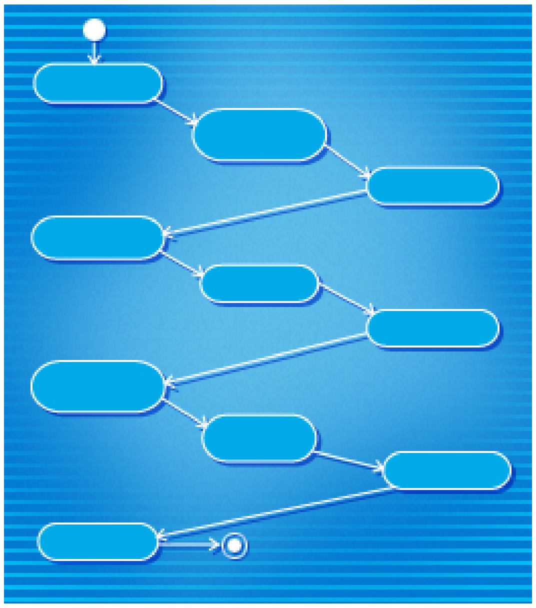
the Rational edge
e-zine for the rational community

UML basics

Part II: The activity diagram

by **Donald Bell**
IBM Global Services

In June 2003, The Rational Edge *introduced a new article series by Donald Bell, IBM Global Services, called* **UML basics**. *The purpose of this series is to help readers become familiar with the major diagrams that compose much of the UML. Part I offered a general overview of these diagrams; this month, we continue the series with a close look at the activity diagram, including this diagram's complete UML v1.4 notation set.*



The activity diagram's purpose

The purpose of the activity diagram is to model the procedural flow of actions that are part of a larger activity. In projects in which use cases are present, activity diagrams can model a specific use case at a more detailed level. However, activity diagrams can be used independently of use cases for modeling a business-level function, such as buying a concert ticket or registering for a college class. Activity diagrams can also be used to model system-level functions, such as how a ticket reservation data mart populates a corporate sales system's data warehouse.

Because it models procedural flow, the activity diagram focuses on the action sequence of execution and the conditions that trigger or guard those actions. The activity diagram is also focused only on the activity's internal actions and *not* on the actions that call the activity in their process flow or that trigger the activity according to some event (e.g., it's 12:30 on April 13th, and Green Day tickets are now on sale for the group's summer tour).

Although UML sequence diagrams can protray the same information as activity diagrams, I personally find activity diagrams best for modeling business-level functions. This is because activity diagrams show all potential sequence flows in an activity, whereas a sequence diagram typically shows only one flow of an activity. In addition, business managers and business process personnel seem to prefer activity diagrams over sequence diagrams -- an activity diagram is less "techie" in appearance, and therefore less intimidating to business people. Besides, business managers are used to seeing flow diagrams, so the "look" of an activity diagram is familiar.

The notation

The activity diagram's notation is very similar to that of a statechart diagram. In fact, according to the UML specification, an activity diagram is a variation of a statechart diagram[1]. So if you are already familiar with statechart diagrams, you will have a leg up on understanding the activity diagram's notation, and much of the discussion below will be review for you.

The basics

First, let's consider the action element in an activity diagram, whose official UML name is *action state*. In my experience, people rarely if ever call it an action state; usually they call it either *action* or *activity*. In this article, I will always refer to it as *action* and will use the term *activity* only to refer to the whole task being modeled by the activity diagram. This distinction will make my explanations easier to understand.

An action is indicated on the activity diagram by a "capsule" shape -- a rectangular object with semicircular left and right ends (see Figure 1). The text inside it indicates the action (e.g., Customer Calls Ticket Office or Registration Office Opens).

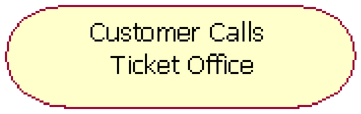


Figure 1: A sample action that is part of an activity diagram.

Because activity diagrams show a sequence of actions, they must indicate the starting point of the sequence. The official UML name for the starting point on the activity diagram is *initial state*, and it is the point at which you begin reading the action sequence. The initial state is drawn as a solid circle with a transition line (arrow) that connects it to the first action in the activity's sequence of actions. Figure 2 shows what an activity diagram's initial state looks like. Although the UML specification does not prescribe the location of the initial state on the activity diagram, it is usually easiest to place the first action at the top left corner of your diagram.

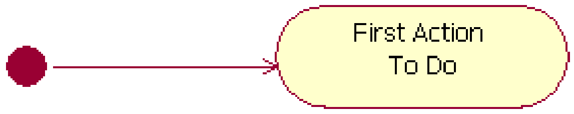


Figure 2: The initial state clearly shows the starting point for the action sequence within an activity diagram.

It is important to note that there can *be only one* initial state on an activity diagram and *only one* transition line connecting the initial state to an action. Although it may seem obvious that an activity can have only one initial state, there are certain circumstances -- namely, the commencement of asynchronous action sequences -- that may suggest that a new initial state should be indicated in the activity diagram. UML does not allow this. Figure 3 offers an example of an incorrect activity diagram, because the initial state has two transition lines that point to two activities.

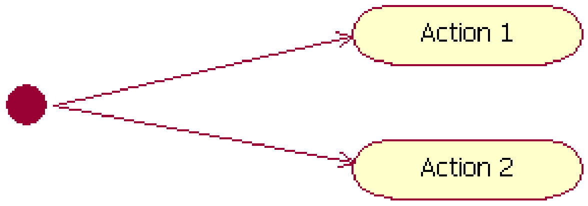


Figure 3: Incorrect rendering of an initial state within an activity diagram. The initial state can indicate only ONE action.

With arrows indicating direction, the transition lines on an activity diagram show the sequential flow of actions in the modeled activity. The arrow will always point to the next action in the activity's sequence. Figure 4 shows a complete activity diagram, modeling how a customer books a concert ticket.

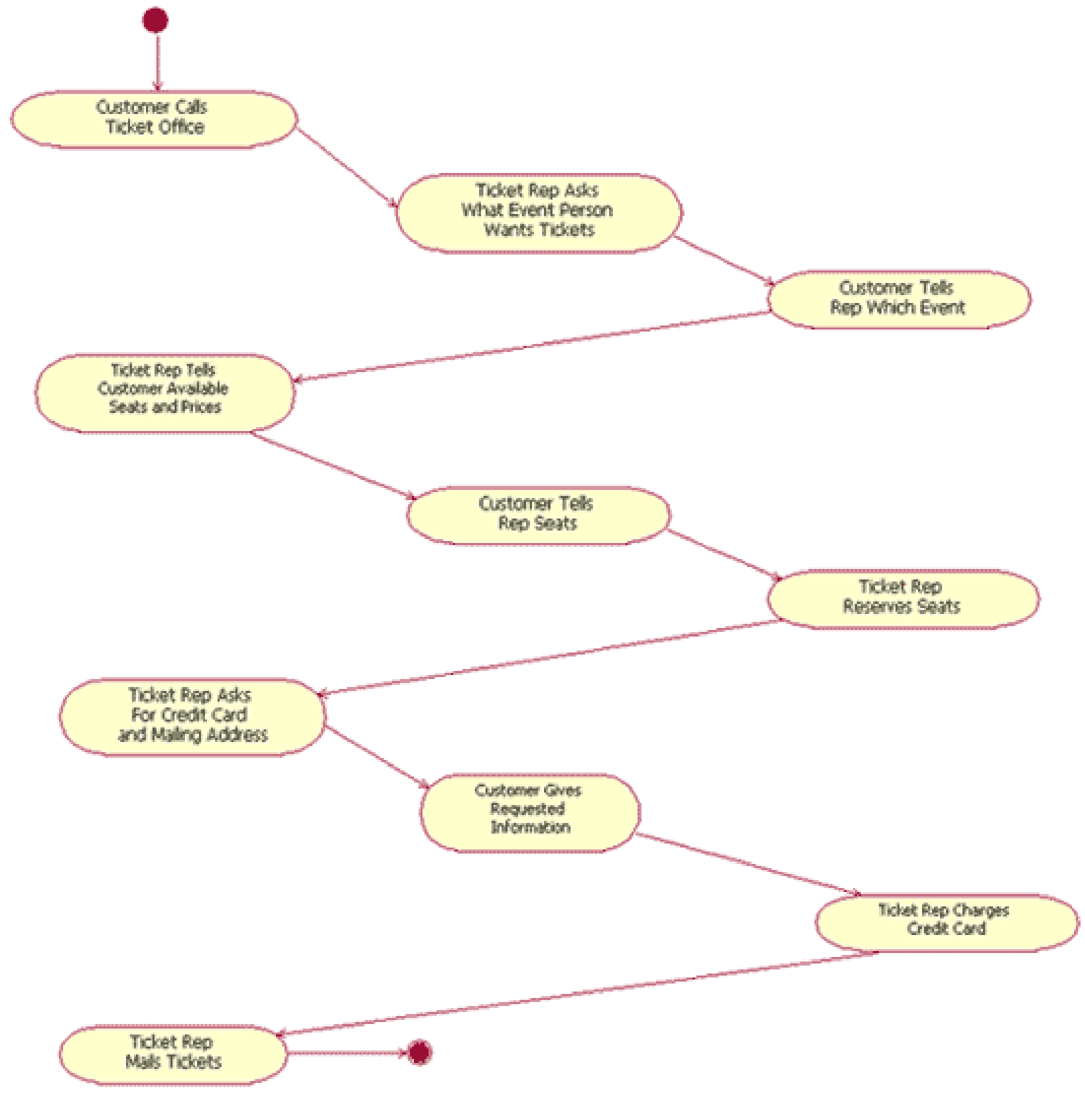

Customer Calls Ticket Office
Ticket Rep Asks What Event Person Wants Tickets
Customer Tells Rep Which Event
Ticket Rep Tells Customer Available Seats and Prices
Customer Tells Rep Seats
Ticket Rep Reserves Seats
Ticket Rep Asks For Credit Card and Mailing Address
Customer Gives Requested Information
Ticket Rep Charges Credit Card
Ticket Rep Mails Tickets

Figure 4: A complete activity diagram makes the sequence of actions easy to understand.

The sample activity diagram in Figure 4 documents the activity "Booking a Concert Ticket," with actions in the following order:

1. Customer calls ticket office.
2. Ticket rep asks what event person wants tickets for.
3. Customer tells rep event choice.
4. Ticket rep tells customer available seats and prices.
5. Customer tells rep seating choice.
6. Ticket rep reserves seats.
7. Ticket rep asks for credit card and billing address.
8. Customer gives requested information.
9. Ticket rep charges credit card.
10. Ticket rep mails tickets.

The above action order is clear from the diagram, because the diagram shows an initial state (starting point), and from that point one can follow the transition lines as they connect the activity's actions.

The activity's flow terminates when the transition line of the last action in the sequence connects to a "final state" symbol, which is a bullseye (a circle surrounding a smaller solid circle). As shown in Figure 4, the action "Ticket Rep Mails Tickets" is connected to a final state symbol, indicating that the activity's action sequence has reached its end. Every activity diagram should have at least one final state symbol; otherwise, readers will be unclear about where the action sequence ends, or perhaps assume that the activity diagram is still a work in progress.

It is possible for an activity diagram to show multiple final states. Unlike initial state symbols, of which there can be only one on an activity diagram, final state symbols can represent the termination of one of many branches in the logic -- in other words, the activity may terminate in different manners.

Beyond the basics

We have covered the basic notation elements of an activity diagram, but there are still more notation elements that can be placed on this type of diagram. Although the Figure 4 diagram is technically complete, the activity modeled in Figure 4 is very simplistic. Typically, activities modeled for real software development projects include decision points that control what actions take place. And sometimes activities have parallel actions.

Decision points

Typically, decisions need to be made throughout an activity, depending on the outcome of a specific prior action. In creating the activity diagram for such cases, you might need to model two or more different sequences of actions. For example, when I order Chinese food for delivery, I call the Chinese food delivery guy, give him my phone number, and his computer will automatically display my address if I've ordered food before. But if I'm a new customer calling for the first time, he must get my address before he takes my order.

The UML specification provides two ways to model decisions like this.

The first way is to show a single transition line coming out of an action and connecting to a decision point. The UML specification name for a decision point is *decision*, and it is drawn as a diamond on an activity diagram. Since a decision will have at least two different outcomes, the decision symbol will have multiple transition lines connecting to different actions. Figure 5 shows a fragment of a sample activity diagram with a decision.

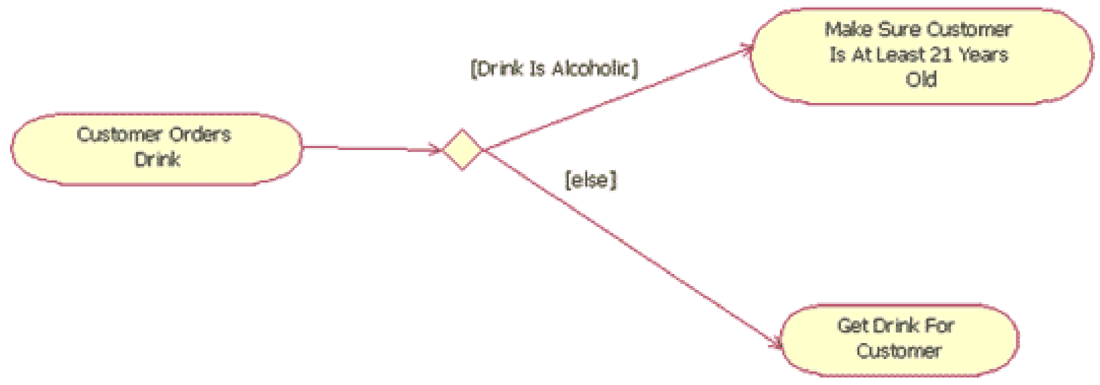


Figure 5: A decision point models a choice that must be made within the sequence of actions.[2]

As shown in Figure 5, each transition line involved in a decision point must be labeled with text above it to indicate "guard conditions," commonly abbreviated as *guards*.

Guard condition text is always placed in brackets -- for example, [guard condition text]. A guard condition explicitly tells when to follow a transition line to the next action. According to the decision point shown in Figure 5, a bartender (user) only needs to "make sure the customer is at least 21 years old" when the customer orders an alcoholic drink. If the customer orders any other type of drink (the "else" condition), then the bartender simply gets the drink for the customer. The [else] guard is commonly used in activity diagrams to mean "if none of the other guarded transition lines matches the actual condition," then follow the [else] transition line.

Merge points

Sometimes the procedural flow from one decision path may connect back to another decision path, as shown in Figure 6 at the "Customer's Age > = 21" condition. In these cases, we connect two or more action paths together using the same diamond icon with multiple paths pointing to it, but with only one transition line coming out of it. This does not indicate a decision point, but rather a *merge*.

Figure 6 shows the same decision as in Figure 5, but Figure 6 expands the activity diagram. Performing a check of the customer's age leads the user to a second decision: If the customer is younger than 21, the bartender must tell the customer to order another non-alcoholic drink, which takes our sequence back to the action "Customer Orders Drink." However, if the customer is 21 years old or older, then our action sequence takes take us to the same action the bartender would follow if the person had ordered a non-alcoholic drink: "Get Drink For Customer."

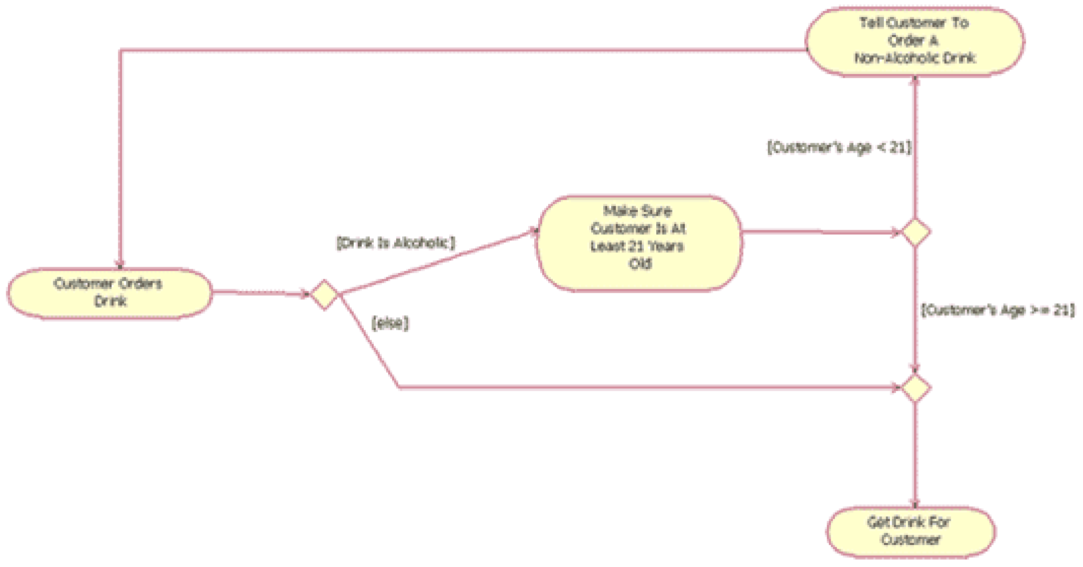


Figure 6: A partial activity diagram, showing two decision points ("Drink is alcoholic" and "Customer's age < 21") and one merge ("else" and "Customer's age >= 21")
Click to enlarge

An alternative approach

The second approach to modeling decisions is to have multiple transition lines coming out of an action, as in Figure 7. If this method is used, then each transition line coming out of the action must have a guard label above it, as with decisions (i.e., the diamond symbols) in the first approach. All the rules that apply to the decision symbol apply to decisions that are modeled out of an action. Personally, I do not recommend this approach, because I prefer a visual queue of the decision. Nevertheless, the UML 1.4 specification allows this approach, as shown in Figure 7, and I mention it here for the sake of completeness.

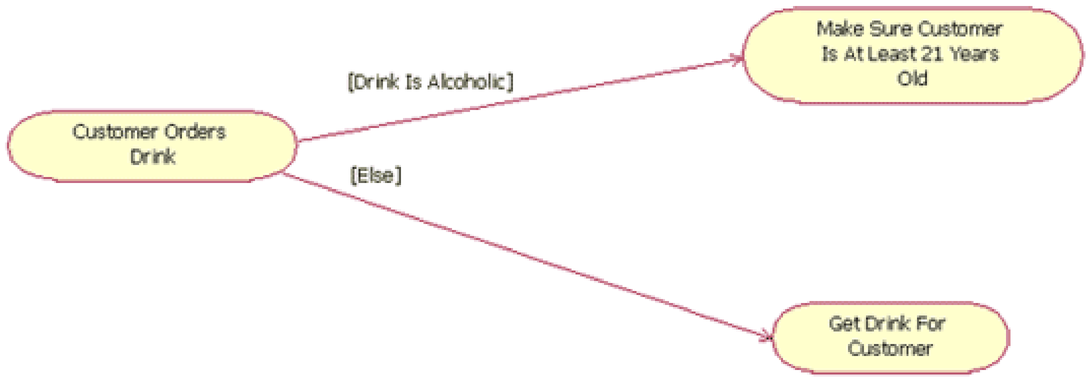


Figure 7: Although I do not recommend this approach, the UML 1.4 specification allows decisions to be modeled as actions with guard conditions.

If you choose this second approach, you must have multiple action sequences merge into a single sequence. To do this, you connect the last transition lines in the specific sequence to the action at which the sequence becomes one again. In Figure 8, this is illustrated with the transition line from "Tell customer to order a non-alcoholic drink" returning to the action "Customer orders drink."

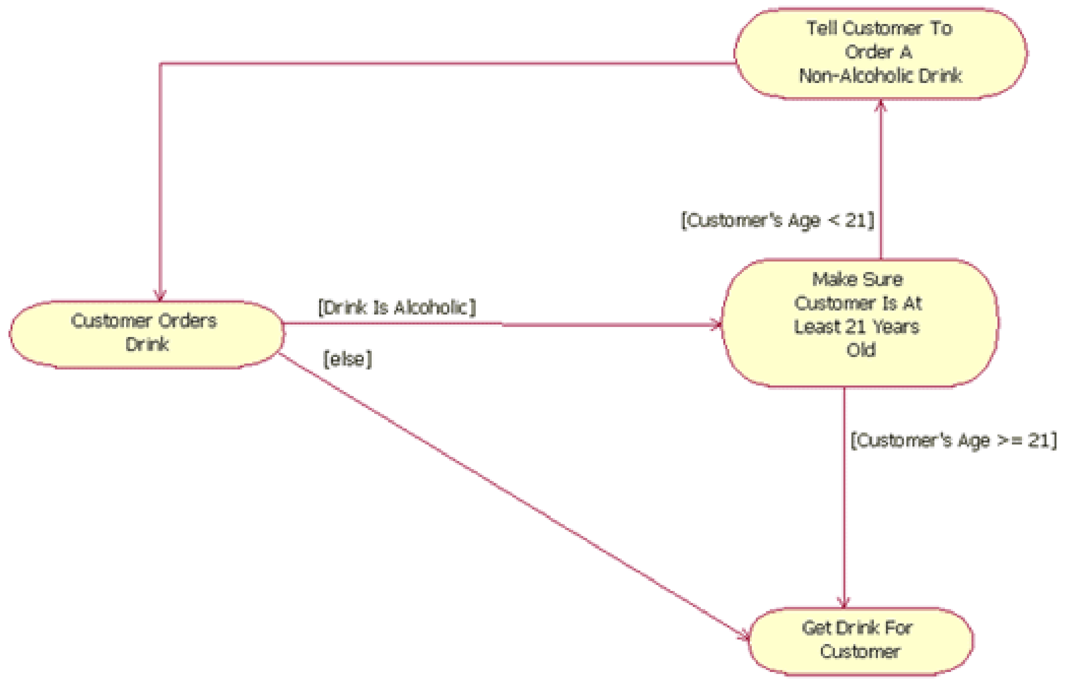


Figure 8: The second approach to modeling decisions

Synch states for asynchronous actions

When modeling activities, you sometimes need to show that certain action sequences can be done in parallel, or asynchronously. A special notation element allows you to show parallel action sequences, or *synch states*, as they are officially named, since they indicate the synchronization status of the flow of activity. Synch states allow the forking and joining of execution threads. To be clear, a synch state that forks actions into two or more threads represents a de-synchronizing of the flow (asynchronous actions), and a synch state that joins actions back together represents a return to synchronized flow. A synch state is drawn as a thick, solid line with transition lines coming into it from the left (usually) and out of it on the right (usually). To draw a synch state that forks the action sequence into mulitple threads, first connect a transition line from the action preceding the parallel sequence to the synch state. Then draw two transition lines coming out of the synch state, each connecting to its own action. Figure 9 is an activity diagram fragment that shows the forking of execution modeled.

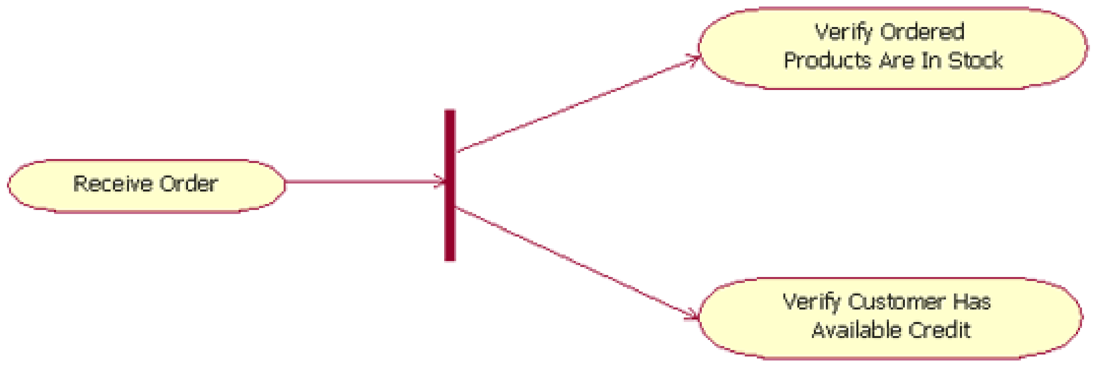

Verify Ordered Products Are In Stock
Receive Order
Verify Customer Has Available Credit

Figure 9: A thick, solid line indicates a *synch state*, allowing two or more action sequences to proceed in parallel.

In the example shown in Figure 9, after the action "Receive Order" is completed, two threads are kicked off in parallel. This allows the system to process both the "Verify Ordered Products Are In Stock" and the "Verify Customer Has Available Credit" actions at the same time.

When you fork execution into multiple threads, typically you have to rejoin them at some point for later processing.[3] Therefore, the synch state element is also used to denote multiple threads joining back together into a single thread. Figure 10 shows an activity diagram fragment that has two threads joining into one.

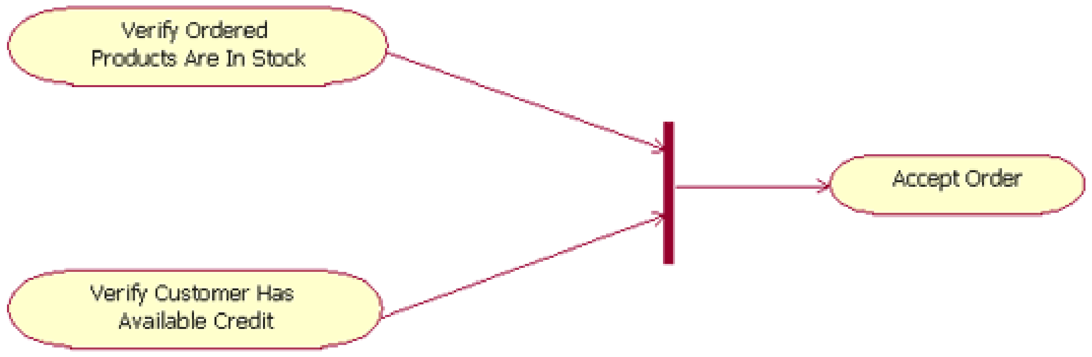

Verify Ordered Products Are In Stock
Accept Order
Verify Customer Has Available Credit

Figure 10: When parallel action sequences terminate, a synch state (thick line) is used to indicate that the muliple threads are joined back into a single thread.

In Figure 10, the "Verify Ordered Products Are In Stock" and the "Verify Customer Has Available Credit" actions shown in Figure 9 have completed processing,[4] and the "Accept Order" action is processed. Note that the single transition line coming out of the synch state means there is now only one thread of execution.

A synch state can also be used as a synchronization point in an activity's overall action execution. In this case, it models how separate threads are forced to join before further execution can proceed. Figure 11 shows an activity diagram fragment that uses a synch state as a synchronization point.

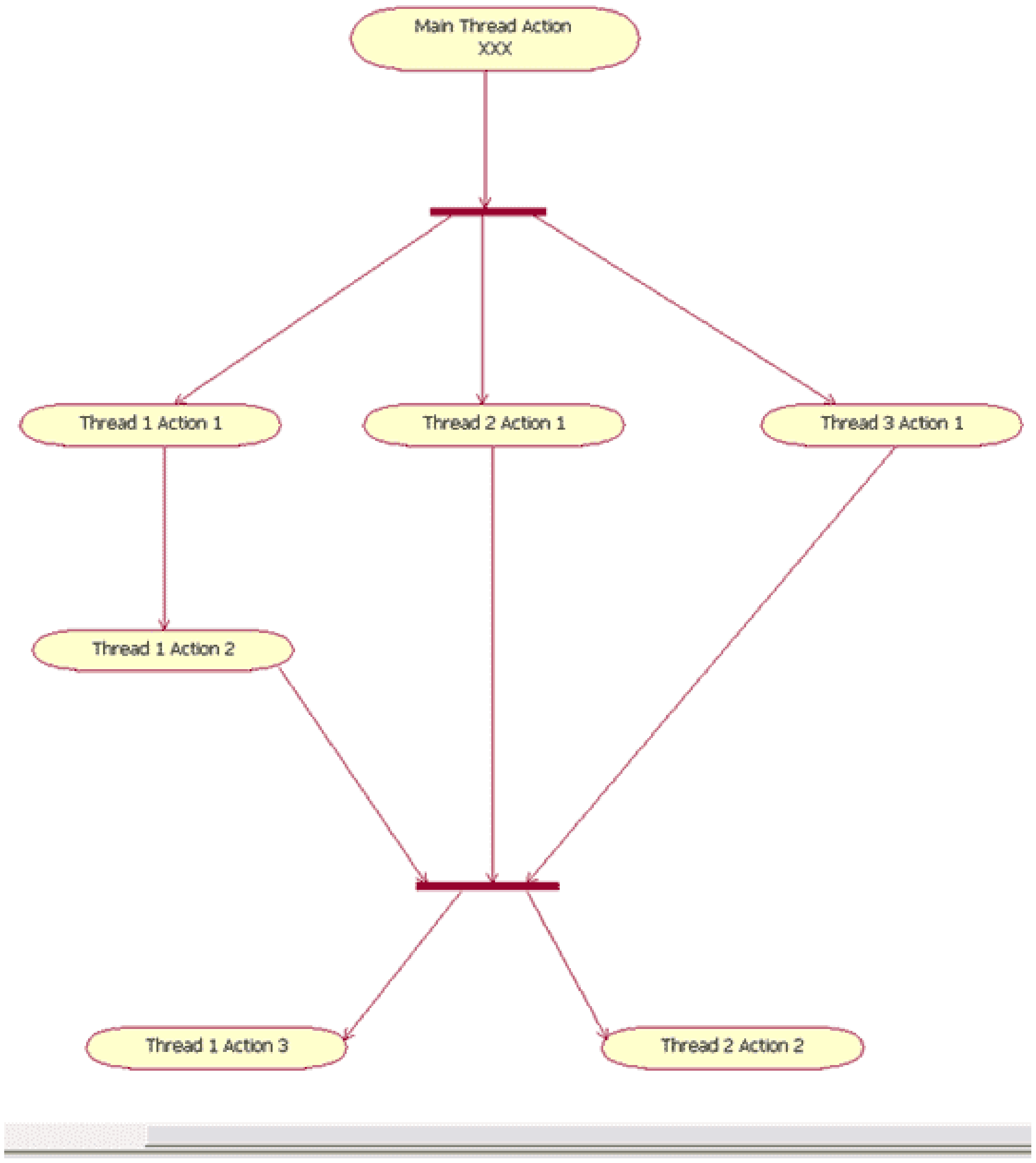


Figure 11: Using a synch state as a synchronization point
Click to enlarge

Although it is abstract, I believe the diagram in Figure 11 provides an easy way to understand how synch states are used as synchronization points. To understand this, let's first be clear about the activity sequence here. The activity starts with all the action in one thread, and when the action "Main Thread Action XXX" is done, the activity breaks into three threads executing in parallel. In the first thread, "Thread 1 Action 1" is executed, then "Thread 1 Action 2" is executed. At the same time this first thread is executing, the second and third thread actions are being executed -- "Thread 2 Action 1" and "Thread 3 Action 1," respectively. By having the second synch state (the one on the right side of Figure 11), we wait until "Thread 1 Action 2," "Thread 2 Action 1," and "Thread 3 Action 1" are completed before proceeding. When all the previous actions are done, the right synch state synchronizes the previous three threads, then two new threads are forked, and both actions on these threads are executed in parallel.

Now, here is what is most significant about this example. Remember that when multiple threads are executing, the actions in one thread must not impact the actions executing in a parallel thread. In Figure 11 the action "Thread 1 Action 1" may be done quickly, and "Thread 1 Action 2" could begin processing before "Thread 2 Action 1" is complete. The only thing that will cause threads to wait for another parallel thread is a synch state, placed as shown in Figure 11.

In all the above examples the synch states are drawn as thick vertical lines; however, the UML specification does not require these lines to be oriented in one way or another. A synch state can be a horizontal thick line or even a diagonal thick line. However, UML diagrams are meant to communicate information as easily as possible; so a person should typically draw a synch state icon as a vertical or horizontal line; only when it makes complete sense (e.g., when running out of space on a whiteboard or piece of paper) should a synch state be drawn at an odd angle.

Swimlanes

In activity diagrams, it is often useful to model the activity's procedural flow of control between the objects (persons, organizations, or other responsible entities) that actually execute the action. To do this, you can add *swimlanes* to the activity diagram (swimlanes are named for their resemblance to the straight-line boundaries between two or more competitors at a swim meet).

To put swimlanes on an activity diagram, use vertical columns. For each object that executes one or more actions, assign a column its name, placed at the top of the column. Then place each action associated with an object in that object's swimlane. Figure 12 shows that two objects execute actions (e.g., Band Manager and Reporting Tool). The Band Manager object executes the "Selects the View Sales For My Band Report" action, and the Reporting Tool executes the "Retrieve Bands the Band Manager Manages" action. It is important to note that, although using swimlanes improves the clarity of an activity diagram (since all the activities are placed in the swimlanes of their respective executor objects), all the previously mentioned rules governing activity diagrams still hold true. In other words, you read the activity diagram just like you would if no swimlanes were used.

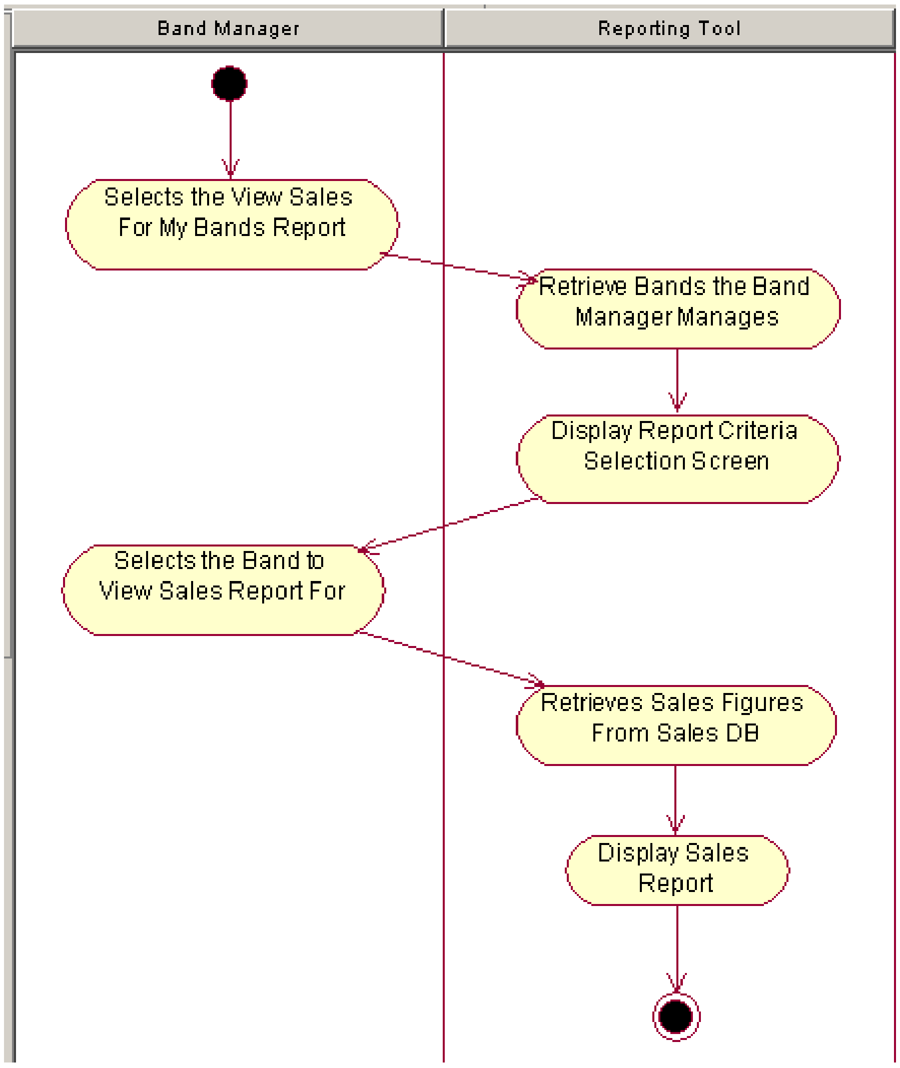

Band Manager
Reporting Tool
Selects the View Sales For My Bands Report
Retrieve Bands the Band Manager Manages
Display Report Criteria Selection Screen
Selects the Band to View Sales Report For
Retrieves Sales Figures From Sales DB
Display Sales Report

Figure 12: Two swimlanes distinguish the actions of the Band Manager object from those of the Reporting Tool object.

Advanced notations elements

As we have seen so far, actions are typically executed by an object. But sometimes an action will output an object that is input to another action. The UML specification does not absolutely require this object output/input to be modeled on an activity diagram, but sometimes it is useful to do so, for the same reason it's helpful to provide swimlanes in the diagram. To model this object flow, UML has a notation called *action-object flow relationship*, and includes two types of notation symbols -- *object flow* and *object in state*.

Object flow

An object flow is the same thing as a transition line, but it is shown as a dashed line instead of a solid one. An object flow *line* is connected to an object in state *symbol*, and another object flow line connects the object in state symbol to the next action. Figure 13 shows this action-object flow relationship.

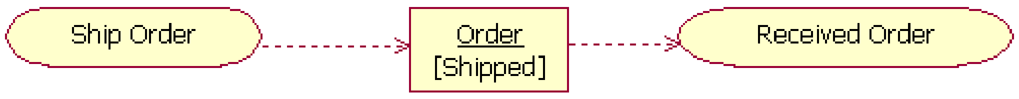
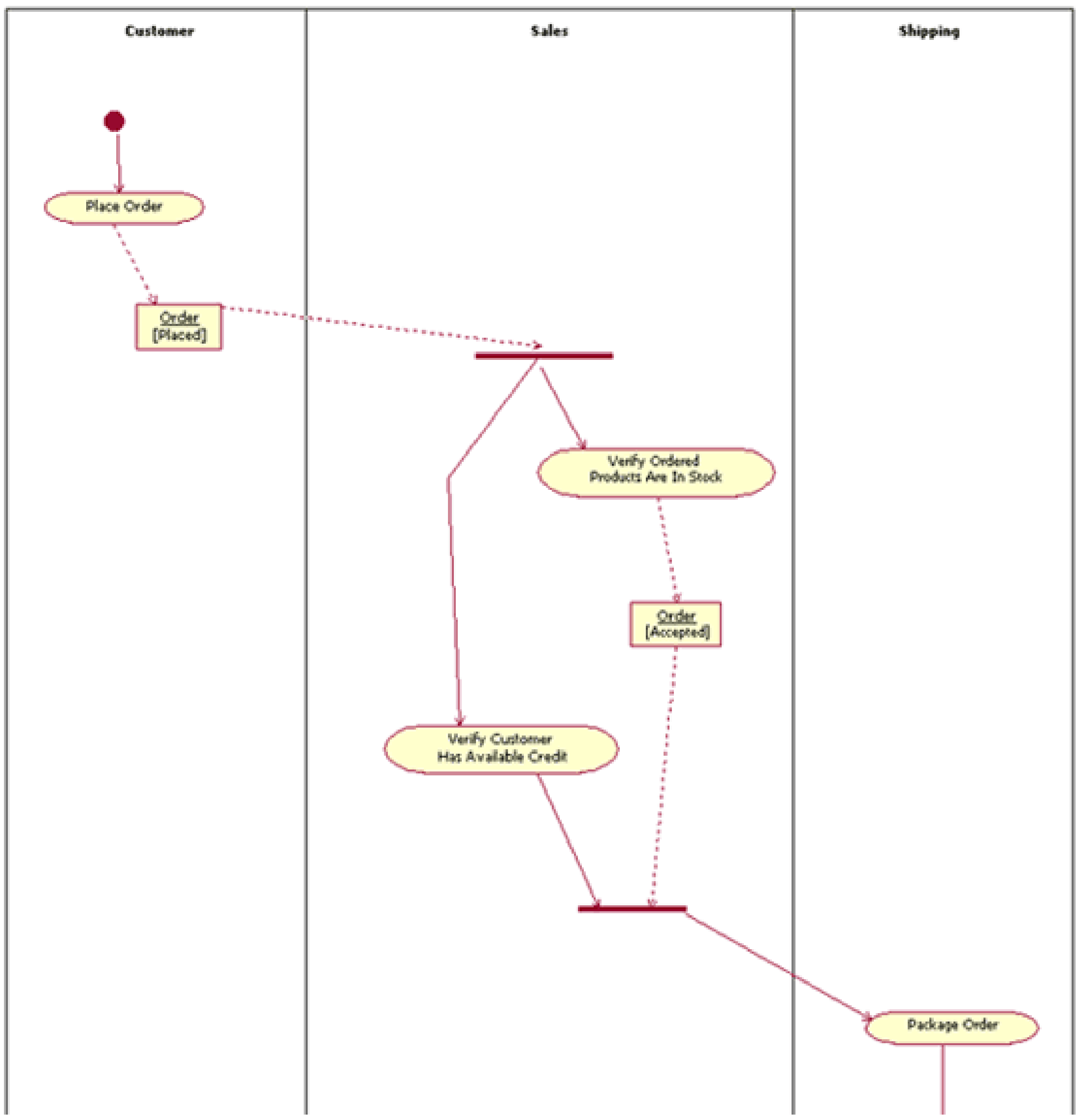


Figure 13: Action-object flow relationship.

Note that in Figure 13, instead of transition lines between the two activities, we see action-object flow relationship symbols. This is because an action-object flow relationship between two actions implies transition to the other action, so transition lines are considered redundant.

Object in state

The object in state symbol is the rectangle in Figure 13. The first part of the object in state symbol is the underlined text. The underlined text in the rectangle is the object's class name -- in our example "Order" -- and this class would be found on one of the modeled system's class diagrams.[5] The second part of the object in state is the text inside the brackets, which is the object's state name. Including the object's state is optional, but I recommend you do so if an action modifies the object's state.[6] Figure 14 shows a complete activity diagram with multiple action-object flow relationships.



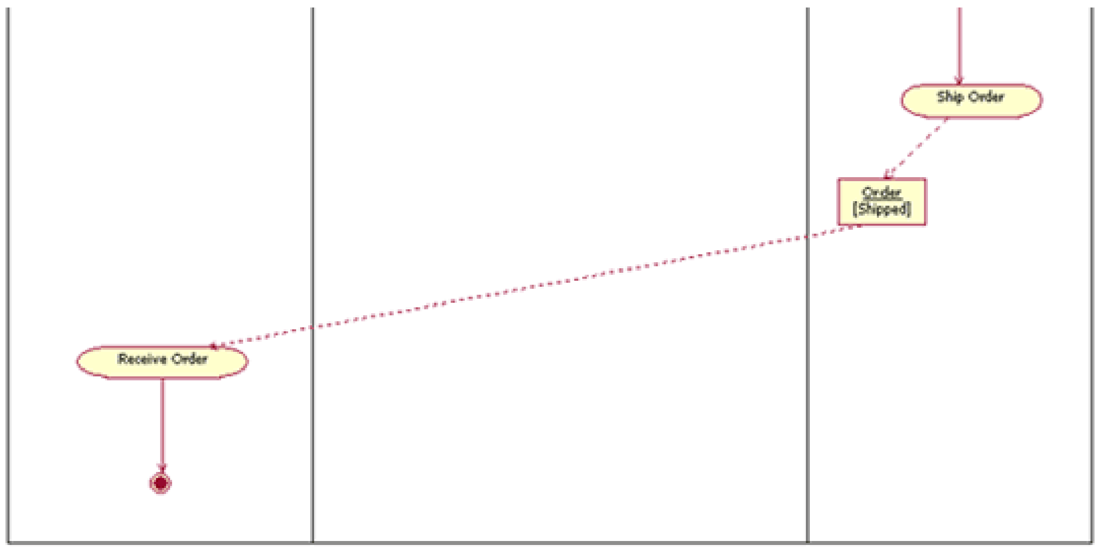


Figure 14: The "Place Order" action puts the Order object into the "Placed" state; then the "Verify Ordered Products Are In Stock" action moves the Order object into the "Accepted" state.
Click to enlarge

The inclusion of action-object flow relationships does not change the way you read an activity diagram; it just provides additional information. In Figure 14, the "Place Order" action puts the Order object into the "Placed" state; later, the "Verify Ordered Products Are In Stock" action moves the Order object into the "Accepted" state. We know that these actions are modifying the Order's states, because the objects in state symbols have the states on them.

Subactivity state

The *subactivity state* represents another activity diagram that you can use when you want to nest another activity in the flow of an activity. A subactivity state is placed where an action state on an activity diagram would be located. You also draw a subactivity state in the same way as an action state, with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol, as shown in Figures 15 and 16.

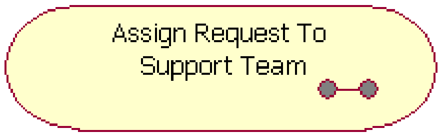


Figure 15: Example of a subactivity state as drawn with IBM Rational XDE™

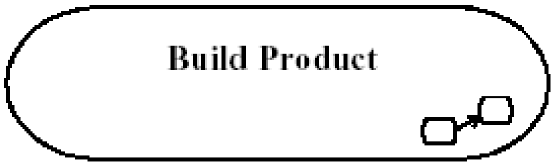


Figure 16: Example of a subactivity state as drawn in the UML specification

Note the slight difference between the subactivity state icons placed in the lower right corners in Figures 15 and 16. The UML 1.4 specification does not explicitly state what the icon should be, but instead says this:

> A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol. The subactivity need not be unique within the diagram. This notation is applicable to any UML construct that supports "nested" structure. The icon must suggest the type of nested structure.

For now, this means that there is not a standard symbol, since subactivity state icons can be different depending on who (or what tool) adds them to the activity diagram. So until firmer agreement is reached on the appearance of this icon, it is best to simply be consistent: Use the same icon every time you represent a subactivity state.

Conclusion

Like all UML diagrams, the number one purpose of the activity diagram is to communicate information effectively. One main reason to include activity diagrams in an overall system model is that they model the procedural flow of control for various activities. This is important, because this sort of model allows business people to get a better understanding of the business environment in which a system will run. Of course, activity diagrams are not limited to modeling business processes; they can also be used to model computer processes.

Typically, you will not use every notation element described in this article when you create your own activity diagrams. But you will make frequent use of the initial state, transition line, action state, and final state notation elements.

In our next installment of this series on essential UML diagrams, we will take a close look at the Class Diagram. See you later this fall.

Notes

1 In the current draft version of UML 2.0, the activity diagram will no longer inherit from the statechart diagram. However, a full discussion of this subject is outside the scope of this article..

2 In the United States, people younger than 21 years of age cannot purchase alcoholic beverages, such as beer or wine.

3 In activity diagrams, if you fork execution into multiple threads there is no requirement to rejoin the threads -- it's possible, though unlikely, to have an activity sequence that breaks off and then just terminates. An example of an activity sequence that might fork off and need not be synchronized back would be a process for notifying an external system or third party of an event.

[4] Remember that one action might take longer to execute than the other.

[5] Although the text implies that every class in a UML model must appear in a class diagram, this is not required by the UML specification. It is completely possible that a class could appear on an activity diagram, but not on any class diagram.

[6] Just as the class Order is modeled on one of the system's class diagrams, the state should also be modeled on a statechart diagram.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!