Features are a good way of chunking up a system for planning an iterative project, whereby each iteration delivers a number of features. Use cases provide a narrative of how the actors use the system. Hence, although both techniques describe requirements, their purposes are different.

Although you can go directly to describing features, many people find it helpful to develop use cases first and then generate a list of features. A feature may be a whole use case, a scenario in a use case, a step in a use case, or some variant behavior, such as adding yet another depreciation method for your asset valuations, that doesn't show up in a use case narrative. Usually, features end up being more fine grained than use cases.

When to Use Use Cases

Use cases are a valuable tool to help understand the functional requirements of a system. A first pass at use cases should be made early on. More detailed versions of use cases should be worked just prior to developing that use case.

It is important to remember that use cases represent an *external* view of the system. As such, don't expect any correlations between use cases and the classes inside the system.

The more I see of use cases, the less valuable the use case diagram seems to be. With use cases, concentrate your energy on their text rather than on the diagram. Despite the fact that the UML has nothing to say about the use case text, it is the text that contains all the value in the technique.

A big danger of use cases is that people make them too complicated and get stuck. Usually, you'll get less hurt by doing too little than by doing too much. A couple of pages per use case is just fine for most cases. If you have too little, at least you'll have a short, readable document that's a starting point for questions. If you have too much, hardly anyone will read and understand it.

Where to Find Out More

Use cases were originally popularized by Ivar Jacobson in [Jacobson, OOSE].

Although use cases have been around for a while, there's been little standardization on their use. The UML is silent on the important contents of a use case and has standardized only the much less important diagrams. As a result, you can find a divergent range of opinions on use cases.

In the last few years, however, [Cockburn, use cases] has become the standard book on the subject. In this chapter, I've followed the terminology and advice of that book for the excellent reason that when we've disagreed in the past, I've usually ended up agreeing with Alistair Cockburn in the end. He also maintains a Web site at **http://usecases.org**. [Constantine and Lockwood] provides a convincing process for deriving user interfaces from use cases; also see **http://foruse.com**.

Chapter 10. State Machine Diagrams

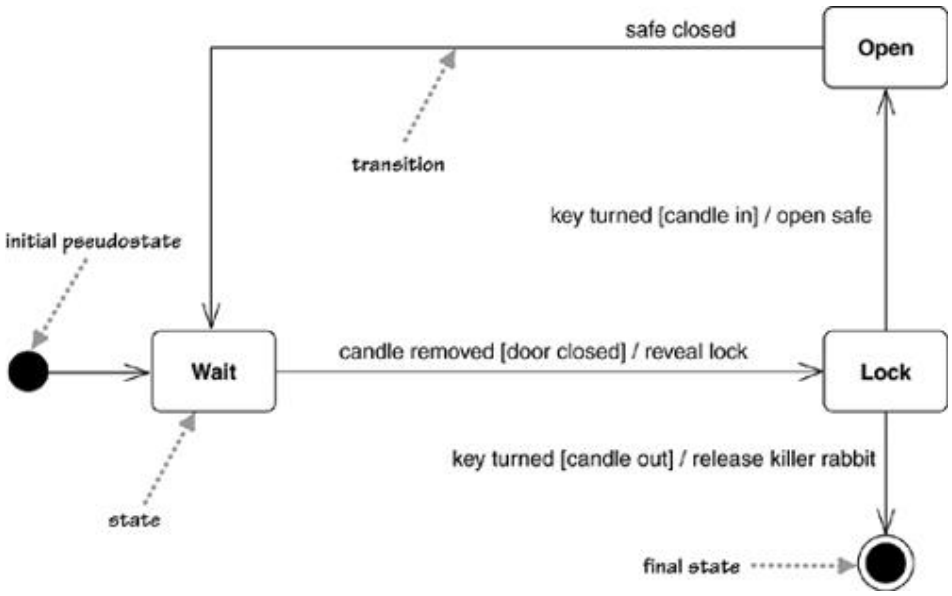
State machine diagrams are a familiar technique to describe the behavior of a system. Various forms of state diagrams have been around since the 1960s and the earliest object-oriented techniques adopted them to show behavior. In object-oriented approaches, you draw a state machine diagram for a single class to show the lifetime behavior of a single object.

Whenever people write about state machines, the examples are inevitably cruise controls or vending machines. As I'm a little bored with them, I decided to use a controller for a secret panel in a Gothic

castle. In this castle, I want to keep my valuables in a safe that's hard to find. So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal the lock only while the door is closed. Once I can see the lock, I can insert my key to open the safe. For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him.

Figure 10.1 shows a state machine diagram of the controller class that directs my unusual security system.The state diagram starts with the state of the controller object when it's created: in Figure 10.1, the Wait state. The diagram indicates this with **initial pseudostate**, which is not a state but has an arrow that points to the initial state.

Figure 10.1. A simple state machine diagram



The diagram shows that the controller can be in three states: Wait, Lock, and Open. The diagram also gives the rules by which the controller changes from state to state. These rules are in the form of transitions: the lines that connect the states.

The **transition** indicates a movement from one state to another. Each transition has a label that comes in three parts: `trigger-signature [guard]/activity`. All the parts are optional. The `trigger-signature` is usually a single event that triggers a potential change of state. The `guard`, if present, is a Boolean condition that must be true for the transition to be taken. The `activity` is some behavior that's executed during the transition. It may be any behavioral expression. The full form of a `trigger-signature` may include multiple events and parameters. So in Figure 10.1, you read the outward transition from the Wait state as "In the Wait state if the candle is removed providing the door is open, you reveal the lock and move to the Lock state."

All three parts to a transition are optional. A missing activity indicates that you don't do anything during the transition. A missing guard indicates that you always take the transition if the event occurs. A missing trigger-signature is rare but does occur. It indicates that you take the transition immediately, which you see mostly with activity states, which I'll come to in a moment.

When an event occurs in a state, you can take only one transition out of it. So if you use multiple transitions with the same event, as in the Lock state of Figure 10.1, the guards must be mutually exclusive. If an event occurs and no transition is valid—for example, a safe-closed event in the Wait state or a candle-removed event with the door closed—the event is ignored.

The final state indicates that the state machine is completed, implying the deletion of the controller object. Thus, if someone should be so careless as to fall for my trap, the controller object terminates, so I would need to put the rabbit in its cage, mop the floor, and reboot the system.

Remember that state machines can show only what the object directly observes or activates. So although you might expect me to add or remove things from the safe when it's open, I don't put that on the state diagram, because the controller cannot tell.

When developers talk about objects, they often refer to the state of the objects to mean the combination of all the data in the fields of the objects. However, the state in a state machine diagram is a more abstract notion of state; essentially, different states imply a different way of reacting to events.

Internal Activities

States can react to events without transition, using **internal activities:** putting the event, guard, and activity inside the state box itself.

Figure 10.2 shows a state with internal activities of the character and help events, as you might find on a UI text field. An internal activity is similar to a **self-transition:** a transition that loops back to the same state. The syntax for internal activities follows the same logic for event, guard, and procedure.

Figure 10.2. Internal events shown with the typing state of a text field

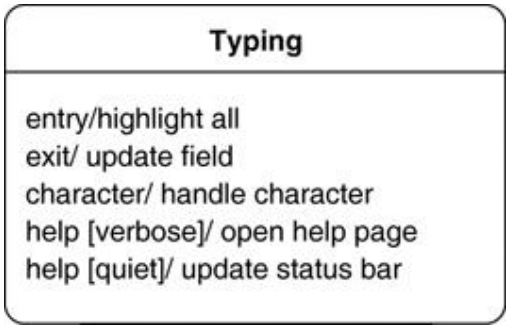


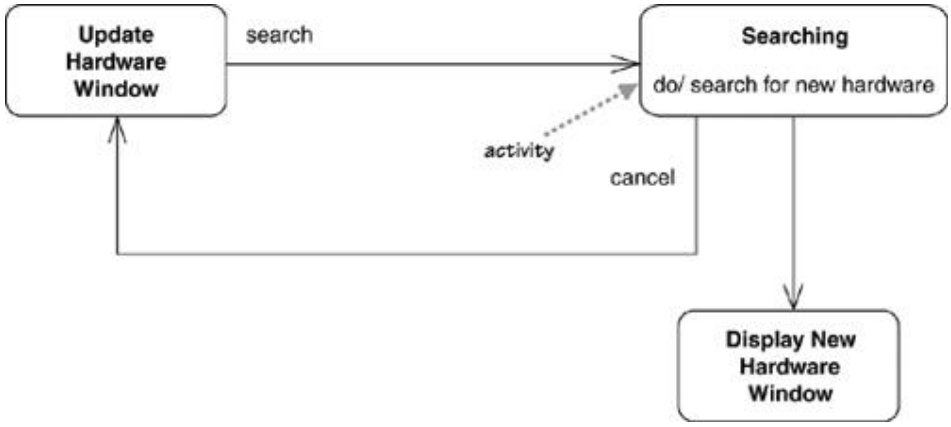
Figure 10.2 also shows two special activities: the entry and exit activities. The **entry activity** is executed whenever you enter a state; the **exit activity**, whenever you leave. However, internal activities do not trigger the entry and exit activities; that is the difference between internal activities and self-transitions.

Activity States

In the states I've described so far, the object is quiet and waiting for the next event before it does something. However, you can have states in which the object is doing some ongoing work.

The Searching state in Figure 10.3 is such an **activity state:** The ongoing activity is marked with the `do/`; hence the term **do-activity**. Once the search is completed, any transitions without an activity, such as the one to display new hardware, are taken. If the cancel event occurs during the activity, the do-activity is unceremoniously halted, and we go back to the Update Hardware Window state.

Figure 10.3. A state with an activity



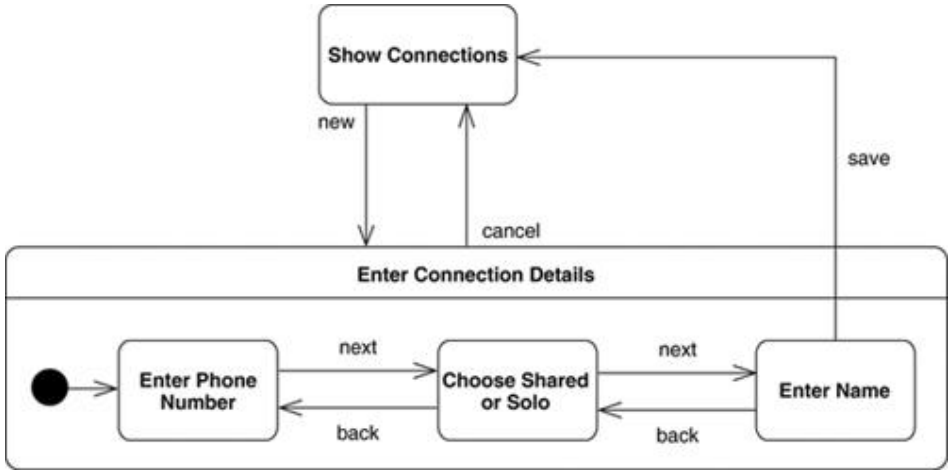
Both do-activities and regular activities represent carrying out some behavior. The critical difference between the two is that regular activities occur "instantaneously" and cannot be interrupted by regular events, while do-activities can take finite time and can be interrupted, as in Figure 10.3. Instantaneous will mean different things for different system; for hard real-time systems, it might be a few machine instructions, but for desktop software might be several seconds.

UML 1 used the term **action** for regular activities and used activity only for do-activities.

Superstates

Often, you'll find that several states share common transitions and internal activities. In these cases, you can make them substates and move the shared behavior into a superstate, as in Figure 10.4. Without the superstate, you would have to draw a cancel transition for all three states within the Enter Connection Details state.

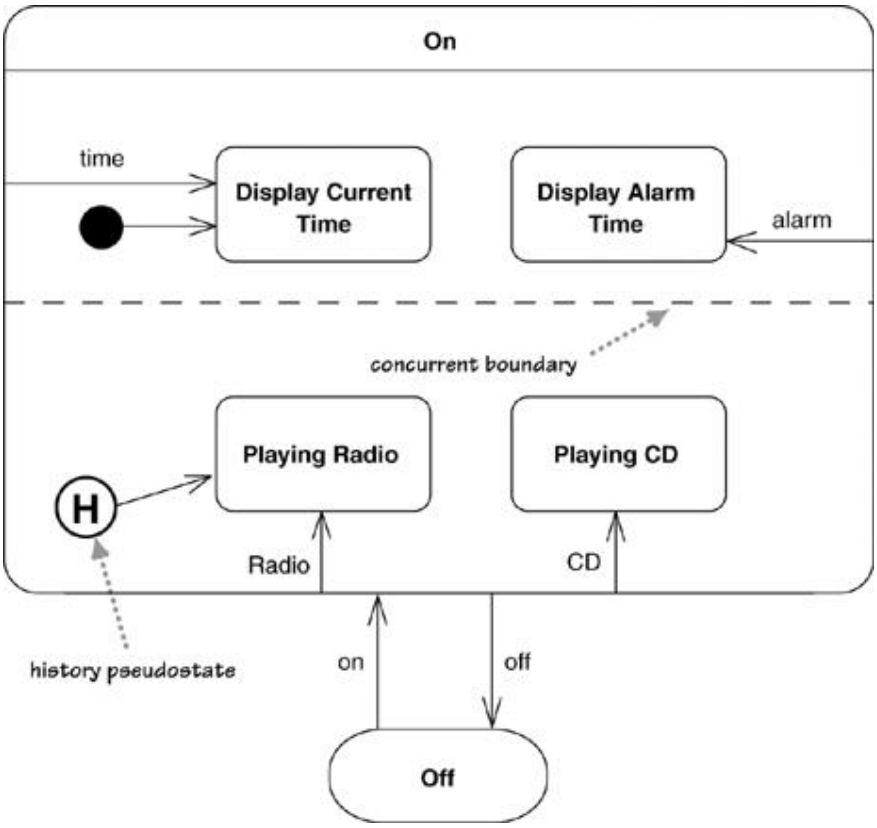
Figure 10.4. Superstate with nested substates



Concurrent States

States can be broken into several orthogonal state diagrams that run concurrently. Figure 10.5 shows a pathetically simple alarm clock that can play either CDs or the radio and show either the current time or the alarm time.

Figure 10.5. Concurrent orthogonal states



The choices CD/radio and current/alarm time are orthogonal choices. If you wanted to represent this with a nonorthogonal state diagram, you would need a messy diagram that would get very much out of hand should you want more states. Separating out the two areas of behavior into separate state diagrams makes it much clearer.

Figure 10.5 also includes a **history pseudostate**. This indicates that when the clock is switched on, the radio/CD choice goes back to the state the clock was in when it was turned off. The arrow from the history pseudostate indicates what state to be in on the first time when there is no history.

Implementing State Diagrams

A state diagram can be implemented in three main ways: nested switch, the State pattern, and state tables. The most direct approach to handling a state diagram is a nested switch statement, such as Figure 10.6. Although it's direct, it's long-winded, even for this simple case. It's also very easy for this approach to get out of control, so I don't like using it even for simple cases.

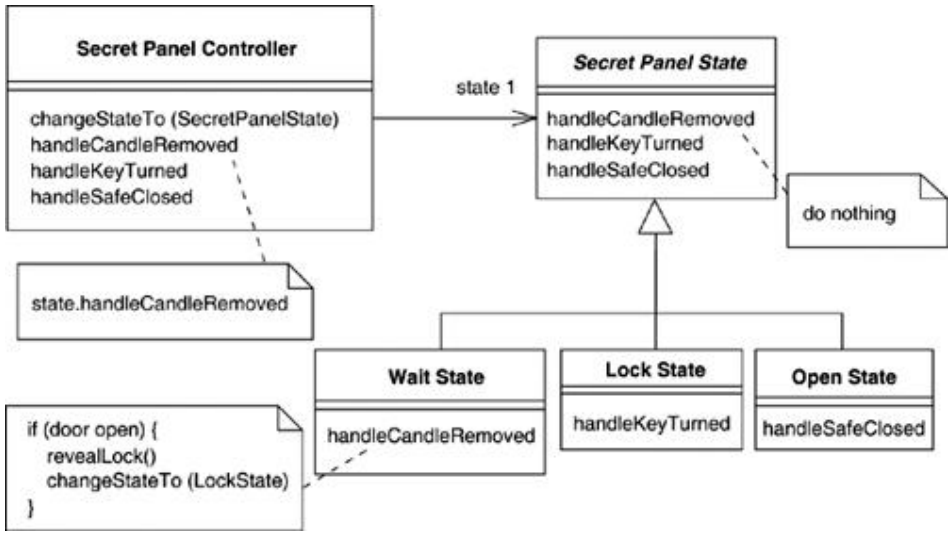
Figure 10.6 A C# nested switch to handle the state transition from Figure 10.1

```
public void HandleEvent (PanelEvent anEvent) {
  switch (CurrentState) {
    case PanelState.Open :
      switch (anEvent) {
        case PanelEvent.SafeClosed :
          CurrentState = PanelState.Wait;
          break;
      }
      break;
    case PanelState.Wait :
```

```
      switch (anEvent) {
        case PanelEvent.CandleRemoved :
          if (isDoorOpen) {
            RevealLock();
            CurrentState = PanelState.Lock;
          }
          break;
      }
      break;
    case PanelState.Lock :
      switch (anEvent) {
        case PanelEvent.KeyTurned :
          if (isCandleIn) {
            OpenSafe();
            CurrentState = PanelState.Open;
          } else {
            ReleaseKillerRabbit();
            CurrentState = PanelState.Final;
          }
          break;
      }
      break;
   }
  }
}
```

The **State pattern** [Gang of Four] creates a hierarchy of state classes to handle behavior of the states. Each state in the diagram has one state subclass. The controller has methods for each event, which simply forwards to the state class. The state diagram of Figure 10.1 would yield an implementation indicated by the classes of Figure 10.7.

Figure 10.7. A State pattern implementation for Figure 10.1



The top of the hierarchy is an abstract class that implements all the event-handling methods to do nothing. For each concrete state, you simply override the specific event methods for which that state has transitions.

The **state table** approach captures the state diagram information as data. So Figure 10.1 might end up represented in a table like Table 10.1. We then build either an interpreter that uses the state table at runtime or a code generator that generates classes based on the state table.

Obviously, the state table is more work to do once, but then you can use it every time you have a state problem to hold. A runtime state table can also be modified without recompilation, which in

some contexts is quite handy. The state pattern is easier to put together when you need it, and although it needs a new class for each state, it's a small amount of code to write in each case.

These implementations are pretty minimal, but they should give you an idea of how to go about implementing state diagrams. In each case, implementing state models leads to very boilerplate code, so it's usually best to use some form of code generation to do it.

Table 10.1. A State Table for Figure 10.1

Source State	Target State	Event	Guard	Procedure
Wait	Lock	Candle removed	Door open	Reveal lock
Lock	Open	Key turned	Candle in	Open safe
Lock	Final	Key turned	Candle out	Release killer rabbit
Open	Wait	Safe closed		

When to Use State Diagrams

State diagrams are good at describing the behavior of an object across several use cases. State diagrams are not very good at describing behavior that involves a number of objects collaborating. As such, it is useful to combine state diagrams with other techniques. For instance, interaction diagrams (see Chapter 4) are good at describing the behavior of several objects in a single use case, and activity diagrams (see Chapter 11) are good at showing the general sequence of activities for several objects and use cases.

Not everyone finds state diagrams natural. Keep an eye on how people are working with them. It may be that your team does not find state diagrams useful to its way of working. That is not a big problem; as always, you should remember to use the mix of techniques that works for you.

If you do use state diagrams, don't try to draw them for every class in the system. Although this approach is often used by high-ceremony completists, it is almost always a waste of effort. Use state diagrams only for those classes that exhibit interesting behavior, where building the state diagram helps you understand what is going on. Many people find that UI and control objects have the kind of behavior that is useful to depict with a state diagram.

Where to Find Out More

Both the *User Guide* [Booch, UML user] and the *Reference Manual* [Rumbaugh, UML Reference] have more information on state diagrams. Real-time designers tend to use state models a lot, so it's no surprise that [Douglass]) has a lot to say about state diagrams, including information on how to implement them. [Martin] contains a very good chapter on the various ways of implementing state diagrams.

Chapter 11. Activity Diagrams

Activity diagrams are a technique to describe procedural logic, business process, and work flow. In many ways, they play a role similar to flowcharts, but the principal difference between them and flowchart notation is that they support parallel behavior.

Activity diagrams have seen some of the biggest changes over the versions of the UML, so they have, not surprisingly, been significantly extended and altered again for UML 2. In UML 1, activity

diagrams were seen as special cases of state diagrams. This caused a lot of problems for people modeling work flows, which activity diagrams are well suited for. In UML 2, that tie was removed.

Figure 11.1 shows a simple example of an activity diagram. We begin at the **initial node** action and then do the action Receive Order. Once that is done, we encounter a fork. A **fork** has one incoming flow and several outgoing concurrent flows.

Figure 11.1. A simple activity diagram

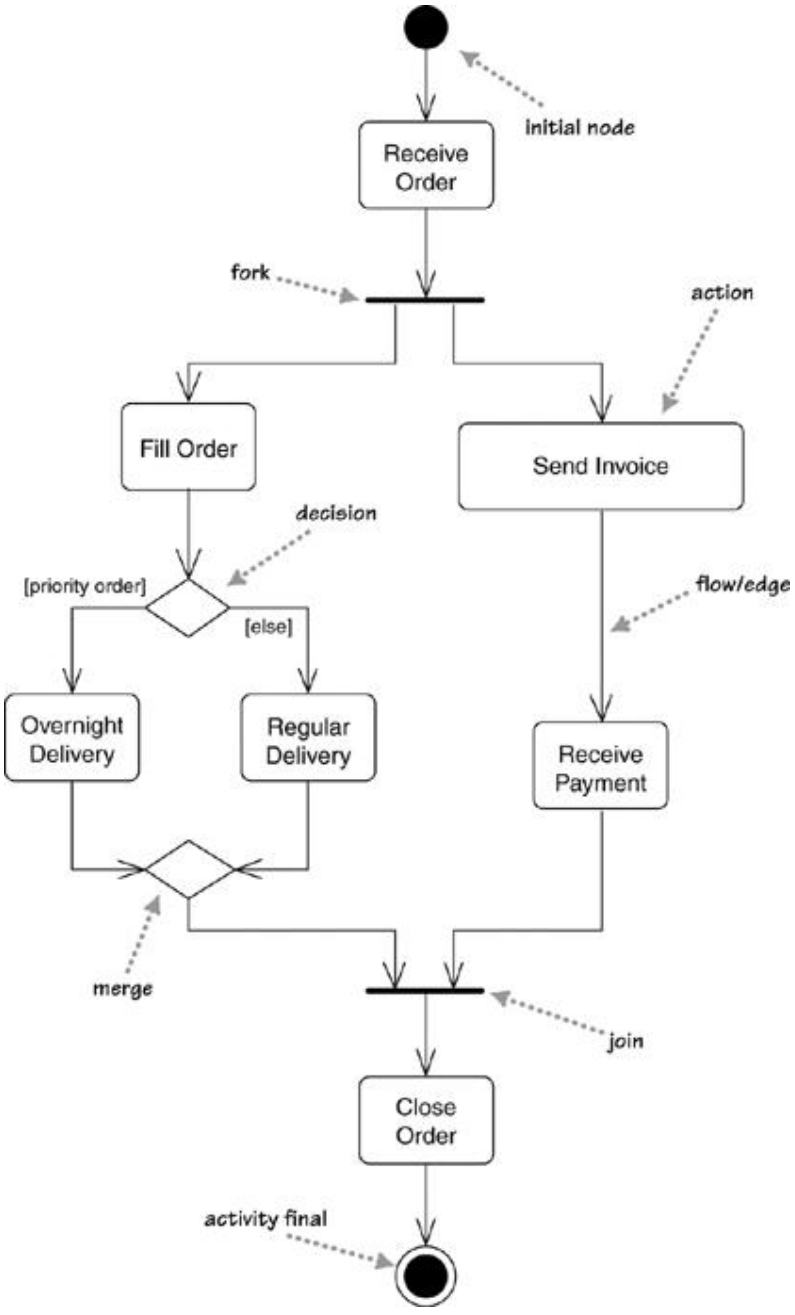


Figure 11.1 says that Fill Order, Send Invoice, and the subsequent actions occur in parallel. Essentially, this means that the sequence between them is irrelevant. I could fill the order, send the invoice, deliver, and then receive payment; or, I could send the invoice, receive the payment, fill the order, and then deliver: You get the picture.

I can also do these actions by interleaving. I grab the first line item from stores, type up the invoice, grab the second line item, put the invoice in an envelope, and so forth. Or, I could do some of this

simultaneously: type up the invoice with one hand while I reach into my stores with another. Any of these sequences is correct, according to the diagram.

The activity diagram allows whoever is doing the process to choose the order in which to do things. In other words, the diagram merely states the essential sequencing rules I have to follow. This is important for business modeling because processes often occur in parallel. It's also useful for concurrent algorithms, in which independent threads can do things in parallel.

When you have parallelism, you'll need to synchronize. We don't close the order until it is delivered and paid for. We show this with the **join** before the Close Order action. With a join, the outgoing flow is taken only when all the incoming flows reach the join. So you can close the order only when you have both received the payment and delivered.

UML 1 had particular rules for balancing the forks and joins, as activity diagrams were special cases of state diagrams. With UML 2, such balancing is no longer needed.

You'll notice that the nodes on an activity diagram are called actions, not activities. Strictly, an activity refers to a sequence of actions, so the diagram shows an activity that's made up of actions.

Conditional behavior is delineated by decisions and merges. A **decision**, called *branch* in UML 1, has a single incoming flow and several guarded out-bound flows. Each outbound flow has a guard: a Boolean condition placed inside square brackets. Each time you reach a decision, you can take only one of the outbound flows, so the guards should be mutually exclusive. Using `[else]` as a guard indicates that the `[else]` flow should be used if all the other guards on the decision are false.

In Figure 11.1, after an order is filled, there is a decision. If you have a rush order, you do an Overnight Delivery; otherwise, you do a Regular Delivery.

A **merge** has multiple input flows and a single output. A merge marks the end of conditional behavior started by a decision.

In my diagrams, each action has a single flow coming in and a single flow going out. In UML 1, multiple incoming flows had an implicit merge. That is, your action would execute if any flow triggered. In UML 2, this has changed so there's an implicit join instead; thus, the action executes only if all flows trigger. As a result of this change, I recommend that you use only a single incoming and outgoing flow to an action and show all joins and merges explicitly; that will avoid confusion.

Decomposing an Action

Actions can be decomposed into subactivities. I can take the delivery logic of Figure 11.1 and define it as its own activity (Figure 11.2). Then I can call it as an action (Figure 11.3 on page 121).

Figure 11.2. A subsidiary activity diagram

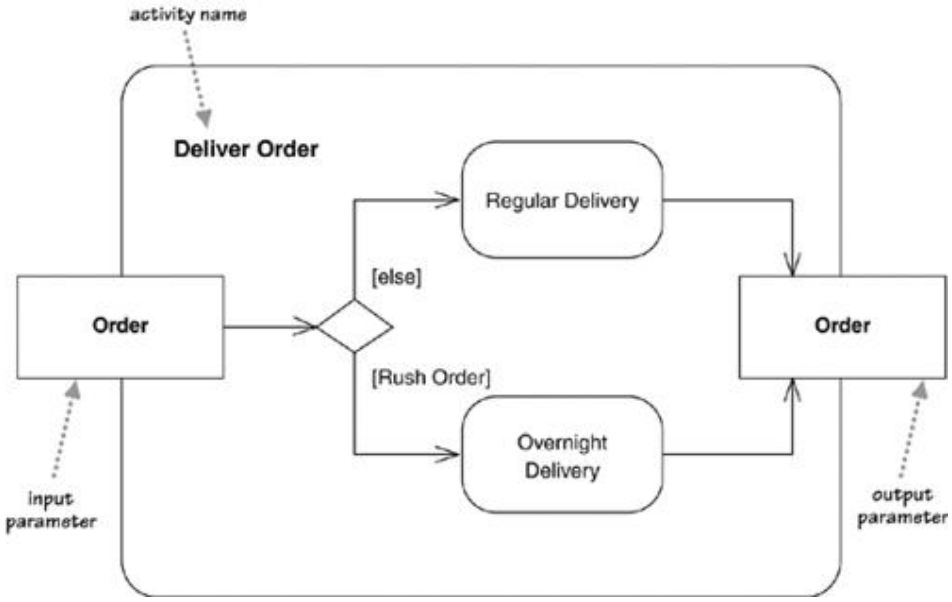
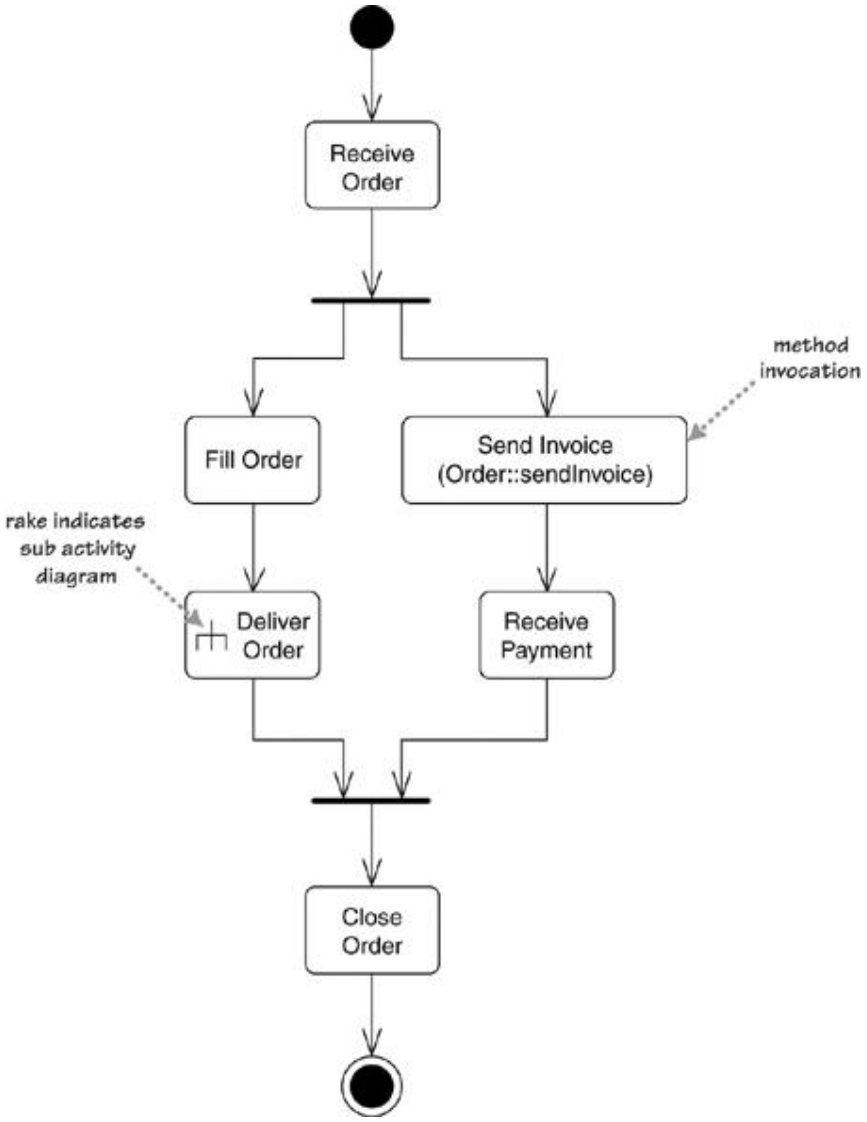


Figure 11.3. The activity of Figure 11.1 modified to invoke the activity in Figure 11.2



Actions can be implemented either as subactivities or as methods on classes. You can show a subactivity by using the rake symbol. You can show a call on a method with syntax `class-name::method-name`. You can also write a code fragment into the action symbol if the invoked behavior isn't a single method call.

And There's More

I should stress that this chapter only scratches the surface on activity diagrams. As with so much of the UML, you could write a whole book on this one technique alone. Indeed, I think that activity diagrams would make a very suitable topic for a book that really dug into the notation and how to use it.

The vital question is how widely they get used. Activity diagrams aren't the most widely used UML technique at the moment, and their flow-modeling progenitors weren't very popular either. Diagrammatic techniques haven't yet caught on much for describing behavior in this kind of way. On the other hand, there are signs in a number of communities of a pent-up demand that a standard technique will help to satisfy.

When to Use Activity Diagrams

The great strength of activity diagrams lies in the fact that they support and encourage parallel behavior. This makes them a great tool for work flow and process modeling, and indeed much of the push in UML 2 has come from people involved in work flow.

You can also use an activity diagram as a UML-compliant flowchart. Although this allows you to do flowcharts in a way that sticks with the UML, it's hardly very exciting. In principle, you can take advantages of the forks and joins to describe parallel algorithms for concurrent programs. Although I don't travel in concurrent circles that much, I haven't seen much evidence of people using them there. I think the reason is that most of the complexity of concurrent programming is in avoiding contention on data, and activity diagrams don't help much with that.

The main strength of doing this may come with people using UML as a programming language. In this case, activity diagrams represent an important technique to represent behavioral logic.

I've often seen activity diagrams used to describe a use case. The danger of this approach is that often, domain experts don't follow them easily. If so, you'd be better off with the usual textual form.

Where to Find Out More

Although activity diagrams have always been rather complicated and are even more so with UML 2, there hasn't been a good book that describes them in depth. I hope this gap will get filled someday.

Various flow-oriented techniques are similar in style to activity diagrams. One of the better known—but hardly well known—is Petri Nets, for which **http://www.daimi.au.dk/PetriNets/** is a good Web site.

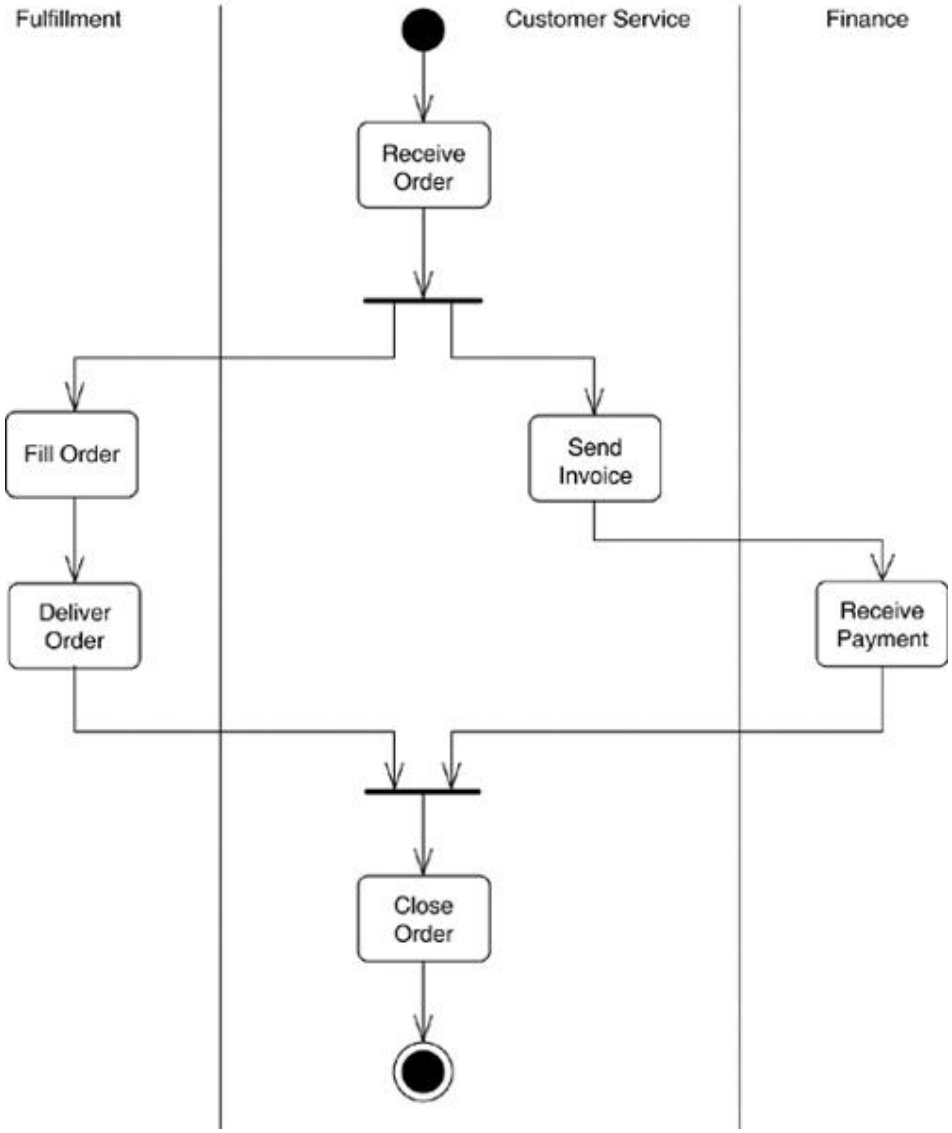
Partitions

Activity diagrams tell you what happens, but they do not tell you who does what. In programming, this means that the diagram does not convey which class is responsible for each action. In business

process modeling, this does not convey which part of an organization carries out which action. This isn't necessarily a problem; often, it makes sense to concentrate on what gets done rather than on who does what parts of the behavior.

If you want to show who does what, you can divide an activity diagram into **partitions**, which show which actions one class or organization unit carries out. Figure 11.4 (on page 122) shows a simple example of this, showing how the actions involved in order processing can be separated among various departments.

Figure 11.4. Partitions on an activity diagram



The partitioning of Figure 11.4 is a simple one-dimensional partitioning. This style is often referred to as **swim lanes**, for obvious reasons and was the only form used in UML 1.x. In UML 2, you can use a two-dimensional grid, so the swimming metaphor no longer holds water. You can also take each dimension and divide the rows or columns hierarchically.

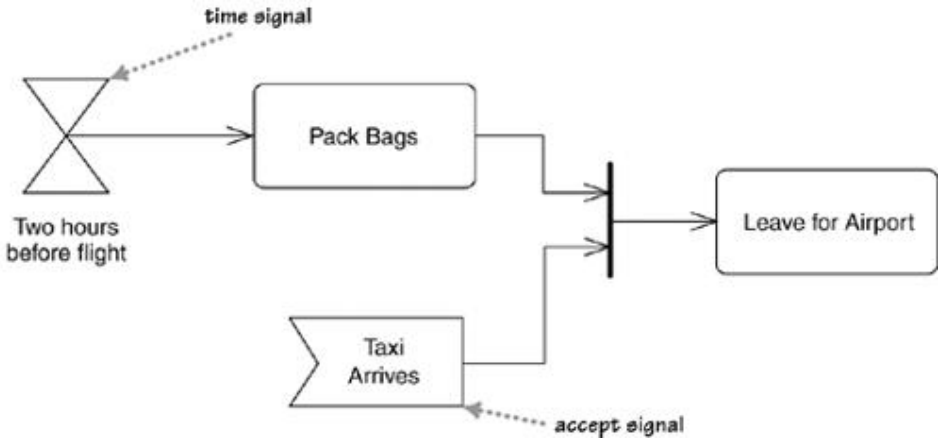
Signals

In the simple example of Figure 11.1, activity diagrams have a clearly defined start point, which corresponds to an invocation of a program or routine. Actions can also respond to signals.

A **time signal** occurs because of the passage of time. Such signals might indicate the end of a month in a financial period or each microsecond in a real-time controller.

Figure 11.5 shows an activity that listens for two signals. A **signal** indicates that the activity receives an event from an outside process. This indicates that the activity constantly listens for those signals, and the diagram defines how the activity reacts.

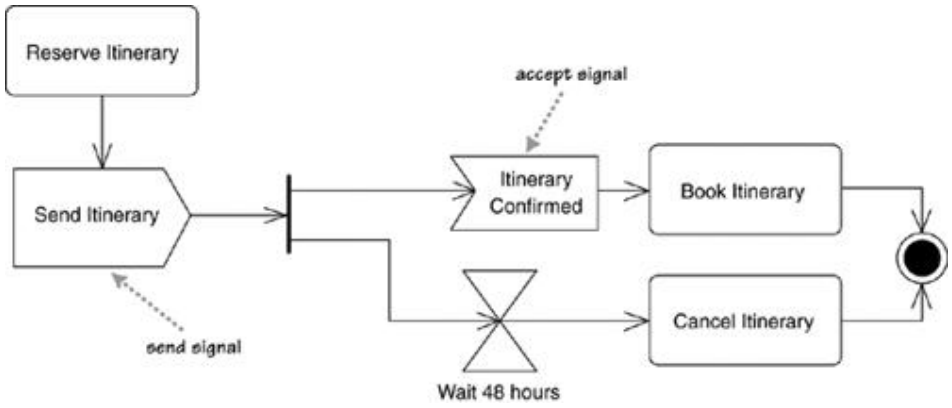
Figure 11.5. Signals on an activity diagram



In the case of Figure 11.5, 2 hours before my flight leaves, I need to start packing my bags. If I'm quick to pack them, I still cannot leave until the taxi arrives. If the taxi arrives before my bags are packed, it has to wait for me to finish before we go.

As well as accepting signals, we can send them. This is useful when we have to send a message and then wait for a reply before we can continue. Figure 11.6 shows a good example of this with a common idiom of timing out. Note that the two flows are in a race: The first to reach the final state will win and terminate the other flow.

Figure 11.6. Sending and receiving signals



Although accepts are usually just waiting for an external event, we can also show a flow going into them. That indicates that we don't start listening until the flow triggers the accept.

Tokens

If you're sufficiently brave to venture into the demonic depths of the UML specification, you'll find that the activity section of the specification talks a lot about tokens and their production and consumption. The initial node creates a token, which then passes to the next action, which executes and then passes the token to the next. At a fork, one token comes in, and the fork produces a token

on each of its outward flows. Conversely, on a join, as each inbound token arrives, nothing happens until all the tokens appear at the join; then a token is produced on the outward flow.

You can visualize the tokens with coins or counters moving across the diagram. As you get to more complicated examples of activity diagrams, tokens often make it easier to visualize things.

Flows and Edges

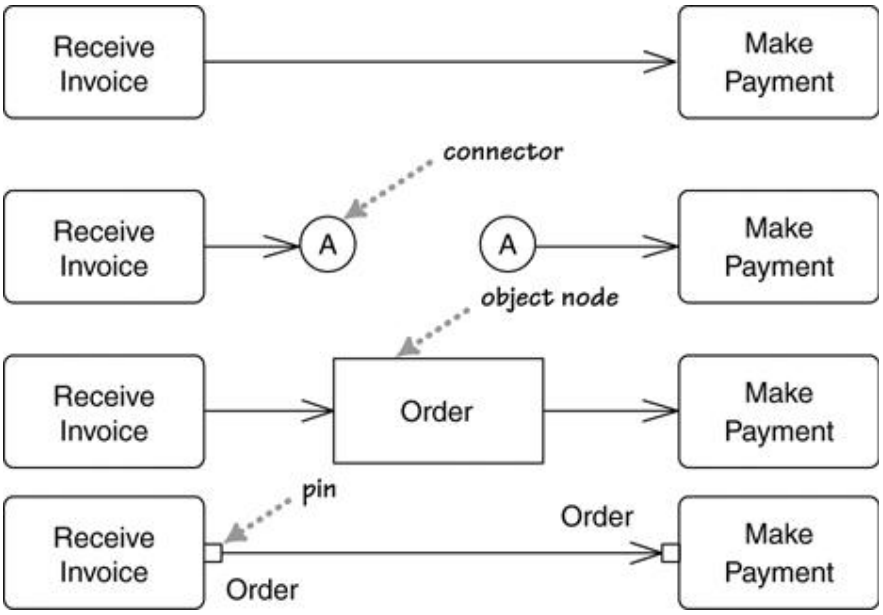
UML 2 uses the terms **flow** and **edge** synonymously to describe the connections between two actions. The simplest kind of edge is the simple arrow between two actions. You can give an edge a name if you like, but most of the time, a simple arrow will suffice.

If you're having difficulty routing lines, you can use connectors, which simply save you having to draw a line the whole distance. When you use connectors, you must use them in pairs: one with incoming flow, one with an outgoing flow, and both with the same label. I tend to avoid using connectors if at all possible, as they break up the visualization of the flow of control.

The simplest edges pass a token that has no meaning other than to control the flow. However, you can also pass objects along edges; the objects then play the role of tokens, as well as carry data. If you are passing an object along the edge, you can show that by putting a class box on the edge, or you can use pins on the actions, although pins imply some more subtleties that I'll describe shortly.

All the styles shown in Figure 11.7 are equivalent; you should use whichever conveys best what you are trying to communicate. Most of the time, the simple arrow is quite enough.

Figure 11.7. Four ways of showing an edge



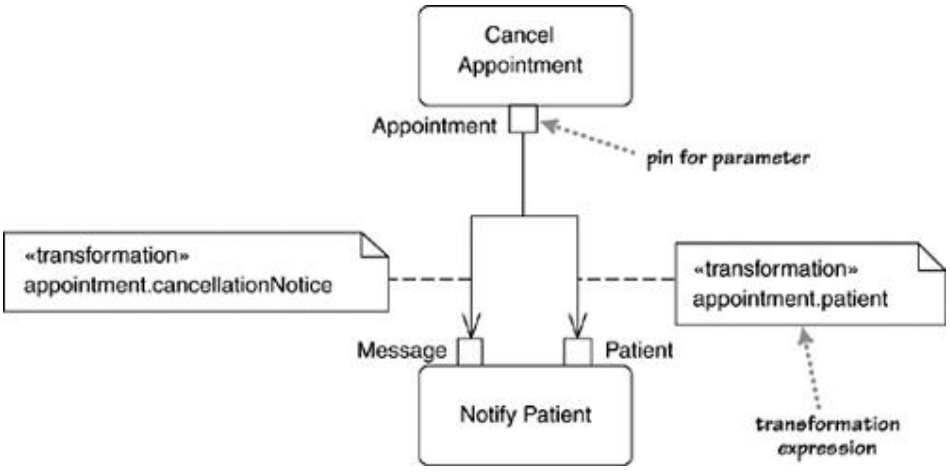
Pins and Transformations

Actions can have parameters, just as methods do. You don't need to show information about parameters on the activity diagram, but if you wish, you can show them with **pins**. If you're decomposing an action, pins correspond to the parameter boxes on the decomposed diagram.

When you're drawing an activity diagram strictly, you have to ensure that the output parameters of an outbound action match the input parameters of another. If they don't match, you can indicate a

transformation (Figure 11.8) to get from one to another. The transformation must be an expression that's free of side effects: essentially, a query on the output pin quary that supplies an object of the right type for the input pin.

Figure 11.8. Transformation on a flow



You don't have to show pins on an activity diagram. Pins are best when you want to look at the data needed and produced by the various actions. In business process modeling, you can use pins to show the resources produced and consumed by actions.

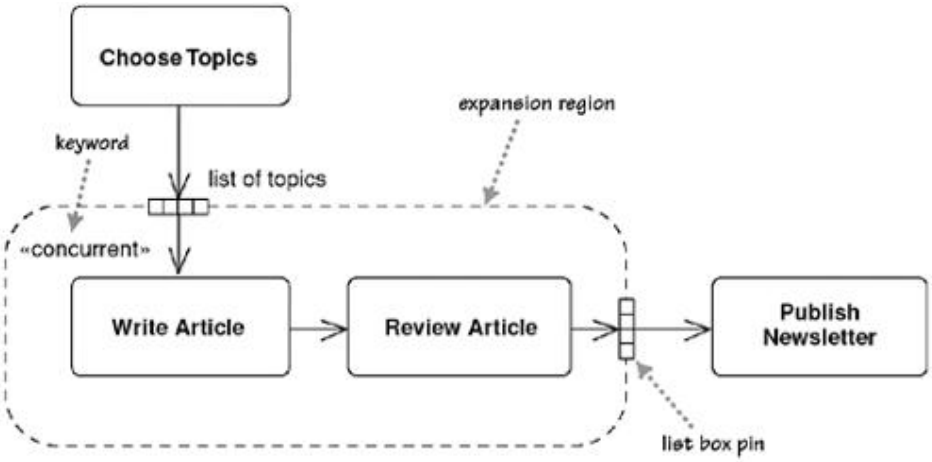
If you use pins, it's safe to show multiple flows coming into the same action. The pin notation reinforces the implicit join, and UML 1 didn't have pins, so there's no confusion with the earlier assumptions.

Expansion Regions

With activity diagrams, you often run into situations in which one action's output triggers multiple invocations of another action. There are several ways to show this, but the best way is to use an expansion region. An **expansion region** marks an activity diagram area where actions occur once for each item in a collection.

In Figure 11.9, the Choose Topics action generates a list of topics as its output. Each element of this list then becomes a token for input to the Write Article action. Similarly, each Review Article action generates a single article that's added to the output list of the expansion region. When all the tokens in the expansion region end up in the output collection, the region generates a single token for the list that's passed to Publish Newsletter.

Figure 11.9. Expansion region

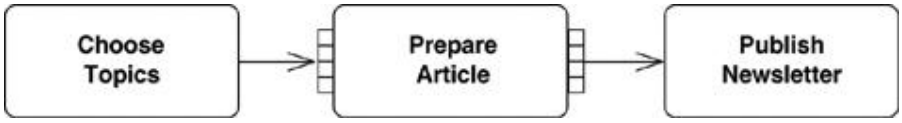


In this case, you have the same number of items in the output collection as you do in the input collection. However, you may have fewer, in which case the expansion region acts as a filter.

In Figure 11.9, all the articles are written and reviewed in parallel, which is marked by the «concurrent» keyword. You can also have an iterative expansion region. Iterative regions must fully process each input element one at a time.

If you have only a single action that needs multiple invocation, you use the shorthand of Figure 11.10. The shorthand assumes concurrent expansion, as that's the most common. This notation corresponds to the UML 1 concept of dynamic concurrency.

Figure 11.10. Shorthand for a single action in an expansion region

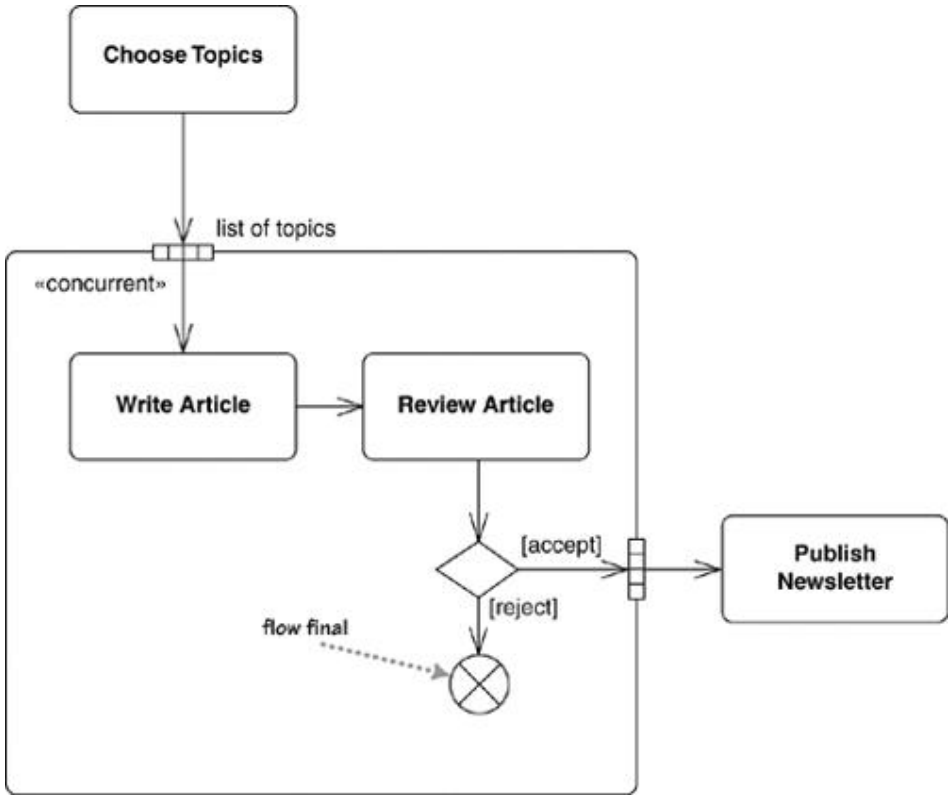


Flow Final

Once you get multiple tokens, as in an expansion region, you often get flows that stop even when the activity as a whole doesn't end. A **flow final** indicates the end of one particular flow, without terminating the whole activity.

Figure 11.11 shows this by modifying the example of Figure 11.9 to allow articles to be rejected. If an article is rejected, the token is destroyed by the flow final. Unlike an activity final, the rest of the activity can continue. This approach allows expansion regions to act as filters, whereby the output collection is smaller than the input collection.

Figure 11.11. Flow finals in an activity



Join Specifications

By default, a join lets execution pass on its outward flow when all its input flows have arrived at the join. (Or in more formal speak, it emits a token on its output flow when a token has arrived on each input flow.) In some cases, particularly when you have a flow with multiple tokens, it's useful to have a more involved rule.

A **join specification** is a Boolean expression attached to a join. Each time a token arrives at the join, the join specification is evaluated and if true, an output token is emitted. So in Figure 11.12, whenever I select a drink or insert a coin, the machine evaluates the join specification. The machine slakes my thirst only if I've put in enough money. If, as in this case, you want to indicate that you have received a token on each input flow, you label the flows and include them in the join specification.

Figure 11.12. Join specification