Ques 30:- Define generalization and aggregation. Define in detail how they could be applied to actual situations.

Ans

**Generalization**

It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

It is a relationship between a general entity and a unique entity which is present inside the system.

In a generalization relationship, the object-oriented concept called **inheritance** can be implemented. A generalization relationship exists between two objects, also called as entities or things. In a generalization relationship, one entity is a parent, and another is said to be as a child. These entities can be represented using inheritance.

In inheritance, a child of any parent can access, update, or inherit the functionality as specified inside the parent object. A child object can add its functionality to itself as well as inherit the structure and behavior of a parent object.

This type of relationship collectively known as a generalization relationship.

**Aggregation.**

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a “part–of” or “has–a” relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

**Example**

In the relationship, “a car has–a motor”, car is the whole object or the aggregate, and the motor is a “part–of” the car. Aggregation may denote −

* **Physical containment** − Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
* **Conceptual containment** − Example, shareholder has–a share.

An aggregation is a subtype of an association relationship in UML. Aggregation and composition are both the types of association relationship in UML. An aggregation relationship can be described in simple words as " an object of one class can own or access the objects of another class."

In an aggregation relationship, the dependent object remains in the scope of a relationship even when the source object is destroyed.

Let us consider an example of a car and a wheel. A car needs a wheel to function correctly, but a wheel doesn't always need a car. It can also be used with the bike, bicycle, or any other vehicles but not a particular car. Here, the wheel object is meaningful even without the car object. Such type of relationship is called an aggregation relation.

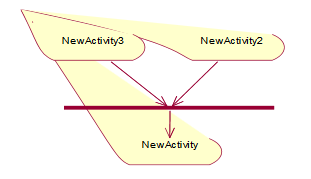
Ques 31:

Ans:

**Join in Activity diagram:**

1. A join in Activity diagram represents the synchronization of two or more concurrent flows of control.
2. A join may have two or more incoming transitions and one outgoing transition.
3. Above the join, activities associated with each of these paths continue in parallel.
4. At the join, the concurrent flows synchronize, means each waits until all incoming flows have reached at the join, at which point one flow of control continues on below the join.

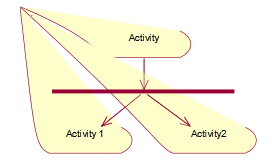
**Notation:**



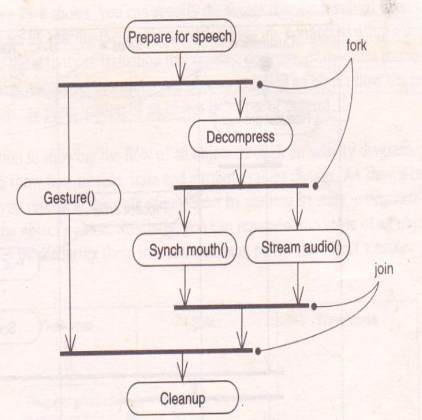
**Fork in Activity diagram:**

1. A fork in activity diagram represents the splitting of a single flow of control into two or more concurrent flows of control.
2. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.
3. Below the fork, the activities associated with each of these paths continue in parallel.

**Notation:**



**Example:**



In the above example there are two forks and two joins. Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join. The activity Stream audio needed to tell the activity Synch mouth when important pauses and intonations occurred. Similarly, for Synch mouth, we would see transitions triggered by these same signals, to which the Synch mouth state machine would respond.

Ques 32:

Ans

## [Events and Signals](https://praveenthomasln.wordpress.com/2012/04/06/events-and-signals-s8-cs/)

**Events**

* An event is the specification of a significant occurrence that has a location in time and space.
* Anything that happens is modeled as an event in UML.
* In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition
* four kinds of events – signals, calls, the passing of time, and a change in state.
* Events may be external or internal and asynchronous or synchronous.

*Asynchronous events* are events that can happen at arbitrary times eg:- signal, the passing of time, and a change of state.

*Synchronous events*, represents the invocation of an operation eg:- Calls

*External events* are those that pass between the system and its actors.

*Internal events* are those that pass among the objects that live inside the system.

*A signal* is an event that represents the specification of an asynchronous stimulus communicated between instances.

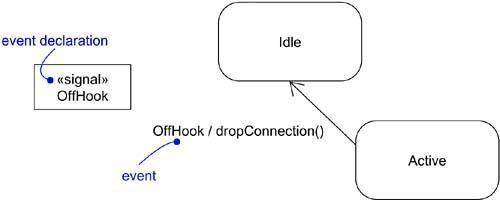


Figure 1: Events

**kinds of events**

## Signal Event

A signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal

* a signal event is an asynchronous event
* signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters
* A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction
* signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send

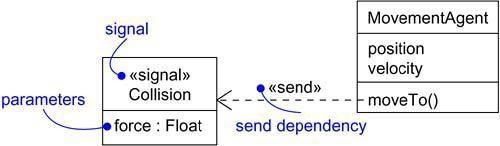


Figure: Signals

## Call Event

* + a call event represents the dispatch of an operation
  + a call event is a synchronous event

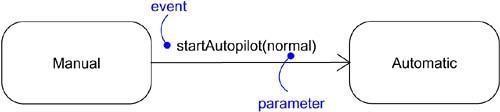


Figure: Call Events

## Time and Change Events

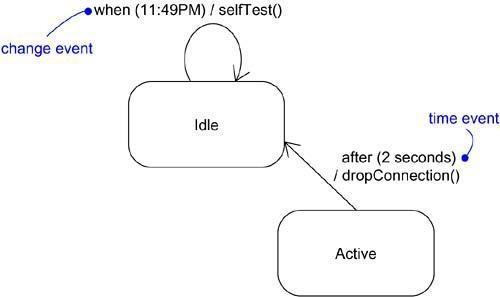
* A *time event* is an event that represents the passage of time.
* modeled by using the keyword ‘after’ followed by some expression that evaluates to a period of time which can be simple or complex.
* A *change event* is an event that represents a change in state or the satisfaction of some condition
* modeled by using the keyword ‘when’ followed by some Boolean expression

Figure : Time and Change Events

## Sending and Receiving Events

*For synchronous events* (Sending or Receiving) like call event, the sender and the receiver are in a rendezvous (the sender dispatches the signal and wait for a response from the receiver) for the duration of the operation. When an object calls an operation, the sender dispatches the operation and then waits for the receiver.

*For asynchronous events* (Sending or Receiving) like signal event, the sender and receiver do not rendezvous ie,the sender dispatches the signal but does not wait for a response from the receiver. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.

Call events can be modeled as operations on the class of the object.

Named signals can be modeled by naming them in an extra compartment of the class

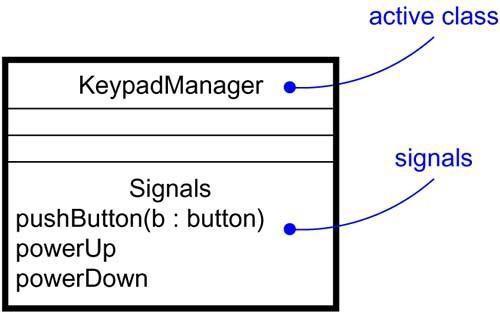


Figure : Signals and Active Classes

# Common Modeling Techniques Modeling family of signals

To model a family of signals,

* Consider all the different kinds of signals to which a given set of active objects may respond.
* Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
* Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

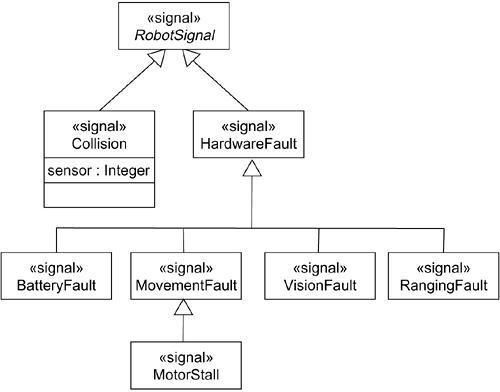


Figure : Modeling Families of Signals

# Modeling Exceptions

To model exceptions,

* For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
* Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
* For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation’s specification.

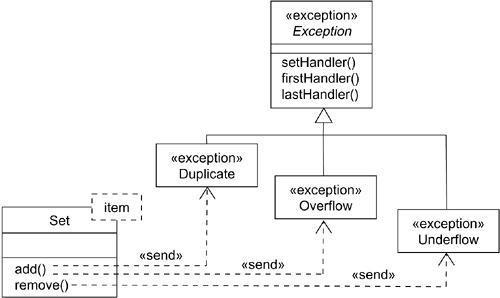


Figure : Modeling Exceptions

# State Machine

**State machine**

* A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events.
* Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.
* State machines are used to specify the behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past.
* state machines are used to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

## States

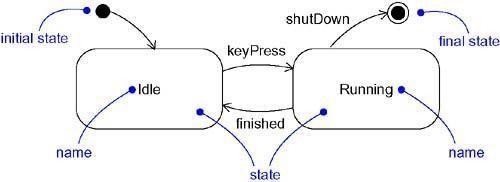
* A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
* An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle, Activating, Active, and Shutting Down.
* a state name must be unique within its enclosing state
* A state has five parts: *Name, Entry/exit actions, Internal transitions* – Transitions that are handled without causing a change in state,
* *Substates* – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates,
* *Deferred events* – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state
* initial state indicates the default starting place for the state machine or substate and is represented as a filled black circle
* final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle
* Initial and final states are pseudo-states

Figure : States

## Transitions

* A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
* Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.
* A transition has five parts:

*Source state* – The state affected by the transition,

*Event trigger* – a stimulus that can trigger a source state to fire on satisfying guard condition,

*Guard condition* – Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger,

*Action* – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object,

*Target state* – The state that is active after the completion of the transition.

* A transition may have multiple sources as well as multiple targets
* A *self-transition*is a transition whose source and target states are the same

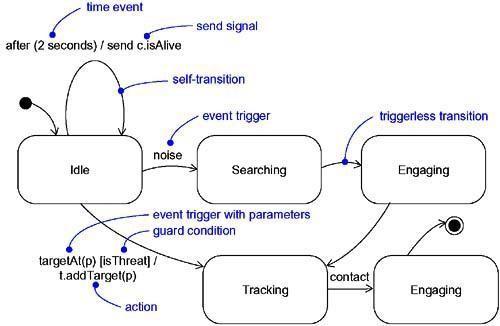


Figure : Transitions

## Event Trigger

* An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
* events may include signals, calls, the passing of time, or a change in state.
* An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.
* An event trigger may be polymorphic

## Guard condition

* a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event
* A guard condition is evaluated only after the trigger event for its transition occurs
* A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

## Action

* An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.
* Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object

*An activity* may be interrupted by other events.

# Advanced States and Transitions

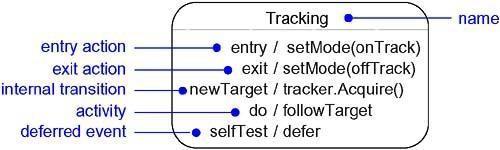


Figure : Advanced States and Transitions

*Entry and Exit Actions*

* Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event ‘entry’ with an appropriate action
* Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event ‘exit’, together with an appropriate action

*Internal Transitions*

* Internal Transitions are events that should be handled internally without leaving the state.
* Internal transitions may have events with parameters and guard conditions.

*Activities*

Activities make use of object’s idle time when inside a state. ‘do’ transition is used to specify the work that’s to be done inside a state after the entry action is dispatched.

*Deferred Events*

A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action ‘defer’.

# Substates

* A substate is a state that’s nested inside another one.
* A state that has substates is called a composite state.
* A composite state may contain either *concurrent (orthogonal) or sequential (disjoint) sub states*.
* Substates may be nested to any level

## Sequential Substates

* Sequential Substates are those sub states in which an event common to the composite states can easily be exercised by each states inside it at any time
* sequential substates partition the state space of the composite state into disjoint states
* A nested sequential state machine may have at most one initial state and one final state

*History States*

* A history state allows composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.
* a shallow history state is represented as a small circle containing the symbol H
* The first time entry to a composite state doesn’t have any history
* the symbol H designates a *shallow history*, which remembers only the history of the immediate nested state machine.
* the symbol H\* designates *deep history*, which remembers down to the innermost nested state at any depth.
* When only one level of nesting, shallow and deep history states are semantically equivalent.

## Concurrent Substates

* concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object
* Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow
* A nested concurrent state machine does not have an initial, final, or history state

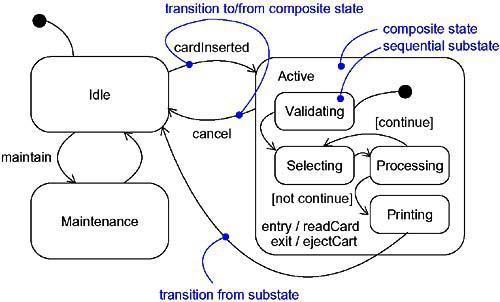


Figure : Sequential Substates

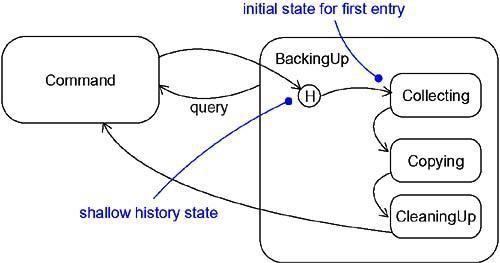


Figure : History State

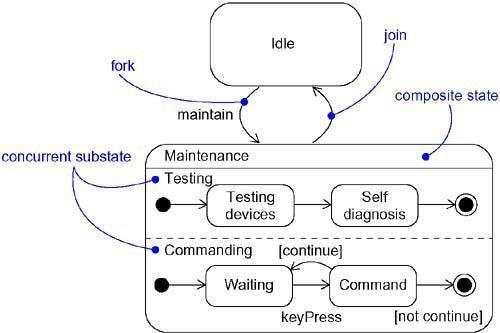


Figure : Concurrent Substates

**Common Modeling Techniques Modeling the Lifetime of an Object** To model the lifetime of an object,

* Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
* *If the context is a class or a use case*, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
* *If the context is the system as a whole*, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system’s lifetime, and except for the most trivial systems, a complete model would be intractable.
* Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
* Decide on the events to which this object may respond. If already specified, you’ll find these in the object’s interfaces; if not already specified, you’ll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
* Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
* Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
* Expand these states as necessary by using substates.
* Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
* Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
* Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
* After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object’s semantics.

For example, Figure shows the state machine for the controller in a home security system

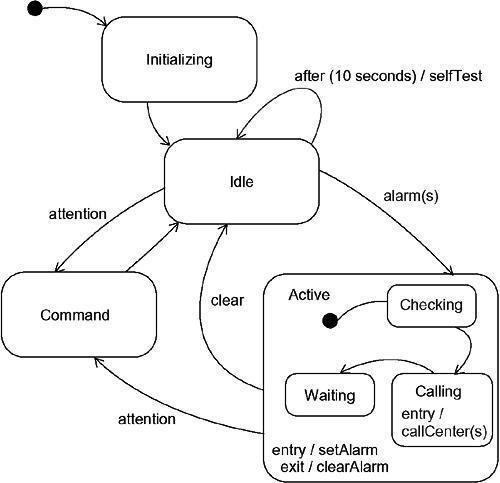


Figure : Modeling the Lifetime of An Object

# Processes and Threads

* A process is a heavyweight flow that can execute concurrently with other processes.
* A thread is a lightweight flow that can execute concurrently with other threads within the same process.
* An active object is an object that owns a process or thread and can initiate control activity.
* An active class is a class whose instances are active objects.
* Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes.

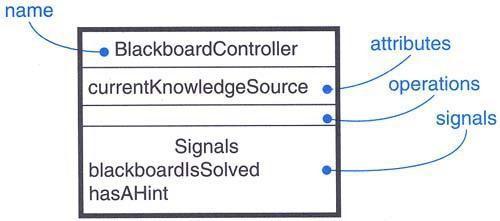


Figure : Active Class

# Flow of Control

*In a sequential system*, there is a single flow of control. i.e, one thing, and one thing only, can take place at a time.

*In a concurrent system*, there is multiple simultaneous flow of control i.e, more than one thing can take place at a time.

# Classes and Events

* Active classes are just classes which represents an independent flow of control
* Active classes share the same properties as all other classes.
* When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated
* *Two standard stereotypes* that apply to active classes are, ***<<process>>*** – Specifies a heavyweight flow that can execute concurrently with other processes. (heavyweight means, a thing known to the OS itself and runs in an independent address space) ***<<thread>>*** – Specifies a lightweight flow that can execute concurrently with other threads within the same process (lightweight means, known to the OS itself.)
* All the threads that live in the context of a process are peers of one another

# Communication

* In a system with both active and passive objects, there are *four possible combinations of interaction*
* *First*, a message may be passed from one passive object to another
* *Second*, a message may be passed from one active object to another
* In *inter-process communication* there are two possible styles of communication. *First*, one active object might synchronously call an operation of another. *Second*, one active object might asynchronously send a signal or call an operation of another object
* a synchronous message is rendered as a full arrow and an asynchronous message is rendered as a half arrow
* *Third*, a message may be passed from an active object to a passive object
* *Fourth*, a message may be passed from a passive object to an active one

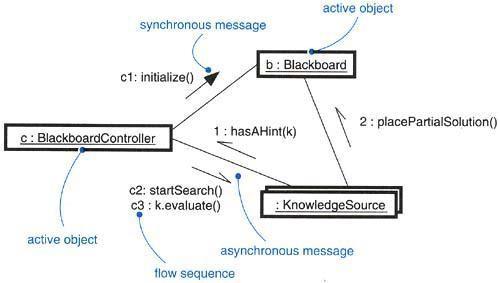


Figure 2: Communication

# Synchronization

* Synchronization means arranging the flow of controls of objects so that mutual exclusion will be guaranteed.
* in object-oriented systems these objects are treated as a critical region
* *three approaches* are there to handle synchronization:
* *Sequential* – Callers must coordinate outside the object so that only one flow is in the object at a time
* *Guarded* – multiple flow of control is sequentialized with the help of object’s guarded operations. in effect it becomes sequential.
* *Concurrent* – multiple flow of control is guaranteed by treating each operation as atomic
* synchronization are rendered in the operations of active classes with the help of constraints

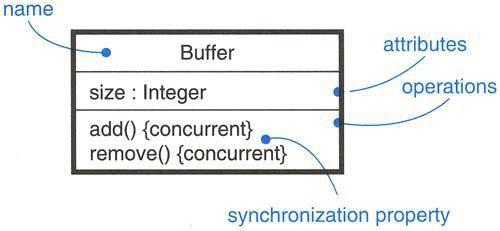


Figure: Synchronization

# Process Views

* The process view of a system encompasses the threads and processes that form the system’s concurrency and synchronization mechanisms.
* This view primarily addresses the performance, scalability, and throughput of the system.

**Common Modeling Techniques Modeling Multiple Flows of Control** To model multiple flows of control,

* Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over engineer the process view of your system by introducing too much concurrency.
* Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that each has the right set of attributes, operations, and signals.
* Capture these static decisions in class diagrams, explicitly highlighting each active class.
* Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
* Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
* Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

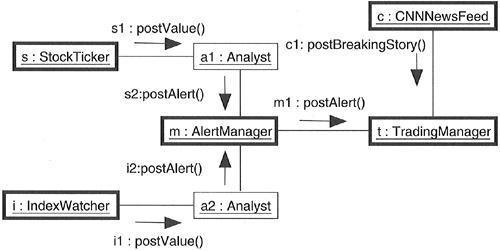


Figure : Modeling Flows of Control

# Modeling InterProcess Communication

To model interprocess communication,

* Model the multiple flows of control.
* Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
* Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
* Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

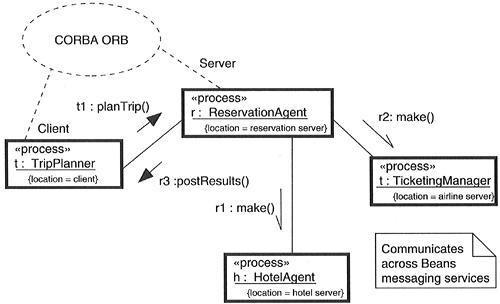


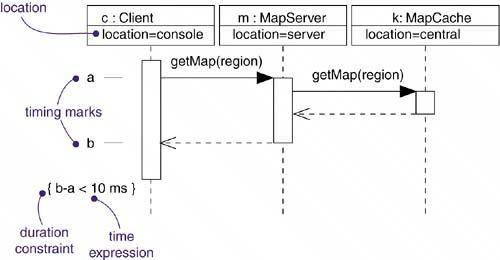
Figure : Modeling Interprocess Communication

# Time and Space

A distributed system is one in which components may be physically distributed across nodes. These nodes may represent different processors physically located in the same box, or they may even represent computers that are located half a world away from one another.

To represent the modeling needs of real time and distributed systems, the UML provides a graphic representation for timing marks, time expressions, timing constraints, and location.

Figure :Timing Constraints and Location



A [*timing mark*](http://umlguide2.uw.hu/gloss01.html#gloss01entry181) is a denotation for the time at which an event occurs. Graphically, a timing mark is depicted as a small hash mark (horizontal line) on the border of a sequence diagram.

A [*time expression*](http://umlguide2.uw.hu/gloss01.html#gloss01entry179) is an expression that evaluates to an absolute or relative value of time. A time expression can also be formed using the name of a message and an indication of a stage in its processing, for

example, request.sendTime or request.receiveTime.

A [*timing constraint*](http://umlguide2.uw.hu/gloss01.html#gloss01entry180) is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint-that is, a string enclosed by brackets and generally connected to an element by a dependency relationship.

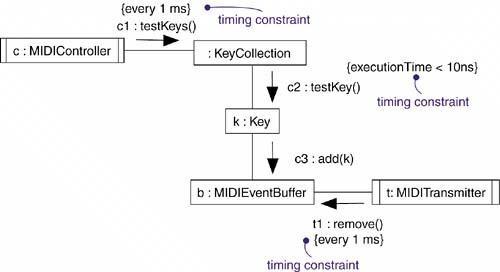
[*Location*](http://umlguide2.uw.hu/gloss01.html#gloss01entry98) is the placement of a component on a node. Location is an attribute of an object.

## Time

Real time systems are, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

The passing of messages represents the dynamic aspect of any system, They are mainly rendered with the name of an event, such as a signal or a call.

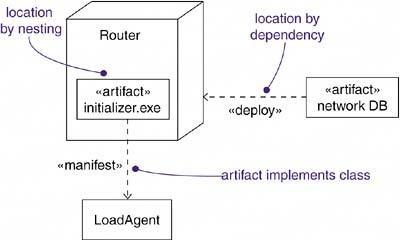
Figure : Time



## Location

Distributed systems, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

Figure : Location



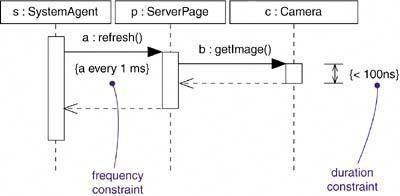
# Common Modeling Techniques

## Modeling Timing Constraints

To model timing constraints,

* For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.
* For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

Figure : Modeling Timing Constraint

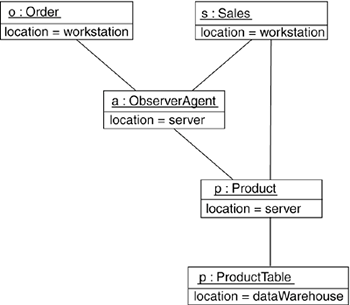


## Modeling the Distribution of Objects

To model the distribution of objects,

* For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbours and their locations. A tightly coupled locality will have neighbouring objects close by; a loosely coupled one will have distant objects.
* Tentatively allocate objects closest to the actors that manipulate them.
* Next consider patterns of interaction among related sets of objects.
* Partition sets of objects that have low degrees of interaction.
* Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
* Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
* Assign objects to components so that tightly coupled objects are on the same component.
* Assign components to nodes so that the computation needs of each node are within capacity. Add additional nodes if necessary.
* Balance performance and communication costs by assigning tightly coupled components to the same node.

Figure : Modeling the Distribution of Objects



**State chart Diagrams**

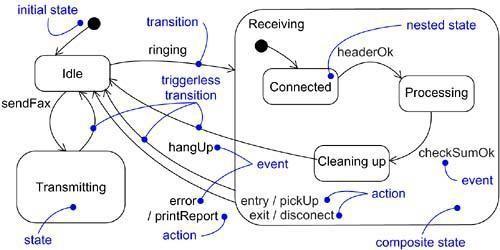


Figure: Statechart Diagram

* State chart diagram is simply a presentation of a state machine which shows the flow of control from state to state.
* State chart diagrams are important for constructing executable systems through forward and reverse engineering.
* state chart diagrams are useful in modeling the lifetime of an object
* State chart diagrams commonly contain – Simple states and composite states, Transitions- including events and actions
* It is one of the five diagrams in UML for modeling the dynamic aspects of systems.
* Graphically, a state chart diagram is a collection of vertices and arcs.

*A state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

*An even*t in the context of state machines is an occurrence of a stimulus that can trigger a state transition. *A transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.

*An activity* is ongoing non atomic execution within a state machine.

An action is an executable atomic computation that results in a change in state of the model or the return of a value.

*A reactive or event-driven object* is one whose behavior is best characterized by its response to events dispatched from outside its context

# Modeling Reactive Objects

To model a reactive object,

* Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
* Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
* Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
* Decide on the meaningful partial ordering of stable states over the lifetime of the object.
* Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
* Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
* Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
* Check that all states are reachable under some combination of events.
* Check that no state is a dead end from which no combination of events will transition the object out of that state.
* Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

The first string represents a tag; the second string represents the body of the message.

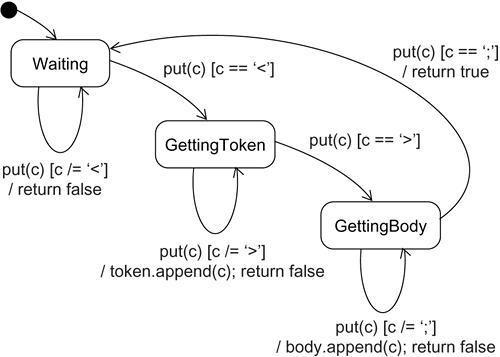
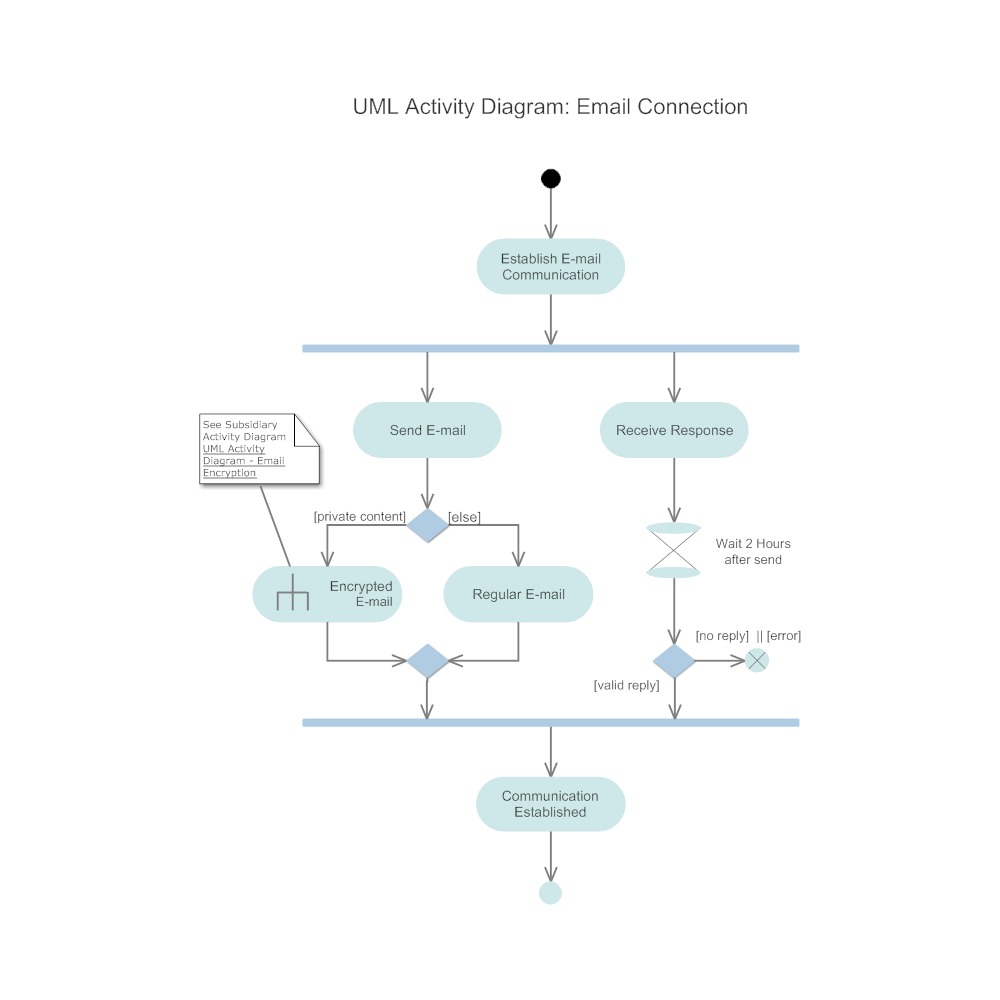


Figure : Modeling Reactive Objects

Ques 33:

Ans:



Ques 34:

Ans:

In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements.

UML relationships are grouped into the following categories:

|  |  |
| --- | --- |
| **Category** | **Function** |
| Activity edges | Represent the flow between activities |
| Associations | Indicate that instances of one model element are connected to instances of another model element |
| Dependencies | Indicate that a change to one model element can affect another model element |
| Generalizations | Indicate that one model element is a specialization of another model element |
| Realizations | Indicate that one model element provides a specification that another model element implements |
| Transitions | Represent changes in state |

You can set properties and use keywords to create variations of these relationships.

Relationships in class diagrams show the interaction between classes and classifiers. Such relationships indicate the classifiers that are associated with each other, those that are generalizations and realizations, and those that have dependencies on other classes and classifiers.

The following topics describe the relationships that you can use in class diagrams:

* [Abstraction relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cabstract.html?view=kc)

An abstraction relationship is a dependency between model elements that represents the same concept at different levels of abstraction or from different viewpoints. You can add abstraction relationships to a model in several diagrams, including use-case, class, and component diagrams.

* [Aggregation relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/caggreg.html?view=kc)

In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier.

* [Association relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cassn.html?view=kc)

In UML models, an association is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.

* [Association classes](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cassnclss.html?view=kc)

In UML diagrams, an association class is a class that is part of an association relationship between two other classes.

* [Binding relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cbind.html?view=kc)

In UML models, a binding relationship is a relationship that assigns values to template parameters and generates a new model element from the template.

* [Composition association relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/ccompasn.html?view=kc)

A composition association relationship represents a whole–part relationship and is a form of aggregation. A composition association relationship specifies that the lifetime of the part classifier is dependent on the lifetime of the whole classifier.

* [Dependency relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cdepend.html?view=kc)

In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier. You can use dependency relationships in class diagrams, component diagrams, deployment diagrams, and use-case diagrams to indicate that a change to the supplier might require a change to the client.

* [Directed association relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cdirasn.html?view=kc)

In UML models, directed association relationships are associations that are navigable in only one direction.

* [Element import relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/celeimport.html?view=kc)

In UML diagrams, an element import relationship identifies a model element in another package, and allows the element in the other package to be referenced by using its name without a qualifier.

* [Generalization relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cgeneral.html?view=kc)

In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.

* [Interface realization relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cimplement.html?view=kc)

In UML diagrams, an interface realization relationship is a specialized type of implementation relationship between a classifier and a provided interface. The interface realization relationship specifies that the realizing classifier must conform to the contract that the provided interface specifies.

* [Instantiation relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cinstantiate.html?view=kc)

In UML diagrams, an instantiation relationship is a type of usage dependency between classifiers that indicates that the operations in one classifier create instances of the other classifier.

* [Package import relationship](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cpkgimport.html?view=kc)

In UML diagrams, a package import relationship allows other namespaces to use unqualified names to refer to package members.

* [Realization relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/creal.html?view=kc)

In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Several clients can realize the behavior of a single supplier. You can use realization relationships in class diagrams and component diagrams.

* [Usage relationships](https://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cusage.html?view=kc)

In UML modeling, a usage relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation.

Ques 35:

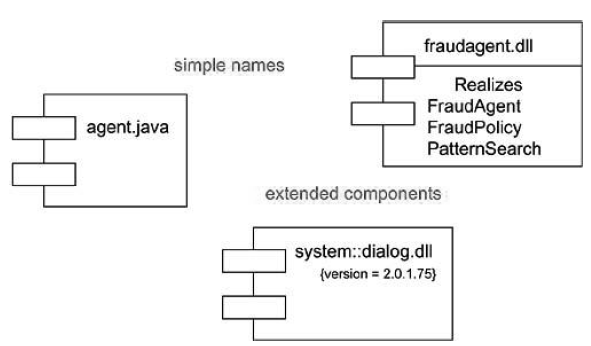
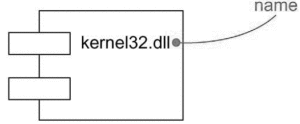
Ans:

**Introduction**

A component is a physical replaceable part of a system that complies with and provides the realization of a set of interfaces. We use components to model the physical things that may reside on a node, such as executables, libraries, tables, files and documents.

A component typically represents the physical packaging of otherwise logical elements such as classes, interfaces and collaborations. We do logical modeling to visualize, specify, and document our decisions about the vocabulary of our domain and the structural and behavioral way those things collaborate.

We do physical modeling to construct the executable system. Object libraries, executables, COM+ components and Enterprise Java Beans are all examples of components.



**Kinds of Components**

Three kinds of components may be distinguished.

First, there are deployment components. These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).

Second, there are work product components. These components are generally the residue of the development process, consisting of things such as the source code files and data files from which deployment components are created.

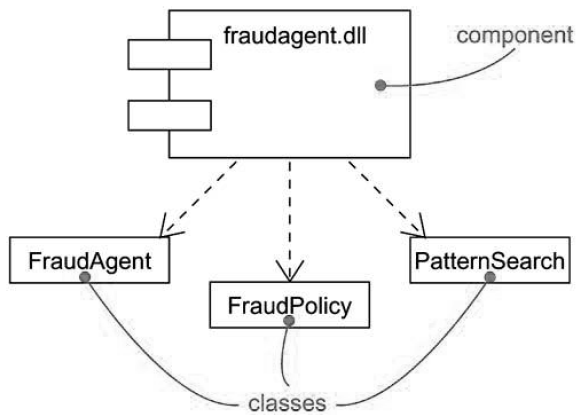
Third are execution components. These components are created as a consequence of an executing system, such as COM+ object, which is instantiated from a DLL.

**Components and Classes**

In many ways, components are like classes. Both have names, both may realize a set of interfaces, both may participate in dependency, generalization and association relationships, both may be nested, and both may have instances. However, there are some significant differences between components and classes:

* Classes represent logical abstractions, components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
* Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
* Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

The relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship as shown below:



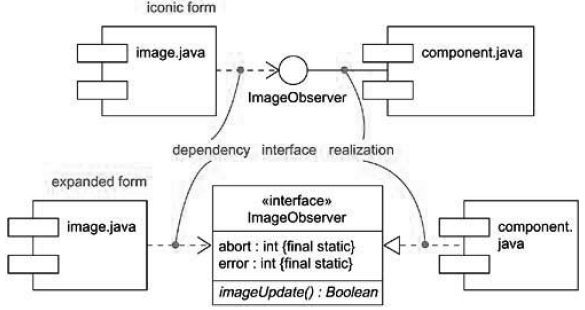
**Components and Interfaces**

An interface is a collection of operations that are used to specify a service of a class or a component. We can show the relationship between a component and its interfaces in one of the two ways.

The first style renders the interfaces in its elided, iconic form. The component that realizes the interfaces is connected to the interface using an elided realization relationship.

The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship.

In both cases, the component that accesses the services of the other component through the interfaces is connected to the interface using a dependency relationship.



Ques 36:

Ans

**What is a Sequence Diagram?**

Sequence diagrams, commonly used by developers, model the interactions between objects in a single use case. They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.

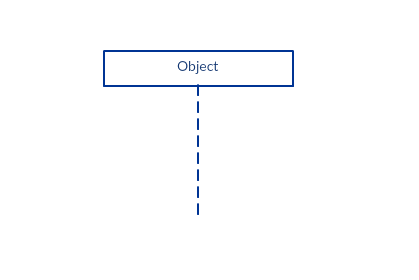
In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

**Sequence Diagram Notations**

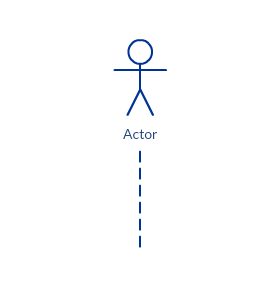
A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions. Each object has a column and the messages exchanged between them are represented by arrows.

**A Quick Overview of the Various Parts of a Sequence Diagram**

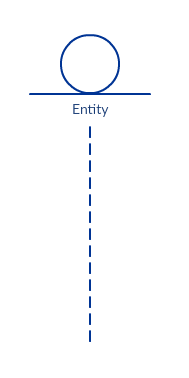
**Lifeline Notation**

A sequence diagram is made up of several of these lifeline notations that should be arranged horizontally across the top of the diagram. No two lifeline notations should overlap each other. They represent the different objects or parts that interact with each other in the system during the sequence.

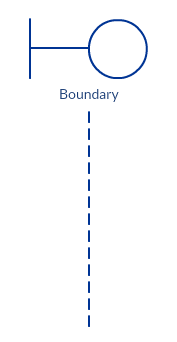
A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.



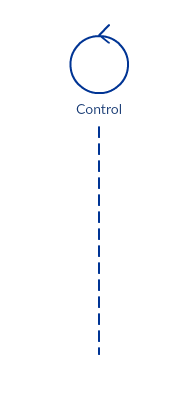
A lifeline with an entity element represents system data. For example, in a customer service application, the Customer entity would manage all data related to a customer.



A lifeline with a boundary element indicates a system boundary/ software element in a system; for example, user interface screens, database gateways or menus that users interact with, are boundaries.



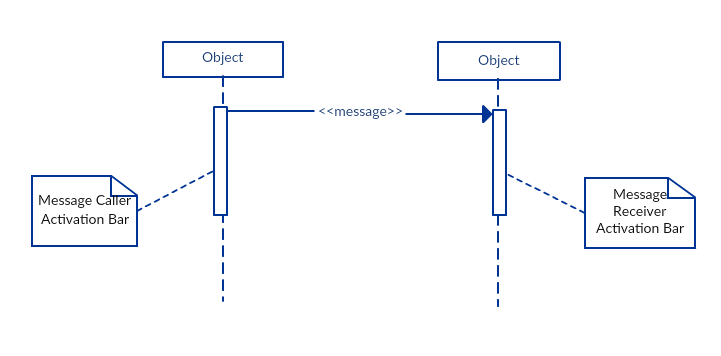
And a lifeline with a control element indicates a controlling entity or manager. It organizes and schedules the interactions between the boundaries and entities and serves as the mediator between them.



**Activation Bars**

Activation bar is the box placed on the lifeline.  It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.

In a sequence diagram, an interaction between two objects occurs when one object sends a message to another. The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.



**Message Arrows**

An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.   A message can flow in any direction; from left to right, right to left or back to the Message Caller itself. While you can describe the message being sent from one object to the other on the arrow, with different arrowheads you can indicate the type of message being sent or received.

The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message\_name are optional.

*attribute = message\_name (arguments): return\_type*

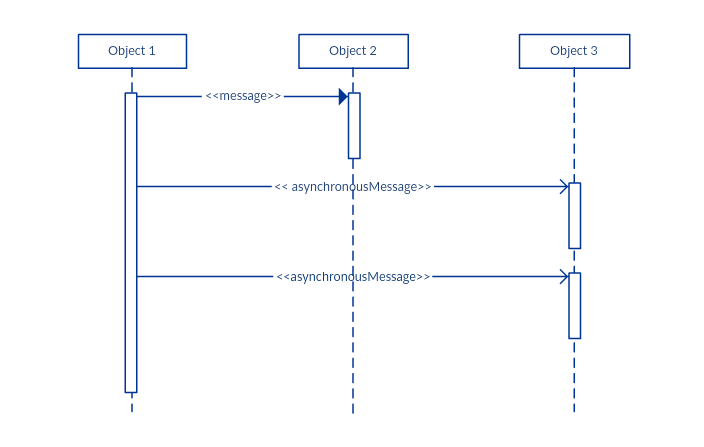
* *Synchronous message*

As shown in the activation bars example, a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message.  The arrowhead used to indicate this type of message is a solid one, like the one below.



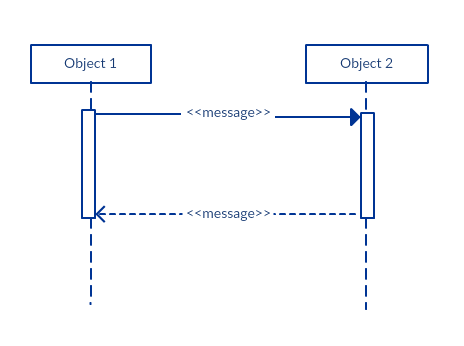
* *Asynchronous message*

An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrowhead used to show this type of message is a line arrow like shown in the example below.



* *Return message*

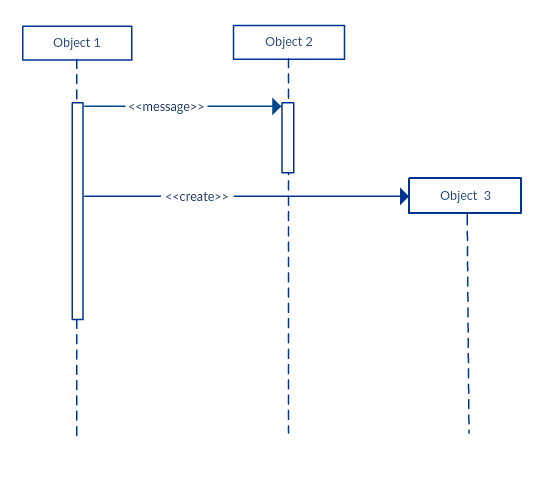
A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller. Return messages are optional notation pieces, for an activation bar that is triggered by a synchronous message always implies a return message.

Tip: You can avoid cluttering up your diagrams by minimizing the use of return messages since the return value can be specified in the initial message arrow itself.  


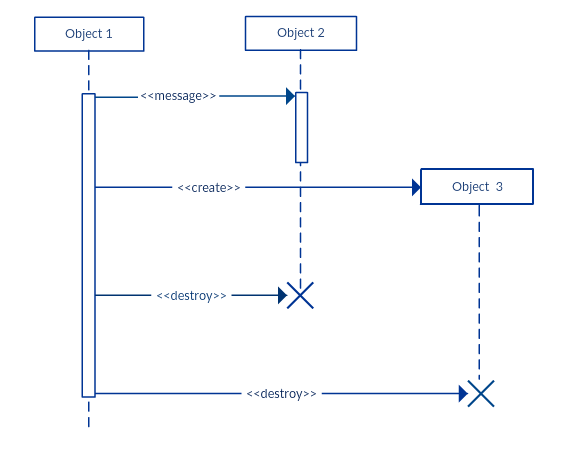
* *Participant  creation message*

Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.

The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent.  If the created participant does something immediately after its creation, you should add an activation box right below the participant box.

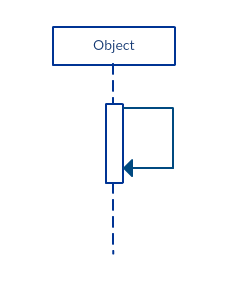


* *Participant destruction message*

Likewise, participants when no longer needed can also be deleted from a sequence diagram. This is done by adding an ‘X’ at the end of the lifeline of the said participant. 

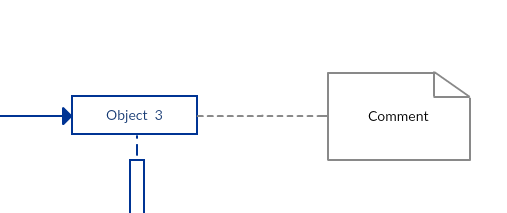
* *Reflexive message*

When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example below.



**Comment**

[UML diagrams](https://creately.com/lp/uml-diagram-tool/) generally permit the annotation of comments in all [UML diagram types](https://creately.com/blog/diagrams/uml-diagram-types-examples/). The comment object is a rectangle with a folded-over corner as shown below. The comment can be linked to the related object with a dashed line.



Note:View Sequence Diagram Best Practices to learn about sequence fragments.

**Sequence Diagram Best Practices**

* **Manage complex interactions with sequence fragments**

A sequence fragment is represented as a box that frames a section of interactions between objects (as shown in the examples below) in a sequence diagram.

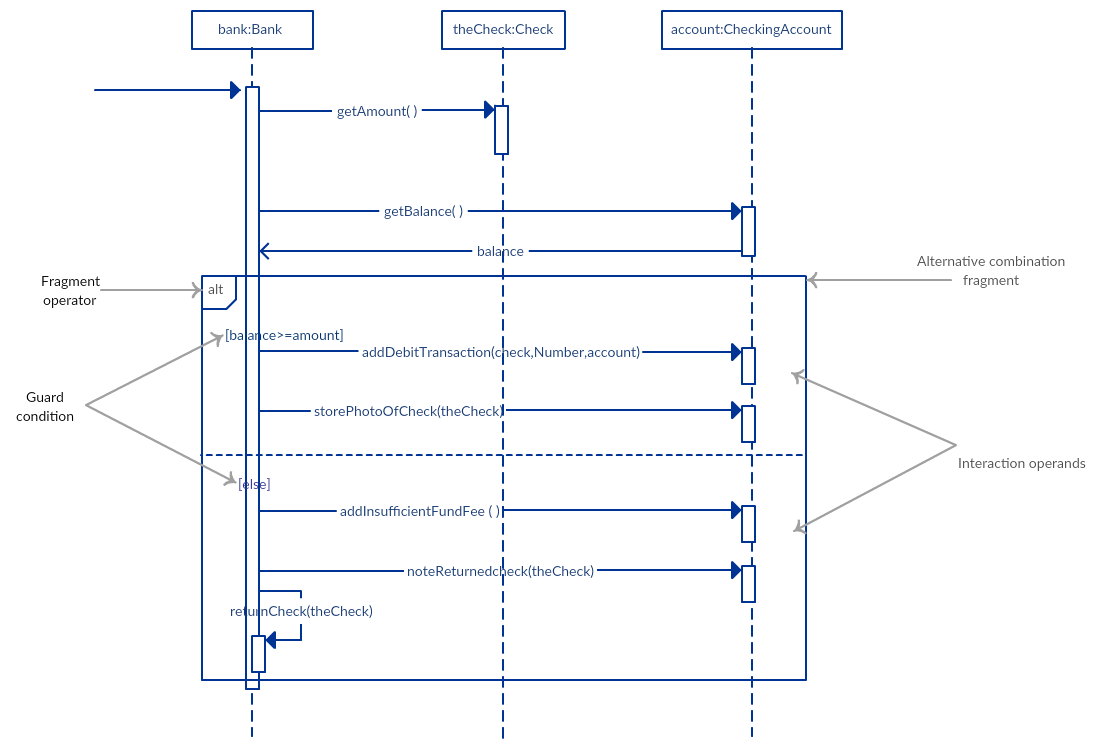
It is used to show complex interactions such as alternative flows and loops in a more structured way. On the top left corner of the fragment sits an operator. This – the fragment operator – specifies what sort of a fragment it is.

*Alternatives*

The alternative combination fragment is used when a choice needs to be made between two or more message sequences. It models the “if then else” logic.

The alternative fragment is represented by a large rectangle or a frame; it is specified by mentioning ‘alt’ inside the frame’s name box (a.k.a. fragment operator).

To show two or more alternatives, the larger rectangle is then divided into what is called interaction operands using a dashed line, as shown in the sequence diagram example above. Each operand has a guard to test against and it is placed at the top left corner of the operand.



*Options*

The option combination fragment is used to indicate a sequence that will only occur under a certain condition, otherwise, the sequence won’t occur. It models the “if then” statement.  

Similar to the alternative fragment, the option fragment is also represented with a rectangular frame where ‘opt’ is placed inside the name box.

Unlike the alternative fragment, an option fragment is not divided into two or more operands. Option’s guard is placed at the top left corner.

*(Find an example sequence diagram with an option fragment in the  Sequence Diagram Templates and Examples section).*

*Loops*

Loop fragment is used to represent a repetitive sequence. Place the words ‘loop’ in the name box and the guard condition near the top left corner of the frame.

In addition to the Boolean test, the guard in a loop fragment can have two other special conditions tested against. These are minimum iterations (written as *minint = [the number]* and maximum iterations (written as maxint = [the number]).

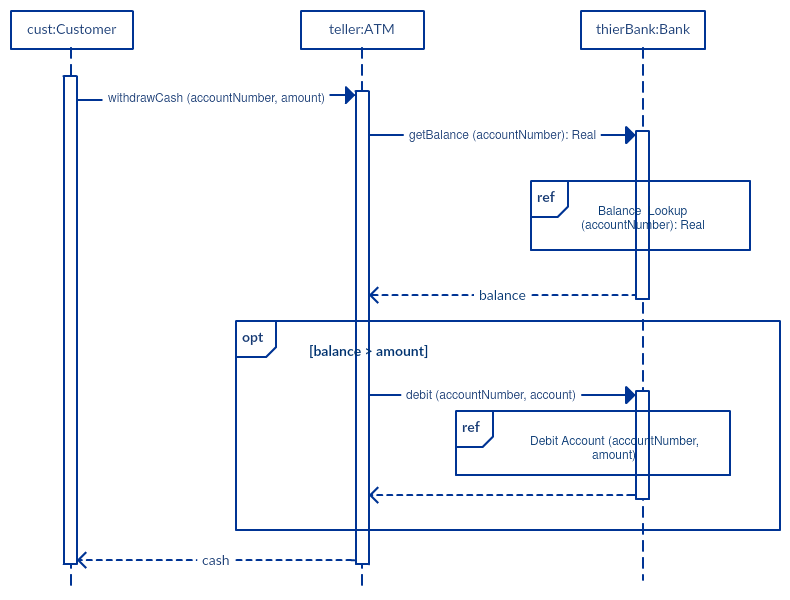
If it is a minimum iterations guard, the loop must execute not less than the number mentioned, and if it is a maximum iterations guard, the loop mustn’t execute more than the number indicated.

(Find an example of a loop fragment below in the sequence diagram templates and example section)

*Reference Fragment*

You can use the ref fragment to manage the size of large sequence diagrams. It allows you to reuse part of one sequence diagram in another, or in other words, you can reference part of a diagram in another diagram using the ref fragment.

To specify the reference fragment, you have to mention ‘ref’ in the name box of the frame and the name of the sequence diagram that is being referred to inside the frame.



*For more sequence fragments refer to Beyond the Basics of Sequence Diagrams:* [*Part 1*](https://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-1/)*,* [*Part 2*](https://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-2/) *and* [*Part 3*](https://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-3/)*.*

* **Draw smaller sequence diagrams that capture the essence of the use case**

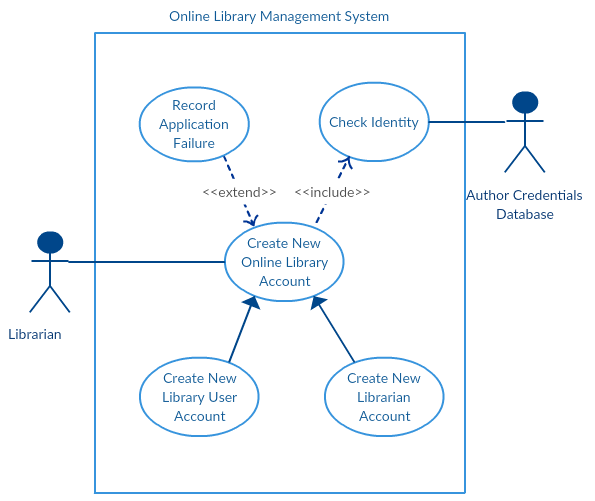
Instead of cluttering your sequence diagram with several objects and groups of messages that will confuse the reader, draw a few smaller sequence diagrams that aptly explain what your system does.  Make sure that the diagram fits on a single page and leaves space for explanatory notes too.

Also instead of drawing dozens of sequence diagrams, find out what is common among the scenarios and focus on that. And if the code is expressive and can stand on its own, there’s no need to draw a sequence diagram in the first place.

**How to Draw a Sequence Diagram**

A sequence diagram represents the scenario or flow of events in one single use case. The message flow of the sequence diagram is based on the narrative of the particular use case.

Then, before you start drawing the sequence diagram or decide what interactions should be included in it, you need to [draw the use case diagram](https://creately.com/diagram-type/use-case) and ready a comprehensive description of what the particular use case does.



From the above use case diagram example of ‘Create New Online Library Account’, we will focus on the use case named ‘Create New User Account’ to draw our sequence diagram example.

Before drawing the sequence diagram, it’s necessary to identify the objects or actors that would be involved in creating a new user account. These would be;

* Librarian
* Online Library Management system
* User credentials database
* Email system

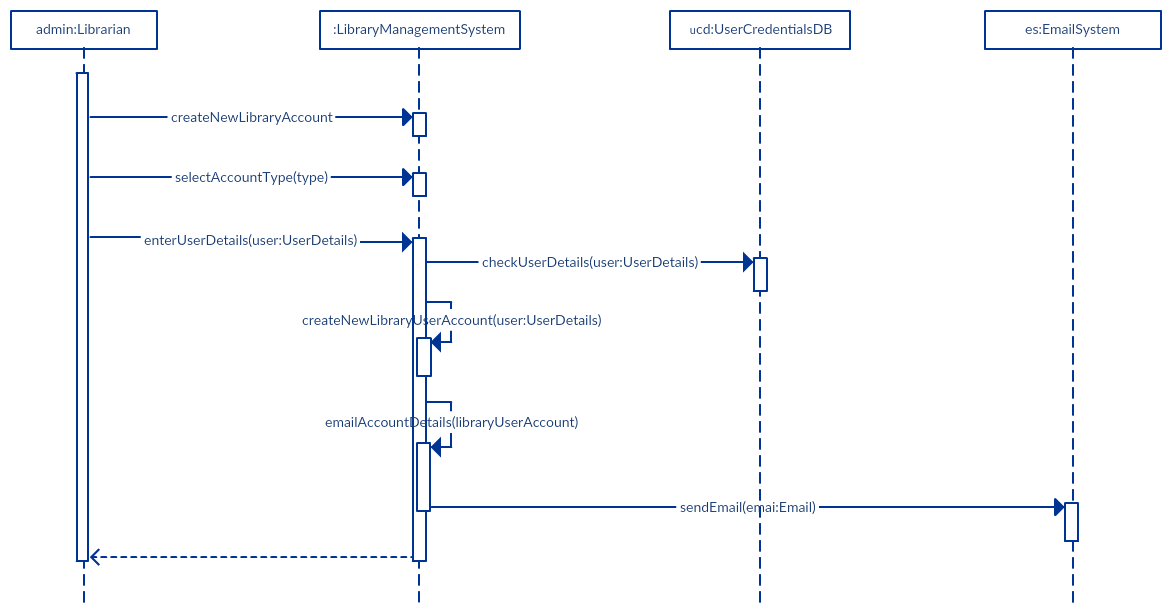
Once you identify the objects, it is then important to write a detailed description on what the use case does. From this description, you can easily figure out the interactions (that should go in the sequence diagram) that would occur between the objects above, once the use case is executed.

Here are the steps that occur in the use case named ‘Create New Library User Account’.

* The librarian request the system to create a new online library account
* The librarian then selects the library user account type
* The librarian enters the user’s details
* The user’s details are checked using the user Credentials Database
* The new library user account is created
* A summary of the of the new account’s details are then emailed to the user

From each of these steps, you can easily specify what messages should be exchanged between the objects in the sequence diagram. Once it’s clear, you can go ahead and start drawing the sequence diagram.

The sequence diagram below shows how the objects in the [online library management system](https://creately.com/diagram/example/hnabkygq/online%20library%20management%20system) interact with each other to perform the function ‘Create New Library User Account’.



**Sequence Diagram Common Mistakes**

When drawing sequence diagrams, designers tend to make these common mistakes. By avoiding these mistakes you can ensure the quality of your diagram.

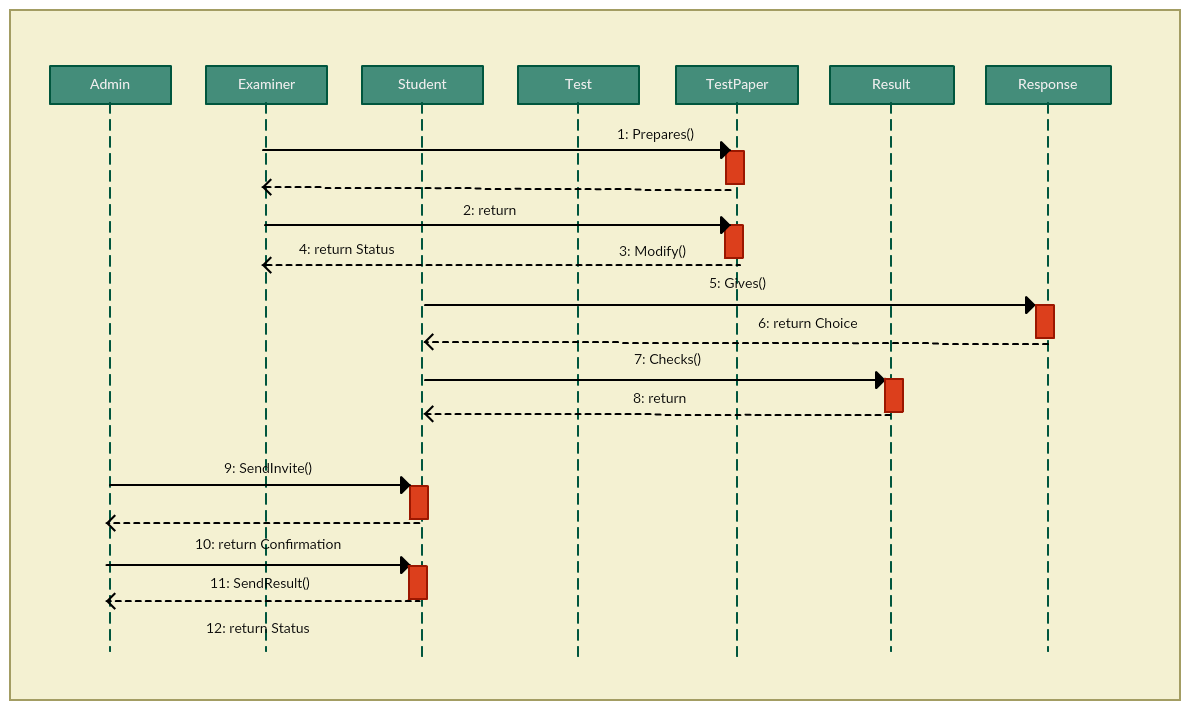
* Adding too much detail. This clutters up the diagram and makes it difficult to read.
* Obsolete and out of date sequence diagrams that are irrelevant when compared to the interfaces, actual architectures etc. of the system. Don’t forget to replace them or modify them.
* Leaving no blank space between the use case text and the message arrow; this makes it difficult for anyone to read the diagram.
* Not considering the origins of message arrows carefully.

See these common mistakes explained in detail in Sequence Diagram Guide: [Common Mistakes to Avoid When Drawing Sequence Diagrams](https://creately.com/blog/diagrams/10-common-mistakes-to-avoid-in-sequence-diagrams/).

**Sequence Diagram Examples and Templates**

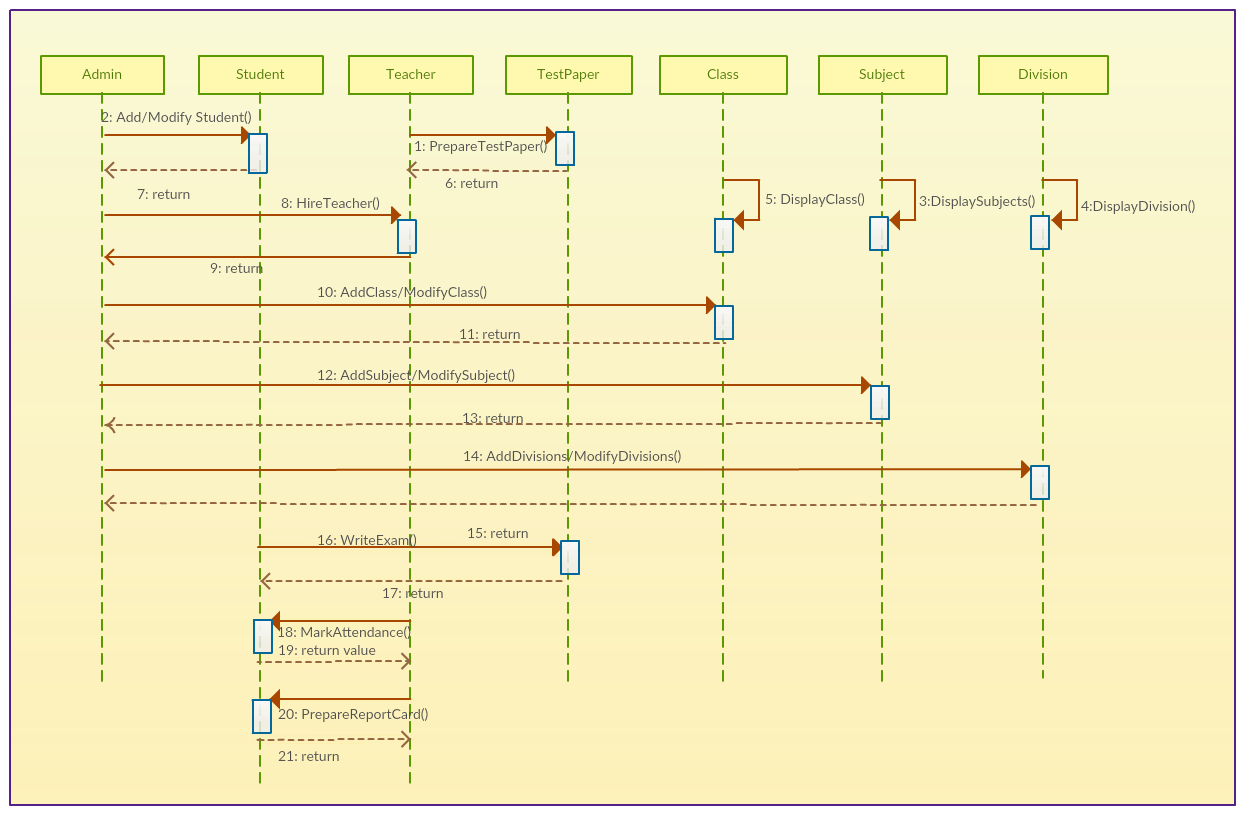
Following are a few [sequence diagram examples](https://creately.com/diagram-community/examples/t/sequence-diagram) and templates that are drawn using Creately. [Create sequence diagrams online](https://creately.com/diagram-type/uml-sequence-diagrams) using Creately’s online tool. Click on the template to open it in the editor.

*Sequence Diagram of an Online Exam System*

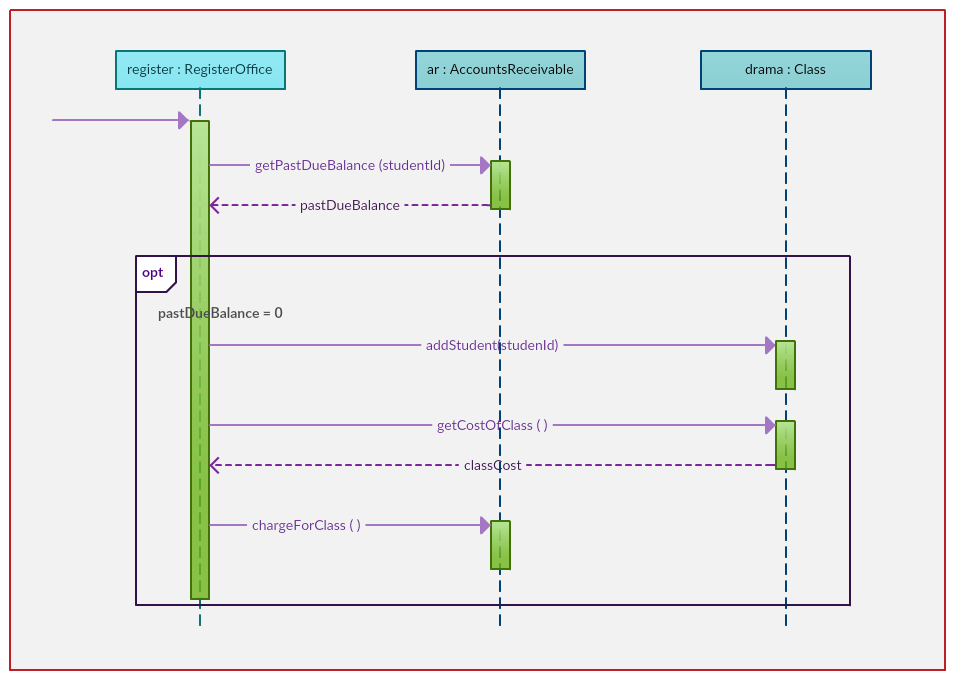


Click the image to edit it online

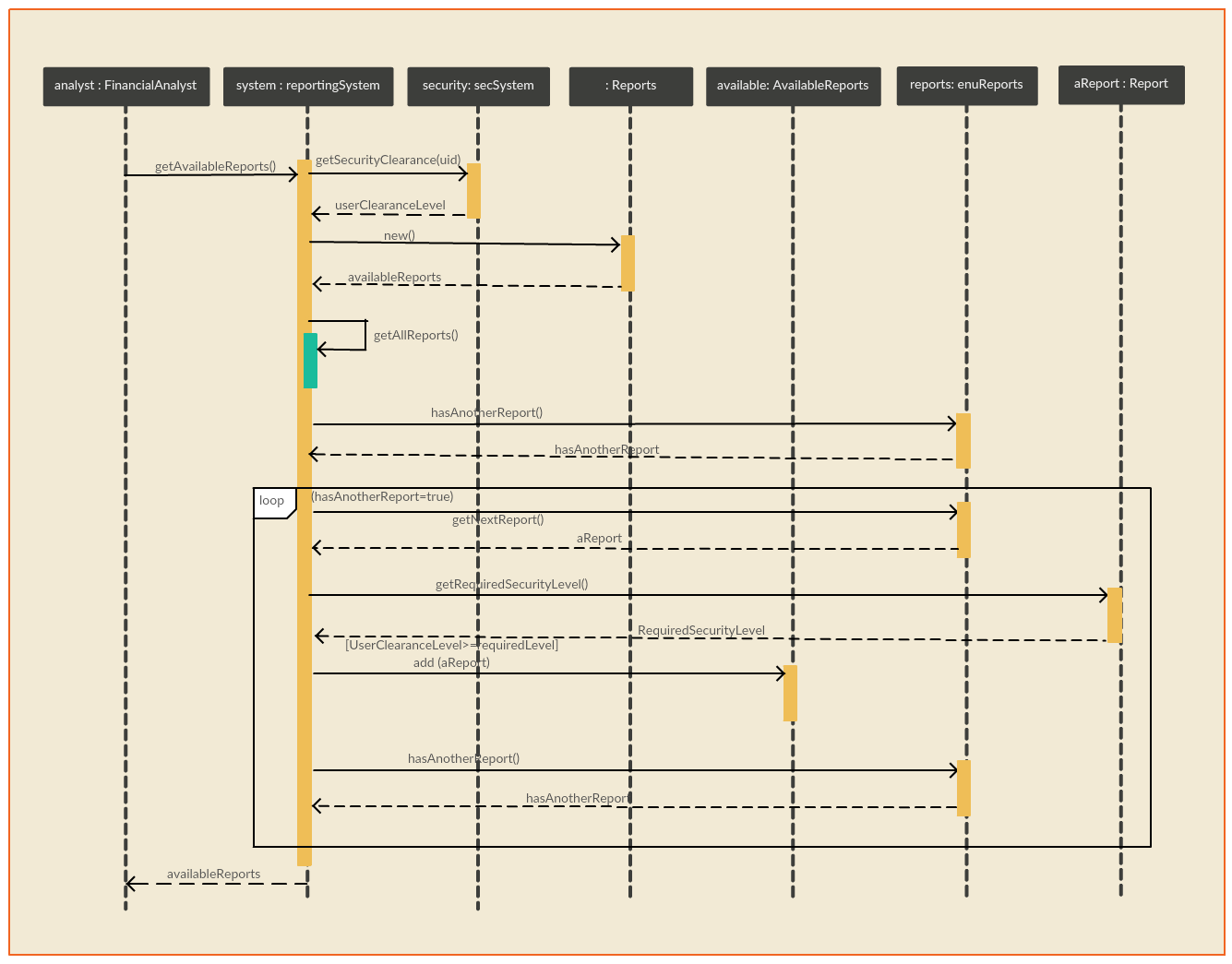
*Sequence Diagram Example of a School Management System*



*Example of an Option Combination Fragment*



*Example of a Loop Sequence*



Ques 37:

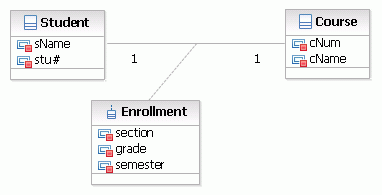
Ans:

In UML diagrams, an association class is a class that is part of an association relationship between two other classes.

You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.

As the following figure illustrates, an association class is connected to an association by a dotted line.



In UML models, an association is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.

An association represents a structural relationship that connects two classifiers. Like attributes, associations record the properties of classifiers. For example, in relationships between classes, you can use associations to show the design decisions that you made about classes in your application that contain data, and to show which of those classes need to share data. You can use an association's navigability feature to show how an object of one class gains access to an object of another class or, in a reflexive association, to an object of the same class.

The name of an association describes the nature of the relationship between two classifiers and should be a verb or phrase.

In the diagram editor, an association appears as a solid line between two classifiers.

**Association ends**

An association end specifies the role that the object at one end of a relationship performs. Each end of a relationship has properties that specify the role of the association end, its multiplicity, visibility, navigability, and constraints.

**Example**

In an e-commerce application, a customer class has a single association with an account class. The association shows that a customer instance owns one or more instances of the account class. If you have an account, you can locate the customer that owns the account. Given a particular customer, you can navigate to each of the customer’s accounts. The association between the customer class and the account class is important because it shows the structure between the two classifiers.

Ques 40:

Ans:

In UML, a communication diagram shows the interactions between the objects or roles associated with lifelines and the messages that pass between lifelines. In earlier versions of UML, this diagram was called a collaboration diagram and had a different notation.

Communication diagrams are a type of interaction diagram that you can use to explore the dynamic behavior of a system or software application. They provide an alternate view of the same information as sequence diagrams. In sequence diagrams, the focus is the ordering of the messages over time; in communication diagrams the focus is the structure of the messages that pass between the objects in the interaction. These diagrams illustrate the flow of messages between objects and the implied relationships between classes.

You can use communication diagrams to explore how objects in a system or application work together. Communication diagrams can identify the following aspects of an interaction or task:

* Objects that participate in the interaction
* Interfaces that the participating classes require
* Structural changes that an interaction requires
* Data that is passed between the objects in an interaction

Communication diagrams look similar to object diagrams, in which a lifeline represent the objects in the interaction and arrows represent the messages that are passed between the lifelines. Arrowheads indicate the direction of the messages, forward or reverse, and sequence numbers indicate the order in which the messages are passed.

The following topics describe the elements in communication diagrams:

* [Interaction frames](https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.7.0/com.ibm.xtools.sequence.doc/topics/cframes_v.html?view=kc)

In sequence diagrams and communication diagrams, an interaction frame provides a context or boundary to the diagram in which you create diagram elements, such as lifelines or messages, and in which you observe behavior.

* [Lifelines in UML diagrams](https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.7.0/com.ibm.xtools.sequence.doc/topics/clifel_v.html?view=kc)

In UML diagrams, such as sequence or communication diagrams, lifelines represent the objects that participate in an interaction. For example, in a banking scenario, lifelines can represent objects such as a bank system or customer. Each instance in an interaction is represented by a lifeline.

* [Message pathways in communication diagrams](https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.7.0/com.ibm.xtools.sequence.doc/topics/cmessagepathway.html?view=kc)

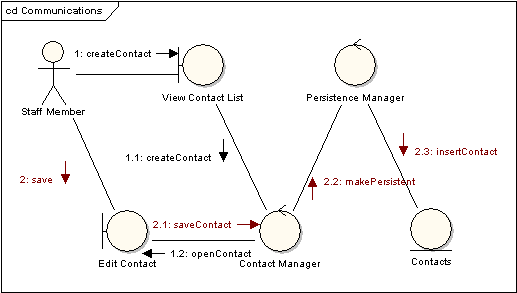
In communication diagrams, a message pathway is a connector between the roles or objects represented by lifelines in the diagram. The pathway identifies the objects that can pass messages in the interaction.

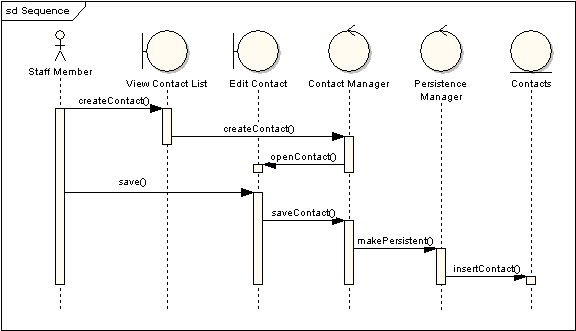
* [Messages in UML diagrams](https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.7.0/com.ibm.xtools.sequence.doc/topics/cmsg_v.html?view=kc)

A message is an element in a Unified Modeling Language (UML) diagram that defines a specific kind of communication between instances in an interaction. A message conveys information from one instance, which is represented by a lifeline, to another instance in an interaction.

They are very useful for visualizing the relationship between objects collaborating to perform a particular task. This is difficult to determine from a sequence **diagram**. In addition, **communication diagrams** can also help you determine the accuracy of your static model (i.e., class **diagrams**).

A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus is on object relationships.  
  
On communication diagrams, objects are shown with association connectors between them. Messages are added to the associations and show as short arrows pointing in the direction of the message flow. The sequence of messages is shown through a numbering scheme.   
  
The following two diagrams show a communication diagram and the sequence diagram that shows the same information. Although it is possible to derive the sequencing of messages in the communication diagram from the numbering scheme, it isn’t immediately visible. What the communication diagram does show quite clearly though, is the full set of messages passed between adjacent objects.





Ques 41:

Ans:

 Both activity and state chart diagrams model the dynamic behavior of the system. Activity diagram is essentially a flowchart showing flow of control from activity to activity. A state chart diagram shows a state machine emphasizing the flow of control from state to state.

 An activity diagram is a special case of a state chart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state (An activity is an ongoing non-atomic execution within a state machine).

 Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. State chart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, and document the dynamics of an individual object.

State chart Diagram:

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

**Purpose of Statechart Diagrams**

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

* To model the dynamic aspect of a system.
* To model the life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model the states of an object.

**How to Draw a Statechart Diagram?**

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

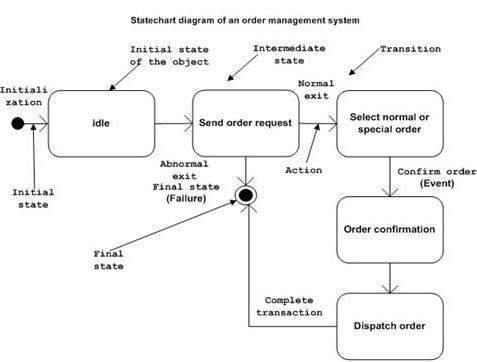
Before drawing a Statechart diagram we should clarify the following points −

* Identify the important objects to be analyzed.
* Identify the states.
* Identify the events.

Following is an example of a Statechart diagram where the state of Order object is analyzed

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



**Where to Use Statechart Diagrams?**

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as −

* To model the object states of a system.
* To model the reactive system. Reactive system consists of reactive objects.
* To identify the events responsible for state changes.
* Forward and reverse engineering.

Activity Diagram:

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

**Purpose of Activity Diagrams**

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

* Draw the activity flow of a system.
* Describe the sequence from one activity to another.
* Describe the parallel, branched and concurrent flow of the system.

**How to Draw an Activity Diagram?**

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

* Activities
* Association
* Conditions
* Constraints

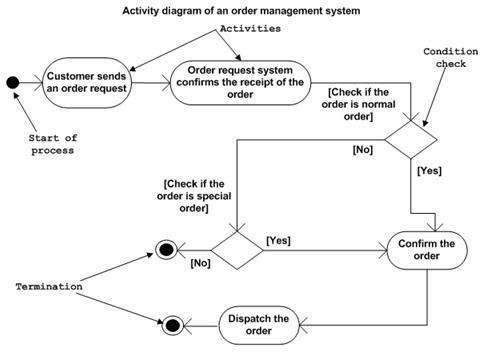
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

* Send order by the customer
* Receipt of the order
* Confirm the order
* Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.



**Where to Use Activity Diagrams?**

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

We will now look into the practical applications of the activity diagram. From the above discussion, it is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

* Modeling work flow by using activities.
* Modeling business requirements.
* High level understanding of the system's functionalities.
* Investigating business requirements at a later stage.

Ques 42

Ans:

**Types of Interaction diagram and Notations**

Following are the different types of interaction diagrams defined in UML:

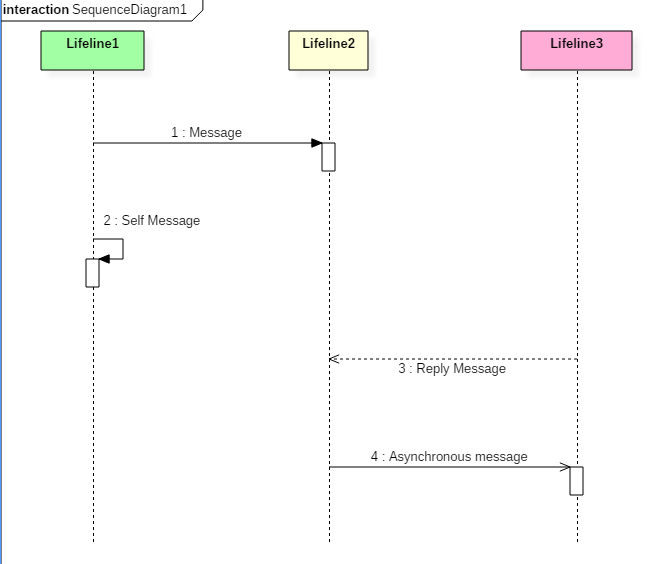
* Sequence diagram
* Collaboration diagram
* Timing diagram

The basic notation of interaction is a rectangle with a pentagon in the upper left corner of a rectangular box.

**What is a Sequence Diagram?**

A **SEQUENCE DIAGRAM** simply depicts interaction between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.

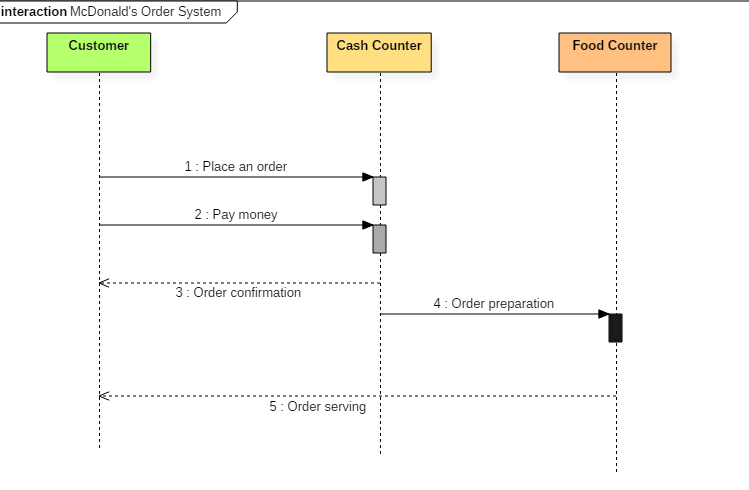
* A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
* In a sequence diagram, a lifeline is represented by a vertical bar.
* A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
* In a sequence diagram, different types of messages and operators are used which are described above.
* In a sequence diagram, iteration and branching are also used.



The above sequence diagram contains lifeline notations and notation of various messages used in a sequence diagram such as a create, reply, asynchronous message, etc.

**Sequence diagram example**

The following sequence diagram example represents McDonald's ordering system:

Sequence diagram of Mcdonald's ordering system

The ordered sequence of events in a given sequence diagram is as follows:

1. Place an order.
2. Pay money to the cash counter.
3. Order Confirmation.
4. Order preparation.
5. Order serving.

If one changes the order of the operations, then it may result in crashing the program. It can also lead to generating incorrect or buggy results. Each sequence in the above-given sequence diagram is denoted using a different type of message. One cannot use the same type of message to denote all the interactions in the diagram because it creates complications in the system.

You must be careful while selecting the notation of a message for any particular interaction. The notation must match with the particular sequence inside the diagram.

**Benefits of a Sequence Diagram**

* Sequence diagrams are used to explore any real application or a system.
* Sequence diagrams are used to represent message flow from one object to another object.
* Sequence diagrams are easier to maintain.
* Sequence diagrams are easier to generate.
* Sequence diagrams can be easily updated according to the changes within a system.
* Sequence diagram allows reverse as well as forward engineering.

**Drawbacks of a sequence diagram**

* Sequence diagrams can become complex when too many lifelines are involved in the system.
* If the order of message sequence is changed, then incorrect results are produced.
* Each sequence needs to be represented using different message notation, which can be a little complex.
* The type of message decides the type of sequence inside the diagram.

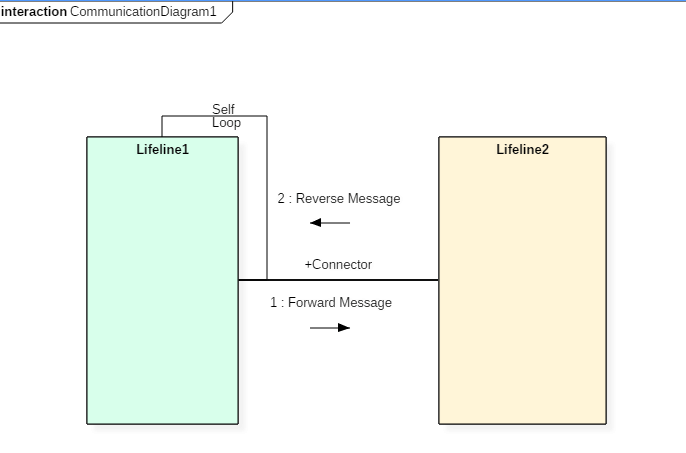
**What is the Collaboration diagram?**

**COLLABORATION DIAGRAM** depicts the relationships and interactions among software objects. They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram. They are also known as “Communication Diagrams.”

As per Object-Oriented Programming (OOPs), an object entity has various attributes associated with it. Usually, there are multiple objects present inside an object-oriented system where each object can be associated with any other object inside the system. Collaboration Diagrams are used to explore the architecture of objects inside the system. The message flow between the objects can be represented using a collaboration diagram.

**Benefits of Collaboration Diagram**

* It is also called as a communication diagram.
* It emphasizes the structural aspects of an interaction diagram - how lifeline connects.
* Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
* Messages passed over sequencing is indicated by numbering each message hierarchically.
* Compared to the sequence diagram communication diagram is semantically weak.
* Object diagrams are special case of communication diagram.
* It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
* Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.
* While modeling collaboration diagrams w.r.t sequence diagrams, some information may be lost.



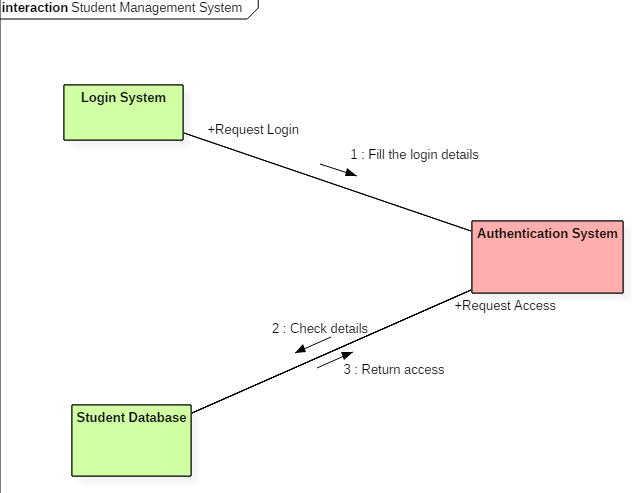
The above collaboration diagram notation contains lifelines along with connectors, self-loops, forward, and reverse messages used in a collaboration diagram.

**Drawbacks of a Collaboration Diagram**

* Collaboration diagrams can become complex when too many objects are present within the system.
* It is hard to explore each object inside the system.
* Collaboration diagrams are time consuming.
* The object is destroyed after the termination of a program.
* The state of an object changes momentarily, which makes it difficult to keep track of every single change the occurs within an object of a system.

**Collaboration diagram Example**

Following diagram represents the sequencing over student management system:

Collaboration diagram for student management system

The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,

1. A student requests a login through the login system.
2. An authentication mechanism of software checks the request.
3. If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

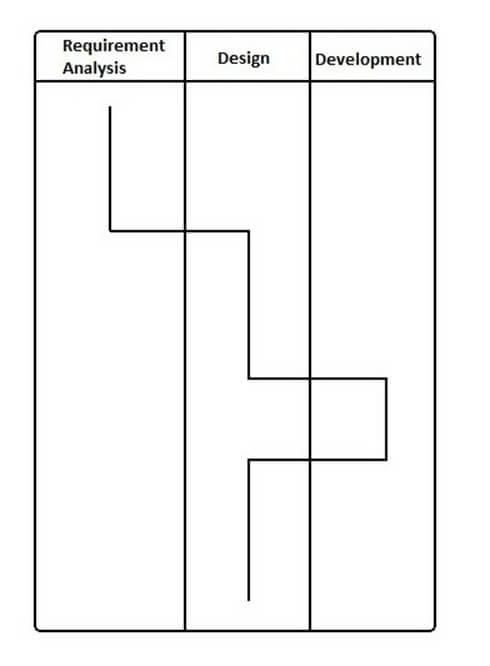
**What is Timing diagram?**

**TIMING DIAGRAM** is a waveform or a graph that is used to describe the state of a lifeline at any instance of time. It is used to denote the transformation of an object from one form into another form. Timing diagram does not contain notations as required in the sequence and collaboration diagram. The flow between the software program at various instances of time is represented using a waveform.

* It is a proper representation of interactions that focuses upon the specific timings of messages sent between various objects.
* Timing diagrams are used to explain the detailed time processing of a particular object.
* Timing diagrams are used to explain how an object changes within its lifetime.
* Timing diagrams are mostly used with distributed and embedded systems.
* In UML, timing diagrams are read from left to right according to the name of a lifeline specified at the left edge.
* Timing diagrams are used to represent various changes that occur within a lifeline from time to time.
* Timing diagrams are used to display a graphical representation of various states of a lifeline per unit time.
* UML provides various notations to simplify the transition state between two lifelines per unit time.

**Timing diagram Example**

The timing diagram given below represents a few phases of a software development life cycle.



In the above diagram, first, the software passes through the requirements phase then the design and later the development phase. The output of the previous phase at that given instance of time is given to the second phase as an input. Thus, the timing diagram can be used to describe SDLC (Software Development Life Cycle) in UML.

**Benefits of a Timing Diagram**

* Timing diagrams are used to represent the state of an object at a particular instance of time.
* Timing diagram allows reverse as well as forward engineering.
* Timing diagram can be used to keep track of every change inside the system.

**Drawbacks of a Timing Diagram**

* Timing diagrams are difficult to understand.
* Timing diagrams are difficult to maintain.

Ques 43:

Ans:

**What are the Strengths and Weaknesses of Sequence vs. Communication Diagrams?**

Each diagram type has advantages, and modelers have idiosyncratic preference—there isn't an absolutely "correct" choice. However, *UML* tools usually emphasize sequence diagrams, because of their greater notational power.

Sequence diagrams have some advantages over communication diagrams. Perhaps first and foremost, the *UML* specification is more sequence diagram centric—more thought and effort has been put into the notation and semantics. Thus, tool support is better and more notation options are available. Also, it is easier to see the call-flow sequence with sequence diagrams—simply read top to bottom. With communication diagrams we must read the sequence numbers, such as "1:" and "2:". Hence, sequence diagrams are excellent for documentation or to easily read a reverse-engineered call-flow sequence, generated from source code with a *UML* tool.

But on the other hand, communication diagrams have advantages when applying "*UML* as sketch" to draw on walls (an Agile Modeling practice) because they are *much* more space-efficient. This is because the boxes can be easily placed or erased anywhere—horizontal or vertical. Consequently as well, *modifying* wall sketches is easier with communication diagrams—it is simple (during creative high-change *OO* design work) to erase a box at one location, draw a new one elsewhere, and sketch a line to it. In contrast, new objects in a sequence diagrams must always be added to the right edge, which is limiting as it quickly consumes and exhausts right-edge space on a page (or wall); free space in the vertical dimension is not efficiently used. Developers doing sequence diagrams on walls rapidly feel the drawing pain when contrasted with communication diagrams.

**Note**

*three ways to use UML p.11*

Likewise, when drawing diagrams that are to be published on narrow pages (like this book), communication diagrams have the advantage over sequence diagrams of allowing *vertical* expansion for new objects—much more can be packed into a small visual space.

|  |  |  |
| --- | --- | --- |
| **Type** | **Strengths** | **Weaknesses** |
| sequence | clearly shows sequence or time ordering of messages  large set of detailed notation options | forced to extend to the right when adding new objects; consumes horizontal space |
| communication | space economical—flexibility to add new objects in two dimensions | more difficult to see sequence of messages  fewer notation options |

Ques 44:

Ans:

**State**

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

**Parts of a state**

* **Name** − A string differentiates one state from another. A state may not have any name.
* **Entry/Exit Actions** − It denotes the activities performed on entering and on exiting the state.
* **Internal Transitions** − The changes within a state that do not cause a change in the state.
* **Sub–states** − States within states.

**Initial and Final States**

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

**Transition**

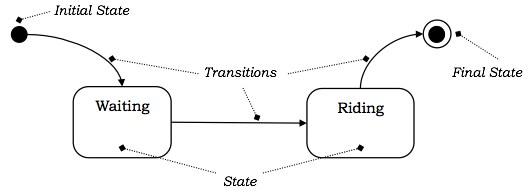
A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state−transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination state.

The five parts of a transition are −

* **Source State** − The state affected by the transition.
* **Event Trigger** − The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.
* **Guard Condition** − A Boolean expression which if True, causes a transition on receiving the event trigger.
* **Action** − An un-interruptible and atomic computation that occurs on the source object due to some event.
* **Target State** − The destination state after completion of transition.

**Example**

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.



**Events**

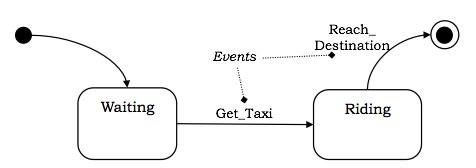
Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.

Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Events that trigger transitions are written alongside the arc of transition in state diagrams.

**Example**

Considering the example shown in the above figure, the transition from Waiting state to Riding state takes place when the person gets a taxi. Likewise, the final state is reached, when he reaches the destination. These two occurrences can be termed as events Get\_Taxi and Reach\_Destination. The following figure shows the events in a state machine.



**External and Internal Events**

External events are those events that pass from a user of the system to the objects within the system. For example, mouse click or key−press by the user are external events.

Internal events are those that pass from one object to another object within a system. For example, stack overflow, a divide error, etc.

**Deferred Events**

Deferred events are those which are not immediately handled by the object in the current state but are lined up in a queue so that they can be handled by the object in some other state at a later time.

**Event Classes**

Event class indicates a group of events with common structure and behavior. As with classes of objects, event classes may also be organized in a hierarchical structure. Event classes may have attributes associated with them, time being an implicit attribute. For example, we can consider the events of departure of a flight of an airline, which we can group into the following class −

Flight\_Departs (Flight\_No, From\_City, To\_City, Route)

Ques 45:

Ans:

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

**Purpose of Statechart Diagrams**

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

* To model the dynamic aspect of a system.
* To model the life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model the states of an object.

## Where to Use Statechart Diagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

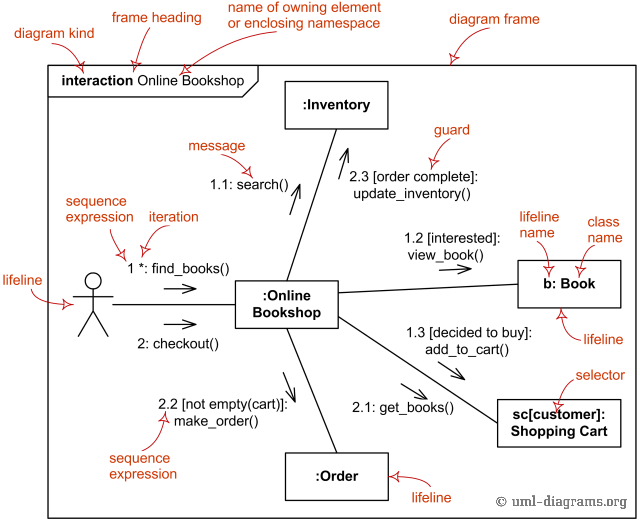
The main usage can be described as −

* To model the object states of a system.
* To model the reactive system. Reactive system consists of reactive objects.
* To identify the events responsible for state changes.
* Forward and reverse engineering.

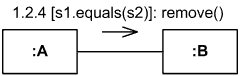
Ques 46:

Ans:

|  |  |  |
| --- | --- | --- |
| **Synchronous call** | Synchronous calls, which are associated with an operation, have a send and a receive message. A message is sent from the source lifeline to the target lifeline. The source lifeline is blocked from other operations until it receives a response from the target lifeline. | A bank teller might send a credit request to the bank manager for approval and must wait for a response before further serving the customer. |
| **Asynchronous call** | Asynchronous calls, which are associated with operations, typically have only a send message, but can also have a reply message. In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages. You can also move the send and receive points individually to delay the time between the send and receive events. You might choose to do this if a response is not time sensitive or order sensitive. | A bank customer could apply for credit but can receive banking information over the phone or request money from an ATM, while waiting to hear about the credit application. |



**Message** in **communication diagram** is shown as a line with [sequence expression](https://www.uml-diagrams.org/communication-diagrams.html#sequence-expression) and **arrow** above the line. The arrow indicates direction of the communication.



Instance of class A sends remove() message to instance of B if s1 is equal to s2

Ques 47:

Ans:

Component modeling is a specialized type of structural modeling concerned with modeling the implementation of a system. Using the UML, you can communicate the implementation of a system using component diagrams. You usually apply component modeling during design activities to determine how implementation activities will build the system; that is, to determine the elements of the system on which implementation activities will focus. Component modeling typically starts after the design of the system is fairly complete, as determined by your system development process.

Deployment modeling is a specialized type of structural modeling concerned with modeling the implementation environment of a system. In contrast to modeling the components of a system, a deployment model shows you the external resources that those components require. You typically apply deployment modeling during design activities to determine how deployment activities will make the system available to its users; that is, to determine the elements of the system on which deployment activities will focus. Like component modeling, deployment modeling usually starts after the design of the system is fairly complete, as determined by your system development process.

Simply put, a Component diagram shows you how different elements of your system have been grouped together (into assemblies / dlls etc) - and the link between these components. A Deployment diagram takes you one step further and describes on which hardware elements do these components reside.

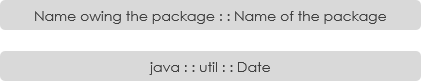
So for example, if "Utility.dll" is a **component** and say it is **deployed** on the Client Machine (hardware). Then, the Component Diagram of this system will show Utility and its link with other components in the system (say.. Customer / SQL Packages). Whereas, the Deployment Diagram will show the hardware configuration - DB Server / Web Server / Client Machine .. and Utility component will be placed into the Client Machine Node.

Ques 48:

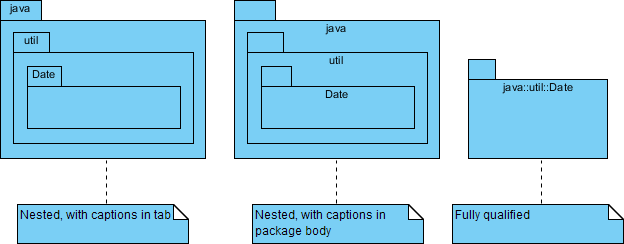
Ans:

Package diagram follows hierarchal structure of nested packages. Atomic module for nested package are usually class diagrams. There are few constraints while using package diagrams, they are as follows.

* Package name should not be the same for a system, however classes inside different packages could have the same name.
* Packages can include whole diagrams, name of components alone or no components at all.
* Fully qualified name of a package has the following syntax.



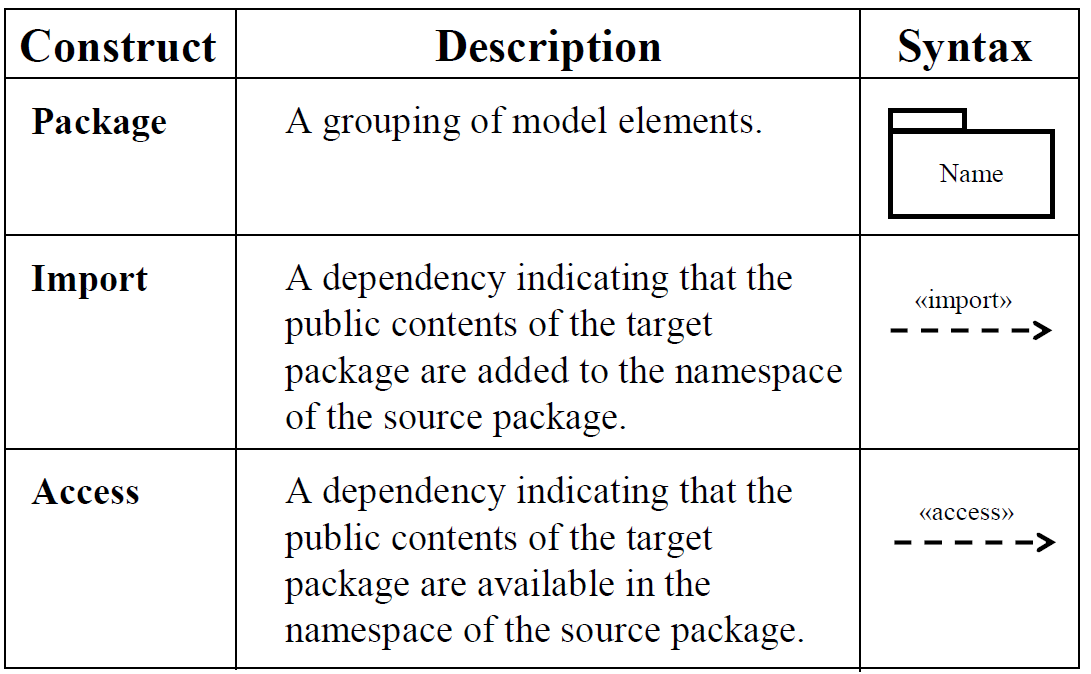
Packages can be represented by the notations with some examples shown below:



**Key Elements of Package Diagram**

Packages are used to organize a large set of model elements:

* Visibility
* Import
* Access



**When to Use Packages?**

To create an overview of a large set of model elements

* To organize a large model
* To group related elements
* To separate namespaces

**Visibility of Packages**

Each contained element has a visibility relative to the containing package.

* A public element is visible to elements outside the package, denoted by ‘+’
* A protected element is visible only to elements within inheriting packages, denoted by ‘#’
* A private element is not visible at all to elements outside the package, denoted by ‘-’
* Same syntax for visibility of attributes and operations in classes

Ques 49:

The deployment diagram is mostly employed by network engineers, system administrators, etc. with the purpose of representing the deployment of software on the hardware system. It envisions the interaction of the software with the hardware to accomplish the execution. The selected hardware must be of good quality so that the software can work more efficiently at a faster rate by producing accurate results in no time.

The software applications are quite complex these days, as they are standalone, distributed, web-based, etc. So, it is very necessary to design efficient software.

Deployment diagrams can be used for the followings:

1. To model the network and hardware topology of a system.
2. To model the distributed networks and systems.
3. Implement forwarding and reverse engineering processes.
4. To model the hardware details for a client/server system.
5. For modeling the embedded system.

## Purpose of Deployment Diagram

The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.

Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.

The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.

Following are the purposes of deployment diagram enlisted below:

1. To envision the hardware topology of the system.
2. To represent the hardware components on which the software components are installed.
3. To describe the processing of nodes at the runtime.

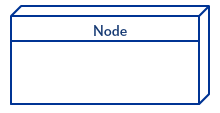
Ques 50:

Ans:

**Deployment Diagram Notations**

In order to draw a deployment diagram, you need to first become familiar with the following deployment diagram notations and deployment diagram elements.

**Nodes**

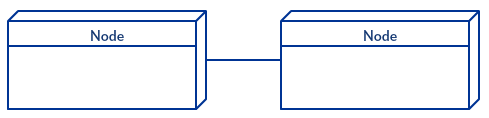


A node, represented as a cube, is a physical entity that executes one or more components, subsystems or executables. A node could be a hardware or software element.

**Artifacts**

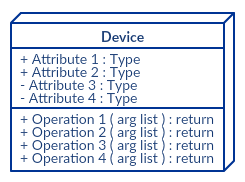


Artifacts are concrete elements that are caused by a development process. Examples of artifacts are libraries, archives, configuration files, executable files etc.

**Communication Association**

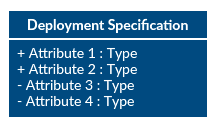
This is represented by a solid line between two nodes. It shows the path of communication between nodes.

**Devices**



A device is a node that is used to represent a physical computational resource in a system. An example of a device is an application server.

**Deployment Specifications**



Deployment specifications is a configuration file, such as a text file or an XML document. It describes how an artifact is deployed on a node.

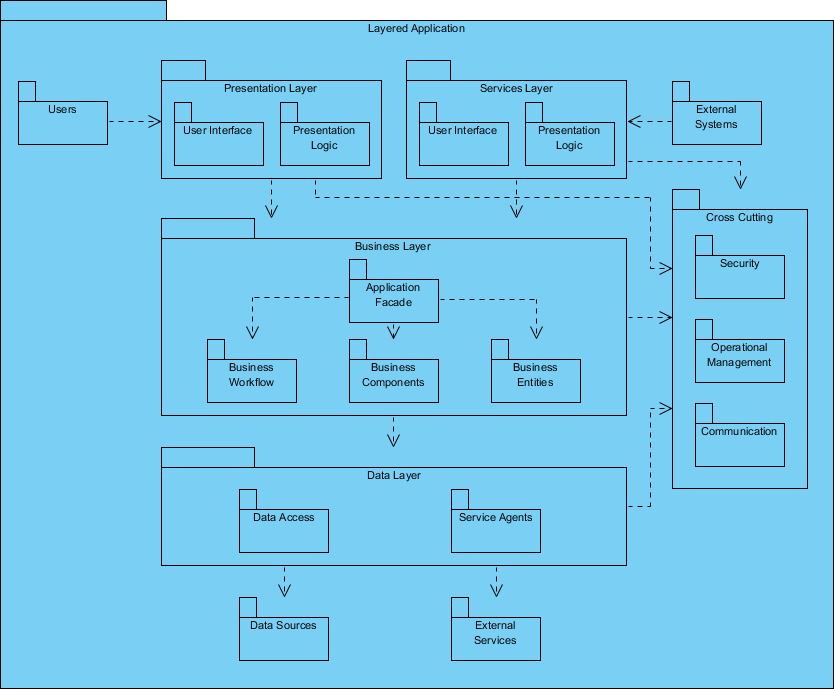
Ques 51:

Ans:

Package diagram, a kind of structural diagram, shows the arrangement and organization of model elements in middle to large scale project. Package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered (aka multi-tiered) application - multi-layered application model.

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

* Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
* A package is a collection of logically related UML elements.
* Packages are depicted as file folders and can be used on any of the UML diagrams.



Ques 52:

Ans :

The purpose of a component diagram is to show the relationship between different components in a system. For the purpose of UML 2.0, the term "component" refers to a module of classes that represent independent systems or subsystems with the ability to interface with the rest of the system.

There exists a whole development approach that revolves around components: component-based development (CBD). In this approach, component diagrams allow the planner to identify the different components so the whole system does what it's supposed to do.

More commonly, in an OO programming approach, the component diagram allows a senior developer to group classes together based on common purpose so that the developer and others can look at a software development project at a high level.

**Benefits of component diagrams**

Though component diagrams may seem complex at first glance, they are invaluable when it comes to building your system. Component diagrams can help your team:

* Imagine the system’s physical structure.
* Pay attention to the system’s components and how they relate.
* Emphasize the service behavior as it relates to the interface.

**How to use component diagrams**

A component diagram in UML gives a bird’s-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a better developer. Component diagrams can describe software systems that are implemented in any programming language or style.

**Where to Use Component Diagrams?**

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to −

* Model the components of a system.
* Model the database schema.
* Model the executables of an application.
* Model the system's source code.

Ques 53:

Ans:

There are two types of nodes in a deployment diagram: device nodes and execution environment nodes. Device nodes are computing resources with processing capabilities and the ability to execute programs. Some examples of device nodes include PCs, laptops, and mobile phones.

An execution environment node, or EEN, is any computer system that resides within a device node. It could be an operating system, a JVM, or another servlet container.

Ques 54:

Ans:

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

We will now look into the practical applications of the activity diagram. From the above discussion, it is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

* Modeling work flow by using activities.
* Modeling business requirements.
* High level understanding of the system's functionalities.
* Investigating business requirements at a later stage.

