UML

UML basics: An introduction to the Unified Modeling Language

A little background

As I mentioned, UML was meant to be a unifying language enabling IT professionals to model computer applications. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady Booch, who originally had their own competing methods (OMT, OOSE, and Booch). Eventually, they joined forces and brought about an open standard. (Sound familiar? A similar phenomenon spawned J2EE, SOAP, and Linux.) One reason UML has become a *standard* modeling language is that it is programming-language independent. (UML modeling tools from IBM Rational are used extensively in J2EE shops as well in .NET shops.) Also, the UML notation set is a language and not a methodology. This is important, because a language, as opposed to a methodology, can easily fit into any company's way of conducting business without requiring change.

Since UML is not a methodology, it does not require any formal work products (i.e., "artifacts" in IBM Rational Unified Process® lingo). Yet it does provide several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. There is more to UML than these diagrams, but for my purposes here, the diagrams offer a good introduction to the language and the principles behind its use. By placing standard UML diagrams in your methodology's work products, you make it easier for UML-proficient people to join your project and quickly become productive. The most useful, standard UML diagrams are: use case diagram, class diagram, sequence diagram, statechart diagram, activity diagram, component diagram, and deployment diagram.

It is beyond the scope of this introductory article to go into great detail about each type of diagram. Instead, I will provide you with enough information for a general understanding of each one and then supply more details in later articles.

Use-case diagram

A use case illustrates a unit of functionality provided by the system. The main purpose of the use-case diagram is to help development teams visualize the functional requirements of a system, including the relationship of "actors" (human beings who will interact with the system) to essential processes, as well as the relationships among different use cases. Use-case diagrams generally show groups of use cases -- either all use cases for the complete system, or a breakout of a particular group of use cases with related functionality (e.g., all security administration-related use cases). To show a use case on a use-case diagram, you draw an oval in the middle of the diagram and put the name of the use case in the center of, or below, the oval. To draw an actor (indicating a system user) on a use-case diagram, you draw a stick person to the left or right of your diagram (and just in case you're wondering, some people draw prettier stick people than others). Use simple lines to depict relationships between actors and use cases, as shown in Figure 1.

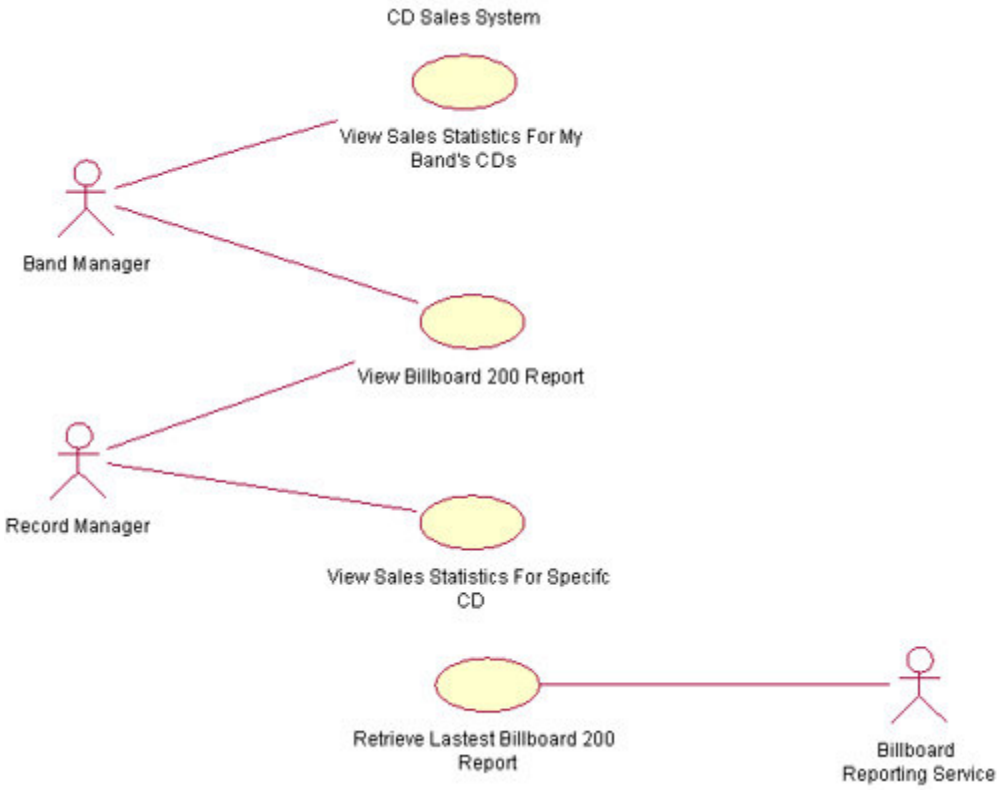

Figure 1: Sample use-case diagram

A use-case diagram is typically used to communicate the high-level functions of the system and the system's scope. By looking at our use-case diagram in Figure 1, you can easily tell the functions that our example system provides. This system lets the band manager view a sales statistics report and the Billboard 200 report for the band's CDs. It also lets the record manager view a sales statistics report and the Billboard 200 report for a particular CD. The diagram also tells us that our system delivers Billboard reports from an external system called Billboard Reporting Service.

In addition, the absence of use cases in this diagram shows what the system *doesn't* do. For example, it does not provide a way for a band manager to listen to songs from the different albums on the Billboard 200 -- i.e., we see no reference to a use case called Listen to Songs from Billboard 200. This absence is not a trivial matter. With clear and simple use-case descriptions provided on such a diagram, a project sponsor can easily see if needed functionality is present or not present in the system.

Class diagram

The class diagram shows how the different entities (people, things, and data) relate to each other; in other words, it shows the static structures of the system. A class diagram can be used to display logical classes, which are typically the kinds of things the business people in an organization talk about -- rock bands, CDs, radio play; or loans, home mortgages, car loans, and interest rates. Class diagrams can also be used to show implementation classes, which are the things that programmers typically deal with. An implementation class diagram will probably show some of the same classes as the logical classes diagram.The implementation class diagram won't be drawn with the same attributes, however, because it will most likely have references to things like Vectors and HashMaps.

A class is depicted on the class diagram as a rectangle with three horizontal sections, as shown in Figure 2. The upper section shows the class's name; the middle section contains the class's attributes; and the lower section contains the class's operations (or "methods").

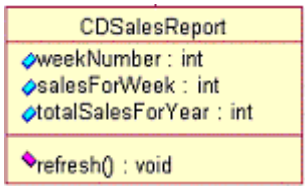

Figure 2: Sample class object in a class diagram

In my experience, almost every developer knows what this diagram is, yet I find that most programmers draw the relationship lines incorrectly. For a class diagram like the one in Figure 3,you should draw the inheritance relationship[1] using a line with an arrowhead at the top pointing to the super class, and the arrowhead should a *completed triangle*. An association relationship should be a solid line if both classes are aware of each other and a line with an *open arrowhead* if the association is known by only one of the classes.

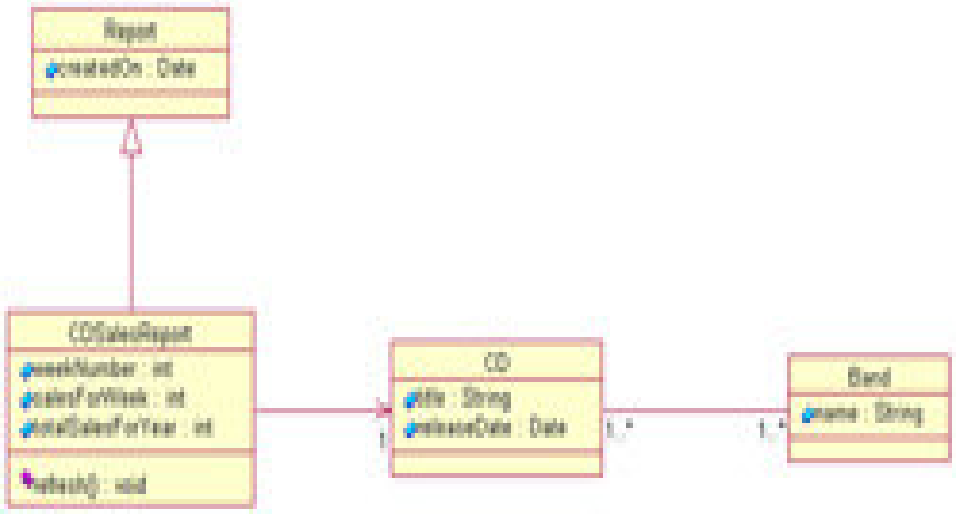

Figure 3: A complete class diagram, including the class object shown in Figure 2

In Figure 3, we see both the inheritance relationship and two association relationships. The CDSalesReport class inherits from the Report class. A CDSalesReport is associated with one CD, but the CD class doesn't know anything about the CDSalesReport class. The CD and the Band classes both know about each other, and both classes can be associated to one or more of each other.

A class diagram can incorporate many more concepts, which we will cover later in this article series.

Sequence diagram

Sequence diagrams show a detailed flow for a specific use case or even just part of a specific use case. They are almost self explanatory; they show the calls between the different objects in their sequence and can show, at a detailed level, different calls to different objects.

A sequence diagram has two dimensions: The vertical dimension shows the sequence of messages/calls in the time order that they occur; the horizontal dimension shows the object instances to which the messages are sent.

A sequence diagram is very simple to draw. Across the top of your diagram, identify the class instances (objects) by putting each class instance inside a box (see Figure 4). In the box, put the class instance name and class name separated by a space/colon/space " : " (e.g., myReportGenerator : ReportGenerator). If a class instance sends a message to another class instance, draw a line with an open arrowhead pointing to the receiving class instance; place the name of the message/method above the line. Optionally, for important messages, you can draw a dotted

line with an arrowhead pointing back to the originating class instance; label the return value above the dotted line. Personally, I always like to include the return value lines because I find the extra details make it easier to read.

Reading a sequence diagram is very simple. Start at the top left corner with the "driver" class instance that starts the sequence. Then follow each message down the diagram. Remember: Even though the example sequence diagram in Figure 4 shows a return message for each sent message, this is optional.

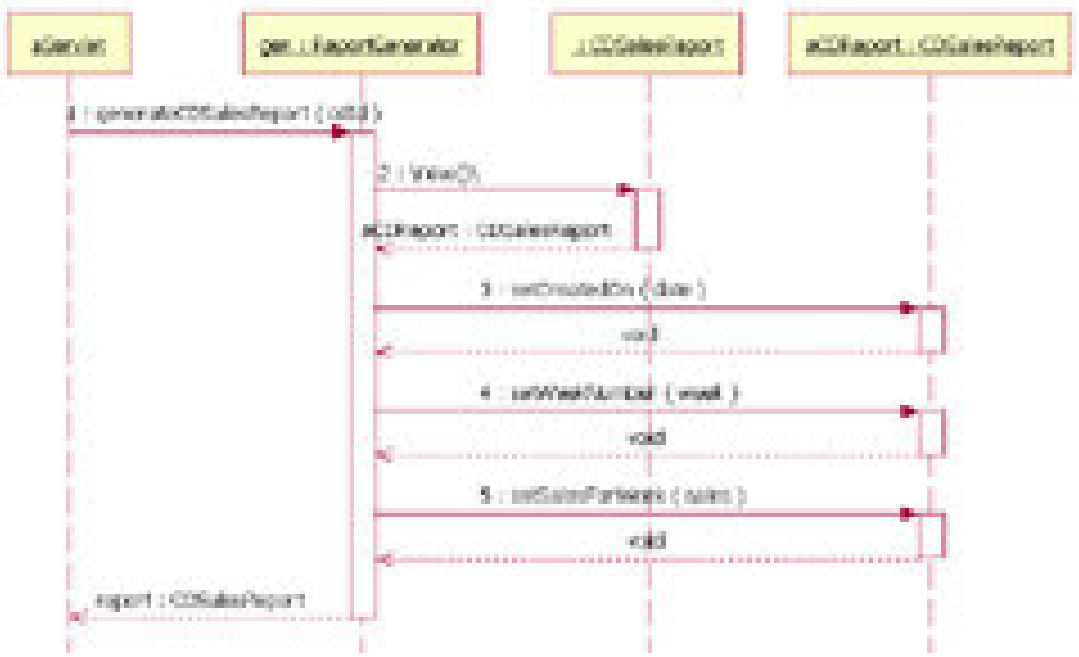

Figure 4: A sample sequence diagram

By reading our sample sequence diagram in Figure 4, you can see how to create a CD Sales Report. The aServlet object is our example driver. aServlet sends a message to the ReportGenerator class instance named gen. The message is labeled generateCDSalesReport, which means that the ReportGenerator object implements this message handler. On closer inspection, the generateCDSalesReport message label has cdId in parentheses, which means that aServlet is passing a variable named cdId with the message. When gen instance receives a generateCDSalesReport message, it then makes subsequent calls to the CDSalesReport class, and an actual instance of a CDSalesReport called aCDReport gets returned. The gen instance then makes calls to the returned aCDReport instance, passing it parameters on each message call. At the end of the sequence, the gen instance returns aCDReport to its caller aServlet.

Please note: The sequence diagram in Figure 4 is arguably too detailed for a typical sequence diagram. However, I believe it is simple enough to understand, and it shows how nested calls are drawn. Also, with junior developers, sometimes it is necessary to break down sequences to this explicit level to help them understand what they are supposed to do.

Statechart diagram

The statechart diagram models the different states that a class can be in and how that class transitions from state to state. It can be argued that every class has a state, but that every class shouldn't have a statechart diagram. Only classes with "interesting" states -- that is, classes with three or more potential states during system activity -- should be modeled.

As shown in Figure 5, the notation set of the statechart diagram has five basic elements: the initial starting point, which is drawn using a solid circle; a transition between states, which is drawn using a line with an open arrowhead; a state, which is drawn using a rectangle with rounded corners; a decision point, which is drawn as an open circle; and one or more termination points, which are drawn using a circle with a solid circle inside it. To draw a statechart diagram, begin with a starting point and a transition line pointing to the initial state of the class. Draw the states themselves anywhere on the diagram, and then simply connect them using the state transition lines.

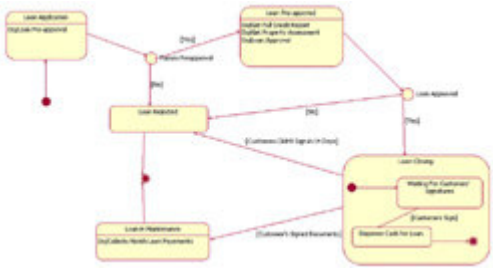

Figure 5: Statechart diagram showing the various states that classes pass through in a functioning system

The example statechart diagram in Figure 5 shows some of the potential information they can communicate. For instance, you can tell that loan processing begins in the Loan Application state. When the pre-approval process is done, depending on the outcome, you move to either the Loan Pre-approved state or the Loan Rejected state. This decision, which is made during the transition process, is shown with a decision point -- the empty circle in the transition line. By looking at the example, a person can tell that a loan cannot go from the Loan Pre-Approved state to the Loan in Maintenance state without going through the Loan Closing state. Also, by looking at our example diagram, a person can tell that all loans will end in either the Loan Rejected state or the Loan in Maintenance state.

Activity diagram

Activity diagrams show the procedural flow of control between two or more class objects while processing an activity. Activity diagrams can be used to model higher-level business process at the business unit level, or to model low-level internal class actions. In my experience, activity diagrams are best used to model higher-level processes, such as how the company is currently doing business, or how it would like to do business. This is because activity diagrams are "less technical" in appearance, compared to sequence diagrams, and business-minded people tend to understand them more quickly.

An activity diagram's notation set is similar to that used in a statechart diagram. Like a statechart diagram, the activity diagram starts with a solid circle connected to the initial activity. The activity is modeled by drawing a rectangle with rounded edges, enclosing the activity's name. Activities can be connected to other activities through transition lines, or to decision points that connect to different activities guarded by conditions of the decision point. Activities that terminate the modeled process are connected to a termination point (just as in a statechart diagram). Optionally, the activities can be grouped into swimlanes, which are used to indicate the object that actually performs the activity, as shown in Figure 6.

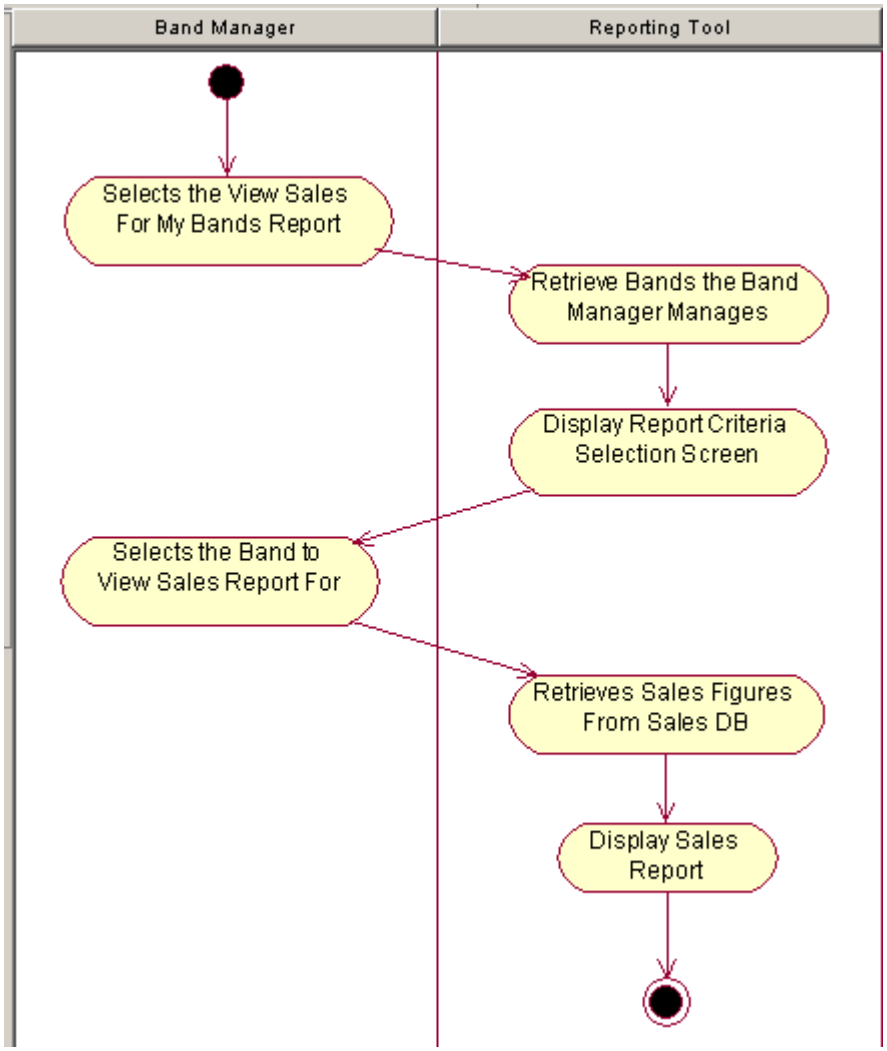

Figure 6: Activity diagram, with two swimlanes to indicate control of activity by two objects: the band manager, and the reporting tool

In our example activity diagram, we have two swimlanes because we have two objects that control separate activities: a band manager and a reporting tool. The process starts with the band manager electing to view the sales report for one of his bands. The reporting tool then retrieves and displays all the bands that person manages and asks him to choose one. After the band manager selects a band, the reporting tool retrieves the sales information and displays the sales report. The activity diagram shows that displaying the report is the last step in the process.

Component diagram

A component diagram provides a physical view of the system. Its purpose is to show the dependencies that the software has on the other software components (e.g., software libraries) in the system. The diagram can be shown at a very high level, with just the large-grain components, or it can be shown at the component package level.[2]

Modeling a component diagram is best described through an example. Figure 7 shows four components: Reporting Tool, Billboard Service, Servlet 2.2 API, and JDBC API. The arrowed lines from the Reporting Tool component to the Billboard Service, Servlet 2.2 API, and JDBC API components mean that the Reporting Tool is dependent on those three components.

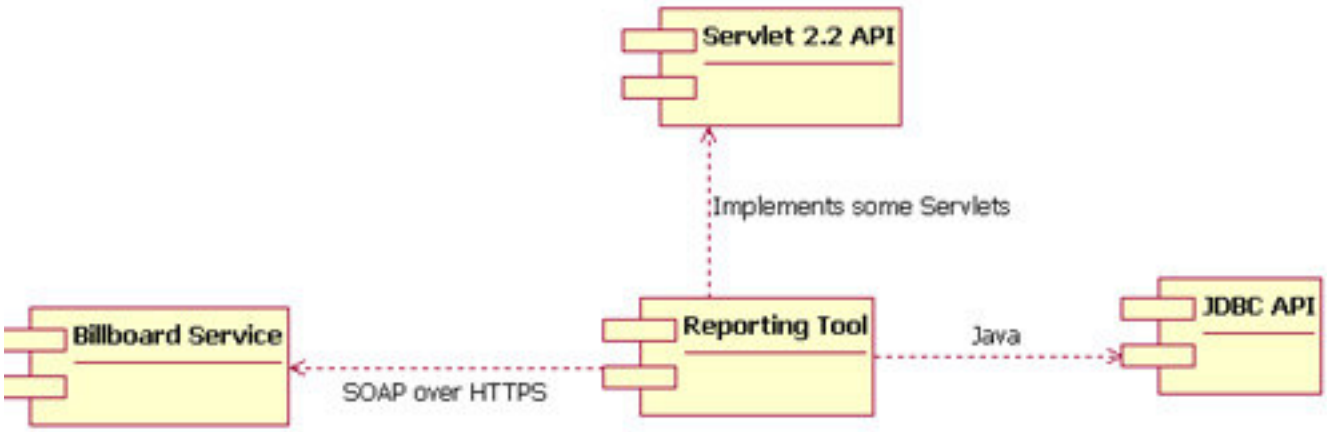

Figure 7: A component diagram shows interdependencies of various software components the system comprises

Deployment diagram

The deployment diagram shows how a system will be physically deployed in the hardware environment. Its purpose is to show where the different components of the system will physically run and how they will communicate with each other. Since the diagram models the physical runtime, a system's production staff will make considerable use of this diagram.

The notation in a deployment diagram includes the notation elements used in a component diagram, with a couple of additions, including the concept of a node. A node represents either a physical machine or a virtual machine node (e.g., a mainframe node). To model a node, simply draw a three-dimensional cube with the name of the node at the top of the cube. Use the naming convention used in sequence diagrams: [instance name] : [instance type] (e.g., "w3reporting.myco.com : Application Server").

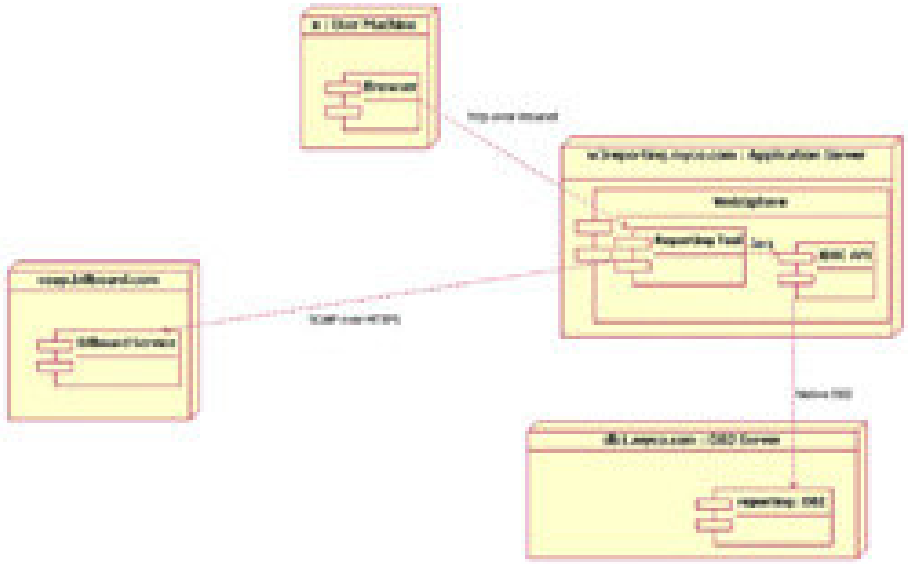

Figure 8: Deployment diagram. ***Because the Reporting Tool component is drawn inside of IBM WebSphere, which in turn is drawn inside of the node w3.reporting.myco.com, we know that users will access the Reporting Tool via a browser running on their local machine, and connecting via HTTP over their company's intranet.***

The deployment diagram in Figure 8 shows that the users access the Reporting Tool by using a browser running on their local machine and connecting via HTTP over their company's intranet to the Reporting Tool. This tool physically runs on the Application Server named w3reporting.myco.com. The diagram shows the Reporting Tool component drawn inside of IBM WebSphere, which in turn is drawn inside of the node w3.reporting.myco.com. The Reporting Tool connects to its reporting database using the Java language to IBM DB2's JDBC interface, which then communicates to the actual DB2 database running on the server named db1.myco.com using native DB2 communication. In addition to talking to the reporting database, the Report Tool component communicates via SOAP over HTTPS to the Billboard Service.

Conclusion

Although this article provides only a brief introduction to Unified Modeling Language, I encourage you to start applying the information you have learned here to your own projects and to dig more deeply into UML. There are several software tools that help you to integrate UML diagrams into your software development process, but even without automated tools, you can use markers on a whiteboard or paper and pencils to draw your UML diagrams and still achieve benefits.

Notes

[1] For more information on inheritance and other object-oriented principles, see http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

[2] The phrase component package level is a programming language-neutral way of referring to class container levels such as .NET's namespaces (e.g., System.Web.UI) or Java's packages (e.g., java.util).

UML basics

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).

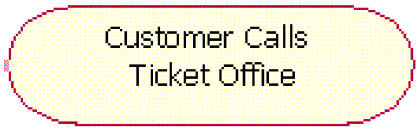

Figure 1: A sample action that is part of an activity diagram.

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.

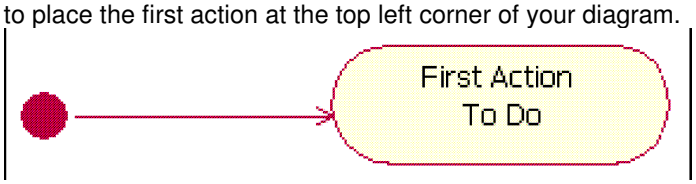

Figure 2: The initial state clearly shows the starting point for the action sequence within an activity diagram.

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.

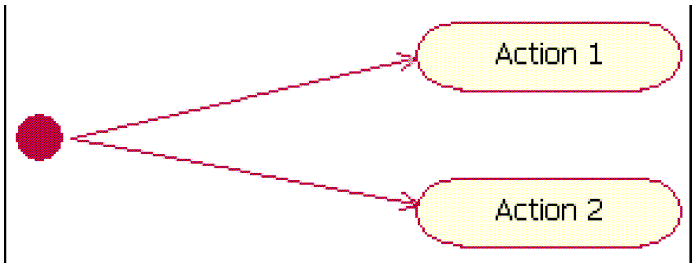

Figure 3: Incorrect rendering of an initial state within an activity diagram. The initial state can indicate only ONE action.

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.

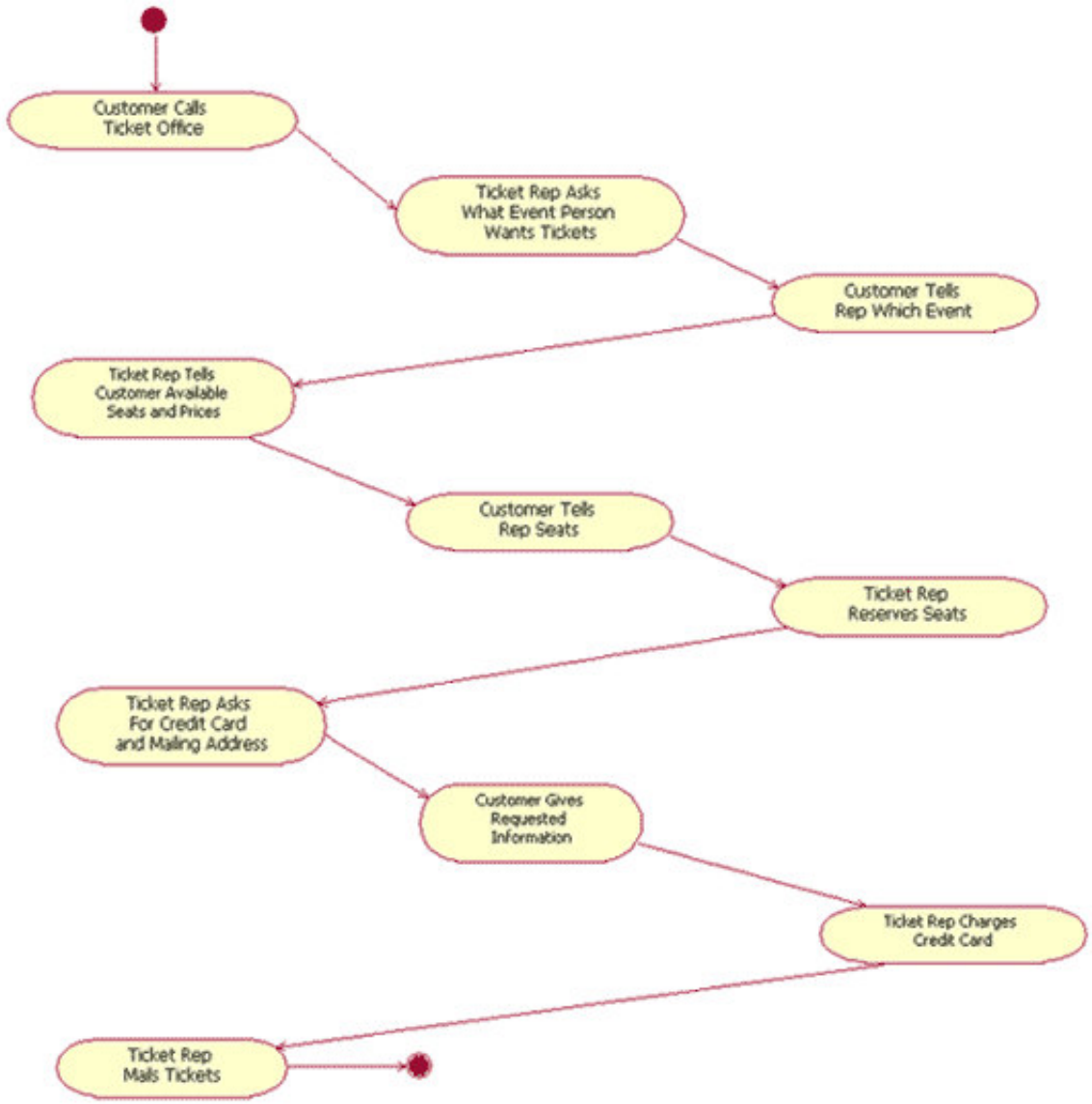

Figure 4: A complete activity diagram makes the sequence of actions easy to understand.

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.

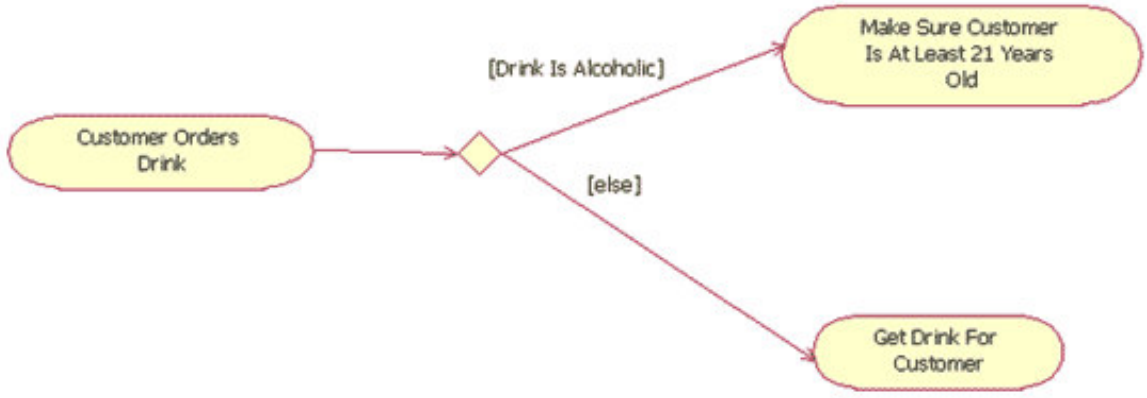

Figure 5: A decision point models a choice that must be made within the sequence of actions.[2]

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."

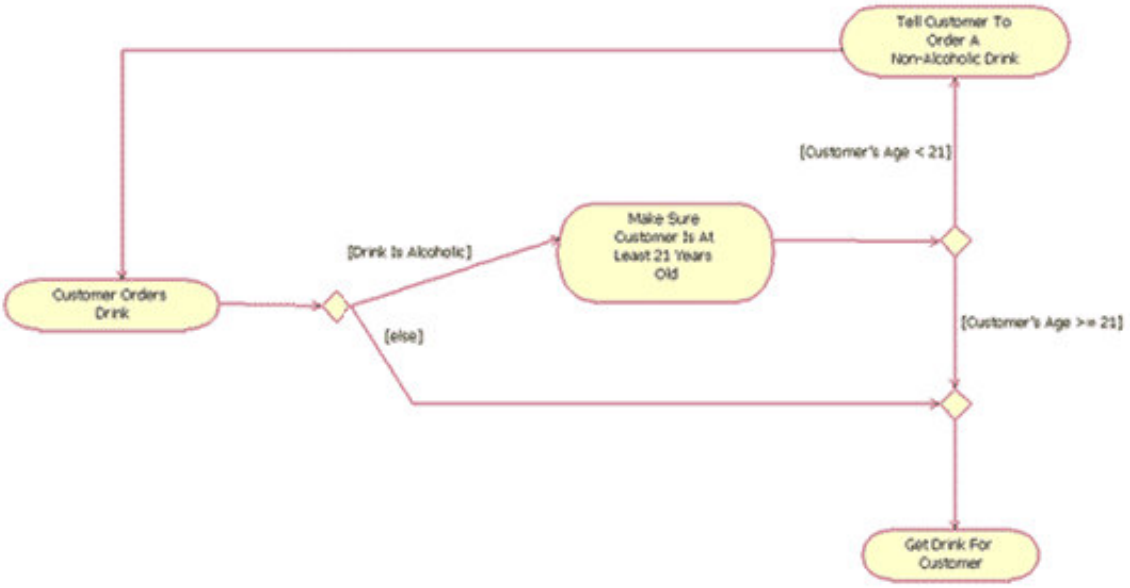

Figure 6: A partial activity diagram, showing two decision points ("Drink is alcoholic" and "Customer's age < 21") and one merge ("else" and "Customer's age >= 21")

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.

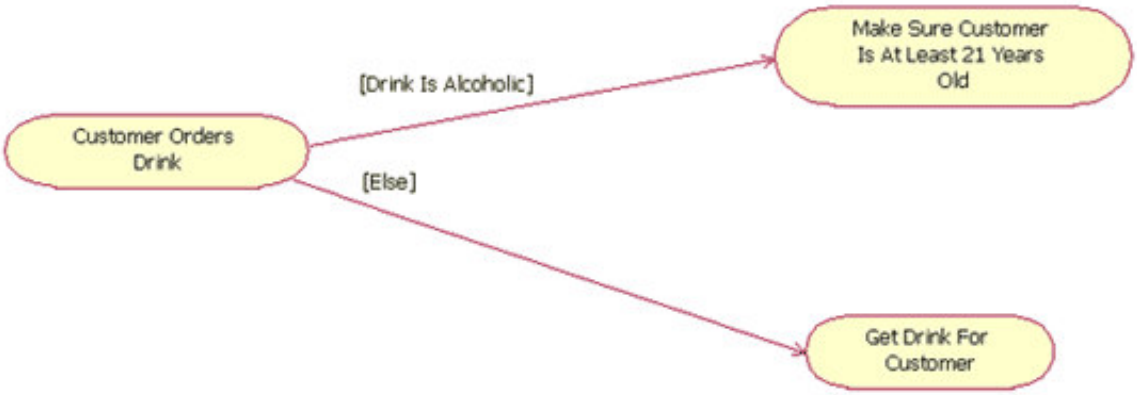

Figure 7: Although I do not recommend this approach, the UML 1.4 specification allows decisions to be modeled as actions with guard conditions.

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."

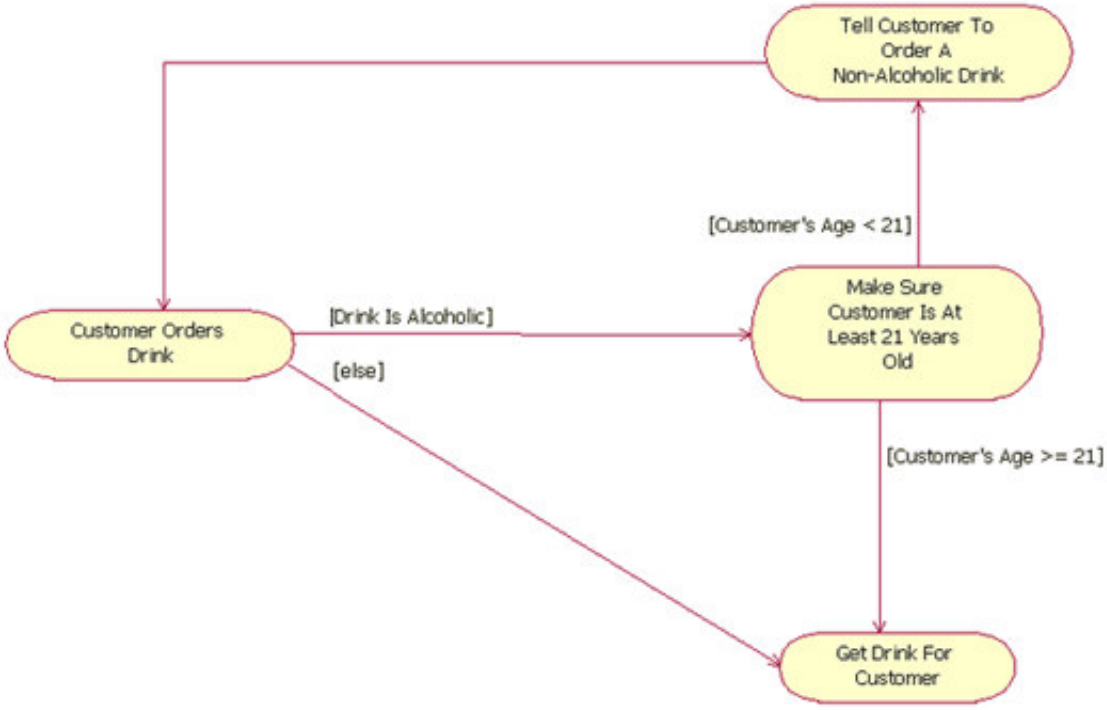

Figure 8: The second approach to modeling decisions

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.

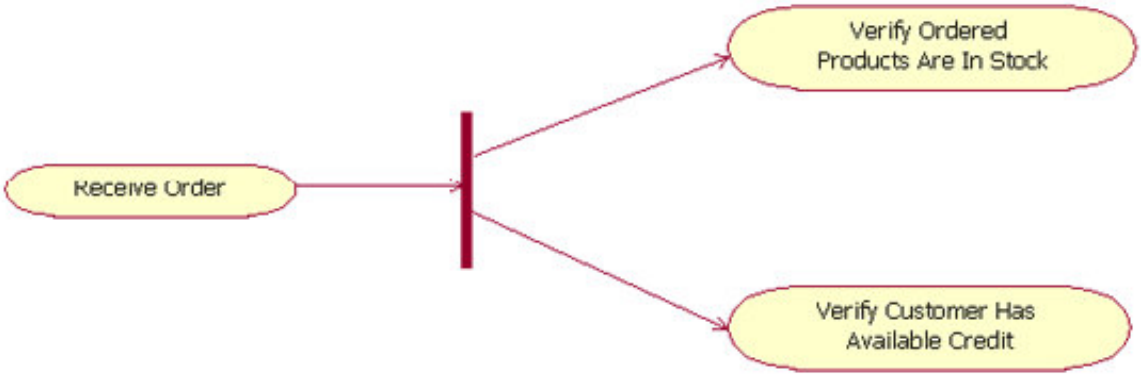

Figure 9: A thick, solid line indicates a *synch state*, allowing two or more action sequences to proceed in parallel.

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.

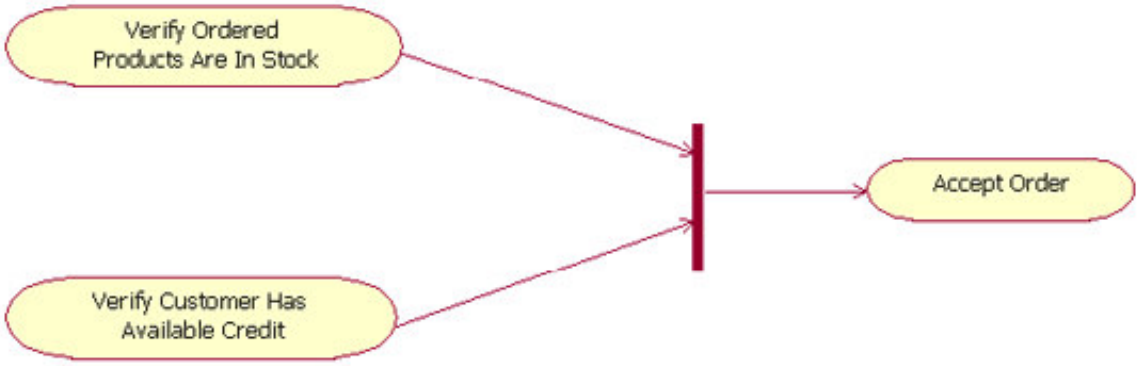

Figure 10: When parallel action sequences terminate, a synch state (thick line) is used to indicate that the muliple threads are joined back into a single thread.

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.

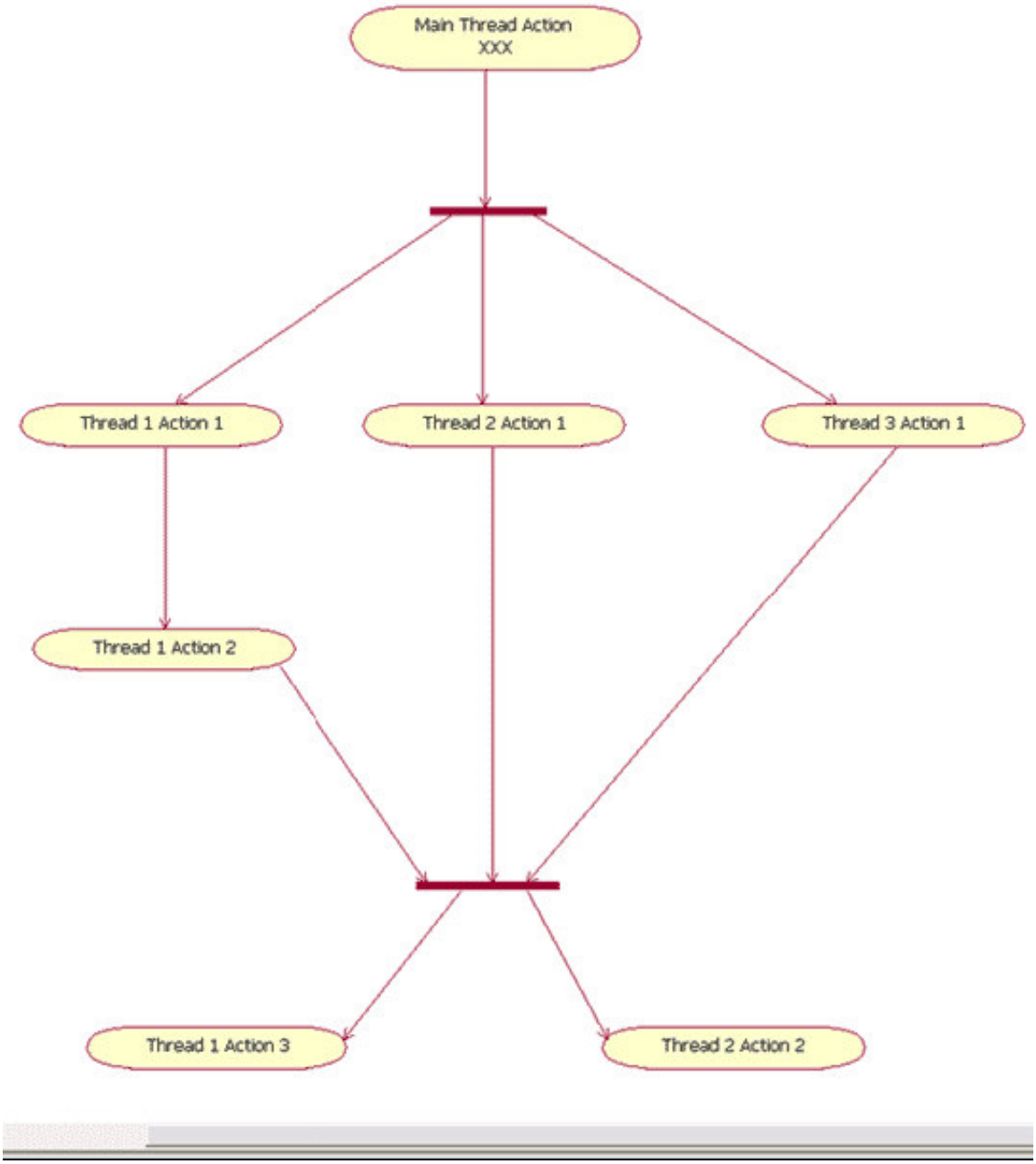

Figure 11: Using a synch state as a synchronization point
Click to enlarge

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.

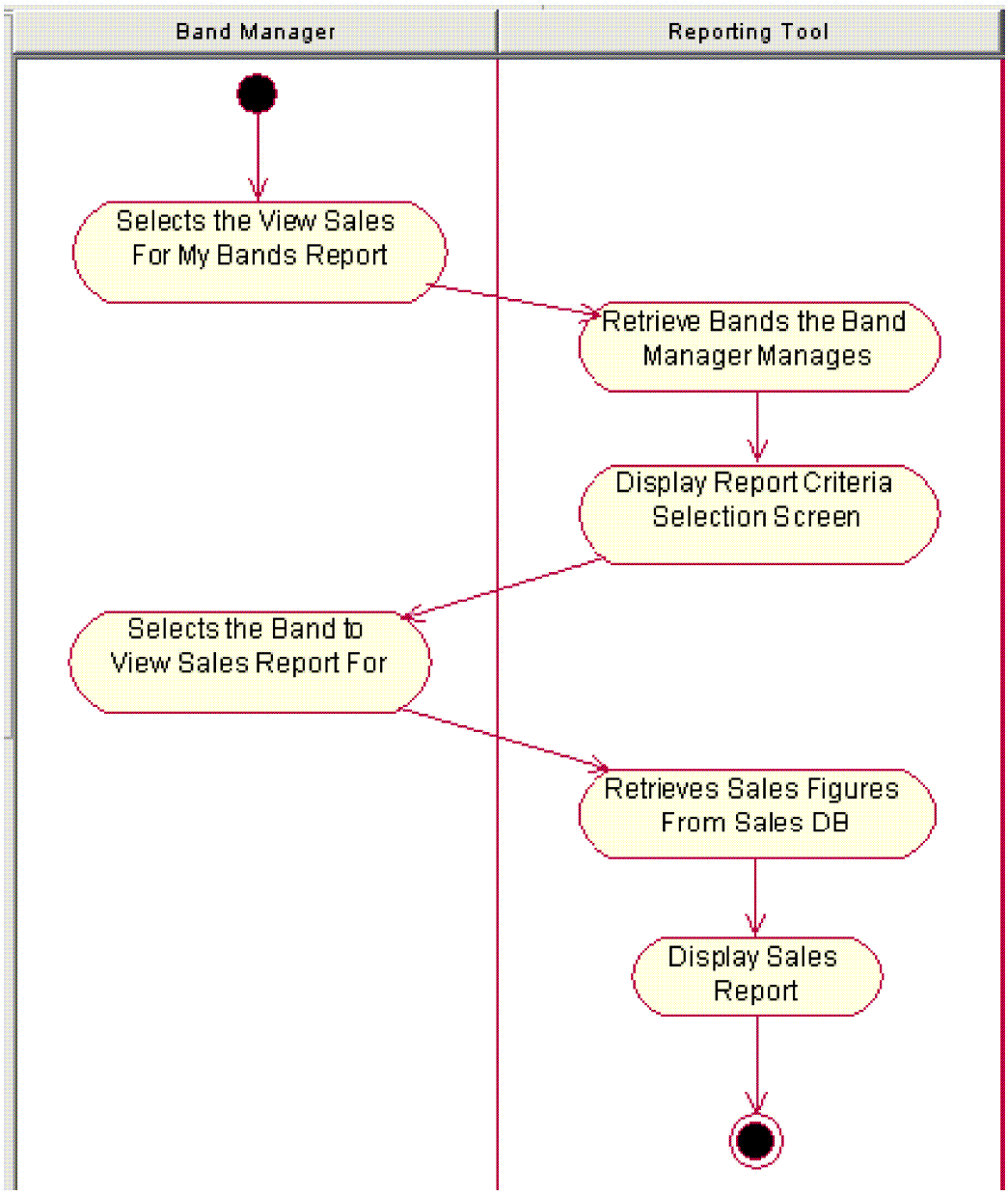

Figure 12: Two swimlanes distinguish the actions of the Band Manager object from those of the Reporting Tool object.

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.

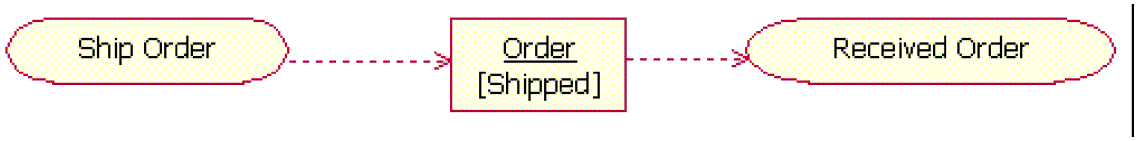

Figure 13: Action-object flow relationship.

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.

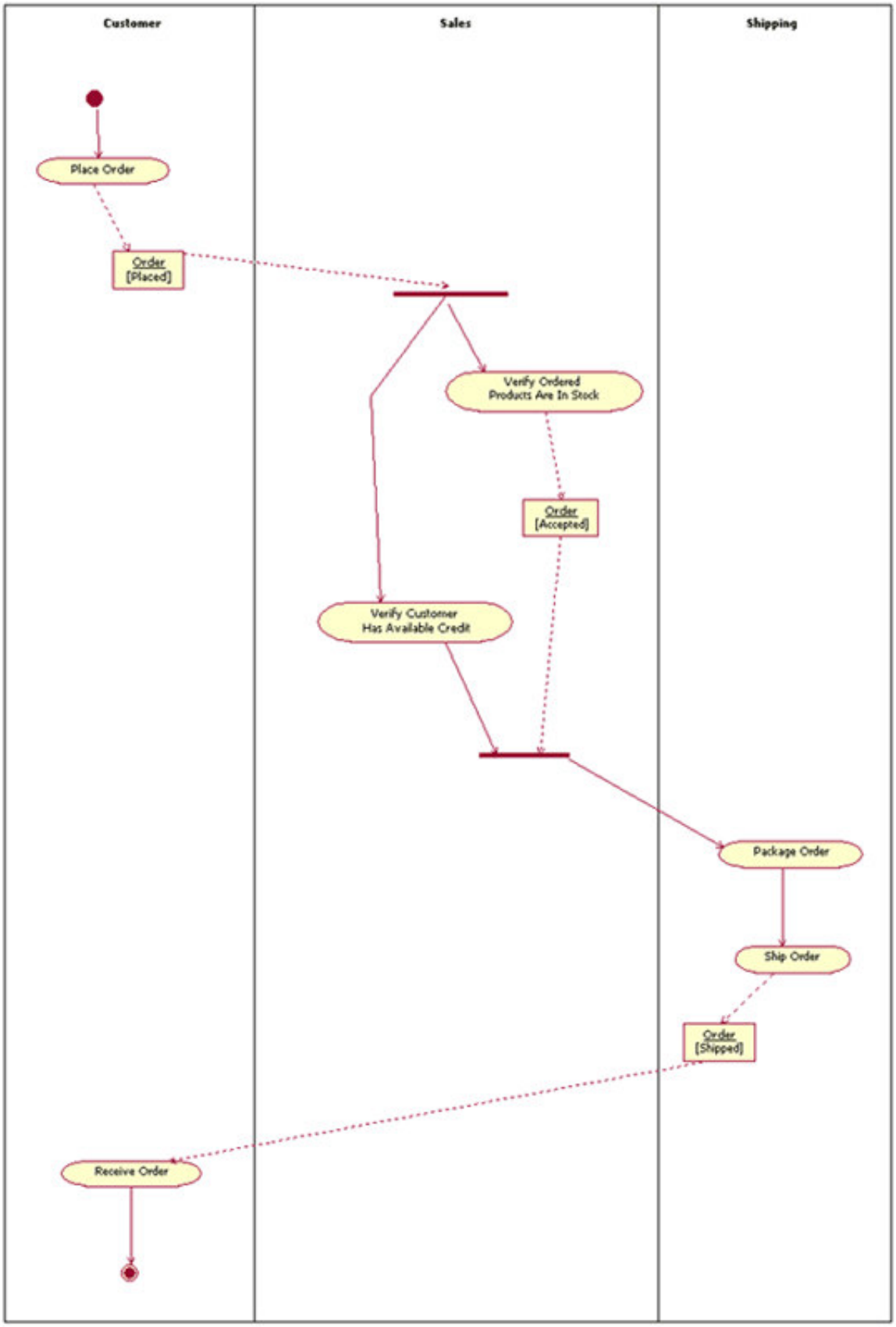
Customer
Sales
Shipping
Place Order
Order
[Placed]
Verify Ordered
Products Are In Stock
Order
[Accepted]
Verify Customer
Has Available Credit
Package Order
Ship Order
Order
[Shipped]
Receive Order

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

Haaris Infotech

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.

Introduction to Sequence Diagram

Level: Introductory

The diagram's purpose

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. Much like the class diagram, developers typically think sequence diagrams were meant exclusively for them. However, an organization's business staff can find sequence diagrams useful to communicate how the business currently works by showing how various business objects interact. Besides documenting an organization's current affairs, a business-level sequence diagram can be used as a requirements document to communicate requirements for a future system implementation. During the requirements phase of a project, analysts can take use cases to the next level by providing a more formal level of refinement. When that occurs, use cases are often refined into one or more sequence diagrams.

An organization's technical staff can find sequence diagrams useful in documenting how a future system should behave. During the design phase, architects and developers can use the diagram to force out the system's object interactions, thus fleshing out overall system design.

One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams. In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an

existing (call it "legacy") system currently interact. This documentation is very useful when transitioning a system to another person or organization.

The notation

Since this is the first article in my UML diagram series that is based on UML 2, we need to first discuss an addition to the notation in UML 2 diagrams, namely a notation element called a frame. The frame element is used as a basis for many other diagram elements in UML 2, but the first place most people will encounter a frame element is as the graphical boundary of a diagram. A frame element provides a consistent place for a diagram's label, while providing a graphical boundary for the diagram. The frame element is optional in UML diagrams; as you can see in Figures 1 and 2, the diagram's label is placed in the top left corner in what I'll call the frame's "namebox," a sort of dog-eared rectangle, and the actual UML diagram is defined within the body of the larger enclosing rectangle.

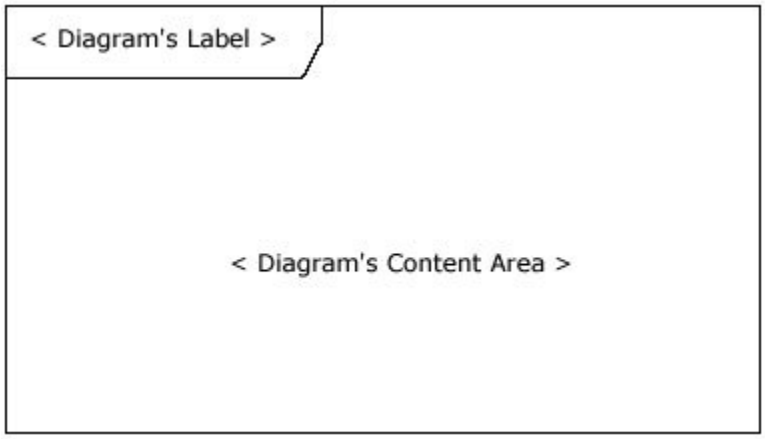

Figure 1: An empty UML 2 frame element

In addition to providing a visual border, the frame element also has an important functional use in diagrams depicting interactions, such as the sequence diagram. On sequence diagrams incoming and outgoing messages (a.k.a. interactions) for a sequence can be modeled by connecting the messages to the border of the frame element (as seen in Figure 2). This will be covered in more detail in the "Beyond the basics" section below.

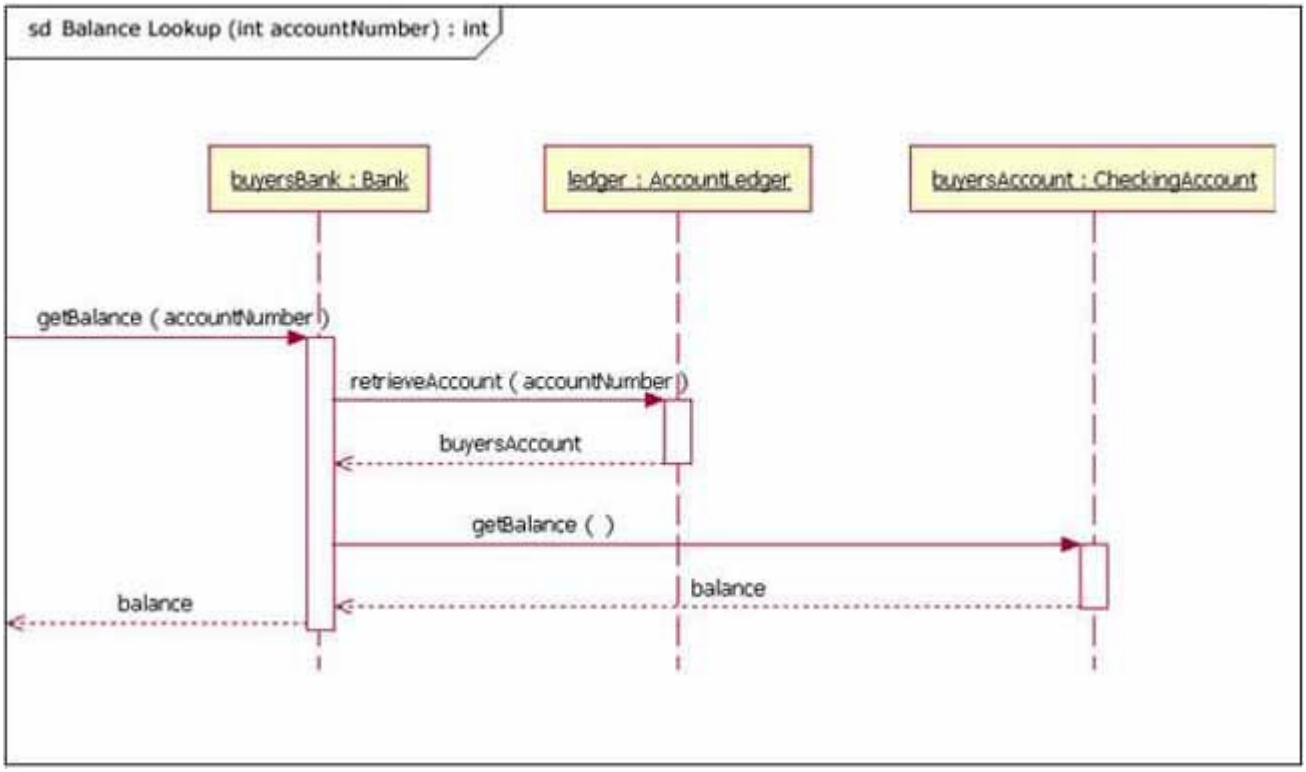

Figure 2: A sequence diagram that has incoming and outgoing messages

Notice that in Figure 2 the diagram's label begins with the letters "sd," for Sequence Diagram. When using a frame element to enclose a diagram, the diagram's label needs to follow the format of:

```
Diagram Type Diagram Name
```

The UML specification provides specific text values for diagram types (e.g., sd = Sequence Diagram, activity = Activity Diagram, and use case = Use Case Diagram).

The basics

The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on messages themselves and more on the order in which messages occur; nevertheless, most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

Lifelines

When drawing a sequence diagram, lifeline notation elements are placed across the top of the diagram. Lifelines represent either roles or object instances that participate in the sequence being modeled.[1] Lifelines are drawn as a box with a dashed line descending from the center of the bottom edge (Figure 3). The lifeline's name is placed inside the box.

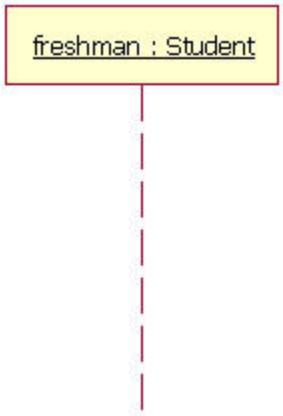

Figure 3: An example of the Student class used in a lifeline whose instance name is freshman

The UML standard for naming a lifeline follows the format of:

```
Instance Name : Class Name
```

In the example shown in Figure 3, the lifeline represents an instance of the class Student, whose instance name is freshman. Note that, here, the lifeline name is underlined. When an underline is used, it means that the lifeline represents a specific instance of a class in a sequence diagram, and not a particular kind of instance (i.e., a role). In a future article we'll look at structure modeling. For now, just observe that sequence diagrams may include roles (such as *buyer* and *seller*) without specifying who plays those roles (such as **Bill** and **Fred**). This allows diagram reuse in different contexts. Simply put, instance names in sequence diagrams are underlined; roles names are not.

Our example lifeline in Figure 3 is a named object, but not all lifelines represent named objects. Instead a lifeline can be used to represent an anonymous or unnamed instance. When modeling an unnamed instance on a sequence diagram, the lifeline's name follows the same pattern as a named instance; but instead of providing an instance name, that portion of the lifeline's name is left blank. Again referring to Figure 3, if the lifeline is representing an anonymous instance of the Student class, the lifeline would be: " Student." Also, because sequence diagrams are used during the design phase of projects, it is completely legitimate to have an object whose type is unspecified: for example, "freshman."

Messages

The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability. Subsequent messages are then added to the diagram slightly lower then the previous message.

To show an object (i.e., lifeline) sending a message to another object, you draw a line to the receiving object with a solid arrowhead (if a synchronous call operation) or with a stick arrowhead (if an asynchronous signal). The message/method name is placed above the arrowed line. The message that is being sent to the receiving object represents an operation/method that the receiving object's class implements. In the example in Figure 4, the analyst object makes a call to the system object which is an instance of the ReportingSystem class. The analyst object is calling the system object's getAvailableReports method. The system object then calls the getSecurityClearance method with the argument of userId on the secSystem object, which is of the class type SecuritySystem.[2]

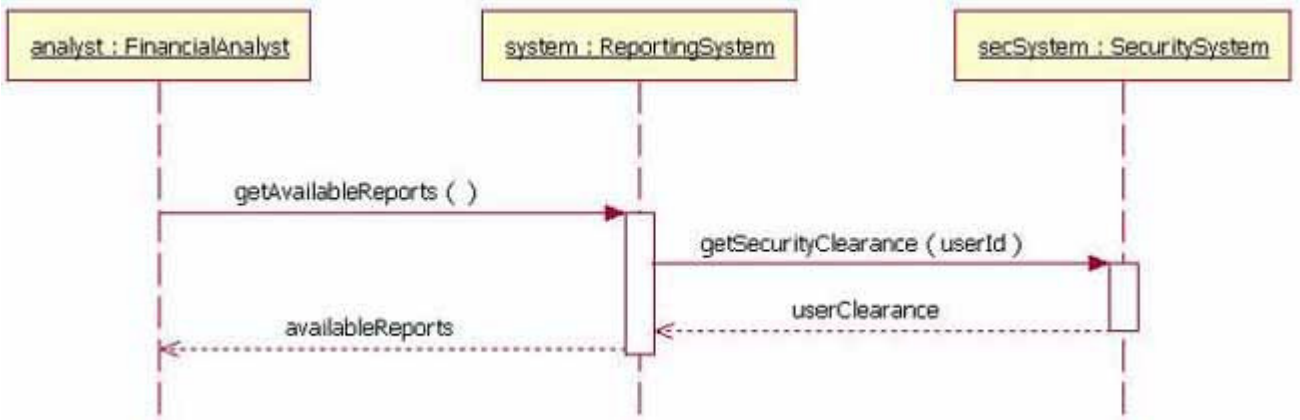

Figure 4: An example of messages being sent between objects

Besides just showing message calls on the sequence diagram, the Figure 4 diagram includes return messages. These return messages are optional; a return message is drawn as a dotted line with an open arrowhead back to the originating lifeline, and above this dotted line you place the return value from the operation. In Figure 4 the secSystem object returns userClearance to the system object when the getSecurityClearance method is called. The system object returns availableReports when the getAvailableReports method is called.

Again, the return messages are an optional part of a sequence diagram. The use of return messages depends on the level of detail/abstraction that is being modeled. Return messages are useful if finer detail is required; otherwise, the invocation message is sufficient. I personally like to include return messages whenever a value will be returned, because I find the extra details make a sequence diagram easier to read.

When modeling a sequence diagram, there will be times that an object will need to send a message to itself. When does an object call itself? A purist would argue that an object should never send a message to itself. However, modeling an object sending a message to itself can be useful in some cases. For example, Figure 5 is an improved version of Figure 4. The Figure 5 version shows the system object calling its determineAvailableReports method. By showing the system sending itself the message "determineAvailableReports," the model draws attention to the fact that this processing takes place in the system object.

To draw an object calling itself, you draw a message as you would normally, but instead of connecting it to another object, you connect the message back to the object itself.

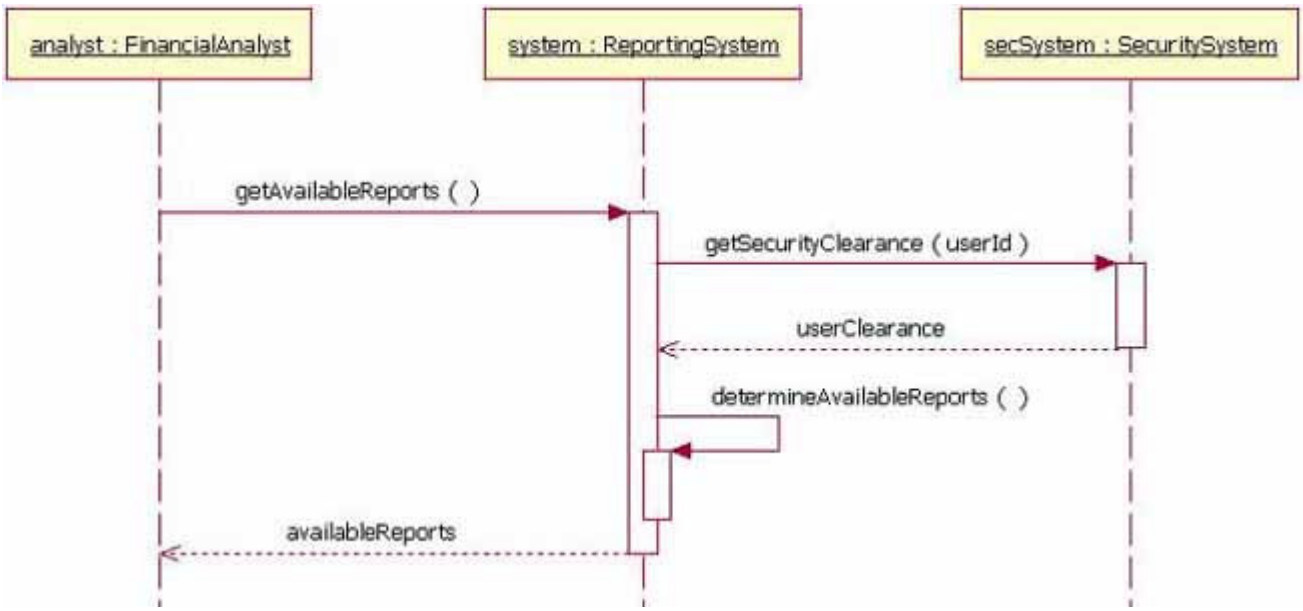

Figure 5: The system object calling its determineAvailableReports method

The example messages in Figure 5 show synchronous messages; however, in sequence diagrams you can model asynchronous messages, too. An asynchronous message is drawn similar to a synchronous one, but the message's line is drawn with a stick arrowhead, as shown in Figure 6.

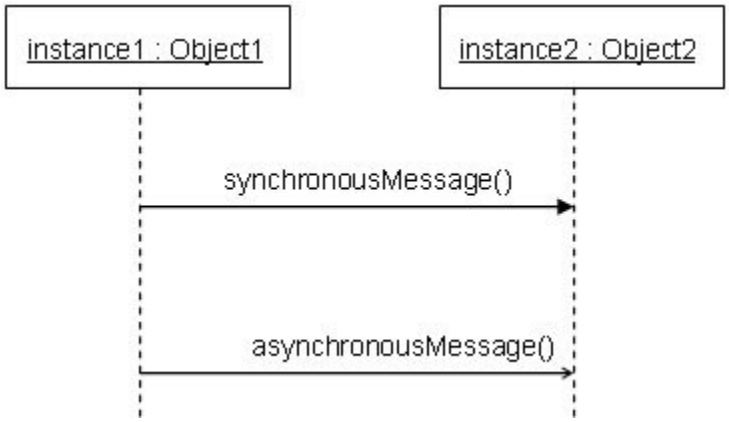

Figure 6: A sequence diagram fragment showing an asynchronous message being sent to instance2

Guards

When modeling object interactions, there will be times when a condition must be met for a message to be sent to the object. Guards are used throughout UML diagrams to control flow. Here, I will discuss guards in both UML 1.x as well as UML 2.0. In UML 1.x, a guard could only be assigned to a single message. To draw a guard on a sequence diagram in UML 1.x, you placed the guard element above the message line being guarded and in front of the message name. Figure 7 shows a fragment of a sequence diagram with a guard on the message addStudent method.

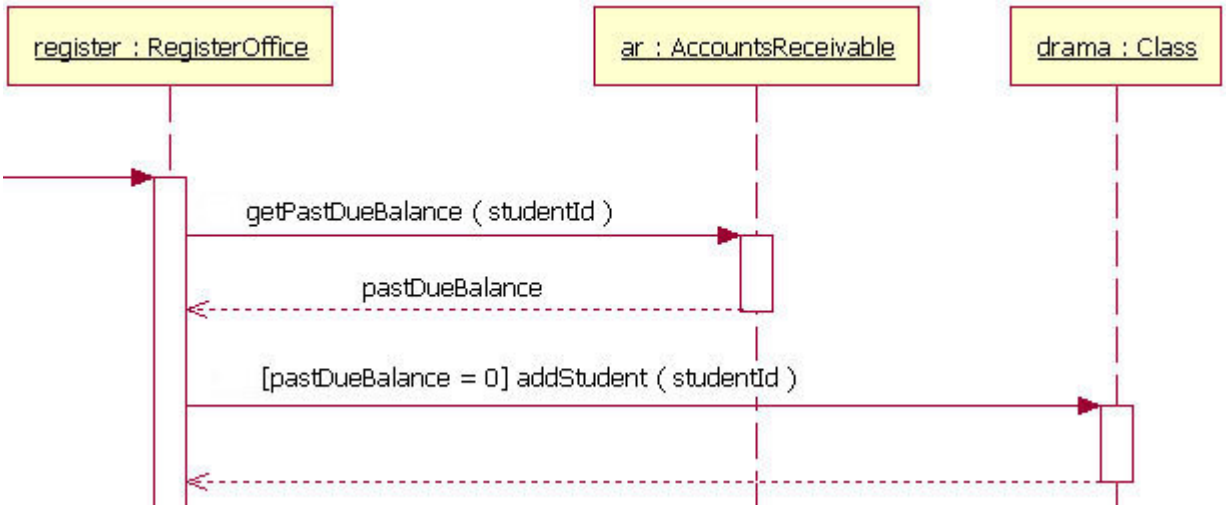

Figure 7: A segment of a UML 1.x sequence diagram in which the addStudent message has a guard

In Figure 7, the guard is the text "[pastDueBalance = 0]." By having the guard on this message, the addStudent message will only be sent if the accounts receivable system returns a past due balance of zero. The notation of a guard is very simple; the format is:

> [Boolean Test]

For example,

> [pastDueBalance = 0]

Combined fragments (alternatives, options, and loops)

In most sequence diagrams, however, the UML 1.x "in-line" guard is not sufficient to handle the logic required for a sequence being modeled. This lack of functionality was a problem in UML 1.x. UML 2 has addressed this problem by removing the "in-line" guard and adding a notation element called a Combined Fragment. A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram. The UML 2 specification identifies 11 interaction types for combined fragments. Three of the eleven will be covered here in "The Basics" section, two more types will be covered in the "Beyond The Basics" section, and the remaining six I will leave to be covered in another article. (Hey, this is an article, not a book. I want you to finish this piece in one day!)

Alternatives

Alternatives are used to designate a mutually exclusive choice between two or more message sequences.[3] Alternatives allow the modeling of the classic "if then else" logic (e.g., **if** I buy three items, **then** I get 20% off my purchase; **else** I get 10% off my purchase).

As you will notice in Figure 8, an alternative combination fragment element is drawn using a frame. The word "alt" is placed inside the frame's namebox. The larger rectangle is then divided into what UML 2 calls operands.[4] Operands are separated by a dashed line. Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand on top of a lifeline.[5] If an operand's guard equates to "true," then that operand is the operand to follow.

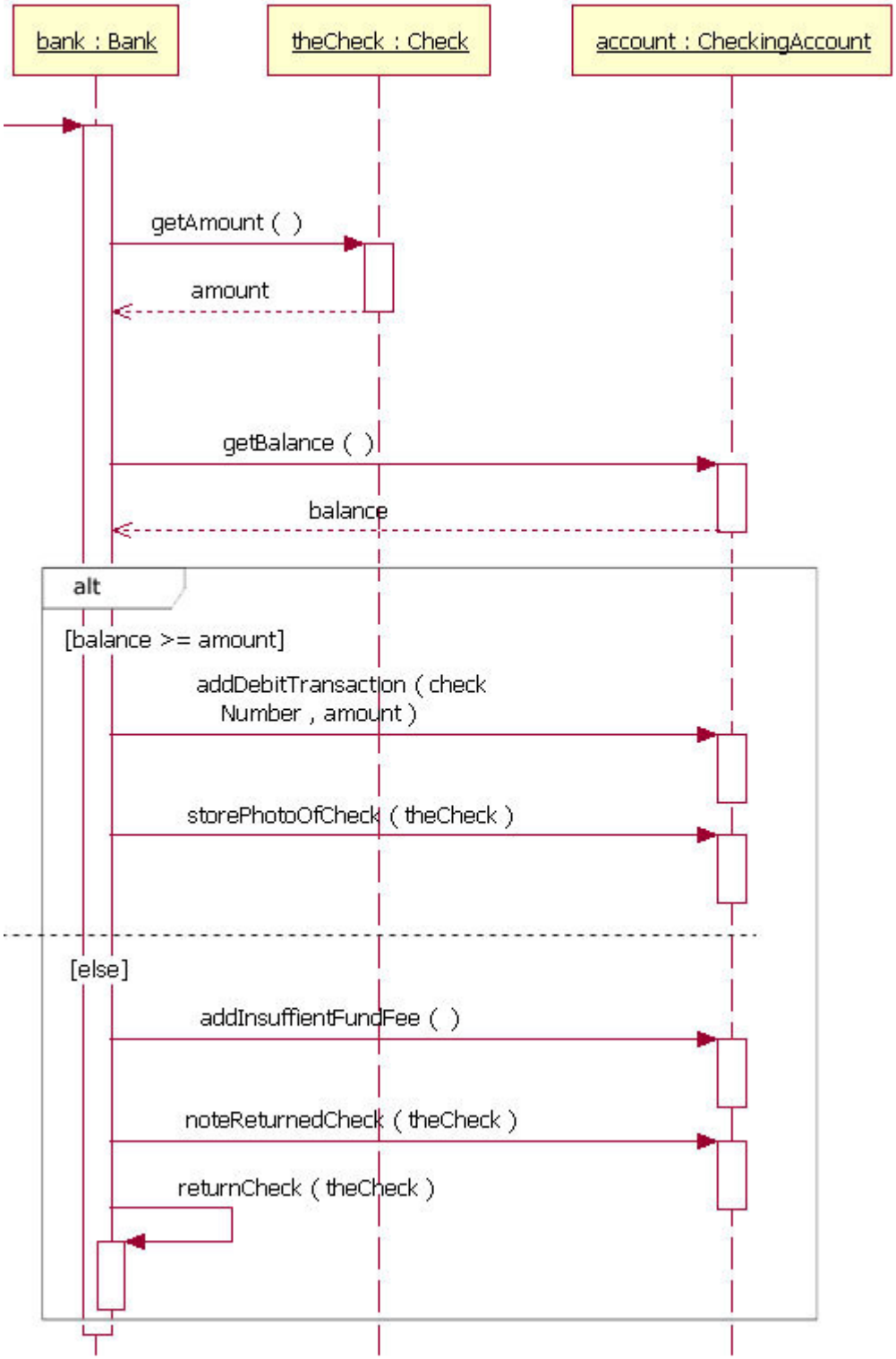

Figure 8: A sequence diagram fragment that contains an alternative combination fragment

As an example to show how an alternative combination fragment is read, Figure 8 shows the sequence starting at the top, with the bank object getting the check's amount and the account's balance. At this point in the sequence the alternative combination fragment takes over. Because of the guard "[balance >= amount]," if the account's

balance is greater than or equal to the amount, then the sequence continues with the bank object sending the addDebitTransaction and storePhotoOfCheck messages to the account object. However, if the balance is not greater than or equal to the amount, then the sequence proceeds with the bank object sending the addInsuffientFundFee and noteReturnedCheck message to the account object and the returnCheck message to itself. The second sequence is called when the balance is not greater than or equal to the amount because of the "[else]" guard. In alternative combination fragments, the "[else]" guard is not required; and if an operand does not have an explicit guard on it, then the "[else]" guard is to be assumed.

Alternative combination fragments are not limited to simple "if then else" tests. There can be as many alternative paths as are needed. If more alternatives are needed, all you must do is add an operand to the rectangle with that sequence's guard and messages.

Option

The option combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. An option is used to model a simple "if then" statement (i.e., if there are fewer than five donuts on the shelf, then make two dozen more donuts).

The option combination fragment notation is similar to the alternation combination fragment, except that it only has one operand and there never can be an "else" guard (it just does not make sense here). To draw an option combination you draw a frame. The text "opt" is placed inside the frame's namebox, and in the frame's content area the option's guard is placed towards the top left corner on top of a lifeline. Then the option's sequence of messages is placed in the remainder of the frame's content area. These elements are illustrated in Figure 9.

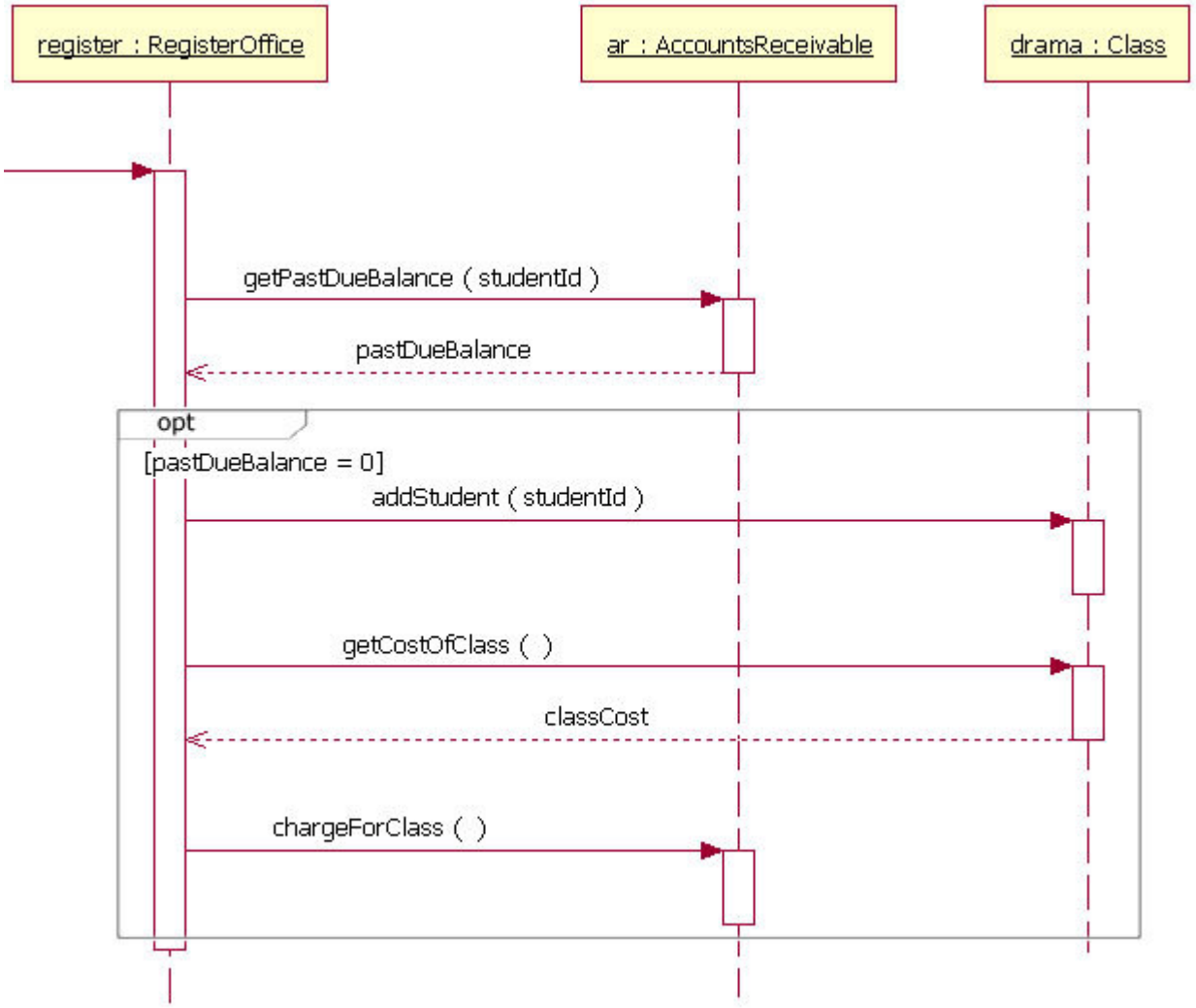

Figure 9: A sequence diagram fragment that includes an option combination fragment

Reading an option combination fragment is easy. Figure 9 is a reworking of the sequence diagram fragment in Figure 7, but this time it uses an option combination fragment because more messages need to be sent if the student's past due balance is equal to zero. According to the sequence diagram in Figure 9, if a student's past due balance equals zero, then the addStudent, getCostOfClass, and chargeForClass messages are sent. If the student's past due balance does not equal zero, then the sequence skips sending any of the messages in the option combination fragment.

The example Figure 9 sequence diagram fragment includes a guard for the option; however, the guard is not a required element. In high-level, abstract sequence diagrams you might not want to specify the condition of the option. You may simply want to indicate that the fragment is optional.

Loops

Occasionally you will need to model a repetitive sequence. In UML 2, modeling a repeating sequence has been improved with the addition of the loop combination fragment.

The loop combination fragment is very similar in appearance to the option combination fragment. You draw a frame, and in the frame's namebox the text "loop" is placed. Inside the frame's content area the loop's guard[6] is placed towards the top left corner, on top of a lifeline. Then the loop's sequence of messages is placed in the

remainder of the frame's content area. In a loop, a guard can have two special conditions tested against in addition to the standard Boolean test. The special guard conditions are minimum iterations written as "minint = [the number]" (e.g., "minint = 1") and maximum iterations written as "maxint = [the number]" (e.g., "maxint = 5"). With a minimum iterations guard, the loop must execute at least the number of times indicated, whereas with a maximum iterations guard the number of loop executions cannot exceed the number.

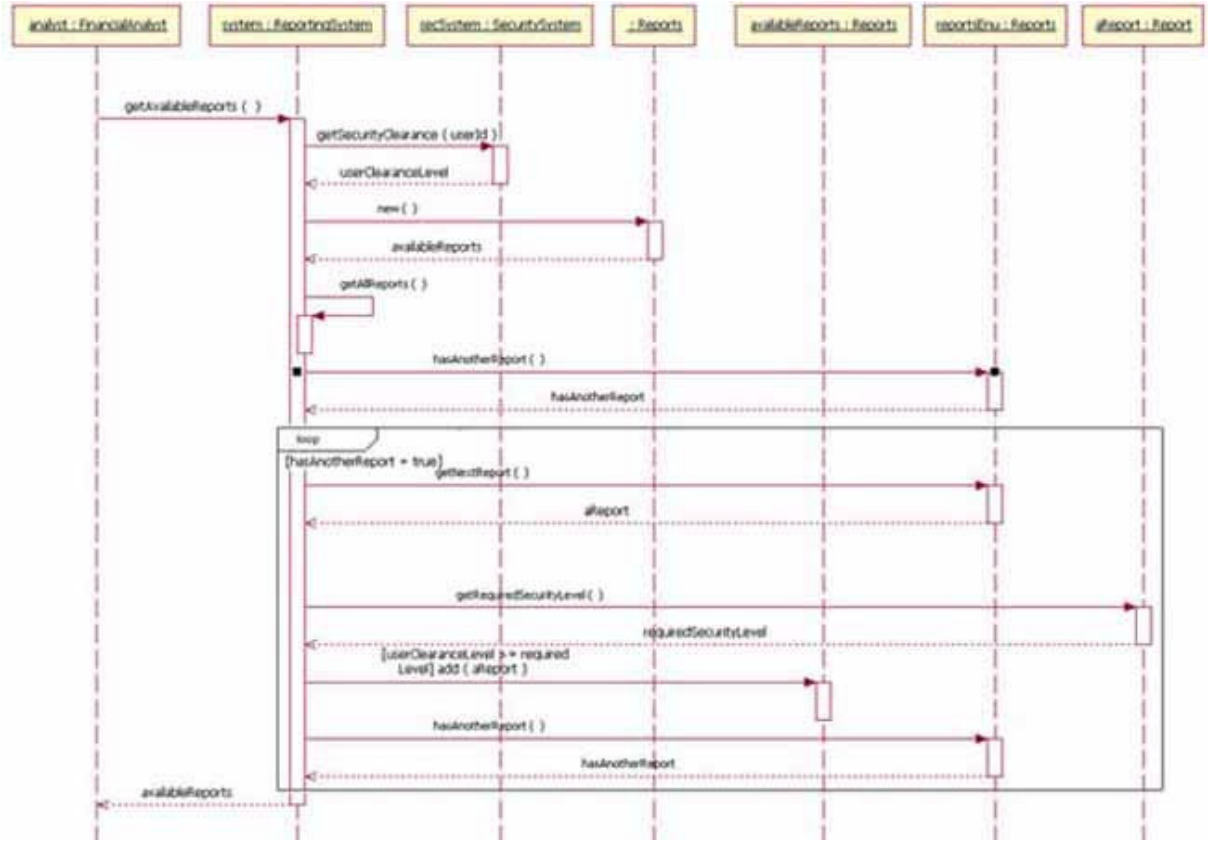

Figure 10: An example sequence diagram with a loop combination fragment
click to enlarge

The loop shown in Figure 10 executes until the reportsEnu object's hasAnotherReport message returns false. The loop in this sequence diagram uses a Boolean test to verify if the loop sequence should be run. To read this diagram, you start at the top, as normal. When you get to the loop combination fragment a test is done to see if the value hasAnotherReport equals true. If the hasAnotherReport value equals true, then the sequence goes into the loop fragment. You can then follow the messages in the loop as you would normally in a sequence diagram

Beyond the basics

I've covered the basics of the sequence diagram, which should allow you to model most of the interactions that will take place in a common system. The following section will cover more advanced notation elements that can be used in a sequence diagram.

Referencing another sequence diagram

When doing sequence diagrams, developers love to reuse existing sequence diagrams in their diagram's sequences.[7] Starting in UML 2, the "Interaction Occurrence" element was introduced. The addition of interaction occurrences is arguably the most important innovation in UML 2 interactions modeling. Interaction occurrences add the ability to compose primitive sequence diagrams into complex sequence diagrams. With these you can combine (reuse) the simpler sequences to produce more complex sequences. This means that you can abstract out a complete, and possibly complex, sequence as a single conceptual unit.

Haaris Infotech

An interaction occurrence element is drawn using a frame. The text "ref" is placed inside the frame's namebox, and the name of the sequence diagram being referenced is placed inside the frame's content area along with any parameters to the sequence diagram. The notation of the referenced sequence diagram's name follows the pattern of:

```
sequence diagram name[(arguments)] [: return value]
```

Two examples:

1. Retrieve Borrower Credit Report(ssn) : borrowerCreditReport

or

2. Process Credit Card(name, number, expirationDate, amount : 100)

In example 1, the syntax calls the sequence diagram called Retrieve Borrower Credit Report and passes it the parameter ssn. The Retreive Borrower Credit Report sequence returns the variable borrowerCreditReport.

In example 2, the syntax calls the sequence diagram called Process Credit Card and passes it the parameters of name, number, expiration date, and amount. However, in example 2 the amount parameter will be a value of 100. And since example 2 does not have a return value labeled, the sequence does not return a value (presumably, the sequence being modeled does not need the return value).

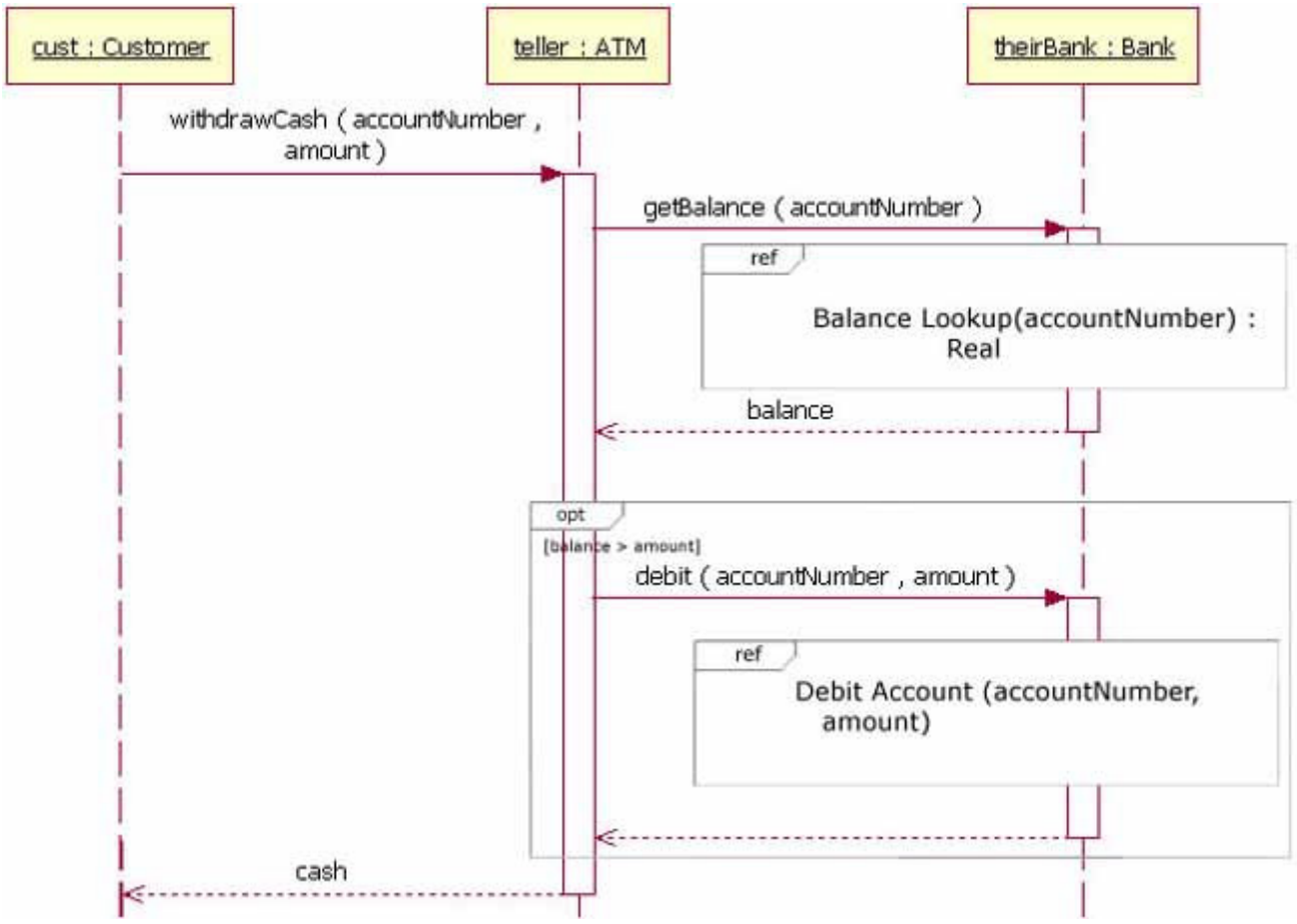

Figure 11: A sequence diagram that references two different sequence diagrams

Figure 11 shows a sequence diagram that references the sequence diagrams "Balance Lookup" and "Debit Account." The sequence starts at the top left, with the customer sending a message to the teller object. The teller object sends a message to the theirBank object. At that point, the Balance Lookup sequence diagram is called, with the accountNumber passed as a parameter. The Balance Lookup sequence diagram returns the balance variable. Then the option combination fragment's guard condition is checked to verify the balance is greater then the amount variable. In cases where the balance is greater than the amount, the Debit Account sequence diagram is called, passing it the accountNumber and the amount as parameters. After that sequence is complete, the withdrawCash message returns cash to the customer.

It is important to notice in Figure 11 that the lifeline of theirBank is hidden by the interaction occurrence Balance Lookup. Because the interaction occurrence hides the lifeline, that means that the theirBank lifeline is referenced in the "Balance Lookup" sequence diagram. In addition to hiding the lifeline in the interaction occurrence, UML 2 also specifies that the lifeline must have the same theirBank in its own "Balance Lookup" sequence.

There will be times when you model sequence diagrams that an interaction occurrence will overlap lifelines that are *not* referenced in the interaction occurrence. In such cases the lifeline is shown as a normal lifeline and is not hidden by the overlapping interaction occurrence.

In Figure 11, the sequence references the "Balance Lookup" sequence diagram. The "Balance Lookup" sequence diagram is shown in Figure 12. Because the example sequence has parameters and a return value, its label -- located in the diagram's namebox--follows a specific pattern:

```
Diagram Type Diagram Name [(Parameter Type : Parameter Name)] :
```

```
[: Return Value Type]
```

Two examples:

1. SD Balance Lookup(Integer : accountNumber) : Real

or

2. SD Available Reports(Financial Analyst : analyst) : Reports

Figure 12 illustrates example 1, in which the Balance Lookup sequence uses parameter accountNumber as a variable in the sequence, and the sequence diagram shows a Real object being returned. In cases such as this, where the sequence returns an object, the object being returned is given the instance name of the sequence diagram.

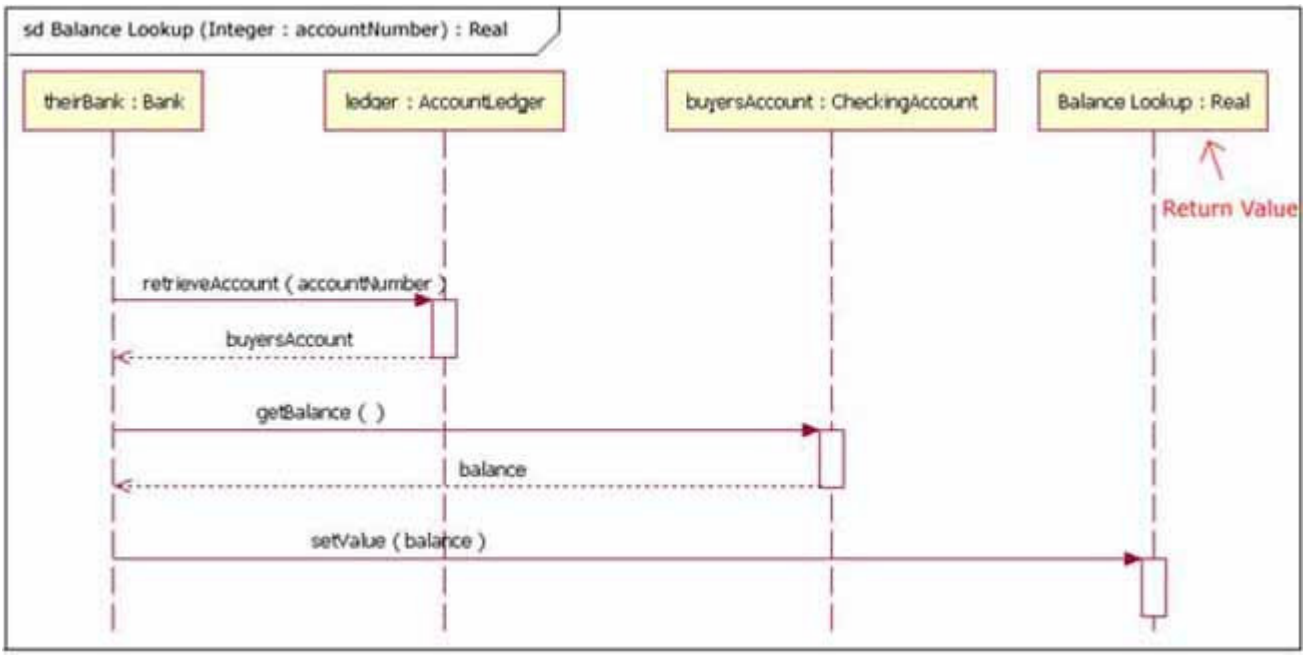

Figure 12: A sequence diagram that takes the parameter of accountNumber and returns a Real object

Figure 13 illustrates example 2, in which a sequence takes a parameter and returns an object. However, in Figure 13 the parameter is used in the sequence's interaction.

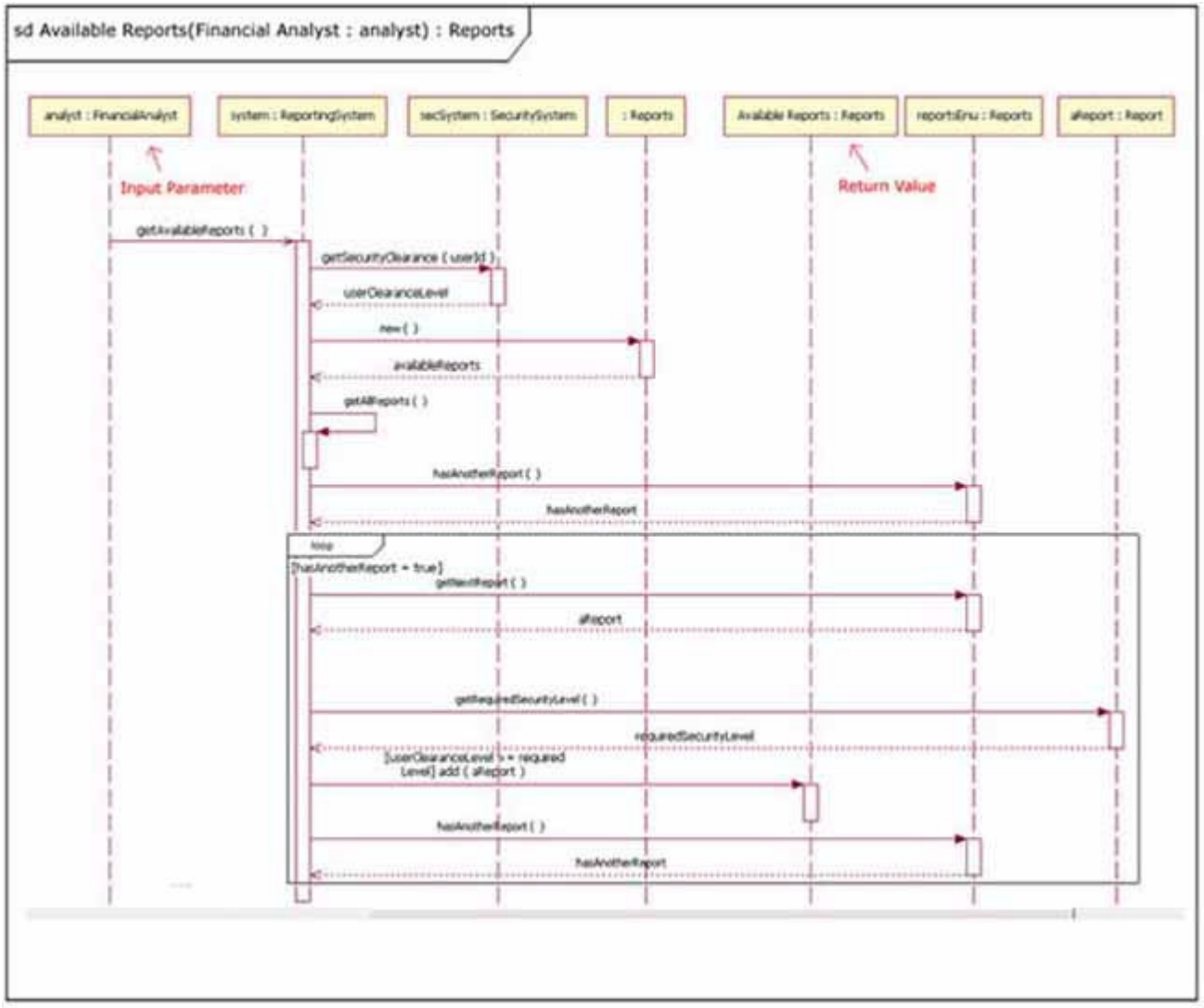

Figure 13: A sequence diagram that uses its parameter in its interaction and returns a Reports object
Click to enlarge

Gates

The previous section showed how to reference another sequence diagram by passing information through parameters and return values. However, there is another way to pass information between sequence diagrams. Gates can be an easy way to model the passing of information between a sequence diagram and its context. A gate is merely a message that is illustrated with one end connected to the sequence diagram's frame's edge and the other end connected to a lifeline. A reworking of Figures 11 and 12 using gates can be seen in Figures 14 and 15. The example diagram in Figure 15 has an entry gate called getBalance that takes the parameter of accountNumber. The getBalance message is an entry gate, because it is the arrowed line that is connected to the diagram's frame with the arrowhead connected to a lifeline. The sequence diagram also has an exit gate that returns the balance variable. The exit gate is known, because it's a return message that is connected from a lifeline to the diagram's frame with the arrowhead connected to the frame.

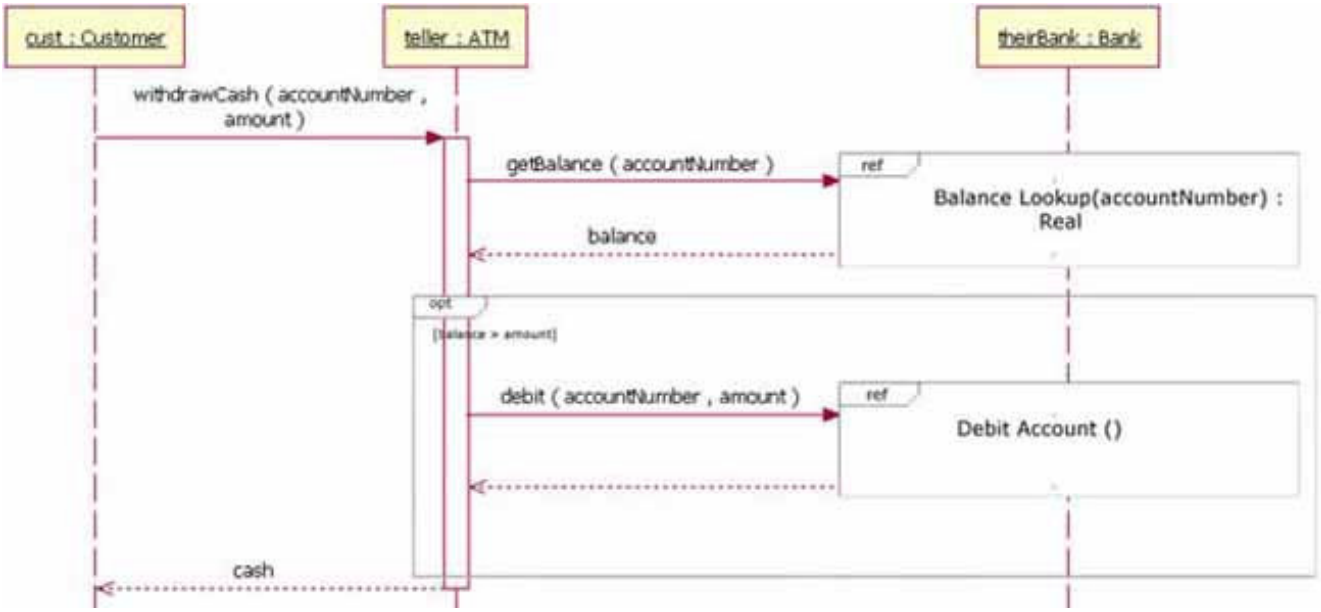

Figure 14: A reworking of Figure 11, using gates this time

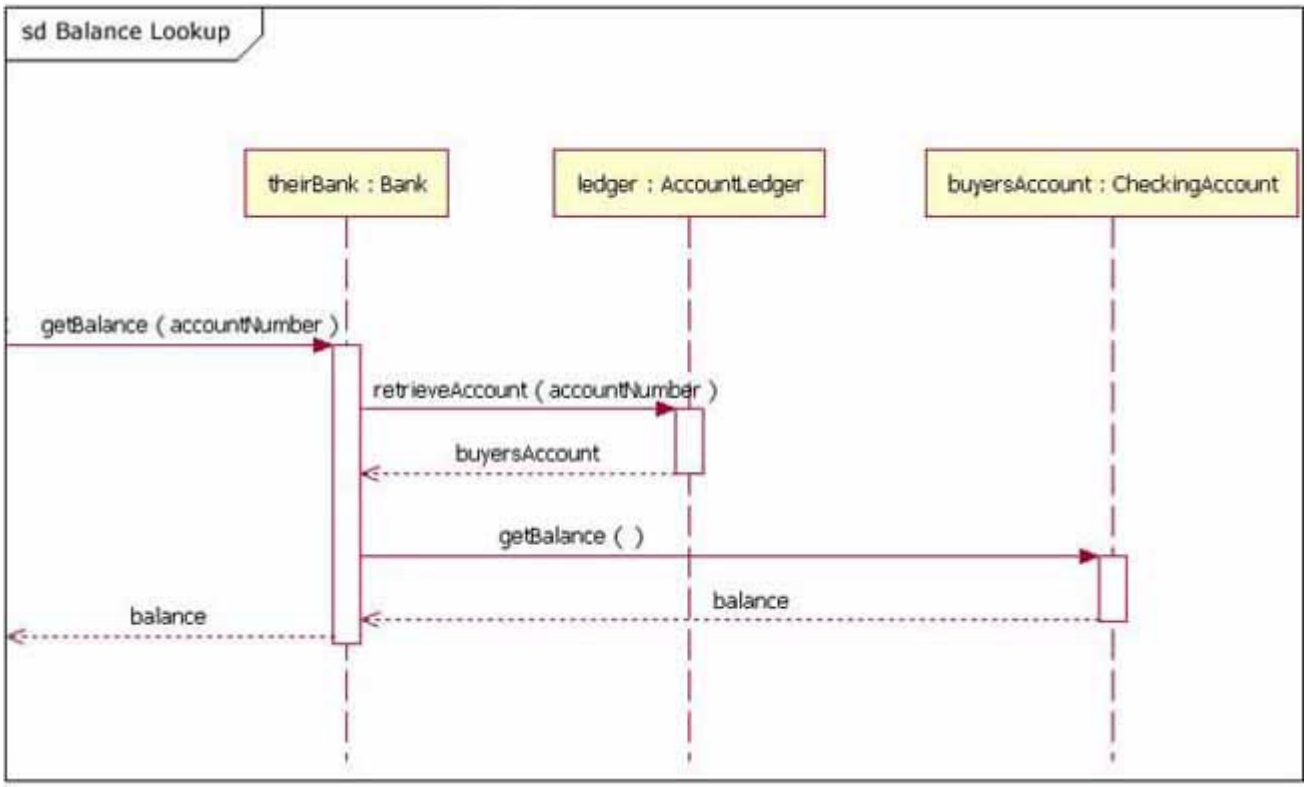

Figure 15: A reworking of Figure 12, using gates this time

Combined fragments (break and parallel)

In the "basics" section presented earlier in this paper, I covered the combined fragments known as "alternative," "option," and "loop." These three combined fragments are the ones most people will use the most. However, there are two other combined fragments that a large share of people will find useful – break and parallel.

Break

The break combined fragment is almost identical in every way to the option combined fragment, with two exceptions. First, a break's frame has a namebox with the text "break" instead of "option." Second, when a break combined fragment's message is to be executed, the enclosing interaction's remainder messages will not be executed because the sequence breaks out of the enclosing interaction. In this way the break combined fragment is much like the break keyword in a programming language like C++ or Java.

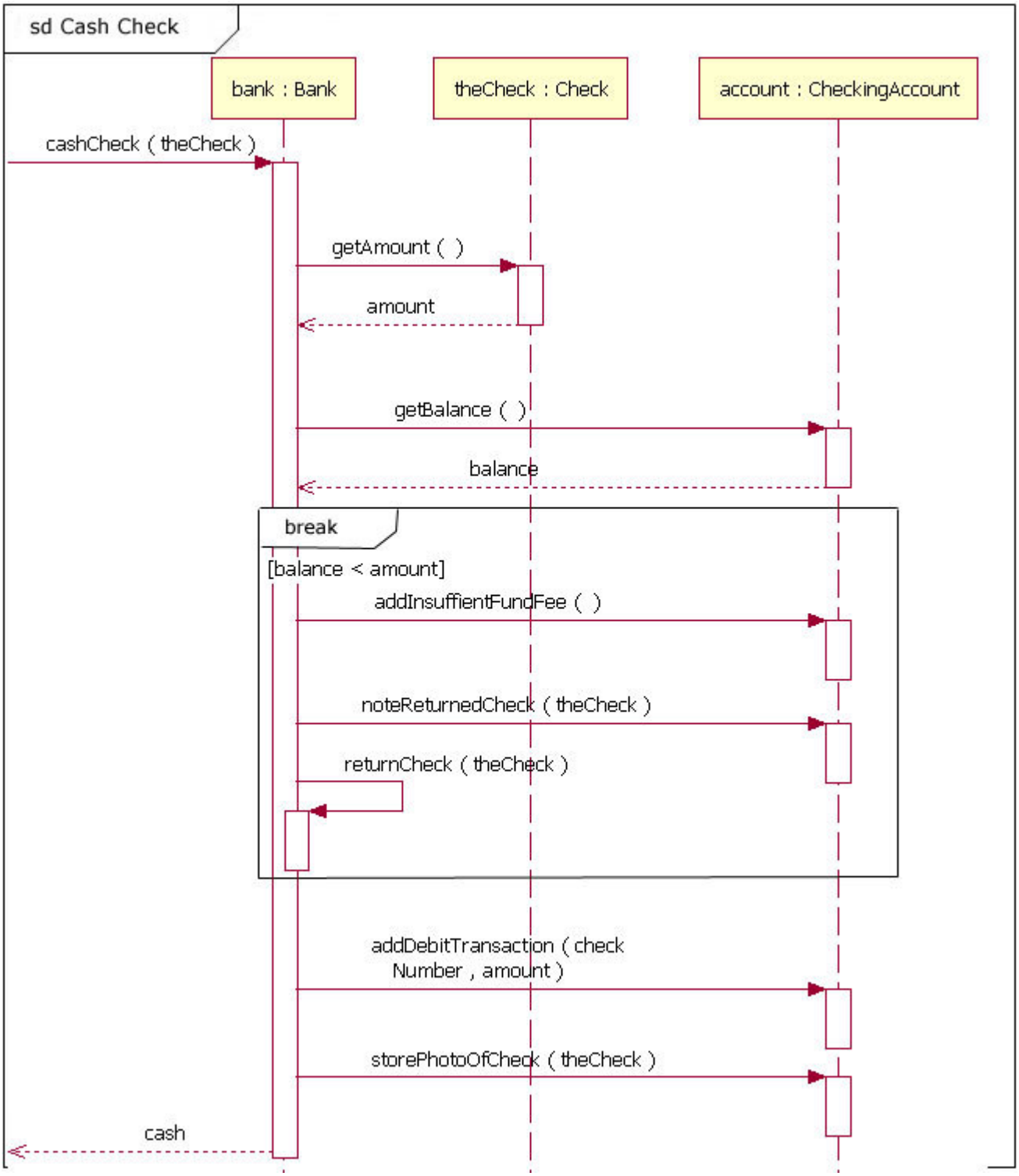

Figure 16: A reworking of the sequence diagram fragment from Figure 8, with the fragment using a break instead of an alternative

Breaks are most commonly used to model exception handling. Figure 16 is a reworking of Figure 8, but this time Figure 16 uses a break combination fragment because it treats the balance < amount condition as an exception instead of as an alternative flow. To read Figure 16, you start at the top left corner of the sequence and read down.

When the sequence gets to the return value "balance," it checks to see if the balance is less than the amount. If the balance is not less than the amount, the next message sent is the addDebitTransaction message, and the sequence continues as normal. However, in cases where the balance is less than the amount, then the sequence enters the break combination fragment and its messages are sent. Once all the messages in the break combination have been sent, the sequence exits without sending any of the remaining messages (e.g., addDebitTransaction).

An important thing to note about breaks is that they only cause the exiting of an enclosing interaction's sequence and not necessarily the complete sequence depicted in the diagram. In cases where a break combination is part of an alternative or a loop, then only the alternative or loop is exited.

Parallel

Today's modern computer systems are advancing in complexity and at times perform concurrent tasks. When the processing time required to complete portions of a complex task is longer than desired, some systems handle parts of the processing in parallel. The parallel combination fragment element needs to be used when creating a sequence diagram that shows parallel processing activities.

The parallel combination fragment is drawn using a frame, and you place the text "par" in the frame's namebox. You then break up the frame's content section into horizontal operands separated by a dashed line. Each operand in the frame represents a thread of execution done in parallel.

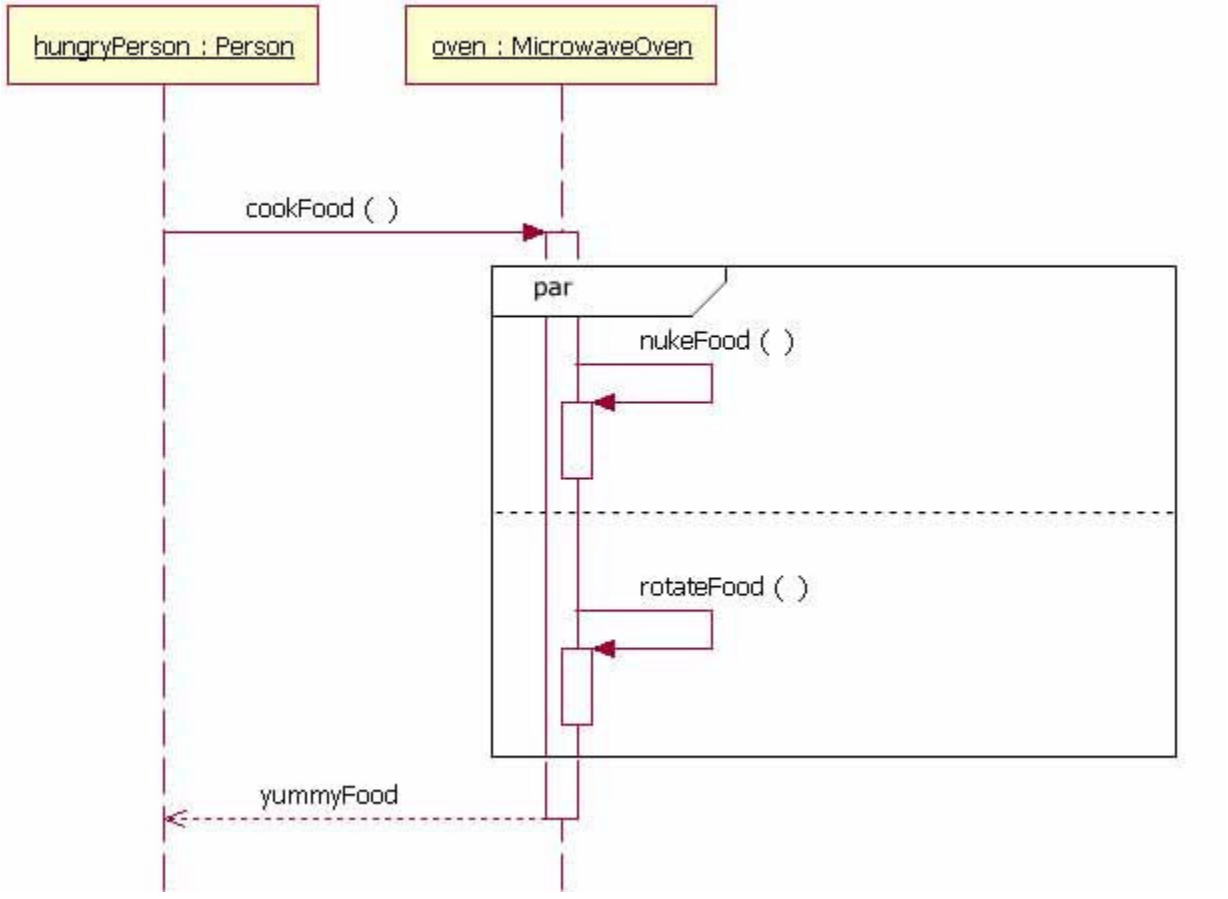

Figure 17: A microwave is an example of an object that does two tasks in parallel

While Figure 17 may not illustrate the best computer system example of an object doing activities in parallel, it offers an easy-to-understand example of a sequence with parallel activities. The sequence goes like this: A hungryPerson sends the cookFood message to the oven object. When the oven object receives that message, it

sends two messages to itself at the same time (nukeFood and rotateFood). After both of these messages are done, the hungryPerson object is returned yummyFood from the oven object.

Conclusion

The sequence diagram is a good diagram to use to document a system's requirements and to flush out a system's design. The reason the sequence diagram is so useful is because it shows the interaction logic between the objects in the system in the time order that the interactions take place.

Introduction to Class Diagram

Level: Introductory

The yin and yang of UML 2

In UML 2 there are two basic categories of diagrams: structure diagrams and behavior diagrams. Every UML diagram belongs to one these two diagram categories. The purpose of structure diagrams is to show the static structure of the system being modeled. They include the class, component, and or object diagrams. Behavioral diagrams, on the other hand, show the dynamic behavior between the objects in the system, including things like their methods, collaborations, and activities. Example behavior diagrams are activity, use case, and sequence diagrams.

Structure diagrams in general

As I have said, structure diagrams show the static structure of the system being modeled. focusing on the elements of a system, irrespective of time. Static structure is conveyed by showing the types and their instances in the system. Besides showing system types and their instances, structure diagrams also show at least some of the relationships among and between these elements and potentially even show their internal structure.

Structure diagrams are useful throughout the software lifecycle for a variety of team members. In general, these diagrams allow for design validation and design communication between individuals and teams. For example, business analysts can use class or object diagrams to model a business's current assets and resources, such as account ledgers, products, or geographic hierarchy. Architects can use the component and deployment diagrams to test/verify that their design is sound. Developers can use class diagrams to design and document the system's coded (or soon-to-be-coded) classes.

The class diagram in particular

UML 2 considers structure diagrams as a classification; there is no diagram itself called a "Structure Diagram." However, the class diagram offers a prime example of the structure diagram type, and provides us with an initial set of notation elements that all other structure diagrams use. And because the class diagram is so foundational, the remainder of this article will focus on the class diagram's notation set. By the end of this article you should have an understanding of how to draw a UML 2 class diagram and have a solid footing for understanding other structure diagrams when we cover them in later articles.

The basics

As mentioned earlier, the purpose of the class diagram is to show the types being modeled within the system. In most UML models these types include:

- a class
- an interface
- a data type
- a component.

UML uses a special name for these types: "classifiers." Generally, you can think of a classifier as a class, but technically a classifier is a more general term that refers to the other three types above as well.

Class name

The UML representation of a class is a rectangle containing three compartments stacked vertically, as shown in Figure 1. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations. When drawing a class element on a class diagram, you must use the top compartment, and the bottom two compartments are optional. (The bottom two would be unnecessary on a diagram depicting a higher level of detail in which the purpose is to show only the relationship between the classifiers.) Figure 1 shows an airline flight modeled as a UML class. As we can see, the name is *Flight*, and in the middle compartment we see that the Flight class has three attributes: flightNumber, departureTime, and flightDuration. In the bottom compartment we see that the Flight class has two operations: delayFlight and getArrivalTime.

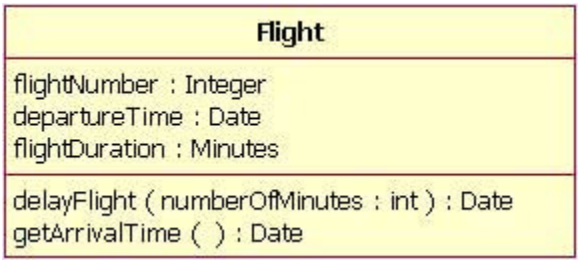

Figure 1: Class diagram for the class Flight

Class attribute list

The attribute section of a class (the middle compartment) lists each of the class's attributes on a separate line. The attribute section is optional, but when used it contains each attribute of the class displayed in a list format. The line uses the following format:

```
name : attribute type
```

```
flightNumber : Integer
```

Continuing with our Flight class example, we can describe the class's attributes with the attribute type information, as shown in Table 1.

Table 1: The Flight class's attribute names with their associated types

Attribute Name	Attribute Type
flightNumber	Integer
departureTime	Date
flightDuration	Minutes

In business class diagrams, the attribute types usually correspond to units that make sense to the likely readers of the diagram (i.e., minutes, dollars, etc.). However, a class diagram that will be used to generate code needs classes whose attribute types are limited to the types provided by the programming language, or types included in the model that will also be implemented in the system.

Sometimes it is useful to show on a class diagram that a particular attribute has a default value. (For example, in a banking account application a new bank account would start off with a zero balance.) The UML specification allows for the identification of default values in the attribute list section by using the following notation:

```
name : attribute type = default value
```

For example:

```
balance : Dollars = 0
```

Showing a default value for attributes is optional; Figure 2 shows a Bank Account class with an attribute called *balance*, which has a default value of 0.

Figure 2: A Bank Account class diagram showing the balance attribute's value defaulted to zero dollars.

Class operations list

The class's operations are documented in the third (lowest) compartment of the class diagram's rectangle, which again is optional. Like the attributes, the operations of a class are displayed in a list format, with each operation on its own line. Operations are documented using the following notation:

```
name(parameter list) : type of value returned
```

The Flight class's operations are mapped in Table 2 below.

Table 2: Flight class's operations mapped from Figure 2

Operation Name	Parameters Return		Value Type
delayFlight	**Name**	**Type**	N/A
	numberOfMinutes	Minutes	
getArrivalTime	N/A		Date

Figure 3 shows that the delayFlight operation has one input parameter -- numberOfMinutes -- of the type Minutes. However, the delayFlight operation does not have a return value.[1] When an operation has parameters, they are put inside the operation's parentheses; each parameter uses the format "parameter name : parameter type".

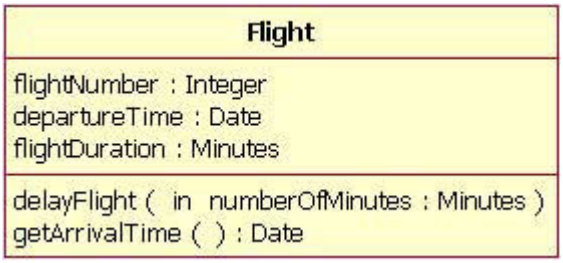

Figure 3: The Flight class operations parameters include the optional "in" marking.

When documenting an operation's parameters, you may use an optional indicator to show whether or not the parameter is input to, or output from, the operation. This optional indicator appears as an "in" or "out" as shown in the operations compartment in Figure 3. Typically, these indicators are unnecessary unless an older programming language such as Fortran will be used, in which case this information can be helpful. However, in C++ and Java, all parameters are "in" parameters and since "in" is the parameter's default type according to the UML specification, most people will leave out the input/output indicators.

Inheritance

A very important concept in object-oriented design, *inheritance*, refers to the ability of one class (child class) to *inherit* the identical functionality of another class (super class), and then add new functionality of its own. (In a very non-technical sense, imagine that I inherited my mother's general musical abilities, but in my family I'm the only one who plays electric guitar.) To model inheritance on a class diagram, a solid line is drawn from the child class (the class inheriting the behavior) with a closed, unfilled arrowhead (or triangle) pointing to the super class. Consider types of bank accounts: Figure 4 shows how both CheckingAccount and SavingsAccount classes inherit from the BankAccount class.

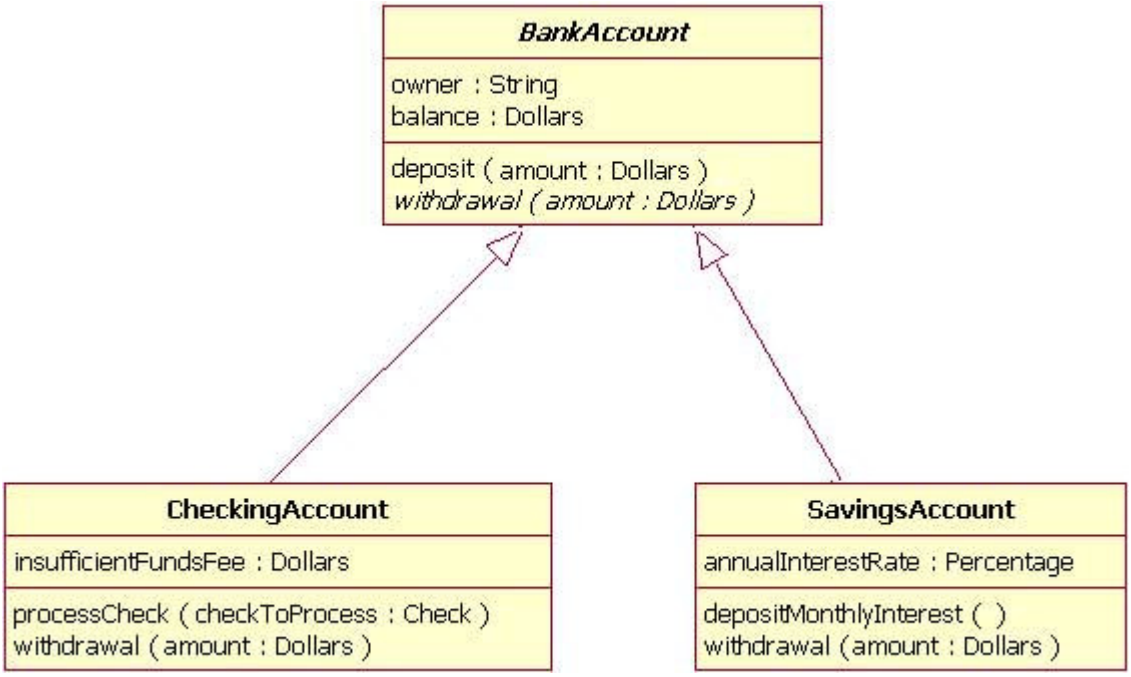

Figure 4: Inheritance is indicated by a solid line with a closed, unfilled arrowhead pointing at the super class.

In Figure 4, the inheritance relationship is drawn with separate lines for each subclass, which is the method used in IBM Rational Rose and IBM Rational XDE. However, there is an alternative way to draw inheritance called *tree notation*. You can use tree notation when there are two or more child classes, as in Figure 4, except that the inheritance lines merge together like a tree branch. Figure 5 is a redrawing of the same inheritance shown in Figure 4, but this time using tree notation.

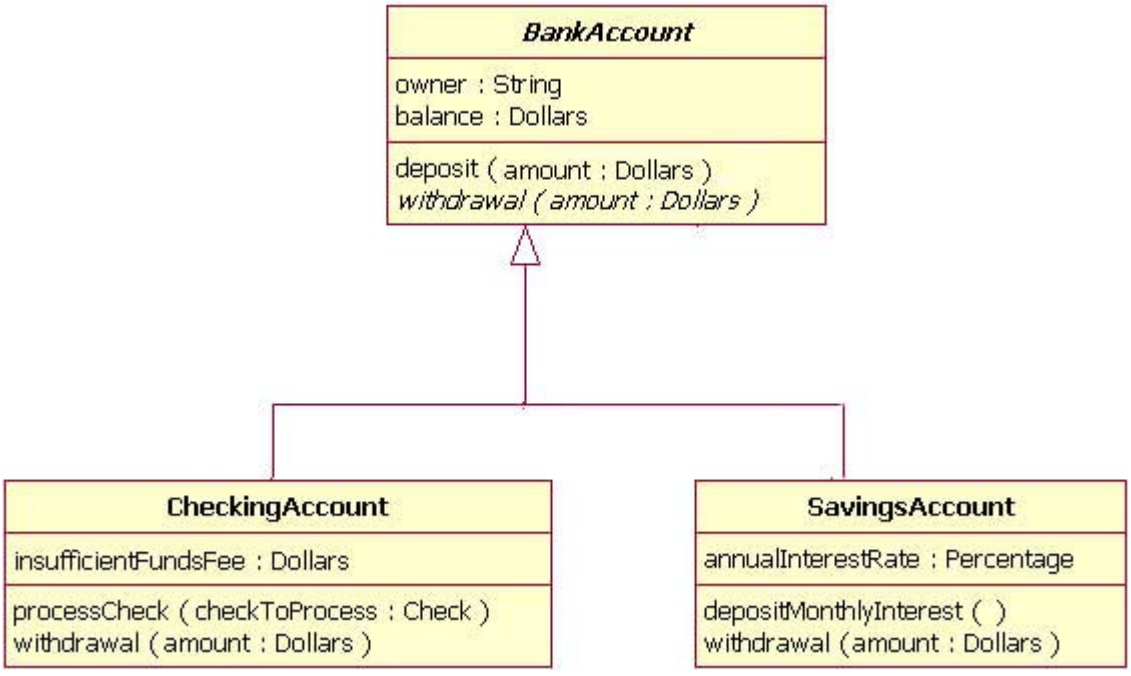

Figure 5: An example of inheritance using tree notation

Abstract classes and operations
The observant reader will notice that the diagrams in Figures 4 and 5 use italicized text for the BankAccount class name and withdrawal operation. This indicates that the BankAccount class is an abstract class and the withdrawal method is an abstract operation. In other words, the BankAccount class provides the abstract operation signature of withdrawal and the two child classes of CheckingAccount and SavingsAccount each implement their own version of that operation.

However, super classes (parent classes) do not have to be abstract classes. It is normal for a standard class to be a super class.

Associations
When you model a system, certain objects will be related to each other, and these relationships themselves need to be modeled for clarity. There are five types of associations. I will discuss two of them -- bi-directional and uni-directional associations -- in this section, and I will discuss the remaining three association types in the *Beyond the basics* section. Please note that a detailed discussion of when to use each type of association is beyond the scope of this article. Instead, I will focus on the purpose of each association type and show how the association is drawn on a class diagram.

Bi-directional (standard) association
An association is a linkage between two classes. Associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship, unless you qualify the association as some other type. Going back to our Flight example, Figure 6 shows a standard kind of association between the Flight class and the Plane class.

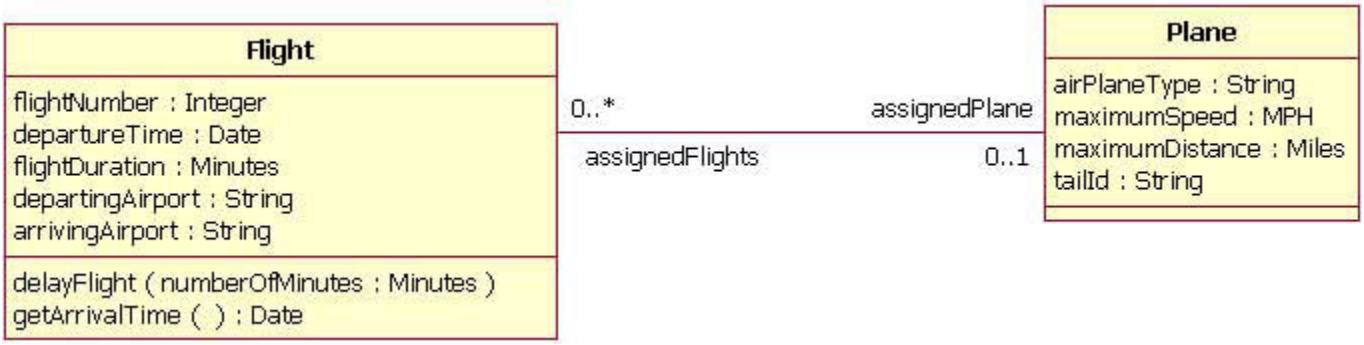

Figure 6: An example of a bi-directional association between a Flight class and a Plane class

A bi-directional association is indicated by a solid line between the two classes. At either end of the line, you place a role name and a multiplicity value. Figure 6 shows that the Flight is associated with a specific Plane, and the Flight class knows about this association. The Plane takes on the role of "assignedPlane" in this association because the role name next to the Plane class says so. The multiplicity value next to the Plane class of 0..1 means that when an instance of a Flight exists, it can either have one instance of a Plane associated with it or no Planes associated with it (i.e., maybe a plane has not yet been assigned). Figure 6 also shows that a Plane knows about its association with the Flight class. In this association, the Flight takes on the role of "assignedFlights"; the diagram in Figure 6 tells us that the Plane instance can be associated either with no flights (e.g., it's a brand new plane) or with up to an infinite number of flights (e.g., the plane has been in commission for the last five years).

For those wondering what the potential multiplicity values are for the ends of associations, Table 3 below lists some example multiplicity values along with their meanings.

Table 3: Multiplicity values and their indicators

Potential Multiplicity Values	
Indicator	Meaning
0..1	Zero or one

1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen

Uni-directional association
In a uni-directional association, two classes are related, but only one class knows that the relationship exists. Figure 7 shows an example of an overdrawn accounts report with a uni-directional association.

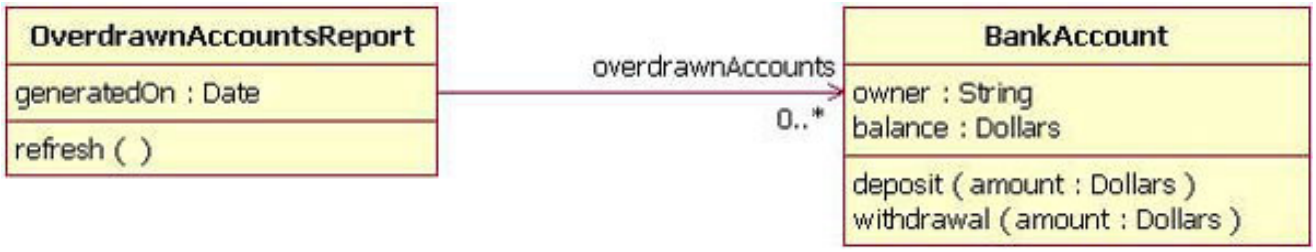

Figure 7: An example of a uni-directional association: The OverdrawnAccountsReport class knows about the BankAccount class, but the BankAccount class does not know about the association.

A uni-directional association is drawn as a solid line with an open arrowhead (not the closed arrowhead, or triangle, used to indicate inheritance) pointing to the known class. Like standard associations, the uni-directional association includes a role name and a multiplicity value, but unlike the standard bi-directional association, the uni-directional association only contains the role name and multiplicity value for the known class. In our example in Figure 7, the OverdrawnAccountsReport knows about the BankAccount class, and the BankAccount class plays the role of "overdrawnAccounts." However, unlike a standard association, the BankAccount class has no idea that it is associated with the OverdrawnAccountsReport.[2]

Packages
Inevitably, if you are modeling a large system or a large area of a business, there will be many different classifiers in your model. Managing all the classes can be a daunting task; therefore, UML provides an organizing element called a *package*. Packages enable modelers to organize the model's classifiers into namespaces, which is sort of like folders in a filing system. Dividing a system into multiple packages makes the system easier to understand, especially if each package represents a specific part of the system.[3]

There are two ways of drawing packages on diagrams. There is no rule for determining which notation to use, except to use your personal judgement regarding which is easiest to read for the class diagram you are drawing. Both ways begin with a large rectangle with a smaller rectangle (tab) above its upper left corner, as seen in Figure 8. But the modeler must decide how the package's membership is to be shown, as follows:

- If the modeler decides to show the package's members within the large rectangle, then all those members[4] need to be placed within the rectangle. Also the package's name needs to be placed in the package's smaller rectangle (as show n in Figure 8).

- If the modeler decides to show the package's members outside the large rectangle then all the members that will be shown on the diagram need to be placed outside the rectangle. To show what classifiers belong to the package, a line is drawn from each classifier to a circle that has a plus sign inside the circle attached to the package (Figure 9).

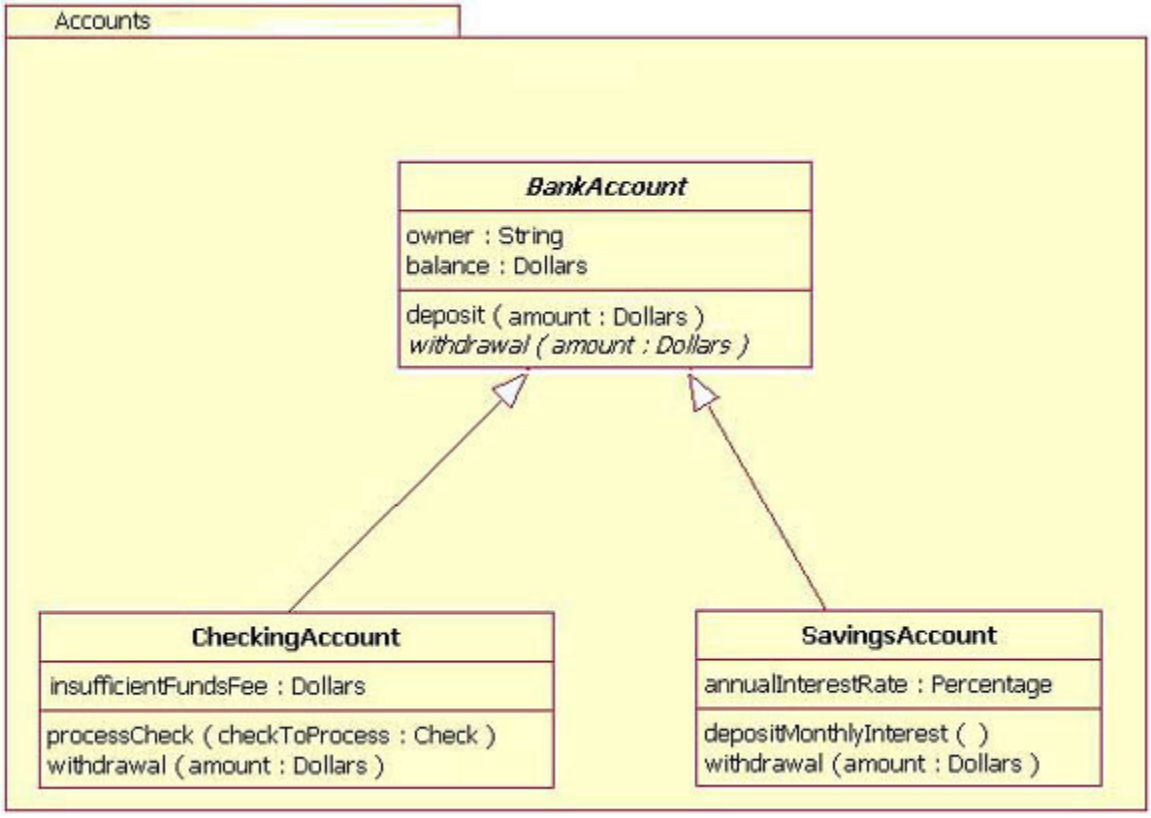

Figure 8: An example package element that shows its members inside the package's rectangle boundaries

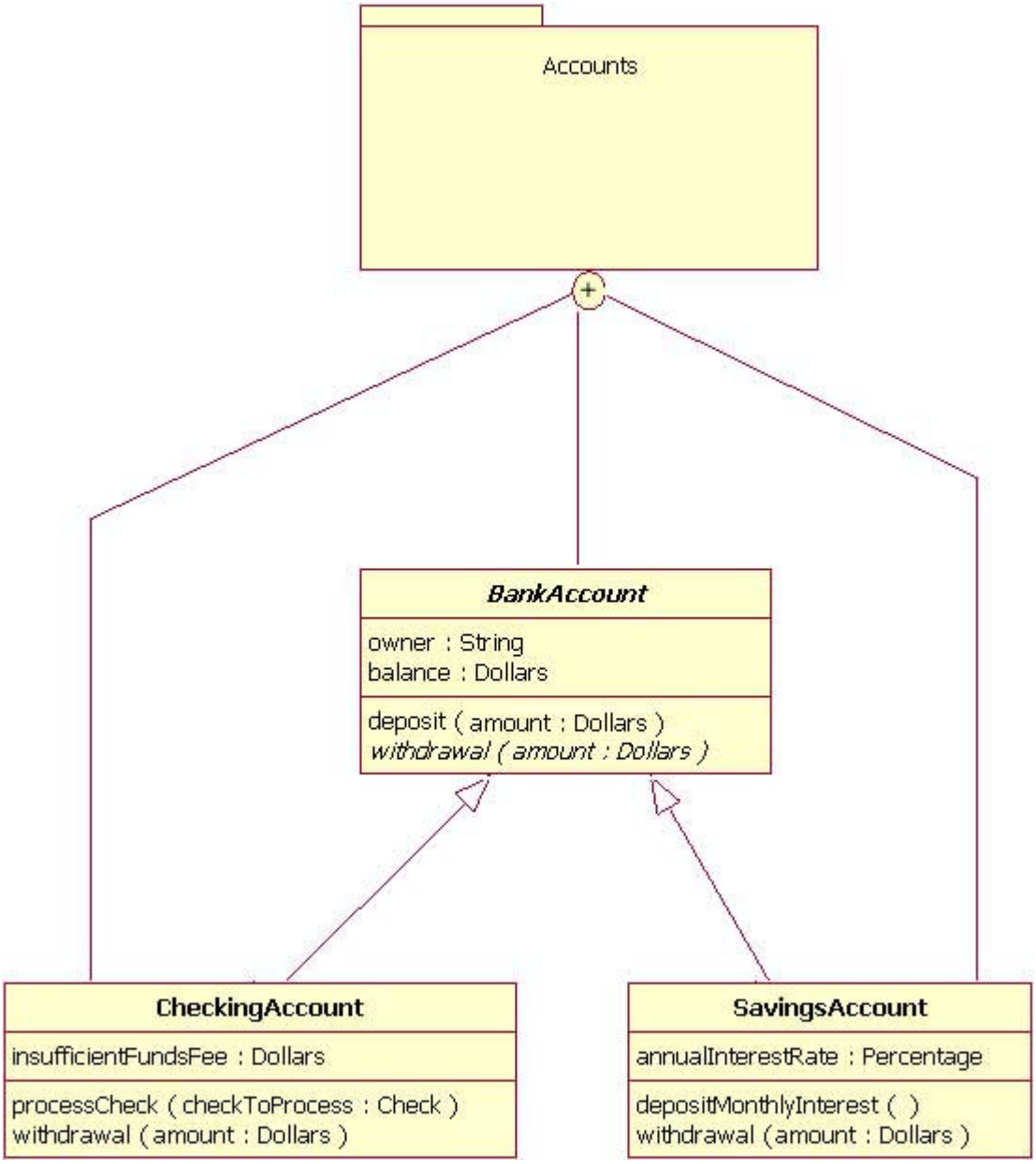

Figure 9: An example package element showing its membership via connected lines

Importance of understanding the basics

It is more important than ever in UML 2 to understand the basics of the class diagram. This is because the class diagram provides the basic building blocks for all other structure diagrams, such as the component or object diagrams (just to name a few).

Beyond the basics

At this point, I have covered the basics of the class diagram, but do not stop reading yet! In the following sections, I will address more important aspects of the class diagram that you can put to good use. These include interfaces, the three remaining types of associations, visibility, and other additions in the UML 2 specification.

Interfaces
Earlier in this article, I suggested that you think of *classifiers* simply as classes. In fact, a classifier is a more general concept, which includes data types and interfaces.

A complete discussion of when and how to use data types and interfaces effectively in a system's structure diagrams is beyond the scope of this article. So why do I mention data types and interfaces here? There are times when you might want to model these classifier types on a structure diagram, and it is important to use the proper notation in doing so, or at least be aware of these classifier types. Drawing these classifiers incorrectly will likely confuse readers of your structure diagram, and the ensuing system will probably not meet requirements.

A class and an interface differ: A class can have an actual instance of its type, whereas an interface must have at least one class to implement it. In UML 2, an interface is considered to be a specialization of a class modeling element. Therefore, an interface is drawn just like a class, but the top compartment of the rectangle also has the text "«interface»", as shown in Figure 10.[5]

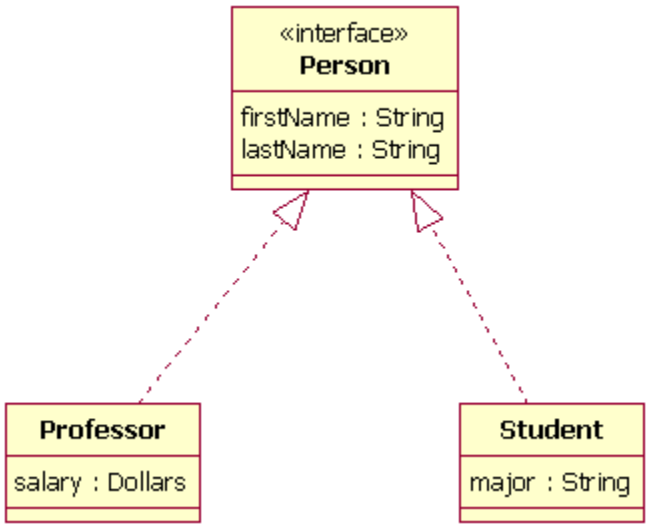

Figure 10: Example of a class diagram in which the Professor and Student classes implement the Person interface

In the diagram shown in Figure 10, both the Professor and Student classes implement the Person interface and do not inherit from it. We know this for two reasons: 1) The Person object is defined as an interface -- it has the "«interface»" text in the object's name area, and we see that the Professor and Student objects are *class* objects because they are labeled according to the rules for drawing a class object (there is no additional classification text in their name area). 2) We know inheritance is not being shown here, because the line with the arrow is dotted and not solid. As shown in Figure 10, a *dotted* line with a closed, unfilled arrow means realization (or implementation); as we saw in Figure 4, a *solid* arrow line with a closed, unfilled arrow means inheritance.

More associations
Above, I discussed bi-directional and uni-directional associations. Now I will address the three remaining types of associations.

Association class
In modeling an association, there are times when you need to include another class because it includes valuable information about the relationship. For this you would use an *association class* that you tie to the primary association. An association class is represented like a normal class. The difference is that the association line between the primary classes intersects a dotted line connected to the association class. Figure 11 shows an association class for our airline industry example.

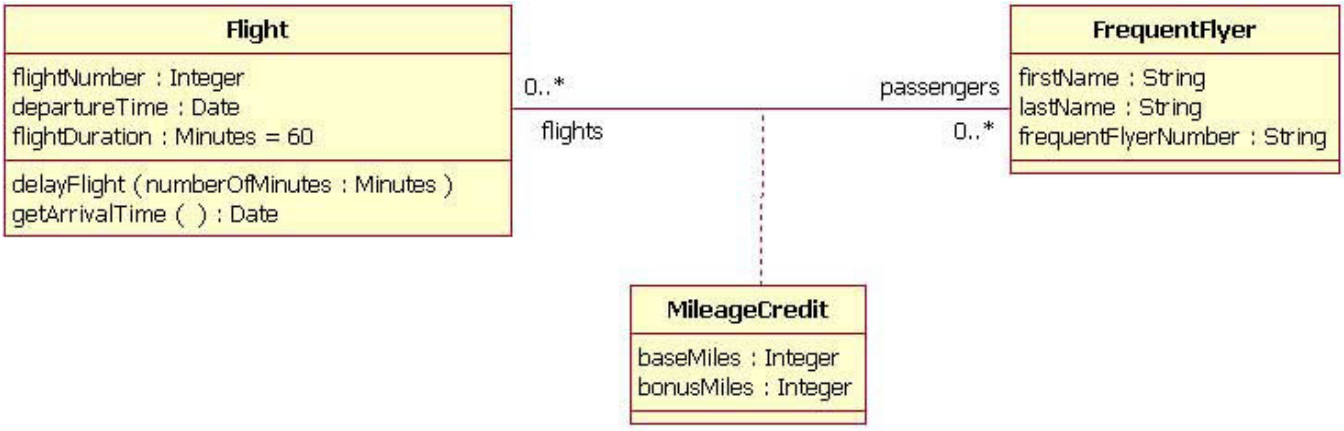

Figure 11: Adding the association class MileageCredit

In the class diagram shown in Figure 11, the association between the Flight class and the FrequentFlyer class results in an association class called MileageCredit. This means that when an instance of a Flight class is associated with an instance of a FrequentFlyer class, there will also be an instance of a MileageCredit class.

Aggregation
Aggregation is a special type of association used to model a "whole to its parts" relationship. In basic aggregation relationships, the lifecycle of a *part* class is independent from the *whole* class's lifecycle.

For example, we can think of *Car* as a whole entity and *Car Wheel* as part of the overall Car. The wheel can be created weeks ahead of time, and it can sit in a warehouse before being placed on a car during assembly. In this example, the Wheel class's instance clearly lives independently of the Car class's instance. However, there are times when the *part* class's lifecycle *is not* independent from that of the *whole* class -- this is called composition aggregation. Consider, for example, the relationship of a company to its departments. Both *Company and Departments* are modeled as classes, and a department cannot exist before a company exists. Here the Department class's instance is dependent upon the existence of the Company class's instance.

Let's explore basic aggregation and composition aggregation further.

Basic aggregation
An association with an aggregation relationship indicates that one class is a part of another class. In an aggregation relationship, the child class instance can outlive its parent class. To represent an aggregation relationship, you draw a solid line from the parent class to the part class, and draw an unfilled diamond shape on the parent class's association end. Figure 12 shows an example of an aggregation relationship between a Car and a Wheel.

Figure 12: Example of an aggregation association

Composition aggregation
The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In Figure 13, which shows a composition relationship between a Company class and a Department class, notice that the composition relationship is drawn like the aggregation relationship, but this time the diamond shape is filled.

Figure 13: Example of a composition relationship

In the relationship modeled in Figure 13, a Company class instance will always have at least one Department class instance. Because the relationship is a composition relationship, when the Company instance is removed/destroyed, the Department instance is automatically removed/destroyed as well. Another important feature of composition aggregation is that the part class can only be related to one instance of the parent class (e.g. the Company class in our example).

Reflexive associations
We have now discussed all the association types. As you may have noticed, all our examples have shown a relationship between two different classes. However, a class can also be associated with itself, using a reflexive association. This may not make sense at first, but remember that classes are abstractions. Figure 14 shows how an Employee class could be related to itself through the manager/manages role. When a class is associated to itself, this does not mean that a class's instance is related to itself, but that an instance of the class is related to another instance of the class.

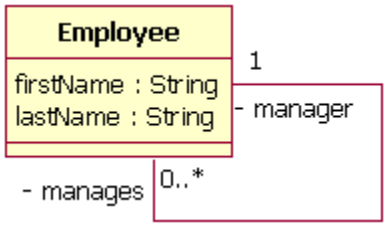

Figure 14: Example of a reflexive association relationship

The relationship drawn in Figure 14 means that an instance of Employee can be the manager of another Employee instance. However, because the relationship role of "manages" has a multiplicity of 0..*; an Employee might not have any other Employees to manage.

Visibility
In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation. To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilities. Table 4 displays the different marks for the UML-supported visibility types.

Table 4: Marks for UML-supported visibility types

Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Now, let's look at a class that shows the visibility types indicated for its attributes and operations. In Figure 15, all the attributes and operations are public, with the exception of the updateBalance operation. The updateBalance operation is protected.

Figure 15: A BankAccount class that shows the visibility of its attributes and operations

UML 2 additions

Now that we have covered the basics and the advanced topics, we will cover some of the new notations added to the class diagram from UML 1.x.

Instances
When modeling a system's structure it is sometimes useful to show example instances of the classes. To model this, UML 2 provides the *instance specification* element, which shows interesting information using example (or real) instances in the system.

The notation of an instance is the same as a class, but instead of the top compartment merely having the class's name, the name is an underlined concatenation of:

```
Instance Name : Class Name
```

For example:

```
Donald : Person
```

Because the purpose of showing instances is to show interesting or relevant information, it is not necessary to include in your model the entire instance's attributes and operations. Instead it is completely appropriate to show only the attributes and their values that are interesting as depicted in Figure 16.

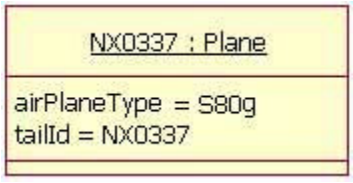

Figure 16: An example instance of a Plane class (only the interesting attribute values are shown)

However, merely showing some instances without their relationship is not very useful; therefore, UML 2 allows for the modeling of the relationships/associations at the instance level as well. The rules for drawing associations are the same as for normal class relationships, although there is one additional requirement when modeling the associations. The additional restriction is that association relationships must match the class diagram's relationships and therefore the association's role names must also match the class diagram. An example of this is shown in Figure 17. In this example the instances are example instances of the class diagram found in Figure 6.

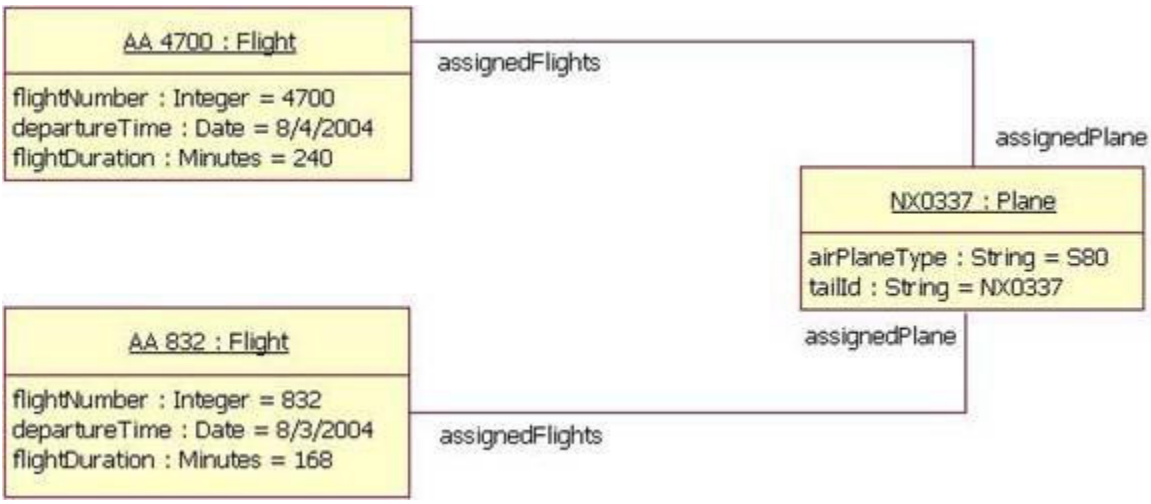

Figure 17: An example of Figure 6 using instances instead of classes

Figure 17 has two instances of the Flight class because the class diagram indicated that the relationship between the Plane class and the Flight class is *zero-to-many*. Therefore, our example shows the two Flight instances that the NX0337 Plane instance is related to.

Roles
Modeling the instances of classes is sometimes more detailed than one might wish. Sometimes, you may simply want to model a class's relationship at a more generic level. In such cases, you should use the *role* notation. The role notation is very similar to the instances notation. To model a class's role, you draw a box and place the class's role name and class name inside as with the instances notation, but in this case you do not underline the words. Figure 18 shows an example of the roles played by the Employee class described by the diagram at Figure 14. In Figure 18, we can tell, even though the Employee class is related to itself, that the relationship is really between an Employee playing the role of manager and an Employee playing the role of team member.

Figure 18: A class diagram showing the class in Figure 14 in its different roles

Note that you cannot model a class's role on a plain class diagram, even though Figure 18 makes it appear that you can. In order to use the role notation you will need to use the Internal Structure notation, discussed next.

Internal Structures
One of the more useful features of UML 2 structure diagrams is the new internal structure notation. It allows you to show how a class or another classifier is internally composed. This was not possible in UML 1.x, because the notation set limited you to showing only the aggregation relationships that a class had. Now, in UML 2, the internal structure notation lets you more clearly show how that class's parts relate to each other.

Let's look at an example. In Figure 18 we have a class diagram showing how a Plane class is composed of four engines and two control software objects. What is missing from this diagram is any information about how airplane parts are assembled. From the diagram in Figure 18, you cannot tell if the control software objects control two engines each, or if one control software object controls three engines and the other controls one engine.

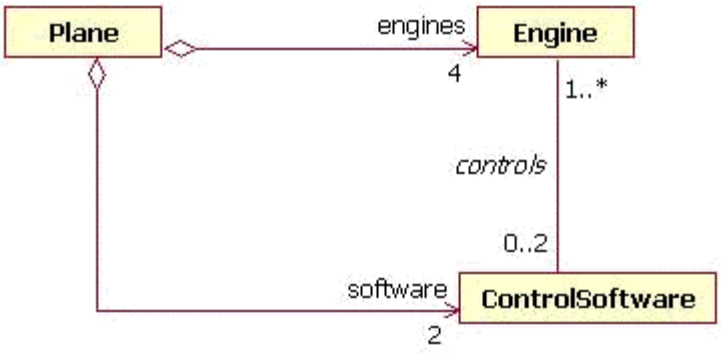

Figure 19: A class diagram that only shows relationships between the objects

Drawing a class's internal structure will improve this situation. You start by drawing a box with two compartments. The top compartment contains the class name, and the lower compartment contains the class's internal structure, showing the parent class's part classes in their respective roles, as well as how each particular class relates to others in that role. Figure 19 shows the internal structure of Plane class; notice how the internal structure clears up the confusion.

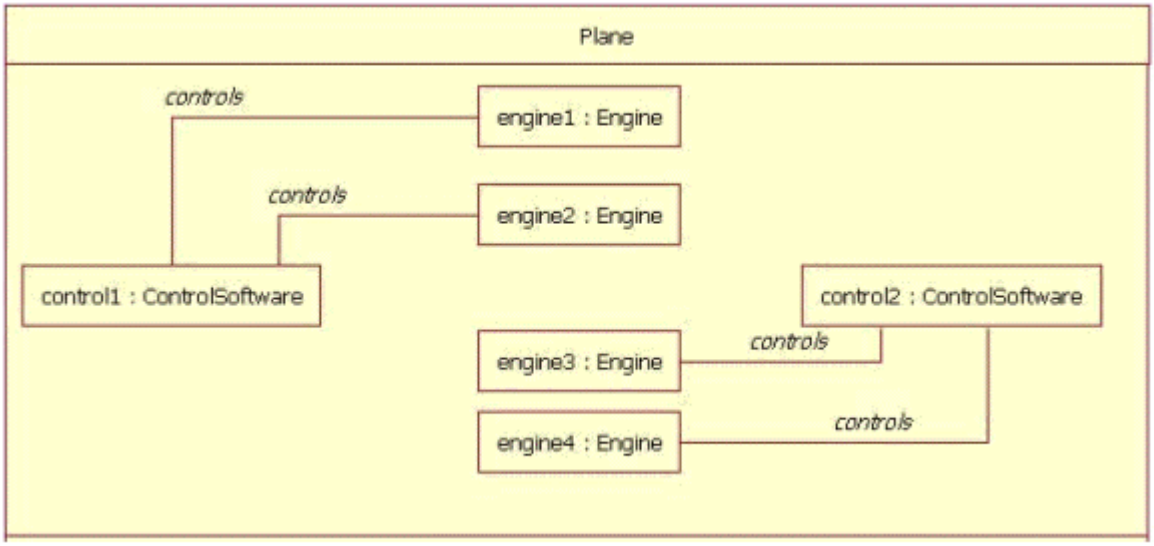

Figure 20: An example internal structure of a Plane class.

In Figure 20 the Plane has two ControlSoftware objects and each one controls two engines. The ControlSoftware on the left side of the diagram (control1) controls engines 1 and 2. The ControlSoftware on the right side of the diagram (control2) controls engines 3 and 4.

Conclusion

There are at least two important reasons for understanding the class diagram. The first is that it shows the static structure of classifiers in a system; the second reason is that the diagram provides the basic notation for other structure diagrams prescribed by UML. Developers will think the class diagram was created specially for them; but other team members will find them useful, too. Business analysts can use class diagrams to model systems from the business perspective. As we will see in other articles in this series on UML basics, other diagrams -- including the activity, sequence, and statechart diagrams -- refer to the classes modeled and documented on the class diagram.

Haaris Infotech

Next in this series on "UML basics:" the component diagram.

Notes

[1] The delayFlight does not have a return value because I made a design decision not to have one. One could argue that the delay operation should return the new arrival time, and if this were the case, the operation signature would appear as delayFlight(numberOfMinutes : Minutes) : Date.

[2]It may seem strange that the BankAccount class does not know about the OverdrawnAccountsReport class. This modeling allows report classes to know about the business class they report, but the business classes do not know they are being reported on. This loosens the coupling of the objects and therefore makes the system more adaptive to changes.

[3] Packages are great for organizing your model's classes, but it's important to remember that your class diagrams are supposed to easily communicate information about the system being modeled. In cases where your packages have lots of classes, it is better to use multiple topic-specific class diagrams instead of just producing one large class diagram.

[4]It's important to understand that when I say "all those members," I mean only the classes that the current diagram is going to show. A diagram showing a package with contents does not need to show all its contents; it can show a subset of the contained elements according to some criterion, which is not necessarily all the package's classifiers.

[5] When drawing a class diagram it is completely within UML specification to put «class» in the top compartment of the rectangle, as you would with «interface»; however, the UML specification says that placing the "class" text in this compartment is optional, and it should be assumed if «class» is not displayed.

Introduction to Component Diagram

Level: Introductory

The diagram's purpose

The component diagram's main purpose is to show the structural relationships between the components of a system. In UML 1.1, a component represented implementation items, such as files and executables. Unfortunately, this conflicted with the more common use of the term component," which refers to things such as COM components. Over time and across successive releases of UML, the original UML meaning of components was mostly lost. UML 2 officially changes the essential meaning of the component concept; in UML 2, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces. Although the UML 2 specification does not strictly state it, components are larger design units that represent things that will typically be implemented using replaceable" modules. But, unlike UML 1.x, components are now strictly logical, design-time constructs. The idea is that you can easily reuse and/or substitute a different component implementation in your designs because a component encapsulates behavior and implements specified interfaces. [1]

In component-based development (CBD), component diagrams offer architects a natural format to begin modeling a solution. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

In addition, component diagrams are useful communication tools for various groups. The diagrams can be presented to key project stakeholders and implementation staff. While component diagrams are generally geared towards a system's implementation staff, component diagrams can generally put stakeholders at ease because the diagram presents an early understanding of the overall system that is being built.

Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building, which helps developers begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed skill enhancements. System administrators find component diagrams useful because they get an early view of the logical software components that will be running on their systems. Although system administrators will not be able to identify the physical machines or the physical executables from the diagram, a component diagram will nevertheless be welcomed because it provides early information about the components and their relationships (which allows sys-admins to loosely plan ahead).

The notation

The component diagram notation set now makes it one of the easiest UML diagrams to draw. Figure 1 shows a simple component diagram using the former UML 1.4 notation; the example shows a relationship between two components: an Order System component that uses the Inventory System component. As you can see, a component in UML 1.4 was drawn as a rectangle with two smaller rectangles protruding from its left side.

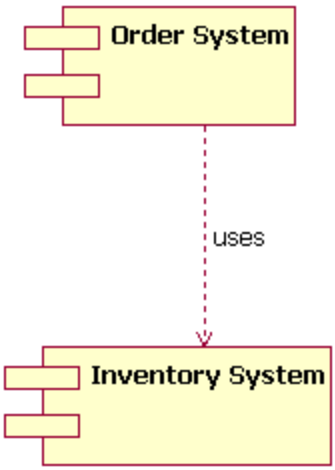

Figure 1: This simple component diagram shows the Order System's general dependency using UML 1.4 notation

The above UML 1.4 notation is still supported in UML 2. However, the UML 1.4 notation set did not scale well in larger systems. For that reason, UML 2 dramatically enhances the notation set of the component diagram, as we will see throughout the rest of this article. The UML 2 notation set scales better, and the notation set is also more informative while maintaining its ease of understanding.

Let's step through the component diagram basics according to UML 2.

The basics

Drawing a component in UML 2 is now very similar to drawing a class on a class diagram. In fact, in UML 2 a component is merely a specialized version of the class concept. Which means that the notation rules that apply to the class classifier also apply to the component classifier

In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as just a rectangle with the component's name and the component stereotype text and/or icon. The component stereotype's text is «component»" and the component stereotype icon is a rectangle with two smaller rectangles protruding on its left side (the UML 1.4 notation element for a component). Figure 2 shows three different ways a component can be drawn using the UML 2 specification.

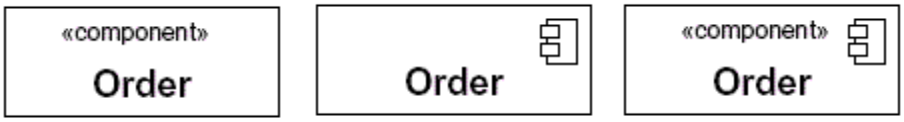

Figure 2: The different ways to draw a component's name compartment

When drawing a component on a diagram, it is important that you always include the component stereotype text (the word "component" inside double angle brackets, as shown in Figure 2) and/or icon. The reason? In UML, a rectangle without any stereotype classifier is interpreted as a class element. The component stereotype and/or icon distinguishes this rectangle as a component element.

Modeling a component's interfaces Provided/Required

The Order components drawn in Figure 2 all represent valid notation elements; however, a typical component diagram includes more information. A component element can have additional compartments stacked below the name compartment. As mentioned earlier, a component is an autonomous unit that provides one or more public interfaces. The interfaces provided represent the formal contract of services the component provides to its consumers/clients. Figure 3 shows the Order component having a second compartment that denotes what interfaces the Order component provides and requires.[2]

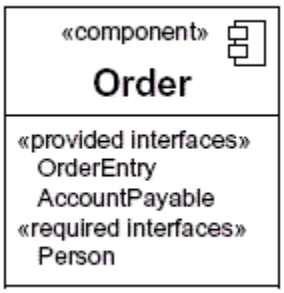

Figure 3: The additional compartment here shows the interfaces that the Order component provides and requires.

In the example Order component shown in Figure 3, the component provides the interfaces of OrderEntry and AccountPayable. Additionally, the component also requires another component that provides the Person interface.[3]

Another approach to modeling a component's interfaces

UML 2 has also introduced another way to show a component's provided and required interfaces. This second way builds off the single rectangle, with the component's name in it, and places what the UML 2 specification calls interface symbols" connected to the outside of the rectangle. This second approach is illustrated in Figure 4.

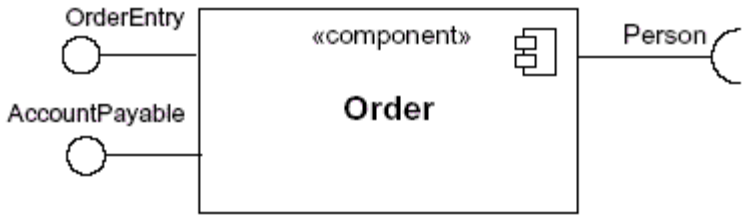

Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols

In this second approach the interface symbols with a complete circle at their end represent an interface that the component provides -- this lollipop" symbol is shorthand for a realization relationship of an interface classifier. Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself). Even though Figure 4 looks much different from Figure 3, both figures provide the same information -- i.e., the Order component *provides* two interfaces: OrderEntry and AccountPayable, and the Order component *requires* the Person interface.

Modeling a component's relationships

When showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop, as shown in Figure 5.

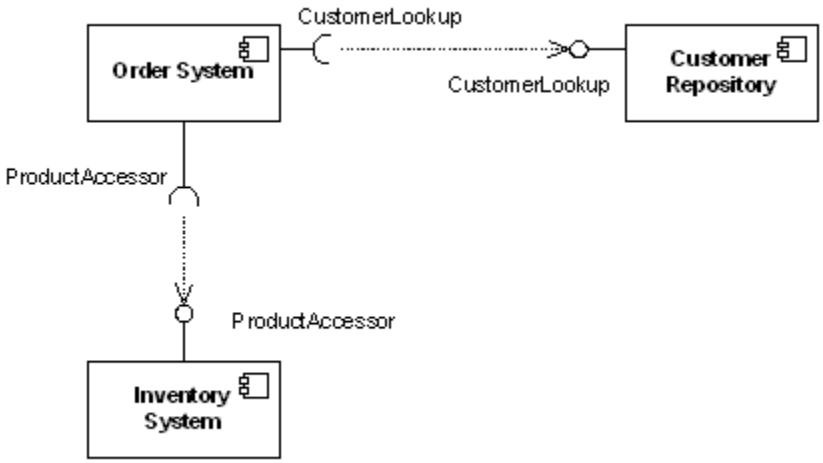

Figure 5: A component diagram that shows how the Order System component depends on other components

Figure 5 shows that the Order System component depends both on the Customer Repository and Inventory System components. Notice in Figure 5 the duplicated names of the interfaces CustomerLookup" and ProductAccessor." While this may seem unnecessarily repetitive in this example, the notation actually allows for different interfaces (and differing names) on each component depending on the implementation differences (e.g., one component provides an interface that is a subclass of a smaller required interface).

Subsystems

In UML 2 the subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem" instead of component," as shown in Figure 6.

Figure 6: An example of a subsystem element

The UML 2 specification is quite vague on how a subsystem is different from a component. The specification does not treat a component or a subsystem any differently from a modeling perspective. Compared with UML 1.x, this UML 2 modeling ambiguity is new. But there's a reason. In UML 1.x, a subsystem was considered a package, and this package notation was confusing to many UML practitioners; hence UML 2 aligned subsystems as a specialized component, since this is how most UML 1.x users understood it. This change did introduce fuzziness into the picture, but this fuzziness is more of a reflection of reality versus a mistake in the UML 2 specification.

So right now you are probably scratching your head wondering when to use a component element versus a subsystem element. Quite frankly, I do not have a direct answer for you. I can tell you that the UML 2 specification says that the decision on when to use a component versus a subsystem is up to the methodology of the modeler. I personally like this answer because it helps ensure that UML stays methodology independent, which helps keep it universally usable in software development.

Beyond the basics

The component diagram is one of the easier-to-understand diagrams, so there is not much to cover beyond the basics. However, there is one area you may consider somewhat advanced.

Showing a component's internal structure

There will be times when it makes sense to display a component's internal structure. In my previous article on the class diagram, I showed how to model a class's internal structure; here I will focus on how to model a component's internal structure when it is composed of other components.

To show a component's inner structure, you merely draw the component larger than normal and place the inner parts inside the name compartment of the encompassing component. Figure 7 show's the Store's component inner structure.

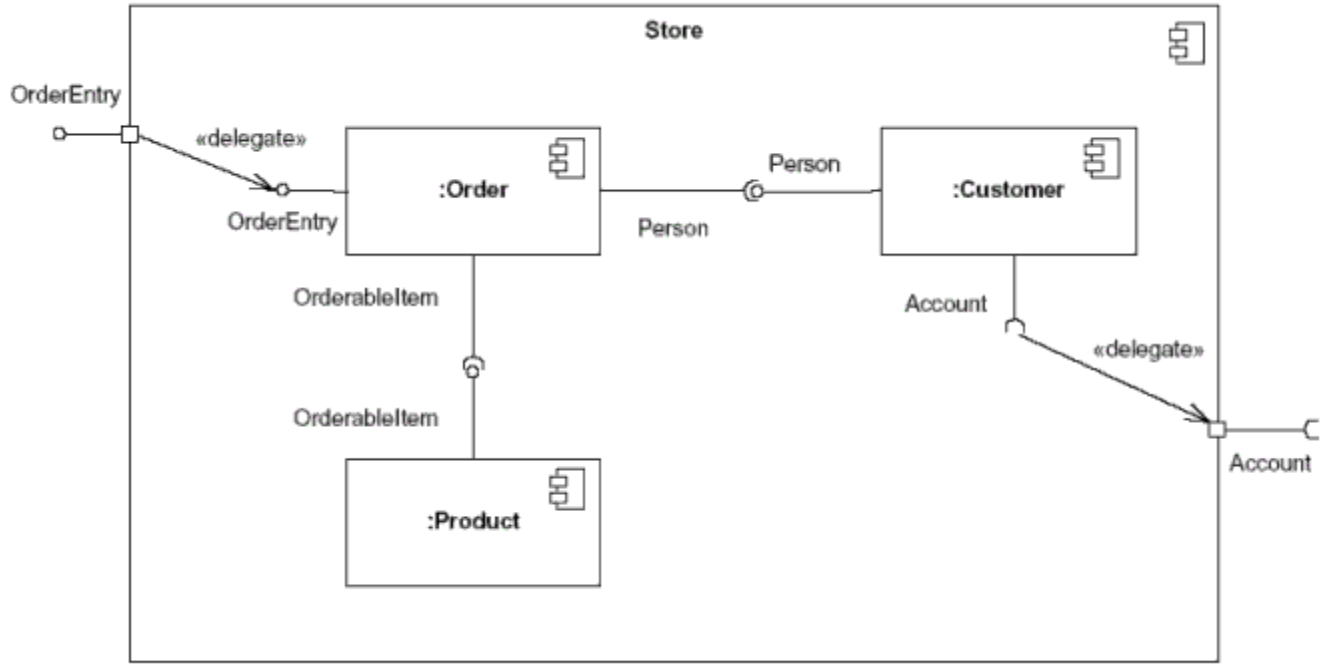

Figure 7: This component's inner structure is composed of other components.

Using the example shown in Figure 7, the Store component provides the interface of OrderEntry and requires the interface of Account. The Store component is made up of three components: Order, Customer, and Product components. Notice how the Store's OrderEntry and Account interface symbols have a square on the edge of the component. This square is called a port. In a simplistic sense, ports provide a way to model how a component's *provided/required* interfaces relate to its internal parts.[4] By using a port, our diagram is able to de-couple the internals of the Store component from external entities. In Figure 7, the OrderEntry port delegates to the Order component's OrderEntry interface for processing. Also, the internal Customer component's required Account interface is delegated to the Store component's required Account interface port. By connecting to the Account port, the internals of the Store component (e.g. the Customer component) can have a local representative of some unknown external entity which implements the port's interface. The required Account interface will be implemented by a component outside of the Store component.[5]

You will also notice in Figure 7 that the interconnections between the inner components are different from those shown in Figure 5. This is because these depictions of internal structures are really collaboration diagrams nested inside the classifier (a component, in our case), since collaboration diagrams show instances or roles of classifiers. The relationship modeled between the internal components is drawn with what UML calls an assembly connector." An assembly connector ties one component's *provided* interface with another component's *required* interface. Assembly connectors are drawn as lollipop and socket symbols next to each other. Drawing these assembly connectors in this manner makes the lollipop and socket symbols very easy to read.

Conclusion

The component diagram is a very important diagram that architects will often create early in a project. However, the component diagram's usefulness spans the life of the system. Component diagrams are invaluable because they model and document a system's architecture. Because component diagrams document a system's architecture, the developers and the eventual system administrators of the system find this work product-critical in helping them understand the system.

Component diagrams also serve as input to a software system's deployment diagram, which will be the topic of my next article in this series.

Notes

[1]The physical items that UML1.x called components are now called "artifacts" in UML 2. An artifact is a physical unit, such as a file, executable, script, database, etc. Only artifacts live on physical nodes; classes and components do not have "location." However, an artifact may manifest components and other classifiers (i.e., classes). A single component could be manifested by multiple artifacts, which could be on the same or different nodes, so a single component could indirectly be implemented on multiple nodes.

[2]Even though components are autonomous units they still may depend on the services provided by other components. Because of this, documenting a component's required interfaces is useful.

[3]Figure 3 does not show the Order component in its complete context. In a real-world model the OrderEntry, AccountPayable, and Person interfaces would be present in the system's model.

[4]In actuality, ports are applicable to any type of classifier (i.e., to a class or some other classifier your model might have). To keep this article simple, I refer to ports in their use on component classifiers.

[5]Typically, when you draw a dependency relationship between a port and an interface, the dependent (requiring) interface will handle all the processing logic at execution time. However, this is not a hard and fast rule -- it is completely acceptable for the encompassing component (e.g., the Store component in our example) to have its own processing logic instead of merely delegating the processing to the dependant interface.