Unit 2 Software Requirement Analysis and Design

❖ Requirement Gathering & Analysis

- Requirement of an End User plays a key role in developing software product.
- The task of requirement gathering and analysis is done by the system analyst.
- Collecting all the information from the end user and then analyze the collected information to remove all ambiguity and inconsistencies from customer perception.
- Mainly two activities are performed during this task.
  - Requirement gathering
  - Requirement analysis

Requirement gathering

- Requirement gathering is the first activity performed during the software development.
- The goal of requirement gathering activity is to collect all relevant information from the customer regarding the product to be developed.
- In this phase, meeting with customers, analyzing market demand and features of the product are mainly focused.
- Requirement gathering activities are
  - Studying the existing documents
  - Interview with end users
  - Task analysis
  - Scenario analysis
  - Brainstorming
  - Questionnaires

Requirement Analysis

- The goal of requirement analysis activity is to clearly understand the exact requirements of the customer.
- Requirement analysis helps to understand, interpret, classify and organize the software requirements in order to access the feasibility, completeness and consistency of the requirements.
- This activity involves

- Eliciting requirements: requirements are eliciting by communicating with customers and find their exact need.
- Analyzing requirements: requirements are then analyzed to make it complete, clear and unambiguous.
- Requirement recording: all the requirements are recorded in form of use-case, process specifications etc.

- The output of requirement gathering and analysis activity is SRS document.
- SRS means System/Software requirement specification.

❖ Software Requirement Specification (SRS)

- SRS is the output of requirement gathering and analysis activity.
- SRS is a detailed description of the software that is to be developed.
- It describes the complete behavior of the system.
- SRS describe *what* the proposed system should do without describing *how* the software will do.
- It is working as a reference document for the developer.
- The SRS translate the ideas of the customer into the formal documents.
- The SRS document is prepared by system analyst.

SRS – Benefits

- SRS provides foundation for design work. Because it works as a input to the design phase.
- It enhances communication between customer and developer.
- Developers can get the idea what exactly the customer wants.
- It reduces the development cost and time.
- It enables project planning and helps in verification and validation process.

SRS – Content

- **Customer Requirements:**
  - Customer requirement are the important aspect of the SRS document.

  - It represents the needs, expectations, and desires of the end-user, customers and other stack holders who will be using the software product.
- **Functional requirement:**
  - The functional requirement is the services or functionalities that the system is expected to provide.
- **Non-Functional requirement:**
  - The non functional requirements describe the characteristics of the system that can't be expresses functionality.
  - Ex: portability, maintainability, usability, security etc.
- **Constraint:**
  - That describes what the system should do or should not do.

Characteristics of SRS

- **Concise**
  - SRS should contain brief and concise information regarding the project; no more detailed description of the system should be there.
- **Complete**
  - It should be complete regarding to the project. So, it can be easily understood by the analyst and developer.
- **Consistent**
  - An SRS should be consistent through the project development.
  - Requirement may not conflict at later stage..
- **Structured**
  - SRS should be well structured to understand and to implement.
- **Black-Box View**
  - SRS should have block box view means; there should not be much detailing of the project in it.
- **Verifiable**
  - It should be verifiable by the clients or the customers for whom the project is being made.
- **Adaptable**

- It should be adaptable in both sides from the clients as well as from the developers.

- **Maintainable**
  - SRS should be maintainable so in the future change can be made easily.
- **Portable**
  - It should be portable as we can use the contents of it for the same types of development.
- **Unambiguous**
  - There should not be any alternates of SRS that creates ambiguity.
- **Traceable**
  - Each of the requirements should be clear and refer to the future development.

i. Customer Requirements:

- The SRS document should accurately capture all customer requirements, which are typically gathered through various methods,such as interviews,surveys,focus groups,and market research.
- The requirements should be clear,unambiguous,specific,measurable,and traceable to ensure thar the software meets the customer’s expectations.

How to identify customer’s requirements:

The are several ways by which we can help to identify the requirements of customers.

- Conduct interviews
- Analyse existing system
- conduct workshops
- Have customer feedback
- Analyze competitors

- Once you have identified customer requirements, it is important to document them in a clear and concise manner in the SRS document.
- This document will serve as a blueprint for the development team to ensure that the software product meets the needs and expectations of the customers.

Steps to write customer requirements in SRS

- Identify the customers – determine for whom the software is being developed.

- Gather requirements – conduct interviews, surveys and feedback sessions to get exact customer requirements.
- Organize the requirements - Organize the requirements based on their importance and properties.
- Use consistent format - Use consistent format (i.e. Requirement ID,rational,description,functional and Non-functional requirements).

ii. Functional Requirement of SRS

- The functional requirements define the functions of the software and its components.
- It forms the core of requirement documents.
- The functional requirements for a system describe the functionalities or services that the system is expected to provide.
- They provide how the system should react to particular input and how the system should behave in a particular condition.
- The Goal of functional requirement is to capture the behavior of the software in terms of functions and technology.
- **How to identify functional requirement?**
    a. From informal problem description or from conceptual understanding of the system.
    b. Identify from user perspective.
- **How to document the functional requirements?**
    a. Document the functionalities supported by the system.
    b. Specify the input data domain, processing and output data domain.

iii. Non Functional Requirement of SRS

- Non functional requirements are requirements that specify criteria that can be used to judge the operation of a system in particular conditions, rather than specific behavior.
- Sometimes these requirements are known as quality attributes.
- Non functional requirements describes the characteristics that cant be expressed functionally,lke…
- Portability
- Reliability

- Security
- Scalability
- Usability
- Performance
- Inter operability
- Availability

❖ Software Design Process

- The design process is a sequence of steps to describe all aspects of the system.
- Design process specifies how the system will be implemented and how it will work.
- The goal of design process is to plan a solution of the problem specified in SRS.
- It includes user interface design, input output design, data design, process and program design etc.
- The output of design process is design documents.

1. Characteristics of Good Software Design

i. **Functionality:** The software meets the requirements and specifications that it was designed for, and it behaves as expected when it is used in its intended environment.

ii. **Usability:** The software is easy to use and understand, and it provides a positive user experience.

iii. **Reliability:** The software is free of defects and it performs consistently and accurately under different conditions and scenarios.

iv. **Performance:** The software runs efficiently and quickly, and it can handle large amounts of data or traffic.

v. **Security:** The software is protected against unauthorized access and it keeps the data and functions safe from malicious attacks.

vi. **Maintainability:** The software is easy to change and update, and it is well-documented, so that it can be understood and modified by other developers.

vii. **Reusability:** The software can be reused in other projects or applications, and it is designed in a way that promotes code reuse.

viii. **Scalability:** The software can handle an increasing workload and it can be easily extended to meet the changing requirements.

ix. **Testability:** The software is designed in a way that makes it easy to test and validate, and it has comprehensive test coverage.

2. Analysis Vs Design

Sr. No.	**System Analysis**	**System design**
1.	It is done before Design.	It is done after Design.
2.	It is like examining the problem.	It is like finding the solution.
3.	It is considering "what" part of software development.	It is considering "how" part of software development.
4.	It deals with data collection.	It deals with design specification.
5.	Its output is SRS document.	Its output is design documents with technical specification.
6.	It includes requirement gathering, use case analysis, requirement specification,etc.	It includes architecture design, user interface design, component design,etc.

❖ Cohesion and Coupling

- Modularity is clearly a desirable property of software development
- A system is considered modular if it consist of discrete modules show that each module can be implemented separately and debugged separately.
- Cohesion and coupling are two modularization criteria that are often used together.

Classification of Cohesion

- Cohesion is a measure of function strength of a module.
- Cohesion keeps the internal module together and represent the function strength.
- Cohesion of a module represents how tightly bound the internal element of a module are to one another.

- **Classification of cohesion (Types)**

i. Coincidental(Low)
ii. Logical
iii. Temporal
iv. Procedural
v. Communicational
vi. Sequential
vii. Functional(High)

I. Coincidental cohesion

- It is lowest cohesion.
- It occurs when there is no meaningful relationship between the elements.
- A module is said to have coincidental cohesion if it perform a set of task that relate each other very loosely.

II. Logical Cohesion

- A module is said to be logically cohesive if there is some logical relationship between the element of module and element perform functions that fall into same logical class.
- Example: the task of error handling, input and output of data.

III. Temporal Cohesion

- It is same as logical cohesion except that the elements are related in time and are executed together.
- A module is in temporal cohesion when a module contains functions that must be executed in the same time span.
- Example: modules that perform activities like initialization, cleanup, startup and shut down.

IV. Procedural cohesion

- A module has procedural cohesion when it contains elements that belongs to common procedural unit.
- A module is said to have procedural cohesion, if the set of the module and all the part of a procedure in which certain sequence of steps are carried out achieve an objective.

V. Communicational Cohesion

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

VI. Sequential Cohesion.

- When the output of one element in a module forms the input to another module, we get sequential cohesion.
- Sequential cohesion does not provide any guideline how to combine these elements into modules.

VII. Functional Cohesion

- Functional cohesion is the strongest cohesion.
- In it, all the elements of the module are related to perform a single task.
- All elements are achieving a single goal of a module.

Classification of Coupling

- Coupling between two modules is a measure of the degree of interaction between two modules.
- Coupling refers to the no of connections between 'calling' and a 'called' module.

- There must be at least one connection between them.
- It refers to the strengths of relationship between modules in a system. It indicates how closely two modules interact and how they are interdependent.
- As modules become more interdependent, the coupling increases. And loose coupling minimize interdependency that is better for any system development.
- High coupling between modules make the system difficult to understand and increase the development efforts. So low coupling is the best.

- **Classification of coupling**
  1. Data (Low) (Best)
  2. Stamp
  3. Control
  4. Common
  5. Content (High) (Worst)

I. Data Coupling

- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two.
- Ex: an int, a char etc
- It is a lowest coupling and best for software development.

II. Stamp Coupling

- Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

III. **Control Coupling**

- Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.
- Ex: - flag set in one module and tested in another module.

IV. **Common Coupling**

- Two modules are common coupled, if they share data through some global data items.

V. **Content Coupling**

- Content coupling exists between two modules, if they share code.
- Ex:- a branch from one module into another module.
- It is a highest coupling and creates more problems in S/W development.

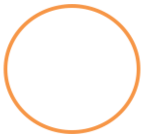
❖ Function Oriented Software Design: Data Flow Diagrams

- DFD is also called as bubble chart or data flow graph.
- DFDs are very useful in understanding the system and can be effectively used during analysis.
- It shows the flow of data through a system visually.
- The DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.
- It views a system as a function that transforms the inputs into desired outputs.
- Each function is considered as a process that consumes some input data and produces some output data.

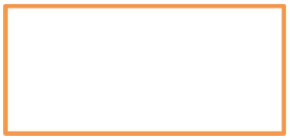
Primitive Symbols of DFD

- **Process (Function)**

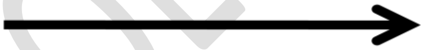
  i. Process is represented by circle or bubble.
  ii. Circles are annotated with names of the corresponding functions.
  iii. A process shows the part of the system that transforms inputs into outputs.
  iv. The process is named using a single word that describe what the system does functionality.



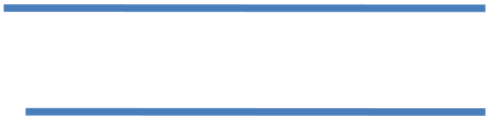
- **External Entity**
  1. Entity is represented by rectangle.
  2. Entities are external to the system which interacts by inputting data to the system or by consuming data produced by the system.
  3. It can also define source or destination of the system.



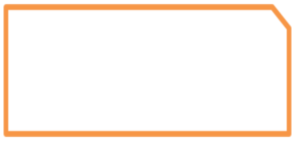
- **Data Flow**
  1. Data flow is represented by an arc or by an arrow.
  2. It is used to describe the movement of the data.
  3. It represents the data flow occurring between two processes or between an external entity and a process.
  4. It passes data from one part of the system to another part.
  5. Data flow arrows usually annotated with the corresponding data names.



- **Data Store**
  1. Data store is represented by two parallel lines.
  2. Generally it is a logical file or database.
  3. It can be either a data structure or a physical file on the disk.



- **Output**
  1. Output is used when a hardcopy is produced.
  2. It is graphically represented by a rectangle cut either a side.

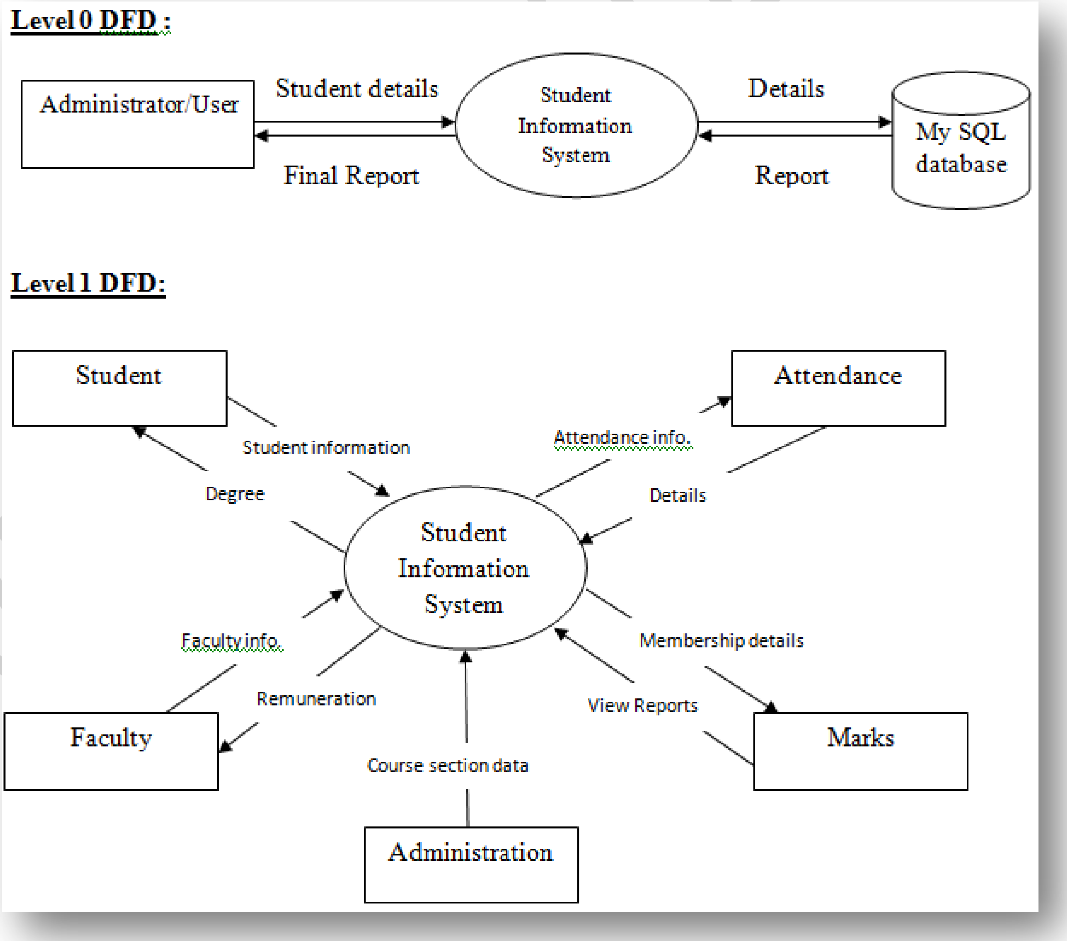


Developing DFD model of the system

- DFD graphically shows the transformation of the data input to the system to the final result through a hierarchy of levels.
- DFD starts with the most abstract level of the system and at each higher level, more details are introduced.
- **Context diagram:**
  1. To develop higher level DFDs, processes are decomposed into their sub functions. The abstract representation of the problem is called as context diagram.
  2. The context diagram is top level diagram.
  3. It only contains one process node that generalizes the function of entire system with external entities.
  4. It represents the entire system as a single bubble.
  5. Data input and output are represented using incoming and outgoing arrows.
  6. These arrows should be annotated with the corresponding data items.
  7. The bubbles (Circles) are decomposed into sub-functions at the successive levels of the DFD.
- **Numbering of bubbles**
  1. It is necessary to number the different bubbles occurring in the DFD.
  2. The bubble at the context level is usually assigned the number 0 to indicate that it is 0 levels DFD.
  3. Bubbles at level 1 are numbered as 0.1, 0.2 etc

- **Example (Draw DFD for Student Information System)**
  - Starting from the registration process, a student may come to the college for the admission for a particular course. He/She submits the registration form and student registration form processing is handled by the College Administration Information Process.

- **College administration:** College administration stores the student information in student information record. It too collects information from the account database to know about the student's payments records and from the faculty process about the faculty information. It sends the course information to the course information process for further processing.
- **Student Details:** In Student Details the whole personal details of any student is stored. Which can be edited only by authorized administrator? And can be used to view through anyone for any particular details.
- **Faculties:** Faculty information which will be useful for the instructors and separating the sections of the students.



❖ Object Modeling with UML

- UML is **not a programming language**; it is rather a visual language. We use UML diagrams to portray the **behavior and structure** of a system.

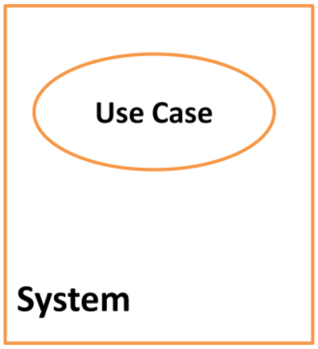
- UML can be used to construct nine types of diagrams to capture five different views of a system.

- Diagrams in UML can be broadly classified as:

  1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
  2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

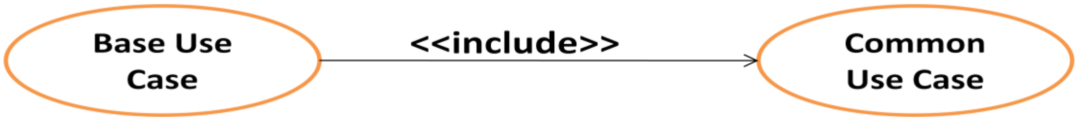
❖ Use-Case Diagram

- Use case model in UML provides system behavior.
- Use cases represent the different ways in which a system can be used by the users.
- The purpose of use case is to define the logical behavior of the system without knowing the internal structure of it.
- UML describes "***who can do what in a system***"
- A use case represents a sequence of interactions between the user and the system.
- **Components of Use Case Diagram:**
  - **Use Case**
    - i. Each use case is represented by ellipse with the name of the use case written inside the ellipse.
    - ii. All the use cases are enclosed with a rectangle representing system boundary.
    - iii. Rectangle contains the name of the system.

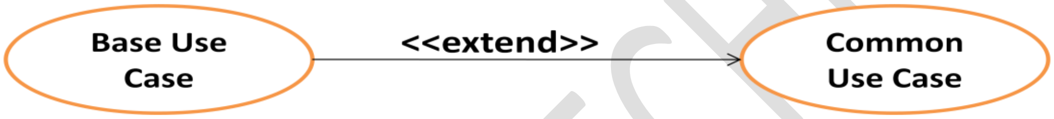


- **Actor**
  1. An actor is anything outside the system that interacts with it.
  2. Actors in the use case diagram are represented by using the stick person icon.
  3. An actor may be a person, machine or any external system.
  4. In use case diagrams, actors are connected to use cases by drawing a simple line connected to it.
  5. Actor triggers use case.
  6. Each actor must be linked to a use case, while some use cases may not be linked to actors.
- **Relationship**
  1. Actors are connected to the use cases through relationship.
  2. An actor may have relationship with more than one use case and one use case may relate to more than one actor.
  3. **Types of relationship**
     1. Association
     2. Include Relationship
     3. Extend Relationship.
- **Association**
  1. This relationship is the interface between an actor and a use case.
  2. It is represented by joining a line from actor to use case.
- **Include Relationship**
  1. It involves one use case including the behavior of another use case.
  2. The “include” relationship occurs when a chunk of behavior that is similar across a number of use cases.

iii. It is represented using predefined stereotype <<include>>.

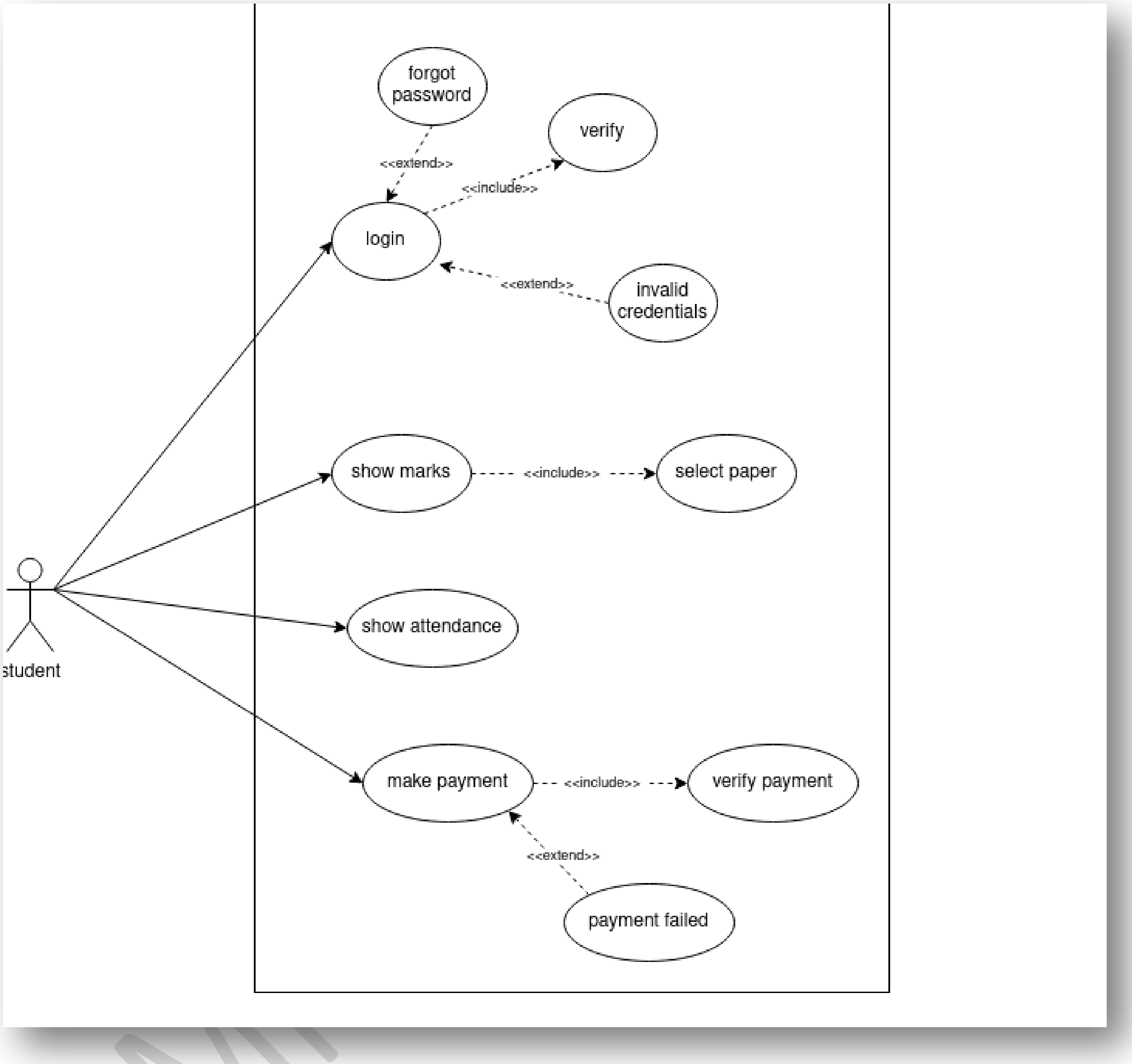


- **Extend Relationship**
  i. It allows you to show optional behavior of the system.
  ii. This relationship among use cases is represented as a stereotype <<extend>>.
  iii. Extend relationship exists when one use case calls another use case under certain condition.



VI. Use Case Guidelines

- Identify all different users and give suitable names.
- For each user, identify tasks. These tasks will be the use cases.
- Use case name should be user perspective.
- Show relationship and dependencies.
- **Example (Use Case for Student Information System)**



❖ Class Diagram

- Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration.

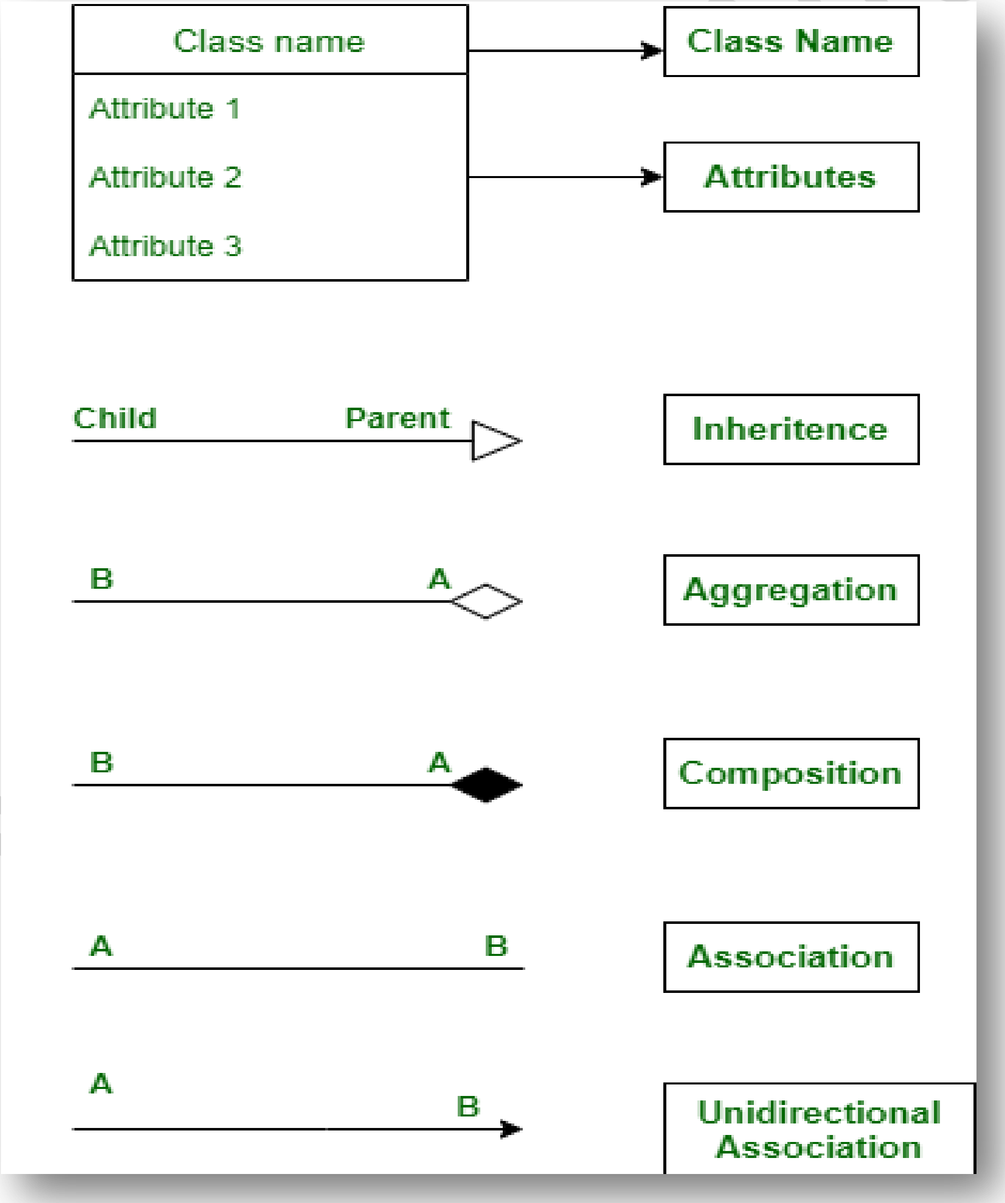
- Class diagrams basically represent the object-oriented view of a system, which is static in nature.

- Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.

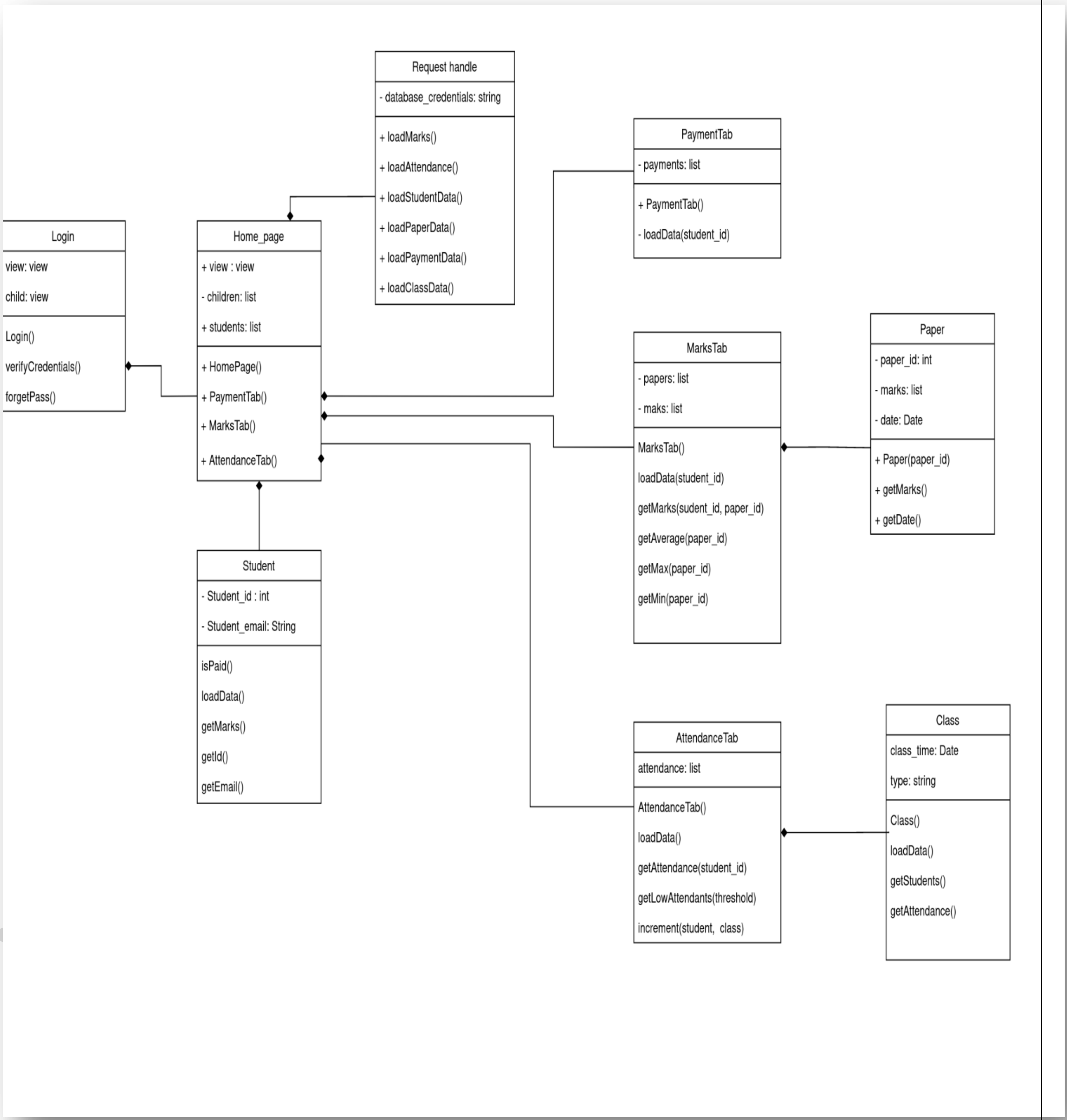
The following points should be remembered while drawing a class diagram −

- The name of the class diagram should be meaningful.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified.
- Use notes whenever required to describe some aspect of the diagram.

Notations of the class diagram:



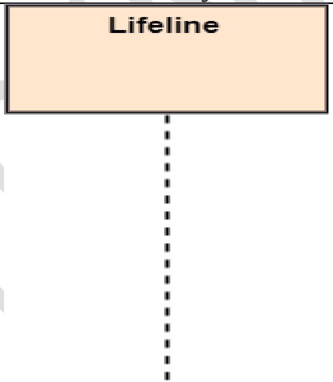
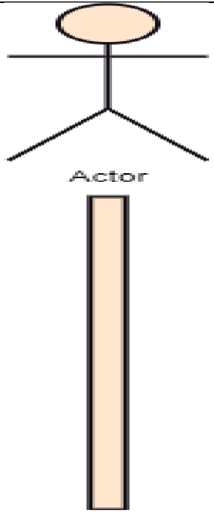
Example of Class Diagram of student management system

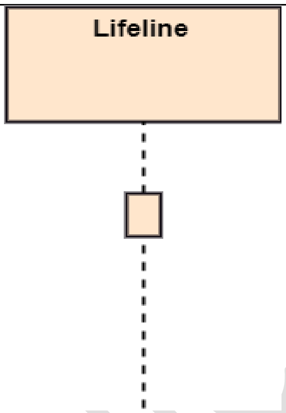
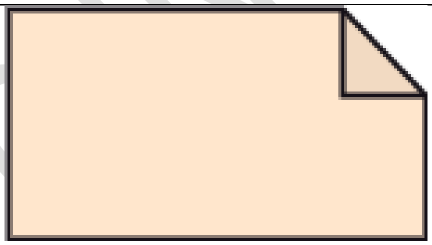
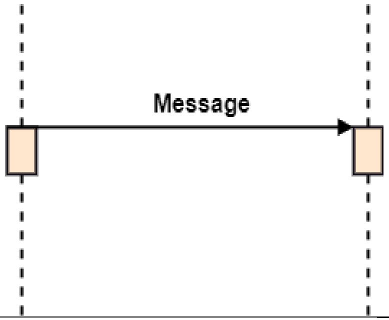
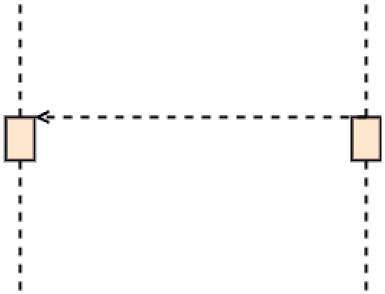


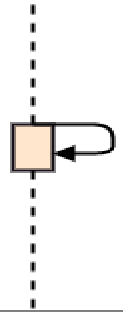
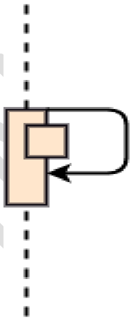
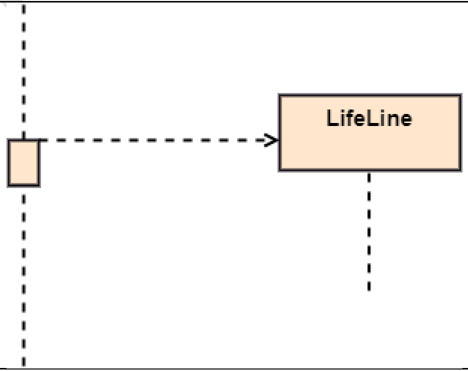
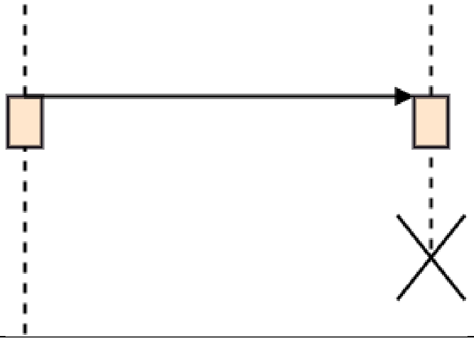
❖ Sequence /Event Diagram

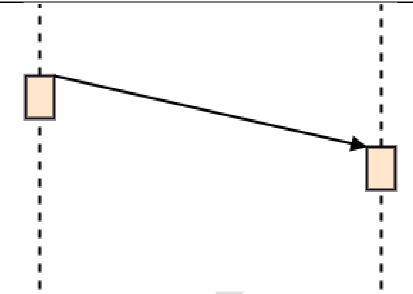
- The sequence diagram represents the flow of messages in the system and is also termed as an event diagram.
- It represents the communication between any two lifelines as a sequence of events.
- In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page.

- **Notations of a Sequence Diagram**

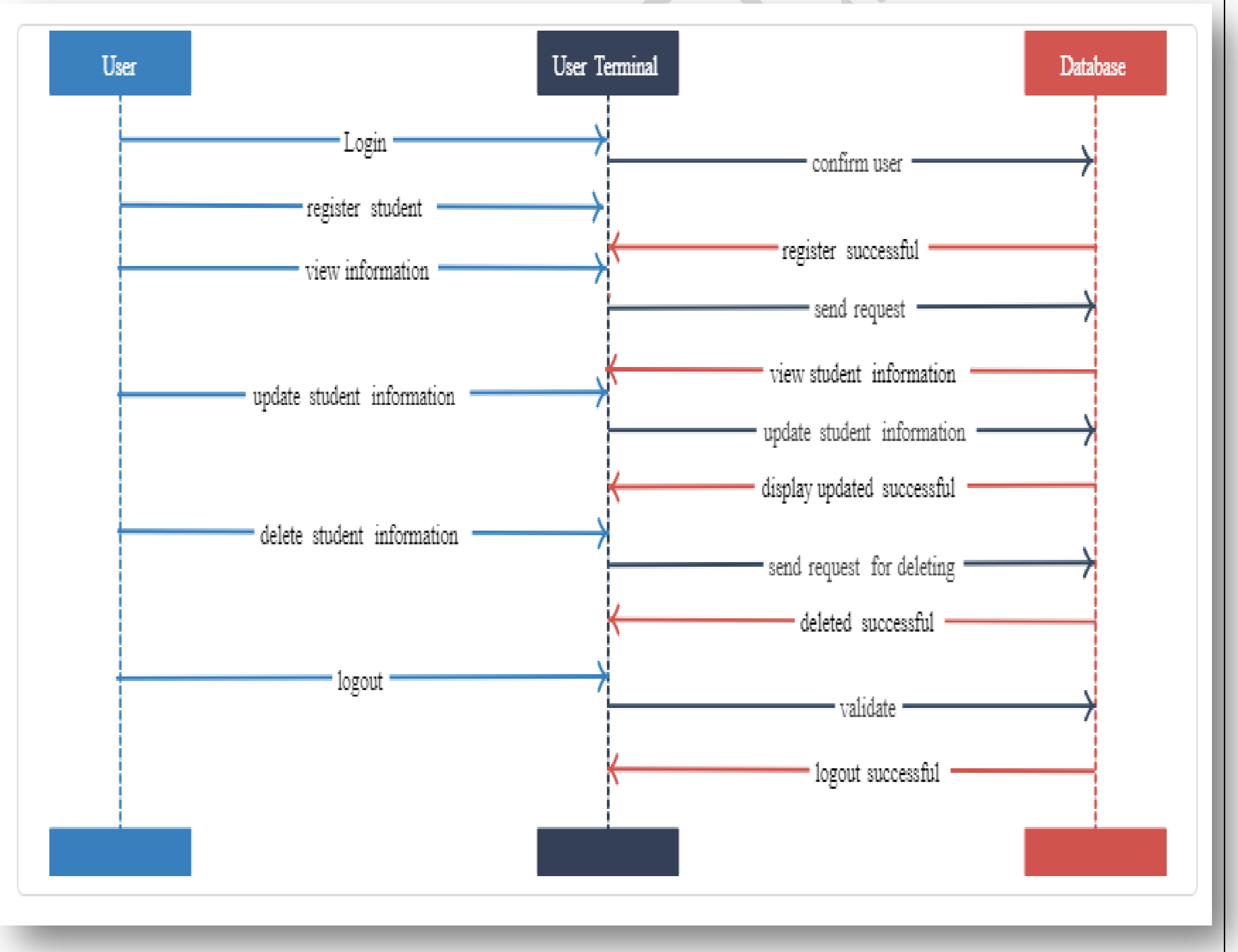
Sr No.	Symbol Name	Description	Symbol
1	Lifeline	An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram	
2	Actor	A role played by an entity that interacts with the subject is called as an actor. It represents the role, which involves 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.	

3	**Activation**	It is represented by a thin rectangle on the lifeline. It describes that 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, each respectively.	
4	**Note**	A note is the capability of attaching several remarks to the element. It basically carries useful information for the modelers.	
5	**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:			
5.1	**Call Message**	It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.	
5.2	**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.	

5.3	**Self Message**	It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.	
5.4	**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.	
5.5	**Create Message**	It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.	
5.6	**Destroy Message**	It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.	

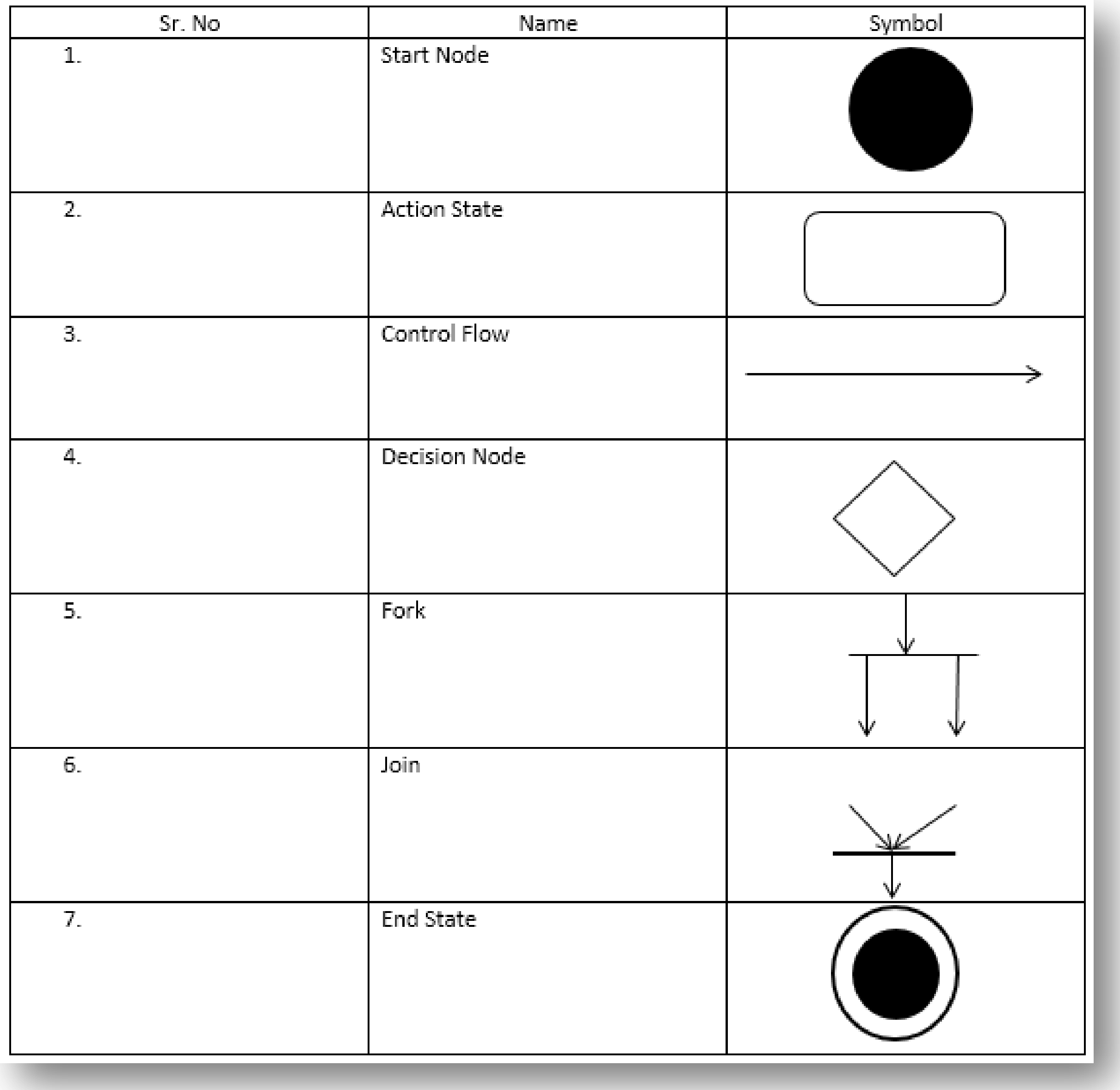
5.7	**Duration Message**	It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.	

Example of Sequence Diagram of student management system



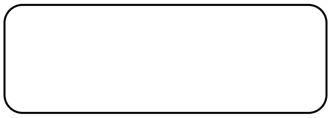
❖ Activity Diagram

- Activity diagram is a UML diagram that is used to model a process.
- Activity diagrams consist of activities, states and transitions between activities and states.
- It describes how the events in a single use case relate to one another.
- Activity diagrams represent workflows in a graphical ways.
- The AIM of Activity diagram is to record the flow of control from one activity to another of each actor and show interaction between them.
- Activity diagrams are similar to procedural flow charts. The difference is that activity diagram support parallel activities.
- **Elements of Activity Diagram**

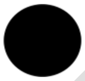


Sr. No	Name	Symbol
1.	Start Node	●
2.	Action State	▢
3.	Control Flow	→
4.	Decision Node	◇
5.	Fork	
6.	Join	
7.	End State	◉

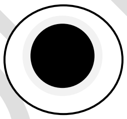
- **Activity**
  i. It represents a particular action taken in the flow of control.
  ii. It is denoted by a rectangle with rounded edges and labeled inside it describing corresponding activity.



  iii. Initial Activity
    1. It shows the starting point or first activity of the flow. It is denoted by solid circle.

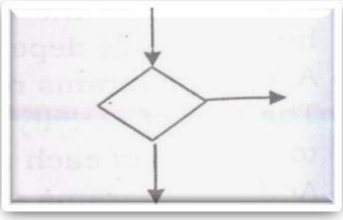


  iv. Final Activity
    1. It shows the end point of all activities. It is represented by bull's eye symbol.

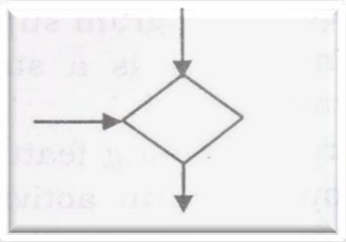


- **Flow**
  i. A FLOW is represented with a directed arrow.
  ii. This is used to show transfer of control from one activity to another.

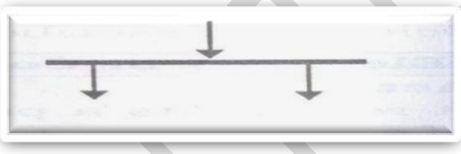
- **Decision**
  i. A decision node represented by diamond.



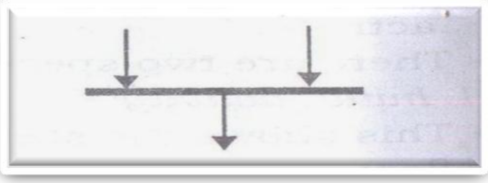
- **Merge**
  1. This is represented with a diamond shape with two or more input transitions and a single output.



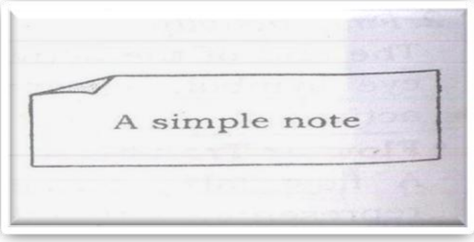
- **Fork**
  1. Fork is a point where parallel activities begin.
  2. Fork is denoted by black bar with one incoming transition and several outgoing transitions.



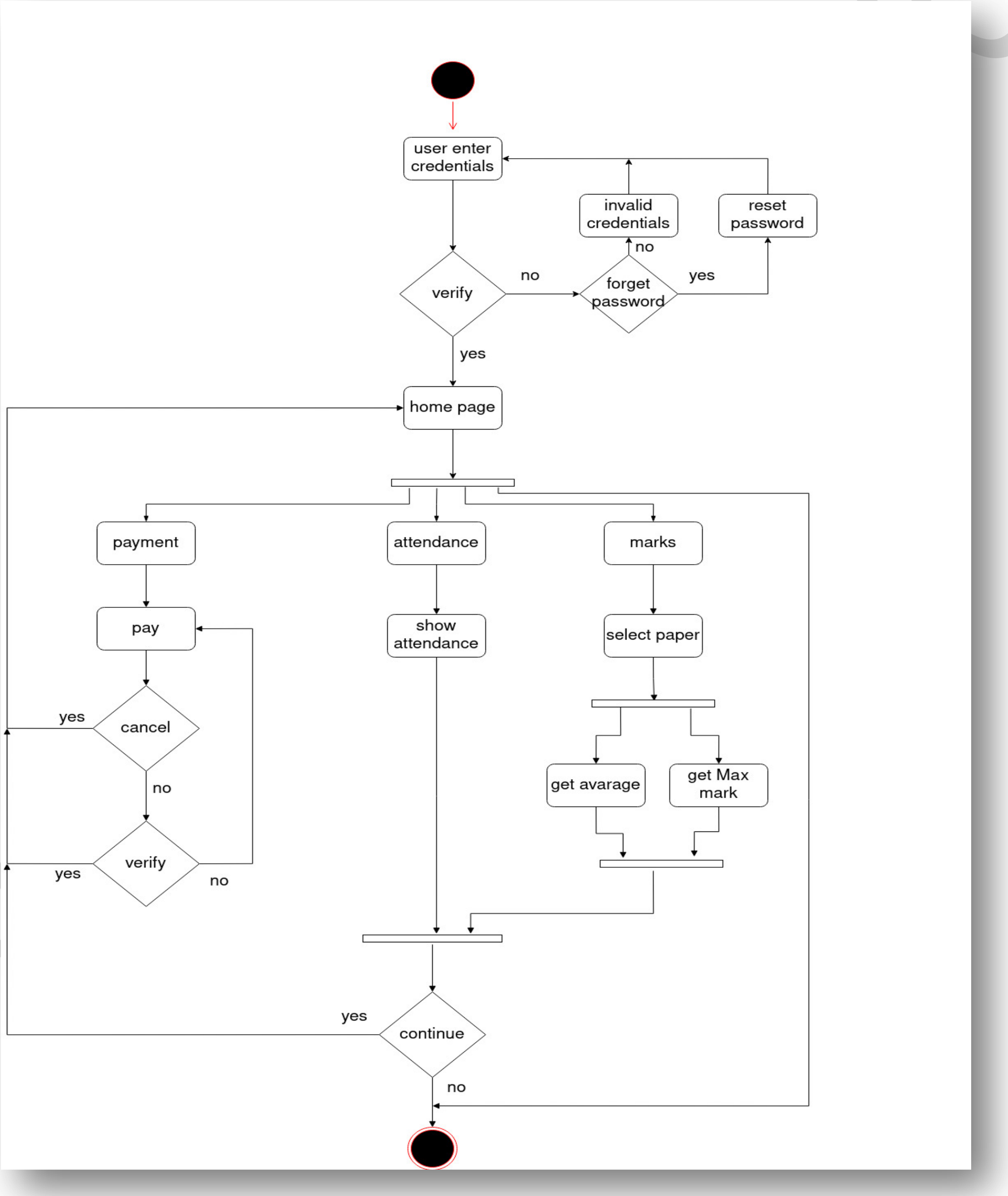
- **Join**
  1. Join is denoted by a black bar with multiple incoming transitions and single outgoing transition.
  2. It represents the synchronization of all concurrent activities.



- **Note**
  1. UML allows attaching a note to different components of diagram to present some textual information.
  2. It could be some comments or may be some constraints.
  3. It is denoted by a rectangle with cut a side.



Example of Activity Diagram of student management system


user enter credentials
invalid credentials
reset password
no
verify
no
forget password
yes
yes
home page
payment
attendance
marks
pay
show attendance
select paper
yes
cancel
no
get avarage
get Max mark
verify
yes
no
yes
continue
no