SE (Diploma 4th Computer Engineering)

Unit : 2

Software Requirement Analysis and Design

Prepared by:

MS. JEMI M. PAVAGADHI

Lecturer, Computer Engg. Dept.
Govt. Polytechnic PORBANDAR

CO (Course Outcome) & UOs (Unit Outcomes)

CO (b)

Prepare software analysis and design using SRS, DFD and object oriented UML diagrams.

UOs

2a. Identify Software requirements for the given problem
2b. Prepare SRS from the requirement analysis
2c. Represent the specified problem in the given design notation – DFD
2d. Draw the relevant UML diagrams for the given problem

2.1 Requirement Gathering & Analysis

- As per IEEE the definition of requirement is → *A condition or capability needed by a user to solve a problem or achieve an objective.*
- Requirements are the description of features and functionalities of the target system.
- Requirements of customer play an important role.
- Requirements convey the expectation of users.
- Done by system analysis.
- **Requirement Gathering and Analysis** is → "Collecting all the information from the customer and then analyze the collected information to remove all ambiguities and inconsistencies from customer perception."

2.1 Requirement Gathering & Analysis

- The most important phase of the SDLC is the requirement gathering and analysis phase because this is when the project team begins to understand what the customer wants from the project.
- During the requirements gathering sessions, the project team meets with the customer to outline each requirement in detail.

2.1 Requirement Gathering & Analysis

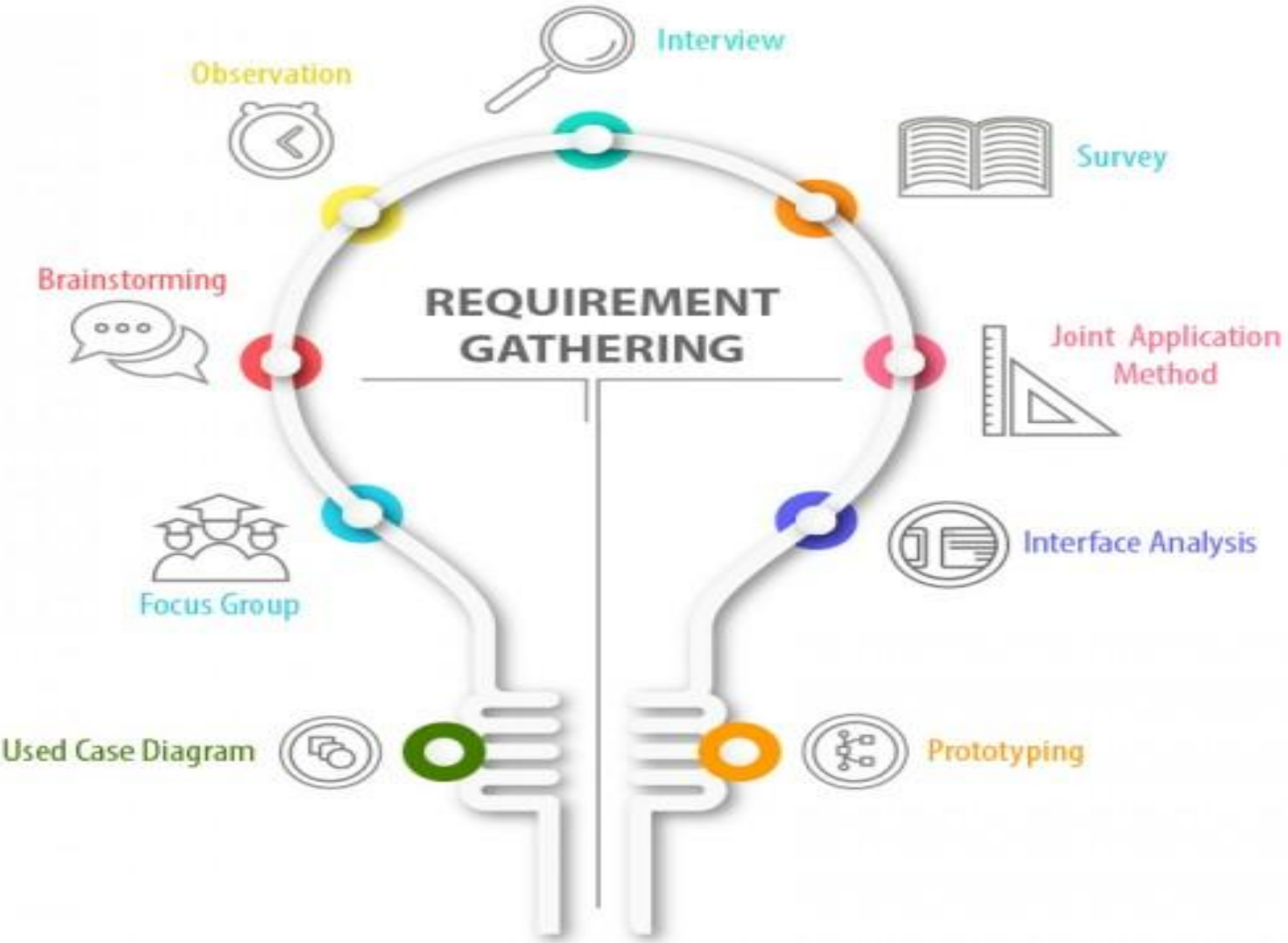
- It is the discovery phase of software development.
- Two sub processes → Requirement gathering and Requirement analysis.

❖ Requirement Gathering:

- Collecting requirements from the stakeholders.
- First part of any software product development.
- **Requirements gathering** is the process of understanding what you are trying to build and why you are building it.
- **Goal** → to collect all related 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 & Analysis

- Activity of market research (for competitive analysis) is done at this time.
- It involves interviewing the end-users and studying the existing documents to collect all possible information.
- Different requirement gathering activities:
  - Interview with end users or customers
  - Survey / Questionnaire
  - Brainstorming
  - Observation
  - Interface analysis
  - Task analysis
  - Form analysis
  - Group discussion
  - Prototyping
  - Analyse existing documents


REQUIREMENT GATHERING
Interview
Observation
Survey
Brainstorming
Joint Application Method
Interface Analysis
Focus Group
Used Case Diagram
Prototyping

2.1 Requirement Gathering & Analysis

❖ Requirement Analysis:

- Requirement analysis is the process of evaluating, interpreting, and refining the requirements gathered during the requirement gathering process to ensure they are clear, complete, accurate, and feasible.
- Goal → to clearly understand the exact requirements of the customers.
- *IEEE defines requirement analysis as (1) the process of studying user needs and (2) The process of studying and refining system hardware or software requirements.*
- **Software requirement analysis** simply means complete study, analyzing, describing software requirements so that requirements that are genuine and needed can be fulfilled to solve problem.

2.1 Requirement Gathering & Analysis

- **Requirement analysis involves:**
  - ***Eliciting requirements*** (requirements ને ઉઘેાડવી)
  - ***Analyzing requirements*** (ઉઘડેલી requirements નુ analysis કરવુ)
  - ***Requirements recording or storing*** (બધીજ જરૂરી requirements ને record કરવી)
- Performed by system analyst.
- For that, analyst has to identify and eliminate the problemsof anomalies, inconsistencies and incompleteness. **Anomaly** is the ambiguity in the requirement, **Inconsistency** contradicts the requirements, and **Incompleteness** may overlook some requirements.

Analyzing Gathered Requirements

It’s an essential step in requirement analysis due to the following reasons:

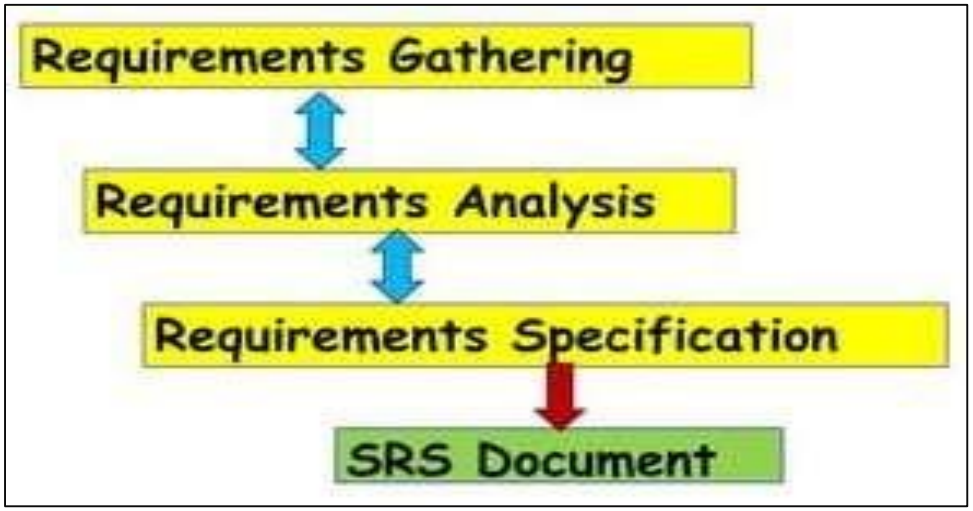
(1) Customer may provide some requirements that could be impossible to implement due to various dependencies.

(2) A business analyst may understand the customer requirement differently than a programmer interprets it.

2.1 Requirement Gathering & Analysis

- Discussion with end users is performed.
- Finally, make sure that requirements should be specific, measurable, timely, achievable and realistic.
- Output of this activity is → Software Requirement Specification (SRS).

 Requirement gathering and requirement analysis & specification activities collectively called 'Requirement Engineering'.



2.2 Software Requirement Specification (SRS)

- It is the output of requirement gathering and analysis activity.
- Created by system analyst.
- It is a detailed description of the software that is to be developed.
- It describes 'what' the proposed system should do without describing 'how' the software will do (*what part, not how*).
- Working as a reference document.
- SRS is actually a contract between developer and end user.
- The SRS translates the ideas of the customers (input) into the formal documents (output).
- The SRS document is known as black-box specification.
- SRS is a formal report, which acts as a representation of software that enables the **customers to review whether it (SRS) is according to their requirements.** Also, it comprises user requirements for a system as well as **detailed specifications of the system requirements**.

2.2 Software Requirement Specification (SRS)

❖ Important of SRS (Features 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.
- Reduces the development cost and time efforts.
- We can get the partial satisfaction of the end user for the final product.
- SRS is also useful during the maintenance phase.

2.2 Software Requirement Specification (SRS)

❖ Users of SRS

- ✓ *Customers and end users*
- ✓ *Project managers*
- ✓ *Designers and programmers*
- ✓ *Testers*
- ✓ *Maintenance teams*
- ✓ *Trainers*

❖ Characteristics of a good SRS

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

2.2 Software Requirement Specification (SRS)

❖ Examples of bad SRS (Characteristics of bad SRS)

- Over specification (try to address 'how')
- Forward references (aspects that to be referred late)
- Wishful thinking (contents that are difficult to implement)
- The SRS documents that contain incompleteness, ambiguity and contradictions are considered as bad SRS documents.

❖ Types of requirements in SRS

- Customer requirements
- Functional requirements
- Non-functional requirements

2.2 Software Requirement Specification (SRS)

❑ Customer requirements

- It represent the needs, expectations, and desires of the end-users, customers or stakeholders who will be using the software product.
- Motivate customers to buy a product or service.
- 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 that the software meets the customer's expectations.
- Requirements should be prioritized to ensure that the most critical requirements are addressed first.

2.2 Software Requirement Specification (SRS)

☐How to identify customer's requirements

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

- All the customer requirements should be written in SRS with clear and concise approach.

2.2 Software Requirement Specification (SRS)

❑ Functional requirements

- It describes the functionalities or services that the system is expected to provide.
- It forms the core of requirement documents.
- It also describe how the system should behave in different scenarios.
- **Key goal** → to capture the behaviour of software in terms of functions and technology.
- It is the list of actual services which a system will provide.
- 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 supposed to do.

2.2 Software Requirement Specification (SRS)

☐How to identify functional requirements

- From following factors:
  - ✓ From informal problem description or from conceptual understanding of the system
  - ✓ From user perspective
  - ✓ Find out higher level function requirements

☐How to document functional requirements

- Specified by different scenarios
- Specify the input data domain, processing and output data domain
- Document the functionalities supported by the system

2.2 Software Requirement Specification (SRS)

☐ Example: operations at ATM

Functional requirements of ATM system (likely)

- Withdraw cash
- Fast cash
- Deposit cash
- Check balance
- Print mini statement
- Change pin number
- Generate PIN
- Transfer money etc.

When you design SRS, all the functional requirements must start with a 'verb'.

(we can do many different tasks at ATM centre i.e withdraw cash, check balance, print mini statement, deposit cash etc. All of these tasks will be Functional Requirements of SRS and we note them as R1, R2,,. And we will have a step by step process for each requirement i.e. R1.1, R1.2,,. Likewise we have to write detailed step by step process for every functional requirement.)

2.2 Software Requirement Specification (SRS)

Sample SRS for one of the requirements of ATM system.

R1: Withdraw cash

- **R 1.1: enter the card**
  Input: ATM card
  Output: user prompted to enter PIN showing the display with various operations
- **R 1.2: enter PIN**
  Input: valid PIN
  Output: showing the display with various operations
- **R 1.3: select operation type**
  Input: select proper option (withdraw amount option)
  Output: user prompted to enter the account type
- **R 1.4: select account type**
  Input: user option (for example: saving account or current account)
  Output: user prompted to enter amount
- **R 1.5: get required amount**
  Input: amount to be withdrawn
  Output: requested cash and printed transaction

Customer management system

Functional requirements for a customer management system allow companies to interact with customers, store customer information and share information within a company's network.

For example, an insurance agency might use a customer management system where **insurance agents can interact with clients, update and change policy information and exchange customer information with other insurance agents at their company.**

Here are some functional requirements they might consider for the customer management system:

The system allows insurance agents to access it using a password and their employee identification number.

Insurance agents must receive management verification before making a change to customer information.

The software archives all deleted policy information.

The system tracks and records all customer interactions.

A software feature enables the system to perform routine security checks to verify the identity of insurance agents.

Insurance agents must receive management verification before making a change to customer information.

The software archives all deleted policy information.
The system tracks and records all customer interactions.

A software feature enables the system to perform routine security checks to verify the identity of insurance agents.

Video game software

Video game software includes features that allow users to defeat levels, win challenges and build up their problem-solving skills. For example, if a video game company wants to release a game that encourages users to figure out challenges and increases the amount of difficulty for each level, the functional requirements may include:

Users can operate a controller that allows them to control characters in the game.

Each level increases the difficulty of challenges.

Users must create a username and password to play the game. The game prompts users to verify their identity before starting each game.

The colors of the game change depending on the level.

2.2 Software Requirement Specification (SRS)

❑ Non Functional requirements

- Non-functional requirements are the constraints or restrictions on the design of the system.
- It is called quality attributes.
- It describes the characteristics of the system that can't be expressed functionally.
- Some non-functional requirements are:

- Portability
- Maintainability
- Reliability
- Usability
- Security
- Performance
- Security
- Manageability
- Data integrity
- Inter-operability
- Scalability
- Availability

Non Functional requirements

INTRODUCTION:

Non-functional requirements in software engineering refer to the characteristics of a software system that are not related to specific functionality or behavior.

They describe how the system should perform, rather than what it should do.

Examples of non-functional requirements include:

Performance: This includes requirements related to the speed, scalability, and responsiveness of the system. For example, a requirement that the system should be able to handle a certain number of concurrent users or process a certain amount of data within a specific time frame.

Security: This includes requirements related to the protection of the system and its data from unauthorized access, as well as the ability to detect and recover from security breaches.

Usability: This includes requirements related to the ease of use and understandability of the system for the end-users.

Reliability: This includes requirements related to the system's ability to function correctly and consistently under normal and abnormal conditions.
Maintainability: This includes requirements related to the ease of maintaining the system, including testing, debugging, and modifying the system.
Portability: This includes requirements related to the ability of the system to be easily transferred to different hardware or software environments.
Compliance: This includes requirements related to adherence to laws, regulations, industry standards, or company policies.
Scalability :assesses the highest workloads under which the system will still meet the performance requirements.
Example of scalability requirements:
The system must be scalable enough to support 1,000,000 visits at the same time while maintaining optimal performance.

2.2 Software Requirement Specification (SRS)

- *Functional requirements drive the application architecture of the system while non-functional requirements drive the technical architecture.*
- *In one statement: Functional requirement is "what the system should do." And Non-functional requirement is "how the system should behave while performing."*

2.2 Software Requirement Specification (SRS)

❖ **Difference between functional and non-functional requirements.**

Functional requirements	Non-functional requirements
- These describe **what the system should do.**	- These describe **how the system should behave.**
- 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.

2.4 Software Design

- It is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- It is affecting the quality of the software product.
- It starts after requirement gathering and analysis.

❖ Levels of software design:

➢ Architectural design

- System can be viewed in terms of sub-systems, modules and relationship between these modules.
- It defines the framework of the system.

➢ High level design:

- Problem is decomposed in set of modules.
- Represented using structure chart and different UML diagrams

2.4 Software Design

➢ Detailed design

- It deals with implementation part.
- In it, each module is examined carefully to design data structure and algorithms.

❖ Characteristics of good software design:

- **Correctness**	- **Usability**
- **Completeness**	- **Maintainability**
- **Modularity**	- **Efficiency**
- **Scalability**	- **Testability**
- **Robustness**	- **Readability**
- **Flexibility**	- **Performance**

2.4 Software Design

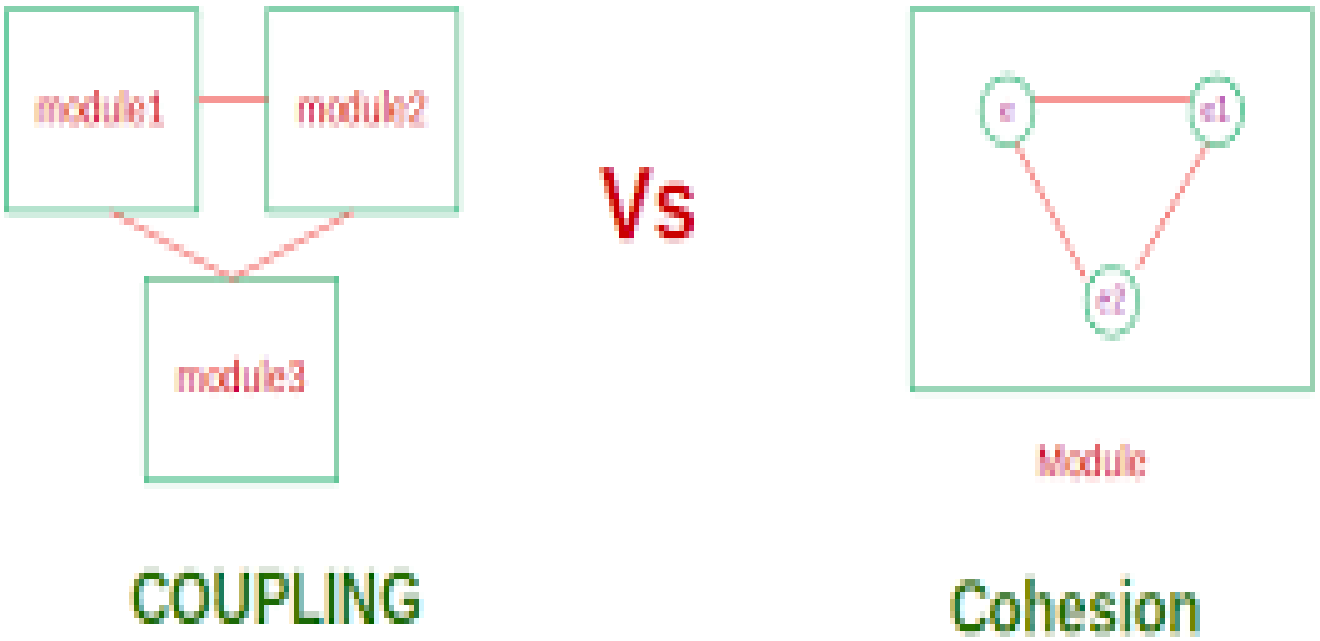
❖ Analysis vs Design

Analysis	Design
- It carried out **before design**.	- It carried out **after analysis.**
- It is a kind of examination of the problem.	- It is finding the solution of the problem.
- It concerned with '**what**' part of the software development.	- It concerned with '**how**' part of the software development.
- It deals with data collection.	- It deals with detailed design specification.
- **Output** is → Software Requirement Specification (SRS)	- **Output** is → Design documents with technical specifications.
- Analysis typically involves requirement gathering, use case analysis, requirement specification, domain analysis etc.	- Designing involves architecture design, user interface design, detailed design, component design etc.

2.5 Cohesion and Coupling

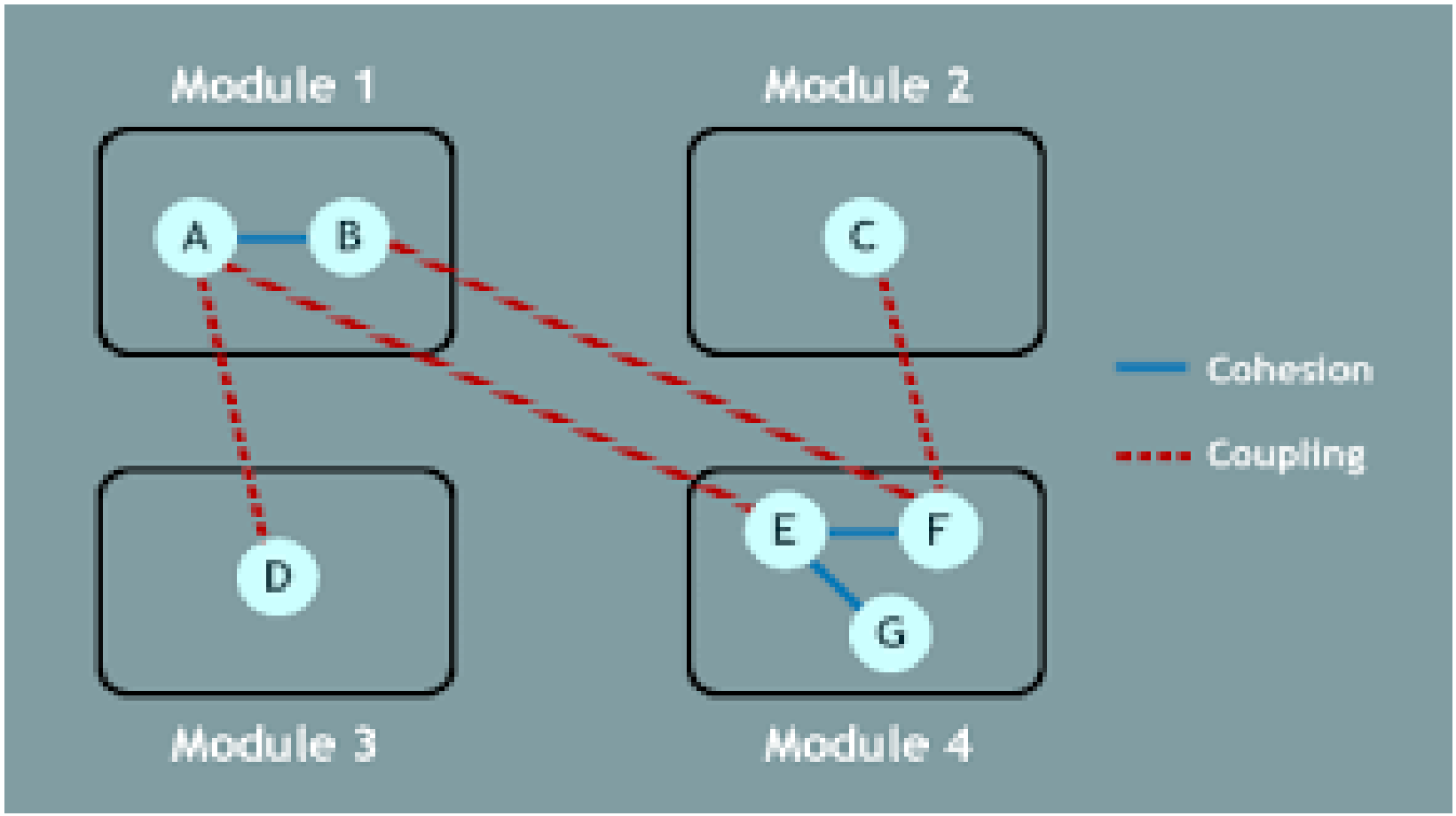
- Concept of modularity
- Software modularity is measured by how well software is decomposed into smaller pieces with standardized interfaces.
- Due to modularity:
  - Easy to understand the system.
  - System maintenance is easy.
  - Provide reusability.
- Cohesion and coupling are two modularization criteria that are often used together.
- '***high cohesion and low coupling***' is a primary need of any software development.

cohesion and coupling

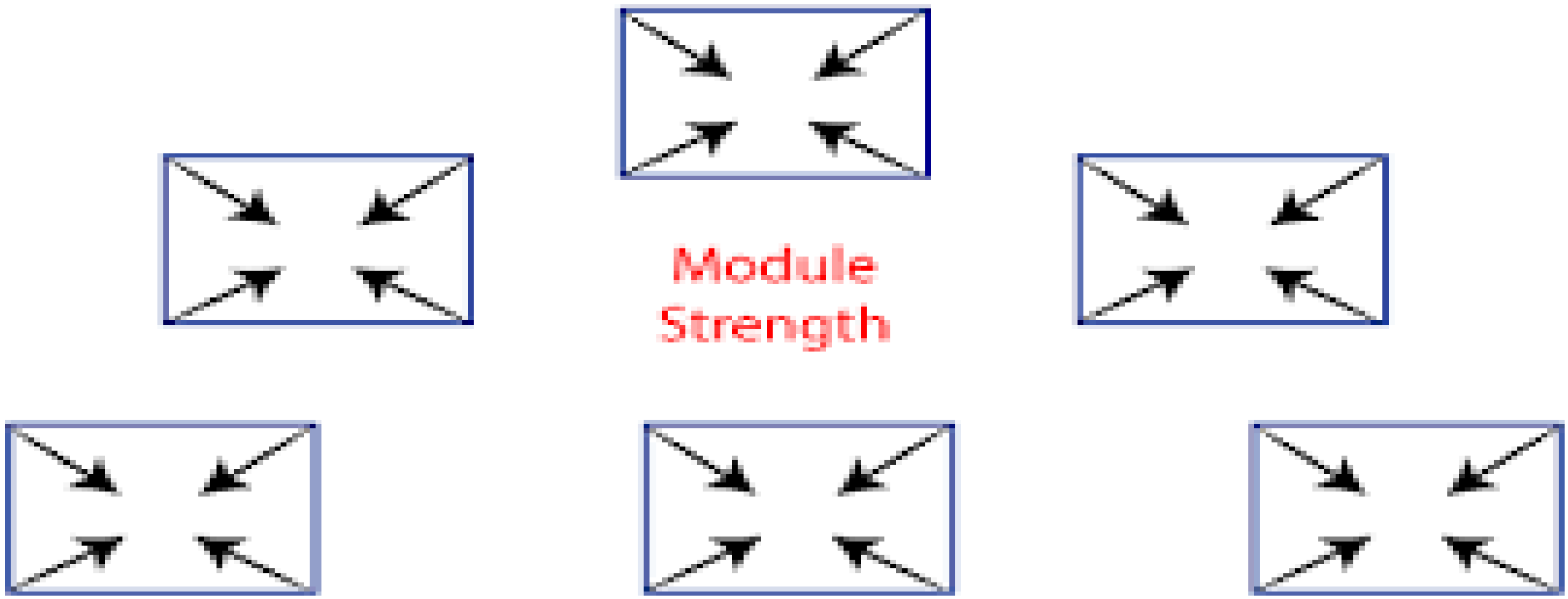


- ***high cohesion and low coupling*** is a primary need of any software development.

cohesion and coupling

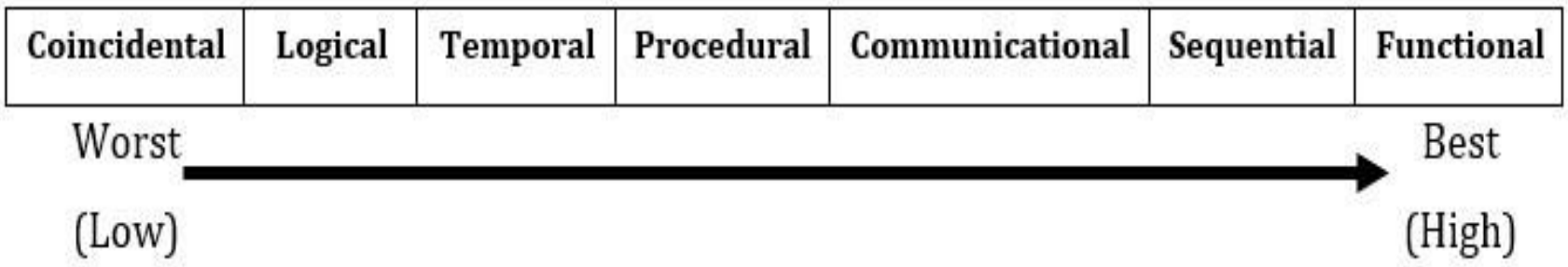


cohesion



Cohesion= Strength of relations within Modules

Cohesion

- **Cohesion** is → a measure of functional strength of a module.
- It refers to what a module can do internally. So it is called inter-module binding.
- Cohesion is the indication of the relationship within module. It is concept of intra-module.
- Cohesion has many types but usually highly cohesion is good for software.
- Cohesion keeps the internal modules together, and represents the functional strength.
- It represents how tightly bound the internal elements of a module
- 

ty,

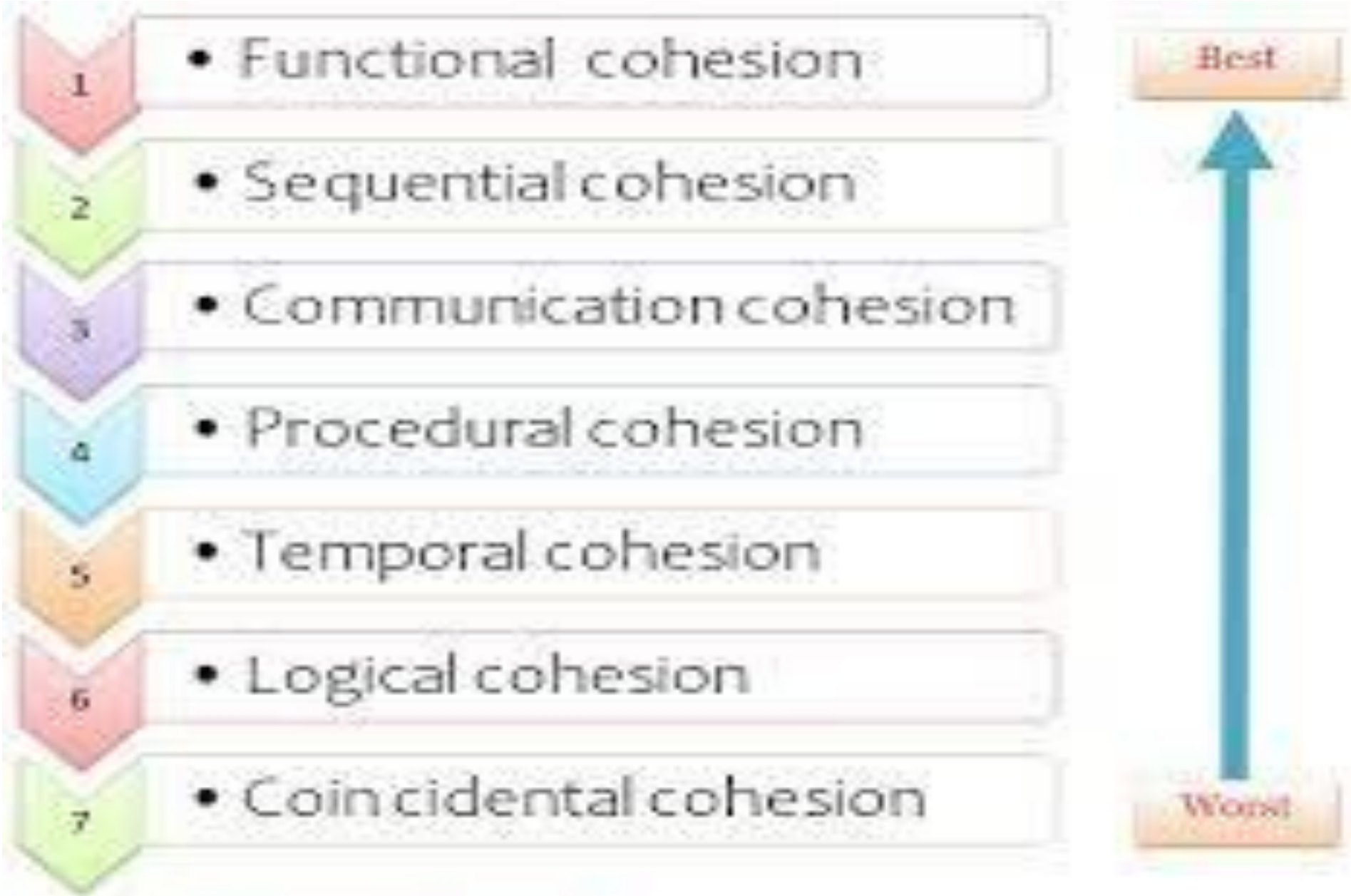


Figure 7: Types of Module cohesion

Cohesion

≫ Coincidental cohesion

- Lowest cohesion.
- Coincidental cohesion occurs when there are no meaningful relationships between the elements.
- it performs a set of tasks in a module that relate to each other very loosely or not relate to each other.
- The Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.
- Coincidental cohesion is can be avoid as far as possible.
- It is also called random or unplanned cohesion.
- **For example**, suppose we are working on a module that calculates the area and perimeter of a shape in a geometry application, the module performing tasks of 1. Retrieving the color of the shape, 2. Check the shape (concave or convex) and print message.
- Here both the tasks are unrelated to each other, and do not share any common information or data structure.

Cohesion

Logical cohesion

- A module is said to be logically cohesive if there is some logical relationships between the elements of a module, and the elements perform functions that fall into same logical class.
- For example: the tasks of error handling, input and output of data.
- The activities of the same type or same general category is contributed by the elements in the module

Temporal cohesion

- In it, elements are related in time and they 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, clean-up, and start-up, shut down are usually having temporal cohesion.

Cohesion

≫ Procedural cohesion

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

- For example, we are working on a module that calculates the total cost of a customer's order in an e-commerce application. The module performs the tasks of collect customer order details, calculate the cost of each item, calculate the total cost of the order and apply applicable discount.

- In this case, the tasks performed by the module are all related to the same procedure.

Cohesion

≫ Communicational cohesion

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, for example the set of functions defined on an array or a stack.
- Also, two elements operate on the same input data or contribute towards the same output data.
- These modules may perform more than one functions together.
- For example, user authentication module in web application, doing the tasks of verifying user's credential, retrieving user information and logging user activity. All the tasks are related to user authentication and focus on same data structure and same output data.

Cohesion

≫ Sequential cohesion

- When the output of one element in a module forms the input to another, we get sequential cohesion.
- For example, suppose you are working on a module that reads a file, processes the data, and then writes the results to another file. The module performs three tasks in sequence: reading, performing and writing. These tasks are arranged in a specific sequence with the output of one task serving as input to another task. This module is said to be in sequential cohesion.

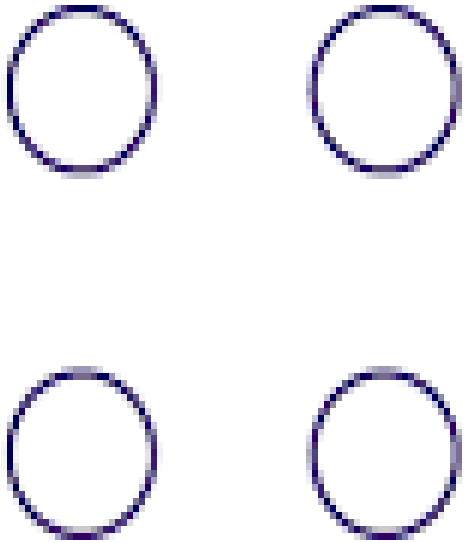
≫ Functional cohesion

- Functional cohesion is the strongest cohesion. And it is very good for any software development.
- 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.
- Functions like: reading transaction records, compute square root, sort the array are examples of these modules.

Coupling

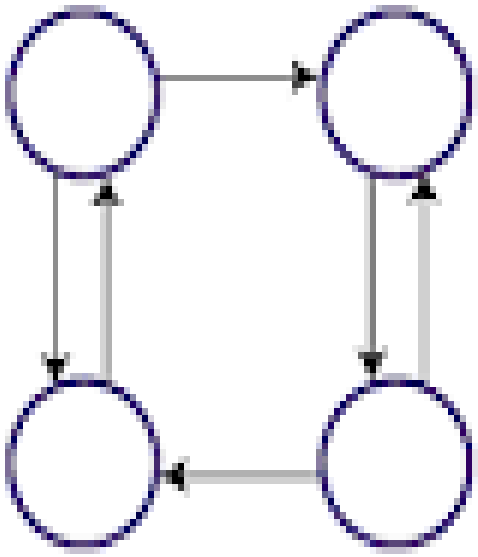
- **Coupling** between two modules is → a measure of the degree of interdependence or interaction between these two modules.
- Coupling refers to the number of connections between 'calling' and a 'called' module. There must be at least one connection between them.
- It refers to the strengths of relationship between modules in a system.
- As modules become more interdependent, the coupling increases. And loose coupling minimize interdependency that is better for any system development.
- If two modules interchange large amount of data, then they are highly interdependent or we can say they are highly coupled.
- High coupling between modules make the system difficult to understand and increase the development efforts. So low (OR loose) coupling is the best.

Module Coupling



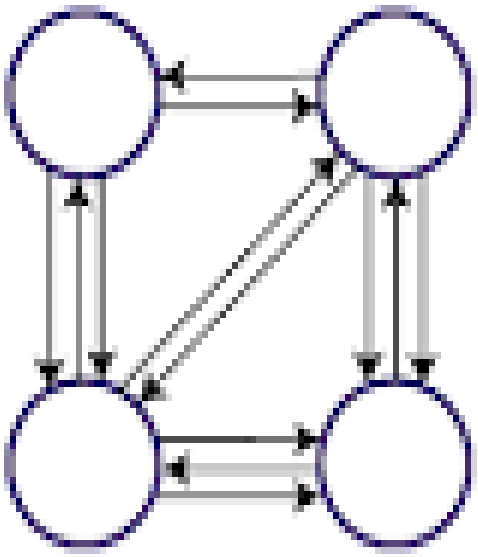
Uncoupled: no dependencies

(a)



Loosely Coupled: Some dependencies

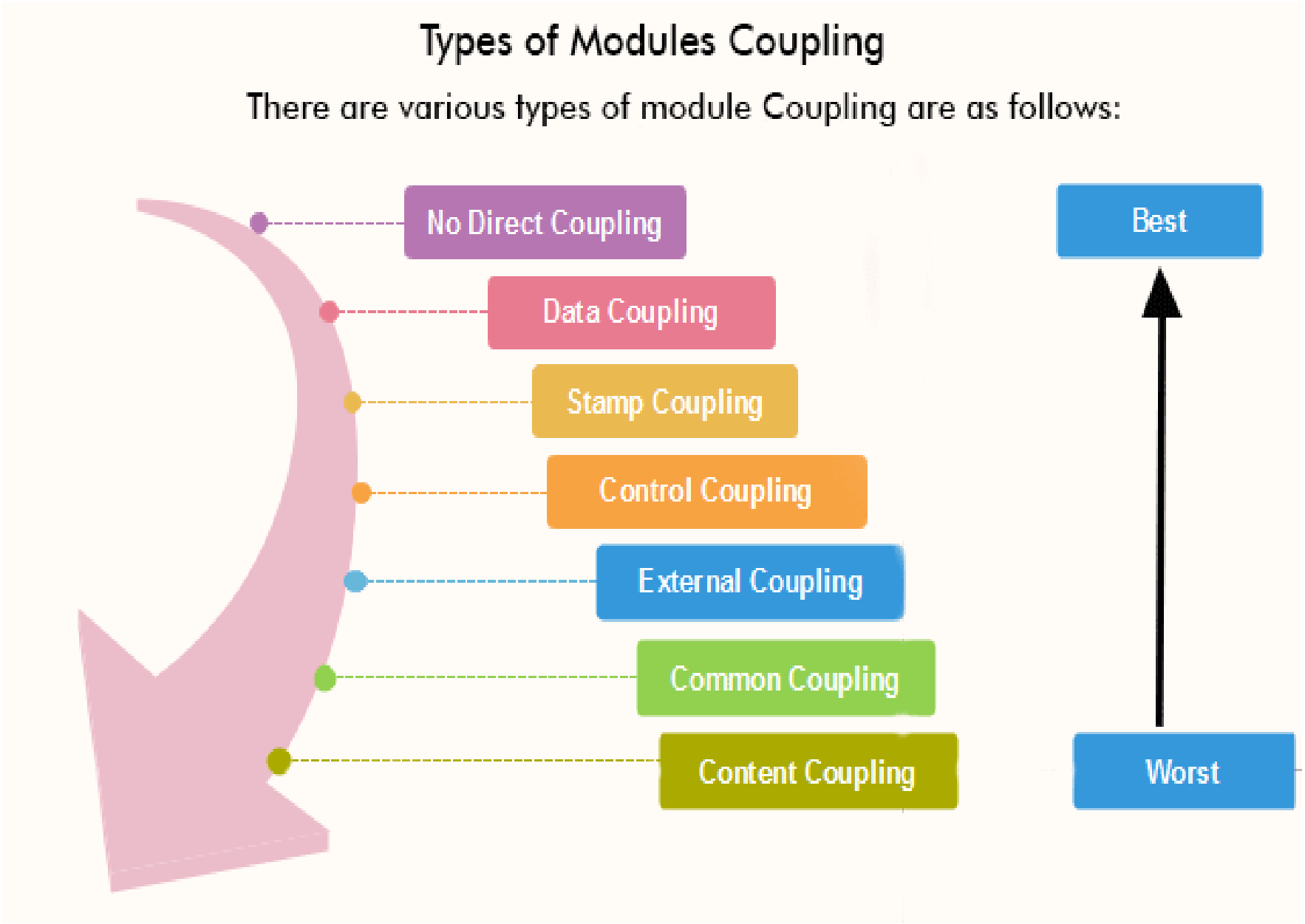
(b)



Highly Coupled: Many dependencies

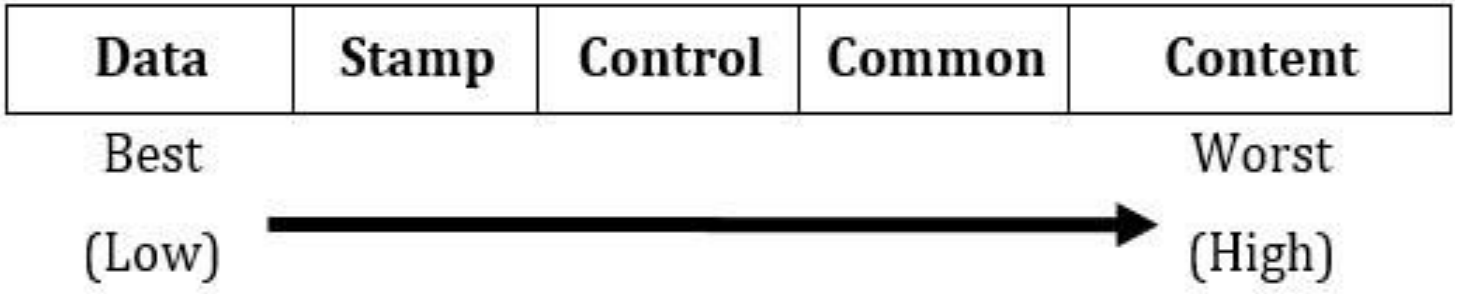
(c)

Classification of coupling



Coupling

❖ Classification of coupling



≫Data coupling

- It is lowest coupling and best for the software development.
- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between these two.
- One module passes data to another module and the second module uses that data to perform its function.
- For example → an int, a char, a float etc.

Coupling

≫Stamp coupling

- Two modules are stamp coupled when modules are connected based on a common data structure.
- Stamp coupling enables the programmer to pass an entire data structure from one module to another.
- For example, a database application where different modules interact with the same data tables.

≫Control coupling

- Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another module.
- In other term, one module controls the flow of execution of another module by passing it control information.
- For example, in a file secure system, security module controls the execution of file accessing module by passing the control information based on the user's authentication.

Coupling

≫ Common coupling

- Two modules are common coupled, if they share data through some global data items. It means two or more modules are communicating using common global data.
- Common coupling can be problematic in software engineering because it can increase the complexity of the code and make it more difficult to maintain and test.

- It can also make the software more prone to errors and bugs.

- For example, if the user interface module stores the user input in a shared data structure and the data processing module reads from the same data structure to perform calculations, then the two modules have common coupling.

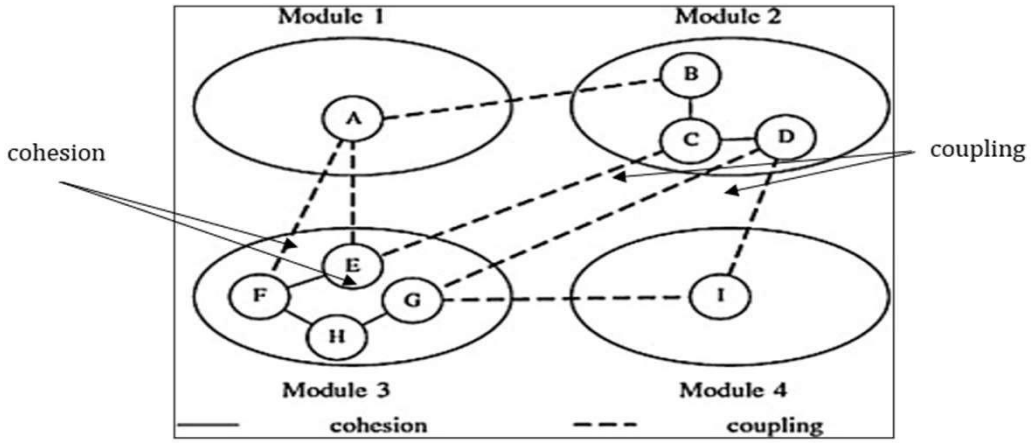
Coupling

≫ Content coupling

- It is the highest coupling and creates more problems in software development.
- It is worst coupling for any software development.
- This creates strong dependency between modules. And increase the complexity of the modules.

- Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

- It is also known as 'pathological coupling'.



Cohesion and Coupling

Difference between Cohesion and Coupling

Cohesion	Coupling
- Cohesion is the indication of the relationship within module.	- Coupling is the indication of the relationships between modules.
- Cohesion shows the module's relative functional strength.	- Coupling shows the relative interdependence among the modules.
- Cohesion is a degree (quality) to which a component / module focuses on the single thing.	- Coupling is a degree to which a component / module is connected to the other modules.
- While designing you should go for high cohesion. i.e. a cohesive component/ module focus on a single task with little interaction with other modules of the system.	- While designing you should go for low coupling i.e. dependency between modules should be less.
- Cohesion is the kind of natural extension of data hiding for example, class having all members visible with a package having default visibility.	- Making private fields, private methods and non-public classes provides loose coupling.
- Cohesion is Intra-Module Concept.	- Coupling is Inter -Module Concept

Function Oriented Software Design

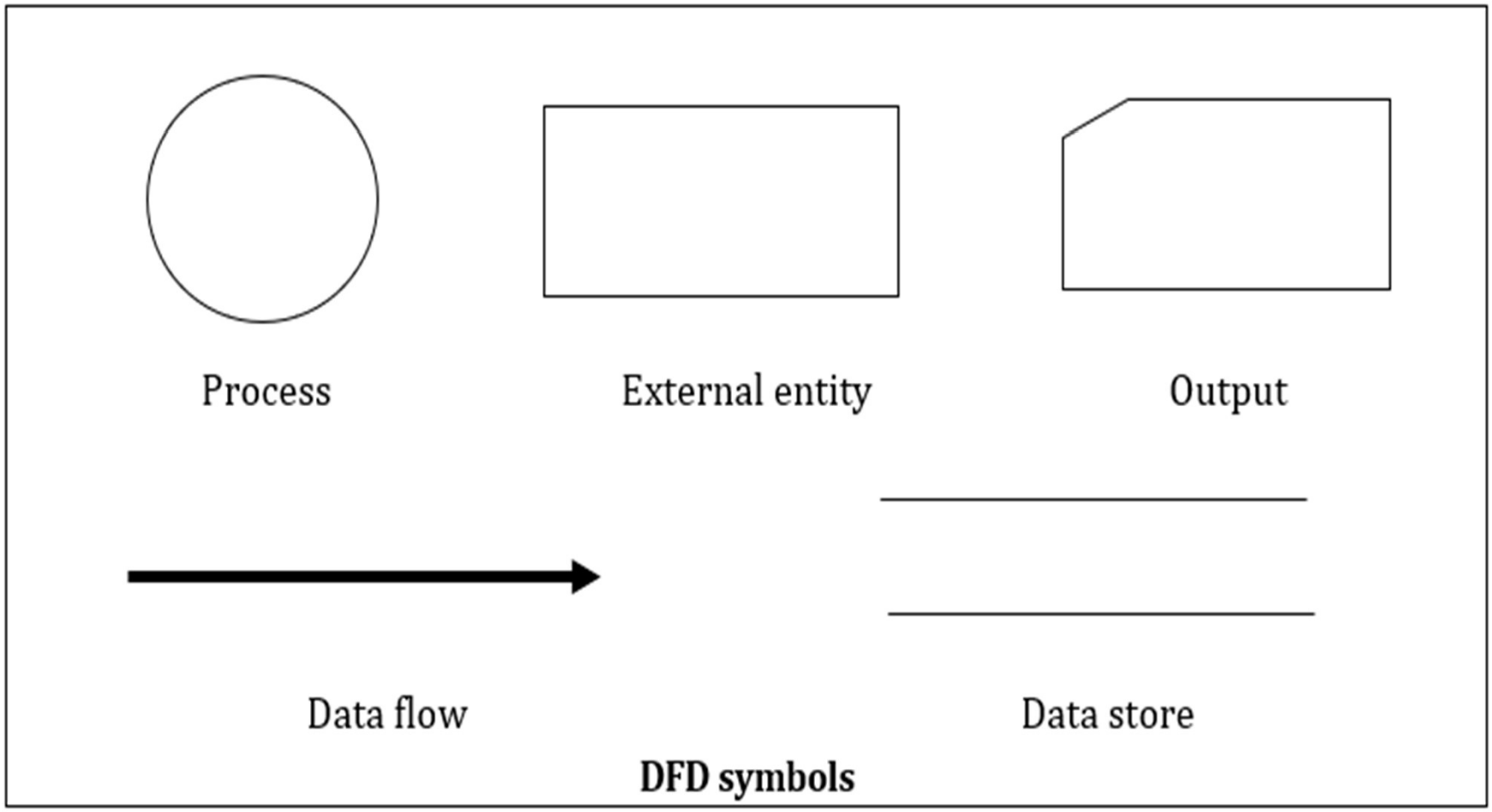
- That focuses on decomposing software systems into smaller, more manageable functions. So that the testing, development and maintenance will be easier.
- It is carried out in top down approach
- Function-oriented design is commonly used in procedural programming languages.
- Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

2.6 Data Flow Diagram (DFD)

- Also known as bubble chart or data flow graph.
- DFDs are very useful in understanding the system and can be effectively used during analysis.
- DFD is a graphical representation of how data moves through a system.
- The **purpose** of a DFD is to provide a visual representation of the system's data flow that is easily understood by both technical and non-technical stakeholders.
- It views a system as a function that transforms the inputs into desired outputs.
- 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 Diagram (DFD)

- Primitive symbols used in construction of DFD



2.6 Data Flow Diagram (DFD)

➢ Process (function)

- Represented by circle or bubble.
- Circles are annotated with names of the corresponding functions.
- Transforms inputs into outputs.
- The process is named using a single word that describes what the system does functionally, generally using 'verb'.

➢ External entity

- It 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.

2.6 Data Flow Diagram (DFD)

➢ Data flow

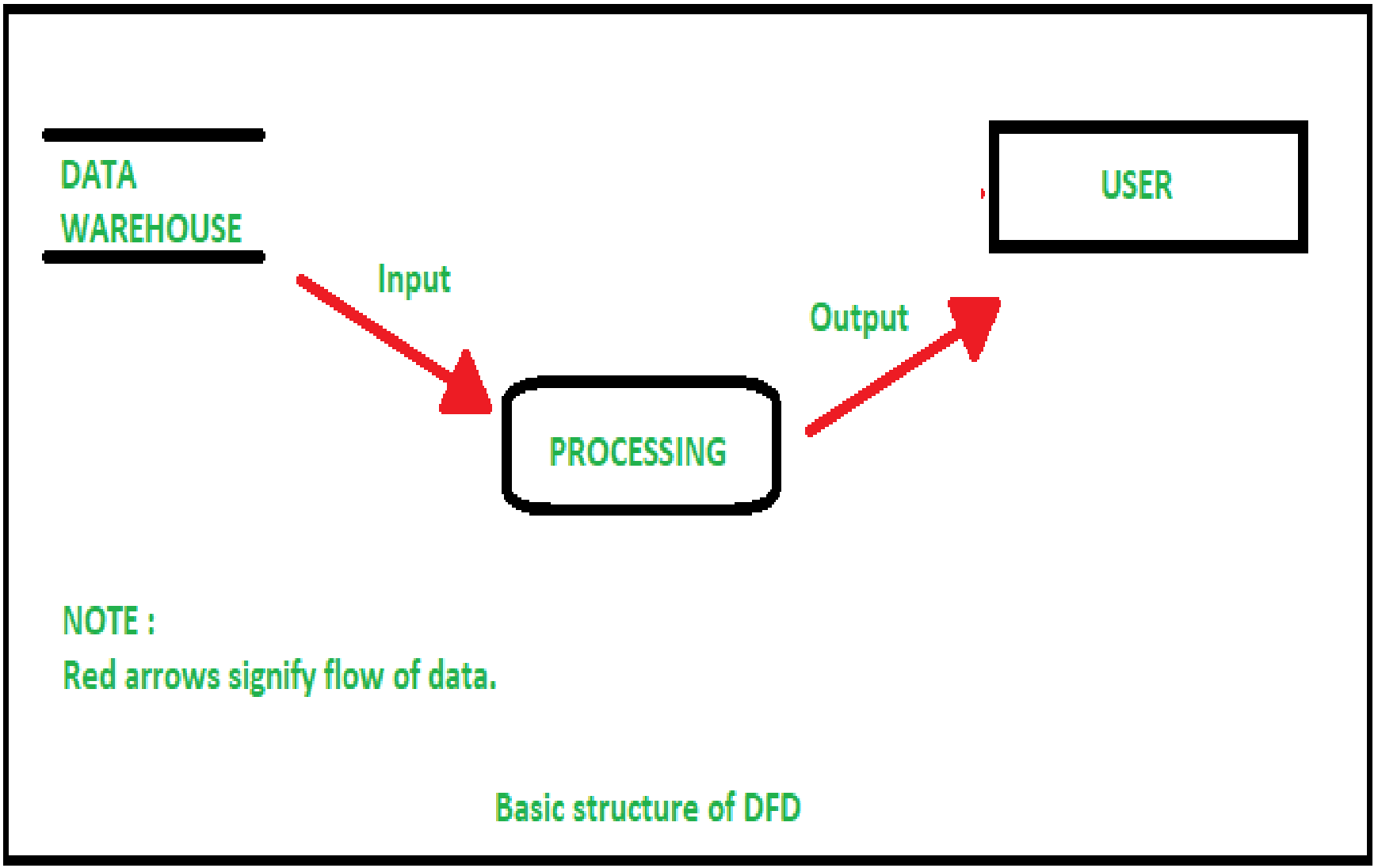
- Represented by an arc or by an arrow.
- It used to describe the movement of the data.
- It represents the data flow occurring between two processes, or between an external entity and a process. It passes data from one part of the system to another part.
- Generally, name of the corresponding data is in '*noun*' form.

➢ Data store

- Represented by two parallel lines.
- It is generally a logical file or database where the data is stored.

➢ Output

- Used when a hardcopy is produced.
- Represented by a rectangle cut either a side.



Basic structure of DFD

2.6 Data Flow Diagram (DFD)

❖ Developing DFD

- 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 (0 Level DFD)

- It is top level DFD.
- Most abstract view of the system.
- It is representing simplified view of targeted system.
- Entire system is represented as a single bubble (process).

2.6 Data Flow Diagram (DFD)

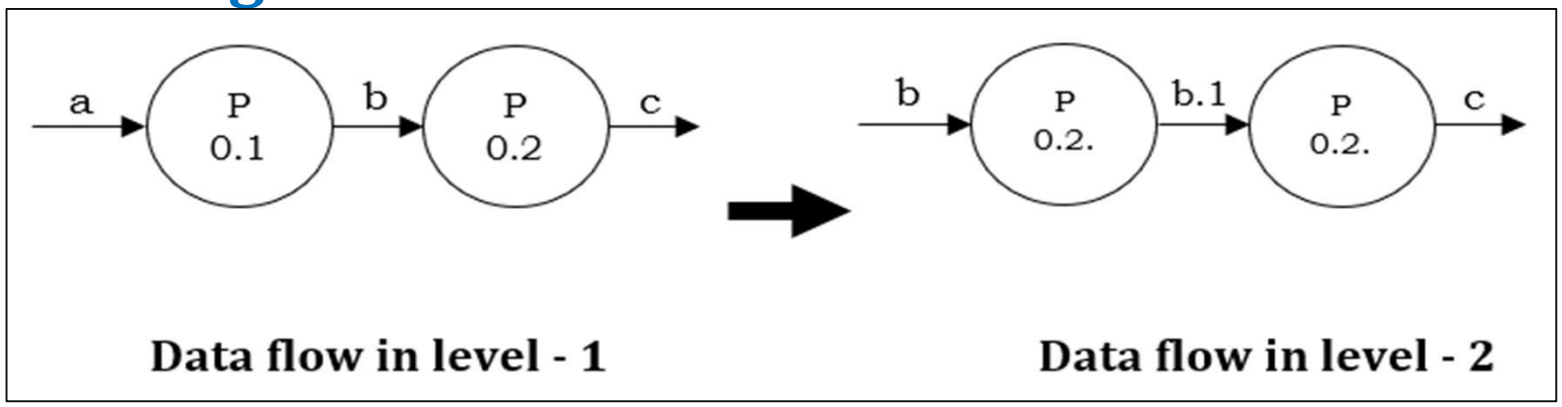
❖ Level 1 DFD

- It is more detailed version of context diagram.
- It is representing higher level functional requirements.
- Generally 3 to 7 processes are shown in level 1 DFD.

❖ Further decomposition

- It is also known as factoring or exploding a bubble.
- Successive versions of DFD give more detailed description of the system.

❖ Balancing DFD



2.6 Data Flow Diagram (DFD)

❖ Guidelines while drawing DFD

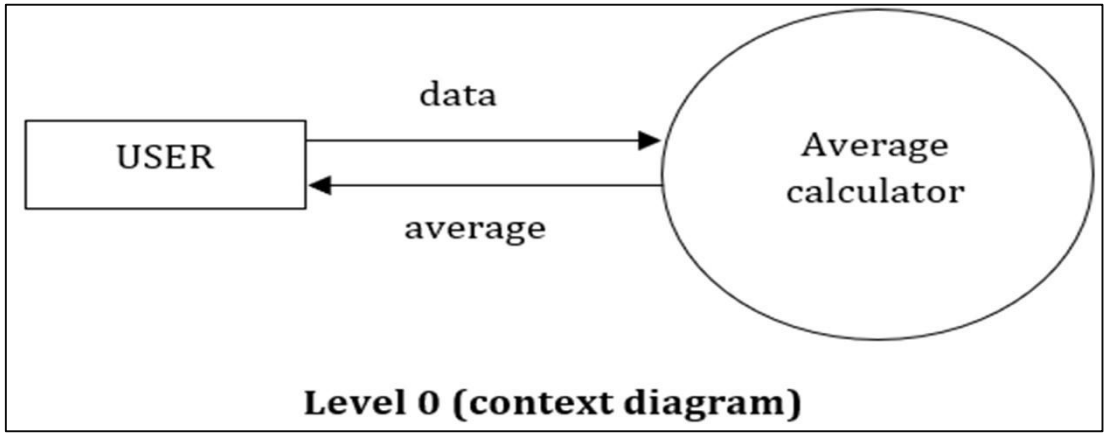
- A process must have at least one input and one output data flow.
- No control information (IF-THEN-ELSE).
- Data flows must be named.
- No detailed description in context level.
- No more than one bubble in context level.
- Name of data flow → *Noun* and process → *Verb*
- A data store must always be connected with a process. A data store cannot be connected to another data store or to an external entity.
- All the functionalities of the system specified in SRS must be captured by the DFD model.
- Data flows from entities must flow into processes, and data flows to entities must come from processes.

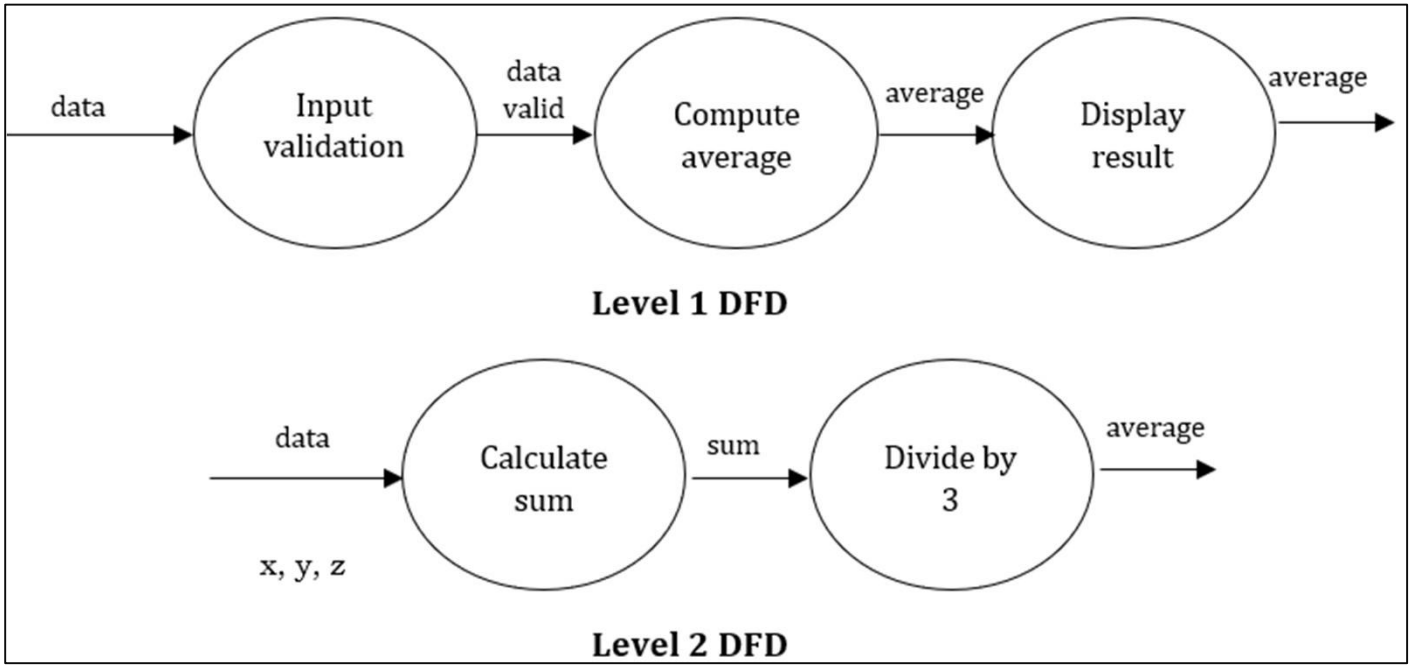
- **Levels of DFD**
- DFD uses hierarchy to maintain transparency thus multilevel DFD's can be created. Levels of DFD are as follows:

- **0-level DFD**: It represents the entire system as a single bubble and provides an overall picture of the system.
- **1-level DFD**: It represents the main functions of the system and how they interact with each other.
- **2-level DFD**: It represents the processes within each function of the system and how they interact with each other.
- **3-level DFD:** It represents the data flow within each process and how the data is transformed and stored.

2.6 Data Flow Diagram (DFD)

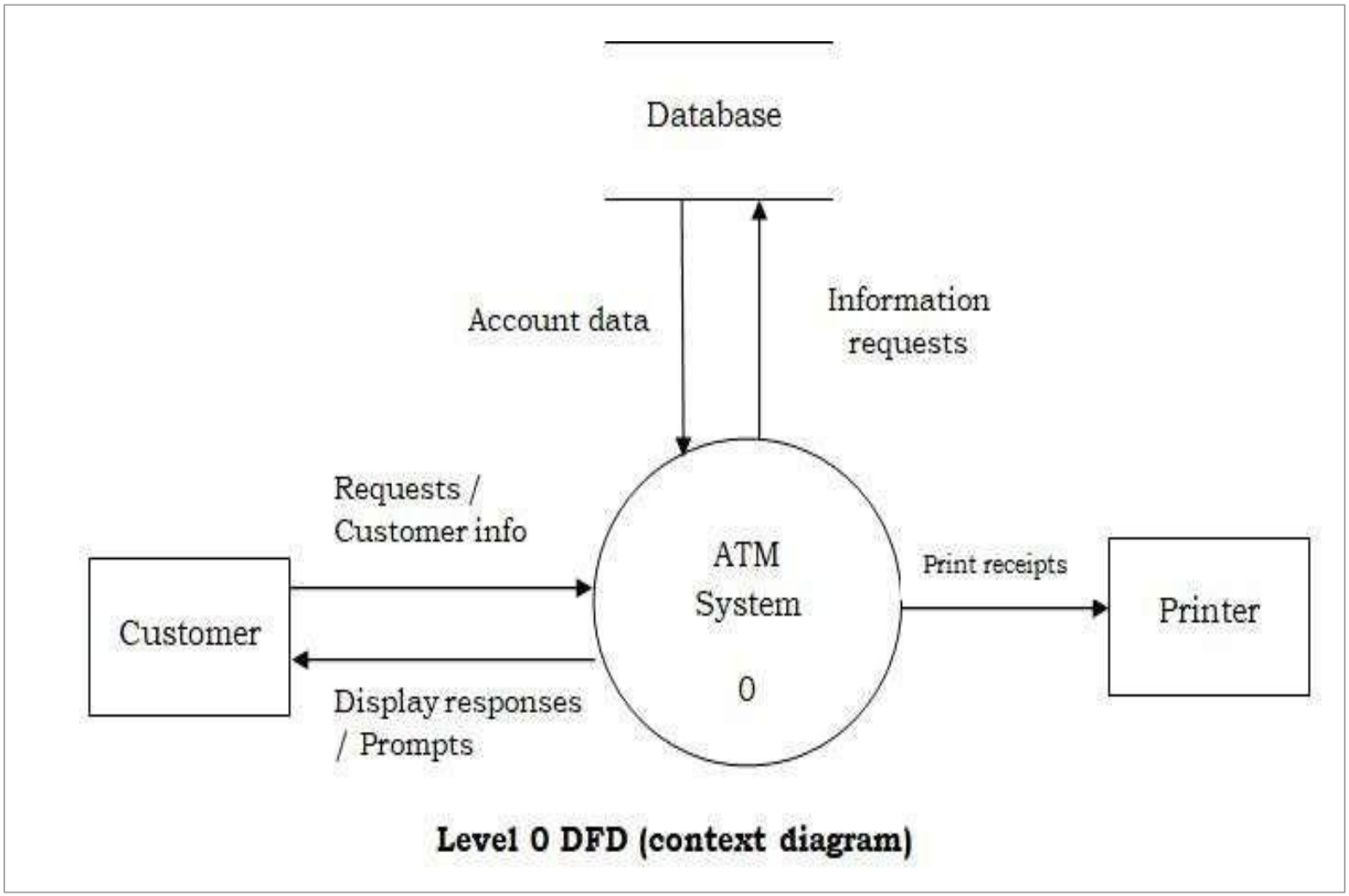
❖ Example (Average calculator)





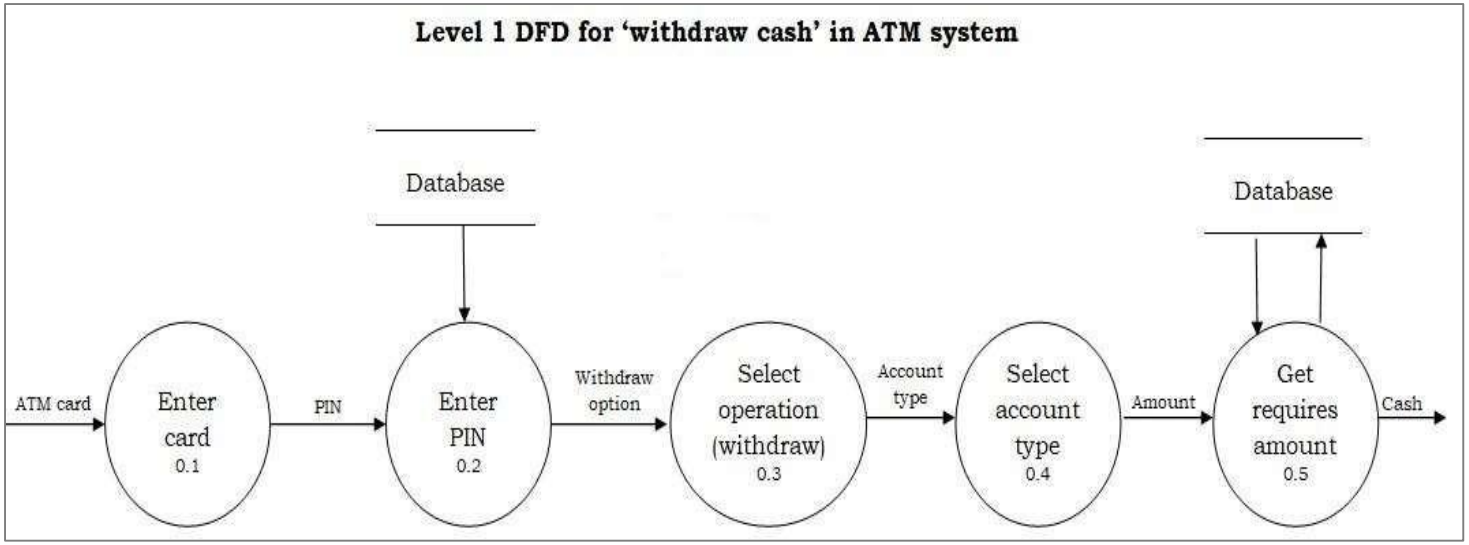
2.6 Data Flow Diagram (DFD)

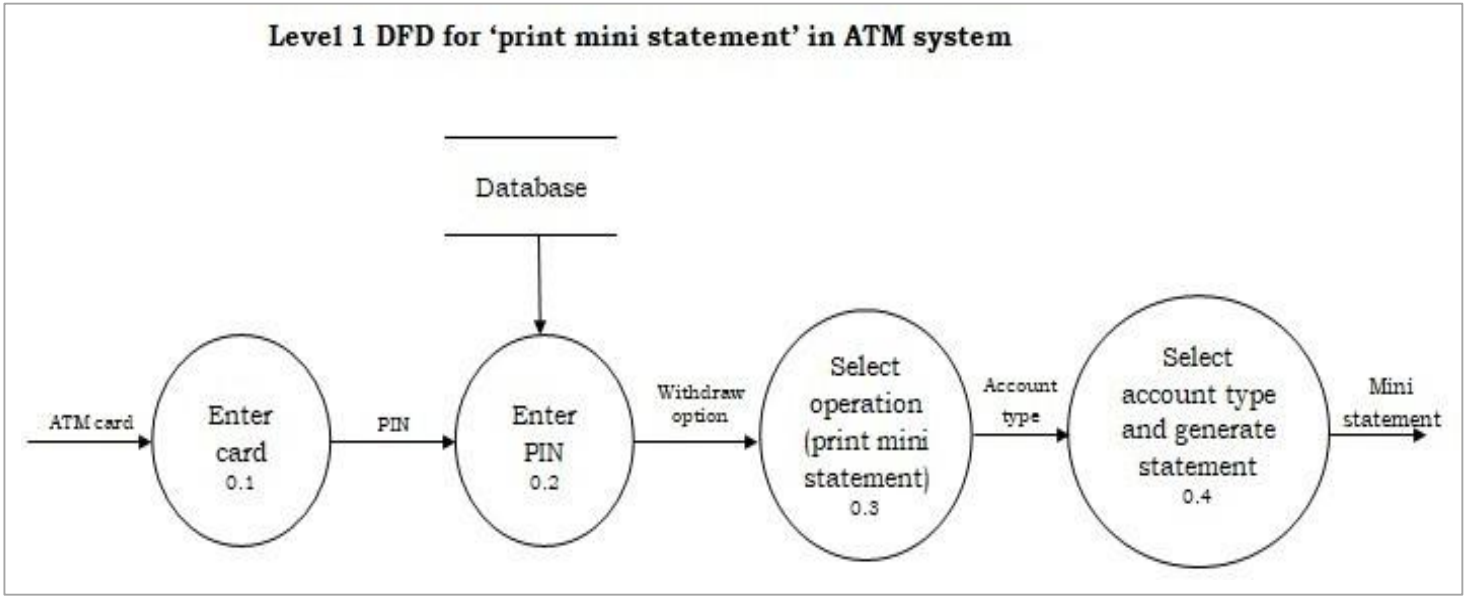
❖ **Another example (ATM system)**



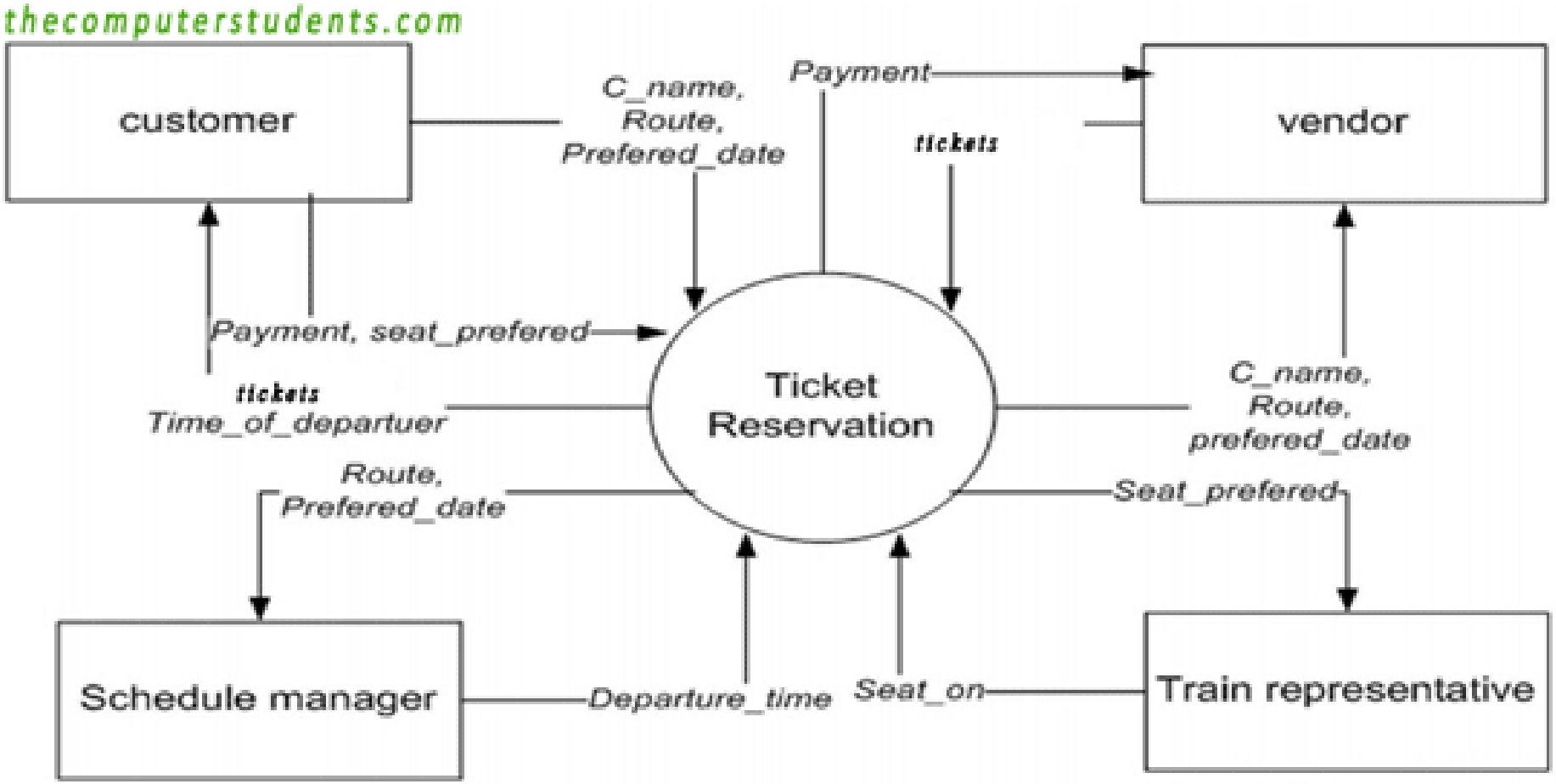
Level 0 DFD (context diagram)

2.6 Data Flow Diagram (DFD)

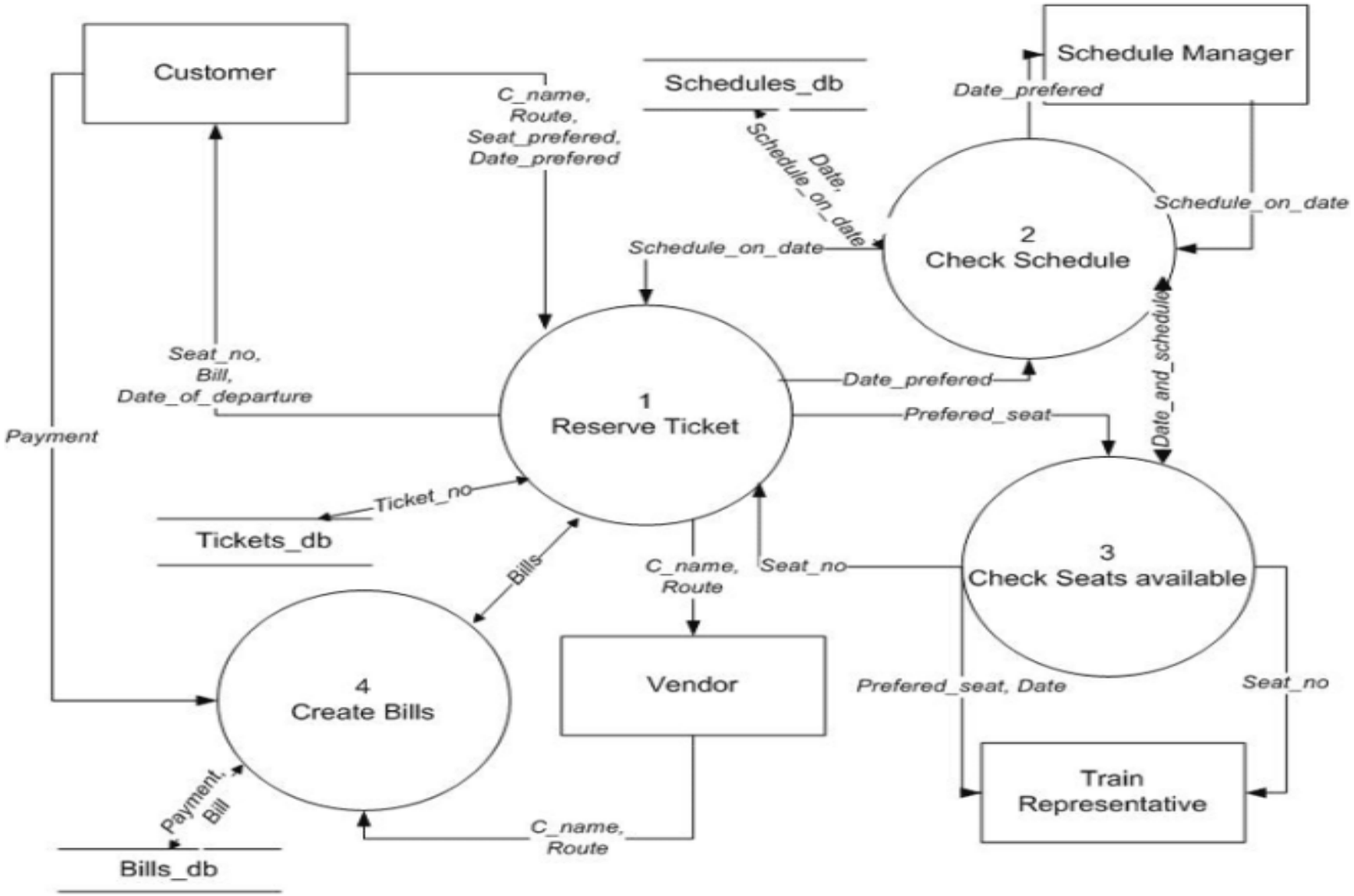




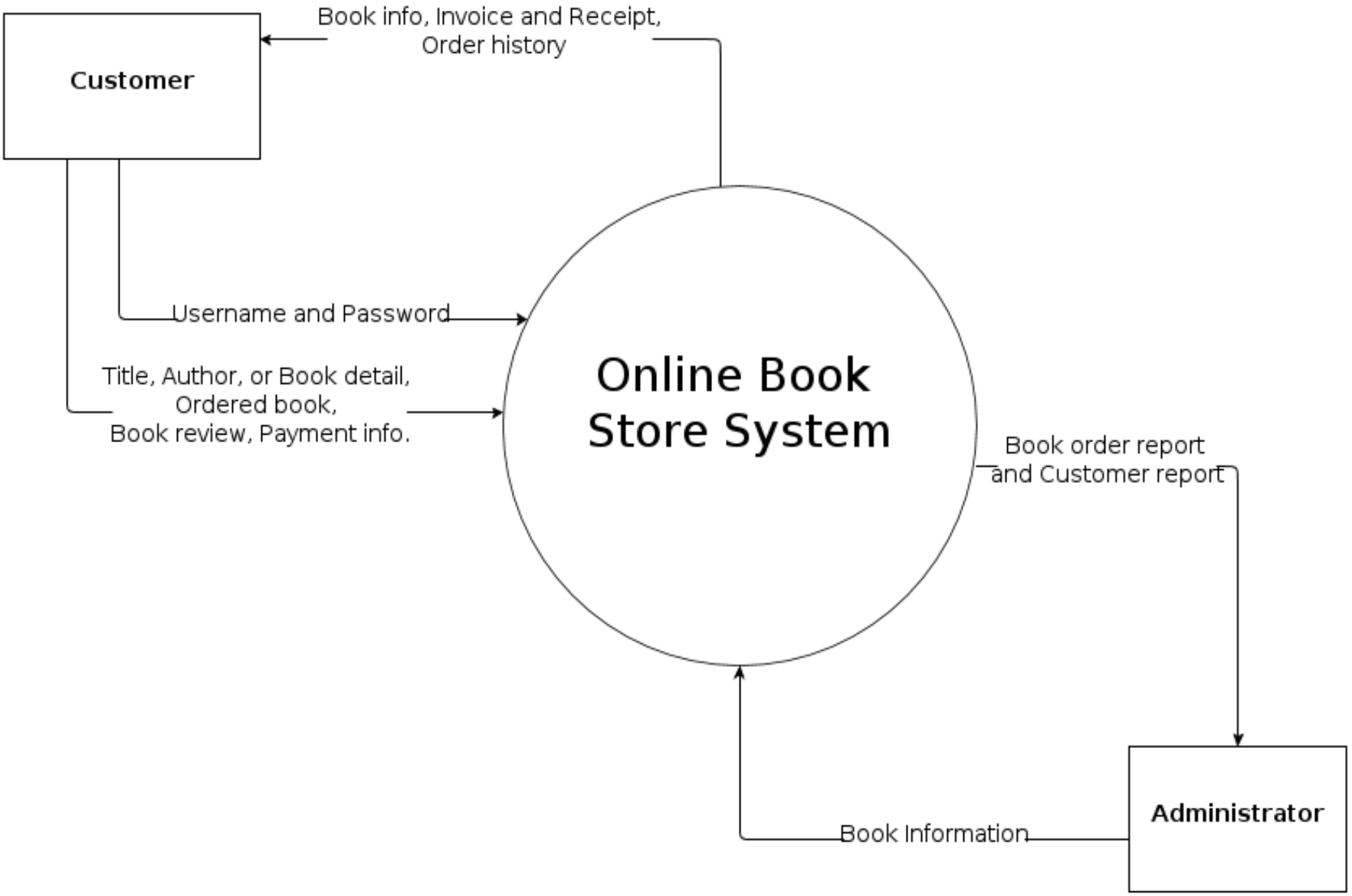
Level 0 DFD


thecomputerstudents.com
customer
C_name,
Route,
Prefered_date
Payment
tickets
vendor
Payment, seat_prefered
tickets
Time_of_departuer
Ticket
Reservation
C_name,
Route,
prefered_date
Route,
Prefered_date
Seat_prefered
Schedule manager
Departure_time
Seat_on
Train representative

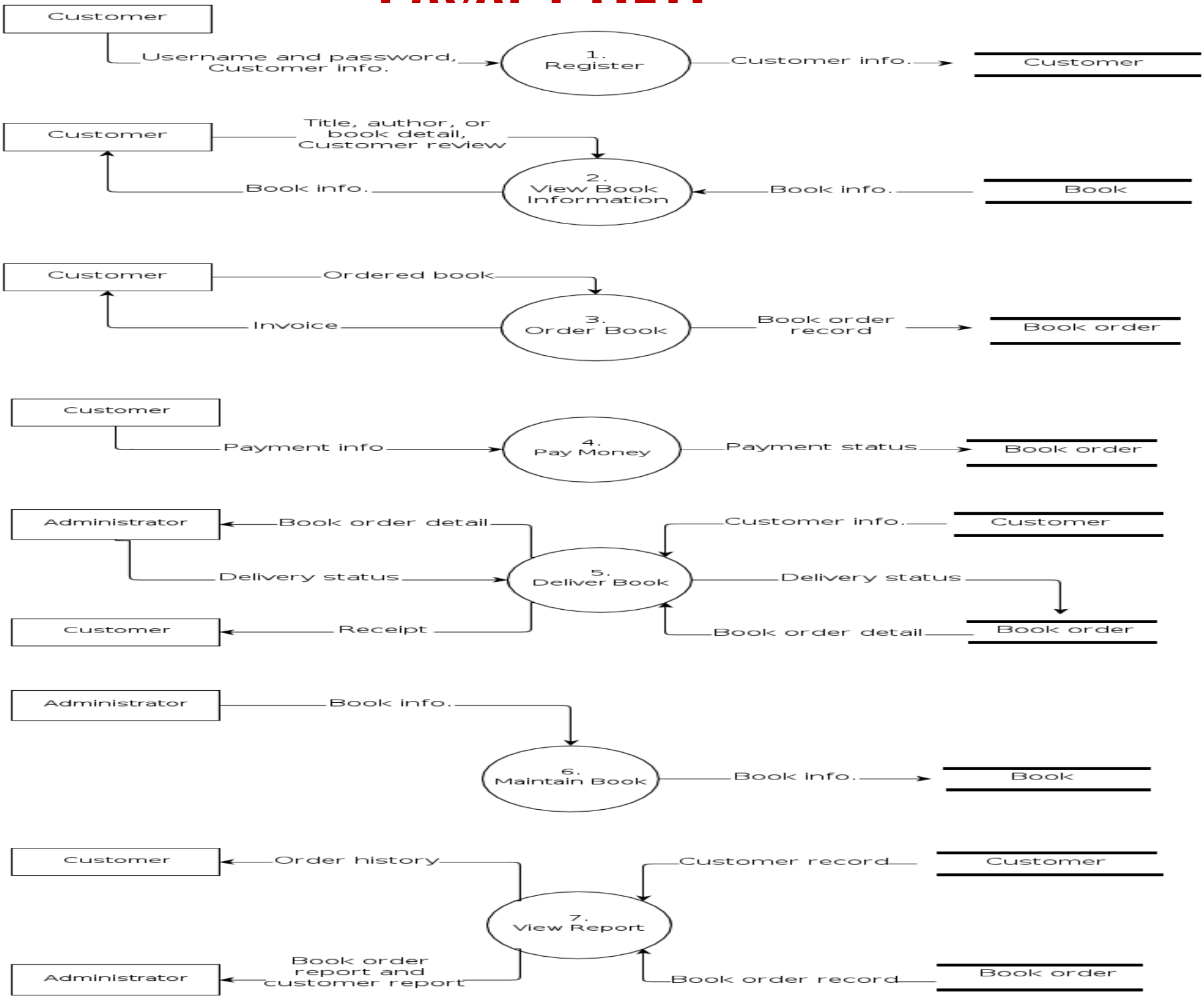
Level 1 DFD

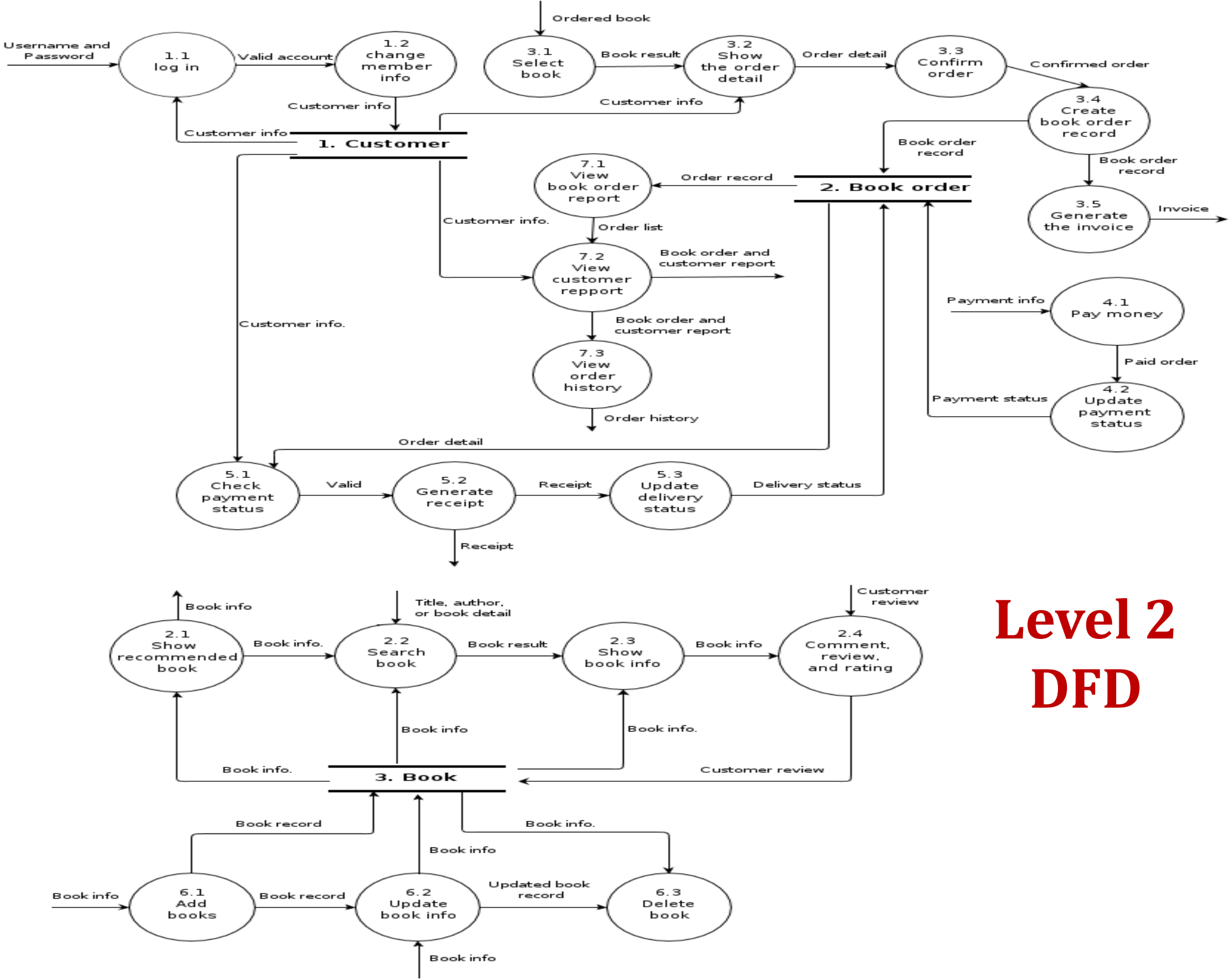


Level 0 DFD


Customer
Book info, Invoice and Receipt,
Order history
Username and Password
Title, Author, or Book detail,
Ordered book,
Book review, Payment info.
Online Book
Store System
Book order report
and Customer report
Administrator
Book Information

Level 1 DFD


Customer
Username and password, Customer info.
1. Register
Customer info.
Customer
Customer
Title, author, or book detail, Customer review
Book info.
2. View Book Information
Book info.
Book
Customer
Ordered book
Invoice
3. Order Book
Book order record
Book order
Customer
Payment info
4. Pay Money
Payment status
Book order
Administrator
Book order detail
Delivery status
Customer
Receipt
5. Deliver Book
Customer info.
Customer
Delivery status
Book order
Book order detail
Administrator
Book info.
6. Maintain Book
Book info.
Book
Customer
Order history
7. View Report
Customer record
Customer
Administrator
Book order report and customer report
Book order record
Book order


Ordered book
Username and Password
1.1 log in
Valid account
1.2 change member info
3.1 Select book
Book result
3.2 Show the order detail
Order detail
3.3 Confirm order
Confirmed order
3.4 Create book order record
Customer info
Customer info
Customer info
1. Customer
Book order record
Book order record
7.1 View book order report
Order record
2. Book order
3.5 Generate the invoice
Invoice
Customer info.
Order list
7.2 View customer repport
Book order and customer report
Payment info
4.1 Pay money
Customer info.
Book order and customer report
7.3 View order history
Paid order
Payment status
4.2 Update payment status
Order history
Order detail
5.1 Check payment status
Valid
5.2 Generate receipt
Receipt
5.3 Update delivery status
Delivery status
Receipt
Book info
Title, author, or book detail
Customer review
2.1 Show recommended book
Book info.
2.2 Search book
Book result
2.3 Show book info
Book info
2.4 Comment, review, and rating
Book info
Book info.
Book info.
Book info.
Customer review
3. Book
Book record
Book info.
Book info
Book info
Book record
6.1 Add books
Updated book record
6.2 Update book info
6.3 Delete book
Book info

Level 2 DFD

2.6 Data Flow Diagram (DFD)

❖ Advantages of DFD

- Very simple to understand and easy to use.
- Used foe documentation.
- Provide detailed description.
- Provide clear understanding.
- It explains logic of the system.
- Less number of symbols.
- Provides structured analysis.

❖ Disadvantages of DFD

- No control information.
- Difficult to represent real time systems.
- Different symbols in different notations.

Object Modeling with UML

- Very popular now a days.
- Found in 1980s.
- Real world based modeling approach for design.
- Support object oriented design.
- Also provide general framework for information system.
- It is visual representation of software or system objects, attributes, actions, and relationships.

❖ UML

- Full form.
- UML is a general purpose modeling language in the field of software engineering, which is designed a standard way to visualize the design of the system. Developed in 1990s.
- First step of developing real world object oriented design.
- Goal → common language for object modeling.

Object Modeling with UML

- UML is not a design methodology.
- UML making system easy to understand using less number of primitive symbols.
- At a glance, UML is a powerful and widely-used tool for software development that helps to ensure that software systems are well-designed, well-documented, and meet the needs of their users.

❖ UML Diagrams

Structure Diagrams

Class Diagram

Component Diagram

Object Diagram

Package Diagram

Composite Structure Diagram

Profile Diagram

Deployment Diagram

Behaviour Diagrams

Use case Diagram

Activity Diagram

Sequence Diagram

Communication Diagram

State machine Diagram

Interaction overview Diagram

Timing Diagram

2.7 Use case Diagram

- It is a kind of behavioural diagrams in UML.
- Use cases represent the different ways in which a system can be used by the users.
- Consists of 'Use cases'.

It is a representation of a user's interaction with the system that shows the relationship between the users and different use cases in which user is involved.

- Purpose → define logical behaviour of the system.
- Identifies functional requirements.
- It describes "who can do what in a system".
- Represents a sequence of interactions between the user and the system.
- Use cases corresponding to the high-level functional requirements.

- A use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.
- Scenarios in which your system or application interacts with people, organizations, or external systems

2.7 Use case Diagram

⇨ **For example**: Use cases in ATM system

- Check balance
- Change PIN
- Print mini statement
- Transfer money
- Withdraw money

❖ Components of use case diagram

- Three main components along with relationship.

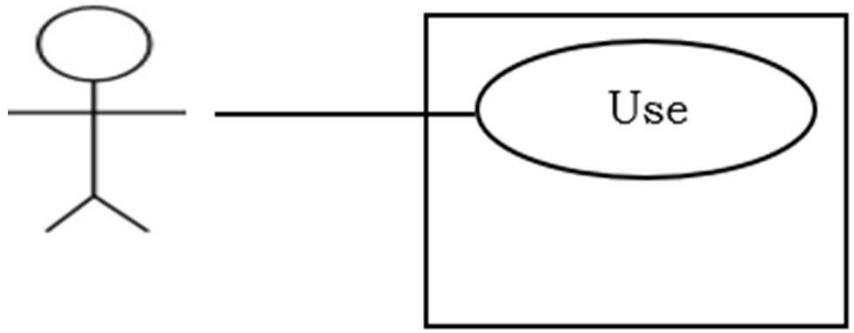
1. Use case

- Represented by an ellipse.
- All the use cases are enclosed with a rectangle boundary.
- It identifies the functional requirements of the system.

2.7 Use case Diagram

2. Actor

- Outside of the system.
- Represented by stick person icon.
- It may be a person, machine or any external system.
- Actors are connected to use cases by drawing a simple line connected to it.
- *Each actor must be linked to at least one use case, while some use cases may not be linked to actors.*



Actor

2.7 Use case Diagram

3. Relationship

- Represented using ‘a line’ between an actor and a use case.
- An actor may have relationship with more than one use case and one use case may relate to more than one actor.

≫ Different relationships in use case diagram

➢ Association

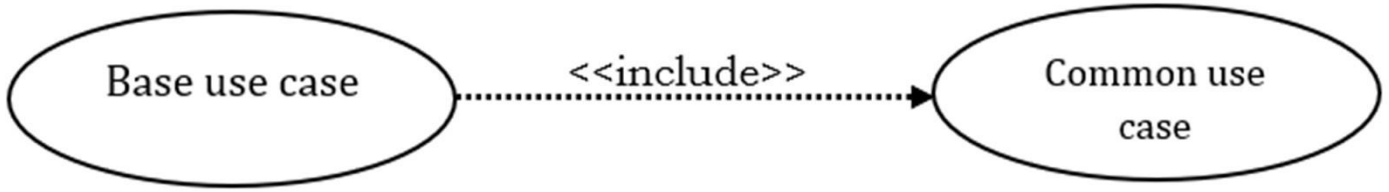
- Interface between an actor and a use case.
- Represented by joining a line from actor to use case.

➢ Include relationship

- It involves one use case including the behaviour of another use case.
- The “include” relationship occurs when a chunk of behaviour that is similar across a number of use cases.
- Represented using **<<include>>** stereotype.

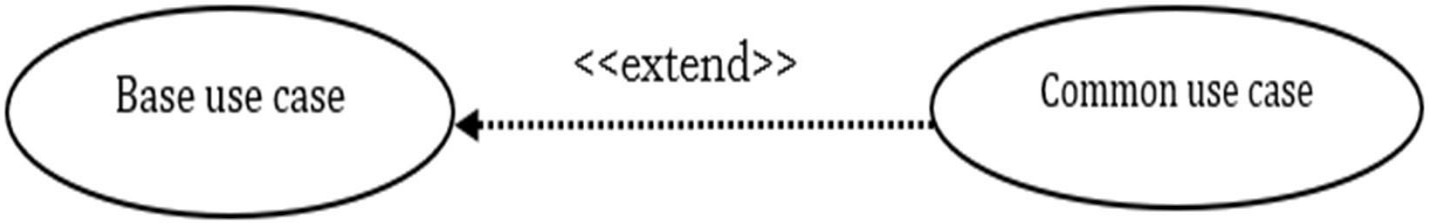
Include relationship: The use case is mandatory(compulsory) and part of the base use case. The include relationship is intended for reusing behavior modeled by another use case .

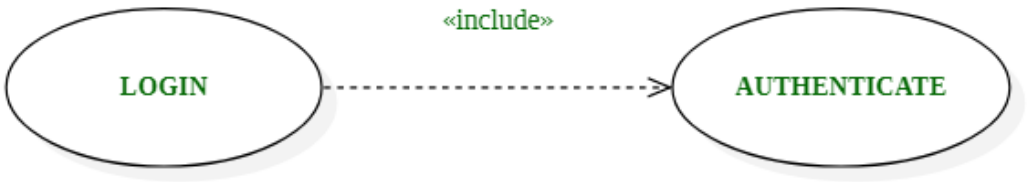
It is represented by a dashed arrow in the direction of the included use case with the notation **<<include>>.**



Extend relationship: The use case is optional and comes after the base use case. It is represented by a dashed arrow in the direction of the base use case with the notation <<extend>> .

• the extend relationship is intended for adding parts to existing use cases as well as for modeling optional system services"



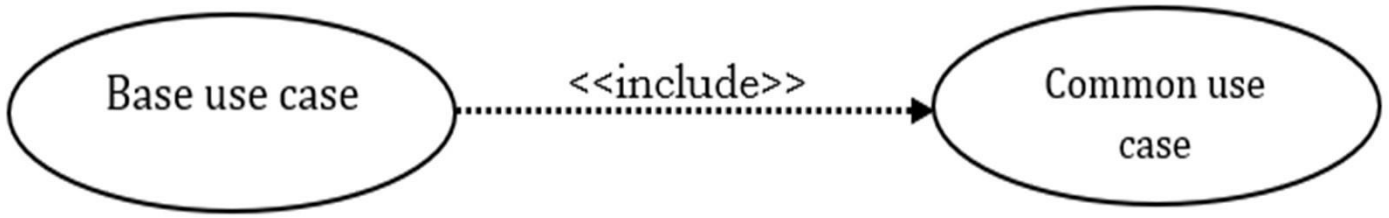


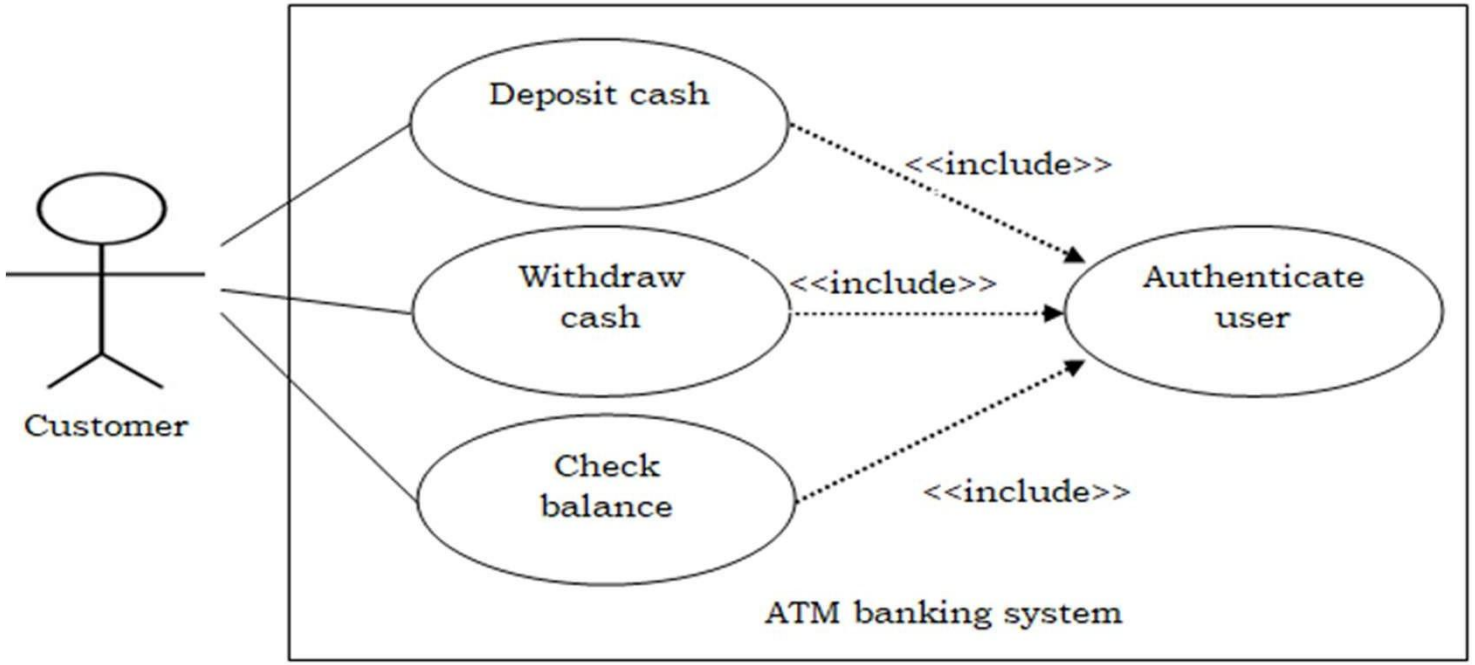
LOGIN is the BASE USE CASE and AUTHENTICATE is the INCLUDED USE CASE.

ILOGIN is the BASE USE CASE and INVALID PASSWORD is the EXTENDED USE CASE.

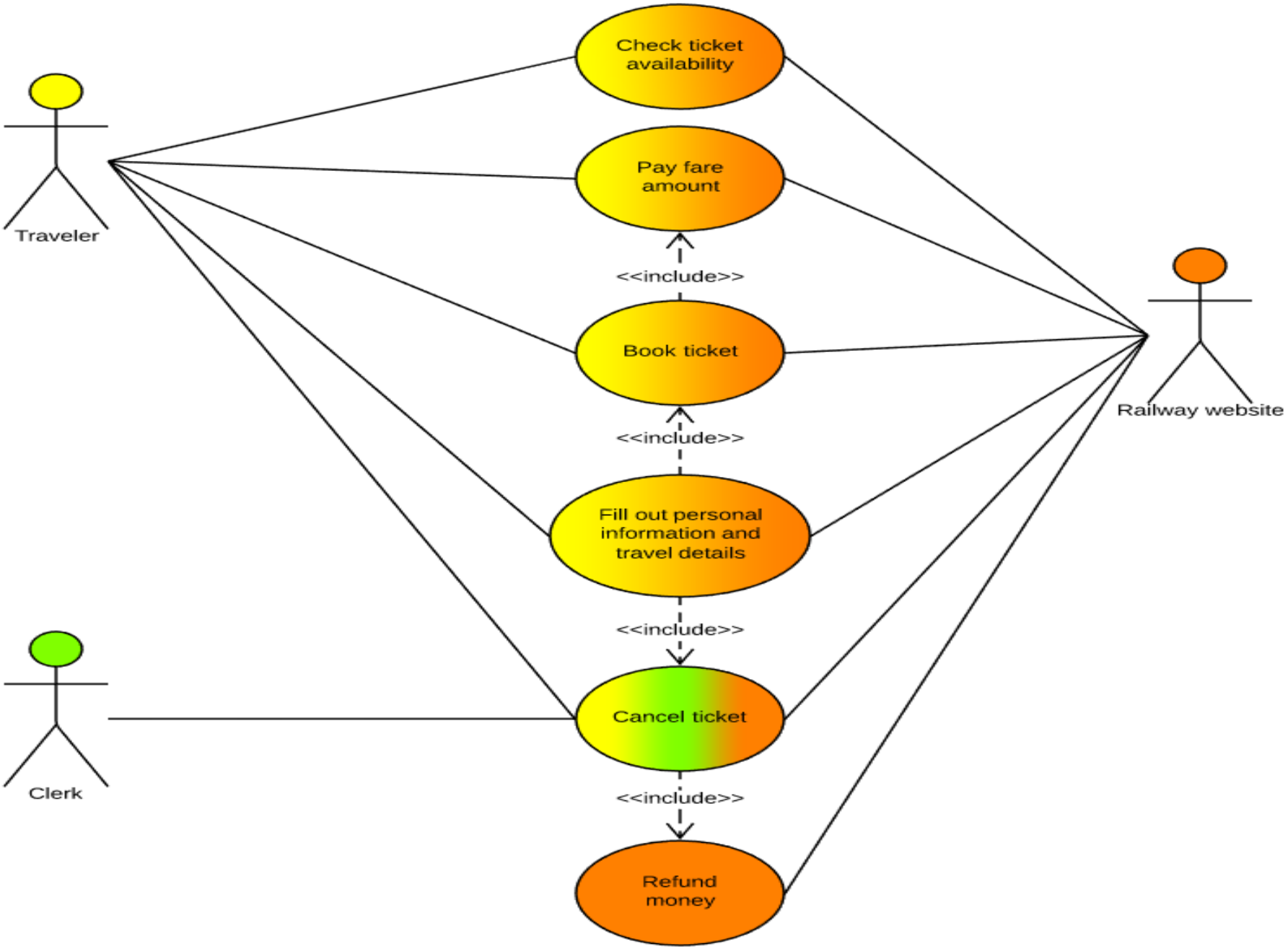
2.7 Use case Diagram

- **Include relationship**





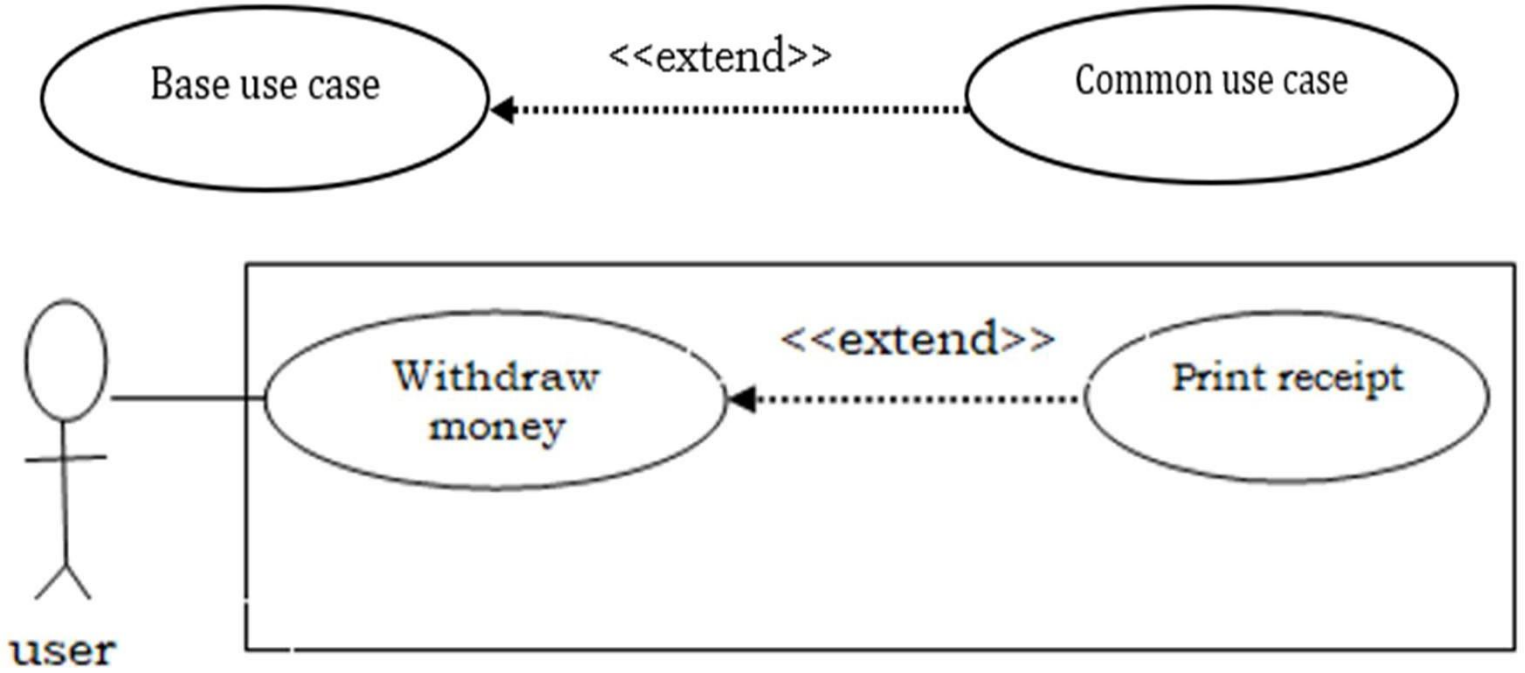
Railway reservation use case diagram example



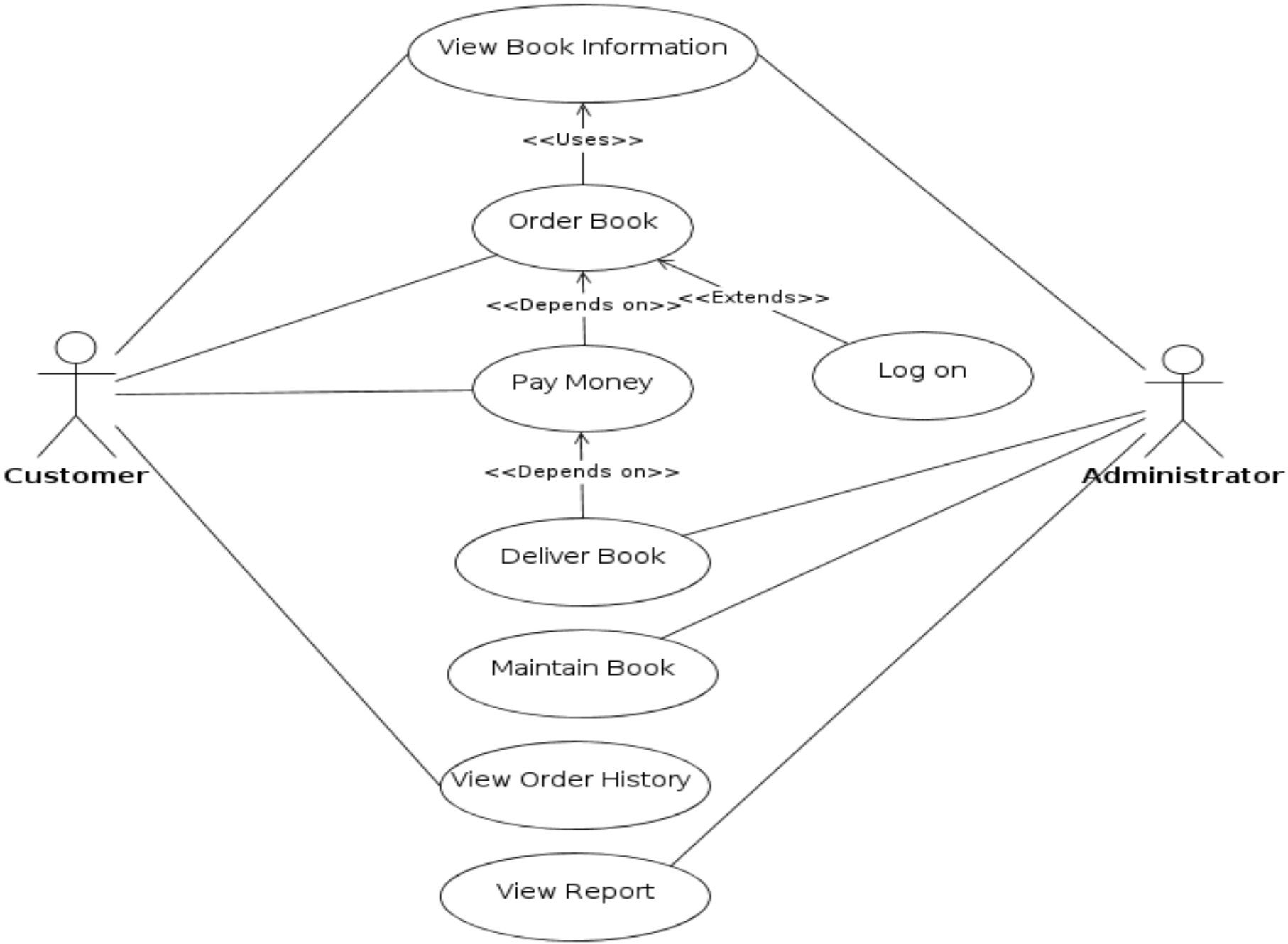
2.7 Use case Diagram

➢ Extend relationship

- Shows optional behaviour of the system.
- Represented using **<<extend>>** stereotype.
- Extend relationship exists when one use case calls another use case under certain condition (like: If – then condition).



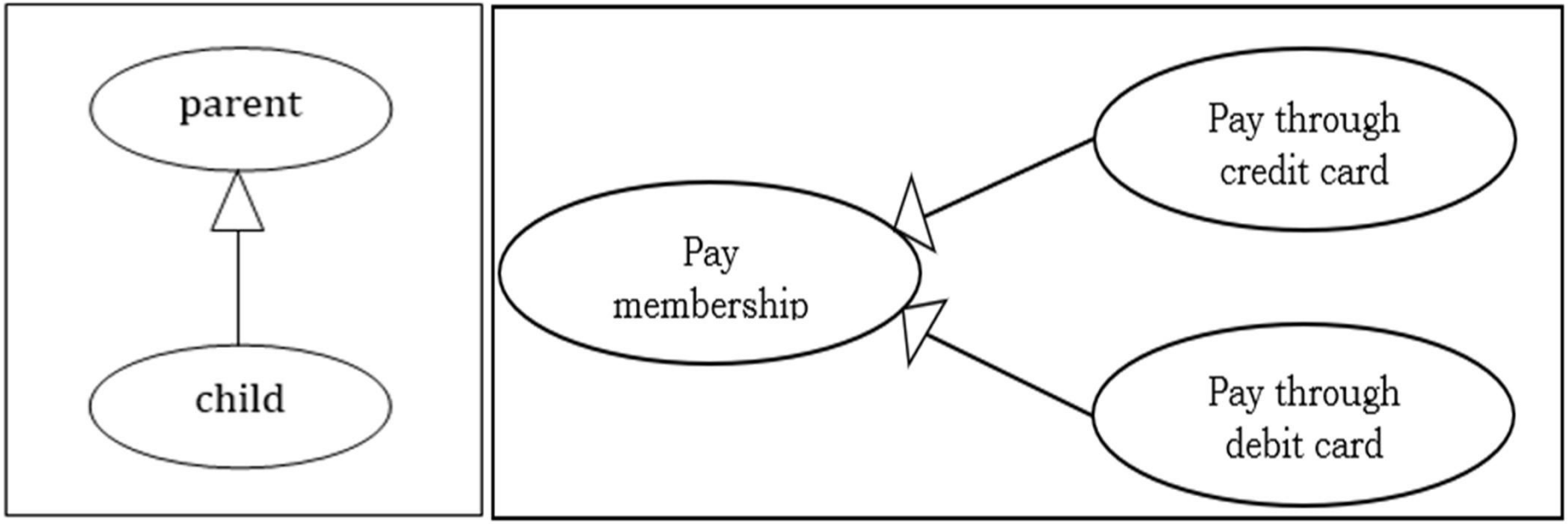
Use case Diagram



2.7 Use case Diagram

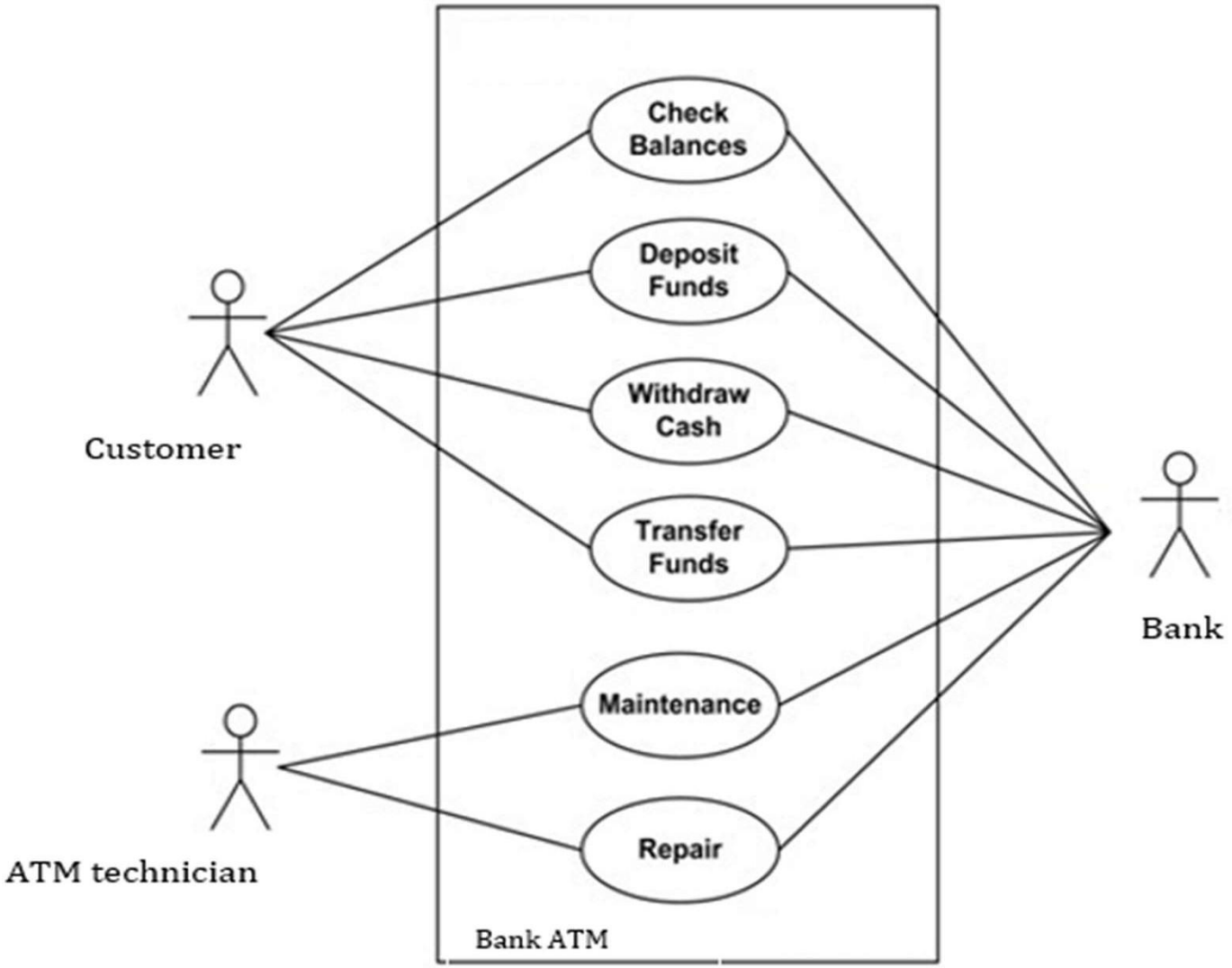
➢ Generalization

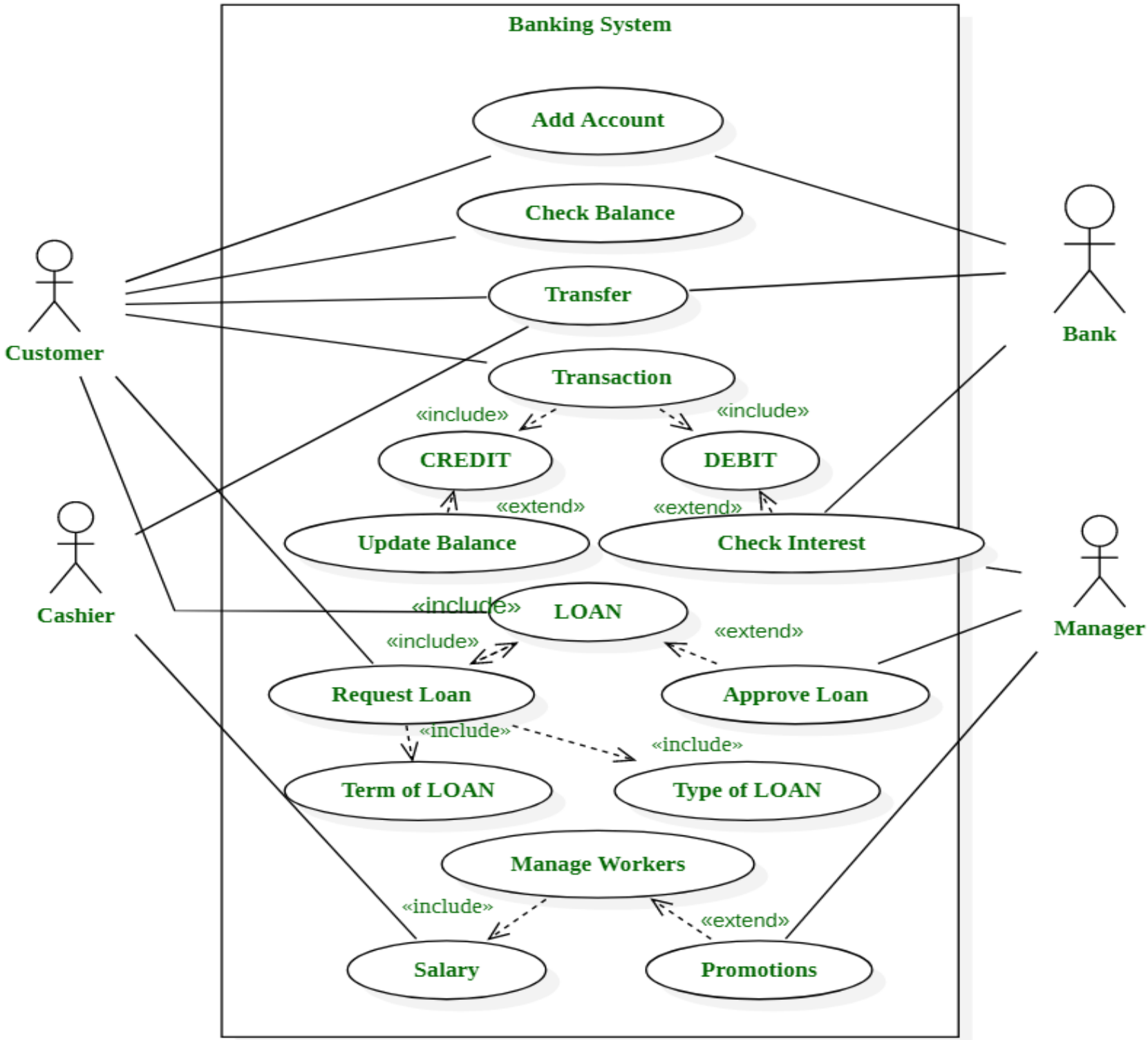
- It is a parent-child relationship between use cases.
- It can be used when you have one use case that is similar to another, but does slightly different.
- Represented as a directed arrow with a triangle arrowhead



2.7 Use case Diagram

- **Example (ATM System)**




Banking System
Add Account
Check Balance
Transfer
Transaction
CREDIT
DEBIT
Update Balance
Check Interest
LOAN
Request Loan
Approve Loan
Term of LOAN
Type of LOAN
Manage Workers
Salary
Promotions
«include»
«include»
«extend»
«extend»
«include»
«extend»
«include»
«include»
«include»
«extend»
Customer
Cashier
Bank
Manager

2.7 Use case Diagram

☐ Advantages of use case diagram

- Easy to understand and draw.
- Used to capture the functional requirements of the system.
- Used to analyse the system behaviour.
- Provide base for scheduling.
- Provides outside view of the system.

☐ Disadvantages of use case diagram

- Not fully object oriented.
- Do not provide information about the internal workings of the system.
- Becomes difficult in large system.
- Does not provide any guideline when to stop.
- Create ambiguity if use cases ae not defined clearly.

2.7 Use case Diagram

☐ Applications of use case diagram

- Requirement analysis
- High level design
- Reverse engineering
- Forward engineering
- Identifying test cases
- Business process modeling

2.8 Class Diagram

- It is a kind of static structure diagram of UML.
- Visually showing different classes, their attributes, methods, and the relationships between them.
- It shows how a system is structured rather than how it behaves.
- Inheritance can be represented.

≫ Purpose of class diagram

- Main purpose → to build static view of the system.
- It can be used as a base for component and deployment diagrams.
- We can get the idea of object behaviour and relationship between them.

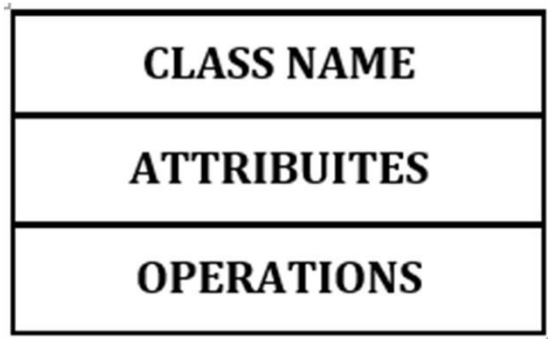
≫ Components of class diagram

- A UML class diagram made up of
  - Set of classes
  - Set of relationships between classes

2.8 Class Diagram

➢ Class

- A class is a group of objects all with common structure and behaviour.
- Class is used to create objects.
- In class diagram, class is represented using rectangle with three parts:



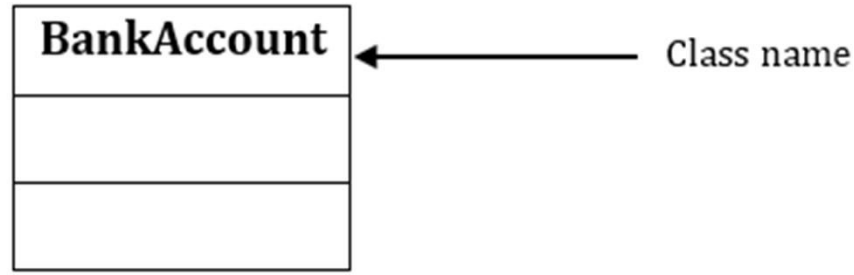
☐ Class name

- Top most part of class in class diagram.
- A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics.

2.8 Class Diagram

- Guidelines while representing class in diagram:
  - Class name should start with capital letter and capitalize the initial letters.
  - Place the class name in the centre of the first part.
  - Name of the abstract class should be written in italic format.

Example



Class: Bank Account

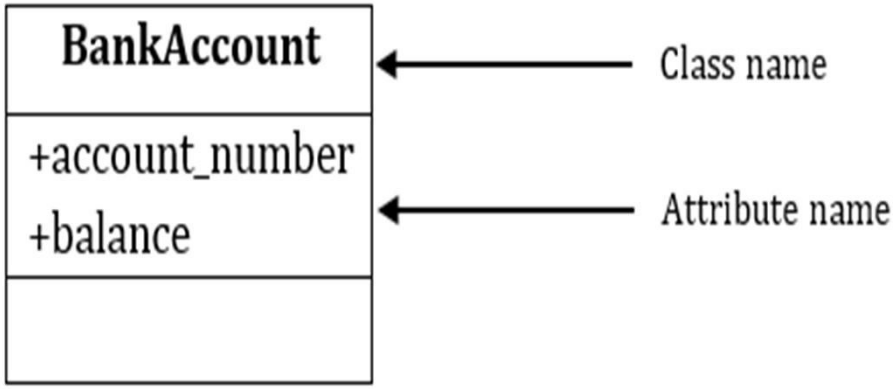
☐ Attributes

- Appears in the middle section of the class in class diagram.
- Attributes describe the properties or qualities of the class.

2.8 Class Diagram

- Characteristics of attributes:
  - Written along with visibility : Public, Private, Protected and Package
  - Accessibility of class determined by visibility factors
  - Meaningful name should be assigned to the attribute
- At the time of coding, attributes are mapped onto variables.

Example:



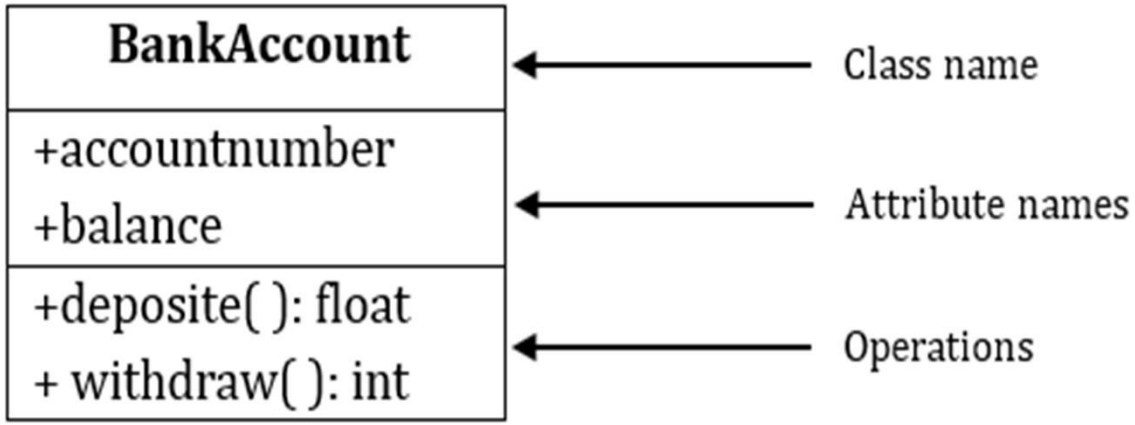
Class with attributes

2.8 Class Diagram

☐ Operations (Method) of the class

- Appears in the last section of the class in class diagram.
- Written in the list, one method in single line.
- Operations are the services that the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- At the time of coding, operations are mapped onto methods.

Example:



Class with attributes and operations (methods)

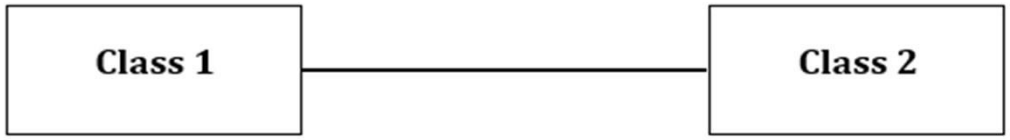
2.8 Class Diagram

☐ Relationship

- It is a connection or association between two or more classes.
- Describes functionality, relation and exchange of data.
- Different types of relationships:

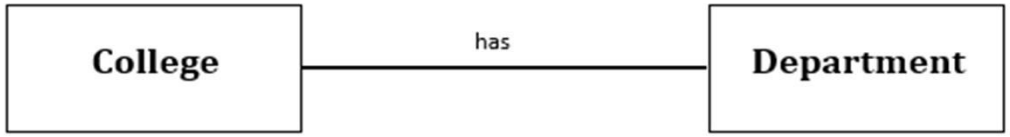
❑ Association

- It is a connection between two or more classes.
- Number of objects of one class connected to number of objects of another class.
- It can be one-to-one, one-to-many, many-to-many.
- Represented using sloid line between linked classes. Using '*verb*'.



Class 1 is associated with Class 2

For example – a department is associated with the college



2.8 Class Diagram

❑ Dependency

- Used where change in one class cause changes in another class.
- That indicates one class is depend on another.
- It is represented using a dashed arrow.

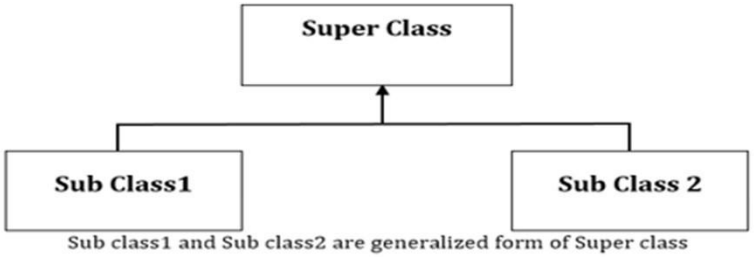
Class1 depends on Class2

For example – a student name is depended on his id.

2.8 Class Diagram

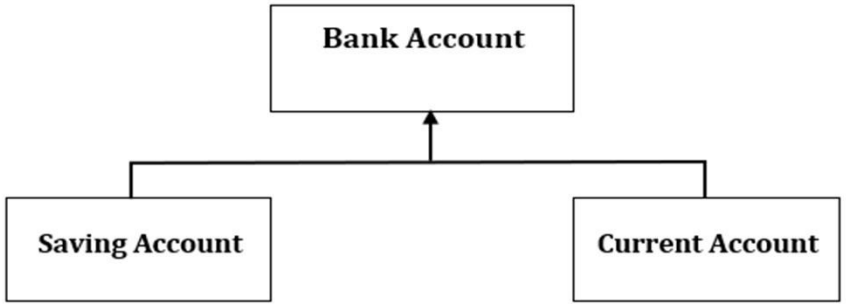
❑ Generalization

- Like inheritance.
- It is a relationship between a parent class (super class) and a child class (sub class).
- It represents '*is-a*' relationship.
- Generalization is represented using a solid line with arrow head that points from the child class to parent class.



Sub class1 and Sub class2 are generalized form of Super class

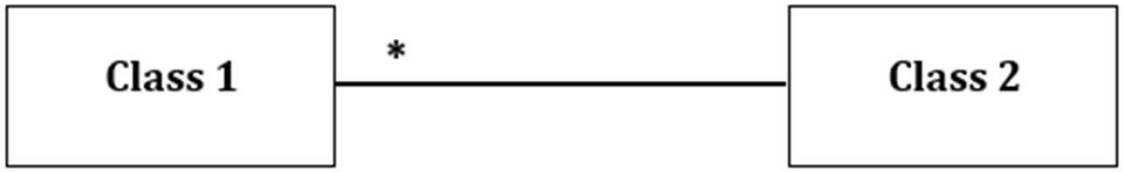
For example – Current Account and Saving Account are the generalized form of Bank Account.



2.8 Class Diagram

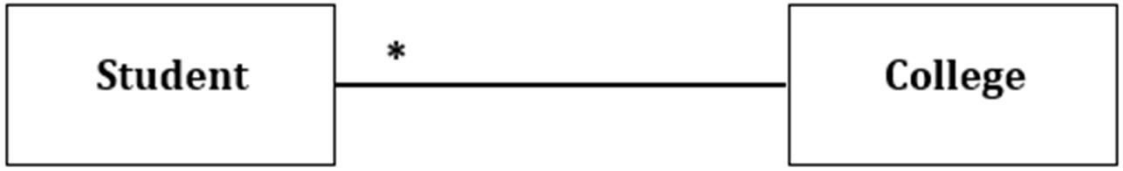
❑ Multiplicity

- It is associated with attributes. It shows haw many objects of each class take part in relationship.
- If it is not defined, it is considered as a default multiplicity.
- It can be expressed as:
  - Exactly one – 1
  - Zero or One – 0..1
  - Many - *
  - One or more – 1..*
  - Exact number – 2..3 or 5



Class 1 is associated with Class 2 with many multiplicity relationship

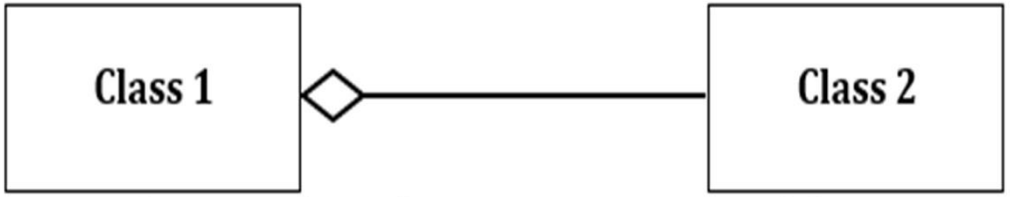
For example – many students are studying in college



2.8 Class Diagram

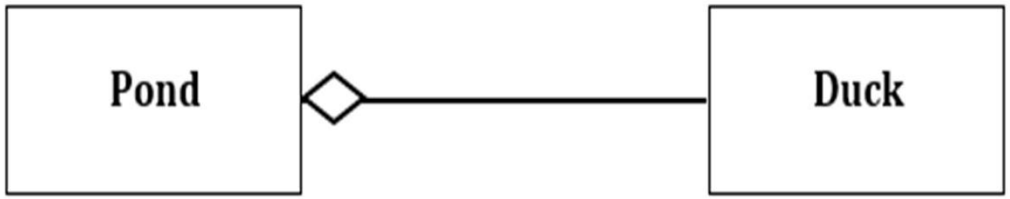
❑ Aggregation

- It is a type of association that represents a part-whole or part-of relationship between two classes.
- It is denoting '*consist-of*' OR '*has-a*' hierarchy.
- It shows that one class is composed of one or more instances of another class.
- It is represented using diamond symbol.



Class 1 is made up of one or more components of Class 2

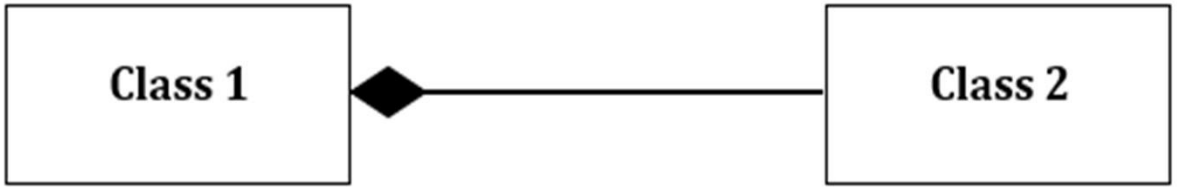
For Example – a pond is having many ducks and many other animals. So here if we consider pond and duck as class, we can show them as:



2.8 Class Diagram

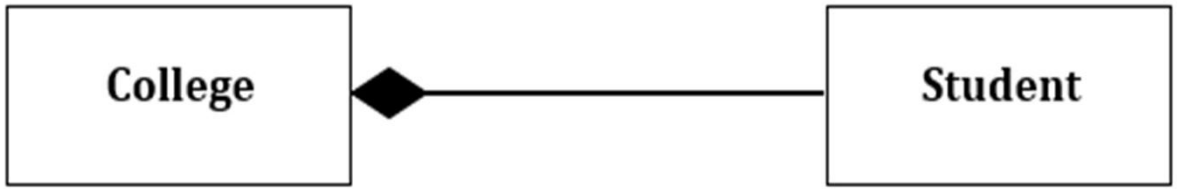
❑ Composition

- It is subset of aggregation, which denotes strong relationship between two classes when one class is part of another class.
- In this type of relationship if the parts are destroyed then the whole is destroyed.
- It is represented using a filled diamond symbol at the whole end.



(Class 1 and Class 2 are in composition relationship, where objects of Class 2 live and die with Class 1 and Class 2 cannot stand by itself.)

For Example – Class College is composed of class Student. The college could contain many students, while each student belongs to only one college. So, if college is not functioning all the students also removed.



2.8 Class Diagram

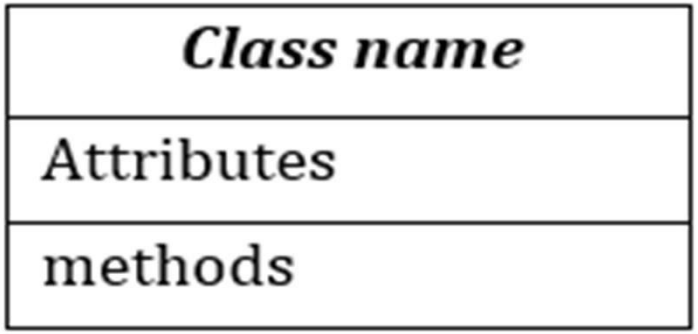
❑ Difference between aggregation and composition

Aggregation	Composition
- It illustrates 'has-a' relationship.	- It illustrates 'part-of' relationship.
- It can be represented using hollow diamond.	- It can be represented using filled diamond.
- In aggregation, child classes can have their own lifetime.	- Here, child classes don't have their own lifetime.
- In Aggregation, the child object will remain in the system even if we remove the parent object.	- In Composition, the child object will get deleted if we delete the parent object.
- In it, the objects are independent of each other.	- In it, the objects are dependent on each other.
- It is less strong association.	- It is more strong association.

2.8 Class Diagram

❑ Abstract class

- Abstract class is a special type of class where no objects can be a direct entity of the abstract class.
- The abstract class can neither be declared nor be instantiated.
- The notation of the abstract class is similar to that of class; the only difference is that the name of the class is written in italics.



2.8 Class Diagram

☐ Advantages of class diagram

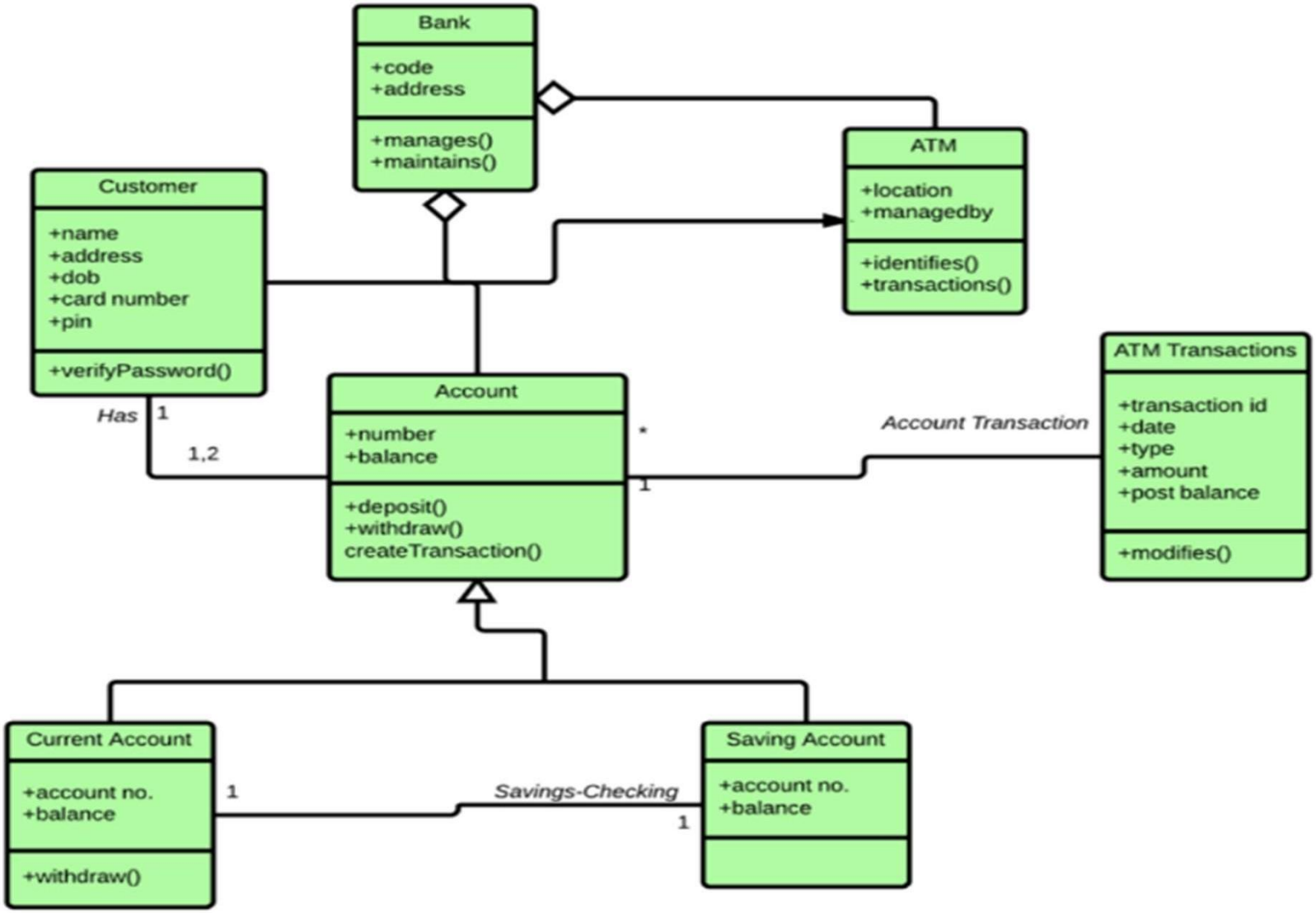
- Describe static view of the system.
- Describe functionalities of the system.
- Used to construct object oriented applications.
- Used to plan testing scenario.
- Improve the quality and efficiency of the product.

☐ Disadvantages of the class diagram

- Overcomplicated or overwhelming sometime.
- Take long time to manage.
- Tedious and time consuming at the time of changes in system.
- Ambiguous when multiple relationships and associations are there.
- Difficult to manage it when requirements are changed.

2.8 Class Diagram

- **Example (ATM system)**



2.9 Sequence Diagram

- It is a kind of behavioural diagram in UML.
- It shows the interactions between objects in time-ordered sequence.
- It is used to visualize the flow of messages, events, or actions between the different elements in a system.

≫ Purpose of sequence diagram

- To model high level interactions among active objects.
- Used to visualize the dynamic behaviour of the system.

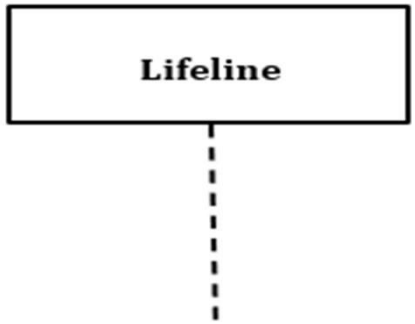
≫ Components (notations) of sequence diagram

➢ Lifeline

- It is individual participant.
- Located at the top of the diagram.
- Each lifeline is represented by a vertical dashed line that extends down the length of the diagram and a rectangle box at the top of line is labelled with the name of the object or component it represents.

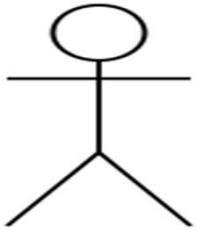
2.9 Sequence Diagram

- It indicates the periods during which an object exists or is active.



➢ Actor

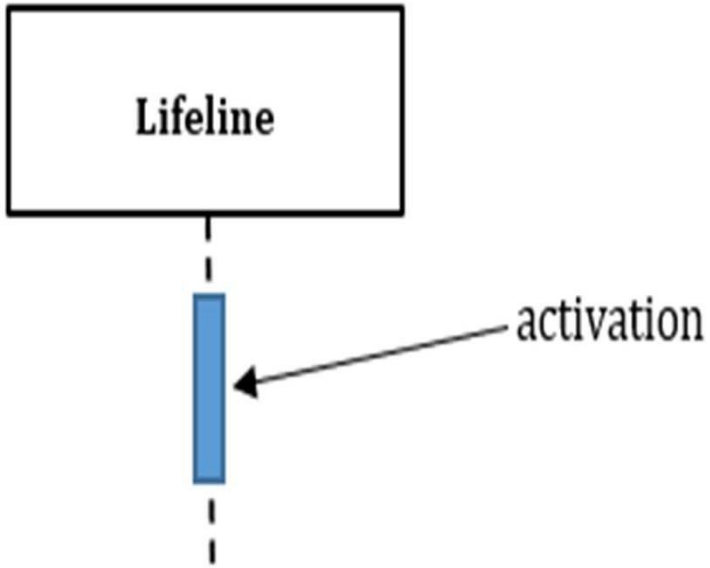
- An actor represents a user, system, or external entity that interacts with the system.
- Actors are represented as stick person icon outside the system boundary, and they are connected to the system by a dashed line.



2.9 Sequence Diagram

➢ Activation

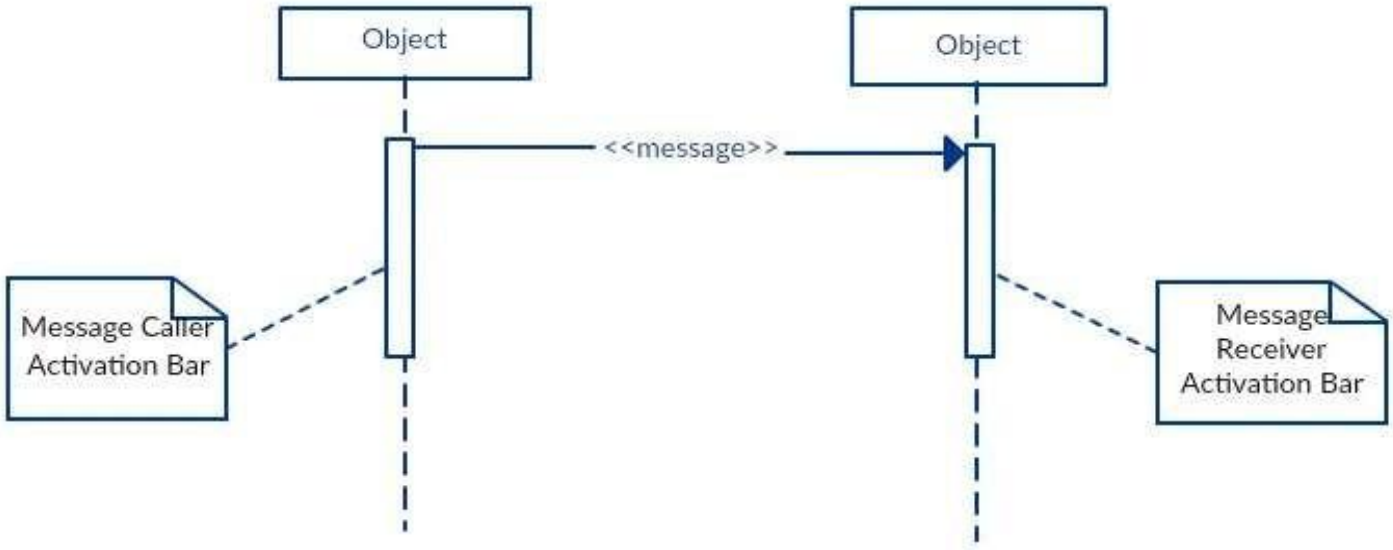
- It is runtime behaviour of the object.
- It is showing the time of object where it is activated for particular duration.
- It is represented by a thin rectangle on the lifeline.



2.9 Sequence Diagram

➢ Message

- Message represents the communication between objects.
- Used to convey the information from one object to another. (typically in case of request or response)
- Messages are represented by arrows that connect the lifelines of objects. The arrowhead indicates the direction of the message, and the label on the arrow indicates the type of message being sent.

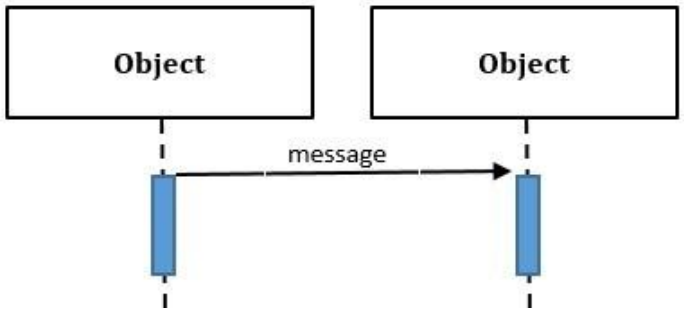


2.9 Sequence Diagram

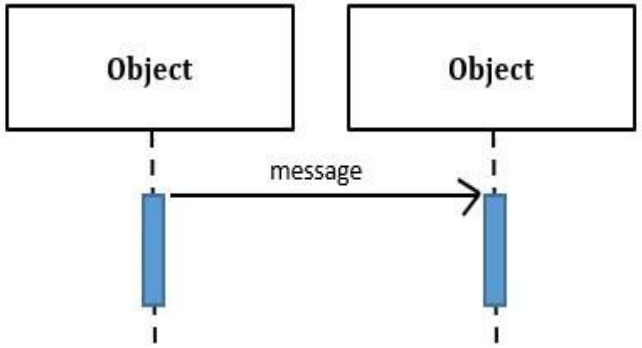
Different types of messages used in class diagram

↦ Call message

- It defines a particular communication between Lifelines.
- Call message is of two types:
  - ***Synchronous call message***



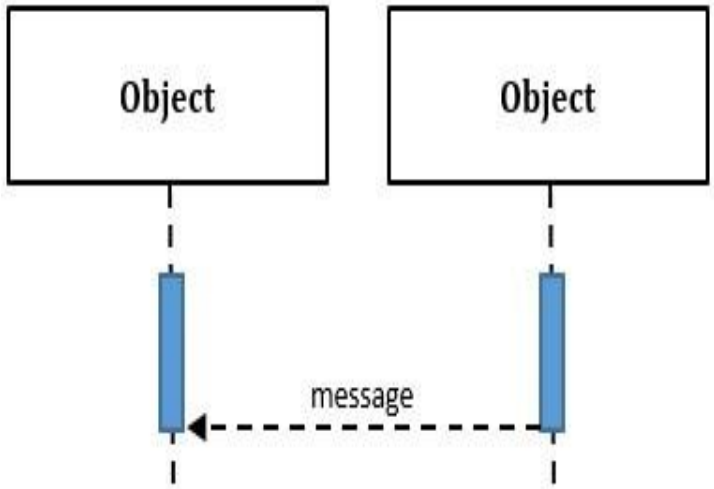
  - ***Asynchronous call message***



2.9 Sequence Diagram

↦ Return message

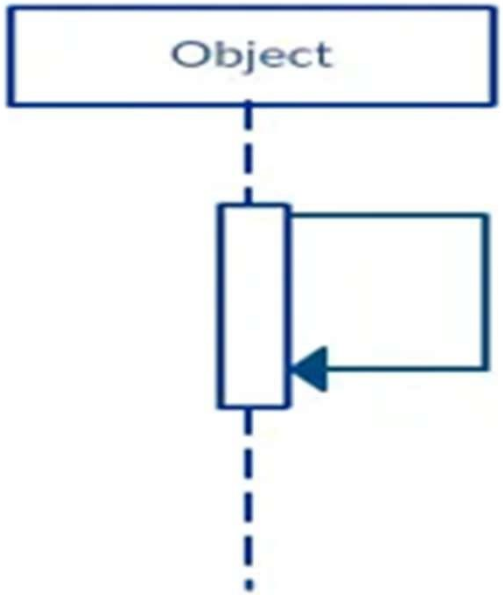
- A return message is used to return the result of an operation to the calling object.
- It is represented by dashed arrow.
- It is only used in response to synchronous messages.



2.9 Sequence Diagram

↦ Self message

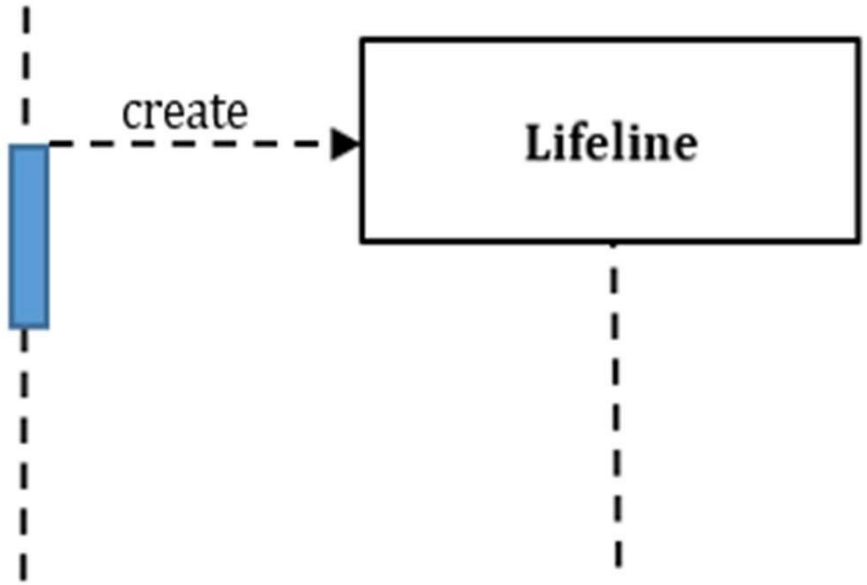
- A message from an object to itself.
- It is represented by a looped arrow.
- Self-messages are often used when an object needs to perform an internal operation or to trigger another operation within itself.



2.9 Sequence Diagram

↦ Create message

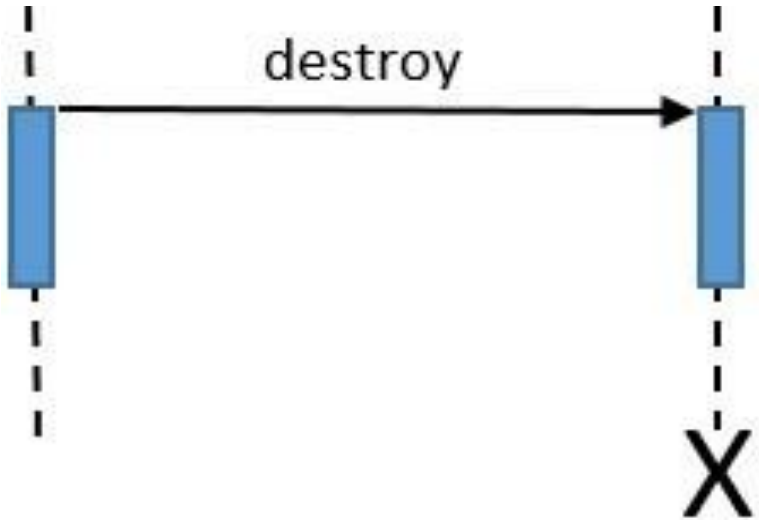
- It describes creation of a new object or instance of a class.
- It is shown as a dashed line with open arrowhead (looks the same as reply message), and pointing to the created lifeline's head.



2.9 Sequence Diagram

↦ Destroy message

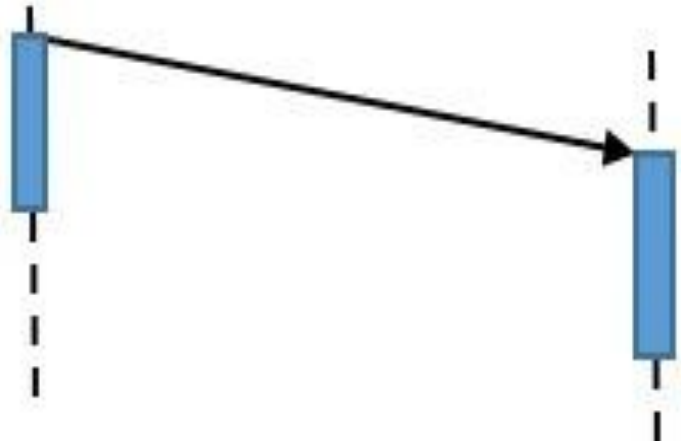
- It represents the deletion or destruction of an object.
- It is represented by an arrow terminating with a X.



2.9 Sequence Diagram

↦ Duration message

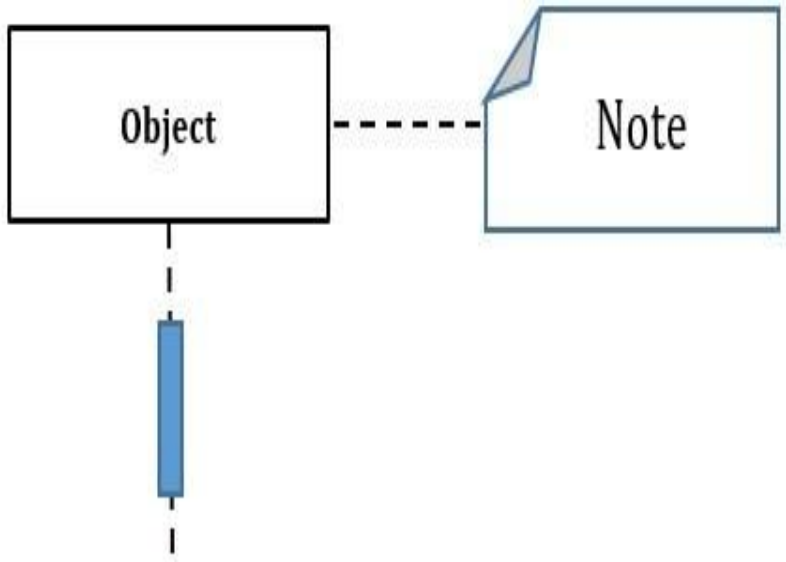
- That represents the duration of an activity in the system.
- It is denoted by a horizontal line with an optional label indicating the duration of the activity.



2.9 Sequence Diagram

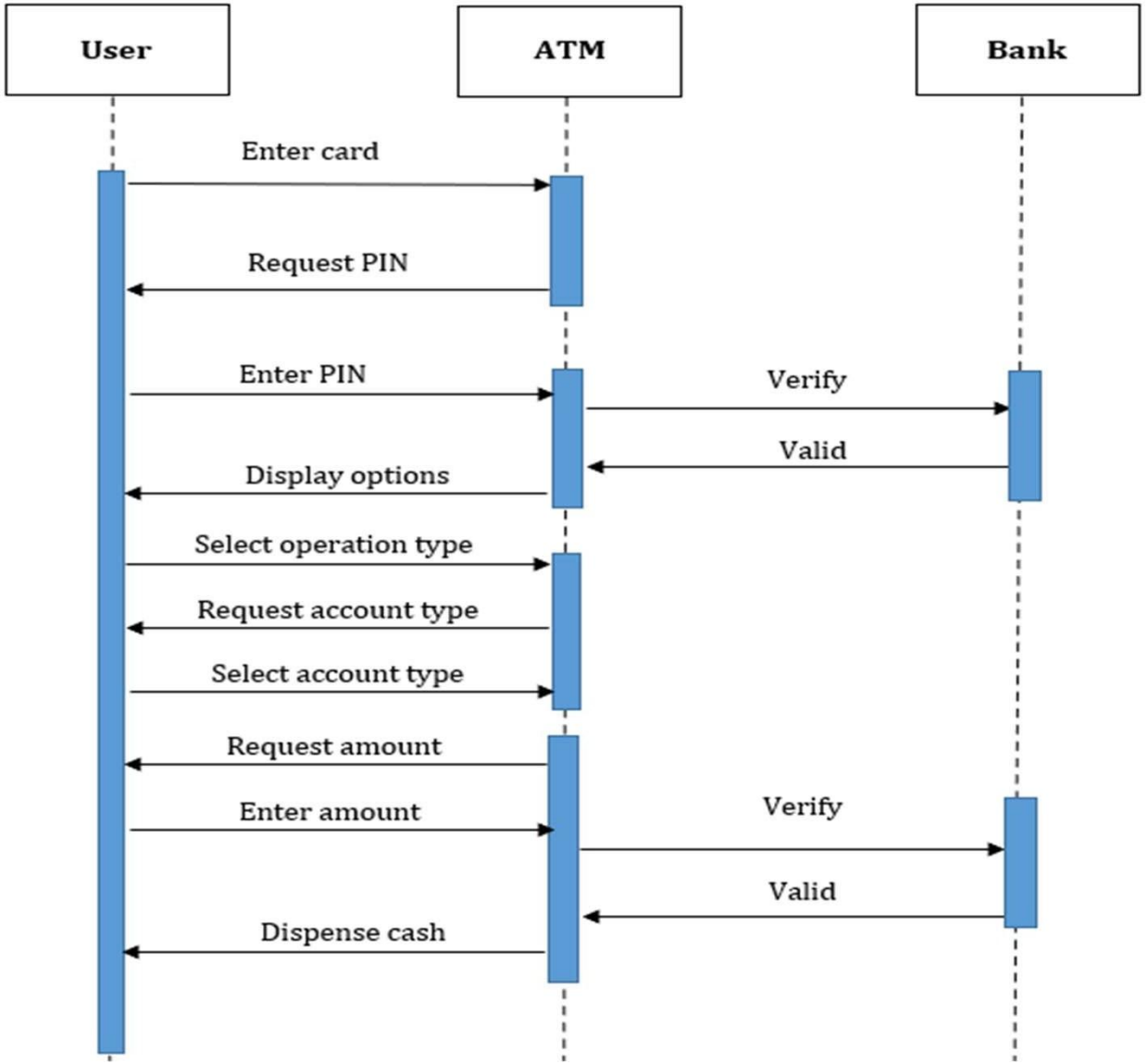
↦ Note (Or Comment)

- Provide remarks.
- Basically carries useful information.
- It is represented by a rectangle with a folded corner. And it can be linked with the related object with dashed line.



2.9 Sequence Diagram

- **Example of Sequence diagram (ATM system – *'withdraw amount'*)**



2.10 Activity Diagram

- It is a kind of behavioural diagram in UML.
- It models the actions (behaviour) performed by the system components, the order in which the actions take place and conditions related to actions.
- It consists of activities, states and transitions between activities and states.
- **Aim** → record the flow of control from one activity to another.
- Same as procedural work flow, but difference is that, activity diagram supports parallel activities.
- Important feature is '*swimlanes*', that enables group activities.

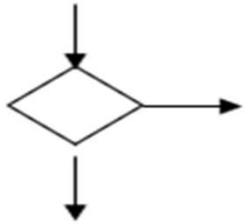
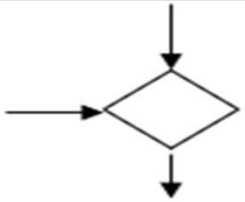
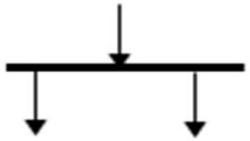
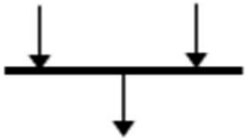
2.10 Activity Diagram

➢ Elements (Components) of activity diagram

Elements or Components and its description	Symbol
Activity - It represents a particular action taken in the flow of control. - It is denoted by a rectangle with rounded edges. And labelled inside it describing corresponding activity. - There are two special type of activity nodes: ***1. Initial activity (OR Start activity)*** - This shows the starting point or first activity of the flow. - It is denoted by a solid circle. ***2. Final activity (OR End activity)*** - The end of the activity diagram shown by a bull's eye symbol. It represents the end point of all activities.	
Flow or Transition - A flow (also termed as edge or transition) is represented with a directed arrow. - This is used to show transfer of control from one activity to another.	

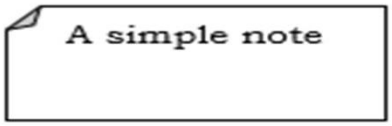
2.10 Activity Diagram

➢ Elements (Components) of activity diagram

Decision (Branch) - A decision node represented with a diamond. - It is a branch where single transition (flow) enters and several outgoing transitions.	
Merge - This is represented with a diamond shape with two or more input transitions and a single output transition.	
Fork - Fork is a point where parallel activities begin. - Fork is denoted by black bar with one incoming transition and several outgoing transitions. - 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.	

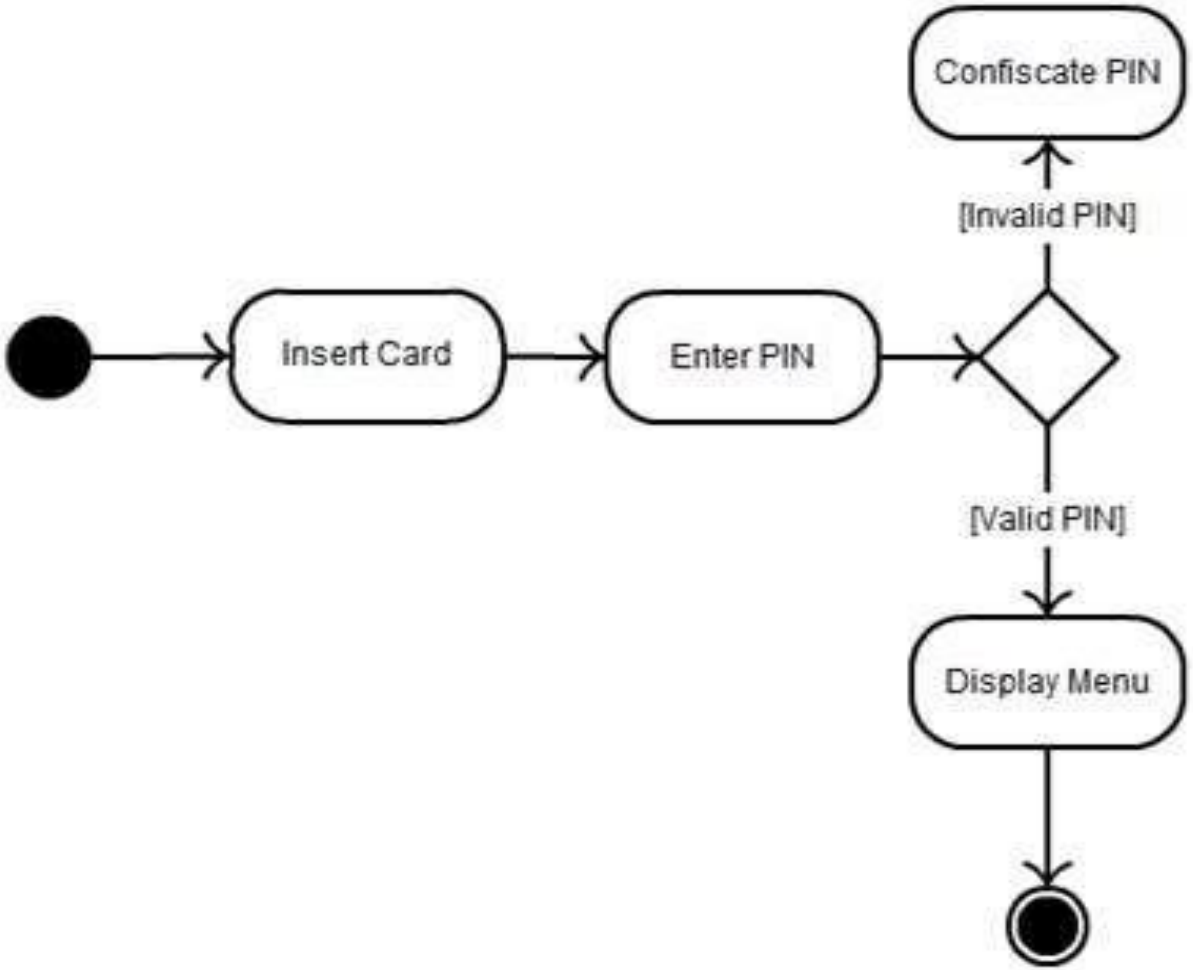
2.10 Activity Diagram

➢ Elements (Components) of activity diagram

Note - UML allows attaching a note to different components of diagram to present some textual information. - It could be some comments or may be some constraints. - A note generally attached to a decision point to indicate the branching criteria. - It is denoted by a rectangle with cut a side.	
Partition or Swimlanes - Different components of an activity diagram can be logically grouped into different areas, called partition or swimlanes. - They often correspond to different users or different units of organization. - It is denoted by drawing vertical parallel lines. - Partitions in an activity diagram are not mandatory.	
Guard conditions - Guard conditions control transition from alternative transitions based on condition. - These are represented by square brackets.	**[ ]**

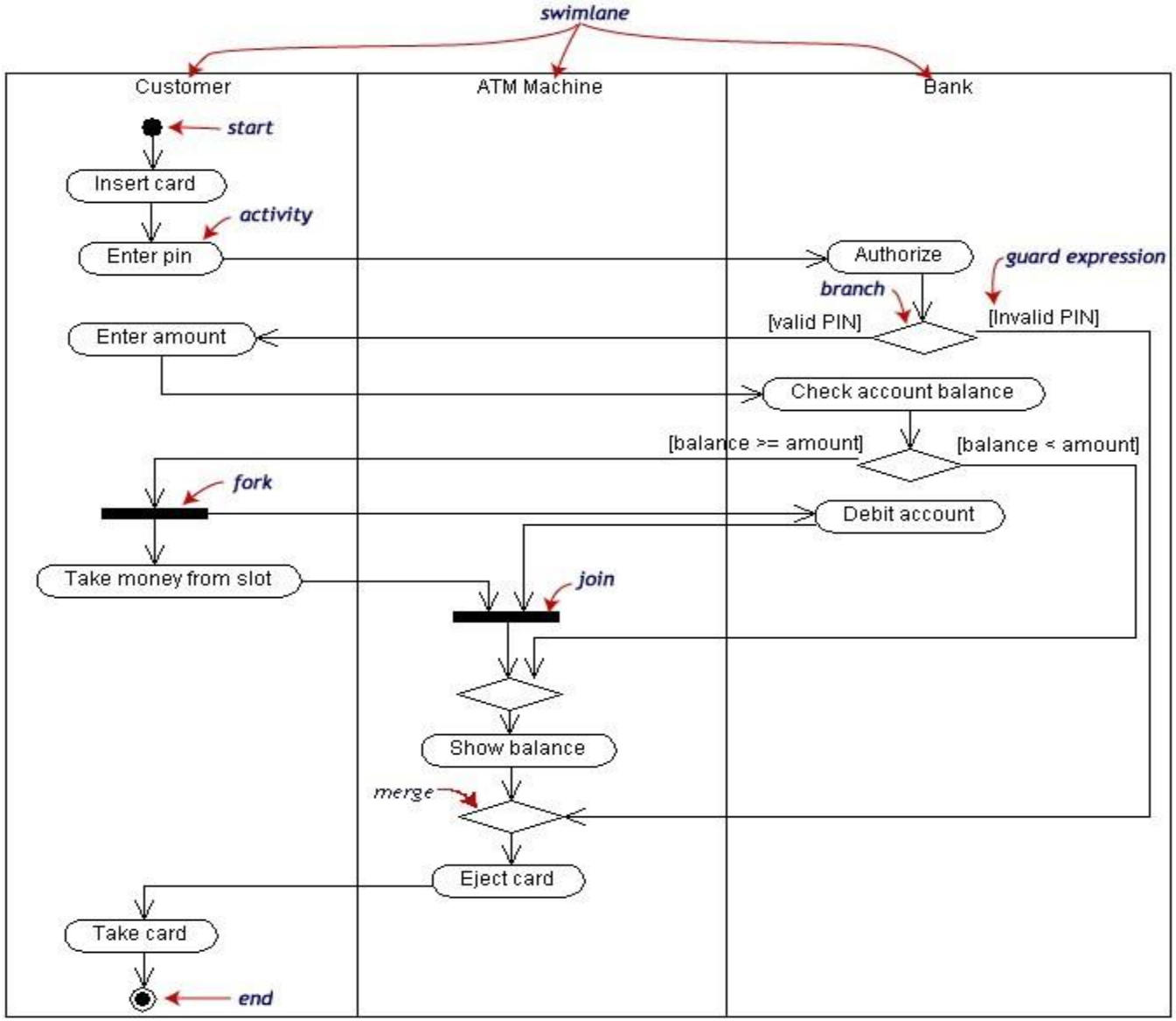
2.10 Activity Diagram

- **Example (Activity diagram for ATM system – card validation)**



2.10 Activity Diagram

- **Example (Activity diagram for ATM system)**



2.10 Activity Diagram

☐ Advantages of activity diagram

- Used to show the flow between activities.
- Provides a clear and concise visual representation of complex processes or workflows.
- It is easy to understand, even for non-technical stakeholders.
- Used to model parallel activities.
- It is good for describing synchronization and concurrency between activities.

☐ Disadvantages of the activity diagram

- Does not provide message part.
- It can't describe how objects collaborate.
- Lot many symbols sometimes create confusion.
- Time consuming and required significant efforts.
- Difficult to maintain when frequent change in requirements.
- Not suited for real time systems.

2.10 Activity Diagram

☐ Applications of activity diagram

- Mainly used to model business modeling.
- To document system behaviour.
- Can be used in requirement gathering from stake holders.

Thank you...

Prepared by:

Uresh N. Parmar

Lecturer, Computer Dept.
C U Shah Govt. Polytechnic
Surendranagar