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

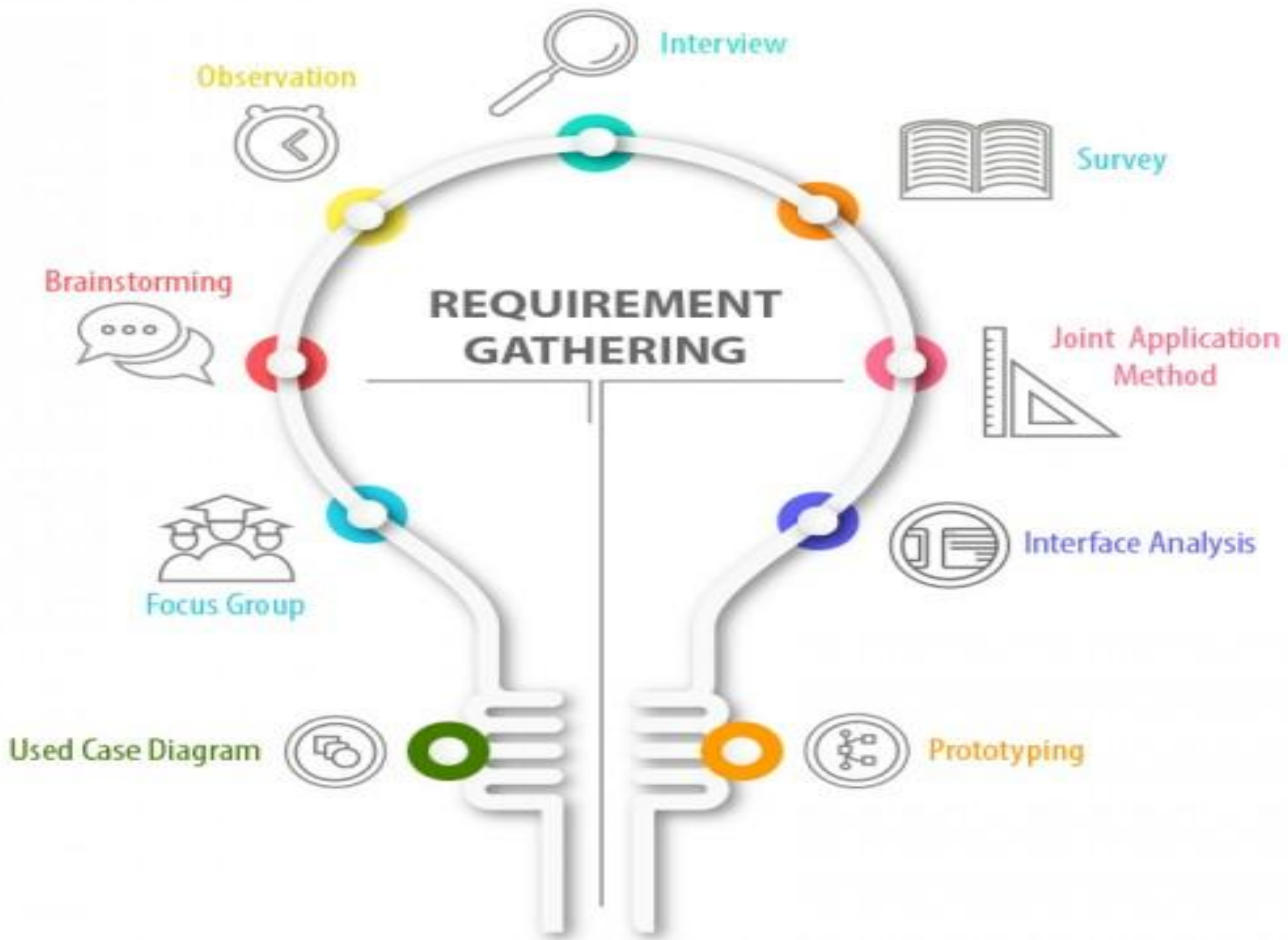
- It is the discovery phase of software development.
- Two sub processes → Requirement gathering and Requirement analysis.

❖ 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


Interview
Observation
Survey
Brainstorming
REQUIREMENT GATHERING
Joint Application Method
Focus Group
Interface Analysis
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 છૂટી પાડવી)
  - ***Analyzing requirements*** (છૂટીŽ પાડેલી requirements નું analysis કરવું)
  - ***Requirements recording or storing*** (બધીજ જરૂરીŽ requirements ને record કરવી)
- Performed by system analyst.
- For that, 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.

Analyzing Gathered Requirements

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

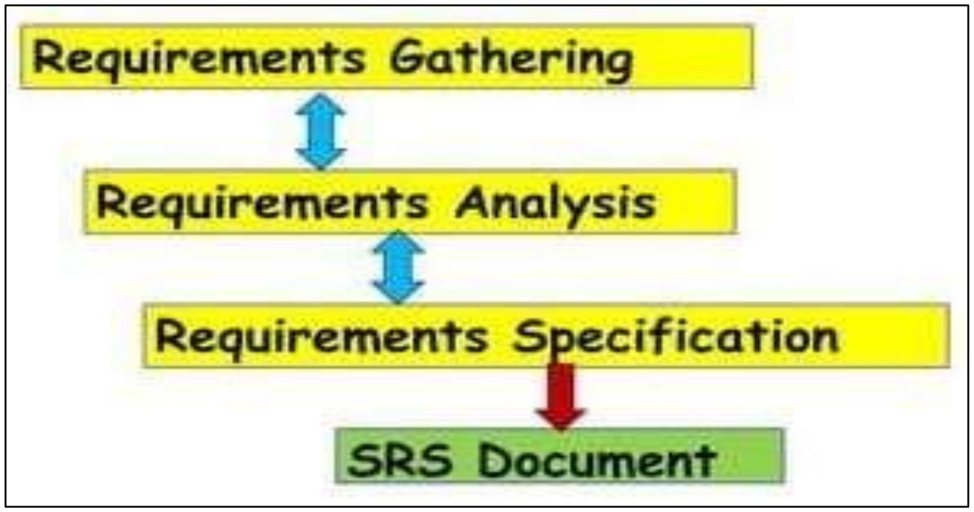
(1) Customer may provide some requirements that could be impossible to implement due to various dependencies.

(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

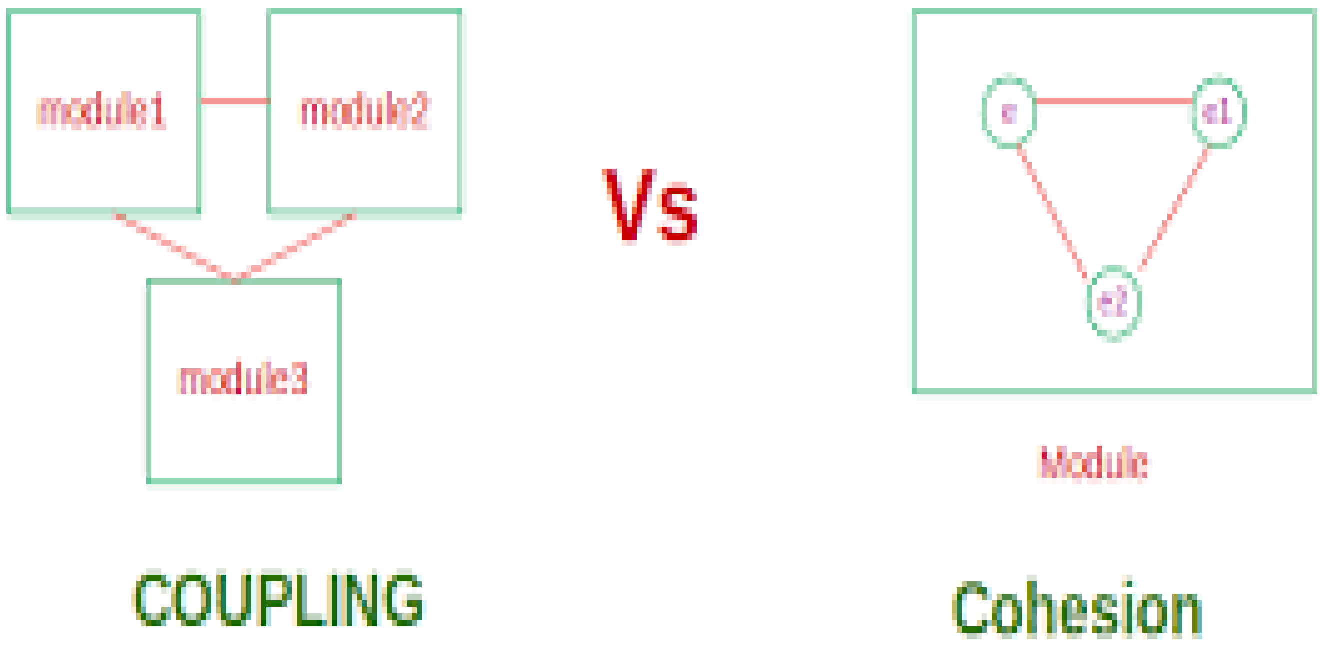
❖ Analysis vs Design

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

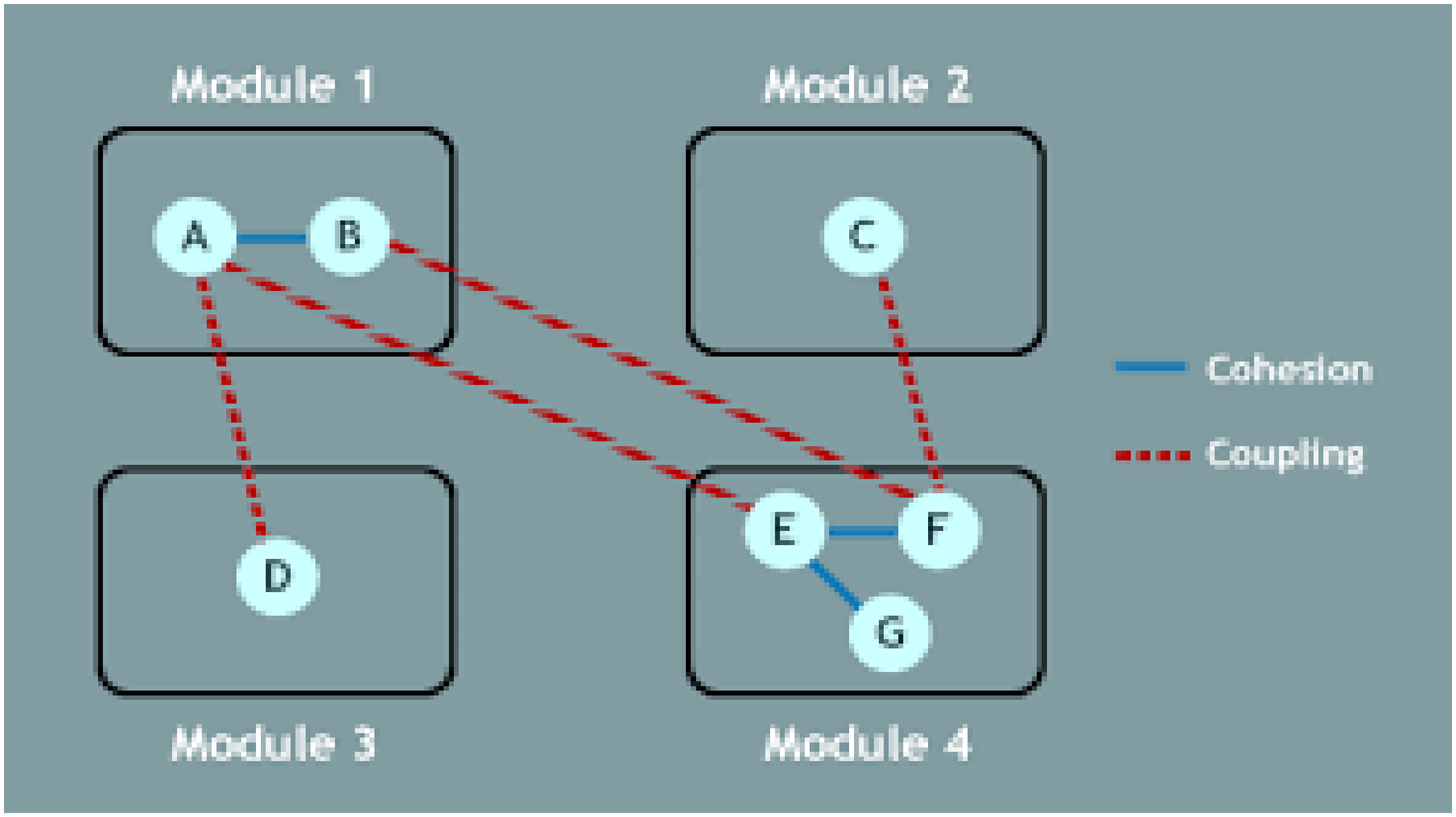
- Concept of modularity
- Software modularity is measured by how well software is decomposed into smaller pieces with standardized interfaces.
- Modularity - decomposition of program into smaller programs
- Due to modularity:
  - Easy to understand the system.
  - System maintenance is easy.
  - Provide 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

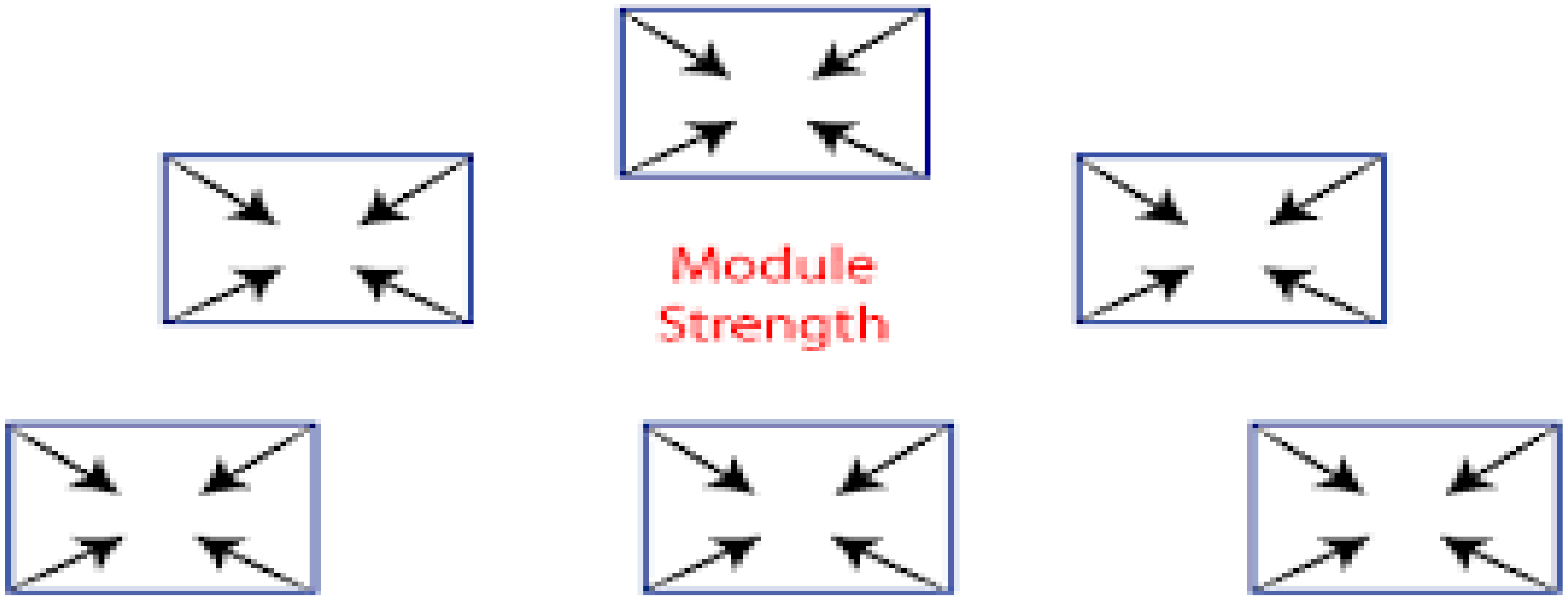


- ***high cohesion and low coupling*** is a primary need of any software development.

cohesion and coupling



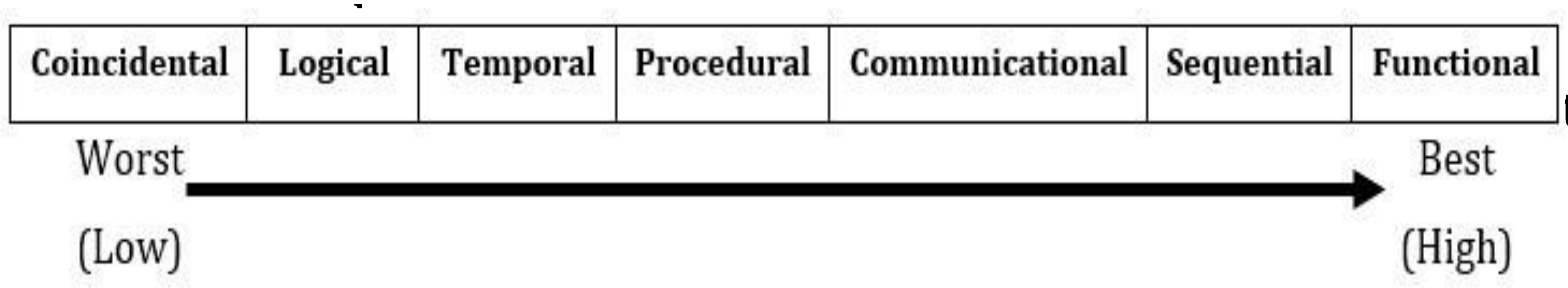
cohesion



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,



❖

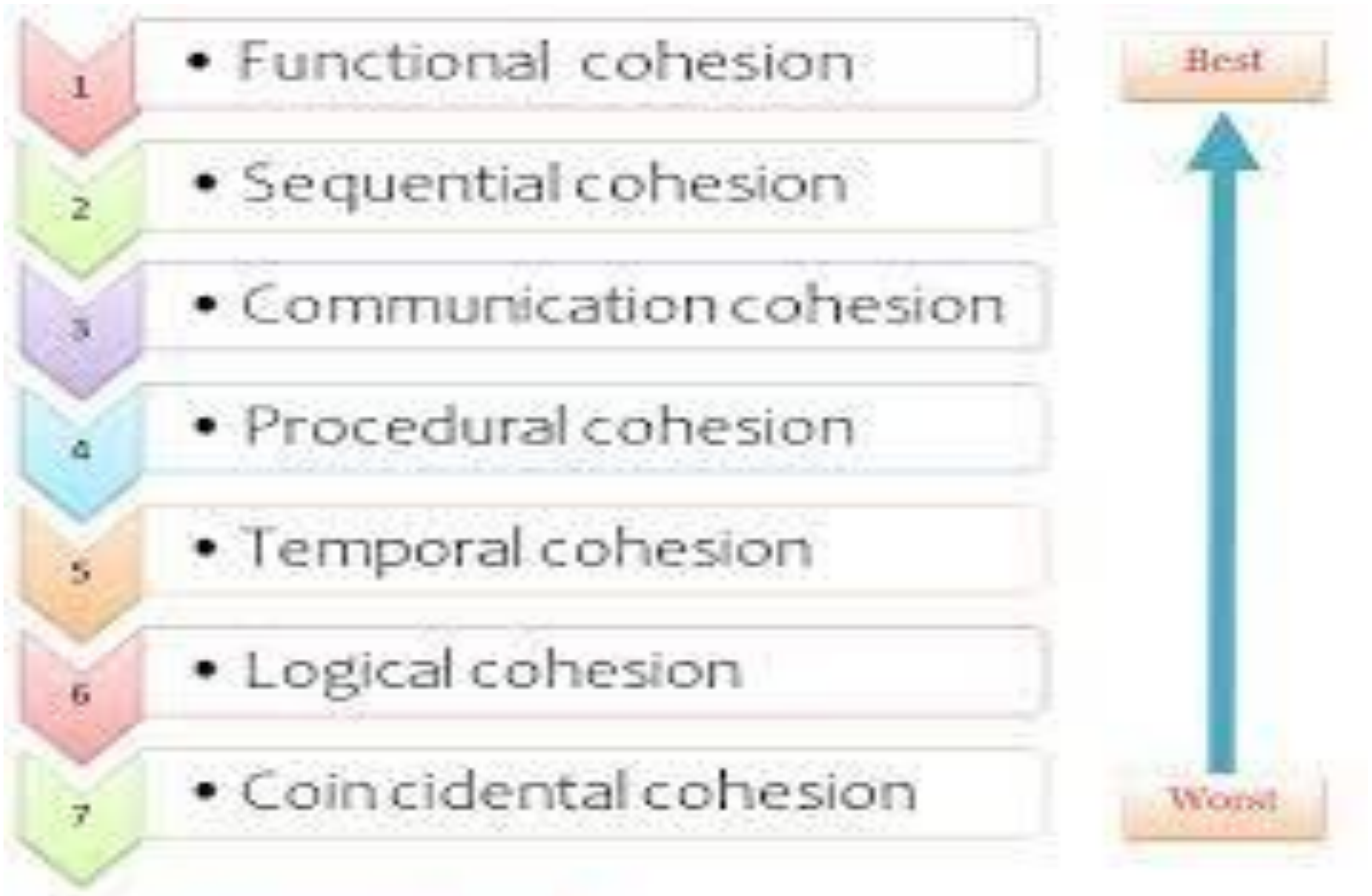


Figure 7 Types of Module cohesion

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.

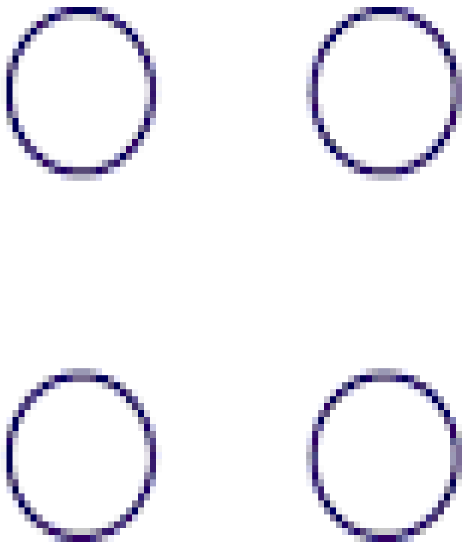
≫ Functional cohesion

- 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

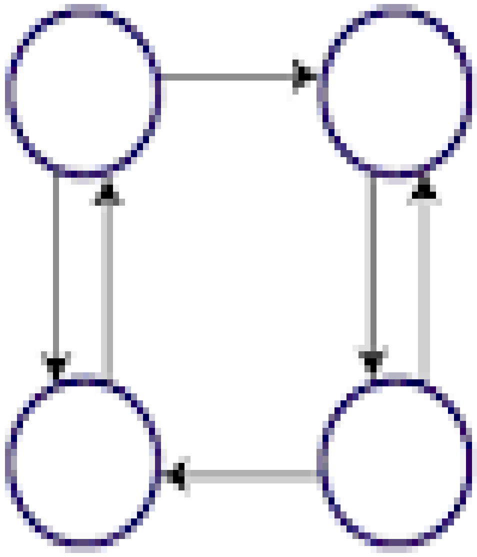
- **Coupling** between two modules is → a measure of the degree of interdependence or interaction between these two modules.
- Coupling refers to the number of connections between 'calling' and a 'called' module. There must be at least one connection between them.
- It refers to the strengths of relationship 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



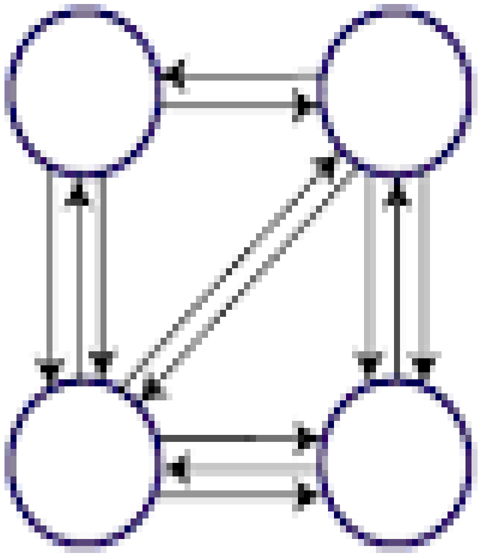
Uncoupled: no dependencies

(a)



Loosely Coupled: Some dependencies

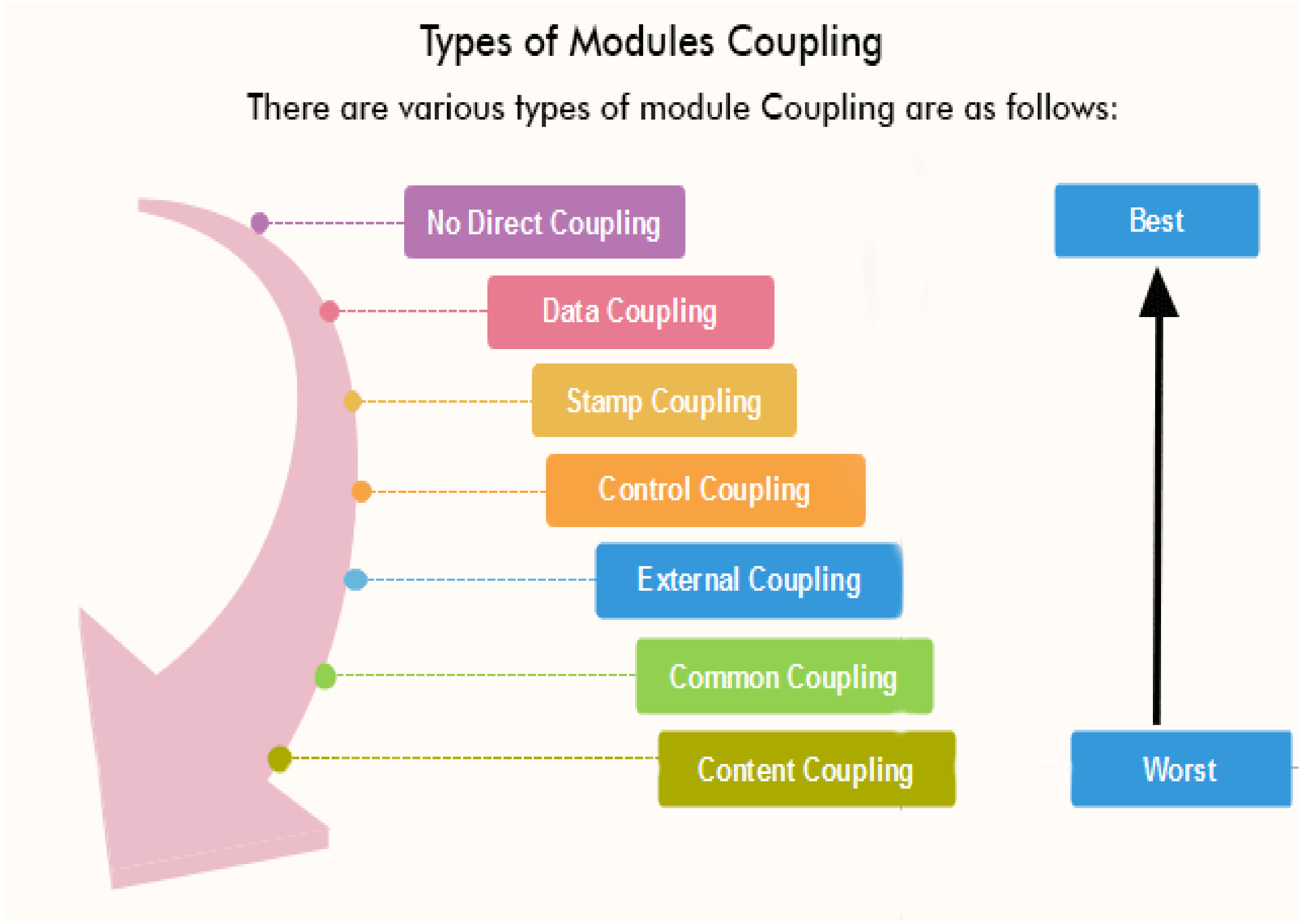
(b)



Highly Coupled: Many dependencies

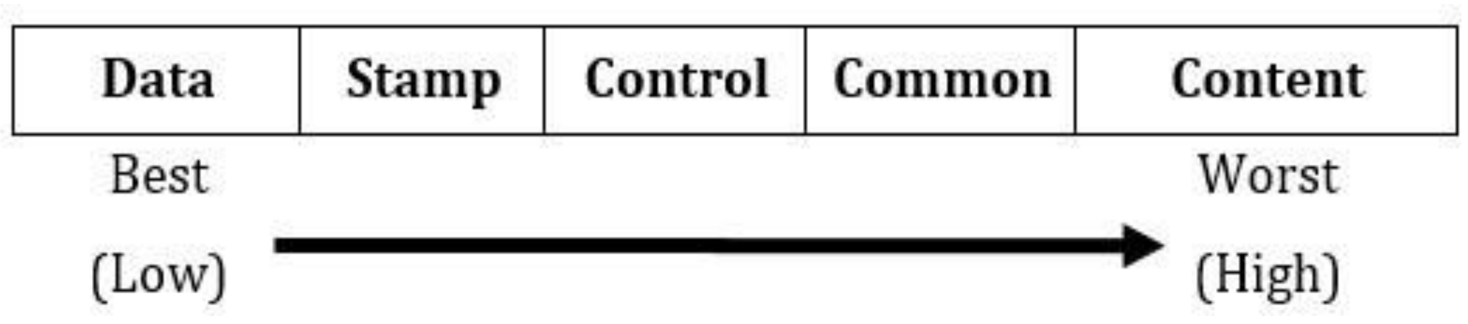
(c)

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.
- In online shopping system the shopping cart module shares data with the payment processing module.

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.
- ≫In inventory management system purchase and sales modules interact with product record module in stamp coupled manner.
- **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

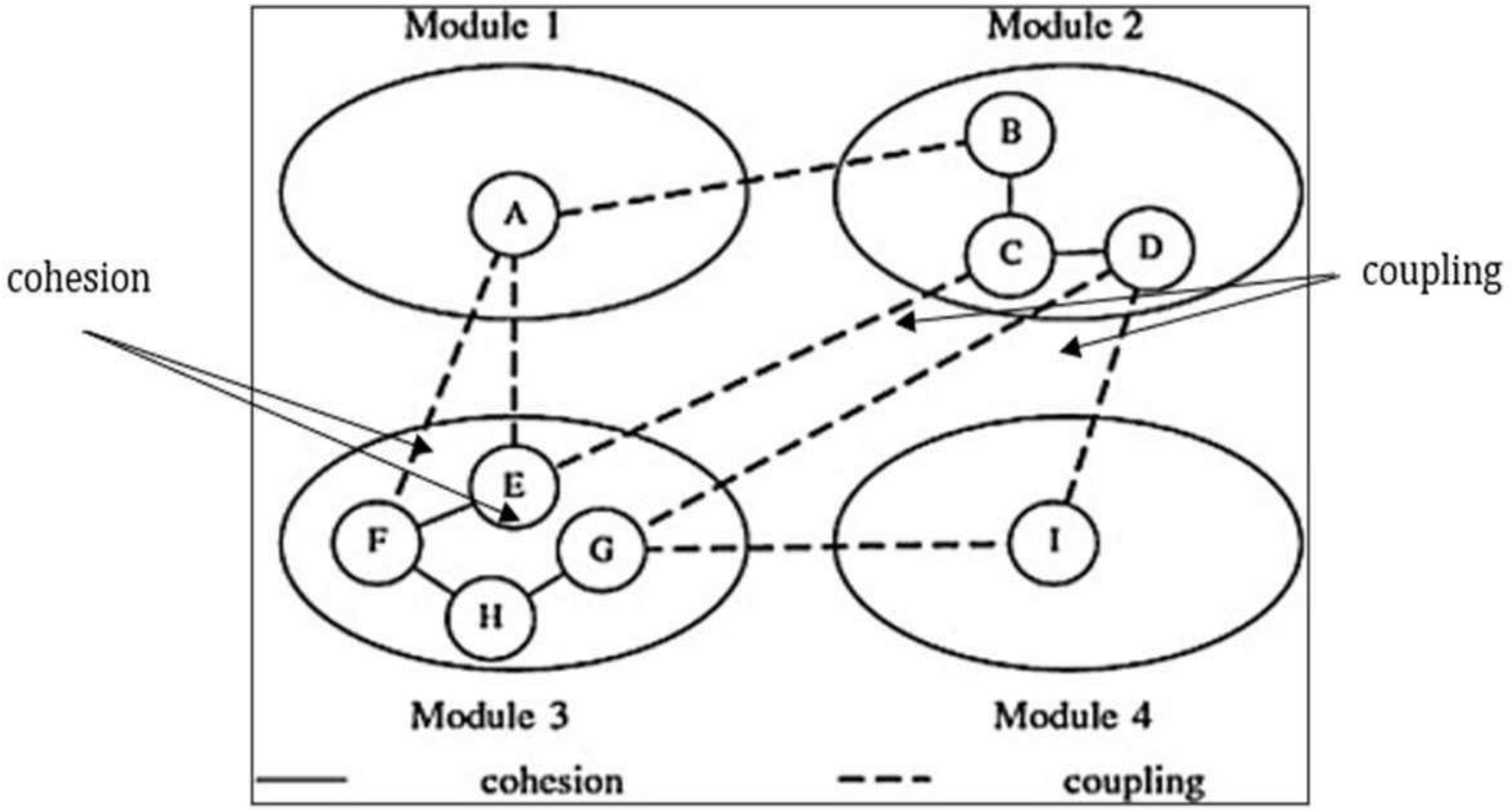
- It is the highest coupling and creates more problems in software development.
- It is worst coupling for any software development.
- This creates strong dependency between modules. And increase the complexity of the modules.

- 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**'.
- For example one module performing mathematical calculation and another module for formatting and displaying the result of that calculation . So formatting module requires knowledge of the internal data structures of the calculation module.

Cohesion and 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

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.

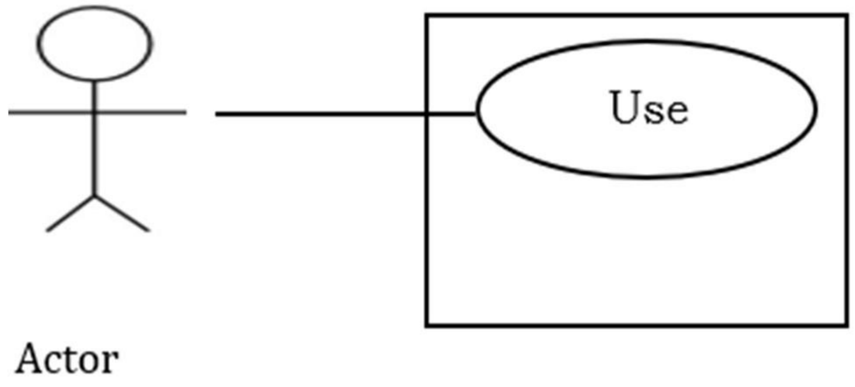
1. Use case

- 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

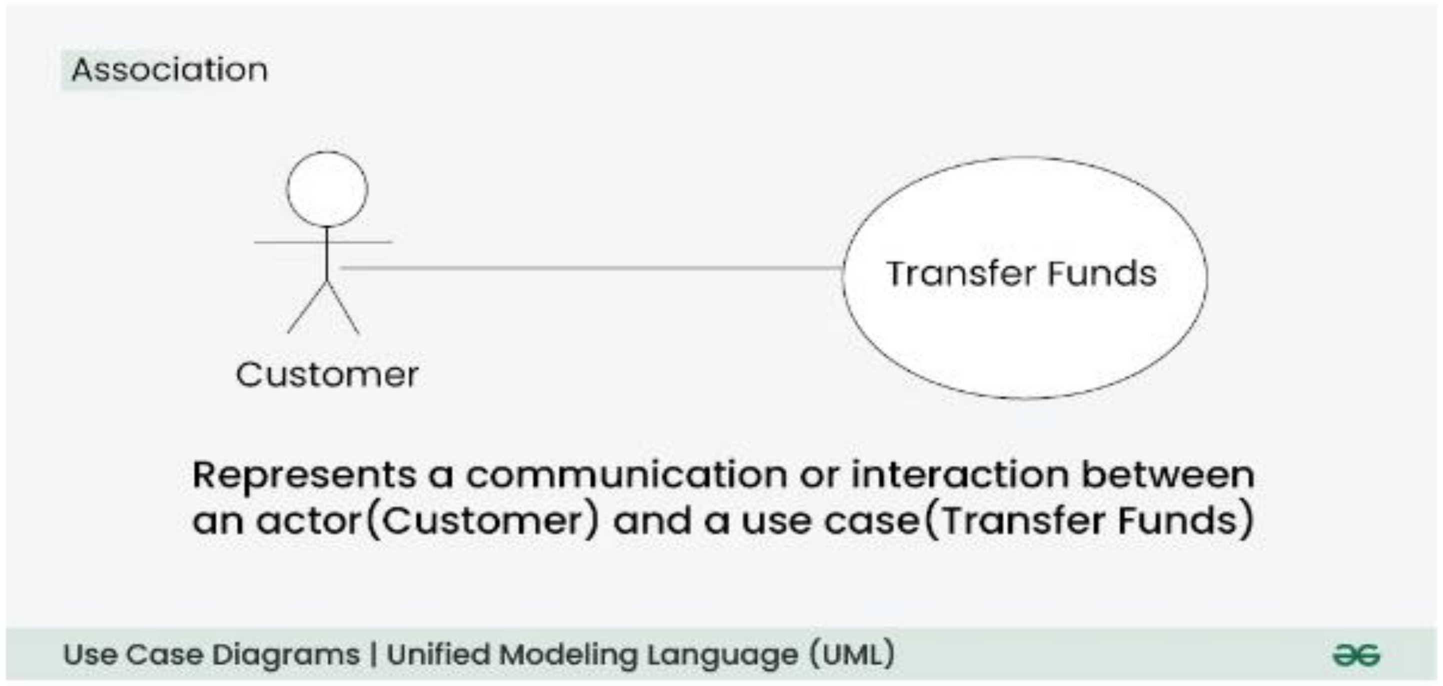
2. Actor

- 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.*



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.


Association
Customer
Transfer Funds
Represents a communication or interaction between an actor(Customer) and a use case(Transfer Funds)
Use Case Diagrams | Unified Modeling Language (UML)

ionship

2.7 Use case Diagram

Different relationships in use case diagram

➢ **Association**

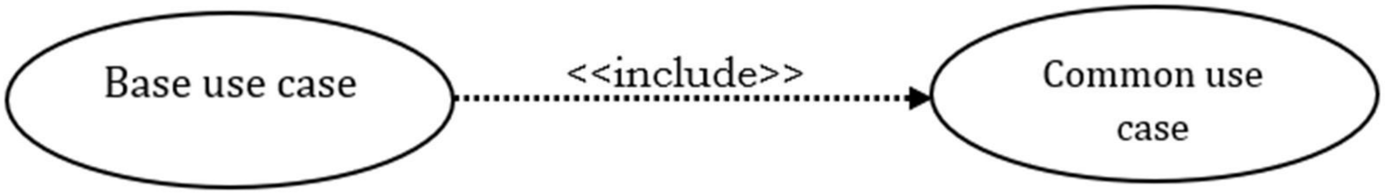
- Interface between an actor and a use case.
- Represented by joining a line from actor to use case.

➢ **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.

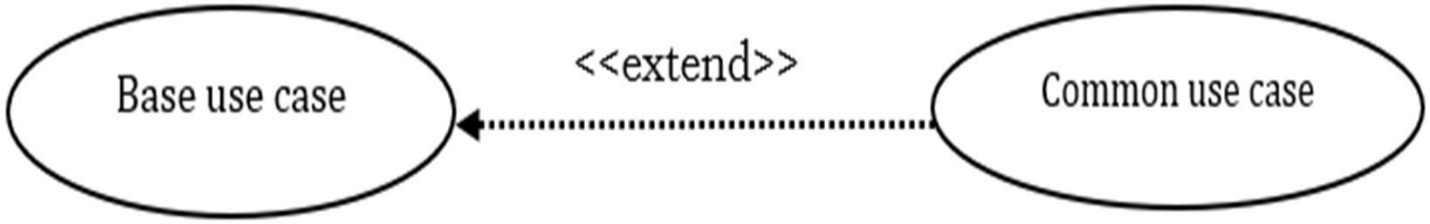
Include relationship: The use case is mandatory(compulsory) and part of the base use case. The include relationship is intended for reusing behavior modeled by another use case .

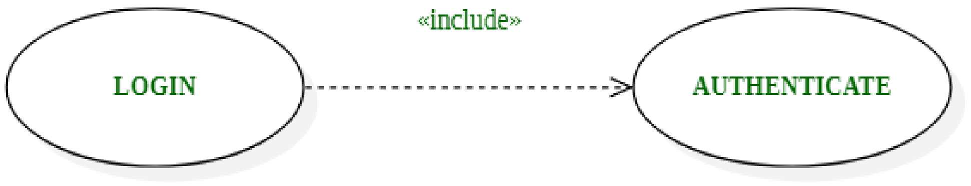
It is represented by a dashed arrow in the direction of the included use case with the notation **<<include>>**.



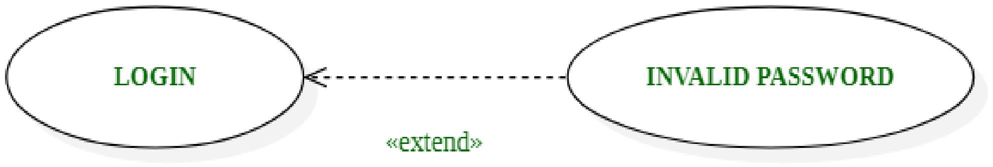
Extend relationship: The use case is optional and comes after the base use case. It is represented by a dashed arrow in the direction of the base use case with the notation <<extend>> .

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





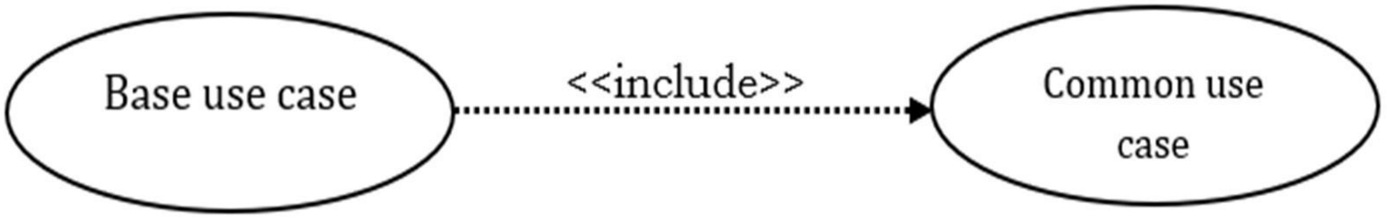
LOGIN is the BASE USE CASE and AUTHENTICATE is the INCLUDED USE CASE.

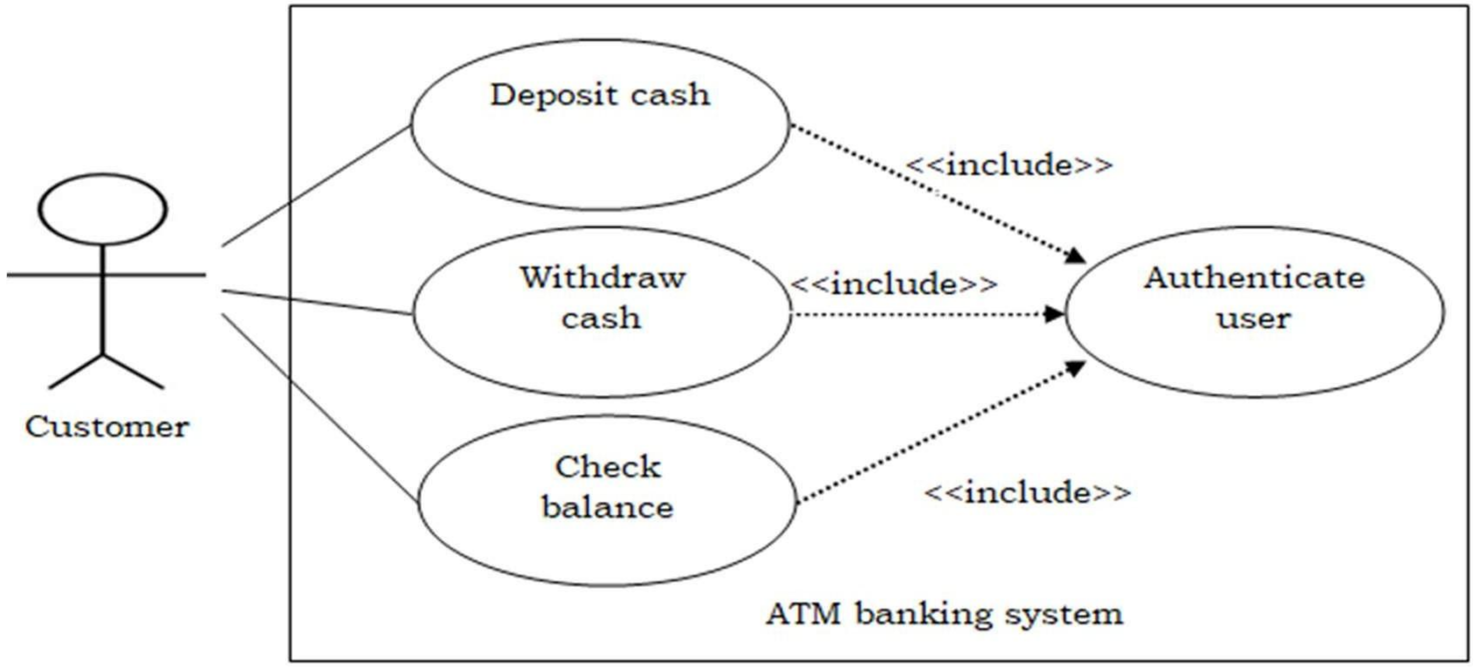


ILOGIN is the BASE USE CASE and INVALID PASSWORD is the EXTENDED USE CASE.

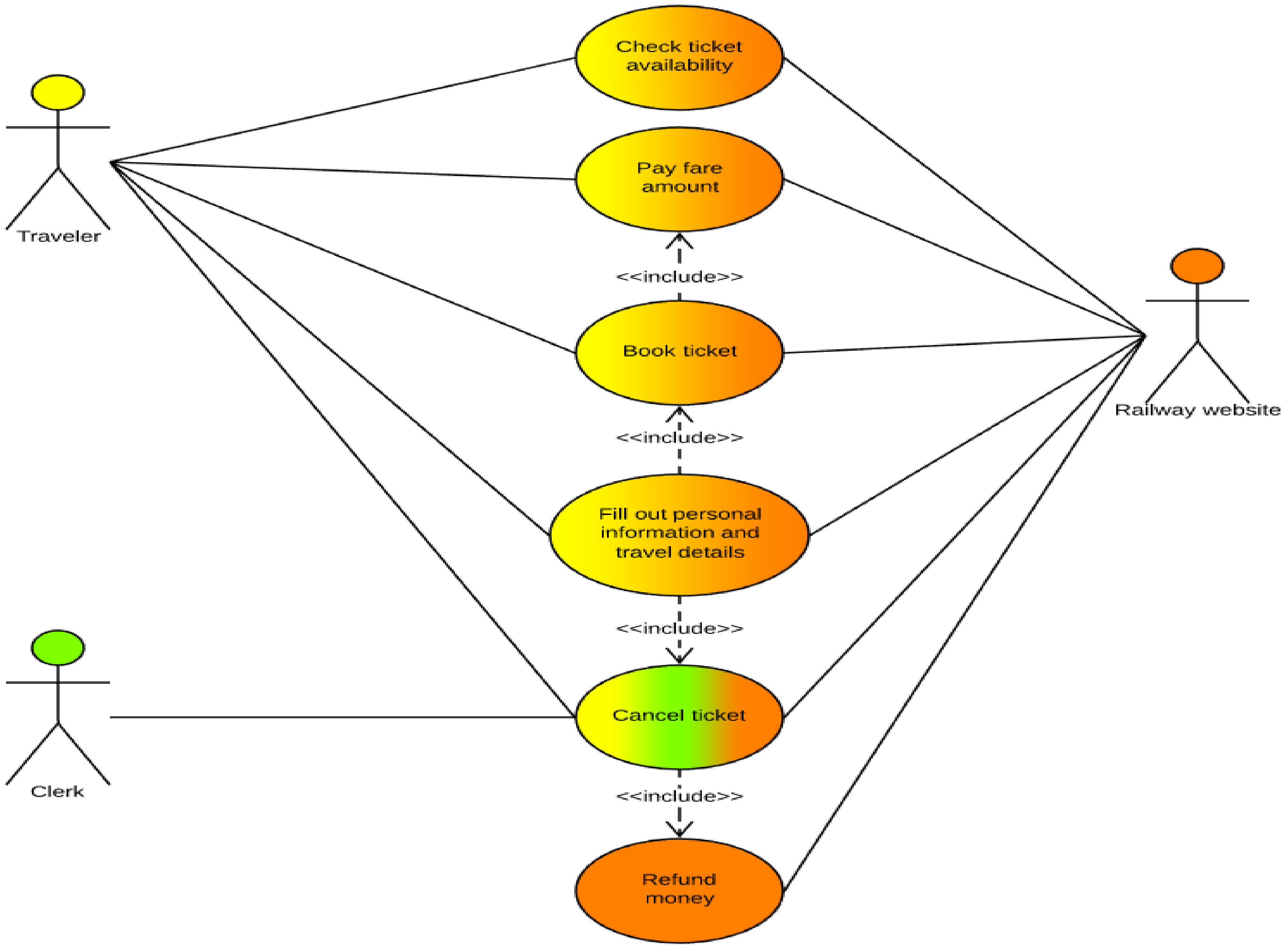
2.7 Use case Diagram

- **Include relationship**





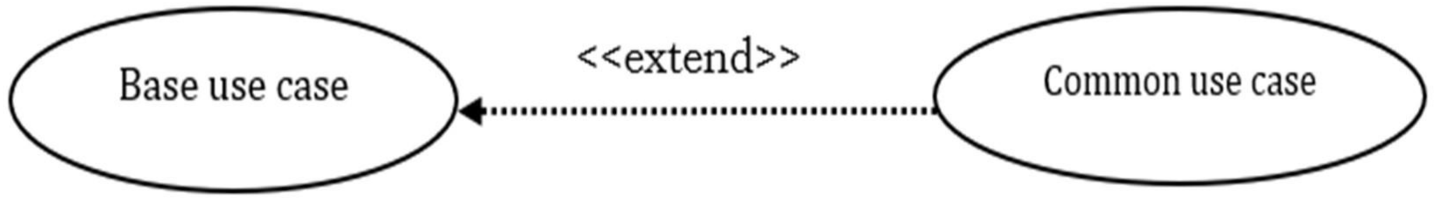
Railway reservation use case diagram example

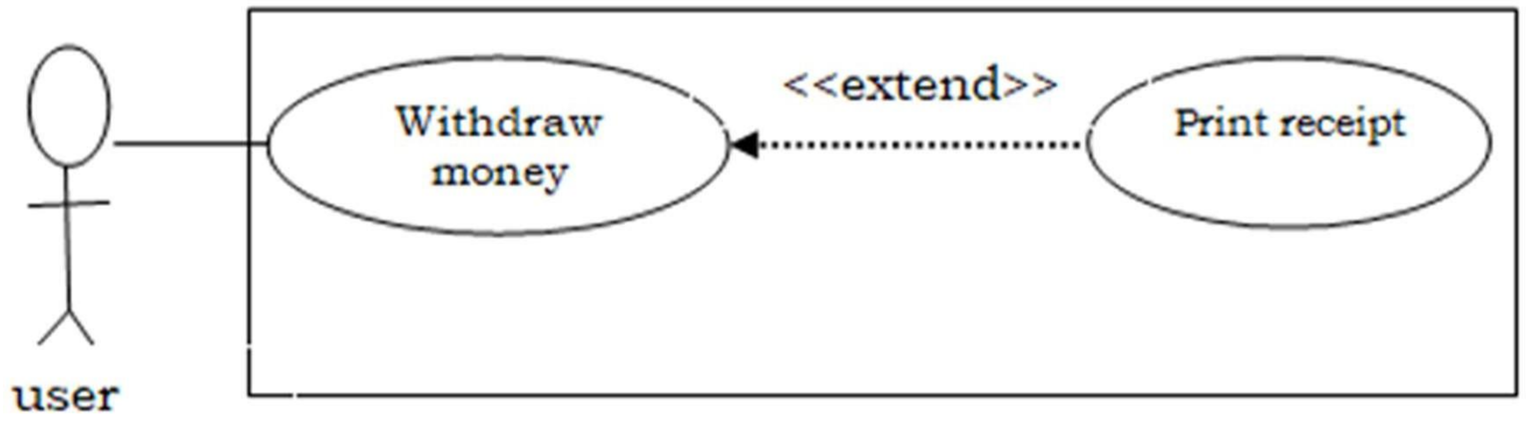


2.7 Use case Diagram

➢ 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).

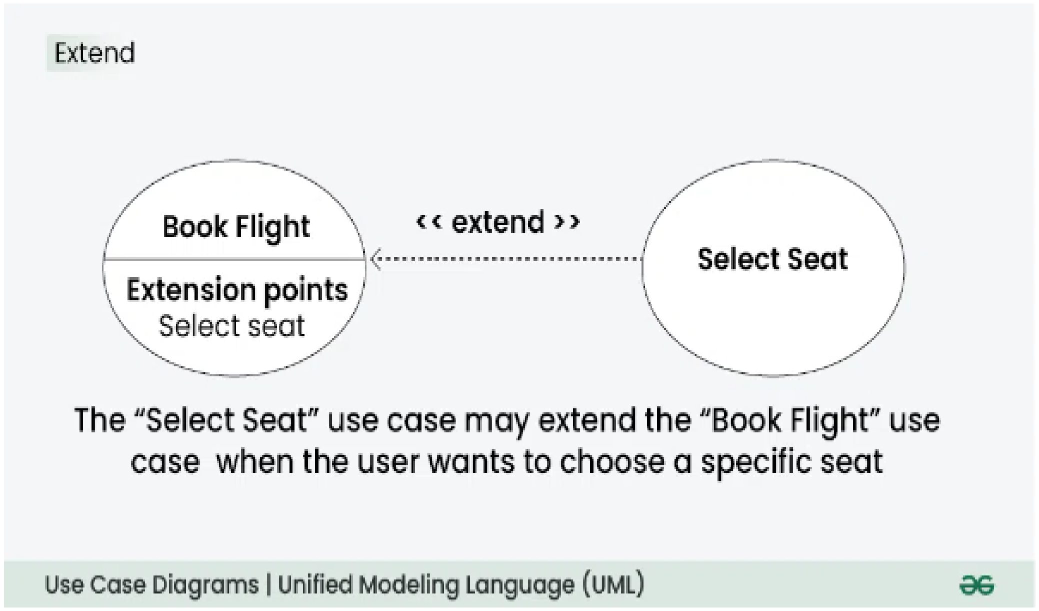




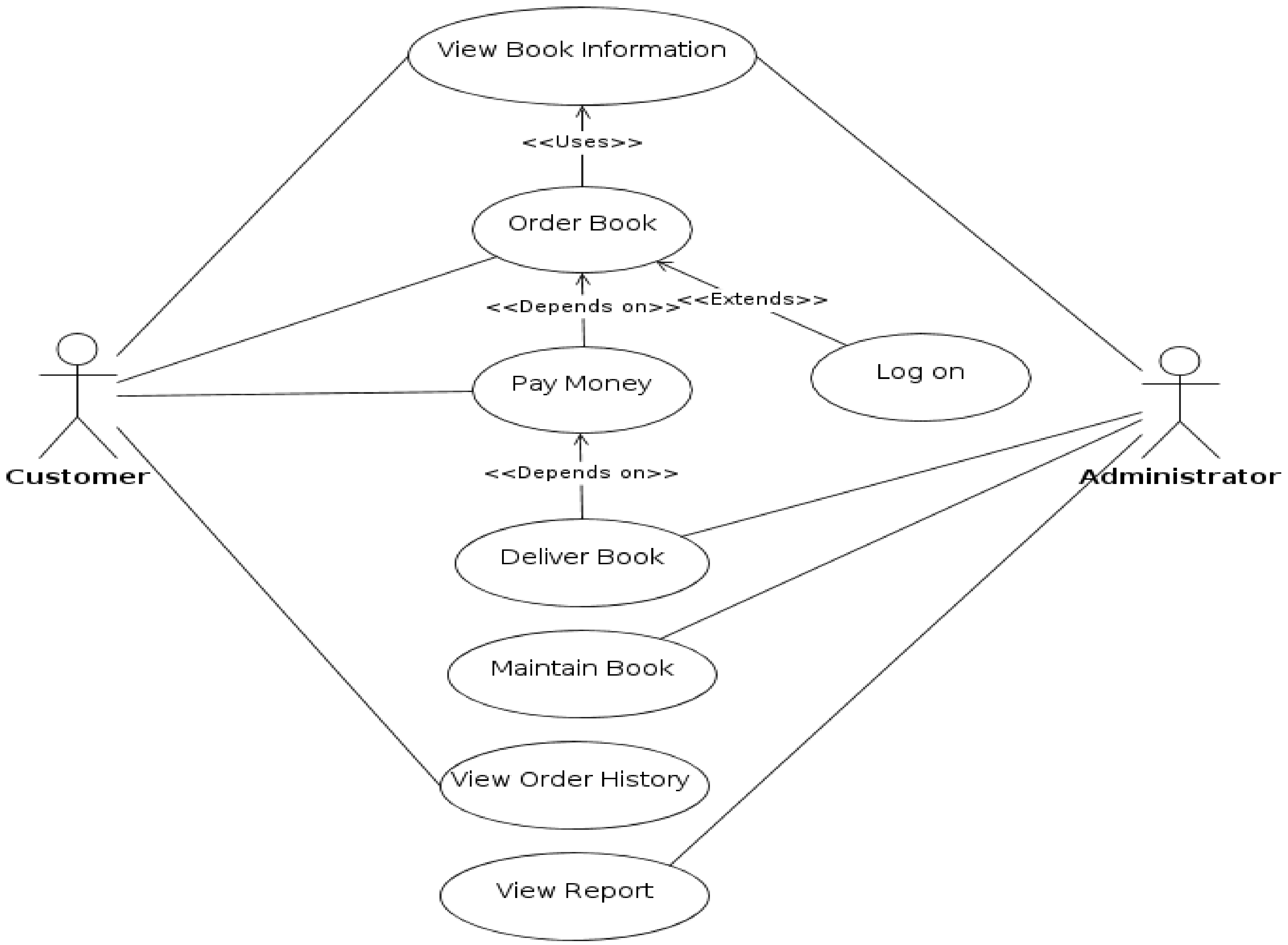
Extend relationship

Example: Flight Booking System

- **Use Cases:** Book Flight, Select Seat
- **Extend Relationship:** The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat, but it is an optional step.


Extend
Book Flight
Extension points
Select seat
<< extend >>
Select Seat
The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat
Use Case Diagrams | Unified Modeling Language (UML)

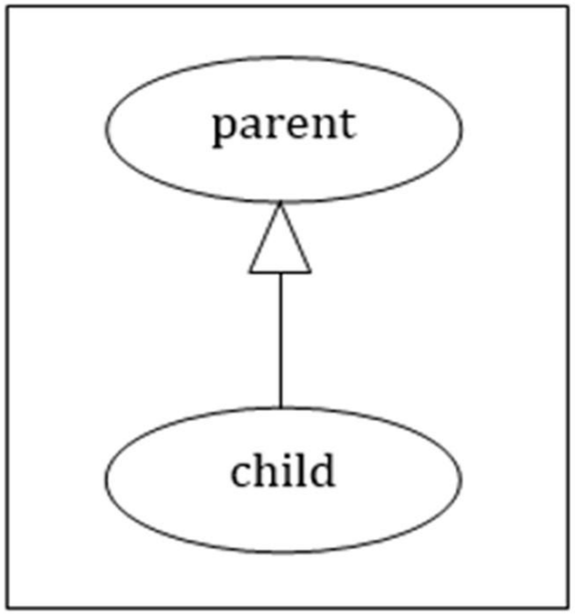
Use case Diagram

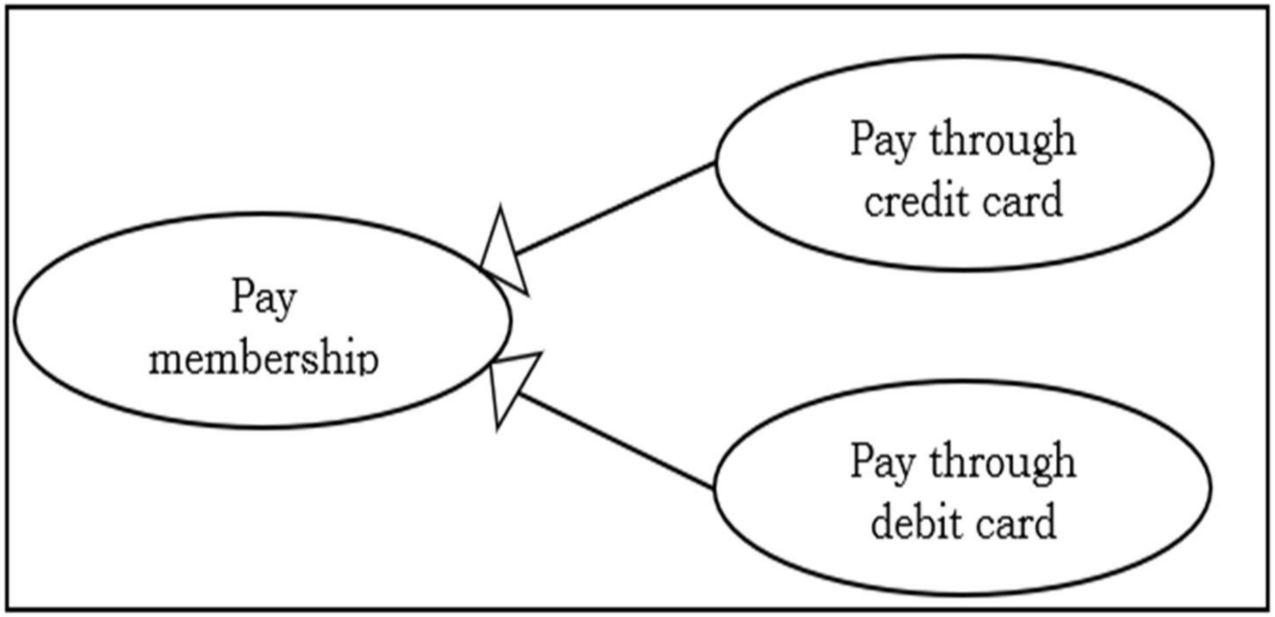


2.7 Use case Diagram

➢ 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

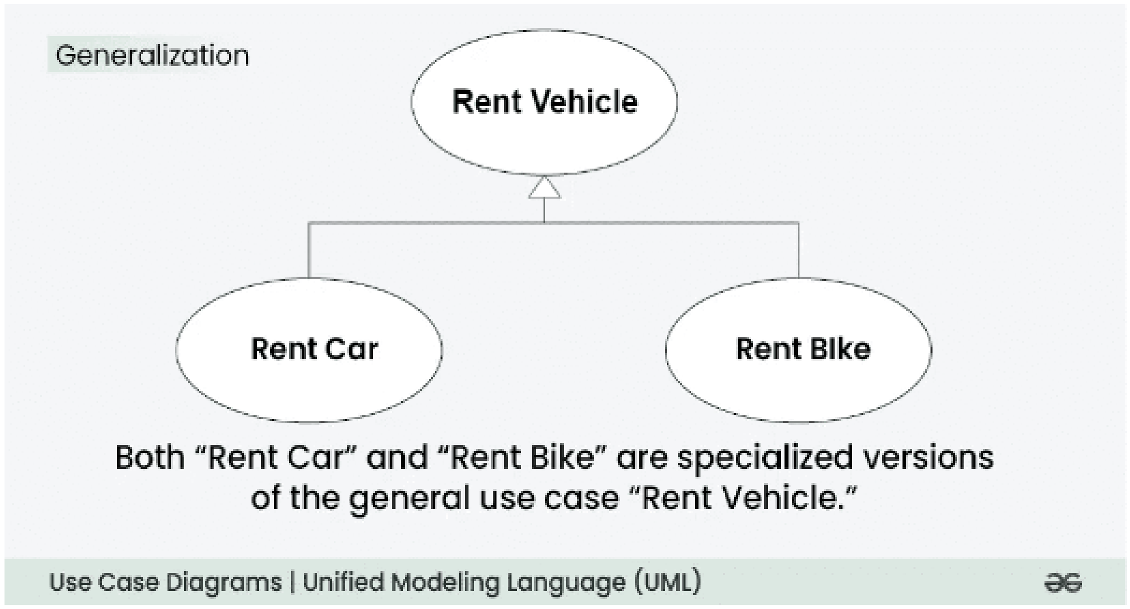




Generalization

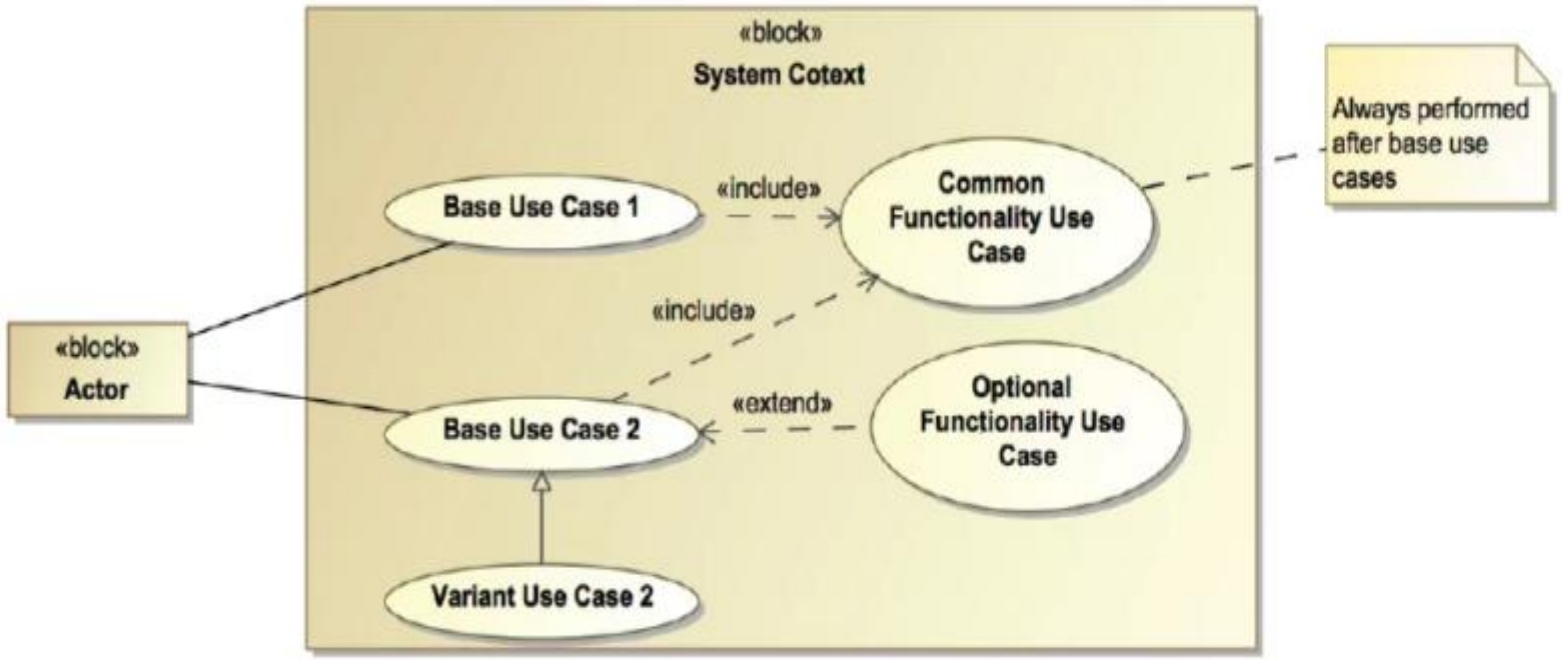
Example: Vehicle Rental System

- **Use Cases:** Rent Car, Rent Bike
- **Generalization Relationship:** Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."



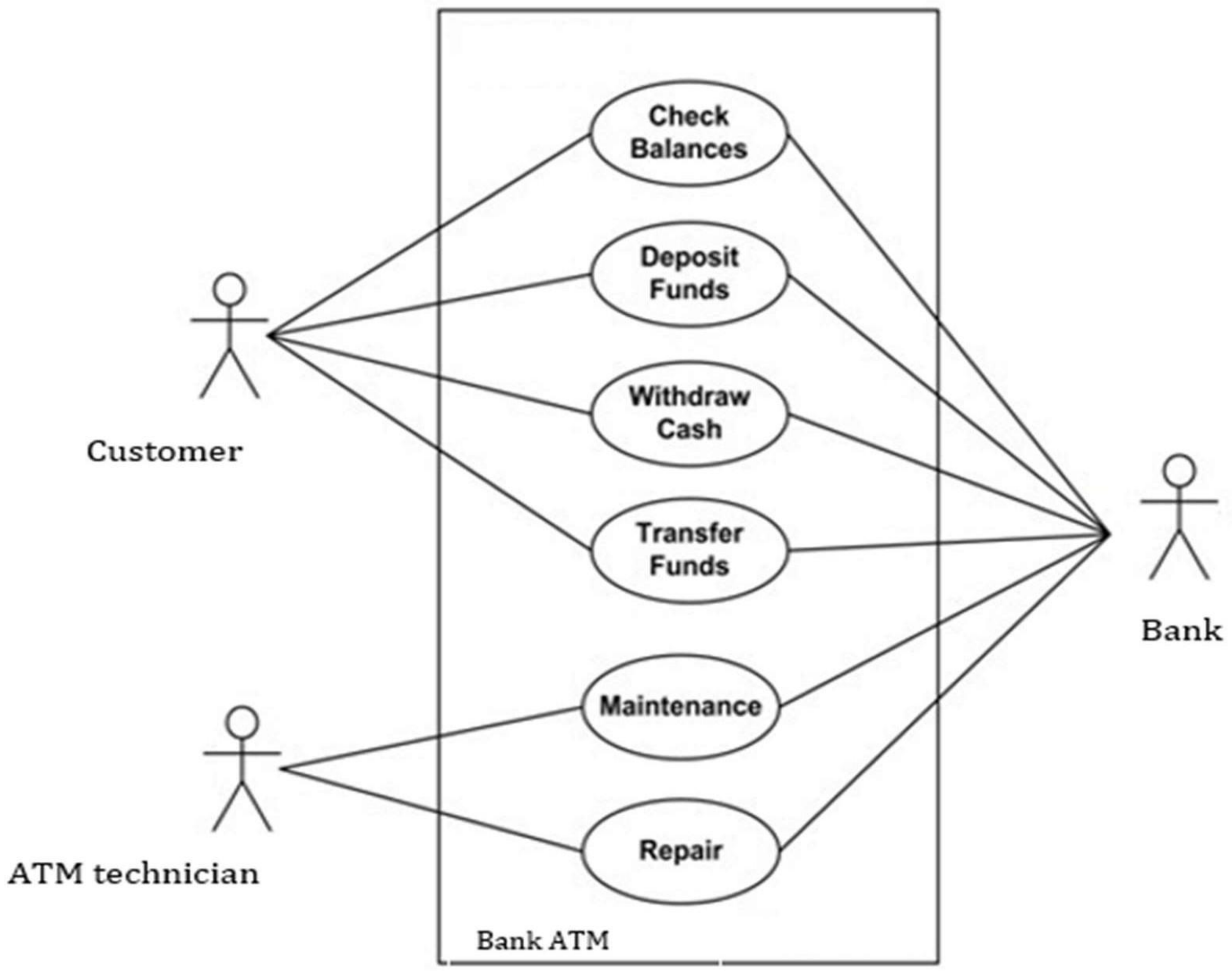
Both “Rent Car” and “Rent Bike” are specialized versions of the general use case “Rent Vehicle.”

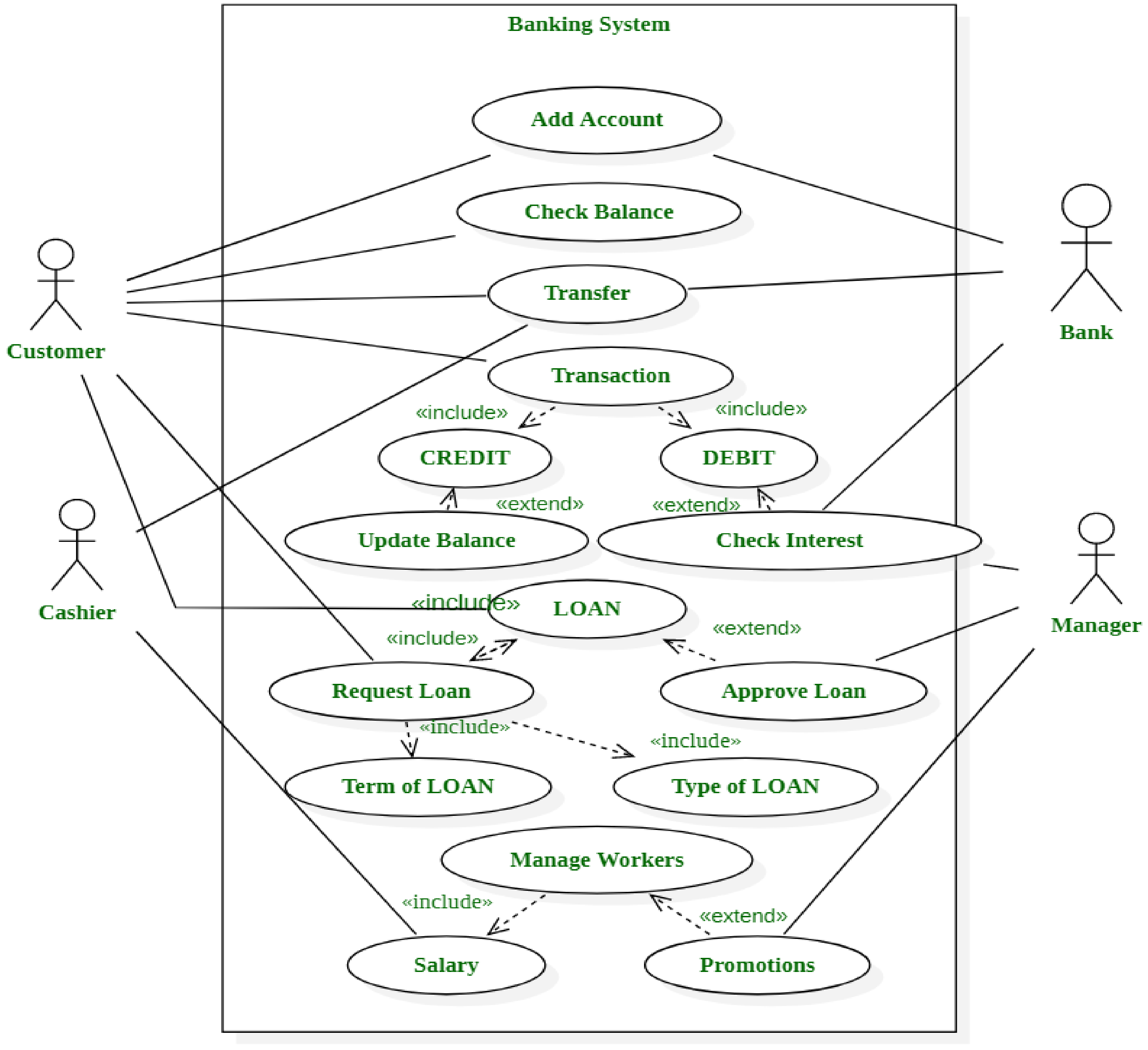
Generalization Relationship


«block»
System Cotext
Base Use Case 1
«include»
Common Functionality Use Case
Always performed after base use cases
«include»
«block»
Actor
Base Use Case 2
«extend»
Optional Functionality Use Case
Variant Use Case 2

2.7 Use case Diagram

- **Example (ATM System)**




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

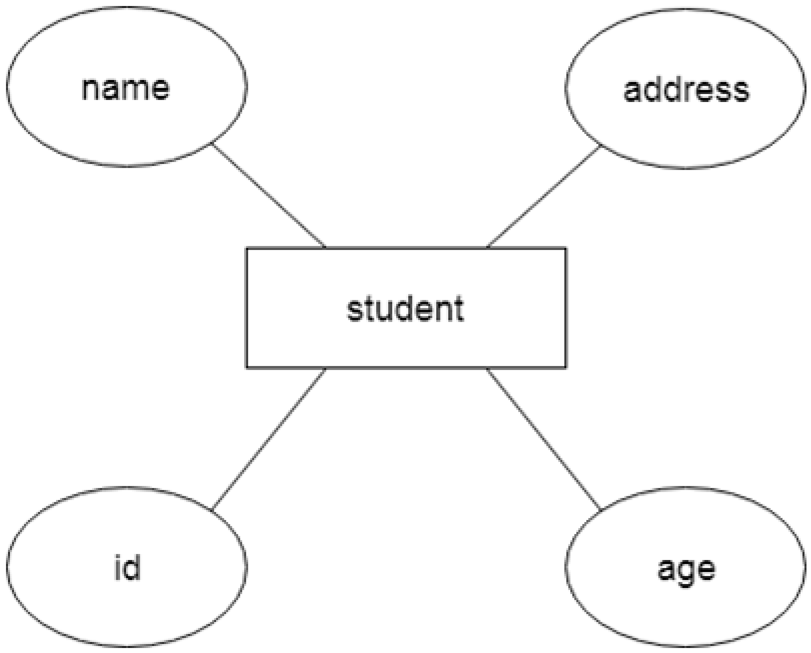
2.7 Use case Diagram

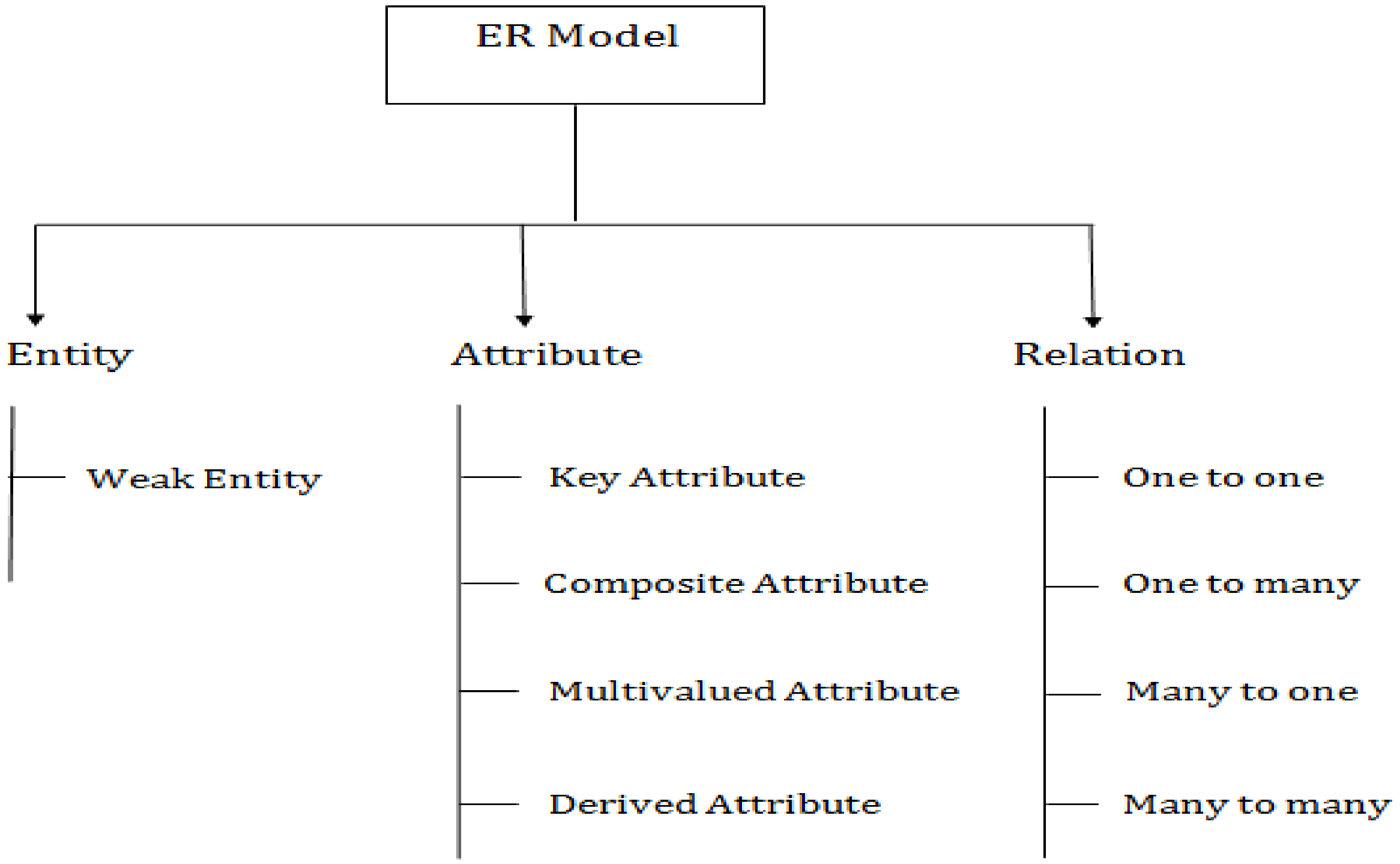
☐ Applications of use case diagram

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

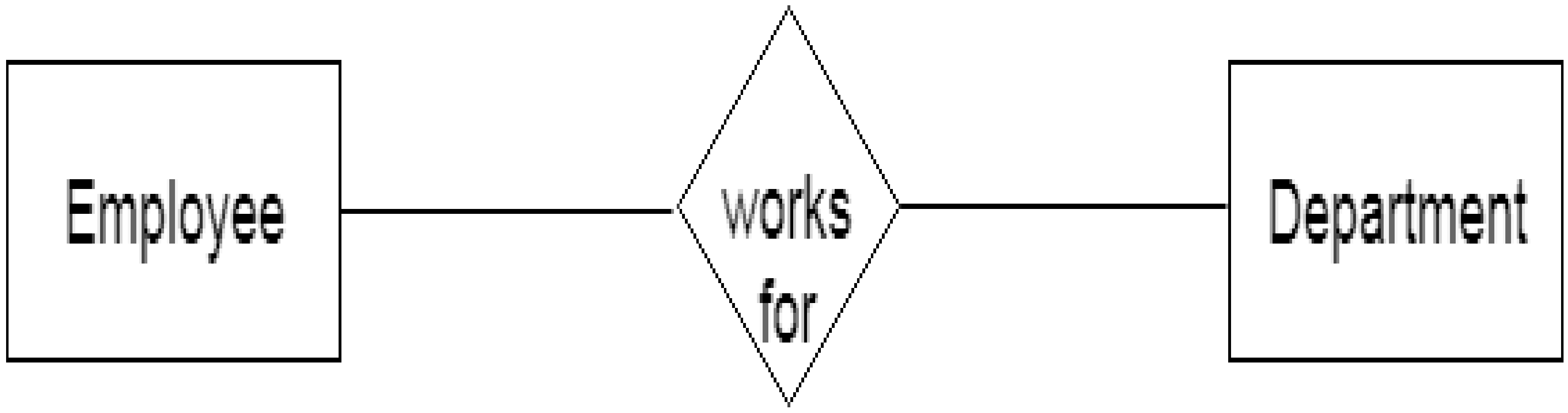
ER (Entity Relationship) Diagram

- ER model stands for an Entity-Relationship model. It is a high-level data model.
- This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc.




ER Model
Entity
Weak Entity
Attribute
Key Attribute
Composite Attribute
Multivalued Attribute
Derived Attribute
Relation
One to one
One to many
Many to one
Many to many

1. Entity:
An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.
Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.

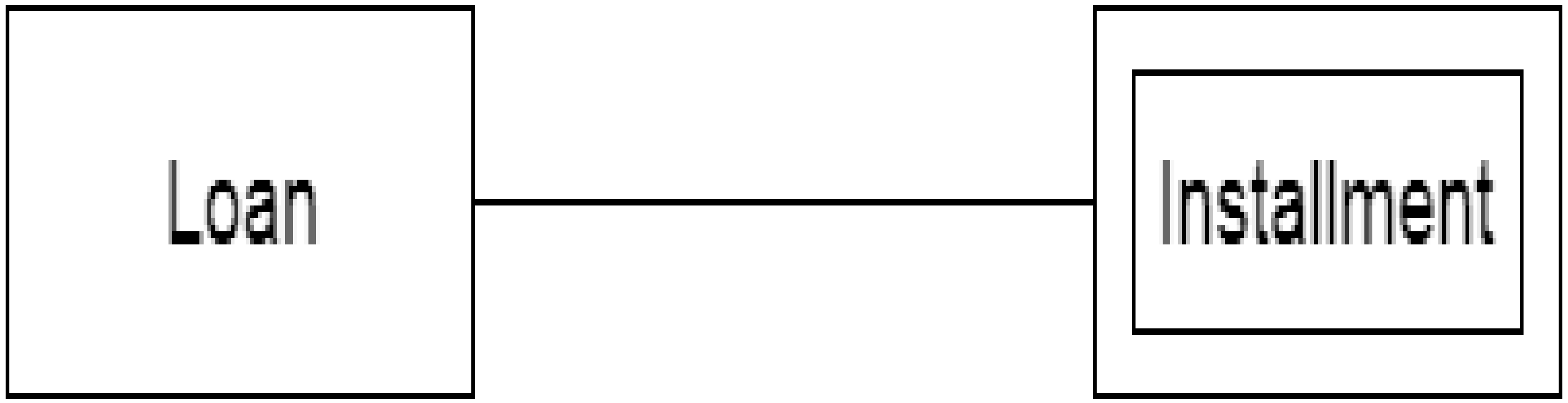


Weak Entity

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.

Strong Entity

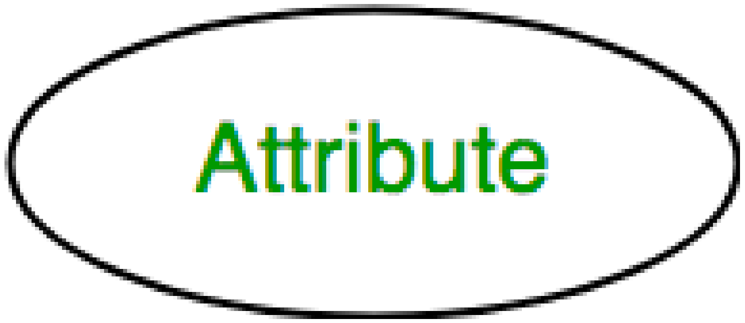
A Strong Entity is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.



2. Attribute

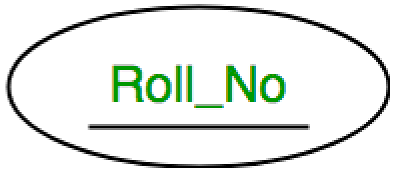
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.



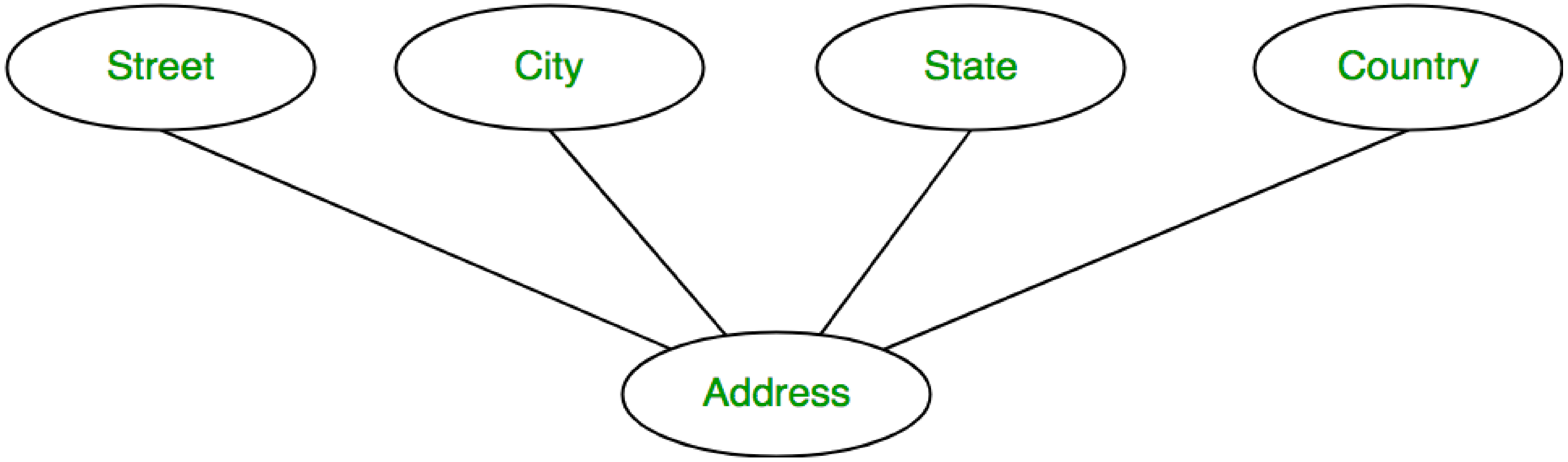
Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



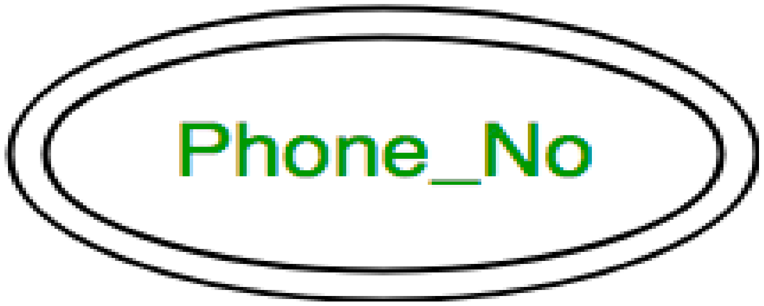
Composite Attribute

An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.



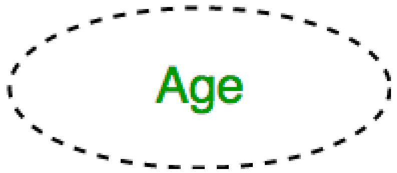
Multivalued Attribute

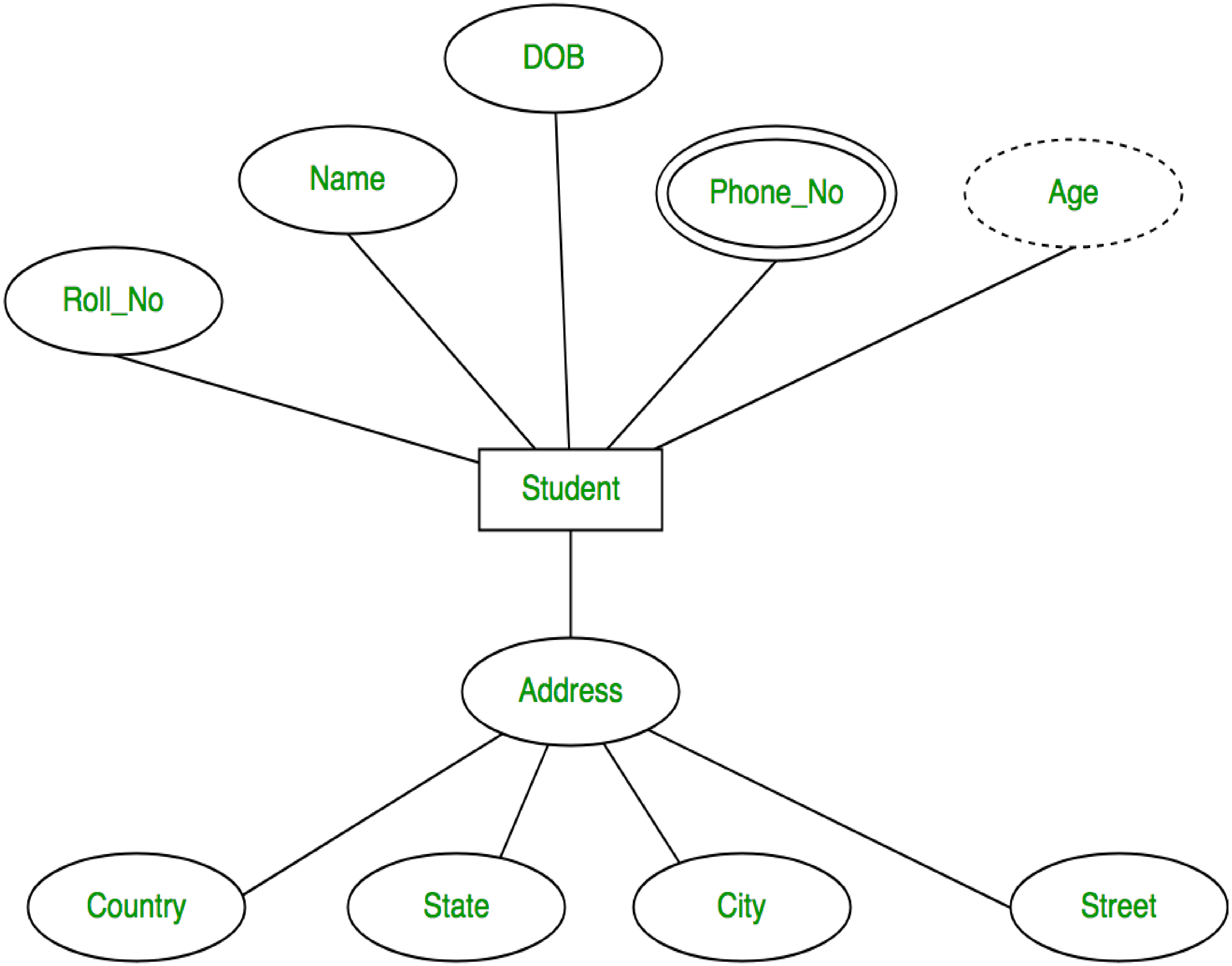
An attribute consisting of more than one value for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.



Derived Attribute

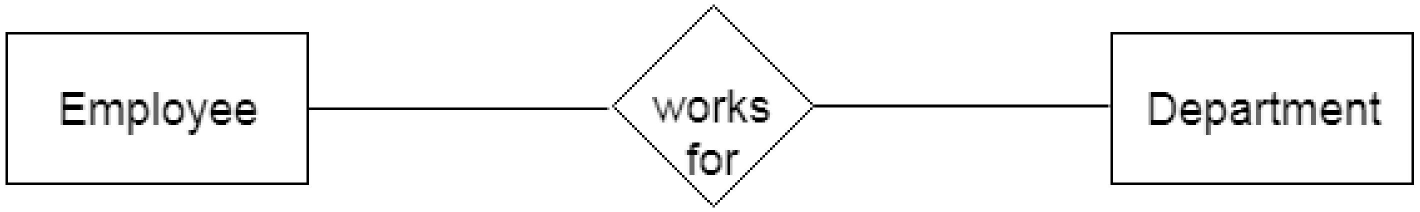
An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.

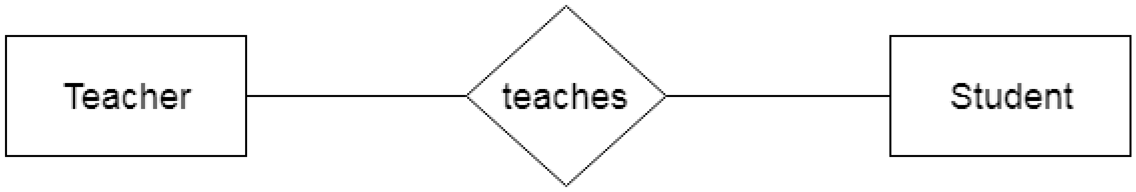



DOB
Name
Phone_No
Age
Roll_No
Student
Address
Country
State
City
Street

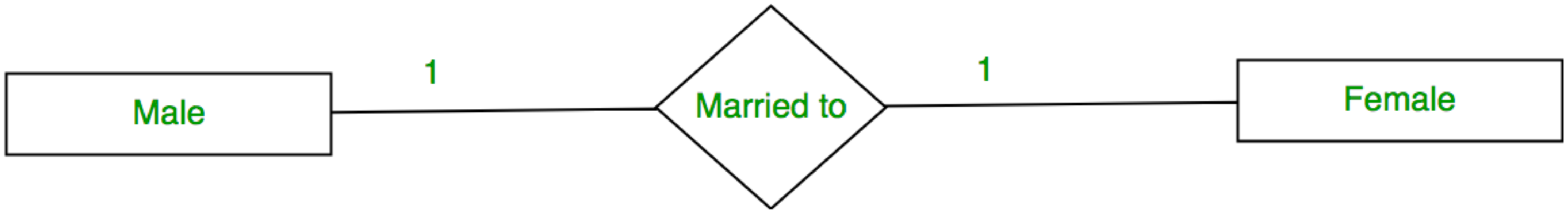
3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

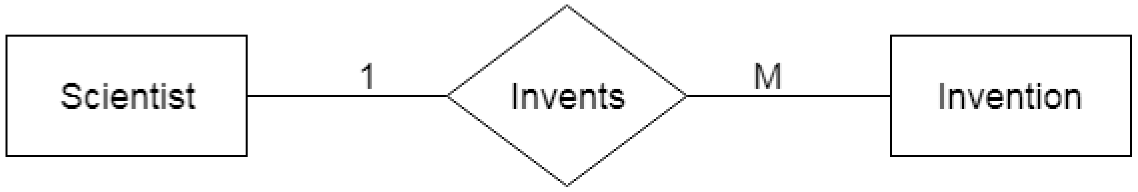




One-to-One Relationship:- When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one.

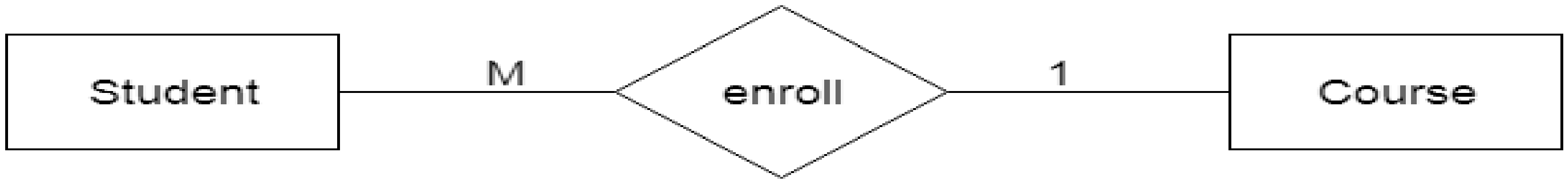


One-to-many relationship



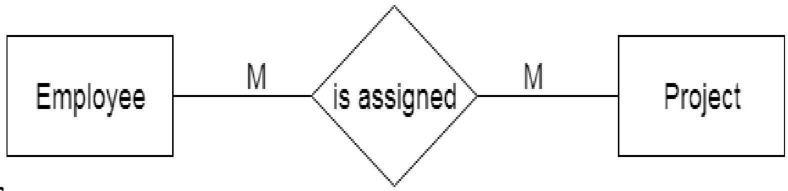
Many-to-one relationship

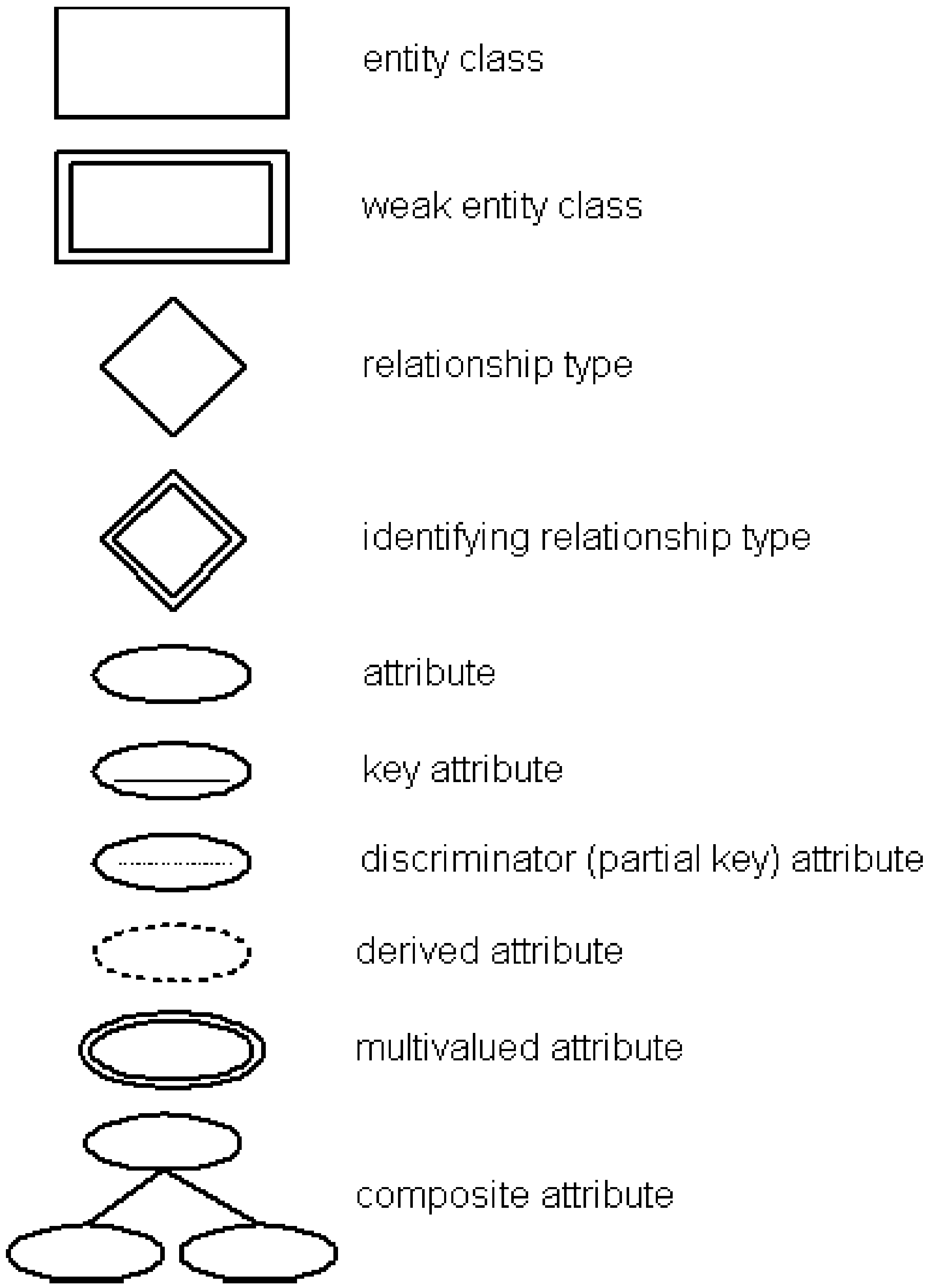
When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.
For example, Student enrolls for only one course, but a course can have many students.



Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.
For example, Employee can assign by many projects and project can have many employees.




entity class
weak entity class
relationship type
identifying relationship type
attribute
key attribute
discriminator (partial key) attribute
derived attribute
multivalued attribute
composite attribute

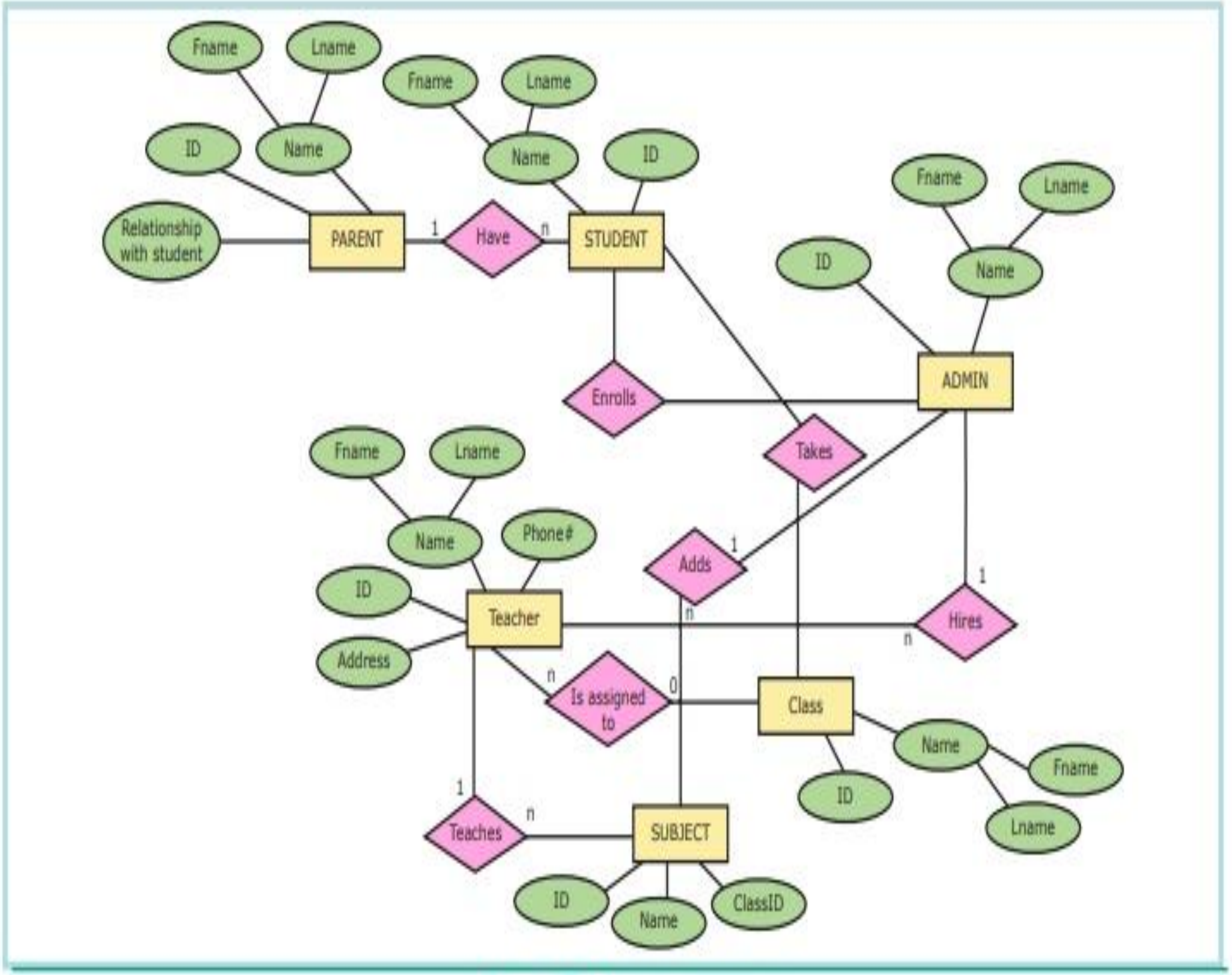
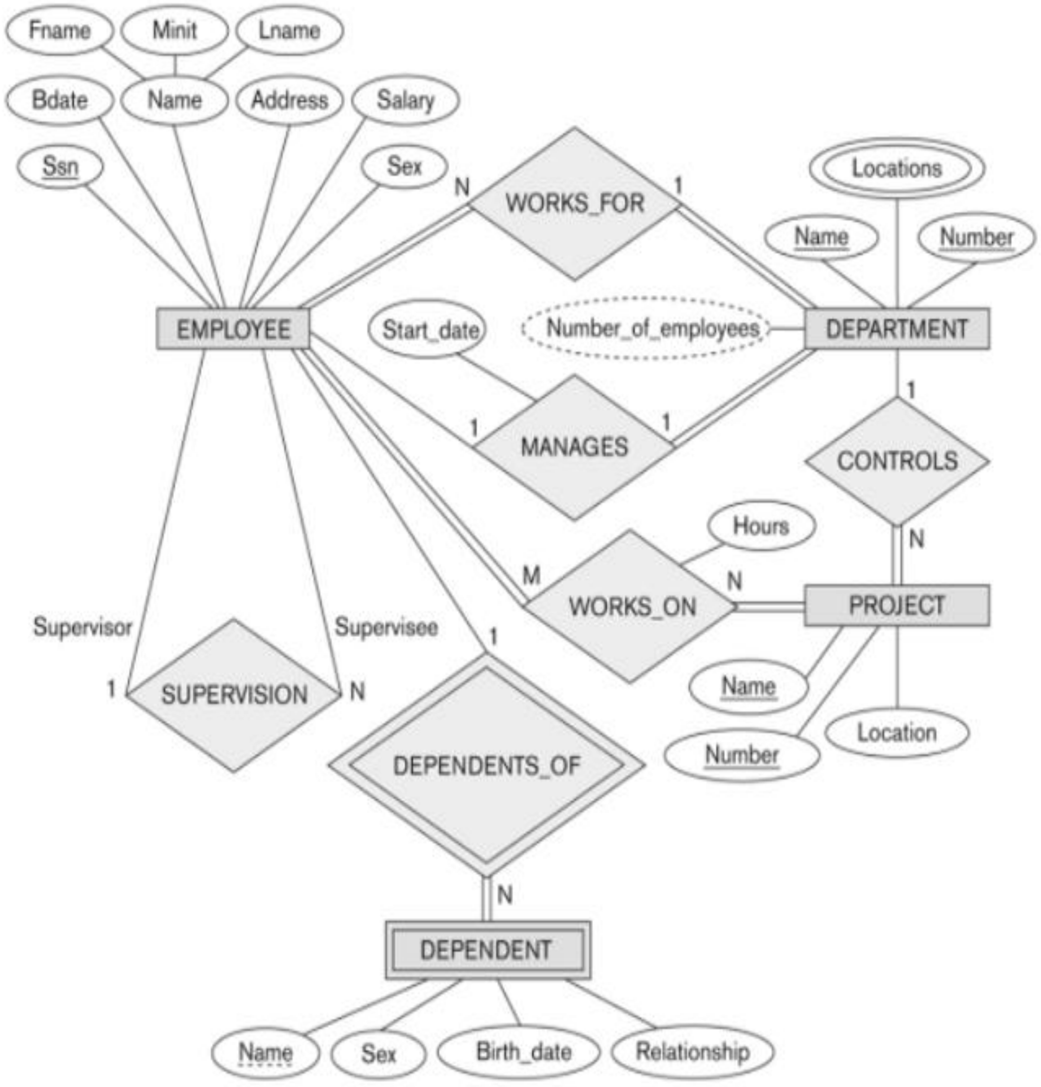


Figure: 3.12 ER-Diagram


Fname
Minit
Lname
Bdate
Name
Address
Salary
Ssn
Sex
N
WORKS_FOR
1
Locations
Name
Number
EMPLOYEE
Start_date
Number_of_employees
DEPARTMENT
1
MANAGES
1
1
CONTROLS
Hours
N
M
N
WORKS_ON
PROJECT
Supervisor
Supervisee
1
1
SUPERVISION
N
Name
Location
Number
DEPENDENTS_OF
N
DEPENDENT
Name
Sex
Birth_date
Relationship

ThankYou...

Prepared by:

Jemi M. Pavagadhi

Lecturer, Computer Dept. G. P. Porbandar