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 value). 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.

4.2 Association Ends

As the name implies, an ***association end*** is an end of an association. A binary association has two ends, a ternary association (Section 4.3) has three ends, and so forth. Chapter 3 discussed the following properties.

- **Association end name**. An association end may have a name. The names disambiguate multiple references to a class and facilitate navigation. Meaningful names often arise, and it is useful to place the names within the proper context.
- **Multiplicity**. You can specify multiplicity for each association end. The most common multiplicities are "1" (exactly one), "0..1" (at most one), and "*" ("many"—zero or more).
- **Ordering**. The objects for a "many" association end are usually just a set. However, sometimes the objects have an explicit order.
- **Bags and sequences**. The objects for a "many" association end can also be a bag or sequence.
- **Qualification**. One or more qualifier attributes can disambiguate the objects for a "many" association end.

Association ends have some additional properties.

- **Aggregation**. The association end may be an aggregate or constituent part (Section 4.4). Only a binary association can be an aggregation; one association end must be an aggregate and the other must be a constituent.
- **Changeability**. This property specifies the update status of an association end. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).
- **Navigability**. Conceptually, an association may be traversed in either direction. However, an implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class. Arrowheads may be attached to zero, one, or both ends of an association.
- **Visibility**. Similar to attributes and operations (Section 4.1.4), association ends may be *public*, *protected*, *private*, or *package*.

4.3 N-ary Associations

Chapter 3 presented binary associations (associations between two classes). However, you may occasionally encounter ***n-ary associations*** (associations among three or more classes.) You should try to avoid n-ary associations—most of them can be decomposed into binary associations, with possible qualifiers and attributes. Figure 4.5 shows an association that at first glance might seem to be an n-ary but can readily be restated as binary associations.

A nonatomic n-ary association—a person makes the purchase of stock in a company...

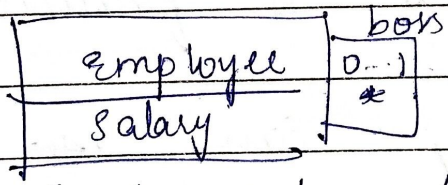
Can be restated as...

Q:- What is Constraints, explain Constrains on Object, generilization set & link with eg:-

Ans:- A Constraint is boolean condition which contains model elements such as objects, classes, attributes, links, generilization set etc. It restricts the values that elements can assume.

(i) Constraints on objects.

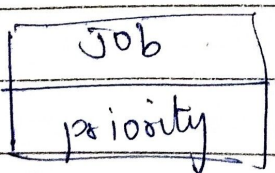
eg:- (1)



{Salary ≤ boss.salary}.

No Employee's Salary can exceed the Salary of the employee's boss.

(2)



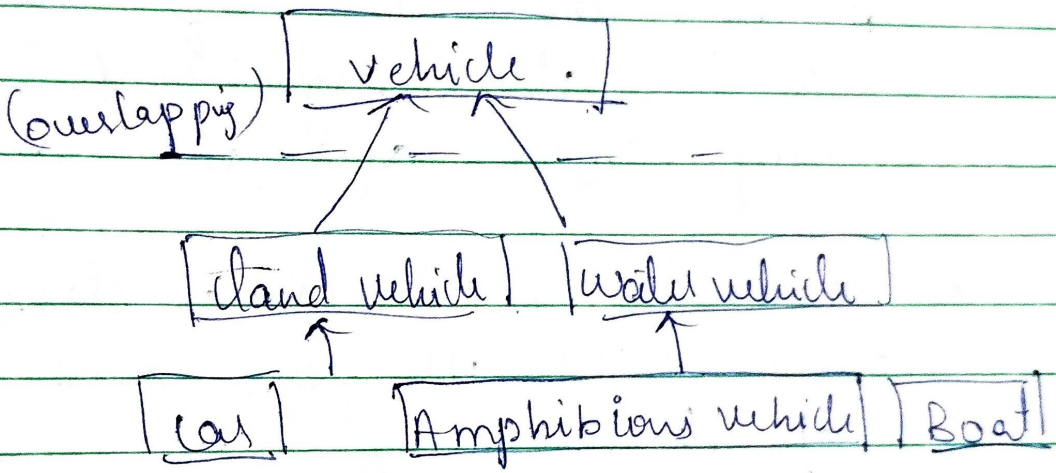
{Pribrity never increases}.

The pribrity of job may not increase

(2) Constraints on Generalization sets.

Class models captures many constraints through their structure. With single inheritance the subclasses are mutually exclusive. Each instance of an abstract superclass corresponds to exactly one subclass instance.

Eg:



97 Contains some of the keywords.

(1) Disjoint :- Each object belongs to exactly one of the subclass

(2) Overlapping :- An object may belong to more than one subclass

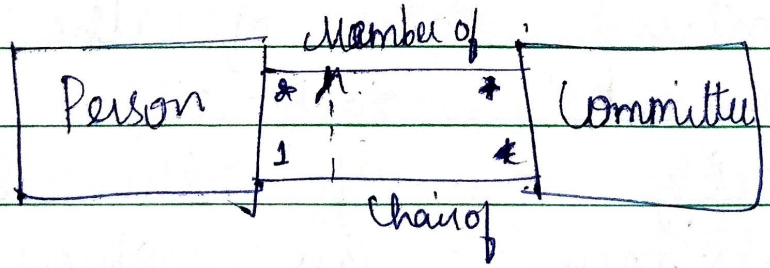
(3) complete :- The generalization lists all possible subclass

(4) Incomplete :- The generalization may be missing some subclass

③ Constraints on links :

- Multiplicity for an association restricts the number of objects related to a given object.
- The qualifier attribute does not describe the links of an association but is also significant in resolving "many" object at an association end.
- Association class is a class in every right it can have attributes & operations.

Eg



Subset Constraint b/w association

Page No.:
Date : / / 20

3. Define events explain different types of event

Ans: An event is an occurance at a point in time. An event happens instantaneously with regard to time scale of an application.

Events include errors condition as well as normal occurances.

↓ types of event.

1. Signal event
2. Change event
3. time event

a Signal

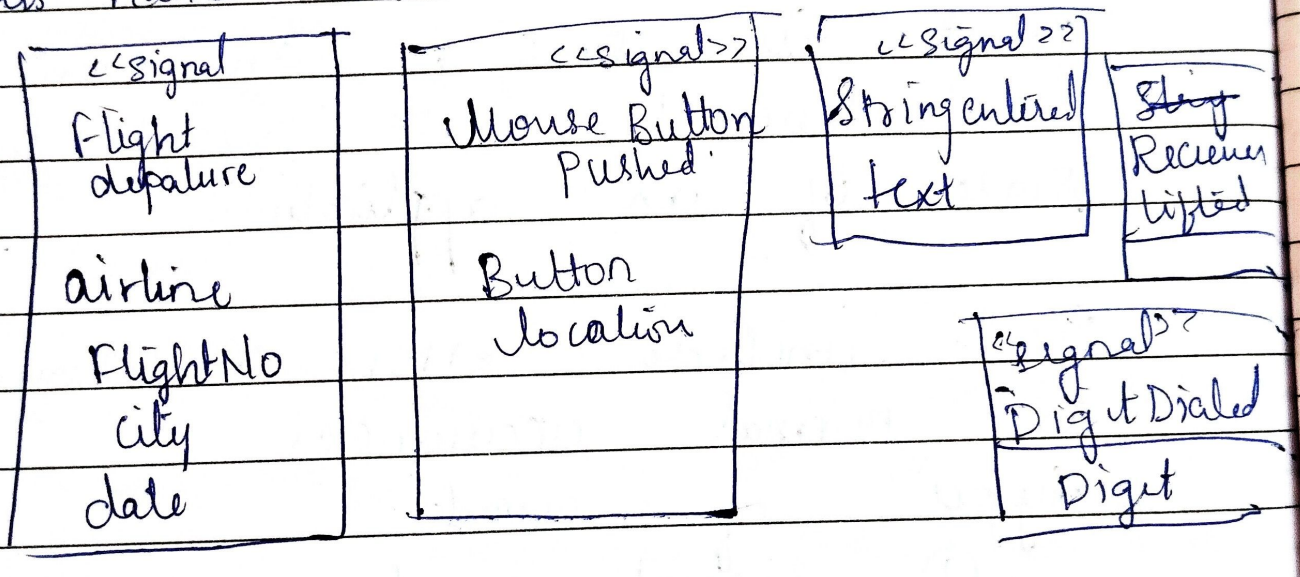
① Signal event :- is an one way transmission of information from one object to another.

whereas Signal event.

- is the event of sending or receiving a Signal

- The difference b/w Signal & Signal event is a Signal is a message b/w objects while Signal event is occurance in time

- Every signal transmission is a unique occurance & we group them into signal classes. & give each signal class name to indicate common structure.



② Change event

cerned 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 *FlightDeparture*. 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.

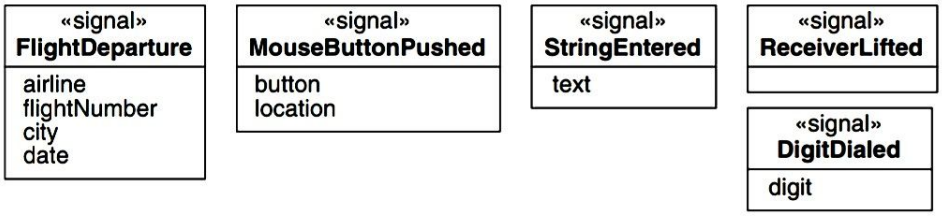


Figure 5.1 Signal classes and attributes. A signal is an explicit one-way transmission of information from one object to another.

5.1.2 Change Event

A ***change event*** is an event that is caused by the satisfaction of a boolean expression. The intent of a change event is that the expression is continually tested—whenever the expression changes from false to true, the event happens. Of course, an implementation would not *continuously* check a change event, but it must check often enough so that it seems continuous from an application perspective.

The UML notation for a change event is the keyword *when* followed by a parenthesized boolean expression. Figure 5.2 shows several examples of change events.

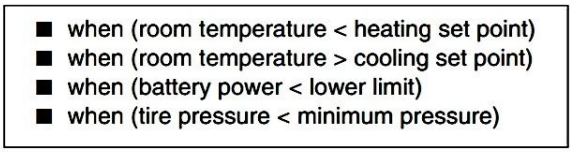


Figure 5.2 Change events. A change event is an event that is caused by the satisfaction of a boolean expression.

5.1.3 Time Event

A ***time event*** is an event caused by the occurrence of an absolute time or the elapse of a time interval. As Figure 5.3 shows, the UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time. The notation for a time interval is the keyword *after* followed by a parenthesized expression that evaluates to a time duration.

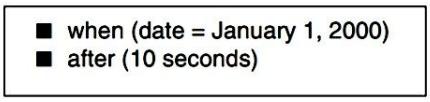


Figure 5.3 Time events. A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

5.2 States

A ***state*** is an abstraction of the values and links of an object. Sets of values and links are

confine transition line segments to a rectilinear grid.

5.5 State Diagram Behavior

State diagrams would be of little use if they just described events. A full description of an object must specify what the object does in response to events.

5.5.1 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, *disconnectPhoneLine* 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.

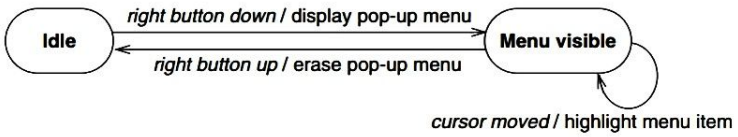


Figure 5.12 Activities for pop-up menu. An activity is behavior that can be executed in response to an event.

5.5.2 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.

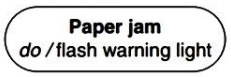


Figure 5.13 Do-activity for a copy machine. A do-activity is an activity that continues for an extended time.

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.

5.5.3 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.

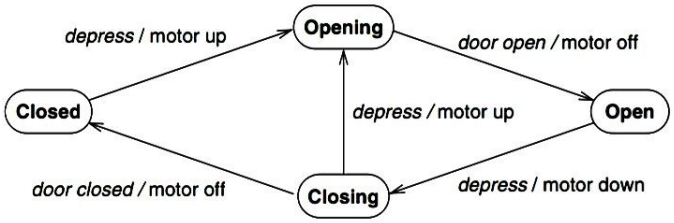


Figure 5.14 Activities on transitions. An activity may be bound to an event that causes a transition.

Figure 5.15 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.

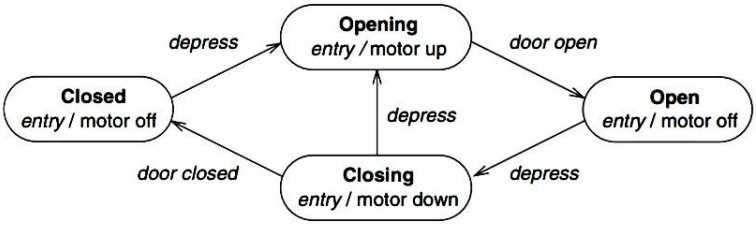


Figure 5.15 Activities on entry to states. An activity may also be bound to an event that occurs within a state.

Exit activities are less common than entry activities, but they are occasionally useful. An 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 do-activity 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.

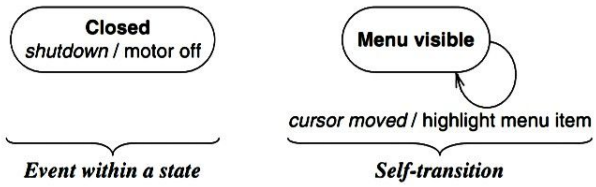


Figure 5.16 Event within a state vs. self-transition. A self-transition causes entry and exit activities to be executed. An event within a state does not.

5.5.4 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

Also remember that you must assume perfect technology. Be sure that the use cases are based on business events, not on technical activities such as logging on to the system. Given those preconditions, you can develop the use case diagram.

USE CASE DETAILED DESCRIPTIONS

As indicated earlier, creating a use case diagram is only one part of use case analysis. The use case diagram helps identify the various processes that users perform and that the new system must support. But careful system development requires us to go to a much more detailed level of description or diagram. To create a comprehensive, robust system that truly meets users' needs, we must understand all of the detailed steps. Internally, a use case includes a whole sequence of steps to complete a business process, and frequently several variations of the business steps exist within a single use case. The use case *Create new order* will have a separate flow of activities for each actor that invokes the use case. The processes for an order clerk creating a new order over the telephone may be quite different from the processes for a customer creating an order over the Internet. Each flow of activities is a valid sequence for the *Create new order* use case. These different flows of activities are called *scenarios*, or sometimes *use case instances*. Thus, a scenario is a unique set of internal activities within a use case and represents a unique path through the use case.

use case
instance
ce of steps
e case may
t scenarios

Analysts elaborate a use case with various diagrams and descriptions. One of the more useful diagramming techniques for documenting a use case is an activity diagram, discussed later. Activity diagrams were first introduced in Chapter 4. Many analysts prefer to write narrative descriptions of use cases. Typically, use case descriptions are written at three separate levels of detail: brief description, intermediate description, and fully developed description. Written descriptions and activity diagrams can be used in any combination, depending on an analyst's needs.

BRIEF DESCRIPTION

A brief description can be used for very simple use cases, especially when the system to be developed is also a small, well-understood application. A simple use case would normally have a single scenario and very few, if any, exception conditions. A brief description used in conjunction with an activity diagram adequately describes a simple use case. Figure 6-7 provides a brief description of the *Create new order* use case. Generally, a use case such as *Create new order* is complex enough that either an intermediate or fully developed description is developed. We illustrate those descriptions next.

***Create new order* description**

When the customer calls to order, the order clerk and system verify customer information, create a new order, add items to the order, verify payment, create the order transaction, and finalize the order.

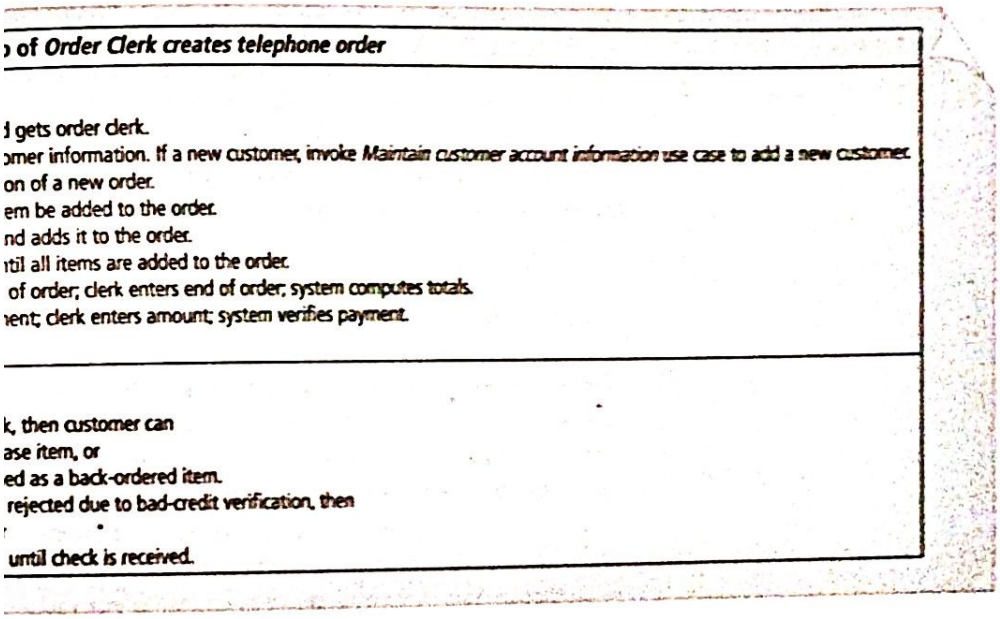
INTERMEDIATE DESCRIPTION

The intermediate-level use case description expands the brief description to include the internal flow of activities for the use case. If there are multiple scenarios, then each flow of activities is described individually. Exception conditions can be documented, if they are needed. Figures 6-8 and 6-9 show intermediate descriptions that document the two scenarios of *Order clerk creates telephone order* and *Customer creates Web order*. These two scenarios were identified earlier as separate workflows for the *Create new order* use case. Notice that each describes what the user and the system need to carry out the processing for the scenario. Exception conditions are also listed. Each step is identified with a number to make it easier to read. In many ways, this description resembles a type of writing called *structured English*, which can include sequence, decision, and repetition blocks.

rio of *Order Clerk creates telephone order*

INTERMEDIATE DESCRIPTION

The intermediate-level use case description expands the brief description to include the internal flow of activities for the use case. If there are multiple scenarios, then each flow of activities is described individually. Exception conditions can be documented, if they are needed. Figures 6-8 and 6-9 show intermediate descriptions that document the two scenarios of *Order clerk creates telephone order* and *Customer creates Web order*. These two scenarios were identified earlier as separate workflows for the *Create new order* use case. Notice that each describes what the user and the system need to carry out the processing for the scenario. Exception conditions are also listed. Each step is identified with a number to make it easier to read. In many ways, this description resembles a type of writing called *structured English*, which can include sequence, decision, and repetition blocks.



› of *Order Clerk creates telephone order*

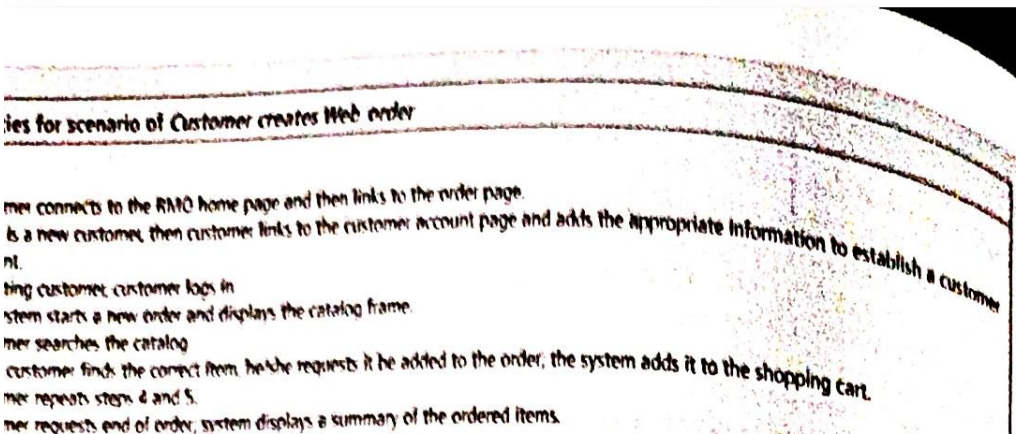
ł gets order clerk.
›mer information. If a new customer, invoke *Maintain customer account information* use case to add a new customer.
on of a new order.
em be added to the order.
nd adds it to the order.
ıtil all items are added to the order.
of order; clerk enters end of order; system computes totals.
ıent; clerk enters amount; system verifies payment.

k, then customer can
ase item, or
ed as a back-ordered item.
rejected due to bad-credit verification, then

until check is received.

FULLY DEVELOPED DESCRIPTION

The fully developed description is the most formal method for documenting a use case. Even though it takes a little more work to define all the components at this level, it is the preferred method of describing the internal flow of activities for a use case. One of the major difficulties that software developers have is struggling to obtain a deep understanding of the users' needs. But if you create a fully developed use case description, you increase the probability that you thoroughly understand the business processes and the ways the system must support them. Figure 6-10 is an example of a fully developed use case description of the telephone order scenario of the *Create new order* use case, and Figure 6-11 shows the Web order scenario for the same use case.



ies for scenario of Customer creates Web order

mer connects to the RMO home page and then links to the order page.
is a new customer, then customer links to the customer account page and adds the appropriate information to establish a customer
nt.
ting customer, customer logs in
stem starts a new order and displays the catalog frame.
mer searches the catalog
customer finds the correct item, he/she requests it be added to the order, the system adds it to the shopping cart.
mer repeats steps 4 and 5.
mer requests end of order, system displays a summary of the ordered items.

Figure 6-14 shows a generic SSD. As with a use case diagram, the stick figure represents an actor—a person (or role) that interacts with the system. In a use case diagram, the actor "uses" the system, but the emphasis in an SSD is on how the actor "interacts" with the system by entering input data and receiving output data. The idea is the same with both diagrams; the level of detail is different.

The box labeled :System is an object that represents the entire automated system. In SSDs and all interaction diagrams, instead of using class notation, analysts use object notation. Object notation indicates that the box refers to an individual object and not to the class of all similar objects. The notation is simply a rectangle with the name of the object underlined. The colon before the underlined class name is a frequently used, but optional, part of the object notation. In an interaction diagram, the messages are

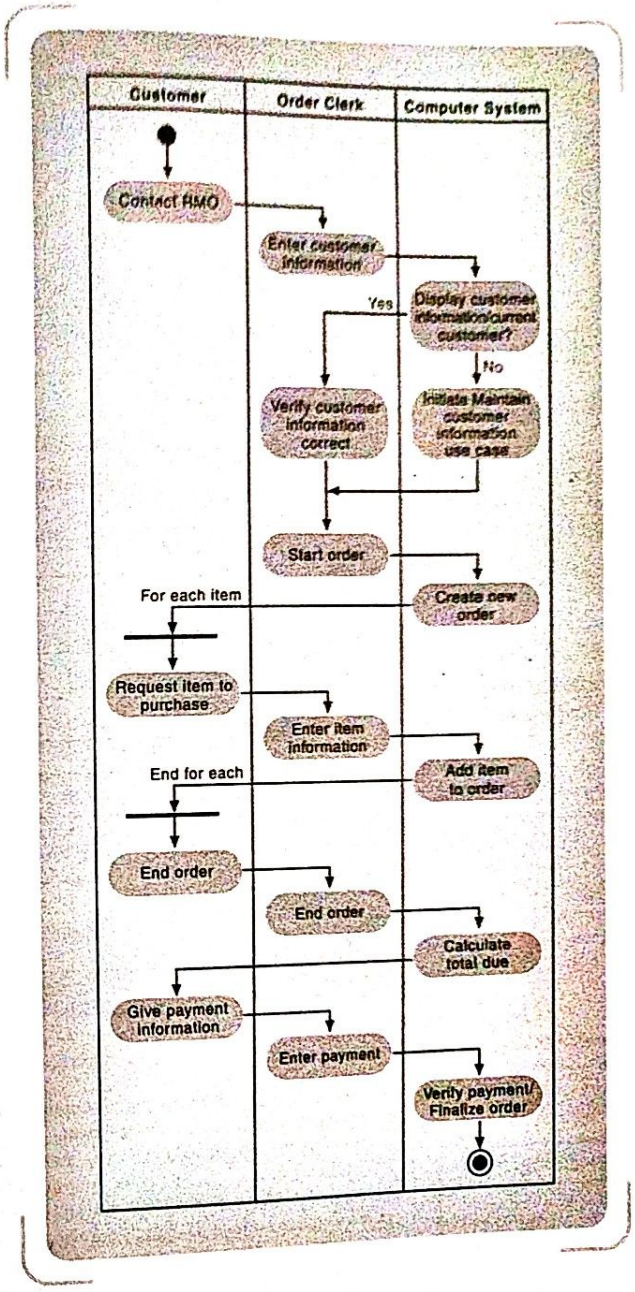
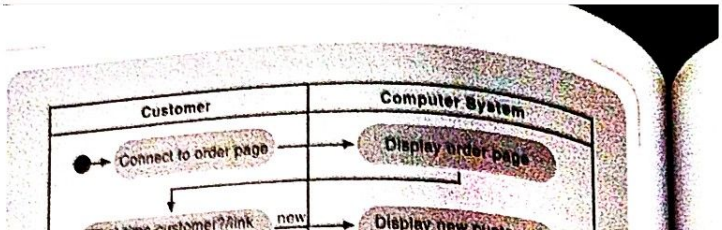


FIGURE 6-12
Activity diagram of the telephone order scenario



State Machine Diagrams | Unified Modeling Language (UML)

Last Updated : 03 Jan, 2025

A State Machine Diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. In this article, we will explain what is a state machine diagram, the components, and the use cases of the state machine diagram.

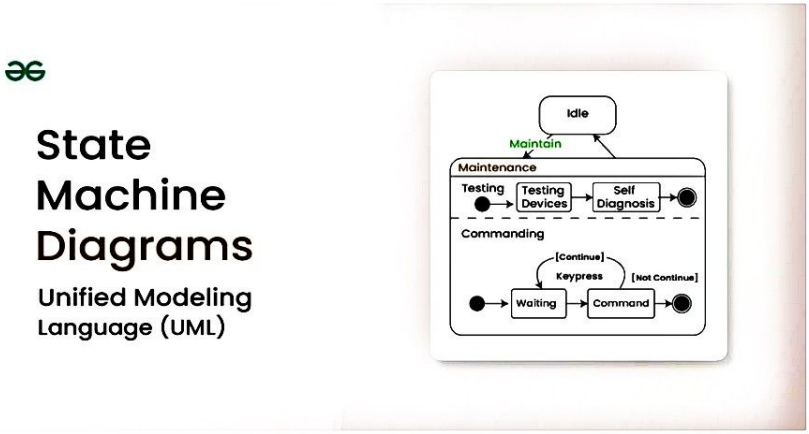


Table of Content

What is a State Machine Diagram?

A state diagram is a uml diagram which is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions.

- State Machine diagrams are also known as State Diagrams and State-Chart Diagrams. These both terms can be used interchangeably.
- A state machine diagram is used to model the dynamic behavior of a class

RULES FOR DEVELOPING STATECHARTS

Statechart development follows a set of rules. The rules help you to develop statecharts for classes in the problem domain. Usually the primary challenge in building a statechart is to identify the *right* states for the object. It may be helpful to pretend that you are the object itself. It is easy to pretend to be a customer, but a little more difficult to say, "I am an order," or, "I am a shipment. How do I come into existence? What states am I in?" However, if you can begin to think this way, it will help you develop statechart diagrams.

The other major area of difficulty for new analysts is to identify and handle composite states with nested threads. Usually, the primary cause of this difficulty is a lack of experience in thinking about concurrent behavior. The best solution is to remember that developing statecharts is an iterative behavior, more so than developing any other type of diagram. Analysts seldom get a statechart right the first time. They always draw it and then refine it again and again. Also, remember that when you are defining requirements, you are only getting a general idea of the behavior of an object. During design, as you build detailed sequence diagrams, you will have an opportunity to refine and correct important statecharts.

Finally, don't forget to ask about an exception condition—especially when you see the words *verify* or *check*. Normally, there will be two transitions out of states that verify something—one for acceptance and one for rejection.

Here is a list of steps that will help you get started in developing statecharts:

[1] **Review the class diagram and select the classes that will require statecharts.** Remember, we normally include only those that have various status conditions that are important to track in the system. Then begin with the classes that appear to have the simplest statecharts, such as the OrderItem class.

[2] **For each selected class in the group, make a list of all the status conditions you can identify.** At this point, simply brainstorm. If you are working in a team, have a brainstorming session with the whole team. Remember that you are defining states of being of the software classes. However, these states must also reflect the states for the real-world objects that are represented in software. Sometimes it is helpful to think of the physical object, identify states of the physical object, then translate those that are appropriate into corresponding system states or status conditions. It is also helpful to think of the life of the object. How does it come into existence in the system? When and how is it deleted from the system? Does it have active states? Does it have inactive states? Does it have states in which it is waiting? Think of activities done to the object or by the object. Often, the object will be in a particular state as these actions are occurring.

[3] **Begin building statechart fragments by identifying the transitions that cause an object to leave the identified state.** For example, if an Order is in a state of *Ready to be shipped*, then a transition such as beginShipping will cause the Order to leave that state.

[4] **Sequence these state-transition combinations in the correct order. Then aggregate these combinations into larger fragments.** As the fragments are being aggregated into larger paths, it is natural to begin to look for a natural life cycle for the object. Continue to build longer paths in the statechart by combining the fragments.

[5] **Review the paths and look for independent, concurrent paths.** When an item can be in two states concurrently, there are two possibilities. The two states may be on independent paths, as in the printer example of *Working* and *Full*. This occurs when the states and paths are independent, and one can

change without affecting the other. Alternately, one state may be a [illegible] state, so that the two states should be nested, one inside the other. One key to identify a candidate for a composite state is to determine if it is [illegible] with several other states, and these other states depend on the original state. For example, the On state has several other states and paths that can occur while the printer is in the On state, and those states depend on the printer being in the On state.

[6] **Look for additional transitions.** Often, during a first iteration, several of the possible combinations of state transition state are missed. One method to identify them is to take every paired combination of states and ask whether there is a valid transition between the states. Test for transitions in both directions.

[7] **Expand each transition with the appropriate message event, guard condition, and action-expression.** Include with each state appropriate action-expressions. Much of this may have been done as the statechart fragments were being built.

[8] **Review and test each statechart.** We test statecharts by "desk-checking" them. Review each of your statecharts by doing the following:

a. Make sure your states are really states of the object in the class. Ensure that the names of states truly describe states of being of the object.

b. Follow the life cycle of an object from its coming into existence to its being deleted from the system. Be sure that all possible combinations are covered and that the paths on the statechart are accurate.

c. Be sure your diagram covers all exception conditions as well as the normal expected flow of behavior.

d. Look again for concurrent behavior (multiple paths) and the possibility of nested paths (complex states).