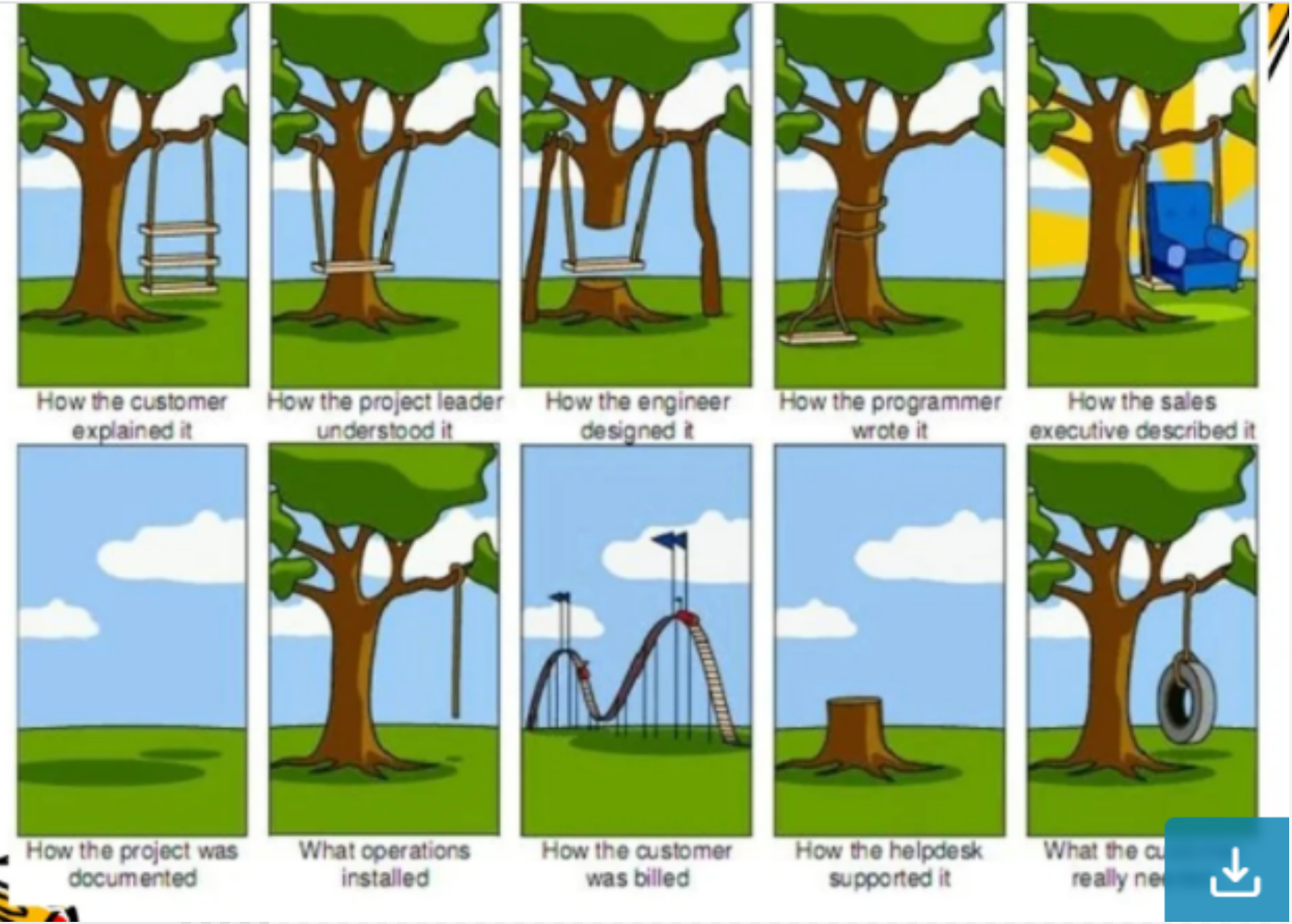
UNIT 2

- **Understanding Requirements**


How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it

Requirement Engineering

Broad spectrum of task and techniques that lead to an understanding of requirements is called ***Requirement engineering***

Builds a bridge to design and construction.

Requirements Engineering-I

- Inception
- Elicitation—elicit requirements from all stakeholders
- Elaboration—create an analysis model that identifies data, function and behavioral requirements.
- Negotiation—agree on a deliverable system that is realistic for developers and customers.

Requirements Engineering-II

- Specification—can be any one (or more) of the following:
  - A written document
  - A set of models
  - A formal mathematical model
  - A collection of user scenarios (use-cases)
  - A prototype
- Validation—a review mechanism that looks for
  - errors in content or interpretation
  - areas where clarification may be required
  - missing information
  - inconsistencies (a major problem when large products or systems are engineered)
  - conflicting or unrealistic (unachievable) requirements.
- Requirements management

Inception

- Ask a set of questions that establish …
  - Basic understanding of the problem
  - People who want a solution
  - Nature of the solution that is desired
  - Effectiveness of preliminary communication and collaboration between the customer and the developer

- Recognize multiple points of view
- Work toward collaboration
- The first questions
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution?
  - Is there another source for the solution that you need?

Eliciting Requirements

- Meetings are conducted and attended by both software engineers and customers
- Rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- the goal is
  - to identify the objective
  - propose elements of the solution
  - specify a preliminary set of solution requirements

- **Problem of Scope:** The requirements given are of unnecessary detail, ill-defined, or not possible to implement.

- **Problem of Understanding:** Not having a clear-cut understanding between the developer and customer when putting out the requirements needed. Sometimes the customer might not know what they want or the developer might misunderstand one requirement for another.

- **Problem of Volatility:** Requirements changing over time can cause difficulty in leading a project. It can lead to loss and wastage of resources and time.

Elicitation Work Products

- a statement of need and feasibility.
- a bounded statement of scope for the system or product.
- a list of customers, users, and other stakeholders who participated in requirements elicitation
- a description of the system's technical environment.
- a list of requirements (preferably organized by function) and the domain constraints that apply to each.
- a set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- any prototypes developed to better define requirements.

Elaboration

- This is the third phase of the requirements analysis process.
- This phase is the result of the inception and elicitation phase.
- In the elaboration process, it takes the requirements that have been stated and gathered in the first two phases and refines them. Expansion and looking into it further are done as well.
- The main task in this phase is to indulge in modeling activities and develop a prototype that elaborates on the features and constraints using the necessary tools and functions.

Negotiation

- This is the fourth phase of the requirements analysis process.
- This phase emphasizes discussion and exchanging conversation on what is needed and what is to be eliminated.
- Negotiation is between the developer and the customer and they dwell on how to go about the project with limited business resources.

- Customers are asked to prioritize the requirements and make guesstimates on the conflicts that may arise along with it.
- Risks of all the requirements are taken into consideration and negotiated in a way where the customer and developer are both satisfied with reference to the further implementation.

The following are discussed in the negotiation phase:

- Availability of Resources.
- Delivery Time.
- Scope of requirements.
- Project Cost.
- Estimations on development.

Specification

This is the fifth phase of the requirements analysis process. This phase specifies the following:

- Written document.
- A set of models.
- A collection of use cases.
- A prototype.

- This final working product will be the basis of any functions, features or constraints to be observed.
- The models used in this phase include ER (Entity Relationship) diagrams, DFD (Data Flow Diagram), FDD (Function Decomposition Diagrams), and Data Dictionaries.
- A software specification document is submitted to the customer in a language that he/she will understand, to give a glimpse of the working model.

Validation

- This is the sixth phase of the requirements analysis process.
- This phase focuses on checking for errors and debugging.
- In the validation phase, the developer scans the specification document and checks for the following:
  1. All the requirements have been stated and met correctly
  2. Errors have been debugged and corrected.
  3. Work product is built according to the standards.

Requirements Management

- This is the last phase of the requirements analysis process.
- Requirements management is a set of activities where the entire team takes part in identifying, controlling, tracking, and establishing the requirements for the successful and smooth implementation of the project.

Software Requirement Specifications (SRS)

- SRS is a complete specification and description of requirements of the software that need to be fulfilled for the successful development of the software system.
- These requirements can be functional as well as non-functional depending upon the type of requirement.
- The interaction between different customers and contractors is done because it is necessary to fully understand the needs of customers.

Features of a good SRS document:

1. Correctness: User review is used to provide the accuracy of requirements stated in the SRS.

SRS is said to be perfect if it covers all the needs that are truly expected from the system.

2. Completeness: The SRS is complete if, and only if, it includes the following elements:

(1). All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

(2). Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

(3). Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

3. Consistency: The SRS is consistent if, and only if, no subset of individual requirements described in its conflict.

4. Unambiguousness: SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

5.Modifiability: SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced.

6.Verifiability: SRS is correct when the specified requirements can be verified with a cost-effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

7. Traceability: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

9. Design Independence: There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

10. Testability: An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

11. Understandable by the customer: An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

Properties of a good SRS document

- **Concise:** The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.
- **Structured:** It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

- **Black-box view:** It should only define what the system should do and not stating how to do these. This means that the SRS document should define the external behaviour of the system and not discuss the implementation issues. The SRS report should view the system to be developed as a black box and should define the externally visible behaviour of the system. For this reason, the SRS report is also known as the black-box specification of a system.

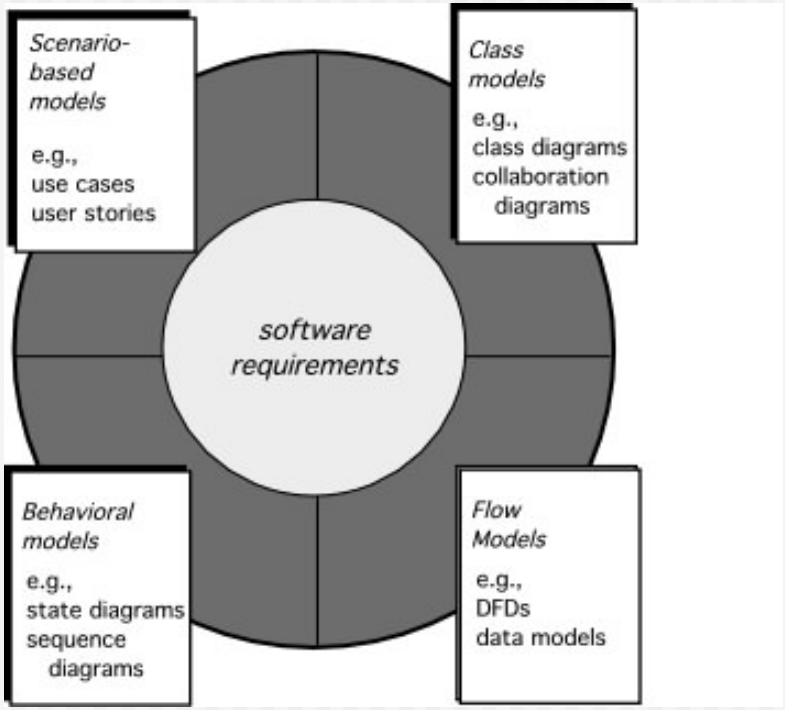
- **Conceptual integrity:** It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.
- **Verifiable:** All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

Requirement Analysis

- Requirement analysis is significant and essential activity after elicitation.
- It reviews all the requirements and may provide a graphical view of the entire system.
- The understandability of the project improves significantly.

Requirement analysis models/Elements

- Scenario-based modelling
- Flow oriented modelling
- Behavioral modelling
- Class-based modelling



Scenario-based modelling

- It identifies the primary use cases for the proposed software system or application, to which later stages of requirements modelling will refer.
- For building design and analysis model, it is important for software engineers to understand how end users and others want to interact with the system.

1. Use case diagram
2. Activity diagram

What is a use case diagram?

- A use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.
- To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

    1. Scenarios in which your system or application interacts with people, organizations, or external systems.

    2. Goals that your system or application helps those entities (known as actors) achieve.

    3. The scope of your system

1. Use case Diagram:

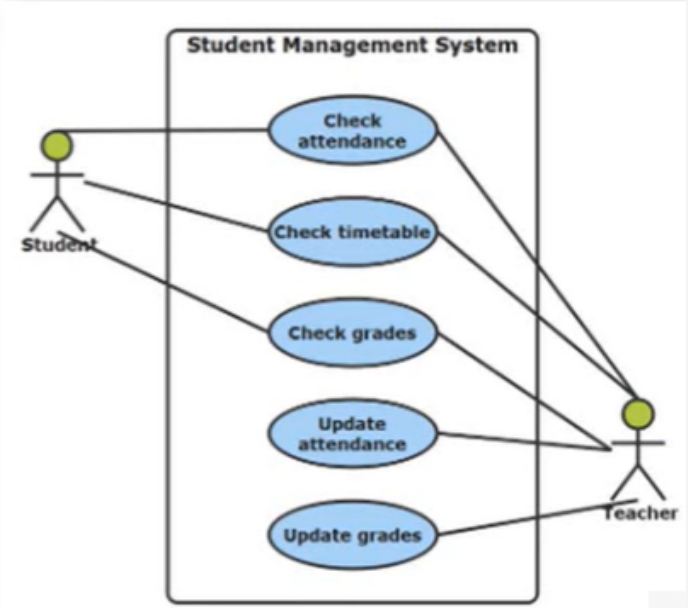
- A use case diagram is used to represent the dynamic behavior of a system.

: Actors

: Use cases

: Communication Link

: System Boundary



Use case diagrams are ideal for:

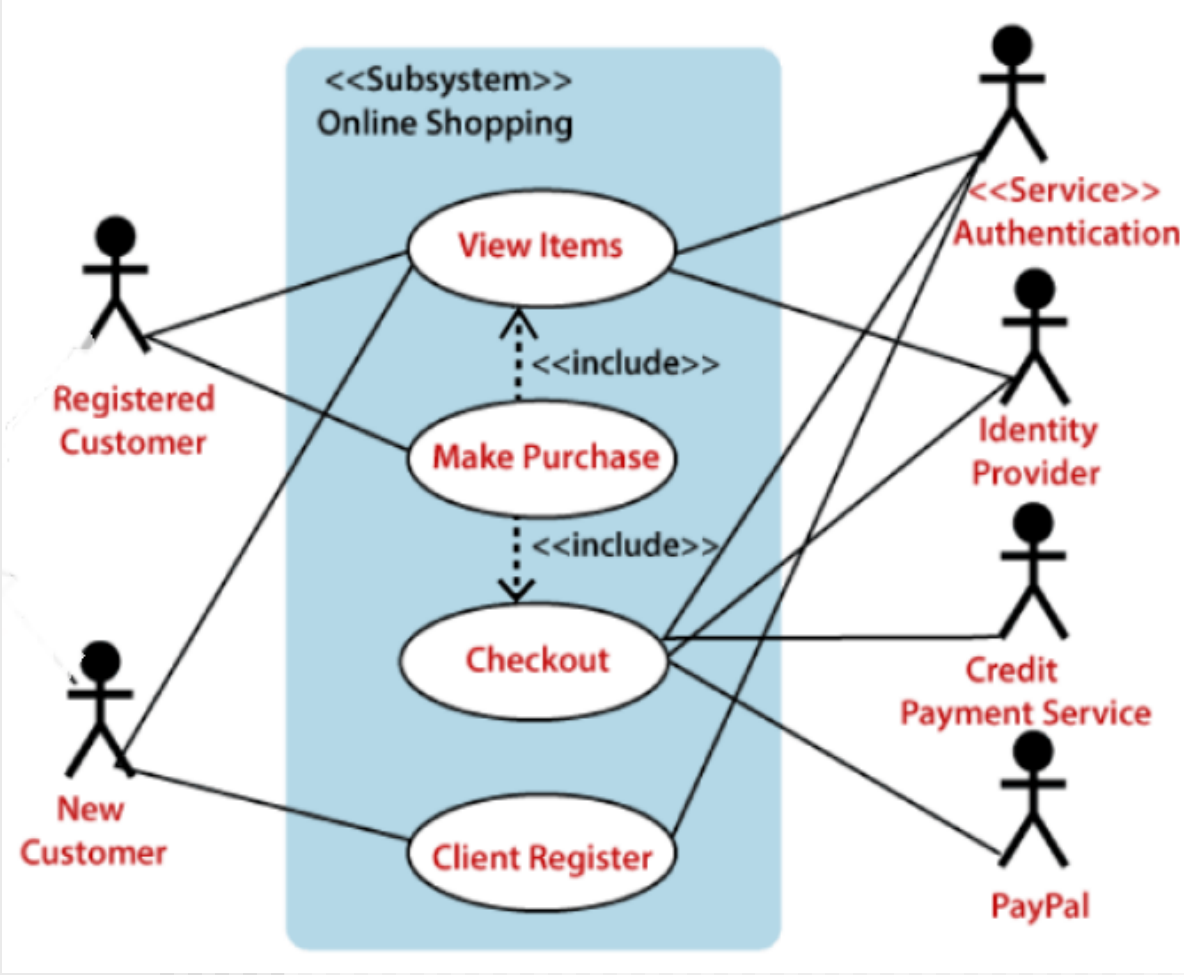
- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

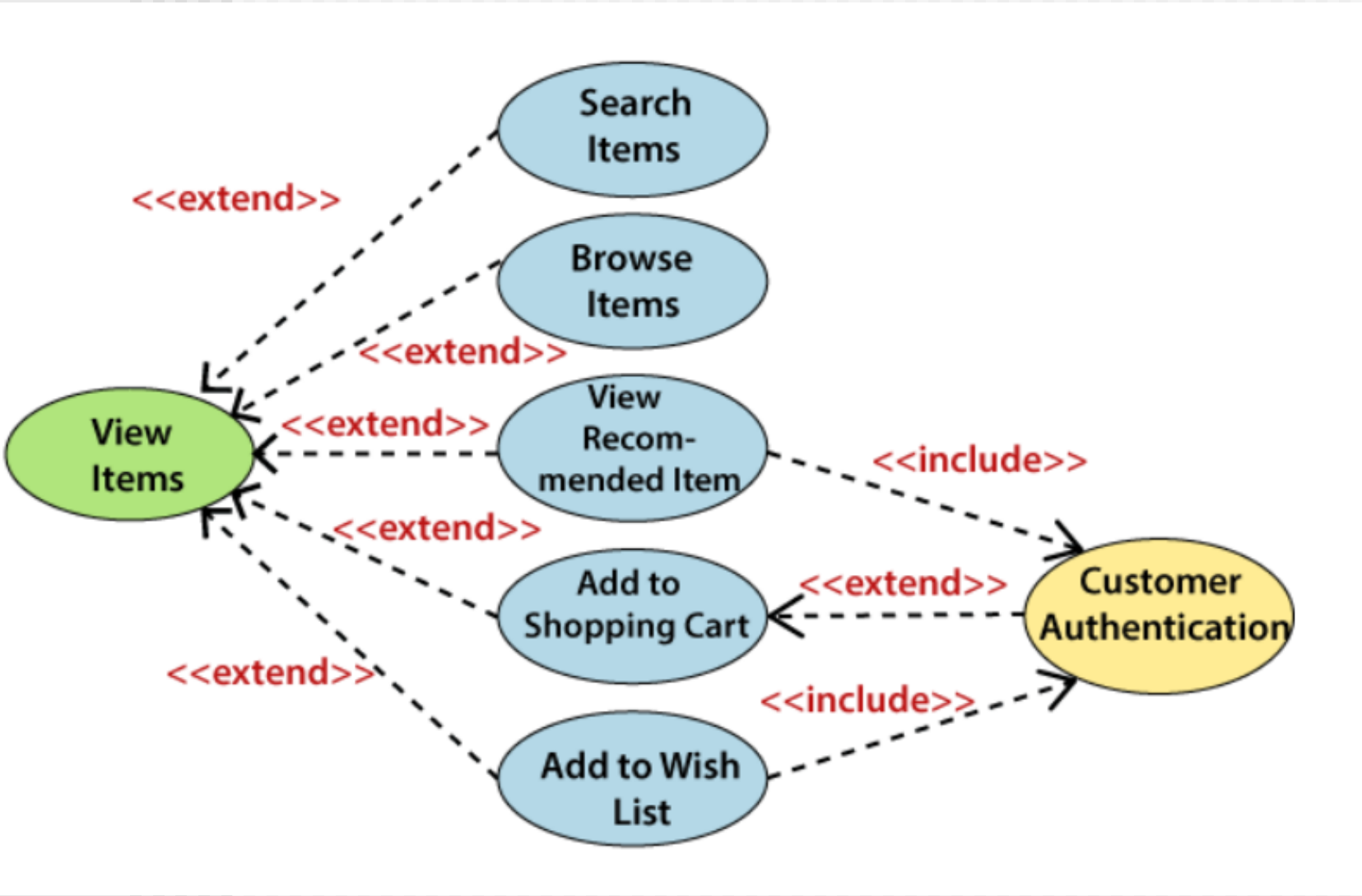
- **Use cases**: Horizontally shaped ovals that represent the different uses that a user might have.
- **Actors**: Stick figures that represent the people actually employing the use cases.
- **Associations**: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes**: A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system.

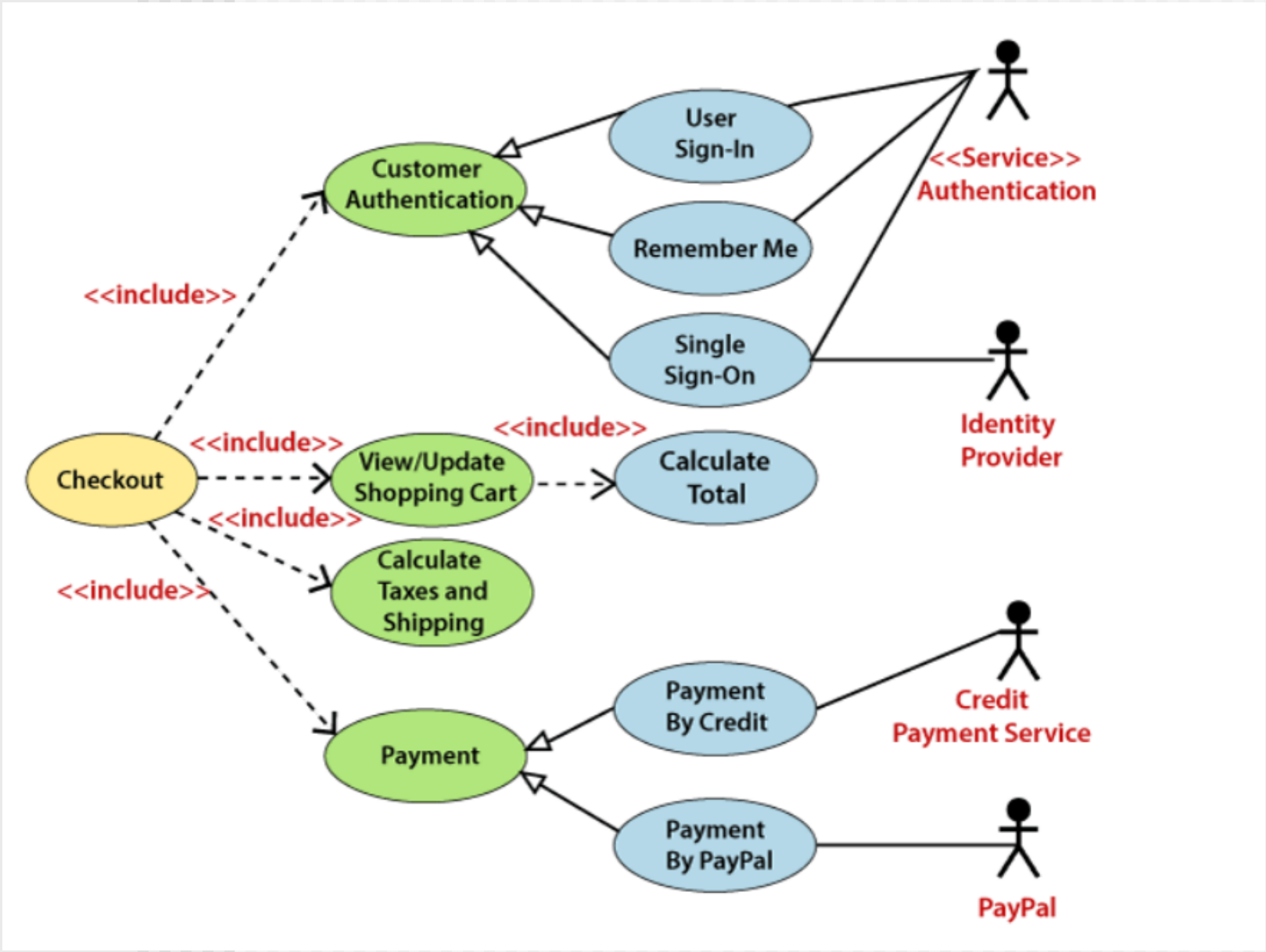
Rules that must be followed while drawing a use case diagram:

1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notations to be used as and when required.
4. The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example: Online shopping system




Search Items
<<extend>>
Browse Items
<<extend>>
View Items
<<extend>>
View Recom-mended Item
<<include>>
<<extend>>
Add to Shopping Cart
<<extend>>
Customer Authentication
<<extend>>
<<include>>
Add to Wish List

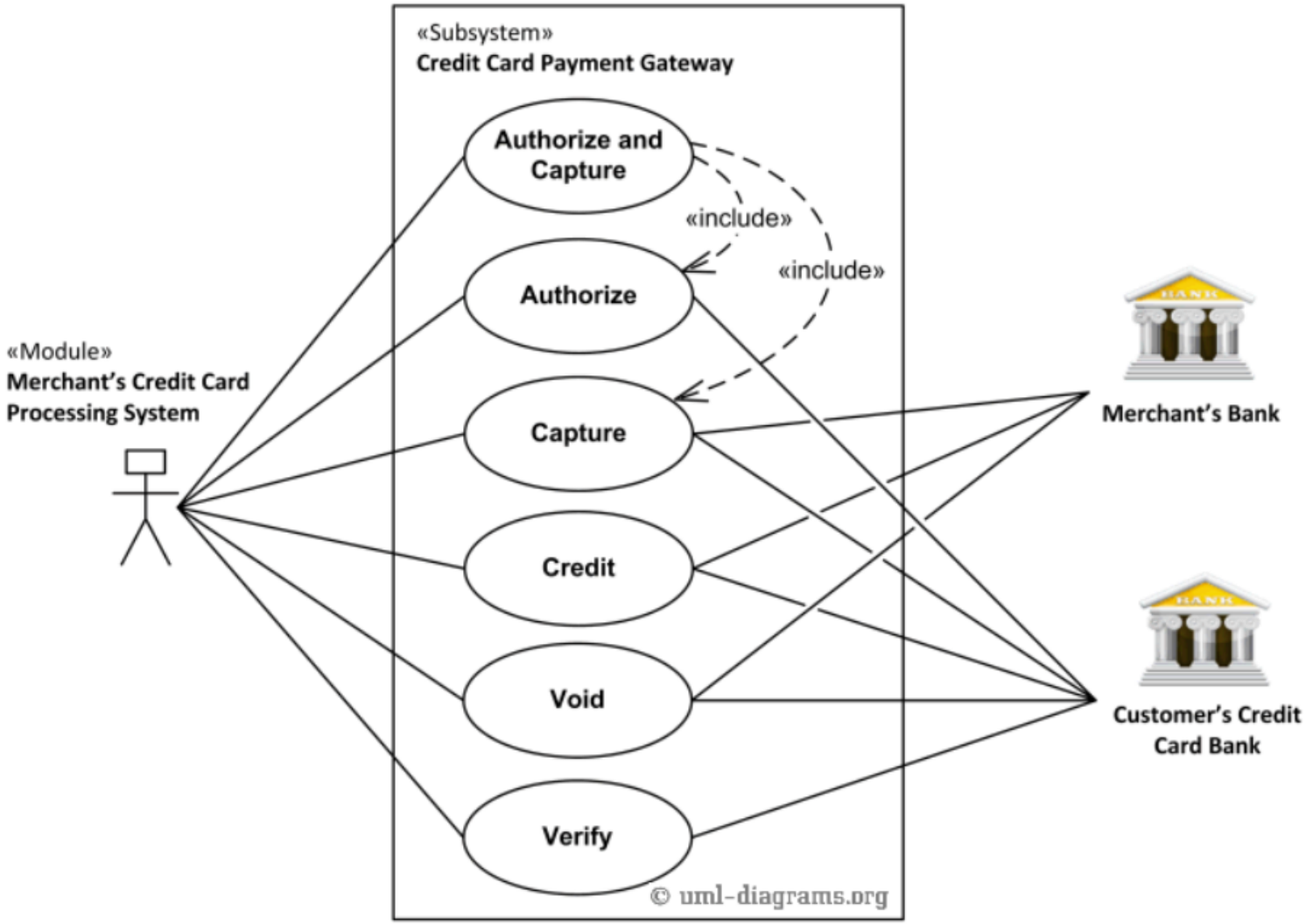

User
Sign-In
Customer
Authentication
Remember Me
<<Service>>
Authentication
<<include>>
Single
Sign-On
Identity
Provider
<<include>>
<<include>>
Checkout
View/Update
Shopping Cart
Calculate
Total
<<include>>
Calculate
Taxes and
Shipping
<<include>>
Payment
By Credit
Credit
Payment Service
Payment
Payment
By PayPal
PayPal

Tips to draw use case diagrams

1. A simple and complete use case diagram should be articulated.
2. A use case diagram should represent the most significant interaction among the multiple interactions.
3. At least one module of a system should be represented by the use case diagram.
4. If the use case diagram is large and more complex, then it should be drawn more generalized.

Exercise 1

1. Draw use case diagram for credit card processing system.
2. Draw use case diagram for bank management system.


«Subsystem»
Credit Card Payment Gateway
Authorize and Capture
«include»
«include»
Authorize
Capture
Credit
Void
Verify
© uml-diagrams.org
«Module»
Merchant's Credit Card Processing System
Merchant's Bank
Customer's Credit Card Bank

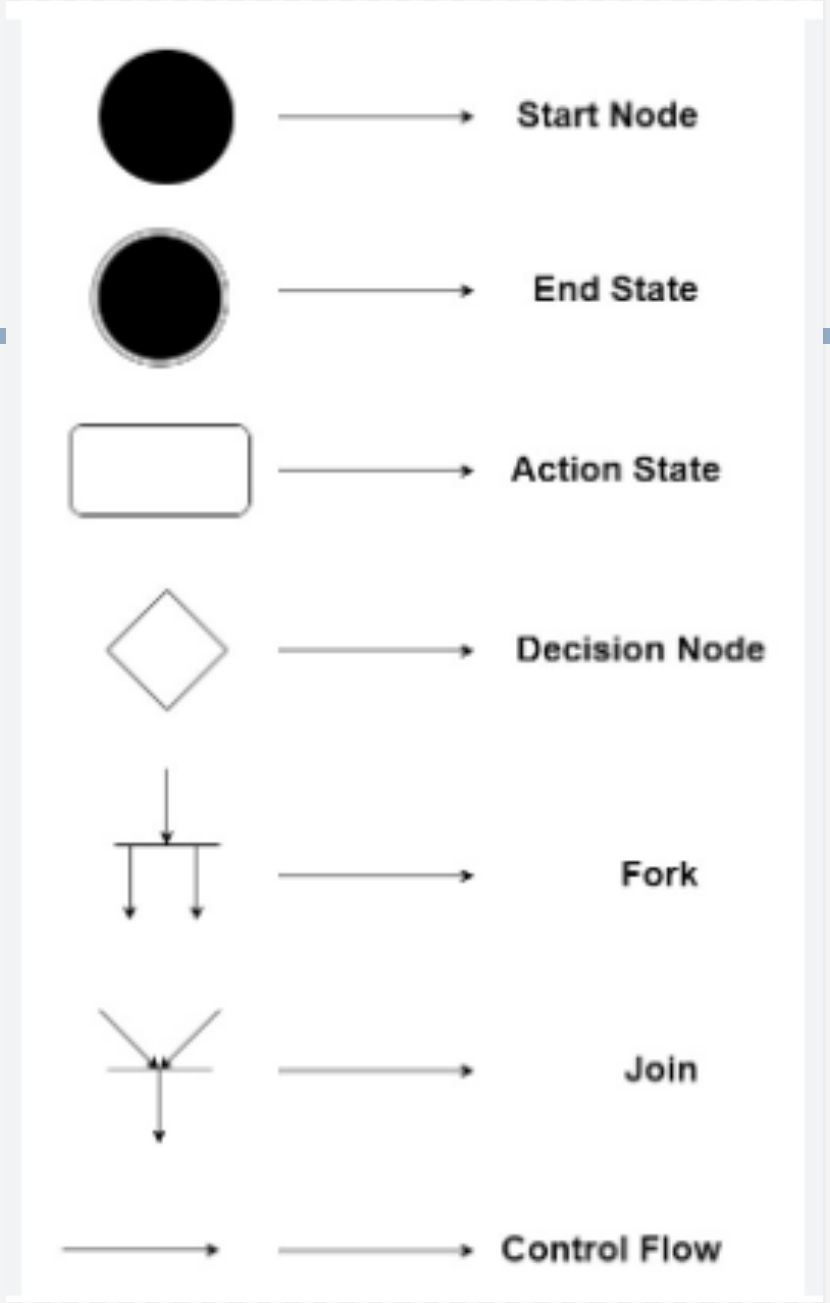
2. Activity Diagram

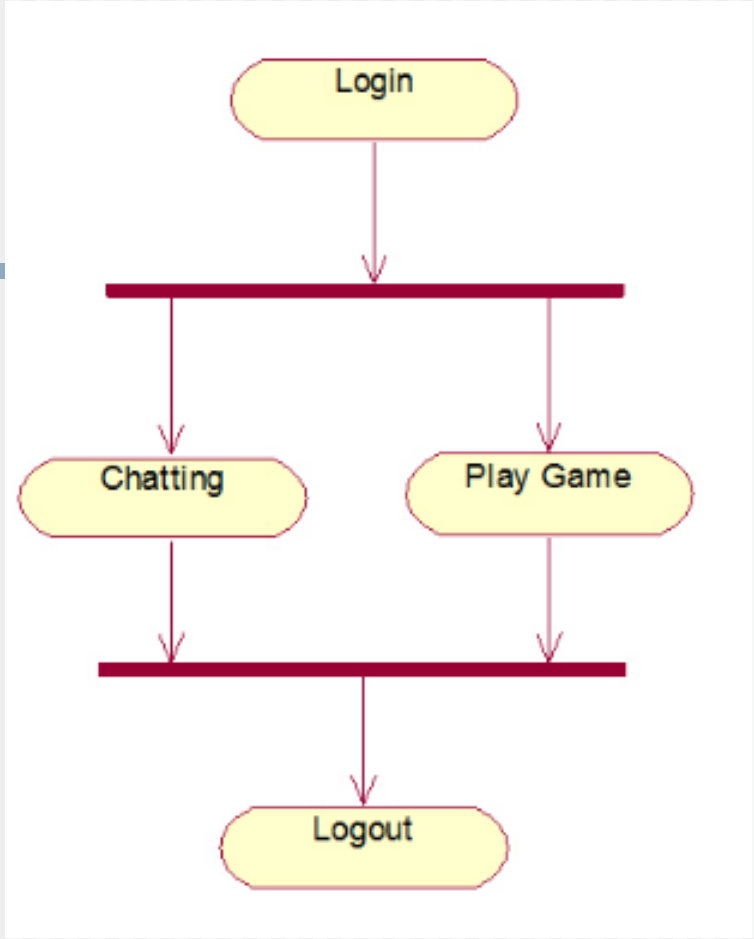
Activity diagram is designed to model the system's workflow.

The diagram can include nodes, which represent activities or actions, and edges, which represent the flow of control between activities.

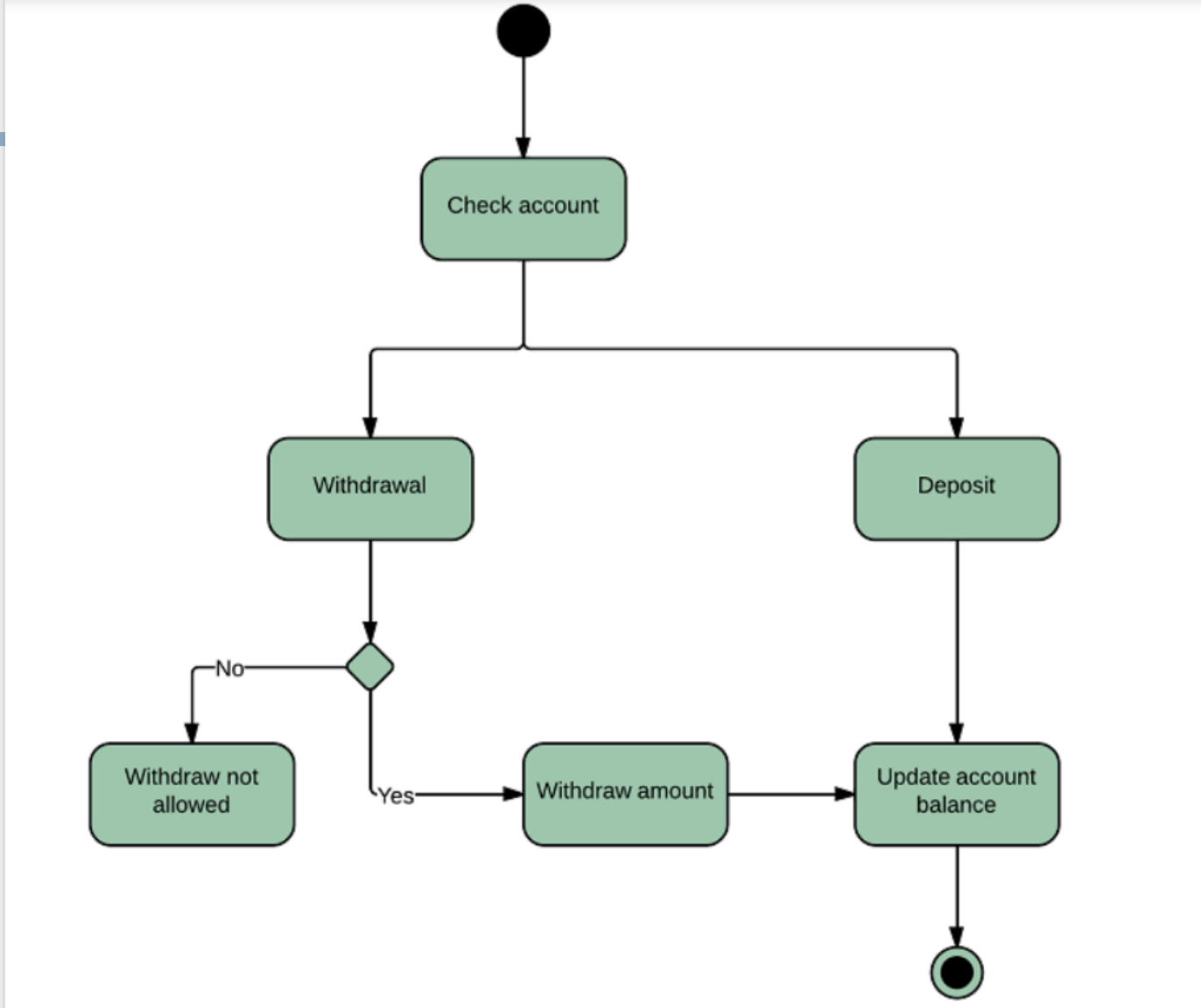
Basic components of an activity diagram

- Action: A step in the activity wherein the users or software perform a given task. actions are symbolized with round-edged rectangles.
- Decision node: A conditional branch in the flow that is represented by a diamond. It includes a single input and two or more outputs.
- Control flows: Another name for the connectors that show the flow between steps in the diagram.
- Start node: Symbolizes the beginning of the activity. The start node is represented by a black circle.
- End node: Represents the final step in the activity. The end node is represented by an outlined black circle.
- Fork: Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
- Join: Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.

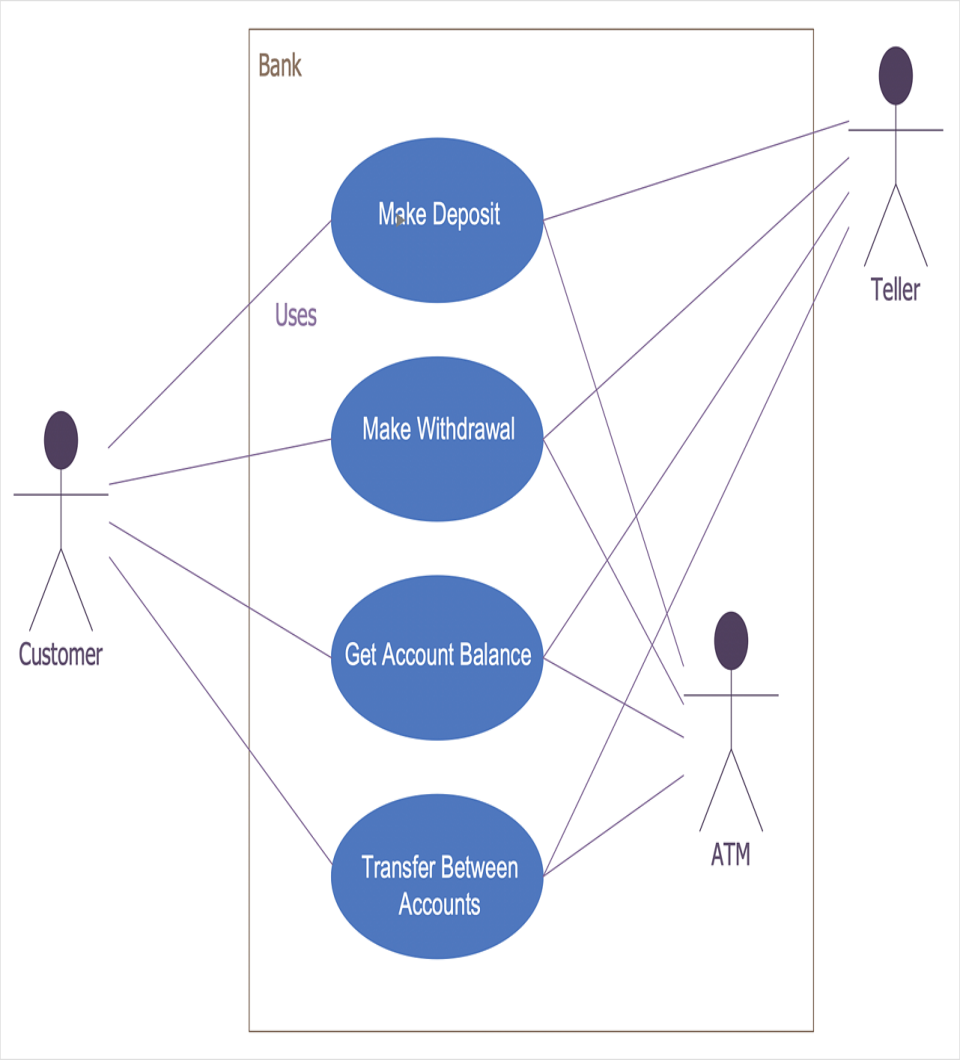

Start Node
End State
Action State
Decision Node
Fork
Join
Control Flow


Login
Chatting
Play Game
Logout

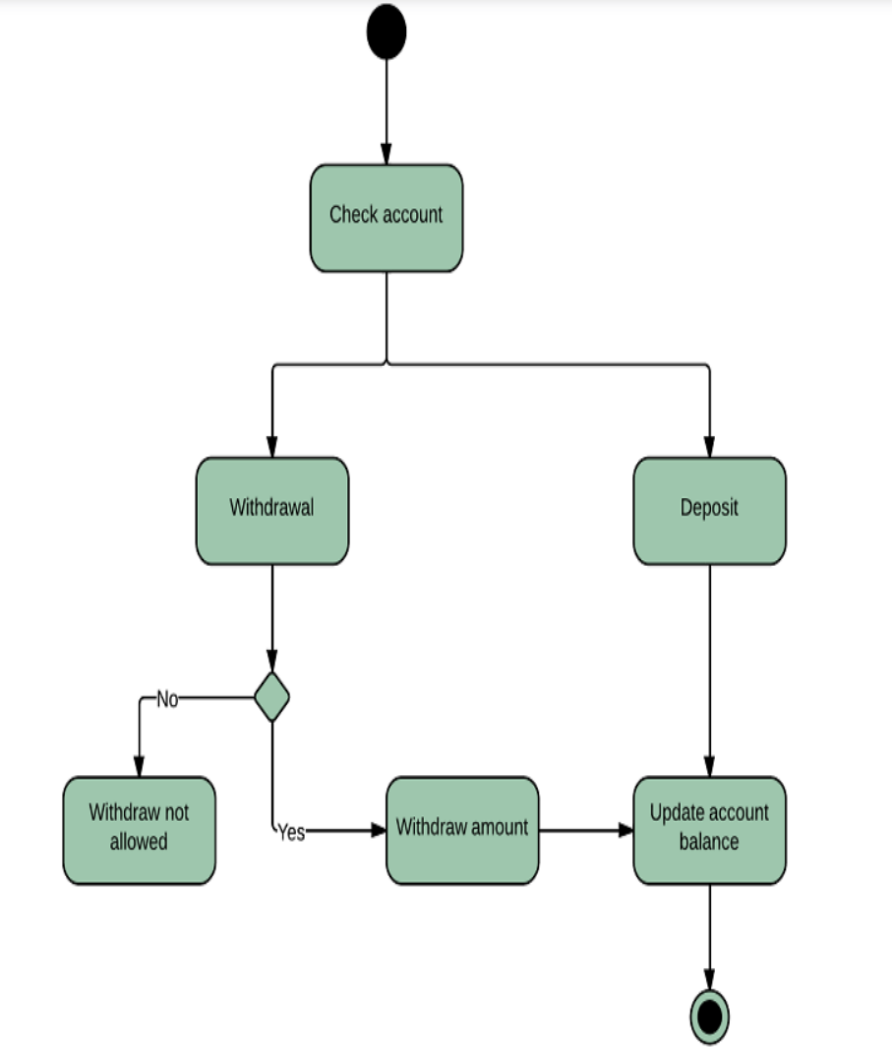
Examples : banking system



Banking System



Use case diagram

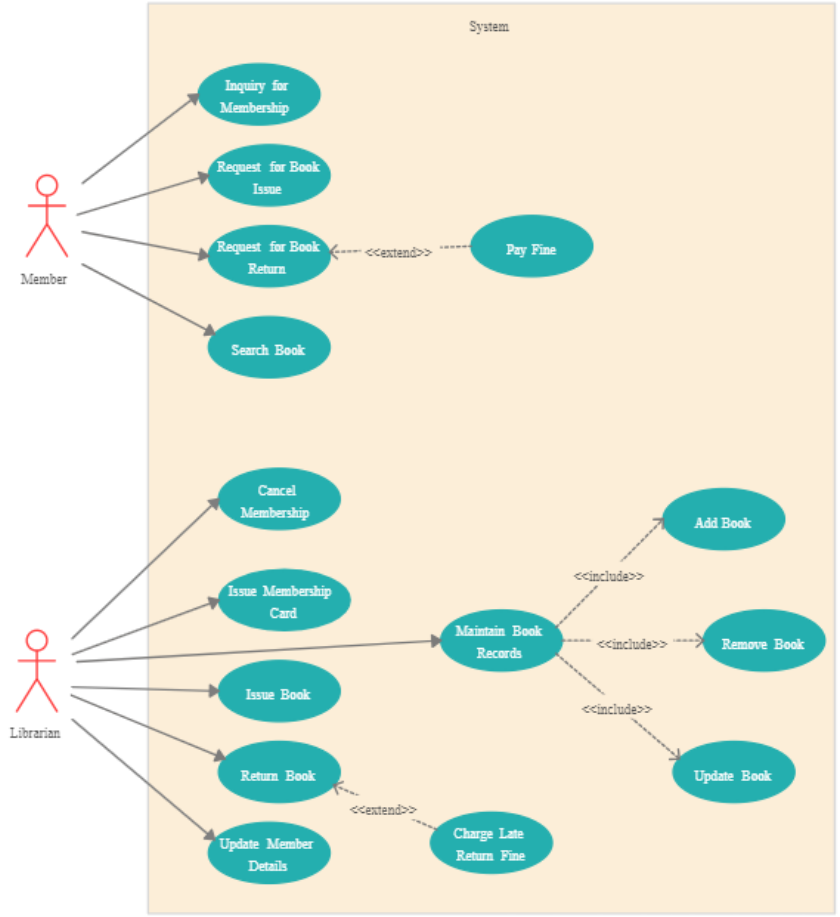


Activity diagram

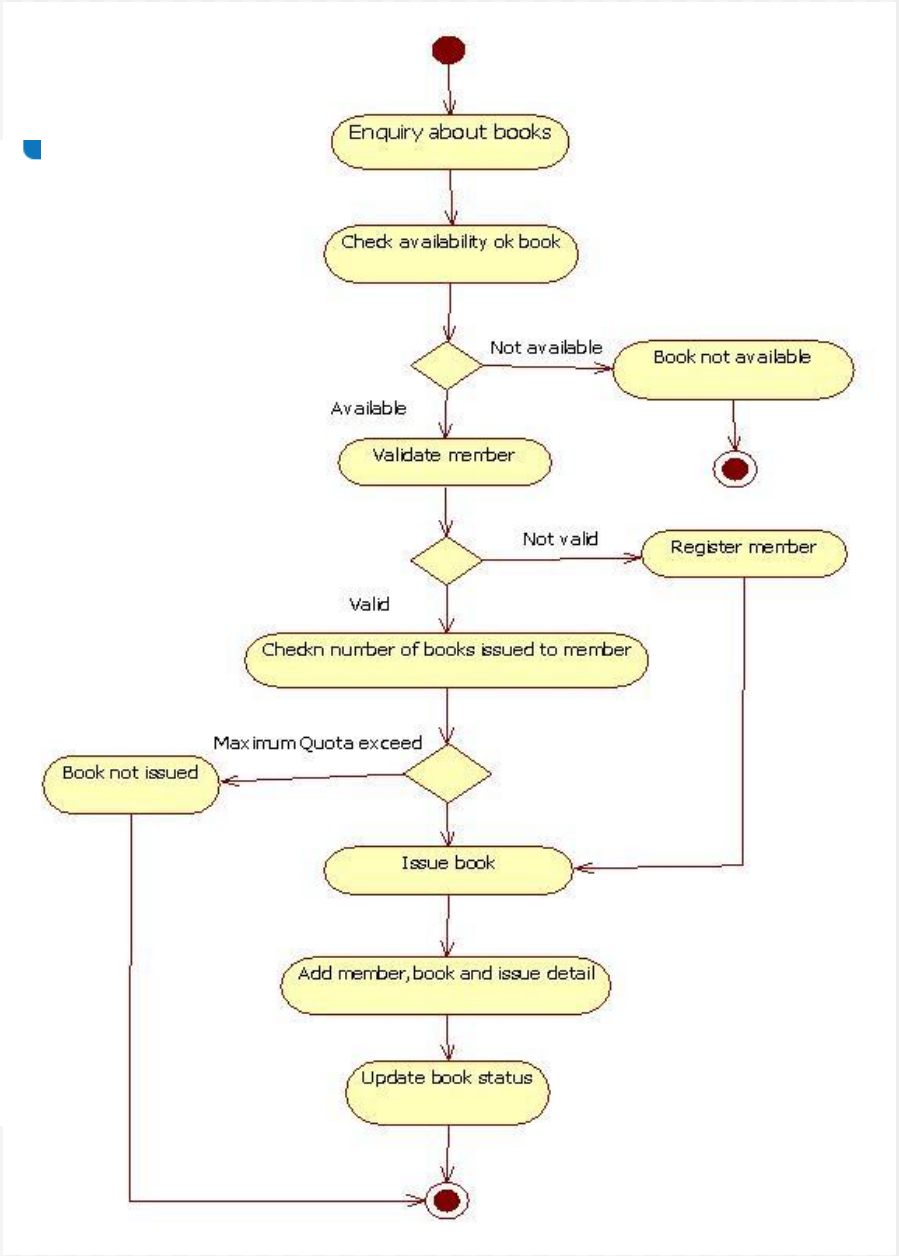
Exercise

1. Draw Activity diagram for online shopping system.
2. Draw use case diagram and activity diagram for library management system.

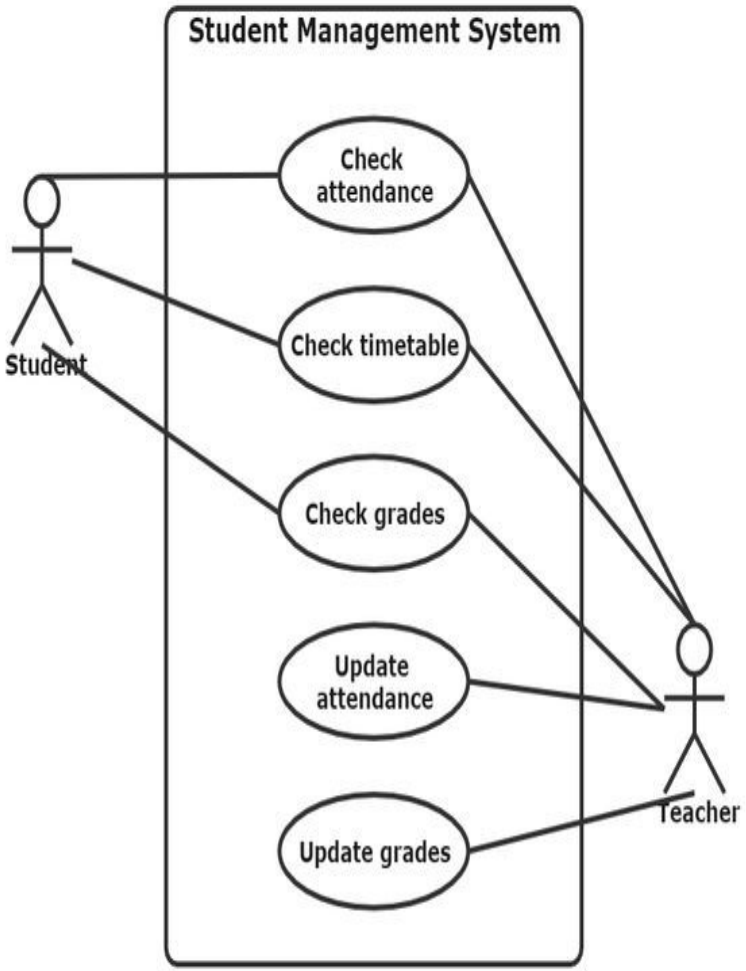
LIBRARY MANAGEMENT SYSTEM



USE CASE DIAGRAM



ACTIVITY DIAGRAM


Student Management System
Check attendance
Check timetable
Check grades
Update attendance
Update grades
Student
Teacher

A Data Flow Diagram (DFD)

- A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically.

There are two types of DFDs

logical and physical.

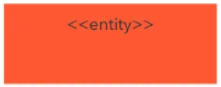
Logical diagrams display the theoretical process of moving information through a system, like where the data comes from, where it goes, how it changes, and where it ends up.

Physical diagrams show you the practical process of moving information through a system. It can show how your system's specific software, hardware, files, employees, and customers influence the flow of information.

Data Flow Diagram Symbols

Data flow diagram symbols are standardized notations, like rectangles, circles, arrows, and short-text labels. These symbols represent a system's data flow direction, inputs, outputs, storage points, and sub-processes.

Four common methods of notation are used in DFDs: Yourdon & De Marco, Gene & Sarson, SSADM, and Unified.

Notation	Yourdon & De Marco	Gene & Sarson	SSADM	Unified
External Entity				
Process				
Data Store				
Data Flow				

- 1. External Entity
- Also known as terminators, sources, sinks, or actors are outside systems that send or receive data to and from the diagrammed system.They're either the sources or destinations of information, so they're usually placed on the diagram's edges.External entity symbols are similar across models except for Unified, which uses a stick-figure drawing instead of a rectangle, circle, or square.
- 2. Process
- process is a procedure that manipulates the data and its flow by taking incoming data, changing it, and producing an output. A process can do this by performing computations and using logic to sort the data or change its flow of direction.Processes usually start from the top left of the DFD and finish on the bottom right of the diagram.

- 3. Data Store
- Data stores hold information for later use, like a file of documents that's waiting to be processed. Data inputs flow through a process and then through a data store, while data outputs flow out of a data store and then through a process.
- 4. Data Flow
- Data flow is the path the system's information takes from external entities through processes and data stores. With arrows and succinct labels, the DFD can show you the direction of the data flow.

DFD Levels

- DFDs can range from simple overviews to complex, granular representations of a system or process with multiple levels, starting with level 0.
- The most common and intuitive DFDs are level 0 DFDs, also called context diagrams.
- They're digestible, high-level overviews of the flow of information through a system or process, so almost anyone can understand it.

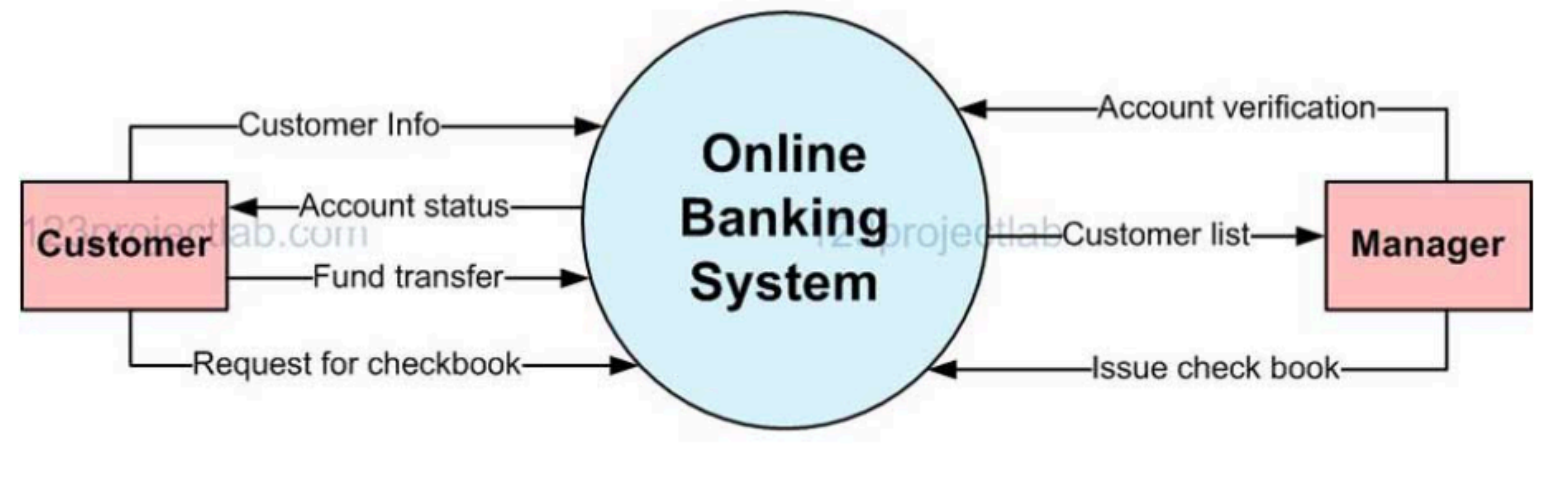
- **Level 0**: Context DiagramThis DFD level focuses on high-level system processes or functions and the data sources that flow to or from them.
- Level 0 diagrams are designed to be simple, straightforward overviews of a process or system.

- **Level 1:** Process Decomposition, level 1 DFDs are still broad overviews of a system or process, they're also more detailed
- — they break down the system's single process node into subprocesses.

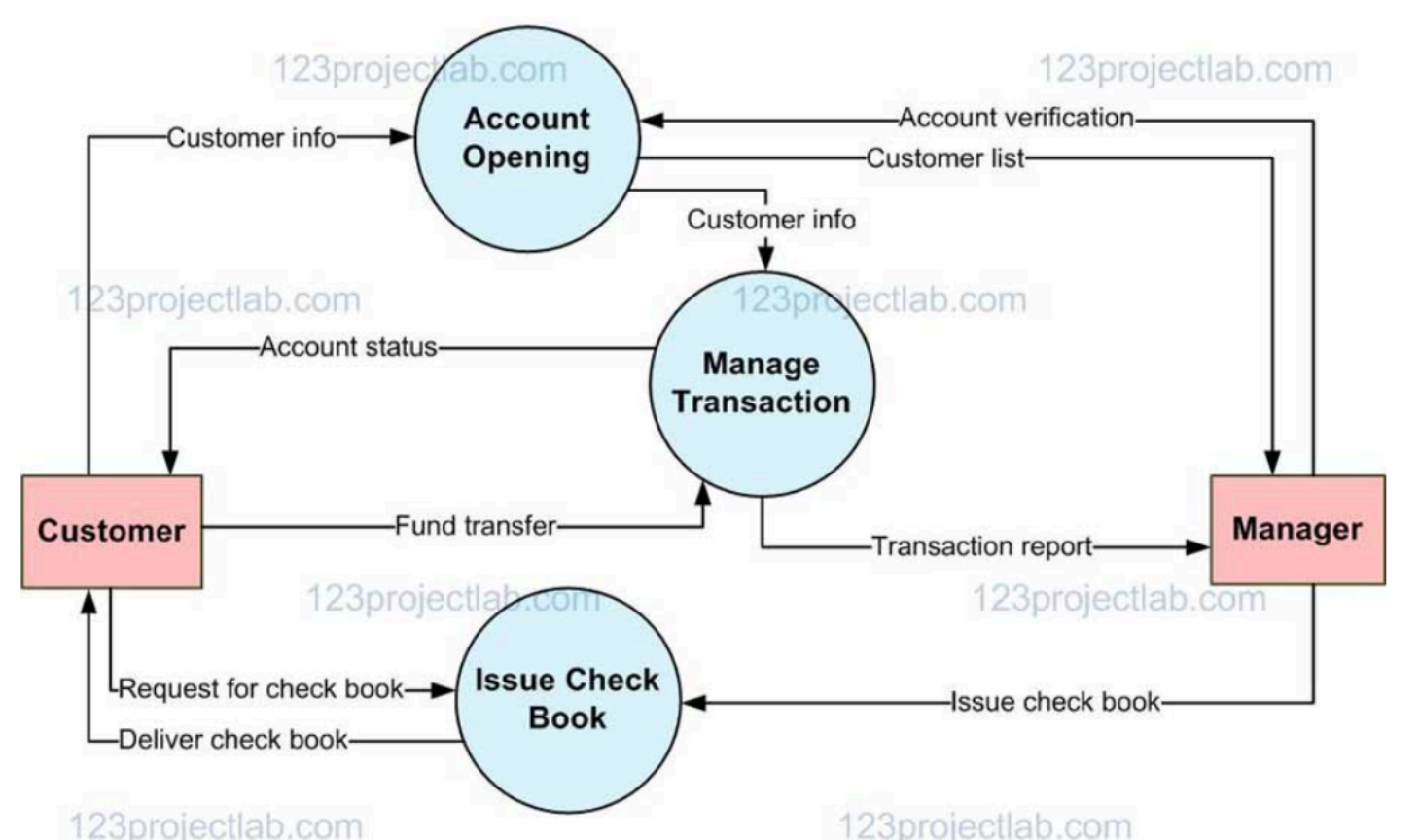
- **Level 2**: Deeper DivesThe next level of DFDs dives even deeper into detail by breaking down each level 1 process into granular subprocesses.

Online banking system

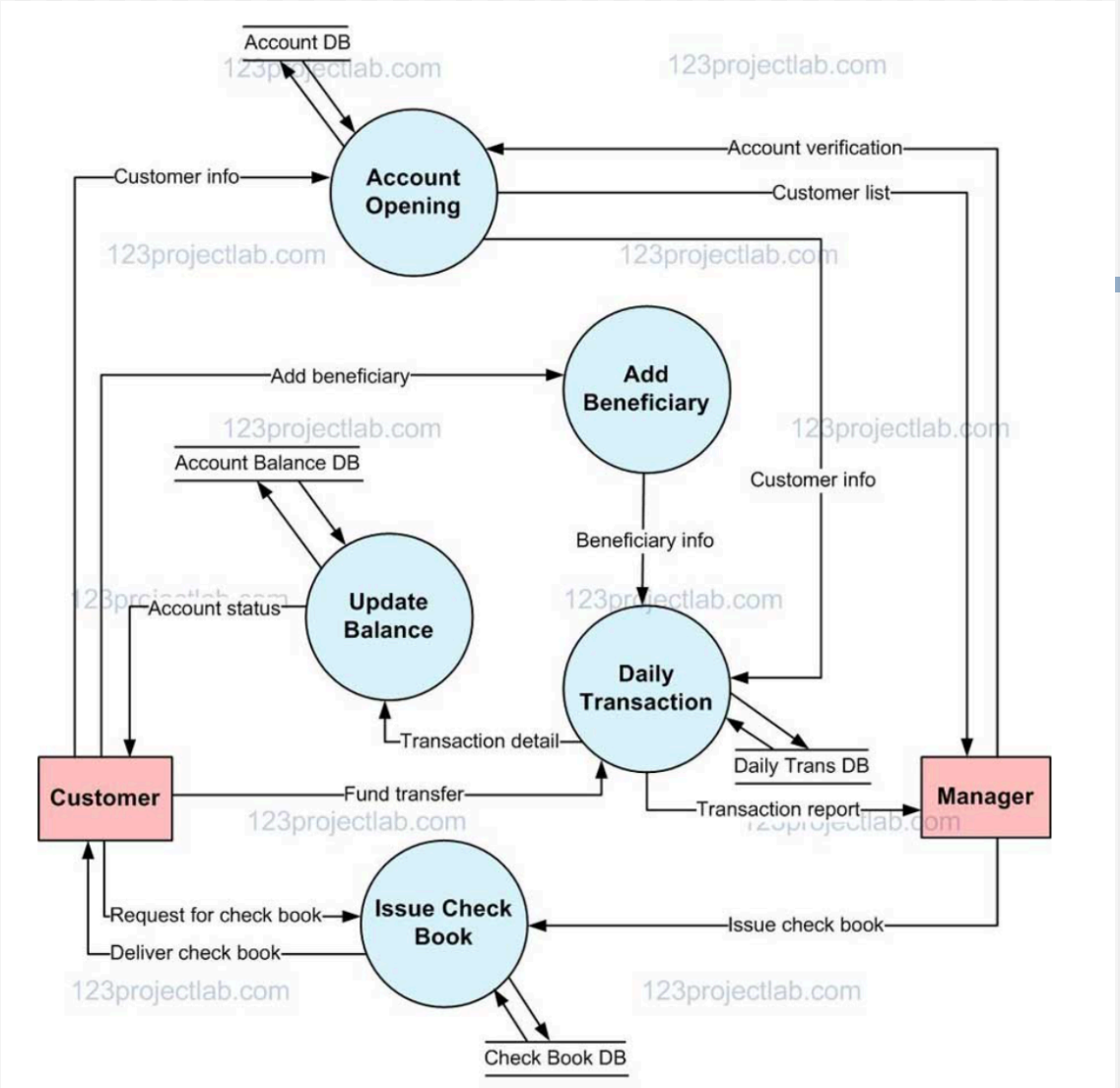
- Level 0 DFD



Level 1 DFD


Account Opening
Customer info
Account verification
Customer list
Customer info
Manage Transaction
Account status
Customer
Fund transfer
Transaction report
Manager
Issue Check Book
Request for check book
Deliver check book
Issue check book
123projectlab.com

Level 2 DFD


Account DB
Account Opening
Customer info
Account verification
Customer list
Add Beneficiary
Add beneficiary
Account Balance DB
Update Balance
Account status
Beneficiary info
Customer info
Daily Transaction
Transaction detail
Daily Trans DB
Customer
Fund transfer
Transaction report
Manager
Issue Check Book
Request for check book
Deliver check book
Issue check book
Check Book DB
123projectlab.com

SOFTWARE DESIGN

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form easily implementable using programming language.

Characteristics of a good Software design

1. **Correctness**:Software design should be correct as per requirement.
2. **Completeness**:The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency**:Resources should be used efficiently by the program.
4. **Flexibility**:Able to modify on changing needs.
5. **Consistency**:There should not be any inconsistency in the design.
6. **Maintainability**: The design should be so simple so that it can be easily maintainable by other designers.

Preliminary Design

- The preliminary design, or high-level design (also called FEED or Basic design), often bridges a gap between design conception and detailed design, particularly in cases where the level of conceptualization achieved during ideation is not sufficient for full evaluation.

- So in this task, the overall system configuration is defined, and schematics, diagrams, and layouts of the project may provide early project configuration. (This notably varies greatly by field, industry, and product.)

- During detailed design and optimization, the parameters of the part being created will change, but the preliminary design focuses on creating the general framework to build the project.

After Preliminary design…

-You have a graphical representation of the structure of your software system

- A document that defines high-level details of each module in your system including,
  - A module's interface (including input and output data types)
  - Notes on possible algorithms or data structures the module can use to meet its responsibilities.
  - A list of any non-functional requirements that might impact the module

Detailed design

After high-level design,

- A designer's focus shifts to low-level design
- Each module's responsibilities should be specified as precisely as possible
- Constraints on the use of its interface should be specified
- pre and post conditions can be identified
- module-wide invariants can be specified
- internal data structures and algorithms can be suggested

A designer must exhibit caution,

- To not over specify a design.
- we do not want to express a module's detailed design using a programming language.
- you would naturally end up implementing the module perhaps unnecessarily constraining the approaches a developer would use to accomplish the same job.
- nor do we (necessarily) want to use only natural language text to specify a module's detailed design

Software Design Principles

-Software design principles are concerned with providing means to handle the complexity of the design process effectively.

-Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

1. Problem Partitioning
2. Abstraction
3. Modularity
4. Strategy of design

1. Problem Partitioning

- For small problems, we can handle the entire problem at once but for the significant problem, divide them and conquer the problem.

- It means to divide the problem into smaller pieces so that each piece can be captured separately.

- For software design, the goal of problem partitioning is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

-These pieces cannot be entirely independent of each other as they together form the system.

-They have to cooperate and communicate to solve the problem.

- This communication adds complexity.

2. Abstraction

- An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation.

- Abstraction can be used for existing element as well as the component being designed.

- Two common abstraction mechanisms

1. Functional Abstraction

2. Data Abstraction

1. Functional Abstraction

- A module is specified by the method it performs.
- The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

2. Data Abstraction

Details of the data elements are not visible to the users of data.

Data Abstraction forms the basis for **Object Oriented design approaches**.

3. Modularity

- Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software.

- It is the only property that allows a program to be intellectually manageable.

- Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test.
- It produced the well designed and more readable program.

Disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

Modular Design

Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system.

- Functional independence
- Information hiding

1. Functional Independence

- Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules.
- Independence is important because it makes implementation more accessible and faster.
- The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well.
- Functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

2. Information hiding

- Modules can be characterized by the design decisions that protect from the others.

- Modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

- The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing and later during software maintenance.

- This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

4. Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change.

Structured design methods help developers to deal with the size and complexity of programs.

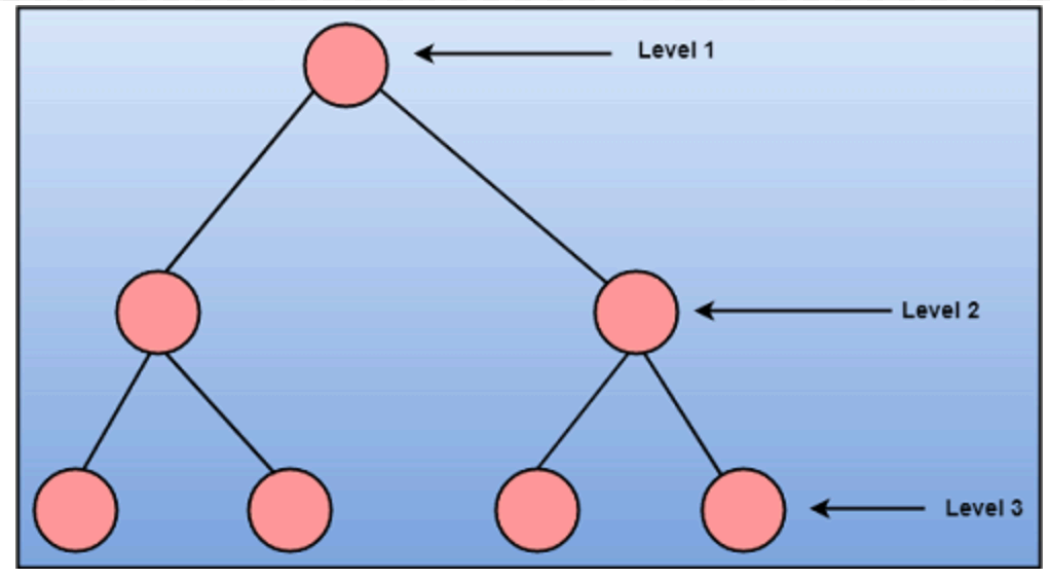
Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

Two possible approaches:

1. Top-down Approach
2. Bottom-up Approach

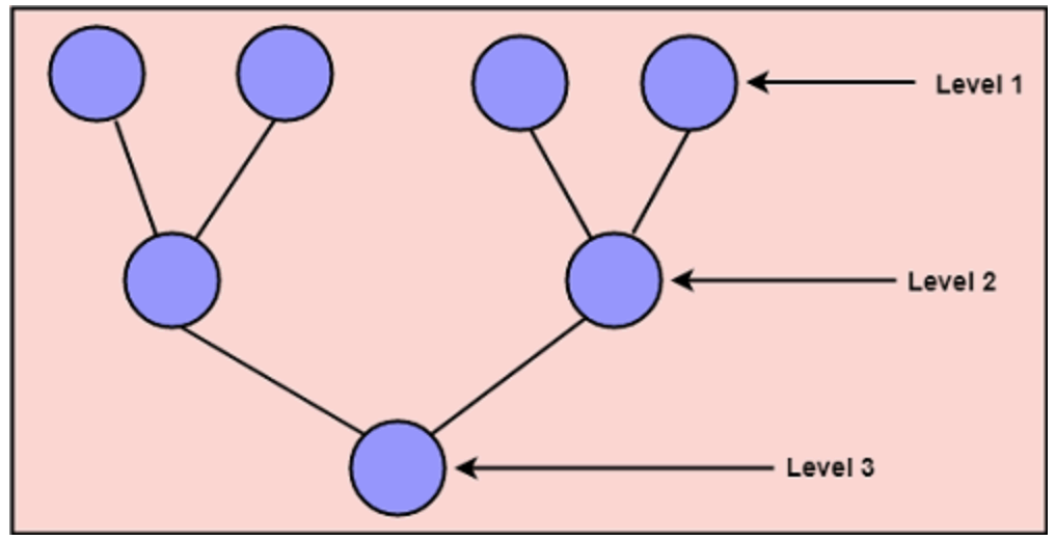
Top down approach

This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.



Bottom up approach

A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.



Coupling and Cohesion

Module Coupling

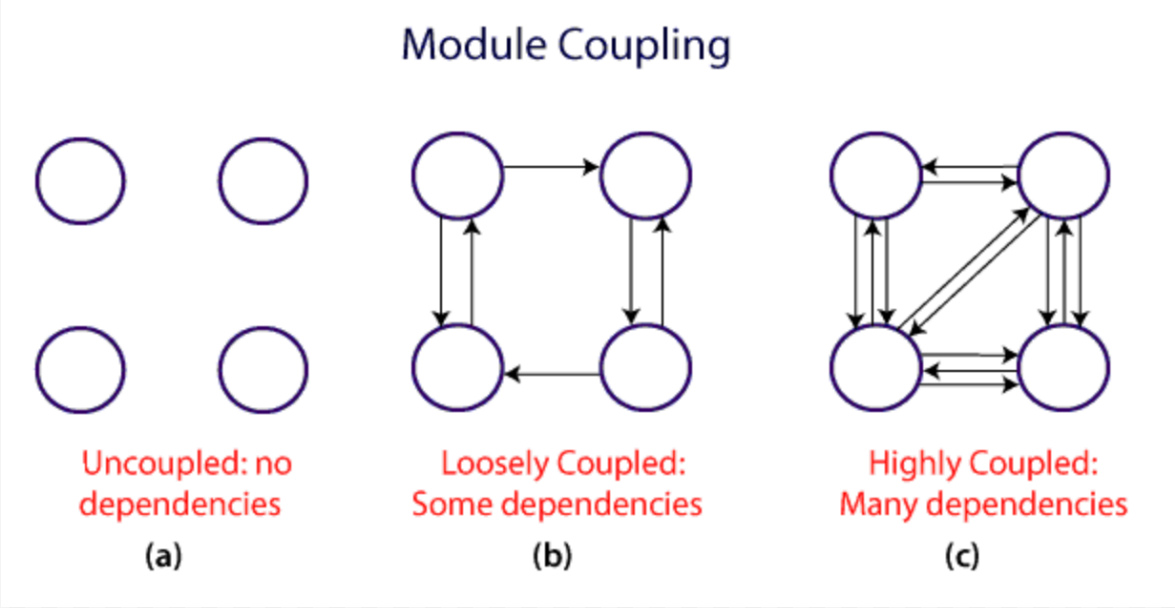
- Coupling is the degree of interdependence between software modules.

- Two modules that are tightly coupled are strongly dependent on each other.

- However, two modules that are loosely coupled are not dependent on each other.

- Uncoupled modules have no interdependence at all within them.

The various types of coupling techniques are shown in fig:

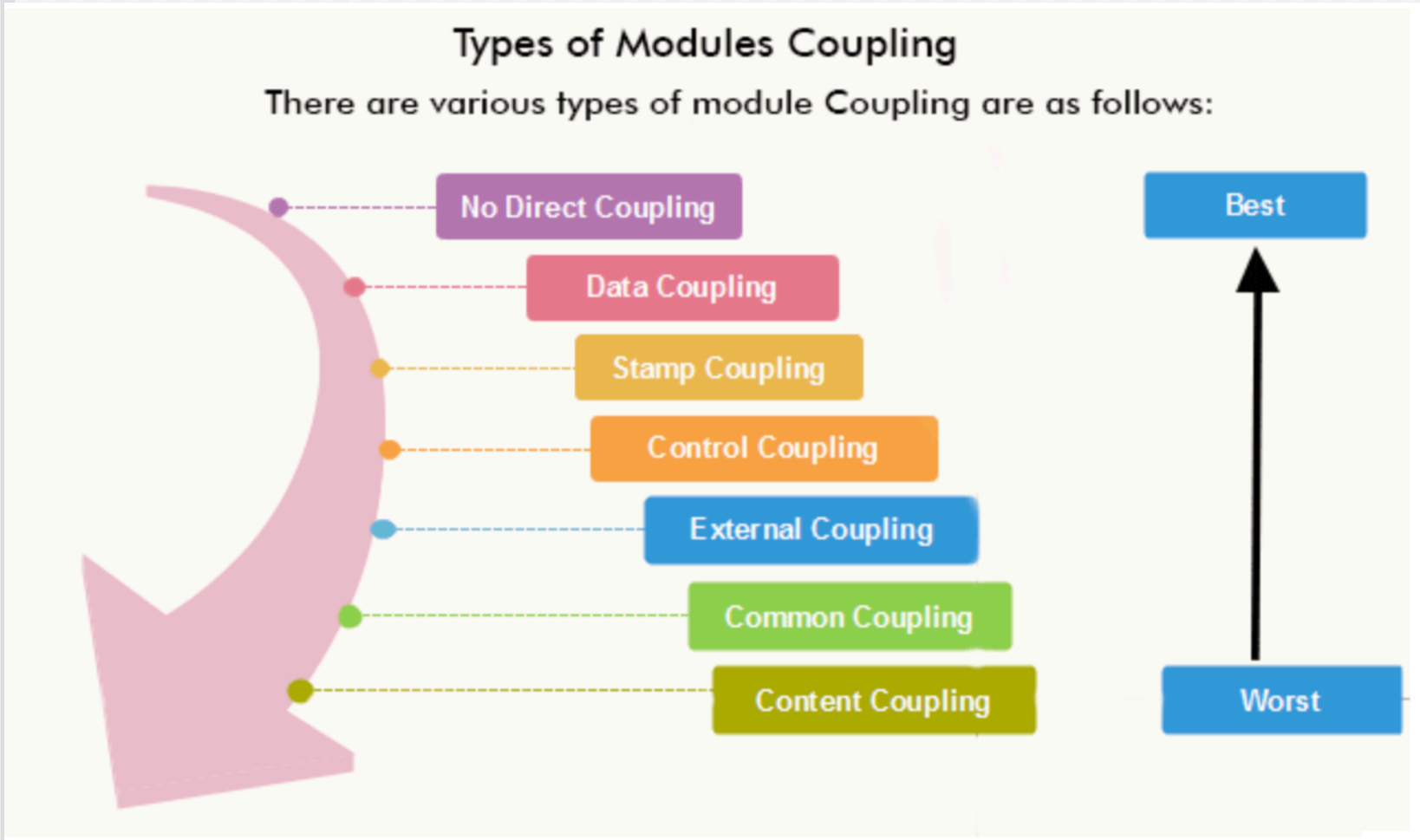


- A good design is the one that has low coupling.

- Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large.

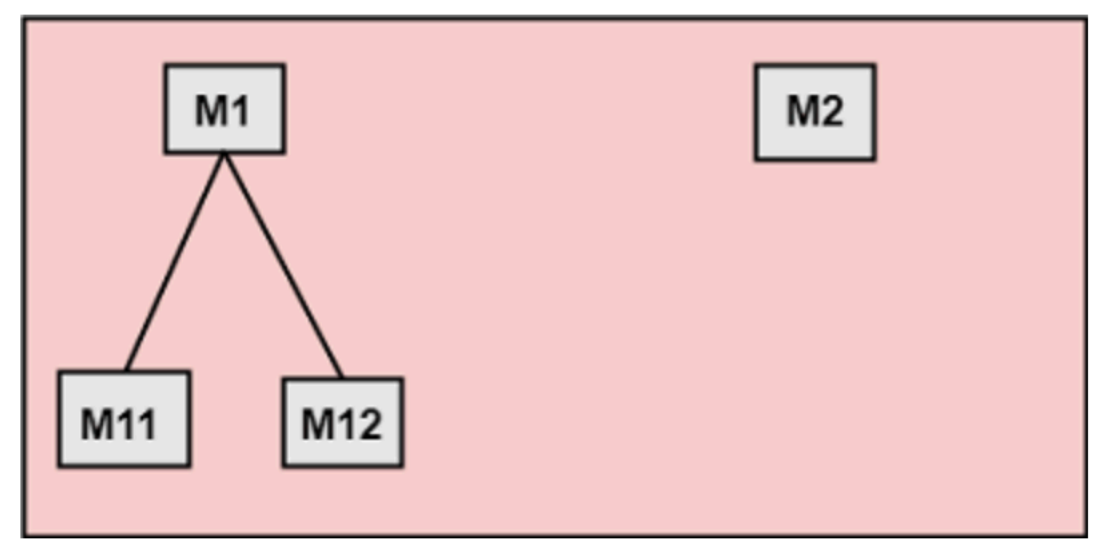
- A design with high coupling will have more errors.

Types of Module Coupling



1. No Direct Coupling

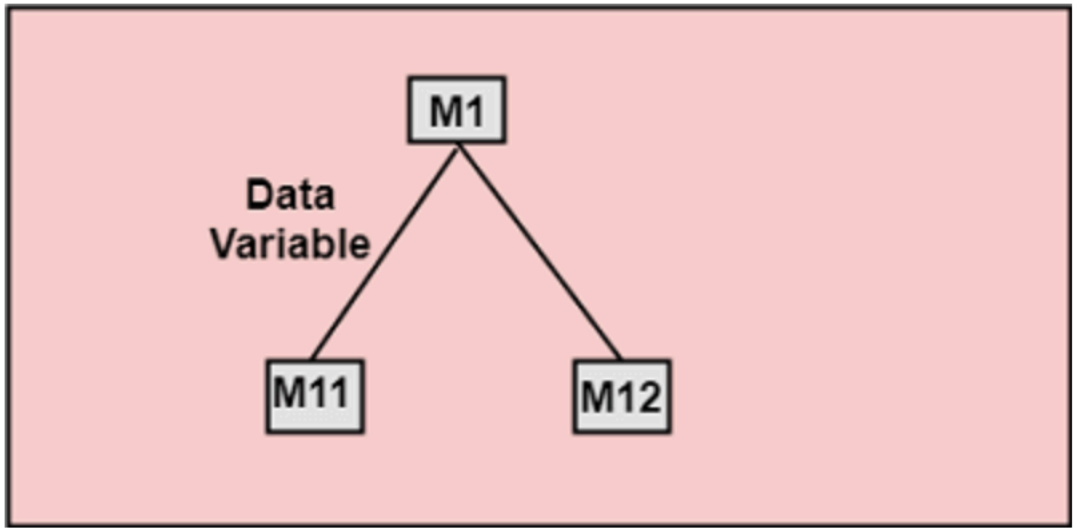
There is no direct coupling between M1 and M2



Modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling

When data of one module is passed to another module, this is called data coupling.



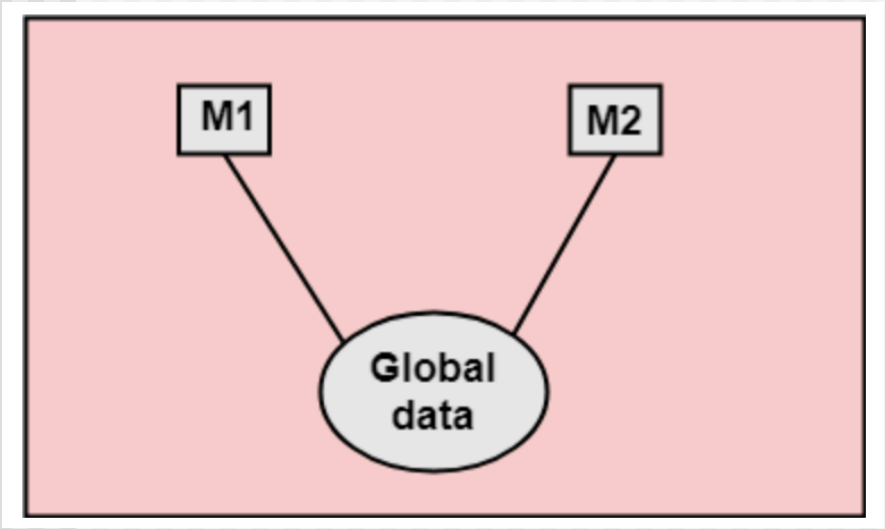
3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5**. Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

6. **External Coupling**: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

7. **Common Coupling**: Two modules are common coupled if they share information through some global data items.

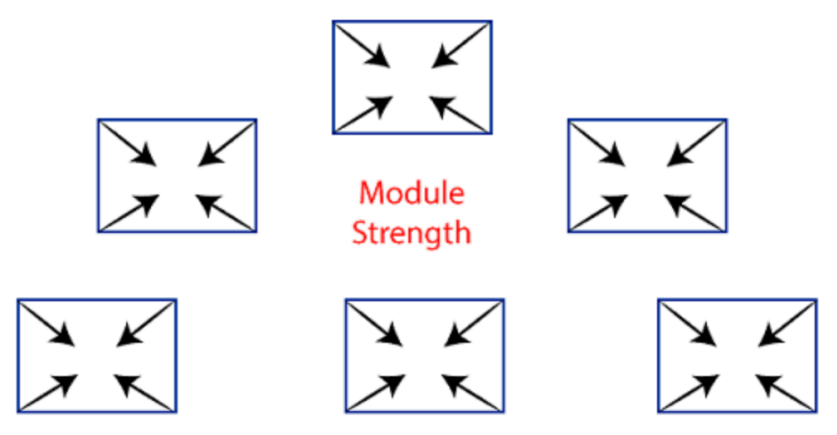


Module Cohesion

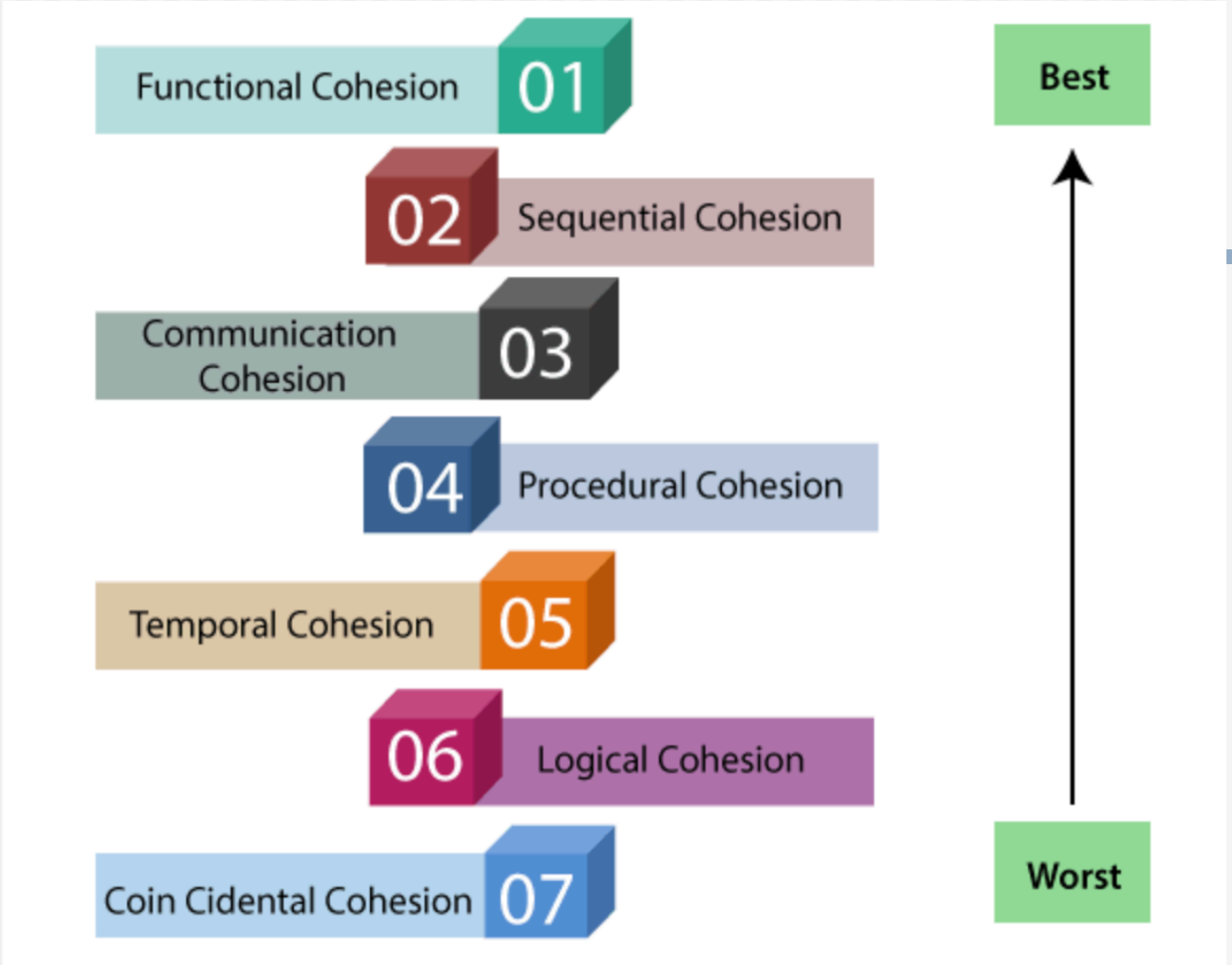
- Cohesion defines to the degree to which the elements of a module belong together.

- Cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

- Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Types of module cohesion



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3. **Communicational Cohesion**: A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack

1. **Procedural Cohesion**: A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

2. **Temporal Cohesion**: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

3. **Logical Cohesion**: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

4. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely.

Differentiate between coupling and cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

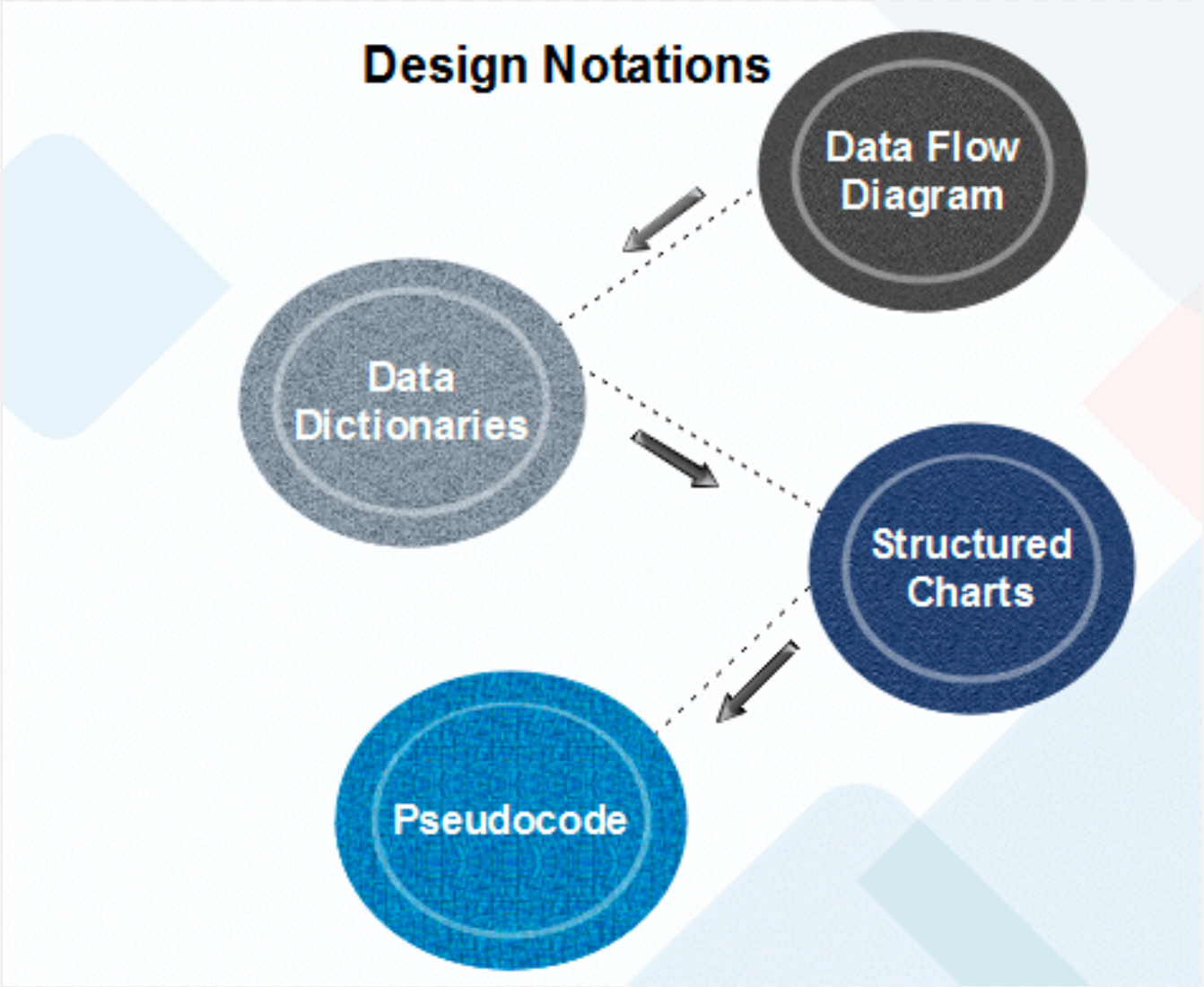
Function oriented design

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.

Generic Procedure

Start with a high-level description of what the software/program does. Refine each part of the description by specifying in greater detail the functionality of each part. These points lead to a Top-Down Structure.


Design Notations
Data Flow Diagram
Data Dictionaries
Structured Charts
Pseudocode

Dataflow diagram

- Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs.

- The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

- Data-flow diagrams are a useful and intuitive way of describing a system.

- They are generally understandable without specialized training, notably if control information is excluded.

- They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

Data Dictionaries

- A data dictionary lists all data elements appearing in the DFD model of a system.

- The data items listed contain all data flows and the contents of all data stores looking on the DFDs in the DFD model of a system.

- A data dictionary lists the objective of all data items and the definition of all composite data elements in terms of their component data items.

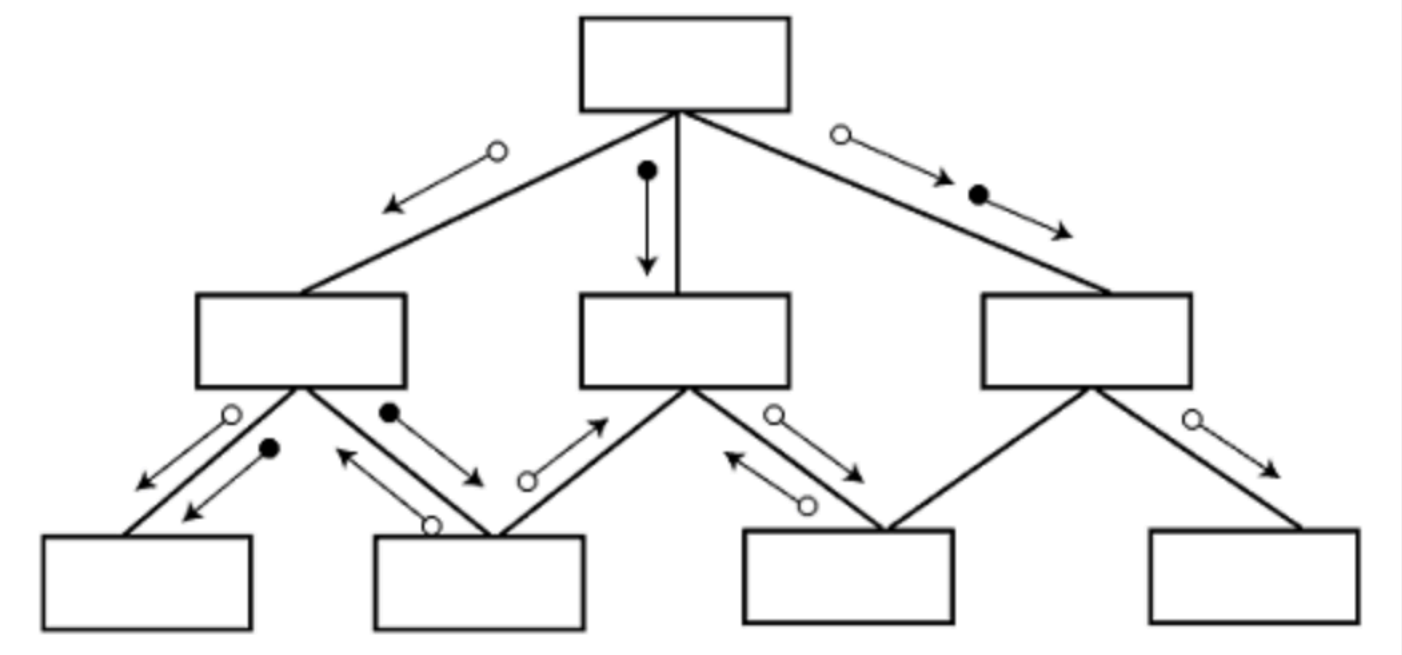
- For the smallest units of data elements, the data dictionary lists their name and their type.

A data dictionary plays a significant role in any software development process because of the following reasons:

- A Data dictionary provides a standard language for all relevant information for use by engineers working in a project.
- A consistent vocabulary for data items is essential since, in large projects, different engineers of the project tend to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of various data structures in terms of their component elements.

Structured Charts

It partitions a system into block boxes. A Black box system that functionality is known to the user without the knowledge of internal design.



Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

The following notations are used in structured chart:

SYMBOL	DESCRIPTION
	Module
	Arrow
	Data couple
	Control Flag
	Loop
	Decision

Pseudocode

Pseudo-code notations can be used in both the preliminary and detailed design phases.

Using pseudo-code, the designer describes system characteristics using short, concise, English Language phases that are structured by keywords such as If-Then-Else, While-Do, and End.

Object oriented design

- The system is viewed as a collection of objects (i.e., entities).

- The state is distributed among the objects, and each object handles its state data.

For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.

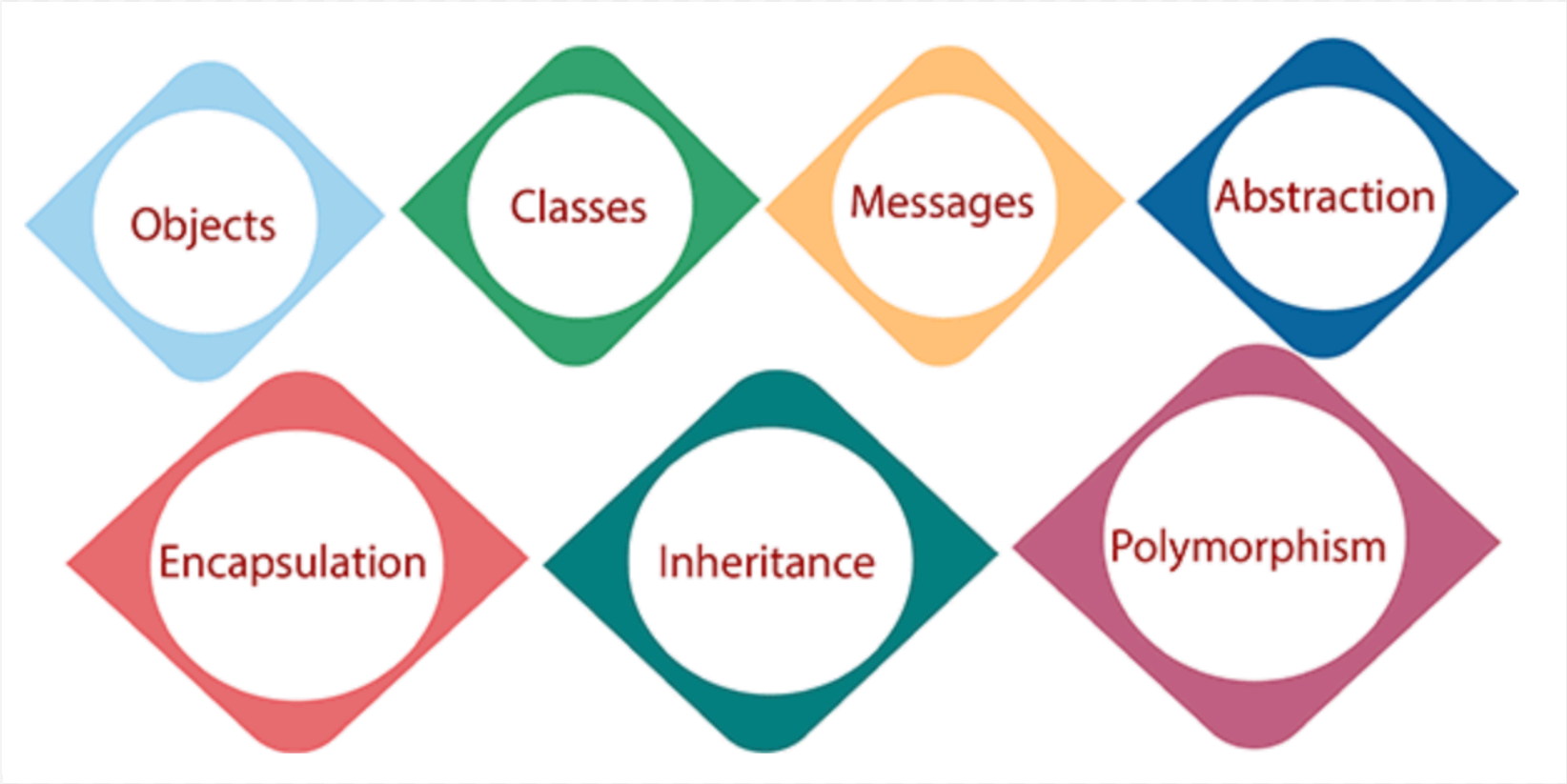
- The tasks defined for one purpose cannot refer or change data of other objects.

- Objects have their internal data which represent their state.

- Similar objects create a class. In other words, each object is a member of some class.

- Classes may inherit features from the superclass.

The different terms related to object design are:



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.

2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

1. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

2. **Encapsulation**: Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

3. **Inheritance**: OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

4. **Polymorphism**: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.