Unit 3: Dynamic Models, Diagrams and Architecture design and principles

Section 1: Activity Model

In UML, the activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.

- The activity diagram helps in envisioning the workflow from one activity to another.
- It puts emphasis on the condition of flow and the order in which it occurs. The flow can be sequential, branched, or concurrent, and to deal with such kinds of flows, the activity diagram has come up with a fork, join, etc.
- It is particularly useful when you know that an operation has to achieve a number of different things, and we want to model what the essential dependencies between them are, before we decide in what order to do them.
- Records the dependencies between activities, such as which things can happen in parallel and what must be finished before something else can start.

1.1 Purpose of Activity Model

- The purpose of the activity diagram is to model the procedural flow of actions that are part of a larger activity.
- In projects in which use cases are present, activity diagrams can model a specific use case at a more detailed level.
- However, activity diagrams can be used independently of use cases for modeling a business-level function, such as buying a concert ticket or registering for a college class.
- Activity diagrams can also be used to model system-level functions, such as how a ticket reservation data mart populates a corporate sales system's data warehouse.
- It shows sequential and concurrent flow of controls

1.2 Activities

The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that 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.

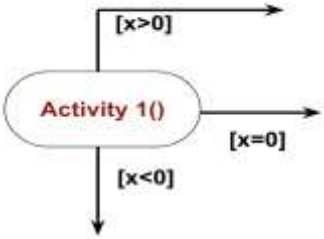
The control flow of activity is represented by control nodes and object nodes that illustrate the objects used within an activity. The activities are initiated at the initial node

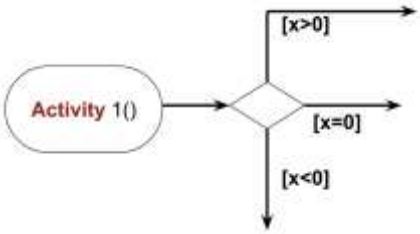
and are terminated at the final node.

They are graphically rendered as rounded rectangle.An action is indicated on the activity diagram by a "capsule" shape – a rectangular object with semi circular left and right ends. The text inside it indicates the action (e.g., Customer Calls Ticket Office or Registration Office Opens)

1.3 Decision Diamonds

These are used when the continuation of a flow depends on condition



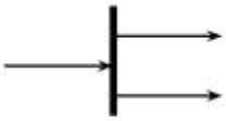


Without decision diamond

With decision diamond

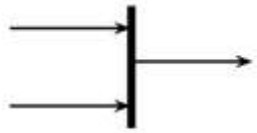
1.4 Join and Fork

- Use a synchronization bar to specify the forking and joining of parallel flows of control.
- A synchronization bar is rendered as a thick horizontal or vertical line.



Splitting Bar (Fork)

At the Fork, the activities of each outgoing transition are concurrent.A fork may have one incoming transition and two or more outgoing transitions. Each transition represents an independent flow of control. Conceptually, the activities of each outgoing transition are concurrent. Either truly concurrent (multiple nodes) or sequential yet interleaved (one node)



Synch. Bar (Join)

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

1.5 Start and Stop Activity

Start Marker

A solid circle with an outgoing arrow shows the starting point of the activity diagram so control starts at a solid circle.The initial state clearly shows the starting point for the action sequence within an activity diagram. Because activity diagrams show a sequence of actions, they must indicate the starting point of the sequence.

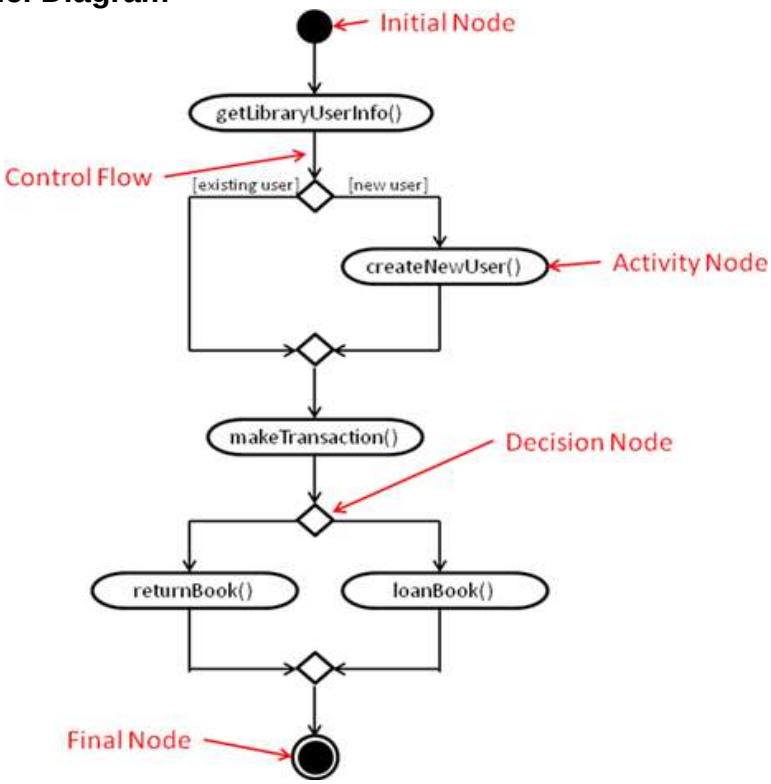
The official UML name for the starting point on the activity diagram is *initial state*, and it is the point at which you begin reading the action sequence.

It is important to note that there can *be only one* initial state on an activity diagram and *only one* transition line connecting the initial state to an action.

Stop Marker

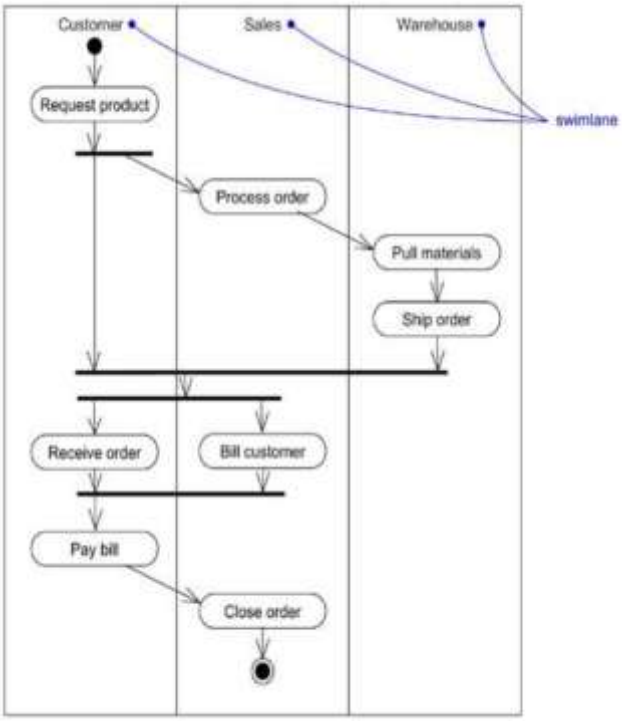
A bull's eye shows the termination point which has an incoming arrow.At bull's eye activity is completed & execution is completed.

1.6 Activity Model Diagram



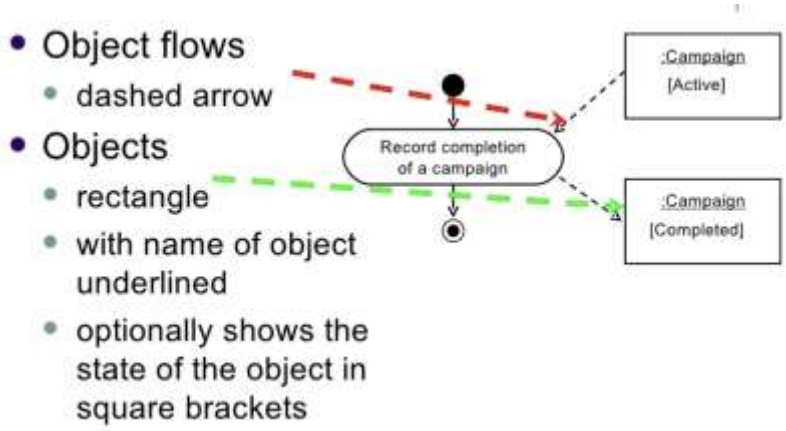
1.7 Swimlanes

The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram.To partition the activity states on an activity diagram into groups each group representing the business organization responsible for those activities each group is called a swimlane. It is not necessary to incorporate swimlanes in the activity diagram. But it is used to add more transparency to the activity diagram.



1.7 Objects and Object Flow

- Object flow refers to the creation and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object



1.8 Guidelines for Activity Diagram

- **Don't misuse activity diagram** :- Elaborate use case & sequence model to

study algorithm & workflow.

- **Level diagram :**- Activities on a diagram should be a consistent level of detail. place additional details for activity in a separate diagram.
- **Be careful with branches & conditions**:- At least one condition should be satisfied.
- **Be careful with concurrent activities** :- Before a merge can happen all inputs must first complete.

1.9 How to draw Activity Diagram

An activity diagram is a flowchart of activities, as it represents the workflow among various activities.

Since it incorporates swimlanes, branching, parallel flows, join nodes, control nodes, and forks, it supports exception handling. A system must be explored as a whole before drawing an activity diagram to provide a clearer view of the user. All of the activities are explored after they are properly analyzed for finding out the constraints applied to the activities. Each and every activity, condition, and association must be recognized.

After gathering all the essential information, an abstract or a prototype is built, which is then transformed into the actual diagram.

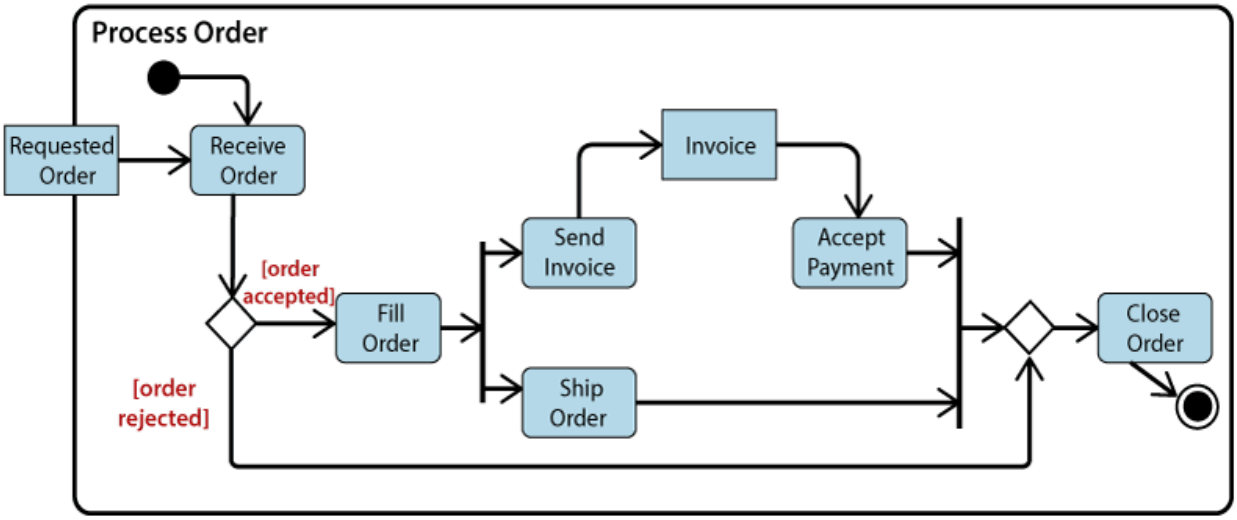
Following are the rules that are to be followed for drawing an activity diagram:

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

1.10 Example of Activity Diagram

An example of an activity diagram showing the business flow activity of order processing is given below.

Here the input parameter is the Requested order, and once the order is accepted, all of the required information is then filled, payment is also accepted, and then the order is shipped. It permits order shipment before an invoice is sent or payment is completed.



Section 2: State Model

- The state of an object is a condition or a situation during the life of an object during which it satisfies some condition, performs some activity or waits for an event
- When an event occurs some activity will take place based on the current state of the object
- Activity which is an ongoing non atomic execution within the state machine, results in some action which could be made of some atomic computation that results in the change in the state of the model or a return of a value
- State diagrams can be visualized as representation of potential states of the objects and the transitions among those states.
- An object must be in some specific state at any given time during its lifecycle.
- An object transitions from one state to another as a result of some event that affects it.
- There can be only one start state in a state diagram, but there may be many intermediate and final states
- We take the approach of looking at the different terms and concepts which are part of this and look at state diagrams.

2.1 UML Notations of State Diagram

- **State**

  Is shown as a rectangle with rounded corners.Has a name that represents the state of the object

- **An initial state**

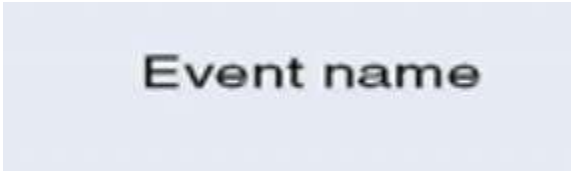
  Is shown as a small filled in circle.Represent the point at which an object begins to exist

- **A final state**

  Is shown as a circle surrounding a small filled-in circle(bull's eye).Represents the completion of an activity

- **An event**

  Is an occurrence that triggers a change in state.Can be a designated condition becoming true, a receipt of an explicit signal from one subject to another, or the passage of a designated period of time. Is used to label a transition.



- **Transition**

Indicates that an object in the first state will enter the second state. Is triggered by occurrence of the event labeling the transition.Is shown as a solid arrow from one state to another,labeled by the event name

2.3 Events

- An Event is an occurrence at a given point in time. It often corresponds to a verb in past tense or on the onset of a condition other E.g. User depresses a button, customer reports a problem, shipment arrives, phone taken off hook, paper try becomes empty, temperature falls below zero
- A event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another
- These events could be
    - Normal as well as error conditions
    - External or internal (e.g. page loaded)
- Concurrent and Related Events
    - Related events if one event logically or causally follows another
    - If the events are causally unrelated they are concurrent events.

Types of events:

2.3.1 Signal Event

- Signal – One way transmission of information/message from one object to another
- An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.
- ***Signal Event*** is the event of sending or receiving a signal
- Contrasting signal and signal event, signal is a message between objects, whereas the signal event is an occurrence in time
- Signal occurrences are unique, but are grouped to ***signal classes*** with a class name to indicate common structure and behavior. They also have attributes

indicating values they convey
- They are represented in UML as <<signal>> on top of class name

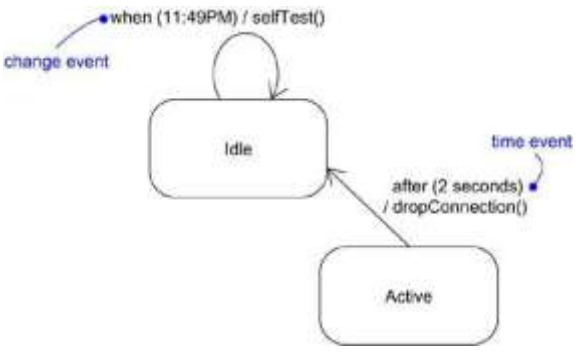
2.3.2 Change Event

- Caused by satisfaction of Boolean expression
- Modeled using
    - when (expression)
- Expression below is continuously (practically) tested & whenever the expression evaluates to true the event happens
- Generally detected by polling or other application logic

2.3.3 Time Event

- Represents elapse of time or absolute time
- Modeled using
    - when (expression)
    - after (expression)
- UML notation for an absolute time is the keyword when followed by a parenthesized expression involving time.
- Eg:

    when (date = November 1, 2011)
    after (10 seconds)



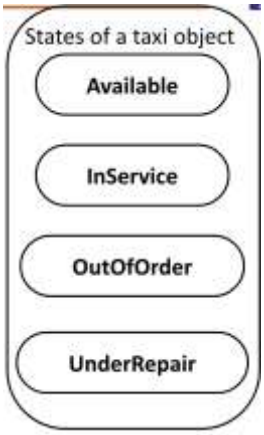
2.4 State

- State is an abstraction of values and links of an object (some internal

condition).

- Objects of a class have a finite number of possible states. Each object can be in one state at a time.
- State determines response of the object to input events. All inputs other than ones which are defined for are ignored
- State of the object depends on the past events which in most cases are eventually hidden by subsequent events
- Events correspond to a specific point in time and states represent intervals of time between two events received by the object.

Example:



2.5 Transition and Conditions

- **Transition**: an instantaneous change in state from one state to another
  - triggered by an event
  - Transition is said to fire upon the change from source to target state
  - Original and target state of a transition depends on the original state and could be different or the same.
  - e.g. when a phone line is answered, the phone line transitions from the ***Ringing*** state to the ***Connected*** state.
- **Guard Condition**:
  - boolean expression that must be true for transition to occur
  - checked only once, at the time event occurs; transition fires if true
  - e.g. when you go out in the morning ***(event)***, if the temperature is below freezing ***(condition)***, then put on your gloves ***(next state)***.

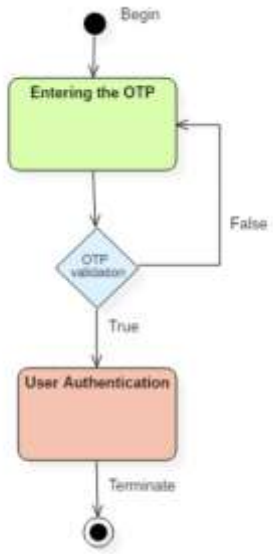
2.5.1 Difference between guard condition and change event

Guard Condition	Change Event
It is checked only once	Checked continuously
UML notation is a transition line	UML notation may include event label in italics
It is followed by guard condition in square brackets	An arrowhead points to target state from origin state

2.6 State Diagram

- State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions
- It represents
    - State sequences caused by event sequences
    - State names must be unique within the scope of a state diagram
    - Common behavior of all instances of the class

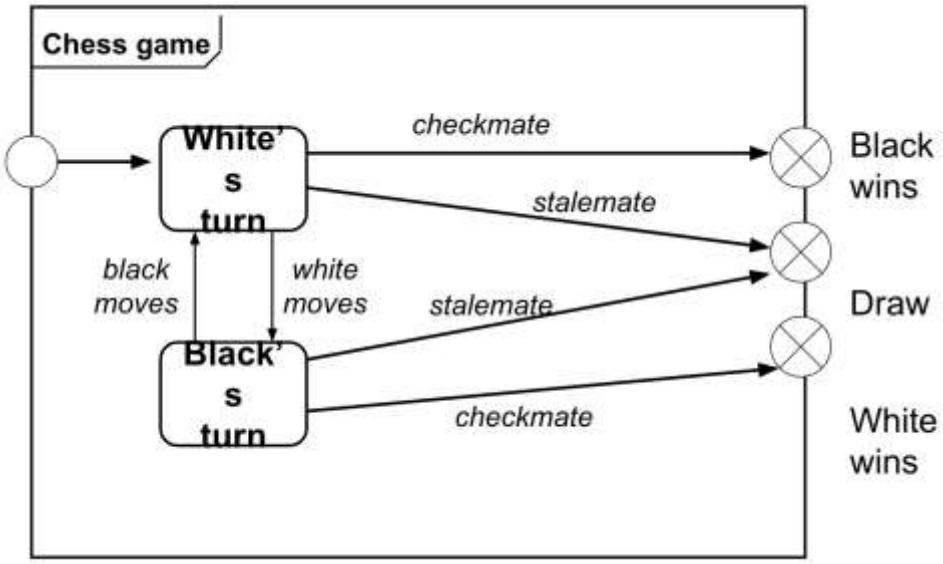
Example of User Authentication Process



2.6.1 One-shot state diagrams

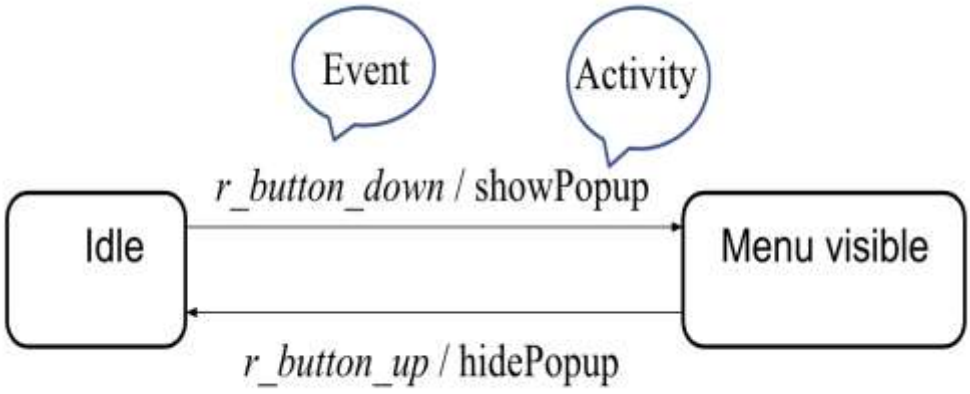
- State diagrams can represent continuous loops or one-shot life cycles. Eg: Diagram for the phone line is a continuous loop
- One – shot state diagrams represent objects with finite levels and have initial and final states. The initial state is entered on creation of an object ;entry of the final state implies destruction of the object.
- Object created in a state and change of the state deletes the object.

- As an alternate notation, you can indicate initial and final states via **entry** and **exit** points. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.



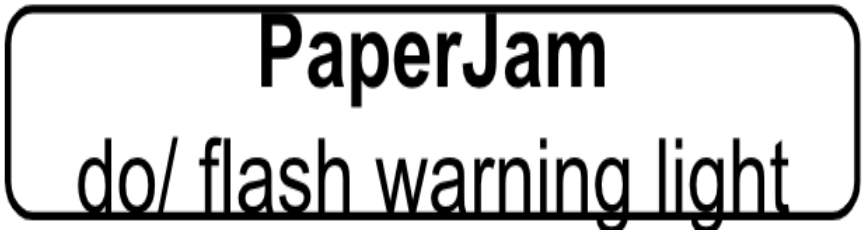
2.7 Activities

- This refers to the behavior of the object with state change or the activity (an action of an activity) that is invoked in response to an event. An activity is the actual behavior that can be invoked by any number of effects.
- An activity may be performed upon
    - a transition
    - the entry to or exit from a state
    - some other event within a state



2.7.1 Types of Activities

- **Do-Activities**
    - Continue for an extended time
    - Can occur only within a state
    - Cannot be attached to a transition
    - Continuous operations, such as displaying a picture on a television screen.
    - Sequential operations that terminate by themselves after an interval of time



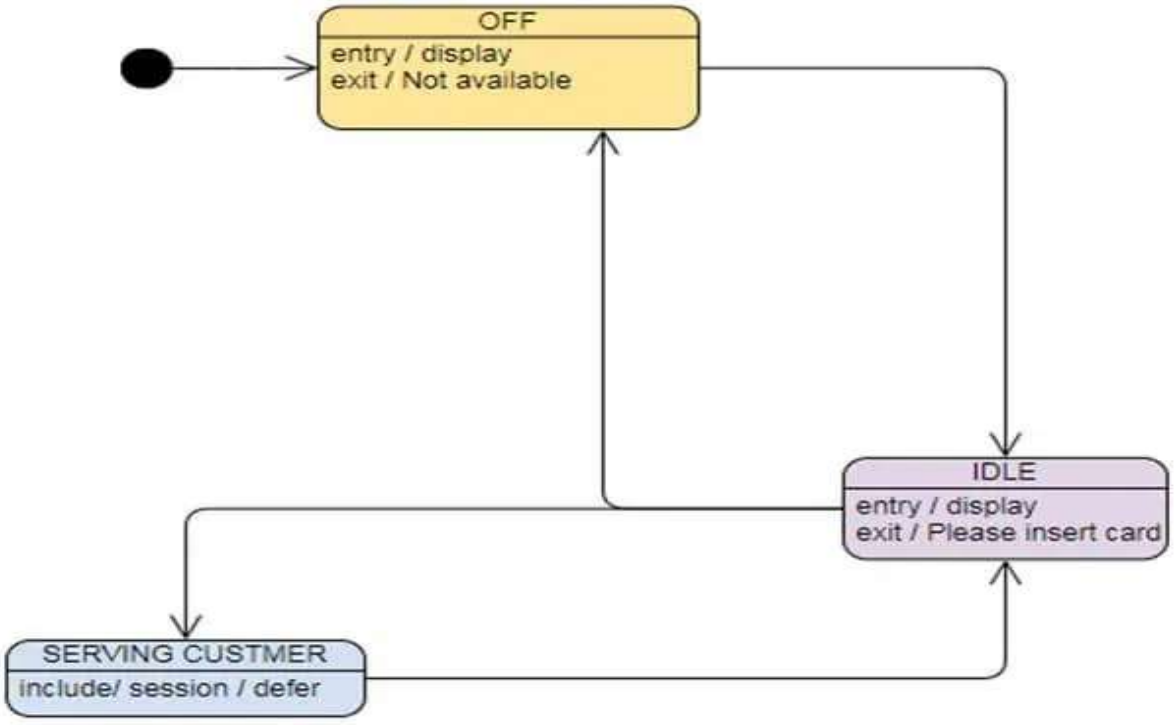
- **Entry and Exit Activities**
    - can bind activities to entry to/ exit from a state
    - All transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state.

2.7.2 Order of Activities

1. activities on incoming transition
2. entry activities
3. do-activities
4. exit activities
5. activities on outgoing transition

Note: Events that cause transitions out of the state can interrupt do-activities. If a do-activity is interrupted, the exit activity is still performed

Example of ATM



Section 3: Sequence Model

- Sequence diagrams are derived from Use Cases
- A Use Cases describes a particular execution of the system by an actor
- A sequence diagram depicts the interactions that are needed between objects to implement a particular use case
- It represents the flow of messages in the system and is also termed as an event diagram.
- It helps in envisioning several dynamic scenarios. It portrays the communication

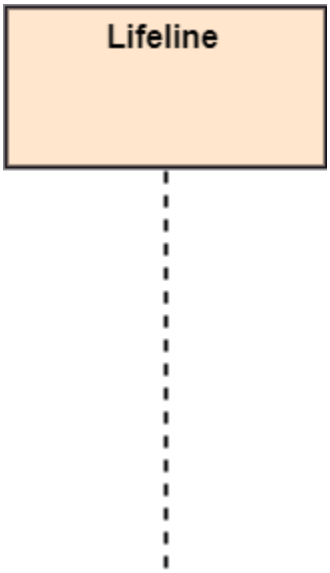
between any two lifelines as a time-ordered sequence of events, such that these lifelines took part at the run time.

- In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page.
- It incorporates the iterations as well as branching.

3.1 UML Notations

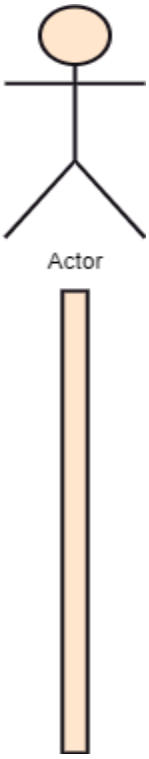
Lifeline

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



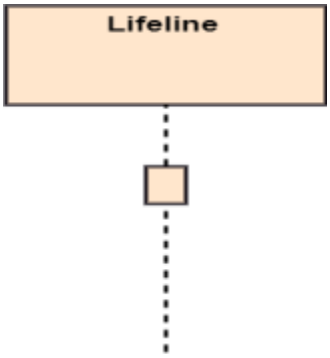
Actor

A role played by an entity that interacts with the subject is called an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.



Activation

It is represented by a thin rectangle on the lifeline. It describes the time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, respectively.

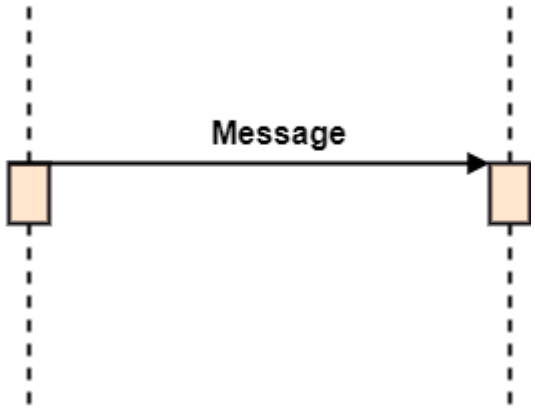


Messages

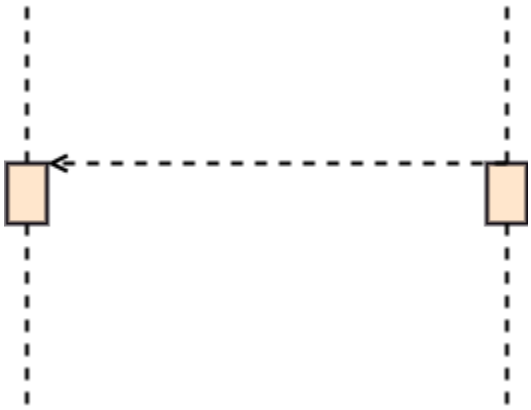
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

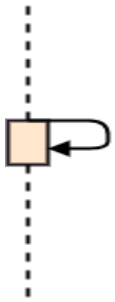
- **Call Message**: It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



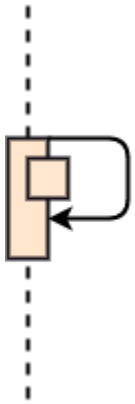
- **Return Message**: It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



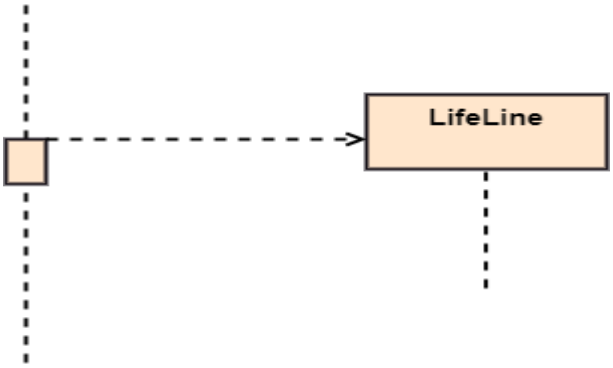
- **Self Message**: It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



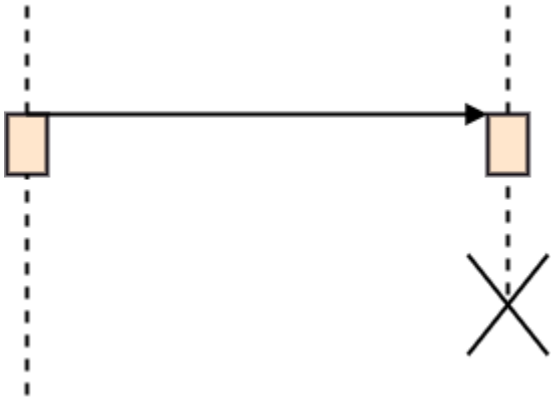
- **Recursive Message**: A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



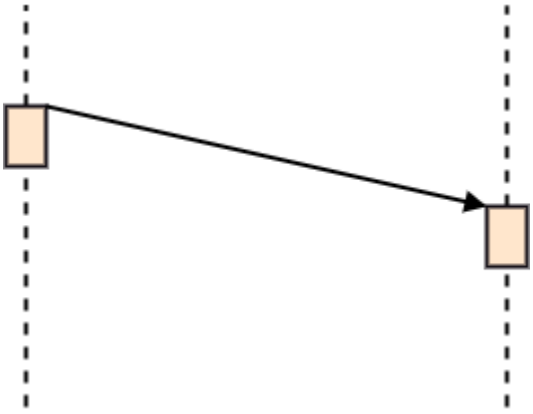
- **Create Message**: It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **Destroy Message**: It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.

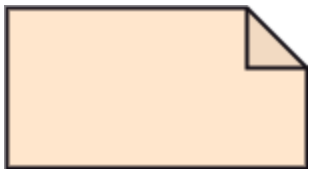


- **Duration Message**: It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.



Note

A note is the capability of attaching several remarks to the element. It basically carries useful information for the modelers.

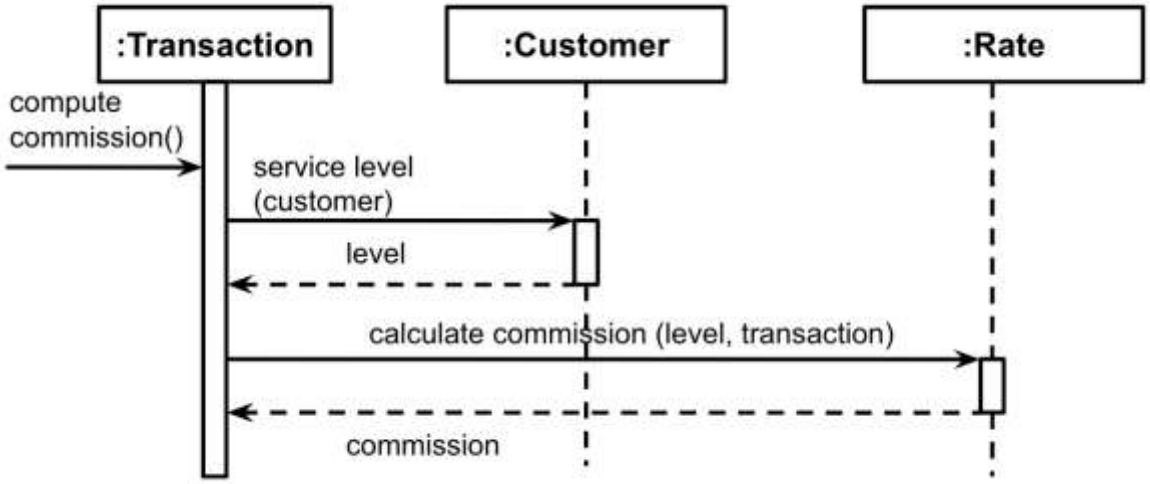


3.2 Sequence Diagram guidelines

- Prepare at least one scenario per use case
- Steps should be logical commands
- Abstract the scenarios into sequence diagrams
- Divide complex interactions
- Prepare a sequence diagram for each error conditio

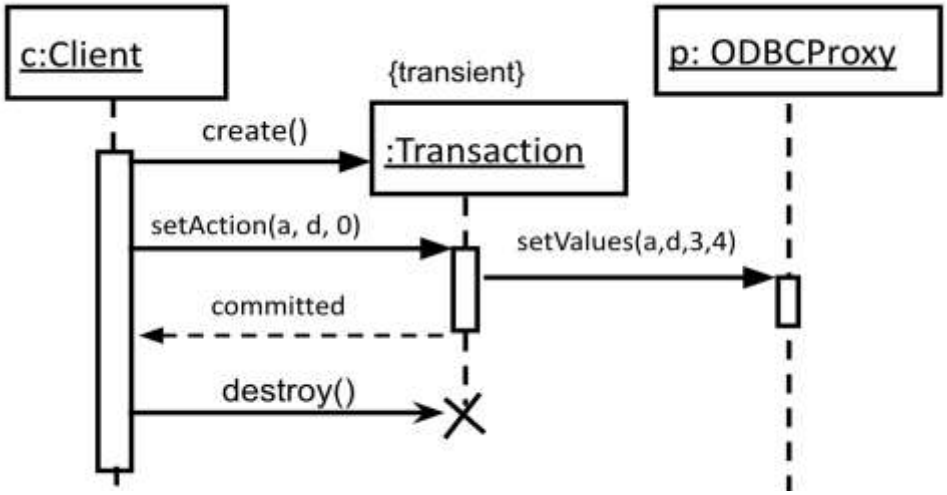
3.3 Passive Objects

- Not constantly active
- Activated only when called
- On return of control, becomes inactive
- Period of time for objects execution shown as a thin rectangle



3.4 Transition Objects

Created and destroyed within the scenarios



3.5 Fragments of Sequence Diagram

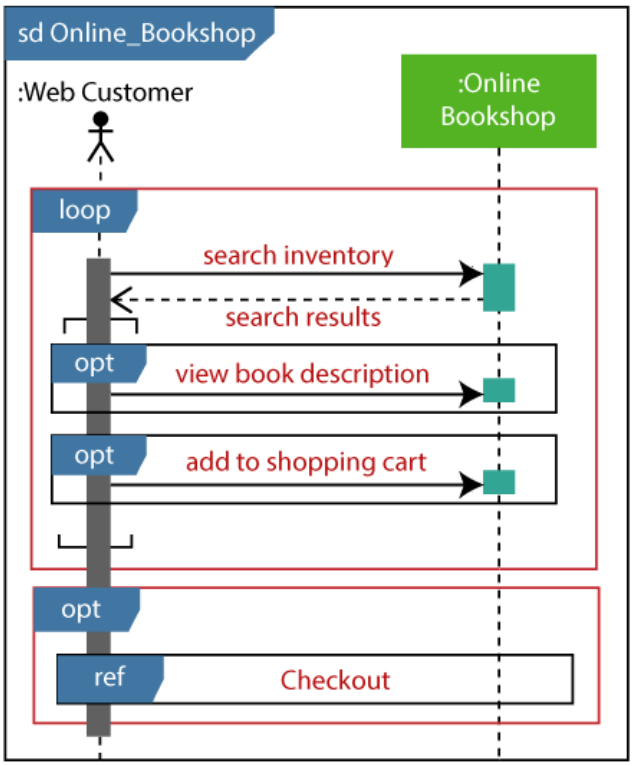
Operator	Fragment Type
alt	Alternative multiple fragments: The only fragment for which the condition is true, will execute.
opt	Optional: If the supplied condition is true, only then the fragments will execute. It is similar to alt with only one trace.
par	Parallel: Parallel executes fragments.
loop	Loop: Fragments are run multiple times, and the basis of interaction is shown by the guard.
region	Critical region: Only one thread can execute a fragment at once.

neg	Negative: A worthless communication is shown by the fragment.
ref	Reference: An interaction portrayed in another diagram. In this, a frame is drawn so as to cover the lifelines involved in the communication. The parameter and return value can be explained.
sd	Sequence Diagram: It is used to surround the whole sequence diagram.

3.6 Example of Sequence Diagram

An example of a high-level sequence diagram for online bookshops is given below.

Any online customer can search for a book catalog, view a description of a particular book, add a book to its shopping cart, and do checkout.



Section 4: OO Development Process,System Design and Frameworks

4.1 OO Development

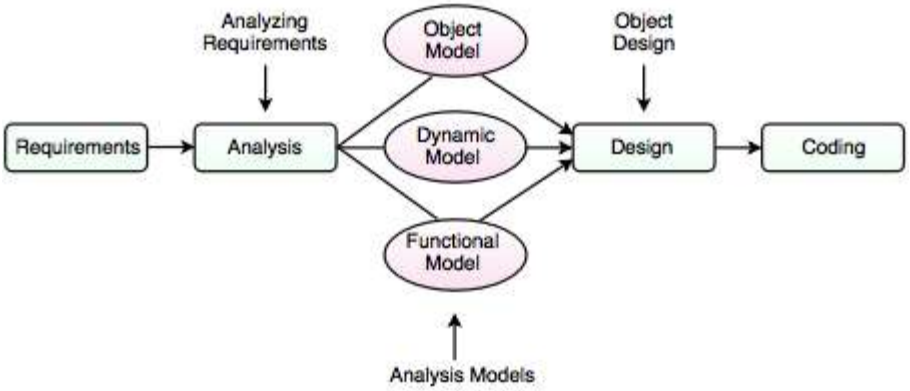
- The essence of OO development is the identification and organization of application concepts rather than their final representation in programming language.
- It encourages software developers to work and think in terms of application throughout the software life cycle.
- It is a conceptual process independent of the programming language until final stages.
- It is a way of thinking and not a programming technique.
- Its greatest benefit comes from helping developers, customers express abstract concepts clearly and communicate them to each other.
- The OO concepts are used to express a design and also provide documentation.

4.2 OO Methodology

The process for OO development and graphical notation for representing OO concepts consists of building a model of an application and then adding details to it during design.

OO Methodology has the following stages:

- **System conception** : Software development begins with business analysis or users conceiving an application and formulating tentative Requirements
- **System Analysis** : The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a precise abstraction of what the desired system must do, not how it will be done. It should not contain implementation decisions.
- **System design**: The development teams devise a high – level strategy called the system architecture for solving the application problem.
- **Class design** : The class designer adds details to the analysis model in accordance with the system design strategy. The focus of class design is the data structures and algorithms needed to implement each class.
- **Implementation** : Implementers translate the classes and relationships developed during class design into particular programming language, database or hardware.



The System analysis model has 2 parts:

- **Domain model -** a description of the real-world objects reflected within the system Eg: Domain objects for a stockbroker

- **Application model -** a description of the parts of the application system itself that are visible to the user. Eg:- Application might include stock, bond, trade and commission. Application objects might control the execution of trades and present the results.

4.3 System Design

- During System design, decide how the problem will be solved.
- Need to apply a high level strategy – *System Architecture* – for solving the problem and building a solution.
- During System design, developers decide the overall structure and style.
- System architecture provides the organization of the system into subsystems.

To construct system architecture, following decision must be made:

1. Estimating performance
2. Making a Reuse plan
3. Breaking system into subsystem
4. Identifying Concurrency
5. Allocation of Subsystems
6. Management of Data Storage
7. Handling Global Resources
8. Choosing a Software Control strategy
9. Handling Boundary Conditions
10. Common Architectural Style.

- **Estimating Performance**

    - You should prepare a rough performance estimate.
    - Purpose of this task to determine if the system is feasible.
    - You will have to make simplifying assumptions.(i.e. assume factors)
    - Don’t worry about detail – just approximate, estimate and guess.

- **Reuse Plan**
    - Two different aspect of reuse are possible:-
        - Using existing thing
        - Creating reusable new things
    - Much easier to reuse existing things than to design new things.
    - Most developers reuse existing things and only a small fraction of developers create new things.
    - Creating reusable new things is not an easy task. It require good experience

  Reusability includes mainly:

    - Libraries
    - Frameworks
    - Patterns

4.4 Libraries

It is a collection of classes that are useful in many contexts. Classes must be organized, so users can find them. Classes must have accurate and thorough descriptions to help users determine their relevance.
Good qualities of libraries are:

- **Coherence**: organized well focus
- **Completeness**: complete behavior
- **Consistency**: consistent names and signature.
- **Efficiency**: provide alternative to implement
- **Extensibility**: able to define subclasses
- **Genericity**: should be parameterized class where appropriate

Some of the most useful and popular libraries are as follows:

- Java Standard libraries
- Apache Commons
- Jackson

- Maven

4.5 Patterns

A pattern is a proven solution to a general problem.

- There are patterns for analysis, architecture, design and implementation.
- A pattern comes with guidelines on when to use it, as well as exchange on its use.
- Software Architecture Patterns are of two types
    - Layered Pattern
    - Client-Server Pattern
- **Design pattern**: The Gangs of Four(GOF) Design Patterns are broken into three categories:
    - **Creational Patterns** for the creation of objects.
    - **Structural Patterns** to provide relationships between objects.
    - **Behavioral Patterns** to help define how objects interact

4.6 Frameworks

- A Framework is a set of cooperating classes that makes up a reusable design for a specific class of software.
- Frameworks are larger architectural elements than design patterns.
- A typical framework contains several design patterns, but the reverse is never true
- Frameworks are the bodies that contain the pre-written codes (classes and functions) in which we can add our code to overcome the problem. We can also say that frameworks use programmer's code because the framework is in control of the programmer.

Some of the most popular Java frameworks are:

- Spring
- Hibernate
- Grails

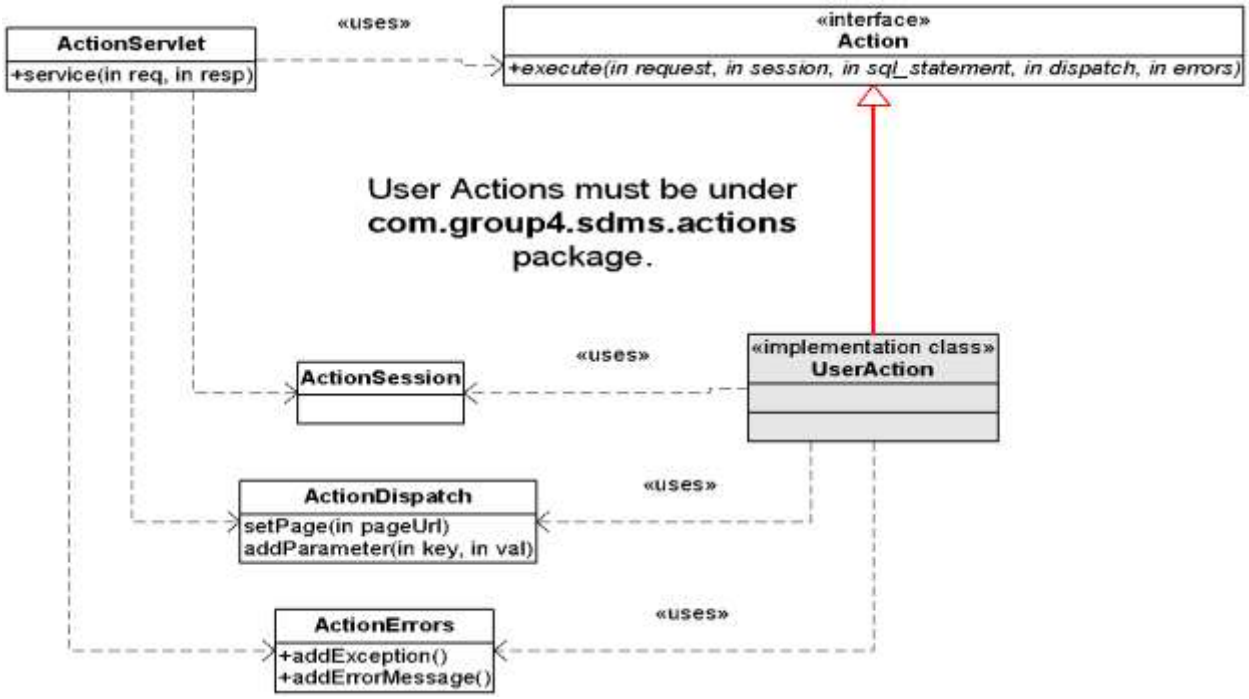
The advantages of the Java Frameworks are as follows:

- **Security:** It is the most important advantage of the Java framework. If we found any security loophole or vulnerability in an application, we can directly move to the official website of the framework to fix the security related issues.

- **Support:** The widely used framework provides large forums or groups where we can put our problem for the solution. It also provides the documentation of the framework that helps us to understand the working of the framework.
- **Efficiency:** If we are doing a task without using the framework, it may take time to complete. But if we are doing the same work by using the framework, we can complete that task in an easy and fast way. Therefore, using the Java framework development becomes faster, easier and effective. It also saves time and effort.
- **Expenses:** Another advantage of using the framework is to reduce the cost of the application. Because the maintenance cost of the framework is low.

In Java, **Collection** is an example of the framework. It reduces the programming efforts because it provides useful data structure and algorithms. It is referred to as a library that does not provide inversion of control.

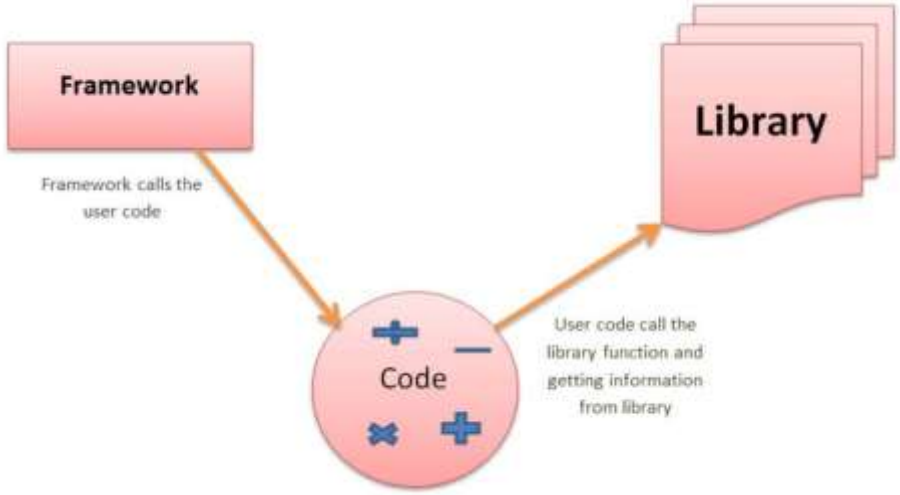
A framework relies on the 'Hollywood Principle' – 'don't call us, we'll call you."
This means that the user-defined classes (for example new subclasses) will receive messages from the predefined framework classes.
These are usually handled by implementing superclass abstract methods. (like via generalization)



In the above figure the framework provides all the classes you see except

the <<implementation>> class that fills the ‘User Action” role.This is the whole idea about Frameworks. The framework “calls” the UserAction code, which is something that application developers provide.

4.7 Libraries vs Frameworks



Library	Framework
Library is the collection of frequently used, pre-compiled classes.	Framework is the collection of libraries.
It is a set of reusable functions used by computer programs.	It is a piece of code that dictates the architecture of your project and aids in programs.
You are in full control when you call a method from a library and the control is then returned.	The code never calls into a framework, instead the framework calls you.

It is corporate seamlessly into existing projects to add functionality that you can access using an API.	It cannot be seamlessly incorporated into an existing project. Instead it can be used when a new project is started.
They are important in programs for linking and binding processes.	They provide a standard way to build and deploy applications.
Libraries do not employ an inverted flow of control between itself and its clients.	Framework employs an inverted flow of control between itself and its clients.
Example: jQuery is a JavaScript library that simplifies DOM manipulation.	Example: AngularJS is a JavaScript-based framework for dynamic web applications.

Section 5: Architectural Patterns

- Architecture is the blueprint of building software. It shows the overall structure of the software, the collection of components in it, and how they interact with one another while hiding the implementation.
- Some architecture patterns naturally lend themselves toward highly scalable applications, whereas other architecture patterns naturally lend themselves toward applications that are highly agile.
- Knowing the characteristics, strengths, and weaknesses of each architecture pattern is necessary in order to choose the one that meets the specific business needs and goals.
- Address various issues in software engineering, such as **computer hardware performance limitations, high availability and minimization of a business risk**
- Different Software Architecture Patterns :
    - Layered Pattern
    - Client-Server Pattern

- Event-Driven Pattern
- Microkernel Pattern
- Microservices Pattern
- MVC Patterns

5.1 Model View Controller

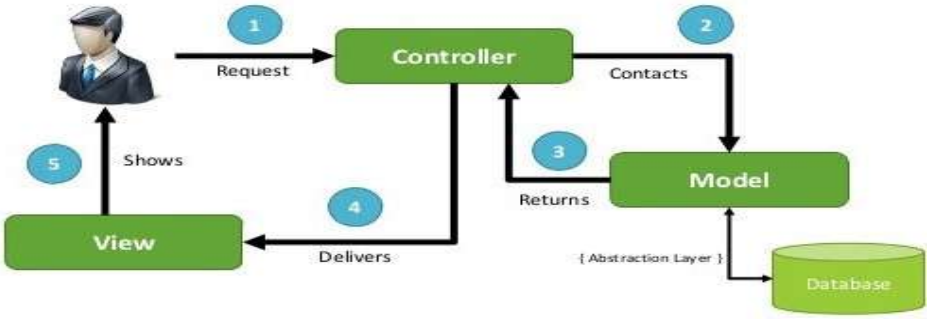
The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is a way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information.The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

- **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update the controller if data is changed.
- **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
- **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.

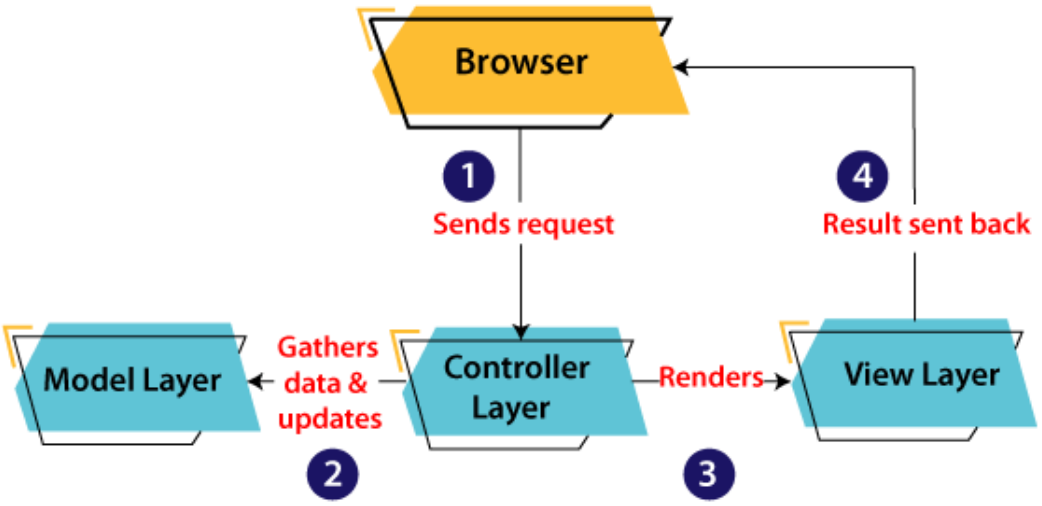
5.2 How does MVC work?

Whenever the controller receives a request from the user (either directly or via the view), it puts the model to work. And when the model delivers the data requested in the right format, the controller forwards it to the view.



5.3 MVC in Java

In Java Programming, the Model contains the simple Java classes, the View used to display the data and the Controller contains the servlets. Due to this separation the user requests are processed as follows:



1. A client (browser) sends a request to the controller on the server side, for a page.
2. The controller then calls the model. It gathers the requested data.
3. Then the controller transfers the data retrieved to the view layer.
4. Now the result is sent back to the browser (client) by the view.

5.4 Implementation of MVC using Java

- **Employee Class, will act as model layer:**

  It represents the business logic for application and also the state of application. The model object fetches and stores the model state in the database. Using the model layer, rules are applied to the data that represents the concepts of application.

Employee.java

```java
public class Employee {

      // declaring the variables
       private String EmployeeName;
       private String EmployeeId;
       private String EmployeeDepartment;

      // defining getter and setter methods
       public String getId() {
          return EmployeeId;
       }

       public void setId(String id) {
          this.EmployeeId = id;
       }

       public String getName() {
          return EmployeeName;
       }

       public void setName(String name) {
          this.EmployeeName = name;
       }

       public String getDepartment() {
              return EmployeeDepartment;
           }

       public void setDepartment(String Department) {
              this.EmployeeDepartment = Department;
           }

    }
```

- **EmployeeView Class, will act as a view layer:**

  View represents the visualization of data received from the model. The view layer consists of the output of the application or user interface. It sends the requested data to the client, that is fetched from the model layer by controller.

EmployeeView.java

```
// class which represents the view
public class EmployeeView {

      // method to display the Employee details
public void printEmployeeDetails (String EmployeeName, String
EmployeeId, String EmployeeDepartment){
          System.out.println("Employee Details: ");
          System.out.println("Name: " + EmployeeName);
          System.out.println("Employee ID: " + EmployeeId);
          System.out.println("Employee Department: " +
EmployeeDepartment);
       }
    }
```

- **EmployeeContoller Class, will act a controller layer:**

  The controller layer gets the user requests from the view layer and processes them, with the necessary validations. It acts as an interface between Model and View. The requests are then sent to the model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

EmployeeController.java

```
// class which represent the controller
public class EmployeeController {

      // declaring the variables model and view
       private Employee model;
       private EmployeeView view;

      // constructor to initialize
       public EmployeeController(Employee model, EmployeeView view) {
          this.model = model;
          this.view = view;
       }

      // getter and setter methods
       public void setEmployeeName(String name){
          model.setName(name);
       }

       public String getEmployeeName(){
          return model.getName();
       }

       public void setEmployeeId(String id){
          model.setId(id);
       }

       public String getEmployeeId(){
          return model.getId();
       }

       public void setEmployeeDepartment(String Department){
              model.setDepartment(Department);
       }

           public String getEmployeeDepartment(){
              return model.getDepartment();
       }

       // method to update view
       public void updateView() {
          view.printEmployeeDetails(model.getName(), model.getId(),
model.getDepartment());
       }
    }
```

- **Main Class Java file**

The following example displays the main file to implement the MVC architecture. Here, we are using the MVCMain class.

MVCMain.java

```
// main class
public class MVCMain {
	public static void main(String[] args) {

		// fetching the employee record based on the employee_id
from the database
		Employee model = retriveEmployeeFromDatabase();

		// creating a view to write Employee details on console
		EmployeeView view = new EmployeeView();

		EmployeeController controller = new
EmployeeController(model, view);

		controller.updateView();

		//updating the model data
		controller.setEmployeeName("Nirnay");
		System.out.println("\n Employee Details after updating: ");

		controller.updateView();
	}

	private static Employee retriveEmployeeFromDatabase(){
		Employee Employee = new Employee();
		Employee.setName("Anu");
		Employee.setId("11");
		Employee.setDepartment("Salesforce");
		return Employee;
	}
    }
```

Section 6: GRASP Principles

It stands for **General Responsibility Assignment Software Patterns**.It guides in assigning responsibilities to collaborating objects.
There are 9 GRASP patterns

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Indirection
7. Polymorphism
8. Protected Variations
9. Pure Fabrication

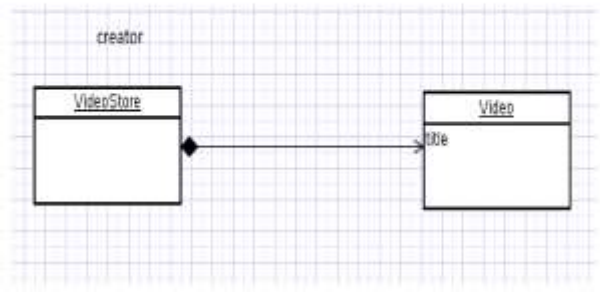
6.1 Creator

- Who creates an Object? Or who should create a new instance of some class? “Container” objects create “contained” objects.
- Decide who can be the creator based on the objects association and their interaction.

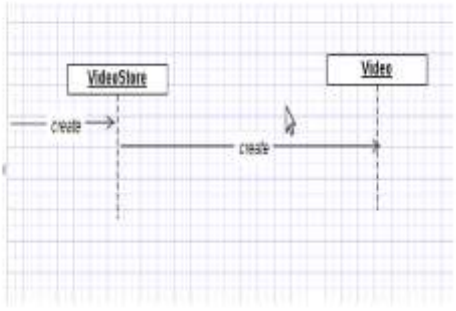
Example for Creator

- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in VideoStore class

Example diagram



Example for creator

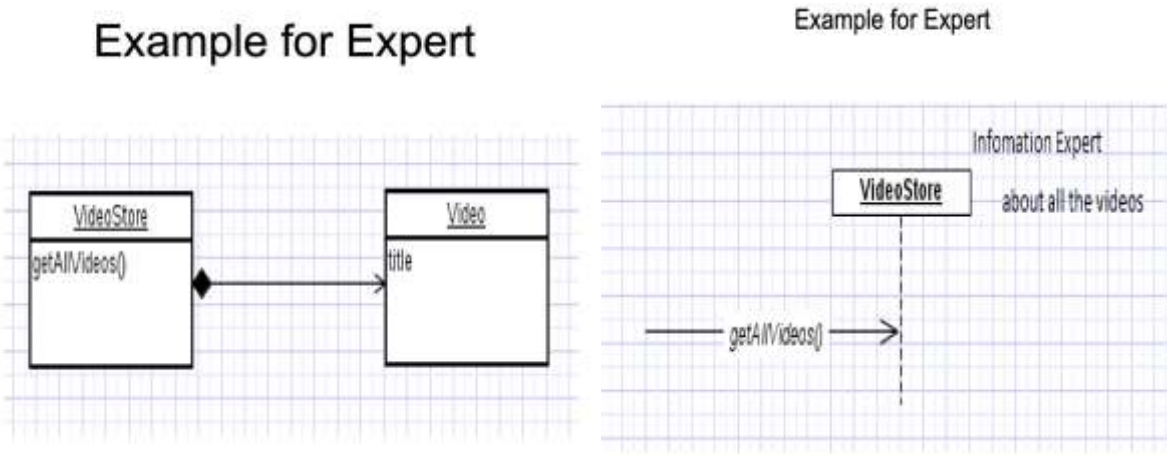


6.2 Information Expert

- Given an object o, which responsibilities can be assigned to o? Expert principle says – assign those responsibilities to o for which o has the information to fulfill that responsibility.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

Example for Expert

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to the VideoStore class.
- VideoStore is the information expert.

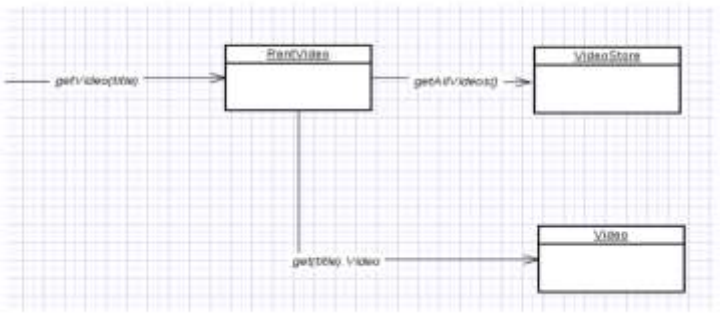


6.3 Low Coupling

- How strongly the objects are connected to each other? Coupling – objects depending on other objects.
- When dependent upon element changes, it affects the dependent also.
- Low Coupling – How can we reduce the impact of change in dependent elements on dependent elements.
- Prefer low coupling – assign responsibilities so that coupling remains low.
- Minimizes the dependency hence making the system maintainable, efficient and code reusable.

- Two elements are coupled, if
    - One element has aggregation/composition association with another element.
    - One element implements/extends another element.

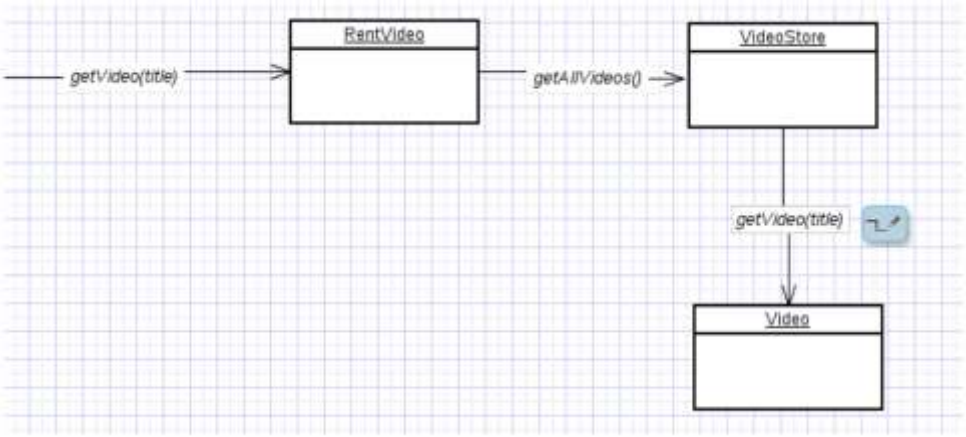
Example for poor coupling



Here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.

Example for low coupling

VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.

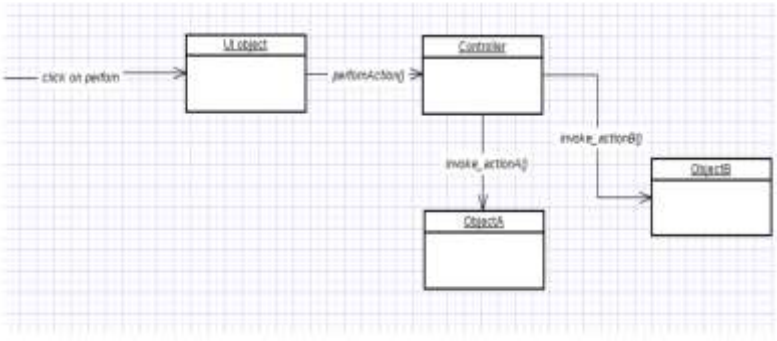


6.4 Controller

- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- When a request comes from a UI layer object, the Controller pattern helps us in determining what is that first object that receives the message from the UI layer objects.

- This object is called controller object which receives requests from UI layer objects and then controls/coordinates with other objects of the domain layer to fulfill the request.
- It delegates the work to other classes and coordinates the overall activity.
- We can make an object as Controller, if
    - Object represents the overall system (facade controller)
    - Objects represent a use case, handling a sequence of operations (session controller).
- **Benefits**
    - Can reuse this controller class.
    - Can be used to maintain the state of the use case.
    - Can control the sequence of the activities

Example for Controller



Bloated Controllers

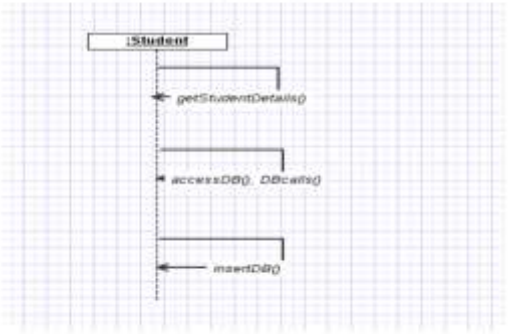
- Controller class is called bloated, if the class is overloaded with too many responsibilities.
    - Solution: Add more controllers
- The Controller class also performs many tasks instead of delegating to other classes.
    - Solution: The Controller class has to delegate things to others

6.5 High Cohesion

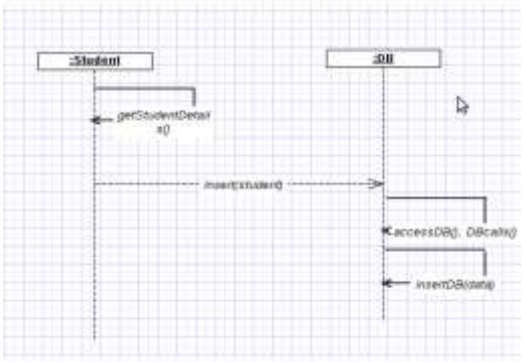
- How are the operations of any element functionally related? High cohesion helps in keeping related responsibilities into one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
    - Easily understandable and maintainable.

- Code reuse.
- Low coupling.

Example for low cohesion



Example for High Cohesion

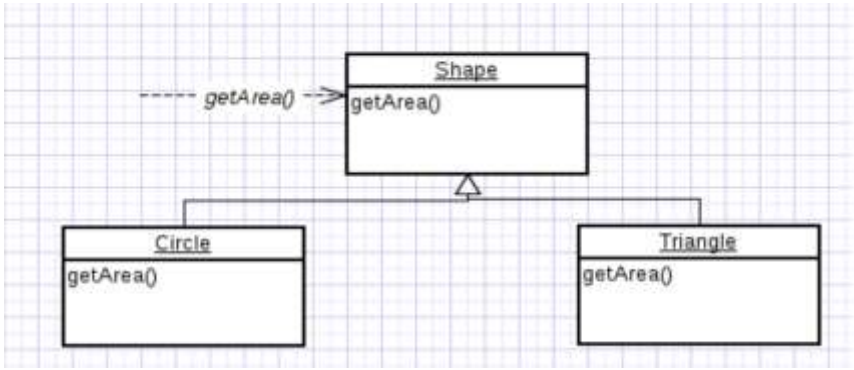


6.6 Polymorphism

- How to handle related but varying elements based on element type? Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- **Benefits:**
  - Handling new variations will become easy.

Example for Polymorphism

The getArea() varies by the type of shape, so we assign that responsibility to the subclasses.



By sending a message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

6.7 Pure Fabrication

- Fabricated class/ artificial class:assign a set of related responsibilities that

doesn't represent any domain object.

- Provides a highly cohesive set of activities.
- Behavioral decomposition – implements some algorithms.
- Examples: Adapter, Strategy
- Benefits: High cohesion, low coupling and can reuse this class.

Example

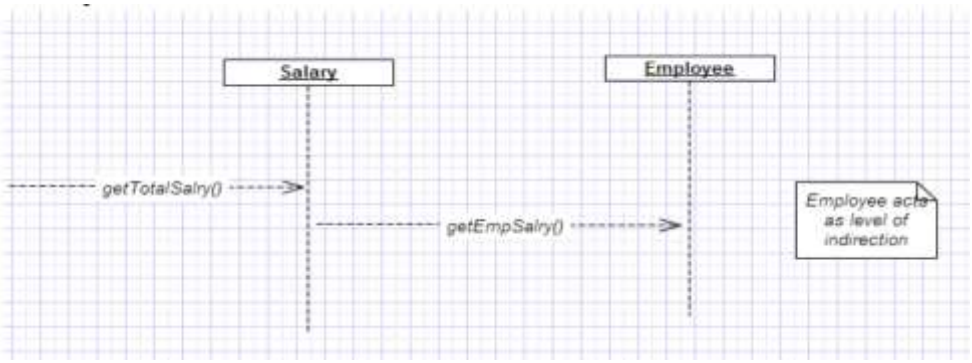
- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible for performing all database operations.
- Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

6.8 Indirection

- How can we avoid a direct coupling between two or more elements?Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Facade, Observer

Example for Indirection

- Here polymorphism illustrates indirection
- Class Employee provides a level of indirection to other units of the system.



6.9 Protected Variation

- How to avoid the impact of variations of some elements on the other elements?Protected variation provides a well defined interface so that there will be no effect on other units.
- It provides flexibility and protection from variations.
- It provides more structured design.
- Example: polymorphism, data encapsulation, interfaces