UNIT-II Classes-9

State Modelling: Events, States, Transitions and Conditions, State diagrams, State diagram behaviour. Need, purpose Advanced object and class concepts, Use Case Modelling: Actor Identification, Actor Classification, Actor Generalization, Use Cases Identification, Communication, Uses/Include and Extend Associations, writing a Formal Use Cases, Use Case realizations. Need, purpose

Events

An **event** is an occurrence at a point in time, such as user depresses left button or flight 123 departs from Chicago. Events often correspond to verbs in the past tense (power turned on, alarm set) or to the onset of some condition (paper tray becomes empty, temperature becomes lower than freezing). By definition, an event happens instantaneously with regard to the time scale of an application. Of course, nothing is really instantaneous; an event is simply an occurrence that an application considers atomic and fleeting. The time at which an event occurs is an implicit attribute of the event. Temporal phenomena that occur over an interval of time are properly modeled with a state. One event may logically precede or follow another, or the two events may be unrelated. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after flight 456 departs Rome; the two events are causally unrelated. Two events that are causally unrelated are said to be **concurrent**; they have no effect on each other. If the communications delay between two locations exceeds the

difference in event times, then the events must be concurrent because they cannot influence each other. Even if the physical locations of two events are not distant, we consider the events concurrent if they do not affect each other. In modeling a system we do not try to establish an ordering between concurrent events because they can occur in any order.

Events include error conditions as well as normal occurrences. For example, motor

jammed, transaction aborted, and timeout are typical error events. There is nothing different about an error event; only our interpretation makes it an “error.” The term event is often used ambiguously. Sometimes it refers to an instance, at other times to a class. In practice, this ambiguity is usually not a problem and the precise meaning is apparent from the context. If necessary, you can say event occurrence or event type to be precise. There are several kinds of events. The most common are the signal event, the change event, and the time event.

Signal Event

A **signal** is an explicit one-way transmission of information from one object to another. It is different from a subroutine call that returns a value. An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.

A **signal event** is the event of sending or receiving a signal. Usually we are more concerned

about the receipt of a signal, because it causes effects in the receiving object. Note

the difference between signal and signal event—a signal is a message between objects while a signal event is an occurrence in time. Every signal transmission is a unique occurrence, but we group them into signal classes and give each signal class a name to indicate common structure and behavior. For example, UA flight 123 departs from Chicago on January 10, 1991 is an instance of signal class Flight- Departure. Some signals are simple occurrences, but most signal classes have attributes indicating

the values they convey. For example, as Figure 5.1 shows, FlightDeparture has

attributes airline, flightNumber, city, and date. The UML notation is the keyword signal in guillemets («») above the signal class name in the top section of a box. The second section lists the signal attributes

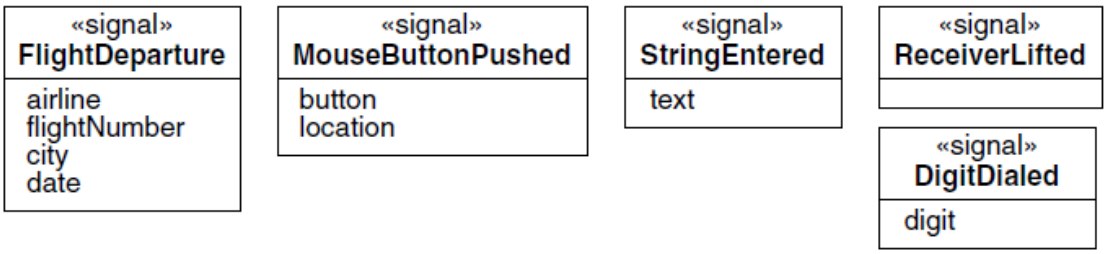


Figure 5.1 Signal classes and attributes. A signal is an explicit one-way transmission of information from one object to another.

States

A **state** is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behavior of objects. For example, the state of a bank is either solvent or insolvent, depending on whether its assets exceed its liabilities. States often correspond to verbs with a suffix of "ing" (Waiting, Dialing) or the duration of some condition (Powered, BelowFreezing).

Figure 5.4 shows the UML notation for a state—a rounded box containing an optional

state name. Our convention is to list the state name in boldface, center the name near the top of the box, and capitalize the first letter.

Figure 5.4 States. A state is an abstraction of the values and links of an object.

In defining states, we ignore attributes that do not affect the behavior of the object, and lump together in a single state all combinations of values and links with the same response to events. Of course, every attribute has some effect on behavior or it would be meaningless, but often some attributes do not affect the sequence of control and you can regard them as simple parameter values within a state. Recall that the purpose of modeling is to focus on qualities that are relevant to the solution of an

application problem and abstract away those that are irrelevant. The three UML models (class, state, and interaction) present different views of a system for which the particular choice of attributes and values are not equally important. For example, except for leading 0s and 1s, the exact digits dialed do not affect the control of the phone line, so we can summarize them all with state Dialing and track the phone number as a parameter. Sometimes, all possible values of an attribute are important,

but usually only when the number of possible values is small.

The objects in a class have a finite number of possible states—one or possibly some larger number. Each object can only be in one state at a time. Objects may parade through one or more states during their lifetime. At a given moment of time, the various objects for a class can exist in a multitude of states.

A state specifies the response of an object to input events. All events are ignored in a state, except those for which behavior is explicitly prescribed. The response may include the invocation of behavior or a change of state. For example, if a digit is dialed in state Dial tone, the phone line drops the dial tone and enters state Dialing; if the receiver is replaced in state Dial tone, the phone line goes dead and enters state Idle.

There is a certain symmetry between events and states as Figure 5.5 illustrates. Events represent points in time; states represent intervals of time. A state corresponds to the interval between two events received by an object. For example, after the receiver is lifted and before the first digit is dialed, the phone line is in state Dial tone. The state of an object depends on past events, which in most cases are eventually hidden by subsequent events. For example,

events that happened before the phone is hung up do not affect future behavior; the Idle state "forgets" events received prior to the receipt of the hang up signal.

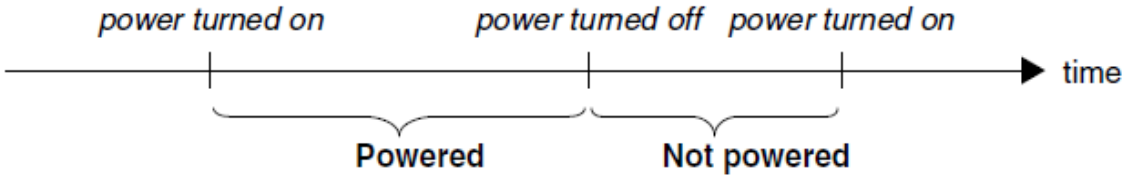


Figure 5.5 Event vs. state. Events represent points in time; states represent intervals of time.

Both events and states depend on the level of abstraction. For example, a travel agent

planning an itinerary would treat each segment of a journey as a single event; a flight status board in an airport would distinguish departures and arrivals; an air traffic control system would break each flight into many geographical legs.

You can characterize a state in various ways, as Figure 5.6 shows for the state Alarm

ringing on a watch. The state has a suggestive name and a natural-language description of its purpose. The event sequence that leads to the state consists of setting the alarm, doing anything that doesn't clear the alarm, and then having the target time occur. A declarative condition for the state is given in terms of parameters, such as current and target time; the alarm stops ringing after 20 seconds. Finally, a stimulus-response table shows the effect of events current time and button pushed,

including the response that occurs and the next state. The different descriptions of a state may overlap.

State: *AlarmRinging*

Description: alarm on watch is ringing to indicate target time

Event sequence that produces the state:

setAlarm (*targetTime*)
any sequence not including *clearAlarm*
when (*currentTime* = *targetTime*)

Condition that characterizes the state:

alarm = on, alarm set to *targetTime*, *targetTime* ≤ *currentTime* ≤ *targetTime* + 20 seconds, and no button has been pushed since *targetTime*

Events accepted in the state:

event	response	next state
when (*currentTime* = *targetTime* + 20)	*resetAlarm*	*normal*
buttonPushed (any button)	*resetAlarm*	*normal*

Figure 5.6 Various characterizations of a state. A state specifies the response of an object to input events.

Transitions and Conditions

A **transition** is an instantaneous change from one state to another. For example, when a called phone is answered, the phone line transitions from the Ringing state to the Connected state. The transition is said to **fire** upon the change from the source state to the target state.The origin and target of a transition usually are different states, but may be the same. A transition fires when its event occurs (unless an optional guard condition causes the event to be ignored). The choice of next state depends on both the original state and the event received.

An event may cause multiple objects to transition; from a conceptual point of view such transitions occur concurrently.

A **guard condition** is a boolean expression that must be true in order for a transition to occur. For example, a traffic light at an intersection may change only if a road has cars waiting. A guarded transition fires when its event occurs, but only if the guard condition is true.For example, “when you go out in the morning (event), if the temperature is below freezing (condition), then put on your gloves (next state).” A guard condition is checked only once,at the time the event occurs, and the transition fires if the condition is true. If the condition becomes true later, the transition does not then fire. Note that a guard condition is different from a change event—a guard condition is checked only once while a change event is, in effect, checked continuously.

Figure 5.7 shows guarded transitions for traffic lights at an intersection. One pair of electric eyes checks the north-south left turn lanes; another pair checks the east-west turn lanes.

If no car is in the north-south and/or east-west turn lanes, then the traffic light control logic is smart enough to skip the left turn portion of the cycle.

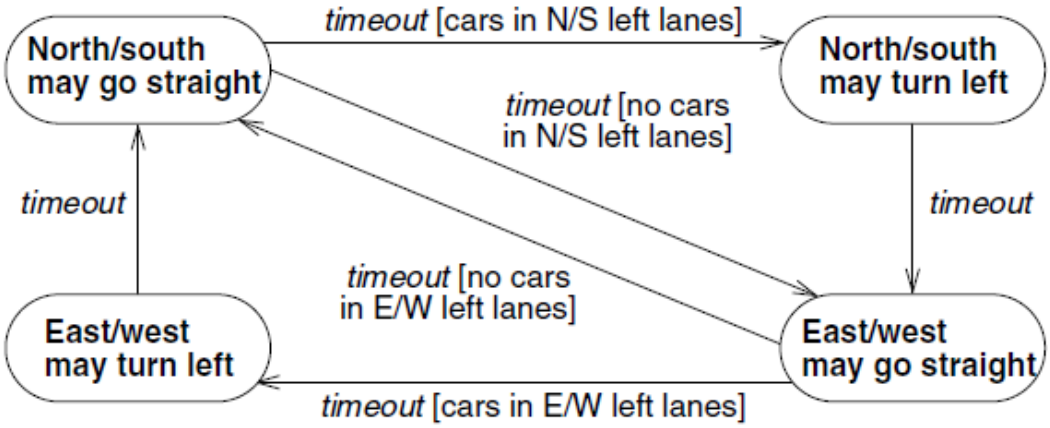


Figure 5.7 Guarded transitions. A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

The UML notation for a transition is a line from the origin state to the target state. An

arrowhead points to the target state. The line may consist of several line segments. An event

may label the transition and be followed by an optional guard condition in square brackets.

By convention, we usually confine line segments to a rectilinear grid. We italicize the event

name and show the condition in normal font.

State Diagram

A **state diagram** is a graph whose nodes are states and whose directed arcs are transitions

between states. A state diagram specifies the state sequences caused by event sequences.

State names must be unique within the scope of a state diagram. All objects in a class execute

the state diagram for that class, which models their common behavior. You can implement

state diagrams by direct interpretation or by converting the semantics into equivalent programming code.

The **state model** consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces events and guard conditions. The individual state diagrams interact by passing events and through the side effects of guard conditions. Some events and guard conditions appear in a single state diagram; others appear in multiple state diagrams for the purpose of coordination.This chapter covers only individual state diagrams; Chapter 6 discusses state models of interacting diagrams.

A class with more than one state has important temporal behavior. Similarly, a class is temporally important if it has a single state with multiple responses to events. You can represent state diagrams with a single state in a simple nongraphical form—a stimulus–response table listing events and guard conditions and the ensuing behavior.

One-shot State Diagrams

State diagrams can represent continuous loops or one-shot life cycles. The diagram for the phone line is a continuous loop. In describing ordinary usage of the phone, we do not know or care how the loop is started. (If we were describing installation of new lines, the initial state would be important.)

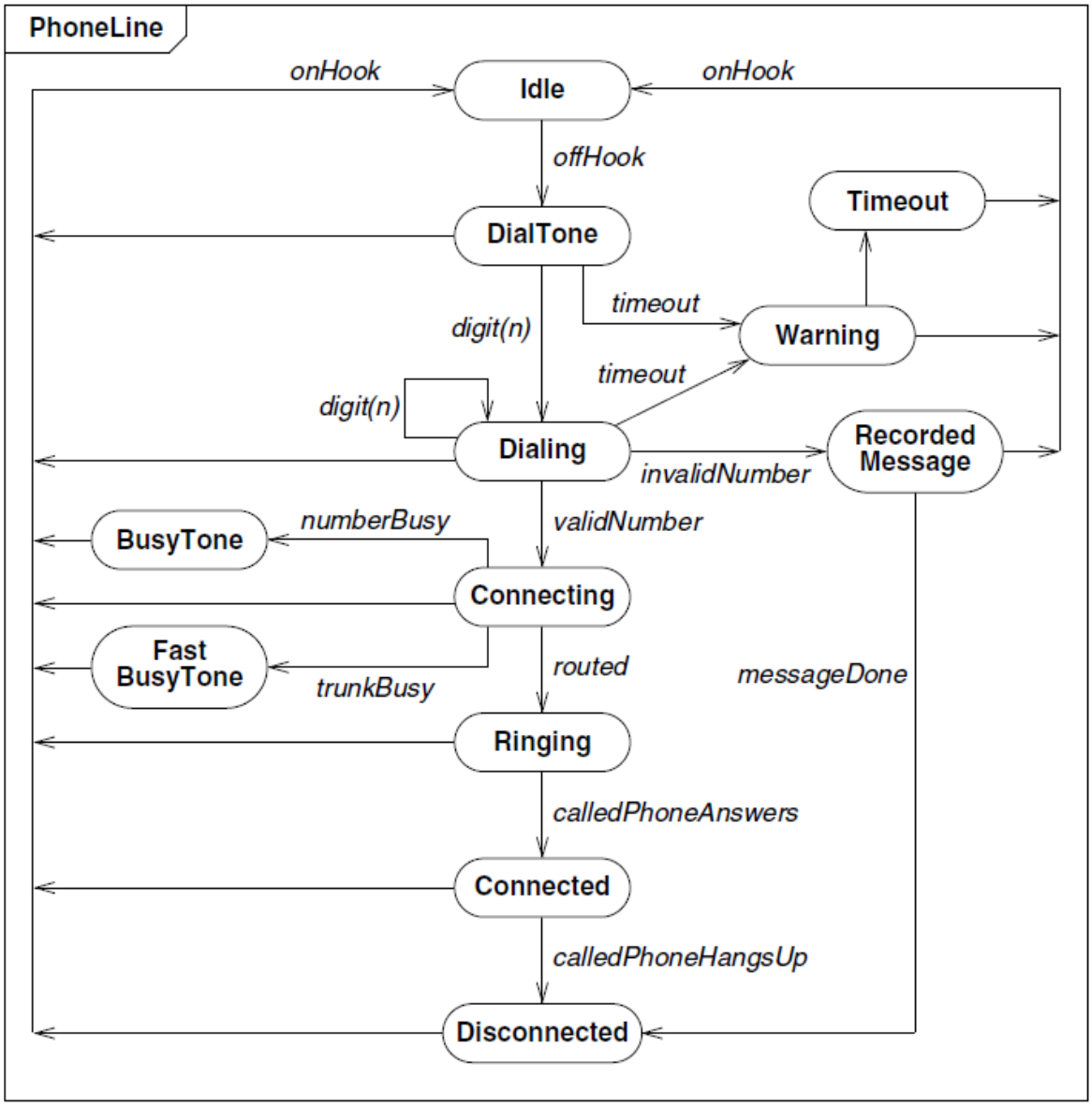


Figure 5.8 State diagram for a telephone line. A state diagram specifies the state sequences caused by event sequences.

One-shot state diagrams represent objects with finite lives and have initial and final states. The initial state is entered on creation of an object; entry of the final state implies destruction of the object. Figure 5.9 shows a simplified life cycle of a chess game with a default initial state (solid circle) and a default final state (bull's eye).

As an alternate notation, you can indicate initial and final states via entry and exit points.

In Figure 5.10 the start entry point leads to white's first turn, and the chess game eventually ends with one of three possible outcomes. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.

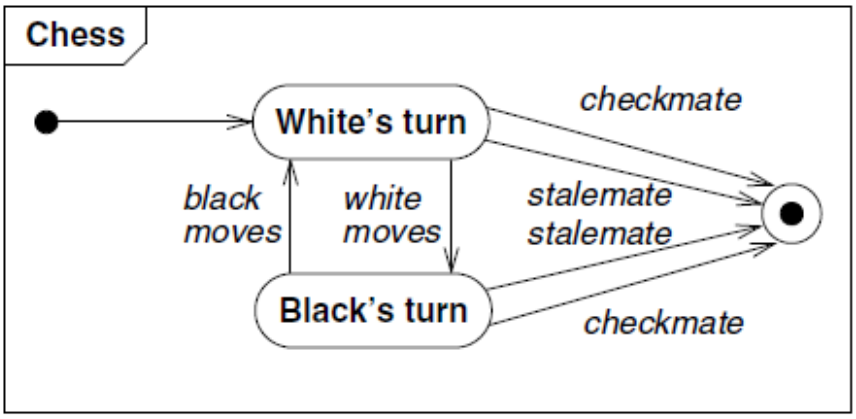


Figure 5.9 State diagram for chess game. One-shot diagrams represent objects with finite lives.

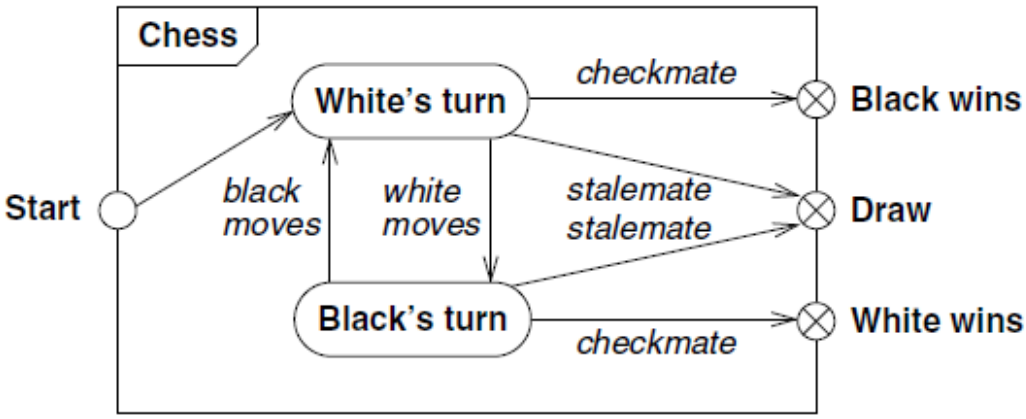


Figure 5.10 State diagram for chess game. You can also show one-shot diagrams by using entry and exit points.

Summary of Basic State Diagram Notation

Figure 5.11 summarizes the basic UML syntax for state diagrams.

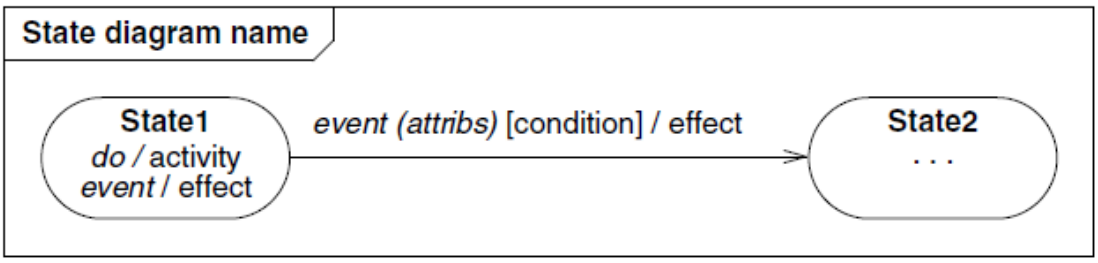


Figure 5.11 Summary of basic notation for state diagrams.

State. Drawn as a rounded box containing an optional name. A special notation is available

for initial states (a solid circle) and final states (a bull's-eye or encircled "x").

Transition. Drawn as a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments.

■ **Event**. A signal event is shown as a label on a transition and may be followed by parenthesized attributes. A change event is shown with the keyword when followed by a parenthesized boolean expression. A time event is shown with the keyword when followed by a parenthesized expression involving time or the keyword after followed by a parenthesized expression that evaluates to a time duration.

■ **State diagram**. Enclosed in a rectangular frame with the diagram name in a small pentagonal tag in the upper left corner.

■ **Guard condition**. Optionally listed in square brackets after an event.

■ **Effects** (to be explained in next section). Can be attached to a transition or state and are listed after a slash ("/"). Multiple effects are separated with a comma and are performed concurrently. (You can create intervening states if you want multiple effects to be performed in sequence.)

State Diagram Behavior

Activity Effects An **effect** is a reference to a behavior that is executed in response to an event. An **activity** is the actual behavior that can be invoked by any number of effects. For example, disconnect-PhoneLine might be an activity that is executed in response to an onHook event for Figure 5.8. An activity may be performed upon a transition, upon the entry to or exit from a state, or upon some other event within a state.

Activities can also represent internal control operations, such as setting attributes or generating other events. Such activities have no real-world counterparts but instead are mechanisms for structuring control within an implementation. For example, a program might increment an internal counter every time a particular event occurs. The notation for an activity is a slash ("/") and the name (or description) of the activity, following the event that causes it. The keyword do is reserved for indicating an ongoing activity (to be explained) and may not be used as an event name. Figure 5.12 shows the state diagram for a pop-up menu on a workstation.When the right button is depressed, the menu is displayed; when the right button is released, the menu is erased. While the menu is visible, the highlighted menu item is updated whenever the cursor moves.

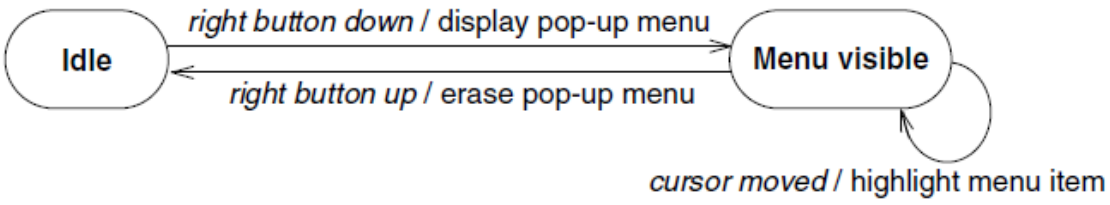


Figure 5.12 Activities for pop-up menu. An activity is behavior that can be executed in response to an event.

Do-Activities A **do-activity** is an activity that continues for an extended time. By definition, a do-activity can only occur within a state and cannot be attached to a transition. For example, the warning light may flash during the Paper jam state for a copy machine (Figure 5.13). Do-activities include continuous operations, such as displaying a picture on a television screen, as well as sequential operations that terminate by themselves after an interval of time, such as closing a valve.

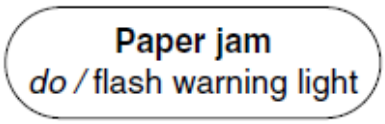


Figure 5.13 Do-activity for a copy machine. A do-activity is an activity that continues for an extended time.

The notation "do /" denotes a do-activity that may be performed for all or part of the

duration that an object is in a state. A do-activity may be interrupted by an event that is received during its execution; such an event may or may not cause a transition out of the state containing the do-activity. For example, a robot moving a part may encounter resistance, causing it to cease moving. Entry and Exit Activities

As an alternative to showing activities on transitions, you can bind activities to entry or to exit from a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state.

For example, Figure 5.14 shows the control of a garage door opener. The user generates depress events with a pushbutton to open and close the door. Each event reverses the direction of the door, but for safety the door must open fully before it can be closed. The control generates motor up and motor down activities for the motor. The motor generates door open and door closed events when the motion has been completed. Both transitions entering state Opening cause the door to open.

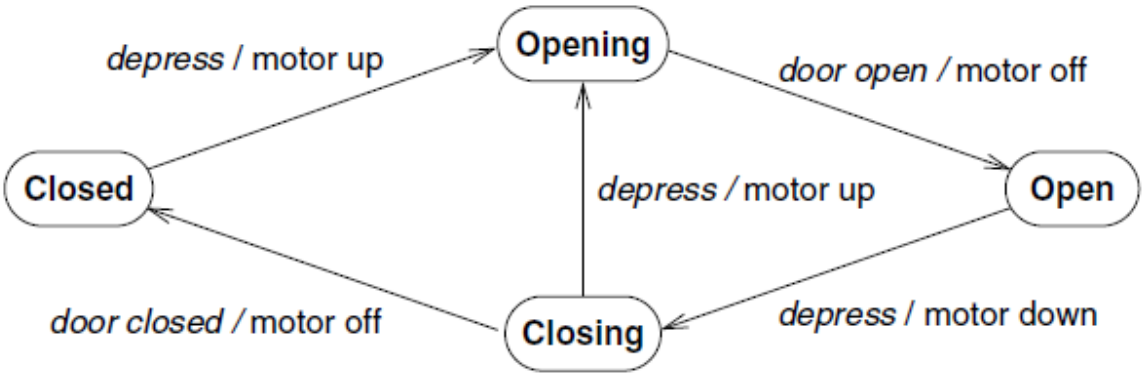


Figure 5.14 Activities on transitions. An activity may be bound to an event that causes a transition.

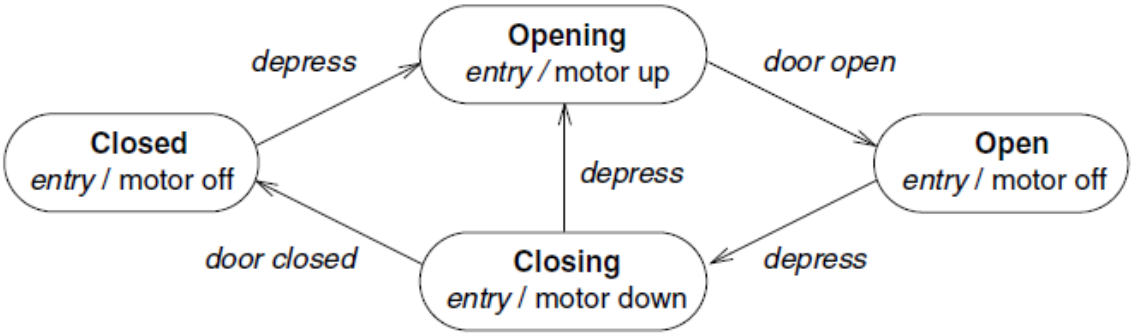


Figure 5.15 Activities on entry to states. An activity may also be bound to an event that occurs within a state.

Figure 5.15 shows the same model using activities on entry to states. An entry activity is shown inside the state box following the keyword entry and a "/" character. Whenever the state is entered, by any incoming transition, the entry activity is performed. An entry activity is equivalent to attaching the activity to every incoming transition. If an incoming transition already has an activity, its activity is performed first exit activity is shown inside the state box following the keyword exit and a "/" character.

Whenever the state is exited, by any outgoing transition, the exit activity is performed first. If a state has multiple activities, they are performed in the following order: activities on the incoming transition, entry activities, do-activities, exit activities, activities on the outgoing transition. Events that cause transitions out of the state can interrupt do-activities. If a doactivity is interrupted, the exit activity is still performed.

In general, any event can occur within a state and cause an activity to be performed. Entry and exit are only two examples of events that can occur. As Figure 5.16 shows, there is a difference between an event within a state and a self-transition; only the self-transition causes the entry and exit activities to be executed.

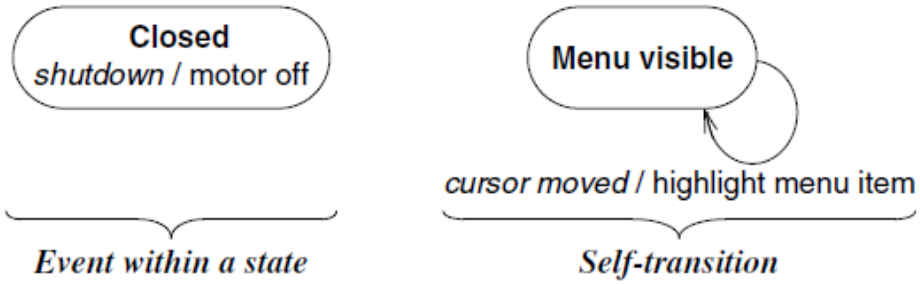


Figure 5.16 Event within a state vs. self-transition. A self-transition causes entry and exit activities to be executed. An event within a state does not.

Completion Transition

Often the sole purpose of a state is to perform a sequential activity. When the activity is completed, a transition to another state fires. An arrow without an event name

indicates an automatic transition that fires when the activity associated with the source state is completed. Such unlabeled transitions are called **completion transitions** because they are triggered by the completion of activity in the source state.

A guard condition is tested only once, when the event occurs. If a state has one or more completion transitions, but none of the guard conditions are satisfied, then the state remains active and may become "stuck"—the completion event does not occur a second time, therefore no completion transition will fire later to change the state. If a state has completion transitions leaving it, normally the guard conditions should cover every possible outcome. You can use the special condition else to apply if all the other conditions are false. Do not use a guard condition on a completion transition to model waiting for a change of value. Instead model the waiting as a change event.

Advanced object and class concept

Enumerations

A data type is a description of values. Data types include numbers, strings, and enumerations. An **enumeration** is a data type that has a finite set of values. For example, the attribute access Permission in Figure 3.17 is an enumeration with possible values that include read and read-write. Figure 3.25 also has some enumerations that Figure 4.1 illustrates. Figure.pen-Type is an enumeration that includes solid, dashed, and dotted. TwoDimensional.fillType is an enumeration that includes solid, grey, none, horizontal lines, and vertical lines.

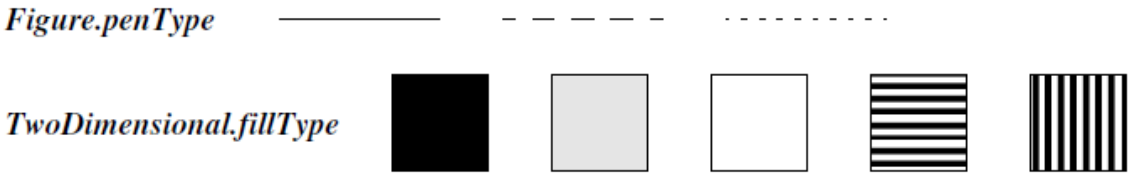


Figure 4.1 Examples of enumerations. Enumerations often occur and are important to users. Implementations must enforce the finite set of values.

When constructing a model, you should carefully note enumerations, because they often occur and are important to users. Enumerations are also significant for an implementation you may display the possible values with a pick list and you must restrict data to the legitimate values.

Do not use a generalization to capture the values of an enumerated attribute. An enumeration is merely a list of values; generalization is a means for structuring the description of objects. You should introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass. As Figure 4.2 shows, you should not introduce a generalization for Card, because most games do not differentiate the behavior of spades, clubs, hearts,

and diamonds.

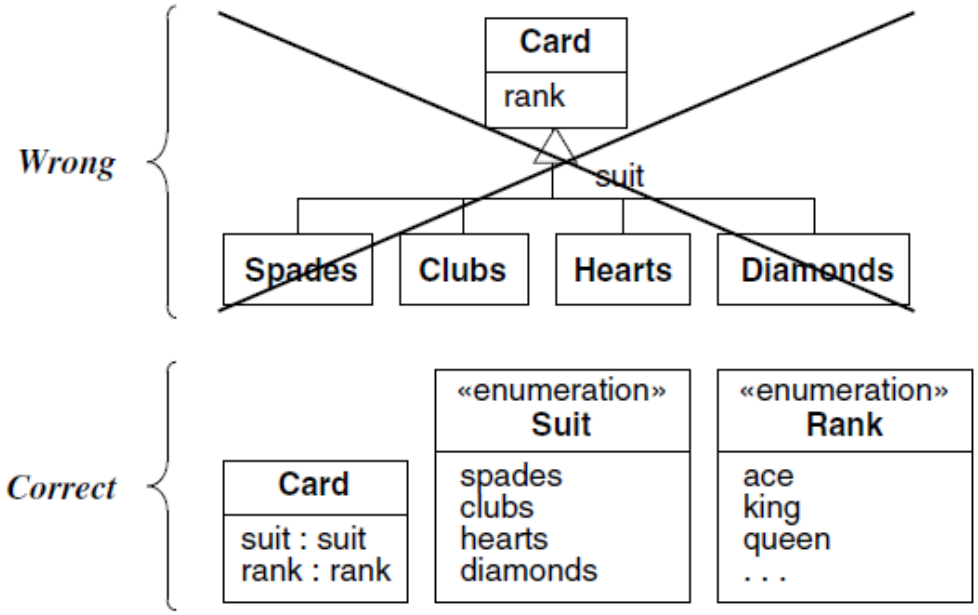


Figure 4.2 Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute.

In the UML an enumeration is a data type. You can declare an enumeration by listing the keyword enumeration in guillemets («») above the enumeration name in the top section of a box. The second section lists the enumeration values.Multiplicity

Multiplicity is a constraint on the cardinality of a set. Chapter 3 explained multiplicity for associations. Multiplicity also applies to attributes.It is often helpful to specify multiplicity for an attribute, especially for database applications.

Multiplicity for an attribute specifies the number of possible values for each instantiation of an attribute. The most common specifications are a mandatory single value [1], an optional single value [0..1], and many [*]. Multiplicity specifies whether an attribute is mandatory or optional (in database terminology whether an attribute can be null). Multiplicity also indicates if an attribute is single valued or can be a collection. If not specified, an attribute is assumed to be a mandatory single value ([1]). In Figure 4.3 a person has one name, one or more addresses, zero or more phone numbers, and one birthdate.

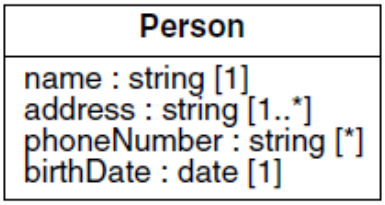


Figure 4.3 Multiplicity for attributes. You can specify whether an attribute is single or multivalued, mandatory or optional.

Scope

Chapter 3 presented features for individual objects. This is the default usage, but there can also be features for an entire class. The **scope** indicates if a feature applies to an object or a class. An underline distinguishes features with class scope (static) from those with object scope. Our convention is to list attributes and operations with class scope at the top of the attribute and operation boxes, respectively.

It is acceptable to use an attribute with class scope to hold the **extent** of a class (the set of objects for a class)—this is common with OO databases. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model. It is better to model groups explicitly and assign attributes to them. For example, the upper model in Figure 4.4 shows a simple model of phone mail. Each message has an owner mailbox, date recorded, time recorded, priority, message contents, and a flag indicating if it has been received. A message may have a mailbox as the source or it may be from an external call. Each mailbox has a phone number, password, and recorded greeting. For the PhoneMessage class we can store the maximum duration for a message and the maximum days a message will be retained. For

the PhoneMailbox class we can store the maximum number of messages that can be stored. The upper model is inferior, however, because the maximum duration, maximum days retained, and maximum message count have a single value for the entire phone mail system. In the lower model these limits can vary for different kinds of users, yielding a more flexible and extensible phone mail system.

In contrast to attributes, it is acceptable to define operations of class scope. The most

common use of class-scoped operations is to create new instances of a class. Sometimes it is convenient to define class-scoped operations to provide summary data. You should be careful with the use of class-scoped operations for distributed applications.

Visibility

Visibility refers to the ability of a method to reference a feature from another class and has the possible values of public, protected, private, and package. The precise meaning depends on the programming language. (See Chapter 18 for details.) Any method can freely access **public** features. Only methods of the containing class and its descendants via inheritance can access **protected** features. (Protected features also have package accessibility in Java.) Only methods of the containing class can access **private** features. Methods of classes defined in the same package as the target class can access **package** features.

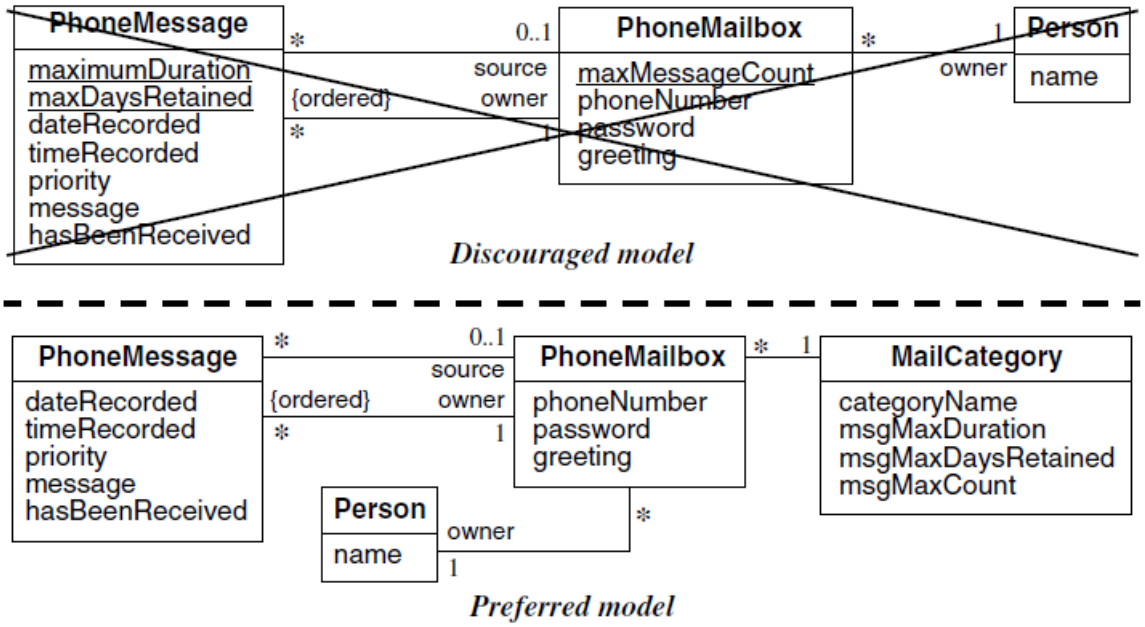


Figure 4.4 Attribute scope. Instead of assigning attributes to classes, model groups explicitly.

The UML denotes visibility with a prefix. The character "+" precedes public features.

The character "#" precedes protected features. The character "-" precedes private features. And the character "~" precedes package features. The lack of a prefix reveals no information about visibility.

There are several issues to consider when choosing visibility.

■ **Comprehension**. You must understand all public features to understand the capabilities of a class. In contrast, you can ignore private, protected, and package features—they are merely an implementation convenience.

■ **Extensibility**. Many classes can depend on public methods, so it can be highly disruptive to change their signature (number of arguments, types of arguments, type of return 0value). Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.

■ **Context**. Private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

Use case Modelling

Use Cases

The various interactions of actors with a system are quantized into use cases. A **use case** is a coherent piece of functionality that a system can provide by interacting with actors. For example, a customer actor can buy a beverage from a vending machine. The customer inserts money into the machine, makes a selection, and ultimately

receives a beverage. Similarly, a repair technician can perform scheduled maintenance on a vending machine. Figure 7.1

summarizes several use cases for a vending machine.

Each use case involves one or more actors as well as the system itself. The use case buy a beverage involves the customer actor and the use case perform scheduled maintenance involves the repair technician actor. In a telephone system, the use case make a call involves two actors, a caller and a receiver. The actors need not all be persons. The use case make a trade on an online stock broker involves a customer actor and a stock exchange actor. The stock broker system needs to communicate with both actors to execute a trade. A use case involves a sequence of messages among the system and its actors. For example, in the buy a beverage use case, the customer first inserts a coin and the vending machine displays the amount deposited. This can be repeated several times. Then the customer pushes

- **Buy a beverage**. The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance**. A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs**. A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items**. A stock clerk adds items into the vending machine to replenish its stock of beverages.

Figure 7.1 Use case summaries for a vending machine. A use case is a coherent piece of functionality that a system can provide by interacting with actors.

a button to indicate a selection; the vending machine dispenses the beverage and issues change, if necessary.

Some use cases have a fixed sequence of messages. More often, however, the message

sequence may have some variations. For example, a customer can deposit a variable number of coins in the buy a beverage use case. Depending on the money inserted and the item selected, the machine may, or may not, return change. You can represent such variability by showing several examples of distinct behavior sequences. Typically you should first define a mainline behavior sequence, then define optional subsequences, repetitions, and other variations.

Error conditions are also part of a use case. For example, if the customer selects a beverage whose supply is exhausted, the vending machine displays a warning message. Similarly, the vending transaction can be cancelled. For example, the customer can push the coin return on the vending machine at any time before a selection has been accepted; the machine returns the coins, and the behavior sequence for the use case is complete. From the user's point of view, some kinds of behavior may be thought of as errors. The designer, however,

should plan for all possible behavior sequences. From the system's point of view, user errors or resource failures are just additional kinds of behavior that a robust system can accommodate. A use case brings together all of the behavior relevant to

a slice of system functionality. This includes normal mainline behavior, variations on normal behavior, exception conditions, error conditions, and cancellations of a request. Figure 7.2 explains the buy a beverage use case in detail. Grouping normal and abnormal behavior under a single use case helps to ensure that all the consequences of an interaction are considered together. In a complete model, the use cases partition the functionality of the system. They should preferably all be at a comparable level of abstraction. For example, the use cases make telephone call and record voice mail message are at comparable levels. The use case set external speaker volume to high is too narrow. It would be better as set speaker volume (with the volume level selection as part of the use case) or maybe even just set telephone parameters, under which we might group setting volume, display pad settings, setting the clock, and so on.

Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message "Enter coins." A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer's money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message "That item is out of stock" is displayed. The machine continues to accept coins or a selection.

Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message "You must insert $*nn.nn* more for that item" is displayed, where *nn.nn* is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

Figure 7.2 Use case description. A use case brings together all of the behavior relevant to a slice of system functionality.

Use Case Diagrams

A system involves a set of use cases and a set of actors. Each use case represents a slice of the functionality the system provides. The set of use cases shows the complete functionality of the system at some level of detail. Similarly, each actor represents one kind of object for which the system can perform behavior. The set of actors

represents the complete set of objects that the system can serve. Objects accumulate behavior from all the systems with which they interact as actors. The UML has a graphical notation for summarizing use cases and Figure 7.3 shows an example. A rectangle contains the use cases for a system with the actors listed on the outside.

The name of the system may be written near a side of the rectangle. A name within an ellipse

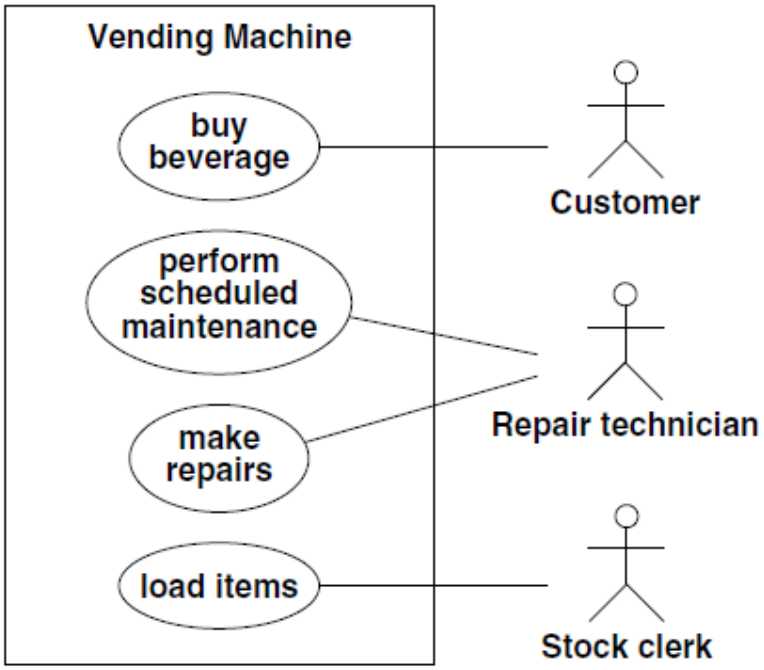


Figure 7.3 Use case diagram for a vending machine. A system involves a set of use cases and a set of actors.

denotes a use case. A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors.

In the figure, the actor Repair technician participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

Guidelines for Use Case Models

Use cases identify the functionality of a system and organize it according to the perspective of users. In contrast, traditional requirements lists can include functionality that is vague to users, as well as overlook supporting functionality, such as initialization and termination. Use cases describe complete transactions and are therefore less likely to omit necessary steps. There is still a place for traditional requirements lists in describing global constraints and other nonlocalized functionality, such as mean time to failure and overall throughput, but you should capture most user interactions with use cases. The main purpose of a system is

almost always found in the use cases, with requirements lists supplying additional implementation constraints. Here are some guidelines for constructing use case models.

■ **First determine the system boundary**. It is impossible to identify use cases or actors if the system boundary is unclear.

■ **Ensure that actors are focused**. Each actor should have a single, coherent purpose. If a real-world object embodies multiple purposes, capture them with separate actors. For example, the owner of a personal computer may install software, set up a database, and send email. These functions differ greatly in their impact on the computer system and the potential for system damage. They might be broken into three actors: system admin-istrator, database administrator, and computer user. Remember that an actor is defined with respect to a system, not as a free-standing concept.

■ **Each use case must provide value to users**. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly. For example, dial a telephone number is not a good use case for a telephone system. It does not represent a complete transaction of value by itself; it is merely part of the use case make telephone call. The latter use case involves placing the call, talking, and terminating the call. By dealing with complete use cases, we focus on the purpose of the functionality provided by the system, rather than jumping into implementation decisions. The details come later. Often there is more than one way to implement desired functionality.

■ **Relate use cases and actors**. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

■ **Remember that use cases are informal**. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.

■ **Use cases can be structured**. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships (see Chapter 8).

Realizing use case

Use cases define the required behavior, but they do not define its realization. That is the purpose of design—to choose among the options and prepare for implementation. Each choice has advantages and disadvantages. It is not sufficient merely to deliver the behavior, although that is a primary goal. You must also consider the consequences of each choice on performance, reliability, ease of future enhancement, and many other “ilities”. Design is the process of realizing functionality while balancing conflicting needs. Use cases define system-level behavior. During design you must invent new operations and new objects that provide this behavior. Then, in turn, you must define each of these new operations in terms of lower-level operations involving more objects. Eventually you can implement operations directly in terms of existing operations. Inventing the right intermediate operations is what we have called “bridging the gap.” To start, list the responsibilities of a use case or operation. A **responsibility** is something

that an object knows or something it must do [Wirfs-Brock-90]. A responsibility is not a precise concept; it is meant to get the thought process going. For example, in an online theatre ticket system, making a reservation has the responsibility of finding unoccupied seats to the desired show, marking the seats as occupied, obtaining payment from the customer, arranging delivery of the tickets, and crediting payment to the proper account. The theater system itself must track which seats are occupied, know the prices of various seats, and so on.

Each operation will have various responsibilities. Some of these may be shared by other operations, and others may be reused in the future. Group the responsibilities into clusters and try to make each cluster coherent. That is, each cluster should consist of related responsibilities that can be serviced by a single lower-level operation. Sometimes, if the responsibilities are broad and independent, each responsibility is in its own cluster. Now define an operation for each responsibility cluster. Define the operation so that is

not restricted to special circumstances, but don't make it so general that it is unfocused. The goal is to anticipate future uses of the new operation. If the operation can be used in several different places in the current design, you probably don't have to make it more general, except to cover the existing uses. Finally, assign the new lower-level operations to classes. If there is no good class to hold an operation, you may need to invent a new lower-level class.

ATM example. One of the use cases from Chapter 13 is process transaction. Recall that a Transaction is a set of Updates and that the logic varies according to withdrawal, deposit, and transfer.

■ **Withdrawal**. A withdrawal involves a number of responsibilities: get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank's policies, verify that ATM has sufficient cash, disburse funds, debit bank account, and post entry on the customer's receipt. Note that some of these responsibilities must be performed within the context of a database transaction. A database transaction ensures all-or-nothing behavior—all operations within the scope of a transaction happen or none of the operations happen. For example, the disbursement of funds and debiting of the bank account must both happen together.

■ **Deposit**. A deposit involves several responsibilities: get amount from customer, accept funds envelope from customer, time-stamp envelope, credit bank account, and post entry on the customer's receipt. Some of these responsibilities must also be performed within the context of a database transaction.

■ **Transfer**. Responsibilities include: get source account, get target account, get amount, verify that source account covers amount, verify that the amount is within the bank's policies, debit the source account, credit the target account, and post an entry on the customer's receipt. Once again some of the responsibilities must happen within a database transaction.

Actor Identification

Actor identification in a use case involves determining the entities (people, systems, or devices) that interact with the system being designed. Actors represent roles that participate in the system's functionality and are external to the system.

Steps to Identify Actors in a Use Case:

1. **Understand the System Scope**:
    - Clearly define the boundaries of the system and what it is intended to do.
2. **Identify Stakeholders**:
    - Determine all the people or systems that have an interest in or need from the system.
3. **Ask Key Questions**:
    - Who will use the system? (Primary Actors)
    - Who will provide inputs to the system?
    - Who will receive outputs from the system?
    - Are there external systems that interact with this system?
    - Are there devices or hardware that need to interact with the system?
4. **Categorize Actors**:
    - **Primary Actors**: Directly interact with the system to achieve their goals (e.g., a user logging into a system).
    - **Secondary Actors**: Support the system in achieving its goals but are not directly involved in the primary workflow (e.g., a database server or an authentication service).
5. **Check for Generalization**:
    - If multiple actors share common behavior or goals, consider abstracting them into a generalized actor.
6. **Validate the Identified Actors**:
    - Review with stakeholders to ensure all relevant actors are included.

Actor Classification

Actors can be classified into the following categories:

---

1. Primary Actors

- **Definition**: Entities that initiate interaction with the system to achieve their goals.
- **Purpose**: Drive the main use cases; their goals are directly addressed by the system.
- **Examples**:
    - A **Customer** placing an order in an e-commerce application.

- An **Employee** applying for leave in an HR system.

---

2. Secondary Actors

- **Definition**: Entities that support the system in achieving the goals of primary actors. They don't directly trigger the system's operations but provide services or perform background tasks.
- **Purpose**: Facilitate the functionality required by the primary actors.
- **Examples**:
  - A **Payment Gateway** processing payments in an online store.
  - A **Database Server** storing transaction records.

---

3. Human Actors

- **Definition**: People interacting with the system.
- **Purpose**: Represent real-world users or stakeholders.
- **Examples**:
  - A **Student** registering for courses in an academic portal.
  - A **Manager** approving leave requests.

---

4. System Actors

- **Definition**: External systems or software interacting with the system being designed.
- **Purpose**: Collaborate with the system to perform tasks or exchange data.
- **Examples**:
  - A **Third-Party API** validating credit card details.
  - An **External Accounting System** generating financial reports.

---

5. Device Actors

- **Definition**: Hardware devices interacting with the system.
- **Purpose**: Serve as interfaces or data sources for the system.
- **Examples**:
  - A **Barcode Scanner** used in inventory management.
  - A **Smart Sensor** collecting environmental data.

---

6. Organizational Actors

- **Definition**: Departments, organizations, or groups that interact with the system.
- **Purpose**: Represent institutional entities rather than individual users.
- **Examples**:
    - A **Logistics Department** using a delivery management system.
    - A **Regulatory Authority** auditing system operations.

Actor Generalization

in use case modeling is the process of identifying common roles, responsibilities, or behaviors among actors and creating a generalized actor to represent shared functionality. It uses inheritance to model relationships between actors, making diagrams simpler and avoiding redundancy.

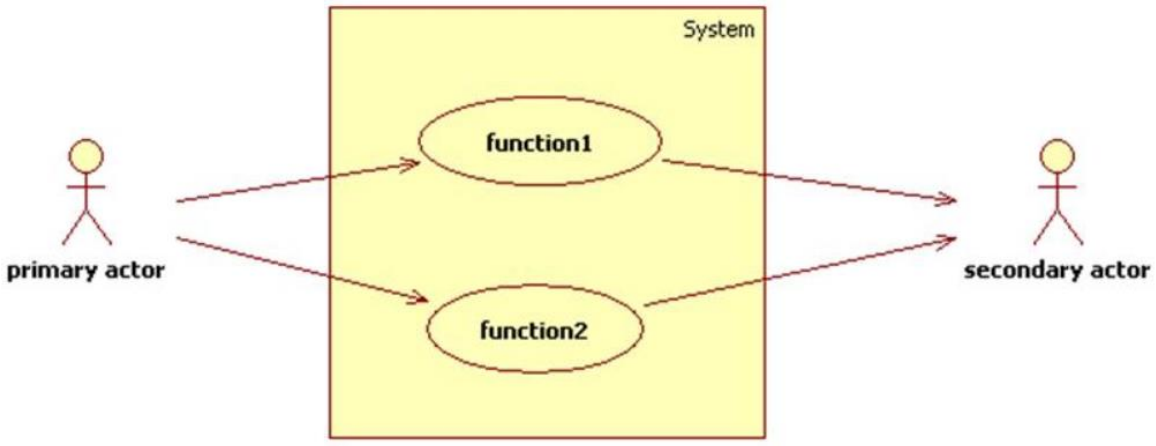
Purpose of Actor Generalization

1. **Avoid Redundancy**: Combine common behaviors shared by multiple actors.
2. **Simplify Models**: Reduce complexity by grouping similar actors under a generalized parent actor.
3. **Enhance Flexibility**: Allow for easy modification or extension of use cases by defining shared and specific behaviors.

Use Cases Identification

A use case represents a function that the system performs. Alternatively, a use case can be thought of as a goal that some actor can achieve with the system.

An actor association is an arrow connecting an actor to a use case. This arrow represents a conversation between the actor and the system component responsible for executing the use case.



A conversation can be viewed as an exchange of messages. For example, the system might send a message to the user by displaying a dialog box. The user might reply to this message by clicking a button on the dialog box. Conversations can be long or short. The initiator of the conversation is indicated by the direction of the arrow, which points from the initiator to the responder.

Primary actors, such as users, always initiate conversations. Secondary actors, such as servers, always respond to an initiating use case.

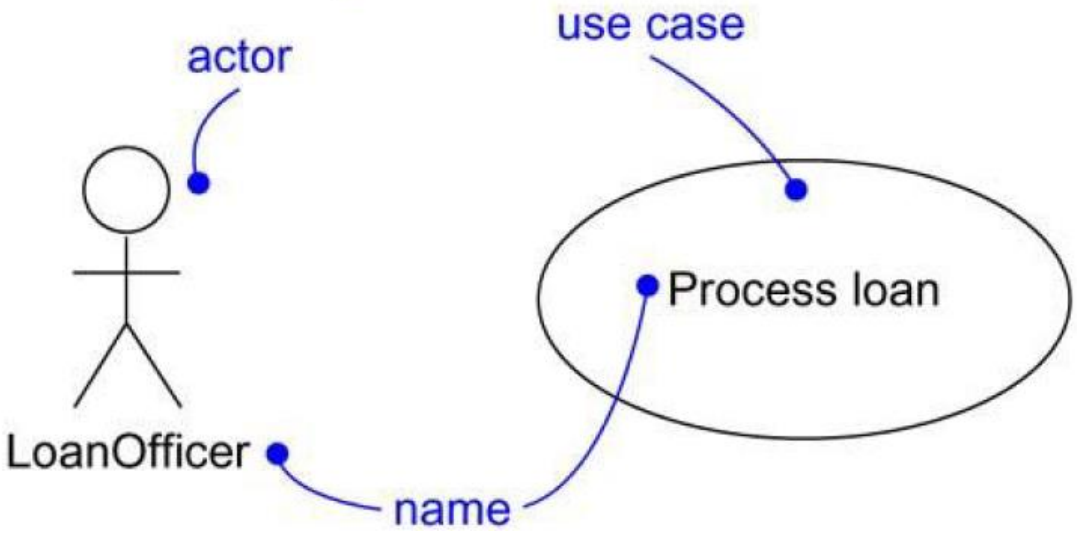
The optional system boundary helps to remind us that use cases are internal to the system and actors are external.

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).

These behaviors are in effect system-level functions that you use to visualize, specify, construct, and document the intended behavior of your system during requirements capture and analysis. A use case represents a functional requirement of your system as a whole. For example, one central use case of a bank is to process loans.

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. For example, in modeling a bank, processing a loan involves, among other things, the interaction between a customer and a loan officer.

Figure 16-1 Actors and Use Cases



Use Cases and Flow of Events

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 re enter 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**.**

Terms and Concepts

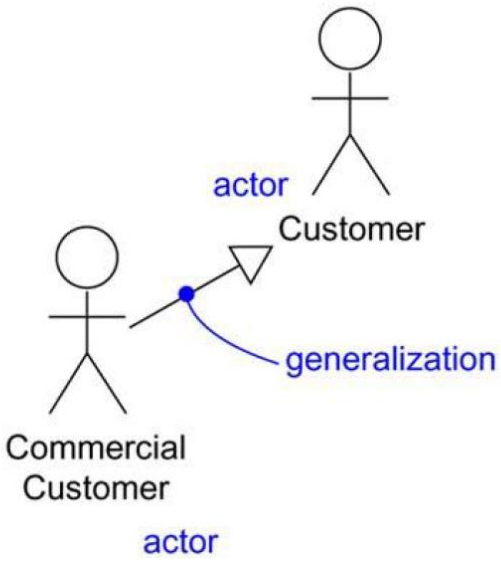
Names

Every use case must have a name that distinguishes it from other use cases. A name is a textual String.

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 system. They live outside the system.



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 Validate User 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.

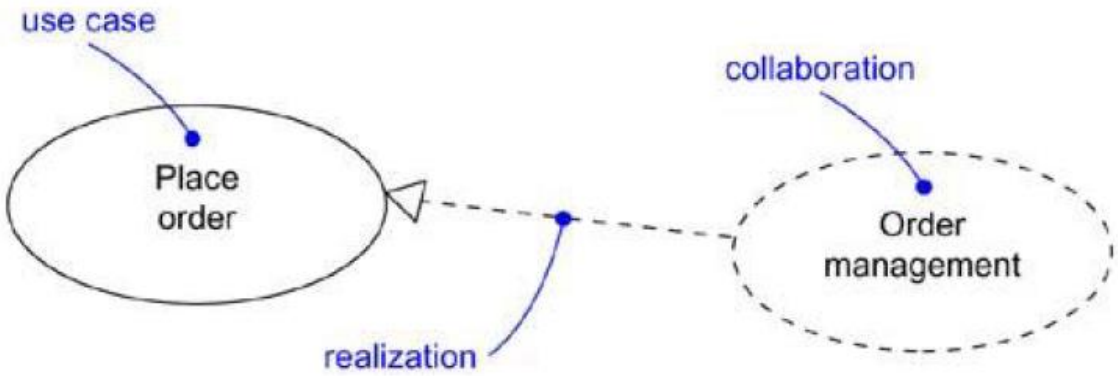
Exceptional flow of events:

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 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.

Figure 16-4 Use Cases and Collaborations

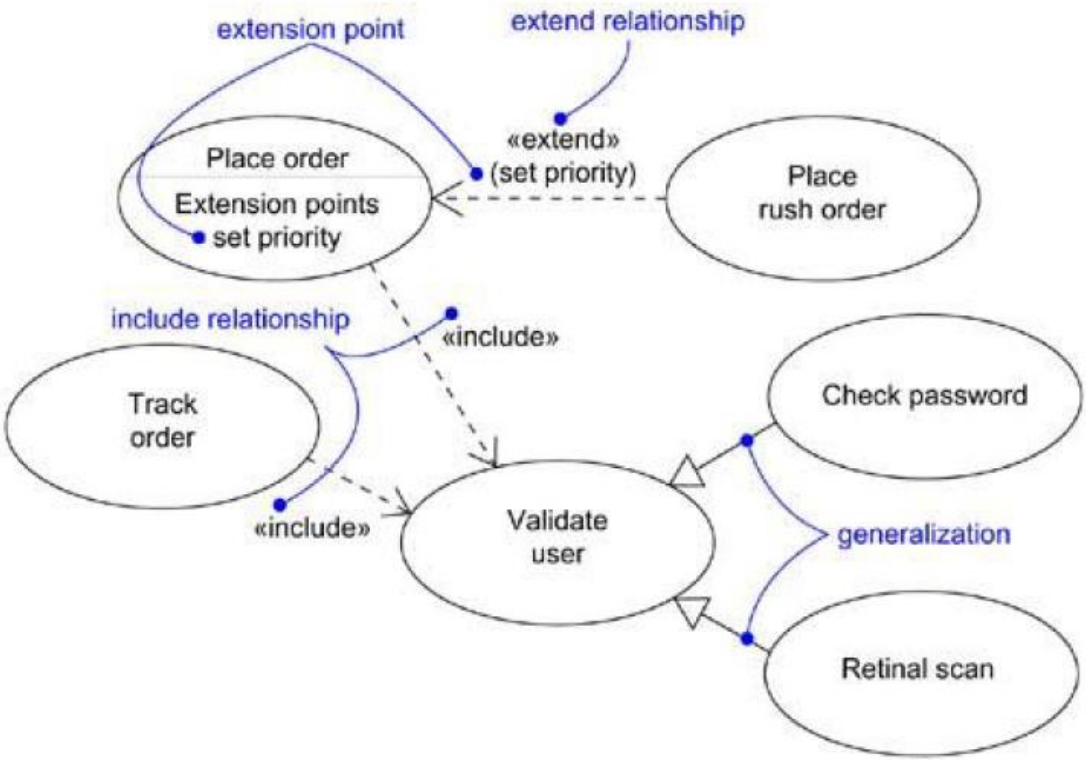


Note

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

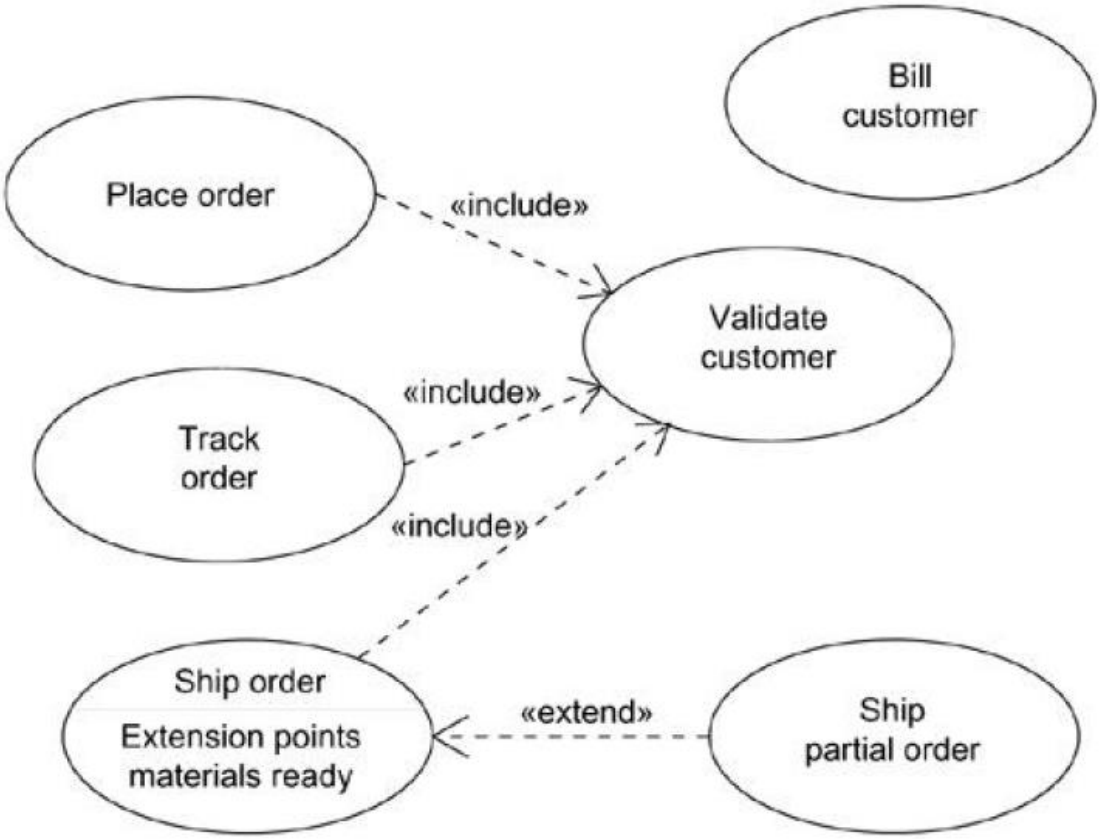
Organizing Use Cases

Figure 16-5 Generalization, Include, and Extend



Modeling the Behavior of an Element

Figure 16-6 Modeling the Behavior of an Element



identifiable behavior of the system or part of the system. A well-structured use case

· Names a single, identifiable, and reasonably atomic behavior of the system or part of the system.

· Factors common behavior by pulling such behavior from other use cases that it includes.

· Factors variants by pushing such behavior into other use cases that extend it.

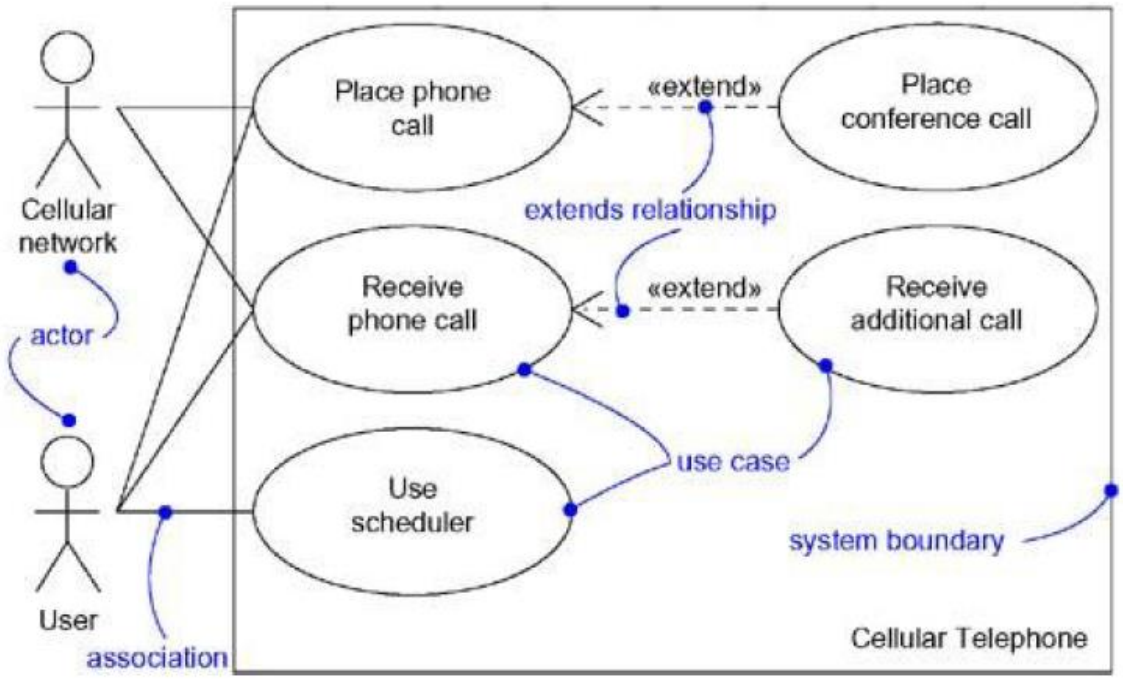
· Describes the flow of events clearly enough for an outsider to easily understand it.

· Is described by a minimal set of scenarios that specify the normal and variant semantics of the use case.When you draw a use case in the UML,

· Show only those use cases that are important to understand the behavior of the system or the part of the system in its context.

· Show only those actors that relate to these use cases.

Figure 17-1 A Use Case Diagram



Use case diagrams commonly contain

· Use cases

· Actors

· Dependency, generalization, and association relationships

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

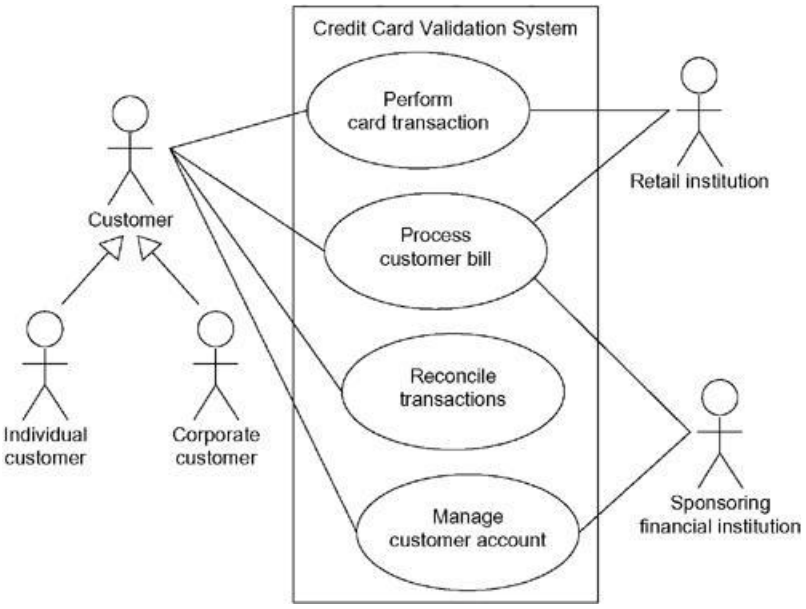
1. To model the context of a system

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.

Requirements are discussed in Chapters 4 and 6.

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.

Figure 17-2 Modeling the Context of a System



Modeling the Requirements of a System

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.

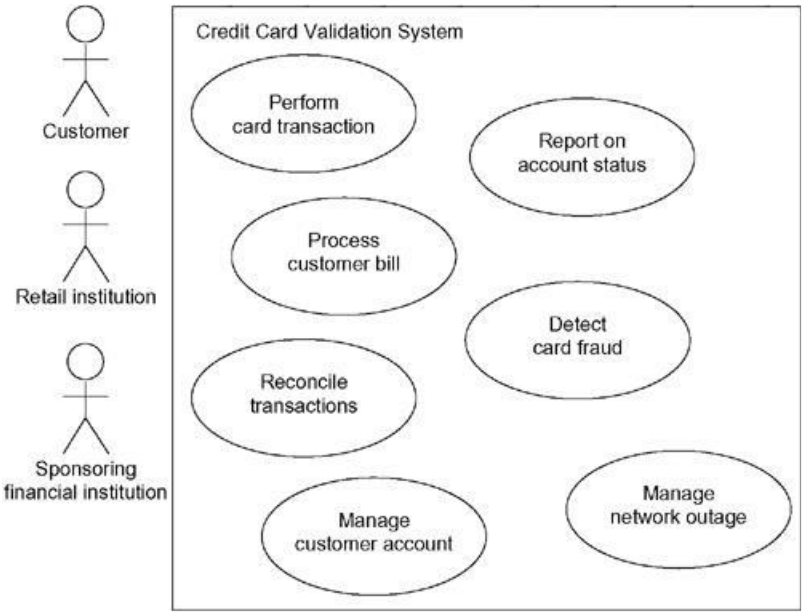
· Name these common behaviors as use cases.

· 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 17-3 Modeling the Requirements of a System



Forward and Reverse Engineering

Forward engineering 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 variants of those flows), 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.

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.