CHAPTER 3

Diagramming Features as Processes

This chapter is about activity diagrams. While my emphasis isn't process, a next step after capturing use cases is to begin describing how the features represented by your use cases will play out. Activity diagrams help you and users to describe visually the sequence of actions that leads you through the completion of a task.

The goal is to converge on code continuously by starting with an understanding of the problem space in general and capturing the problems we will solve—use cases—by describing how those features work and ultimately implementing the solution. Activity diagrams are a useful analysis tool and can be used for process reengineering, i.e., redesigning process. In this way, activity diagrams are a progressive

bridge leading from analysis to design and ultimately to implementation. In this chapter you will learn about

- The symbols used to create activity diagrams
- How to create activity diagrams by describing use cases and scenarios as a series of actions
- Modeling simultaneous behaviors
- Refining physical activities with activity diagrams
- Figuring out when to stop creating activity diagrams

Elaborating on Features as Processes

Few ideas are completely new. Existing concepts are refined and evolve and mature, carrying along some of the old and some of the new. The same is true for analysis and design concepts.

Structured analysis and design emphasized the flowchart. An activity diagram in the Unified Modeling Language (UML) is pretty close to a flowchart. The symbols are similar but not the same. The utility is similar, but there is a difference. Activity diagrams, unlike flowcharts, can model parallel behavior.

Activity diagrams are good analysis diagrams for developers, users, testers, and managers because they use simple symbols, plain text, and a style that is similar to the familiar flowchart. Activity diagrams are good at helping you to capture, visualize, and describe an ordered set of actions from a beginning to an end. Activity diagrams are created as a finite set of serial actions or a combination of serial and parallel actions.

A Journey toward Code

A basic principle of objected-oriented analysis and design is that we want to start from high-level problem-space ideas and concepts and move toward a low-level solution space. The high-level problem space is also referred to as the *problem domain.* The low-level solution space is referred to as the *solution domain.* The UML is a language for capturing and describing our understanding as we move from documenting a problem to coding a solution.

Based on the idea of moving our understanding from concept to design, use cases are a good way to capture the things that describe our problem. For example, we want to match employers to potential employees by providing a job listing board. A use case that supports this is to manage listings. A next step in an abstract sense is to describe how we would go about managing a listing. At this juncture it is still too early

to begin talking about databases and programming languages. Instead, we want to talk about the activities that describe our problem, and these activities consist of actions.

Note *At an ideological level, analysis and design are processes whereby we decompose a problem into smaller discrete problems so that we can compose small solutions for each discrete problem and ultimately orchestrate the small solutions into a coherent whole. The UML is a language for decomposing a problem and recomposing it as the description of a solution. A language such as Visual Basic.NET is useful for implementing the solution description, and process is how we go about it.*

Understanding Activity Diagram Uses

Activity diagrams are not really about methods or classes. It is still too early for that. The reason that it is too early is because technical things such as polymorphism, inheritance, methods, and attributes generally are meaningless concepts to users and sometimes managers.

Activity diagrams are a means by which we can capture the understanding of people we call *domain experts*. For example, if you are building a jail-management system, then a domain expert might be a corrections officer. A corrections officer probably won't understand the difference between a namespace, class, and interface, but as a designer, you may not understand the significance of a purchase of 50 toothbrushes by an inmate. An activity diagram can help.

A true story—and why consulting can be interesting—is behind the toothbrush metaphor. While working for a large county jail system in Oregon, I had to write a pilot application to demonstrate ASP.NET in its early days. The pilot ultimately would be part of an inmate account-management system for the jail. The basic idea was that prisoners can't be in possession of cash, but they can have money on account to purchase personal items and snacks. The county managed the accounts. Some of the rules included limits on the number of candy bars, say, a diabetic might purchase, as well as a limit on the number of toothbrushes that could be purchased. Not being a corrections officer, it seemed weird to me that anyone would buy more than one toothbrush and weirder still why anyone would care. The problem is that when scraped to a sharp point or split with a piece of a safety razorblade wedged into the split and held in place by a rubber band, a toothbrush can become a formidable weapon. (In reality, I knew this because I either learned it as a military policeman or saw it on an episode of "Oz" on HBO.)

In practice, this story is illustrative of the fact that those on the ground—the domain experts—will know details that you will never think of. Activity diagrams are good for capturing these details in a general sense and in a way that the domain experts can examine, clarify, and improve on.

Working backward from my prisoner account-management story, we might have a use case "Make Purchase" and a scenario in that use case that we need to ensure that the purchase does not violate a safety rule. We can capture this in an activity diagram plainly enough that a corrections officer can tell us if we understand the problem and have decomposed it sufficiently. Figure 3-1 shows an activity diagram for this scenario.

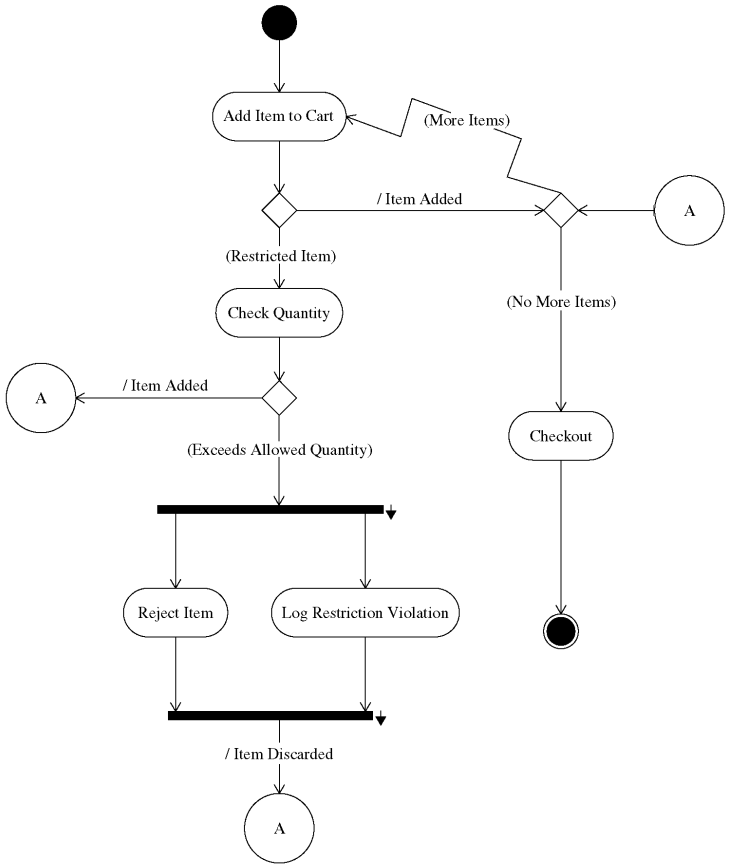

Figure 3-1 An activity diagram that illustrates restrictions on the kind and number of items that can be purchased in prison.

For now, don't worry about what the shapes mean. Simply note the simple text and the flow suggested by the arrows. The general idea is that at a glance—perhaps with a minimum of explanation—this diagram should make sense to users and developers alike. The next section will begin exploring what all these elements and more mean.

Using Activity Diagram Symbols

Activity diagrams can be simple flowcharts that have a finite beginning and ending point or more complex diagrams that model parallel behavior and multiple subflows and define multiple terminuses. I find that diagramming simple activities is an excellent way to get started and that adding too many alternate scenarios in a single diagram makes it both hard to manage and print the diagram and difficult to understand.

Making your activity diagrams comprehensible may be more important than making the diagram comprehensive or all-encompassing. Another mistake is to create activity diagrams for every use case and scenario. Creating diagrams is time-consuming, and a good way to focus your time is by diagramming those aspects that are more critical to solving your problem.

Consider a couple of examples. Programs that store data commonly do so in relational databases. This behavior is called *create, read, update, and delete (CRUD) behavior.* Reading and writing data from a database are so well understood that I wouldn't diagram this behavior as a separate activity. (In fact, the notion of a database really shouldn't show up in an activity diagram.) The entire read-write behavior might be captured at some point in an activity as an action called *fetch and store* or *read and write.* On the other hand—borrowing from Chapter 2—if we are going to expire a customer's job listing and want to give that customer an opportunity to extend the job listing, then this is less common than CRUD behavior, and I would create an activity diagram to explore the sequence of actions. By diagramming the "Expire Listing" activity, I could get the sequence of actions just right, and it might be the catalyst for improving on the quality of service. For example, we might come up with the renew by e-mail feature we discussed in Chapter 2.

If you have created some flowcharts with a tool such as Visio in the past, then activity diagrams will seem pretty straightforward, but keep in mind that activity diagrams can be used to model richer behavior than plain old flowcharts. In order to create activity diagrams, you will need to learn abut the symbols and rules that apply.

Tip *You can think of the symbols and rules of any UML diagram as the* visual grammar *for the language.*

Initial Node

Every activity diagram has one *initial node* symbol. This is a solid circle (see the top of Figure 3-1). You can provide a name and some documentation for the initial node, but generally I do not.

The initial node can have one transition line exiting the node. The transition line is called a *control flow* and is represented by a directed arrow with the arrow pointing away from the initial node. For clarity, just the initial node and a control flow are depicted in Figure 3-2. You can place the initial node anywhere on the diagram you'd like and add the control flow anywhere on the initial node you'd like. Living in the western hemisphere, I have a bias toward upper-left starting points and lower-right ending points.

Control Flow

As mentioned previously, a control flow is a directed arrow. A control flow is also referred to as just a *flow* or an *edge.* The control flow begins at the symbol losing focus and points to and is connected to the thing gaining focus. For example, a control flow might originate at an initial node and terminate at an action, as shown in Figure 3-3.

A common way to adorn a control flow is to add a *guard condition.* A guard condition acts as a sentinel that requires a test be passed before flow continues. In code, commonly this would be implemented as an if-conditional test.

Using Guard Conditions

Without diverting our attention away from guard conditions too much, an *action*—which we will talk about more in the section entitled, "Action"—is something that happens in the flow. An action, like the initial node, is another kind of node. Activity diagrams are wholly composed of various kinds of nodes and flows (or edges).

Figure 3-2 The solid circle is called an *initial node*—or activity diagram starting point—and the directed arrow is called a *control flow*.

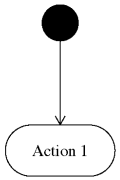

Figure 3-3 An initial node, control flow, and an action.

A guard condition is shown as text in between the left and right square brackets, and you can think of a guard condition as a gatekeeper to the next node (Figure 3-4).

If you have ever served in any kind of militia, then you are familiar with the notion of a password or pass phrase:

> **Guard:** "The sparrow is a harbinger."
>
> **Footsolider:** "Of death, which is the only certainty besides taxes."
>
> **Guard:** "You may pass."

Well, when I was in the army, the pass phrases were never clever, but the idea is the same. The guard represents a test that must be passed to continue. Oddly enough, programmatic tests can be pretty esoteric, but the text you write in your guard conditions will serve your constituency better if they are simple. Figure 3-5 is a practical example of an initial node, an action, and a flow with a guard condition.

In the figure, the initial node transitions to the first action, "Find Customer." The guard condition is that my availability date is known. It doesn't do any good to pile up customers when I have no available time left.

The diagram in Figure 3-5 illustrates how an activity diagram is sort of agnostic when it comes to implementation. The partial activity in Figure 3-5 could be talking about a physical process such as searching the Motown-jobs.com Web site and

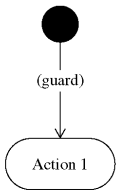

Figure 3-4 An initial node, flow with guard, and a generic action.

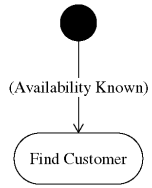

Figure 3-5 Part of an activity diagram for finding customers.

calling past customers or a software process that automatically scans the Motown-jobs.com Web site through a Web service and e-mails past customers, notifying them of my availability. You will see more instances of guard conditions throughout the examples in this chapter.

Different Ways of Showing Flows

The most common way to diagram a flow is to use a single control flow symbol connecting two nodes, but this isn't the only way. If your diagram is very complex with a lot of overlapping edges, then you can use a connector node (Figure 3-6). An edge can transition from an action to an object to an action (Figure 3-7) and between two pins (Figure 3-8).

Using Connector Nodes

You don't have to use connectors, but if your diagrams become very large or complex, then you will find that your flows begin to overlap or that your activity spans multiple pages. The connector node is a good way to simplify overlapping flows or flows that span multiple pages.

Tip *The version of Visio that I used to create Figure 3-6 does not support the connector node. To create this effect, I had to use the Ellipse tool. The result is that the diagram is visually correct, but Visio will report an error. As is true with many tools, tradeoffs have to be made.*

Figure 3-6 A connector node can be used to simplify busy-looking activity diagrams.

Figure 3-7 Inserting a customer object between two actions related to customers.

To use a connector node, draw a flow exiting a node and transitioning to a connector. Where the connection is made to the next node, draw a connector with a flow exiting the connector and transitioning to the next node in the diagram.

Connector nodes come in pairs. Make sure that connector pairs have the same name; naming connectors will help you to match connection points when you have multiple connector pairs in a single diagram.

Using Objects in Activity Diagrams

Earlier I said that diagramming activities occurs too early in analysis to figure out what the objects are; however, the UML supports adding objects to activity diagrams. After you have had a chance to let users provide you with some feedback and your understanding of the problem space has grown, it may be helpful to add objects to your activity diagrams. The key here is to avoid adding technically complex concepts too early. If you get bogged down in discussions about what an object is or whether the object is named correctly or not, then remove the object. On the other hand, if the object is very obvious—as is depicted in Figure 3-7—and it aids everyone's understanding, then add it.

It is valuable to keep in mind who your constituency is for each kind of diagram. Generally, I think of activity diagrams as analysis tools that end users will read to help you understand how they do their job; explaining object-oriented concepts typically seems to be a distraction, so I leave objects out of activity diagrams.

Using Pins

Pins in the UML are analogous to parameters in implementation. The name or value of a pin leaving one action should be thought of as an input parameter to the next action. Figures 3-7 and 3-8 convey the same information—that there is a customer involved in this flow. Pins, like objects, may be too detailed for everyday use

Figure 3-8 An advanced technique includes connecting two pins on action nodes with a control flow.

and may result in tangential, confusing discussions when working out flows with customers. However, if you are explaining the activities to designers and programmers, then it may be helpful to show objects.

In Figures 3-7 and 3-8 the names of actions—"Find Customer" and "Contact Customer"—clearly suggest that a customer is involved. Leaving out the object and pins—see Figure 3-9—still pretty clearly suggests the participation of a customer without risking lengthy, tangential explanations.

Actions

Action nodes are the things that you do or that happen in an activity diagram, and an edge represents the path you follow to leapfrog from action to action. Action nodes are slightly more rectangular in shape than use case shapes. Two of the most important aspects of actions are the order in which they occur and the name you give them. The name should be short and to the point. Using noun and verb pairs in action names can help you to find classes and methods, but action names are not intended solely for this purpose, and again, it is pretty early in analysis and design to get hung up on implementation details such as classes and methods.

Actions are permitted to have one or more incoming flows and only one outgoing flow. If there is more then one incoming flow, then the action will not transition until all incoming flows have reached that action. Actions can split into alternate paths using the *decision node*—refer to the section entitled, "Decision and Merge Nodes"—or transition into parallel flows using the *fork node*—see the section entitled, "Transition Fork and Transition Merge"—but only a single flow actually should be attached as an outgoing flow for an action.

A good rule of thumb for creating activity diagrams is to describe how a use case begins, progresses, and ends with all the actions that must be completed along the way. Decision and merge nodes and forks and joins are a means of modeling parallel behavior or alternations with the activity itself. If alternate flows are very complex, then you can use the subactivity diagram to compartmentalize the subactivity.

Actions also can use preconditions and postconditions to indicate the necessary conditions before and after an action occurs. Let's chunk these aspects—names,

Figure 3-9 This diagram is simpler than the diagrams showing an object or using pins but still suggests the participation of a customer.

subactivities, and conditions—into subsections to examine how we annotate each aspect of an action.

Naming Actions

I prefer actions to have sufficient detail—a noun and a verb—to describe what happens and what or who is involved, e.g., "Find Customer," "E-mail Customer," "Store Job Listing," "Cancel Listing," and "Delete Résumé." Without a tremendous amount of additional text, these names tell me what the action does and what is acted on. This is important because an essential concept in the UML is that a lot of information is conveyed visually as opposed to with a lot of text.

Ultimately, nouns and verbs will help you to find classes and methods, but it is a good idea to defer thinking about implementation details for a while yet. We simply want to understand how we go about performing an activity but not how we implement that activity.

For example, in Chapter 2 we defined a use case "Manage a Job Listing." This is a use case that arguably consists of several activities, including "Post a Job Listing." Posting a job listing is a scenario in the "Manage a Job Listing" use case, but "Post a Job Listing" is not a single action. There are arguably several actions that would have to be completed to capture the entire activity. Here is a written example that describes posting a job listing, followed by a short activity diagram (Figure 3-10) modeling the same thing:

- Provide job description
- Log-in
- Provide payment information
- Process payment
- Store job description
- Provide confirmation

Once we have an initial diagram—shown in Figure 3-10—we have a good basis for holding a discussion about the activity. We can bring in domain experts and ask them about details of the activity diagram and evaluate this information to determine if we need to revise the diagram. For instance, we may want to check if valid payment information on file can be used or we want new payment information. Or if the user is a new user, then we may need to add a decision point that permits the user to register and then log in.

A real implicit benefit here is that a reasonable stab at an activity diagram captures the modeler's understanding and permits others to provide feedback and elaborate on the flow, adding or removing detail as necessary.

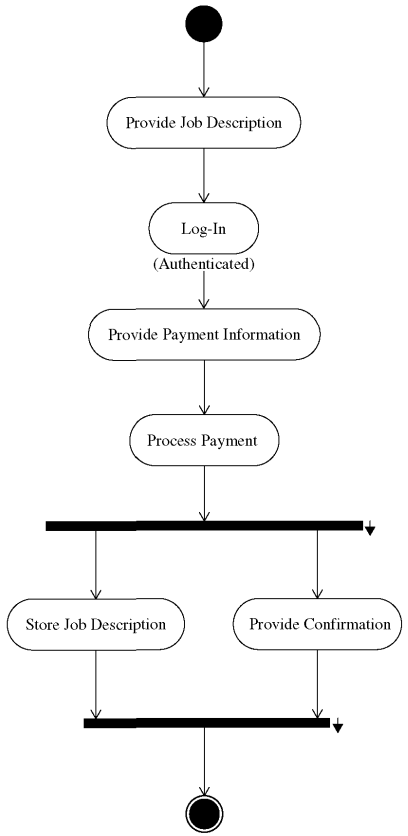

Figure 3-10 A model showing the actions required to post a job.

Adding Preconditions and Postconditions

Preconditions and postconditions can be added to a model using a note—the stereotype symbols with the word *precondition* or the word *postcondition* in between and the name of the condition. The note is attached to the action to which the condition or conditions applies. This is referred to as *design by contract* and often is implemented

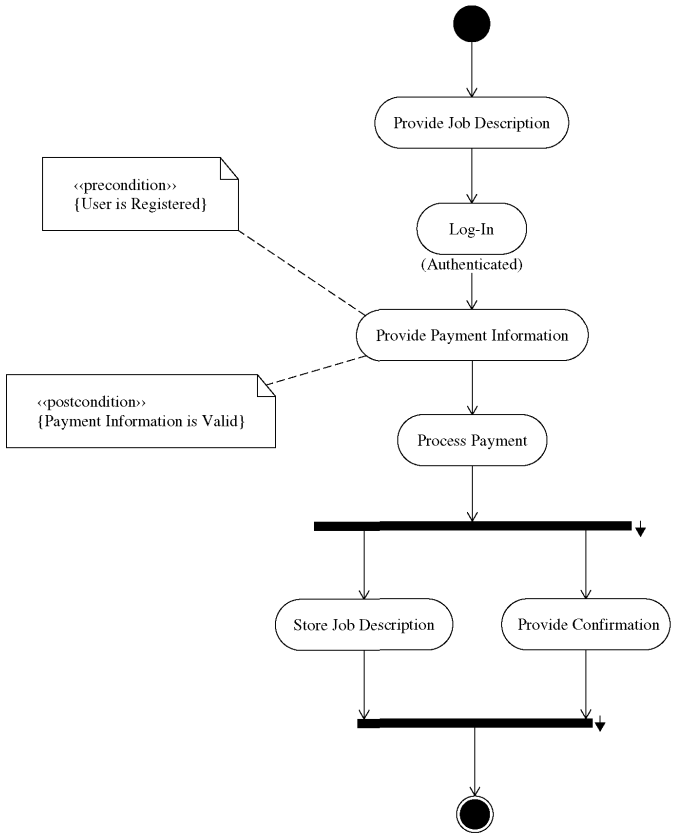

Figure 3-11 Using a precondition and postcondition constraint.

in code as an assertion combined with a conditional test. Figure 3-11 shows a precondition and postcondition applied to the "Provide Payment Information" action.

In Figure 3-11, the diagram requires the precondition that the user is registered and the postcondition that the payment information is valid. As is true with code, there is more than one way to represent this information. For example, we could use

a guard condition before and after the "Provide Payment Information" action (Figure 3-12), or we could use a decision node (see "Decision and Merge Nodes") to branch to a register action before permitting payment information to be provided, and we could have an action to validate payment information after the payment information is provided (Figure 3-13).

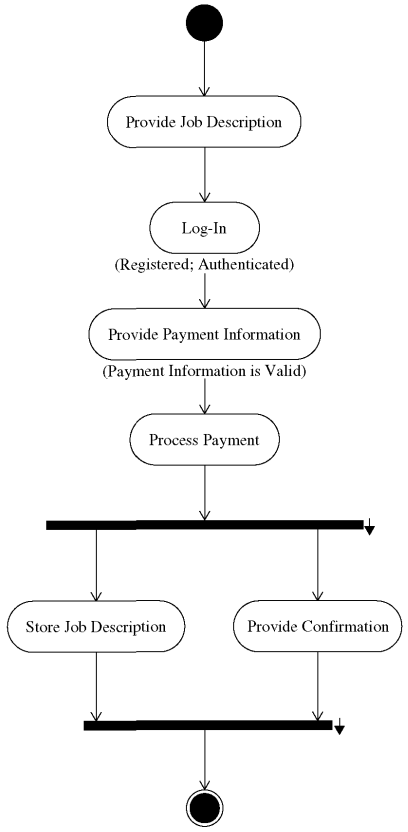

Figure 3-12 Using guards to express a precondition and a postcondition.

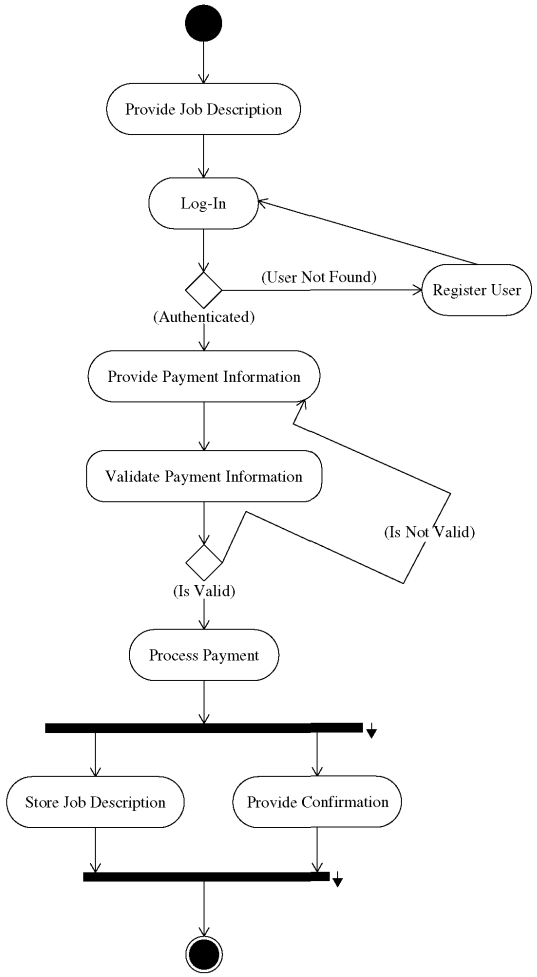

Figure 3-13 Using a decision node to indicate that users must register and provide valid payment information.

All three of these diagrams—Figures 3-11, 3-12, and 3-13—convey the same information. The real difference is stylistic. If you want the diagram to appear less busy, try using the guard condition. If the constraint style—in Figure 3-11—seems more meaningful, then use that style. If you want to explore registration and address validation, then use the decision node styles in Figure 3-13. In that figure, the decision nodes are represented by the diamond-shaped symbols.

Modeling Subactivities

Sometimes it is easy to add too much detail to a single activity diagram, making the diagram busy and confusing. For example, if we expand "Register User" in Figure 3-13 to include all the necessary actions for registering users, such as obtaining a unique user name and password and validating and storing mailing address information, then the main focus of the activity—creating and paying for a job listing—may be lost in the noise of all the additional actions and edges.

If in any instance we find the details of subactivities making a diagram too confusing, or we find that we want to reuse subactivities, then we can mark an action as a subactivity with a fork inside the action. (Visio doesn't support the subsidiary activity symbol, so I drew one from scratch in Microsoft Paint and added it to the "Register User" action in Figure 3-13.)

Tip *If you want to invent or find that an aspect of the UML isn't supported by your specific modeling tool, then consider using a stereotype or a note to document your meaning.*

Decision and Merge Nodes

Decision and merge nodes were called *decision diamonds* in flowcharts. This diamond-shaped symbol is one of the elements that makes an activity diagram reminiscent of a flowchart. Decision and merge nodes use the same symbol and convey conditional branching and merging.

When the diamond-shaped symbol is used as a *decision node*—after "Log-In" in Figure 3-13—it has one edge entering the node and multiple edges exiting the node. When used as a *merge node*, there are multiple entering edges and a single exiting edge. A decision node takes only one exit path, and a merge node doesn't exit until all flows have arrived at the merge node.

The guard conditions on a decision node act like if..else logic and should be mutually exclusive, which necessarily implies that if one guard condition is met, then the other must fail. As depicted in Figure 3-13, you can stipulate both guard conditions literally or stipulate one guard condition and use an [Else] guard for the alternate condition.

A merge node marks the end of conditional behavior started by a decision node. We don't need a merge node in Figure 3-13 because we rerouted the newly registered user back to the "Log-In" action. However, if we wanted to be a little nicer, we might simply authenticate the new user automatically and proceed right to providing payment information where he or she left off. This revision is shown using a merge node in Figure 3-14. (Note that the guard conditions for the decision node following the "Log-In" action were modified to show the use of the [Else] guard style.)

Transition Forks and Joins

A *fork* exists to depict parallel behavior, and a *join* is used to converge parallel behavior back into a single flow. Forked behavior does not specify whether or not the behavior is interleaved or occurs simultaneously; the implication is simply that the forked actions are occurring during a shared, concurrent interval. Usually forked behavior is implemented as multithreaded behavior. (Figure 3-13 presents an example of a fork after the "Process Payment" action and a join immediately before the final node.)

When multiple flows enter an action, this is implicitly a join, and the meaning is that the outgoing flow occurs *only* when all incoming flows have reached the action. Your diagrams will be clearer if you use forks and joins explicitly where you mean to show parallel behavior.

In Figure 3-13 we mean that we can store a job description and provide the user with a confirmation simultaneously or concurrently but that both these things must occur before the activity is considered complete.

Partitioning Responsibility with Swimlanes

Sometimes you want to show who or what is responsible for part of an activity. You can do this with *swimlanes*. Modeling tools typically show swimlanes as a box with a name at the top, and you place whatever nodes and edges that belong to that thing in that swimlane. You can have as many swimlanes as it makes sense to have, but boxy swimlanes can make it hard to organize your activity diagram.

UML version 2.0 supports vertical, horizontal, and gridlike partitions, so the swimlane metaphor is no longer precise. The actual terminology is now *activity partition*, but the word *swimlane* is still employed in general conversation and used in modeling tools.

Using Swimlanes

If we want to show who or what is responsible for various actions in Figure 3-14, then we can add a swimlane (or partition) for what we believe the partitions to be.

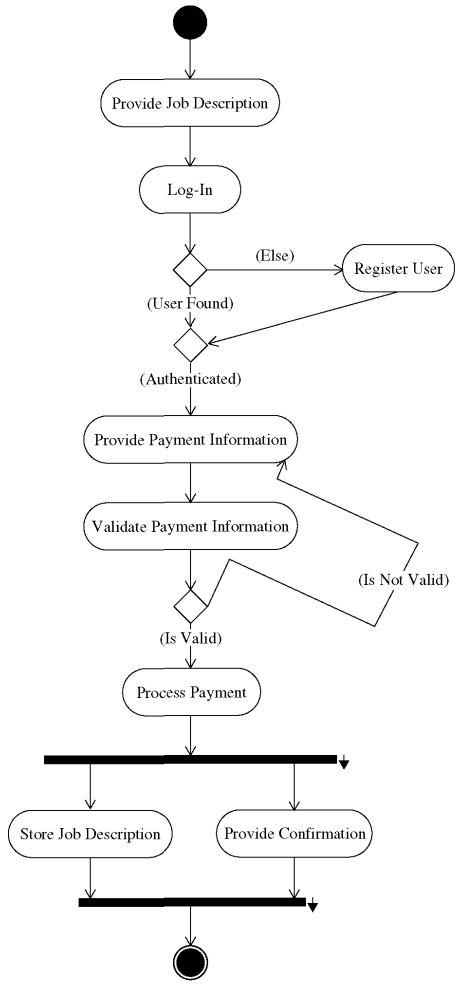

Figure 3-14 A merge node used to converge when a branch is taken after a new user registers.

In the example, we could say that posting a job is divided into two partitions, the user and the system, and add a swimlane for each partition (Figure 3-15). If we decided that payment processing represents a distinct partition, then we could add a third partition and move the process payment action into that partition (Figure 3-16).

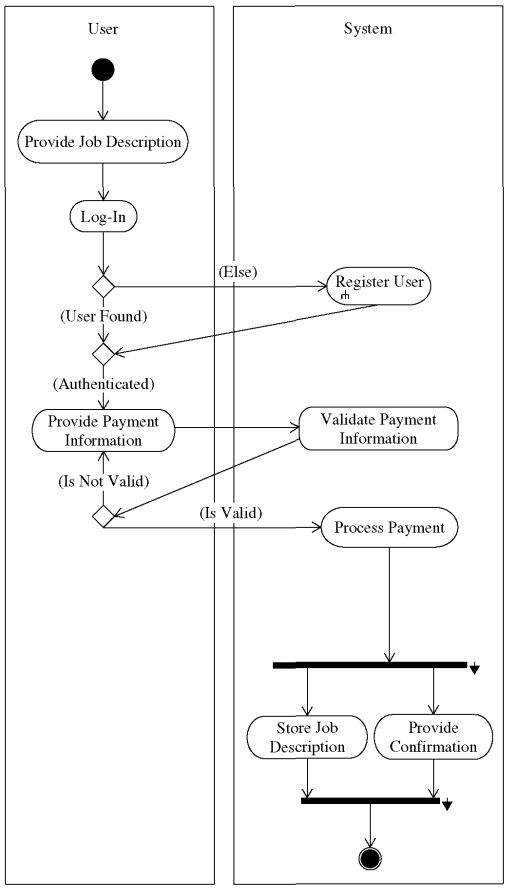

Figure 3-15 Actions are divided between a user and the system.

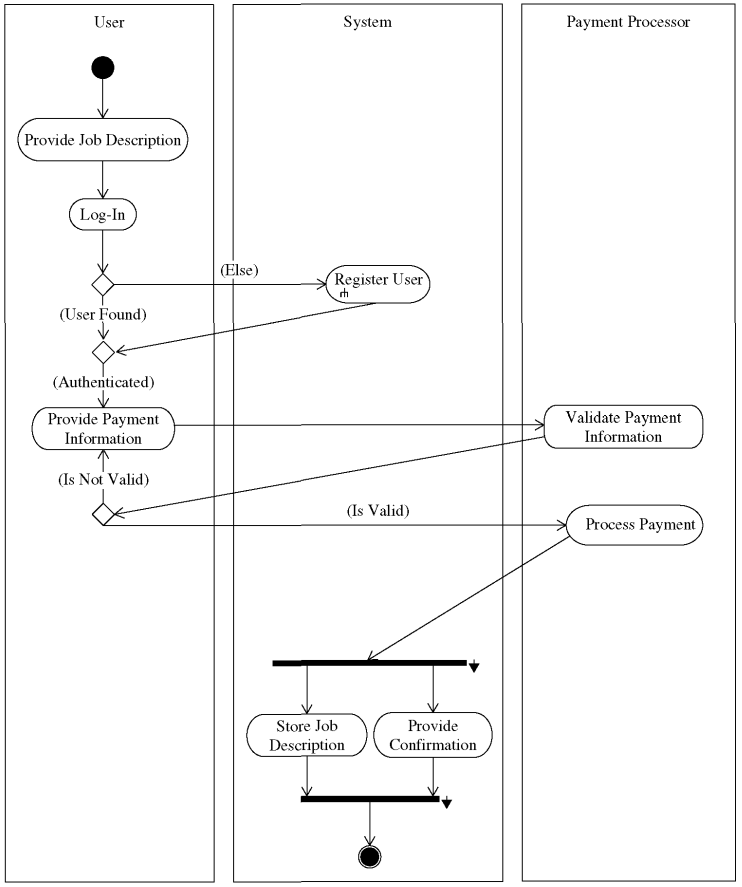

Figure 3-16 Further subdividing responsibilities by placing the "Validate Payment Information" and "Process Payment" actions in a separate partition called the "Payment Processor."

As is true with programming, you can divide your analysis and design into as many partitions as you want. There are tradeoffs for adding partitions in models just

as there are tradeoffs for adding partitions in code. Partitioning models may help you to organize, but all those partitions suggest partitioned software that will have to be orchestrated and reassembled to accomplish the goals of the system.

Modeling Actions that Spans Partitions

Sometimes an action may belong to more than one partition at a time. For example, "Register User" really doesn't belong to the user or the system. We know from earlier discussions that "Register User" is a subsidiary activity that may involve the user providing personal information and the system validating address information and storing the user information. However, the UML doesn't permit a node to span more than one partition in a single dimension. As a result, you will have to pick a partition for the node, and this also suggests what we know to be true about "Register User"—that it can be decomposed into its own activity.

Using Multidimensional Partitions

Modeling multidimensional activity partitions is a relatively new concept. Diagramming multidimensional activity partitions also doesn't seem to be wholly supported by some popular and currently available modeling tools; however, you can simulate a multidimensional partition in Visio by adding two swimlanes (activity partitions) and rotating one of them. (The result is a diagram similar to Figure 3-17.) Now that we have the mechanics for creating a multidimensional partition, you might be wondering how it is used.

An action in an activity partition matrix belongs wholly to both partitions. Suppose, for example, that as we are gearing up to sell job listings on Motown-jobs .com, we decide to use PayPal to process payments. We can say that "Process Payment" is part of both our "Payment Processor" and PayPal's payment-processing system, which is reflected in Figure 3-17.

Indicating Timed Signals

Thus far we haven't talked about when things occur. There are three types of signals that facilitate talking about time in activity diagrams. These are the *time signal*, the *send signal*, and the *accept signal*. A signal indicates that an outside event has fired, and that event initiates the activity.

The hourglass shape of the time signal is used to specify an interval of time. For example, we could use the time signal to indicate that the "Expire Listing" activity will start after the listing has been available for 30 days (Figure 3-18). The receive

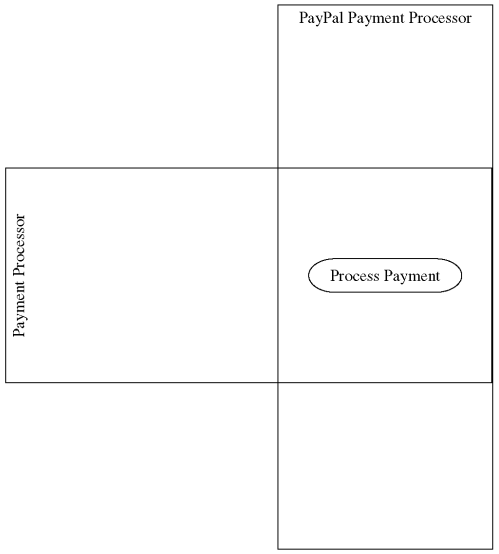

Figure 3-17 Multidimensional partitions, where an action is owned by two partitions in different dimensions at the same time.

signal symbol is a rectangle with a wedge cut into it, and the send signal symbol is a rectangle with a protruding wedge—making the receive and send signal symbols look a bit like jigsaw puzzle pieces (again shown in Figure 3-18).

NOTE *Every tool has its limitations. In Visio, for example, there is no symbol for a time signal, so I contrived one, and the send and receive signals are used as an alternative form of documenting events. Visio's implementation isn't precisely consistent with the UML; it is important not to get hung up on these little inconsistencies that you are bound to run into. Rather than spending your time drawing pictures for unsupported aspects of the UML, try using a note instead.*

The model in Figure 3-18 is understood to mean that 30 days after a listing is posted, it will be expired automatically unless a notified owner elects to extend it. Alternate signals include a user deleting a listing, which causes the listing to be

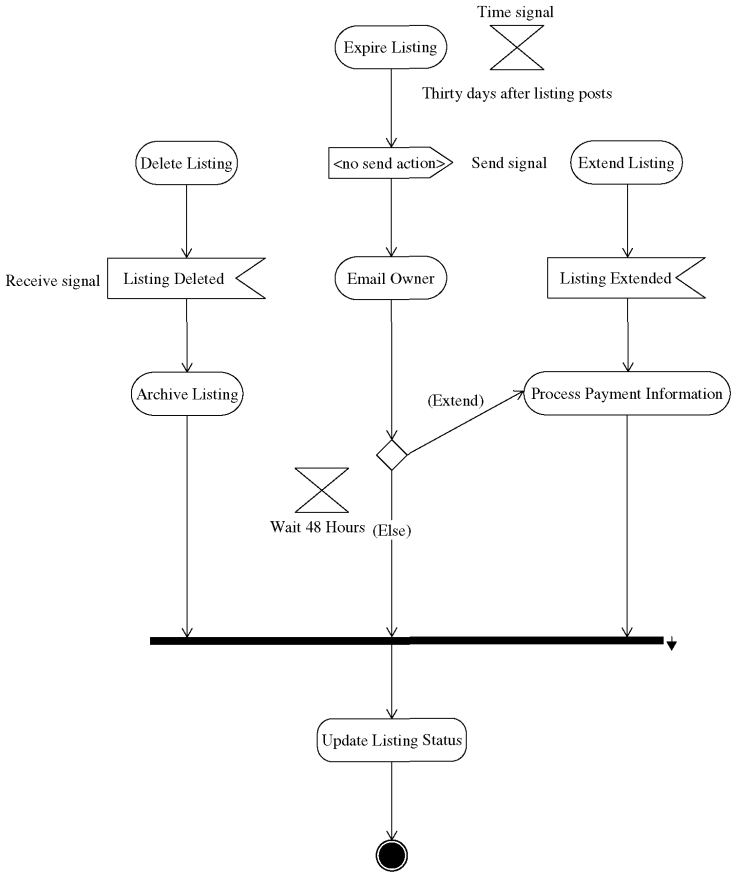

Figure 3-18 A time signal for expiring a listing, two receive signals for extending and deleting a listing, and a send signal for notifying a listing about an impending expiration.

archived before being delisted and an owner taking his or her own initiative to extend the listing prior to its expiration. If the owner extends the listing, then this signals the system to process an additional payment.

Capturing Input Parameters

Activity diagrams can have input parameters, e.g., in Figure 3-18 in every instance that we are talking about doing something with a listing. We could show a "Listing" object as the input for each action in the figure. Grabbing just a small piece of Figure 3-18, we can show the notation and symbol for indicating that the input to that action is a "Listing" object (Figure 3-19).

While input objects can be useful for developers, this is another instance where they may add confusion to the discussion of the activity in a general, analytical sense. At least during early phases of analysis, consider deferring specific reference to implementation details such as classes.

Showing Exceptions in Activity Diagrams

The UML supports modeling *exceptions.* An exception is shown as a zigzagging line (or "lightning bolt") with the name of the class of the exception adorning the zigzagging line. The exception handler can be modeled as an action node with the

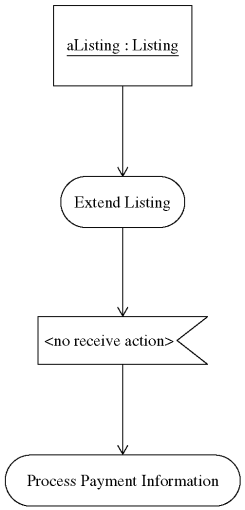

Figure 3-19 The "Listing" object is shown as an input parameter to the "Extend Listing" action and its containing activity diagram.

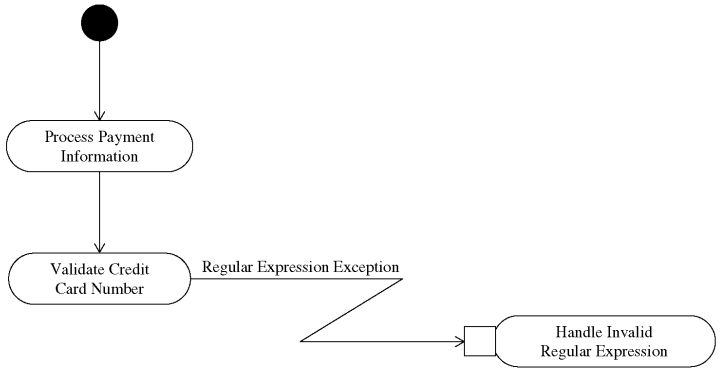

Figure 3-20 Modeling an exception in an activity diagram.

name of the action in the node and the exception flow connecting to an input pin on the exception action node (Figure 3-20).

The node containing the exception handler has no return flow. An exception handler just hangs off the action that caused the error to occur. It is important to remember that we are capturing general flow and actions; during this phase, we do not have to indicate how we are handling the exception.

Concepts such as exception, exception handler, stack unwinding, and performance may add considerably to the confusion for nontechnical users. If you can add an exception and exception action node without getting bogged down in discussions about how exception handlers are implemented or how they work, then go ahead and add them to your activity diagrams.

Terminating Activity Diagrams

When you reach the end of an activity, add an *activity final node.* If you reach the end of a flow and nothing else happens, add a *flow final node* (Figure 3-21). You can have more than one activity final node and flow final node in a single activity diagram.

The activity diagram in Figure 3-21 shows that we process all the expired listings until there are no more, and for each expired listing, we e-mail the owner, providing him or her with an opportunity to renew the listing or let it expire. Notice that when the decision node branches because there are no more expired listings, it simply

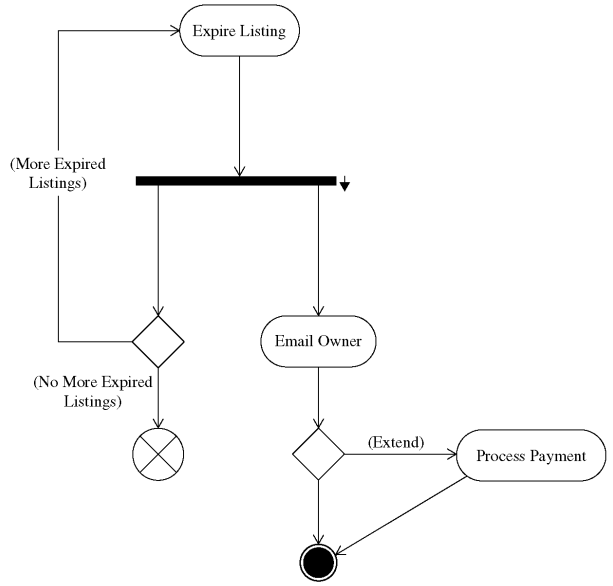

Figure 3-21 An activity showing a flow final node and an activity final node.

dead ends. You might envision this sort of activity implemented as an asynchronous process where each expired listing fires off a process to permit the listing owner to renew the listing.

Creating Activity Diagrams

A decision as important as what goes into an activity diagram is what to diagram. Too often it is easy to keep adding additional models and adding more detail to existing models. The implication, though, is that while you are modeling, someone else is waiting to implement your design, or worse, while you are refining your designs, some poor implementer will have to modify their implementation. For this

reason, it is important to make your activity diagrams relatively simple; limit the activity diagrams you create to important, critical, or challenging aspects of your problem; and avoid trying to make them perfect. A good model that is easy to understand and completed in a timely matter is more valuable than a perfect model later—if there is such a thing as a perfect model.

Examples of the activity diagrams I would create for use cases from Chapter 2 might be an activity for "Manage Job Listing," "Expire Listing," and "Maintain Billing Information." Specifically, I am interested in understanding the critical aspects of the system, especially those for services that are billable items. Common things such as searching or logging are understood well enough that it is unlikely that I would create an activity diagram for them.

Picking what to model and what not to model is a lot like adding salt during cooking: You can always add a little more, but it is hard to remove salt if you have added too much. The same is true with modeling: You cannot recoup time spent modeling obvious activities, but you always can add more activity diagrams later if you need to.

Reengineering Process

Probably the most beneficial use of activity diagrams is to help nondomain personnel—usually the technologists who will implement a solution—understand a domain. Implicit in the preceding statement is that while domain experts and technologists are attempting to reach a common understanding, there is an opportunity to reengineer the process. Let's take a moment to review what is meant by *process reengineering.*

Frequently, people do there jobs every day without ever identifying a formal process. The process knowledge is known only to the practitioners. Often these same organizations are shocked to discover how much overhead and waste exist within their organization. Process reengineering is a kind of pseudoscience that entails first documenting an organization's processes and then looking for ways to optimize those processes.

I am not an expert in process reengineering, but there are historical examples where well-known companies have spent a considerable amount of money and energy to refine their business processes, and the results have led to broad, sweeping changes in industry. An interesting example can be found in *Behind the Golden Arches*, which details the evolutionary path that led McDonald's to use centralized distribution centers for its franchisees.

Note *Ironically enough, software development itself is an example of a domain where the practitioners have defined the process in an ad hoc way. Many software companies are now beginning to realize that they are long overdue for an introspective examination of the processes they follow to build software. Has anyone in your organization every used an activity diagram (or flowchart) to document how your organization builds software?*

Software development is a business of automating solutions to problems. In a general sense it is a useful idea to document critical domain processes and explore some possible optimizations before writing code. If the process is simplified, then the ensuing implementation may be markedly simplified too.

Reengineering a Subactivity

Here is an example involving a subactivity called "Interior Cabin Check" that has to do with the preflight inspection of a small airplane. The idea behind the interior cabin check is that we are looking for required or important things in the interior of the airplane and performing steps to help with some exterior checks. There is a very good likelihood that if we miss something, then we could be taking off with unsafe conditions or not have critical resources during an emergency. (If it bothers you that this doesn't sound like a software problem, then just imagine that we are documenting this problem to write testing or simulation software.)

One of the planes I fly is a Cessna 172 Skyhawk. The interior cabin check (depicted in Figure 3-22) consists of

- Making sure the ignition switch is off
- Turning the master switch on so that we have power
- Lowering the flaps
- Checking for the registration, airworthiness certificate, weight and balance information, and operating handbook, which includes emergency procedures
- Checking the fuel level indicators and fuel selector
- Turning the master switch off

As shown in the activity diagram in Figure 3-22, the steps are carried out consecutively. (This is the way that I performed the inspection the first couple of times I performed it.) An experienced pilot (a domain expert) will tell you that it takes

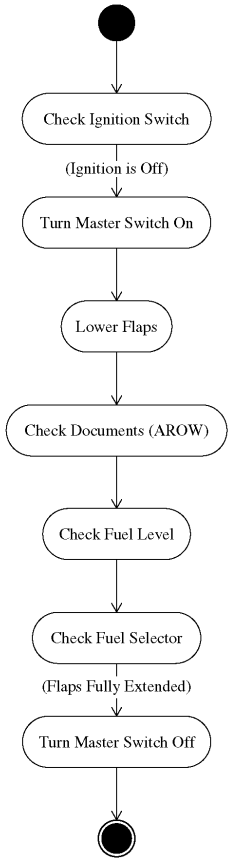

Figure 3-22 Our initial understating is that each task in the activity is performed consecutively.

a few moments for the flaps to come down, so some of the other checks can be performed concurrently. We can tighten up the activity diagram as shown in Figure 3-23.

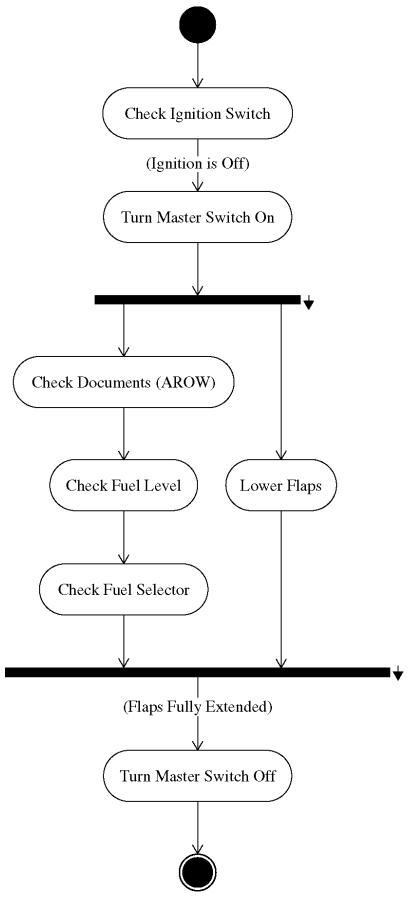

Figure 3-23 Making some tasks concurrent will improve the time to completion of the activity.

Knowing When to Quit

Applying rules consistently will help you to work efficiently during the modeling phase of development. With this in mind, recall that I said that an important idea is to capture the most critical use cases and tackle those first. The same is true of activity diagrams. Identify the use cases that are most critical, and create activity diagrams for those use cases that require some exploration. For instance, authenticating a user in Motown-jobs.com is necessary, but this is a well-understood problem. I wouldn't spend a lot of time creating an activity diagram for this, and I wouldn't work on that activity diagram before I worked on those related to my primary use case, "Manage Job Listing."

If you are not sure how many activity diagrams to create, then try creating an activity diagram for each of the primary functions of your most important use cases. Try to get as much about the activity modeled as quickly and as accurately as you can. Immediately check back with your domain experts, and explore the activities to see if you have captured the most salient points.

Finally, don't permit yourself to get bogged down here. If you can't reach a consensus on the completeness of a particular activity, then set it aside and agree to come back to it. There may be other elements of the problem that will increase your understanding or your users' understanding of the problem in general and resolve the problem you set aside. The key is not to get stuck on any particular problem too early.

Quiz

1. Synonyms for a transition are
   a. connector and flow.
   b. edge and flow.
   c. edge and connector.
   d. action and event.
2. In general, activity diagrams consist of
   a. nodes and edges.
   b. actions and transitions.
   c. actions, decisions, and flows.
   d. symbols and lines.

3. An exception can be shown in an activity diagram with a lightning bolt–shaped edge.
   a. True
   b. False
4. A decision node and merge node use
   a. different symbols.
   b. identical symbols.
   c. either identical or differing symbols depending on context.
5. Multiple flows entering an action node constitute
   a. an implicit merge.
   b. an implicit join.
6. Every flow waits at a merge and join until all flows have arrived.
   a. True
   b. False
7. The swimlane metaphor is no longer used
   a. because swimlanes are no longer part of the UML.
   b. because partitions can be multidimensional and don't look like swimlanes.
   c. The swimlane metaphor is still in use.
   d. Both b and c.
8. Actions can exist in two activity partitions in different dimension at the same time.
   a. True
   b. False
9. A decision and merge node is represented by
   a. an oval.
   b. a circle.
   c. a rectangle.
   d. a diamond.

10. Activity diagrams differ from flowcharts because activity diagrams support
    a. swimlanes.
    b. parallel behavior.
    c. decision nodes.
    d. actions.

Answers

1. b
2. a
3. a
4. b
5. b
6. a
7. d
8. a
9. d
10. b