ABES Engineering College
Estd. 2000
College Code-032

Behavioural Modeling

Activity and State Chart Diagram

OBJECT ORIENTED SYSTEM DESIGN(OOSD)

BATCH 2023-2024

ACTIVITY DIAGRAMS

- It describe **dynamic aspects of the system.**
- It is used to demonstrate the flow of control within the system rather than the implementation.
- Activity diagram is essentially an advanced version of flow chart that modeling the flow from **one activity to another activity.**
- It put emphasis on the condition of flow and the order in which it occurs.
- It is also termed as an object-oriented flowchart. It encompasses activities composed of a set of actions or operations that are applied to model the behavioural diagram.

WHY USE ACTIVITY DIAGRAM

- An event is created as an activity diagram encompassing a group of nodes associated with edges.
- To model the behavior of activities, they can be attached to any modeling element.
- It can model use cases, classes, interfaces, components, and collaborations.
- It mainly models processes and workflows.
- It does not include the message part, which means message flow is not represented in an activity diagram.

COMPONENTS OF AN ACTIVITY DIAGRAM

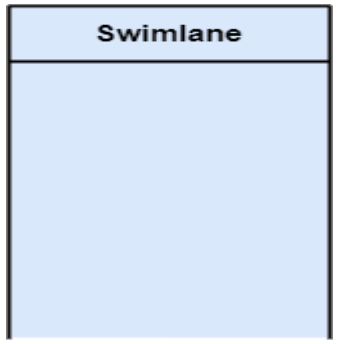
Activities

- The categorization of behavior into one or more actions is termed as an activity.
- An activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes.
- The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity.
- The activities are initiated at the initial node and are terminated at the final node.

COMPONENTS OF AN ACTIVITY DIAGRAM

Activity partition /swimlane

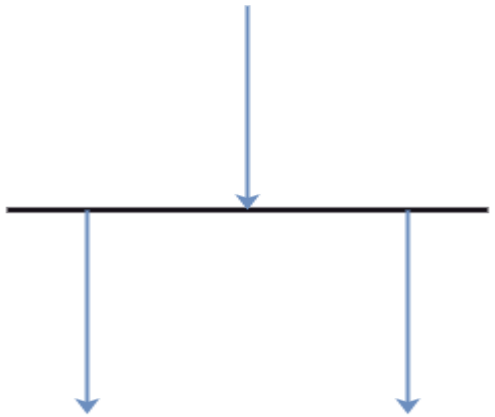
- The swimlane is used to cluster all the related activities in one column or one row.
- It can be either vertical or horizontal.
- It is used to add modularity to the activity diagram.
- It is not necessary to incorporate swimlane in the activity diagram. But it is used to add more transparency to the activity diagram.



COMPONENTS OF AN ACTIVITY DIAGRAM

Forks

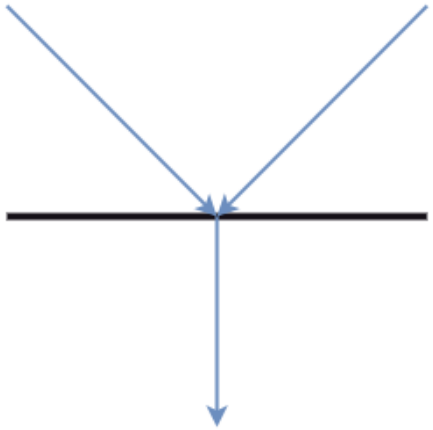
- Forks and join nodes generate the concurrent flow inside the activity.
- A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters.
- Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



COMPONENTS OF AN ACTIVITY DIAGRAM

Join Nodes

- Join nodes are the opposite of fork nodes.
- A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



COMPONENTS OF AN ACTIVITY DIAGRAM

Pins

- It is a small rectangle, which is attached to the action rectangle.
- It clears out all the messy and complicated thing to manage the execution flow of activities.
- It is an object node that precisely represents one input to or output from the action.

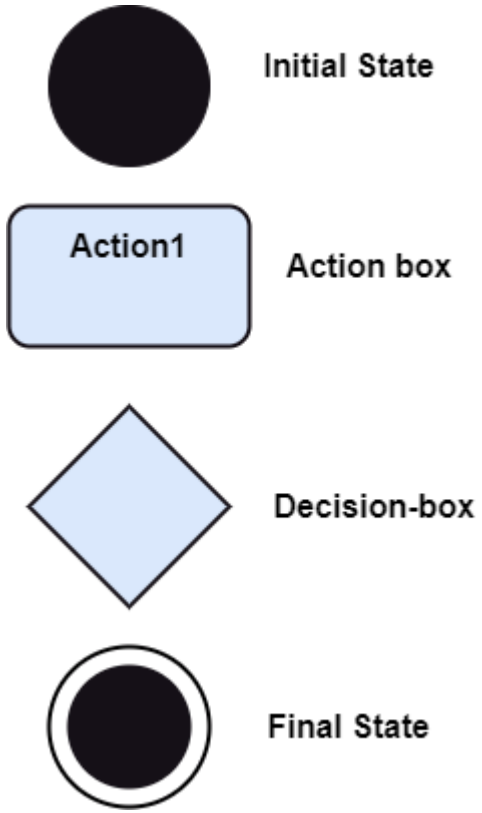
NOTATION OF AN ACTIVITY DIAGRAM

Initial State: It depicts the initial stage or beginning of the set of actions.

Final State: It is the stage where all the control flows and object flows end.

Decision Box: It makes sure that the control flow or object flow will follow only one path.

Action Box: It represents the set of actions that are to be performed.



RULES THAT ARE TO BE FOLLOWED

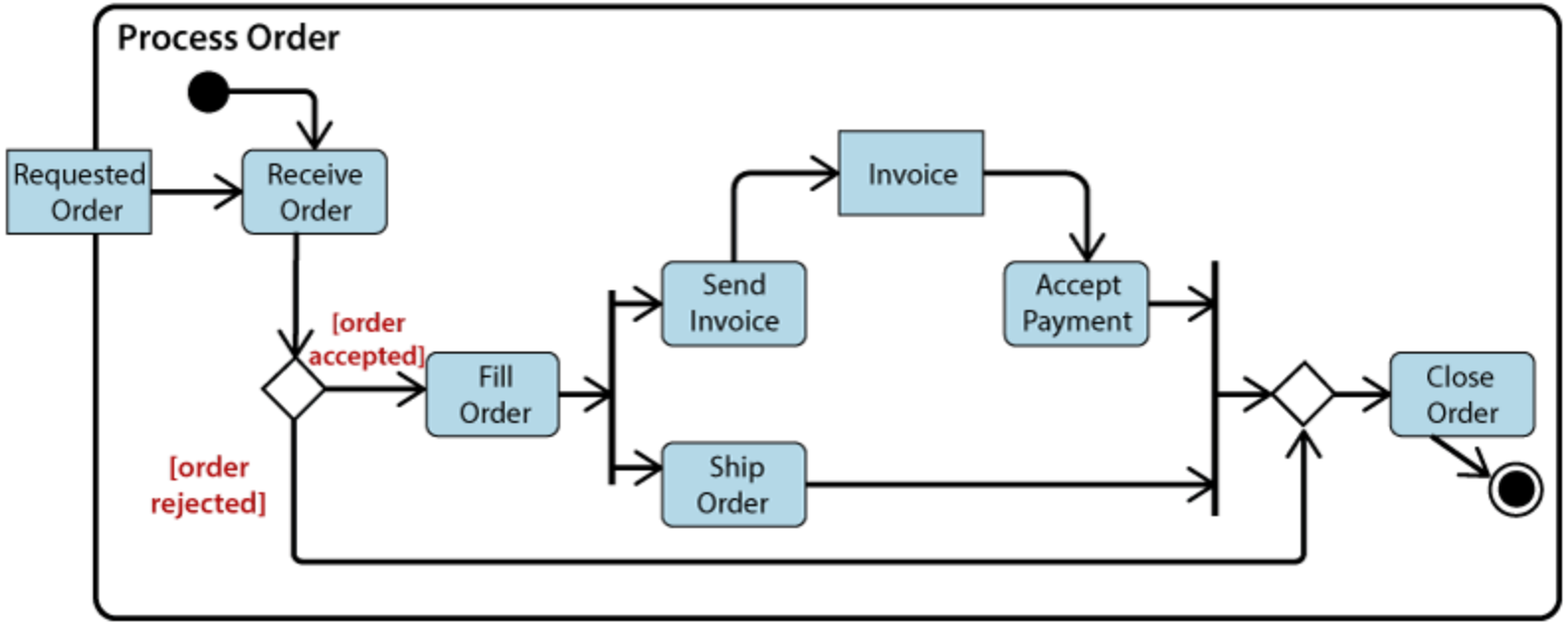
An activity diagram is a flowchart of activities, as it represents the workflow among various activities. They are identical to the flowcharts, but they themself are not exactly the flowchart. In other words, it can be said that an activity diagram is an enhancement of the flowchart, which encompasses several unique skills.

- A meaningful name should be given to each and every activity.
- Identify all of the constraints.
- Acknowledge the activity associations.

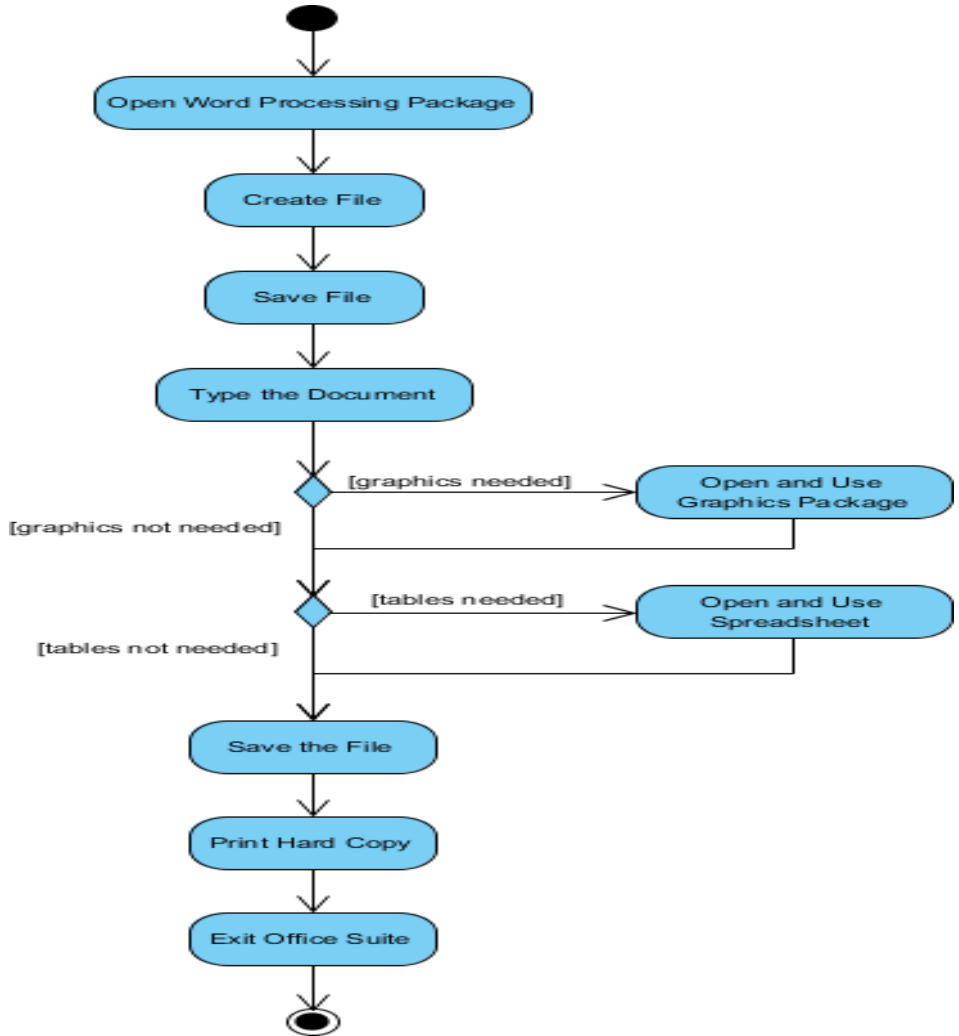
WHEN TO USE ACTIVITY DIAGRAM

- To graphically model the workflow in an easier and understandable way.
- To model the execution flow among several activities.
- To model comprehensive information of a function or an algorithm employed within the system.
- To model the business process and its workflow.
- To envision the dynamic aspect of a system.
- To generate the top-level flowcharts for representing the workflow of an application.
- To represent a high-level view of a distributed or an object-oriented system.

ACTIVITY DIAGRAM - ORDER PROCESSING



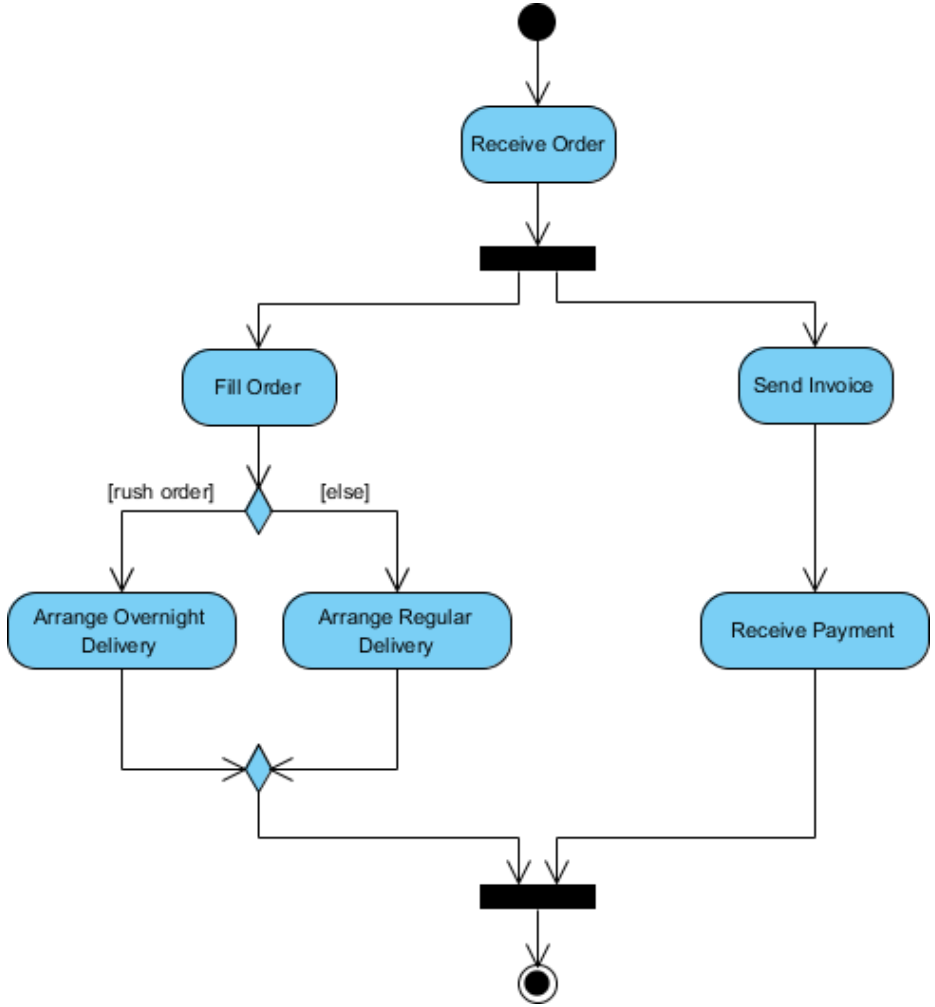
ACTIVITY DIAGRAM - MODELING A WORD PROCESSOR



ACTIVITY DIAGRAM EXAMPLE - PROCESS ORDER

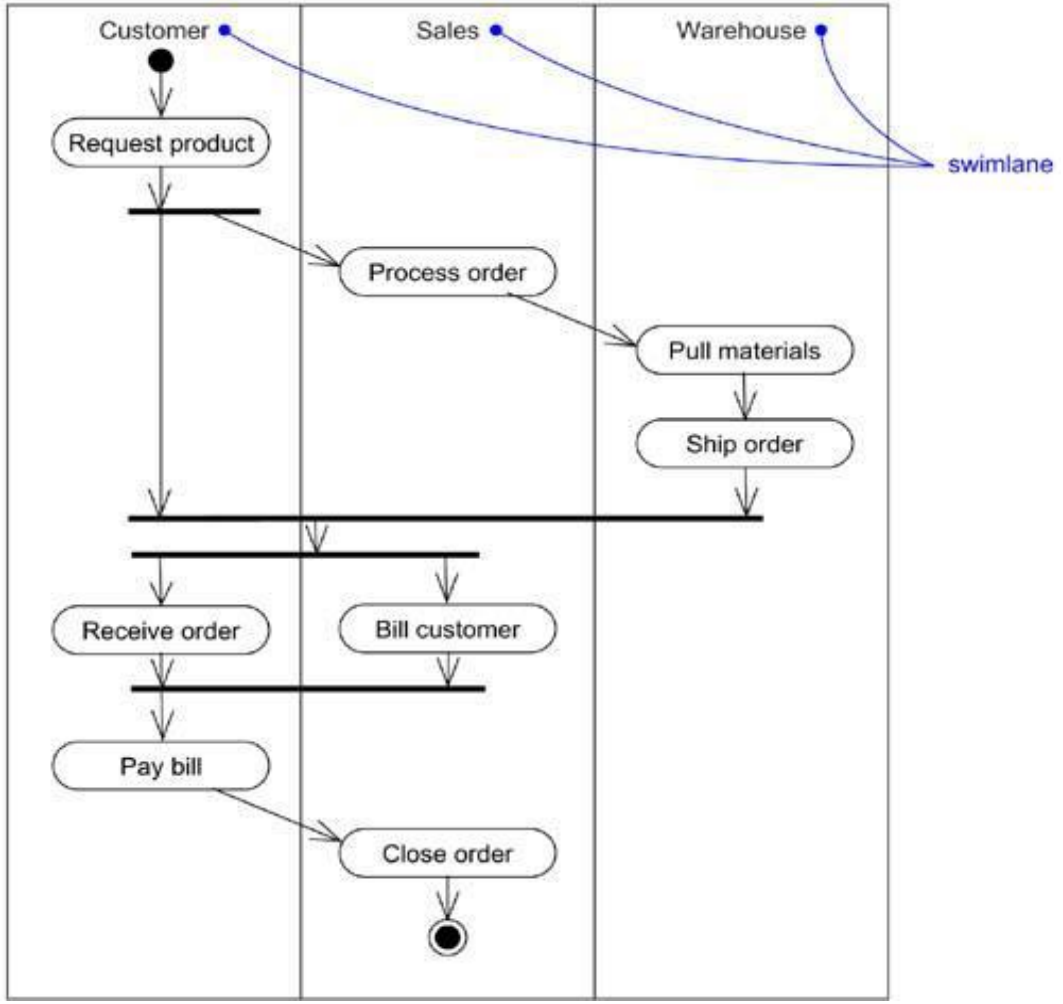
Process Order - Problem Description

- Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- Finally the parallel activities combine to close the order.



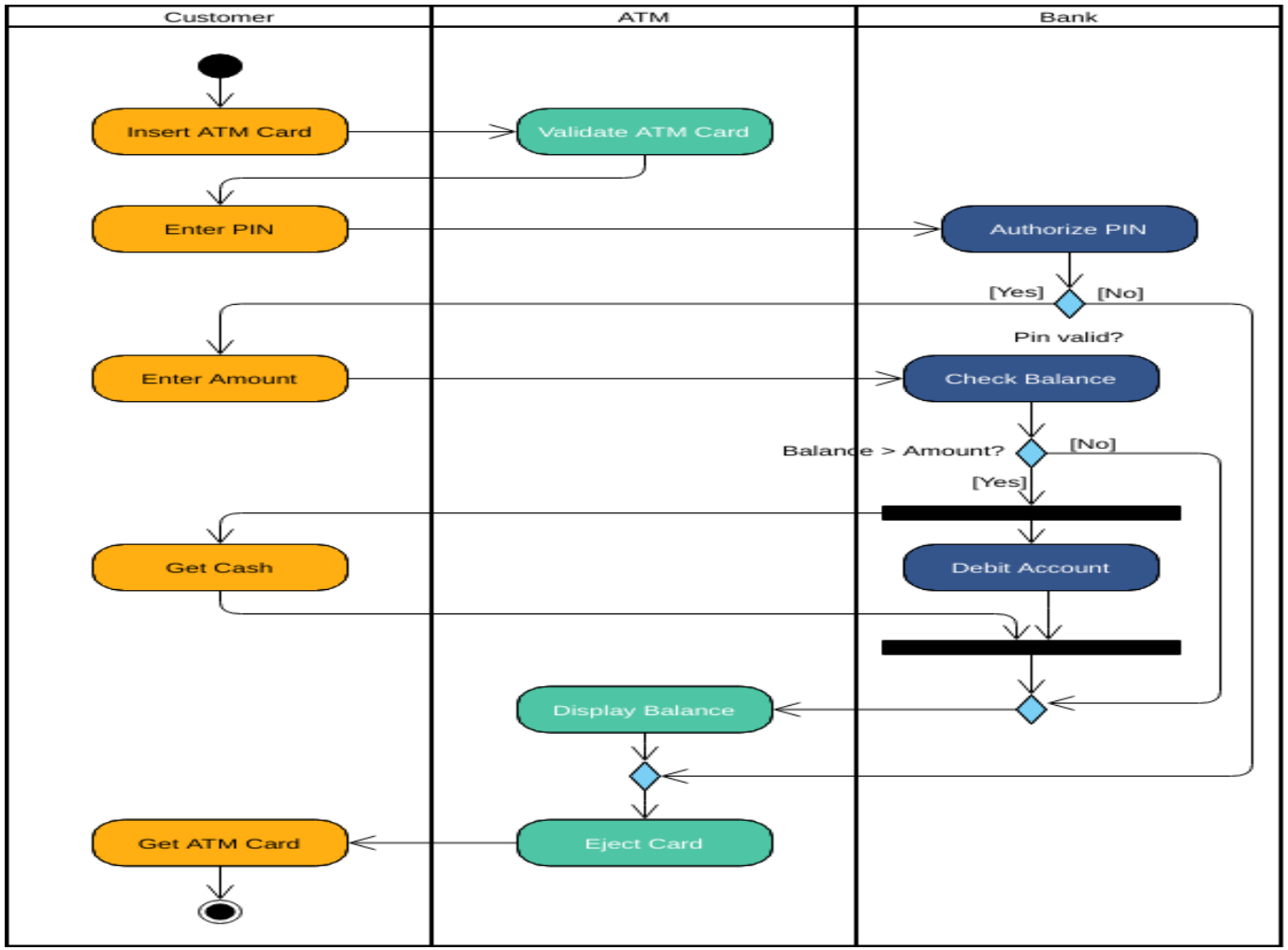
ACTIVITY DIAGRAM - SWIMLANE

- A swimlane is a way to group activities performed by the same actor on an activity diagram or activity diagram or to group activities in a single thread.



ACTIVITY DIAGRAM FOR ATM

- Design **activity** diagram for **ATM** which display all the activities performed during the transaction process of ATM like **insert ATM card, pin, authorize pin, enter amount, get cash, debit account and other activities.**



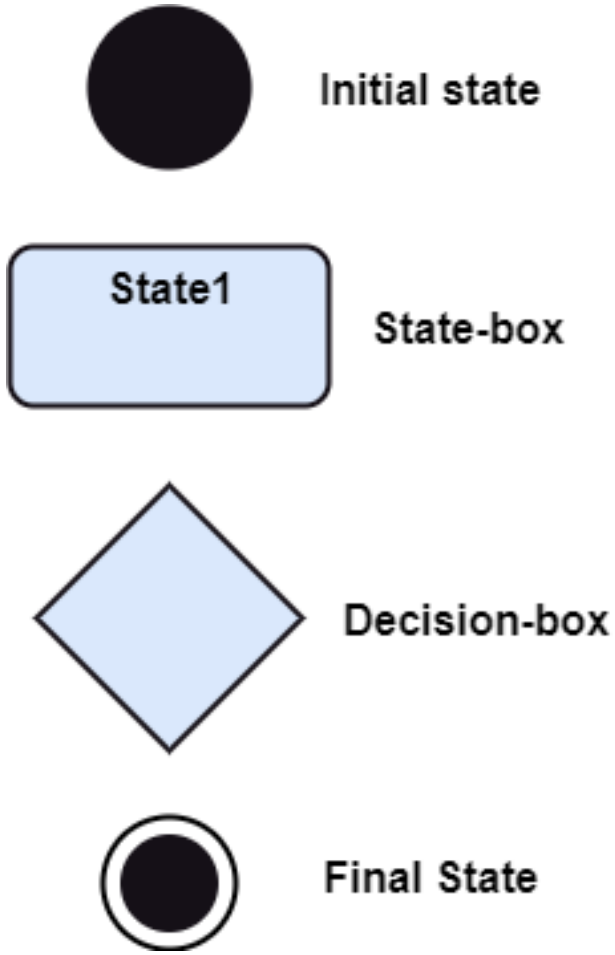
STATE DIAGRAM

- A state diagram, sometimes known as a state machine diagram, is a type of behavioral diagram in the Unified Modeling Language (UML) that **shows transitions between various objects**.
- It captures the software system's behavior.
- It models the behavior of a class, a subsystem, a package, and a complete system.
- It also defines several distinct states of a component within the system. **Each object/component has a specific state.**

WHY STATE MACHINE DIAGRAM?

- Since it records the dynamic view of a system, it portrays the behavior of a software application.
- Each state depicts some useful information about the object.
- It visualizes an object state from its creation to its termination.
- The main purpose is to depict each state of an individual object.

NOTATION OF A STATE MACHINE DIAGRAM



1.Initial state: It defines the initial state (beginning) of a system, and it is represented by a black filled circle.
2.Final state: It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.
3.Decision box: It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.
4.Transition: A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.
5.State box: It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.

TYPES OF STATE

The UML consist of three states:

- **Simple state:** It does not constitute any substructure.

- **Composite state:** It consists of nested states (substates), such that it does not contain more than one initial state and one final state. It can be nested to any level.

- **Submachine state:** The submachine state is semantically identical to the composite state, but it can be reused.

HOW TO DRAW A STATE MACHINE DIAGRAM?

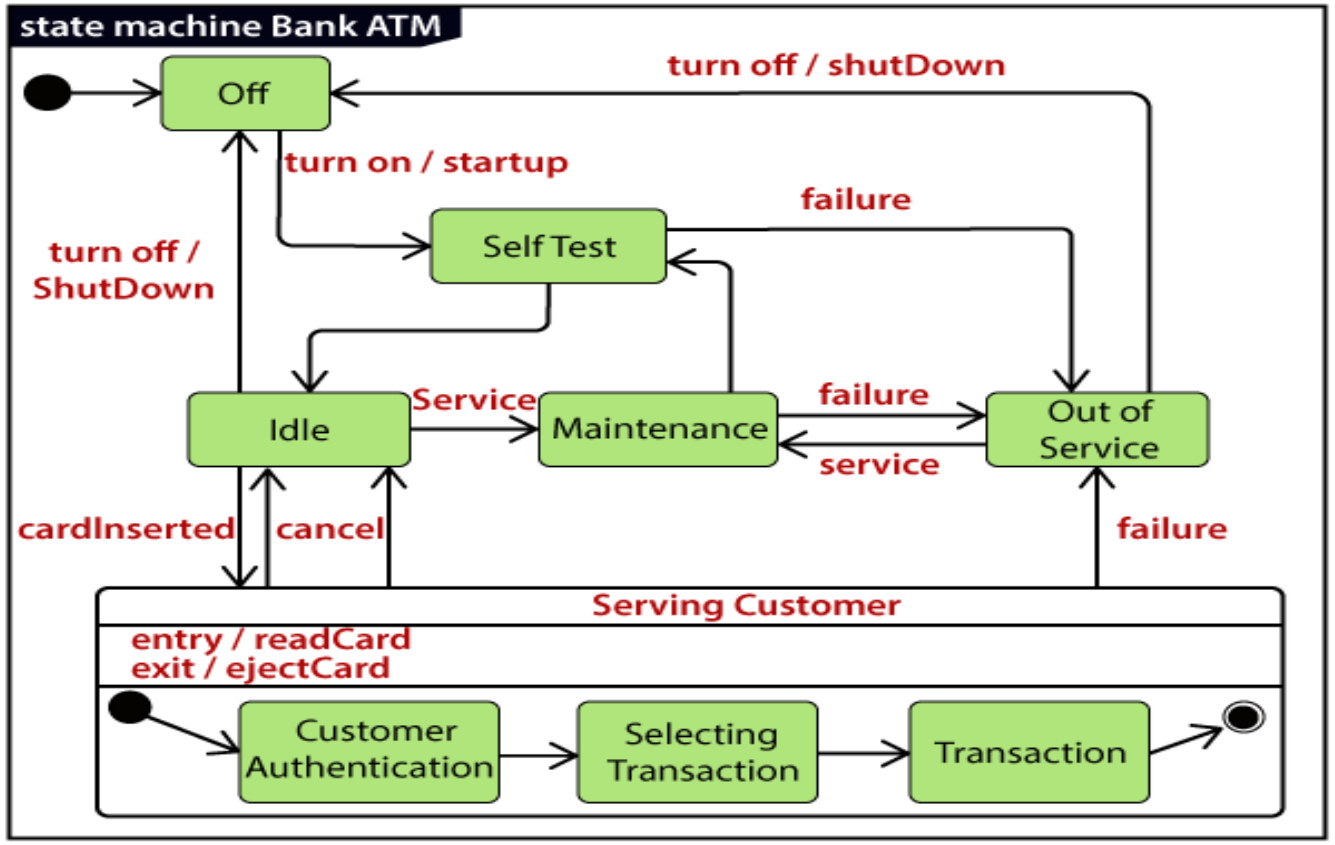
- A unique and understandable name should be assigned to the state transition that describes the behaviour of the system.
- Out of multiple objects, only the essential objects are implemented.
- A proper name should be given to the events and the transitions.

WHEN TO USE?

- For modeling the object states of a system.
- For modeling the reactive system as it consists of reactive objects.
- For pinpointing the events responsible for state transitions.
- For implementing forward and reverse engineering.

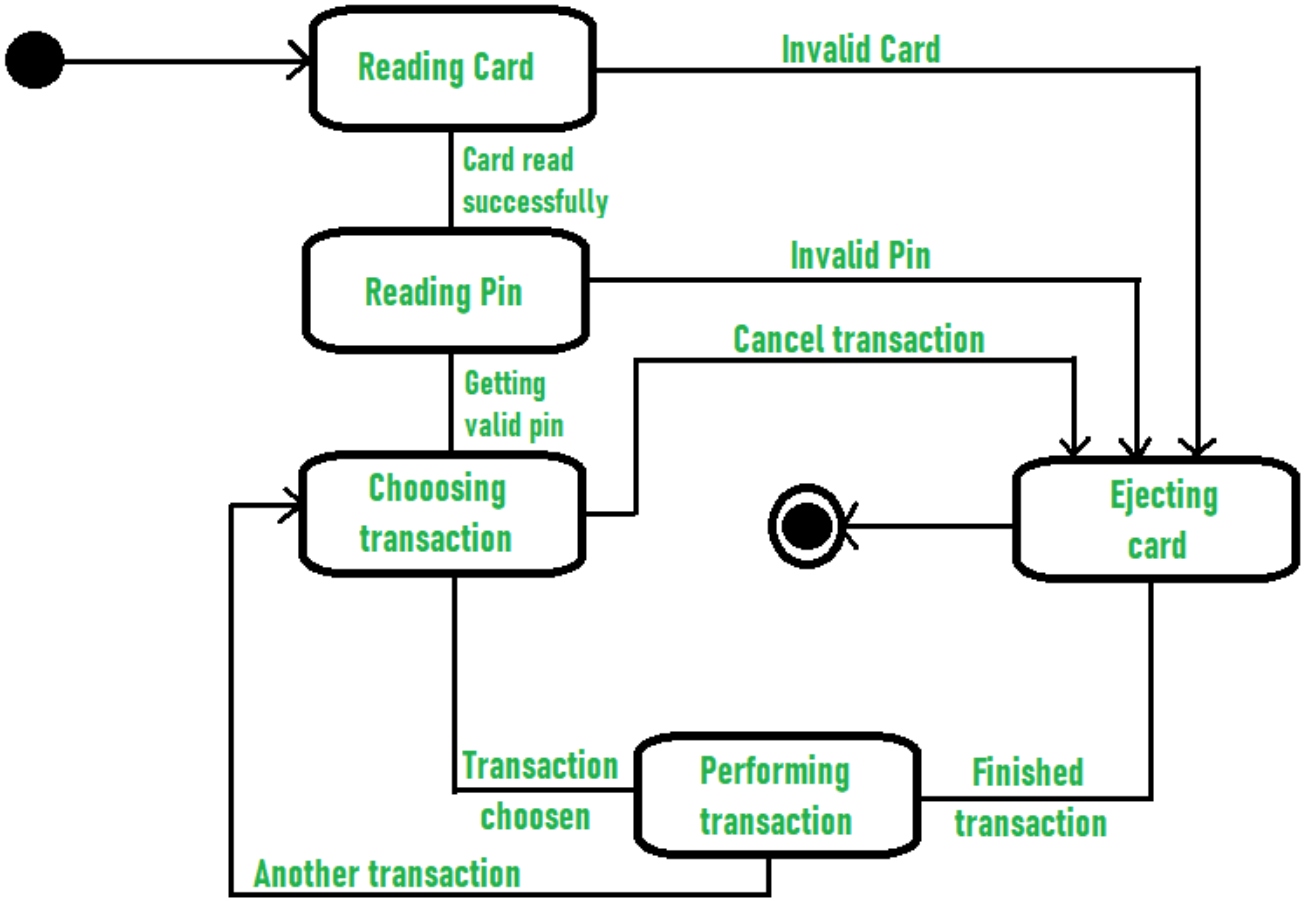
STATE MACHINE DIAGRAM OF ATM

- Draw **state-chart** diagram for **ATM** which describes various states like **reading card, reading pin, choosing transaction, performing transaction and ejecting card**.



STATE MACHINE DIAGRAM OF ATM

- Draw **state-chart** diagram for **ATM**which describes various states like **reading card, reading pin, choosing transaction, performing transaction and ejecting card**.



State Transition Diagram for ATM System

STATE MACHINE VS. FLOWCHART

State Machine	Flowchart
It portrays several states of a system.	It demonstrates the execution flow of a program.
It encompasses the concept of WAIT, i.e., wait for an event or an action.	It does not constitute the concept of WAIT.
It is for real-world modeling systems.	It envisions the branching sequence of a system.
It is a modeling diagram.	It is a data flow diagram (DFD)
It is concerned with several states of a system.	It focuses on control flow and path.

PROCESSES AND THREADS MODEL

- 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.

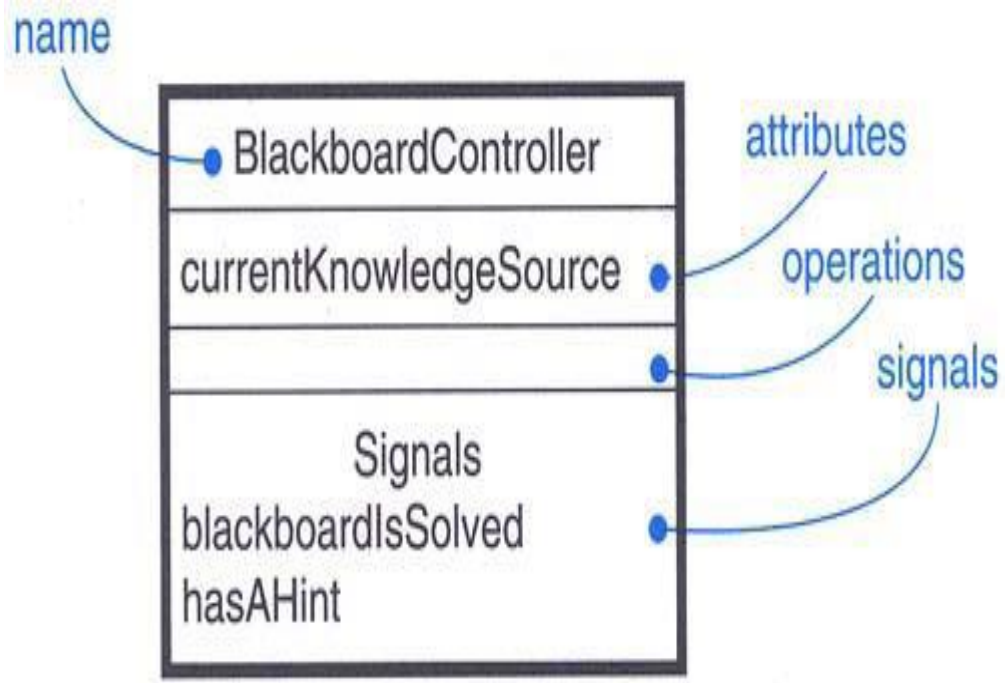


Figure 1: Active Class

- **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
- Figure 2: shows Communication
- *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

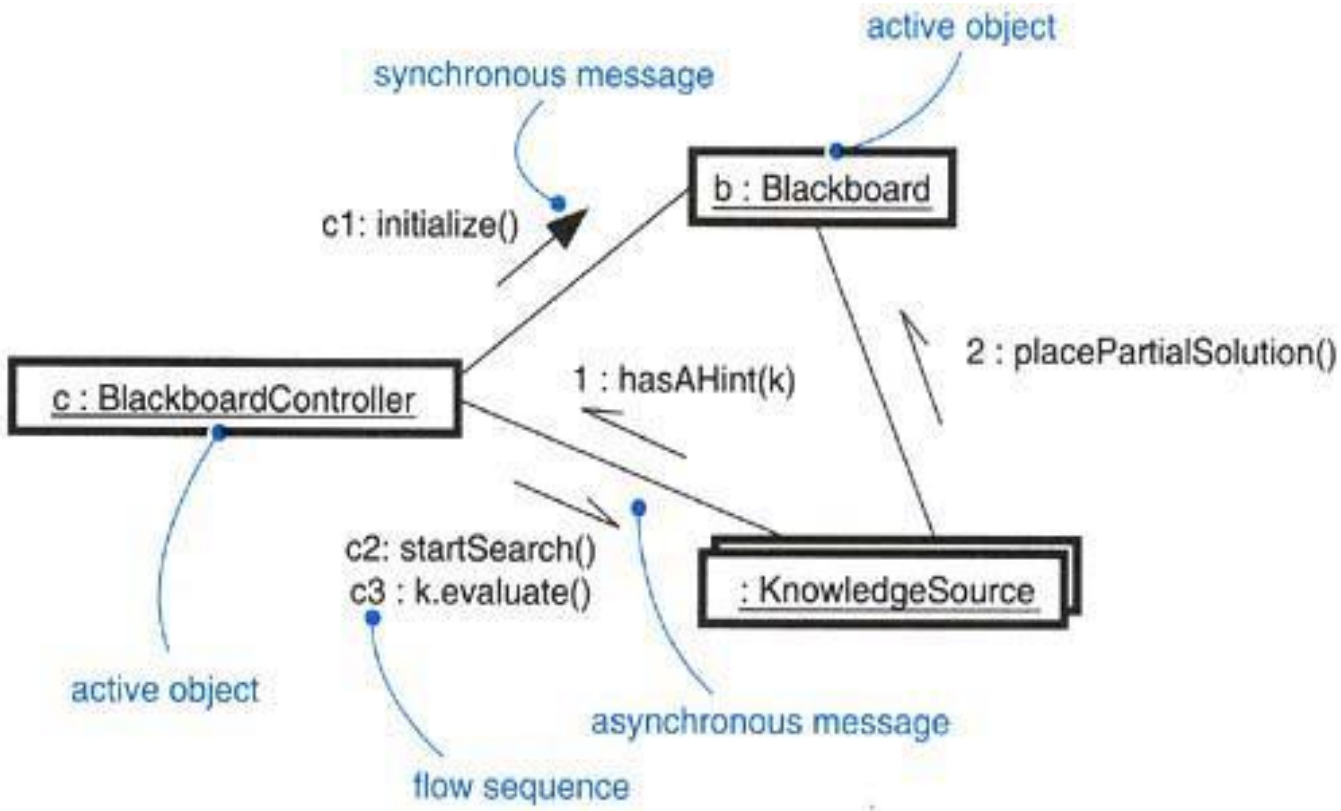


Figure 2: Communication

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
- Figure 3 Synchronization
- *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

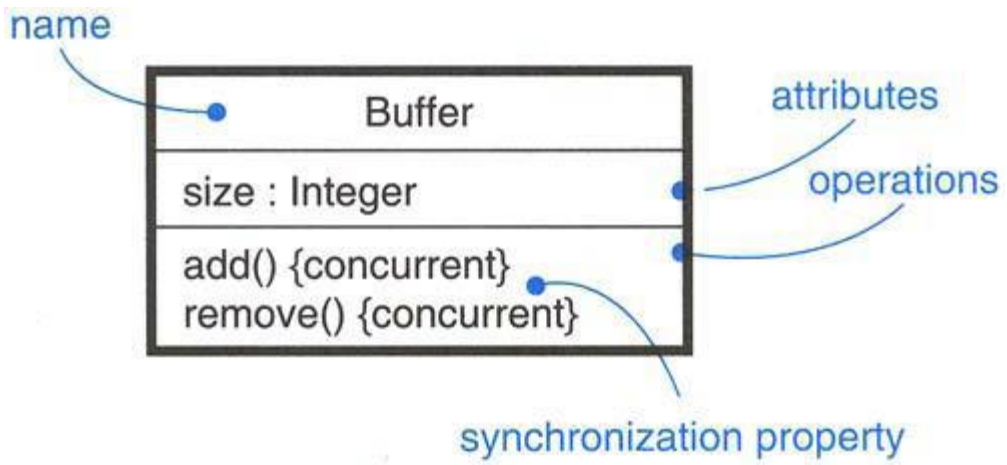
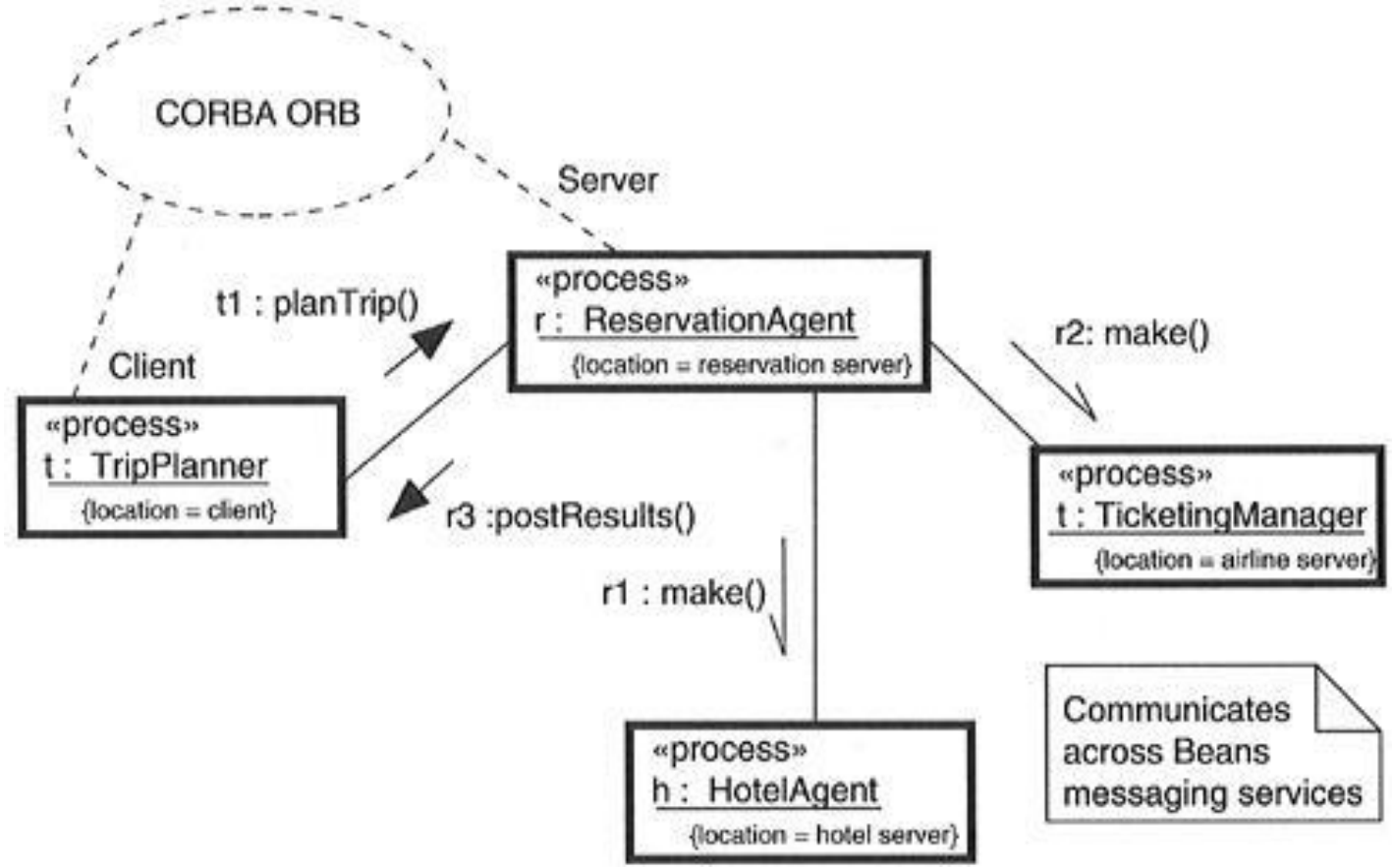


Figure 3: Synchronization

Modeling Interprocess Communication

To model interprocess communication,

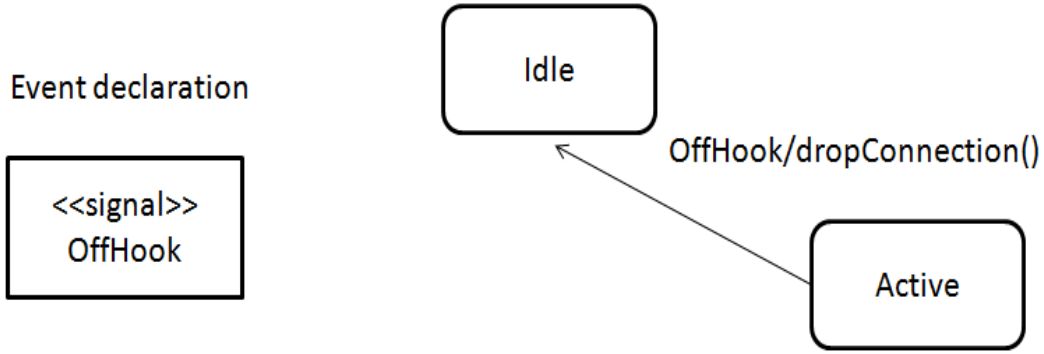
- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
- Informally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


CORBA ORB
Server
t1 : planTrip()
«process»
r : ReservationAgent
{location = reservation server}
Client
«process»
t : TripPlanner
{location = client}
r3 :postResults()
r2: make()
«process»
t : TicketingManager
{location = airline server}
r1 : make()
«process»
h : HotelAgent
{location = hotel server}
Communicates
across Beans
messaging services

EVENTS AND SIGNALS MODEL

- In state machines (sequence of states), we use events to model the occurrence of a stimulus that can trigger an object to move from one state to another state.
- Events may include signals, calls, the passage of time or a change in state.
- In UML, each thing that happens is modeled as an event.
- An event is the specification of a significant occurrence that has a location in time and space.
- A signal, passing of time and change in state are asynchronous events. Calls are generally synchronous events, representing invocation of an operation.

- UML allows us to represent events graphically as shown below. Signals may be represented as stereotyped classes and other events are represented as messages associated with transitions which cause an object to move from one state to another.

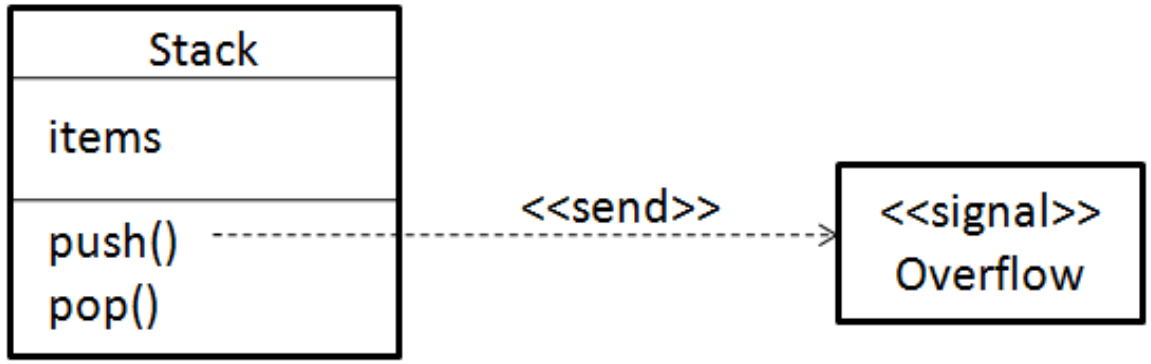


TYPES OF EVENTS

- Events may be external or internal.
- Events passed between the system and its actors are external events. For example, in an ATM system, pushing a button or inserting a card are external events.
- Internal events are those that are passed among objects living inside the system. For example, a overflow exception generated by an object is an internal event.
- In UML, we can model four kinds of events namely: **signals, calls, passing of time and change in state**.

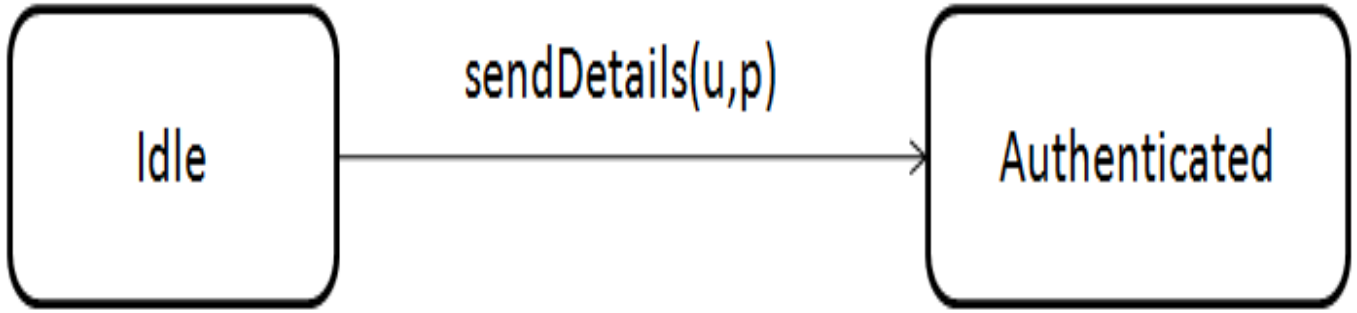
SIGNALS

- A signal is a named object that is sent asynchronously by one object and then received by another. Exceptions are the famous examples for signals. A signal may be sent as the action of a state in a state machine or as a message in an interaction. The execution of an operation can also send signals.
- In UML, we model the relationship between an operation and the events using a dependency stereotyped with "send", which indicates that an operation sends a particular signal.



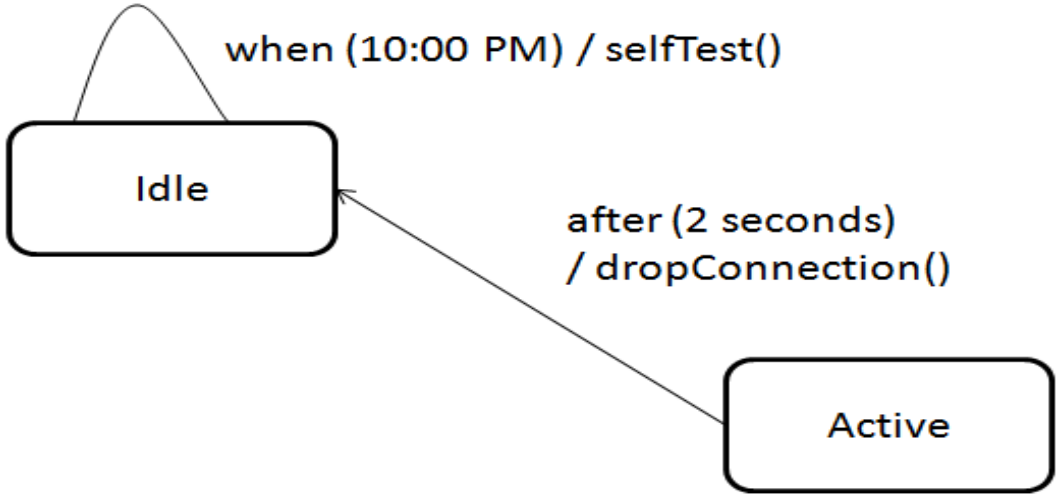
CALL EVENTS

- A call event represents the dispatch of an operation from one object to another. A call event may trigger a state change in a state machine. A call event, in general, is synchronous.
- This means that the sender object must wait until it gets an acknowledgment from the receiver object which receives the call event. For example, consider the states of a customer in an ATM application:



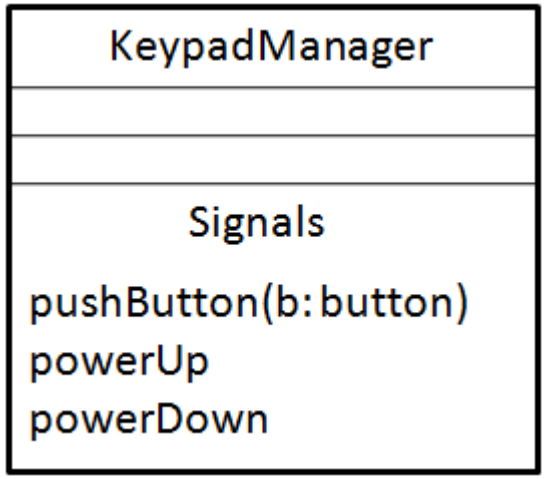
TIME AND CHANGE EVENTS

- A time event represents the passage of time. In UML, we model the time event using the "after" keyword followed by an expression that evaluates a period of time.
- A change event represents an event that represents a change in state or the satisfaction of some condition. In UML, change event is modeled using the keyword "when" followed by some Boolean expression.



SENDING AND RECEIVING EVENTS

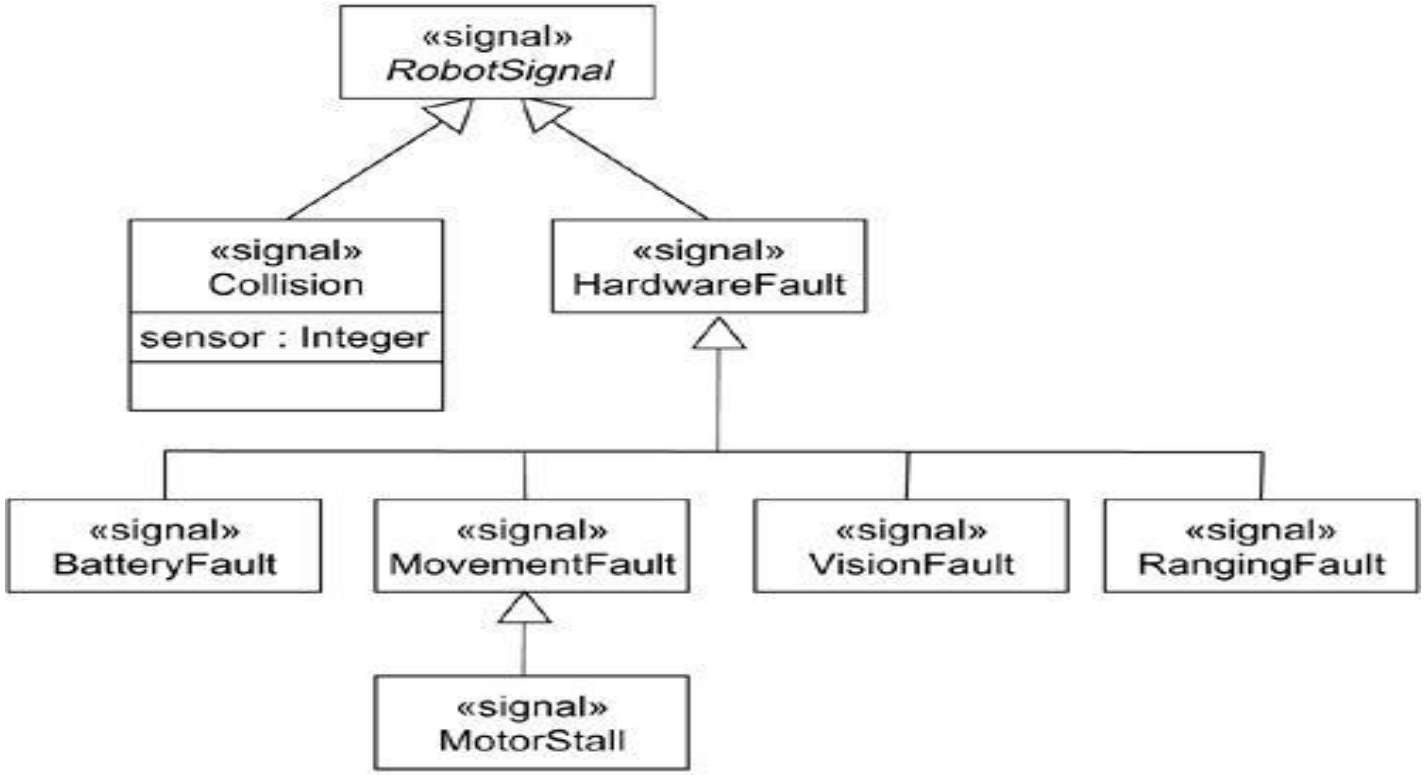
- Any instance of a class can receive a call event or signal. If this is a synchronous call event, the sender is in locked state with receiver. If this is a signal, then the sender is free to carry its operations without any concern on the receiver.
- In UML, call events are modeled as operations on the class of an object and signals that an object can receive are stored in an extra component in the class as shown below:



MODELING A FAMILY OF SIGNALS

- Consider all the signals to which a set of objects can respond.
- Arrange these signals in a hierarchy using generalization-specialization relationship.
- Look out for polymorphism in the state machine of the active objects. When polymorphism is found, adjust the hierarchy by introducing intermediate abstract signals.

MODELING A FAMILY OF SIGNALS



UML TIMING DIAGRAM

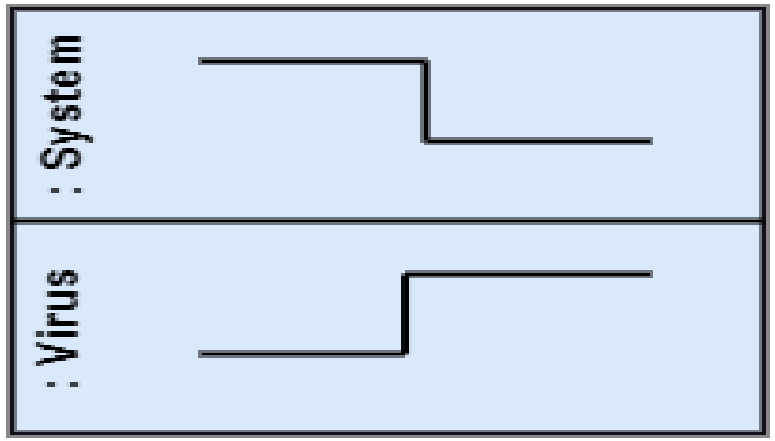
- It emphasizes at that particular time when the message has been sent among objects.
- It explains the time processing of an object in detail.
- It is employed with distributed and embedded systems.
- It also explains how an object undergoes changes in its form throughout its lifeline.
- As the lifelines are named on the left side of an edge, the timing diagrams are read from left to right.
- It depicts a graphical representation of states of a lifeline per unit time.
- In UML, the timing diagram has come up with several notations to simplify the transition state among two lifelines per unit time.

BASIC CONCEPTS OF A TIMING DIAGRAM

In UML, the timing diagram constitutes several major elements, which are as follows:

Lifeline

- As the name suggests, the lifeline portrays an individual element in the interaction.
- It represents a single entity, which is a part of the interaction. It is represented by the classifier's name that it depicts. A lifeline can be placed within a "swimlane" or a diagram frame.

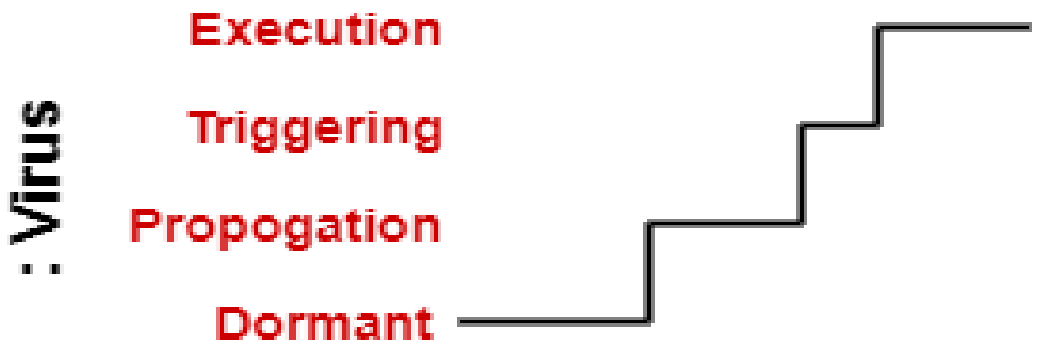


Lifelines representing instances of a System and Virus

BASIC CONCEPTS OF A TIMING DIAGRAM (CONTD..)

State or Condition Timeline

- The timing diagram represents the state of a classifier or attributes that are participating, or some testable conditions, which is a discrete value of the classifier.
- In UML, the state or condition is continuous. It is mainly used to show the temperature and density where the entities endure a continuous state change.

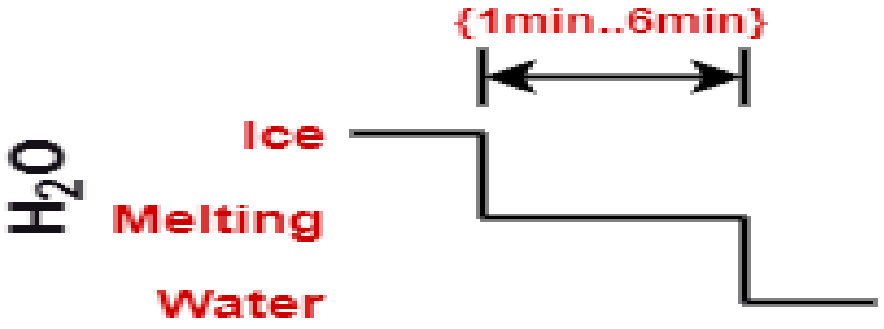


Timeline showing the change in the state of virus between dormant, Propagation, Triggering, Execution

BASIC CONCEPTS OF A TIMING DIAGRAM (CONTD..)

Duration Constraint

- The duration constraint is a constraint of an interval, which refers to duration interval. It is used to determine if the constraint is satisfied for a duration or not. The duration constraint semantics inherits from the constraints.
- The negative trace defines the violated constraints, which means the system is failed. A graphical association between duration interval and the construct, which it constrains, may represent a duration constraint.

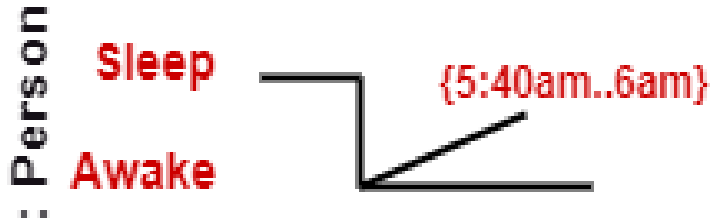


Ice should melt into the water in 1 to 6 mins.

BASIC CONCEPTS OF A TIMING DIAGRAM (CONTD..)

Time Constraint

- It is an interval constraint, which refers to the time interval. Since it is a time expression, it depicts if the constraint is satisfied or not. The constraints dispense its time constraints semantics.
- The negative trace defines the violated constraints, which means the system is failed. The time constraint is represented by a graphical association between the time interval and the construct which it constrains.
- The graphical association is mainly represented by a small line in between a time interval and an occurrence specification.

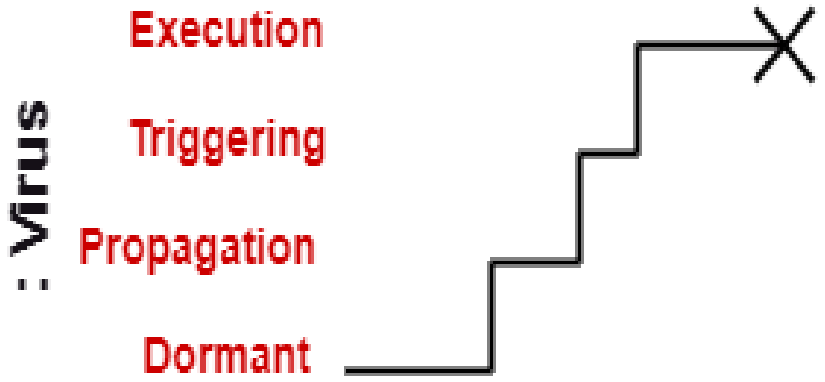


A person should wakeup in between 5:40 am, and 6 am

BASIC CONCEPTS OF A TIMING DIAGRAM (CONTD..)

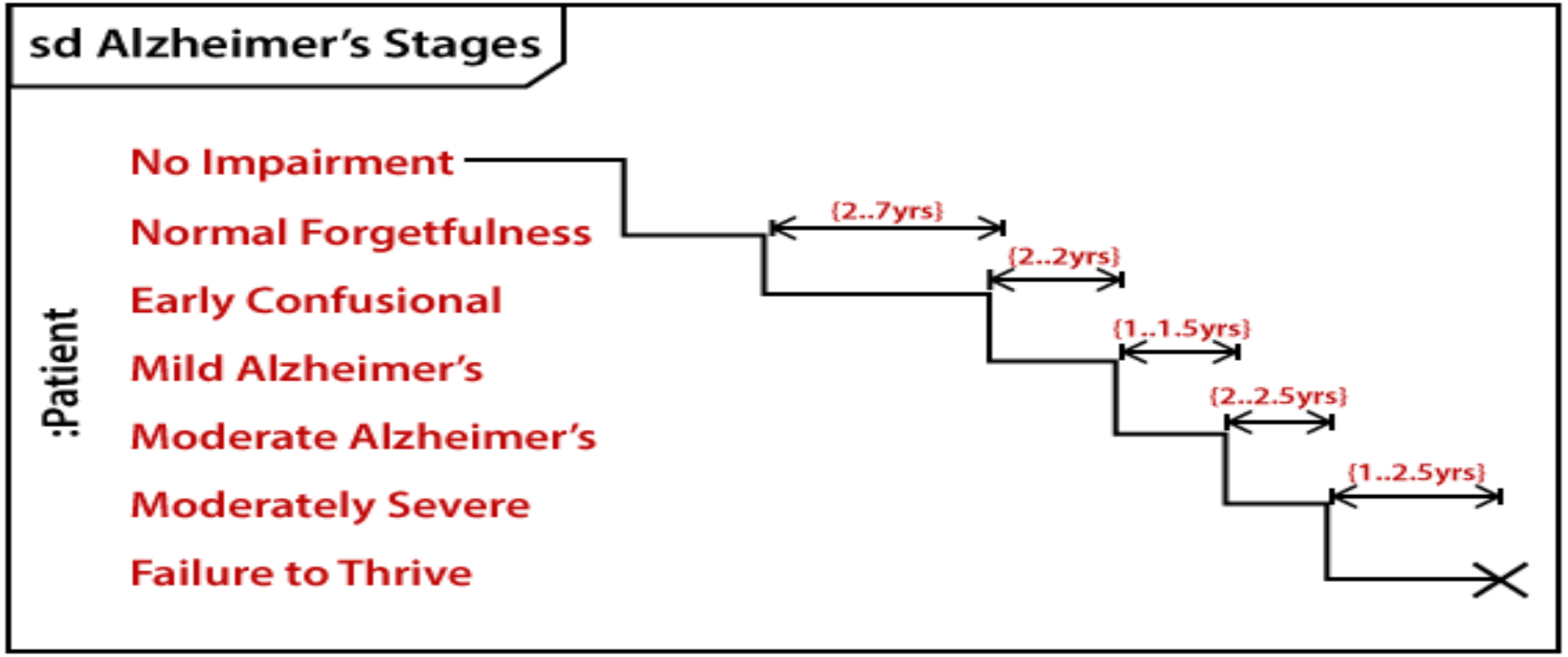
Destruction Occurrence

The destruction occurrence refers to the occurrence of a message that represents the destruction of an instance is defined by a lifeline. It may subsequently destruct other objects owned by the composition of this object, such that nothing occurs after the destruction event on a given lifeline. It is represented by a cross at the end of a timeline.



Virus lifeline is terminated

EXAMPLE OF A TIMING DIAGRAM



Benefits of Timing Diagram

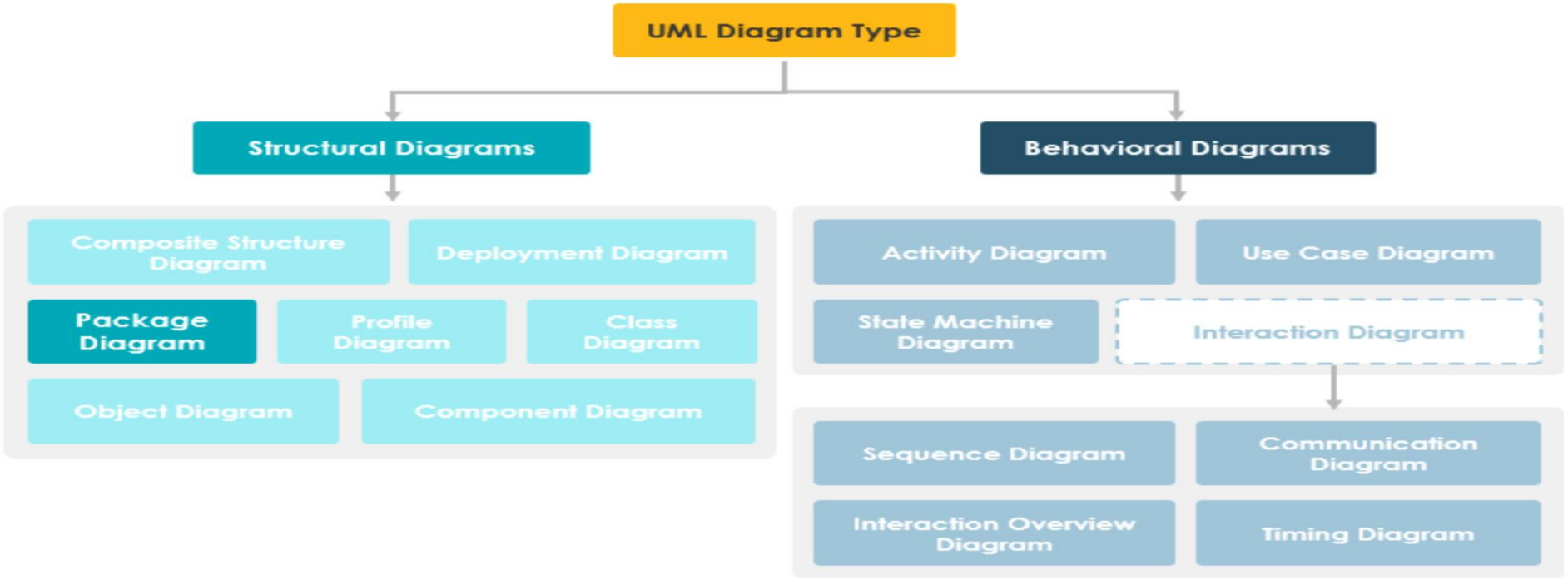
- It depicts the state of an object at a particular point in time.
- It implements forward and reverses engineering.
- It keeps an eye on every single change that happens within the system.

Drawbacks of Timing Diagram

- It is hard to maintain and understand.

PACKAGE DIAGRAM

- 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.



PURPOSE OF PACKAGE DIAGRAMS

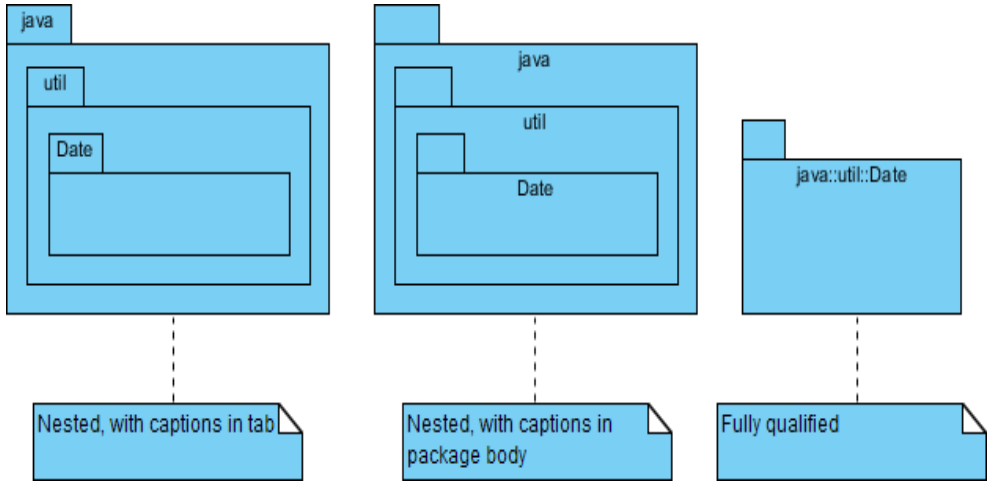
- Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.
- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

BASIC CONCEPTS OF PACKAGE DIAGRAM

- 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.

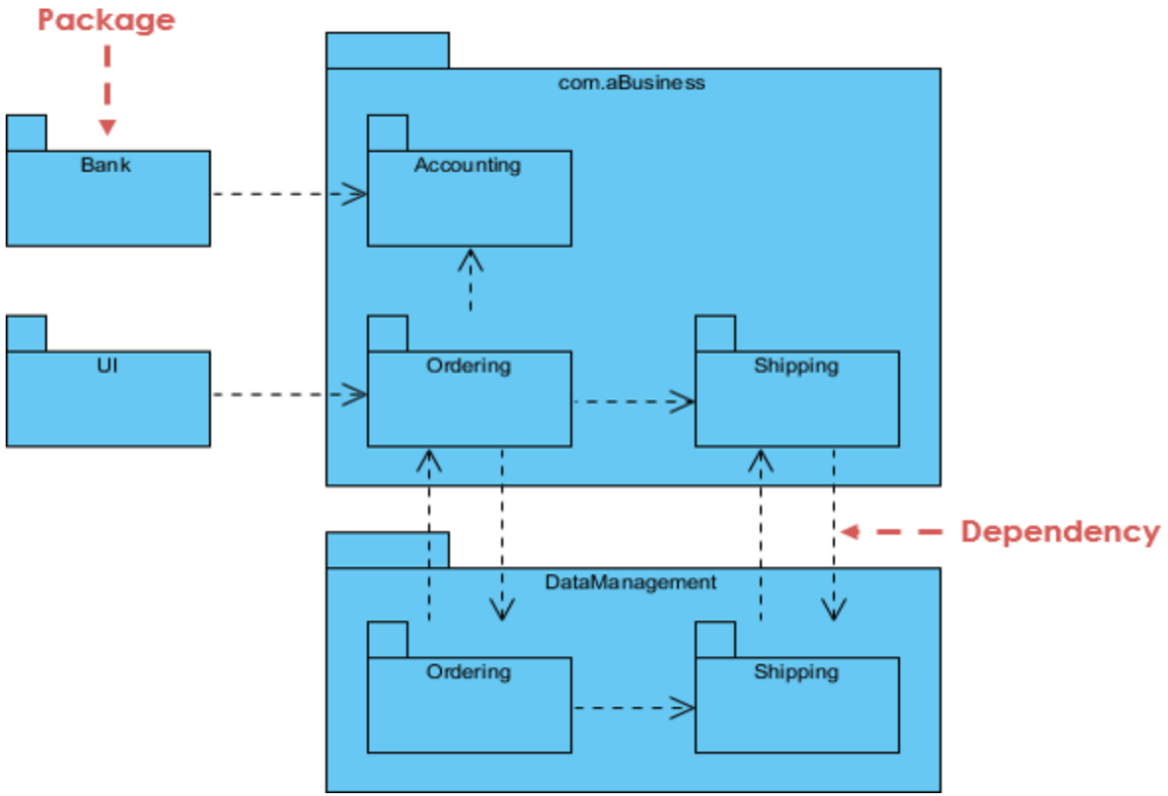
Name owing the package : : Name of the package

java : : util : : Date



PACKAGE DIAGRAM AT A GLANCE

- Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.
- The diagram below is a business model in which the classes are grouped into packages:
- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.


Package
com.aBusiness
Bank
Accounting
UI
Ordering
Shipping
Dependency
DataManagement
Ordering
Shipping

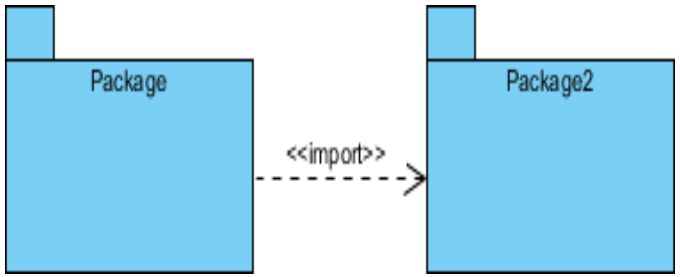
PACKAGE DIAGRAM - DEPENDENCY NOTATION

There are two sub-types involved in dependency. They are <<import>> & <<access>>.

Though there are two stereotypes users can use their own stereotype to represent the type of dependency between two packages.

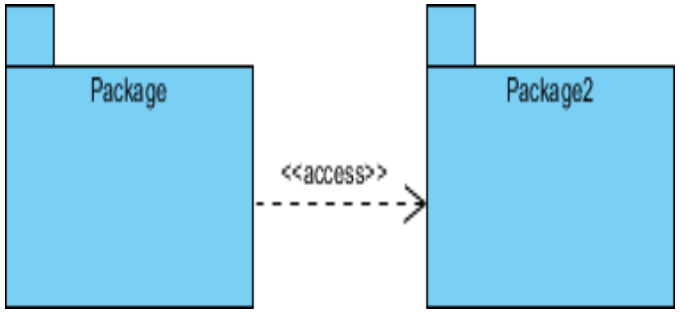
Package Diagram Example - Import

- **<<import>>** - one package imports the functionality of other package

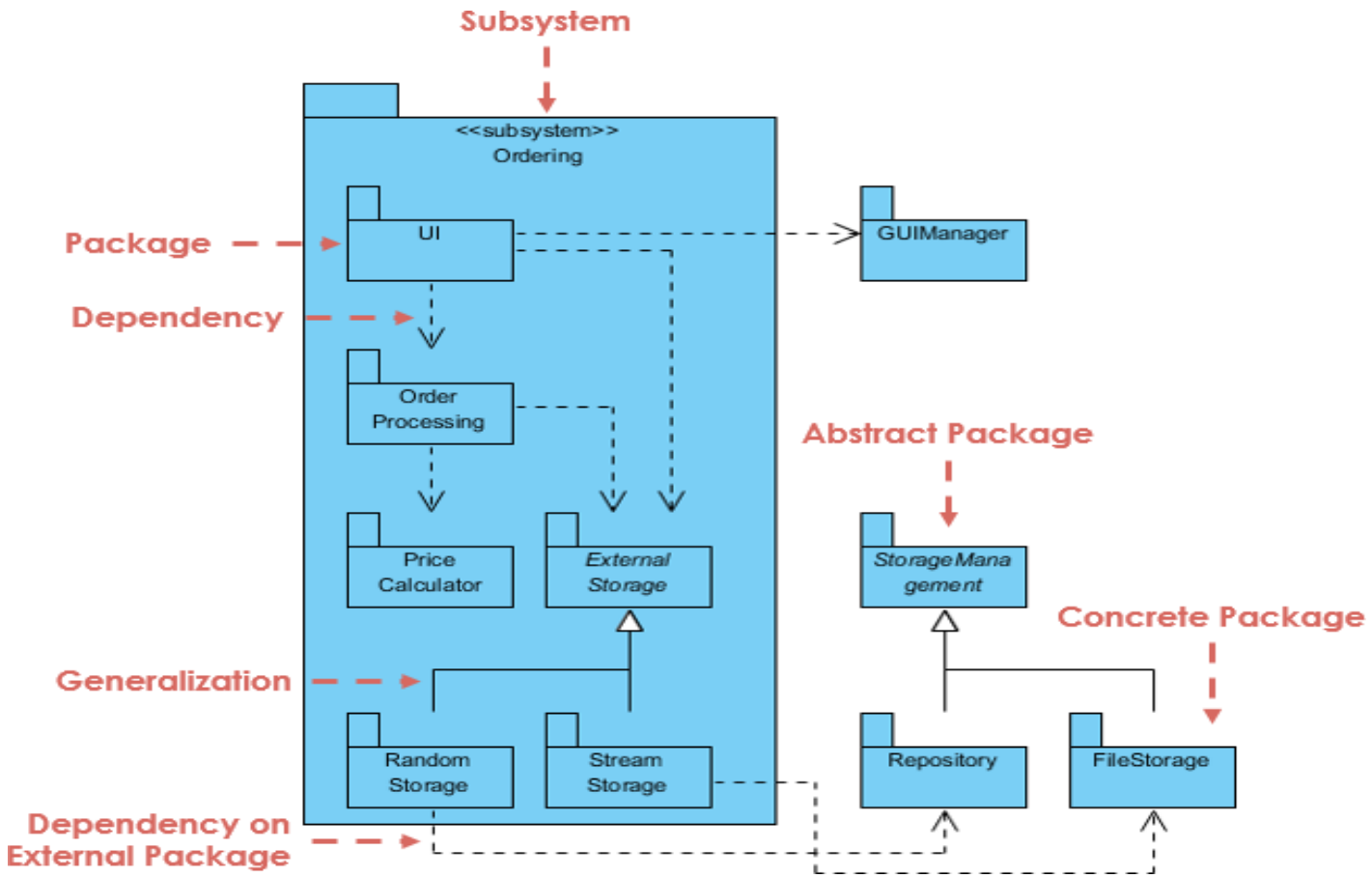


Package Diagram Example - Access

- **<<access>>** - one package requires help from functions of other package.



PACKAGE DIAGRAM EXAMPLE - ORDER SUBSYSTEM



ABES
Estd. 2000
ABES
Engineering College
College Code-032

OBJECT ORIENTED SYSTEM DESIGN(OOSD)

SESSION 2023-24

INTERACTION DIAGRAMS

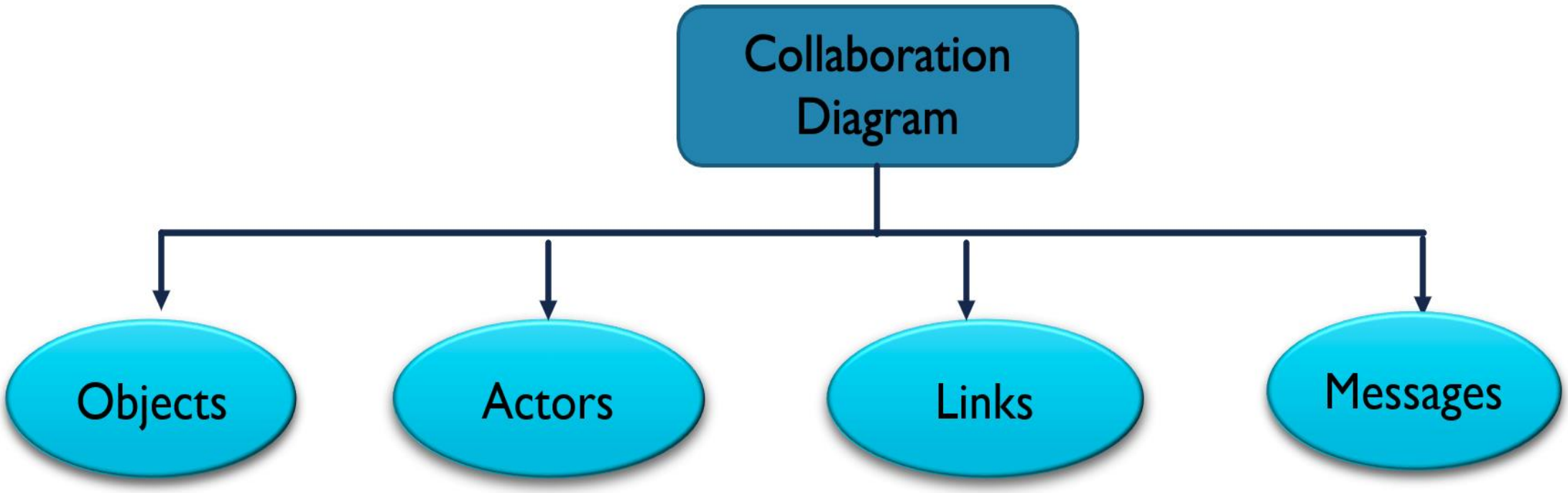
- When a user invokes one of the functions supported by a system, the required behaviour is realised through the interaction of several objects in the system.
- Interaction diagrams are models that describe how groups of objects interact among themselves through message passing to realise some behaviour.
- Sometimes, more than one interaction diagrams may be necessary to capture the behaviour.
- There are two kinds of interaction diagrams—
  - sequence diagrams and
  - collaboration diagrams.

  These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other.
- However, they are both useful. These two actually portray different perspectives of behaviour of a system and different types of inferences can be drawn from them.

COLLABORATION DIAGRAM

- The collaboration diagram is used to show the relationship between the objects in a system.
- It depicts the architecture of the object residing in the system as it is based on object-oriented programming.
- Multiple objects present in the system are connected to each other.
- The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.
- They tend to be better suited to depicting simpler interactions of smaller numbers of objects.
- It is difficult to show additional descriptive information such as timing, or other unstructured information that can be easily added to the notes in a sequence diagram.

COMPONENTS OF COLLABORATION DIAGRAM



OBJECTS

- Each object is also called a collaborator.
- An object is represented by an object symbol showing the name of the object and its class underlined, separated by a colon:

  Object_name : class_name

- Each object in the collaboration is named and has its class specified.
- The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.
- In the collaboration diagram, firstly, the object is created, and then its class is specified.

ACTORS

Normally an actor instance occurs in the collaboration diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

- Each Actor is named and has a role
- One actor will be the initiator of the use case

LINKS

Links connect objects and actors and are instances of associations and each link corresponds to an association in the class diagram

Links are defined as follows:

- A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects.
- An object interacts with, or navigates to, other objects through its links to these objects.
- A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.
- Message flows are attached to links, see Messages.

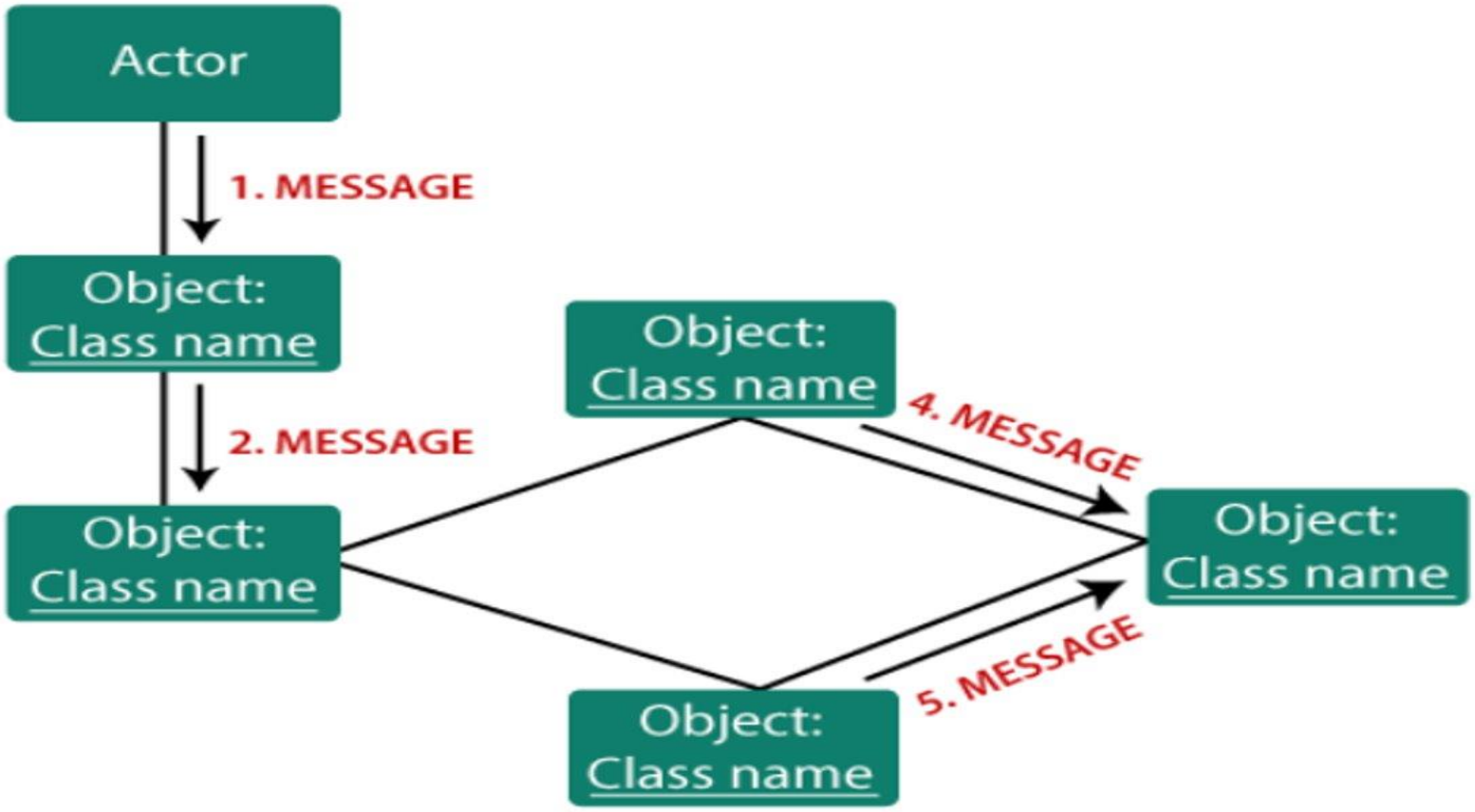
MESSAGES

A message is a communication between objects that conveys information with the expectation that activity will ensue.

In collaboration diagrams, a message is shown as a labeled arrow placed near a link.

- The message is directed from sender to receiver
- The receiver must understand the message
- The association must be navigable in that direction

COMPONENTS OF A COLLABORATION DIAGRAM

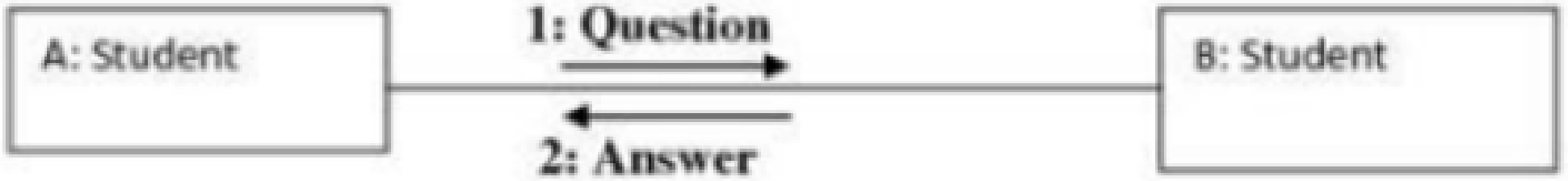


STEPS FOR CREATING COLLABORATION DIAGRAMS

1. Identify behaviour whose realization and implementation is specified
2. Identify the structural elements (class roles, objects, subsystems) necessary to carry out the functionality of the collaboration
    1. Decide on the context of interaction: system, subsystem, use case and operation
3. Model structural relationships between those elements to produce a diagram showing the context of the interaction
4. Consider the alternative scenarios that may be required
    1. Draw instance level collaboration diagrams, if required.
    2. Optionally draw a specification level collaboration diagram to summarize the alternative scenarios in the instance level sequence diagrams

DEPICTING A MESSAGE IN COLLABORATION DIAGRAM

- A message between two objects represents a communication.
- Each message is prefixed by an index value, represents the order in which it will be executed.
- A message can invoke a method, transfer a signal, or create a object.
- An arrow shows the direction of the message.



CONTROL INFORMATION

- Each message is labelled with the message name. Some control information can also be included. Two important types of control information are:
- A **condition** (e.g., [invalid]) indicates that a message is sent, only if the condition is true.
- An **iteration marker** shows that the message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, e.g., [for every book object].

EXAMPLE OF COLLABORATION DIAGRAM

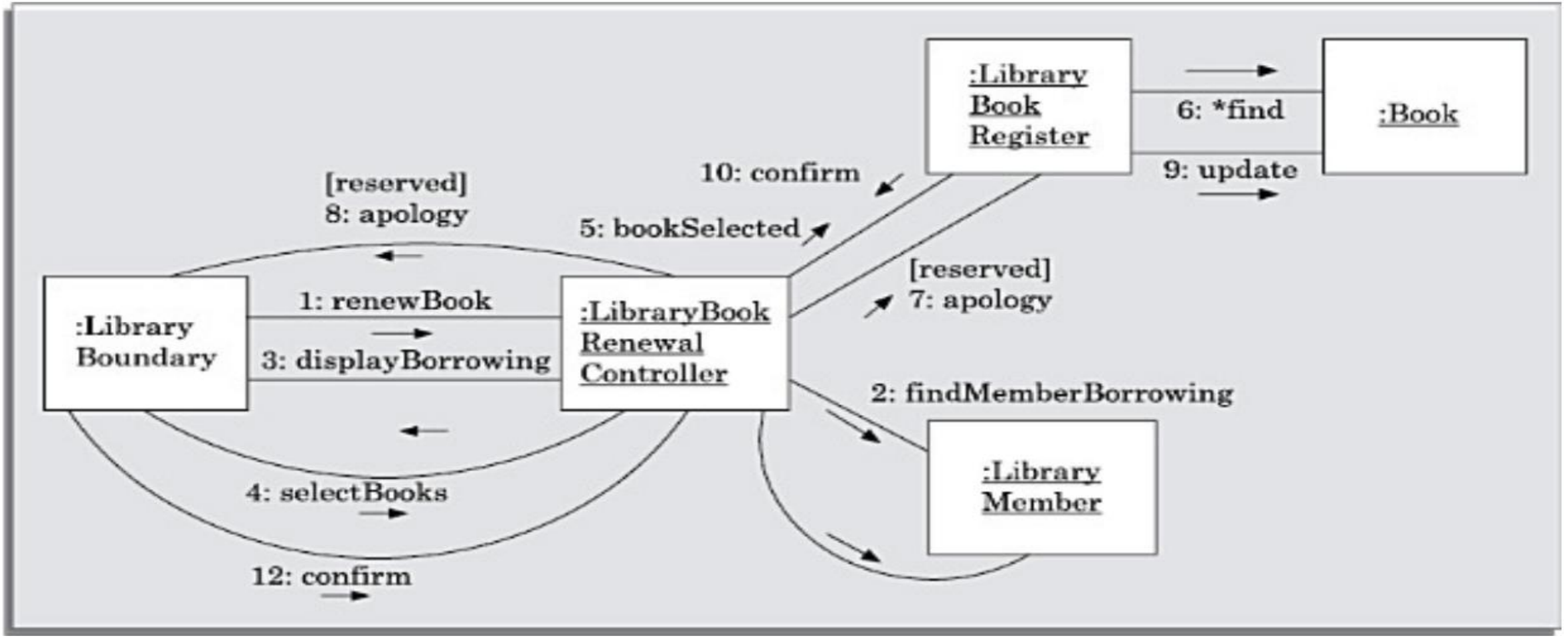
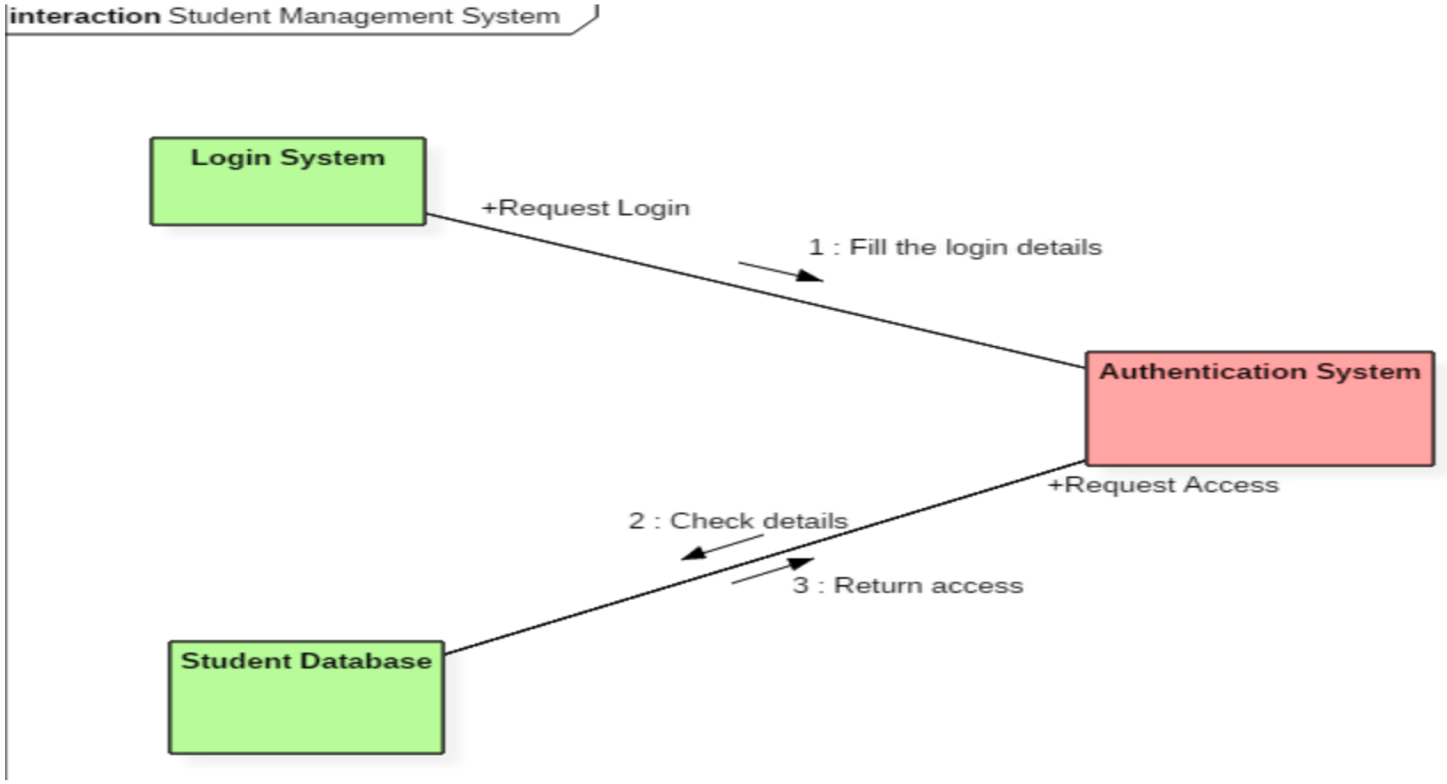


Fig. Collaboration diagram for renewal of a book

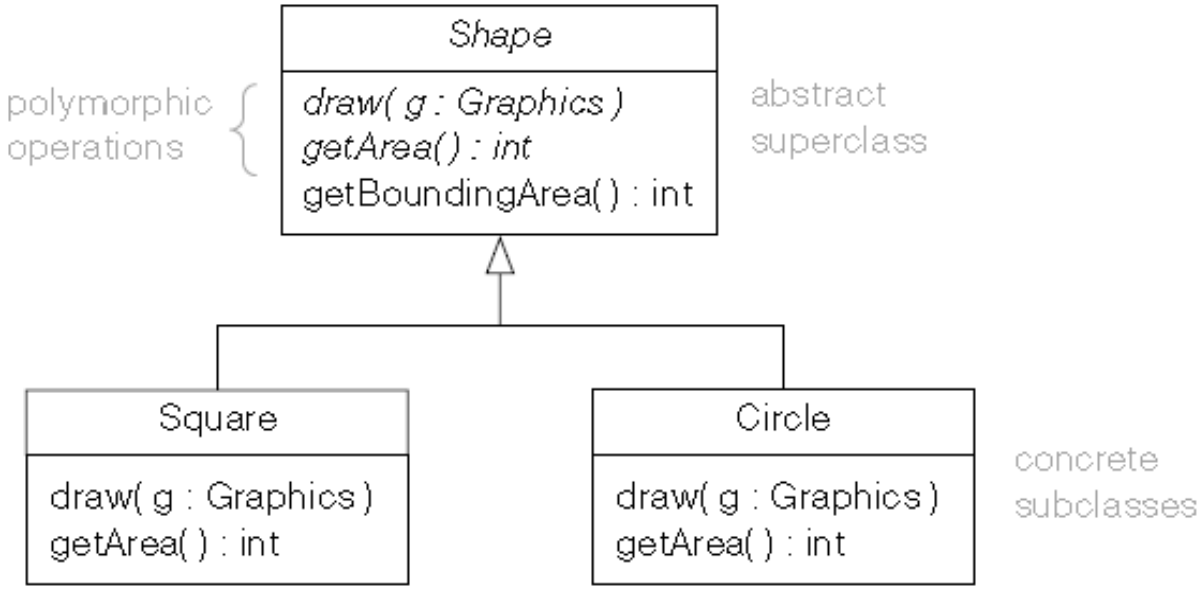
EXAMPLE OF COLLABORATION DIAGRAM



Collaboration diagram for student management system

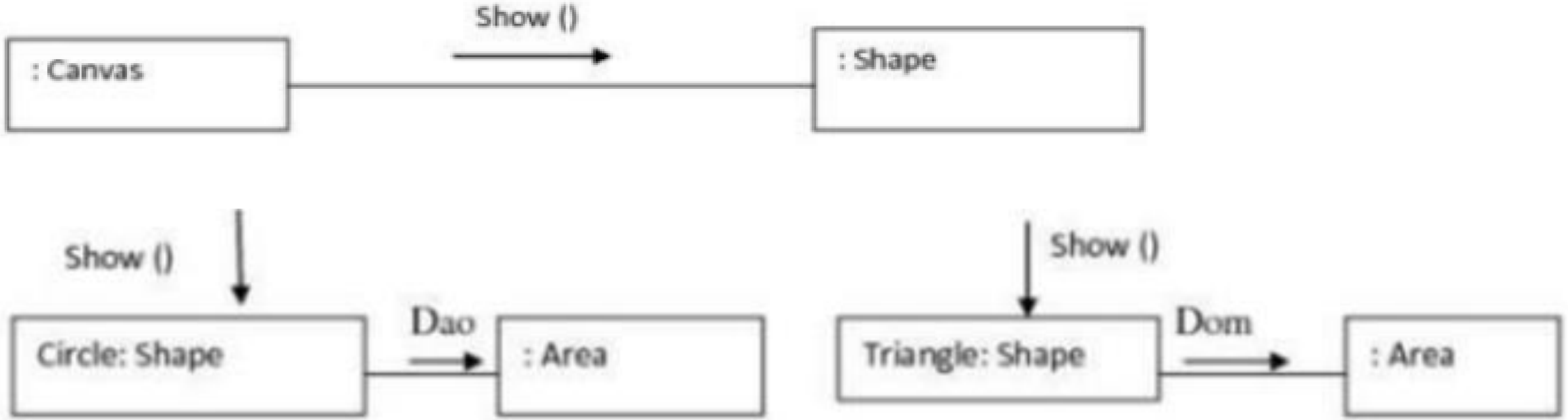
POLYMORPHISM

- **Polymorphism means "many forms". Polymorphic operations have many implementations.**
- **Polymorphism means objects of different classes have operations with the same signature but different implementations.**



POLYMORPHISM IN COLLABORATION DIAGRAM

- Polymorphism is a mechanism in which the operations have the same name but different behaviors in different environments



USE OF SELF IN COLLABORATION DIAGRAM:

- Self is a key word used to represent self message in collaboration diagram. This allows the objects sending message to Sending message to itself

BENEFITS OF A COLLABORATION DIAGRAM

- The collaboration diagram is also known as Communication Diagram.
- It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
- The messages transmitted over sequencing is represented by numbering each individual message.
- The special case of a collaboration diagram is the object diagram.
- It focuses on the elements and not the message flow, like sequence diagrams.

DRAWBACK OF A COLLABORATION DIAGRAM

- Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore all the objects.
- It is a time-consuming diagram.
- After the program terminates, the object is destroyed.
- As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

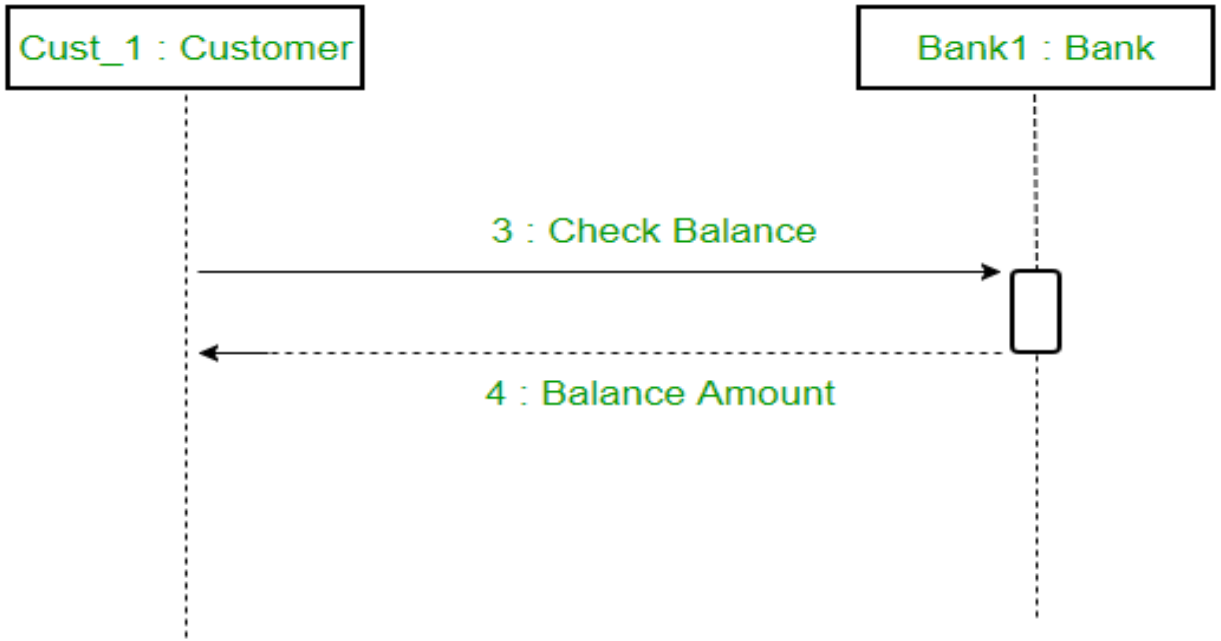
SEQUENCE DIAGRAM

- **UML** Sequence Diagrams are interaction diagrams that detail how operations are carried out.
- They capture the interaction between objects in the context of a collaboration.
- Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when.
- Sequence Diagrams captures high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams).
- Sequence Diagrams show elements as they interact over time and they are organized according to object (horizontally) and time (vertically)

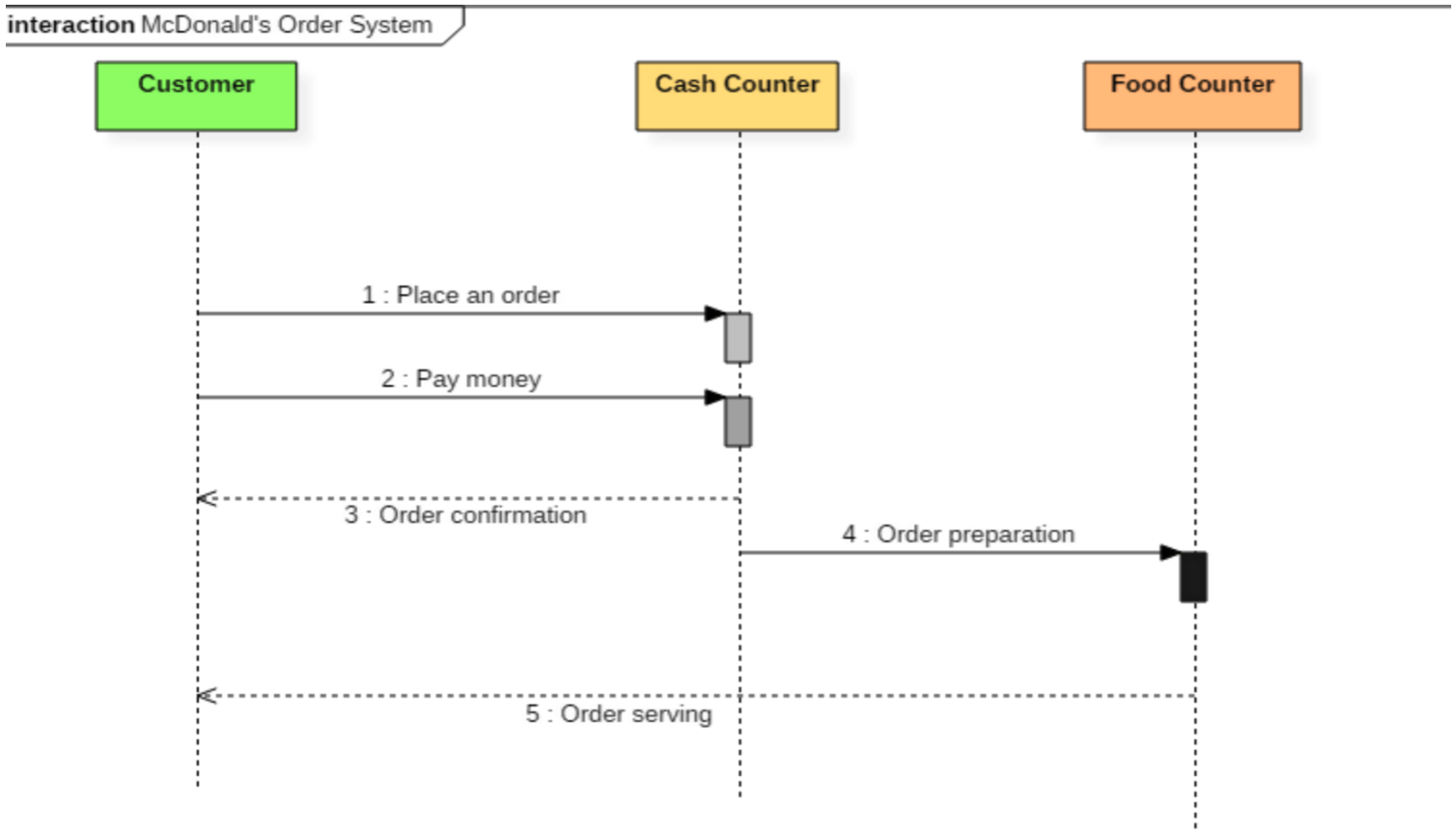
PURPOSE OF SEQUENCE DIAGRAM

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

EXAMPLE OF SEQUENCE DIAGRAM



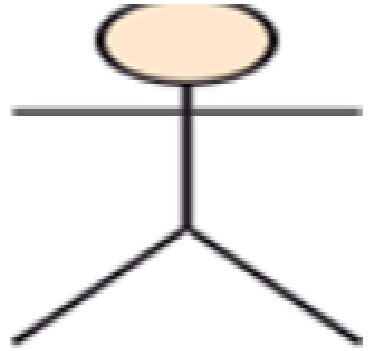
EXAMPLE OF SEQUENCE DIAGRAM



Sequence diagram of Mcdonald's ordering system

NOTATIONS OF SEQUENCE DIAGRAM

1. Actors



- A type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)
- External to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).
- Represent roles played by human users, external hardware, or other subjects.

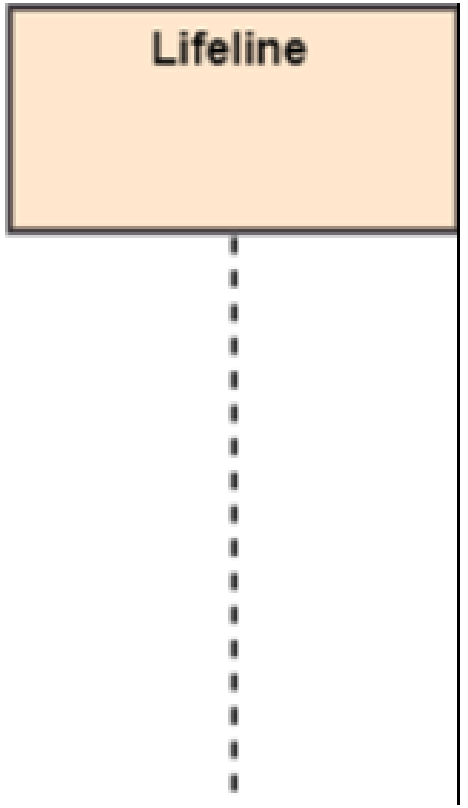
Note that:

- An actor does not necessarily represent a specific physical entity but merely a particular role of some entity
- A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.

NOTATIONS OF SEQUENCE DIAGRAM

Lifeline

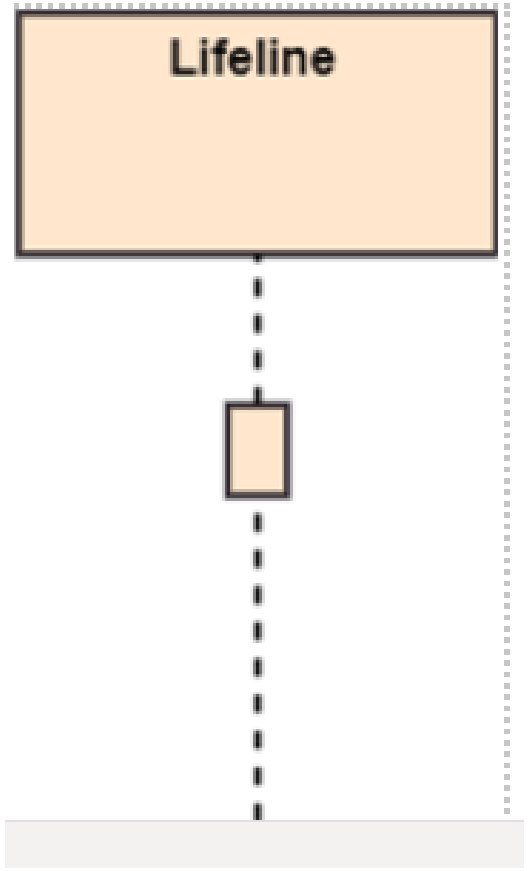
An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.



NOTATIONS OF SEQUENCE DIAGRAM

Activation

- A thin rectangle on a lifeline represents the period during which an element is performing an operation.
- The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively



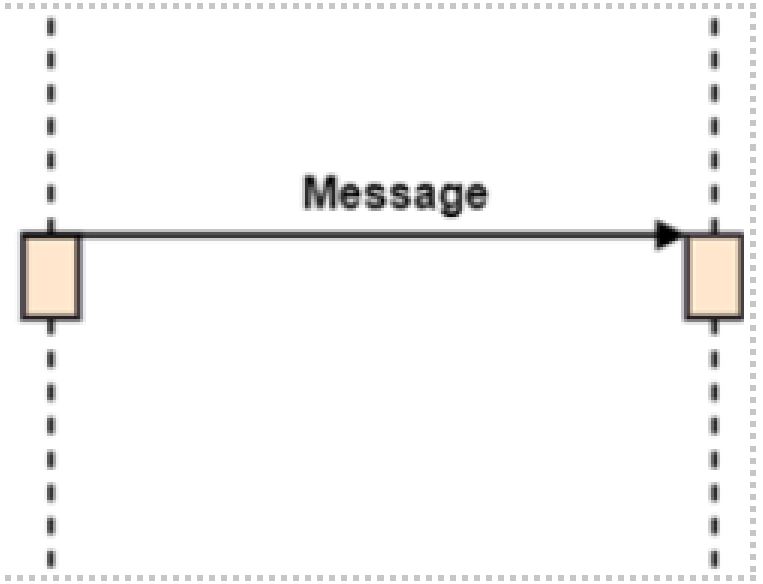
MESSAGES IN SEQUENCE DIAGRAM

Messages

The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

1. Call Message:

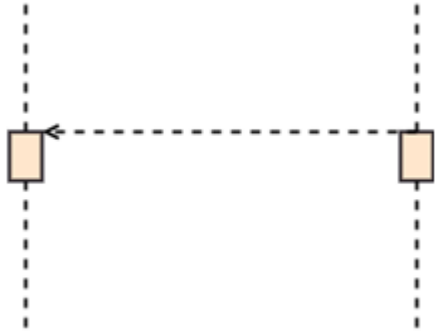
- A message defines a particular communication between Lifelines of an Interaction.
- Call message is a kind of message that represents an invocation of operation of target lifeline.



MESSAGES IN SEQUENCE DIAGRAM

2. Return Messages

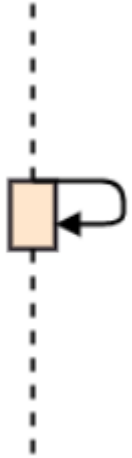
- A message defines a particular communication between Lifelines of an Interaction.
- Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



3. Self **Message**

•A message defines a particular communication between Lifelines of an Interaction.

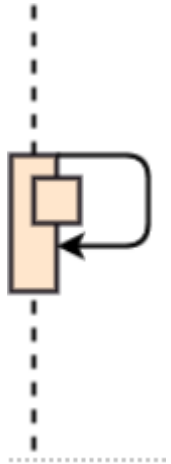
•Self message is a kind of message that represents the invocation of message of the same lifeline.



MESSAGES IN SEQUENCE DIAGRAM

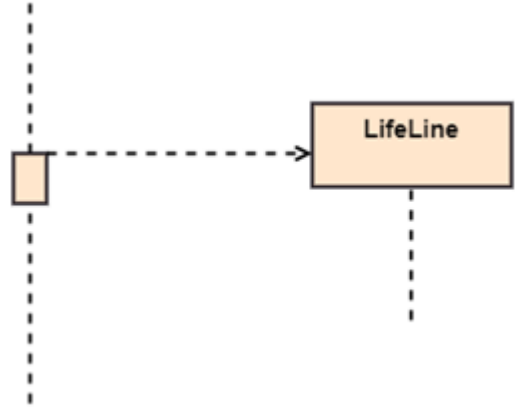
4. Recursive Message

- A message defines a particular communication between Lifelines of an Interaction.
- Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.



5. Create Message

- A message defines a particular communication between Lifelines of an Interaction.
- Create message is a kind of message that represents the instantiation of (target) lifeline.



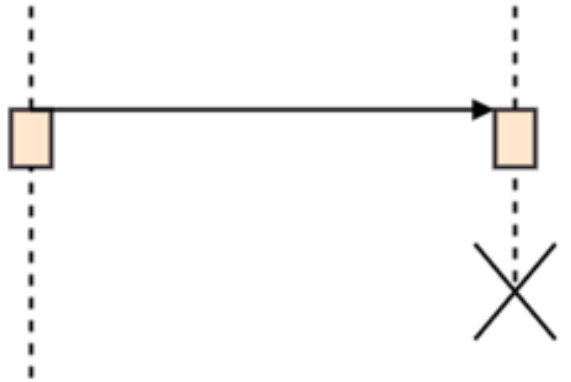
MESSAGES IN SEQUENCE DIAGRAM

ABES Engineering College
Estd. 2000
College Code-032

4. Destroy Message

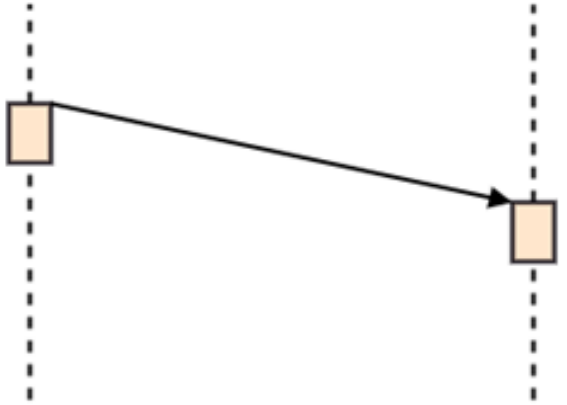
•A message defines a particular communication between Lifelines of an Interaction.

•Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.



5. Duration Message

- A message defines a particular communication between Lifelines of an Interaction.
- Duration message shows the distance between two time instants for a message invocation.



MESSAGES IN SEQUENCE DIAGRAM

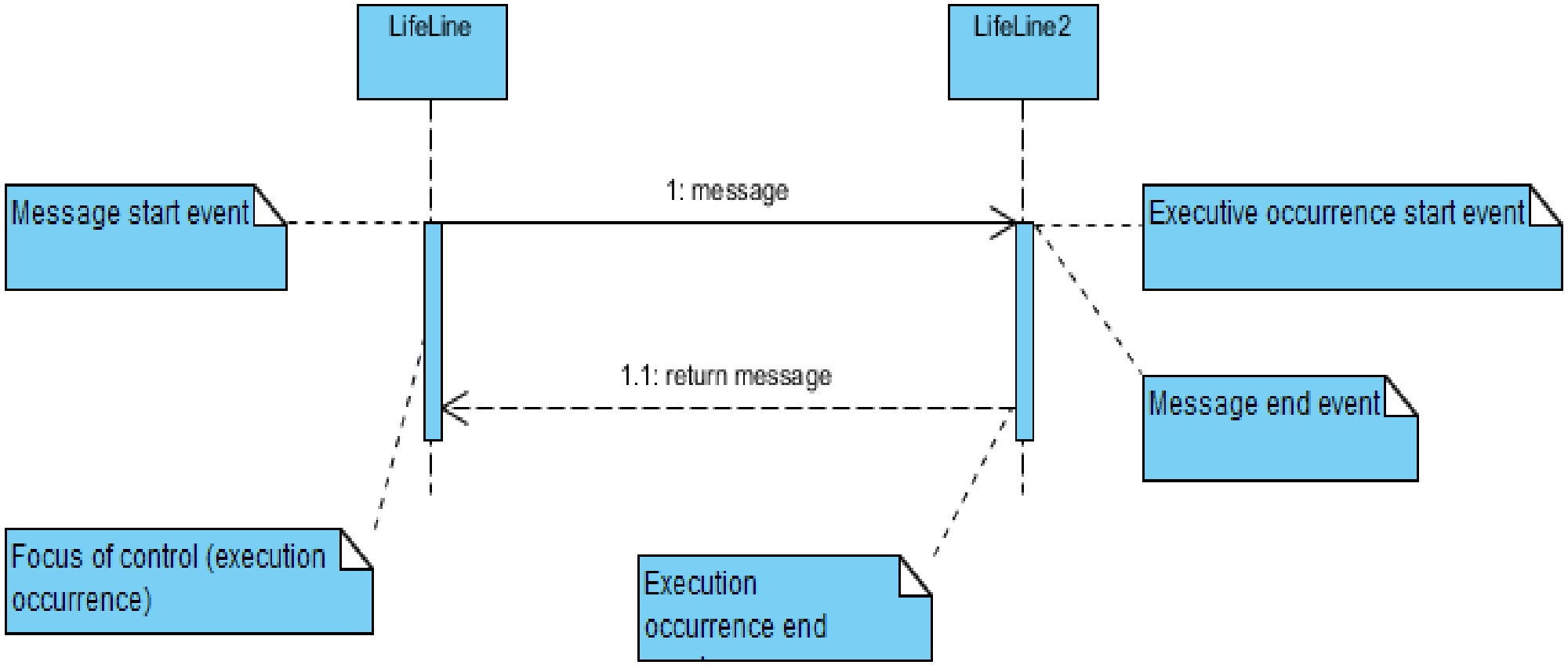
Note

A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

MESSAGE AND FOCUS OF CONTROL

- An Event is any point in an interaction where something occurs.
- Focus of control: also called execution occurrence, an execution occurrence
- It shows as tall, thin rectangle on a lifeline.
- It represents the period during which an element is performing an operation. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.

MESSAGE AND FOCUS OF CONTROL



EXAMPLE OF SEQUENCE DIAGRAM

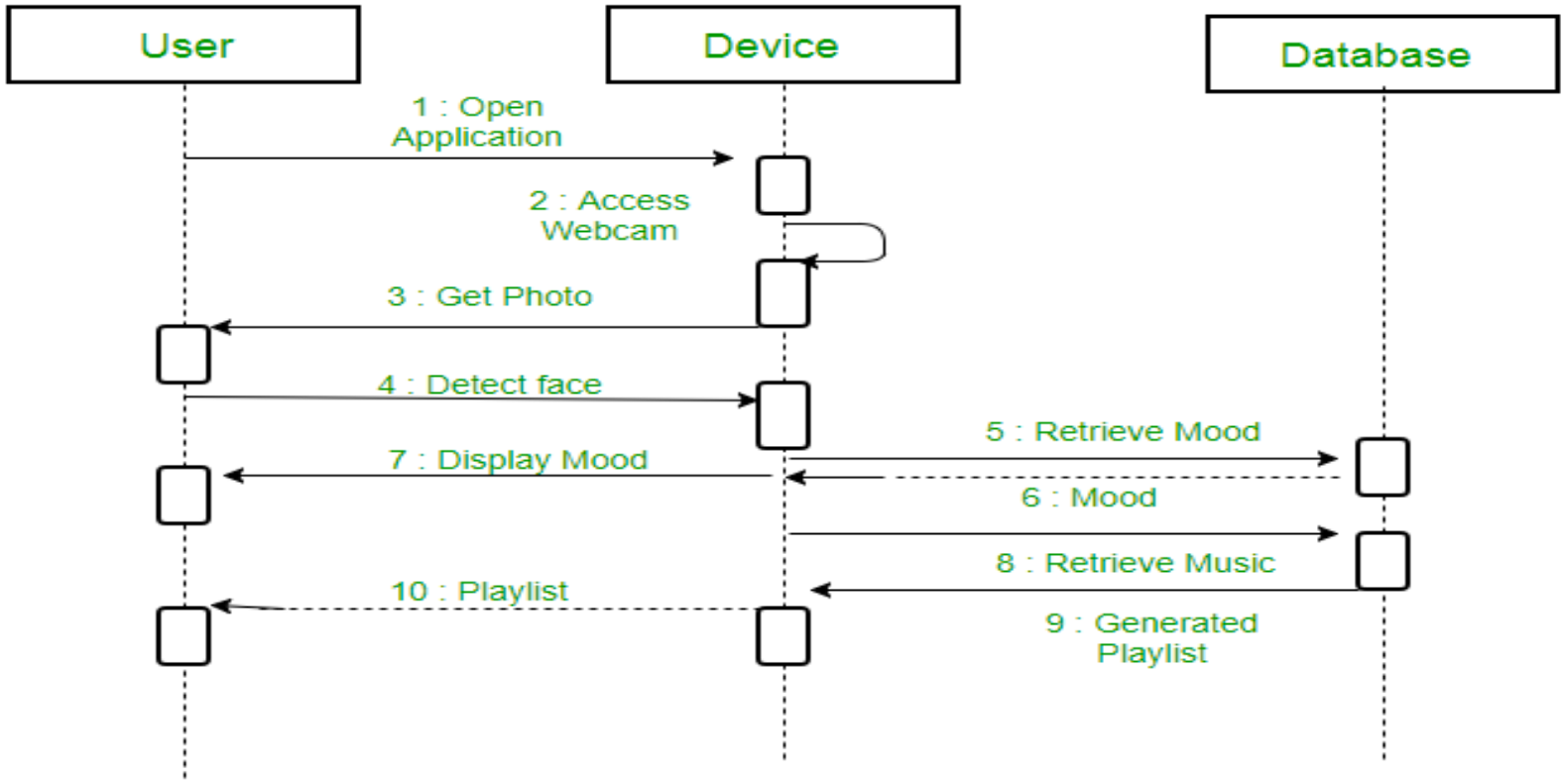
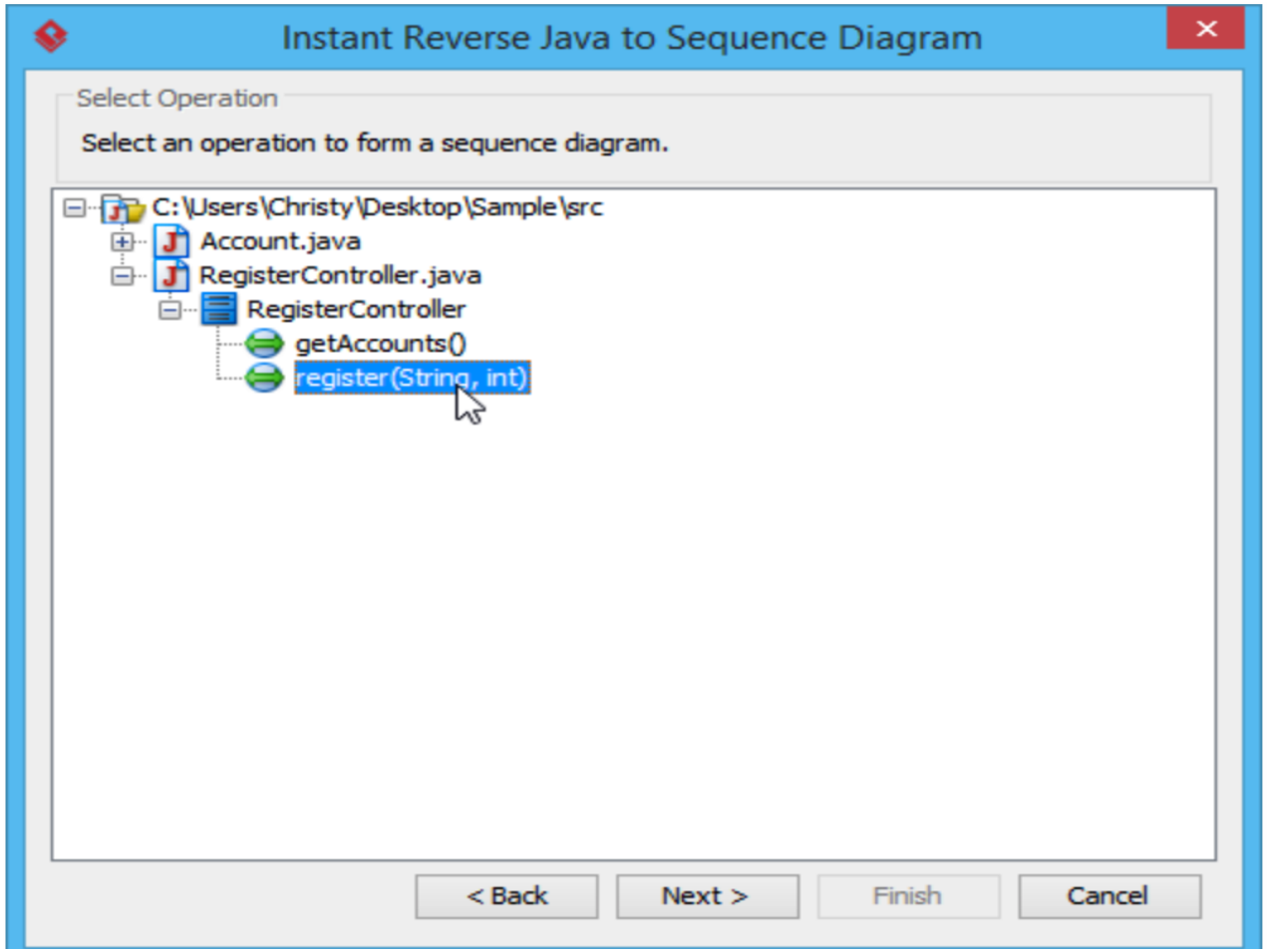
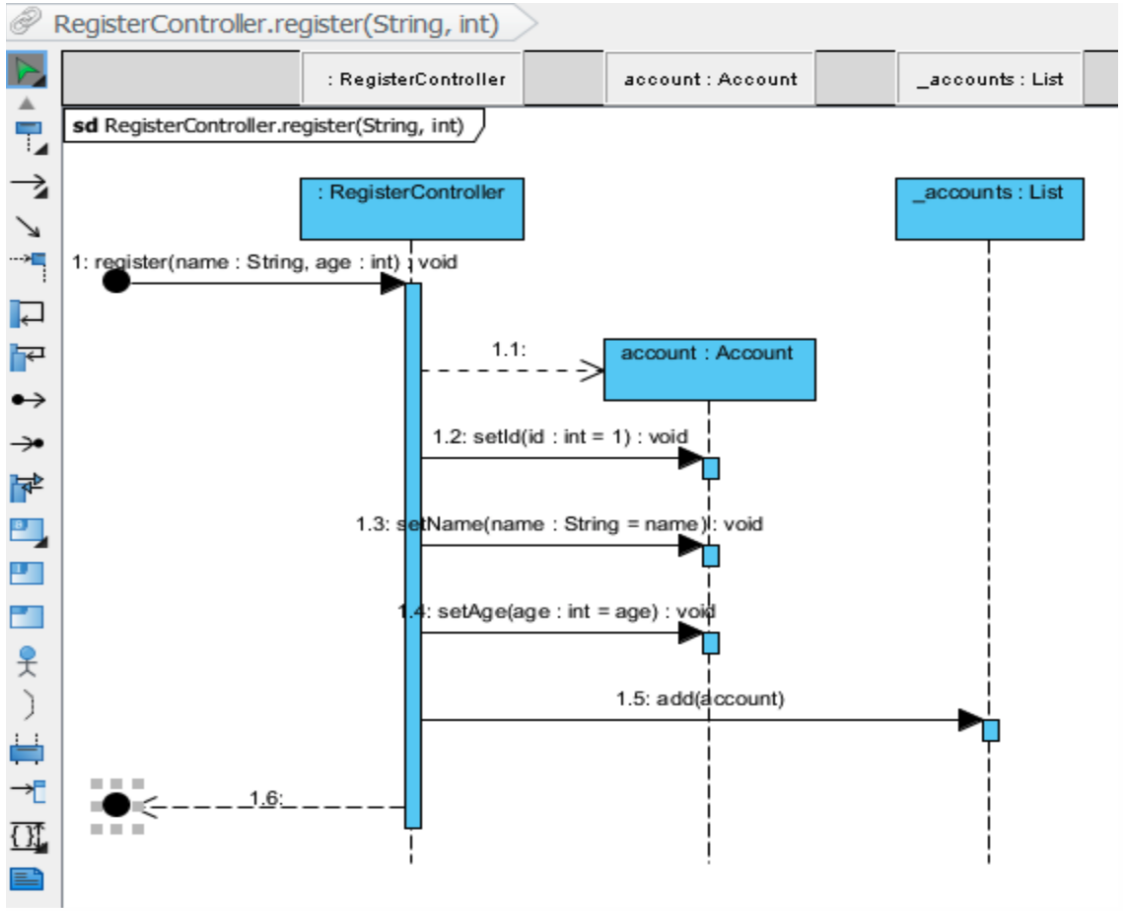


Fig. Sequence diagram for an emotion based music player

EXAMPLE OF SEQUENCE DIAGRAM



EXAMPLE OF SEQUENCE DIAGRAM



RegisterController.java - Notepad

File Edit Format View Help

```
import java.util.*;

public class RegisterController {

  private List _accounts = new ArrayList();

        public void register(String name, int age) {
                Account account = new Account();
                account.setId(1);
                account.setName(name);
                account.setAge(age);
                _accounts.add(account);
        }

        public List getAccounts(){
                return _accounts;
        }
}
```

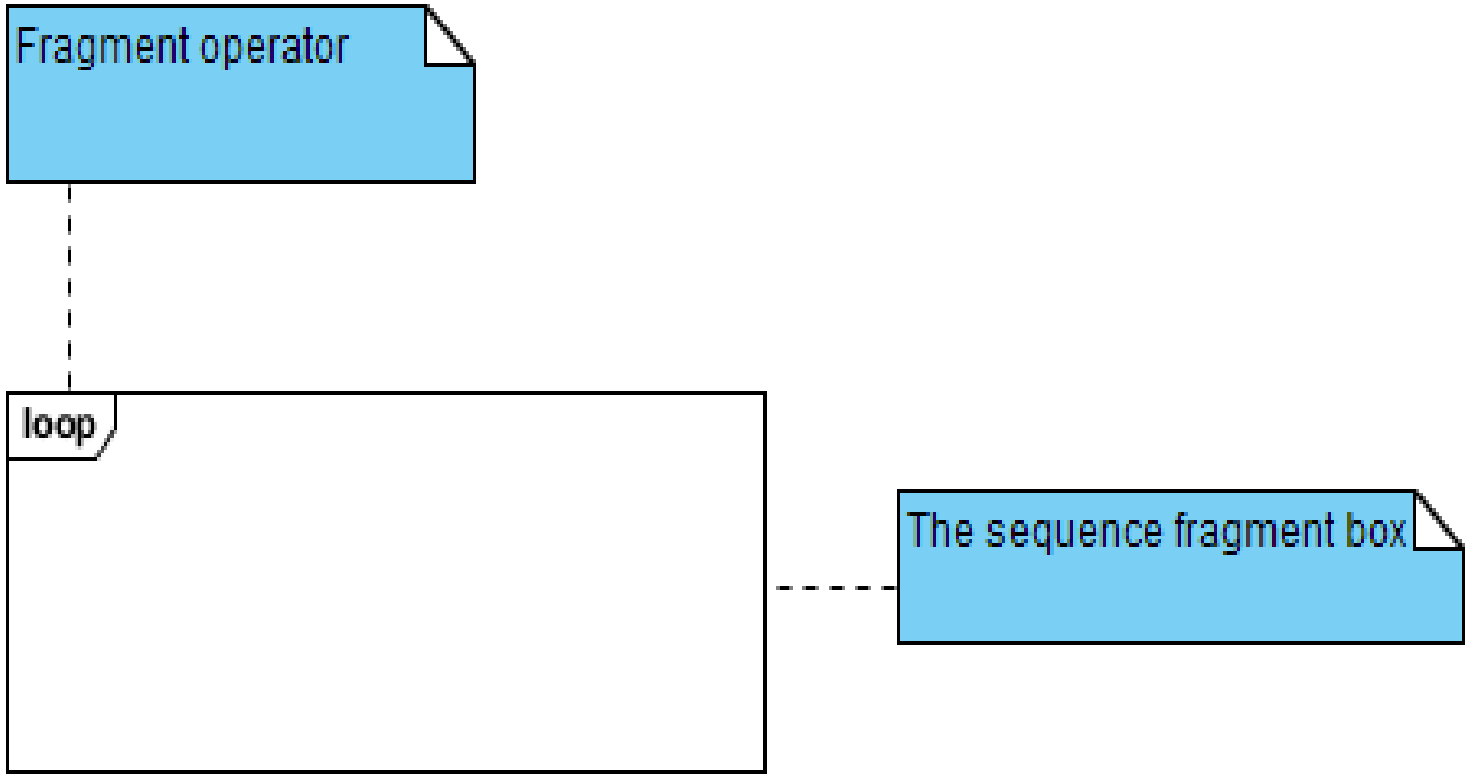
EXPLANATION

When a person invokes *RegisterController*'s register method (message: 1), it creates an account object (message: 1.1). After that, the controller sets the id, name and age to the account object (message 1.2, 1.3, 1.4) and adds itself to the account list (message: 1.5). The invocation ends with a return (message 1.6).

SEQUENCE FRAGMENTS

- **UML 2.0** introduces sequence (or interaction) fragments. Sequence fragments make it easier to create and maintain accurate sequence diagrams
- A sequence fragment is represented as a box, called a combined fragment, which encloses a portion of the interactions within a sequence diagram
- The fragment operator (in the top left corner) indicates the type of fragment
- Fragment types: ref, assert, loop, break, alt, opt, neg

SEQUENCE FRAGMENTS



Operator	Fragment Type
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
region	Critical region: the fragment can have only one thread executing it at once.
neg	Negative: the fragment shows an invalid interaction.
ref	Reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.

SEQUENCE FRAGMENTS (CONTD.)

- It is possible to combine frames in order to capture, e.g., loops or branches.
- **Combined fragment** keywords: alt, opt, break, par, seq, strict, neg, critical, ignore, consider, assert and loop.
- Constraints are usually used to show timing constraints on messages. They can apply to the timing of one message or intervals between messages.

EXAMPLE OF SEQUENCE DIAGRAM

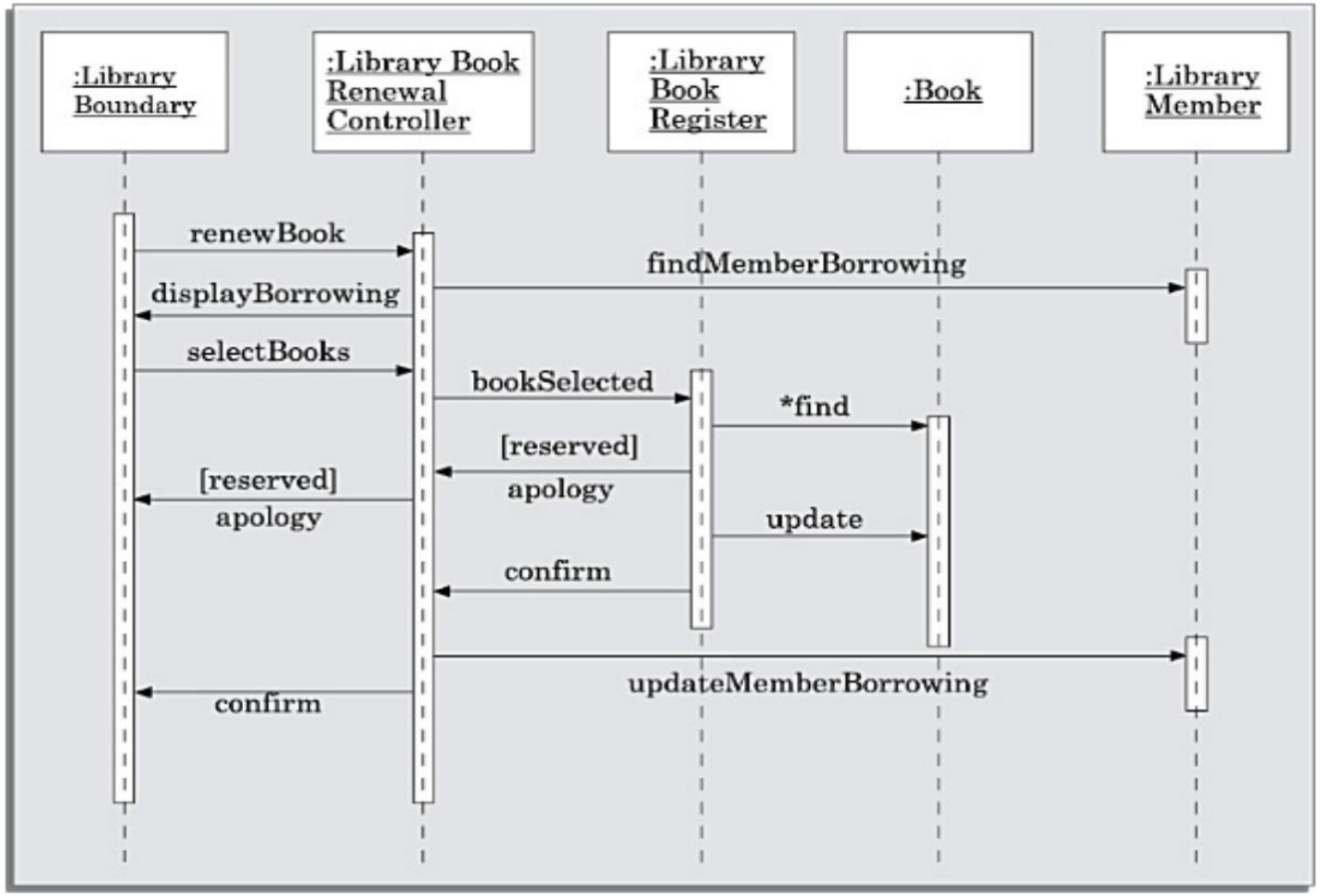


Fig. Sequence diagram for the renew book use case

BENEFITS OF A SEQUENCE DIAGRAM

- It explores the real-time application.
- It depicts the message flow between the different objects.
- It has easy maintenance.
- It is easy to generate.
- Implement both forward and reverse engineering.
- It can easily update as per the new change in the system

THE DRAWBACK OF A SEQUENCE DIAGRAM

- In the case of too many lifelines, the sequence diagram can get more complex.
- The incorrect result may be produced, if the order of the flow of messages changes.
- Since each sequence needs distinct notations for its representation, it may make the diagram more complex.
- The type of sequence is decided by the type of message.

UML DIAGRAMS

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

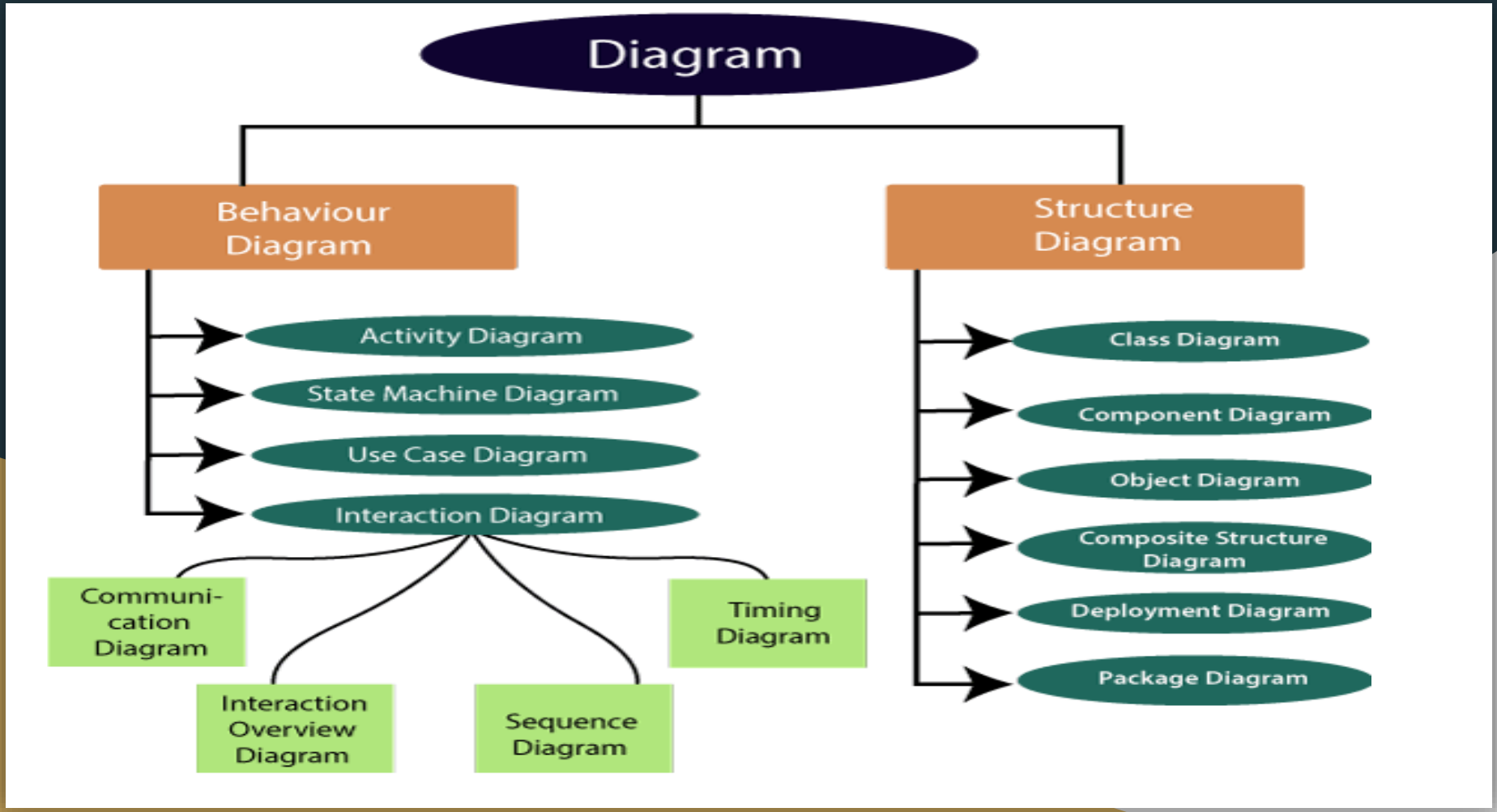


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Use Case Diagram
Interaction Diagram
Communi-cation Diagram
Timing Diagram
Interaction Overview Diagram
Sequence Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram

STRUCTURAL DIAGRAMS

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are −

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram

CLASS DIAGRAMS

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

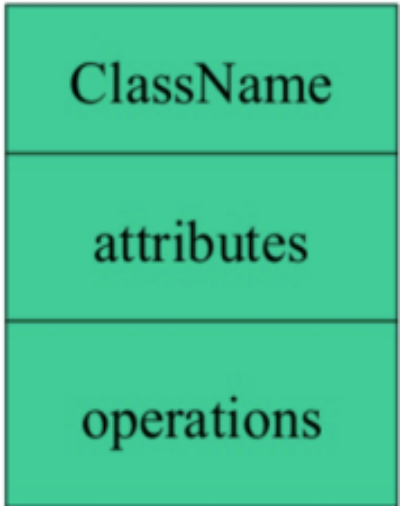
Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

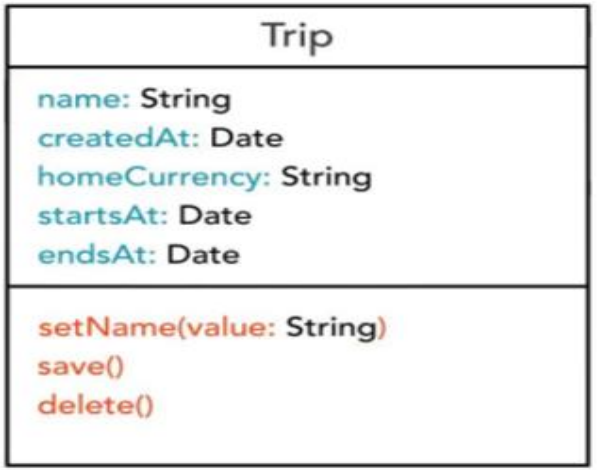
Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Essential Elements of a UML Class Diagram

Essential elements of UML class diagram are:

1. Class Name
2. Attributes
3. Operations





Class Name

The name of the class is only needed in the graphical representation of the class. It appears in the topmost compartment. A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics. The class is rendered as a rectangle, including its name, attributes, and operations in separate compartments.

Following rules must be taken care of while representing a class:

1. A class name should always start with a capital letter.
2. A class name should always be in the center of the first compartment.
3. A class name should always be written in **bold** format.
4. An abstract class name should be written in italics format.

Attributes:

An attribute is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.

Attributes characteristics:

- The attributes are generally written along with the visibility factor.
- Public, private, protected and package are the four visibilities which are denoted by +, -, #, or ~ signs respectively.
- Visibility describes the accessibility of an attribute of a class.
- Attributes must have a meaningful name that describes the use of it in a class.

Class Operations (Methods)

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters is shown after the colon following the parameter name.
- Operations map onto class methods in code.

Relationships in Class Diagrams

Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections. The following are such types of logical connections that are possible in UML:

- Dependency
- Association
- Directed Association
- Reflexive Association
- Multiplicity
- Aggregation
- Composition
- Inheritance/Generalization
- Realization

Dependency

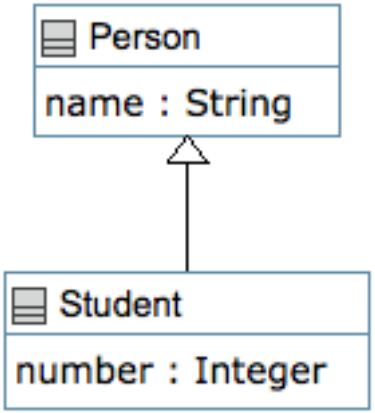
A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another.

In the following example, Student has a dependency on College

Generalization:

A generalization helps to connect a subclass to its superclass. A sub-class is inherited from its superclass. Generalization relationship can't be used to model interface implementation. Class diagram allows inheriting from multiple superclasses.

In this example, the class Student is generalized from Person Class.



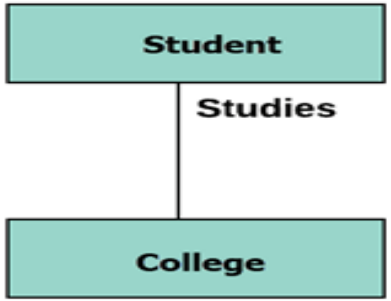
Association:

This kind of relationship represents static relationships between classes A and B. For example; an employee works for an organization.

Here are some rules for Association:

- Association is mostly verb or a verb phrase or noun or noun phrase.
- It should be named to indicate the role played by the class attached at the end of the association path.
- Mandatory for reflexive associations

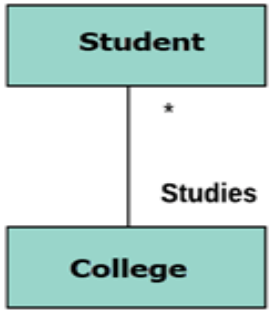
In this example, the relationship between student and college is shown which is studies.



Multiplicity

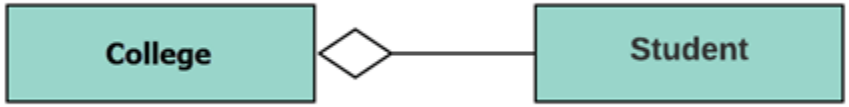
A multiplicity is a factor associated with an attribute. It specifies how many instances of attributes are created when a class is initialized. If a multiplicity is not specified, by default one is considered as a default multiplicity.

Let's say that that there are 100 students in one college. The college can have multiple students.



Aggregation

Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.



For example, the class college is made up of one or more student. In aggregation, the contained classes are never totally dependent on the lifecycle of the container. Here, the college class will remain even if the student is not available.

Composition:

The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.

For example, if college is composed of classes student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.

PURPOSE OF CLASS DIAGRAMS

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Basically, Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the **drawing procedure of class diagram**.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

The following points should be remembered while drawing a class diagram −

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

Where to Use Class Diagrams?

Class diagrams are used for −

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

BENEFITS OF CLASS DIAGRAM

- Class Diagram Illustrates data models for even very complex information systems
- It provides an overview of how the application is structured before studying the actual code. This can easily reduce the maintenance time
- It helps for better understanding of general schematics of an application.
- Allows drawing detailed charts which highlights code required to be programmed
- Helpful for developers and other stakeholders.

Class Diagram in Software Development Lifecycle

Class diagrams can be used in various software development phases. It helps in modeling class diagrams in three different perspectives.

1. Conceptual perspective: Conceptual diagrams are describing things in the real world. You should draw a diagram that represents the concepts in the domain under study. These concepts related to class and it is always language-independent.

2. Specification perspective: Specification perspective describes software abstractions or components with specifications and interfaces. However, it does not give any commitment to specific implementation.

3. Implementation perspective: This type of class diagrams is used for implementations in a specific language or application. Implementation perspective, use for software implementation.

VISIBILITY IN CLASS DIAGRAMS

The visibility of the attributes and operations can be represented in the following ways:

- **Public** − A public member is visible from anywhere in the system. In class diagram, **it is prefixed by the symbol '+'**.
- **Private** − A private member is visible only from within the class. It cannot be accessed from outside the class. A private member **is prefixed by the symbol '−'.**
- **Protected** − A protected member is visible from within the class and from the subclasses inherited from this class, but not from outside. **It is prefixed by the symbol '#'.**

CLASS DIAGRAM EXAMPLES

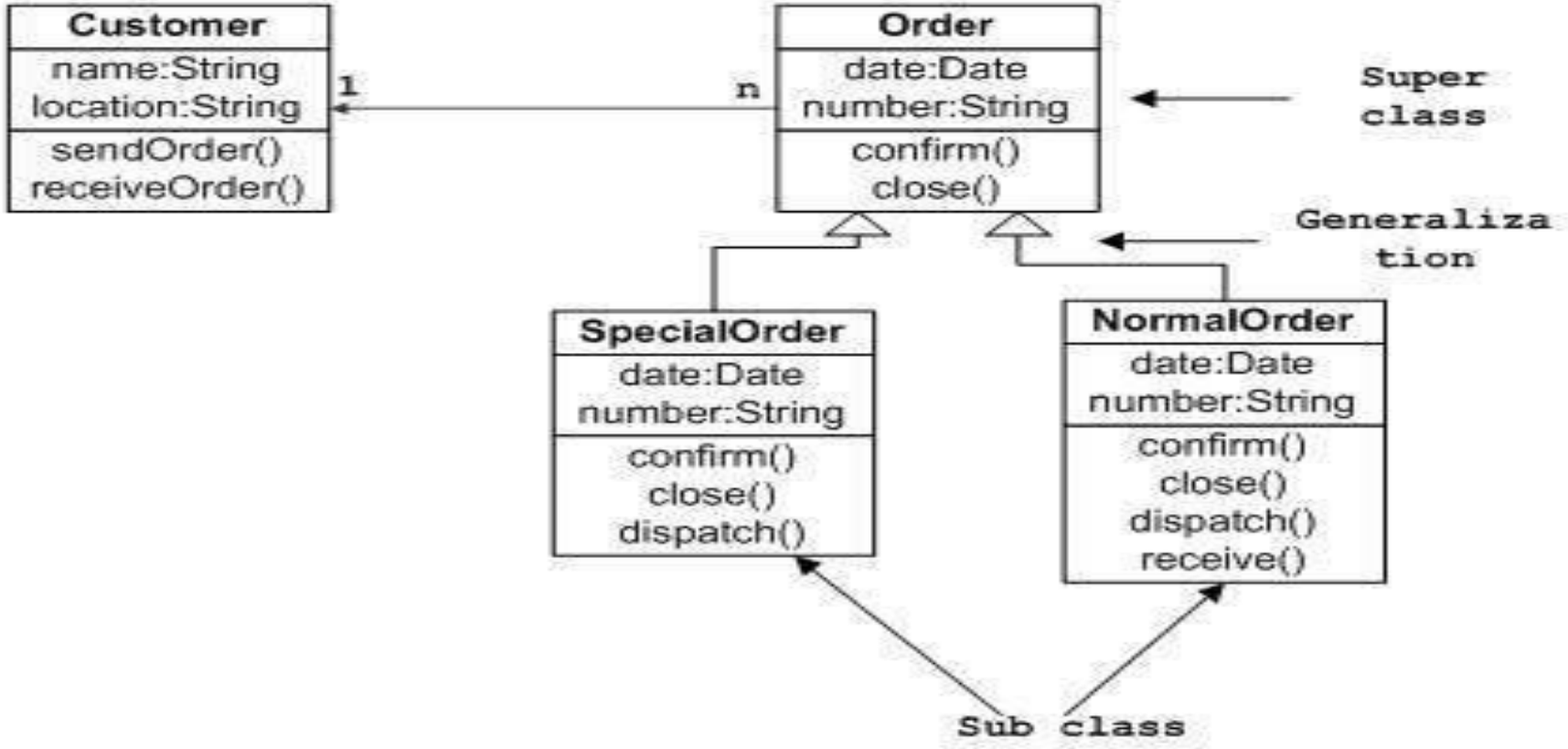
EXAMPLES OF CLASS DIAGRAM

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
- Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.

Sample Class Diagram



CONCLUSION

- UML is the standard language for specifying, designing, and visualizing the artifacts of software systems
- A class is a blueprint for an object
- A class diagram describes the types of objects in the system and the different kinds of relationships which exist among them
- It allows analysis and design of the static view of a software application
- Class diagrams are most important UML diagrams used for software application development
- Essential elements of UML class diagram are 1) Class 2) Attributes 3) Relationships
- Class Diagram provides an overview of how the application is structured before studying the actual code. It certainly reduces the maintenance time
- The class diagram is useful to map object-oriented programming languages like Java, C++, Ruby, Python, etc.

OBJECT DIAGRAMS

- Objects are the real-world entities whose behavior is defined by the classes. Objects are used to represent the static view of an object-oriented system. We cannot define an object without its class. Object and class diagrams are somewhat similar.
- In other words, “An object diagram in the Unified Modeling Language (UML), is a diagram that shows a **complete or partial view** of the structure of a modeled system **at a specific time.**”
- An object diagram shows the relation between the instantiated classes and the defined class, and the relation between these objects in the system. They are be useful to explain smaller portions of your system, when your system class diagram is very complex, and also sometimes modeling recursive relationship in diagram.
- Object diagrams are used to render a set of objects and their relationships as an instance.

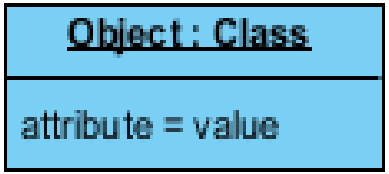
Basic Object Diagram Symbols and Notations

Object Names:

- Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.

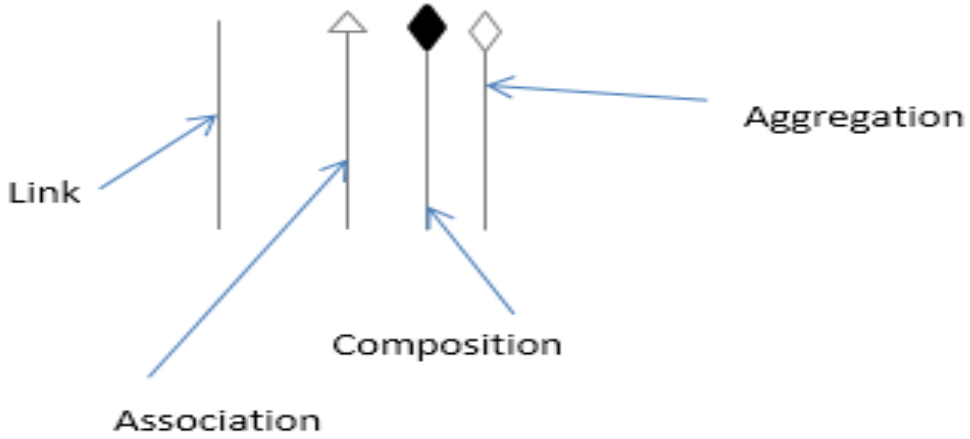
Object Attributes:

- Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.

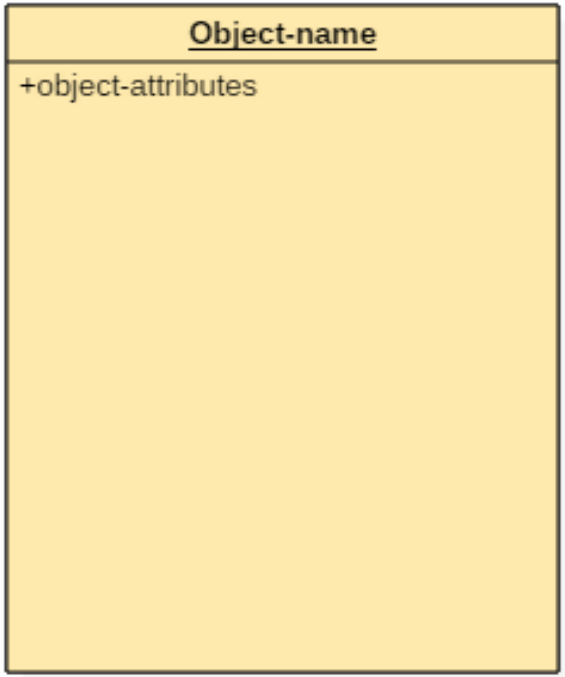


Links:

- Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



Notation of an object diagram:



PURPOSE OF AN OBJECT DIAGRAM:

1. It is used to describe the static aspect of a system.
2. It is used to represent an instance of a class.
3. It can be used to perform forward and reverse engineering on systems.
4. It is used to understand the behavior of an object.
5. It can be used to explore the relations of an object and can be used to analyze other connecting objects.

HOW TO DRAW AN OBJECT DIAGRAM?

1. Before drawing an object diagram, one should analyze all the objects inside the system.
2. The relations of the object must be known before creating the diagram.
3. Association between various objects must be cleared before.
4. An object should have a meaningful name that describes its functionality.
5. An object must be explored to analyze various functionalities of it.

APPLICATIONS OF OBJECT DIAGRAMS:

1. Object diagrams play an essential role while generating a blueprint of an object-oriented system.
2. Object diagrams provide means of modeling the classes, data and other information as a set or a single unit.
3. It is used for analyzing the online or offline system. The functioning of a system can be visualized using object diagrams.

Where to use Object Diagrams?

It can be said that object diagrams are used for −

- Making the prototype of a system.
- Reverse engineering.
- Modeling complex data structures.
- Understanding the system from practical perspective.

EXAMPLES:

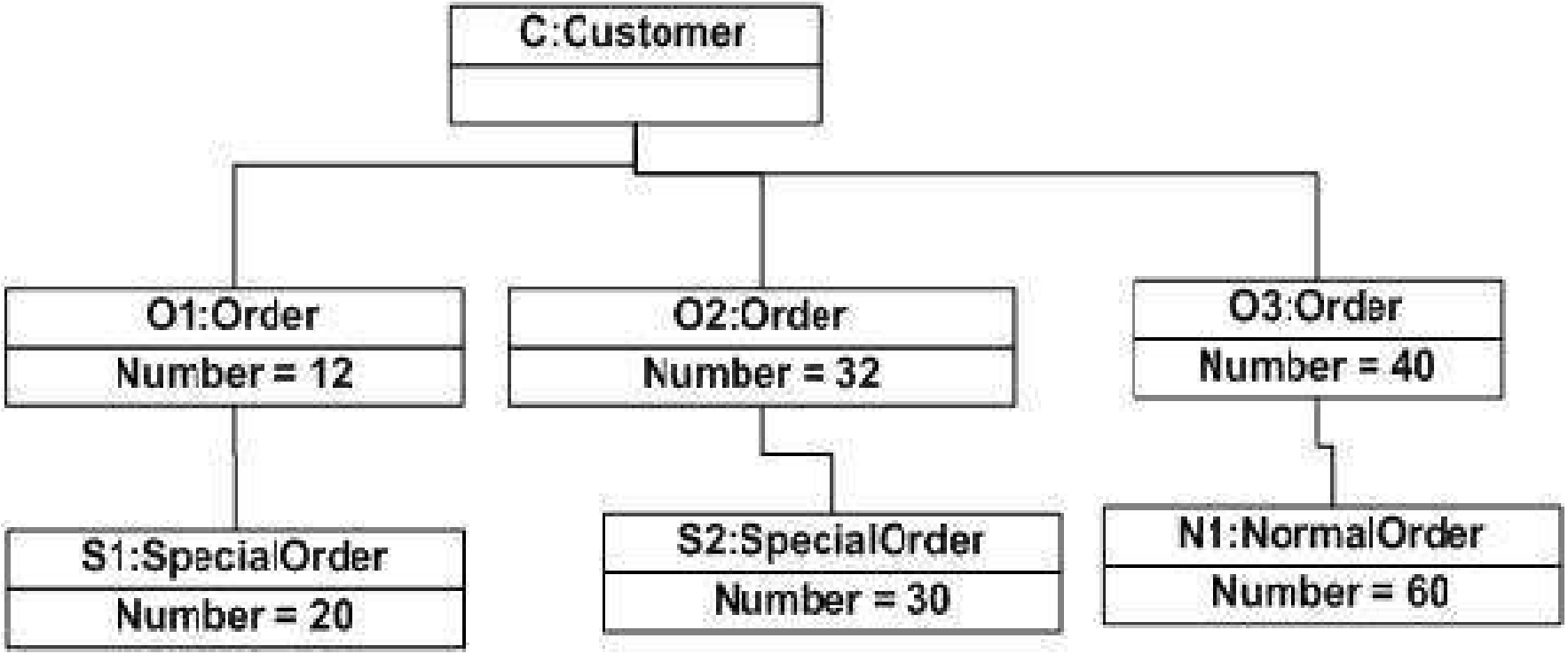
The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

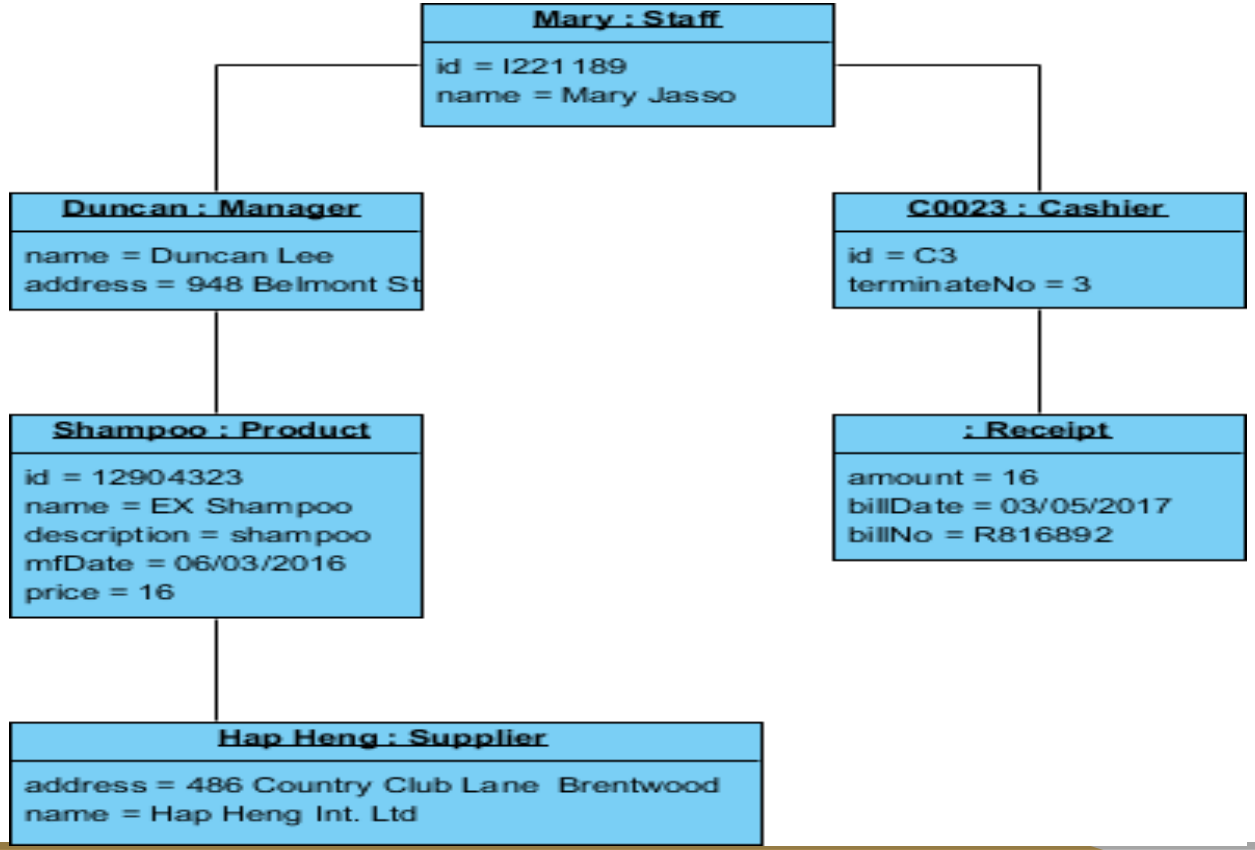
Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

Object diagram of an order management system



Object Diagram Example II - POS



Class vs. Object Diagrams

Serial no.	Class	Object
1	It represents static aspects of a system.	It represents the behavior of a system in real time.
2	It doesn't include dynamic changes.	It captures runtime changes of a system.
3	It never includes attributes or data values of an instance.	It includes attributes and data values of any instance.
4	Class diagram manipulates the behavior of objects.	Objects are instances of classes.

SUMMARY OF OBJECT DIAGRAM

1. Class groups together things that share similar behavior.
2. A class represents a bird's eye view of a system, i.e., an abstraction which is an object-oriented programming concept.
3. An object represents a static view of an object-oriented system.
4. One class can refer to multiple classes.
5. A single class can have any number of objects.
6. Objects are related to one another because they share the same class.
7. The object of different classes can also be connected.

Classification of requirements

Requirements classification

High level

Detailed

Business requirements

User requirements

System requirements

Classification of requirements

- **Business requirements.** These include high-level statements of goals, objectives, and needs.
- **Stakeholder requirements.** The needs of discrete stakeholder groups are also specified to define what they expect from a particular solution.
- **Solution requirements.** Solution requirements describe the characteristics that a product must have to meet the needs of the stakeholders and the business itself.
  - **Nonfunctional** requirements describe the general characteristics of a system. They are also known as *quality attributes*.
  - **Functional** requirements describe how a product must behave, what its features and functions.
- **Transition requirements.** An additional group of requirements defines what is needed from an organization to successfully move from its current state to its desired state with the new product.

Functional Requirements

- Functional requirements describe system behavior under specific conditions and include the product features and functions which web & app developers must add to the solution. Such requirements should be precise both for the development team and stakeholders.
- The list of examples of functional requirements includes:
  - Business Rules
  - Transaction corrections, adjustments, and cancellations
  - Administrative functions
  - Authentication
  - Authorization levels
  - Audit Tracking
  - External Interfaces
  - Certification Requirements
  - Reporting Requirements
  - Historical Data

Example of Functional Requirements

Here, are some examples of non-functional requirement:

1. The software automatically validates customers against the ABC Contact Management System
2. The Sales system should allow users to record customers sales
3. The background color for all windows in the application will be blue and have a hexadecimal RGB color value of 0x0000FF.
4. Only Managerial level employees have the right to view revenue data.
5. The software system should be integrated with banking API
6. The software system should pass Section 508 accessibility requirement.

Non-Functional Requirements

- A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system. Example, how fast does the website load?
- Some typical non-functional requirements are:
  - Performance – for example Response Time, Throughput, Utilization, Static Volumetric
  - Capacity
  - Availability
  - Reliability
  - Recoverability
  - Maintainability
  - Serviceability
  - Security
  - Regulatory
  - Manageability
  - Environmental
  - Data Integrity
  - Usability

Example of Non-Functional Requirements

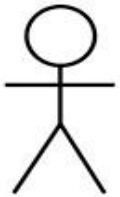
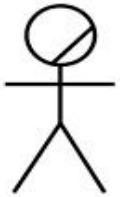
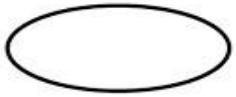
Here, are some examples of non-functional requirement:

1. Users must change the initially assigned login password immediately after the first successful login. Moreover, the initial should never be reused.
2. Employees never allowed to update their salary information. Such attempt should be reported to the security administrator.
3. Every unsuccessful attempt by a user to access an item of data shall be recorded on an audit trail.
4. A website should be capable enough to handle 20 million users with affecting its performance
5. The software should be portable. So moving from one OS to other OS does not create any problem.
6. Privacy of information, the export of restricted technologies, intellectual property rights, etc. should be audited.

Use-Cases

- Use cases describe the interaction between the system and external users that leads to achieving particular goals.
- Each use case includes three main elements:
  - **Actors.** These are the users outside the system that interact with the system.
  - **System**. The system is described by functional requirements that define an intended behavior of the product.
  - **Goals**. The purposes of the interaction between the users and the system are outlined as goals.
- There are two formats to represent use cases:
  - Use case specification/description
  - Use case diagram

Use-Cases Elements

Symbol Name	Symbol
Actor	
Business Actor	
Use Case	
Business Use Case	
Association	
Dependency	
Generalization	

Use-Cases Elements

Connection between Actor and Use Case

Boundary of system

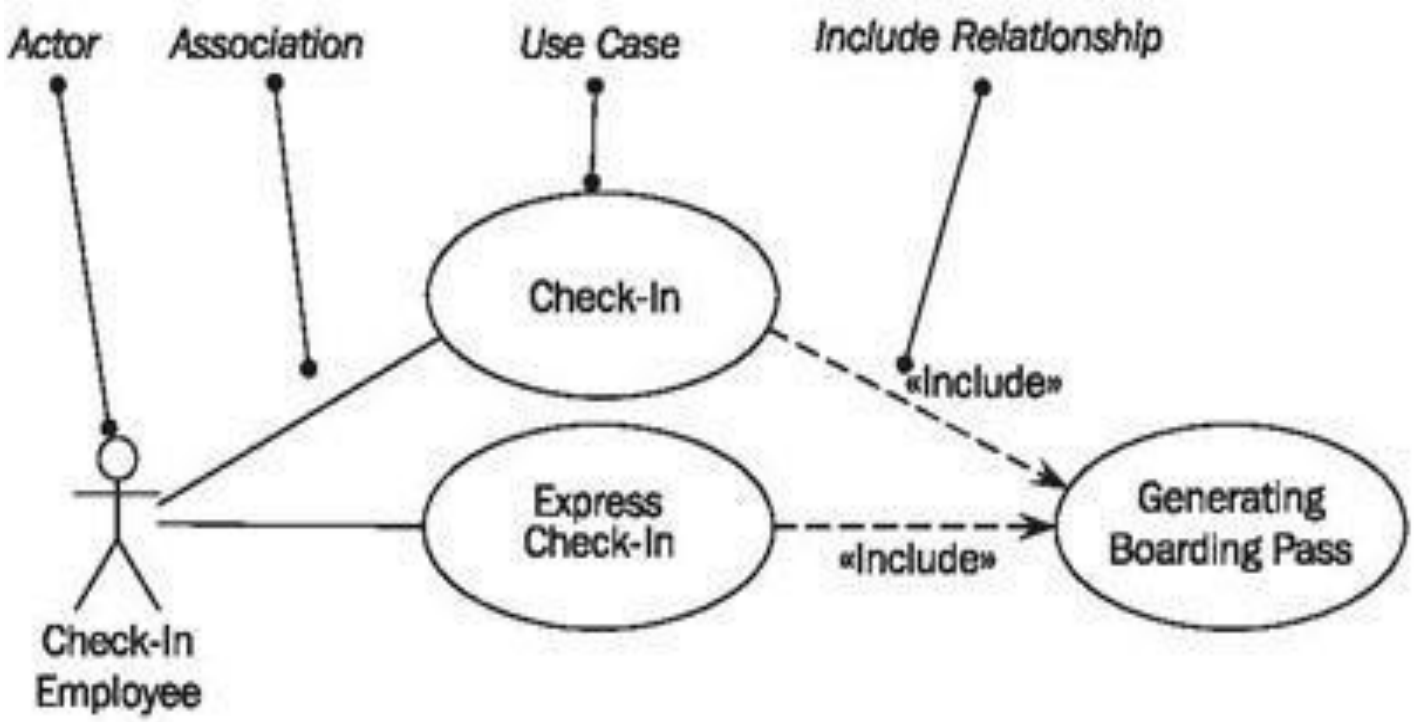
<<include>>

Include relationship between Use Cases (one UC must call another; e.g., Login UC includes User Authentication UC)

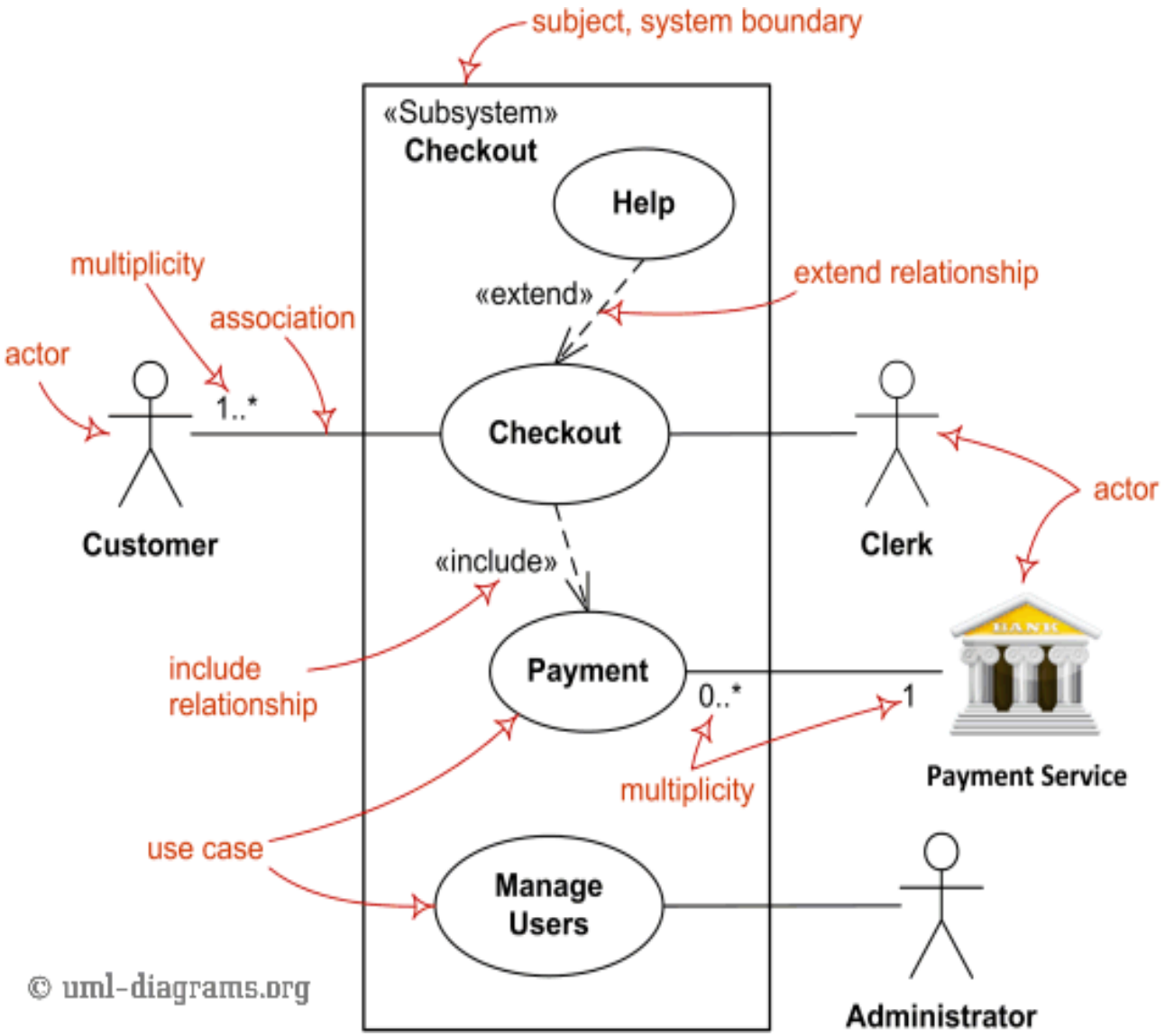
<<extend>>

Extend relationship between Use Cases (one UC calls Another under certain condition; think of if-then decision points)

Use-Cases Elements

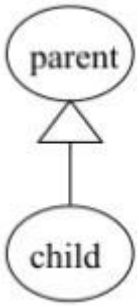


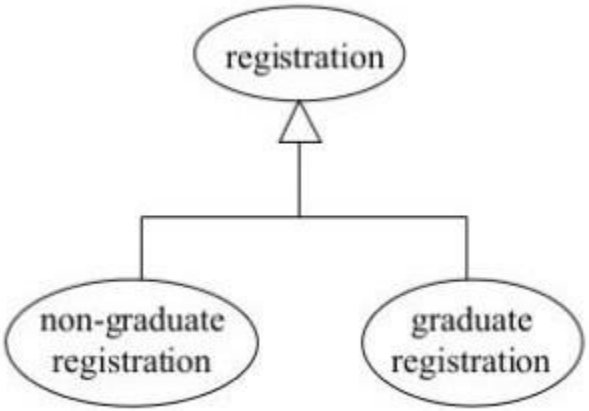
Use Cases Diagram

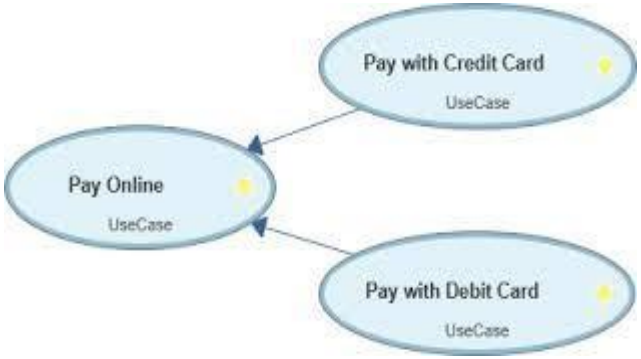


Use-Cases Generalization

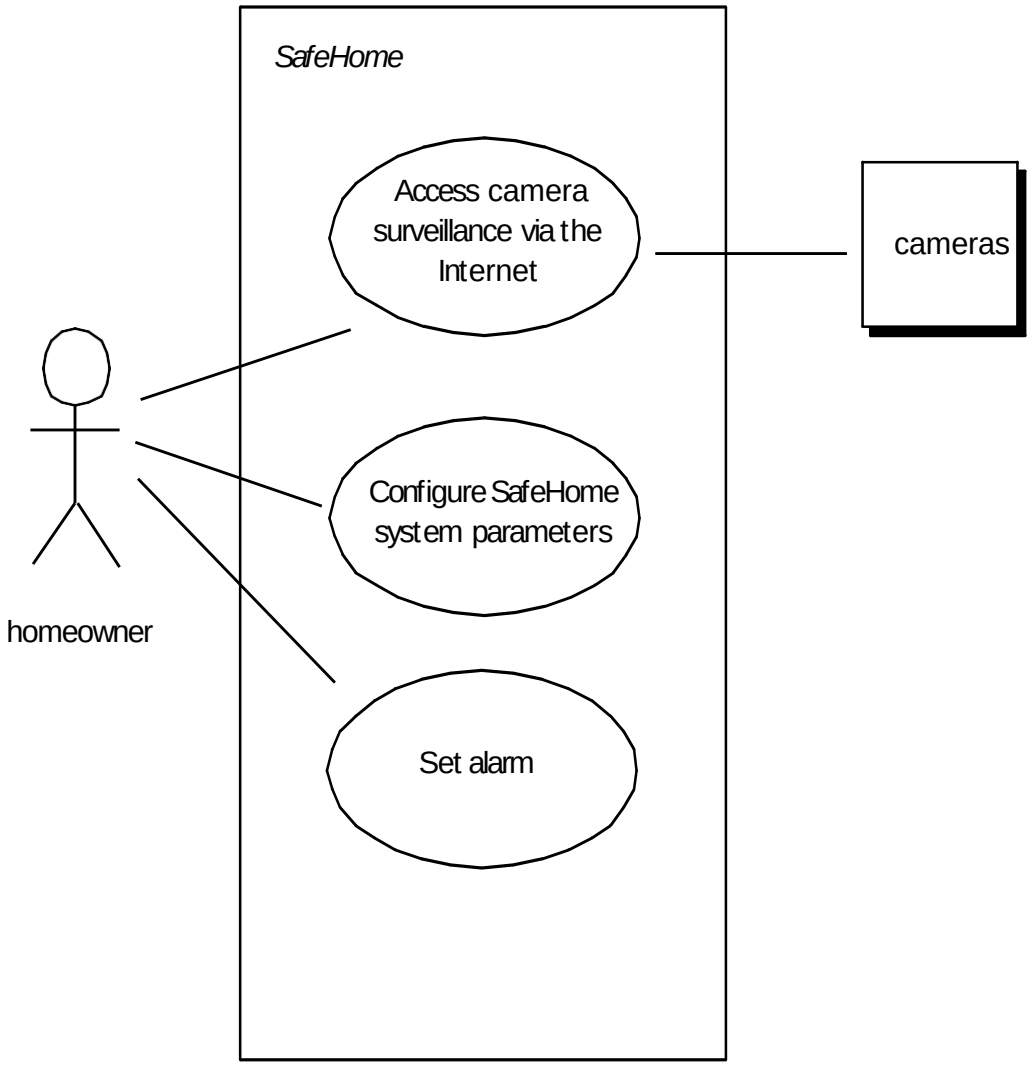
- The child use case inherits the behavior meaning of the parent use case
- The Child may add to or override the behavior of its parent.



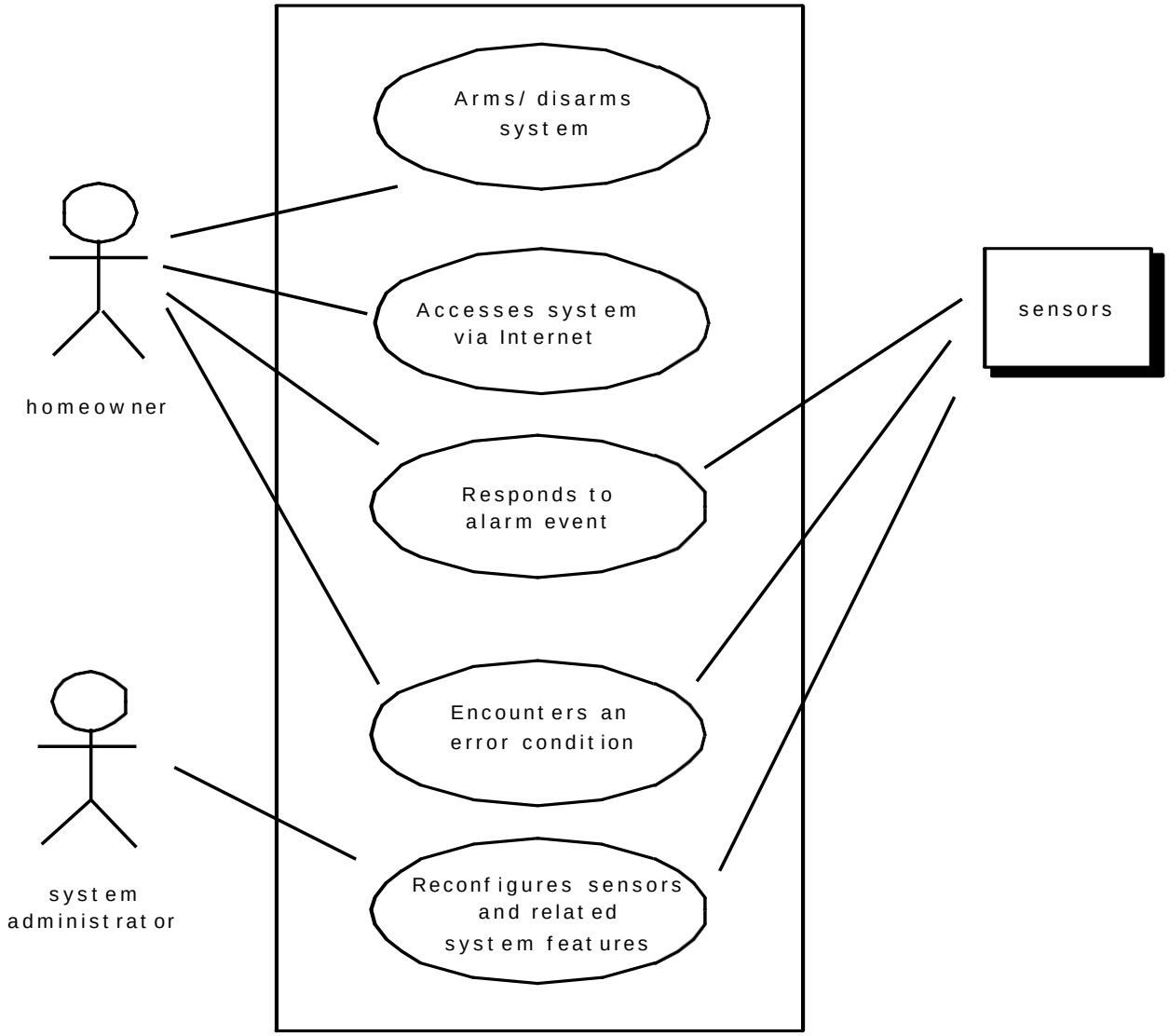




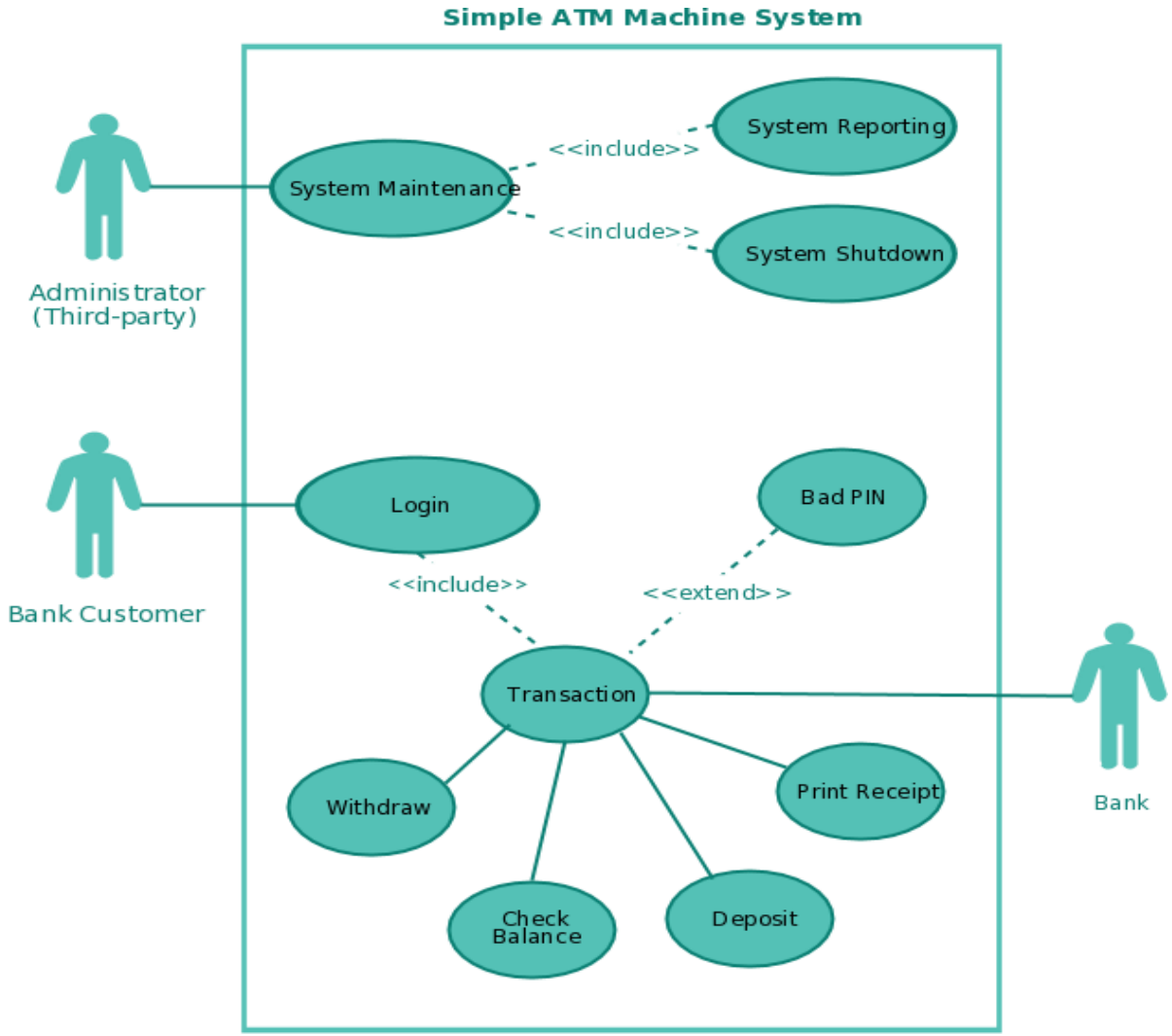
Use-Cases Diagram



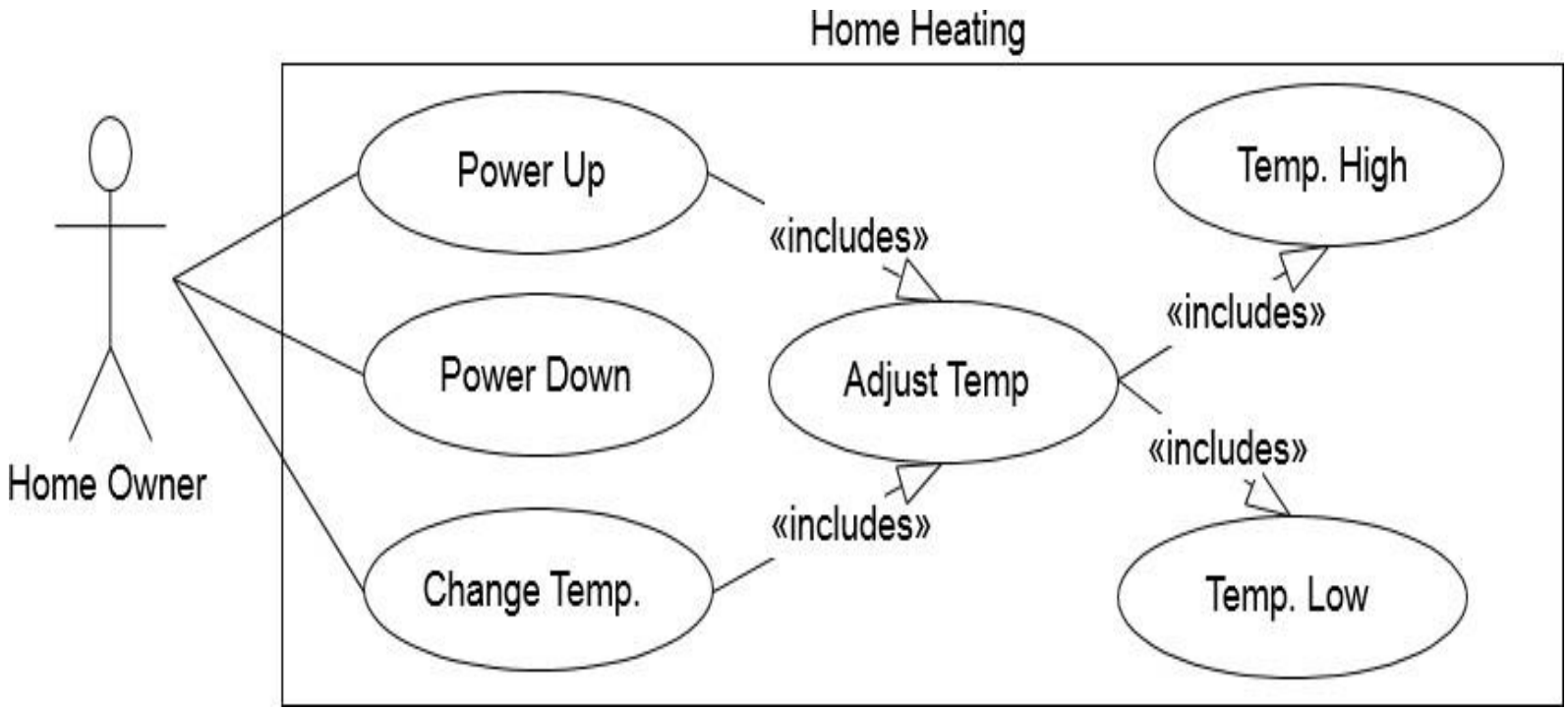
Use-Cases Diagram



Use-Cases Diagram



Home Heating System



Home Heating System

Use Case Description

Use case:	**Power Up**
Actors:	**Home Owner (initiator)**
Type:	Primary and essential
Description:	The Home Owner turns the power on. **Perform Adjust Temp**. If the temperature in all rooms is above the desired temperature, no actions are taken.
Cross Ref.:	Requirements XX, YY, and ZZ
Use-Cases:	Perform Adjust Temp

Home Heating System

Use case: **Adjust Temp**

Actors: **System (initiator)**

Type: Secondary and essential

Description: Check the temperature in each room. For each room:

Below target: **Perform Temp Low**

Above target: **Perform Temp High**

Cross Ref.: Requirements XX, YY, and ZZ

Use-Cases: Temp Low, Temp High

Home Heating System

Use case: **Temp Low**

Actors: **System (initiator)**

Type: Secondary and essential

Description: Open room valve, start pump if not started.
If water temp falls below threshold,
open fuel value and ignite burner.

Cross Ref.: Requirements XX, YY, and ZZ

Use-Cases: None

Use Case Scenario

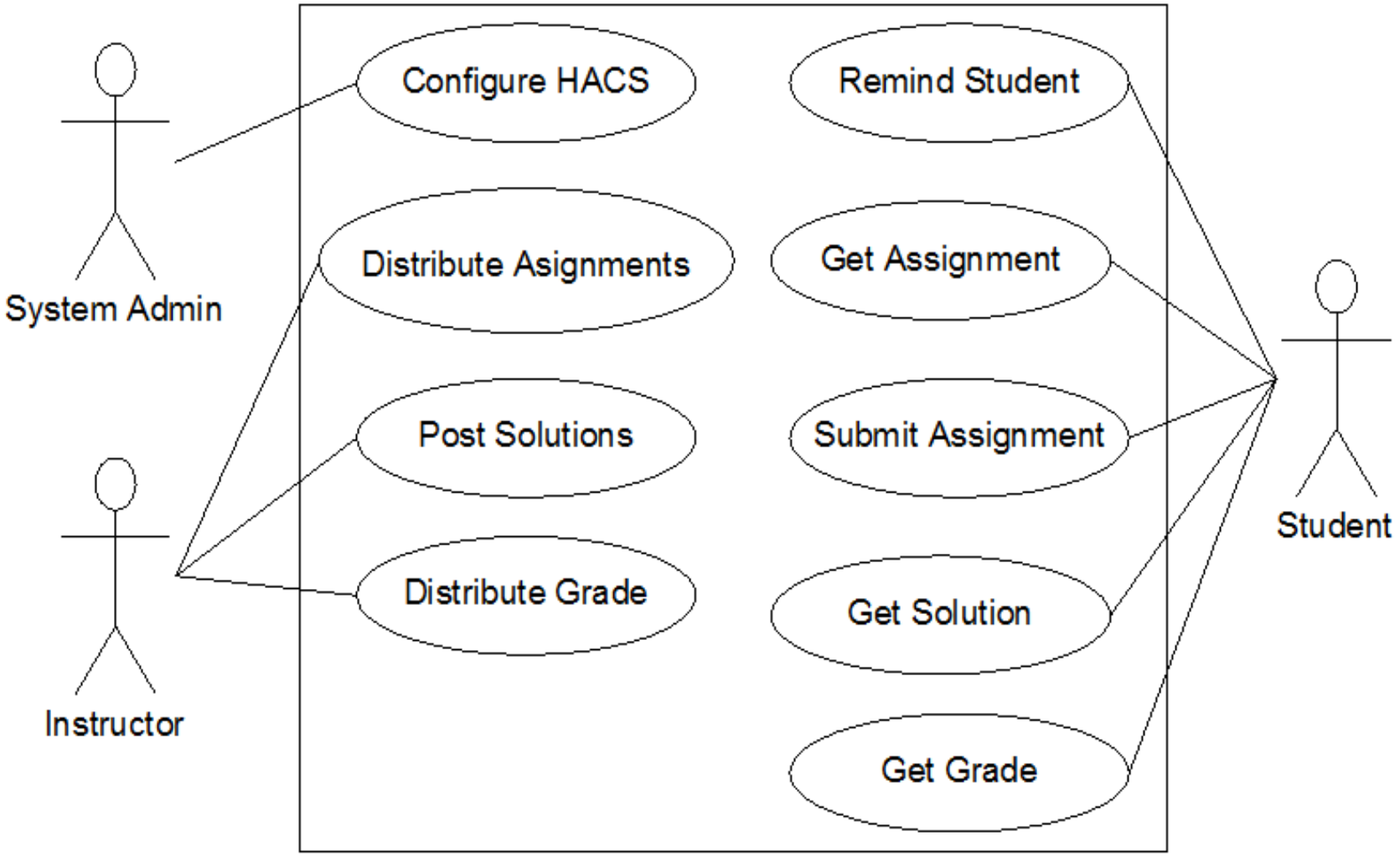
Home Assignment distribution and Collection System(HACS)

Homework assignment and collection are an integral part of any educational system. Today, this task is performed manually. What we want the homework assignment distribution and collection system to do is to automate this process.

The system will be used by the Instructor/Teacher to distribute the homework assignments, review the students' solutions, distribute suggested solution, and distribute student grades on each assignment.

This system will also help the students by automatically distributing the assignments to the students, provide a facility where the students can submit their solutions, remind the students when an assignment is almost due, remind the students when an assignment is overdue.

Home Assignment distribution and Collection System Use Case Diagram

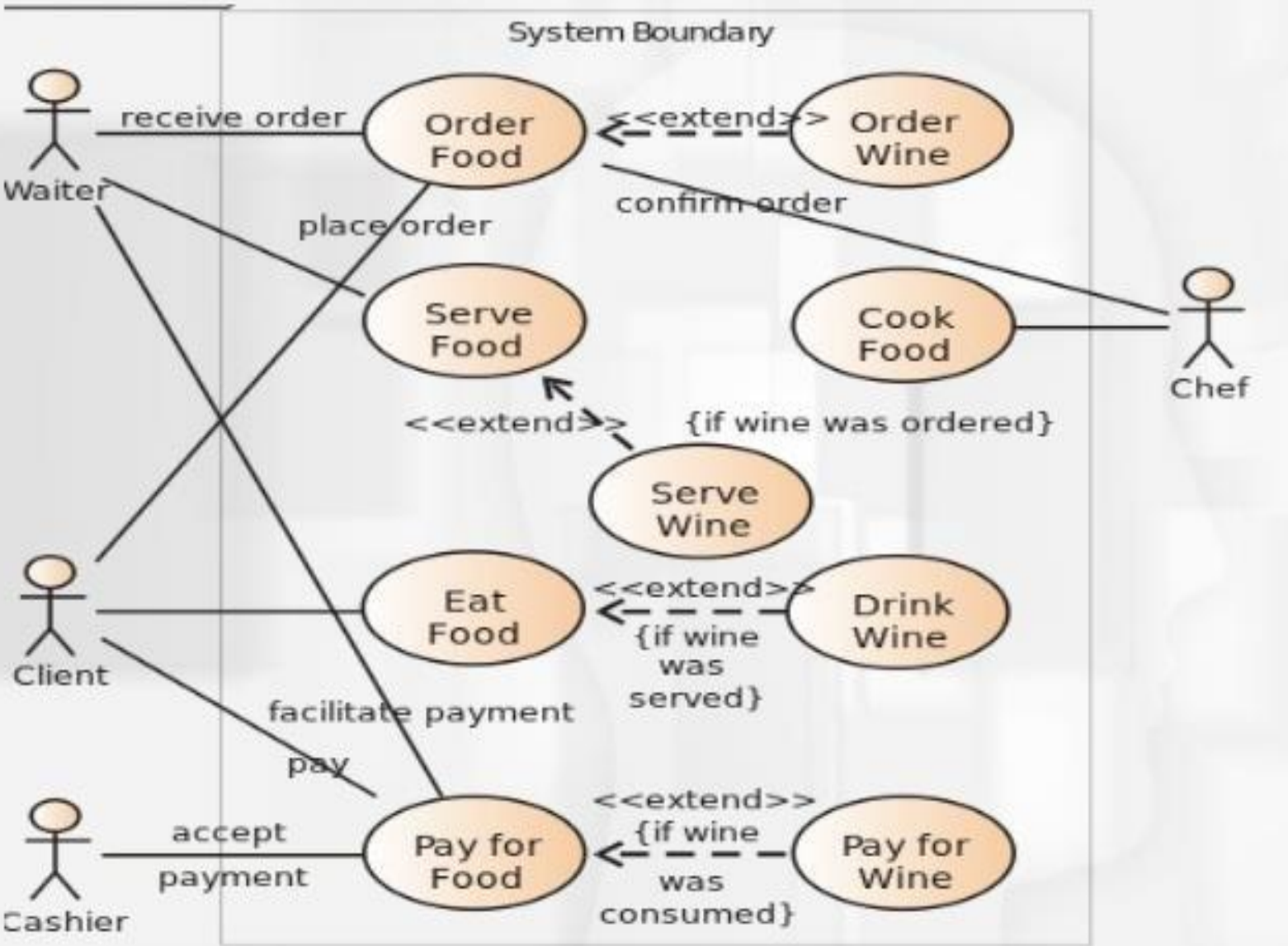


HACS Use Cases Description

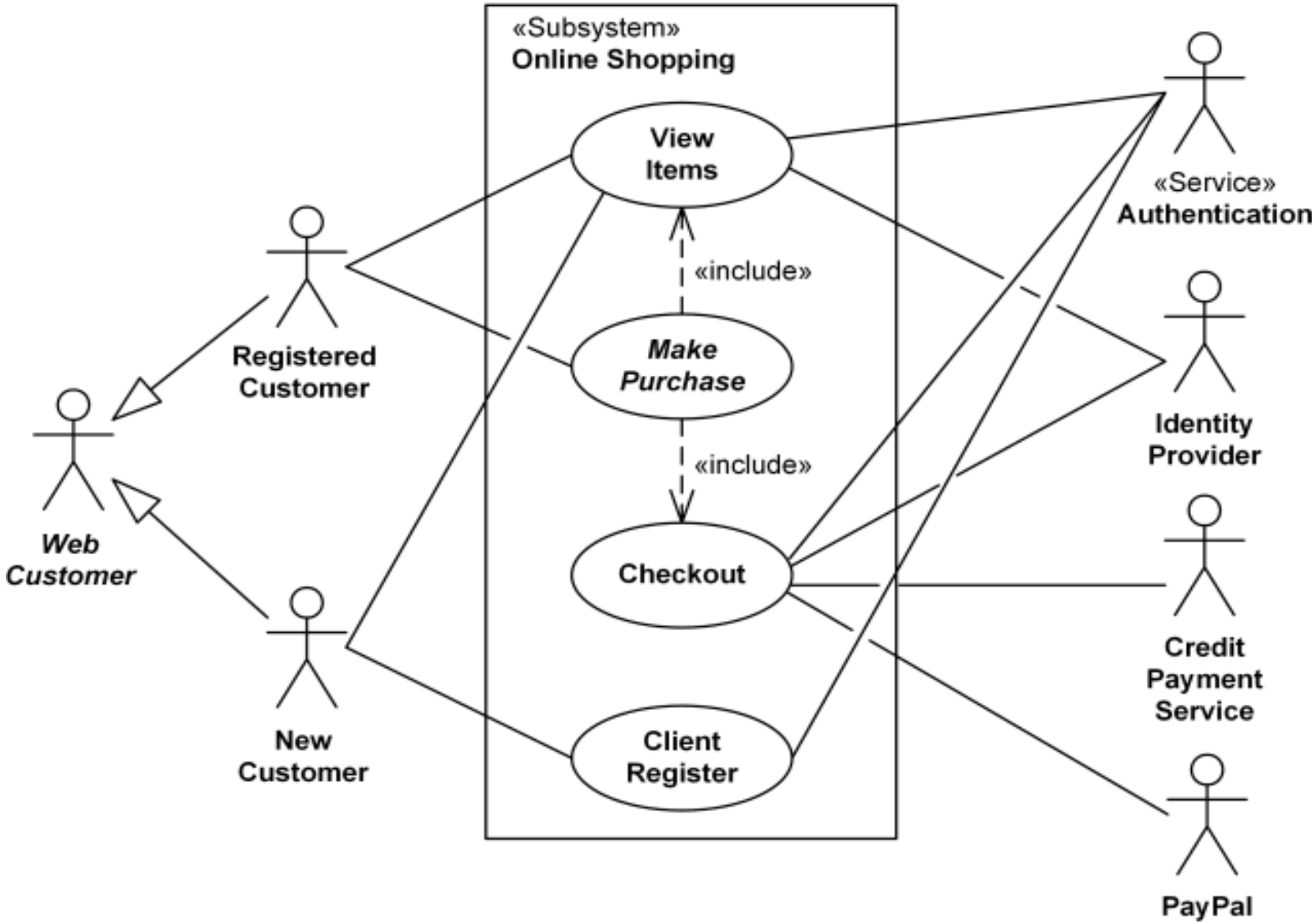
Use case: **Distribute Assignments**
Actors: Instructor (initiator)
Type: Primary and essential
Description: The Instructor completes an assignment and submits it to the system. The instructor will also submit the due date and the class the assignment is assigned for.

Cross Ref.: Requirements XX, YY, and ZZ
Use-Cases: *Configure HACS* must be done before any user (Instructor or Student) can use HACS

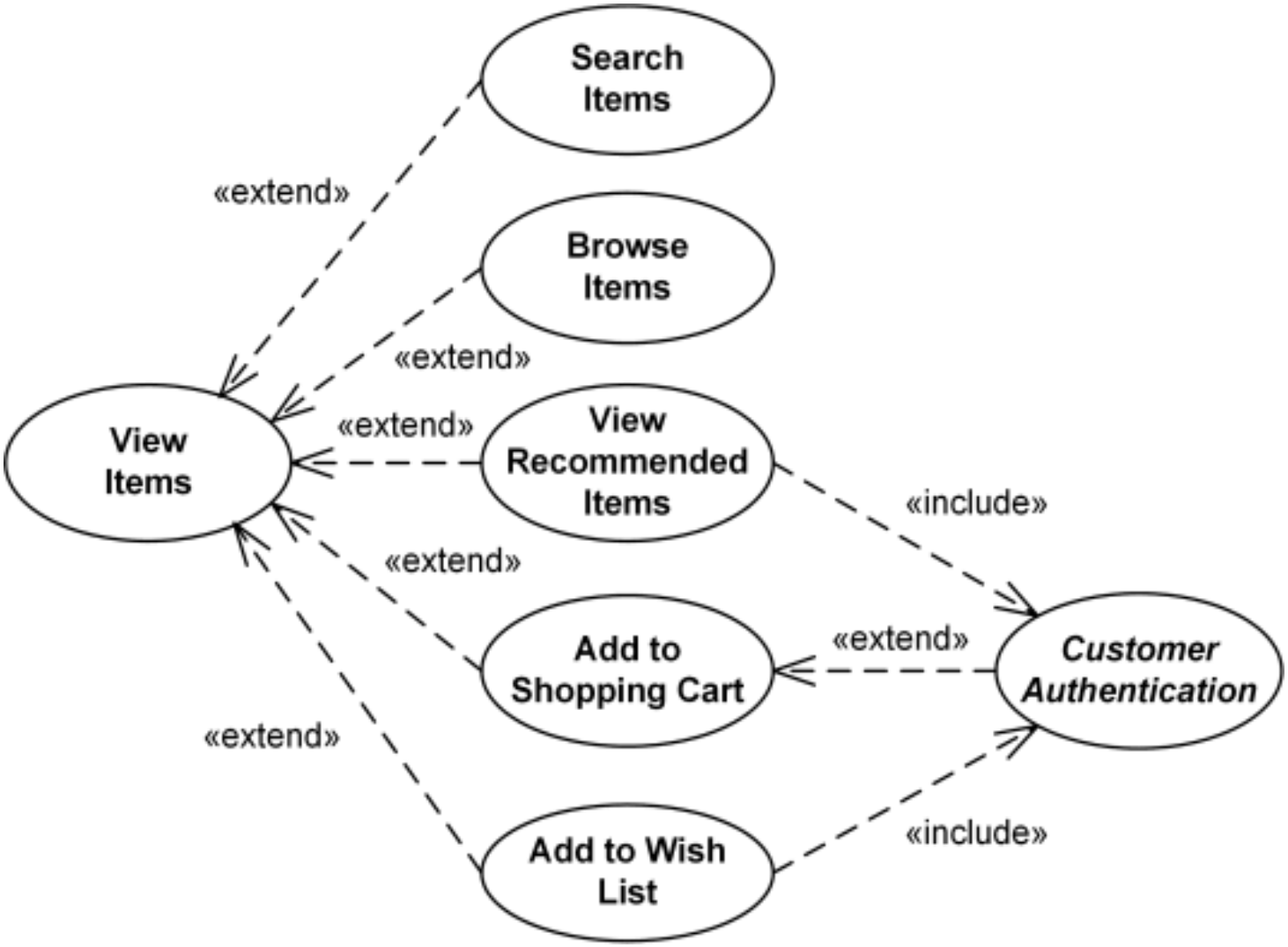
Use Cases Diagram for Restaurant



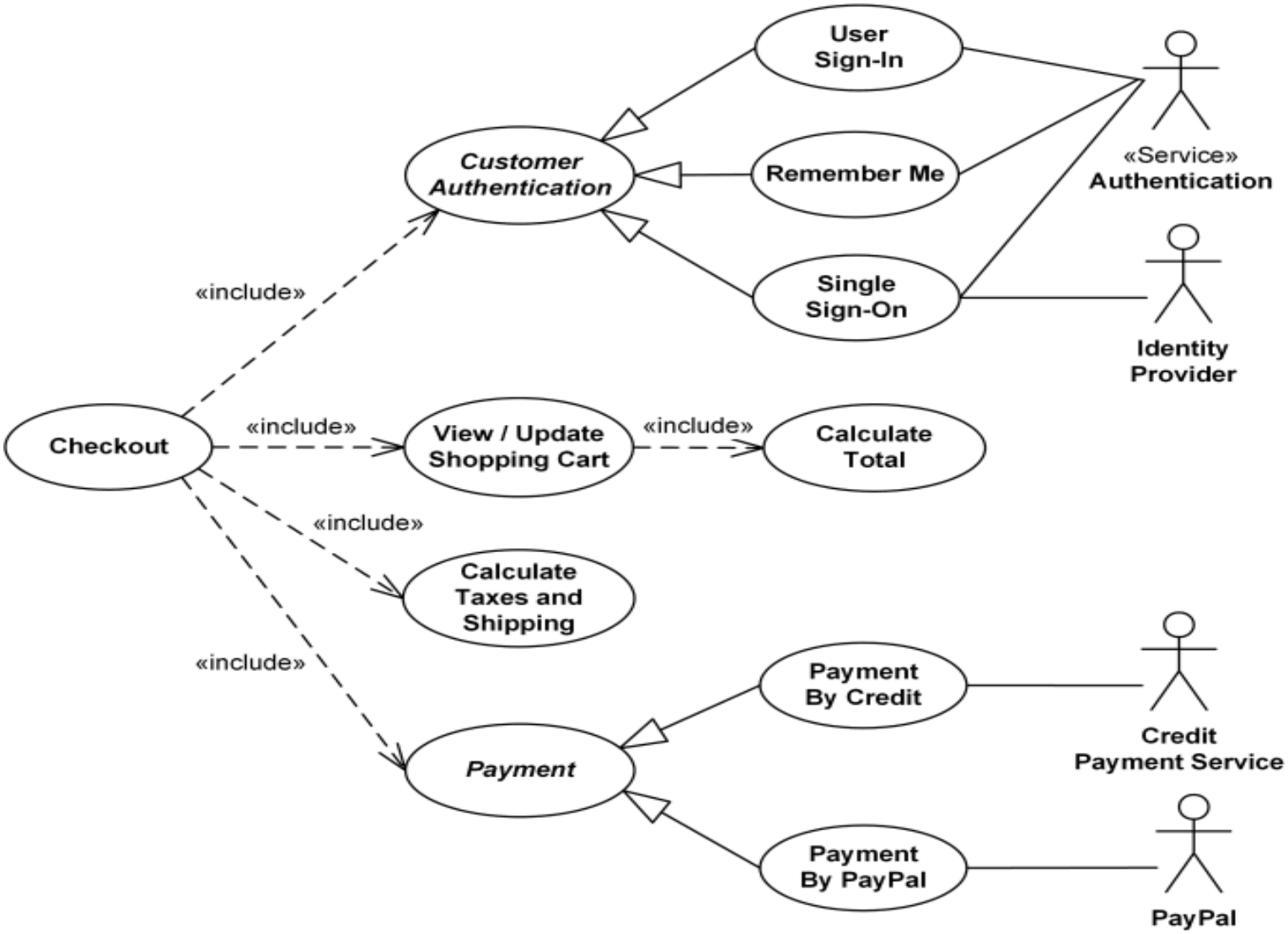
Use Case Diagram Online Shopping



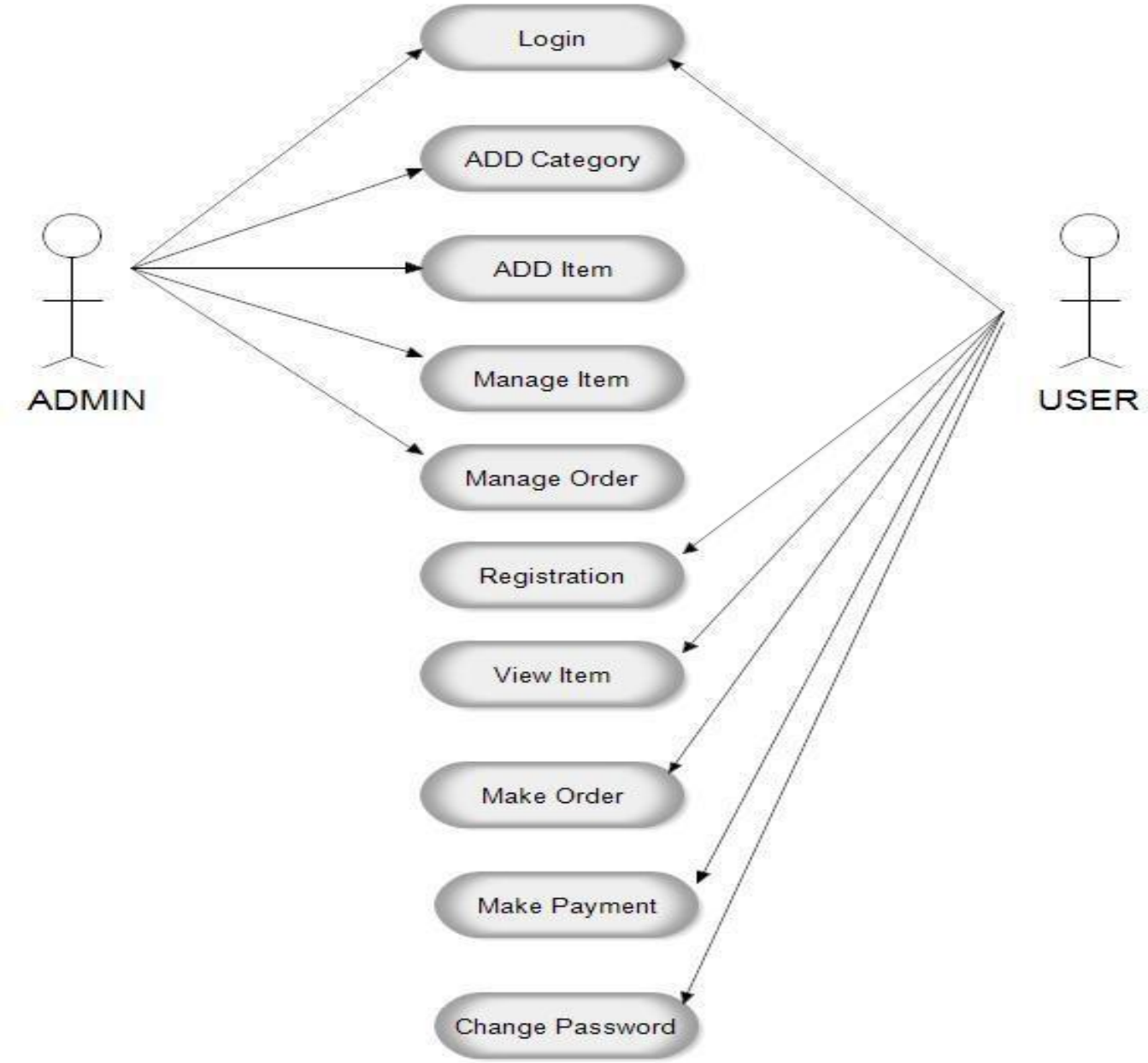
Use Case Diagram- View Items



Use Case Diagram - Checkout



Use Case Diagram for Online Shopping Website



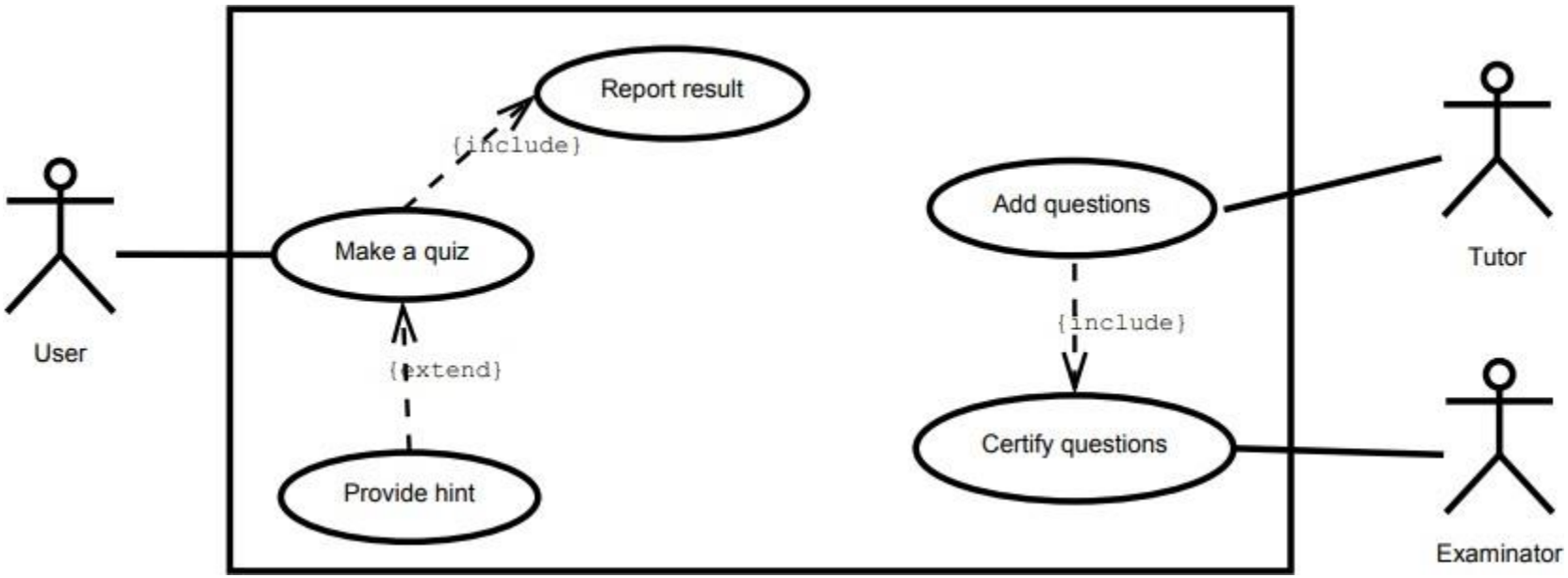
Example-1: Use Case Scenario

A user can request a quiz for the system. The system picks a set of questions from its database, and compose them together to make a quiz. It rates the user's answers, and gives hints if the user requests it.

In addition to users, we also have tutors who provide questions and hints. And also examinations who must certify questions to make sure they are not too trivial, and that they are sensual.

Make a use case diagram to model this system. Work out some of your use cases. Since we don't have real stake holders here, you are free to fill in details you think is sensual for this example.

Use Case Diagram



Use Case Description

Use case: Make quiz.
Primary actor: User
Secondary actors: -
Pre-condition: The system has at least 10 questions.
Post-condition: -
Main flow:
1. The use-case is activated when the user requests it.
2. The user specifies the difficulty level.
3. The system selects 10 questions, and offers them as a quiz to the user.
4. The system starts a timer.
5. For every question:
   5a. The user selects an answer, or skip. [Extension point]
6. If the user is done with the quiz, or the timer runs out, the quiz is concluded, and [include use case 'Report result'].

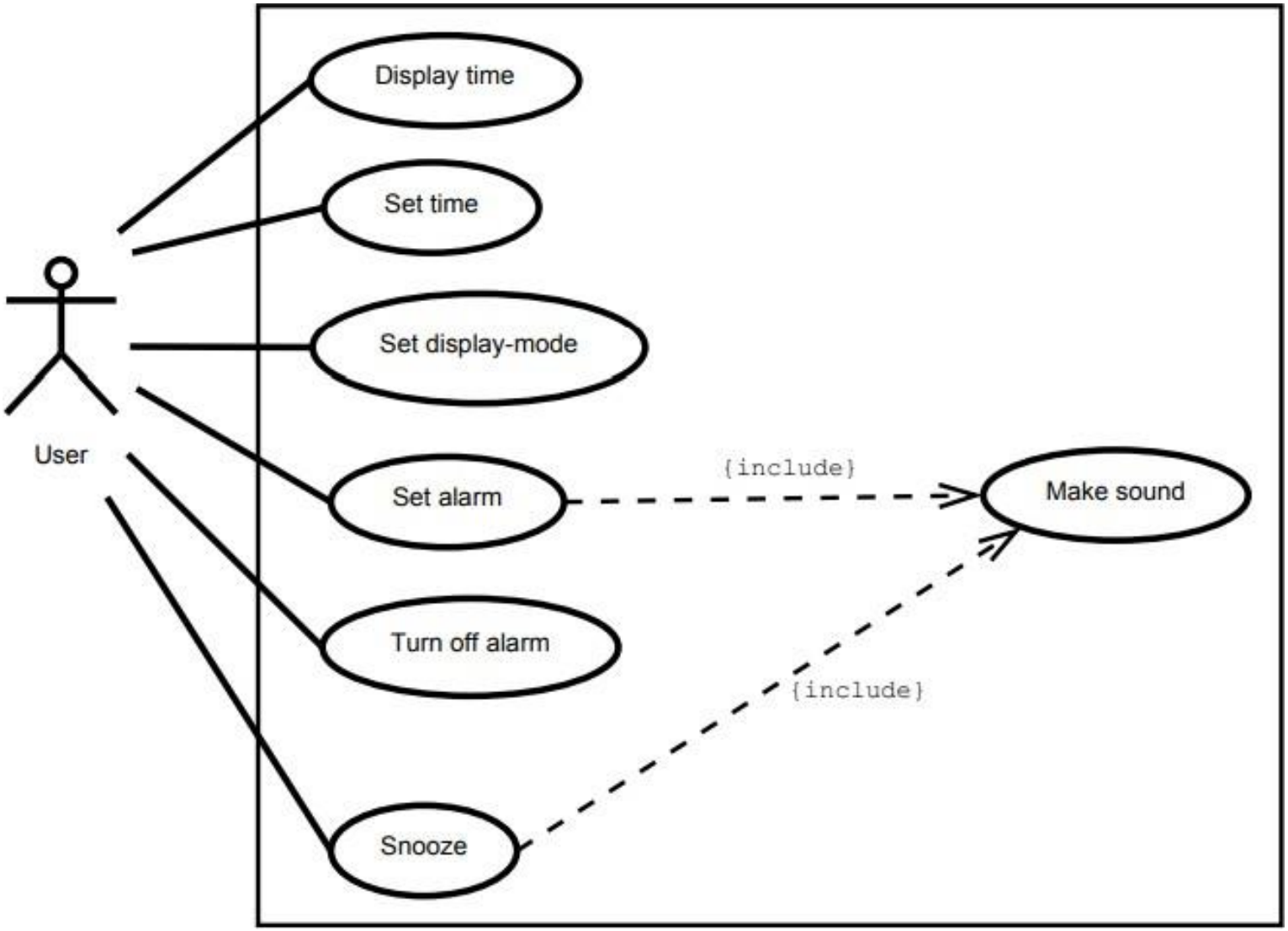
Example-2: Use Case Scenario

Suppose we want to develop software for an alarm clock.

The clock shows the time of day. Using buttons, the user can set the hours and minutes fields individually, and choose between 12 and 24-hour display.

It is possible to set one or two alarms. When an alarm fires, it will sound some noise. The user can turn it off, or choose to ’snooze’. If the user does not respond at all, the alarm will turn off itself after 2 minutes. ’Snoozing’ means to turn off the sound, but the alarm will fire again after some minutes of delay. This ’snoozing time’ is pre-adjustable.

Use Case Diagram



Use Case Description

Use case: Snooze.
Primary actor: User
Secondary actors: -
Pre-condition: An alarm is firing.
Post-condition: -
Main flow:
1. The use-case is activated when the user hits the snooze button.
2. The alarm is turned off.
3. Wait for snooze time.
4. Include use case 'Make sound'

Use Case Description

Use case: Snooze.
Primary actor: User
Secondary actors: -
Pre-condition: An alarm is firing.
Post-condition: -
Main flow:
1. The use-case is activated when the user hits the snooze button.
2. The alarm is turned off.
3. Wait for snooze time.
4. Include use case 'Make sound'

References:

1. **Software Engineering A practitioner's Approach** by Roger S. Pressman, 7th edition, McGraw Hill, 2010.
2. **Software Engineering by Ian Sommerville,** 9th edition, Addison-Wesley, 2011
3. **FUNCTIONAL VS NON-FUNCTIONAL REQUIREMENTS: MAIN DIFFERENCES & EXAMPLES** https://theappsolutions.com/blog/development/functional-vs-non-functional-requirements/

ABES
Estd. 2000
ABES
Engineering College
College Code-032

OBJECT ORIENTED SYSTEM DESIGN(OOSD)

SESSION 2023-24

MAPPING OBJECT ORIENTED CONCEPTS USING NON-OBJECT ORIENTED LANGUAGE

- A use case diagram is used to represent the dynamic behaviour of a system.
- It depicts the high-level functionality of a system and also tells how the user handles a system.
- It encapsulates the system's functionality by incorporating use cases, actors, and their relationships.
- It models the tasks, services, and functions required by a system/subsystem of an application.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- A key concept of use case modeling is that it helps us design a system from the end user's perspective.
- It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.

UML USE CASE DIAGRAM

A use case diagram is usually simple. It does not show the detail of the use cases:

- It only summarizes **some of the relationships** between use cases, actors, and systems.
- It does **not show the order** in which steps are performed to achieve the goals of each use case.

As said, a use case diagram should be simple and contains only a few shapes. If yours contain more than 20 use cases, you are probably misusing use case diagram.

PURPOSE OF USE CASE DIAGRAMS

- It gathers the system's needs.
- It depicts the external view of the system.
- It recognizes the internal as well as external factors that influence the system.
- It represents the interaction between the actors

WHY USE-CASE DIAGRAM?

- A Use Case consists of use cases, persons, or various things that are invoking the features called as actors and the elements that are responsible for implementing the use cases.
- Use case diagrams capture the dynamic behaviour of a live system.
- It drives implementation and helps in generating test cases
- It models how an external entity interacts with the system to make it work. Use case diagrams are responsible for visualizing the external things that interact with the part of the system.

USE OF A USE-CASE DIAGRAM

In general use case diagrams are used for:

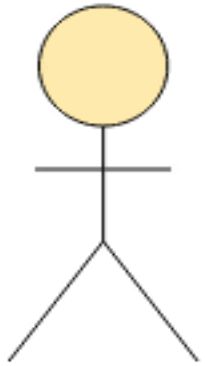
- Analyzing the requirements of a system
- High-level visual software designing
- Capturing the functionalities of a system
- Modeling the basic idea behind the system
- Forward and reverse engineering of a system using various test cases.

USE-CASE DIAGRAM NOTATIONS

ABES Estd. 2000
ABES Engineering College
College Code-032

Actor

- Someone interacts with use case (system function).
- Named by noun.
- Actor plays a role in the business
- Similar to the concept of user, but a user can play different roles
- For example:
  - A prof. can be instructor and also researcher
  - plays 2 roles with two systems
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).



USE-CASE DIAGRAM NOTATIONS

Use Case

- A use case represents a distinct functionality of a system
- Named by verb + Noun (or Noun Phrase).
- i.e. Do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.
- It is denoted by an oval shape with the name of a use case written inside the oval shape.

USE-CASE DIAGRAM NOTATIONS

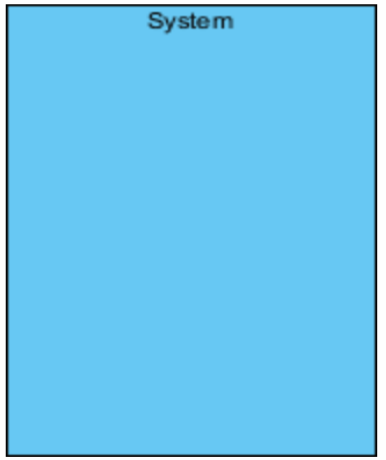
Communication Link

- The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.
- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.

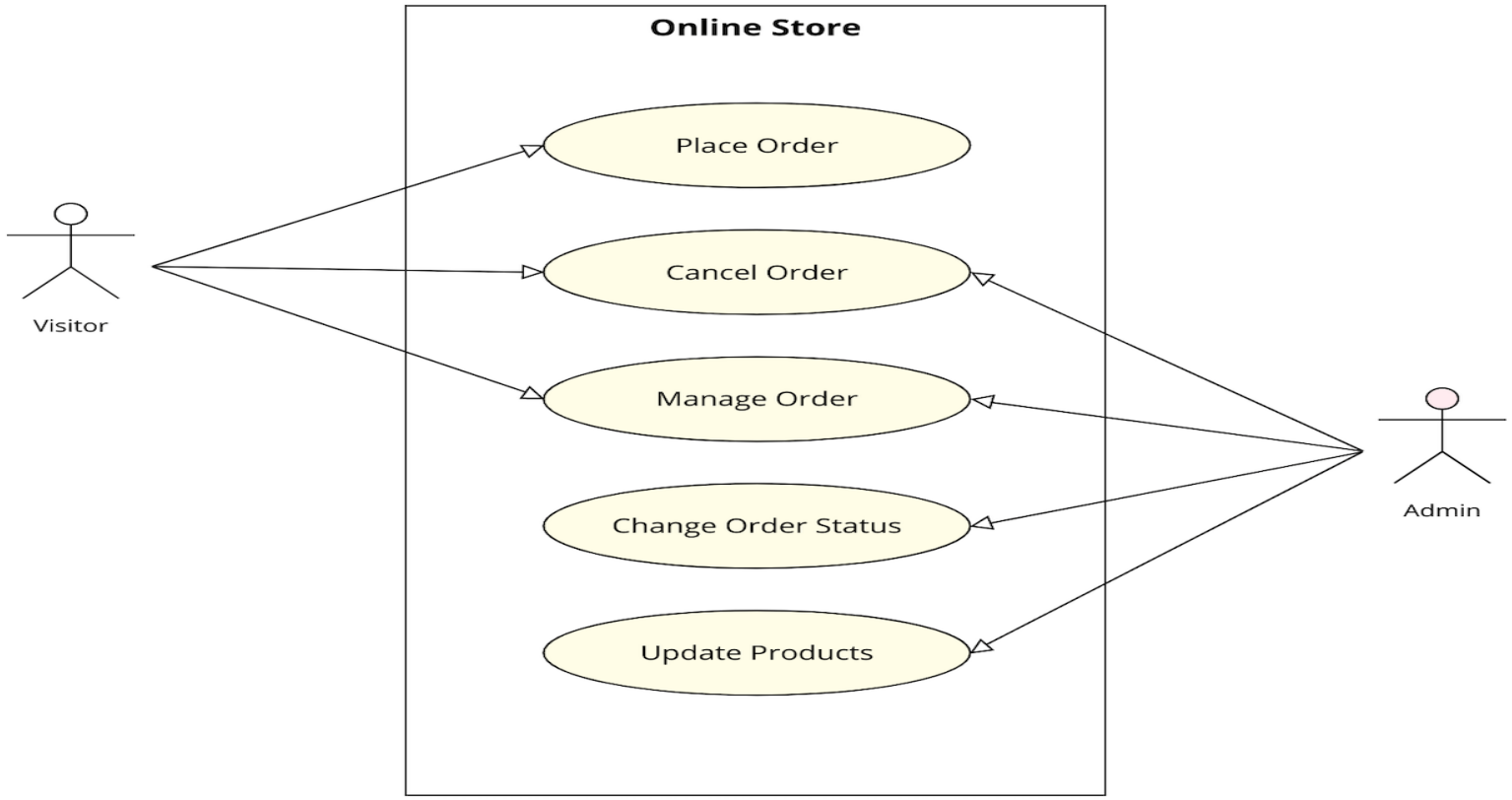
USE-CASE DIAGRAM NOTATIONS

Boundary of system

- The system boundary is potentially the entire system as defined in the requirements document.
- For large and complex systems, each module may be the system boundary.
- For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc.
- can form a system boundary for use cases specific to each of these business functions.
- The entire system can span all of these modules depicting the overall system boundary

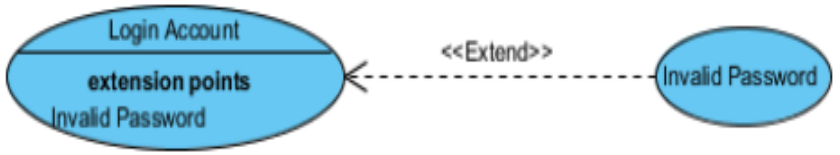


A STANDARD USE CASE DIAGRAM



USE CASE RELATIONSHIP

Extends

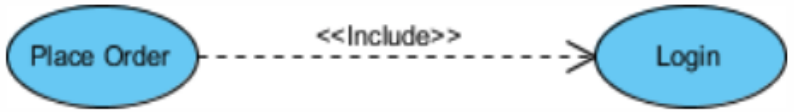


- Indicates that an **"Invalid Password"** use case may include (subject to specified in the extension) the behavior specified by base use case **"Login Account"**.
- Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
- The stereotype "<<extends>>" identifies as an extend relationship

USE CASE RELATIONSHIP

Include

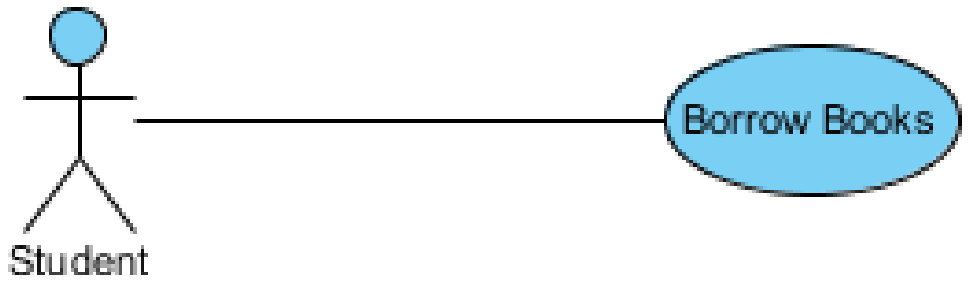
- When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship.



- A use case includes the functionality described in another use case as a part of its business process flow.
- A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow.
- The stereotype "<<include>>" identifies the relationship as an include relationship.

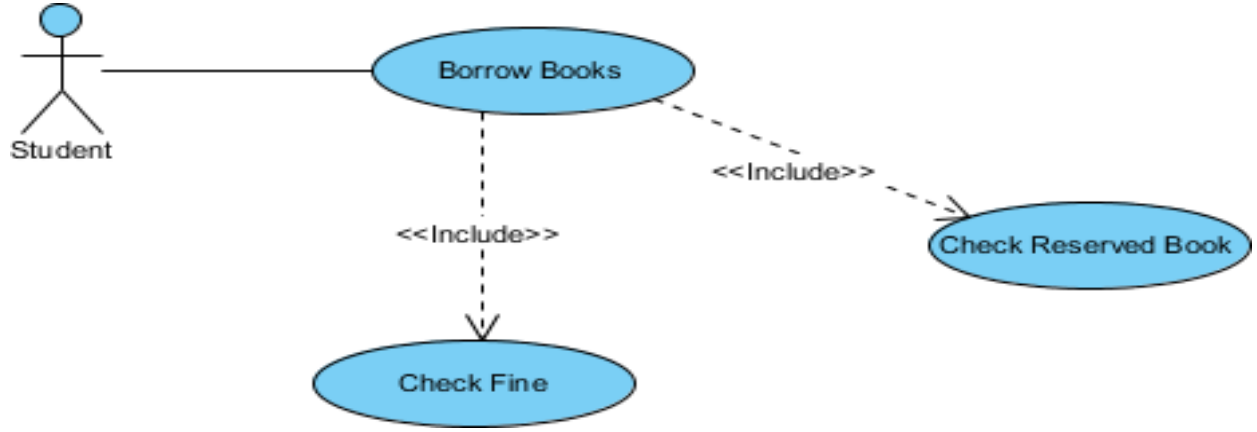
USE CASE EXAMPLE - ASSOCIATION LINK

A Use Case diagram illustrates a set of use cases for a system, i.e. the actors and the relationships between the actors and use cases.



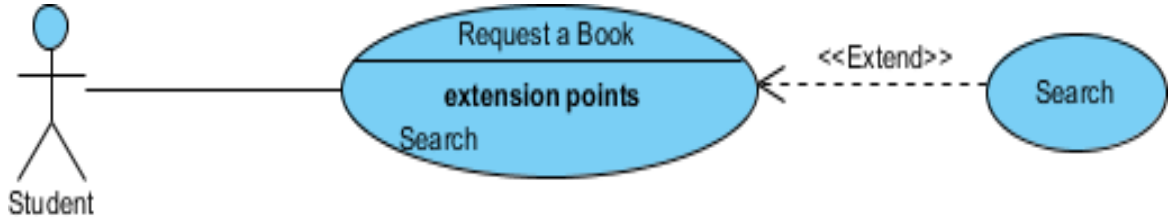
USE CASE EXAMPLE - INCLUDE RELATIONSHIP

The include relationship adds additional functionality not specified in the base use case. The <<Include>> relationship is used to include common behavior from an included use case into a base use case in order to support the reuse of common behavior.



USE CASE EXAMPLE - EXTEND RELATIONSHIP

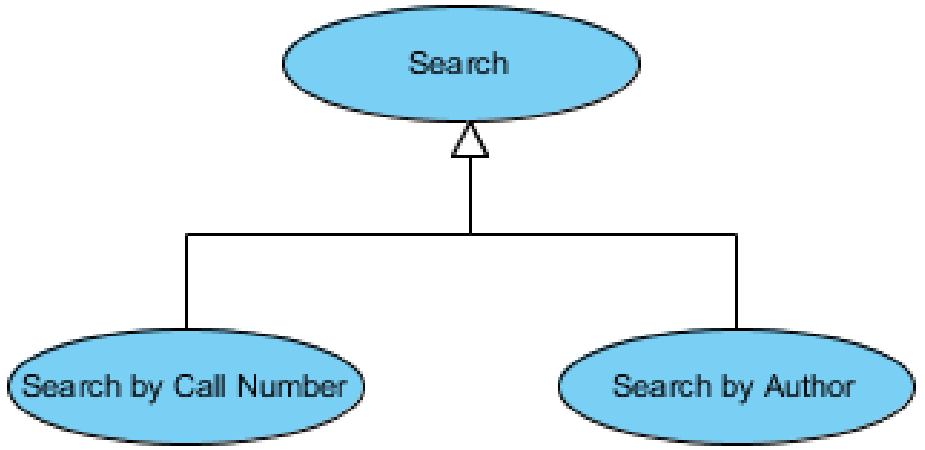
The extend relationships are important because they show optional functionality or system behavior. The <<extend>> relationship is used to include optional behavior from an extending use case in an extended use case. Take a look at the use case diagram example below. It shows an extend connector and an extension point "Search".



USE CASE EXAMPLE - GENERALIZATION RELATIONSHIP

Use Case Example - Generalization Relationship

- A generalization relationship means that a child use case inherits the behavior and meaning of the parent use case. The child may add or override the behavior of the parent. The figure below provides a use case example by showing two generalization connectors that connect between the three use cases.



RULES TO BE FOLLOWED

Following rules must be followed while drawing use-case for any system:

- The name of an actor or a use case must be meaningful and relevant to the system.
- Interaction of an actor with the use case must be defined clearly and in an understandable way.
- Annotations must be used wherever they are required.
- If a use case or an actor has multiple relationships, then only significant interactions must be displayed.

HOW TO IDENTIFY ACTOR

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a present time?

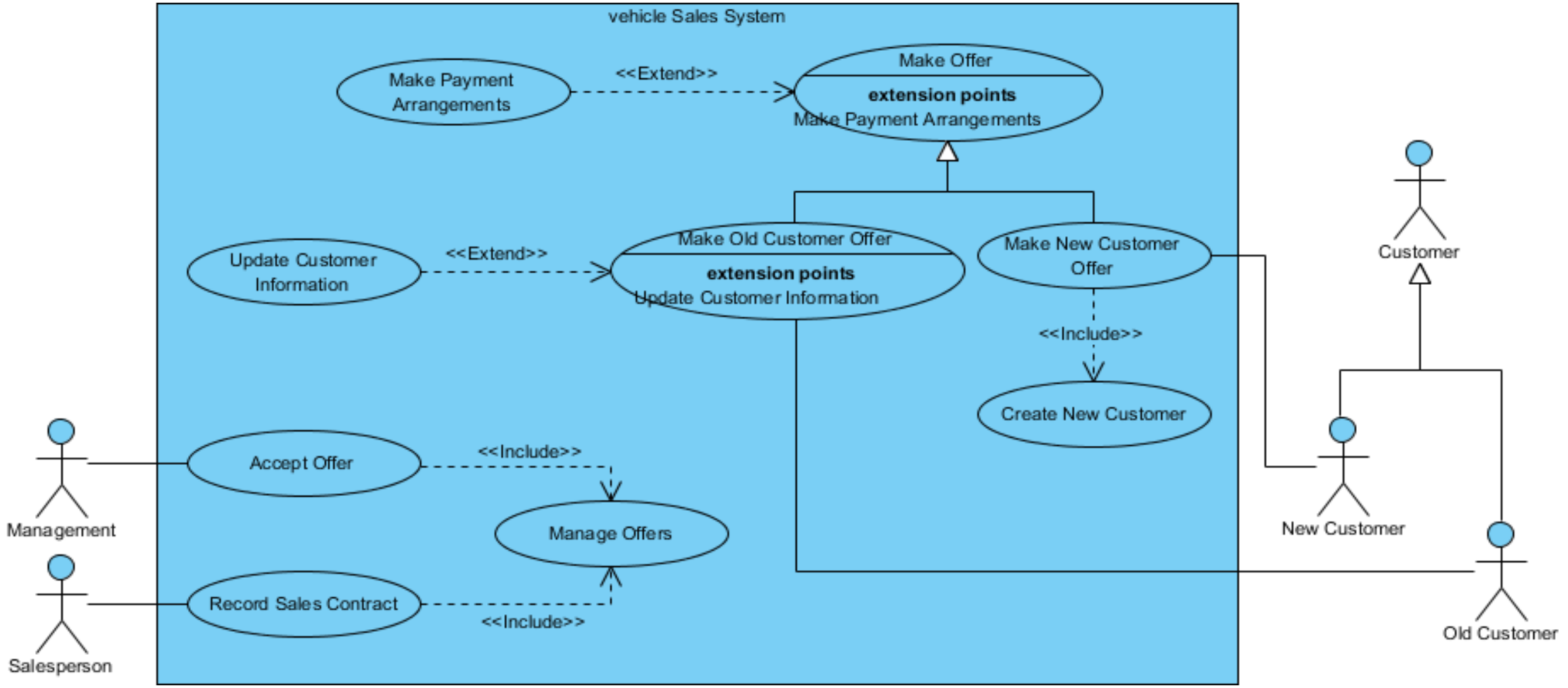
HOW TO IDENTIFY USE CASES?

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about changes in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?

USE CASE DIAGRAM TIPS

- Always structure and organize the use case diagram from the perspective of actors.
- Use cases should start off simple and at the highest view possible. Only then can they be refined and detailed further.
- Use case diagrams are based upon functionality and thus should focus on the "what" and not the "how".

USE CASE DIAGRAM - VEHICLE SALES SYSTEMS



Interaction Diagram :
Sequence diagram Collaboration Diagram

Interaction Diagram :

- From the name *Interaction* it is clear that the diagram is used to describe some type of interactions among the different elements in the model.
- So this interaction is a part of dynamic behavior of the system.

- This interactive behavior is represented in UML by two diagrams known as *Sequence diagram* and *Collaboration diagram*.

- The basic purposes of both the diagrams are similar.

- Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

- The purposes of interaction diagram can be describes as:

  To capture dynamic behavior of a system.

  To describe the message flow in the system.

  To describe structural organization of the objects.

  To describe interaction among objects.

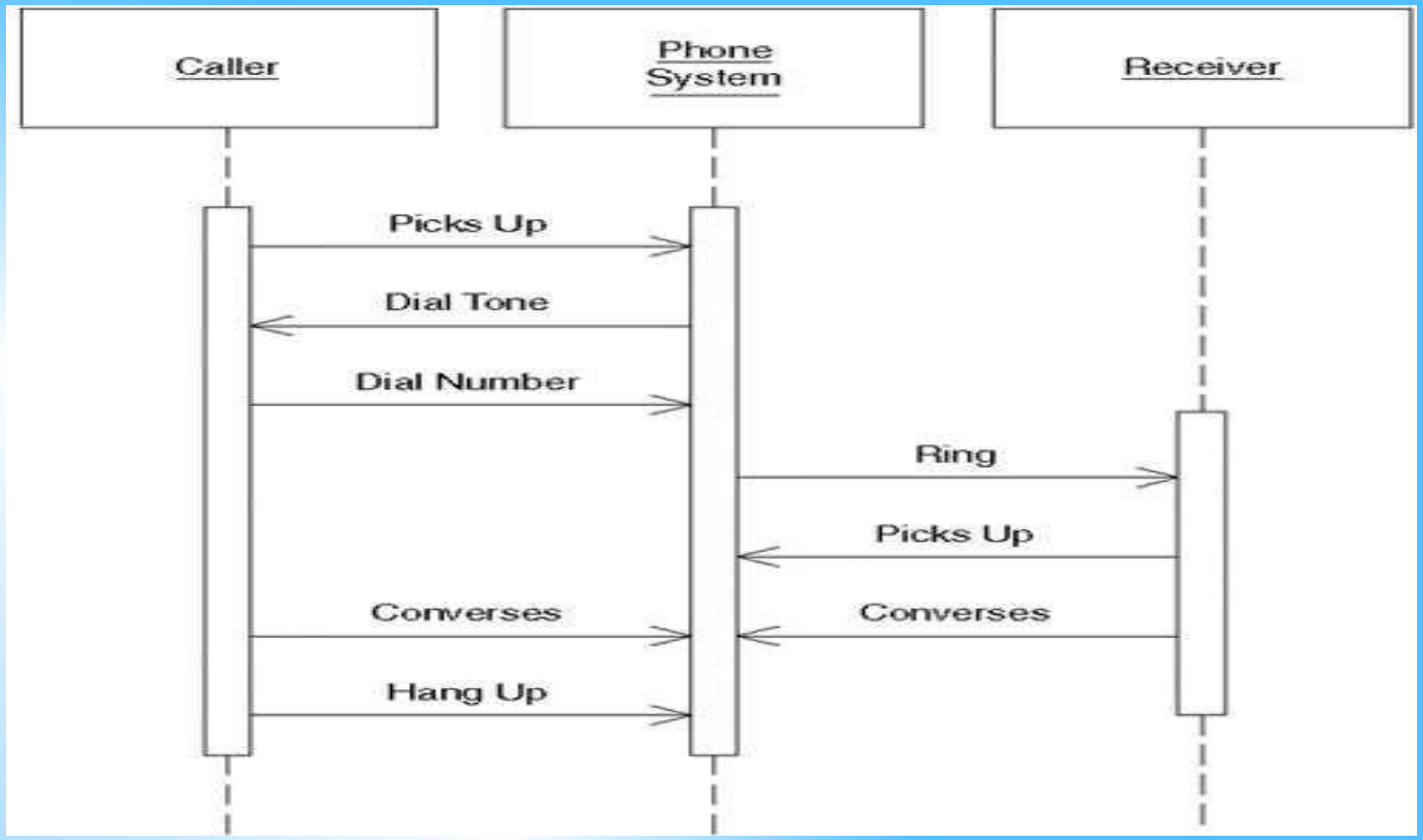
The following factors are to be identified clearly before drawing the interaction diagram:

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

Example :

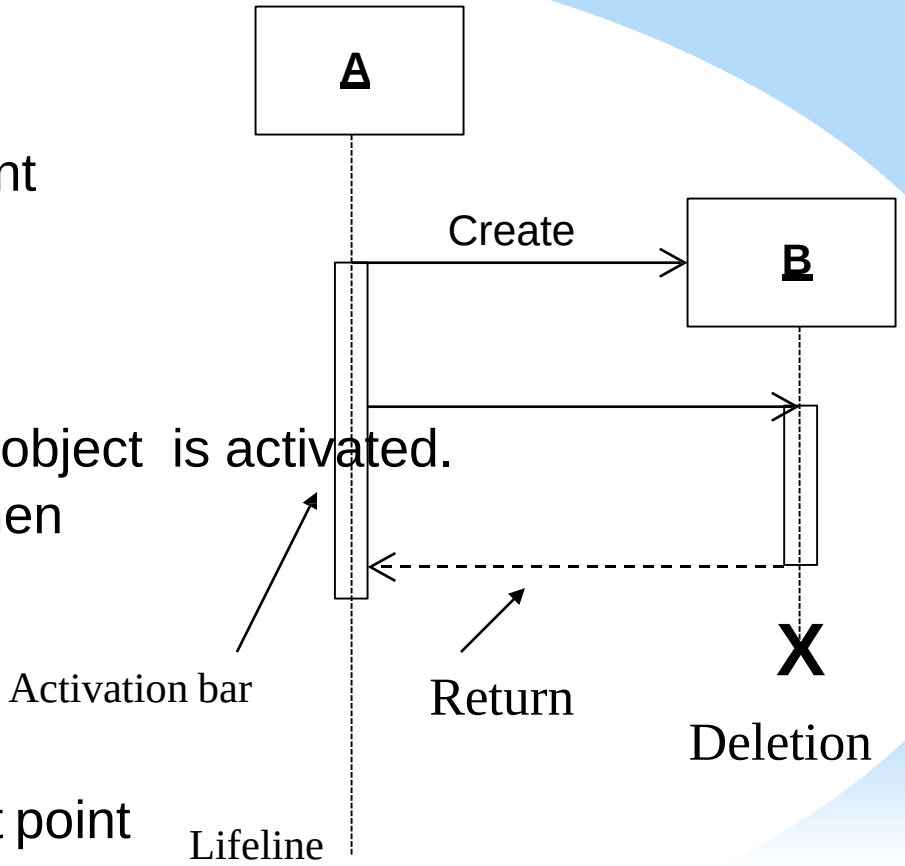
Making a phone call.

Sequence Diagram(Telephone call)



Sequence Diagrams – Object Life Spans

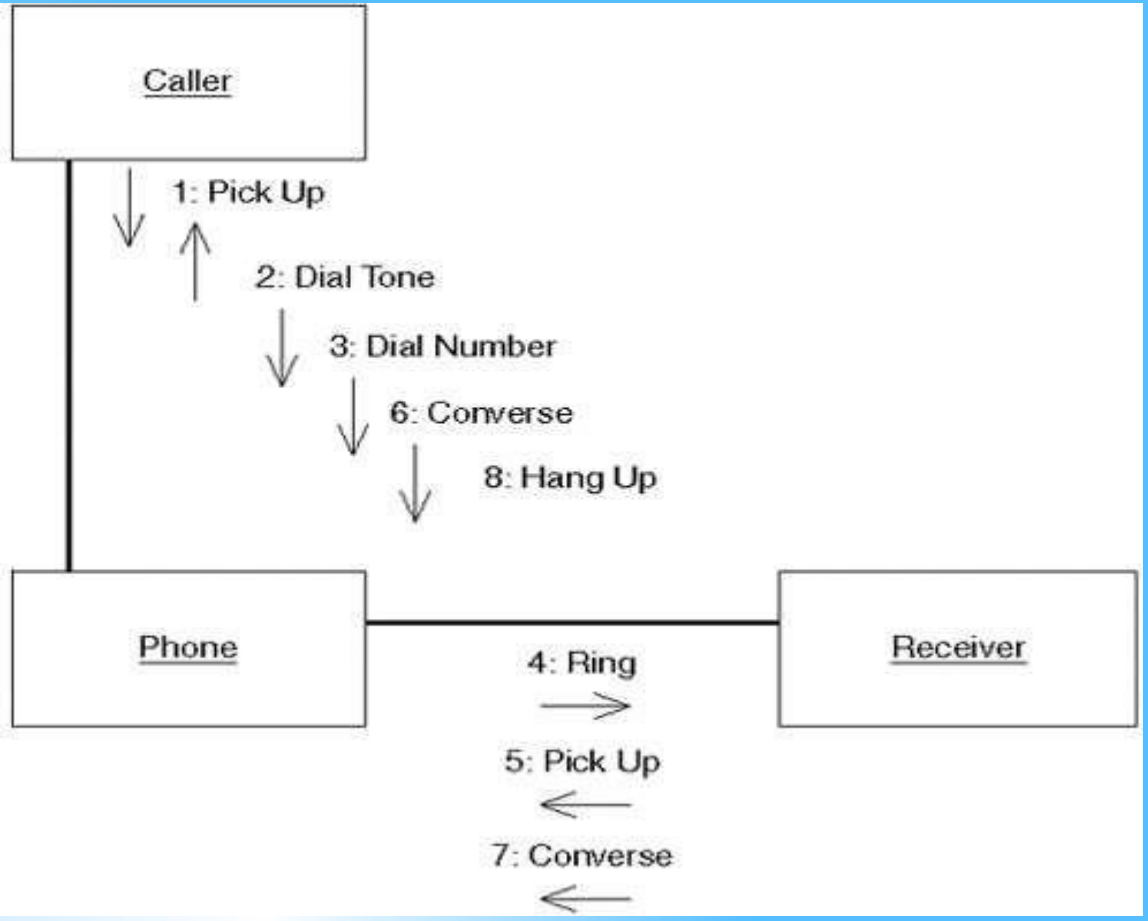
- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point



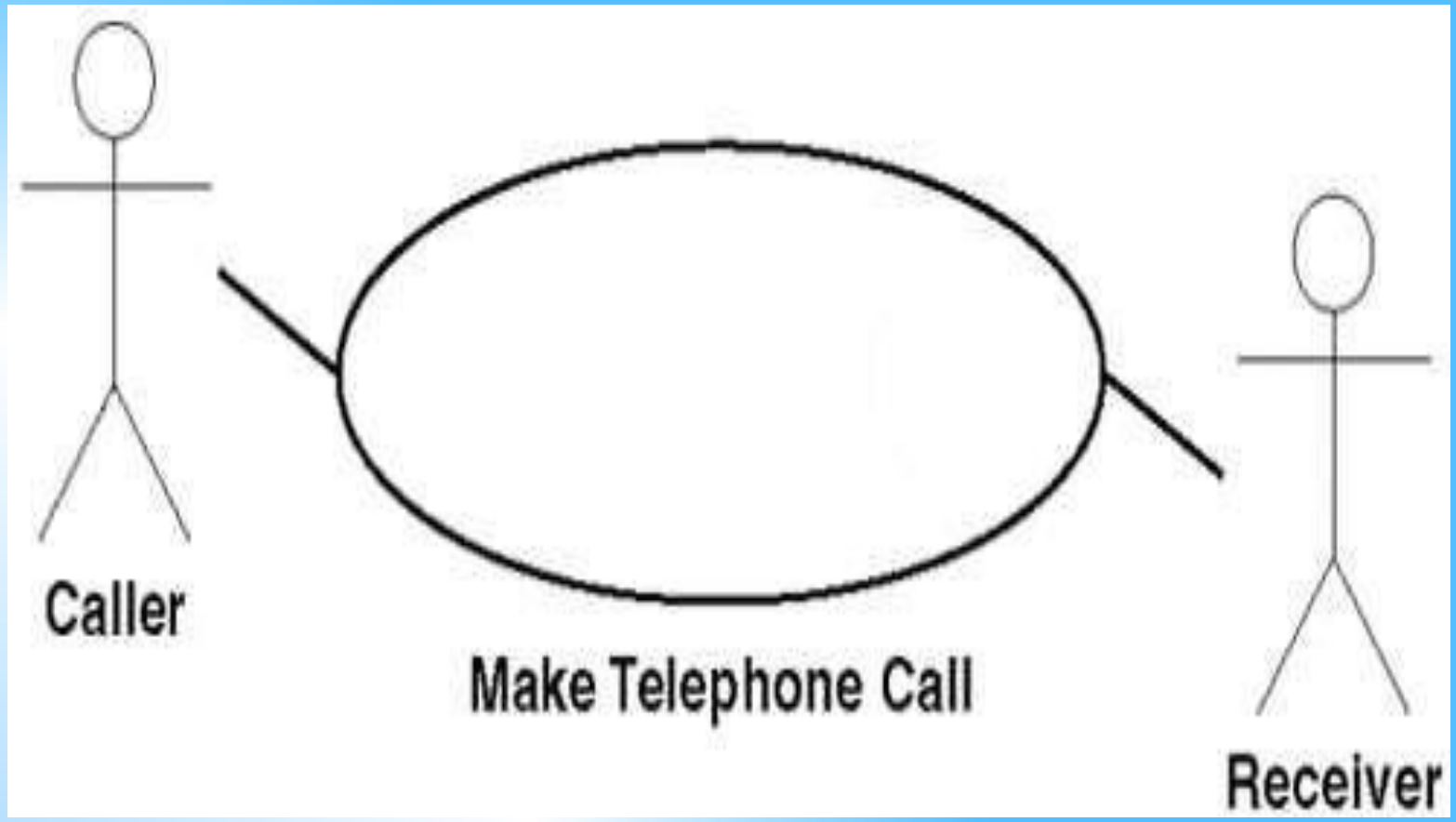
Collaboration Diagram

- A *collaboration diagram* also shows the passing of messages between objects, but focuses on the objects and messages and their order instead of the time sequence.
- The sequence of interactions and the concurrent threads are identified using sequence numbers.
- A collaboration diagram shows an interaction the relationships among the objects playing the different roles.
- The *UML Specification* suggests that collaboration diagrams are better for real-time specifications and for complex scenarios than sequence diagrams.

Collaboration Diagram(Telephone call)



Use Case diagram –Telephone Call



PACKAGE DAIGRAM

- Package diagrams are structural diagrams used to show the organization and arrangement of various model elements in the form of packages.

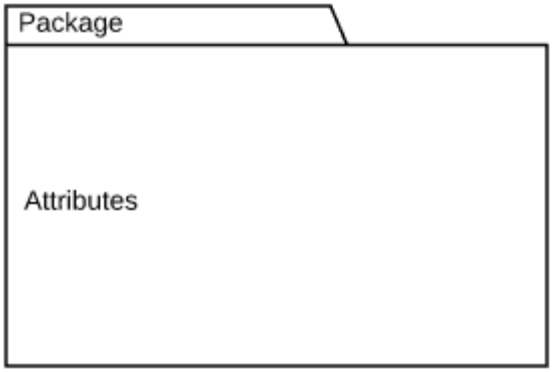
- A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder within the diagram, then arranged hierarchically within the diagram.

Benefits of Package Diagram

- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.

BASIC COMPONENTS

- The makeup of a package diagram is relatively simple. Each diagram includes only symbols:



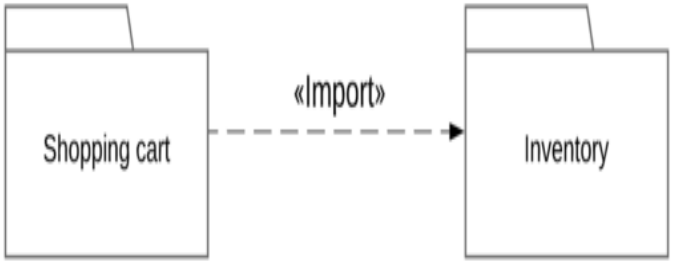
PACKAGE : Groups common elements based on data, behavior, or user interaction

DEPENDENCY :

Depicts the relationship between one element (package, named element, etc.) and another

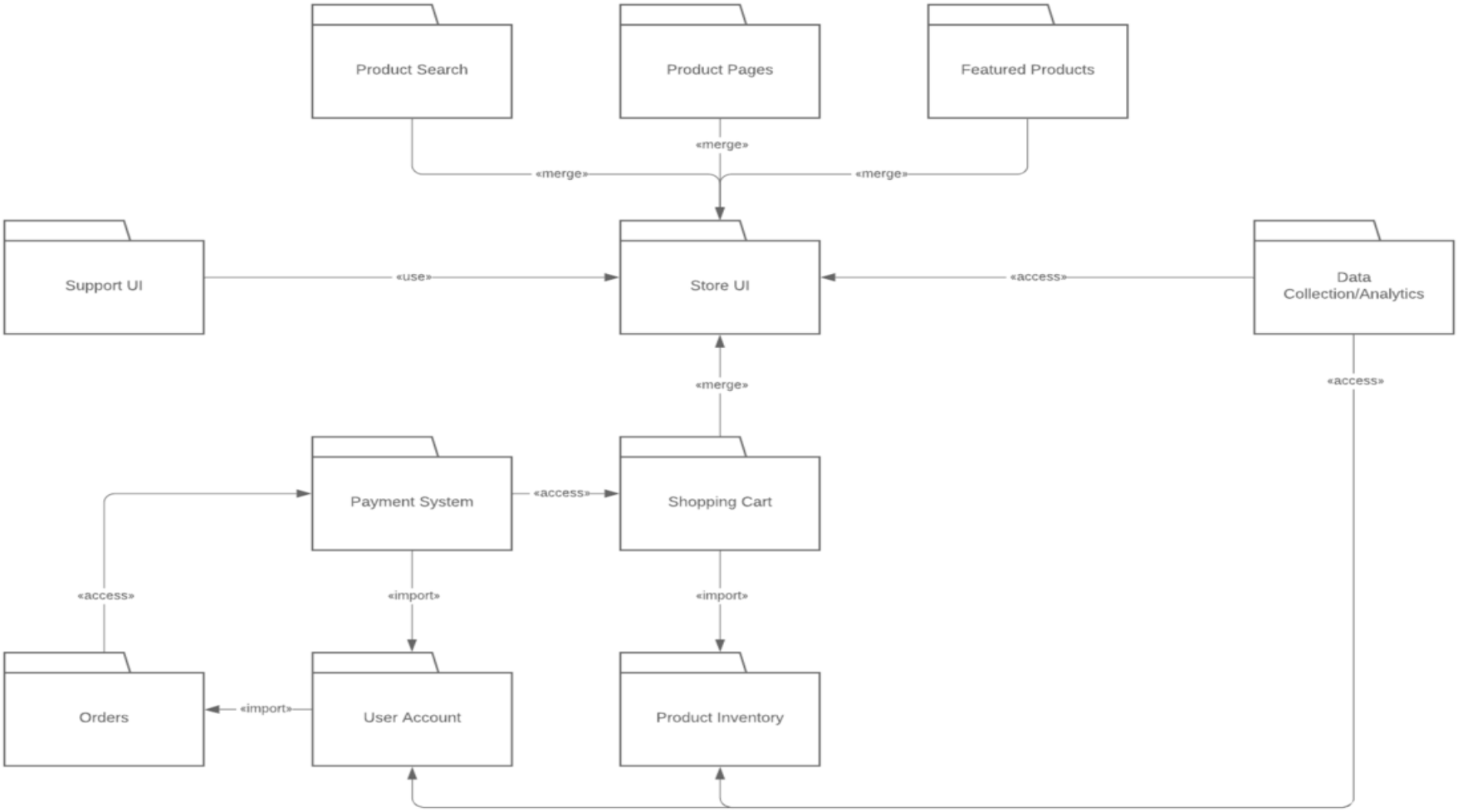
DEPENDANCY NOTATIONS

Access: Indicates that one package requires assistance from the functions of another package.



Import: Indicates that functionality has been imported from one package to another.

PACKAGE DIAGRAM EXAMPLE



Architectural Modeling

Architectural Modeling

* An architecture is the set of significant decisions about the organization of a software system.

* Guides about the elements, their interfaces ,their collaborations and their composition.

* Architecture can be defined at both a logical and physical level.
  * Logical Architecture: It shows more generic view of the architecture.
  * Physical Architecture: Describes in more details how the software and systems are designed.

Component Diagram

* Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems.

* It shows organizations & dependencies among set of components.

* It describe the organization of physical s/w components, including source code, run-time code & executables.

* Addresses static implementation view of a system .it represents the high-level parts that make up the system.

* High level reusable parts of a system are represented in a component diagram.

* Visualize the static aspect of the physical components and their relationships and to specify their details for construction.

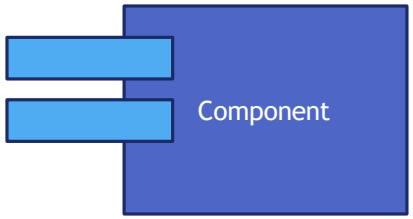
* Collecting various executables, libraries, files, tables(physical things), we build component diagram.

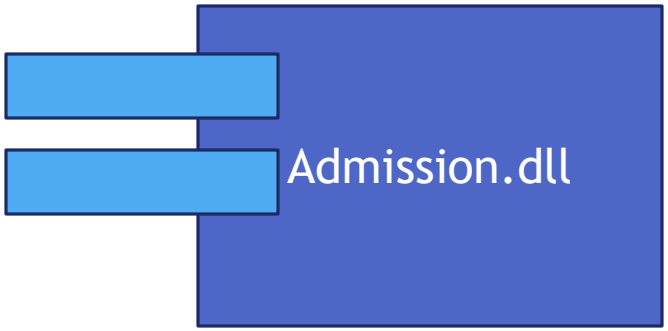
Component Diagram

* Elements Of a Component Diagram
  * Components
  * Interfaces
  * Ports
  * Connectors

Elements Of Component Diagram

* Components
  * Components are made up of one or more classes & describe parts of an application that can be assembled and reused.
  * Defined as “a physical replaceable part that conforms to and provides the realization of a set of interfaces”.
  * Graphically ,a component is rendered as a rectangle with tabs, with the name of the object in it, preceded by a colon and underlined.



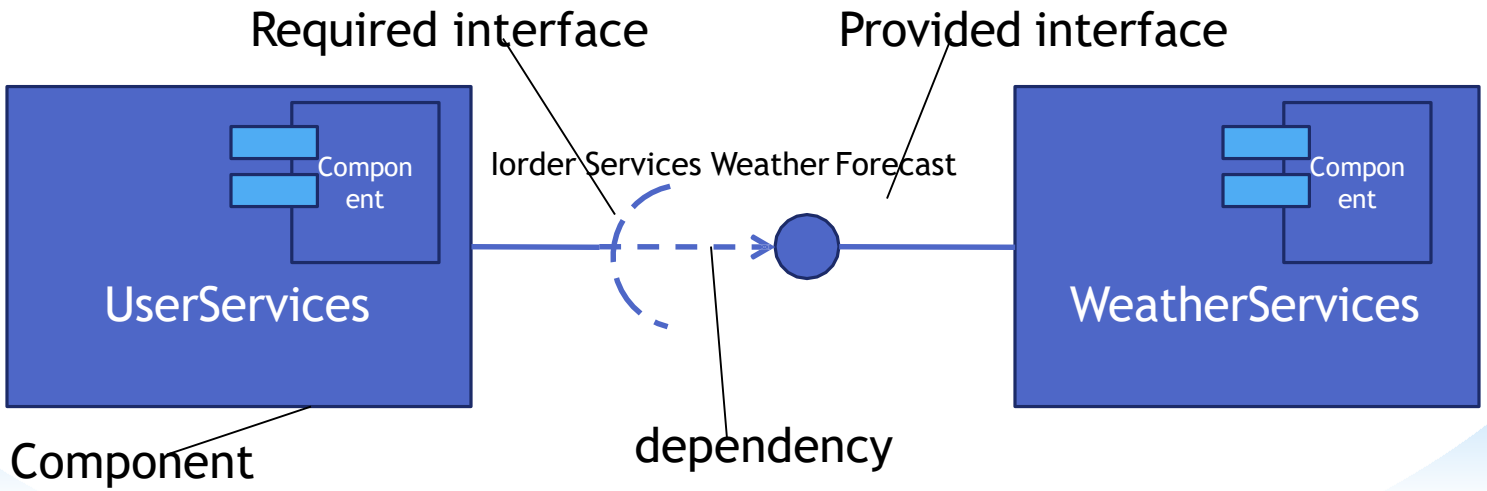


Elements Of Component Diagram

* Interface
  * It is a collection of operations that are used to specify a service of class or components.
  * Graphically it is displayed as a circle or as a typical class with stereotype of <<interface>>
  * **Types of interface**
    * **Provided Interface**
      * An interface that the component provides as a service to other component.
    * **Required Interface**
      * An interface that the component conforms to when requesting services from other components.

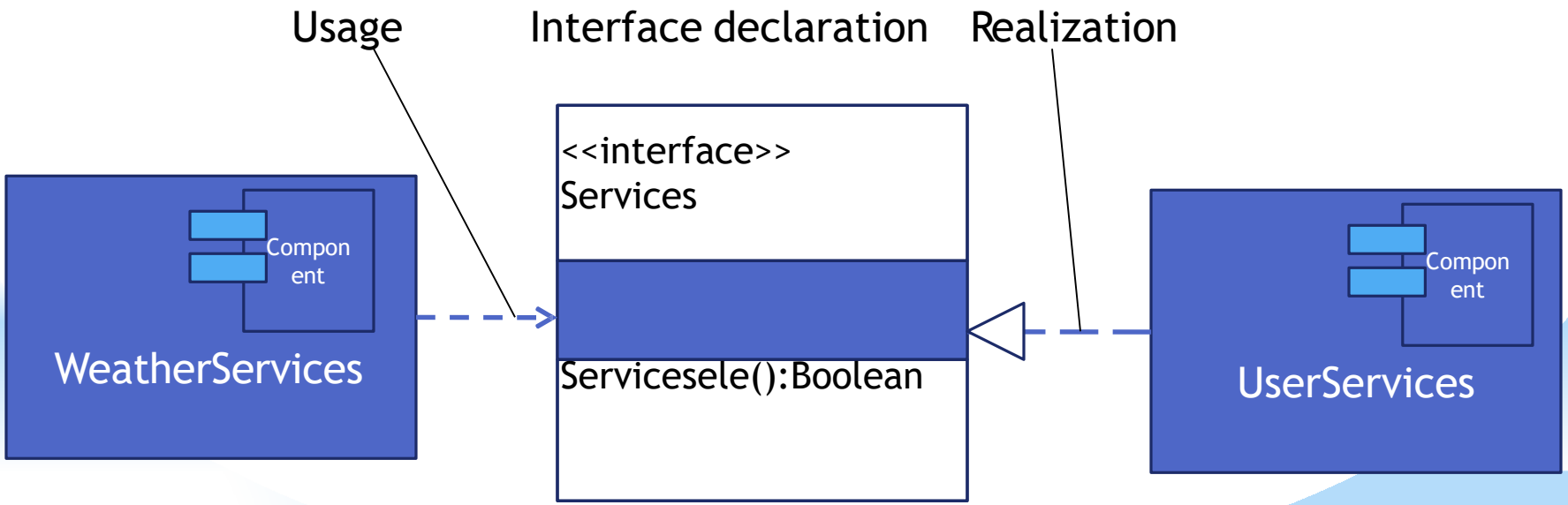
Elements Of Component Diagram

* Relationships between component & its interface
  * **Elided, iconic form.**
    * A provided interface is shown as a circle attached to the component by a line and a “lollipop”. A required interface is shown as a semicircle attached to the component by a line and a “socket”. In both cases, the name of the interface is placed next to the symbol.



Elements Of Component Diagram

* Relationships between component & its interface
  * **Expanded from**
    * The component that realizes the interface is connected to the interface using a full realization relationship. The component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

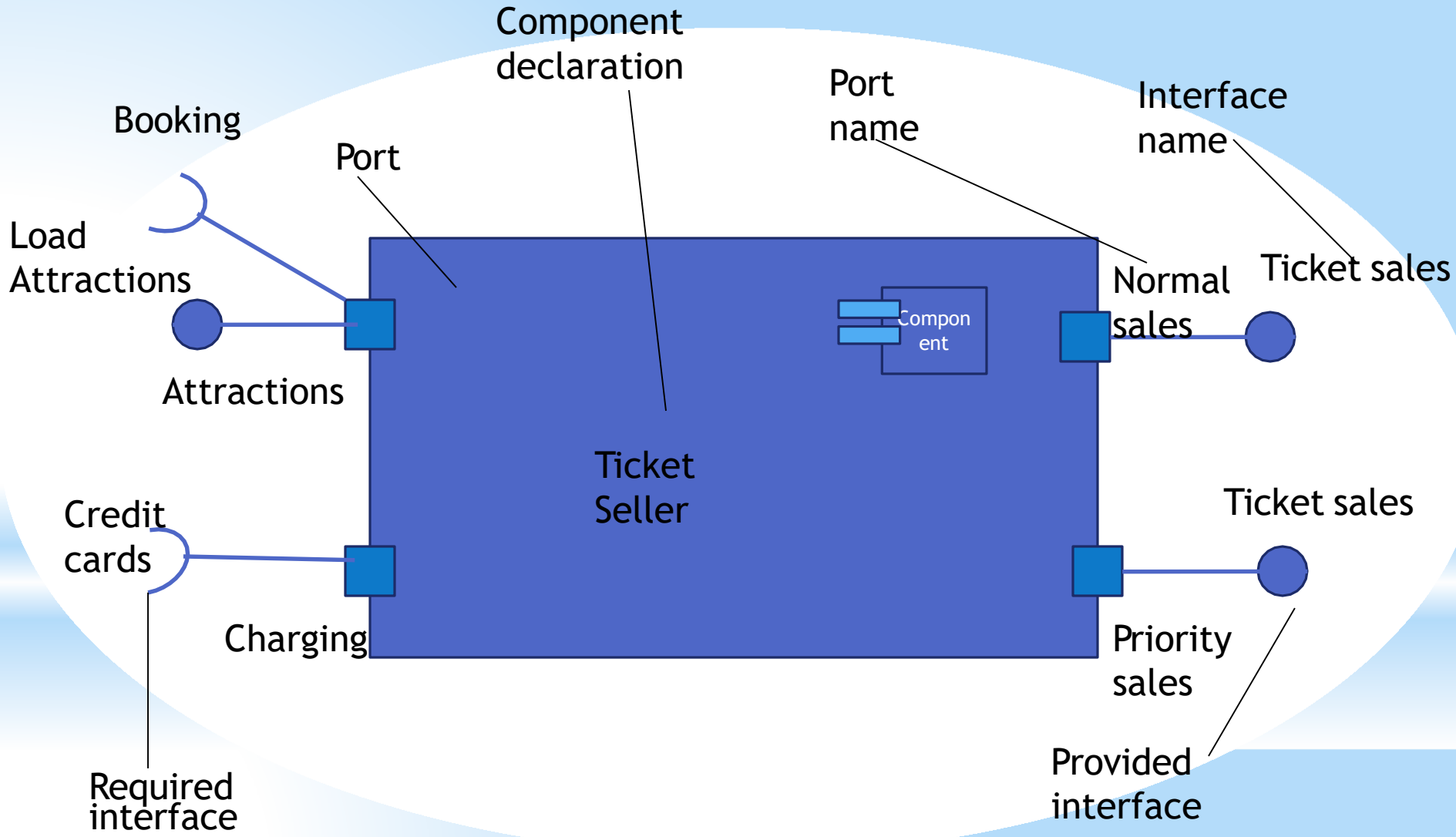


Elements Of Component Diagram

* Ports
  * Ports are used to control the implementation of all the operations in all of the provided interfaces in the component.
  * It is an explicit window into an encapsulated component.
  * All of the interactions into and out of the component pass through ports.
  * It has identity.
  * Component can communicate with the component through a specific port.
  * It is shown as a small square straddling the border of a component. Both provided and required interfaces may be attached to the port symbol.
  * A provided interface represents a service that can be requested through that port. A required interface represents a service that the port needs to obtain from some other component.

Elements Of Component Diagram

* Ports

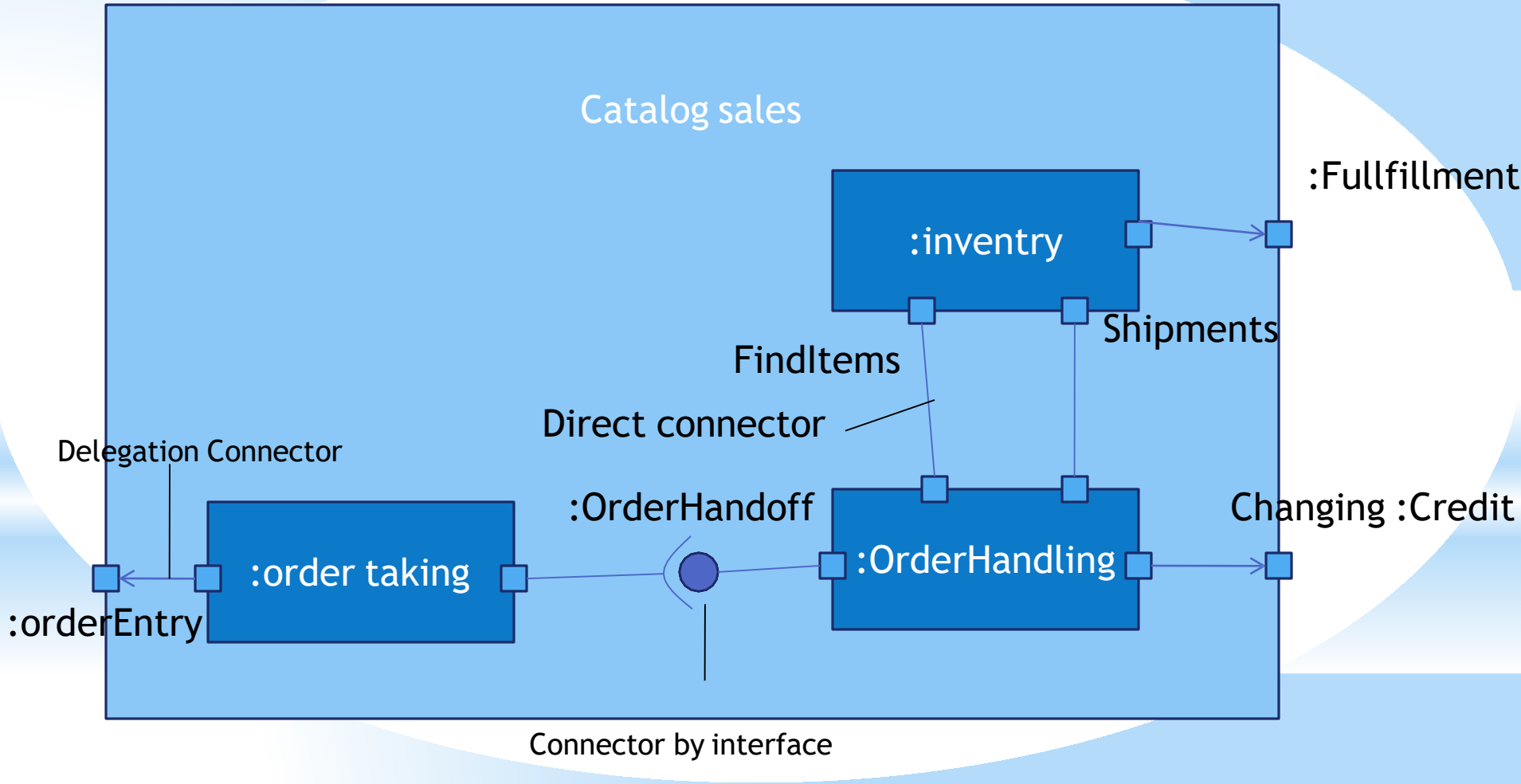


Elements Of Component Diagram

* Connectors
  * A wire between two port is called connector.
  * It represent a link or a transient link. Instance of an ordinary association.
  * A transient link represents a usage relationship between two components.
  * If two components are explicitly wired together, either directly or through ports, just draw a line between them or their ports.
  * If two components are connected because they have compatible interface, you can use a ball-and-socket notation to show that there is no inherent relationship between the components, although they are connected inside this component.
  * Types of Connector
    * Direct connector
    * Delegation connector

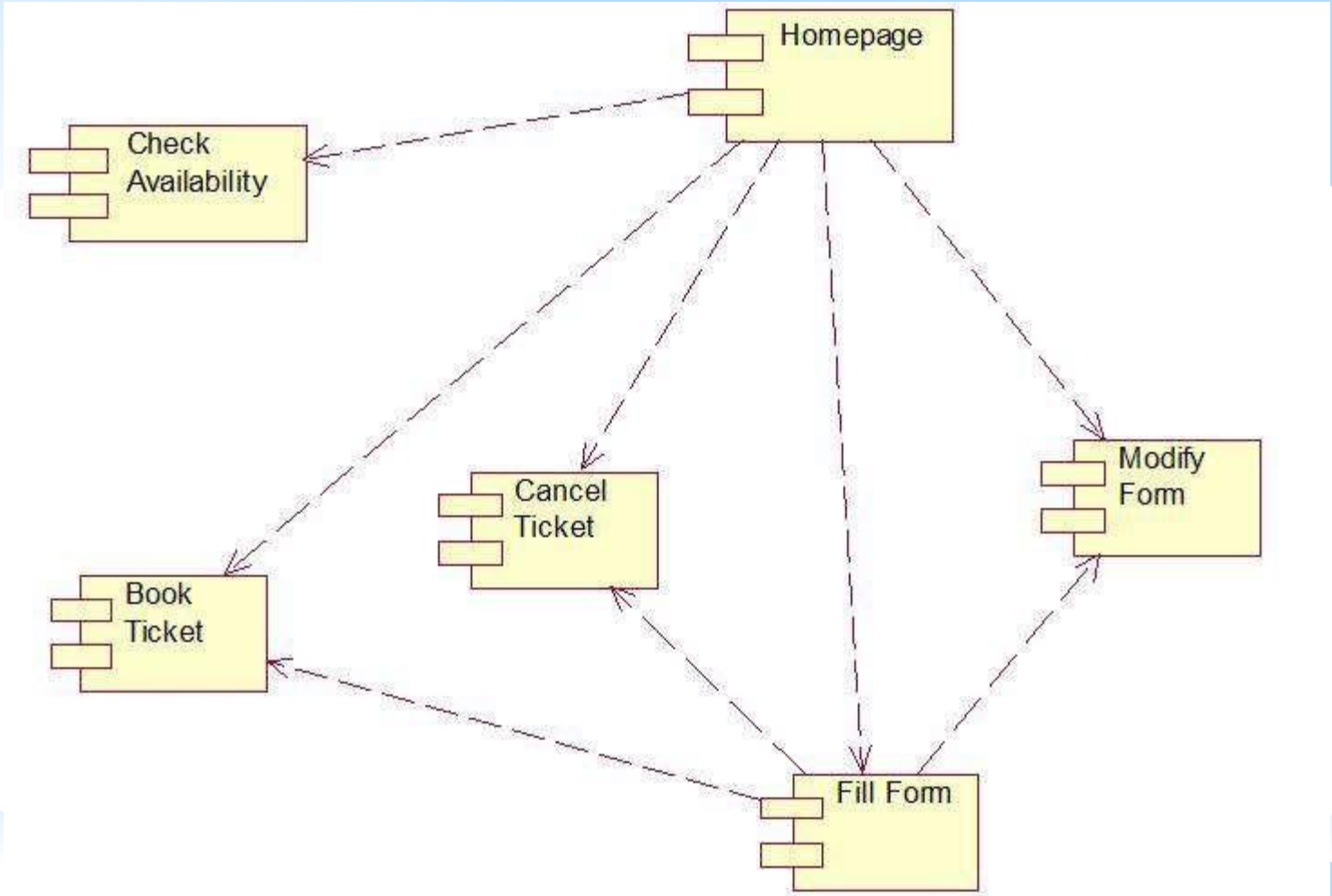
Elements Of Component Diagram

* Connectors



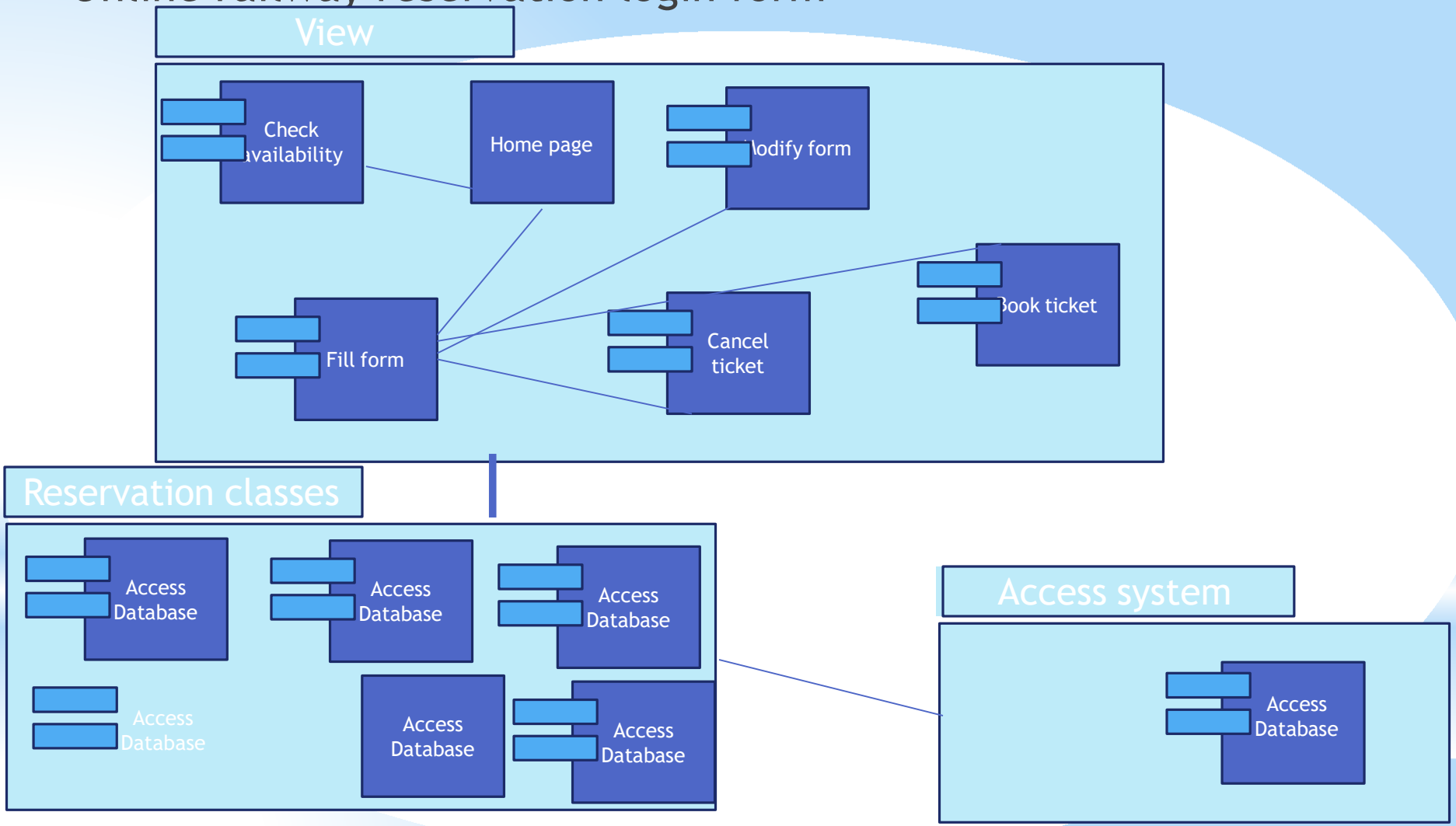
Elements Of Component Diagram

Online railway reservation login form



Elements Of Component Diagram

Online railway reservation login form



Deployment Diagram

* Used to model the static deployment view of a system.
* It is important for visualizing , specifying, and documenting embedded, client/server, & distributed systems.
* It is a diagram that shows the configuration of run time processing nodes & the artifacts that live on them.
* Graphically, a deployment diagram is a collection of vertices and arcs.
* Purpose of deployment Diagrams:
  * Visualize hardware topology of a system
  * Describe the H/W components used to deploy software components.
  * Describe runtime processing nodes.

Deployment Diagram

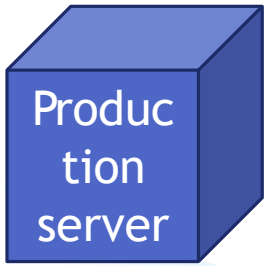
* Elements of Deployment Diagram
  * Nodes
  * Communication between Nodes/Connections
    * Association
    * Dependency
    * Generalization
    * Realization
  * Nodes and Artifacts
  * Common Modeling Techniques of nodes
    * Modeling Processors and Devices
    * Modeling the Distribution of Artifacts

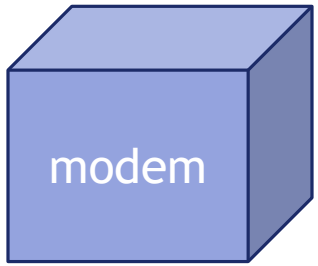
Deployment Diagram

* Elements of Deployment Diagram
  * Artifacts
    * Kinds of Artifacts
      * Deployment artifacts
      * Work product artifacts
      * Execution Artifacts
    * Common Modeling Techniques of Artifacts
      * Modeling Execution and Libraries
      * Modeling Tables, Files , and Documents
      * Modeling Source Code

Elements of Deployment Diagram

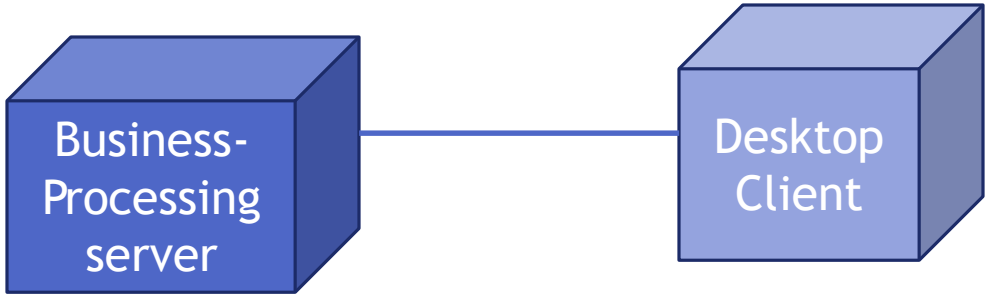
- Nodes:
  - Just like artifacts, are an important building block in modeling the physical aspects of a system.
  - It is a physical elements that exists at run time & represents a computational resource.
  - Graphical representation of node is cube.
  - Types of node
    - Processor:
      - It is a piece of hardware capable of executing programs.
      - A Processor can have list of processes on it.
      - Represented as shaded cube with name of the object.
    - Device:
      - A piece of hardware incapable of executing program is called as device.
      - Device will also have on a cube.





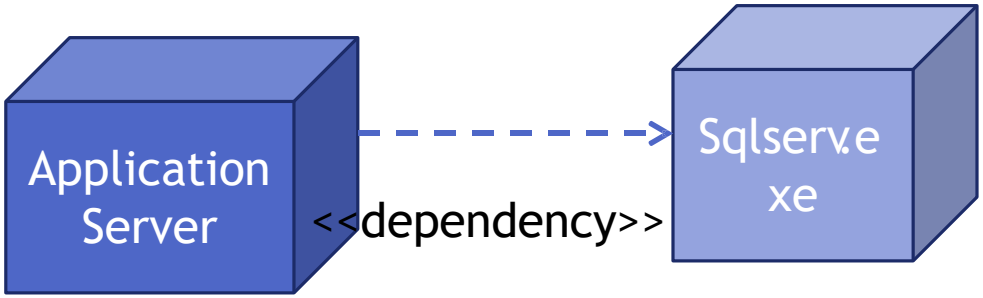
Elements of Deployment Diagram

* Communication between Nodes/ Connections:
  * Association:
    * It refers to a physical connection or link between the nodes.
    * It is shown as a solid-line between nodes.



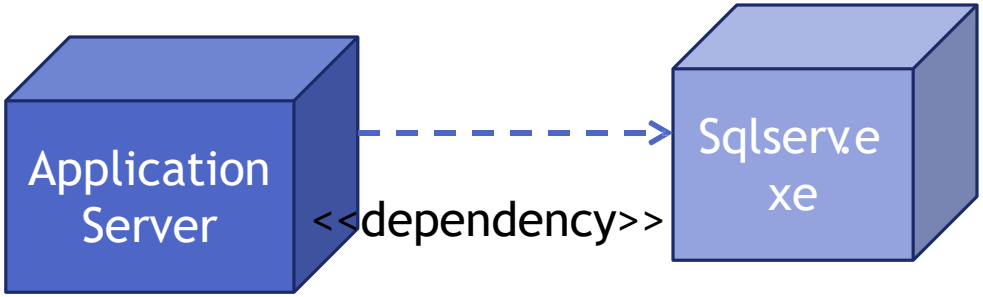
Elements of Deployment Diagram

* Communication between Nodes/ Connections:
  * Dependency
    * It is a relationship that indicates that a model element is in some way dependent of another model element.
    * Dependency of a node on components is depicted using dashed lines.



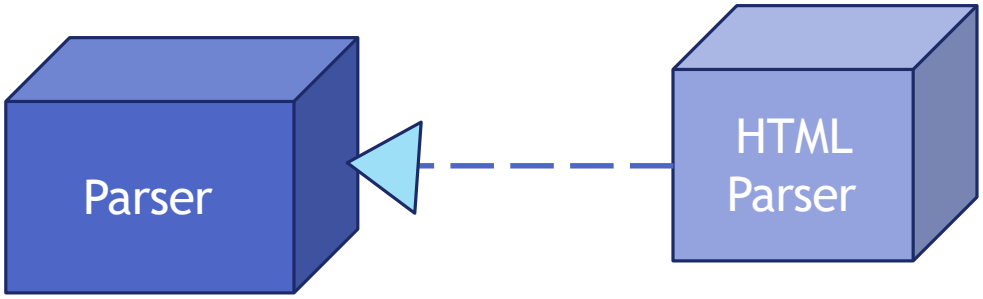
Elements of Deployment Diagram

* Communication between Nodes/ Connections:
  * Generalization
    * It is a relationship that indicates that a model element is in some way dependent of another model element.
    * It is shown as a solid-line between nodes.



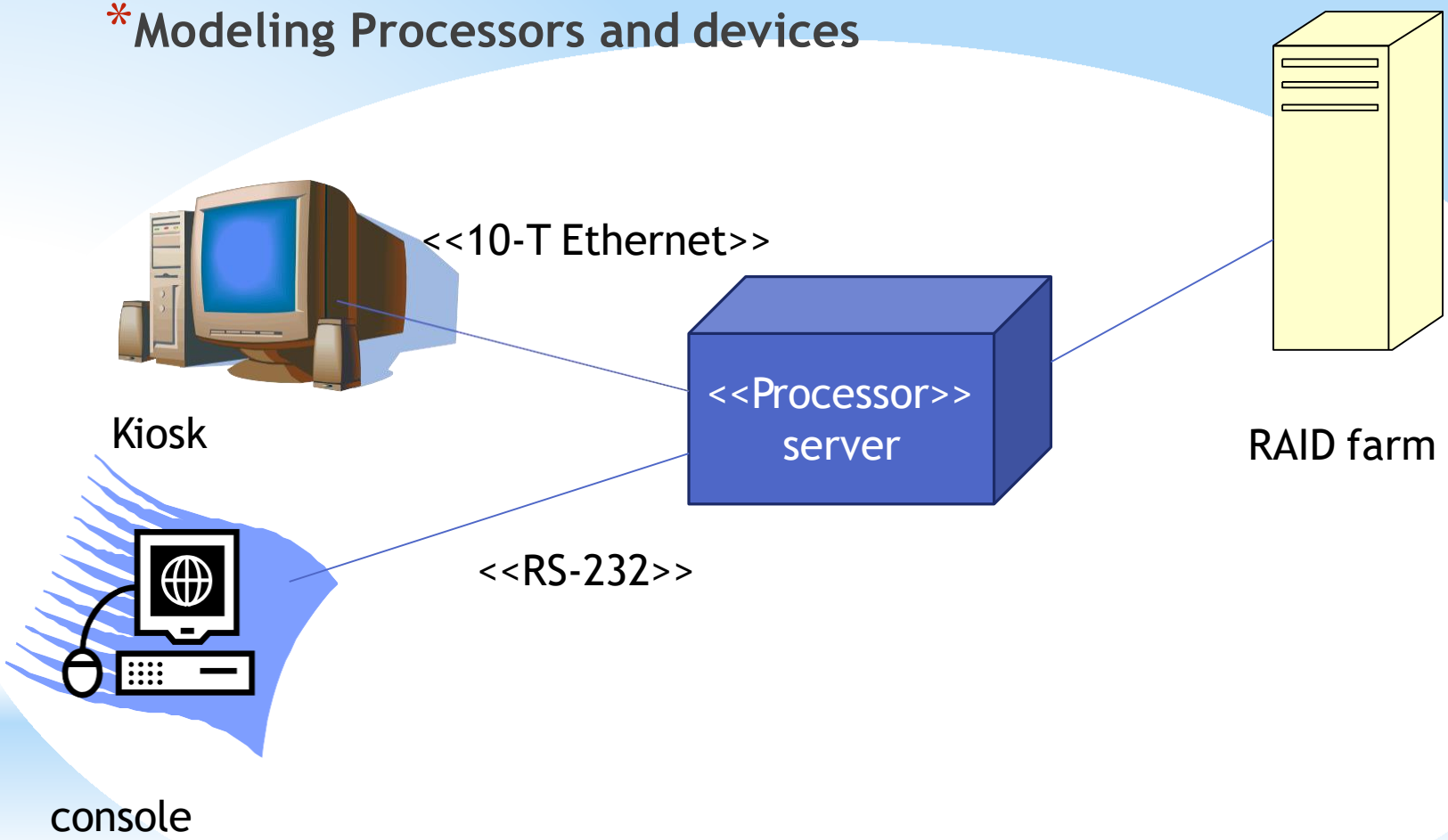
Elements of Deployment Diagram

- Communication between Nodes/ Connections:
  - **Realization**
    - It is a relationship between interface and classes or components that realize them
    - It shows as a dashed line with hollow triangle.
    - Example the relationship between a interface and a class that realizes or execute that interface



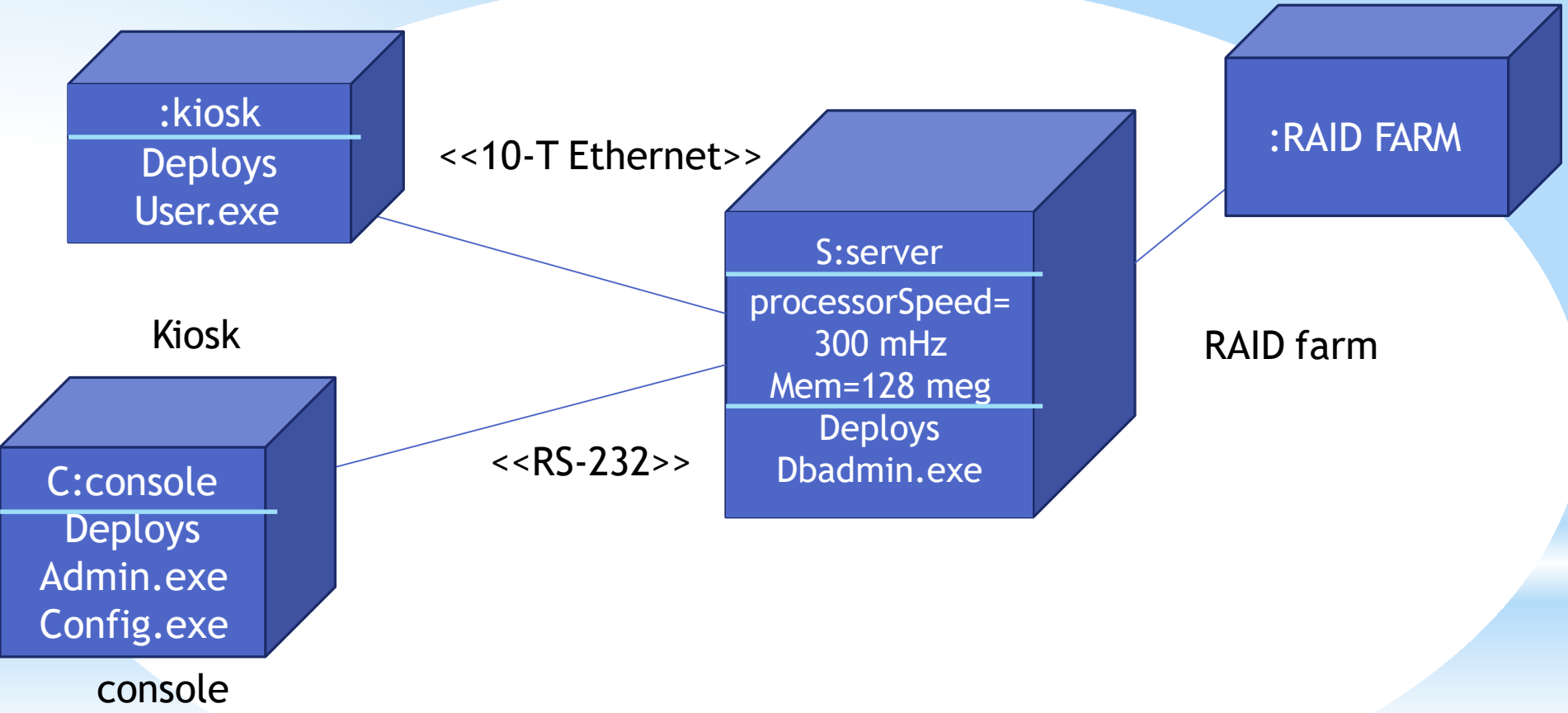
Elements of Deployment Diagram

- Common Modeling Techniques of Nodes
  - **Modeling Processors and devices**



Elements of Deployment Diagram

- Common Modeling Techniques of Nodes
  - **Modeling the Distribution of Artifacts**



Elements of Deployment Diagram

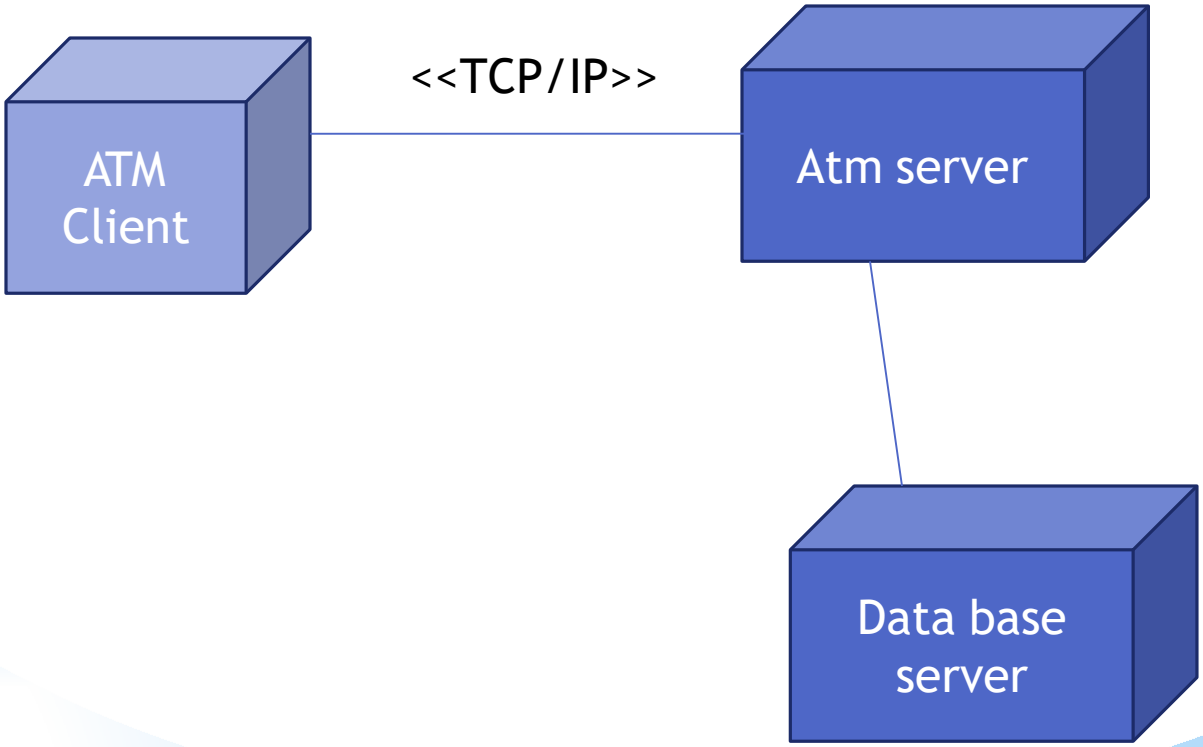
- Artifacts
  - Artifacts are physical entities that are deployed on nodes, devices and executable environments.
  - It is a physical replaceable part of a system.
  - Executables, libraries, tables files and documents.
  - Standard stereotypes for artifacts
    - <<file>>, <<document>>, <<source>>, <<library>>, <<executable>>, <<script>>.
  - Artifact must have a unique name

Deployment Diagram

* Elements of Deployment Diagram
  * Artifacts
    * Kinds of Artifacts
      * Deployment artifacts
      * Work product artifacts
      * Execution Artifacts
    * Common Modeling Techniques of Artifacts
      * Modeling Execution and Libraries
      * Modeling Tables, Files , and Documents
      * Modeling Source Code

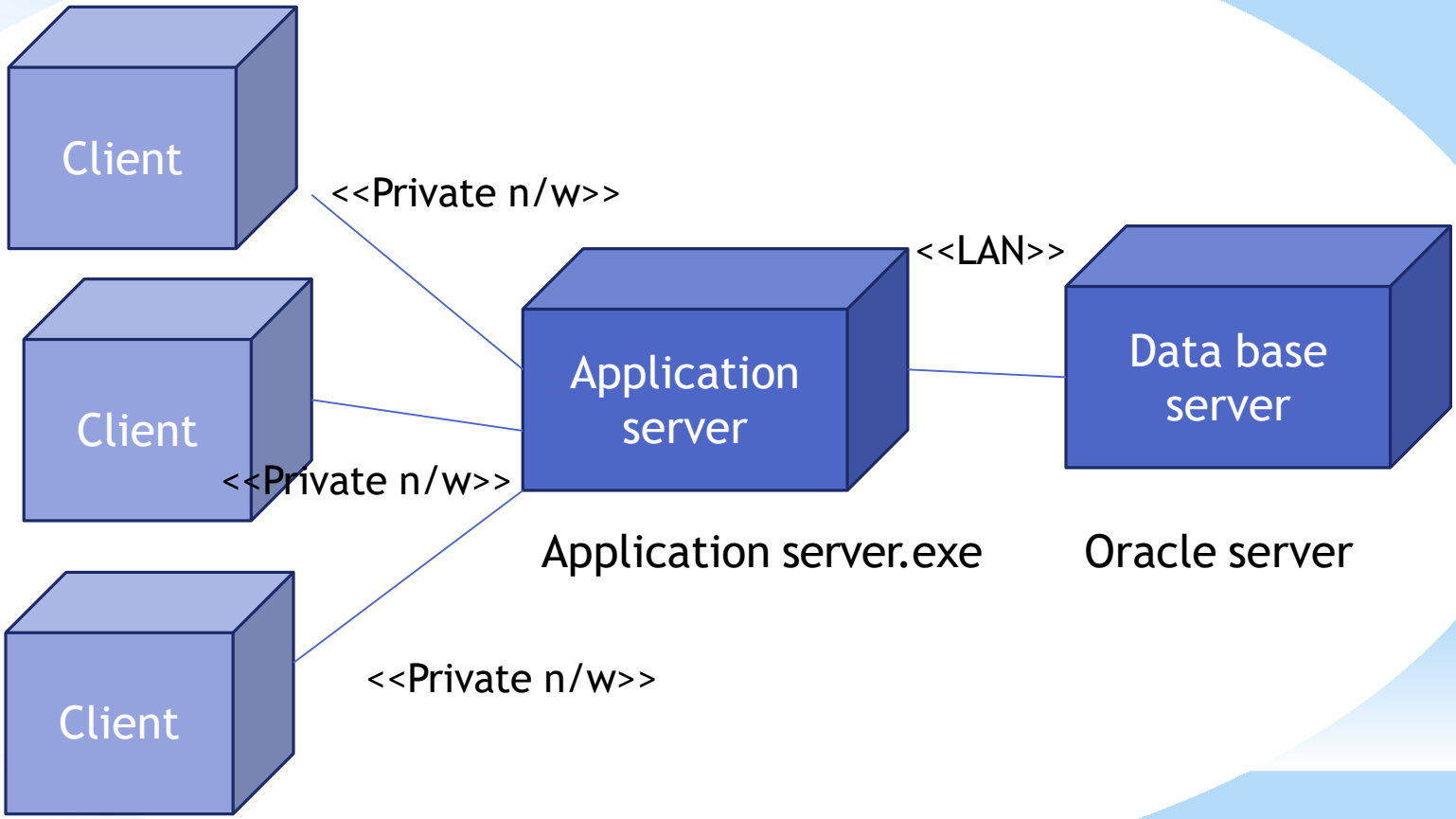
Deployment Diagram

* Atm machine



Deployment Diagram

* Online Shopping



Deployment Diagram

* Online Shopping