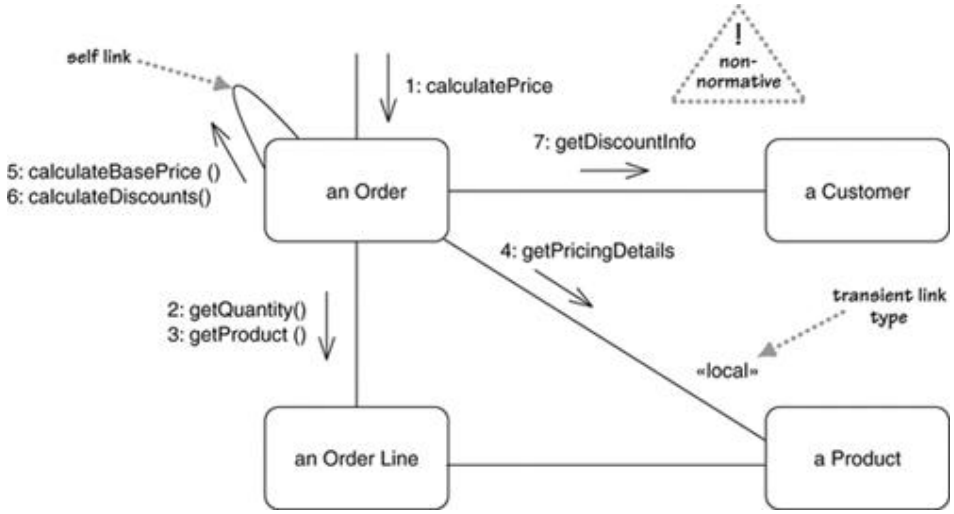
Chapter 12. Communication Diagrams

Communication diagrams, a kind of interaction diagram, emphasize the data links between the various participants in the interaction. Instead of drawing each participant as a lifeline and showing the sequence of messages by vertical direction as the sequence diagrams does, the communication diagram allows free placement of participants, allows you to draw links to show how the participants connect, and use numbering to show the sequence of messages.

In UML 1.x, these diagrams were called **collaboration diagrams**. This name stuck well, and I suspect that it will be a while before people get used to the new name. (These are different from Collaborations [page 143]; hence the name change.)

Figure 12.1 shows a communication diagram for the same centralized control interaction as in Figure 4.2. With a communication diagram, we can show how the participants are linked together.

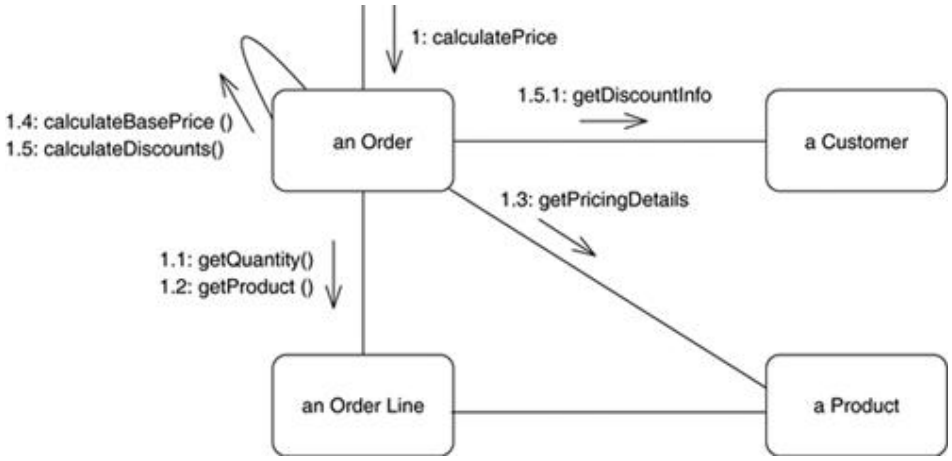
Figure 12.1. Communication diagram for centralized control



As well as showing links that are instances of associations, we can also show transient links, which arise only the context of the interaction. In this case, the `«local»` link from Order to Product is a local variable; other transient links are `«parameter»` and `«global»`. These keywords were used in UML 1 but are missing from UML 2. Because they are useful, I expect them to stay around in conventional use.

The numbering style of Figure 12.1 is straightforward and commonly used, but actually isn't legal UML. To be kosher UML, you have to use a nested decimal numbering scheme, as in Figure 12.2.

Figure 12.2. Communication diagram with nested decimal numbering



The reason for the nested decimal numbers is to resolve ambiguity with self-calls. In Figure 4.2, you can clearly see that `getDiscountInfo` is called within the method `calculateDiscount`. With the flat numbering of Figure 12.1, however, you can't tell whether `getDiscountInfo` is called within `calculateDiscount` or within the overall `calculatePrice` method. The nested numbering scheme resolves this problem.

Despite its illegality, many people prefer a flat numbering scheme. The nested numbers can get very tangled, particularly as calls get rather nested, leading to such sequence numbers as 1.1.1.2.1.1. In these cases, the cure for ambiguity can be worse than the disease.

As well as numbers, you may also see letters on messages; these letters indicate different threads of control. So messages A5 and B2 would be in different threads; messages 1a1 and 1b1 would be different threads concurrently nested within message 1. You also see thread letters on sequence diagrams, although this doesn't convey the concurrency visually.

Communication diagrams don't have any precise notation for control logic. They do allow you to use iteration markers and guards (page 59), but they don't allow you to fully specify control logic. There is no special notation for creating or deleting objects, but the `«create»` and `«delete»` keywords are common conventions.

When to Use Communication Diagrams

The main question with communication diagrams is when to use them rather than the more common sequence diagrams. A strong part of the decision is personal preference: Some people like one over the other. Often, that drives the choice more than anything else. On the whole, most people seem to prefer sequence diagrams, and for once, I'm with the majority.

A more rational approach says that sequence diagrams are better when you want to emphasize the sequence of calls and that communication diagrams are better when you want to emphasize the links. Many people find that communication diagrams are easier to alter on a whiteboard, so they are a good approach for exploring alternatives, although in those cases, I often prefer CRC cards.

Chapter 13. Composite Structures

One of the most significant new features in UML 2 is the ability to hierarchically decompose a class into an internal structure. This allows you to take a complex object and break it down into parts.

Figure 13.1 shows a TV Viewer class with its provided and required interfaces (page 69). I've shown this in two ways: using the ball-and-socket notation and listing them internally.

Figure 13.1. Two ways of showing a TV viewer and its interfaces

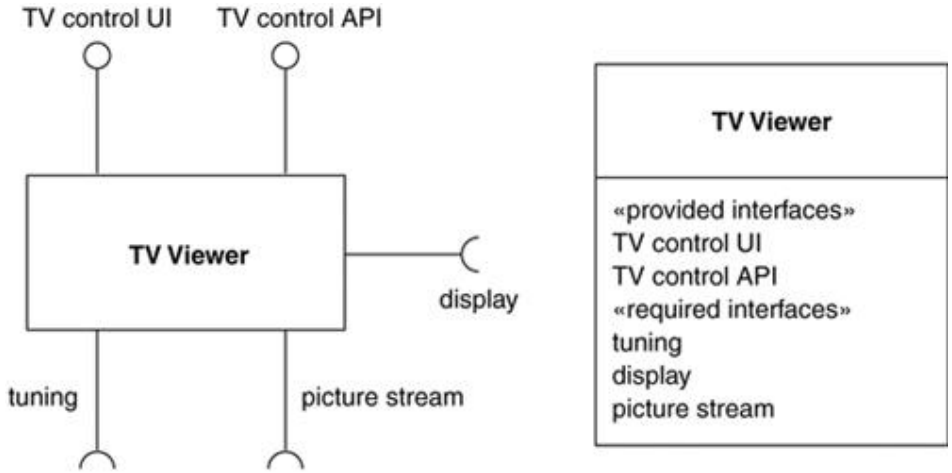
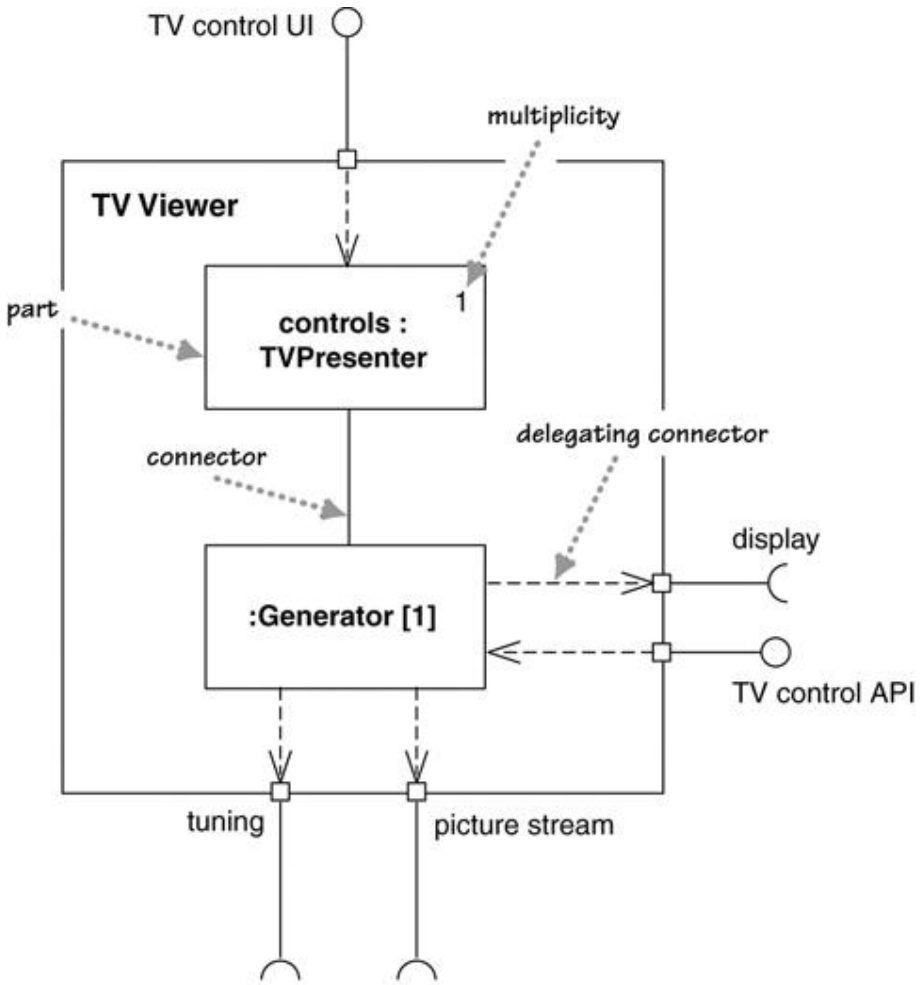


Figure 13.2 shows how this class is decomposed internally into two parts and which parts support and require the different interfaces. Each part is named in the form `name : class`, with both elements individually optional. Parts are not instance specifications, so they are bolded rather than underlined.

Figure 13.2. Internal view of a component (example suggested by Jim Rumbaugh)

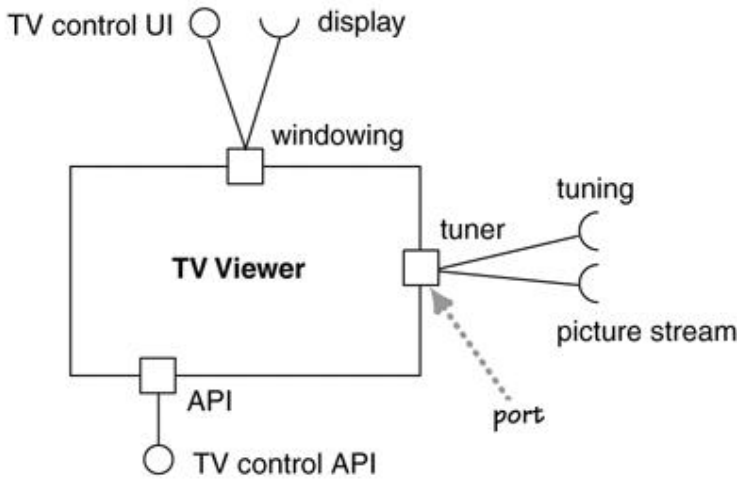


You can show how many instances of a part are present. Figure 13.2 says that each TV Viewer contains one generator part and one controls part.

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