specifying a J2EE implementation of the system.

Figure 15-22 is more specific about the hardware types, the communication protocols, and the allocation of software artifacts to nodes. A detailed deployment diagram, such as Figure 15-22, could be used be used as a blueprint for how to install your system.

You can revisit your deployment diagrams throughout the design of your system to refine the rough initial sketches, adding detail as you decide which technologies, communication protocols, and software artifacts will be used. These refined deployment diagrams allow you to express the current view of the physical system layout with the system's stakeholders.

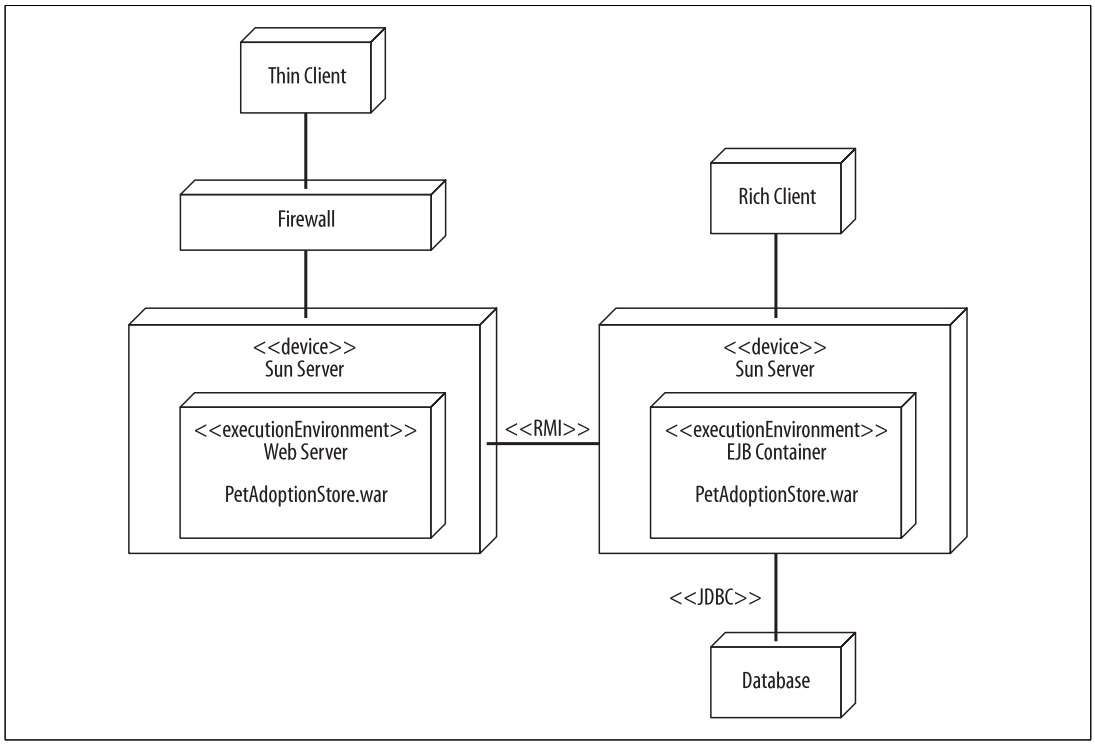


Figure 15-22. You can provide any amount of detail about the physical design of your system

What's Next?

You've finished learning the fundamental UML concepts, but read on to the appendixes for an overview of some advanced modeling techniques. The appendices introduce you to the Object Constraint Language (OCL), which is a rigorous way to show constraints in your diagrams, and Profiles, which allow you to define and use a custom UML vocabulary. It's helpful to review these appendices to get a feel for extra precision you can add to your model and extra capabilities that result from that precision. The Object Constraint Language is covered in Appendix A; UML profiles are described in Appendix B.