Unit-4
Basic Behavioral Modeling-I

Syllabus : Interactions, Interaction diagrams, Use cases, Use case Diagrams, Activity diagrams

Interactions

Terms and Concepts

An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Context

You may find an interaction wherever objects are linked to one another. You'll find interactions in the collaboration of objects that exist in the context of your system or subsystem. You will also find interactions in the context of an operation. Finally, you'll find interactions in the context of a class.

Most often, you'll find interactions in the collaboration of objects that exist in the context of your system or subsystem as a whole. For example, in a system for Web commerce, you'll find objects on the client (such as instances of the classes **BookOrder** and **OrderForm**) interacting with one another. You'll also find objects on the client (again, such as instances of **BookOrder**) interacting with objects on the server (such as instances of **BackOrderManager**). These interactions therefore not only involve localized collaborations of objects (such as the interactions surrounding **OrderForm**), but they may also cut across many conceptual levels of your system (such as theinteractions surrounding **BackOrderManager**).

You'll also find interactions among objects in the implementation of an operation. The parameters of an operation, any variables local to the operation, and any objects global to the operation (but still visible to the operation) may interact with one another to carry out the algorithm of that operation's implementation. For example, invoking the operation **moveToPosition(p :Position)** defined for a class in a mobile robot will involve the interaction of a parameter (**p**), anobject global to the operation (such as the object **currentPosition**), and possibly several local objects (such as local variables used by the operation to calculate intermediate points in a path to the new position).

Finally, you will find interactions in the context of a class. You can use interactions to visualize, specify, construct, and document the semantics of a class. For example, to understand the meaning of a class **RayTraceAgent,** you might create interactions that show how the attributes of that class collaborate with one another (and with objects global to instances of the class and with parameters defined in the class's operations).

Objects and Roles

The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, **p,** an instance of the class **Person,** might denote a particular human. Alternately, as a prototypical thing, **p** might represent any instance of **Person.**

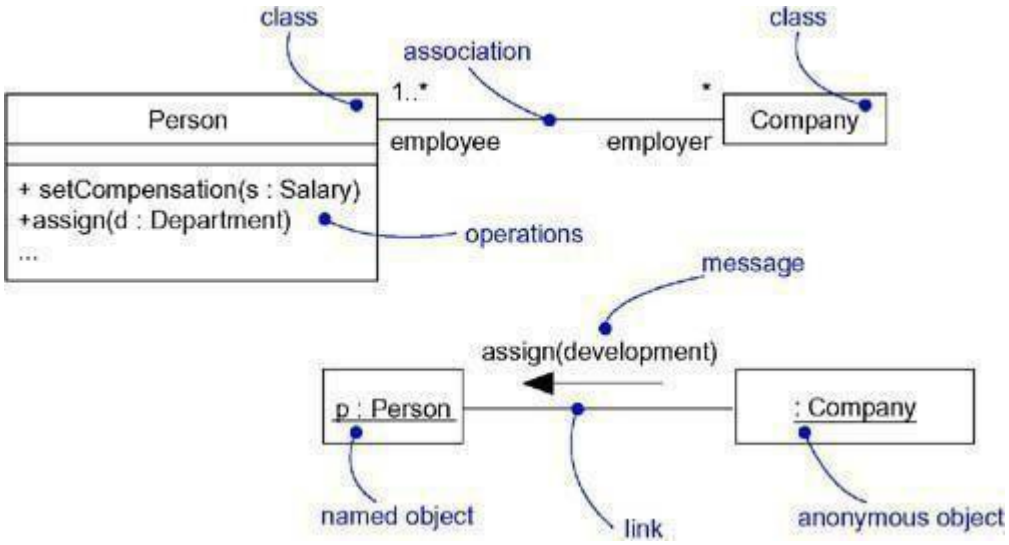
In the context of an interaction, you may find instances of classes, components, nodes, and use cases. Although abstract classes and interfaces, by definition, may not have any direct instances, you may find instances of these things in an interaction. Such instances do not represent direct instances of the abstract class or of the interface, but may represent, respectively, indirect (or prototypical) instances of any concrete children of the abstract class of some concrete class that realizes that interface.

You can think of an object diagram as a representation of the static aspect of an interaction, setting the stage for the interaction by specifying all the objects that work together. An interaction goes further by introducing a dynamic sequence of messages that may pass along the links that connect these objects

Links

A link is a semantic connection among objects. In general, a link is an instance of an association. As Figure shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.

Figure Links and Associations



A link specifies a path along which one object can dispatch a message to another (or the same) object. Most of the time, it is sufficient to specify that such a path exists. If you need to be more precise about how that path exists, you can adorn the appropriate end of the link with any of the following standard stereotypes.

association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible because it is the dispatcher of the operation
global	Specifies that the corresponding object is visible because it is in an enclosing scope
local	Specifies that the corresponding object is visible because it is in a local scope
parameter	Specifies that the corresponding object is visible because it is a parameter

Messages

Call	Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
Return	Returns a value to the caller
Send	Sends a signal to an object
Create	Creates an object
Destroy	Destroys an object; an object may commit suicide by destroying itself

Suppose you have a set of objects and a set of links that connect those objects. If that's all you have, then you have a completely static model that can be represented by an object diagram. Object diagrams model the state of a society of objects at a given moment in time and are useful when you want to visualize, specify, construct, or document a static object structure.

Suppose you want to model the changing state of a society of objects over a period of time. Think of it as taking a motion picture of a set of objects, each frame representing a successive moment in time. If these objects are not totally idle, you'll see objects passing messages to other objects, sending events, and invoking operations. In addition, at each frame, you can explicitly visualize the current state and role of individual instances.

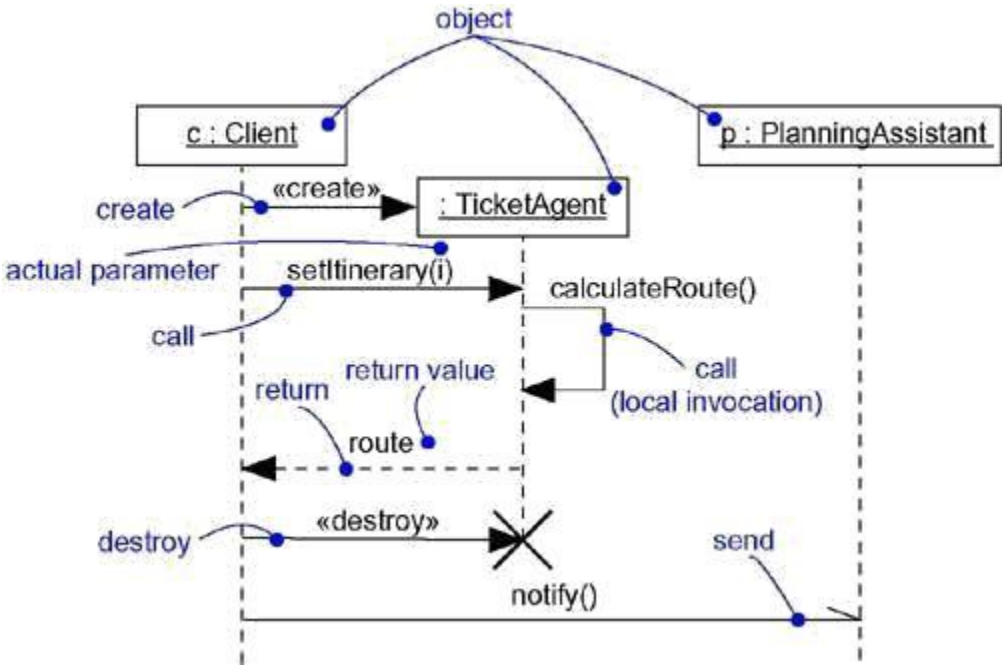
A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue. The receipt of a message instance may be considered an instance of an event.

When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change in state.

In the UML, you can model several kinds of actions.

The UML provides a visual distinction among these kinds of messages, as Figure shows.

Figure Messages



The most common kind of message you'll model is the call, in which one object invokes an operation of another (or the same) object. An object can't just call any random operation. If an object, such as **c** in the example above, calls the operation **setItinerary** on an instance of the class **TicketAgent,** the operation **setItinerary** must not only be defined for the class **TicketAgent**(that is, it must be declared in the class**TicketAgent**or one of its parents), itmust also be visible to the caller **c.**

When an object calls an operation or sends a signal to another object, you can provide actual parameters to the message. Similarly, when an object returns control to another object, you can model the return value, as well.

Sequencing

When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning; the start of every sequence is rooted in some process or thread. Furthermore, any sequence will continue as long as the process or thread that owns it lives. A nonstop system, such as you might find in real time device control, will continue to execute as long as the node it runs on is up.

Each process and thread within a system defines a distinct flow of control, and within each flow, messages are ordered in sequence by time. To better visualize the sequence of a message, you can explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure shows. In this case, the message **findAt**is specified as the first message nested in the second message of the sequence (**2.1**).

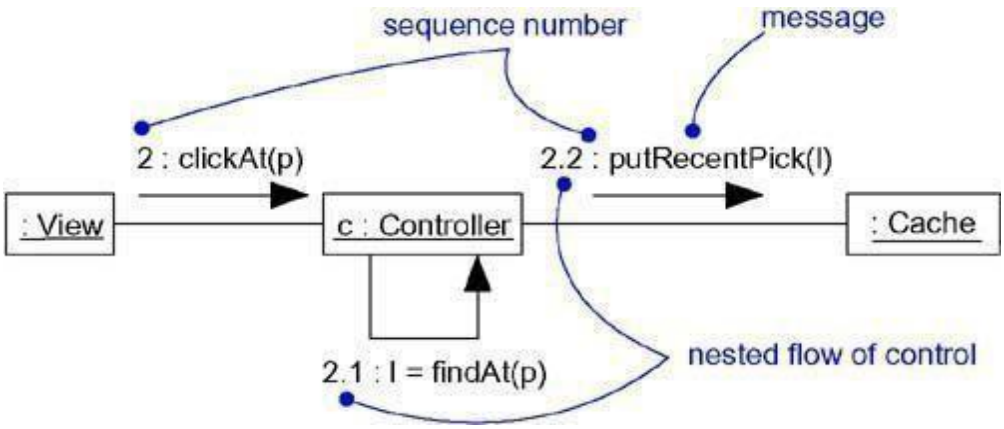


Figure Procedural Sequence

Less common but also possible, as Figure shows, you can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step. In this case, the message **assertCall** is specified as the second message in the sequence.

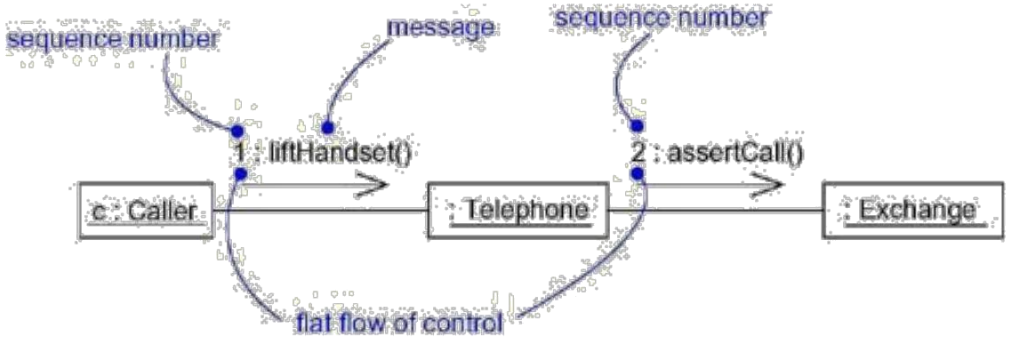


Figure Flat Sequenc

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D.**

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find,** dispatched with the actual parameter **"Rachelle".** This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed the en	Specifies that the instance or link is destroyed prior to completion of execution of losing interaction
transient interact	Specifies that the instance or link is created during execution of the enclosing ion but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and

receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Contro

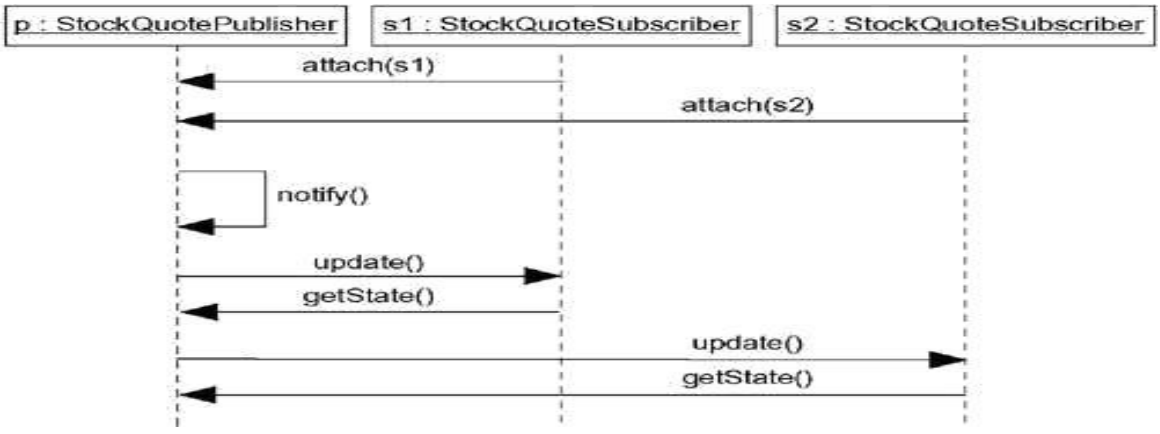
When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.

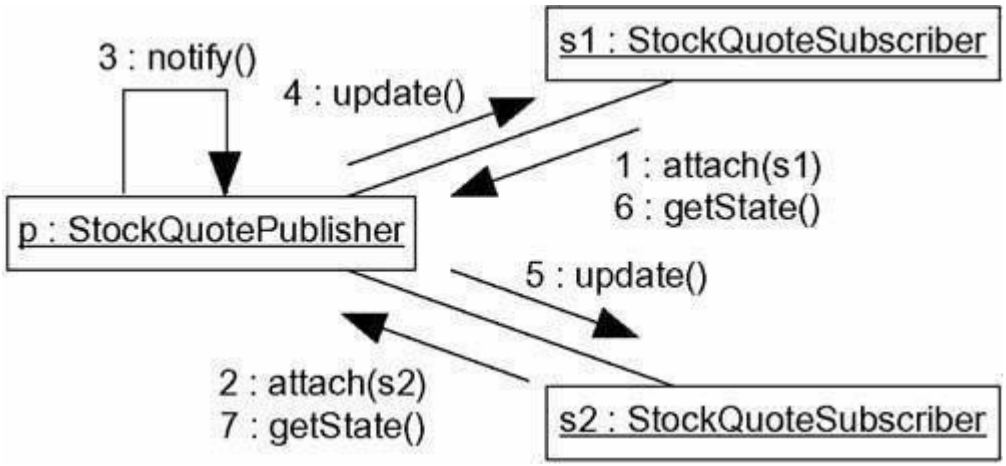
For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1,** and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizesthe time order of messages.

Figure Flow of Control by Time



Figureis semantically equivalent to the previous one, but it is drawn as a collaborationdiagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

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

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

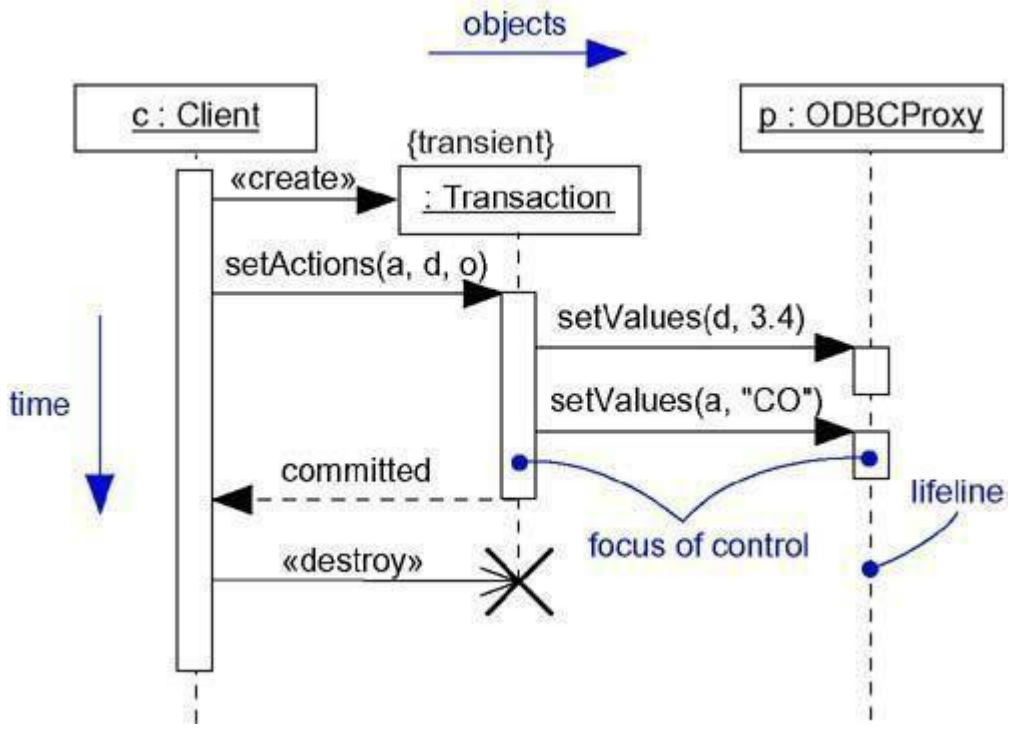
Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

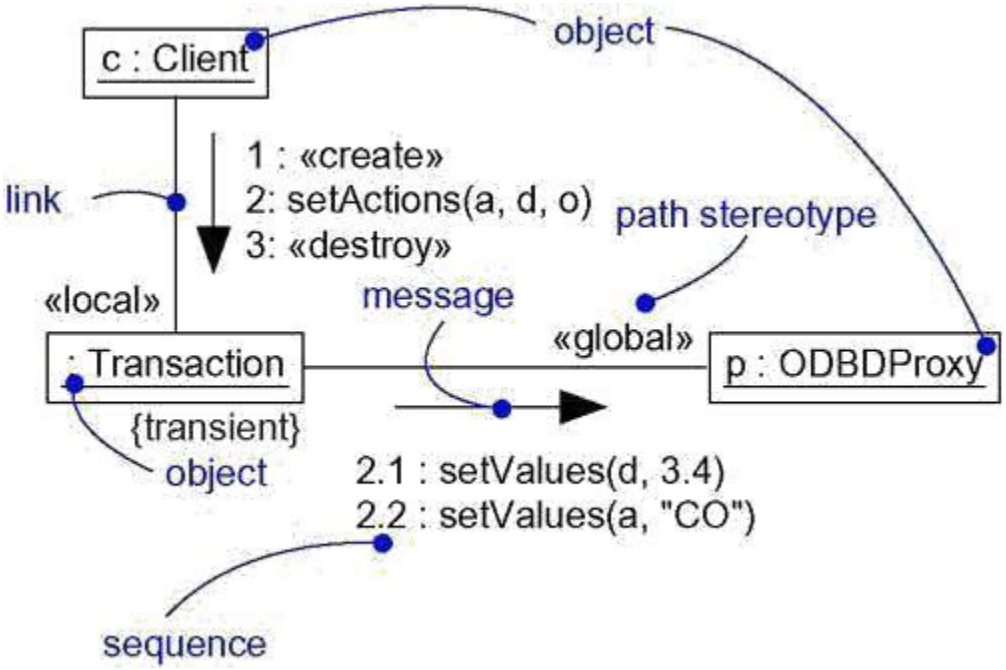
A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the

X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



Sequence diagrams have two features that distinguish them from collaboration diagrams.



First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create.** Objects may be destroyed during the interaction. Their lifelines end with the receipt ofthe message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self- operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collboration Diagram

Collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is localto the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in the flow of control (**2**, **3**, and so on). To show nesting, you use Dewey decimal numbering (**1** is the first message; **1.1** is the first message nested in message **1**; **1.2** is the second message nested in message **1** ; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as ***[i := 1..n]** (or just ***** if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as **[x > 0].** The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, thesequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

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

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

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

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instances in the interaction, along which messages may be

passed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

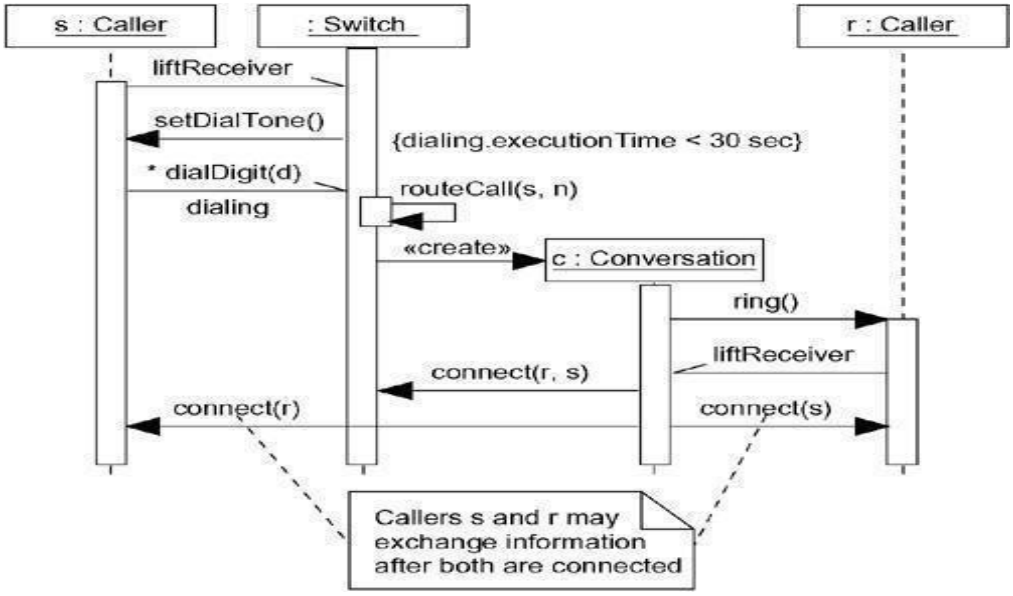
To model a flow of control by time ordering,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, ~~Figure~~ shows a sequence diagram that specifies the flow of control involvedin initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Caller**s (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatchinga signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message**dialDigit.** Note that this message has atiming mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall.** It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver.** The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**, after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as **become** or **copy** (with a suitable sequence number).
- Specify the links among these objects, along which messages may pass.
    1. Lay out the association links first; these are the most important ones, because they represent structural connections.
    2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to oneanother.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey

decimal numbering.

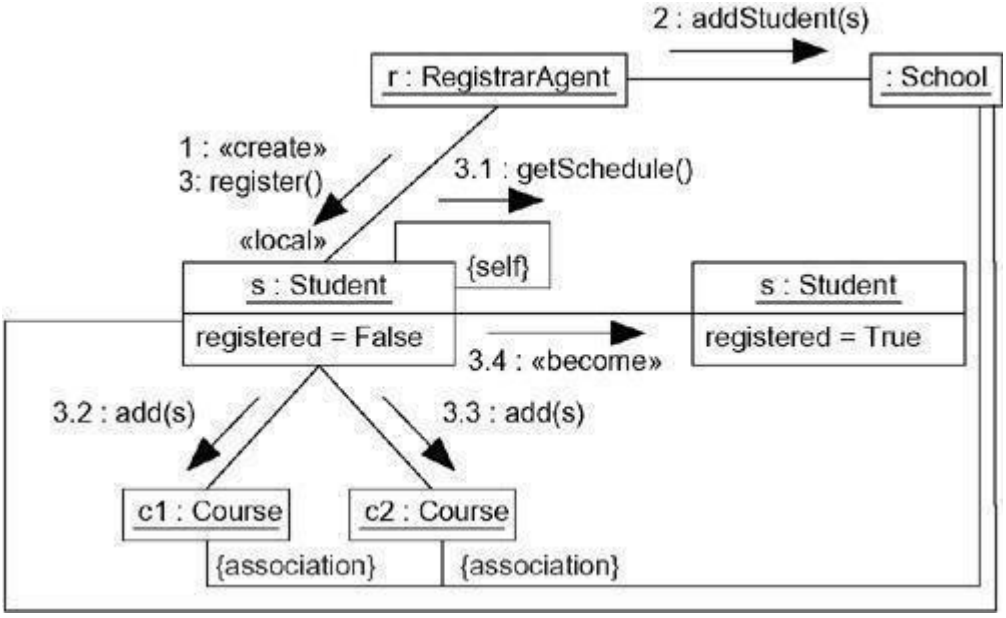
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.

- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, ~~Figure~~ shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (**r**), a **Student** (**s**), two **Course** objects (**c1** and **c2**), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes**getSchedule**on itself, presumably obtaining the **Course** objectsfor which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with **s** rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself.**s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to**getSchedule**(which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence andcollaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =
    getSchedule(); for (inti = 0; i<c.size(); i++)
        c.item(i).add(this);
    this.registered = true;
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection**object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence andcollaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D.**

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find,** dispatched with the actual parameter **"Rachelle".** This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed	Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
transient	Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Control

The most common purpose for which you'll use interactions is to model the flow of control that characterizes the behavior of a system as a whole, including use cases, patterns, mechanisms, and frameworks, or the behavior of a class or an individual operation. Whereas classes, interfaces, components, nodes, and their relationships model the static aspects of your system, interactions model its dynamic aspects.

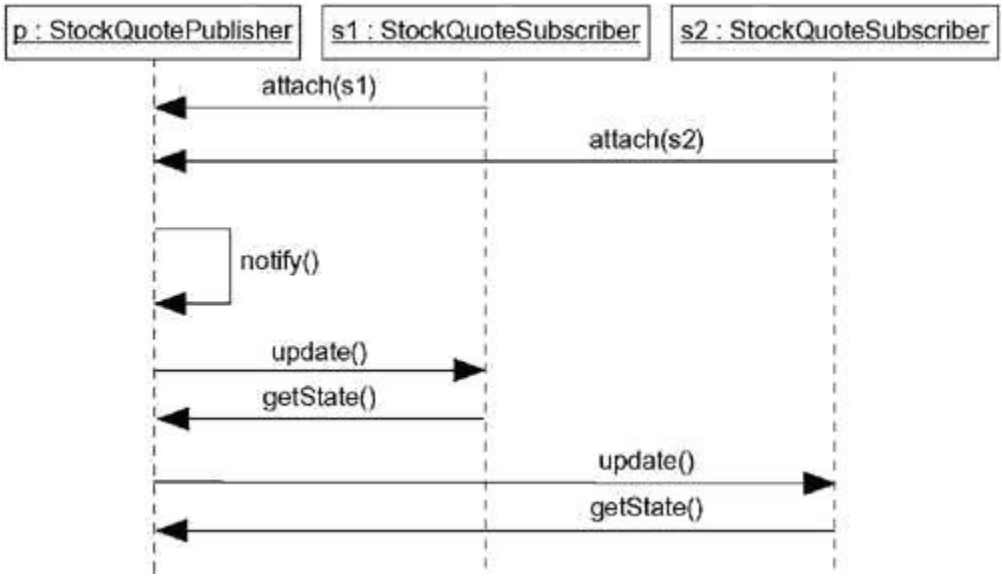
When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.
To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every

moment in time with its state and role.

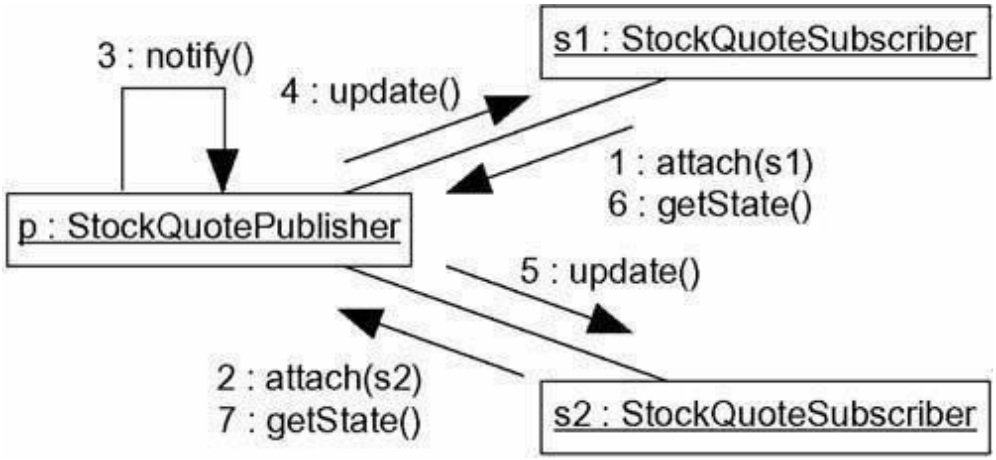
For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1,** and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizesthe time order of messages.

Figure Flow of Control by Time



Eigureis semantically equivalent to the previous one, but it is drawn as a collaborationdiagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

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

Contents

Interaction diagrams commonly contain

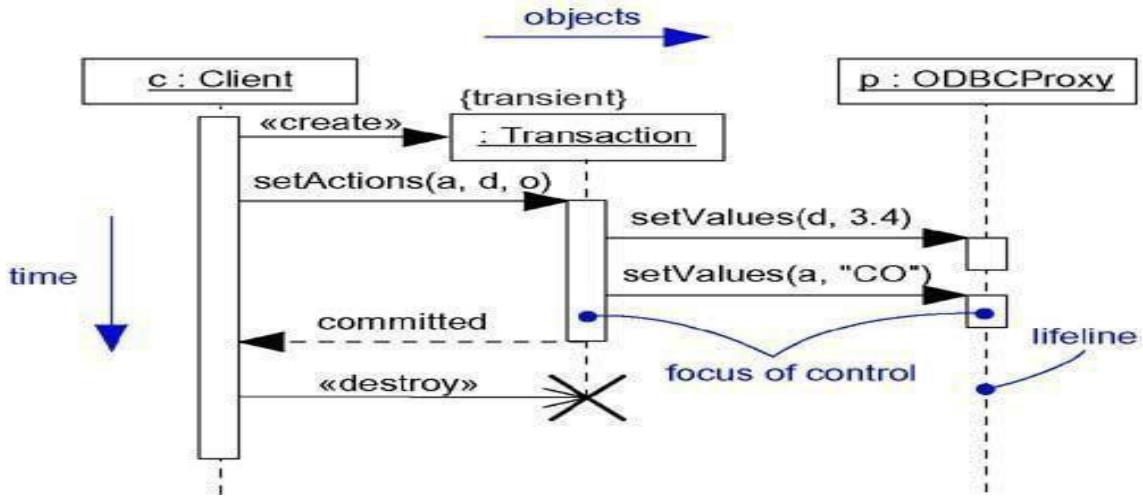
- Objects
- Links
- Messages

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

Sequence Diagrams

A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



sequence diagrams have two features that distinguish them from collaboration diagrams.

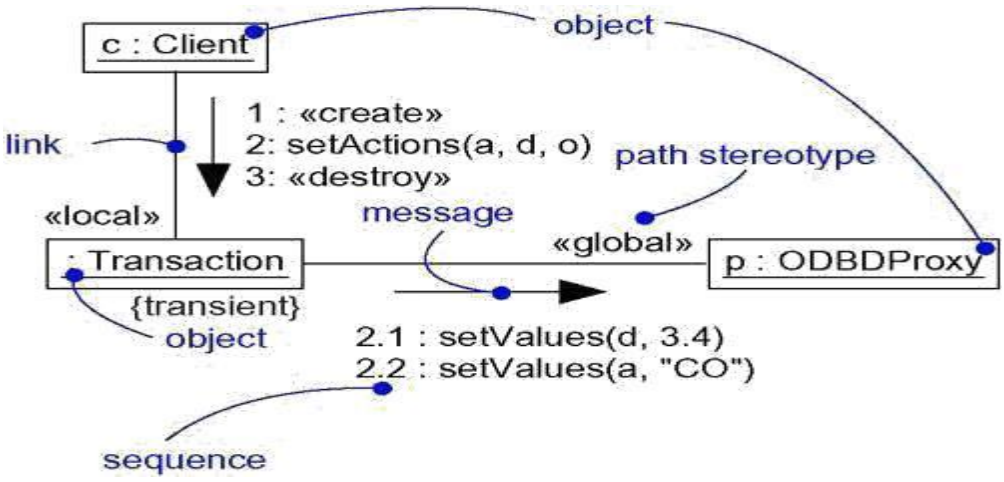
First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create.** Objects may be destroyed during the interaction. Their lifelines end with the receipt ofthe message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self- operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collaboration Diagram



collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is localto the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in

the flow of control (**2**, **3**, and so on). To show nesting, you use Dewey decimal numbering (**1** is the first message; **1.1** is the first message nested in message **1**; **1.2** is the second message nested in message **1** ; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as ***[i := 1..n]** (or just ***** if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as **[x > 0].** The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, thesequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

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

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

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

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instaes in the interaction, along which messages may be

assed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

To model a flow of control by time ordering,

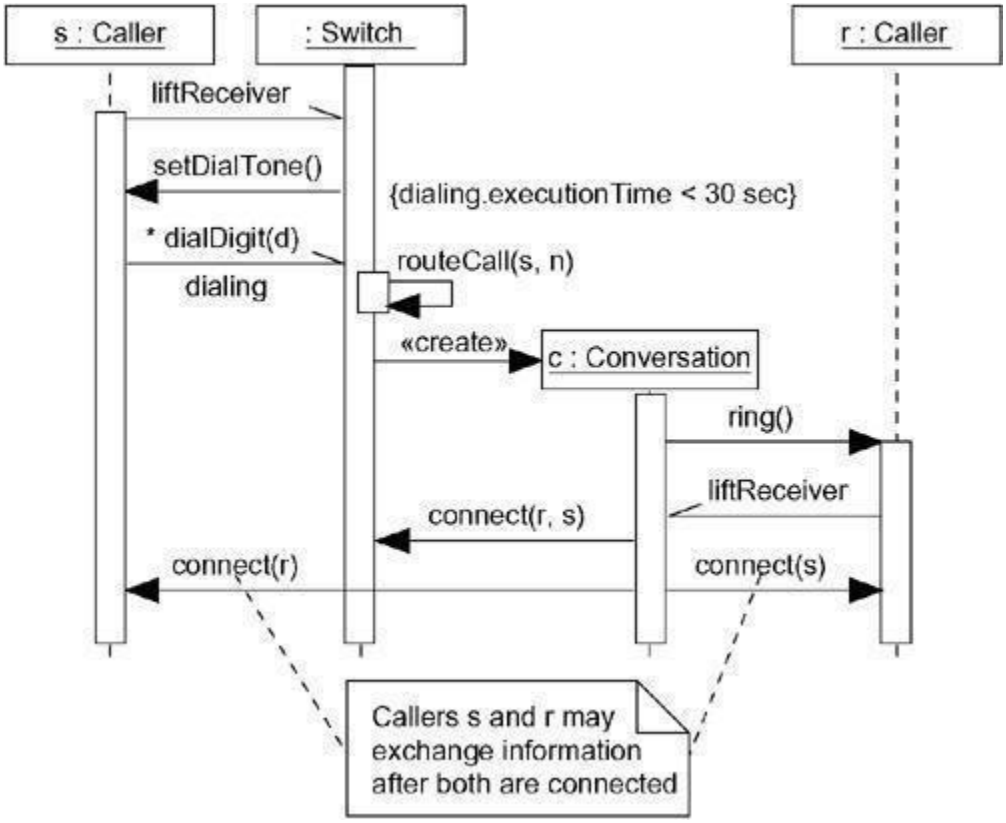
- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, ~~Figure~~ shows a sequence diagram that specifies the fslow of control involvedin initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Caller**s (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatchinga signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message**dialDigit.** Note that this message has atiming mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall.** It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver.** The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**,

after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message

stereotyped as **become** or **copy** (with a suitable sequence number).

- Specify the links among these objects, along which messages may pass.

    1. Lay out the association links first; these are the most important ones, because they represent structural connections.

    2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to oneanother.

- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey decimal numbering.

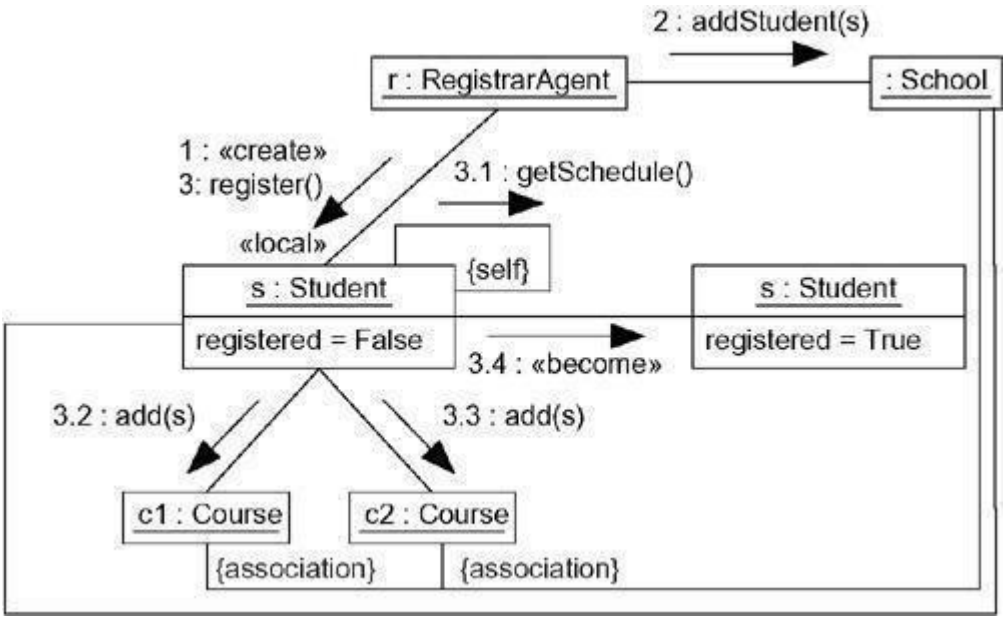
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.

- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (**r**), a **Student** (**s**), two **Course** objects (**c1** and **c2**), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes**getSchedule**on itself, presumably obtaining the **Course** objectsfor which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with **s** rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself.**s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to**getSchedule**(which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence andcollaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =
    getSchedule(); for (inti = 0; i<c.size(); i++)
        c.item(i).add(this);
    this.registered = true;
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection**object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence andcollaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

Use Cases

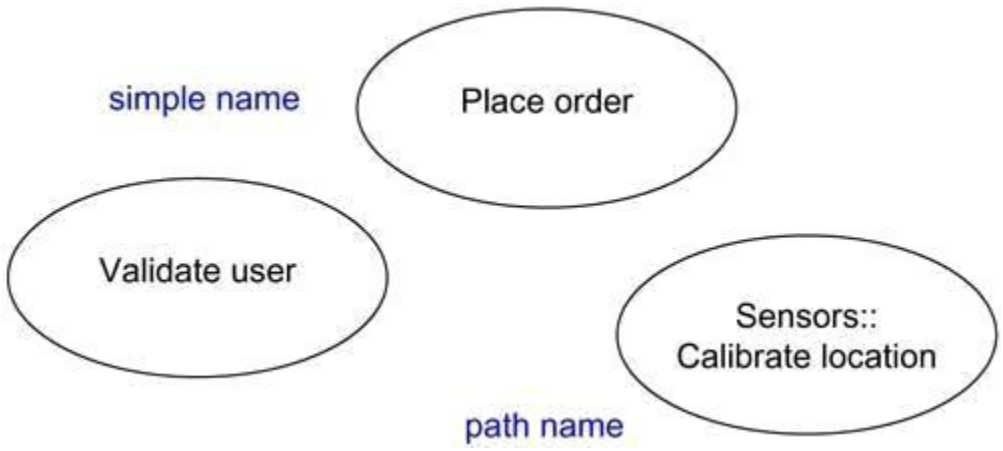
Terms and Concepts

A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor. Graphically, a use case is rendered as an ellipse.

Names

Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the use case name prefixed by the name of the package in which that use case lives. A use case is typically drawn showing only its name, as in Figure.

Figure Simple and Path Names



Note

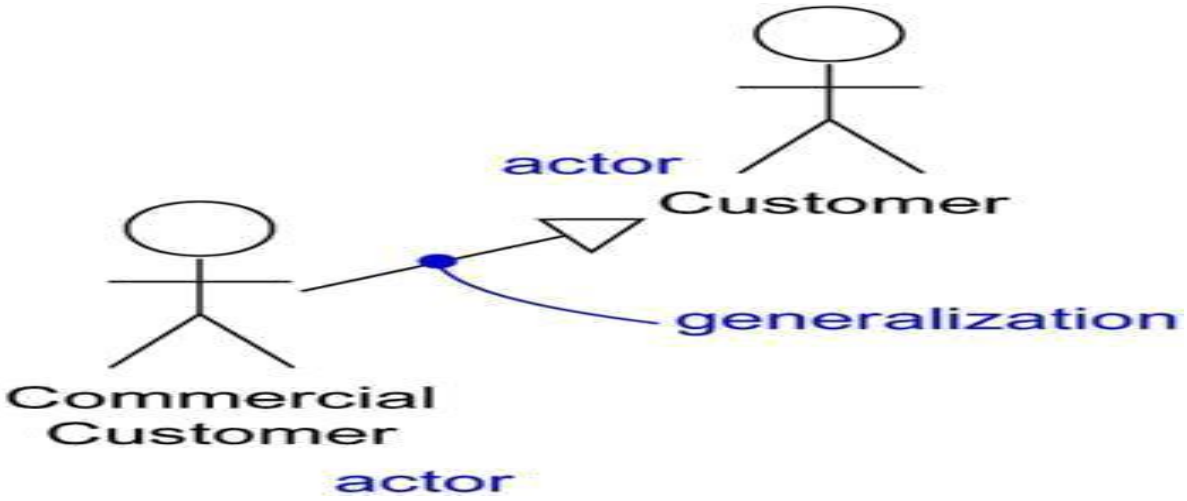
A use case name may be text consisting of any number of letters, numbers, and most punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, use case names are short active verb phrases naming some behavior found in the vocabulary of the system you are modeling.

Use Cases and Actors

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. For example, if you work for a bank, you might be a **LoanOfficer.** If you do your personal banking there, as well, you'll also play the role of **Customer.** An instance of an actor, therefore, represents an individual interacting with thesystem in a specific way. Although you'll use actors in your models, actors are not actually part of the system. They live outside the system.

As Figure indicates, actors are rendered as stick figures. You can define general kinds of actors (such as **Customer**) and specialize them (such as **CommercialCustomer**) using generalization relationships.

Figure Actors



Actors may be connected to use cases only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily. When you write this flow of events, you should include how and when the use case starts and ends, when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

For example, in the context of an ATM system, you might describe the use case **ValidateUser** in the following way:

> **Main flow of events:**
>
> The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.
>
> **Exceptional flow of events:**
>
> The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer*'s account.
>
> **Exceptional flow of events:**

The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

Use Cases and Scenarios

Typically, you'll first describe the flow of events for a use case in text. As you refine your understanding of your system's requirements, however, you'll want to also use interaction diagrams to specify these flows graphically. Typically, you'll use one sequence diagram to specify a use case's main flow, and variations of that diagram to specify a use case's exceptional flows.

It is desirable to separate main versus alternative flows because a use case describes a set of sequences, not just a single sequence, and it would be impossible to express all the details of an interesting use case in just one sequence. For example, in a human resources system, you might find the use case **Hire employee.** This general business function might have many possible variations. You might hire a person from another company (the most common scenario); you might transfer a person from one division to another (common in international companies); or you might hire a foreign national (which involves its own special rules). Each of these variants can be expressed in a different sequence.

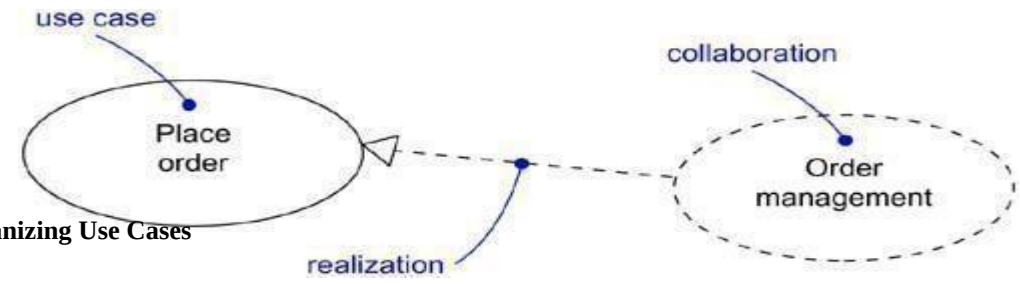
This one use case (**Hire employee**) actually describes a set of sequences in which each sequence in the set represents one possible flow through all these variations. Each sequence is called a scenario. A scenario is a specific sequence of actions that illustrates behavior. Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented. That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out). Ultimately, however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration.

As Figure shows, you can explicitly specify the realization of a use case by a collaboration. Most of the time, though, a given use case is realized by exactly one collaboration, so you will not need to model this relationship explicitly.

Figure Use Cases and Collaborations



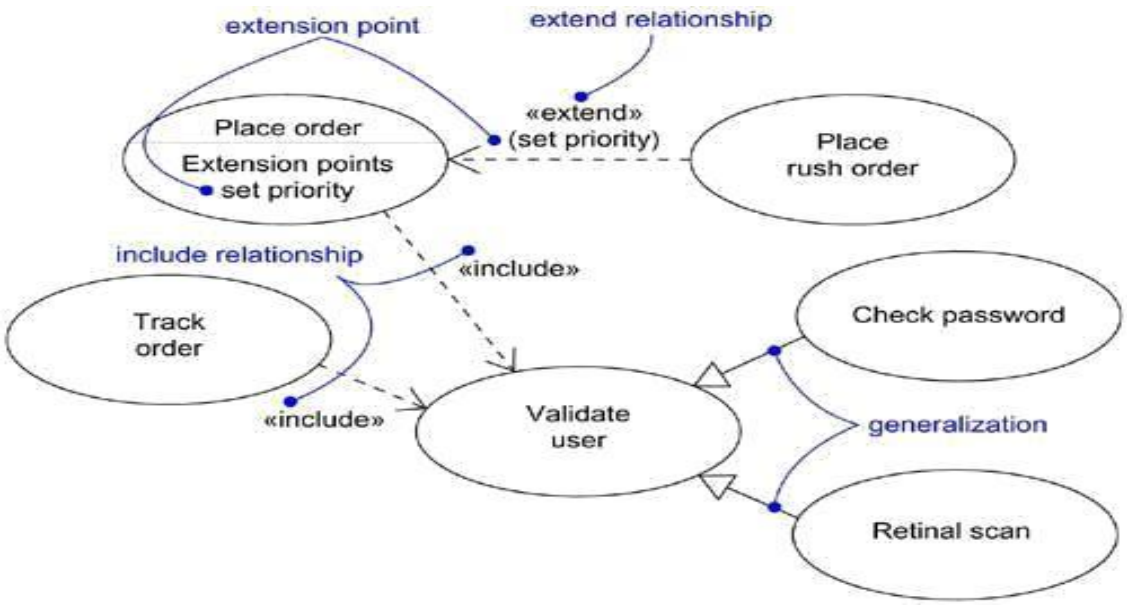
Organizing Use Cases

You can organize use cases by grouping them in packages in the same manner in which you can organize classes.

You can also organize use cases by specifying generalization, include, and extend relationships among them. You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).

Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances). For example, in a banking system, you might have the use case **Validate User**, which is responsible for verifying the identify of the user. You might then have two specialized children of this use case (**Check password** and **Retinal scan**), both of which behave just like **Validate User** and may be applied anywhere **Validate User** appears, yet both of which add their own behavior (the former by checking atextual password, the latter by checking the unique retina patterns of the user). As shown in Figure , generalization among use cases is rendered as a solid directed line with a largeopen arrowhead, just like generalization among classes.

Figure Generalization, Include, and Extend



In include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. You can think of include as the base use case pulling behavior from the supplier use case.

You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own (the use case that is included by a base use case). The include **relationship is essentially an example of delegation• you take a set of responsibilities of the system and** capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.

You render an include relationship as a dependency, stereotyped as **include.** To specify the location in a flow of events in which the base use case includes the behavior of another, you simply write **include** followed by the name of the use case you want to include, as in the following flow for **Track order.**

> **Main flow of events:**
>
> Obtain and verify the order number. **include** (**Validate user**). For each part in the order, query its status, then report back to the user.

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case. This base use case may be extended only at certain points called, not surprisingly, its extension points. You can think of extend as the extension use case pushing behavior to the base use case You use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, you separate optional behavior from mandatory behavior. You may also use an extend relationship to model a separate subflow that is executed only under given conditions. Finally, you may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.

You render an extend relationship as a dependency, stereotyped as **extend.** You may list the extension points of the base use case in an extra compartment. These extension points are just labels that may appear in the flow of the base use case. For example, the flow for **Place order** might read as follows:

> **Main flow of events:**
>
> **include**(**Validate user**). Collect the user's order items. **(set priority).**
> Submit the order for processing.

In this example, **set priority** is an extension point. A use case may have more than one extension point (which may appear more than once), and these are always matched by name. Under normal circumstances, this base use case will execute without regard for the priority of the order. If, on the other hand, this is an instance of a priority order, the flow for this base case will carry out as above. But at the extension point **(set priority)**, the behavior of the extending use case (**Place rush order**) will be performed, then the flow will resume. If there are multiple extension points, the extending use case will simply fold in its flows in order.

Other Features

Use cases are classifiers, so they may have attributes and operations that you may render just as for classes. You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events. These objects and operations may be used in your interaction diagrams to specify the behavior of the use case.As classifiers, you can also attach state machines to use cases. You can use state machines as yet another way to describe the behavior represented by a use case.

Common Modeling Techniques

Modeling the Behavior of an Element

The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class. When you model the behavior of these things, it's important that you focus on what that element does, not how it does it.

Applying use cases to elements in this way is important for three reasons. First, by modeling the behavior

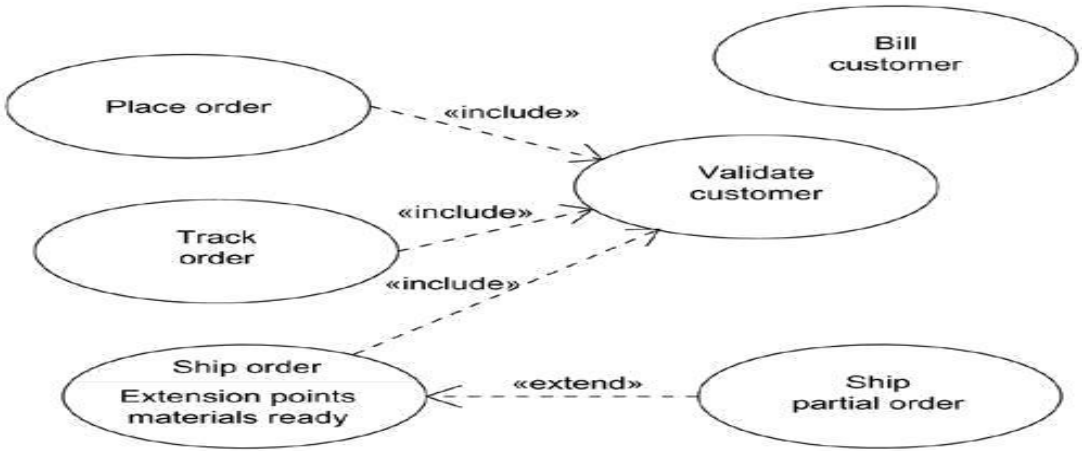
of an element with use cases, you provide a way for domain experts to specify its outside view to a degree sufficient for developers to construct its inside view. Use cases provide a forum for your domain experts, end users, and developers to communicate to one another. Second, use cases provide a way for developers to approach an element and understand it. A system, subsystem, or class may be complex and full of operations and other parts. By specifying an element's use cases, you help users of these elements to approach them in a direct way, according to how they are likely to use them. In the absence of such use cases, users have to discover on their own how to use those elements. Use cases let the author of an element communicate his or her intent about how that element should be used. Third, use cases serve as the basis for testing each element as it evolves during development. By continuously testing each element against its use cases, you continuously validate its implementation. Not only do these use cases provide a source of regression tests, but every time you throw a new use case at an element, you are forced to reconsider your implementation to ensure that this element is resilient to change. If it is not, you must fix your architecture appropriately.

To model the behavior of an element,

- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
- Organize actors by identifying general and more specialized roles.
- For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.
- Consider also the exceptional ways in which each actor interacts with the element.
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.

For example, a retail system will interact with customers who place and track orders. In turn, the system will ship orders and bill the customer. As Figure shows, you can model the behavior of such a system by declaring these behaviors as use cases (**Place order**, **Trackorder**, **Ship order**, and **Bill customer**). Common behavior can be factored out (**Validate custome**r) and variants (**Ship partial order**) can be distinguished, as well. For each ofthese use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

Figure Modeling the Behavior of an Element



Use Case Diagrams

Terms and Concepts

A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

A use case diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams• a name and graphical contents that are a** projection into a model. What distinguishes a use case diagram from all other kinds of diagrams is its particular content.

Contents

Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints.

Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks. Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses

You apply use case diagrams to model the static use case view of a system. This view primarily supports **the behavior of a system• the outwardly visible services that the sy**stem provides in the context of its environment.

When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.

1. To model the context of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it.Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

2. To model the requirements of a system

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

Common Modeling Techniques

Modeling the Context of a System

Given a system• any system• some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that

interact with the system constitute the system's context. This context defines the environment in which that system lives.

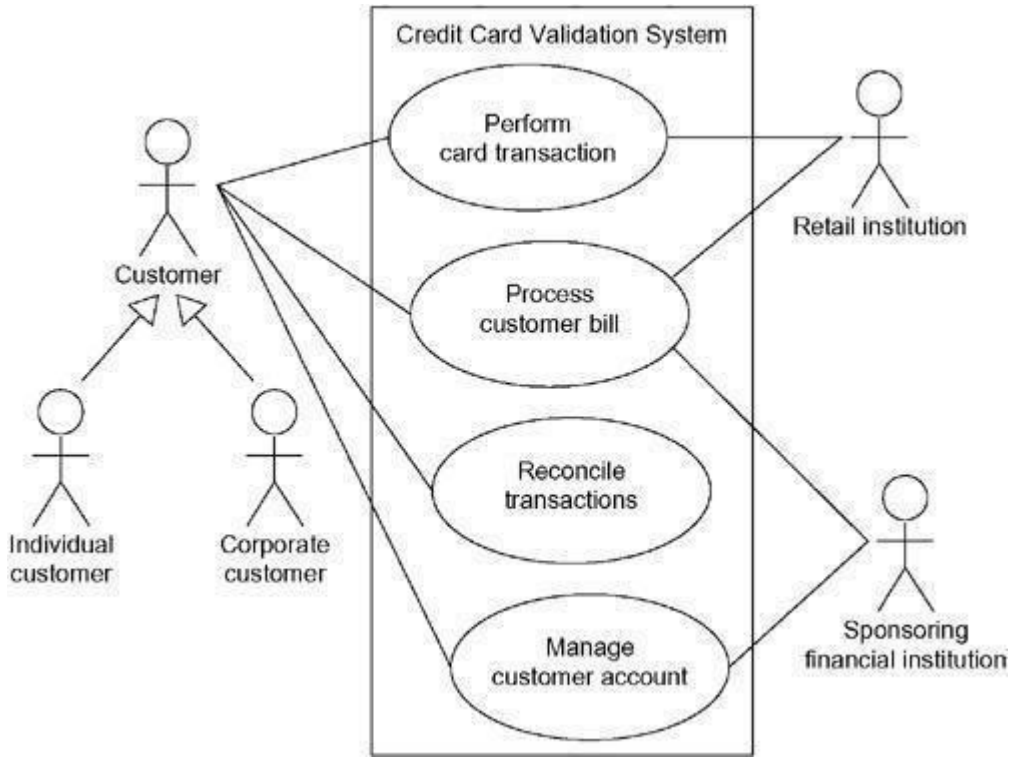
In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system. Deciding what to include as an actor is important because in doing so you specify a class of things that interact with the system. Deciding what not to include as an actor is equally, if not more, important because that constrains the system's environment to include only those actors that are necessary in the life of the system.

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
- Organize actors that are similar to one another in a generalization/specialization hierarchy.
- Where it aids understandability, provide a stereotype for each such actor.
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.

For example, Figure shows the context of a credit card validation system, with anemphasis on the actors that surround the system. You'll find **Customers,** of which there are two kinds (**Individual customer** and **Corporate customer**). These actors are the roles that humans play when interacting with the system. In this context, there are also actors that represent other institutions, such as **Retail institution** (with which a **Customer** performs a card transaction to buy an item or a service) and **Sponsoring financial institution** (which serves as the clearinghouse for the credit card account). In the real world, these latter two actors are likely software-intensive systems themselves.

Figure Modeling the Context of a System



This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

Modeling the Requirements of a System

A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do. For the most part, you don't care how the system does it, you just care *that* it does it. A well-behaved system will carry out all its requirements faithfully, predictably, and reliably. When you build a system, it's important to start with agreement about what that system should do, although you will certainly evolve your understanding of those requirements as you iteratively and incrementally implementthe system. Similarly, when you are handed a system to use, knowing how it behaves is essential to using it properly.

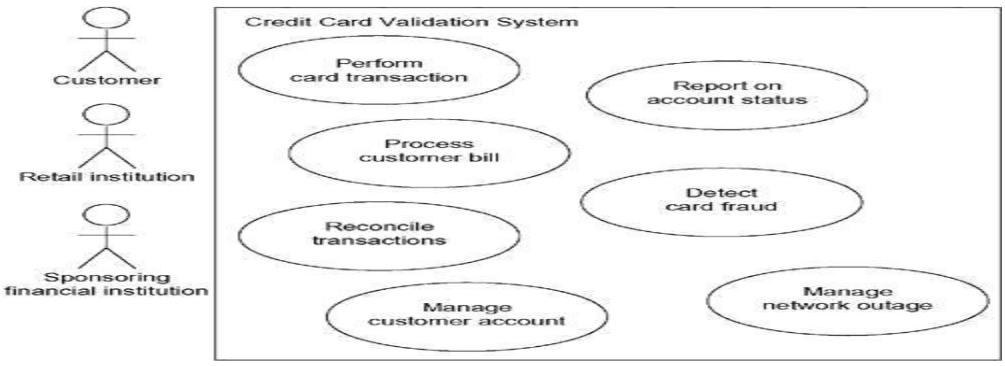
Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between. Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as use cases.
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
- Model these use cases, actors, and their relationships in a use case diagram.
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.

Figure expands on the previous use case diagram. Although it elides the relationshipsamong the actors and the use cases, it adds additional use cases that are somewhat invisible to the average customer, yet are essential behaviors of the system. This diagram is valuable because it offers a common starting place for end users, domain experts, and developers to visualize, specify, construct, and document their decisions about the functional requirements of this system. For example, **Detect card fraud** is a behavior important to both the **Retailinstitution** and the **Sponsoring financial institution.** Similarly, **Report on account status** is another behavior required of the system by the various institutions in itscontext.

Figure Modeling the Requirements of a System



The requirement modeled by the use case **Manage network outage**is a bit different from allthe others because it represents a secondary behavior of the system necessary for its reliable and continuous operation.

This same technique applies to modeling the requirements of a subsystem.

Forward and Reverse Engineering

Most of the UML's other diagrams, including class, component, and statechart diagrams, are clear candidates for forward and reverse engineering because each has an analog in the executable system. Use case diagrams are a bit different in that they reflect rather than specify the implementation of a system, subsystem, or class. Use cases describe how an element behaves, not how that behavior is implemented, so it cannot be directly forward or reverse engineered.

Forward engineering is the process of transforming a model into code through a mapping to animplementation language. A use case diagram can be forward engineered to form tests for the element to which it applies. Each use case in a use case diagram specifies a flow of events (and variants of those fl**ows), and these flows specify how the element is expected to behave• that's something worthy of testing.**

A well-structured use case will even specify pre- and postconditions that can be used to define a test's initial state and its success criteria. For each use case in a use case diagram, you can create a test case that you can run every time you release a new version of that element, thereby confirming that it works as required before other elements rely on it.

To forward engineer a use case diagram,

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
- Use tools to run these tests each time you release the element to which the use case diagram applies.

Reverse engineering is the process of transforming code into a model through a mapping from aspecific implementation language. Automatically reverse engineering a use case diagram is pretty much beyond the state of the art, simply because there is a loss of information when moving from a specification of how an element behaves to how it is implemented. However, you can study an existing system and discern its intended behavior by hand, which you can then put in the form of a use case diagram. Indeed, this is pretty much what you have to do anytime you are handed an undocumented body of software. The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using

extend relationships, and consider modeling common flows by applying include relationships.

- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

Terms and Concepts

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

Common Properties

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

Contents

Activity diagrams commonly contain

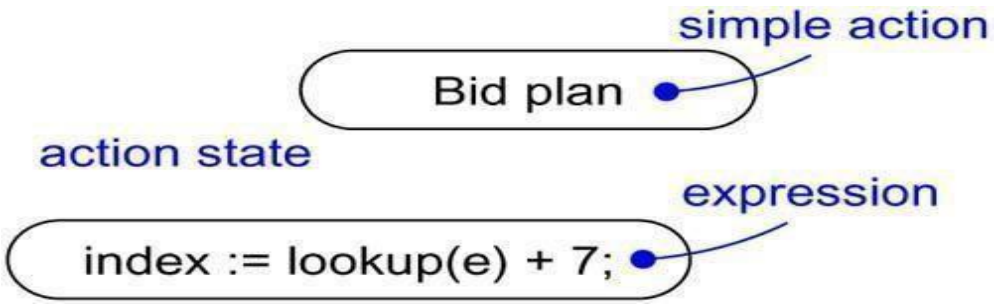
- Activity states and action states
- Transitions
- Objects

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

Action States and Activity States

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

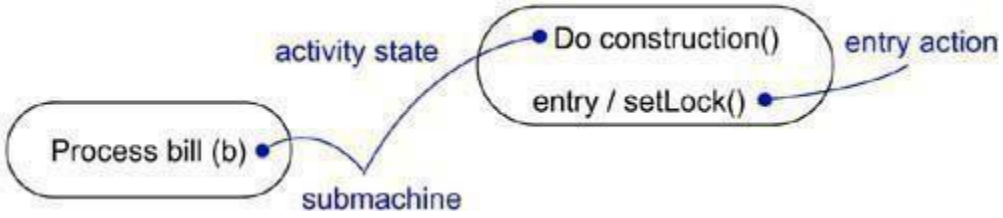
Figure Action States



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

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

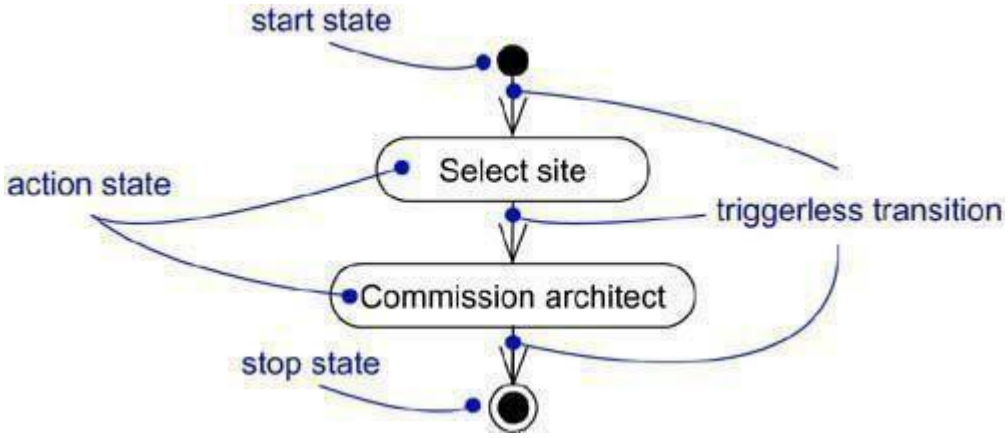
Figure Activity States



Transitions

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

Figure Triggerless Transitions

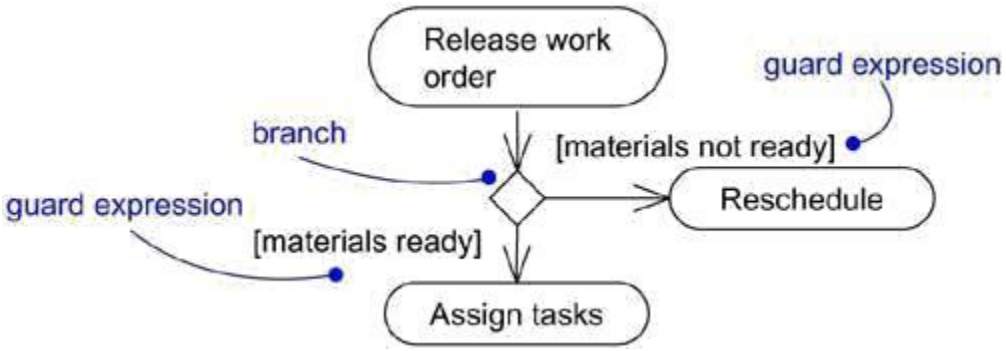


Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in which case it will have a beginning but no end). Therefore, as the figure shows, you may specify this initial state (a solid ball) and stop state (a solid ball inside a circle).

Branching

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

Figure Branching



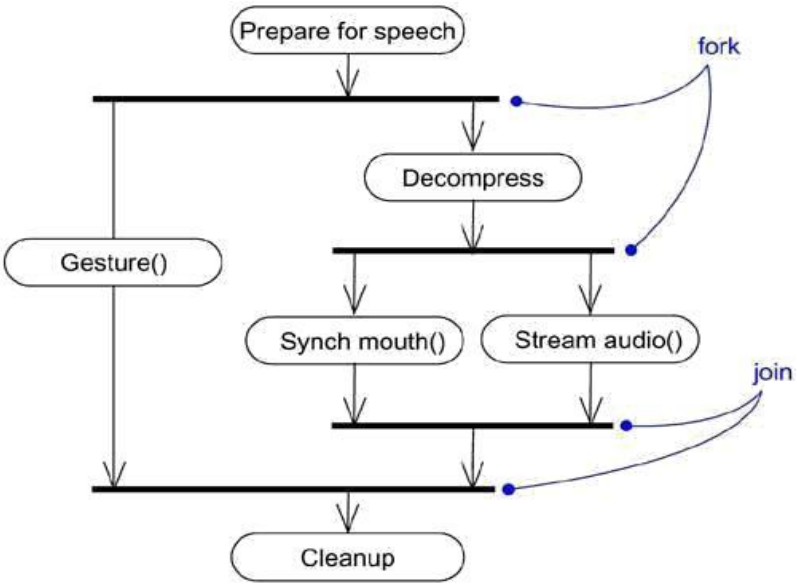
As a convenience, you can use the keyword **else**to mark one outgoing transition, representingthe path taken if no other guard expression evaluates to true.You can achieve the effect of iteration by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished.

Forking and Joining

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

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

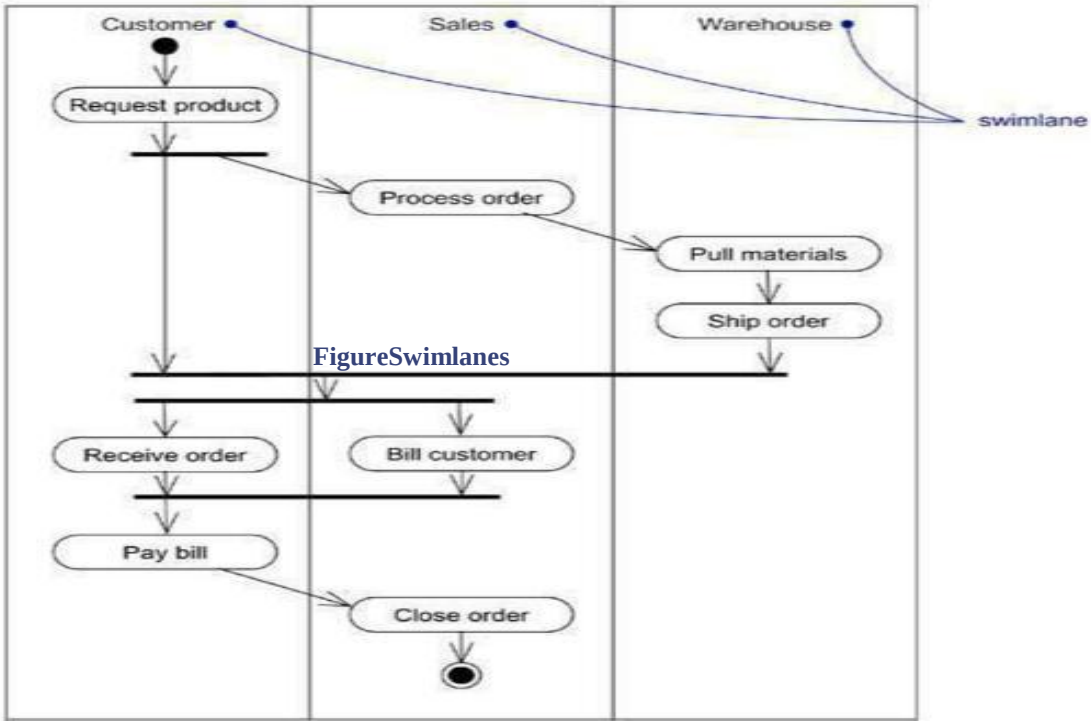
Figure Forking and Joining



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

Swimlanes

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



FigureSwimlanes

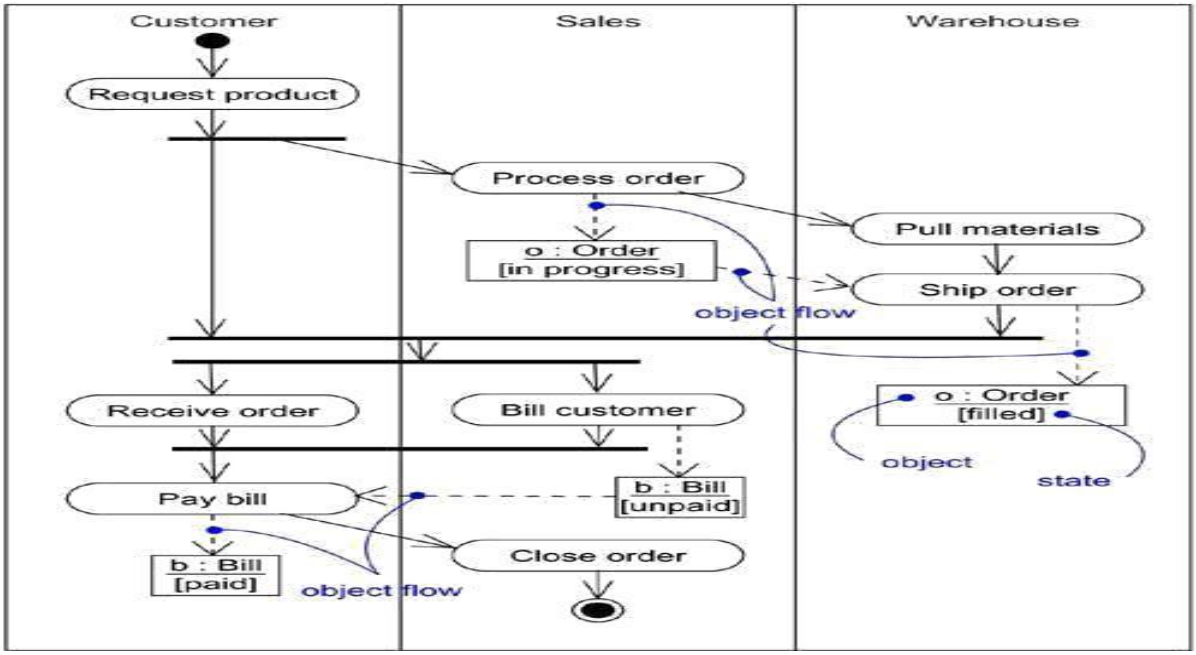
Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane mayeventually be implemented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

Object Flow

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

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

Figure Object Flow



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

Common Uses

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

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

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

1. To model a workflow

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

2. To model an operation

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

Common Modeling Techniques

Modeling a Workflow

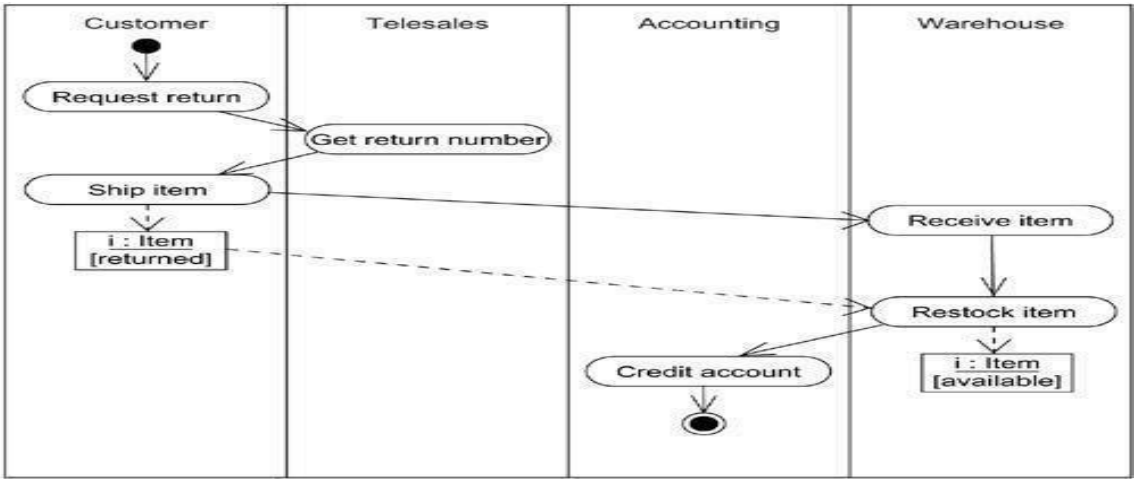
To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

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

the process, changing from the **returned** to the **available** state.

Figure Modeling a Workflow



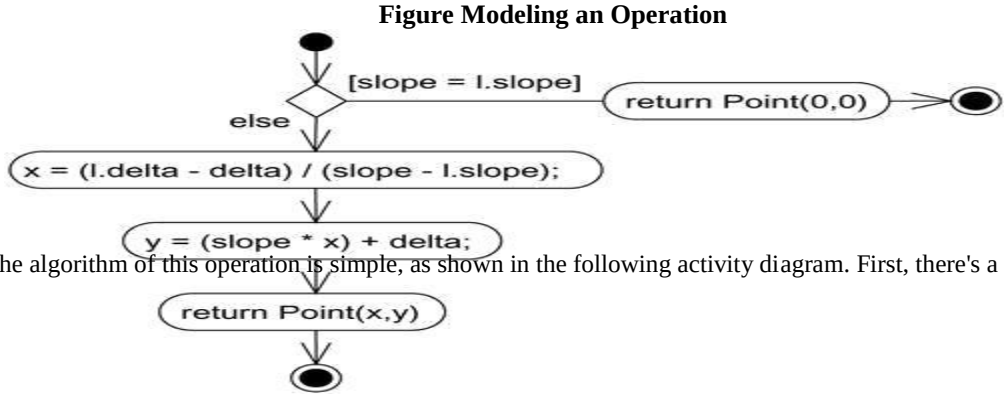
Modeling an Operation

To model an operation,

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

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

Figure Modeling an Operation



The algorithm of this operation is simple, as shown in the following activity diagram. First, there's a guard

that tests whether the **slope** of the current line is the same as the **slope** of parameter **l.** If so, the lines do not intersect, and a **Point** at **(0,0)** is returned. Otherwise, the operationfirst calculates an **x** value for the point of intersection, then a **y** value; **x** and **y**are both objects local to the operation. Finally, a **Point** at **(x,y)** is returned.

Forward and Reverse Engineering

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

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

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

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

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