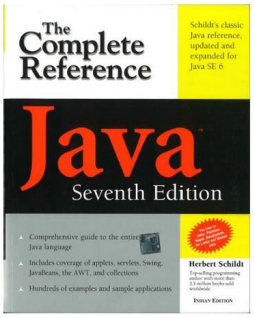
CST281
Object Oriented Programming

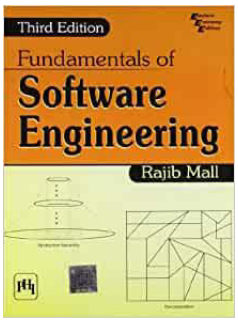
MODULE 1

Syllabus

- Introduction:
  - Approaches to Software Design, Functional Oriented Design, Object Oriented Design, Case Study of Automated Fire Alarm System.
- Object Modeling Using UML
  - Basic Object Oriented concepts, UML (Unified Modeling Language) diagrams, Use case model, Class diagram, Interaction diagram, Activity diagram, State chart diagram.
- Introduction to Java -
  - Java programming Environment and Runtime Environment, Development Platforms Standard, Enterprise Java Virtual Machine (JVM), Java compiler, Byte code, Java applet, Java Buzzwords, Java program structure, Comments, Garbage Collection, Lexical Issues.

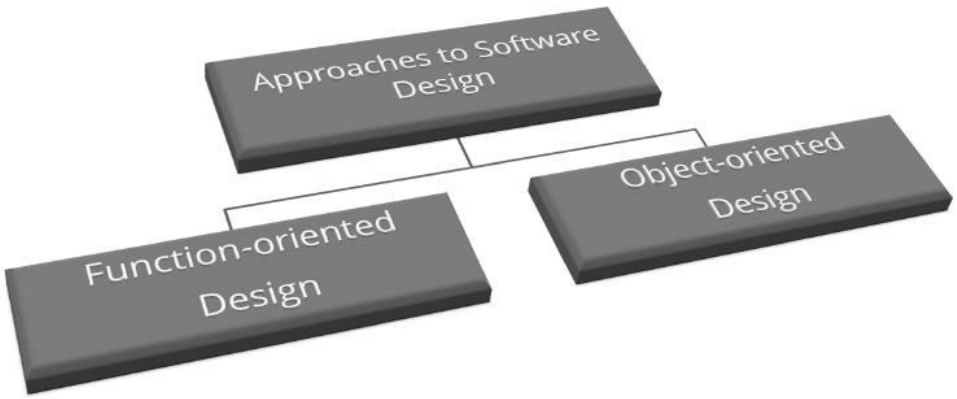
Textbook





- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.
- Rajib Mall, Fundamentals of Software Engineering, 4th edition, PHI, 2014.

Approaches to Software Design



Function Oriented Design

- A system is viewed as something that performs a set of functions.
- Two salient features of the function-oriented design approach:
  a) Top-down decomposition
  b) Centralised system state

a) Top down decomposition

- Starting at this high level view of the system, each function is successively refined into more detailed functions
- For example, consider a function create new library member which essentially creates
  - the record for a new member ,
  - assigns a unique membership number to him
  - prints a bill towards his membership charge

- This high-level function may be refined into the following subfunctions:
  - assign-membership-number
  - create-member-record
  - print-bill
- Each of these subfunctions may be split into more detailed subfunctions and so on.

b) Centralised system state

- The system state is centralised and shared among different functions.
- For example, in the library management system, several functions such as the following share data such as member-records for reference and updation:
  - create-new-member
  - delete-member
  - update-member-record

Object-oriented Design

- A system is viewed as being made up of a collection of objects
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- The system state is de-centralised since there is no globally shared data in the system and data is stored in each object.

- For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data.
- The methods defined for one object cannot directly refer to or change the data of other objects.

Object - oriented versus function-oriented design approaches

Object-Oriented Approach	Function-Oriented Approach
The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.	The basic abstractions, which are given to the user are real world functions.
Functions are grouped together on the basis of the data they operate since the classes are associated with their methods	Functions are grouped together by which a higher level function is obtained.
In this approach a state information is not represented in a centralised memory but is implemented or distributed among the objects of the system.	In this approach the state information is often represented in a centralised state memory.
We decompose in the class level	We decompose in the function/procedure level
Bottom up approach	Top down approach

Case Study on Automatic fire Alarm system

- The owner of a large multi-stored building wants to have a computerized fire alarm system for his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition has occurred and then sound the alarms only in the neighboring locations. The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel
- man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Fire Alarm System Function Oriented Approach

```
/* Global data (system state) accessible by various functions */
BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
BOOL alarm_status[MAX_ROOMS]; /* alarm activated when status is set */
int alarm_locs[MAX_ROOMS]; /* room   number where alarm is located */
int neighbor-alarm[MAX_ROOMS][10];   /* each detector has at most 10 neighboring
locations */
The functions which operate on the    system state are:
interrogate_detectors();
get_detector_location();
determine_neighbor(); ring_alarm();
reset_alarm();
report_fire_location();
```


Fire Alarm System Object Oriented Approach

```
class detector
attributes:
            status, location, neighbors
operations:
            create, sense_status, get_location,find_neighbors
class alarm
attributes:
            location, status
operations:
            create, ring_alarm, get_location,reset_alarm
```

- In the object oriented program, an appropriate number of instances of the class detector and alarm should be created.
- If the function-oriented and the object-oriented programs are examined,
- it can be seen that in the function-oriented program, the system state is centralized and several functions accessing this central data are defined.
- In case of the object-oriented program, the state information is distributed among various sensor and alarm objects.

Syllabus

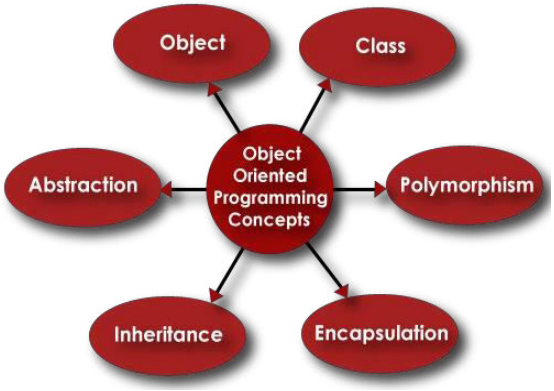
- Introduction:
  - Approaches to Software Design, Functional Oriented Design, Object Oriented Design, Case Study of Automated Fire Alarm System.
- Object Modeling Using UML
  - Basic Object Oriented concepts, UML (Unified Modeling Language) diagrams, Use case model, Class diagram, Interaction diagram, Activity diagram, State chart diagram.
- Introduction to Java -
  - Java programming Environment and Runtime Environment, Development Platforms Standard, Enterprise Java Virtual Machine (JVM), Java compiler, Bytecode, Java applet, Java Buzzwords, Java program structure, Comments, Garbage Collection, Lexical Issues.

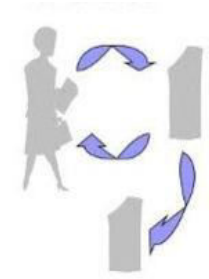
Object Modeling Using UML

Basic Object Oriented concepts, UML (Unified Modeling Language) diagrams, Use case model, Class diagram, Interaction diagram, Activity diagram, State chart diagram.

Basic Object Oriented Concepts

- Object-Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes.
- The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour.

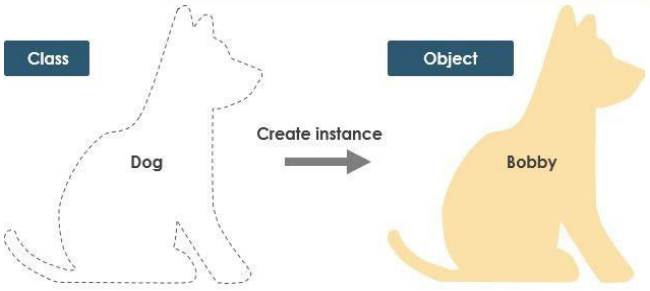


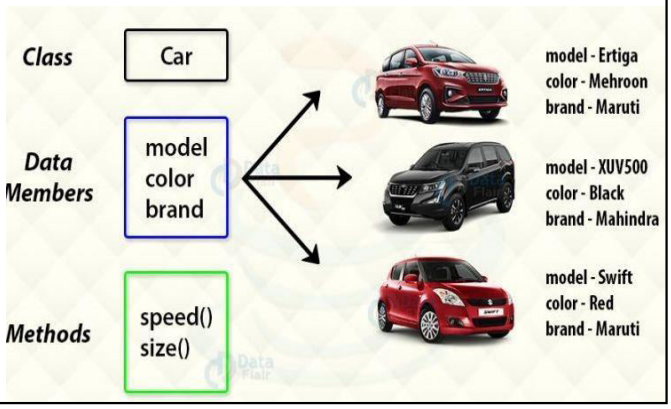


Withdraw, deposit, transfer

Customer, money, account

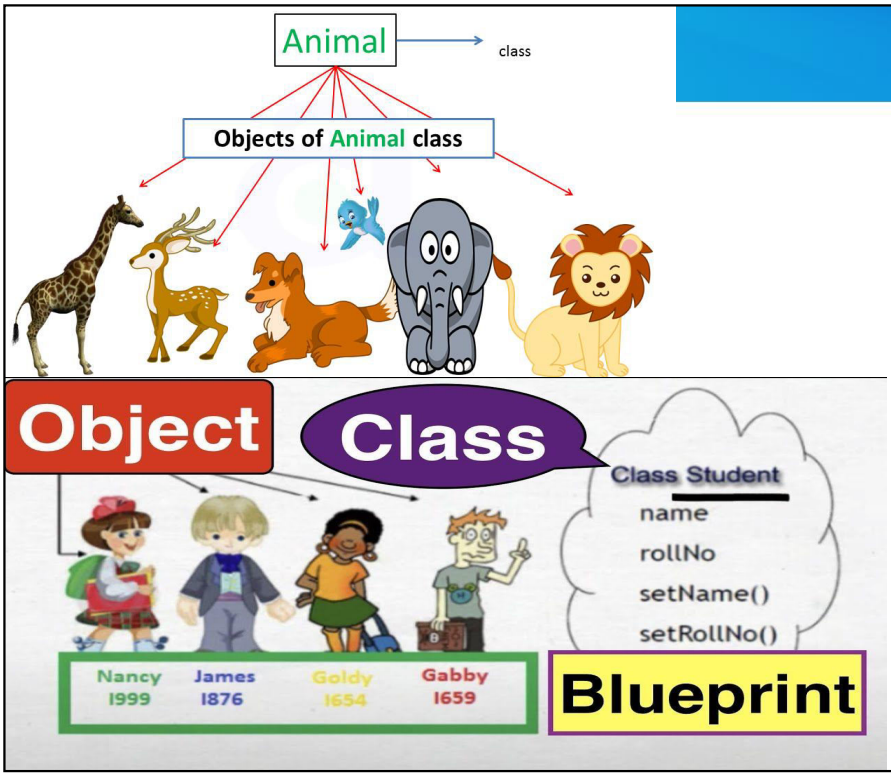
- Major object oriented concepts are
  - Objects
  - Class
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism





Objects

- Complex data type that has an identity, contains other data types called attributes and modules of code called operations or methods
- Attributes and associated values are hidden inside the object.
- Any object that wants to obtain or change a value associated with other object, must do so by sending a message to one of the objects (invoking a method)

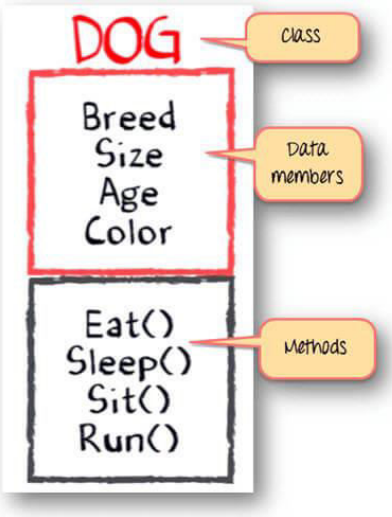


State => Variables
Name RollNo

Behaviors => Functions
setName() setRollNo()

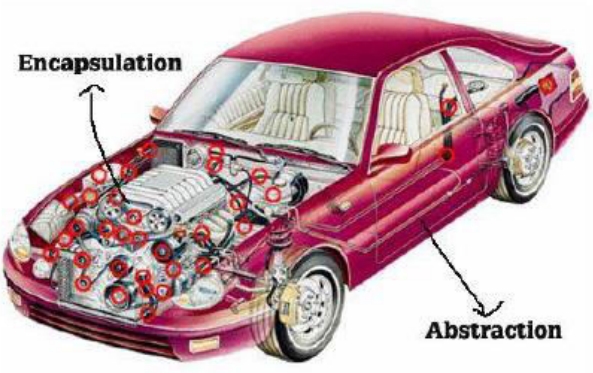
Classes

- Classes are templates that have methods and attribute names and type information, but no actual values!
- Objects are generated by these classes and they actually contain values.
- We design an application at the class level
- When the system is running objects are created by classes as they are needed to contain state information.
- When objects are no longer needed by the application, they are eliminated.



Abstraction

- Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.



Encapsulation

- The **wrapping up** of data(variables) and functions (methods) that operates on the data, into a single unit (called class) is known as encapsulation.
- It is also called "**information hiding**", hides the internal representation, or state, of an object from the outside through access modifies (Private / Public).
- **Hiding unnecessary** details, at implementation level.

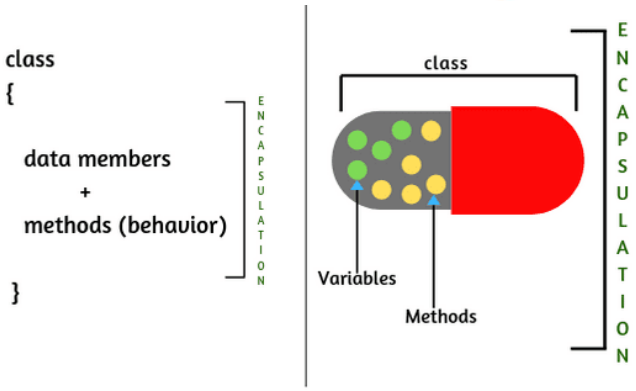


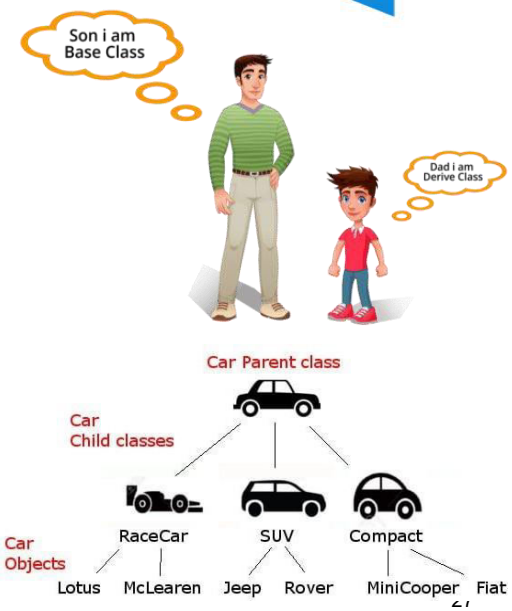
Fig: Encapsulation

Encapsulation Cont.

- **Protection of data** from accidental corruption, by means of **access specifiers** (Private / Public), it **provides security.**
- Flexibility and extensibility of the code and reduction in complexity
- Encapsulation of a class can hide the internal details of how an object does something
- Encapsulation protects abstraction

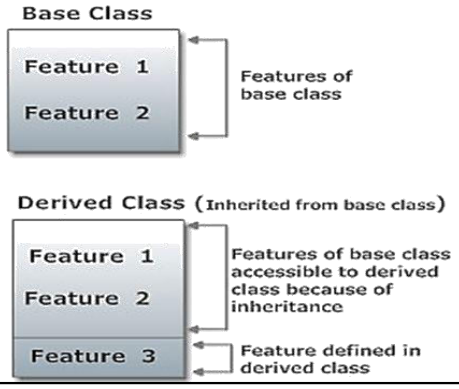
Inheritance

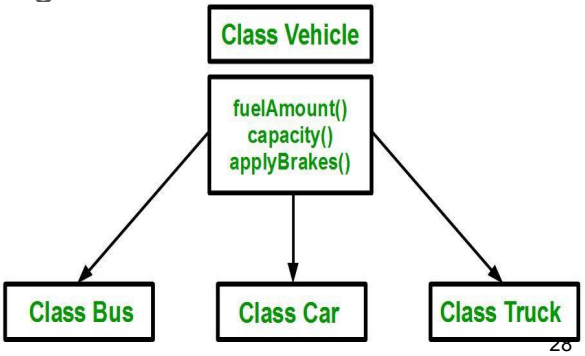
- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- Inheritance is the process by which objects of one class acquired the properties of objects of another classes
- **Base Class** : The class whose properties are inherited by subclass is called Base Class or Super class.
- **Derived Class** : The class that inherits properties from another class is called Subclass or Derived Class.



Inheritance Cont..

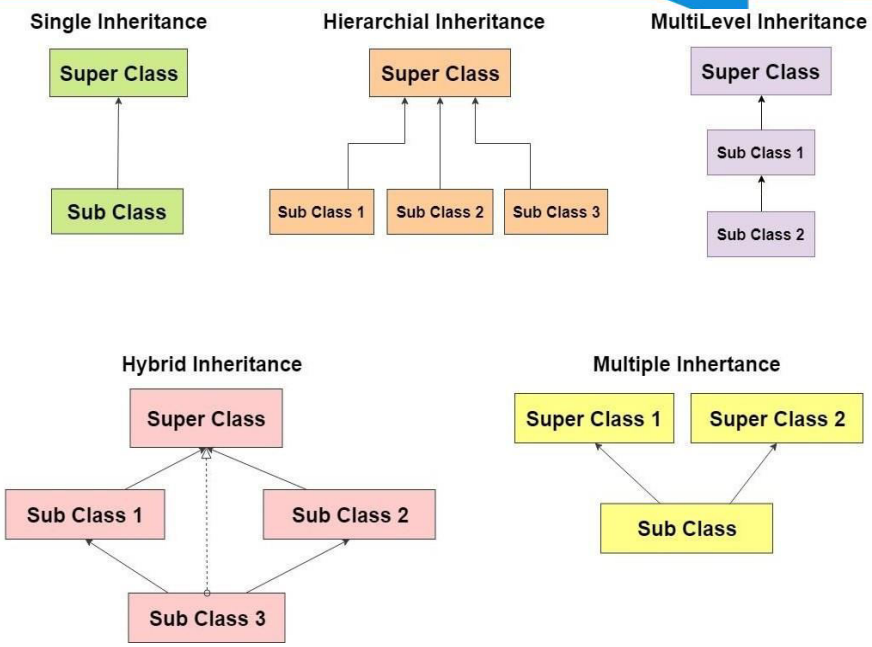
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.





Types of Inheritance

01 Single Inheritance
02 Multilevel Inheritance
03 Hierarchical Inheritance
04 Multiple Inheritance
05 Hybrid Inheritance



Polymorphism

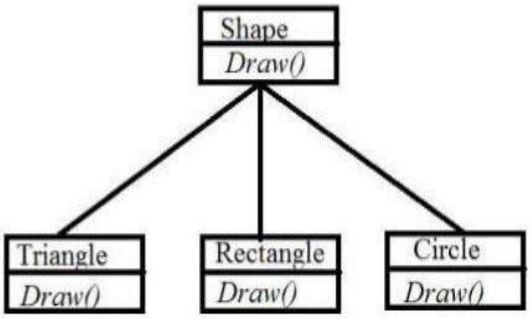
- The word polymorphism means **having many forms.**
- Refers to a programming language's **ability to process objects differently**
- depending on their **data type or class**
- An operation may exhibit different behaviours in different instances. The behaviour depends upon the **types of data used in the operation**.
- Ability for objects of different classes related by inheritance to **respond differently to the same member function** call

In Shopping malls behave like Customer

In Bus behave like Passenger

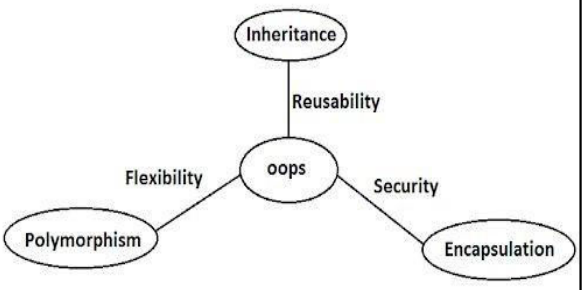
In School behave like Student

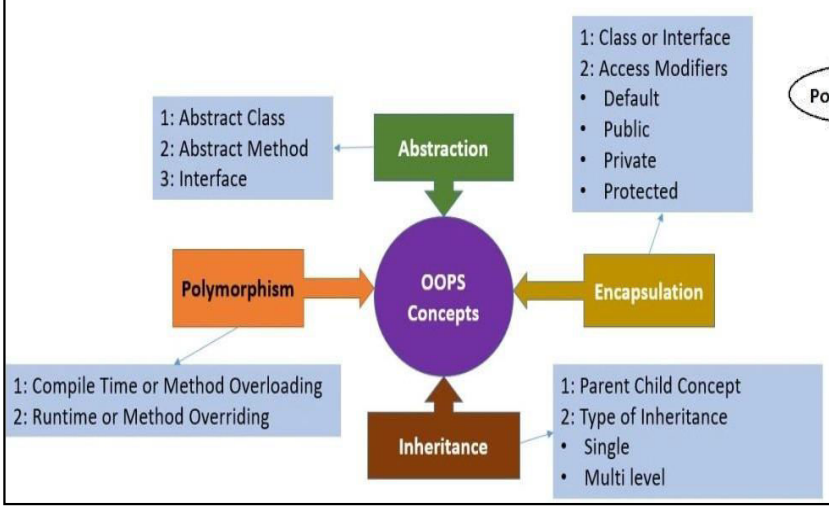
At Home behave like Son



31

OO concepts review





32

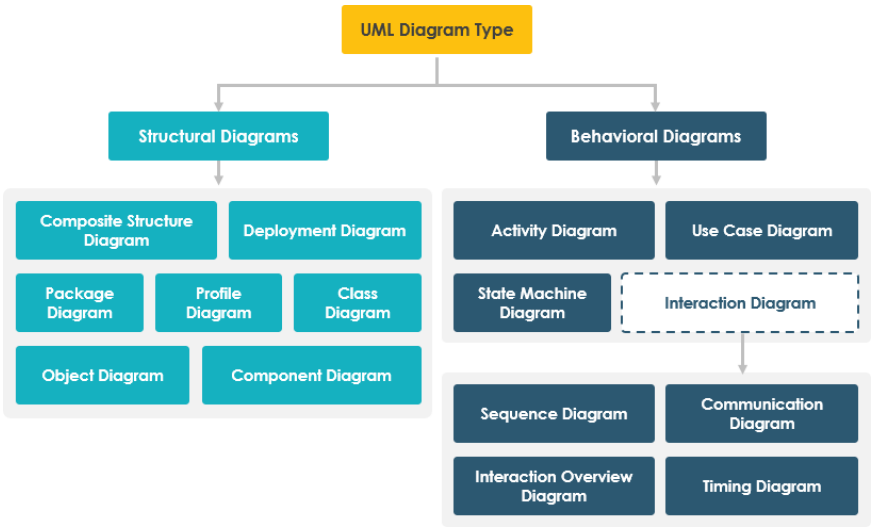
Object Modeling using Unified Modeling Language

UML

Unified Modeling Language(UML)

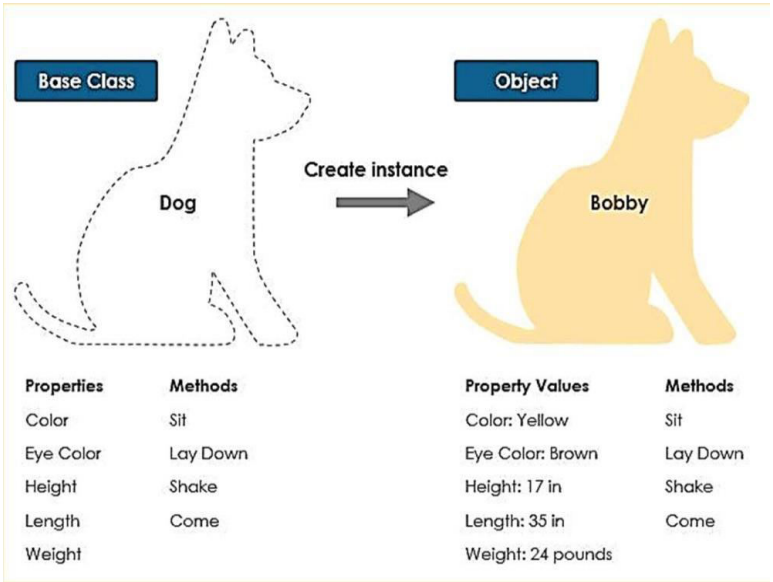
- A standardized modeling language consisting of an integrated set of diagrams,
- developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- UML is a pictorial language used to make software blueprints
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- UML is not a programming language but tools can be used to generate code in various languages using UML diagrams.

UML Diagram Types



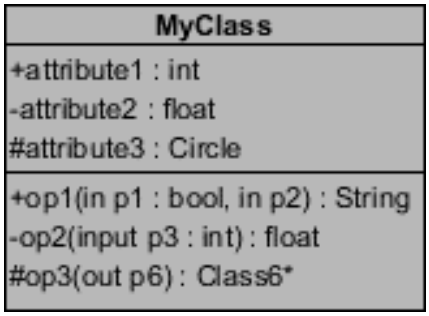
- UML diagrams are broadly classified as structure diagrams and behaviour diagrams.
  - Structure Diagrams : Capture static aspects or structure of a system
  - Behaviour Diagrams : Capture dynamic aspects or behavior of the system

CLASS DIAGRAM



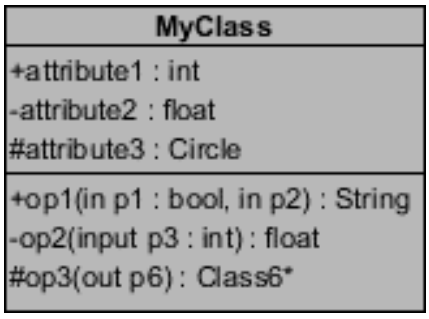
Class diagram

- Using class diagram, we can create the static structure of a system by showing the system's classes, their attributes, operations (or methods).
- Class diagrams helps us to identify relationship between different classes or objects.



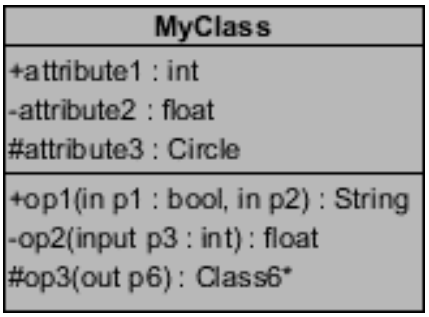
Class diagram

- Class Name - The name of the class appears in the first partition.
- Class Attributes
  - Attributes are shown in the second partition.
  - The attribute type is shown after the colon.
  - Attributes map onto member variables (data members) in code.
- 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 data type of method parameters are shown after the colon following the parameter name.



Class Visibility

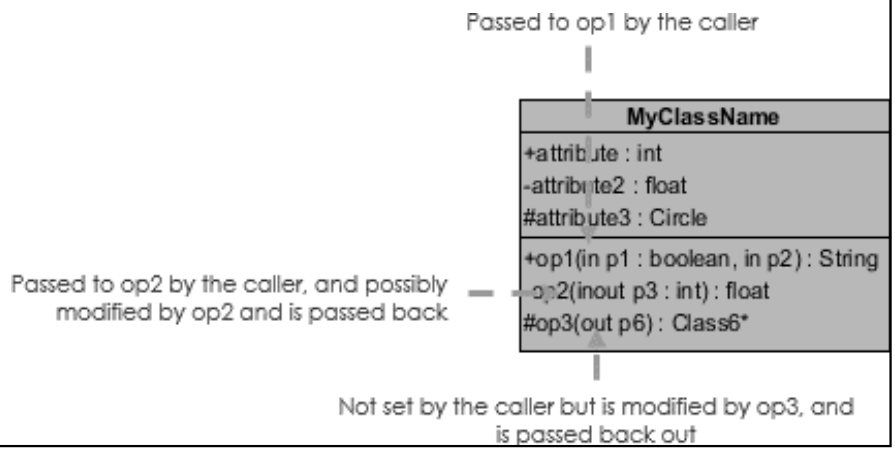
- The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.
  - + denotes public attributes or operations
  - - denotes private attributes or operations
  - # denotes protected attributes or operations



Parameter Directionality

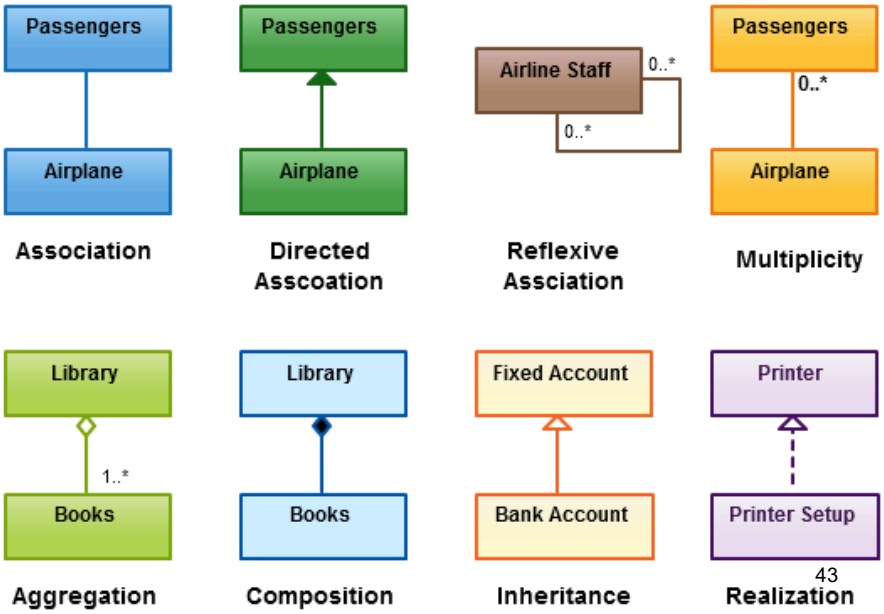
- Each parameter in an operation (method) may be denoted as in, out or in-out which specifies its direction with respect to the caller.
- This directionality is shown before the parameter name.

• **inout**: the parameter comes with a value and after executing the called object, the resulting value is returned to the caller object.
•**in:** the parameter comes with a value and after executing the called object, the resulting value is not returned to the caller object. (*)
•**out:** the parameter doesn't come with a value and after executing the called object, the resulting value is returned to the caller object.



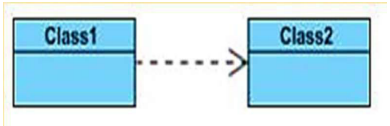
Relationship Between Classes

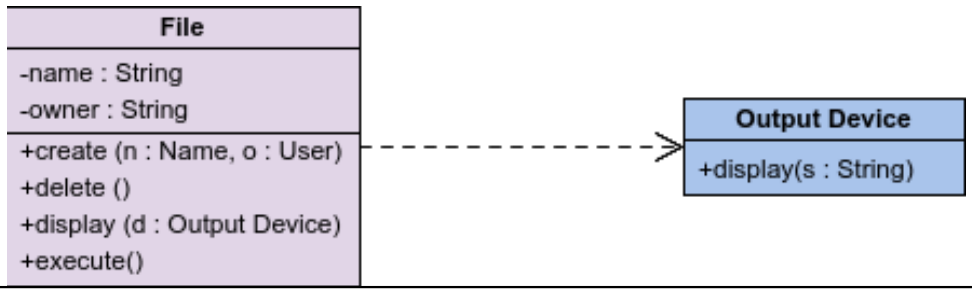
- UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams.
- If precisely interpreted, the implemented code will correctly reflect the intent of the designer.



Relationship Between Classes- Dependency

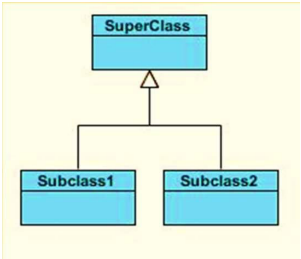
- A dependency means the relation between two or more classes in which a change in one may force changes in the other.
- Dependency indicates that one class depends on another.
- A dashed line with an open arrow

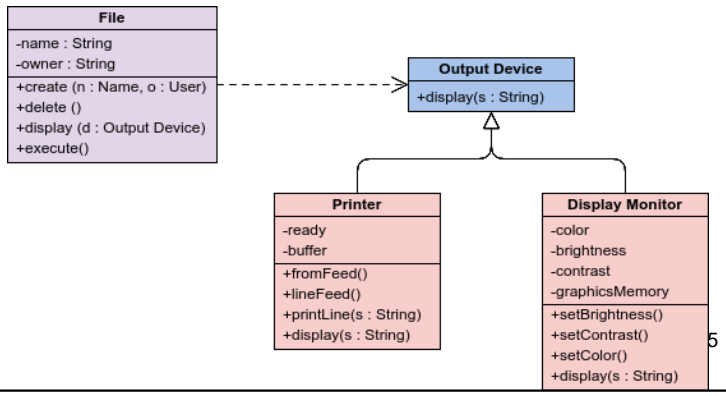




Relationship Between Classes- Generalization

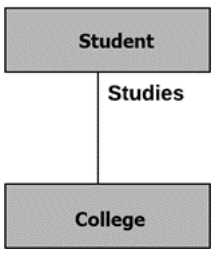
- A generalization (inheritance) helps to connect a subclass to its superclass.
- A sub-class is inherited from its superclass.
- Class diagram allows inheriting from multiple superclasses.
- A solid line with a hollow arrowhead that point from the child to the parent class





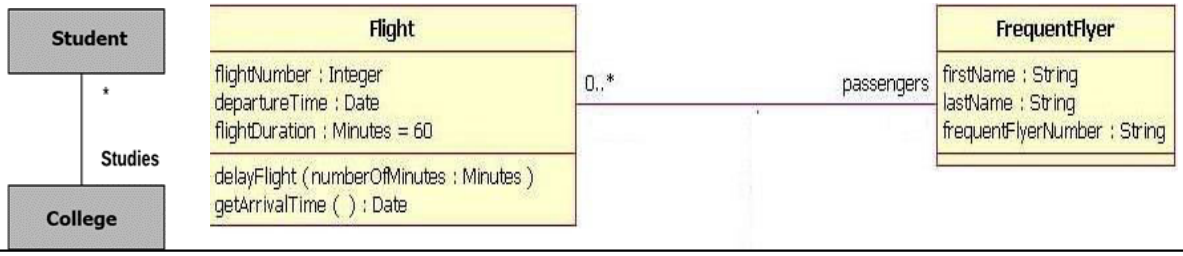
Relationship Between Classes- Asssociation

- This kind of relationship represents static relationships between classes A and B.
- Eg: A Person lives at an location
- It should be named to indicate the role played by the class attached at the end of the association path.
- A solid line connecting two classes



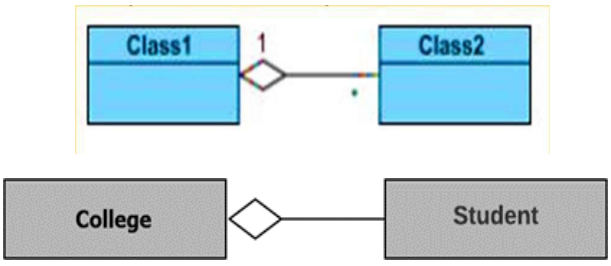
Relationship Between Classes- Multiplicity

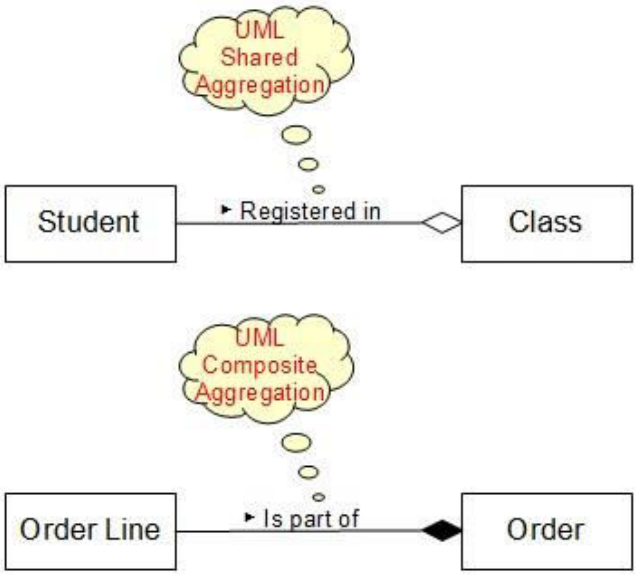
- Multiplicity means how many objects of each class take part in the relationships
  - Exactly one - 1
  - Zero or one - 0..1
  - Many - 0..* or *
  - One or more - 1..*
  - Exact Number - e.g. 3..4 or 6
  - Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5



Relationship Between Classes- Aggregation

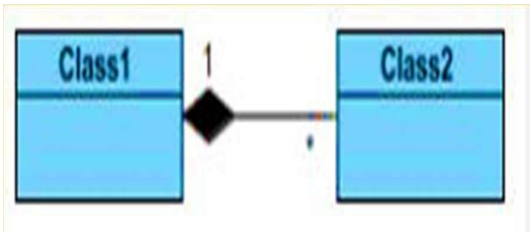
- Aggregation is a special type of association that models a whole- part relationship between aggregate and its parts.
  - For example, the class college is made up of one or more student.
- In aggregation, the contained classes are never totally dependent on the lifecycle of the container.
- Here, the college class will remain even if the student is not available.
- A solid line with an unfilled diamond at the association end connected to the class of composite

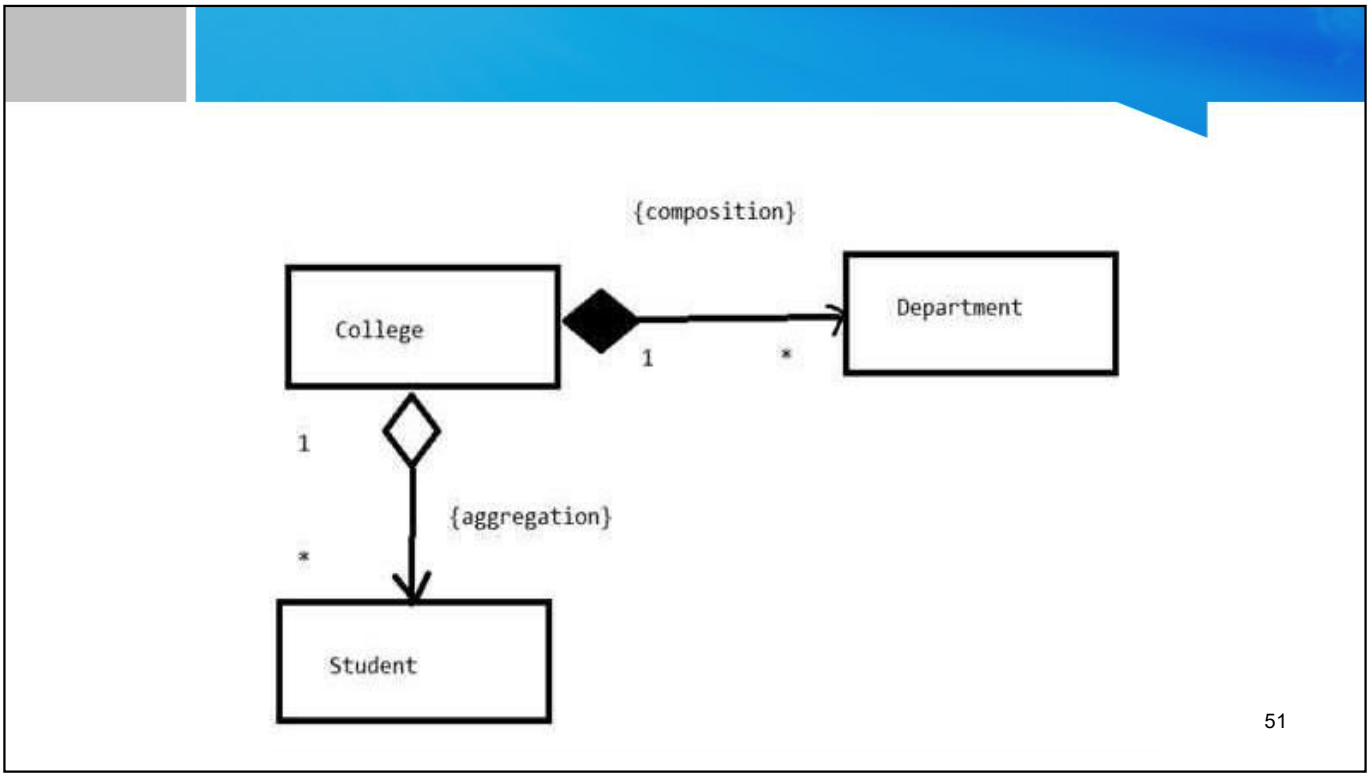


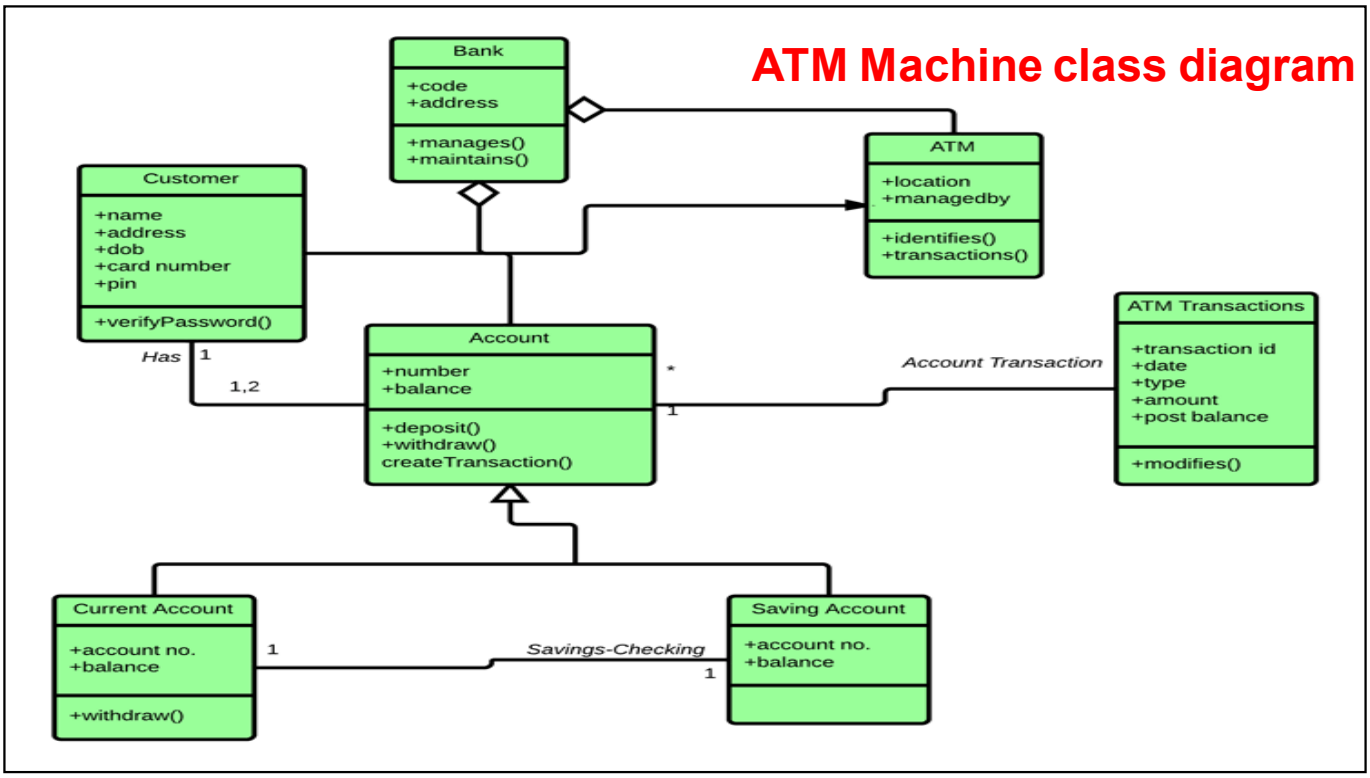


Relationship Between Classes- Composition

- The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.
  - A college has many departments and many students.
  - **The departments of the college can't exist without the college itself.**
  - This means "Departments" are owned by the college. Students can exist without the college, the **ownership of students are not in the college alone.**
  - Students can look for a new college if the college doesn't exist.
- A solid line with a filled diamond at the association connected to the class of composite.




{composition}
College
Department
1
*
1
*
{aggregation}
Student


ATM Machine class diagram
Bank
+code
+address
+manages()
+maintains()
ATM
+location
+managedby
+identifies()
+transactions()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
createTransaction()
*
1
Account Transaction
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Current Account
+account no.
+balance
+withdraw()
1
Savings-Checking
1
Saving Account
+account no.
+balance

USE CASE DIAGRAM

Use case diagram

- A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- Use cases once specified can be denoted both textual and visual representation (i.e. use case diagram).
- 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.

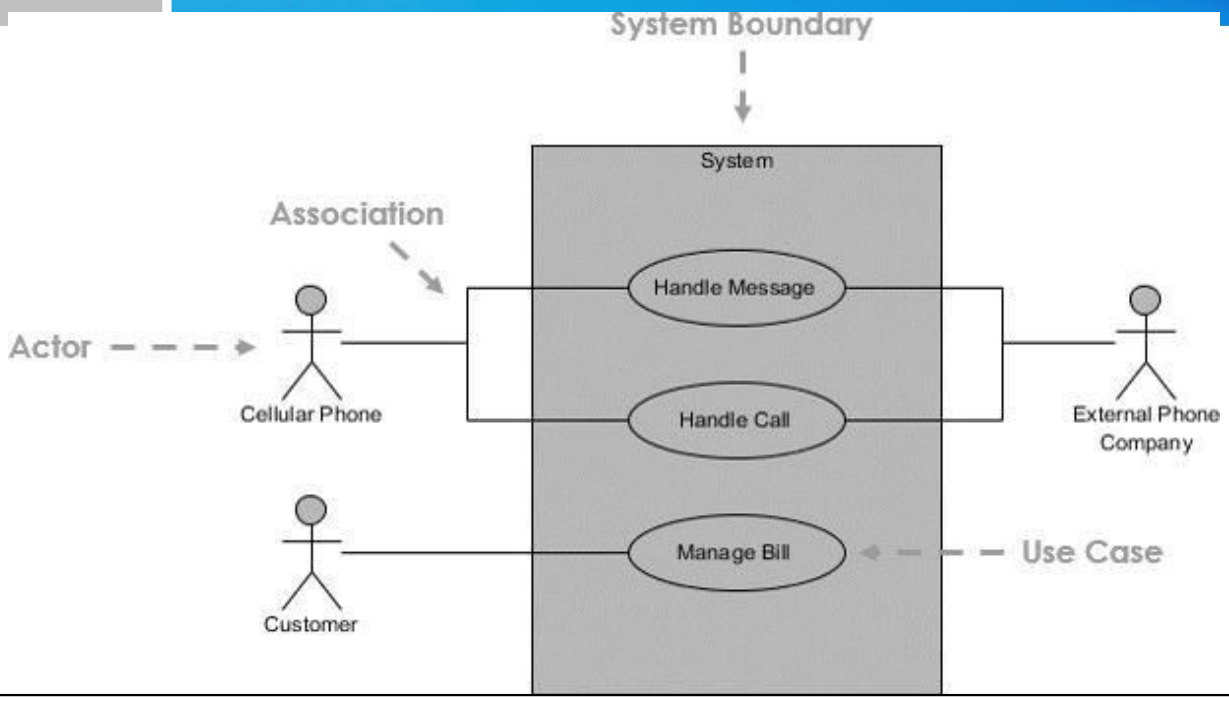
- It only summarizes **some of the relationships** between use cases, actors, and systems.
- It does **not show the order** in which steps are performed to achieve the goals of each use case.

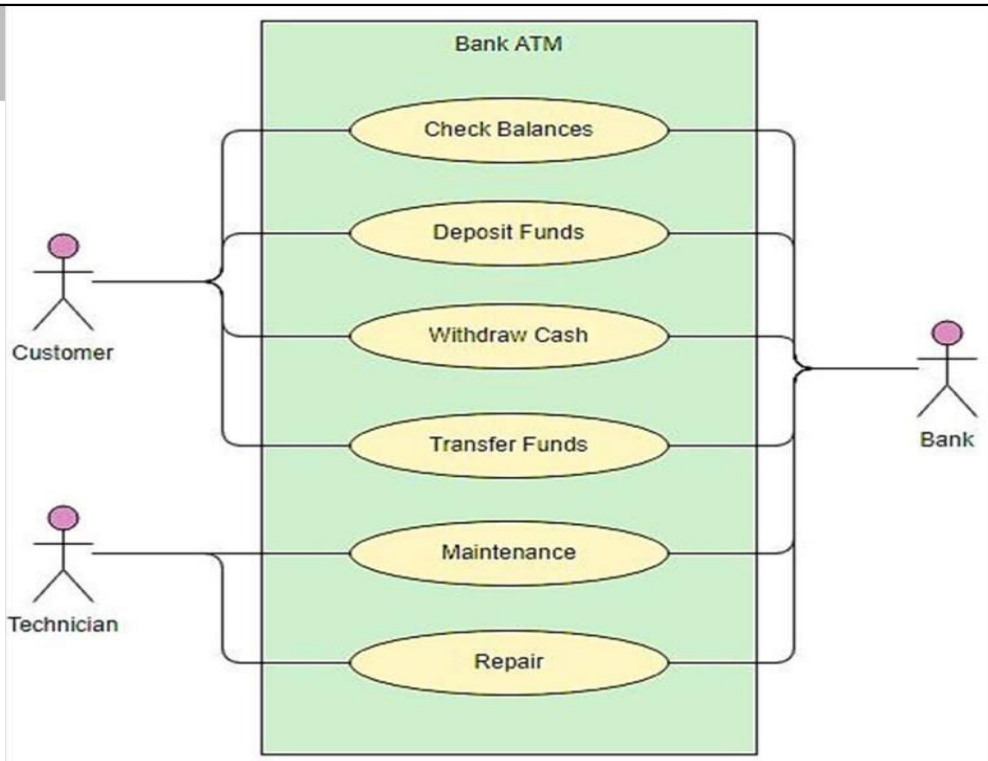
Orgin

- These days use case modeling is often associated with UML, although it has been introduced before UML existed. Its brief history is as follow:
  - In 1986, **Ivar Jacobson** first formulated textual and visual modeling techniques for specifying use cases.
  - In 1992 his co-authored book Object-Oriented Software Engineering - A Use Case Driven Approach helped to popularize the technique for capturing functional requirements, especially in software development.

Purpose

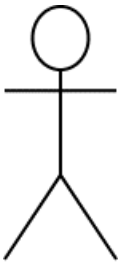
- Use case diagrams are typically developed in the early stage of development
  - Specify the context of a system
  - Capture the requirements of a system
  - Validate a systems architecture
  - Drive implementation and generate test cases
  - Developed by analysts together with domain experts





59

- 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).

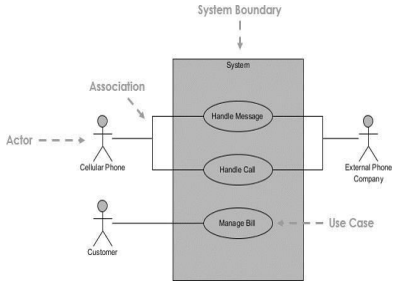


60

- Use Case
  - System function (process - automated or manual)
  - Named by verb + Noun (or Noun Phrase).
  - Each Actor must be linked to a use case, while some use cases may not be linked to actors.

- Communication Link/ Association
  - 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.

- 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.

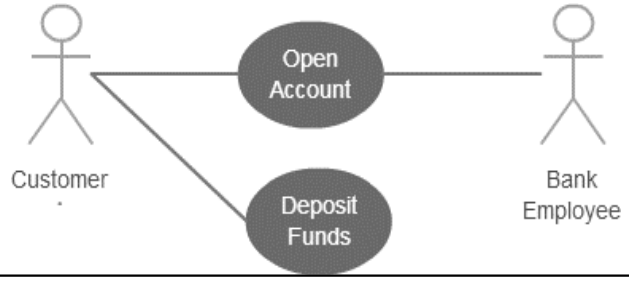


Structuring Use Diagram with Relationships

- There can be 5 relationship types in a use case diagram.
  - Association between actor and use case
  - Generalization of an actor
  - Extend between two use cases
  - Include between two use cases
  - Generalization of a use case

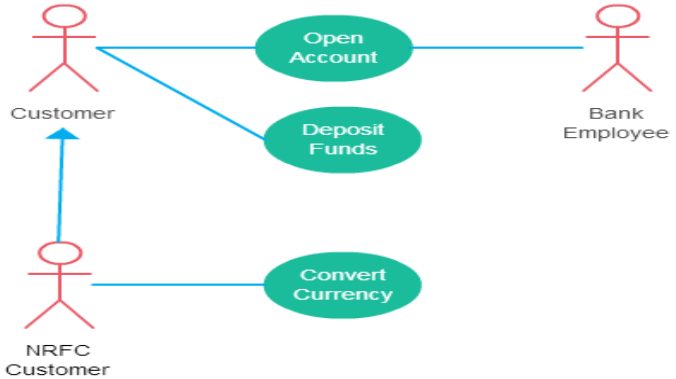
Association between Actors and usecase

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.



Generalization of Actors

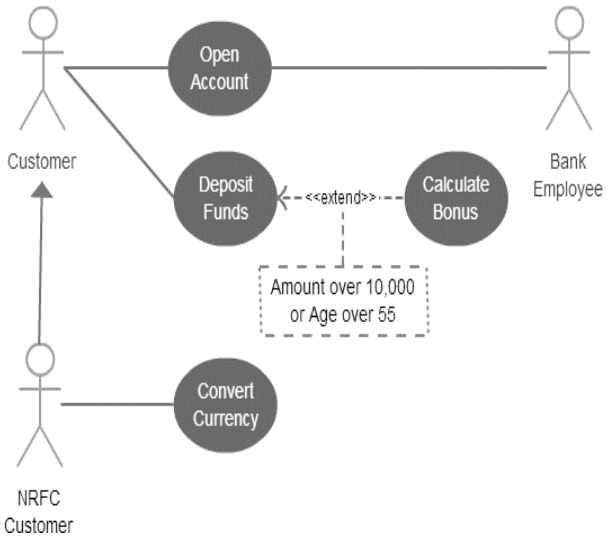
- Generalization of an actor means that one actor can inherit the role of the other actor.
- The descendant inherits all the use cases of the ancestor.
- The descendant has one or more use cases that are specific to that role.



A generalized actor in an use case diagram

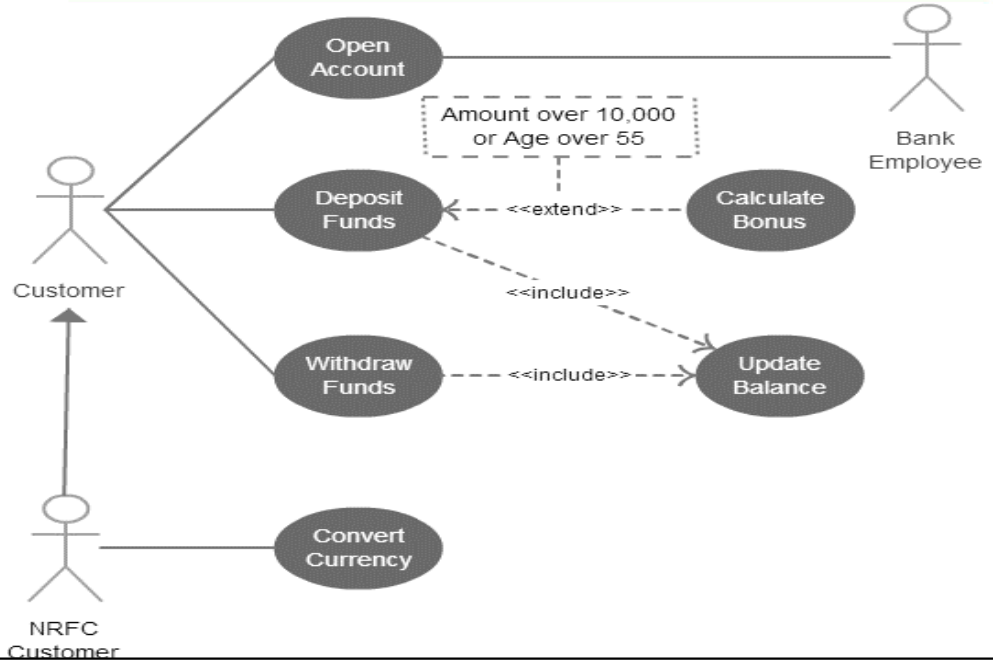
Extend Relationship between two Use Cases

- The <<extend>> use case inserting additional action sequences into the base use-case sequence.
- The extending use case is dependent on the extended (base) use case.
- The extending use case is usually optional and can be triggered conditionally.
- The extended (base) use case must be meaningful on its own.



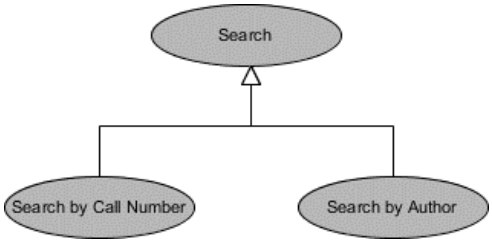
Include Relationship between two Use Cases

- The time to use the <<include>> relationship is after you have completed the first cut description of all your main Use Cases.
- Include relationship implies one use case includes the behaviour of another use case in its sequence of events and actions
- The main reason for this is to reuse the common actions across multiple use cases.
- In some situations, this is done to simplify complex behaviours.
  - The base use case is incomplete without the included use case.
  - The included use case is mandatory and not optional.



Generalization of a use case

- 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.



Use Cases

- 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.

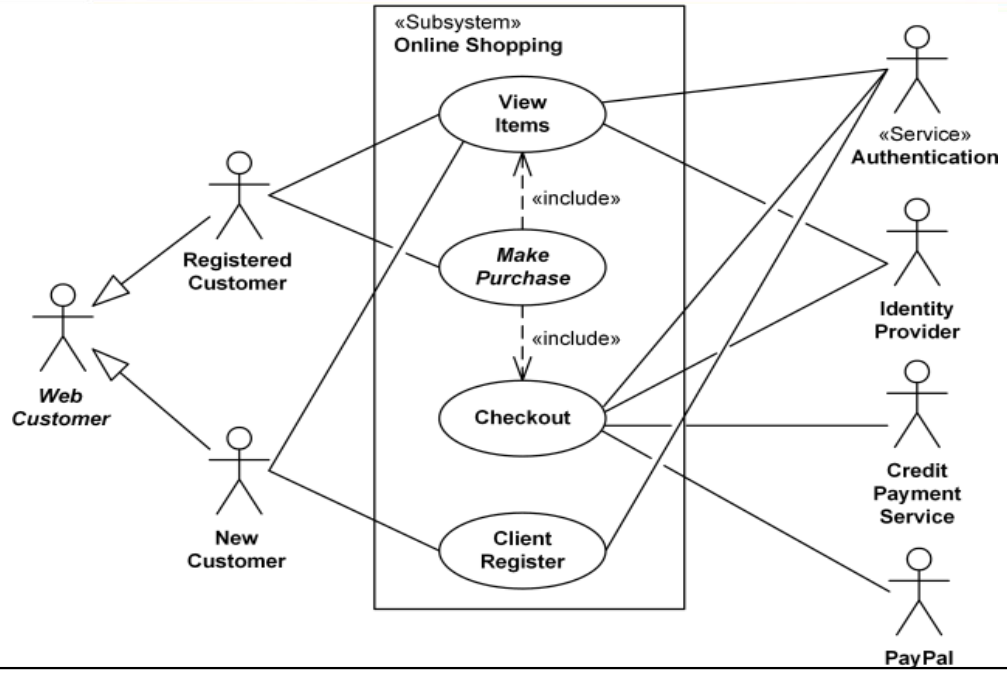
How to Identify Actors

- The following questions can help you identify the actors of your system (Schneider and Winters - 1998):
  - ✓ 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?

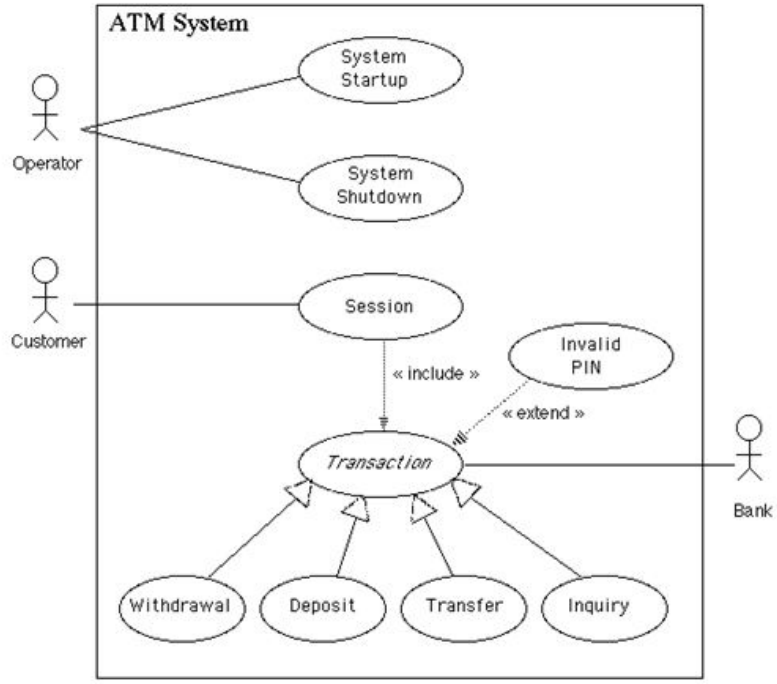
How to Identify Use Cases

- The following questions can be asked to identify use cases, once your actors have been identified (Schneider and Winters - 1998)
  - ✓ 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?

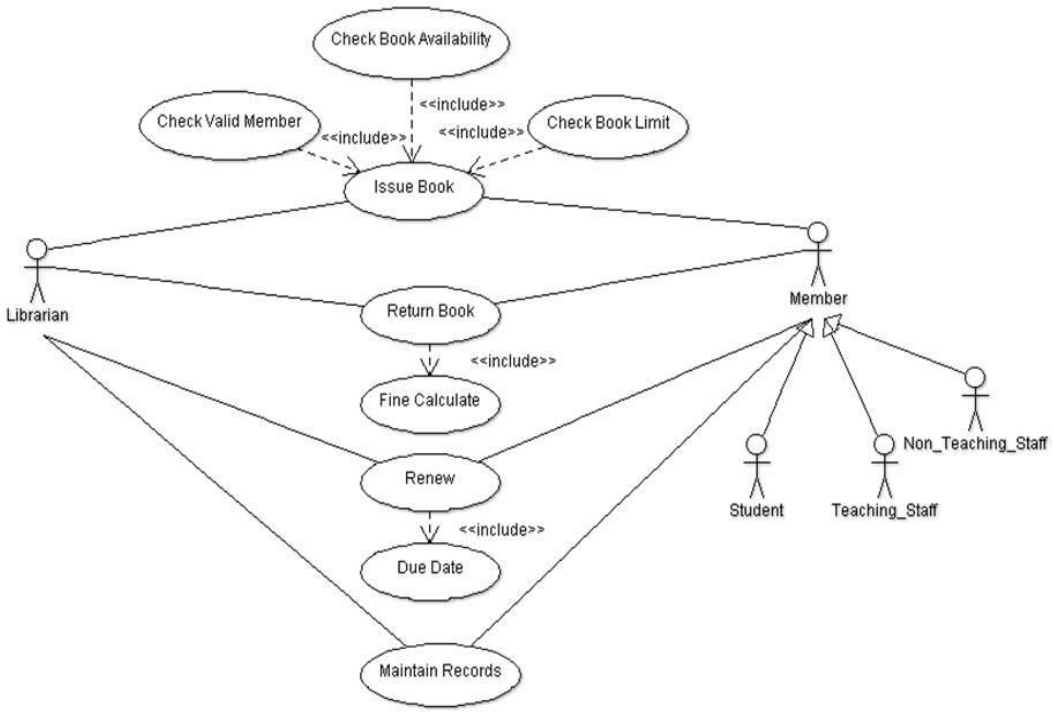
Use Case diagram example-Online Shopping



ATM System



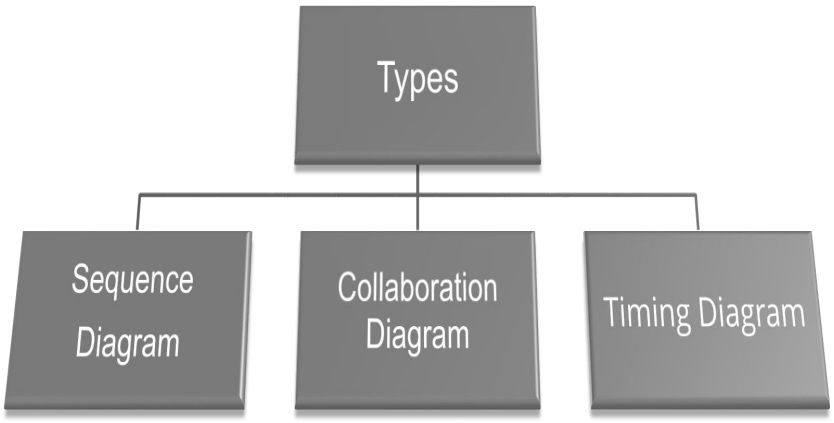
Library



INTERACTION DIAGRAM

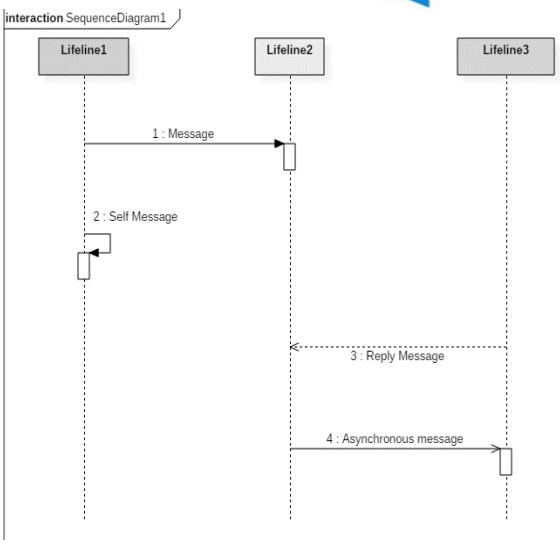
Interaction diagram

- Interaction diagram are used in UML to establish communication between objects.
- It does not manipulate the data associated with the particular communication path.
- Mostly focus on message passing and how these messages make up one functionality of a system.
- The critical component in an interaction diagram is lifeline and messages.
- Interaction diagrams capture the dynamic behavior of any system



Notation of Interaction Diagrams

- Lifeline
  - A lifeline represents a single participant in an interaction.
  - It describes how an instance of a specific classifier participates in the interaction.
    - Name(optional) - It is used to refer the lifeline within a specific interaction.
    - Type - name of a classifier of which the lifeline represents an instance.



Purpose of Interaction Diagrams

- Used to represent the dynamic behavior of a system
- Describes message flow in the system
- Visualizes the communication and sequence of message passing in the system
- Describe structural aspects of various objects in the system
- Represents ordered sequence of interactions within a system

Different types of Interaction Diagrams

1. Sequence diagram
   - Purpose - To visualize the sequence of a message flow in the system
   - Shows the interaction between two lifelines
2. Collaboration diagram
   - Also called as a communication diagram
   - Shows how various lifelines in the system connects.
3. Timing diagram
   - Focus on the instance at which a message is sent from one object to another object.

1. Sequence Diagram

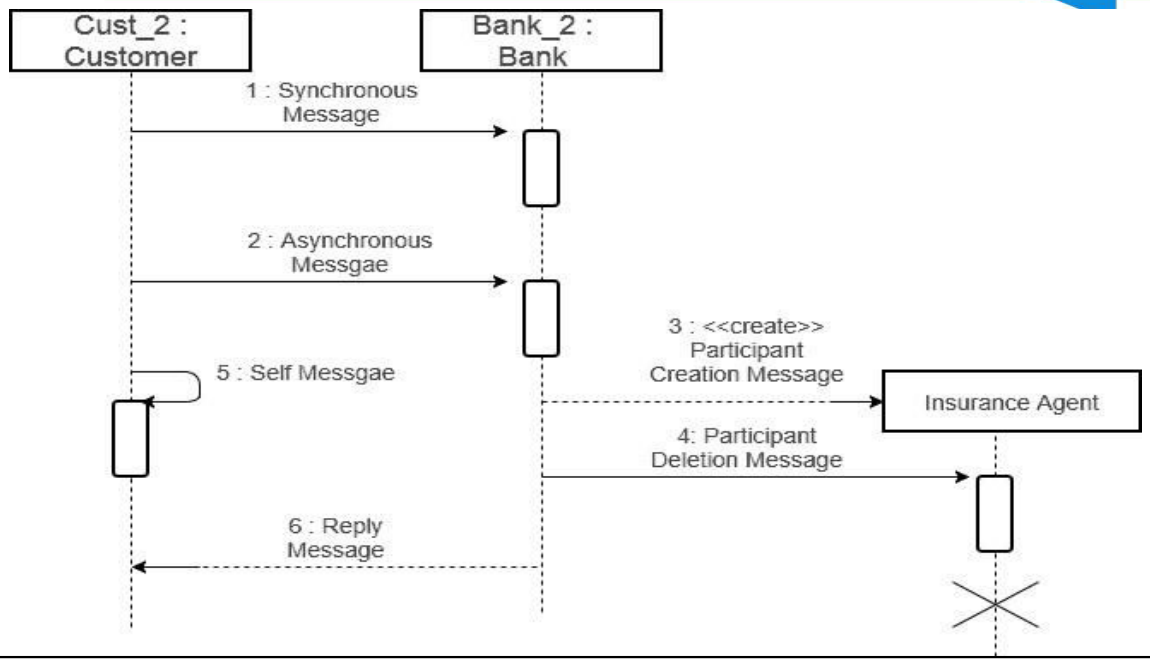
- A Sequence Diagram simply depicts interaction between objects in a sequential order.
- The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system
- Messages – Communication between objects is depicted using messages.
- The messages appear in a sequential order on the lifeline.

- In a sequence diagram, a lifeline is represented by a vertical bar.
- A lifeline represents an individual participant in a sequence diagram
- A lifeline will usually have a rectangle containing its object name
- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- In a sequence diagram, different types of messages and operators are used
- In a sequence diagram, iteration and branching are also used.

Messages used

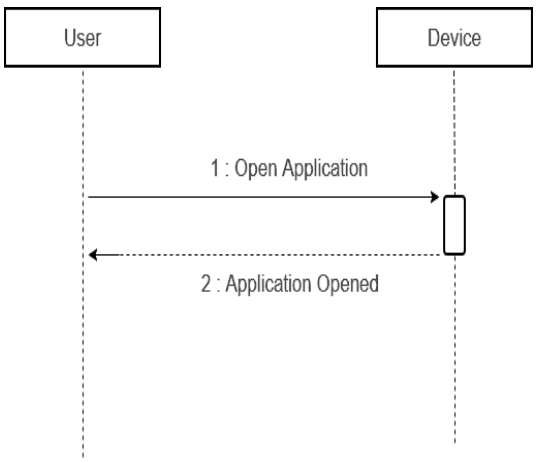
Message Name	Meaning
Synchronous message	The sender of a message keeps waiting for the receiver to return control from the message execution.
Asynchronous message	The sender does not wait for a return from the receiver; instead, it continues the execution of a next message.
Return message	The receiver of an earlier message returns the focus of control to the sender.
Object creation	The sender creates an instance of a classifier.
Object destruction	The sender destroys the created instance.
Found message	The sender of the message is outside the scope of interaction.
Lost message	The message never reaches the destination, and it is lost in the interaction.

Sequence Diagram- Example



Synchronous Messages

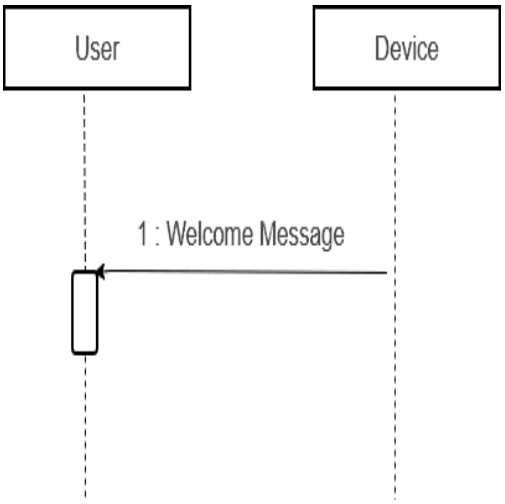
- A synchronous message waits for a reply before the interaction can move forward.
- The sender waits until the receiver has completed the processing of the message.
- The caller continues only when it knows that the receiver has processed the previous message.



87

Asynchronous Messages

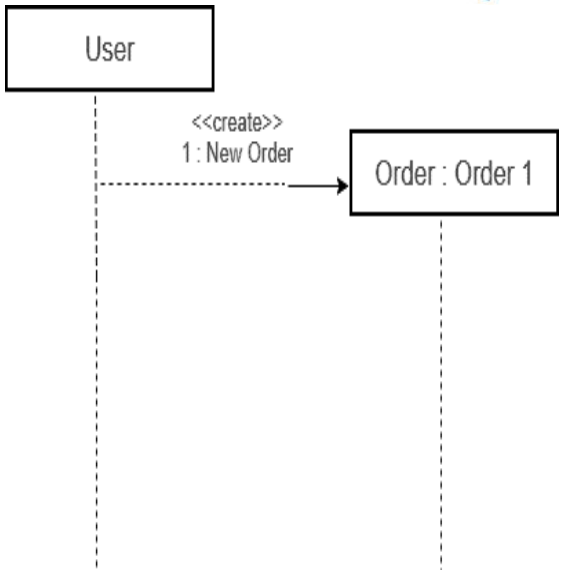
- An asynchronous message does not wait for a reply from the receiver.
- The interaction moves forward irrespective of the receiver processing the previous message or not.



88

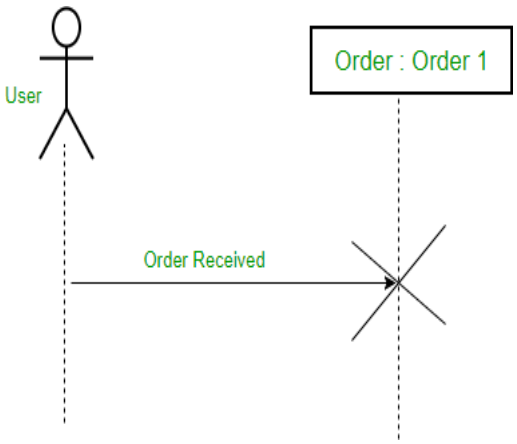
Create message

- We use a Create message to instantiate a new object in the sequence diagram.
- There are situations when a particular message call requires the creation of an object.



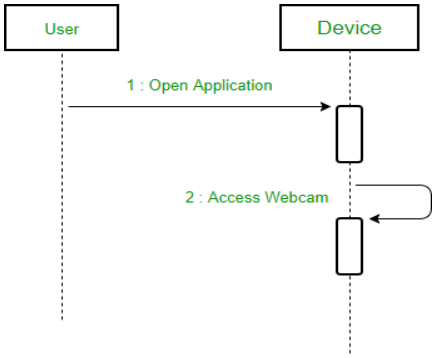
Delete Message

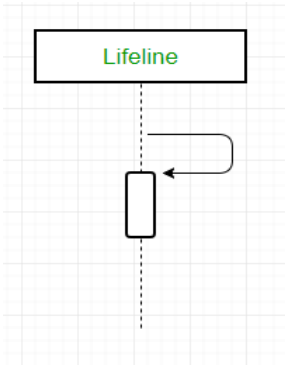
- We use a Delete Message to delete an object.
- When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol.
- It destroys the occurrence of the object in the system.



Self Message

- Certain scenarios might arise where the object needs to send a message to itself.

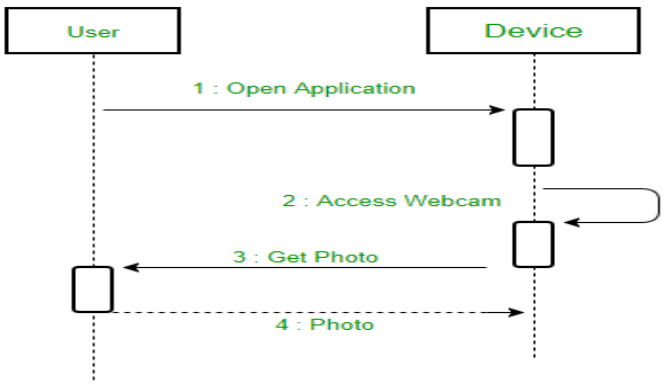




Device wants to access its webcam

Reply Message

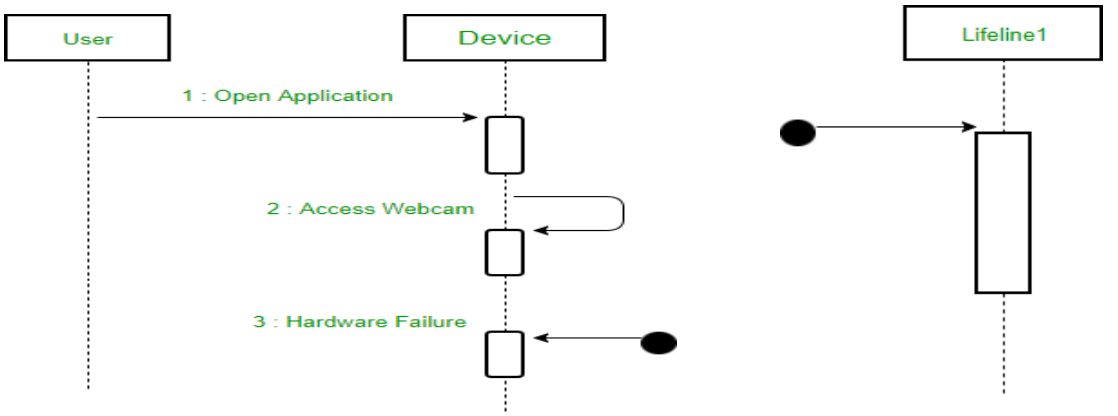
- Reply messages are used to show the message being sent from the receiver to the sender.



A scenario where a reply message is used

Found Message

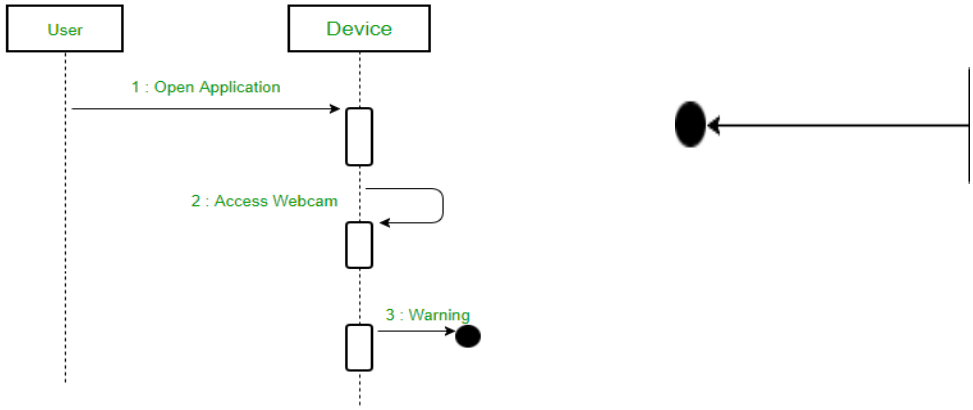
- A Found message is used to represent a scenario where an unknown source sends the message.



A scenario where a found message is used

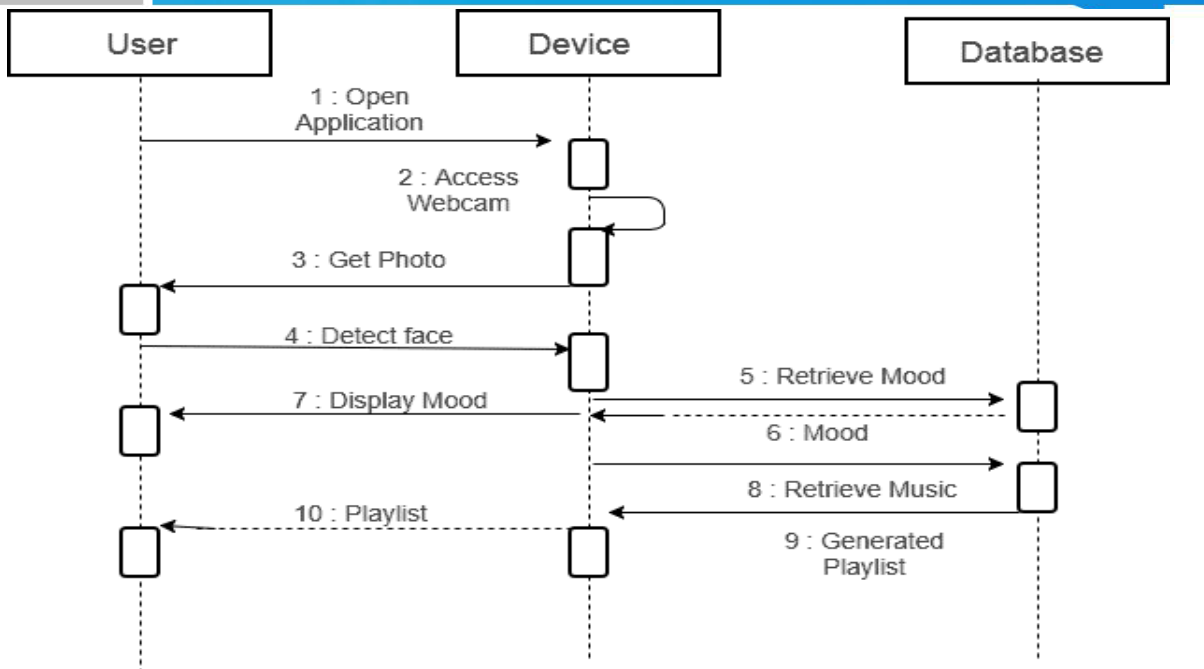
Lost Message

- A Lost message is used to represent a scenario where the recipient is not known to the system.



A scenario where a lost message is used

A sequence diagram for an emotion based Music Player



95

Benefits of a Sequence Diagram

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

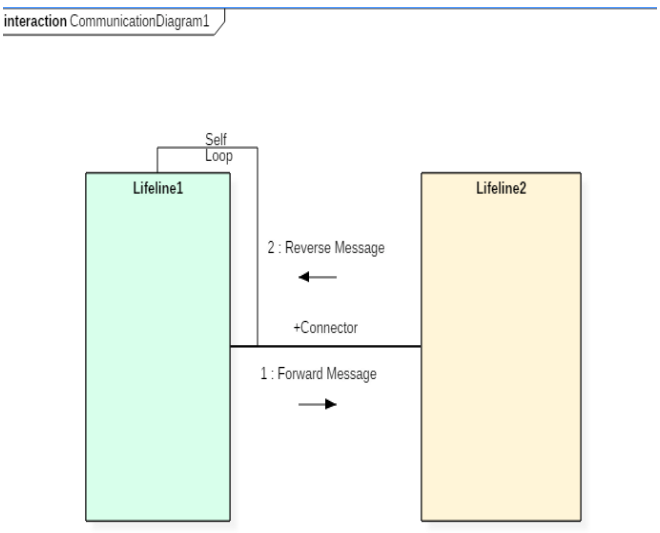
96

Drawbacks of a sequence diagram

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

2. Collaboration Diagram

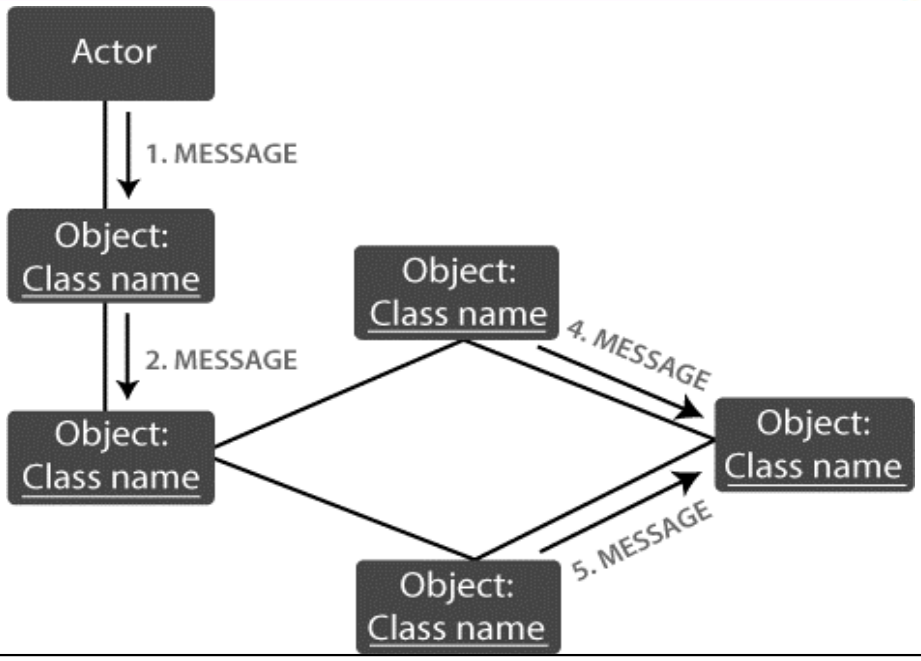
- Collaboration represents the relationships and interactions among software objects.
- They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram.

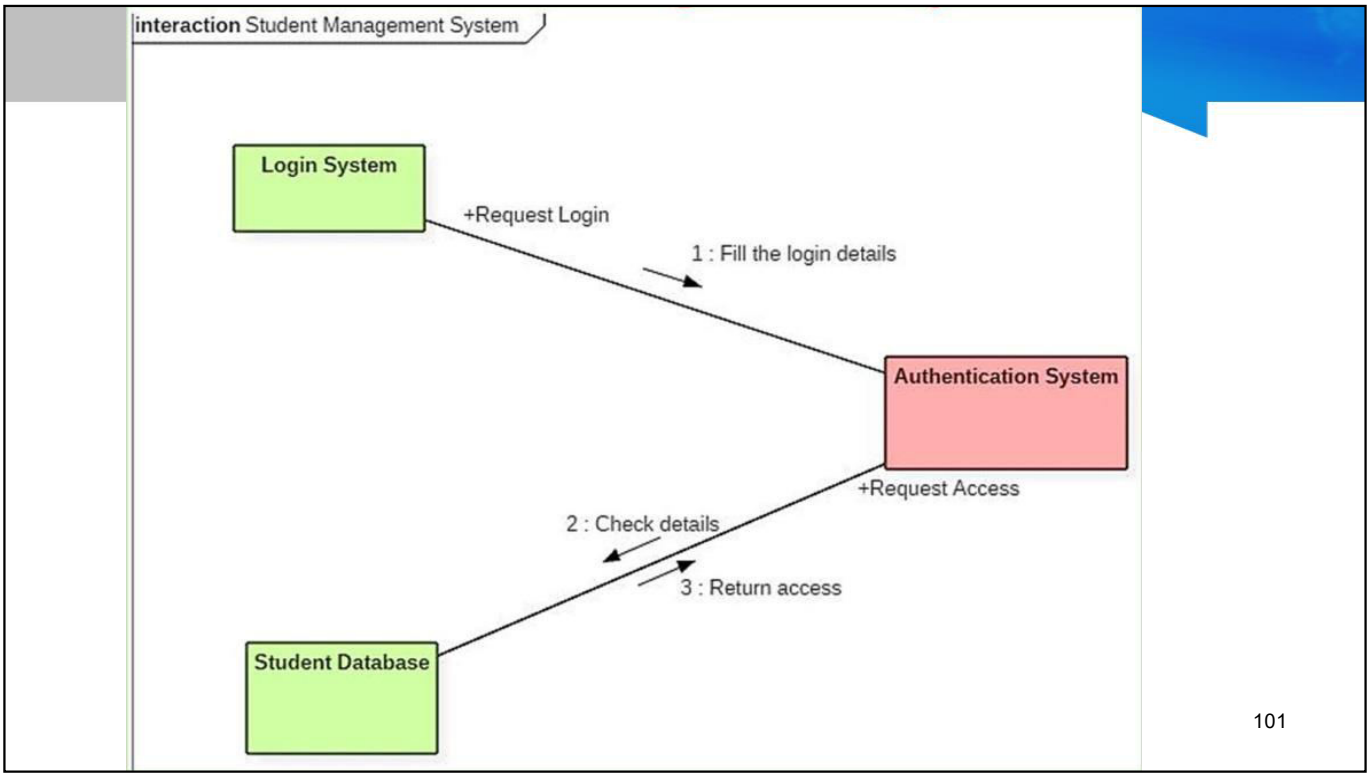


Benefits of Collaboration Diagram

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

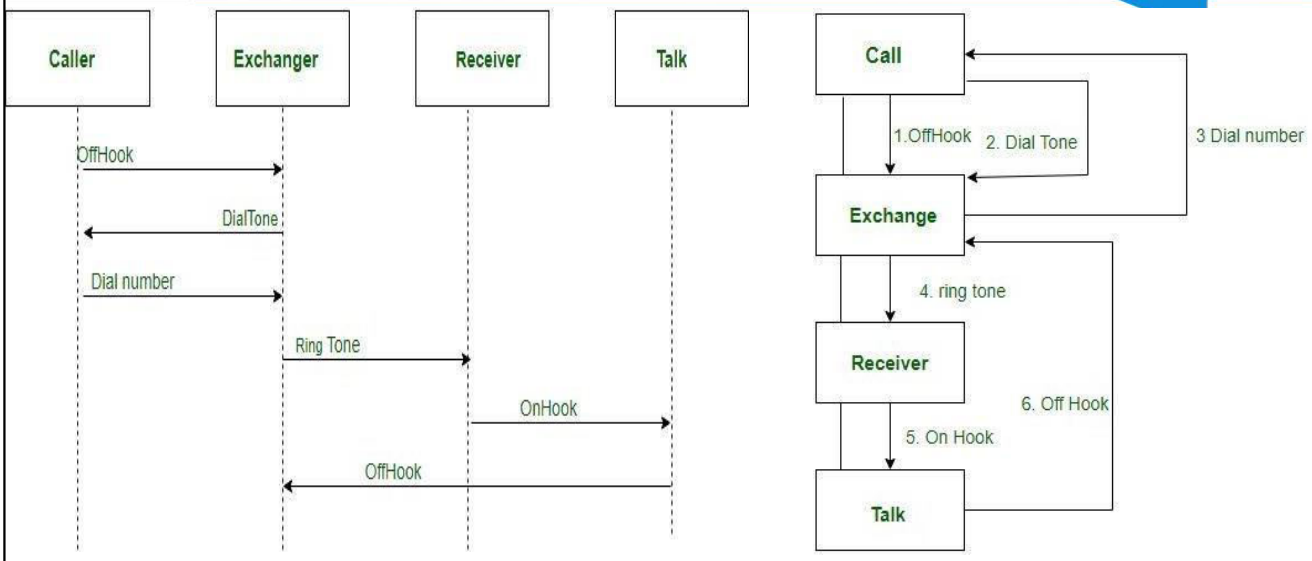
Components of Collaboration Diagram





101

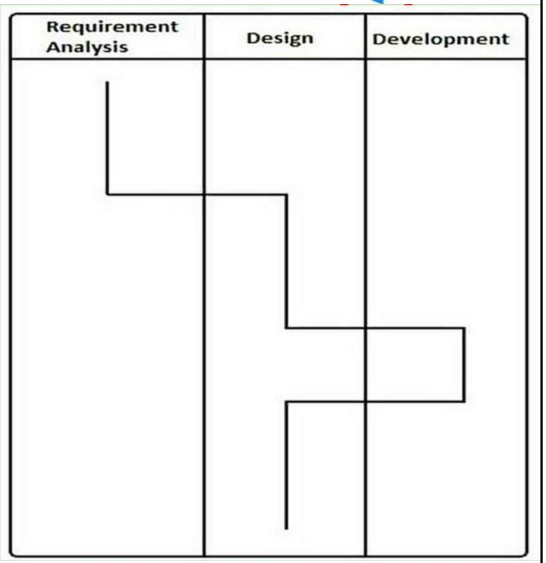
Difference between Sequence and Collaboration Diagram



102

3. Timing Diagram

- Timing diagrams are used to represent the state of an object at a particular instance of time.
- It is used to denote the transformation of an object from one form into another form.
- Timing diagram does not contain notations as required in the sequence and collaboration diagram.
- The flow between the software program at various instances of time is represented using a waveform.
- Timing diagram can be used to keep track of every change inside the system.



Difference between Sequence and Collaboration Diagram

Sequence Diagrams	Collaboration Diagrams
The sequence diagram represents the UML, which is used to visualize the sequence of calls in a system that is used to perform a specific functionality.	The collaboration diagram also comes under the UML representation which is used to visualize the organization of the objects and their interaction.
The sequence diagram are used to represent the sequence of messages that are flowing from one object to another.	The collaboration diagram are used to represent the structural organization of the system and the messages that are sent and received.
The sequence diagram is used when time sequence is main focus.	The collaboration diagram is used when object organization is main focus.
The sequence diagrams are better suited of analysis activities.	The collaboration diagrams are better suited for depicting simpler interactions of the smaller number of objects.

ACTIVITY DIAGRAM

ACTIVITY DIAGRAM

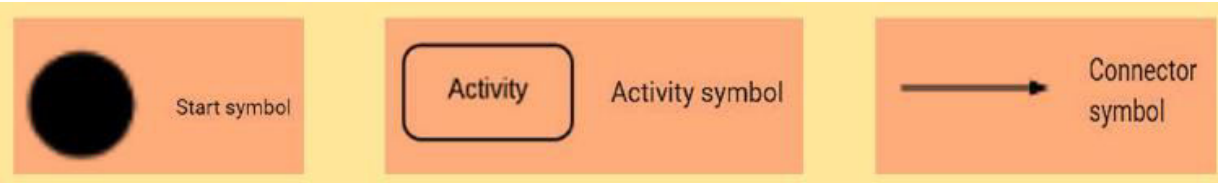
- ACTIVITY DIAGRAM is basically a flowchart to represent the flow from one activity to another activity.
- The activity can be described as an operation of the system
- The basic purpose of activity diagrams is to capture the dynamic behavior of the system
- It is also called object-oriented flowchart
- Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques.

Basic components of an activity diagram

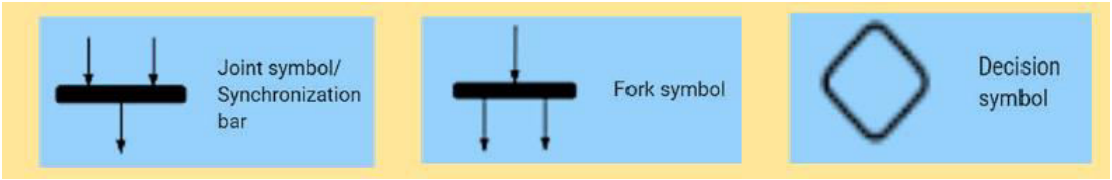
- Action: A step in the activity where in the users or software perform a given task.
- Decision node: A conditional branch in the flow that is represented by a diamond. It includes a single input and two or more outputs.
- Control flows: Another name for the connectors that show the flow between steps in the diagram.
- Start node: Symbolizes the beginning of the activity. The start node is represented by a black circle.
- End node: Represents the final step in the activity. The end node is represented by an outlined black circle.

Activity diagram symbols

- Start symbol - Represents the beginning of a process or workflow in an activity diagram.
- Activity symbol - Indicates the activities that make up a modeled process. These symbols, which include short descriptions within the shape, are the main building blocks of an activity diagram.
- Connector symbol - Shows the directional flow, or control flow, of the activity.

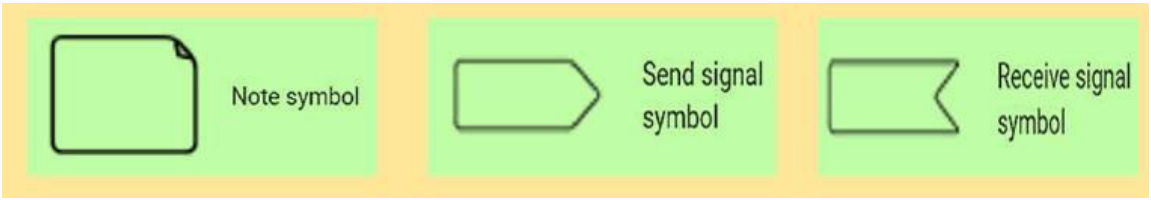


- Joint symbol / Synchronization bar - Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.
- Fork symbol - Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
- Decision symbol - Represents a decision and always has at least two paths branching out with condition text.



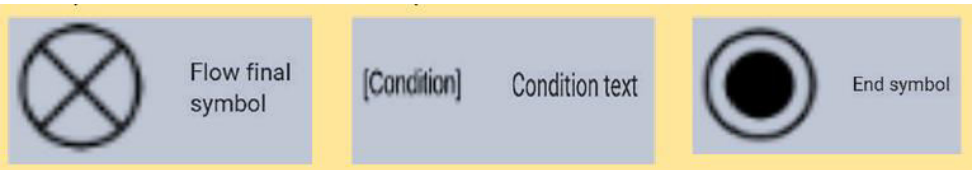
109

- Note symbol - Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.
- Send signal symbol - Indicates that a signal is being sent to a receiving activity
- Receive signal symbol - Demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed.



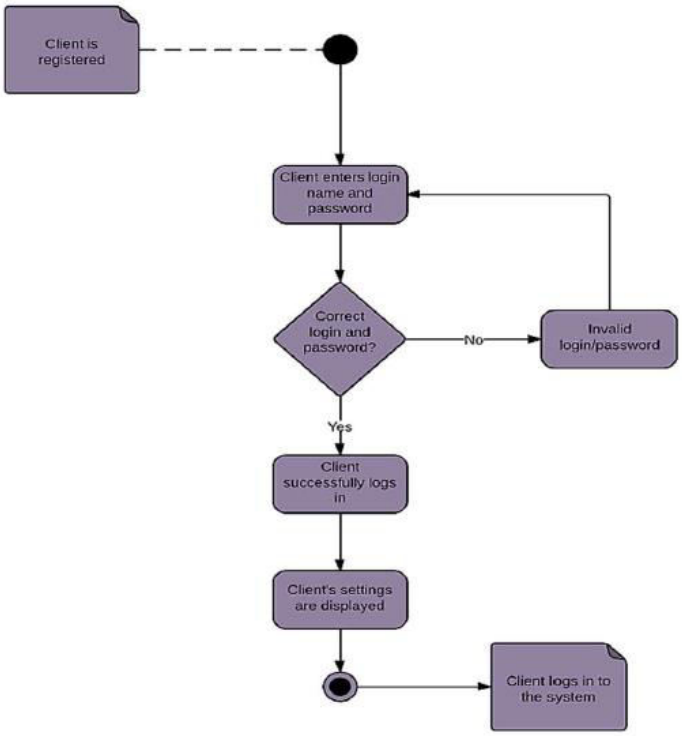
110

- Flow final symbol - Represents the end of a specific process flow. This symbol shouldn't represent the end of all flows in an activity. The flow final symbol should be placed at the end of a single activity flow.
- Condition text - Placed next to a decision marker to let you know under what condition an activity flow should split off in that direction
- End symbol - Marks the end state of an activity and represents the completion of all flows of a process.



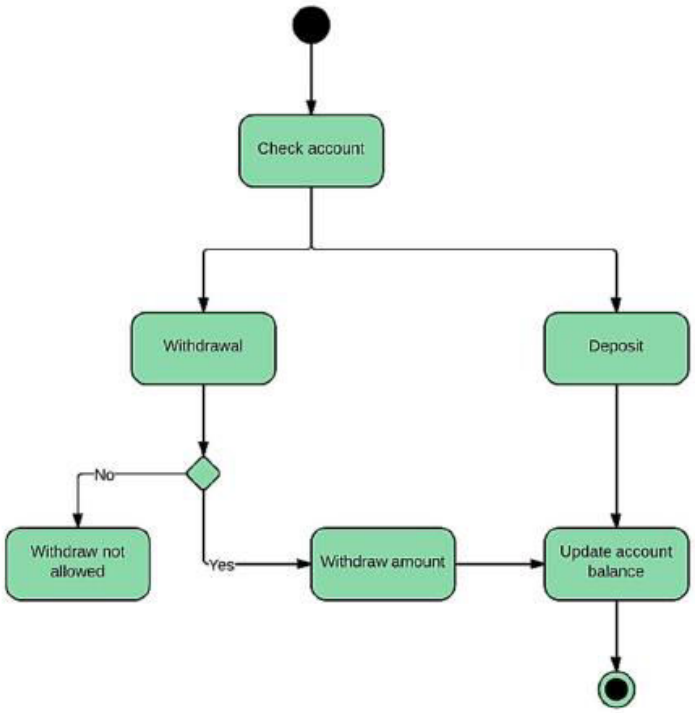
111

Activity diagram a login page



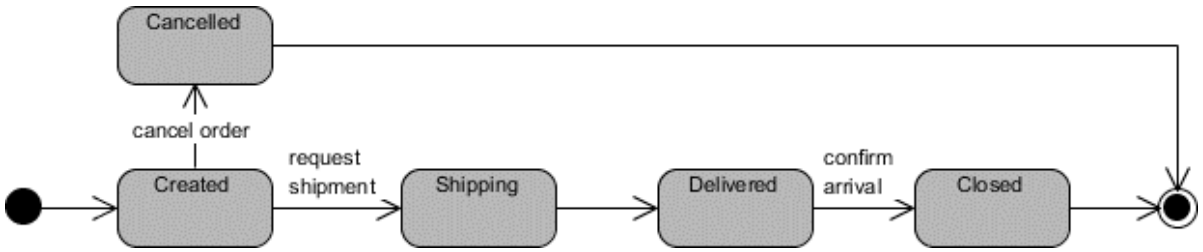
112

Activity Diagram - Banking system

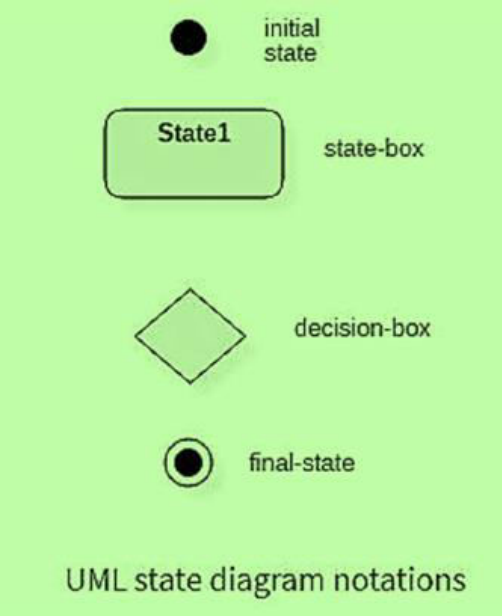


STATE CHART DIAGRAM

- The primary purpose of a state chart diagram is to model interactive systems and define each and every state of an object.
- State chart diagrams are also referred to as State machines and state diagrams.
- A state machine consists of states, linked by transitions. A state is a condition of an object in which it performs some activity or waits for an event



Notation and Symbol for State Machine



- Initial state - The initial state symbol is used to indicate the beginning of a state machine diagram.
- Final state - This symbol is used to indicate the end of a state machine diagram.
- Decision box - It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.
- Transition - A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

State box

- States represent situations during the life of an object.
- It is denoted using a rectangle with round corners.
- The name of a state is written inside the rounded rectangle.
- A state can be either active or inactive.
- When a state is in the working mode, it is active, as soon as it stops executing and transits into another state, the previous state becomes inactive, and the current state becomes active.