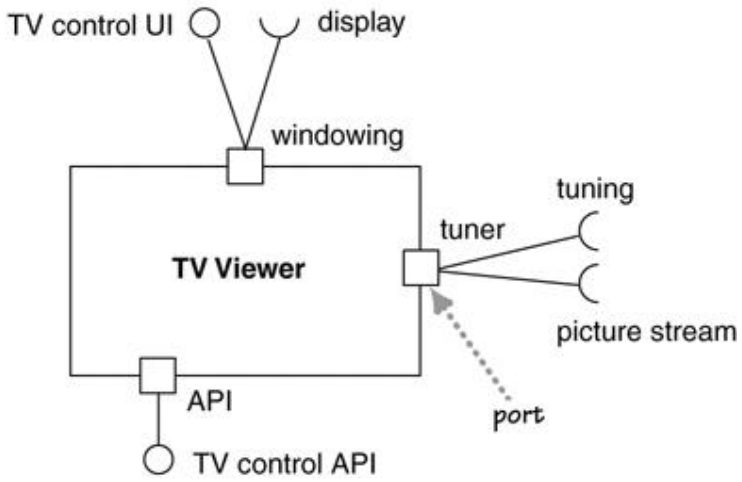
You can show how many instances of a part are present. Figure 13.2 says that each TV Viewer contains one generator part and one controls part.

To show a part implementing an interface, you draw a delegating connector from that interface. Similarly, to show that a part needs an interface, you show a delegating connector to that interface. You can also show connectors between parts with either a simple line, as I've done here, or with ball-and-socket notation (page 71).

You can add ports (Figure 13.3) to the external structure. Ports allow you to group the required and provided interfaces into logical interactions that a component has with the outside world.

Figure 13.3. A component with multiple ports



When to Use Composite Structures

Composite structures are new to UML 2, although some older methods had some similar ideas. A good way of thinking about the difference between packages and composite structures is that packages are a compile-time grouping, while composite structures show runtime groupings. As such, they are a natural fit for showing components and how they are broken into parts; hence, much of this notation is used in component diagrams.

Because composite structures are new to the UML, it's too early to tell how effective they will turn out in practice; many members of the UML committee think that these diagrams will become a very valuable addition.

Chapter 14. Component Diagrams

A debate that's always ranged large in the OO community is what the difference is between a component and any regular class. This is not a debate that I want to settle here, but I can show you the notation the UML uses to distinguish between them.

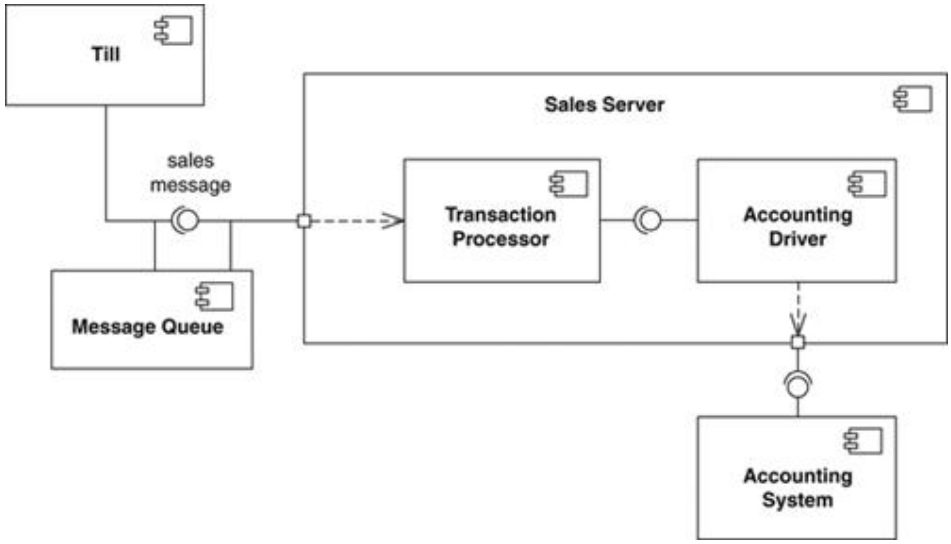
UML 1 had a distinctive symbol for a component (Figure 14.1). UML 2 removed that icon but allows you to annotate a class box with a similar-looking icon. Alternatively, you can use the `«component»` keyword.

Figure 14.1. Notation for components

Other than the icon, components don't introduce any notation that we haven't already seen. Components are connected through implemented and required interfaces, often using the ball-and-socket notation (page 71) just as for class diagrams. You can also decompose components by using composite structure diagrams.

Figure 14.2 shows an example component diagram. In this example, a sales till can connect to a sales server component, using a sales message interface. Because the network is unreliable, a message queue component is set up so the till can talk to the server when the network is up and talk to a queue when the network is down; the queue will then talk to the server when the network becomes available. As a result, the message queue both supplies the sales message interface to talk with the till and requires that interface to talk with the server. The server is broken down into two major components. The transaction processor realizes the sales message interface, and the accounting driver talks to the accounting system.

Figure 14.2. An example component diagram



As I've already said, the issue of what is a component is the subject of endless debate. One of the more helpful statements I've found is this:

> *Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy*.

Ralph Johnson, **http://www.c2.com/cgi/wiki?DoComponentsExist**

The important point is that components represent pieces that are independently purchasable and upgradeable. As a result, dividing a system into components is as much a marketing decision as it is a technical decision, for which [Hohmann] is an excellent guide. It's also a reminder to beware of

overly fine-grained components, because too many components are hard to manage, especially when versioning rears its ugly head, hence "DLL hell."

In earlier versions of the UML, components were used to represent physical structures, such as DLLs. That's no longer true; for this task, you now use artifacts (page 97).

When to Use Component Diagrams

Use component diagrams when you are dividing your system into components and want to show their interrelationships through interfaces or the breakdown of components into a lower-level structure.

Chapter 15. Collaborations

Unlike the other chapters in this book, this one does not correspond to an official diagram in UML 2. The standard discusses collaborations as part of composite structures, but the diagram is really quite different and was used in UML 1 without any link to composite structures. So I felt it best to discuss collaborations as their own chapter.

Let's consider the notion of an auction. In any auction, we might have a seller, some buyers, a lot of goods, and some offers for the sale. We can describe these elements in terms of a class diagram (Figure 15.1) and perhaps some interaction diagrams (Figure 15.2).

Figure 15.1. A collaboration with its class diagram of roles

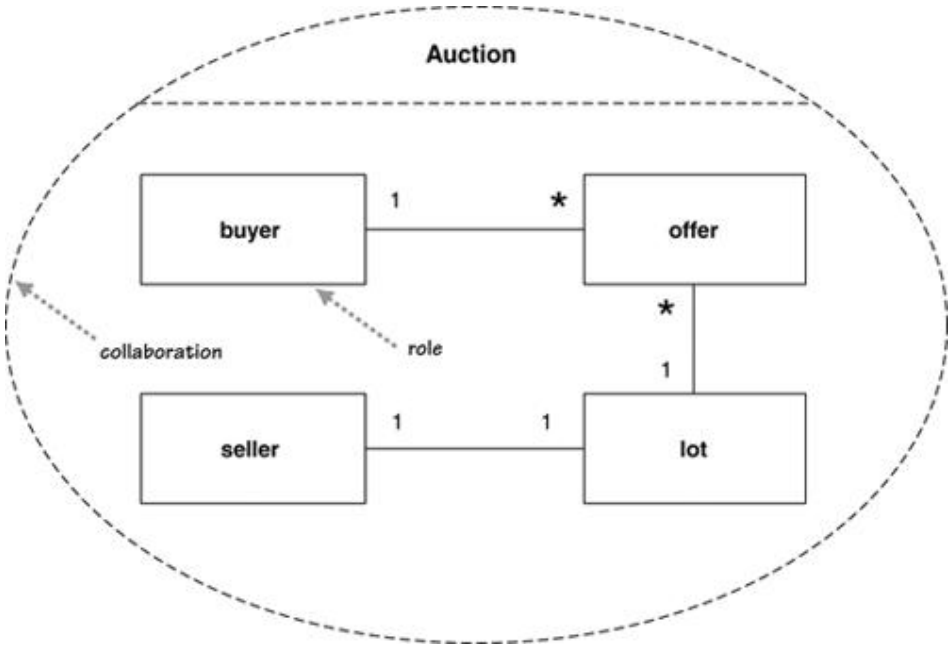


Figure 15.2. A sequence diagram for the auction collaboration

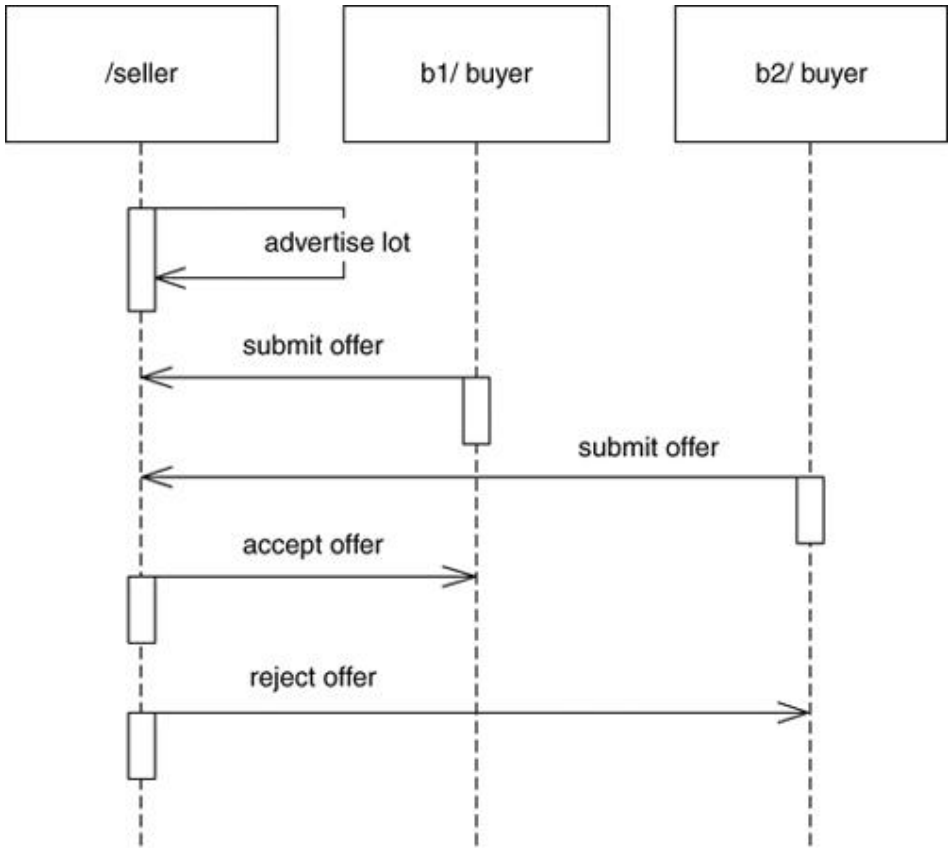
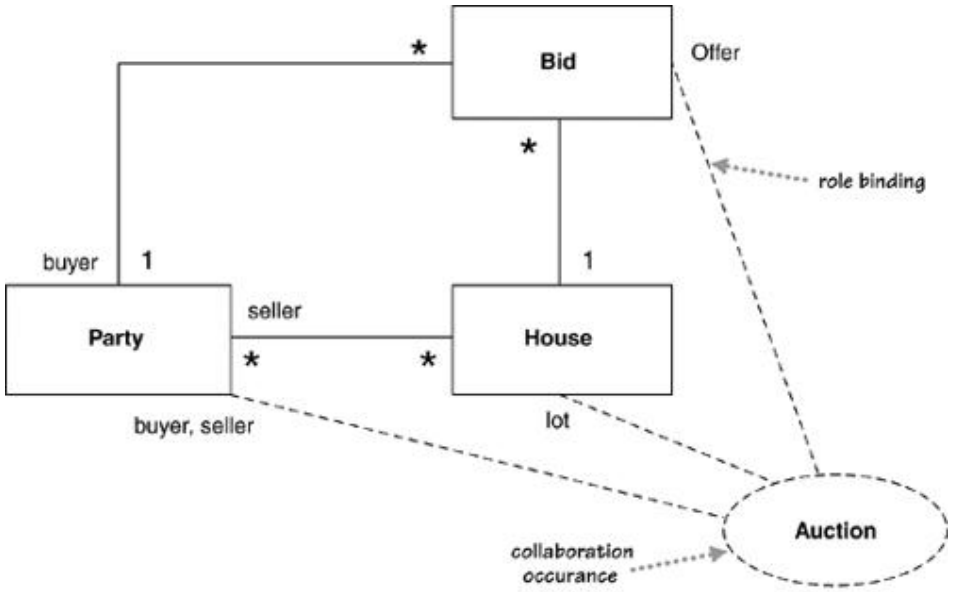


Figure 15.1 is not quite a regular class diagram. For a start, it's surrounded by the dashed ellipse, which represents the auction collaboration. Second, the so-called classes in the collaboration are not classes but **roles** that will be realized as the collaboration is applied—hence the fact that their names aren't capitalized. It's not unusual to see actual interfaces or classes that correspond to the collaboration roles, but you don't have to have them.

In the interaction diagram, the participants are labeled slightly differently from the usual case. In a collaboration, the naming scheme is `participant-name /role-name : class-name`. As usual, all these elements are optional.

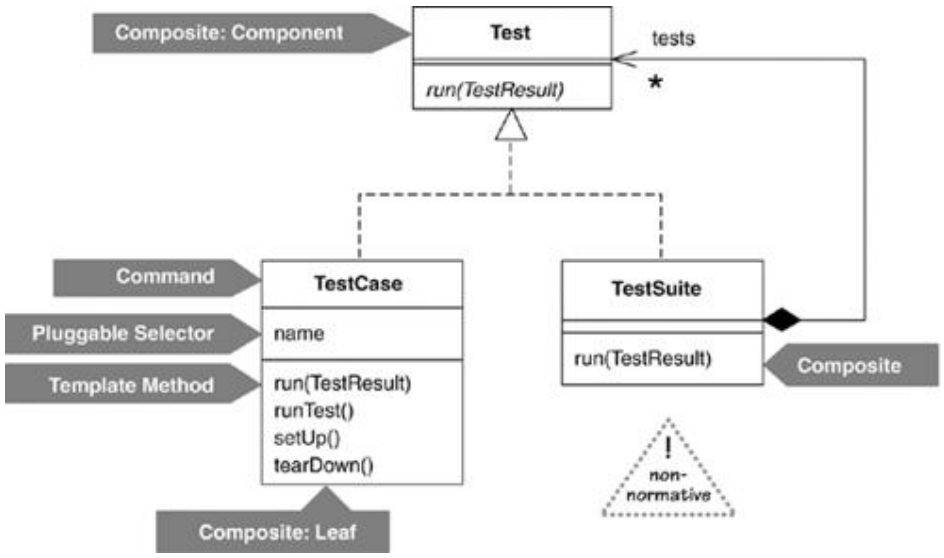
When you use a collaboration, you can show that by placing a collaboration occurrence on a class diagram, as in Figure 15.3, a class diagram of some of the classes in the application. The links from the collaboration to those classes indicate how the classes play the various roles defined in the collaboration.

Figure 15.3. A collaboration occurrence



The UML suggests that you can use the collaboration occurrence notation to show the use of patterns, but hardly any patterns author has done this. Erich Gamma developed a nice alternative notation (Figure 15.4). Elements of the diagram are labeled with either the pattern name or a combination of `pattern:role`.

Figure 15.4. A nonstandard way of showing pattern use in JUnit (junit.org)



When to Use Collaborations

Collaborations have been around since UML 1, but I admit I've hardly used them, even in my patterns writing. Collaborations do provide a way to group chunks of interaction behavior when roles are played by different classes. In practice, however, I've not found that they've been a compelling diagram type.

Chapter 16. Interaction Overview Diagrams

Interaction overview diagrams are a grafting together of activity diagrams and sequence diagrams. You can think of interaction overview diagrams either as activity diagrams in which the activities are