- Use a sequence diagram if you want to emphasize the time ordering of messages. Use a collaboration diagram if you want to emphasize the organization of the objects involved in the interaction.
- Lay out its elements to minimize lines that cross.
- Use notes and color as visual cues to draw attention to important features of your diagram.
- Use branching sparingly; you can represent complex branching much better using activity diagrams.

Chapter 19. Activity Diagrams

In this chapter

- Modeling a workflow
- Modeling an operation
- Forward and reverse engineering

Sequence diagrams, collaboration diagrams, statechart diagrams, and use case diagrams also model the dynamic aspects of systems; sequence and collaboration diagrams are discussed in Chapter 18*; statechart diagrams are discussed in* Chapter 24*; use case diagrams are discussed in* Chapter 17*; actions are discussed in* Chapter 15.

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity.

You use activity diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the sequential (and possibly concurrent) steps in a computational process. With an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control. Activity diagrams may stand alone to visualize, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. An activity is an ongoing nonatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that results in a change in state of the system or the return of a value.

Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Getting Started

Consider the workflow associated with building a house. First, you select a site. Next, you commission an architect to design your house. After you've settled on the plan, your developer asks for bids to price the house. Once you agree on a price and a plan, construction can begin. Permits are secured, ground is broken, the foundation is poured, the framing is erected, and so on, until everything is done. You're then handed the keys and a certificate of occupancy, and you take possession of the house.

Although that's a tremendous simplification of what really goes on in a construction process, it does capture the critical path of the workflow. In a real project, there are lots of parallel activities among various trades. Electricians can be working at the same time as plumbers and carpenters, for example. You'll also encounter conditions and branches. For example, depending on the result of soils tests, you might have to blast, dig, or float. There might even be loops. For example, a building inspection might reveal code violations that result in scrap and rework.

In the construction industry, such techniques as Gantt charts and Pert charts are commonly used for visualizing, specifying, constructing, and documenting the workflow of the project.

Modeling the structural aspects of a system is discussed in Sections 2 *and* 3; *interaction diagrams are discussed in* Chapter 18.

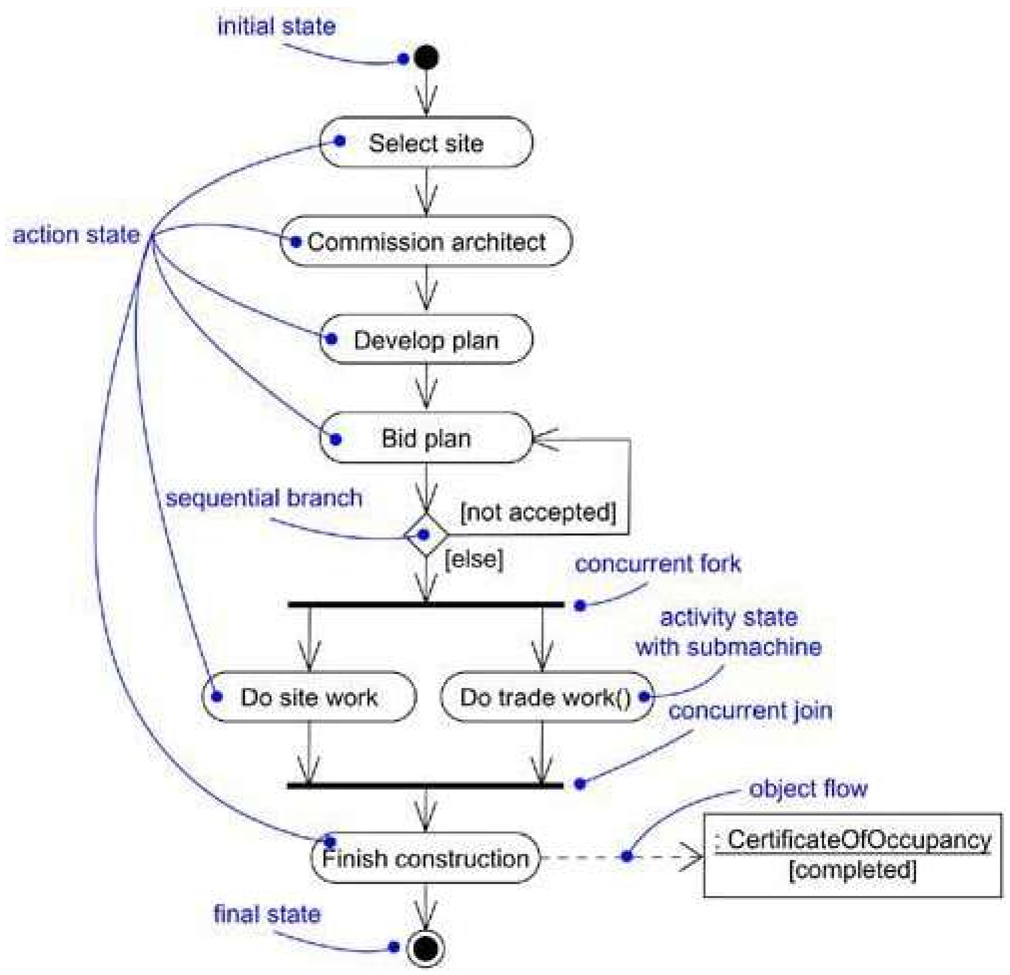
In modeling software-intensive systems, you have a similar problem. How do you best model a workflow or an operation, both of which are aspects of the system's dynamics? The answer is that you have two basic choices, similar to the use of Gantt charts and Pert charts.

On the one hand, you can build up storyboards of scenarios, involving the interaction of certain interesting objects and the messages that may be dispatched among them. In the UML, you can model these storyboards in two ways: by emphasizing the time ordering of messages (using sequence diagrams) or by emphasizing the structural relationships among the objects that interact (using collaboration diagrams). Interaction diagrams such as these are akin to Gantt charts, which focus on the objects (resources) that carry out some activity over time.

Actions are discussed in Chapter 15.

On the other hand, you can model these dynamic aspects using activity diagrams, which focus first on the activities that take place among objects, as Figure 19-1 shows. In that regard, activity diagrams are akin to Pert charts. An activity diagram is essentially a flowchart that emphasizes the activity that takes place over time. You can think of an activity diagram as an interaction diagram turned inside out. An interaction diagram looks at the objects that pass messages; an activity diagram looks at the operations that are passed among objects. The semantic difference is subtle, but it results in a very different way of looking at the world.

Figure 19-1 Activity Diagrams



Terms and Concepts

An *activity diagram* shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine. Activities ultimately result in some *action,* which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression. Graphically, an activity diagram is a collection of vertices and arcs.

Common Properties

The general properties of diagrams are discussed in Chapter 7.

An activity diagram is just a special kind of diagram and shares the same common properties as do all other diagrams—a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its content.

Contents

States, transitions, and state machines are discussed in Chapter 21*; objects are discussed in* Chapter 13.

Activity diagrams commonly contain

- Activity states and action states
- Transitions
- Objects

Note

An activity diagram is basically a projection of the elements found in an activity graph, a special case of a state machine in which all or most states are activity states and in which all or most transitions are triggered by completion of activities in the source state. Because an activity diagram is a kind of state machine, all the characteristics of state machines apply. That means that activity diagrams may contain simple and composite states, branches, forks, and joins.

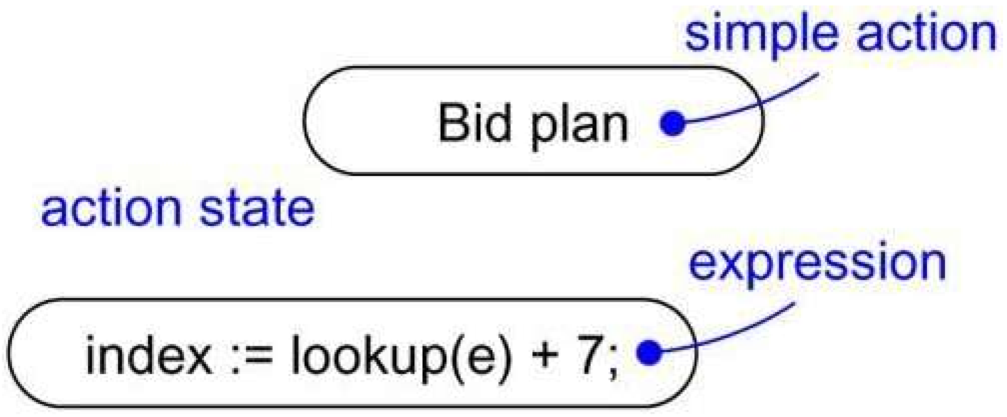
Like all other diagrams, activity diagrams may contain notes and constraints.

Action States and Activity States

Attributes and operations are discussed in Chapters 4 *and* 9*; signals are discussed in* Chapter 20*; creation and destruction of objects are discussed in* Chapter 15 *; states and state machines are discussed in* Chapter 21.

In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object. These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action. As Figure 19-2 shows, you represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.

Figure 19-2 Action States



Note

The UML does not prescribe the language of these expressions. Abstractly, you might just use structured text; more concretely, you might use the syntax and semantics of a specific programming language.

Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted. Finally, the work of an action state is generally considered to take insignificant execution time.

Modeling time and space is discussed in Chapter 23.

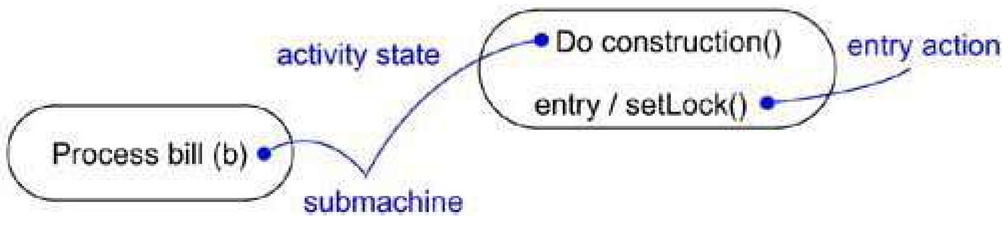
Note

In the real world, of course, every computation takes some amount of time and space. Especially for hard real time systems, it's important that you model these properties.

State machines, the parts of a state (including entry and exit actions) and submachines are discussed in Chapter 21.

In contrast, activity states can be further decomposed, their activity being represented by other activity diagrams. Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete. You can think of an action state as a special case of an activity state. An action state is an activity state that cannot be further decomposed. Similarly, you can think of an activity state as a composite, whose flow of control is made up of other activity states and action states. Zoom into the details of an activity state, and you'll find another activity diagram. As Figure 19-3 shows, there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit actions (actions which are involved on entering and leaving the state, respectively) and submachine specifications.

Figure 19-3 Activity States



Note

Action states and activity states are just special kinds of states in a state machine. When you enter an action or activity state, you simply perform the action or the activity; when you finish, control passes to the next action or activity. Activity states are somewhat of a shorthand, therefore. An activity state is semantically equivalent to expanding its activity graph (and transitively so) in place until you only see actions. Nonetheless, activity states are important because they help you break complex computations into parts, in the same manner as you use operations to group and reuse expressions.

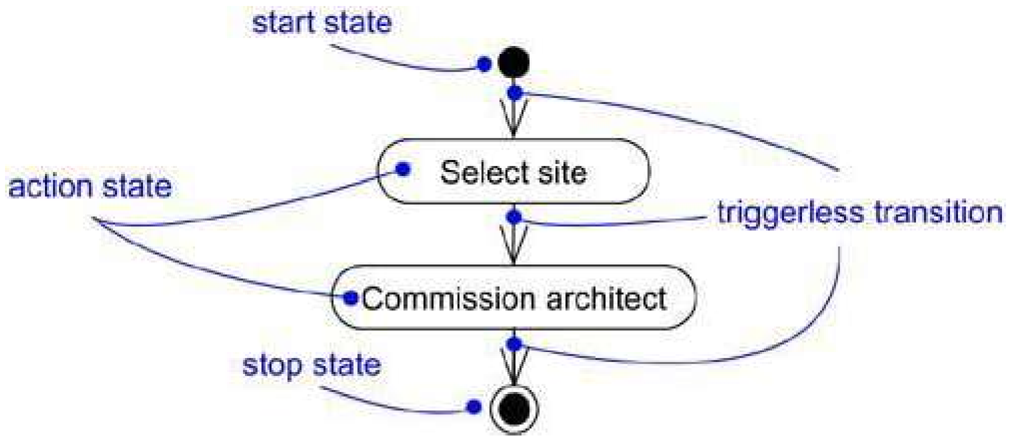
Transitions

Transitions are discussed in Chapter 21.

Triggerless transitions may have guard conditions, meaning that such a transition will fire only if that condition is met; guard conditions are discussed in Chapter 21.

When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state. In the UML, you represent a transition as a simple directed line, as Figure 19-4 shows.

Figure 19-4 Triggerless Transitions



Note

Semantically, these are called triggerless, or completion, transitions because control passes immediately once the work of the source state is done. Once the action of a given source state completes, you execute that state's exit action (if any). Next, and without delay, control follows the transition and passes on to the next action or activity state. You execute that state's entry action (if any), then you perform the action or activity of the target state, again following the next transition once that state's work is done. This flow of control continues indefinitely (in the case of an infinite activity) or until you encounter a stop state.

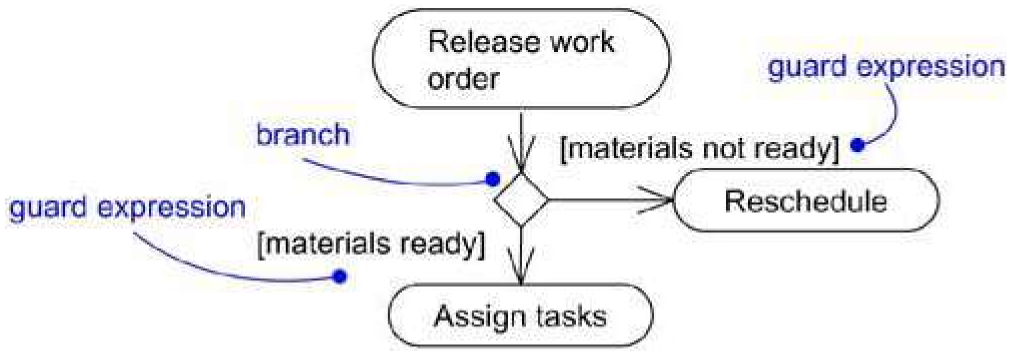
Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in which case it will have a beginning but no end). Therefore, as the figure shows, you may specify this initial state (a solid ball) and stop state (a solid ball inside a circle).

Branching

Branches are a notational convenience, semantically equivalent to multiple transitions with guards, as discussed in Chapter 21.

Simple, sequential transitions are common, but they aren't the only kind of path you'll need to model a flow of control. As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression. As Figure 19-5 shows, you represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones. On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).

Figure 19-5 Branching



As a convenience, you can use the keyword `else` to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.

Branching and iteration are possible in interaction diagrams, as discussed in Chapter 18.

You can achieve the effect of iteration by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished.

Note

The UML does not prescribe the language of these expressions. Abstractly, you might just use structured text; more concretely, you might use the syntax and semantics of a specific programming language.

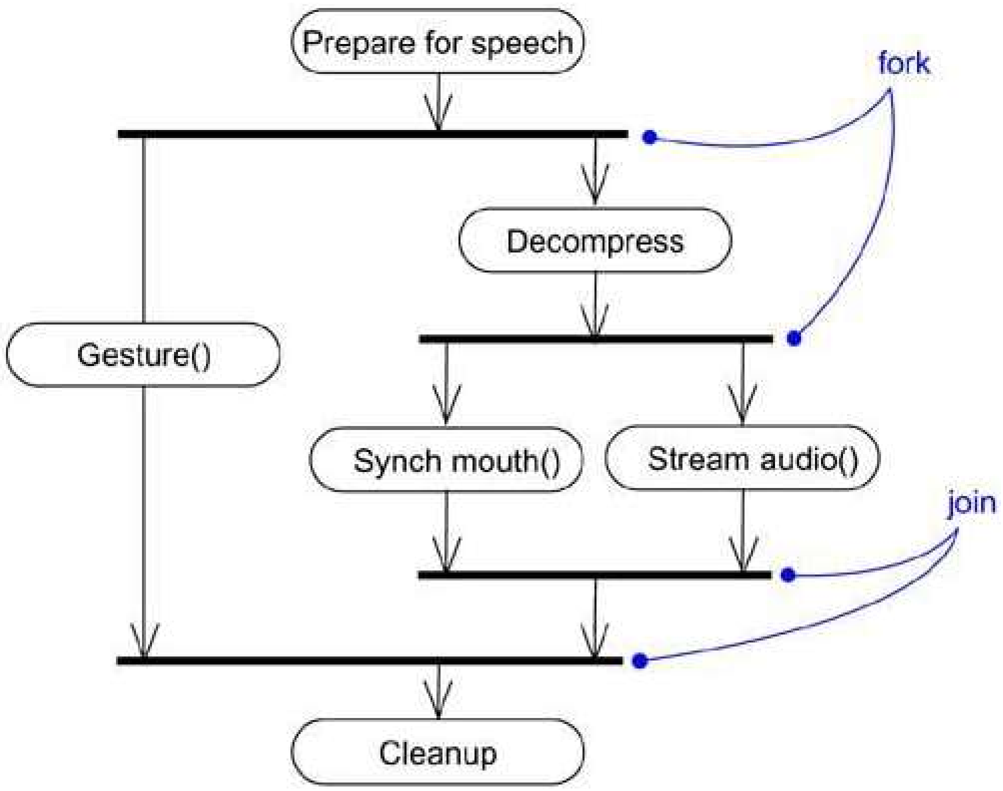
Forking and Joining

Each concurrent flow of control lives in the context of an independent active object, which is typically modeled as either a process or a thread, as discussed in Chapter 22*; nodes are discussed in* Chapter 26.

Simple and branching sequential transitions are the most common paths you'll find in activity diagrams. However—especially when you are modeling workflows of business processes—you might encounter flows that are concurrent. In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures. As Figure 19-6 shows, a fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continues in parallel. Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent (in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.

Figure 19-6 Forking and Joining



Active objects are discussed in Chapter 22*; signals are discussed in* Chapter 20.

As the figure also shows, a join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continues in parallel. At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

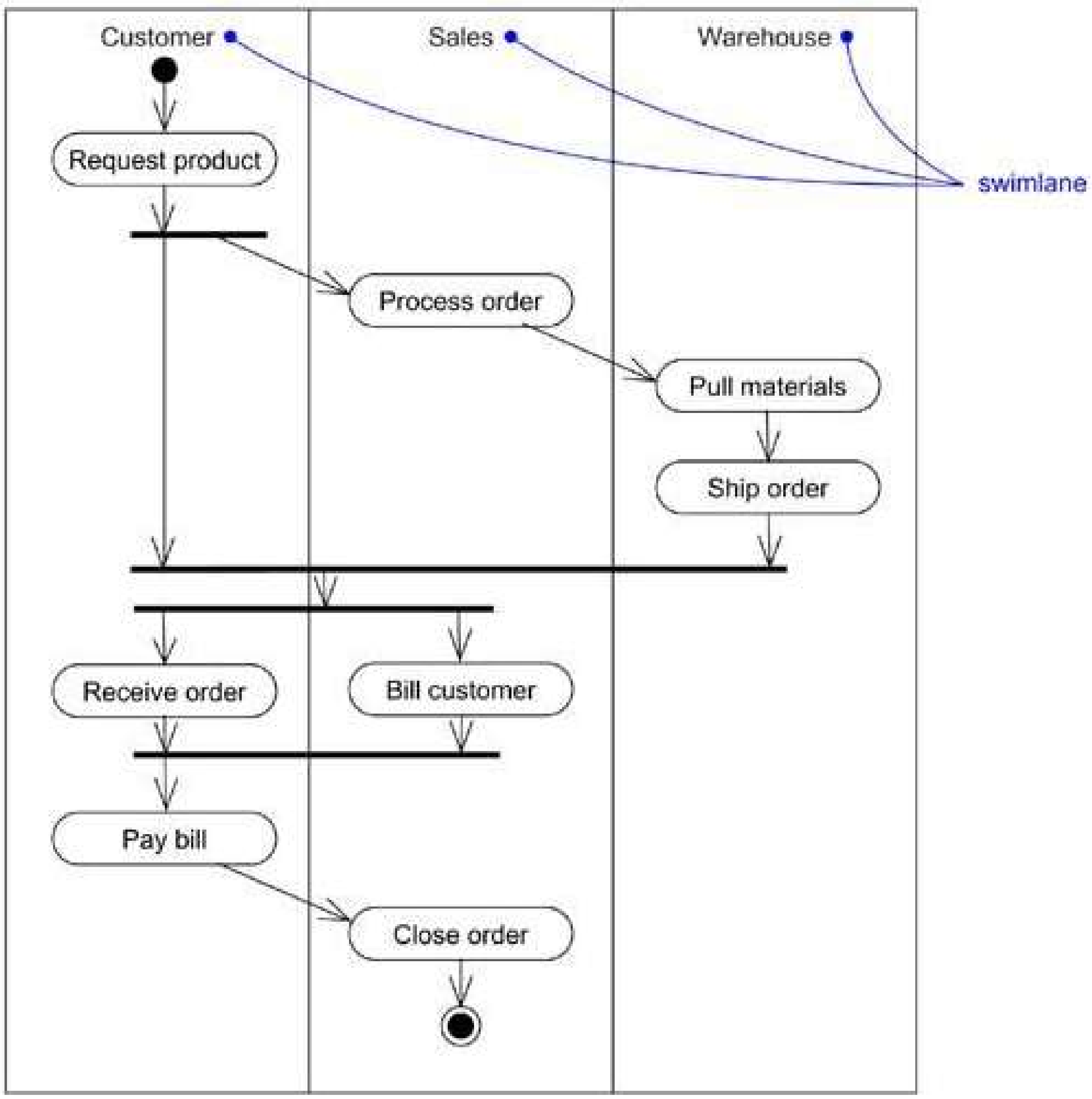
Note

Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join. Also, activities that are in parallel flows of control may communicate with one another by sending signals. This style of communicating sequential processes is called a coroutine. Most of the time, you model this style of communication using active objects. You can also model the sending of and response to these signals in the submachines associated with each communicating activity state. For example, suppose the activity `Stream audio` needed to tell the activity `Synch mouth` when important pauses and intonations occurred. In the state machine for `Stream audio`, you'd see signals sent to the state machine for `Synch mouth.` Similarly, in the state machine for `Synch mouth`, you'd see transitions triggered by these same signals, to which the `Synch mouth` state machine would respond.

Swimlanes

You'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities. In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line, as shown in Figure 19-7. A swimlane specifies a locus of activities.

Figure 19-7 Swimlanes



A swimlane is a kind of package; packages are discussed in Chapter 12; classes are discussed in Chapters 4 and 9; processes and threads are discussed inChapter 22.

Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

Note

There's a loose connection between swimlanes and concurrent flows of control. Conceptually, the activities of each swimlane are generally—but not always—considered separate from the activities of neighboring swimlanes. That makes sense because, in the real world, the business organizations that generally map to these swimlanes are independent and concurrent.

Object Flow

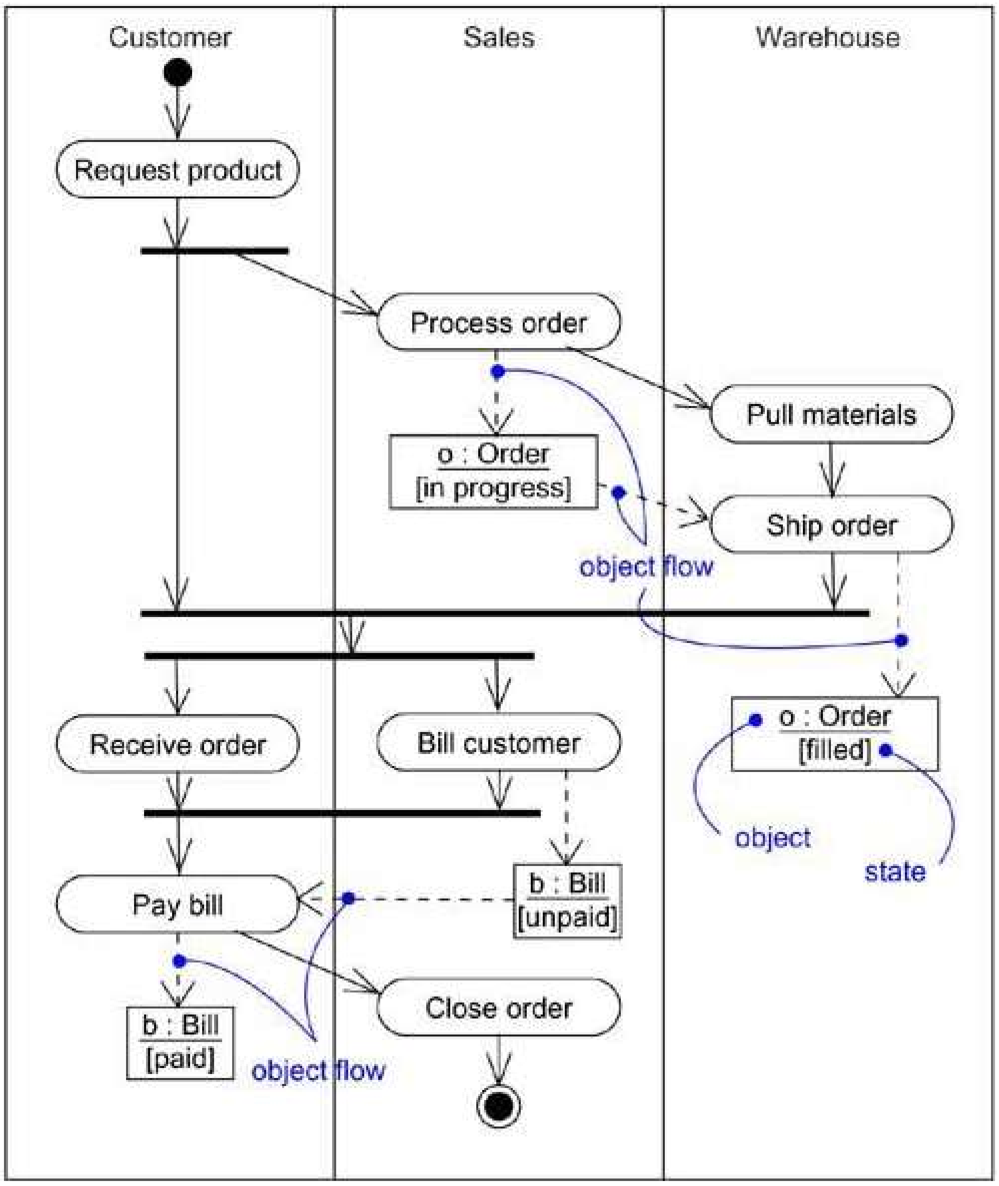
Objects are discussed in Chapter 13 *; modeling the vocabulary of a system is discussed in* Chapter 4.

Objects may be involved in the flow of control associated with an activity diagram. For example, in the workflow of processing an order as in the previous figure, the vocabulary of your problem space will also include such classes as `Order` and `Bill.` Instances of these two classes will be produced by certain activities (`Process order` will create an `Order` object, for example); other activities may modify these objects (for example, `Ship order` will change the state of the `Order` object to `filled`).

Dependency relationships are discussed in Chapters 5 *and* 10.

As Figure 19-8 shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.

Figure 19-8 Object Flow



The values and state of an object are discussed in Chapter 13*; attributes are discussed in* Chapters 4 *and* 9.

In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a compartment below the object's name.

Common Uses

The five views of an architecture are discussed in Chapter 2 *; classes are discussed in* Chapters 4 *and* 9 *; active classes are discussed in* Chapter 22 *; interfaces are discussed in* Chapter 11 *; components are discussed in* Chapter 25 *; nodes are discussed in* Chapter 26 *; systems and subsystems are discussed in* Chapter 31*; operations are discussed in* Chapters 4 *and* 9 *; use cases and actors are discussed in* Chapter 16.

You use activity diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use an activity diagram to model some dynamic aspect of a system, you can do so in the context of virtually any modeling element. Typically, however, you'll use activity diagrams in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach activity diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you'll typically use activity diagrams in two ways.

1. To model a workflow

Here you'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity diagrams, modeling object flow is particularly important.

2. To model an operation

Here you'll use activity diagrams as flowcharts, to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the operation and its local objects.

Common Modeling Techniques

Modeling a Workflow

Modeling the context of a system is discussed in Chapter 17.

No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system. Especially for mission critical, enterprise software, you'll find automated systems working in the context of higher-level business processes. These business processes are kinds of workflows because they represent the flow of work and objects through the business. For example, in a retail business, you'll have some automated systems (for example, point-of-sale systems that interact with marketing and warehouse systems), as well as human systems (the people that work at each retail outlet, as well as the telesales, marketing, buying, and shipping departments). You can model the business processes for the way these various automated and human systems collaborate by using activity diagrams.

Modeling the vocabulary of a system is discussed in Chapter 4 *; preconditions and postconditions are discussed in* Chapter 9.

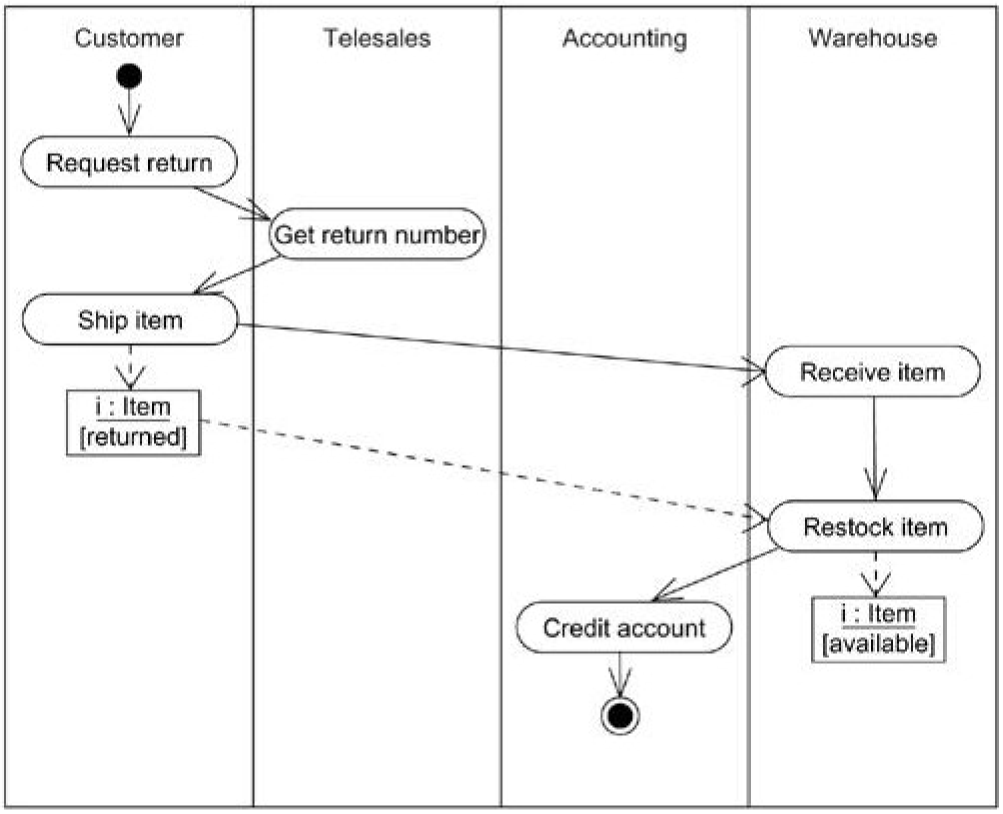
To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.

- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure 19-9 shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order. Work starts with the `Customer` action `Request return` and then flows through `Telesales` (`Get return number`), back to the `Customer` (`Ship item`), then to the `Warehouse` (`Receive item` then `Restock item`), finally ending in `Accounting` (`Credit account`). As the diagram indicates, one significant object (`i`, an instance of `Item`) also flows the process, changing from the `returned` to the `available` state.

Figure 19-9 Modeling a Workflow



Note

Workflows are most often business processes, but not always. For example, you can also use activity diagrams to specify software development processes, such as your process for configuration management. Furthermore, you can use activity diagrams to model nonsoftware systems, such as the flow of patients through a healthcare system.

In this example, there are no branches, forks, or joins. You'll encounter these features in more complex workflows.

Modeling an Operation

Classes and operations are discussed in Chapters 4 *and* 9 *; interfaces are discussed in* Chapter 11.

An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior. You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations. The most common element to which you'll attach an activity diagram is an operation.

Components are discussed in Chapter 25*; nodes are discussed in* Chapter 26 *; use cases are discussed in* Chapter 16 *; collaborations are discussed in* Chapter 27 *; preconditions, postconditions, and invariants are discussed in* Chapter 9.

Used in this manner, an activity diagram is simply a flowchart of an operation's actions. An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model. For example, any other operation or signal that an action state references can be type checked against the class of the target object.

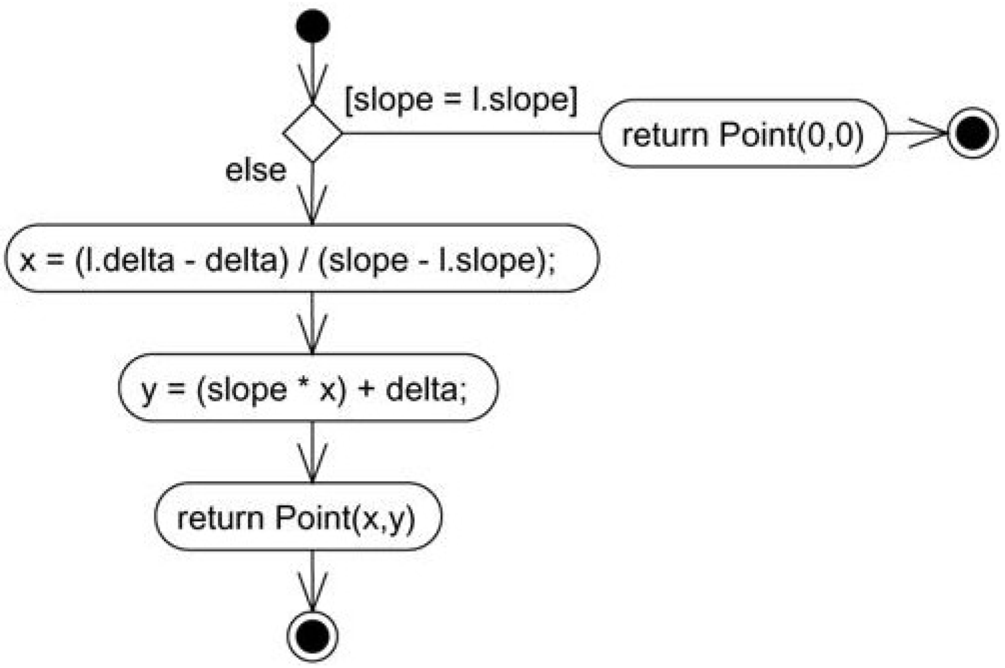
To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Active classes are discussed in Chapter 22.

For example, in the context of the class `Line`, Figure 19-10 shows an activity diagram that specifies the algorithm of the operation `intersection`, whose signature includes one parameter (`l`, an `in` parameter of the class `Line`) and one return value (of the class `Point`). The class `Line` has two attributes of interest: `slope` (which holds the slope of the line) and `delta` (which holds the offset of the line relative to the origin).

Figure 19-10 Modeling an Operation



If an operation involves the interaction of a society of objects, you can also model the realization of that operation using collaborations, as discussed in Chapter 27.

The algorithm of this operation is simple, as shown in the following activity diagram. First, there's a guard that tests whether the `slope` of the current line is the same as the `slope` of parameter `l`. If so, the lines do not intersect, and a `Point` at `(0,0)` is returned. Otherwise, the operation first calculates an `x` value for the point of intersection, then a `y` value; `x` and `y` are both objects local to the operation. Finally, a `Point` at `(x,y)` is returned.

Note

Using activity diagrams to flowchart an operation lies on the edge of making the UML a visual programming language. You can<canII> flowchart every operation, but pragmatically, you won't want to. Writing the body of an operation in a specific programming language is usually more direct. You will want to use activity diagrams to model an operation when the behavior of that operation is complex and therefore difficult to understand just by staring at code. Looking at a flowchart will reveal things about the algorithm you could not have seen just by looking at the code.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation. For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation `intersection`.

```
Point Line::intersection (l : Line) {
  if (slope == l.slope) return Point(0,0);
  int x = (l.delta - delta) / (slope - l.slope);
  int y = (slope * x) + delta;
```

```
        return Point(x, y);
        }
```

There's a bit of cleverness here, involving the declaration of the two local variables. A less-sophisticated tool might have first declared the two variables and then set their values.

Reverse engineering (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation. In particular, the previous diagram could have been generated from the implementation of the class `Line.`

More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the action states in the diagram as they were dispatched in a running system. Even better, with this tool also under the control of a debugger, you could control the speed of execution, possibly setting breakpoints to stop the action at interesting points in time to examine the attribute values of individual objects.

Hints and Tips

When you create activity diagrams in the UML, remember that activity diagrams are just projections on the same model of a system's dynamic aspects. No single activity diagram can capture everything about a system's dynamic aspects. Rather, you'll want to use many activity diagrams to model the dynamics of a workflow or an operation.

A well-structured activity diagram

- Is focused on communicating one aspect of a system's dynamics.
- Contains only those elements that are essential to understanding that aspect.
- Provides detail consistent with its level of abstraction; you expose only those adornments that are essential to understanding.
- Is not so minimalist that it misinforms the reader about important semantics.

When you draw an activity diagram,

- Give it a name that communicates its purpose.
- Start with modeling the primary flow. Address branching, concurrency, and object flow as secondary considerations, possibly in separate diagrams.
- Lay out its elements to minimize lines that cross.
- Use notes and color as visual cues to draw attention to important features of your diagram.

Part V: Advanced Behavioral Modeling