Unit : 2

Software Analysis and Design

2.1 Requirement gathering and Analysis

- Requirement is playing **key role**.
- Done by **system analyst**.
- Removing all **ambiguities and inconsistencies** from customer perception.
- Mainly **two activities** are concerned with this task.

Requirement gathering	Requirement analysis

1. Requirement gathering:

- It is usually **the first part** of any software product.
- **This is the base** for the whole development effort.
- **Goal** → to collect all relevant information from the customer regarding the product to be developed.
- This is **done to clearly understand the customer requirements** so that incompleteness and inconsistencies are removed.

2.1 Requirement gathering and Analysis

- **Focusing on →** market analysis, customer demand
- It involves interviewing the end-users and studying the existing documents to collect all possible info.
- **Requirement gathering activities are**:
  - Studying the existing documents.
  - Interview with end users or customers.
  - Task analysis.
  - Scenario analysis.
  - Form analysis.
  - Brainstorming.
  - Questionnaires.

2. Requirement analysis:

- **Goal →** to clearly understand the exact req^n of the customer.

2.1 Requirement gathering and Analysis

- **IEEE defines requirements analysis** as (1) the process of studying user needs to arrive at a definition of a system, hardware or software requirements. (2) The process of studying and refining system, hardware or software requirements.
- Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.
- **It involves:**
  - *Eliciting requirements*
  - *Analyzing requirements*
  - *Requirements recording OR storing*
- **Some questions** might be understood by the analyst to obtain good system.

- What the problem?	- What i/o?
- Why it is imp to solve?	- What are complexities?
- What are the solutions?	- Data interchange format?

2.1 Requirement gathering and Analysis

- Analyst has to identify and eliminate the problems of anomalies, inconsistencies and incompleteness. **Anomaly** is the ambiguity in the requirement, **Inconsistency** contradicts the requirements, and **Incompleteness** may overlook some requirements.
- Analyst detects these problems by discussing with end-users.
- Finally, make sure that requirements should be specific, measurable, timely, achievable and realistic.
- **Output ➔** SRS (system requirements specification).

2.2 Software requirement specification

- 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** the system.
- It describes ***what*** the proposed system should do without describing ***how*** the software will do.
- It is working as a **reference document** to the developer.
- It **provides guideline** for project development.
- SRS is actually **serves as a contract** between developer and end user.
- The SRS **translates the ideas** of the customers (input) into the formal documents (output).

2.2 Software requirement specification

- The SRS document is known as **black-box specification**, because:
  - In SRS, internal details of the system are not known (as SRS doesn't specify *how* the system will work).
  - Only its visible external (i.e. input/output) behaviour is documented.
- The organization of SRS is **done by the system analyst**.

➥ Benefits of SRS.

- SRS provides **foundation for design** work.
- It **enhances communication** between customer and developer.
- Developers can get the idea **what exactly the customer wants**.
- It enables project planning and helps in verification and validation process.
- High quality SRS **reduces the development cost and time** efforts.

2.2 Software requirement specification

- SRS is also useful during the maintenance phase.

➥ Contents of the SRS document.

- An SRS should clearly document the following things:

1. Functional requirements of the system

- The functional requirements are the services which the end users expect the final product to provide.
- It clearly describes each of the function that the system needs to perform along with the input and output data set.

2. Non-functional requirements of the system

- The non functional requirements describe the characteristics of the system that can't be expresses functionally. E.g. portability, maintainability, usability, security, performance etc.

2.2 Software requirement specification

3. Constraint on the system

- That describes what the system should do or should not do. These are some general suggestions regarding development.
- A constraint can be classified as:
  - Performance constraint
  - Operating constraint
  - Economic constraint
  - Life cycle constraint
  - Interface constraint

➥ Characteristics of a good SRS.

- Characteristics of a good SRS are as follows:

Concise	Structured	Verifiable	Portable
Complete	Conceptual integrity	Adaptable	Unambiguous
Consistent	Black box view	Maintainable	Traceable

2.2 Software requirement specification

➥ Examples of bad SRS.

- Over specification
- Forward references
- Wishful thinking
- The SRS documents that contain incompleteness, ambiguity and contradictions are considered as bad SRS documents.

◈ Functional requirements.

- 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.

2.2 Software requirement specification

- Key goal → to capture the behaviour of software in terms of functions and technology.
- A function is described as a set of inputs, process and a set of outputs.
- Functional requirements may be calculations, data manipulations and processing, technical details or other specific functionalities that define what a system is suppose to do.
- Functional requirements of the system are captured in use cases.
- Functional requirements drive the *application architecture* of the system while non functional requirements drive the *technical architecture.*

2.2 Software requirement specification

➥ How to identify functional requirements?

- We can identify functional requirements:
  - From informal problem description or from conceptual understanding of the system.
  - Identify from user perspective.
  - Find out higher level function requirements.

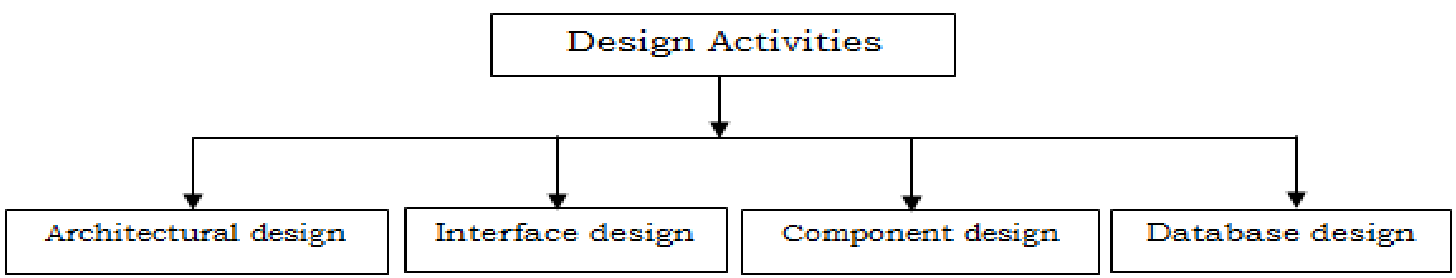
➥ How to document the functional requirements?

- Document the functionalities supported by the system.
- Specify the input data domain, processing and output data domain.
- Functional requirements are specified by different scenarios.

➥ Example: operations at ATM.

2.3 Design Process

- The design process is a sequence of steps to describe all aspects of the software.
- It specifies *how* aspect of the system.
- Purpose → plan a solution of the problem specified in SRS.
- It includes: user interface design, i/o design, data design, process and program design and technical specification etc.
- It convert SRS into program appropriate form for implementation.
- Output → design documents.
- **Classification of design activities:**



Design activities

2.3 Design Process

- *Architectural design:*
- Identify overall structure of the system, subsystem, modules and their relationship.
- Can be represented using DFD.
- *Interface design*
- Defines the interface between system components.
- It describes how system communicates with itself and with the user also.
- It can be derived from DFD and State transition diagram.
- *Component design*
- Defines each system component and show how they operate.
- It can be derived from State transition diagram.

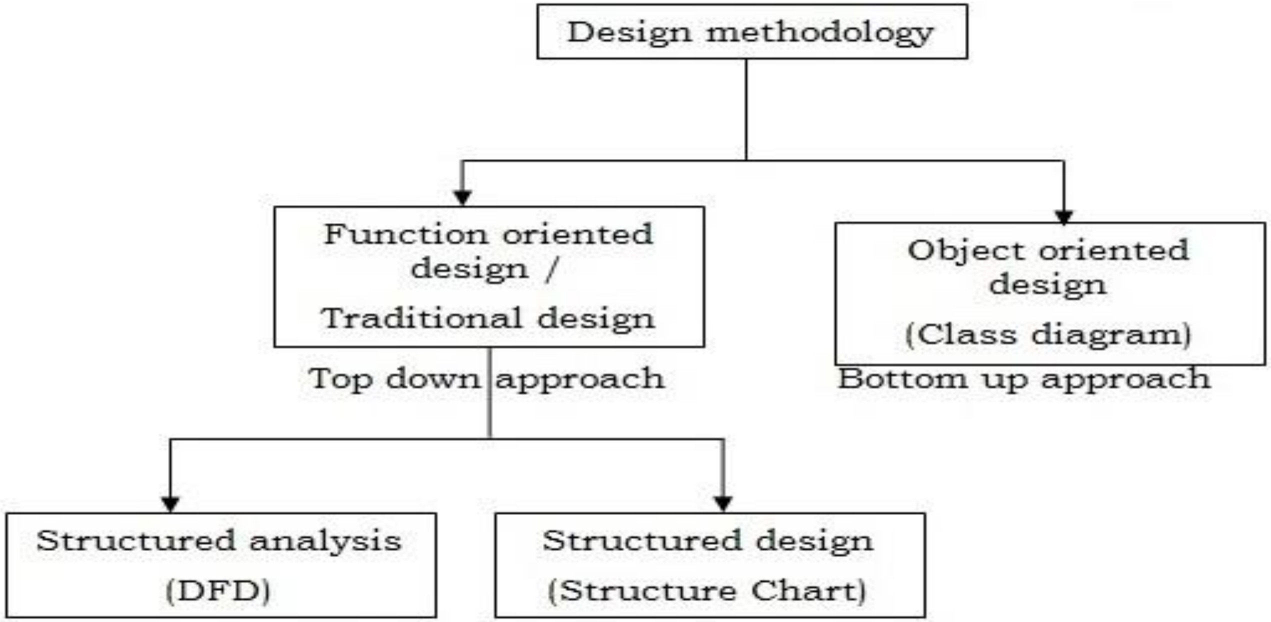
2.3 Design Process

- *Database design:*
- Defines the data structure of the system.
- Existing database can be reused or a new database to be created.
- ER Diagram and DD (data dictionary) used for representation.
- **Classification of design methodologies:**
- Design methodologies are followed in software development from beginning up to the completion of the product.
- Used to provide guidelines for the design activity.
- The nature of the design methodologies are dependent on the following factors:

The software development environment	Qualification and training of the development team
The type of the system being developed	Available software and hardware
User requirements	

2.3 Design Process

- Classification of design methodologies is shown in the figure.



Design methodologies

- There are fundamentally two different approaches:

Function oriented design	Object oriented design

2.3 Design Process

- **Function oriented design.**
- Set of functions are described.
- Top down approach.
- Data in the system is centralized and shared among different functions.
- Function oriented design further classified into → Structure analysis and Structure design.
- **Structure analysis**
- It is used to transform a textual description into graphical form.
- It examines the detail structure of the system.
- It identifies the processes and data flow among these processes.
- In structure analysis → SRS is transformed into DFD.

2.3 Design Process

- **Structure design**
- Results of structured analysis are transformed into the software design.
- Functions are mapped into modules.
- Aim → transform DFD into structure chart.
- Represented using structure chart.
- Two main activities:
- Architectural design (High – level design)
- Detailed design (Low – level design)
- Problem: require entire search when change is made in one part
- **Object oriented design.**
- Objects and their relationships are identified.
- It is built using bottom-up approach.
- Each object is a member of class.

2.3 Design Process

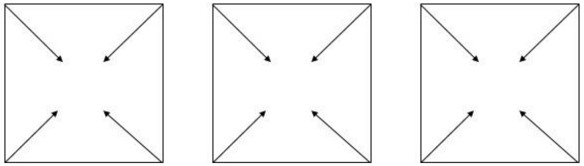
- Data in the system is not centralized and shared but is distributed among the objects in the system.
- Three main concepts:
  - Encapsulation: combining data and functions into a single entity
  - Inheritance: provide reusability
  - Polymorphism: same context can be used for different purposes
- **Advantages of object oriented design.**
- Reduce maintenance.
- Provide code reusability, reliability, modeling and flexibility.
- Provide robustness to the system.
- Provide consistency from analysis through design to coding.

2.4 Cohesion and Coupling

- Modularity is a good property of software development.
- Modular system
- Cohesion and coupling are two modularization criteria.
- ***'high cohesion and low coupling'*** is needed for good development.

Cohesion

- It is a measure of functional strength of a module.
- ***Cohesion keeps the internal modules together, and represents the functional strength.***
- Cohesion of a module represents how tightly bound the internal elements of a module are to one another.



Cohesion = strengths of relations within modules

2.4 Cohesion and Coupling

◈ Classification of cohesion.

Coincidental	Logical	Temporal	Procedural	Communicational	Sequential	Functional

Worst (Low) → Best (High)

- **Coincidental cohesion.**
- Lowest cohesion
- It occurs when there are no meaningful relationships between the elements.
- **Logical cohesion.**
- If there is some logical relationships between the elements of module.
- The elements perform functions that fall into same logical class.
- For example: the tasks of error handling, input and output of data.

2.4 Cohesion and Coupling

- **Temporal cohesion.**
- Temporal cohesion is same as logical cohesion except that the elements are also 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, shut down are usually having temporal cohesion.
- **Procedural cohesion.**
- When module contains elements that belong to common procedural unit.
- A module is said to have procedural cohesion, if the set of the module are all part of a procedure (algorithm) in which certain sequence of steps are carried out to achieve an objective.
- Example: algorithm for decoding a message.

2.4 Cohesion and Coupling

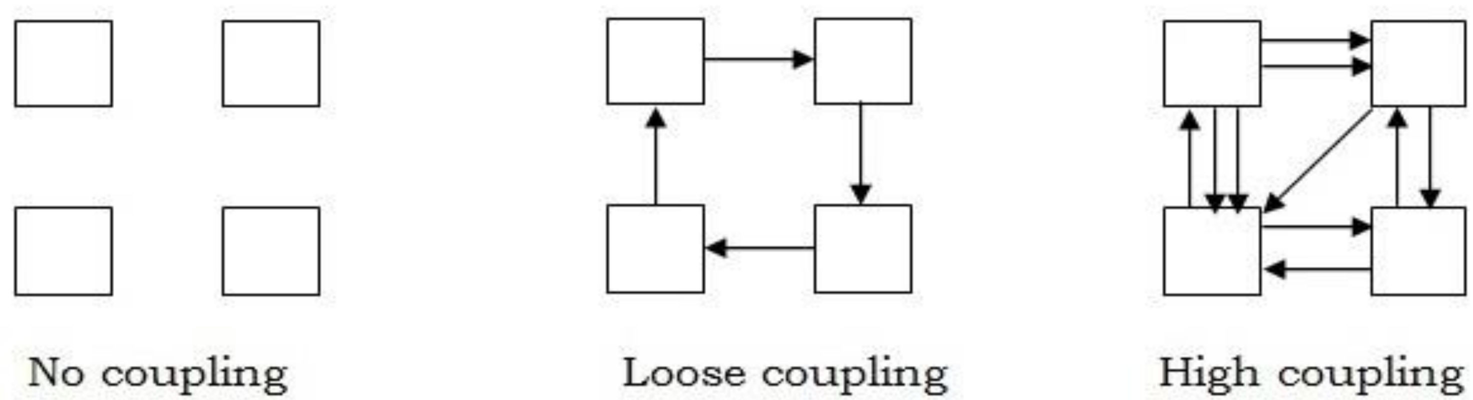
- **Communicational cohesion.**
- If all functions of the module refer to or update the same data structure. e.g. the set of functions defined on an array or a stack.
- These modules may perform more than one function together.
- **Sequential cohesion.**
- When the output of one element in a module forms the input to another, we get sequential cohesion.
- It does not provide any guideline how to combine these elements into modules.
- For example TPS system.
- **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.
- Example: compute square-root of the funcion.

2.4 Cohesion and Coupling

Coupling

- Coupling between two modules is a measure of the degree of interdependence or 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.
- As modules become more interdependent, the coupling increases.
- Loose coupled and high coupled.

2.4 Cohesion and Coupling



◈ Classification of coupling.

Data	Stamp	Control	Common	Content

Best (Low) → Worst (High)

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

2.4 Cohesion and Coupling

- **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.
- **Control coupling.**
- Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.
- Example: is a flag set in one module and tested in another module.
- **Common coupling.**
- Two modules are common coupled, if they share data through some global data items.
- **Content coupling.**
- Content coupling exists, if two modules share code, e.g. a branch from one module into another module.
- It is the highest coupling and creates more problems in software development.

Functional independence

- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- So, that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- For good s/w design neat decomposition is highly needed, and the primary char of neat decomposition is ‘*high cohesion and low coupling*’.

↪ Need of functional independence.

- It is a good key to any software design process due to following reasons.
  - *Error isolation*
  - *Scope of reuse*
  - *Understandability*

2.5 Data modeling concepts

- A data model is a conceptual relationship of data structure (tables) required for a database.
- It concerned with structure rather than rules.
- To avoid the redundancy of database, there is a need to create data model.
- Data model provides abstract and conceptual representation of data.
- Data modeling or ER diagram gives the concepts of objects, attributes and relationship between objects.
- ER diagram is a structured analysis technique. And also describes logical data design that can converted easily into table structure.
- ERD is a snapshot of data structure.
- ER diagram can be used to model the data in the system.
- ER diagram enables a software engineer to identify data objects and their relationships using graphical notations.

2.5 Data modeling concepts

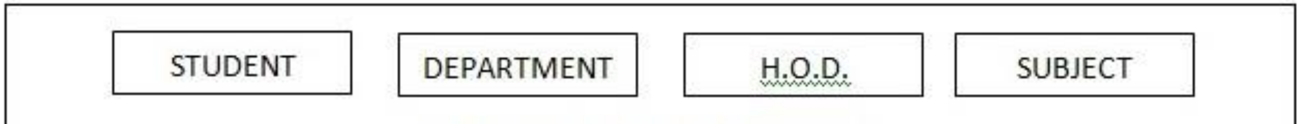
- ERD is a detailed logical representation of any system. It has three main elements → data object (entity), attributes and their relationships.

i. Data objects (Entity set)

- A data object is a real world entity or thing.
- Data object is a fundamental composite information system.
- An entity represents a thing that has meaning and about which you want to store or record data.
- It can be external entity, a thing, an organization, a place or an event. For example: for a college → department, students, head of the department and students may be entities.
- It has number of properties or attributes. Each object has its own attributes.

2.5 Data modeling concepts

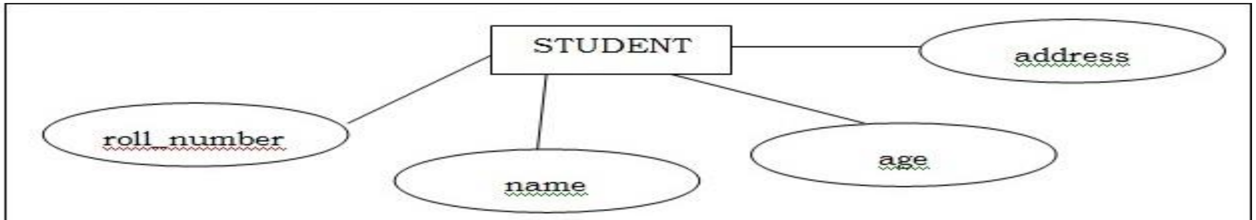
- Entities are represented using *rectangle box* and preferably written in capital letters.



Entity set for a college

ii. Attributes

- An attribute is a property or characteristic of an entity.
- Attributes provide meaning to the objects.
- Attributes must be defined as a identifier, and that become key to find instance of object.
- Attributes represented using *oval*.

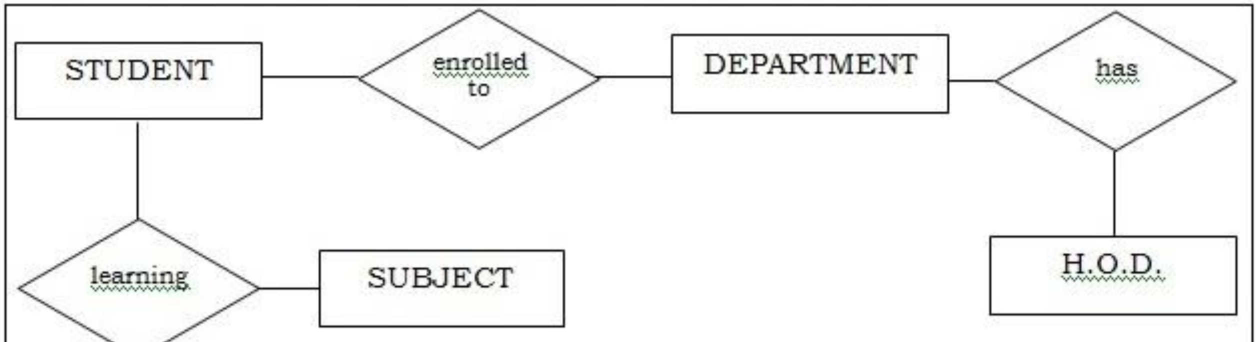


Attributes of entity student

2.5 Data modeling concepts

iii. Relationship

- Entities are connected to each other via relations.
- Generally relationship is binary because there are two entities are related to each other.
- It illustrates sharing of information.
- Relationship of objects is bidirectional, so they can be read in either side.
- Relationship is represented using diamond shape symbol with joined relationship name.



Relationship of entities

2.5 Data modeling concepts

Cardinality

- The elements of data modeling - data objects, attributes, and relationships - provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.
- It is also important how many occurrences of any object are related to how many occurrences of other object. This leads to a data modeling concept called cardinality.
- The concept of cardinality defines the maximum number of objects that can participate in a relationship. That means number of occurrences of one [object] that can be related to the number of occurrences of another [object].
- Cardinality is usually expressed as simply 'one' or 'many.'

2.5 Data modeling concepts

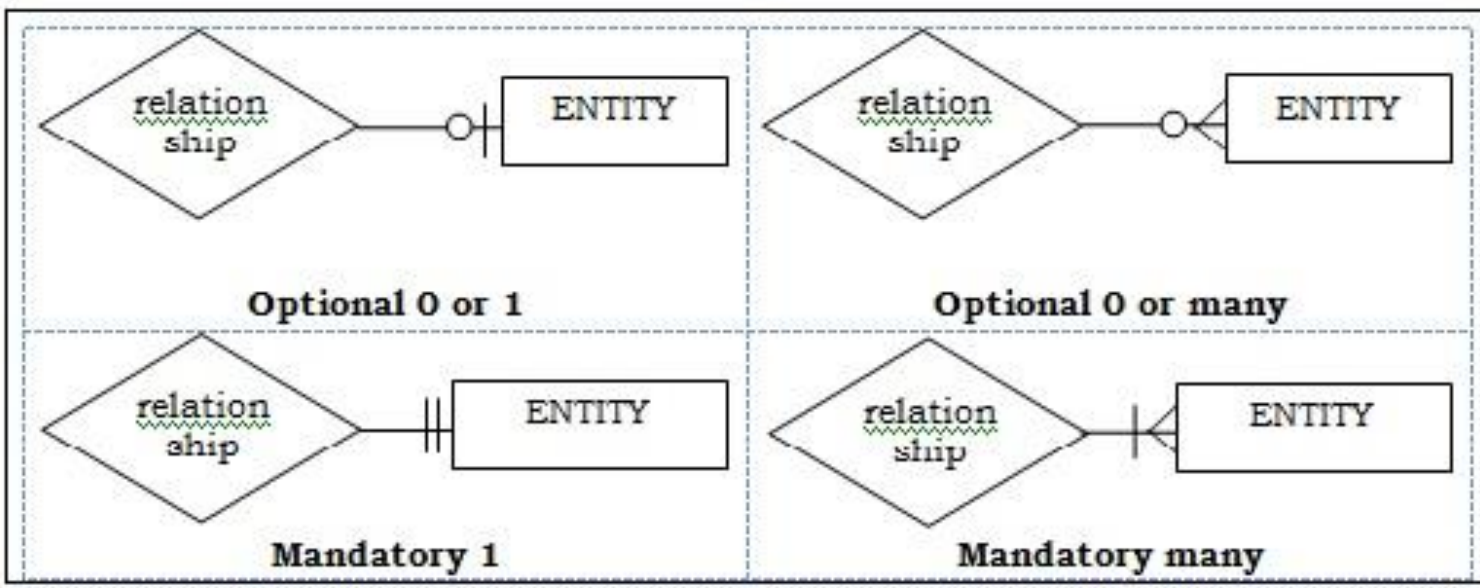
- Different cardinalities are explained below:
- **One to One (1 : 1)**
- Ex → college has principal
- **One to Many (1 : M)**
- Ex → mother and children, college and students
- **Many to Many (M : M)**
- Ex → students and subjects, uncles and nephews

Modality

- Cardinality does not, however, provide an indication of whether or not a particular data object must participate in the relationship.
- To specify this information, the data model adds modality concept to the object/relationship pair.

2.5 Data modeling concepts

- Modality is form of cardinality.
- Modality means a classification of relationships on the basis of whether they claim necessity, possibility or impossibility.
- The modality of relationship is 0 or optional or 1.
- The notations for modality are explained below.



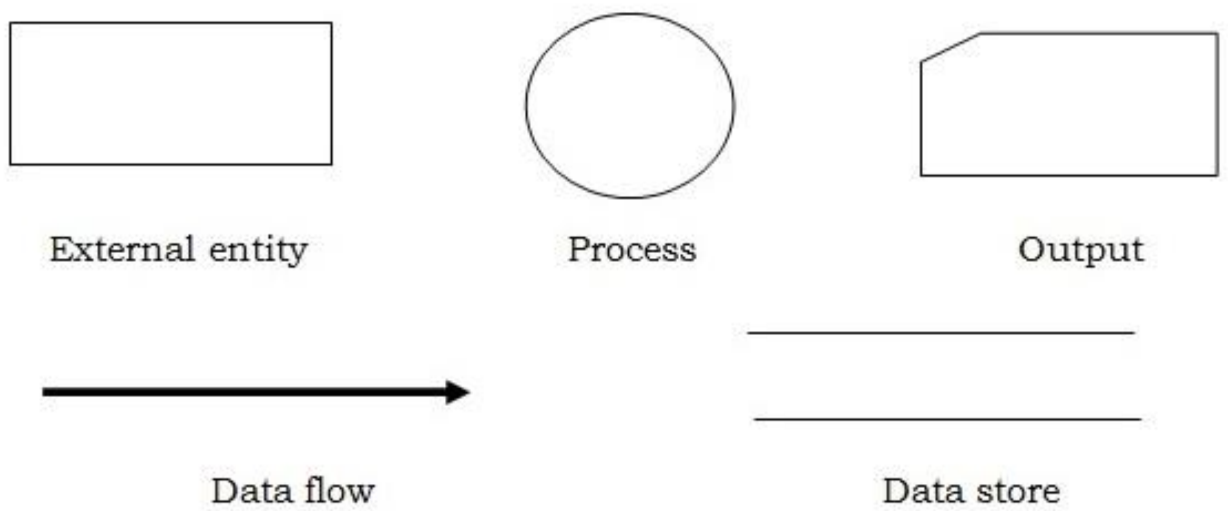
2.6 Data Flow Diagrams

- DFD (Data Flow Diagram) is also known as bubble chart or data flow graph.
- DFDs are very useful in understanding the system and can be effectively used during analysis.
- 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.
- Each function is considered as a process that consumes some input data and produces some output data.
- The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.
- Functional model can be represented using DFD.

2.6 Data Flow Diagrams

◈Primitive symbols used in construction of DFD model.

- DFD used limited number of primitive symbols.



DFD symbols

⤷ Process:

- It is s represented by circle or bubble. Circles are annotated with names of the corresponding functions.

2.6 Data Flow Diagrams

- A process shows the part of the system that transforms inputs into outputs.
- The process is named using a single word that describes *what* the system does functionally.

⤷ **External entity:**

- Entity is represented by a rectangle.
- Entities are external to the system which interacts by inputting data to the system or by consuming data produced by the system.
- It can also define source (originator) or destination (terminator) of the system.

⤷ **Data flow:**

- Data flow is represented by an arc or by an arrow.
- It used to describe the movement of the data.

2.6 Data Flow Diagrams

- It represents the data flow occurring between two processes, or between an external entity and a process. It passes data from one part of the system to another part.
- Data flow arrows usually annotated with the corresponding data names.

➥ **Data store:**

- Data store is represented by two parallel lines.
- It is generally a logical file or database.
- It can be either a data structure or a physical file on the disk.

➥ **Output:**

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

2.6 Data Flow Diagrams

◈ Developing DFD model of the system.

- DFD starts with the most abstract level of the system (lowest level) and at each higher level, more details are introduced.
- To develop higher level DFDs, processes are decomposed into their sub functions.
- The abstract representation of the problem is also called *context diagram*.

↪ Context diagram (Level 0 DFD).

- The context diagram is top level diagram; it is the most abstract data flow representation of a system.
- It only contains one process node that generalizes the function of entire system with external entities. (It represents the entire system as a single bubble.)
- Data input and output are represented using incoming and outgoing arrows.

2.6 Data Flow Diagrams

↪ **Level 1 diagram.**

- To develop the level 1 DFD, we have to examine the high-level functional requirements.
- It is recommended that 3 to 7 functional requirements can be directly represented as bubbles in 1st level.

↪ **Further Decomposition.**

- The bubbles are decomposed into sub-functions at the successive levels.
- Decomposition of a bubble is also known as factoring or exploding a bubble.
- Each bubble at any level of DFD is usually decomposed between 3 to 7 bubbles in its higher level.
- It's not a rule that particular number of levels are needed for the system.

↪ **Numbering the bubbles.**

2.6 Data Flow Diagrams

➥ **Some care should be taken while constructing DFDs (Guidelines when drawing DFDs)..**

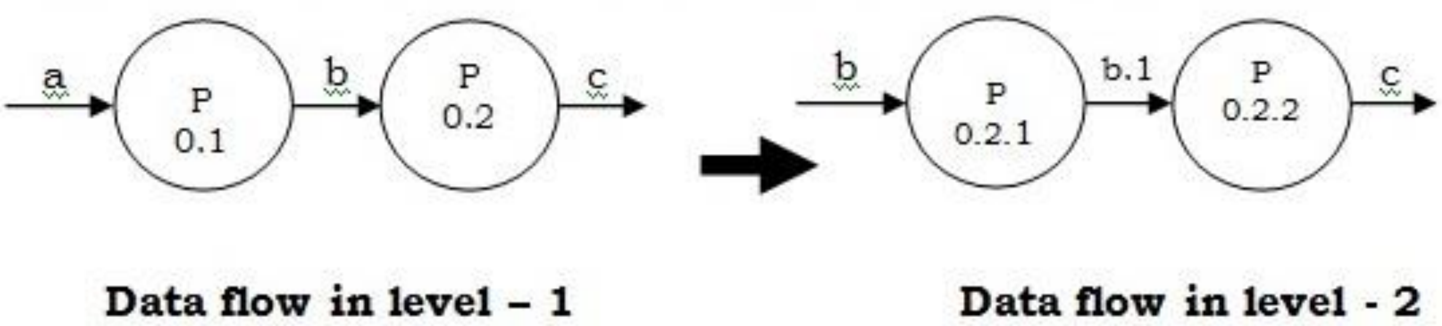
- A process must have at least one input and one output data flow.
- A process must have at least one input and one output data flow.
- No control information (If-THEN-ELSE) should be provided in DFD.
- A data store must always be connected with a process. A data store cannot be connected to another data store or to an external entity.
- Data flows must be named.
- Data flows from entities must flow into processes, and data flows to entities must come from processes.

2.6 Data Flow Diagrams

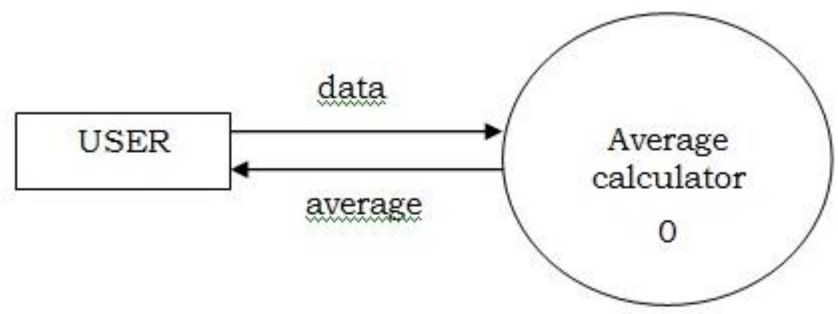
- There should not be detailed description of process in context diagram.
- Name of the data flow should be noun and name of process should be verb.
- Each low level DFD must be balanced to its higher level DFD (input and output of the process must be matched in next level).
- Data that travel together should be one data flow.
- No need to draw more than one bubble in context diagram.
- Generally all external entities interacting with the system should be represented only in the context diagram.
- Be careful with number of bubbles in particular level DFD, as too less or too many bubbles in DFD is oversight.
- All the functionalities of the system specified in SRS must be captured by the DFD model.

2.6 Data Flow Diagrams

- Balancing DFD.

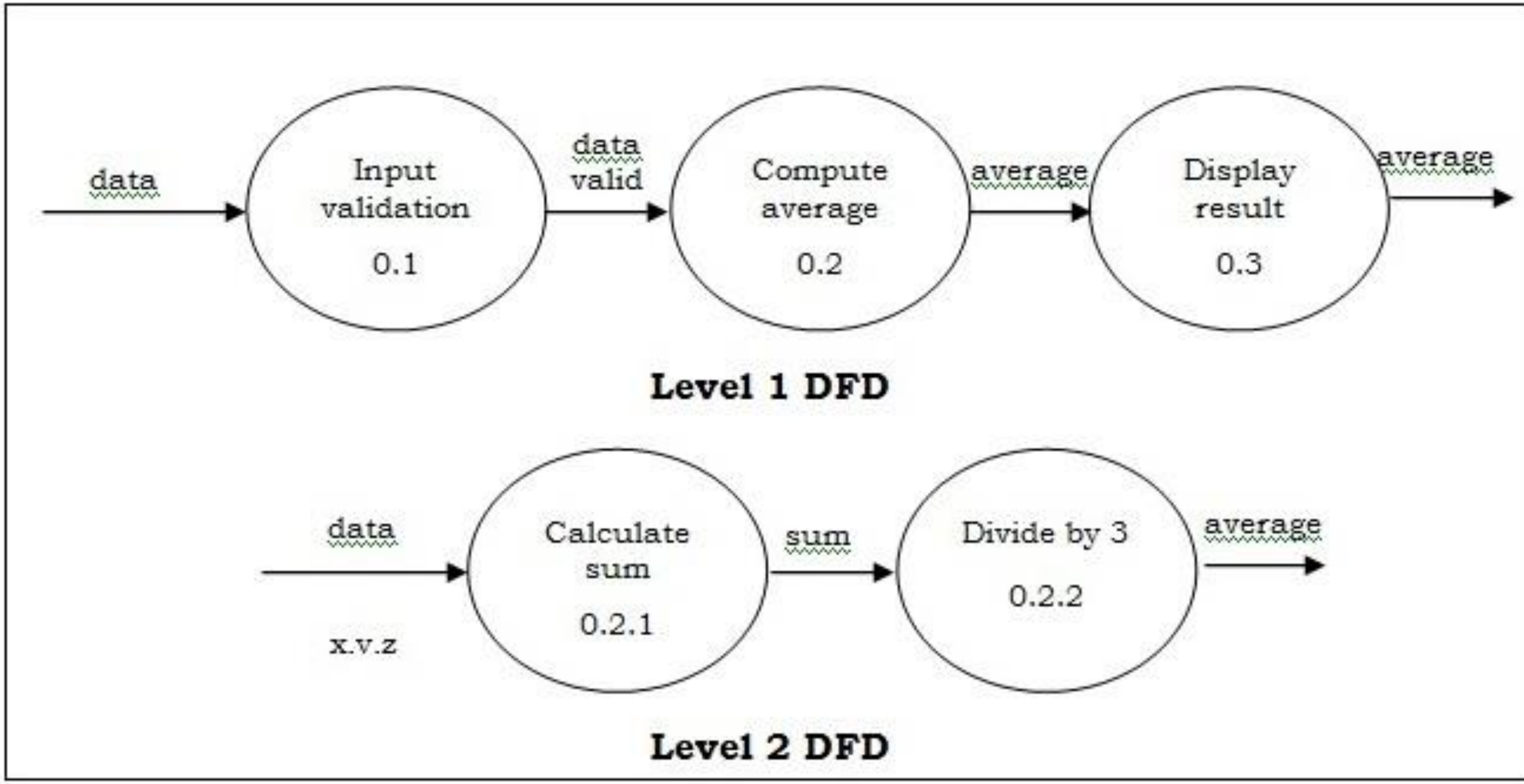


- **Simple example of average calculator of three numbers.**



Level 0 (context diagram)

2.6 Data Flow Diagrams



2.6 Data Flow Diagrams

⤷ Advantages of DFD model.

- It is very simple to understand and easy to use.
- DFD can provide detailed description of the system components.
- It provides clear understanding to the developers about the system boundaries.
- It explains the logic behind the data flow within system.
- It is not only useful to represent the results of structured analysis, but also for several other applications like showing the flow of documents or items in an organization.
- Symbols used in DFD model are very less.

⤷ Disadvantages of DFD model.

- Control information is not defined by a DFD.
- No specific guidance for exact decomposition,
- Sometimes it puts programmers in little confusing state.
- Different models of DFD have different symbols.

2.7 Scenario based modeling

- A scenario describes a set of actions that are performed to achieve some specific condition. And this set is specified as a sequence.
- Each step in scenario is performed by an actor or by a system.

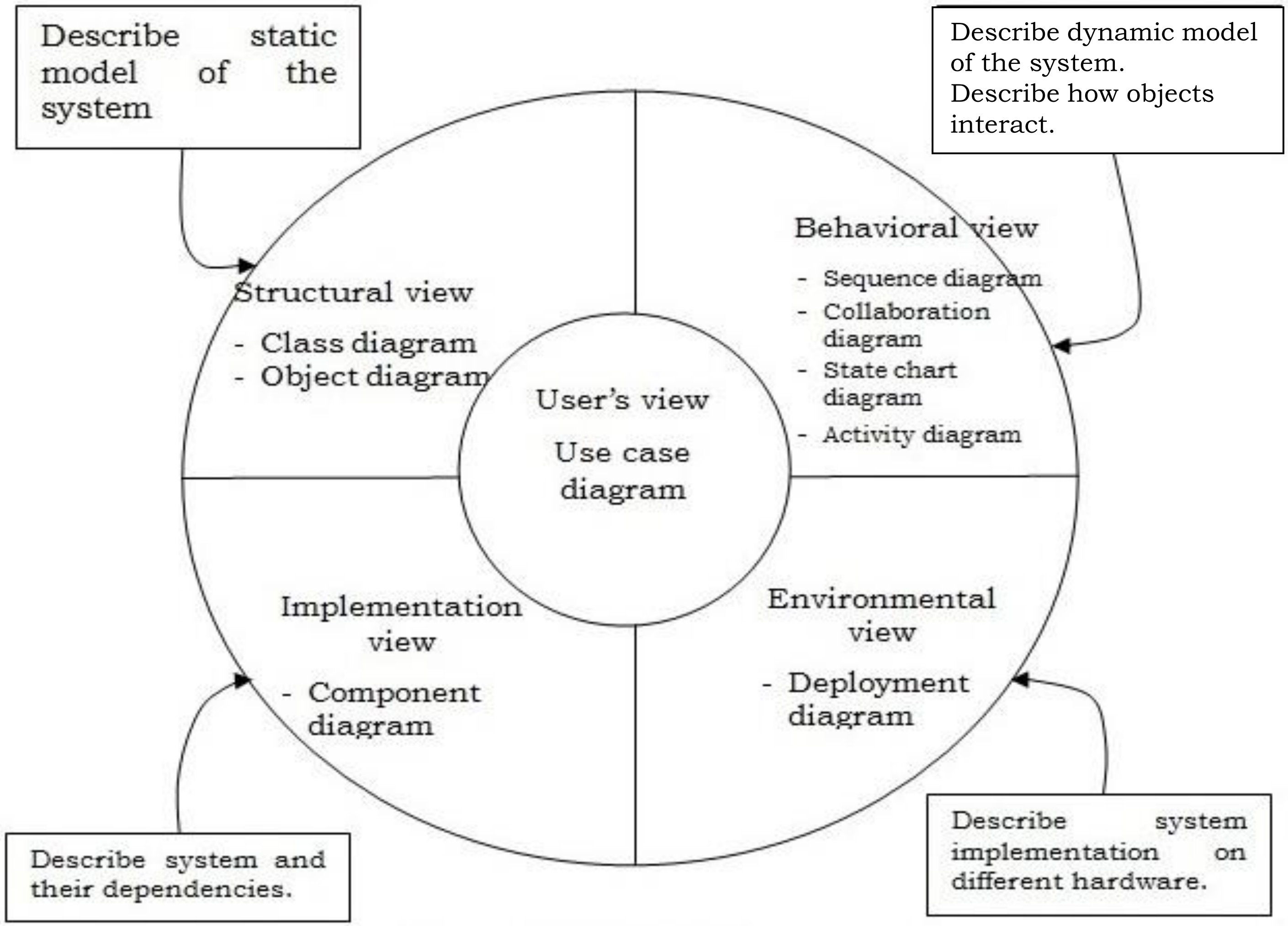
UML (Unified Modeling Language)

- UML is a modeling language.
- UML designs provide a standard way to visualize the design of the system.
- UML is very useful in documenting the design and analysis results.
- UML is not a design methodology.
- UML making system easy to understand using less number of primitive symbols.

2.7 Scenario based modeling

➥ UML Diagrams:

- UM ... grams
  to c
- UM ... ftware
  syst



UML diagrams

2.7 Scenario based modeling

Use-Case diagram

- It provides system behavior.
- Use case model of the system consists of a set of use cases.
- Use cases represent the different ways in which a system can be used by the users.
- The purpose of a use case is to define the logical behavior of the system without knowing the internal structure of it.
- It provides understanding of the system.
- It identifies the functional requirements of the system.
- UML describes "*who can do what in a system*".
- A use case typically represents a sequence of interactions between the user and the system.
- For example → in Bank transaction system, the system should have many use cases like:

2.7 Scenario based modeling

➥ **Components of use case diagram (Representation of use case diagram).**

- Two main components along with relationships are used in use case diagram.

1. Use case:

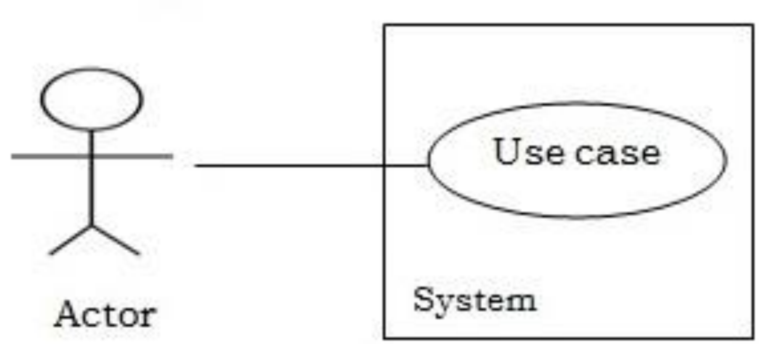
- Represented by an ellipse with the name of the use case written inside the ellipse, named by a verb.
- All the use cases are enclosed with a rectangle representing system boundary. Rectangle contains the name of the system.
- It identifies and analyzes the fun req^n of the system.

2. Actor:

- An actor is anything outside the system that interacts with it, named by noun.
- Actors are represented by using the stick person icon.

2.7 Scenario based modeling

- An actor may be a person, machine or any external system.
- Actors are connected to use cases by drawing a simple line connected to it. Actor triggers use cases.
- Each actor must be linked to a use case, while some use cases may not be linked to actors.



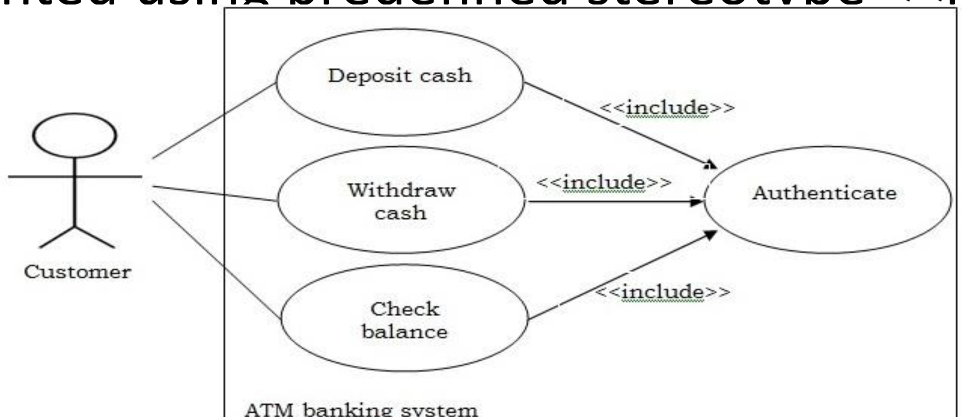
- ***Relationship:***
- It is also called communication relationship. Actors are connected to use cases through relationship lines.
- An actor may have relationship with more than one use case and one use case may relate to more than one actor.

2.7 Scenario based modeling

➥ *Different relationships in use case diagram are explained below:*

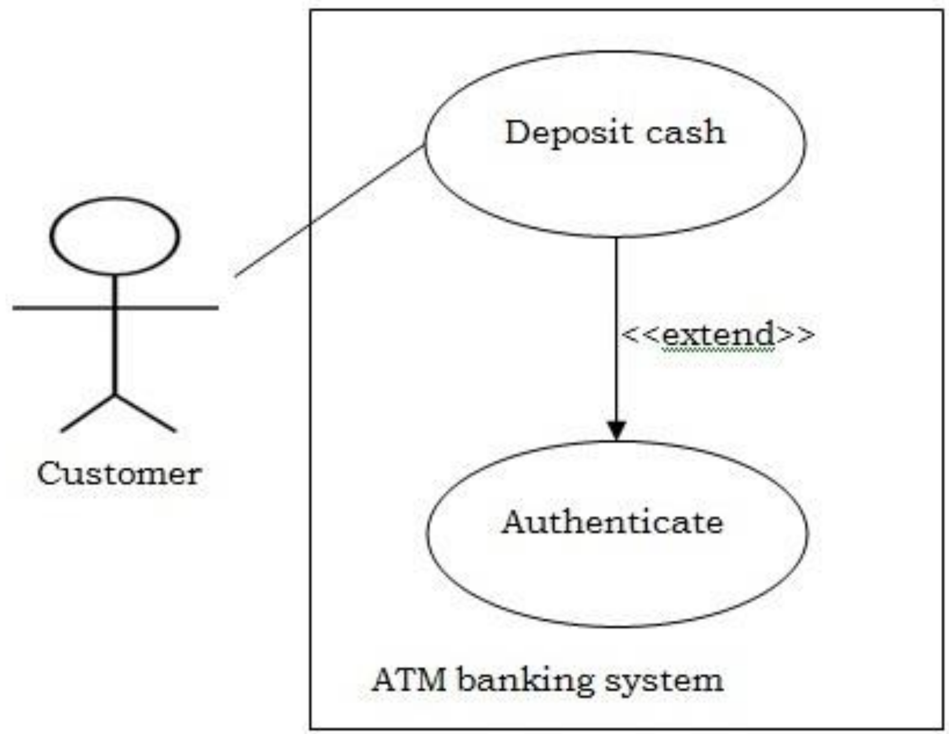
❖ **Association:**

- It is the interface between an actor and a use case.
- It is represented by joining a line from actor to use case.
- **Include relationship:**
- It involves one use case including the behavior of another use case.
- The “include” relationship occurs when a chunk of behavior that is similar across a number of use cases.
- It is represented using predefined stereotype <<include>>.
- Ex.



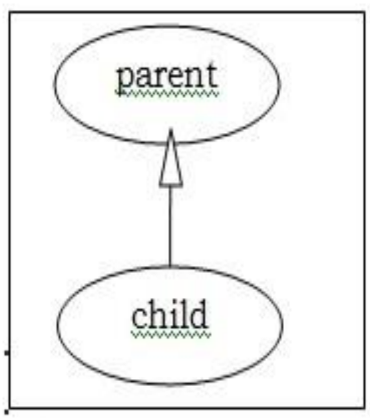
2.7 Scenario based modeling

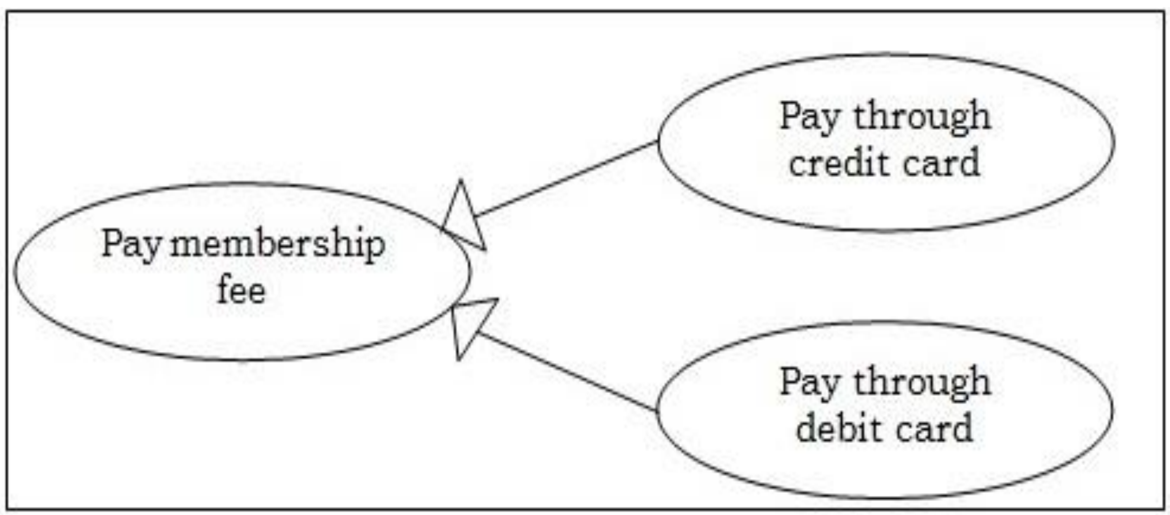
- **Extend relationship**
- It shows optional behavior of the system.
- represented as a stereotype <<extend>>.
- Extend relationship exists when one use case calls another use case under certain condition (like: If – then condition).



2.7 Scenario based modeling

- **Generalization**
- Used when you have one use case that is similar to another, but does slightly different.
- It is a link between use cases. In which the child use case inherits the behavior of parent use case.
- The child may override the behavior of its parent.

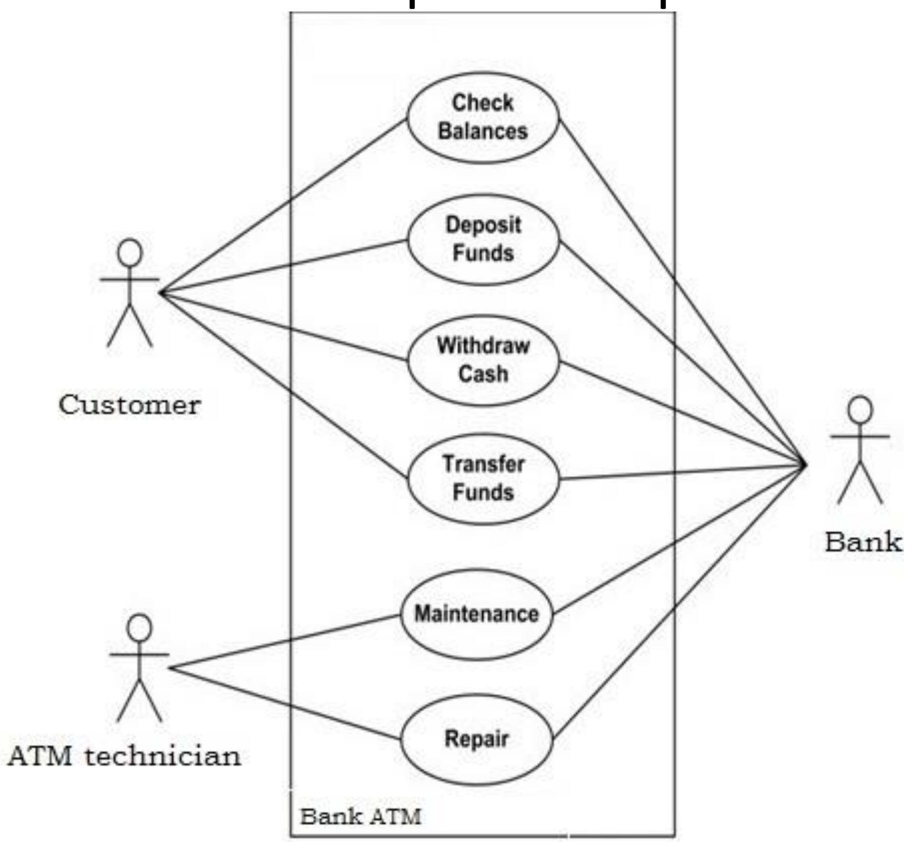




2.7 Scenario based modeling

❖ Use case guidelines

- Identify all different users. Give suitable names.
- For each user, identify tasks. These tasks will be the use cases.
- Use case name should be user perspective.
- Show relationships and dependencies.



- **Application of use case.**
  - Requirement analysis
  - Reverse engineering
  - Forward engineering

2.7 Scenario based modeling

Activity Diagram

- It falls under the category of behavioral diagram in UML.
- Used to model the process. It models the behavior of the system components.
- 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.
- It focuses on the how of activities involved in a single process.
- Activity diagrams represent workflows in a graphical way.
- Aim → to record the flow of control from one activity to another of each actor and to show interaction between them.
- It supports parallel activities.
- An activity is a state with an internal action and one or more outgoing transitions.

2.7 Scenario based modeling

- An interesting feature of the activity diagrams is the ***swimlanes.*** It enables you to group activities based on who is performing them. So, swimlanes make group of activities based on actors.
- **Elements (components) of an activity diagram.**

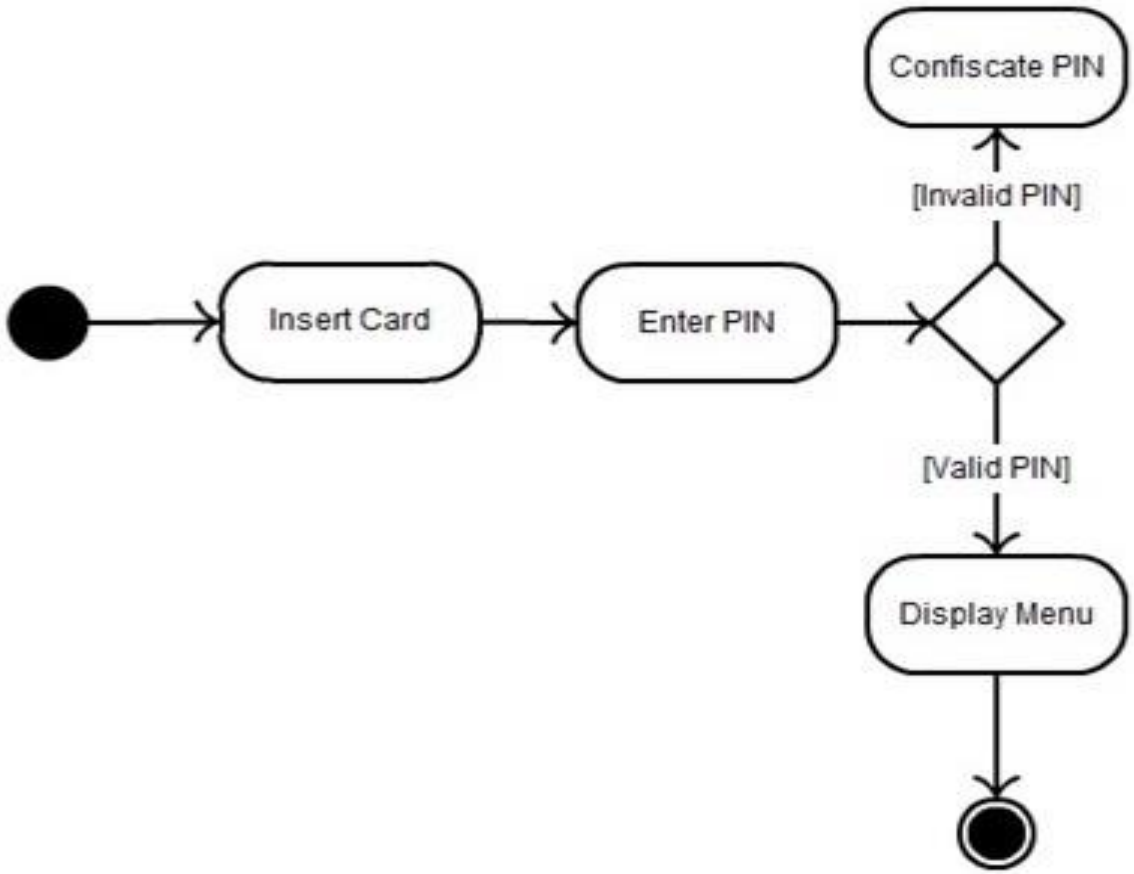
Component	Description
Activity	• It represents a particular action taken in the flow of control. • It is denoted by a rectangle • There are two special type of activity nodes: • Initial activity and final activity
Flow Transition	• Represented with a directed arrow.
Decision	• Represented with a diamond. • Single transition enters and several outgoing transitions
Merge	• This is represented with a diamond shape with two or more input transitions and a single output.

2.7 Scenario based modeling

Component	Description
Fork	• Fork is a point where parallel activities begin. • Fork is denoted by black bar with one incoming transition and several outgoing transition. • When the incoming transition is triggered, all the outgoing transitions are taken into parallel.
Join	• Join is denoted by a black bar with multiple incoming transitions and single outgoing transition. • It represents the synchronization of all concurrent activities.
Note	• UML allows attaching a note to different components of diagram to present some textual information. • It is denoted by a rectangle with cut a side.
Partition / Swimlanes	• Different components of an activity diagram can be logically grouped into different areas, called partition or swimlanes. • It is denoted by drawing vertical parallel lines.

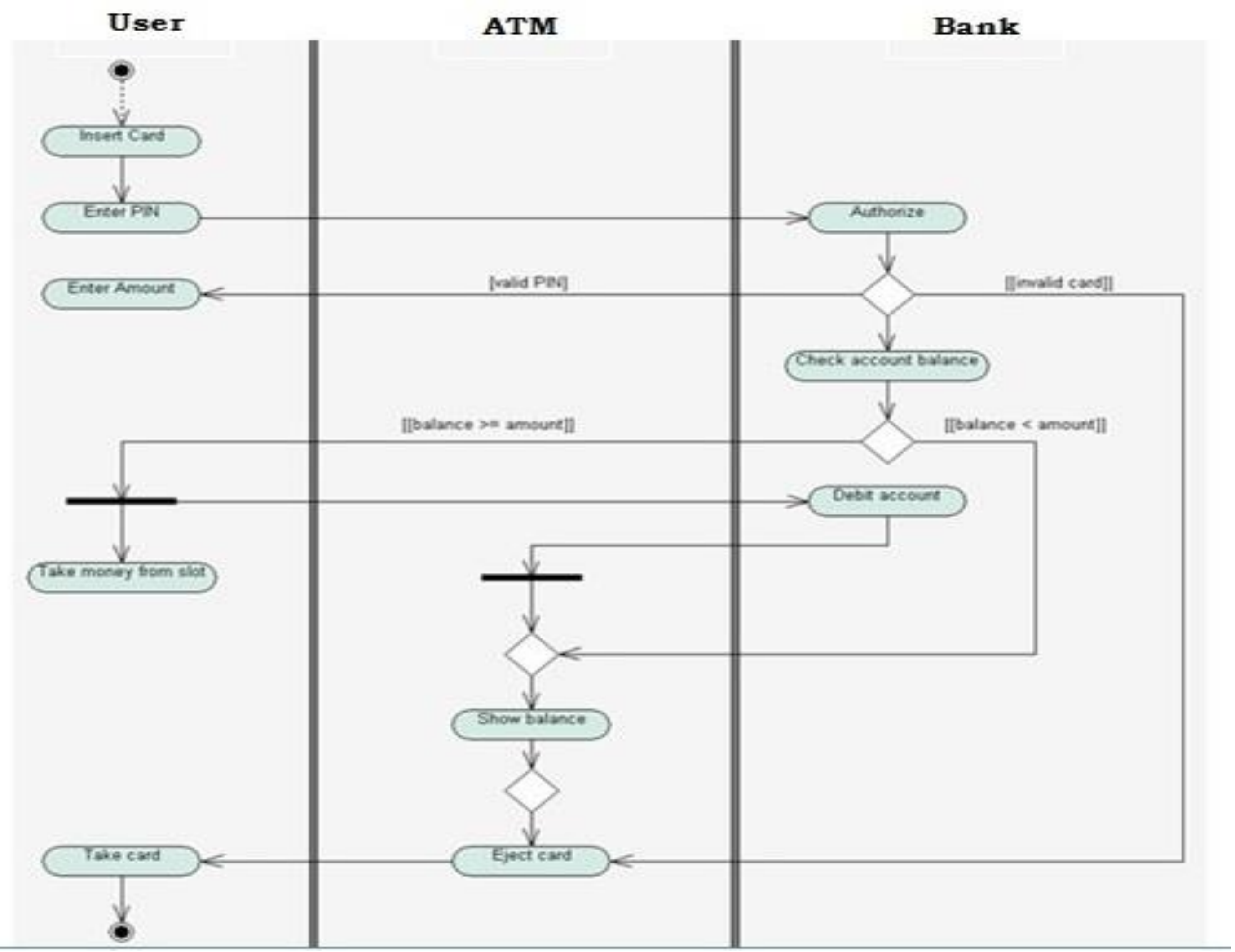
2.7 Scenario based modeling

➡ **A simple example activity diagram (ATM system).**



2.7 Scenario based modeling

➡ Another example showing swimlanes.



2.7 Scenario based modeling

❖ **Advantages of activity diagram.**

- Very useful to understand complex processing activities.
- Swimlanes.
- Provides understating workflow of system.
- It is good for describing synchronization and concurrency between activities.
- Provides responsibilities for interactions and associations between objects and actors.

❖ **Disadvantages of activity diagram.**

- Doesn’t provide message part.
- It can’t describe how objects collaborate.
- Complex logic can’t be represented.

❖ **Application of activity diagram.**

- Used to describe parallel behavior of the system.
- Also used in multi threaded programming application.

2.7 Scenario based modeling

☛ Difference between flowchart and activity diagram.

Flow chart	Activity diagram
- It is limited for sequential access.	- It is used for parallel and concurrent processing.
- It is used for flow of control through an algorithm, not used for object oriented procedure.	- It is usually used for object oriented systems.
- Concept of swimlanes is not there in it.	- It has the functionality of swimlanes.
- It has limited functionalities compare to activity diagram.	- It has more functionalities.

2.8 Architectural design decisions

◈Software architecture.

- It is a framework that describes its form and structure, its components and how they interact together.
- To understand any complex system, we need to have the knowledge of its subsystems and their interactions.
- ***Software architecture*** is a description of the subsystems and components of a software system and the relationships between them.
- Software architecture should be of individual programs or it should be included of different sub-systems.

2.8 Architectural design decisions

◈ Architectural design.

- Software architectural design is a description of how a system is organized.
- You can identify the overall structure of the system, sub-systems, modules and their relationships.
- It is derived from the DFD of the system.
- The output of architectural design is → architectural model.

➥ Architectural design decisions.

- Made by system architects.
- Based on type of system.
- Depend on functional and non-functional requirements.

➥ Some design decisions:

2.8 Architectural design decisions

◈Architectural views.

- Architectural views represent the system as composed of types of elements and relationships between them.
- Different views expose different properties and attributes.
- Different views reduces the complexities of the system and help in understanding and analyzing the system.
- Different types of proposed architectural views are:

⤷ Module view:

- The system is viewed as a collection of code units.
- The main element in this view is modules.
- This view is code based and do not explicitly represent any runtime structure of the system.
- Examples of modules are packages, class, a method, collection of functions etc.

2.8 Architectural design decisions

⤷ **Conceptual view:**

- It is an abstract view of the system.
- It shows detailed decompositions of the system.

⤷ **Logical view:**

- It shows key concepts of the system as objects and classes.
- Objects and their relationships can be identified.

⤷ **Component and connector view.**

- In it, system is viewed as a collection of runtime entities called components, which support in executing the system.
- While executing, components need to interact with others to support the system. And connectors provide mean for this interaction.
- Examples of connectors are pipes and sockets. Shared data can also act as connectors.

2.8 Architectural design decisions

↪ Allocation view:

- It focuses on how different software units are allocated to recourses like hardware, file systems and people.
- Allocation view specifies the relationship between software elements and environmental elements in which the software system is executed.

↪ Process view:

- In it, system is composed of interacting process at run time.

↪ Development view:

- It shows the breakdown structure of software into modules.

↪ Physical view:

- It shows the system hardware and how software components are distributed across them.

2.8 Architectural design decisions

◈ Architectural patterns.

- Patterns are a means of representing, sharing and reusing knowledge.
- Architectural patterns are a means of reusing knowledge about generic system architectures.
- An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture.
- It is the description of system organization.
- It provides system, subsystems and their relationships.
- Architectural patterns are often documented as software design patterns.
- Commonly used architectural patterns are:

  - Layered architecture
  - Client-Server architecture
  - Repository architecture (Shared data)
  - Pipe and filter architecture

2.8 Architectural design decisions

➥ Layered architecture:

- This type of pattern describes separation and independence.
- This architecture uses many layers for allocating the different responsibilities of a software product.
- Each layer works independently and each layer can use the services offered by the layer under it.
- A well suited example for layered architecture is OSI Layer.

❖ Advantages:

- Increases flexibility, maintainability, and scalability.
- Changes in one layer do not affect another.
- Authentication can be provided in each layer.
- Develop loosely coupled systems.
- Helps you to test the components independently of each other.
- It is possible to configure different level of security.

2.8 Architectural design decisions

❖ **Disadvantages:**

- Sometime extra overheads while passing data through layers.
- Sometimes takes long development time.
- More number of layers add complexities.
- Clean separation of each layer is difficult.
- Performance should be degraded due to multiple layers.

❖ **Application:**

- When there is a need of multilevel security.
- Used when building new facilities on top of existing systems.

2.8 Architectural design decisions

➥ Client-Server architecture:

- It is one of the basic paradigms of distributing computing system.
- Main two components: clients and servers.
- A constraint of this style is – a client can communicate with the server and can’t communicate with other clients.
- The communication between these components is initiated by the client when client sends a request for some services to the server and server responds them.
- The server receives the request at its predefined port, performs the service and then returns the results to the particular client.
- In it, request/reply type is working as a connector type.
- This connection is asymmetric.
- Figure:

2.8 Architectural design decisions

❖ Advantages:

- We can use the functionality of the server throughout the network.
- Provide centralized control.
- Data and file back up become easier.
- Provide greater accessibility.
- Provide scalability.
- Provide better security.
- Server can play different role for different clients.

❖ Disadvantages:

- Not robust because of single point failure.
- Performance may be unpredictable.
- Should have problem of overload and congestion.
- More expensive to install and manage.
- Skilled staffs are required for better maintenance.

❖ Application:

- applicable when data in a shared database has to be accessed from different locations

2.8 Architectural design decisions

➥ Repository architecture:

- It in, all the data in a system is managed in a central repository.
- Repository is accessible to all the components which do not interact directly, but only through repository.
- There are two types of components → data repositories and data assessors.
- Figure:
- Large amount of data sharing is possible.
- Different components do not need to communicate each other and not even need to know each others' presence.
- In this style of architecture, read/write data to the repository works as connectors.
- Example → MIS

2.8 Architectural design decisions

❖ **Advantages:**

- All the components are independent; they do not need to know each other.
- Changes made by one component can be propagated or circulated to all.
- Data can be managed consistently, as it is all in one place.

❖ **Disadvantages:**

- The repository is a single point of failure so problem in the repository may affect the whole system.
- Sometimes it may create inefficiency, because all communication made through repository.
- Distributing the repository across several computers may be difficult.

❖ **Application:** when large information storage required.

2.8 Architectural design decisions

➥ Pipe and filter architecture:

- It provides a structure for systems that process a stream of data.
- Each processing step is encapsulated in a filter component.
- Data are passed through pipes between adjacent filters.
- In this architecture, filters are working components and pipes are working as connectors.
- Filter has interfaces from which a set of inputs can flow in and a set of outputs can flow out.
- Filters are independent entities, and they don't know the identity of other filters.
- The pipes are the connectors between a data source and the first filter, between filters, and between the last filter and a data sink.
- Figure and process.

2.8 Architectural design decisions

❖ **Advantages:**

- It is easy to understand and implement.
- Maintenance is easy and provides reusability.
- Filters can work parallel in multi processing environment, so concurrent execution is also possible.
- This work flow style is used in many business processes.

❖ **Disadvantages:**

- As filters are independent entities, designer has to provide complete transformation of input and output to each filter.
- Sometimes this type of architecture may have overhead and latency problems.
- This type of architecture not really suitable for interactive systems.
- Error handling is difficult in this type of architecture.

❖ **Application:** Well suited for batch operating system.

2.8 Architectural design decisions

◈Application Architecture

- It forms the basis of an enterprise architecture.
- Software application architecture is the process of defining a structured solution that meets all of the technical and operational requirements.
- Application architecture is the organizational design of an entire software application, including all sub-components and external applications.
- Application architecture helps us to understand the operations of the system.
- It describes the layout of application's deployment.
- It can be used as a blueprint to ensure that the underlying modules of an application will support future growth of the system.

2.8 Architectural design decisions

- **Use of application architecture:**
- It can be used as a starting point for architectural design.
- It is used as a design check list.
- It is used as a way of organizing the work of the development team.
- Different application architectures:
  - *Data processing application*
  - *Transaction processing application*
  - *Event processing system*
  - *Language processing system*

2.8 Architectural design decisions

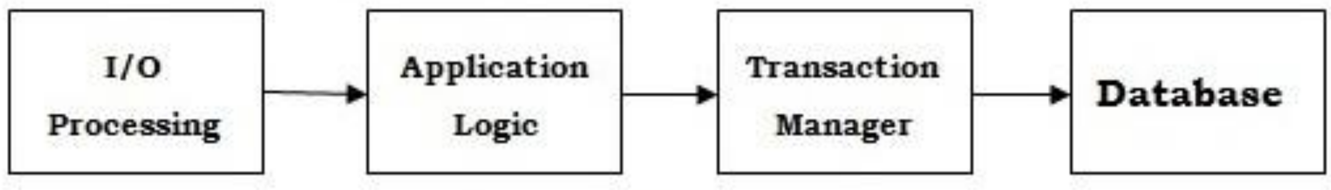
❖ Data processing application

- It is data driven application that processes data in batches without explicit user interference during the processing.
- In it, data is input and output in batches.
- For example, in electricity billing system.
- This type of application usually has an → input-processing-output structure.
- Figure:
- Process:
- Example → DFD.

2.8 Architectural design decisions

❖ Transaction Processing System (TPS)

- This is a data centered application.
- Users make asynchronous requests for service which are then processed by a transaction manager.



Transaction processing system

- Query processing takes place in the system database, and results are sent back to database through transaction manager.
- For example a reservation system.

2.8 Architectural design decisions

❖ **Event Processing System.**

- In which, system's actions are depend on events of system's environment.
- This system responds to events in the system environment.
- Due to unpredictable timing of events, architecture has to be organized to handle this.
- For example word processing system and real time systems.

❖ **Language processing system.**

- In which, accept a natural or artificial language as input and generate some other representation of that language.
- It includes the translator or interpreter to generate the output language.
- Best example for this system is compiler which translates high level programming language into lower level (machine code).

Difference between functional and non-functional requirements.

Functional requirements	Non functional requirements
- These describe what the system should do.	- These describe how the system works.
- These describe features, functionality and usage of the system.	- They describe various quality factors, attributes which affect the system's functionality.
- Describe the actions with which the work is concerned.	- Describe the experience of the user while doing the work.
- Characterized by verbs.	- Characterized by adjectives.
- Ex: business requirements, SRS etc.	- Ex: portability, quality, reliability, robustness, efficiency etc.

Thank You…