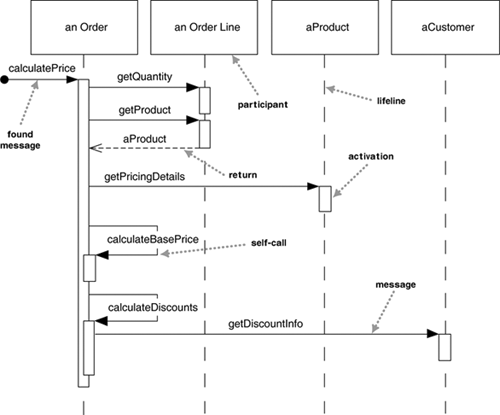
**Sequence Diagrams**

**Interaction diagrams** describe how groups of objects collaborate in some behavior. The UML defines several forms of interaction diagram, of which the most common is the **sequence diagram**.

We have an order and are going to invoke a command on it to calculate its price. To do that, the order needs to look at all the line items on the order and determine their prices, which are based on the pricing rules of the order line’s products. Having done that for all the line items, the order then needs to compute an overall discount, which is based on rules tied to the customer.

Sequence diagrams show the interaction by showing each participant with a lifeline that runs vertically down the page and the ordering of messages by reading down the page.

1



One of the nice things about a sequence diagram is that I almost don’t have to explain the notation. You can see that an instance of order sends getQuantity and getProduct messages to the order line. You can also see how we show the order invoking a method on itself and how that method sends getDiscountInfo to an instance of customer. The diagram, however, doesn’t show everything very well. The sequence of messages getQuantity, getProduct, getPricingDetails, and calculateBasePrice needs to be done for each order line on the order, while calculateDiscounts is invoked just once.

A fuller syntax is name : Class, where both the name and the class are optional, but you must keep the colon if you use the class.

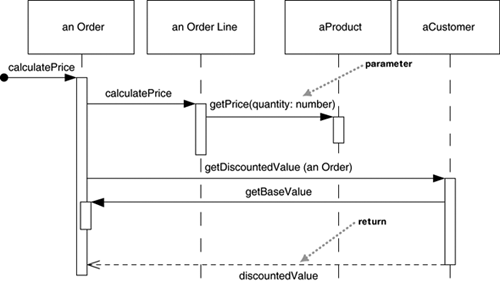
Each lifeline has an activation bar that shows when the participant is active in the interaction. This corresponds to one of the participant’s methods being on the stack. Activation bars are optional in UML.

Naming often is useful to correlate participants on the diagram. The call getProduct is shown returning aProduct, which is the same name, and therefore the same participant, as the aProduct that the getPricingDetails call is sent to.

The first message doesn’t have a participant that sent it, as it comes from an undetermined source. It’s called a found message.

**Another approach to this scenario**

**2**



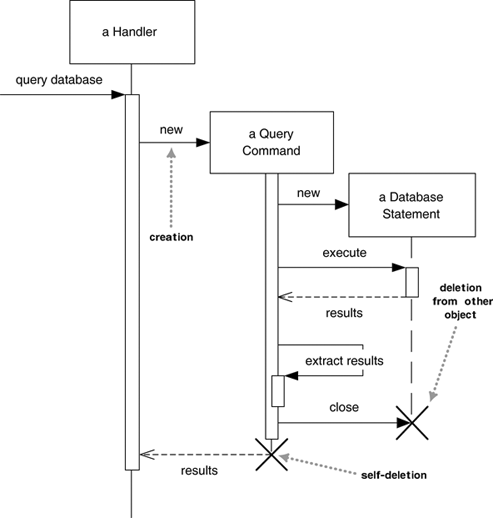
The basic problem is still the same, but the way in which the participants collaborate to implement it is very different. The Order asks each Order Line to calculate its own Price. The Order Line itself further hands off the calculation to the Product; note how we show the passing of a parameter. imilarly, to calculate the discount, the Order invokes a method on the Customer. Because it needs information from the Order to do this, the Customer makes a reentrant call (getBaseValue) to the Order to get the data.

The first thing to note about these two diagrams is how clearly the sequence diagram indicates the differences in how the participants interact. This is the great strength of interaction diagrams. They aren’t good at showing details of algorithms, such as loops and conditional behavior, but they make the calls between participants crystal clear and give a really good picture about which participants are doing which processing. The second thing to note is the clear difference in styles between the two interactions. 1 is centralized control, with one participant pretty much doing all the processing and other participants there to supply data. 2 uses distributed control, in which the processing is split among many participants, each one doing a little bit of the algorithm.

One of the main goals of good design is to localize the effects of change. Data and behavior that accesses that data often change together. So putting the data and the behavior that uses it together in one place is the first rule of object-oriented design. Furthermore, by distributing control, you create more opportunities for using polymorphism rather than using conditional logic. If the algorithms for product pricing are different for different types of product, the distributed control mechanism allows us to use subclasses of product to handle these variations. In general the OO style is to use a lot of little objects with a lot of little methods that give us a lot of plug points for overriding and variation.

**Creating and Deleting Participants**

To create a participant, you draw the message arrow directly into the participant box. A message name is optional here if you are using a constructor, but I usually mark it with “new” in any case. If the participant immediately does something once it’s created, such as the query command, you start an activation right after the participant box. Deletion of a participant is indicated by big X. A message arrow going into the X indicates one participant explicitly deleting another; an X at the end of a lifeline shows a participant deleting itself.



**Loops, Conditionals, and the like**

If you want to show control structures like this, you are better off with an activity diagram or indeed with code itself. Treat sequence diagrams as a visualization of how objects interact rather than as a way of modeling control logic.

Both loops and conditionals use interaction frames, which are ways of marking off a piece of a sequence diagram.

procedure dispatch

foreach (lineitem)

if (product.value > $10K)

careful.dispatch

else

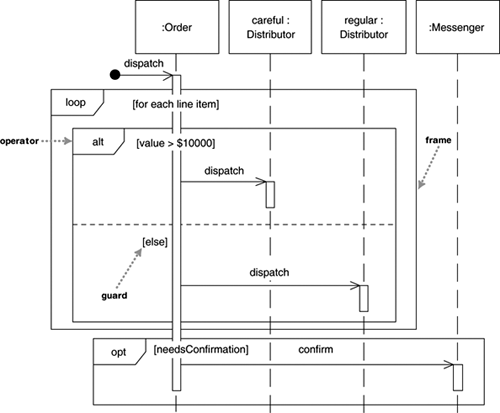
regular.dispatch

end if

end for

if (needsConfirmation) messenger.confirm

end procedure



Note how to add a descriptive note: [for each line item], [value > $10000], [needsConfirmation]

Guards are a conditional expression placed in square brackets and indicate that the message is sent only if the guard is true. Each frame has an operator and each fragment may have a guard. (Table 4.1 lists common operators for interaction frames.) To show a loop, you use the loop operand with a single fragment and put the basis of the iteration in the guard. For conditional logic, you can use an alt operator and put a condition on each fragment. Only the fragment whose guard is true will execute. If you have only one region, there is an opt operator.

Although iteration markers and guards can help, they do have weaknesses. The guards can’t indicate that a set of guards are mutually exclusive, such as the two on Figure 4.5. Both notations work only with a single message send and don’t work well when several messages coming out of a single activation are within the same loop or conditional block.

**Synchronous and asynchronous calls**

In UML 2, filled arrowheads show a synchronous message, while stick arrowheads show an asynchronous message.

If a caller sends a **synchronous** message, it must wait until the message is done, such as invoking a subroutine.

If a caller sends an **asynchronous** message, it can continue processing and doesn’t have to wait for a response. If you want to highlight asynchronous messages, I would recommend using the obsolete half-stick arrowhead.

**When to use sequence diagrams:**

You should use sequence diagrams when you want to look at the behavior of several objects within a single use case. Sequence diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.

Quiz:

1 In sequence diagrams participants contain:

A lifeline

An activation bar

A box describing an object instance

All of these

2 Deletion of a participant is indicated by an "X" on its lifeline.

True

False