**UNIT-IV**

**BASIC BEHAVIORAL MODELING-I**

**Basic Behavioral Modeling**: Interactions, Interaction diagrams, Use cases, Use case diagrams, Activity diagrams.

**CHAPTER-1**

**INTERACTIONS**

**Interactions:**

In every interesting system, objects don't just sit idle; they interact with one another by passing messages. An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.

You use interactions to model the dynamic aspect of collaborations, representing societies of objects playing specific roles, all working together to carry out some behavior that's bigger than the sum of the elements.

Their dynamic aspects are visualized, specified, constructed, and documented as flows of control that may encompass simple, sequential threads through a system, as well as more-complex flows that involve branching, looping, recursion, and concurrency.

You can model each interaction in two ways: by emphasizing its time ordering of messages, or by emphasizing its sequencing of messages in the context of some structural organization of objects.

A building is a living thing. Although every building is constructed of static stuff, such as bricks, mortar, lumber, plastic, glass, and steel, those things work together dynamically to carry out behavior that is useful to those who use the building.

Doors and windows open and close. Lights turn on and off. A building's furnace, air conditioner, thermostat, and ventilation ducts work together to regulate the building's temperature. In intelligent buildings, sensors detect the presence or absence of activity and adjust lighting, heating, cooling, and music as conditions change.

Buildings are laid out to facilitate the flow of people and materials from place to place. More subtly, buildings are designed to adapt to changes in temperature, expanding and contracting during the day and night and across the seasons. All well-structured buildings are designed to react to dynamic forces, such as wind, earthquakes, and the movement of its occupants, in ways that keep the building in equilibrium.

Going beyond object diagrams, however, interactions also introduce messages that are dispatched from object to object. Most often, messages involve the invocation of an operation or the sending of a signal; messages may also encompass the creation and destruction of other objects.

You use interactions to model the flow of control within an operation, a class, a component, a use case, or the system as a whole. Using interaction diagrams, you can reason about these flows in two ways. First, you can focus on how messages are dispatched across time. Second, you can focus on the structural relationships among the objects in an interaction and then consider how messages are passed within the context of that structure.

The UML provides a graphical representation of messages, as Figure 15-1 shows. This notation permits you to visualize a message in a way that lets you emphasize its most important parts: its name, parameters (if any), and sequence. Graphically, a message is rendered as a directed line and almost always includes the name of its operation.

**Figure 15-1 Messages, Links, and Sequencing**



**Terms and Concepts:**

An [interaction](http://umlguide2.uw.hu/gloss01.html#gloss01entry88) is a behavior that comprises a set of messages exchanged among objects in a set of roles within a context to accomplish a purpose. A [message](http://umlguide2.uw.hu/gloss01.html#gloss01entry100) is a specification of a Collaboration 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.

You'll also find interactions among objects in the implementation of an operation, in the context of a class. You can use interactions to visualize, specify, construct, and document the semantics

of a class. An interaction may also be found in the representation of a component, node, or use

case, each of which is really a kind of UML classifier

**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.*

**Links and Connectors**

A link is a semantic connection among objects. In general, a link is an instance of an association. As [Figure 15-2](http://umlguide2.uw.hu/ch16lev1sec2.html#ch16fig02) 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.

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 one of the following constraints:

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

**Figure 15-2 Links and Associations**



**Messages**

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.

A message is the specification of a Collaboration among objects that conveys information with the expectation that activity will ensue. The receipt of a message instance may be considered an occurrence of an event. (An [occurrence](http://umlguide2.uw.hu/gloss01.html#gloss01entry117) is the UML name for an instance of an event.)

When you pass a message, an action usually results on its receipt. An action may result in a change in state of the target object and objects accessible from it.

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

Call Invokes an operation on an object; an object may send a message to itself, resulting in

the local invocation of an operation

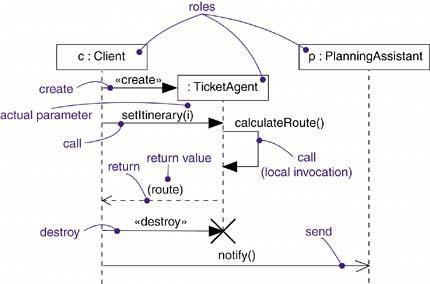
Return Returns a value to the caller

Send Sends a signal to an object

Create Creates an object

Destroy Destroys an object; an object may commit suicide by destroying itself

**Figure 15-3 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), it must also be visible to the caller c.

**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 15-4 shows. In this case, the message findAtis specified as the

first message nested in the second message of the sequence (2.1).

**Figure 15-4 Procedural Sequence**



Less common but also possible, as Figure 15-5 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 15-5 Flat Sequence**



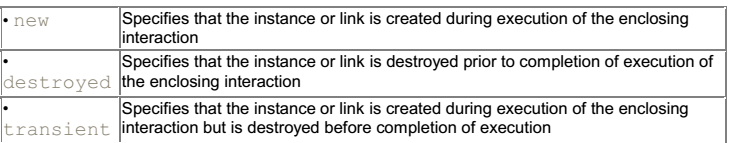
When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process 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.  
1.3.2 : p := find("Rachelle")

**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 a note to its role within a Collaboration diagram.



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 in a sequence diagram by showing the state or the values on the lifeline.

Within a sequence diagram, the lifetime, creation, and destruction of objects or roles are explicitly shown by the vertical extent of their lifelines. Within a Collaboration diagram, creation and destruction must be indicated using notes. Use sequence diagrams if object lifetimes are important to show.

**Representation**

When you model an interaction, you typically include both roles (each one representing objects that appear in an instance of the interaction) and messages (each one representing the Collaboration between objects, with some resulting action).

You can visualize those roles 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 roles 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. (UML also has a more specialized kind of interaction diagram called a timing diagram, which shows the exact times at which messages are exchanged by roles. This diagram is not covered in this book. See the UML Reference Manual for more information.)

Sequence diagrams and Collaboration diagrams are similar, meaning that you can take one and transform it into the other, although they often show different information, so it may not be so useful to go back and forth. There are some visual differences. 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.

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. Name the roles.

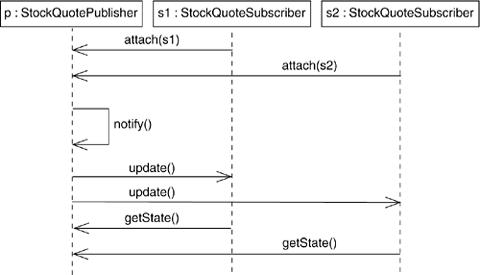
If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of Collaboration that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.

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 15-6 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 emphasizes the time order of messages.

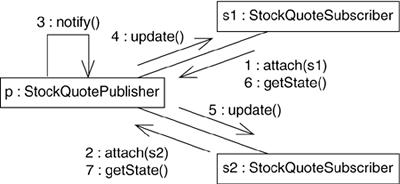
**Figure 15-6 Flow of Control by Time**



The below figure is semantically equivalent to the previous one, but it is drawn as a Collaboration diagram, 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 15-7 is semantically equivalent to the previous one, but it is drawn as a collaboration diagram, 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.

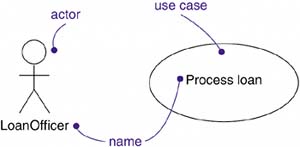
**Figure 15-7 Flow of Control by Organization**



**CHAPTER-2**

**USECASES**

* A use case describes a set of sequences, in which each sequence represents the interaction of the things outside the system (its actors) with the system itself (and its key abstractions).
* A use case involves the interaction of actors and the system. An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Actors can be human or they can be automated systems.
* A use case may have variants.
* A use case carries out some tangible amount of work.
* We can apply use cases to your whole system. We can also apply use cases to part of your system, including subsystems and even individual classes and interfaces.
* The UML provides a graphical representation of a use case and an actor, as Figure 16-1 shows.
* This notation permits you to visualize a use case apart from its realization and in context with other use cases.



**Terms and Concepts**

A [*use case*](http://umlguide2.uw.hu/gloss01.html#gloss01entry189) 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**

A use case name must be unique within its enclosing package, 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 16-2.

**Figure 16-2 Simple and Path Names**



Note

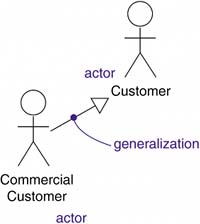
A use case name may be text consisting of any number of letters, numbers, and most punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, use case names are short active verb 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 the system in a specific way. Although you'll use actors in your models, actors are not actually part of the software application. They live outside the application within the surrounding environment.

As [Figure 16-3](http://umlguide2.uw.hu/ch17lev1sec2.html#ch17fig03) 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.



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.

Note

You can specify a use case's flow of events in a number of ways, including informal structured text (as in the example above), formal structured text (with pre- and postconditions), state machines (particularly for reactive systems), activity diagrams (particularly for workflows), and pseudocode.

#### 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.

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.

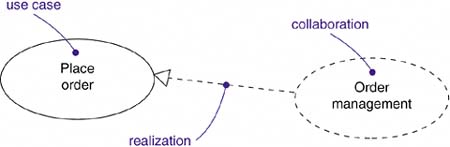
**Note**

There's an expansion factor from use cases to scenarios. A modestly complex system might have a few dozen use cases that capture its behavior, and each use case might expand out to several dozen scenarios. For each use case, you'll find primary scenarios (which define essential sequences) and secondary scenarios (which define alternative sequences).

#### 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. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration

As [Figure 17-4](http://umlguide2.uw.hu/ch17lev1sec2.html#ch17fig04) shows, we 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.



Note

Although you may not visualize this relationship explicitly, the tools you use to manage your models will likely maintain this relationship.

Finding the minimal set of well-structured collaborations that satisfy the flow of events specified in all the use cases of a system is the focus of a system's architecture.

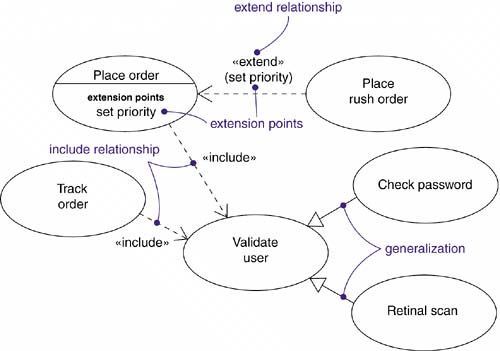
#### Organizing Use Cases

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.

For example, in a banking system, you might have the use case Validate User, which is responsible for verifying the identity 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 a textual password, the latter by checking the unique retina patterns of the user). As shown in [Figure 17-5](http://umlguide2.uw.hu/ch17lev1sec2.html#ch17fig05), generalization among use cases is rendered as a solid directed line with a large triangular arrowhead, just like generalization among classes.



##### Figure 17-5. Generalization, Include, and Extend

An 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.

We 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 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.

We 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:*Track order obtain and verify the order number; include 'Validate user'; for each part in the order, query the order status; report overall status to 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.

We 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.

We 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.

### common Modeling Techniques

#### Modeling the Behavior of an Element

Applying use cases to elements in this way is important for three reasons.

***First,*** by modeling the behavior of an element with use cases, 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. In the absence of such use cases, users have to discover on their own how to use those elements.

***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.

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 17-6](http://umlguide2.uw.hu/ch17lev1sec3.html#ch17fig06) shows, you can model the behavior of such a system by declaring these behaviors as use cases (Place order, track order, Ship order, and Bill customer). Common behavior can be factored out (Validate customer) and variants (Ship partial order) can be distinguished as well. For each of these use cases, you would include a specification of the behaviorr, either by text, state machine, or interactions.

##### Figure 17-6. Modeling the Behavior of an Element

##### 17fig06

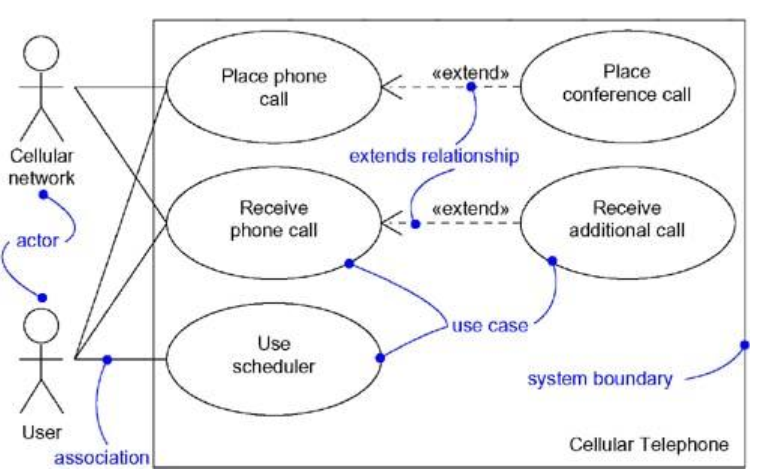
**CHAPTER-3**

**USE CASE DIAGRAMS**

Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. (activity diagrams, statechart diagrams, sequence diagrams, and collaboration diagrams  
are four other kinds of diagrams in the UML). Usecase diagrams are central to modeling the behavior of a system, a subsystem, or a class.

Each one shows a set of use cases and actors and their relationships. We apply use case diagrams to model the use case view of a system. Use case diagrams are important for visualizing, specifying, and documenting the behavior of an element.

##### Figure 18-1. A Use Case Diagram



### Terms and Concepts

A [*use case diagram*](http://umlguide2.uw.hu/gloss01.html#gloss01entry190) 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

* Subject
* 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

#### Notation

The subject is shown as a rectangle containing a set of use case ellipses. The name of the subject is placed within the rectangle. The actors are shown as stick figures placed outside the rectangle; their names are placed under them. Lines connect actor icons to the use case ellipses with which they communicate. Relationships among use cases (such as extend and include) are drawn inside the rectangle.

#### Common Uses

We apply use case diagrams to model the use case view of a subject, such as a system. This view primarily models the external behavior of a system.

When we model the use case view of a subject, we'll typically apply use case diagrams in one of two ways.

1. To model the context of a system

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

1. 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 we 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.

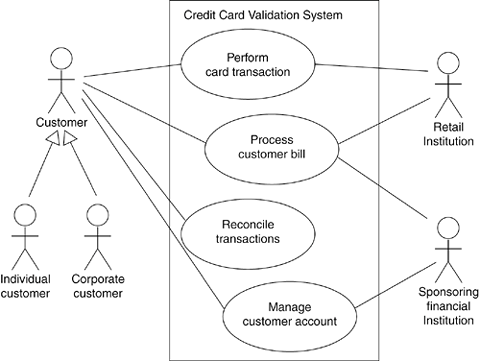
In the UML, you can model the context of a system with a use case diagram, emphasizing the  
actors that surround the system. Deciding what to include as an actor is important because in  
doing so you specify a class of things that interact with the system. Deciding what not to include  
as an actor and 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 18-2](http://umlguide2.uw.hu/ch18lev1sec3.html#ch18fig02) shows the context of a credit card validation system, with an emphasis 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 18-2. Modeling the Context of a System



#### Modeling the Requirements of a System

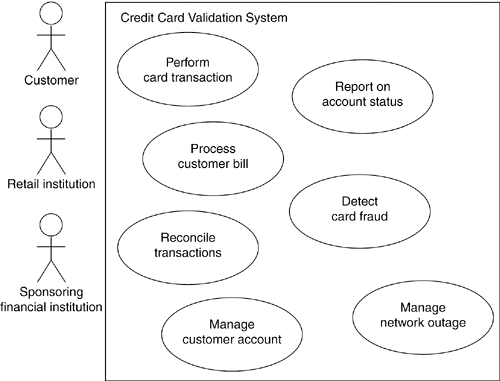
When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself. A well-behaved system will carry out all its requirements faithfully, predictably, and reliably. When you build a system, it's important to start with agreement about what that system should do

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 or constraints that assert nonfunctional requirements; you may have to attach some of these to the whole system.

[Figure 18-3](http://umlguide2.uw.hu/ch18lev1sec3.html#ch18fig03) expands on the previous use case diagram. Although it elides the relationships among 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 Retail institution and the Sponsoring financial institution. Similarly, Report on account status is another behavior required of the system by the various institutions in its context.

##### Figure 18-3. Modeling the Requirements of a System



#### 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*](http://umlguide2.uw.hu/gloss01.html#gloss01entry73) is the process of transforming a model into code through a mapping to an implementation 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 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 post conditions 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*](http://umlguide2.uw.hu/gloss01.html#gloss01entry144) is the process of transforming code into a model through a mapping from a specific implementation language. To automatically reverse engineer a use case diagram is currently 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

**CHAPTER- 4**

**INTERACTION DIAGRAMS**

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

In the UML, you model these storyboards by using interaction diagrams. As Figure 18-1 shows,

you can build up these storyboards in two ways: by emphasizing the time ordering of messages

and by emphasizing the structural relationships among the objects that interact. Either way, the

diagrams are semantically equivalent; you can convert one to the other without loss of

information.

**Figure 18-1 Interaction Diagrams**

****

**Terms and Concepts**

An [interaction diagram](http://umlguide2.uw.hu/gloss01.html#gloss01entry89) shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A [sequence diagram](http://umlguide2.uw.hu/gloss01.html#gloss01entry151) 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](http://umlguide2.uw.hu/gloss01.html#gloss01entry36) 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

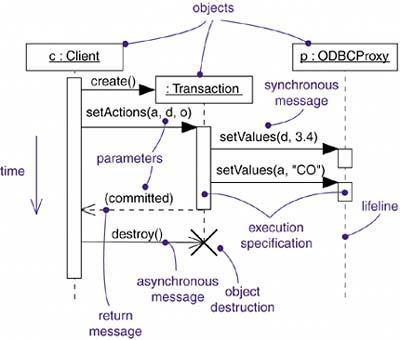
links

Messages



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

**Sequence Diagrams**

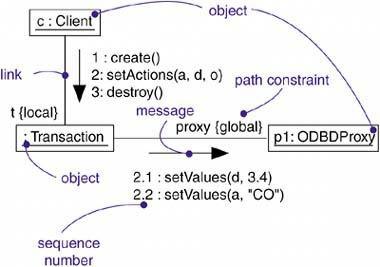
A sequence diagram emphasizes the time ordering of messages. you form a sequence diagram by first placing the objects or roles that participate in the interaction at the top of your diagram, across the horizontal axis. Typically, you place the object or role that initiates the interaction at the left, and increasingly more subordinate objects or roles to the right. Next, you arrange the messages that these objects send and receive along the vertical 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

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

First, there is the 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 create (drawn to box at the head of the lifeline). Objects may be destroyed during the interaction. Their lifelines end with the receipt of the message 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).

**Collaboration Diagrams**

A Collaboration diagram emphasizes the organization of the objects that participate in an interaction. 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. The links may have role names to identify them. 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.

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

First, there is the path. You render a path corresponding to an association. You also render paths corresponding to local variables, parameters, global variables, and self access. A path represents a source of knowledge to an object.

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, which contains message 1.1 and message 1.2, 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.

**Semantic Equivalence**

Because they both derive from the same information in the UML's metamodel, sequence diagrams and communication 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 communication diagram shows how the objects are linked (note the {local} and {global} annotations); the corresponding sequence

diagram does not. Similarly, the sequence diagram shows message return (note the return value *committed*), but the corresponding communication diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not. However, a model entered in one format may lack some of the information shown on the other format, so although the underlying model can include both kinds of information, the two kinds of diagrams may lead to different models.

**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.

We 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, we typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

Here we'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 communication 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.

**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.

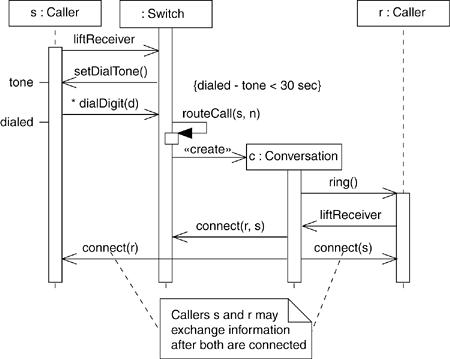


If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

For example, Figure 18-4 shows a sequence diagram that specifies the flow of control involved in initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two Callers (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) dispatching a 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 a timing 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 18-4 Modeling Flows of Control by Time Ordering**



**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 communication 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 communication diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.

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 annotations (such as global and local) to explicitly specify how these objects are related to one another.



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.

For example, Figure 18-5 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 objects for 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 18-5 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](http://umlguide2.uw.hu/gloss01.html#gloss01entry73) (the creation of code from a model) is possible for both sequence and collaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous communication 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 courses = getSchedule(); for (int i = 0; i < courses.size(); i++) courses.item(i).add(this);

this.registered = true;

}

[Reverse engineering](http://umlguide2.uw.hu/gloss01.html#gloss01entry144) (the creation of a model from code) is also possible for both sequence and collaboration 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.

**CHAPTER- 5**

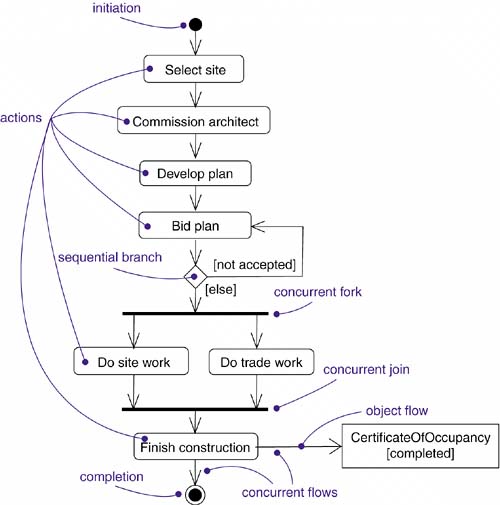
**ACTIVITY DIAGRAMS**

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity. Unlike a traditional flowchart, an activity diagram shows concurrency as well as branches of control.

Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. An activity is an ongoing nonatomic execution within a state machine.

We can model these dynamic aspects using activity diagrams, which focus first on the activities that take place among objects, as Figure 19-1 shows. An activity diagram is essentially a flowchart that emphasizes the activity that takes place over time. You can think of an activity diagram as an interaction diagram turned inside out.

##### Figure 20-1. Activity Diagram

****

### Terms and Concepts

An [*activity diagram*](http://umlguide2.uw.hu/gloss01.html#gloss01entry07) shows the flow from activity to activity. An [*activity*](http://umlguide2.uw.hu/gloss01.html#gloss01entry06) is an ongoing nonatomic execution within a state machine. The execution of an activity ultimately expands into the execution of individual [*actions*](http://umlguide2.uw.hu/gloss01.html#gloss01entry03), each of which may change the state of the system or communicate messages. 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 a kind of diagram and shares the same common properties as do all other diagramsa name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from other kinds of diagrams is its content.

#### Contents

#### Activity diagrams commonly contain

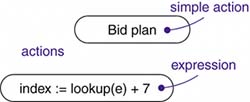
* Actions states
* Activity states
* Transistions
* Objects

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 actions. As [Figure 20-2](http://umlguide2.uw.hu/ch20lev1sec2.html#ch20fig02) shows, you represent an action using a rounded box. Inside that shape, you may write an expression.

##### Figure 20-2. Action states



Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the internal behavior of the action state is not visible. You can't execute part of an action; either it executes completely or not at all. Finally, the work of an action state is often 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 19-3 shows, there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit actions (actions which are involved on entering and leaving the state, respectively) and submachine specifications.

##### Figure 19-3 Activity States



**Note**

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

**Transitions**.

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

**Figure 19-4 Triggerless Transitions**

##### 

**Note**

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

Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in

which case it will have a beginning but no end). Therefore, as the figure shows, you may specify

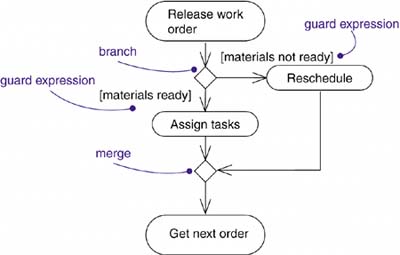
this initial state (a solid ball) and stop state (a solid ball inside a circle).

#### Branching

Simple, sequential flows 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 20-5](http://umlguide2.uw.hu/ch20lev1sec2.html#ch20fig05) shows, you represent a branch as a diamond.

A branch may have one incoming and two or more outgoing flows. On each outgoing flow, you place a Boolean expression, which is evaluated on entering the branch. The guards on the outgoing flows 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 20-5. Branching



**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 processes• you

might encounter flows that are concurrent.

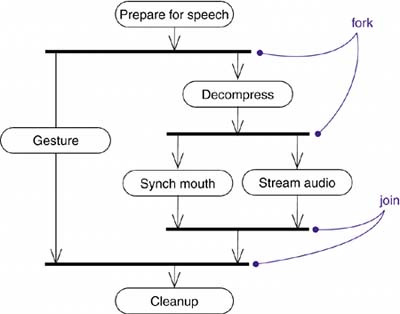
In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

For example, consider the concurrent flows involved in controlling an audio-animatronic device

that mimics human speech and gestures. As Figure 19-6 shows, a fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents anindependent 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 20-6. 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.

**Note**

Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join. Also, activities that are in parallel flows of control may communicate with one another by sending signals

#### Swimlanes

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

Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics,

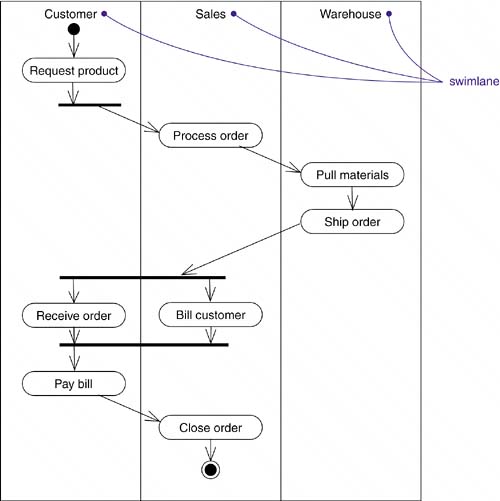
except that it may represent some real-world entity. Each swimlane represents a high-level

responsibility for part of the overall activity of an activity diagram, and each swimlane may

eventually be implemented by one or more classes. In an activity diagram partitioned into

swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

##### Figure 20-7. Swimlanes



#### 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 use or modify these objects (for example, Ship order will change the state of the Order object to filled).

As Figure 20-8 shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.

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.

##### Figure 20-8. Object Flow



#### 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.

We'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.

1. To model an operation

Here we'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

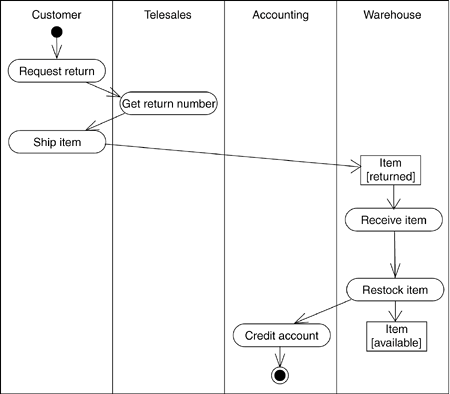
To model a workflow,

* 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 or organization.
* 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 actions that take place over time and render them in the activity diagram.
* For complicated actions or for sets of actions that appear multiple times, collapse these into calls to a separate activity diagram.
* Render the flows that connect these actions and activity nodes. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
* If there are important object values 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 20-10](http://umlguide2.uw.hu/ch20lev1sec3.html#ch20fig10) shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order. Work starts with the Customer action Request return and then flows through Telesales (Get return number), back to the Customer (Ship item), then to the Warehouse (Receive item then Restock item), finally ending in Accounting (Credit account). As the diagram indicates, one significant object (an instance of Item) also flows the process, changing from the returned to the available state.

##### 

##### Figure 20-10. Modeling a Workflow



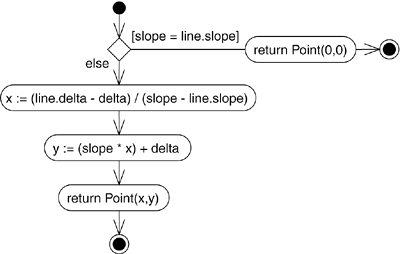
#### Modeling an Operation

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 20-11](http://umlguide2.uw.hu/ch20lev1sec3.html#ch20fig11) shows an activity diagram that specifies the algorithm of the operation intersection, whose signature includes one parameter (line, 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 20-11. Modeling an Operation



**Forward and Reverse Engineering**

[*Forward engineering*](http://umlguide2.uw.hu/gloss01.html#gloss01entry73) (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 (line : Line) {

if (slope == line.slope) return Point(0,0);

int x = (line.delta - delta) /

(slope - line.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*](http://umlguide2.uw.hu/gloss01.html#gloss01entry144) (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.