UNIT-3

Software Requirements and Specification

•It is the first step in the development of a system.

•The requirement is nothing but a condition needed by the user to solve a problem or achieve an objective.

•This activity **acquired the knowledge** about the system.

•It **lists out all the requirements** stated by the user.

•Software documentation is also called a Software Requirements Specification (SRS). It is also known as **requirements document.**

•SRS is a formal document, which acts as a **representation of software** that enables the users to review whether it (SRS) is according to their requirements.

Few Characteristics of Software Requirement Specification

•**Correct:** keeping the specification up to date when you find things that are not correct.

•**Unambiguous:** every requirement stated therein has only one interpretation.

•**Consistent:** requirements specification should be constant

•**Verifiable:** every keystroke should provide a user response within 1000 milliseconds.

•**Modifiable:**

•**Traceable:** to connect the requirements in the requirements specification to a higher-level document.

Feasibility Study

The objective of a software feasibility study, is to assess:

•Operational

•Technical

•Economic

•Organizational

These points are thoroughly view to check whether the project is feasible or not.

Types of Feasibility

•**Organizational Feasibility**: it is related to how much the **solution benefits the organization**. There is full understanding and support from the organization’s top management in relation to the project.

•**Operational Feasibility**: it is related to how much the **solution suits the organization working**.

•**Economic Viability**: It is an **analysis between the development cost and the benefits** after the project is implemented (cost-benefit).

•**Technical Software Feasibility**: linked to the **technical support that the organization will offer** for the development of the project.

•**Timeline Feasibility:** crossing between the activities surveyed and the estimated time to carry them out.

•**Others:** legal, cultural, marketing, etc.

Informal/Formal specifications

Specification:-A software requirements specification is a technical document that **describes in detail** the externally visible **characteristics of a software** product.

•A specification can be a

•**Written Document**,

•A Set Of **Graphical Models**,

•A Formal **Mathematical Model,**

•**A Collection Of Usage Scenarios,**

•**A Prototype, Or**

•**Any Combination Of These**.

•For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.

Formal Specifications:

It can be described as:-

- **“Syntax and semantics”** formally defined
- Easy to show completeness and correctness
- Difficult to write

Informal Specifications:

It can be described as:-

- Natural language is formed with ambiguity
- Implementation is often difficult
- Easy to write

Pre/Post conditions

Preconditions and post conditions are widely used in execution models for software processes.

The conditions that control software processes, however, can be complex and difficult to evaluate in the context of ongoing development activities.

Pre/Post conditions

❖ **Pre condition**

These are a conditions that should be satisfied before the operation can be performed.

❖ **Post condition**

These are a conditions that should be satisfied after the operation has been performed.

Example:-Operation On Bank Account

✓ **Pre condition:** Customer should have sufficient amount in account.

✓ **Post condition:** After withdrawal of amount, account must have minimum balance.

Why:s

Conditions can be used in a test-case design method that exercises the logical conditions contained in a program module.

A simple condition is a Boolean variable or a relational expression.

Algebraic specification

•In the algebraic specification technique an **object, class** or type is specified in terms of relationships existing between the operations defined on that type.

•Representation of algebraic specification

•It defines a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous.

•

•Heterogeneous algebras which **includes automata, state machines etc.**

•A homogeneous algebra consists of a single set and several operations; {I, +, -, *, /}.

An algebraic specification is usually presented in four sections:-

1.Types section

In this section, the sorts (or the **data types**) being used is specified.

2.Exceptions section

This section gives the names of the **exceptional conditions** that might occur when different operations are carried out.

3.Syntax section

This section defines the **signatures of the interface procedures**. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator.

4.Equations section

This section gives a set of rewrite **rules (or equations**) defining the meaning of the interface procedures in terms of each other.

Analysis Modeling

Requirements analysis allows software engineers to define user needs early in the development process. It helps them deliver a system that meets customer’s time, budget and quality expectations.

Analysis modeling represents the user’s requirements by depicting the software in three different domains:

- Information Domain

- Functional Domain

- Behavioral Domain

- The analysis model is **designed by a software engineer** or modeler or **system analysts**, or **project manager**.
- To describe the requirement, it uses diagrammatic form and text.

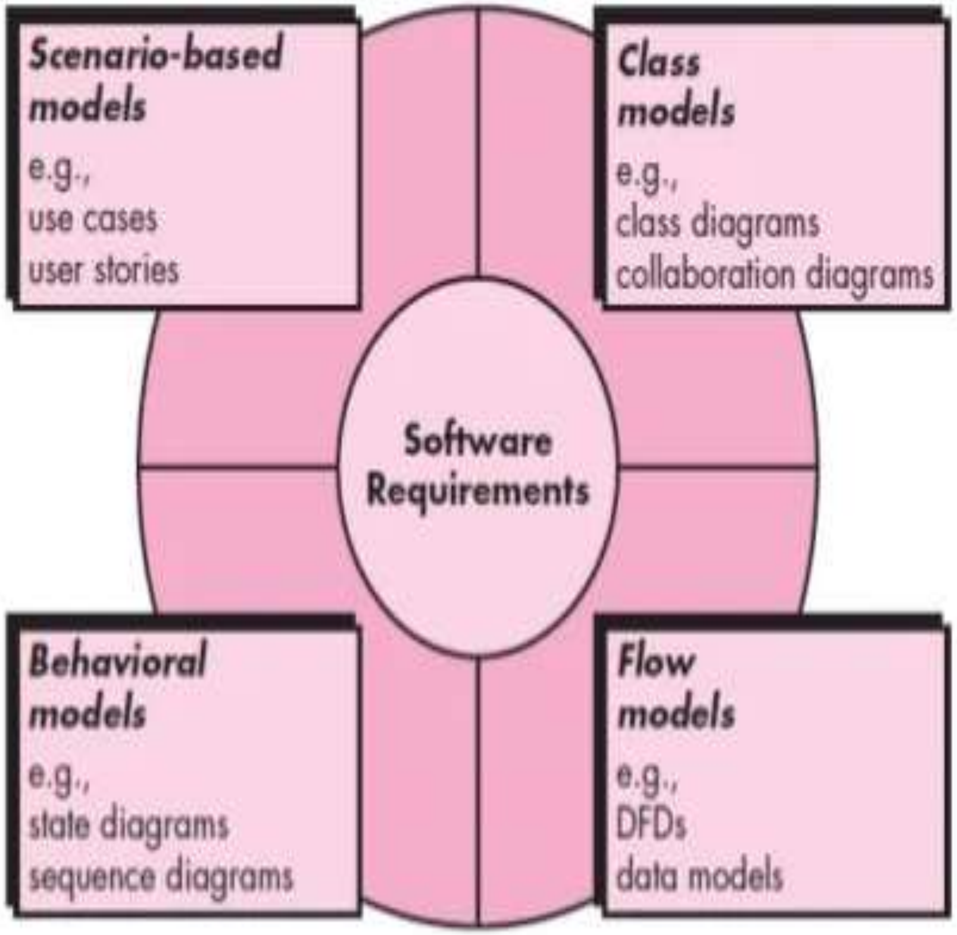
Principles of Analysis Model

•**Represent Information Domain:** The information domain represents the **data flow into** the system from an end-user or external system and devices.

•**Represent Software Functions:** Software functions are responsible for the data flow in the system. It also provides support for those features that are user-visible and they are beneficial to the end-users.

•**Represent Software Behavior:** Software behavior represents **the way the software interacts** with the external environment. Input provided by the users, control data provided by the external system.

•**Partition of Above Three Representations:** The model that depicts information, behavior, and function should be partitioned in a layered manner to get a detailed view of the system.


Scenario-based models
e.g.,
use cases
user stories
Class models
e.g.,
class diagrams
collaboration diagrams
Software Requirements
Behavioral models
e.g.,
state diagrams
sequence diagrams
Flow models
e.g.,
DFDs
data models

Some Rules for Analysis Model

These rules are as follows:

•The model **should focus on requirements** as per business.

•Each element of the analysis model should add to all requirements and provide insight into the system's information domain function and behavior.

•Some models are required more time they need to design after the design process of the software.

•It gives assurance that the analysis model provides value to all stock holders.

•Minimizing coupling through out the system. It is important to represent a relationship between classes and functions. Functions are interconnected to each other very tightly then efforts are put to reduce this interconnectedness.

•It keeps the model as simple as it can be.

Analysis Modeling Approaches

Context Analysis: It is an external viewpoint that shows the overall environment of the system.

Structured Analysis: It shows the architecture of the system or data. It involves the data model like the ER diagram.
It considers data and the processes that transform the data into separate entities called as data objects.
Data objects define the attributes and relationships. It also represents the flow of data objects.

Behavioral Analysis: It shows the behavior of the system.
The analysis model derives modeling elements. The specification of each element may differ from project to project.

Object-Oriented Analysis: It focus on the definition of classes.
It provides object-oriented design methodology such as UML and Unified process analysis.
It performs collaboration of classes to fulfill the user’s requirements.

Elements of Analysis Modeling

Data Dictionary: Data dictionaries are lists of all the names used in the system models; description of the entities, their attributes and relationships are included.

Data Flow Diagram: Data flow diagram model the data flow/processing of the system.

State Transition Diagram: It represents the dynamic model, which shows changes in the state that an object goes through during its lifetime in response to an event.

Entity Relationship Diagram: It consists of information required for each entity or data objects and shows a relationship between objects. It shows the structure of the data in table format.

Data Object Description: It represents the composite information of one entities such that the software understands it.

Software Analysis & Design Tools

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation.
Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.

Structured Approach

Data Modeling -> ERD

Functional Modeling -> DFD

Behavior Modeling -> State chart diagram

Object Oriented Approach (UML)

- Use case
- Class diagram
- Activity diagram
- Sequence diagram
- Deployment diagram
- Component diagram

Data Flow Diagram

•Data flow diagram is graphical representation of flow of data in an information system.

•It represents incoming data flow, outgoing data flow and stored data.

•The DFD does not mention anything about how data flows through the system.

•There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules.

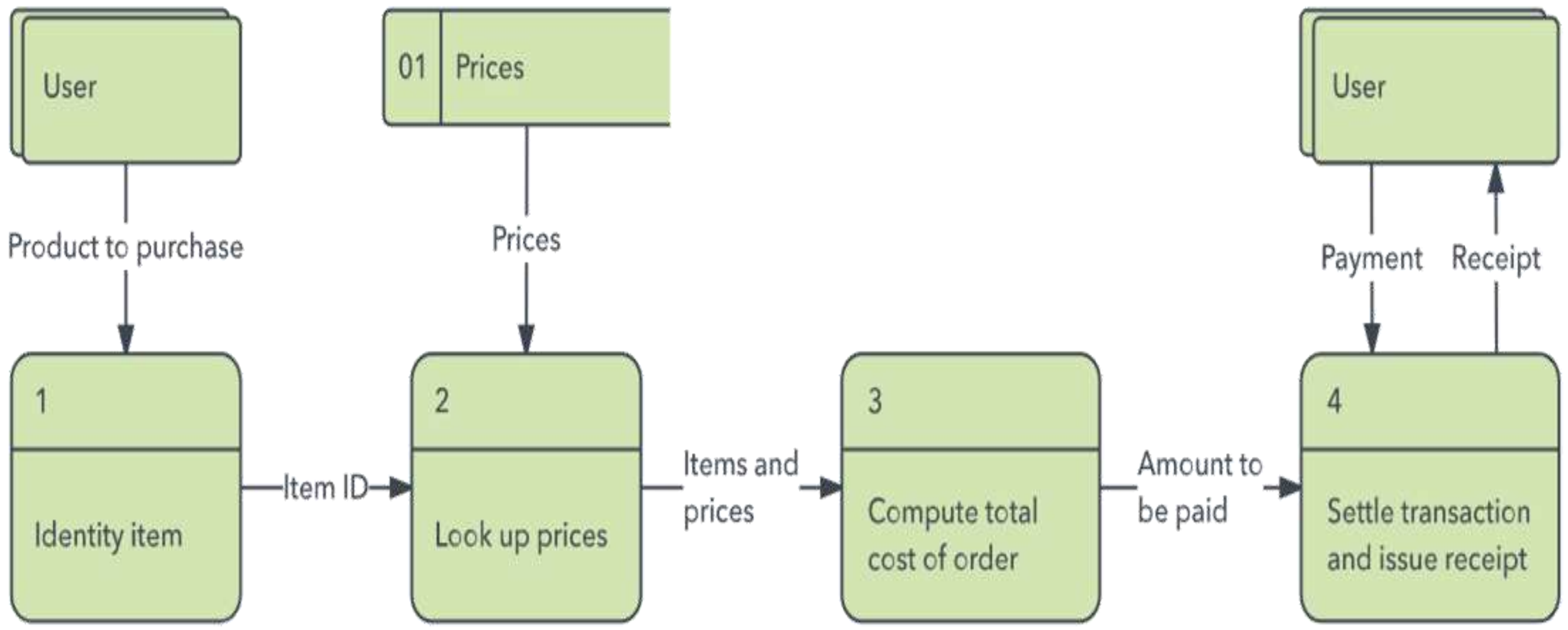
•DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

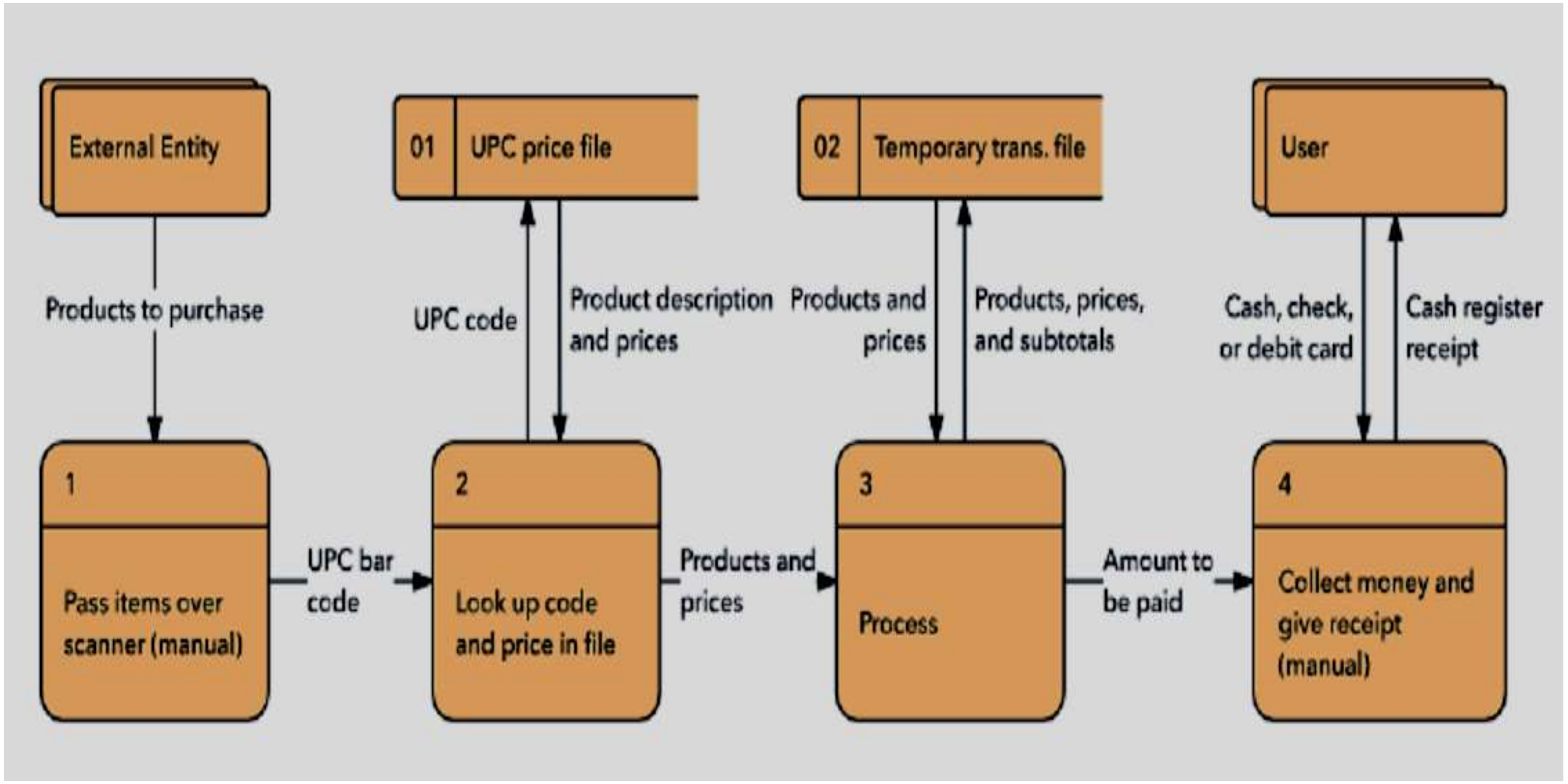
Types of DFD

Data Flow Diagrams are either Logical or Physical.

Logical DFD- This type of DFD concentrates on the **system process**, and flow of data in the system. A logical DFD **focuses on the business and business activities**

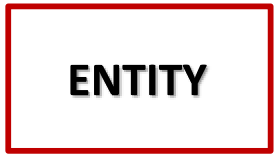
Physical DFD - This type of DFD shows how the **data flow is actually implemented** in the system. It is more specific and close to the implementation. Physical DFD looks at how a system is implemented.


User
01 Prices
User
Product to purchase
Prices
Payment
Receipt
1
Identity item
Item ID
2
Look up prices
Items and prices
3
Compute total cost of order
Amount to be paid
4
Settle transaction and issue receipt


External Entity
Products to purchase
01 UPC price file
UPC code
Product description and prices
02 Temporary trans. file
Products and prices
Products, prices, and subtotals
User
Cash, check, or debit card
Cash register receipt
1 Pass items over scanner (manual)
UPC bar code
2 Look up code and price in file
Products and prices
3 Process
Amount to be paid
4 Collect money and give receipt (manual)

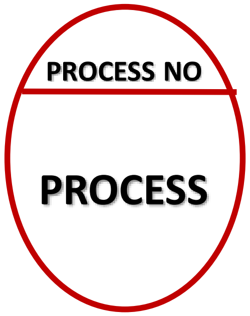
DFD Components

It represent source, destination, storage and flow of data using the following set of components -

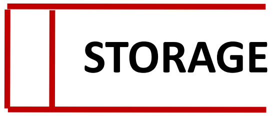


•Entities are source and destination of information data.

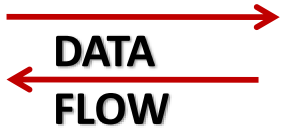
•Entities are represented by a rectangles with their respective names.



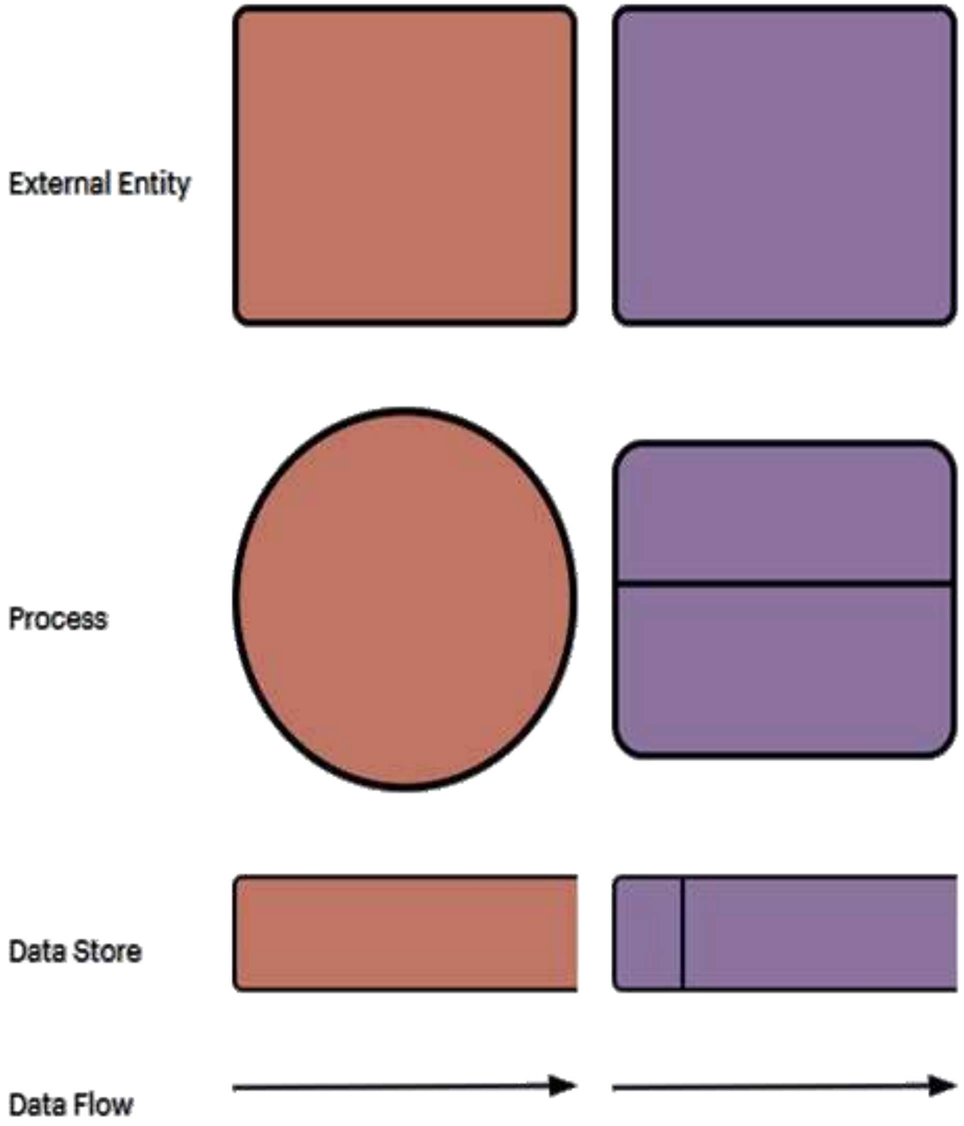
Activities/Process/Action taken on the data are represented by Circle or Round-edged rectangles.



It can be represent in two ways – a) as a rectangle with absence of both smaller sides b) as an open-sided rectangle with only one side missing.

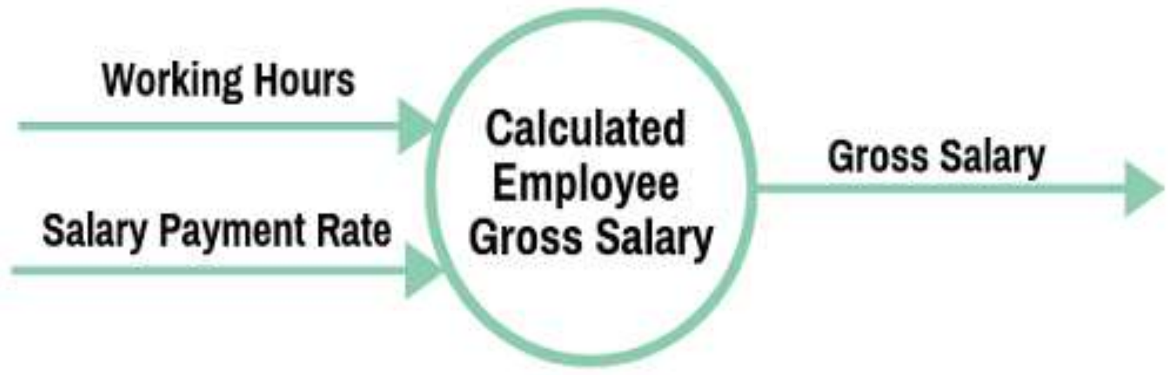


It shows movement of data

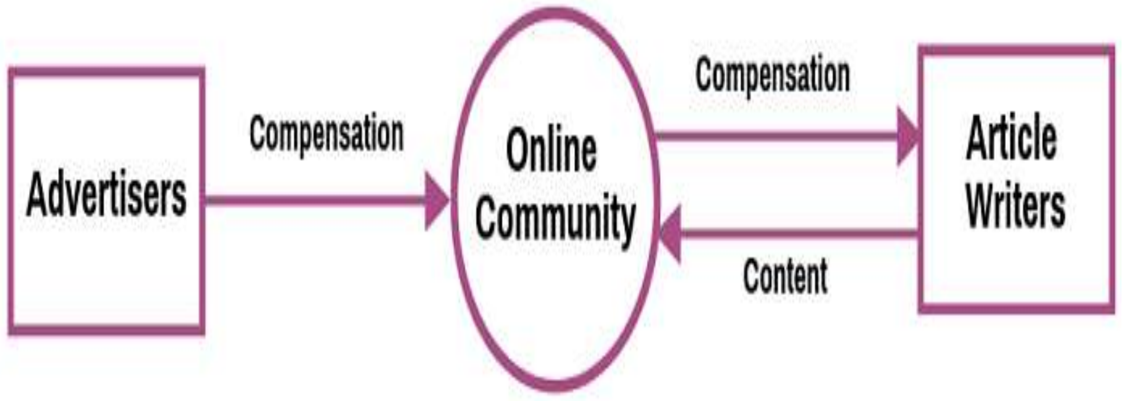

External Entity
Process
Data Store
Data Flow

DFD Rules

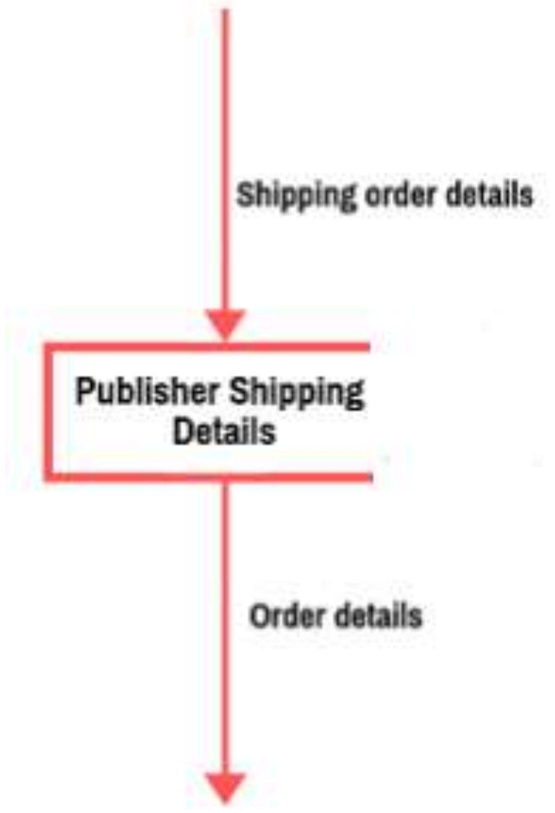
•Each **process** should have at least one input and an output.



Each process can go to any other symbol (other processes, data store, and entities)



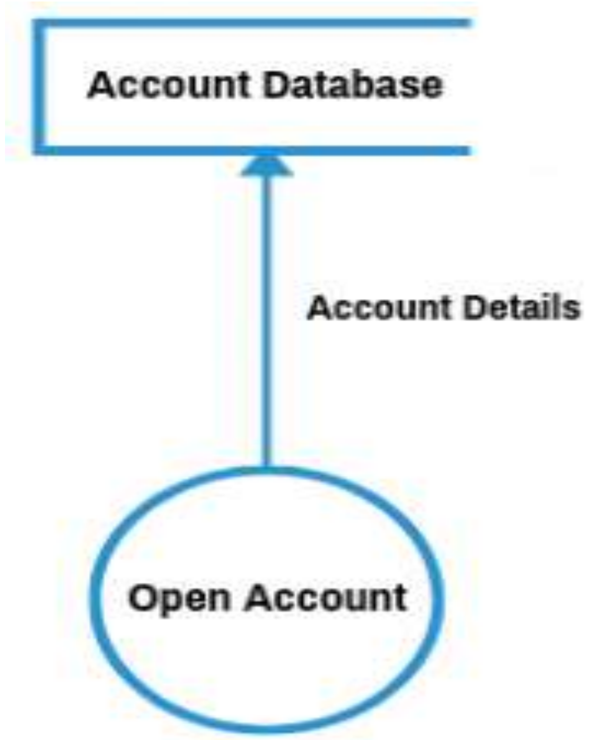
•Each data store should have at least one data flow in and one data flow out.



•Data stored in a system must go through a process.

•All processes in a DFD go to another process or a data store.

Entities must be connected to a process by a data flow.



- Data flows cannot cross with each other.
- Data stores cannot be connected to external entities.

Levels of DFD

The first step in creating DFDs is to identify the DFD elements (Entities, Processes, Data stores, and Data flows).

The next steps involve creating the different levels of the DFDs.

The highest level DFD depicts the synopsis of the system and with decreasing levels of the DFDs, detailed explanations of each segment of the whole process are described.

The following are the levels of data flow diagrams:

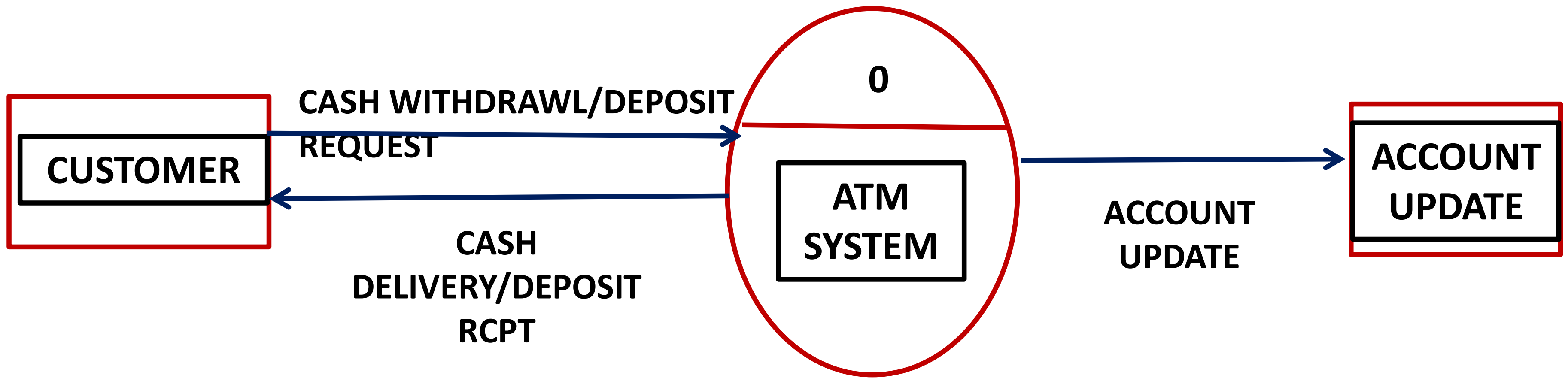
- 0-level DFD
- 1-level DFD
- 2-level DFD

Context Diagrams – Level 0

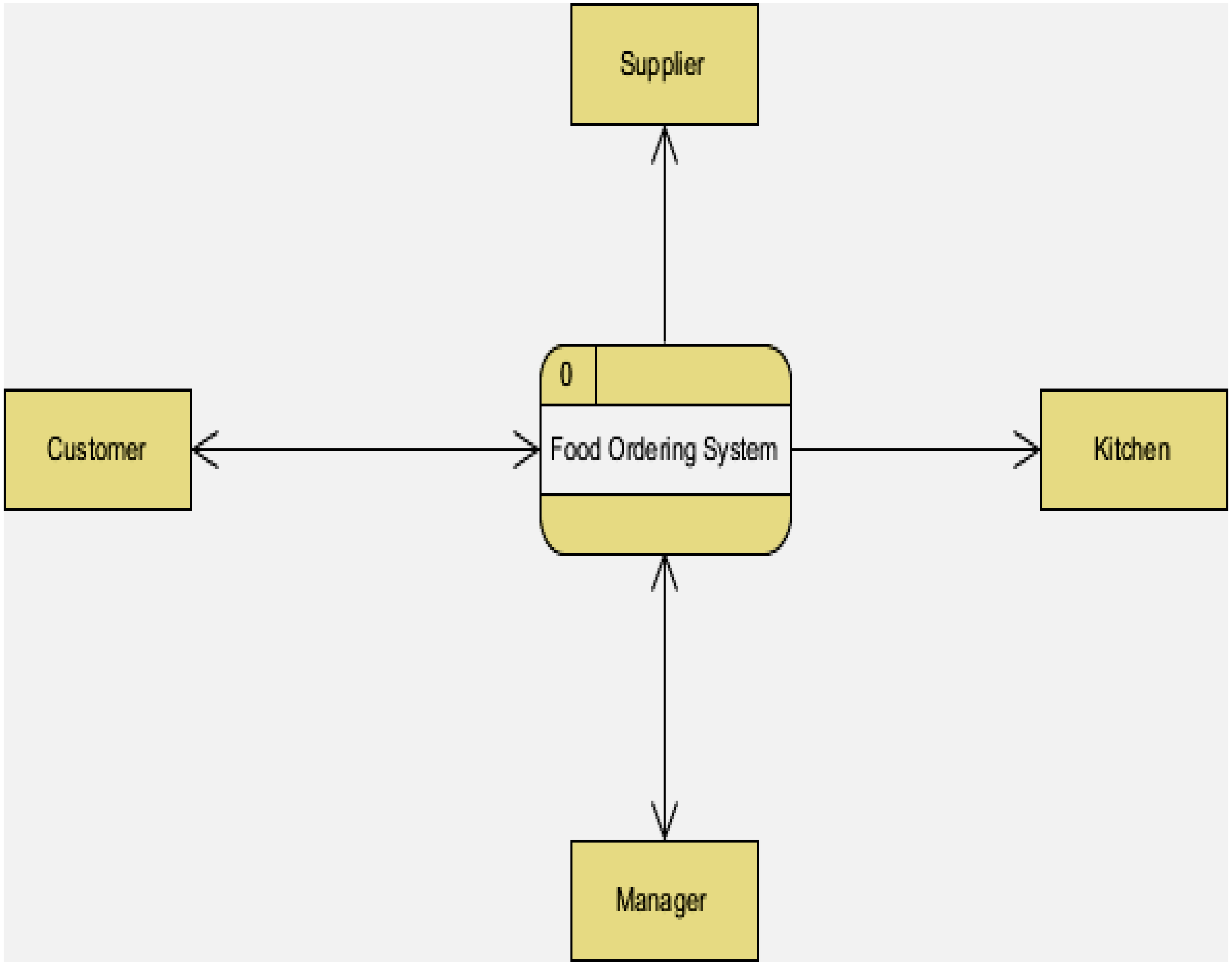
•These diagrams represent the outermost level.

•It gives an overview of the system.

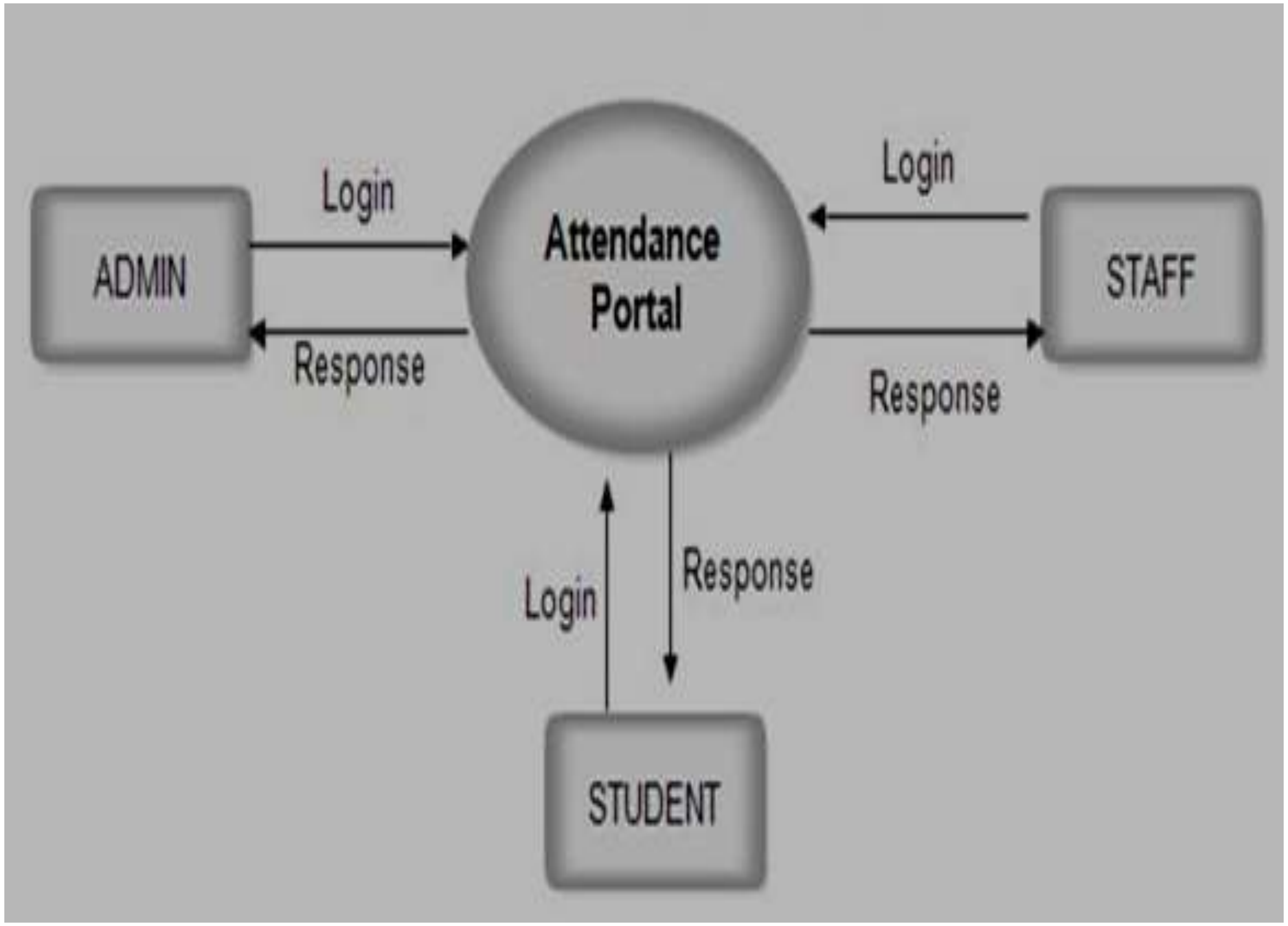
•They show only one process i.e. the entire system and data to and from the external entities.



Context Diagrams – Level 0



Context Diagrams – Level 0



Try to Design 0 Level DFD

1. School Management System
2. Hospital Management System
3. Admission Management System
4. Restaurant Management System
5. Hotel Management System
6. Library Management System
7. Result Analysis Management System
8. Ticket Booking Management System
9. Pathology Management System
10. Gym Management System
11. Police Station Management System
12. Physiotherapy Clinic Management System
13. Car Sales Showroom Management System
14. School Management System
15. Coaching Class Management System
16. Bank Management System

Try to Design 0 Level DFD

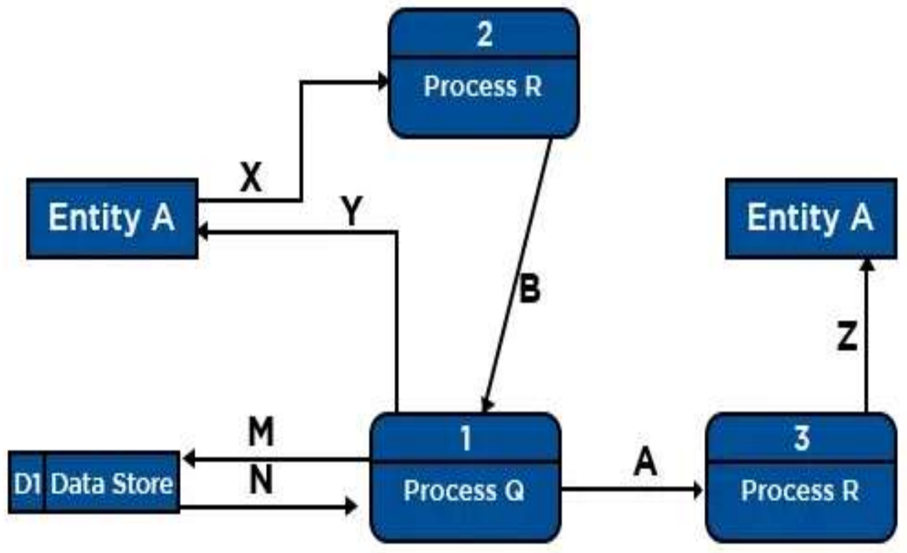
17. Competitive Exam Management System
18. Inventory Management System
19. Employee Management System
20. Production Management System
21. Garage Management System

Level – 1 Data Flow Diagrams

•This level DFD decomposes each parent process of the Level – 0 into more details into child/sub processes.

•It also contains data stores, external entities, and data flows.

•An example of numbering for a process is 1.1.



Level – 2 Data Flow Diagrams

This level further decomposes the child/sub process depicted in Level – 1 DFD into sub-sub processes.

An example of numbering for a process is 1.1.1.

:An Example:
DFD: For A Clothes Ordering System

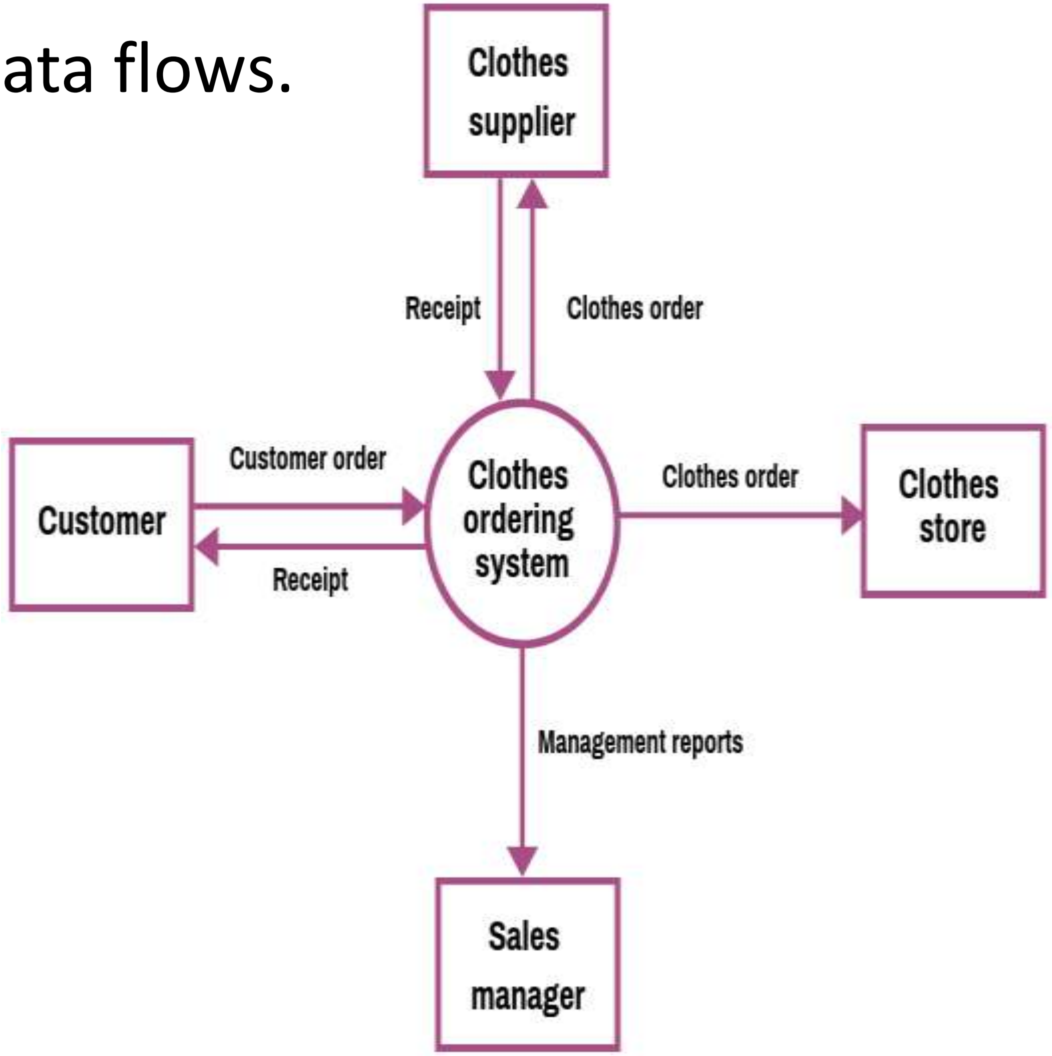
Steps for creating a context DFD:

Step1: Define the process.

Step2: Create a list of all external entities (all people and systems).

Step3: Create a list of the data flows.

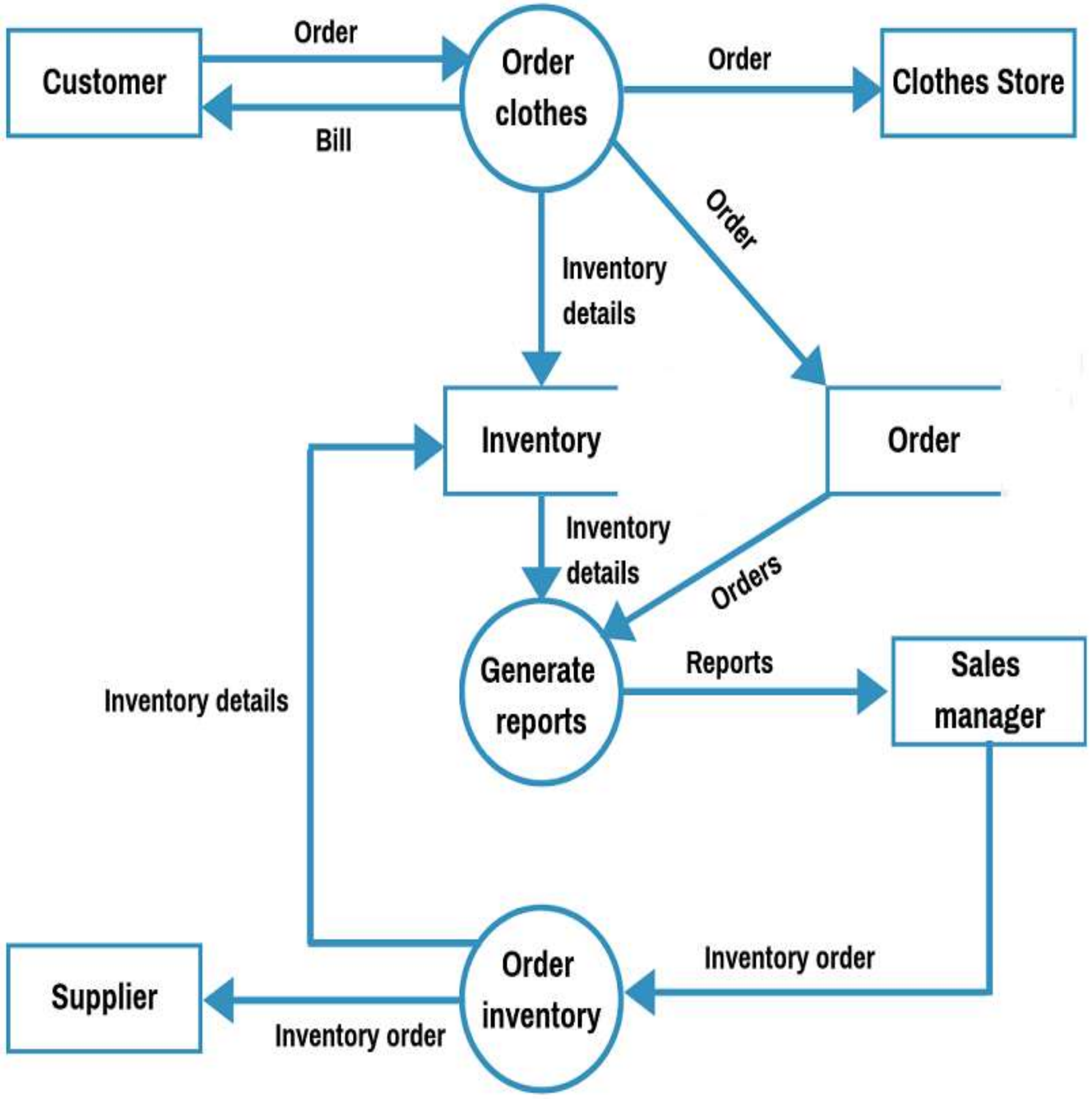
Step4: Draw the diagram.



Level 1 Data Flow Diagram

Context level DFD contains only one process and does not illustrate any data store.
This is the main difference with level 1 DFD.

Steps for creating a context DFD:
Step1: Define the processes (the main process and the sub processes).
Step2: Create a list of all external entities (all people and systems).
Step3: Create a list of the data stores.
Step4: Create a list of the data flows.
Step5: Draw the diagram.


Customer
Order
Bill
Order clothes
Order
Clothes Store
Order
Inventory details
Inventory
Order
Inventory details
Orders
Generate reports
Reports
Sales manager
Inventory details
Order inventory
Inventory order
Supplier
Inventory order

Structure Charts

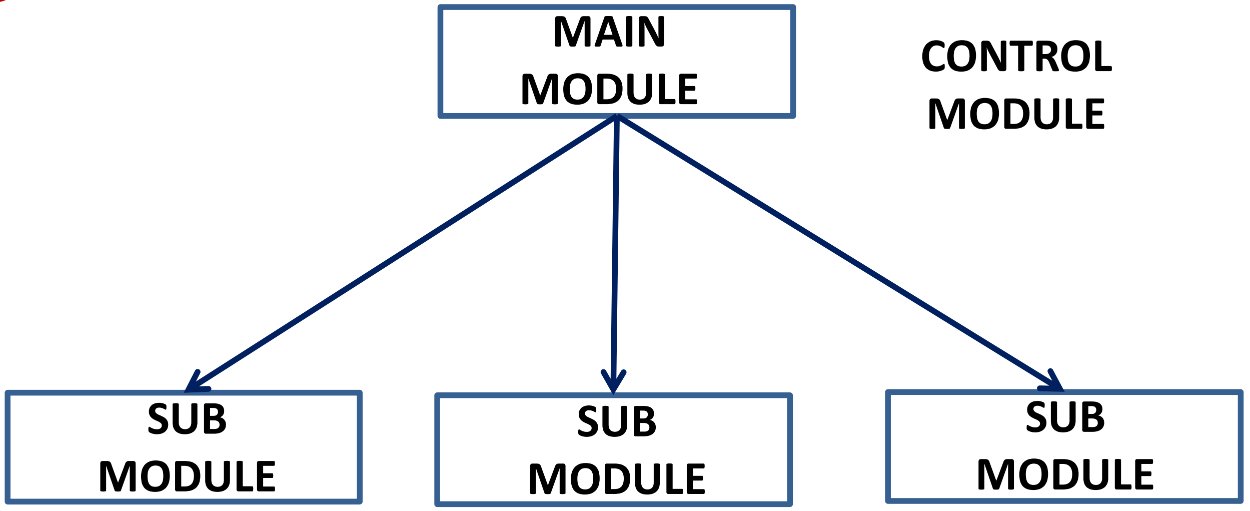
•Structure chart is a chart derived from Data Flow Diagram.

•It represents the system in more detail than DFD.

•It breaks down the entire system into lowest functional modules.

•Describes functions and sub-functions of each module of the system to a greater detail than DFD.

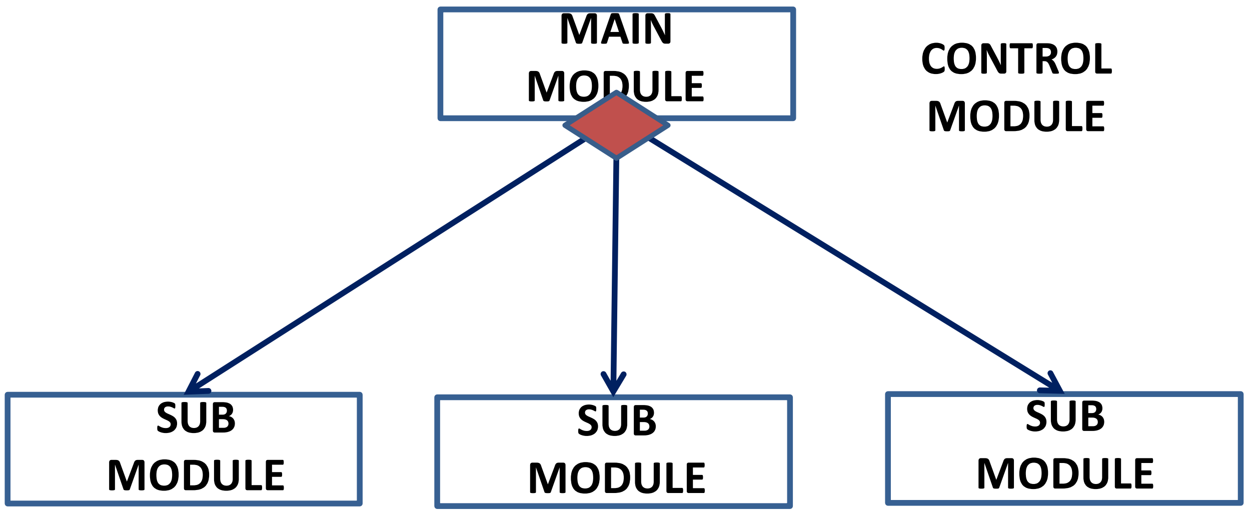
Module



Module - It represents process or subroutine or task.

A main module branches to more than one sub-module.

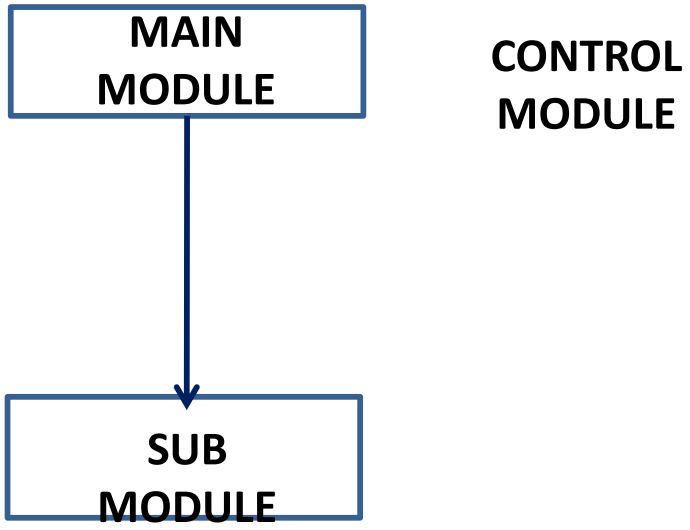
Condition



Condition - It is represented by **small diamond** at the base of module.

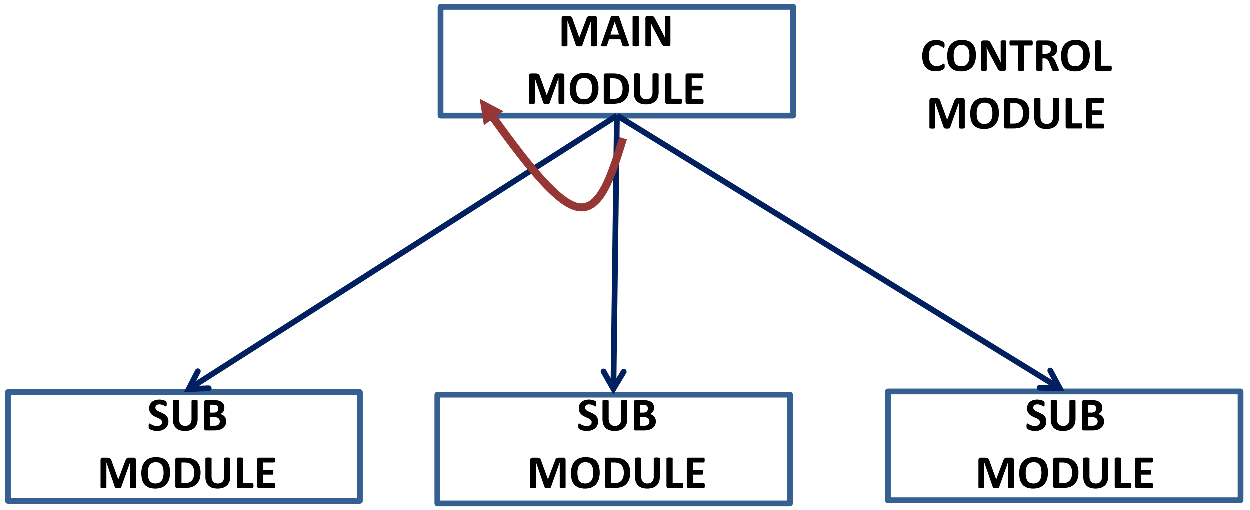
It depicts that control module can select any of sub-routine based on some condition.

Jump



Jump - An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.

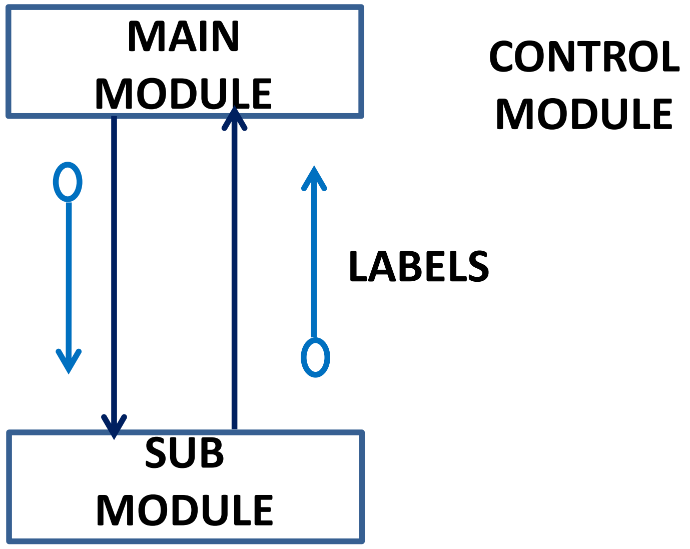
Loop



Loop - A curved arrow represents loop in the module.
All sub-modules covered by loop repeat execution of module.

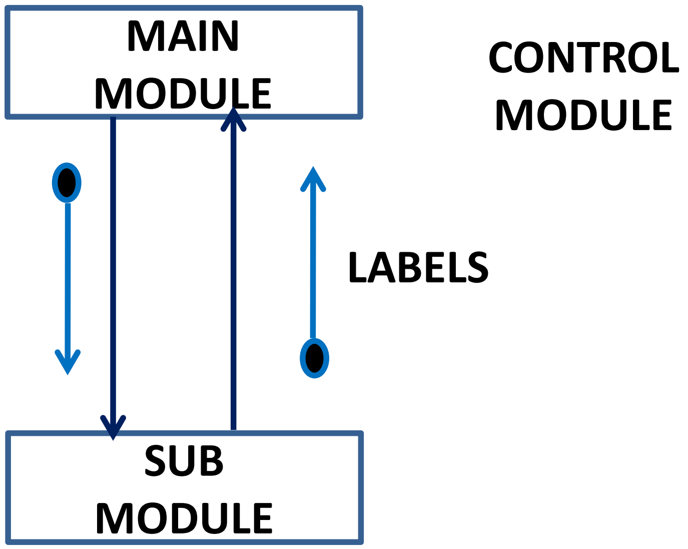
Data Flow

Data Flow - A directed arrow with empty circle at the end represents data flow.



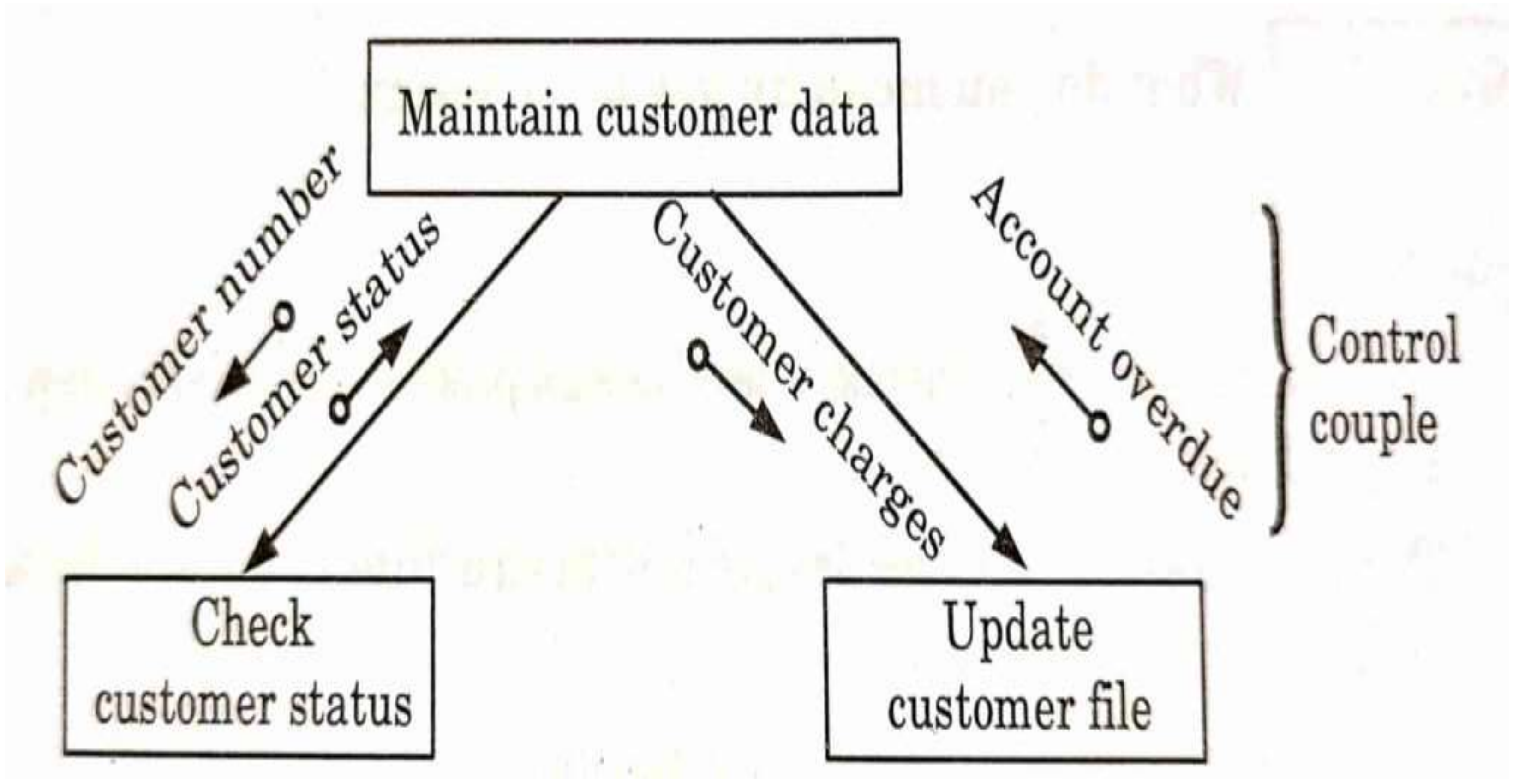
Control Flow

Control Flow - A directed arrow with filled circle at the end represents control flow.



Structure Chart

The symbols used in creation of structure charts are



Structured English

•Software team has to provide accurate information to the programmers to develop accurate code.

•Graphs or diagrams, may are sometimes interpreted differently by different people.

•Analysts and designers use Structured English. It provide what is required and how to code it.

•Structured English is the It **uses plain English words** in structured programming paradigm.

IF-THEN-ELSE, DO-WHILE

Analyst uses the same variable and data name, which are stored in Data Dictionary, making it much simpler to write and understand the code.

Customer Authentication in the Bank application

This procedure to authenticate customer can be written in Structured English as:

```
Enter Customer_Name and Passward

SEEK Customer_Name and Passward in Customer_detail file

          IF Customer_Name and Passward matches THEN

                    Call procedure AUTHENTICATE_USER ()

          ELSE PRINT error message

                    Call procedure NEW_CUSTOMER_REGISTARTION()

          ENDIF
```

Pseudo-Code

- It is written more close to programming language.
- It may be considered as improved programming language, full of comments and descriptions.
- **It avoids variable declaration** but they are written using some actual programming language's constructs, like C, C++ etc.
- It contains more programming details than Structured English.
- It provides a method to perform the task, as if a computer is executing the code.

Pseudo-Code To Add Two Numbers

```
Start Program
Input  two numbers a
and b
sum=a+b
Print sum
End Program
```

```
BEGIN
NUMBER n1, n2, sum
OUTPUT("Input number1:")
INPUT n1
OUTPUT("Input number2:")
INPUT n2
sum=n1+n2
OUTPUT sum
END
```

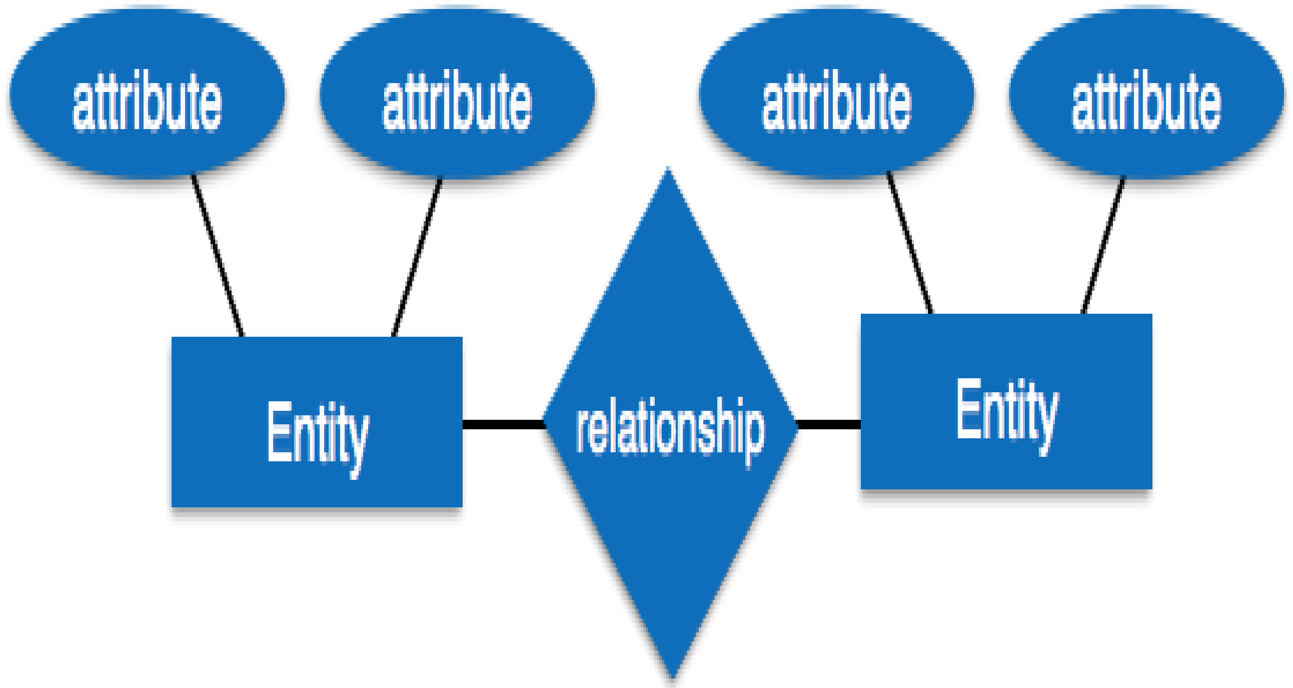
Entity-Relationship Model

•E-R model is a type of database model based on the view of real world entities and relationship among them.

•ER Model creates a set of entities with their attributes, a set of constraints and relation among them.

•ER Model is best used for the conceptual design of database.

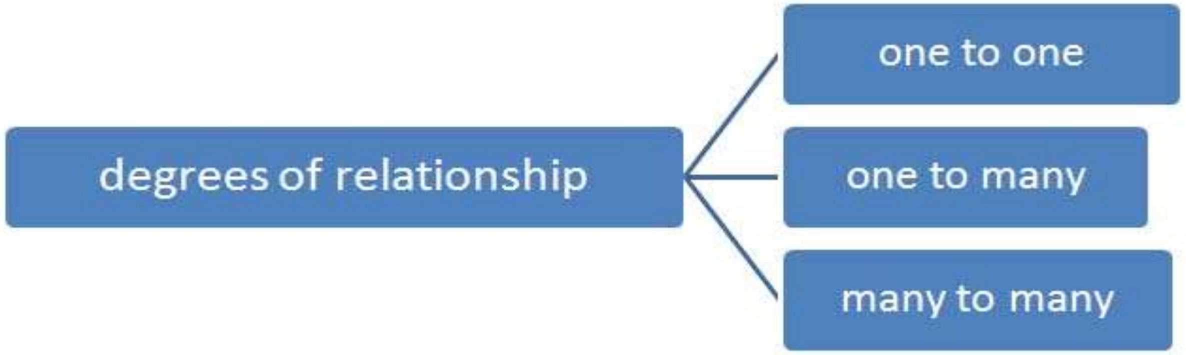
•ER Model can be represented as follows



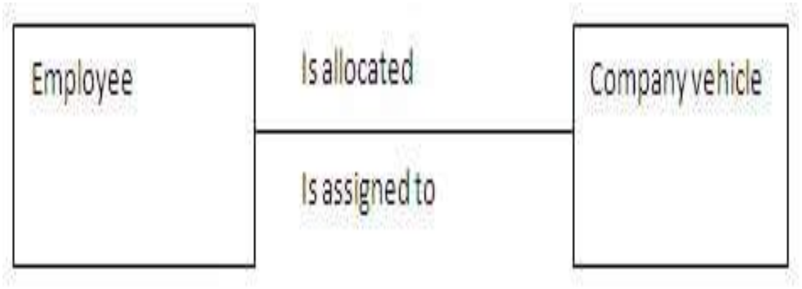
- **Entity** - An entity in ER Model is a real world being, which has some properties called ***attributes.***
- Every attribute is defined by its corresponding set of values, called ***domain***.
- For example, Consider a college database.

Here, a student is an entity. Student has various attributes like name, roll no, age, email, class etc.

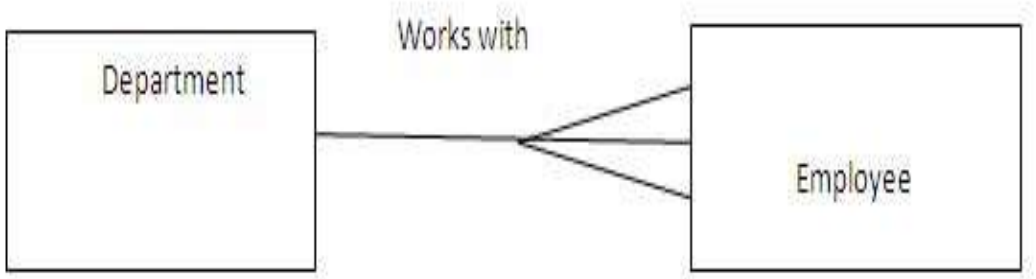
- **Relationship** - The logical association among entities is called ***relationship***.
- Relationships are mapped with entities in various ways.
- Mapping cardinalities define the number of associations between two entities.



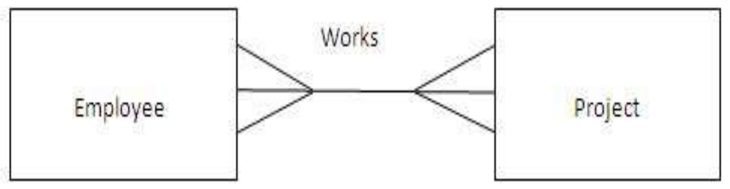
One-to-One (1: 1): An employee is allotted an organization vehicle. Therefore, there's a one-to-one relationship between employee and company vehicle.



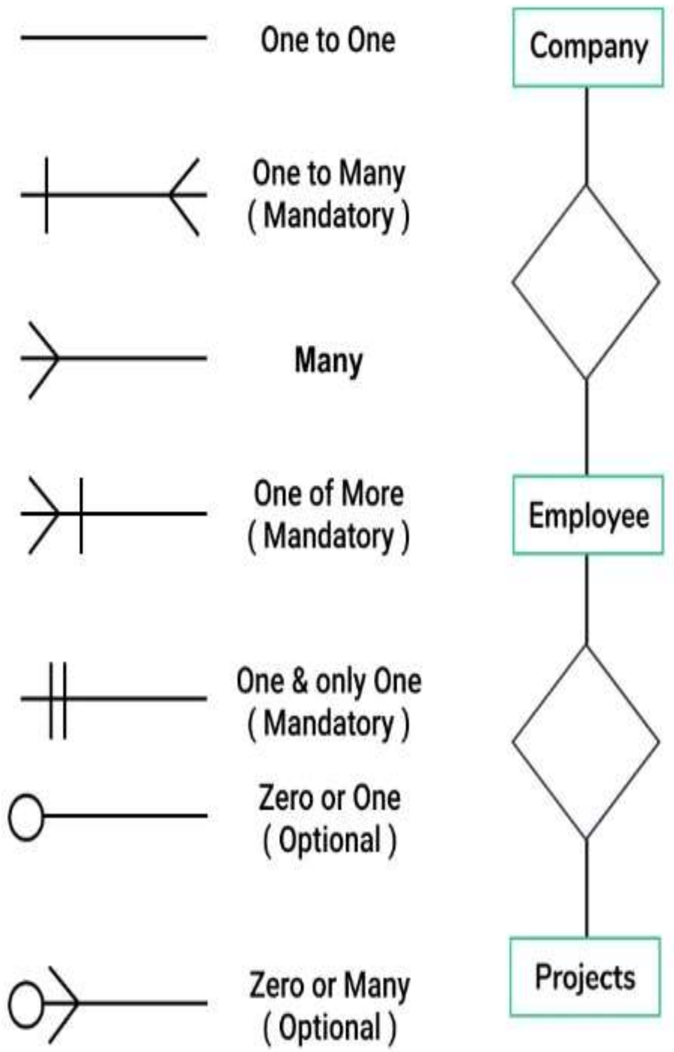
One-to-Many(1: M): In this relationship, one occurrence of an entity relates to many occurrences in another entity.



Many-to-Many (M: M): This many-to-many relationship can be seen between project and employee. An employee works on many projects and at the same time, a project has several employees.



Relationship Cardinality Notations


One to One
One to Many (Mandatory)
Many
One of More (Mandatory)
One & only One (Mandatory)
Zero or One (Optional)
Zero or Many (Optional)
Company
Employee
Projects

Data Dictionary

•It is the centralized collection of information about data.

•It stores meaning and origin of data, its relationship with other data, data format for usage etc.

•It is often referenced as meta-data (data about data) repository.

•It is created along with DFD (Data Flow Diagram) and is expected to be updated whenever DFD is changed or updated.

Data dictionary should contain information about the following

✓**Data Flow**-described by means of DFDs

✓**Data Structure**

✓**Data Elements**-consist of Name and descriptions of Data and Control Items, Internal or External data stores

✓**Data Stores**-It stores the information from where the data enters into the system and exists out of the system.

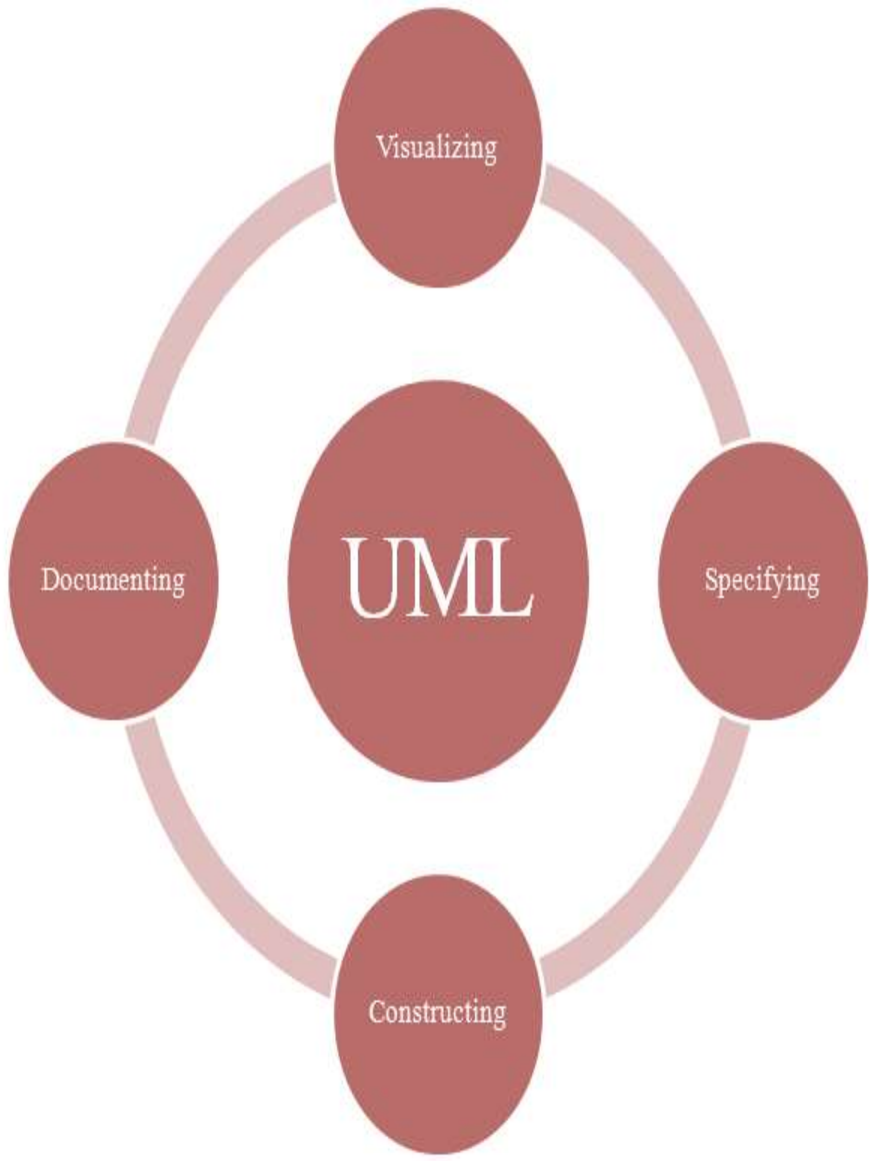
✓**Data Processing**

Object Oriented Approach (UML)

The Object Oriented (OO) approach uses Unified Modeling Language (UML) diagrams for documenting the analysis and design stage, and languages supporting the OO paradigms (C++, Java, C#, etc.) for the programming stage.

To understand and use UML software developers should be familiar with methods of **Object-Oriented Analysis and Design** (**OOAD**) and/or of the **Object-Oriented Development** (**OOD**).

There are several fundamental concepts defining OOD but there is no agreement on the exact list of the concepts, their definition and taxonomy (classification).
Some of OOD concepts that are relevant to the UML:
•Class and Object
•Message, Operation, Method
•Encapsulation
•Abstraction
•Inheritance
•Polymorphism


Visualizing
Documenting
UML
Specifying
Constructing

UML diagrams are classified into three categories that are given below:

- Structural Diagram
- Behavioral Diagram
- Interaction Diagram

- **Structural Diagram:** (Nouns Of UML – Static Parts)

It represents the static view of a system by showing the structure of a system.

Following are the structural diagrams given below:

- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Deployment diagram

•**Behavioral Diagram:** (verbs of UML – Dynamic Parts)

•It depicts the behavioral features of a system.

•It deals with dynamic parts of the system.

It encompasses the following diagrams:

•*Activity diagram*

•*State machine diagram*

•*Use case diagram*

•**Interaction diagram:**

•It is a subset of behavioral diagrams.

•It depicts the interaction between two objects and the data flow between them.

Following are the several interaction diagrams in UML:

•*Timing diagram*

•*Sequence diagram*

•*Collaboration diagram*

The conceptual model of UML contains the fundamentals of UML.
The conceptual model consists of three parts. They are:

1) Building blocks of UML (syntax / vocabulary)
2) Rules (semantics)
3) Common Mechanisms

The building blocks of UML contains three types of elements. They are:

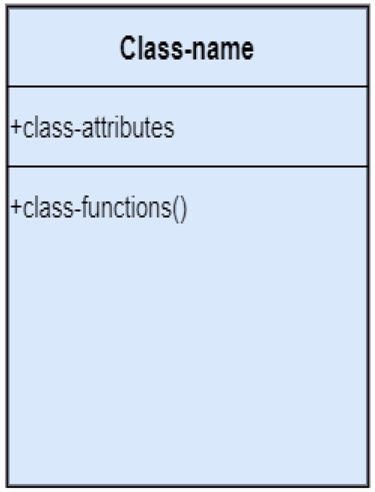
1) Things (Object Oriented parts of UML) [Class, Use Case, Component, Node]

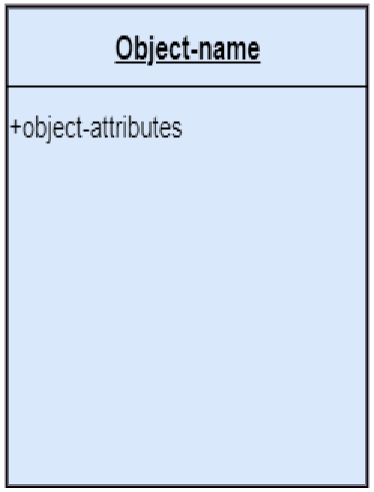
2) Relationships (relational parts of UML) [Interaction, State]

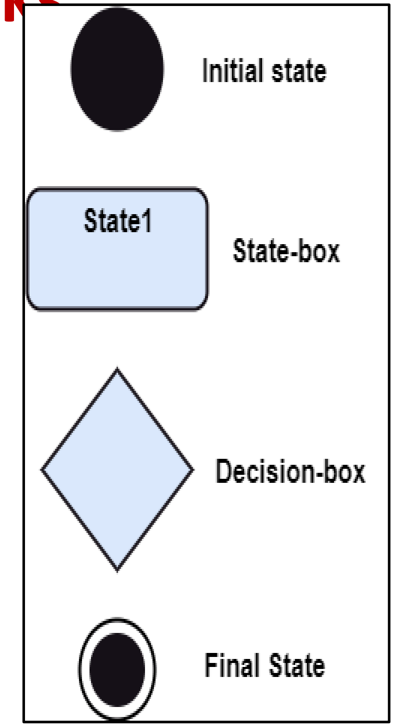
3) Diagrams :

[1) Class diagram 2) Object diagram 3) Use case diagram
4) Component diagram 5) Deployment diagram 6) Sequence diagram
7) Collaboration diagram 8) State chart diagram and 9) Activity diagram]

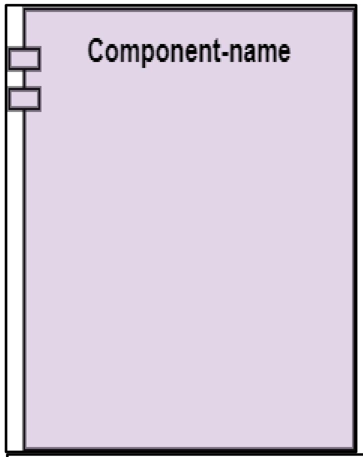
UML-Common Building Blocks



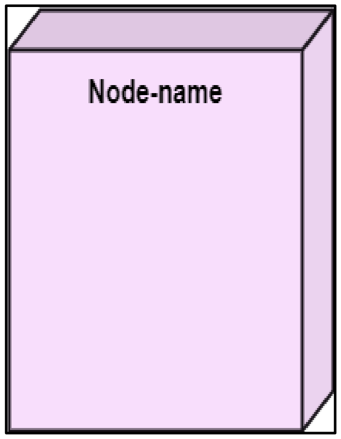




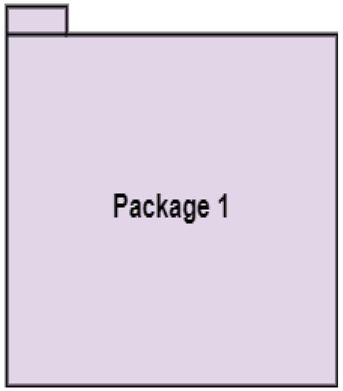
State Machine: It defines a sequence of states that an entity goes through in the software development lifecycle.

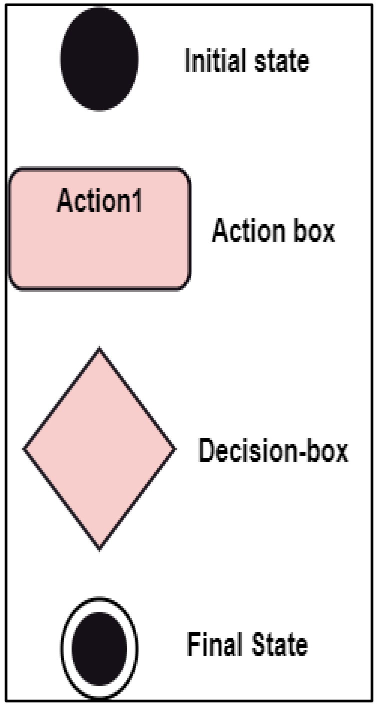


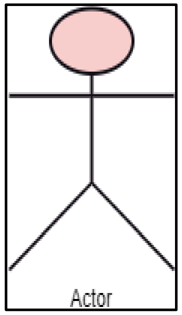
These are executable files, dll files, database tables, files and documents.



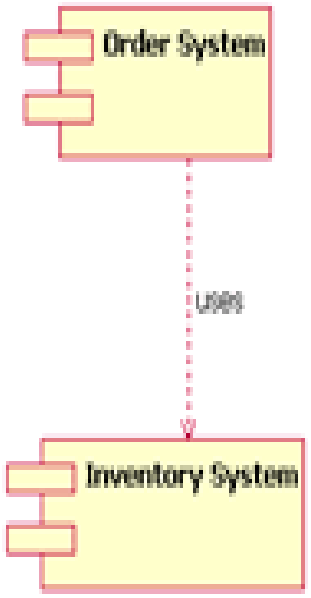
Node can be physical element e.g. PC, LAPTOP, Server





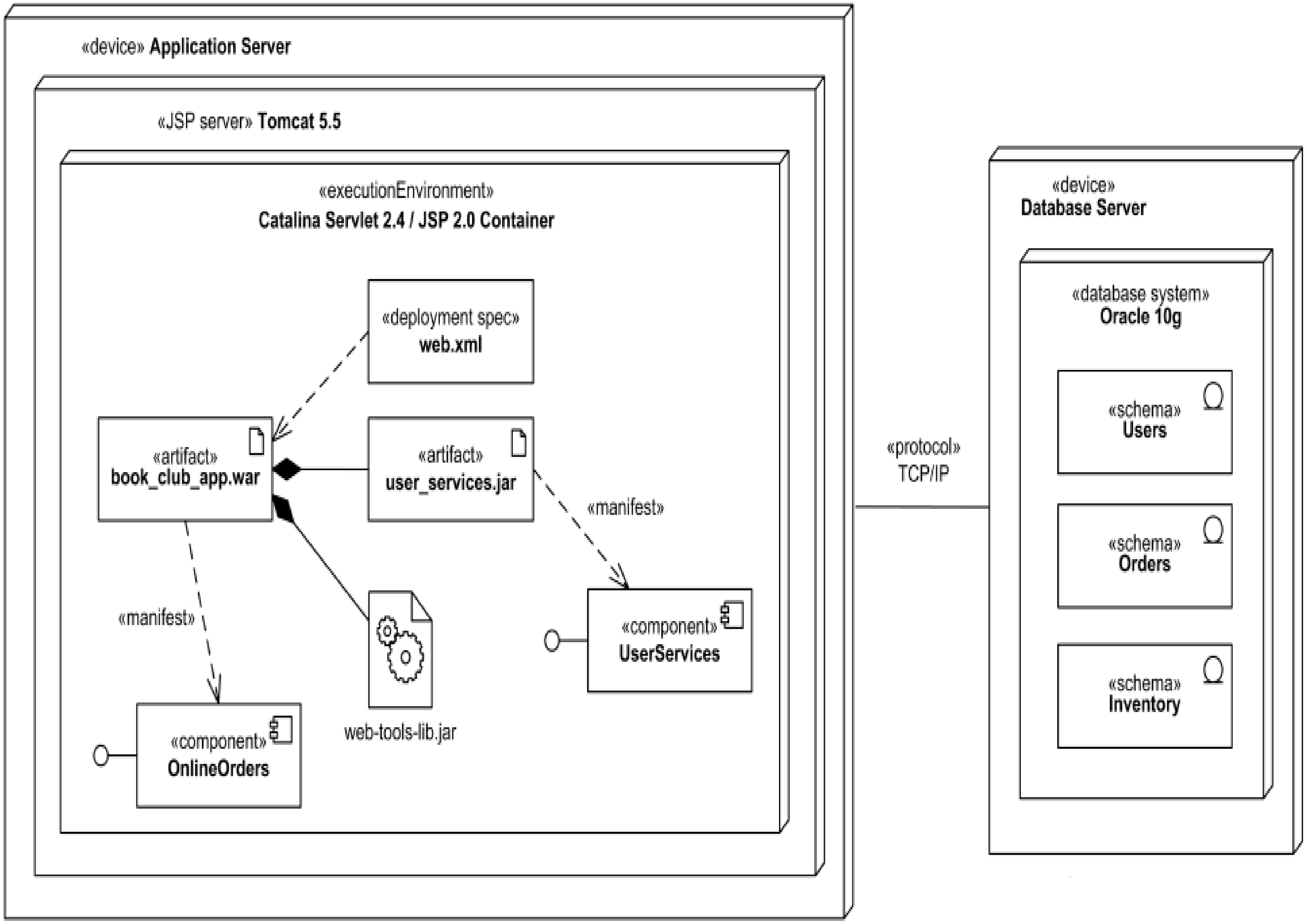


The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system.

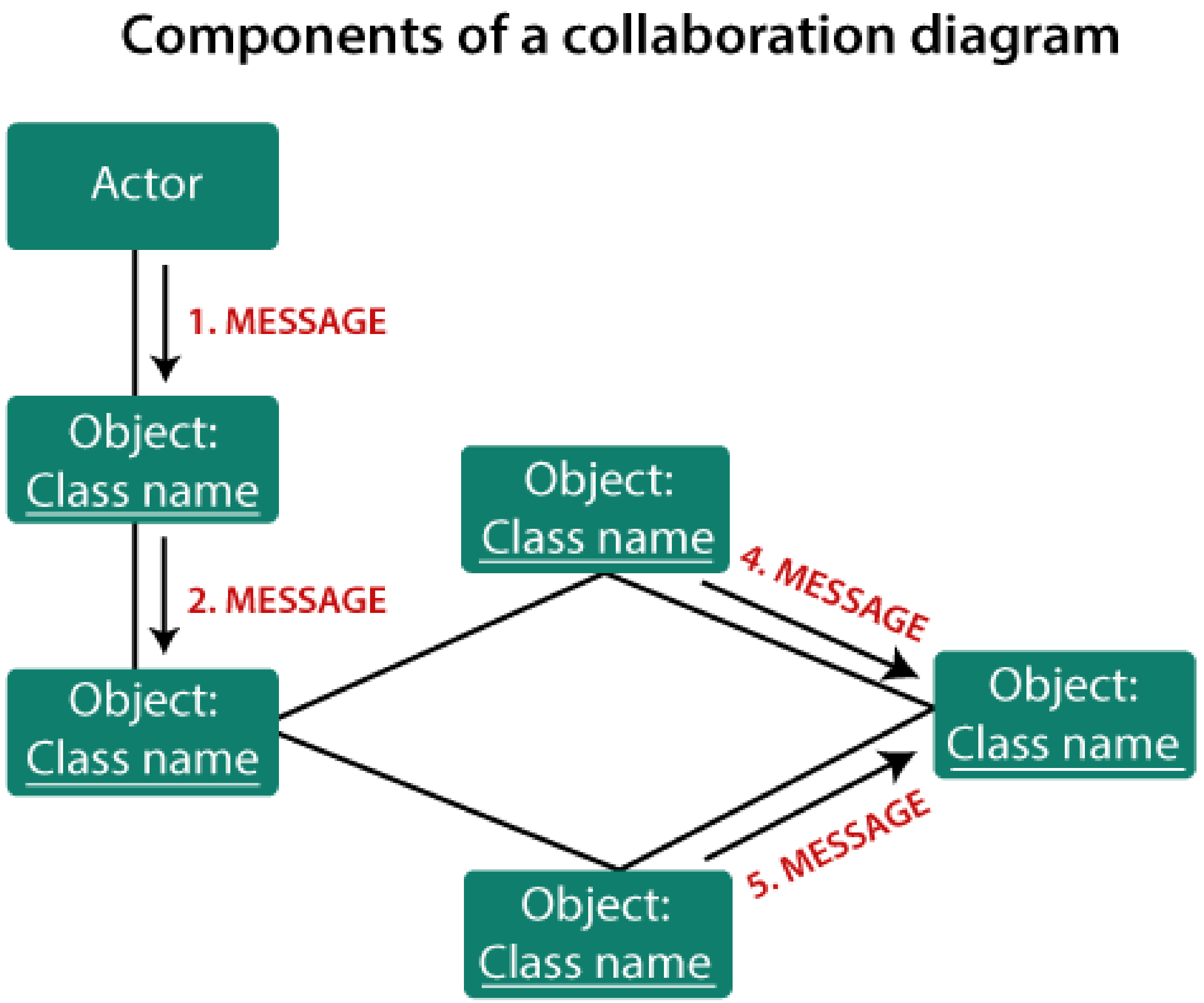


Activity Diagram: It portrays all the activities accomplished by different entities of a system.

Deployment Diagram

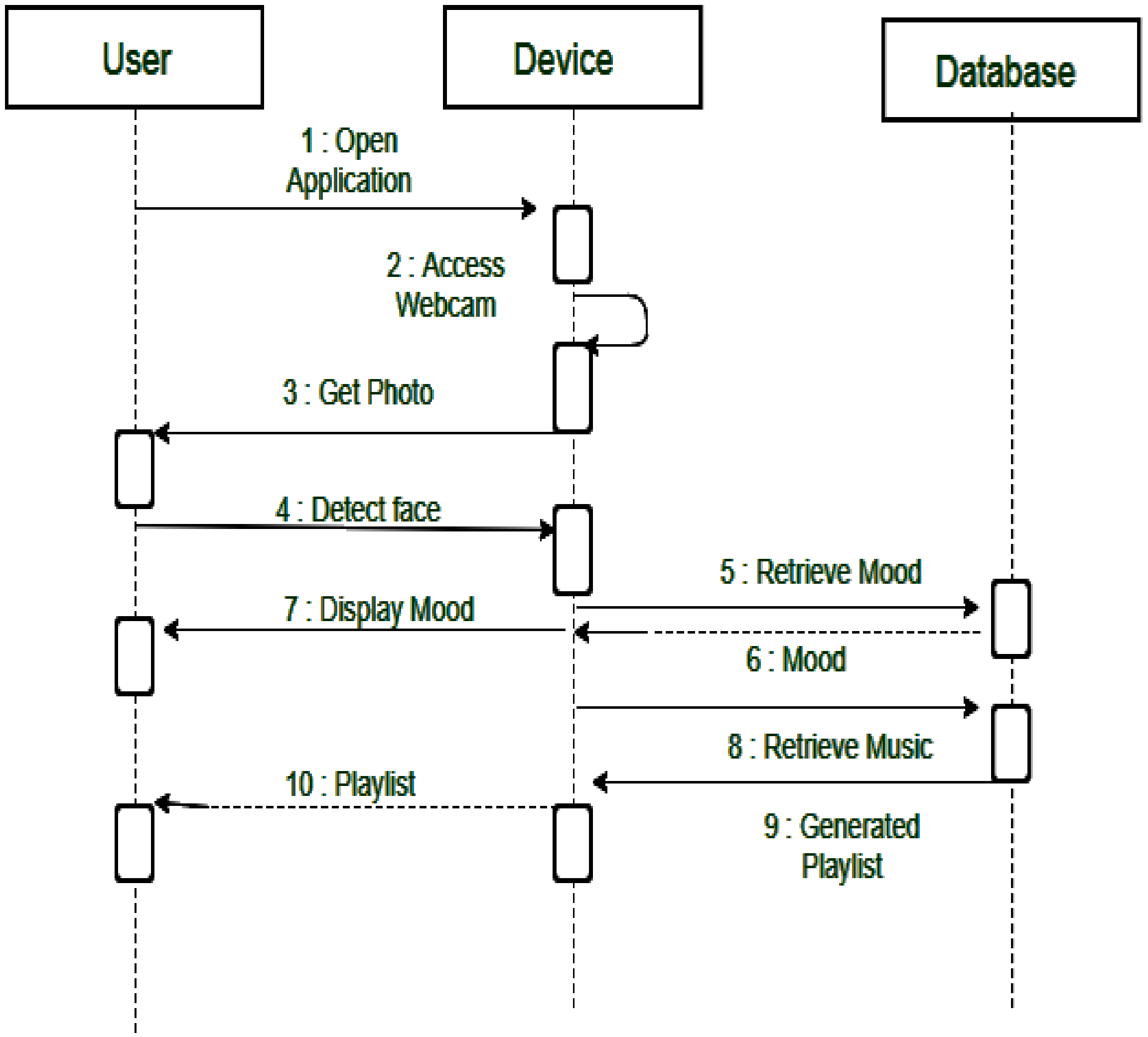


Interaction



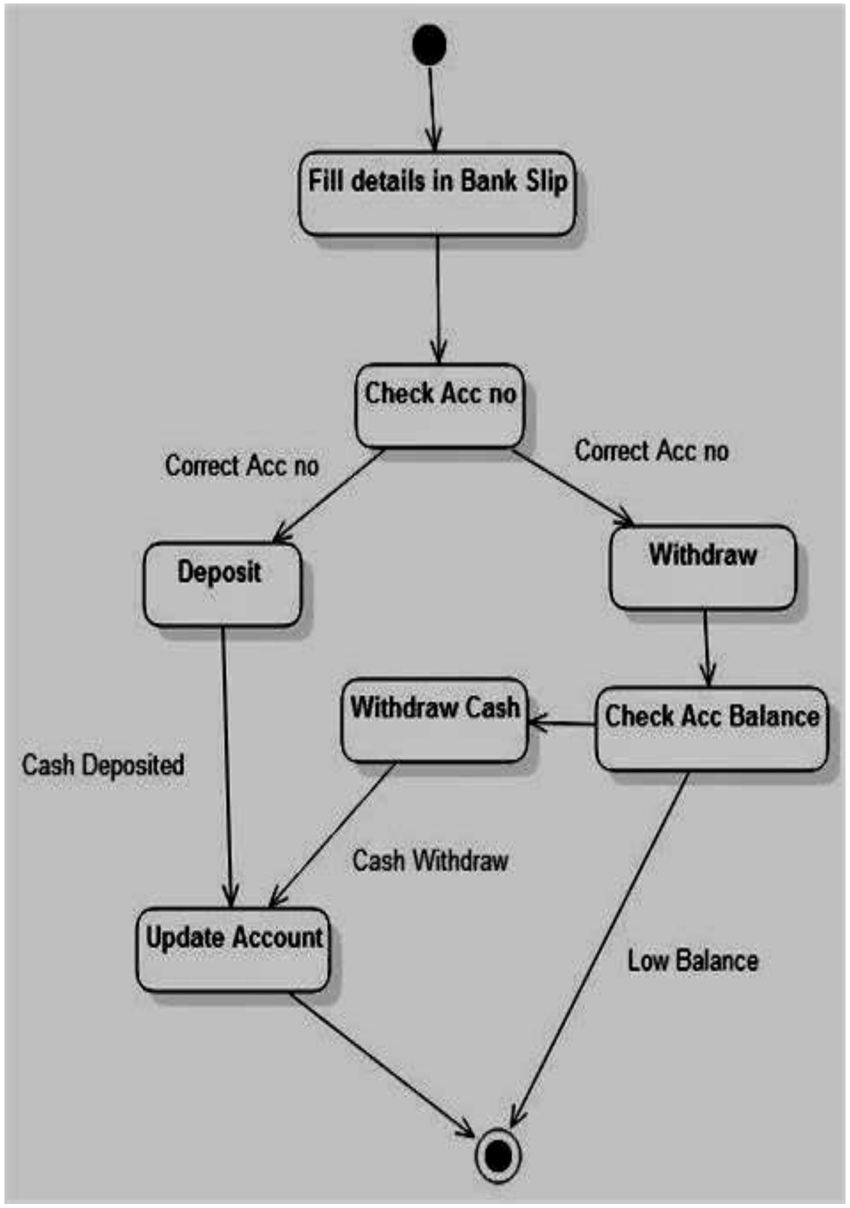
Interaction

Sequence Diagram: For Playing Songs As Per Mood Detection

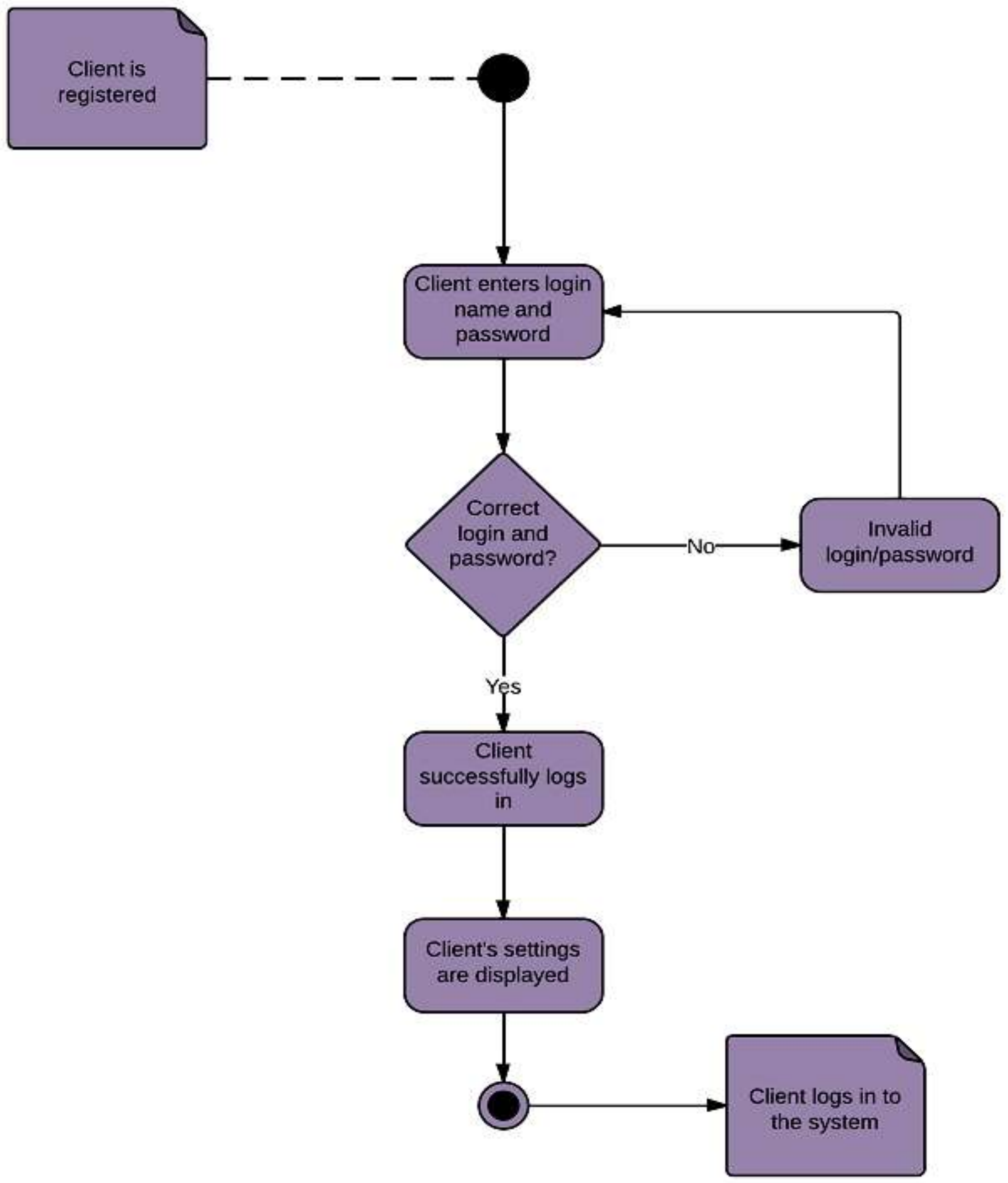


Behavior Diagram

State Chart Diagram For Bank Operations



Activity Diagram For Login Process



Client is registered
Client enters login name and password
Correct login and password?
No
Invalid login/password
Yes
Client successfully logs in
Client's settings are displayed
Client logs in to the system

Use case diagram components

•**Actors:** The *users that interact with a system*. An actor can be a person, an organization, or an outside system that interacts with application or system. They **must be external objects** that produce or consume data.

•**System:** A *specific sequence of actions* and interactions between actors and the system. A system may also be referred to as a scenario.

•**Goals:** The *end result of most use cases*. A successful diagram should describe the activities and variants used to reach the goal.

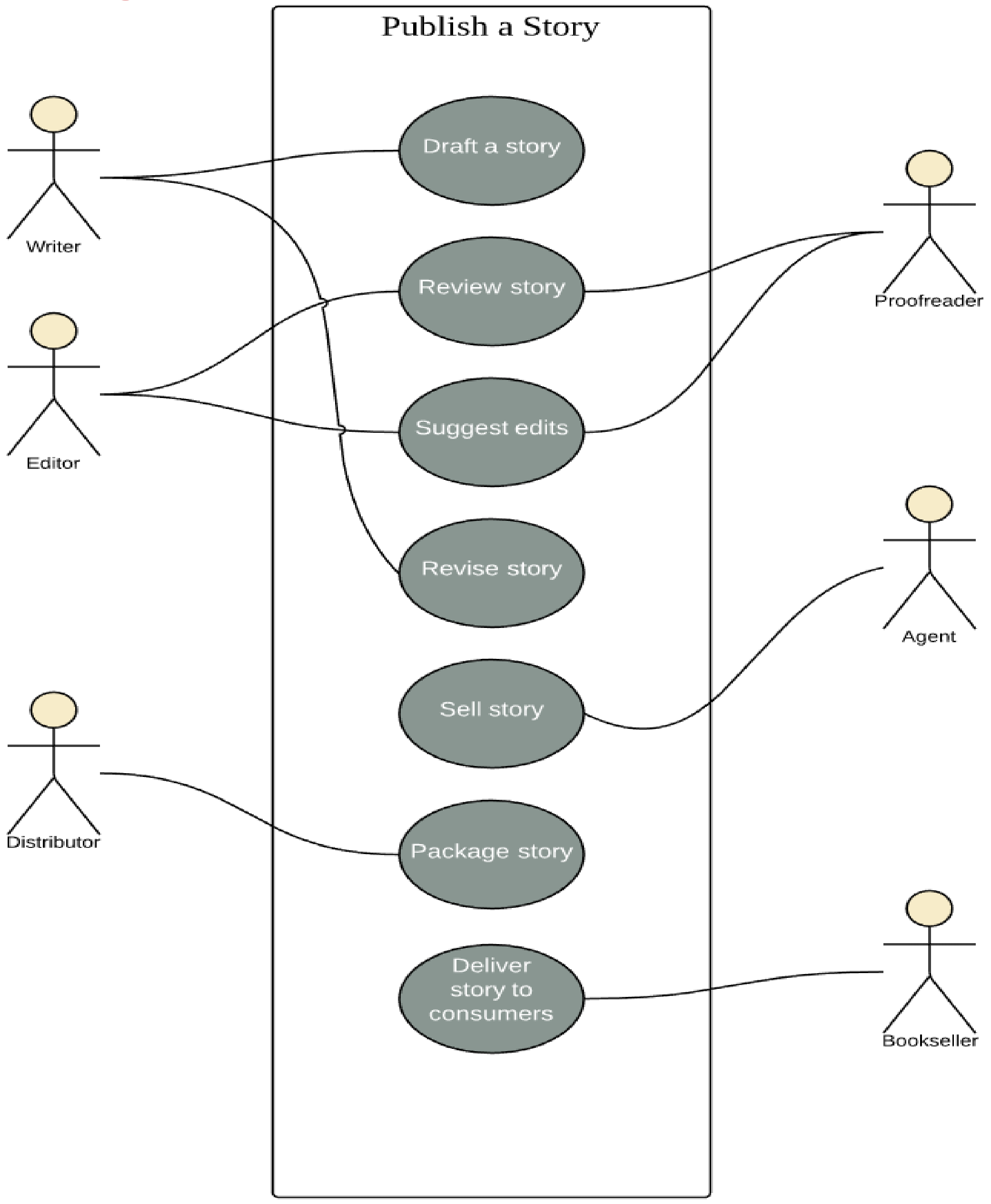
Use Case Diagram Symbols And Notation

The notation for a use case diagram is straightforward and doesn't involve as many types of symbols as other UML diagrams.

Here are all the shapes used to draw Use Case:

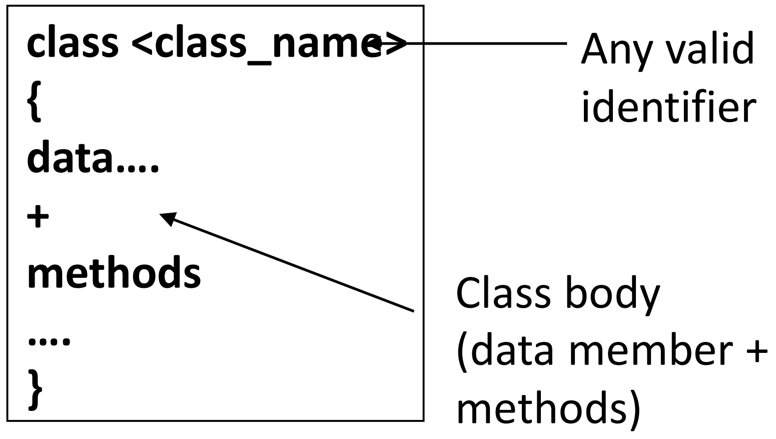
- **Use cases:** Horizontally shaped **ovals** that represent the different uses that a user might have.
- **Actors:** Stick figures that **represent the people** actually employing the use cases.
- **Associations: A line between actors and use cases**. In complex diagrams, it is important to know which 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.
- **Packages:** A UML shape that allows **to put different elements into groups**. Just as with component diagrams, these groupings are represented as file folders

Book Publishing Use Case Diagram

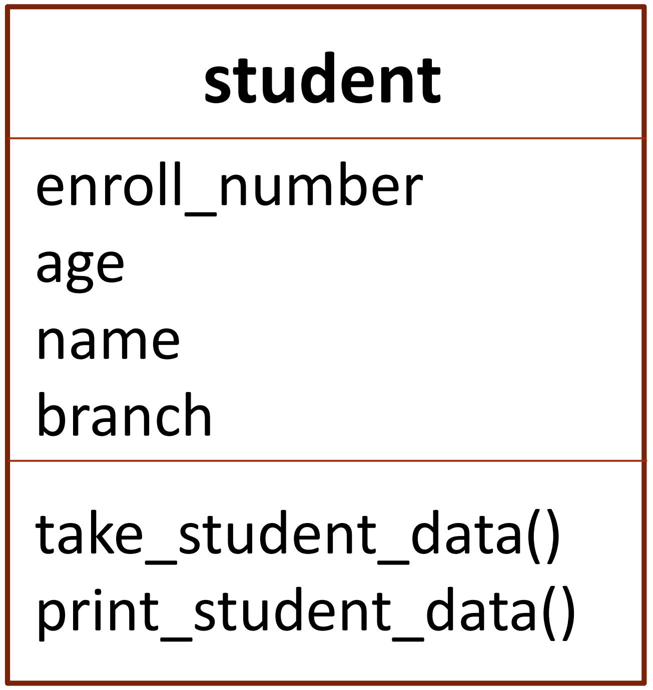


Class

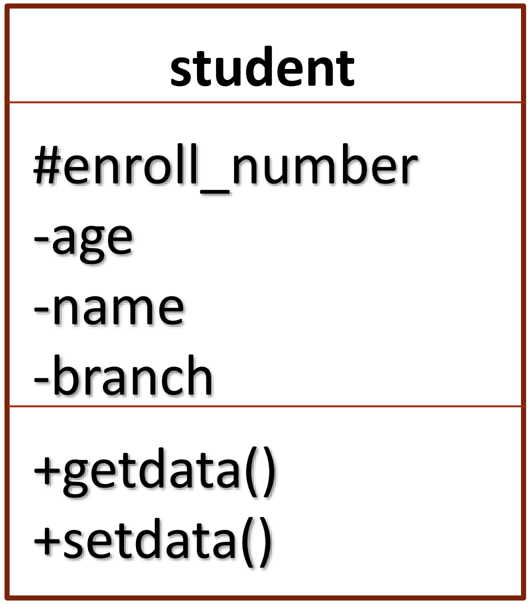
- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { }

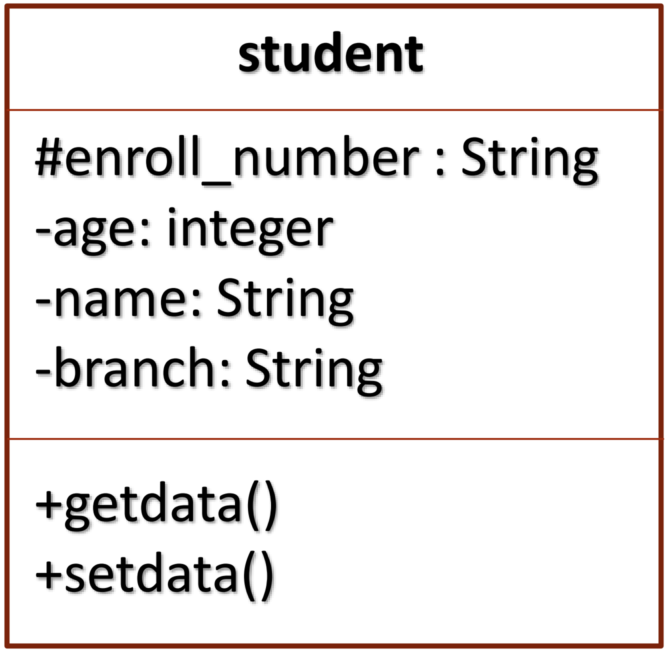


```
class <class_name>
{
data….
+
methods
….
}
```



Class Diagram

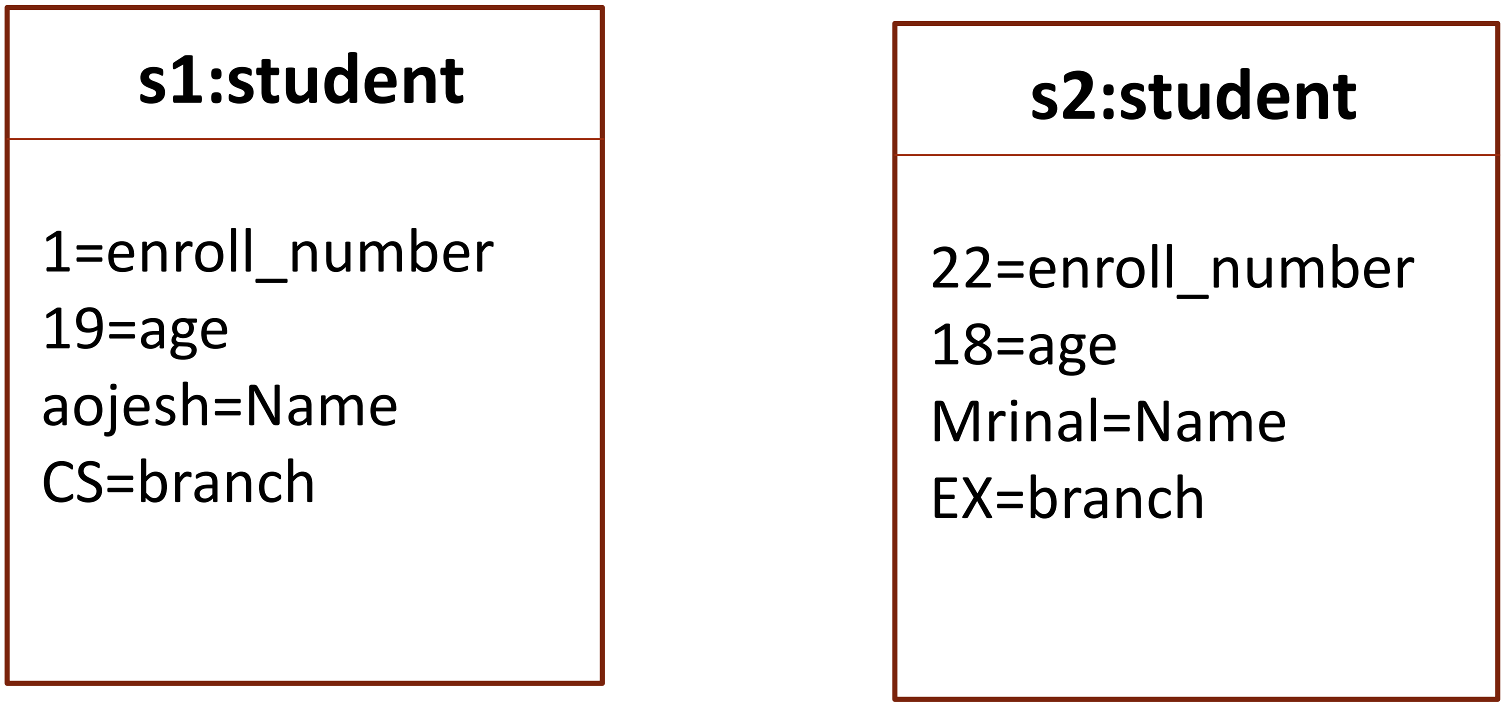




- The top partition contains the name of the class.
- The middle part contains the class attributes.
- The bottom partition shows the possible operations that are associated with the class.

- **Public-** A public member it is prefixed by the symbol **"+"**.
- **Private-** A private member is prefixed by the symbol **"-"**.
- **Protected-** A protected member is prefixed by the symbol **"#"**.
- An abstract class has the class name written in ***italics***.

Object Diagram



Software Design - Objectives

•Software design is both a process and a model.

•The design process is a **sequence of steps** that enable the designer to describe all aspects of the software to be built.

•The design model that is created for software **provides a variety of different views** of the computer software. Basic design principles enable the software engineer to navigate the design process.

•It is like an architect‘s plans for a house. e.g., a three-dimensional model of the house with guidance for constructing each detail (e.g., the plumbing layout).

A set of principles for software design are as follows:

•**The design process should not suffer from “tunnel vision.”** should consider alternative approaches

•**The design should be traceable to the analysis** model- Requirements have been satisfied by the design model.

• **Design is not coding, coding is not design.**

•**The design should exhibit uniformity-**Rules of style and format should be defined for a design team before design work begins.

•**The design should be reviewed to minimize conceptual (semantic) errors.**

•The design should **“minimize the intellectual distance”** between the software and the problem as it exists in the real world-Create the required/desired product.

•**The design should be structured to accommodate change.**

Software Design Approaches

There are mainly 2 approaches

- ***Top-down***
- ***Bottom-up***

- **Top-down** and **bottom-up** are both strategies of information processing.

✓A **top-down** also known as *stepwise design* or *decomposition* is the breaking down of a system to gain insight into its compositional sub-systems in a reverse engineering fashion.

✓ Top down programming is used to create code, while bottom up programming is used to solve problems.

✓In this approach an overview of the system is create but not in detail and its Sub system.

✓Each subsystem is then developed greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements.

✓A top-down model is often specified with the assistance of "black boxes", which makes it easier to manipulate.

✓Top-down approaches emphasize planning and a complete understanding of the system.

•No coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system.

•It is implemented by attaching the stub/sub system in place of the module to complete it.

•This will delays testing of the ultimate functional units of a system.

•Top-down is a programming style, used in traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces.

•write a main procedure that names all the major functions it will need.

•These compartmentalized sub-routines eventually will perform actions so simple they can be easily and concisely coded. When all the various sub-routines have been coded the program is ready for testing.

Bottom-up Approach

•A **bottom-up** approach is the piecing together of systems to give rise to more complex systems. Each individual base elements of the system are specified in great detail.

•These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

•It highlight coding and early testing, which can begin as soon as the first module has been specified.

•This approach runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system but ***Re-usability of code*** is one of the main benefits of the bottom-up approach.

•Object-oriented programming (OOP) is a paradigm that uses **"objects"** to design applications and computer programs. It often resembles a **"seed"** model, by which the beginnings are small, but eventually grow in complexity and completeness

Team For Software Development

•Every product begins with the people.

•Project’s success Or failure depends on software development team.

Scope of Project

•If you’re about to build a product prototype, a team of four may be enough to accomplish the task.

•if you’re planning on launching a brand-new application with multiple features and third-party integrations, the number of team members will certainly increase.

TEAM STRUCTURE

Whether you opt for Waterfall or Agile would directly impact your workflows and a software development team structure.

A traditional ***Waterfall Project*** team is built based on hierarchical relations between team members, so there are managers and subordinates with well-defined responsibilities.

Agile teams, on the other hand, are self-organized and self-managed. Still, there are organizational leaders, like a Scrum Master in Scrum or a Service Delivery Manager in Kanban.

Waterfall Team	Agile Team
Top-down management. The project manager is responsible for delivering results	Self-management. Every team member is responsible for the delivered results
A team may work on many projects at a time	A team focuses on one project
Focus on evaluating the performance of each individual	Focus on assessing the performance of the whole team
Distinct roles and titles	Cross-functional talent
No team size limit	Two-pizza approach with four to ten people per team
A lower degree of team synchronization due to a larger team size and a vertical hierarchy	Small teams with a high degree of coordination and synchronization

Members Of A Software Development Team

A typical software development team structure includes:

- A business analyst
- A product owner
- A project manager
- A product designer
- A software architect
- Software developers,
- Software testing engineers [Including test automation engineers, as well as a DevOps engineer]

Business Analyst

✓Understands customer’s business processes

✓Translates customer business needs into requirements

Product Owner (Decision Maker)

✓Holds responsibility for a product vision and evolution

✓Makes sure a final product meets customer requirements

Project Manager

✓Makes sure a product or its part is delivered on time and within budget

✓Manages and motivates the software development team

UX/UI Designer

✓Transforms a product vision into user-friendly designs

✓Creates user journeys for the best user experience and highest conversion rates

Software Architect

✓Designs a high-level software architecture

✓Selects appropriate tools and platforms to implement the product vision

✓Sets up code quality standards and performs code reviews

Software Developer

✓Engineers and stabilizes the product

✓Solves any technical problems emerging during the development lifecycle

Software Testing Engineer

✓Makes sure an application performs according to requirements

✓Spots functional and non-functional defects

Test Automation Engineer

✓Design a test automation ecosystem

✓Write and maintain test scripts for automated testing

Devops Engineer

✓Facilitates cooperation between development and operations teams
✓Builds CI/CD pipelines for faster delivery

project scope and a software development team structure may correlate:

Project Scope/Stage	Team Size	Team Roles
Discovery / Proof of concept	Up to 5 specialists	Product owner (usually on the client's side), project manager, business analyst, software architect, UI/UX designer
Minimum Viable Product (MVP) development	6+ specialists	Product owner (usually on the client's side), project manager, business analyst, UI/UX designer, software engineers, test engineers
Product development	**Agile:** usually up to 9 specialists; if a project is large-scale, several Agile teams may work together. **Waterfall:** No team size limit; the specific headcount will depend on the type and complexity of an application	Product owner (usually on the client's side), project manager, business analyst, UI/UX designer, software architect, software engineers, test engineers. Optionally: test automation engineers, performance engineers, DevOps engineers, security engineers

User Interface (UI) Design

•UI design refers to Graphical User Interfaces and other forms—e.g., voice-controlled interfaces.
•UI design use to build interfaces in software or computerized devices, focusing on looks or style.
•Interfaces are created, which users find easy to use and pleasurable.

Designing User Interfaces for Users

UI comes in three formats:
•**Graphical user interfaces (GUIs)**—Users interact with visual representations on digital control panels. A computer's desktop is a GUI.

•**Voice-controlled interfaces (VUIs)—**Users interact with these through their voices. Most smart assistants—e.g., Siri on iPhone and Alexa on Amazon devices—are VUIs.

•**Gesture-based interfaces**—Users engage with 3D design spaces through bodily motions: e.g., in *virtual reality (VR)* games.

UI vs. User Experience (UX) Design

•Often confused with UX design

•UI design is more concerned with **the surface and overall feel of a design**.

•UI design is a technique where you the designer build an essential part of the user experience.

•UX design covers the *entire range* of the user experience.

•One analogy is to picture UX design as a car with UI design as the driving console.

Basic Principles Of UI Design

✓**Make buttons and other common elements perform predictably** (including responses such as click-to-zoom)

✓**Maintain high discoverability-**Label icons and include well-indication.

✓**Keep interfaces simple**.

✓**Respect the user's eye and attention regarding layout**. Focus on hierarchy and readability.

✓**Minimize the number of actions for performing tasks but focus on one chief function per page**. Guide users by indicating preferred actions. Ease complex tasks by using progressive disclosure.

✓**Put controls near objects that users want to control**. E.g. a button to submit a form should be near the form.

✓**Keep users informed regarding system responses/actions with feedback**.

✓**Use appropriate UI design patterns to help guide users and reduce burdens** Style/Color Combinations

✓**Maintain brand consistency**.

✓**Always provide next/previous steps which users can figure out *naturally*, whatever their context**.

Modularity

It can be defined as a mechanism where a complex system is divided into several components that are referred to as ‘modules’.

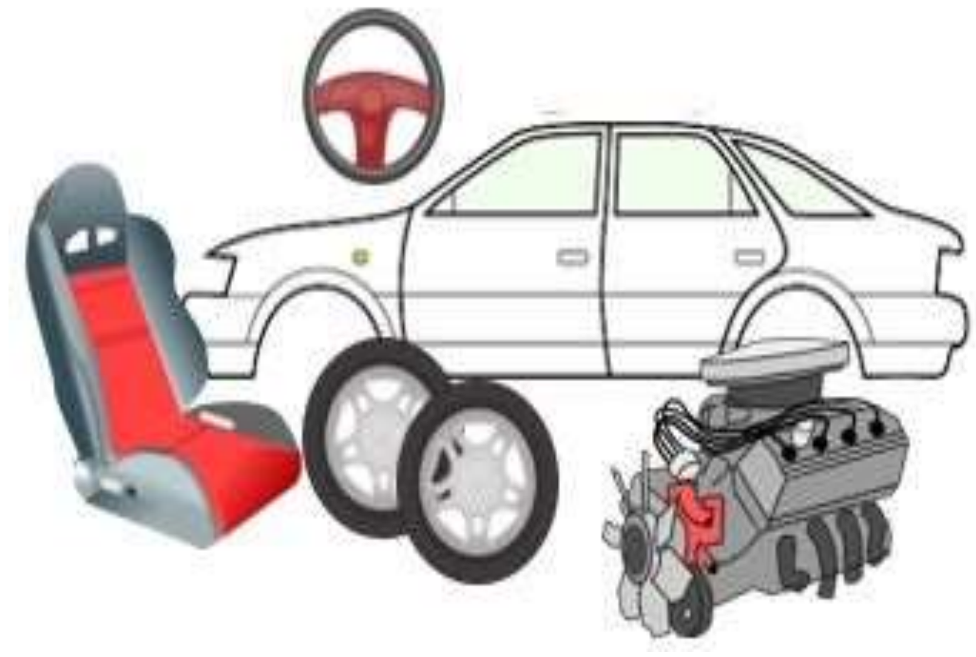


Figure shows division of an automobile into several subsystems then the components or subsystems would be: engines, breaks, wheels, chassis etc.

Why Modularity?

To understand the importance of modularity, consider that we have a *Monolithic Software i.e.* A single-tiered software application in which the user interface and data access code are combined into a single program from a single platform.

Any software engineer can not understand large program in one go.

There will be a lot of local variables, global variables their span of reference, several control paths etc. This will increase the complexity of the large program.

As a solution to this, the large program must be divided into several components or modules as it will become easy to understand the modules.

if modules get increased, the cost required to integrate the several modules also get increased.

Functional Decomposition

•Functional decomposition is a method used to design a detailed structure of components or modules of the software.

•Functional decomposition specifies the functions, activities, processes or actions that the component or module of the software has to perform.

Functional Decomposition Process

•Functional decomposition is a standard technique to design any software.

•It is a top-down method as it keeps refining the functions of the software.

Steps For Functional Decomposition Process

1. Identify the General Function

Developer must be very clear about what task software, system, device, process or component
must do.

2. Identify Set of Sub-Functions

To identify the set of sub-functions to the most general function

3. Identify Next Level Sub-Function

Now, whatever sub-functions you have identified in step 2 you have to refine each of them.

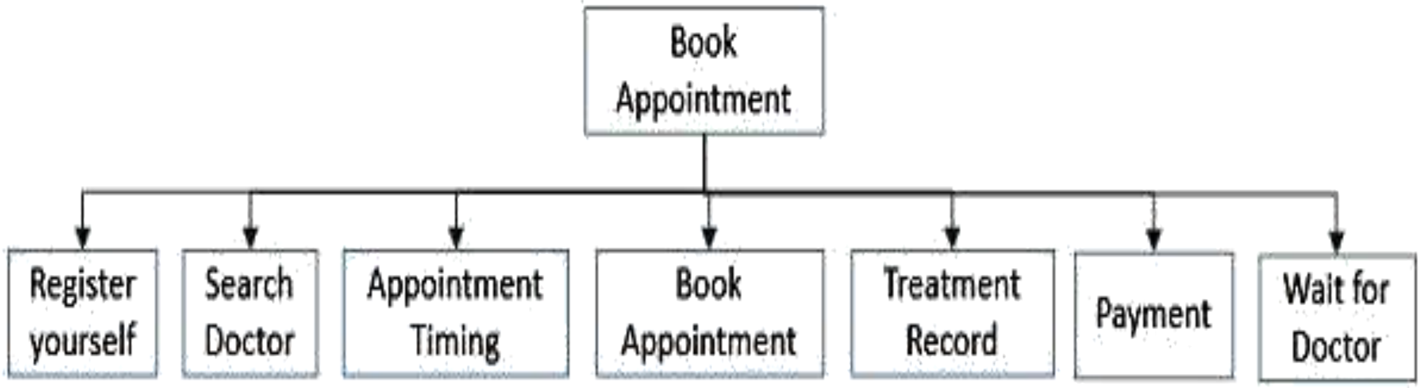
Keep refining the functions until the functions in your functional decomposition diagram cannot break down further.

4. Check the FDD

Assure that you have not left any function that can be included in the diagram.

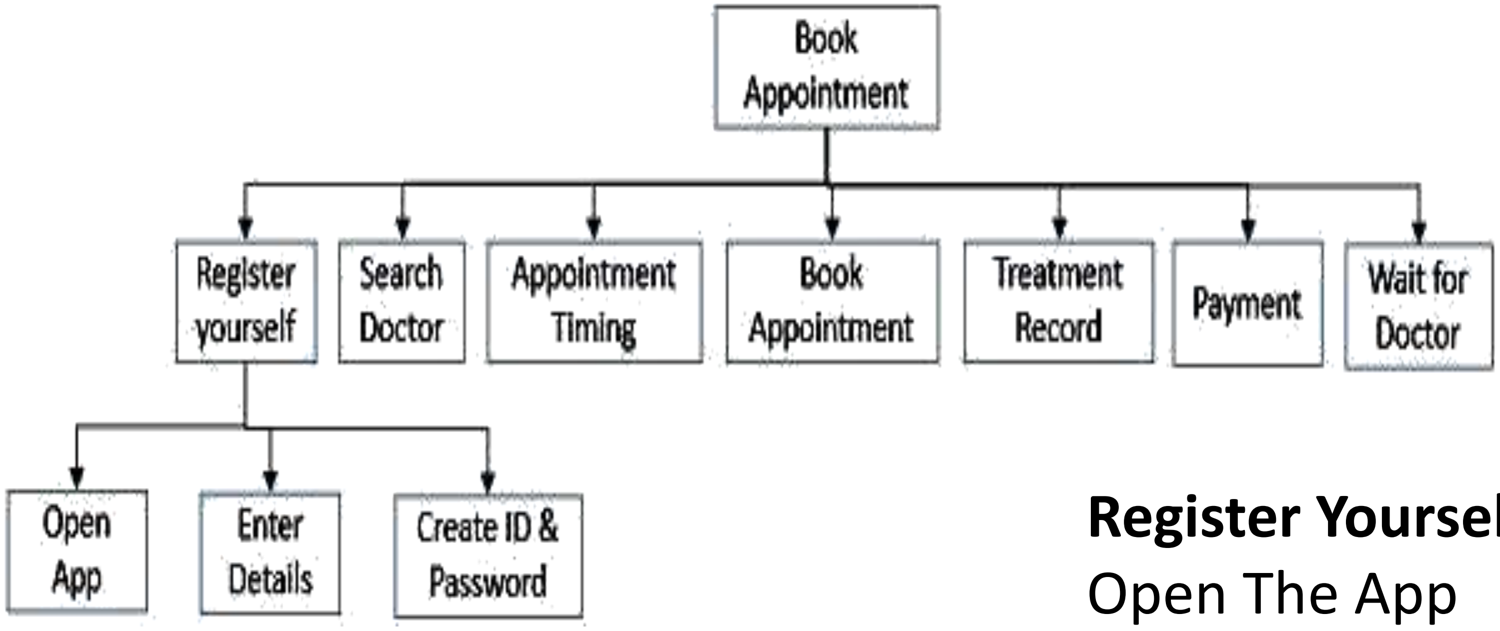
Example

Design FDD that can be used by the patients to make an appointment with the doctor.



1. Register Yourself
2. Search **For** Doctor
3. Manage The Appointment Timing
4. Book Appointment
5. Treatment Record
6. Payment
7. Wait **For** Doctor

Next step will be to break down refine or elaborate each of these statements into more detail.



Register Yourself

Open The App

If New Customer

Then

Enter The Details

And

Create Id And Password

Else

Enter Email-id And

Password To Login

Object Oriented Design

•In the Object-Oriented Design Method, the system is considered a **collection of objects** (i.e., entities).

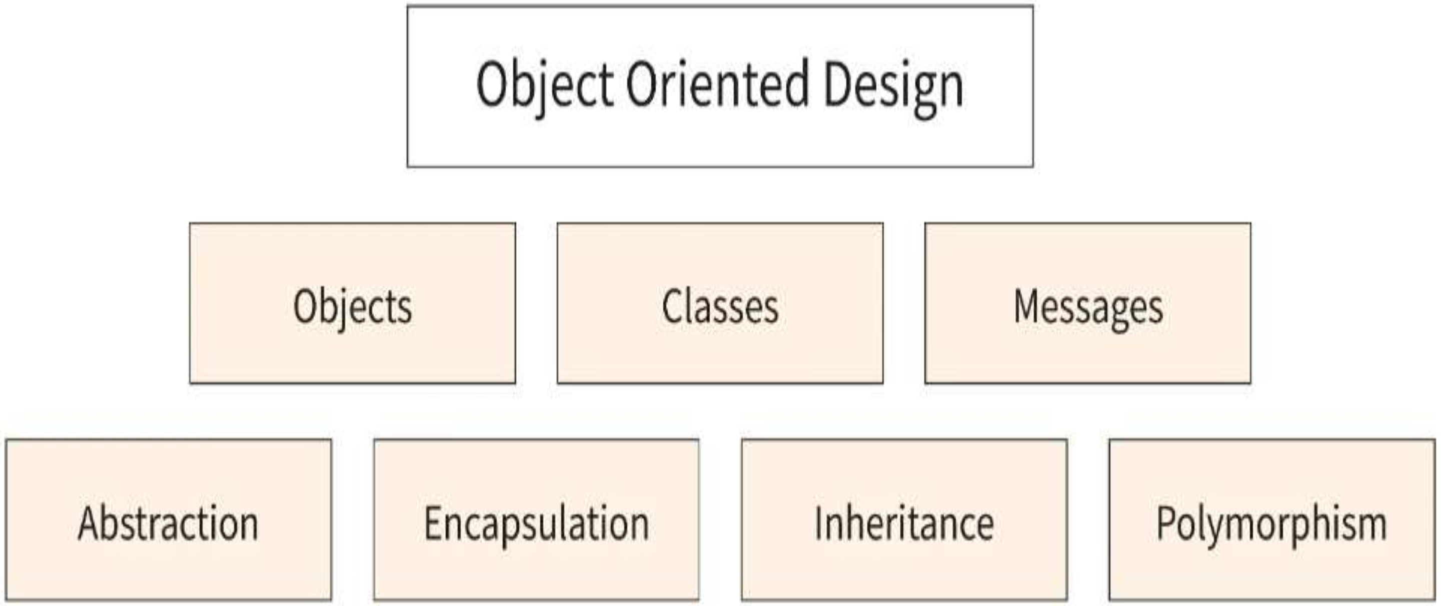
•The state is shared among the objects, and each object is responsible for its own state data.

•Tasks designed for a specific purpose cannot refer to or update data from **other objects**.

•Objects have internal data that represents their current state.

•Similar objects form a class. In other words, **every object** belongs to a class.

Important Terms Related to Object-Oriented Design



Stages of Object-Oriented Design

The Object-Oriented Design process includes two main stages: **System Design** and **Object Design**.

System design

•The entire architecture of the planned system is designed at this stage.

•The system is imagine as a collection of **interacting subsystems**, each comprising a hierarchy of interacting objects classified into classes.

•The system analysis model and the proposed **system architecture** are used to design the system.

Object Design

•A design model is created in this phase based on the models created in the system analysis phase and the architecture built in the system design phase.

•All of the requisite classes have been recognized.

•The relationships between the specified classes are **established**, and class hierarchies are identified.

•In addition, the developer creates the internal details of the classes and their relationships, such as the data structure for each attribute and the **algorithms for the operations**.

Design Pattern in Software Engineering?

•A software design pattern is a general, reusable solution of how to solve a common problem when designing an application or system.

•Unlike a library or framework, which can be inserted and used right away, a design pattern is more of a template to approach the problem at hand.

•Design patterns are used to support Object-Oriented Programming (OOP), a paradigm that is based on the concepts of both objects (instances of a class; data with unique attributes) and classes (user-defined types of data).

Why Do We Need Design Patterns?

1.**Proven Solution:** Design patterns provide a proven, reliable solution to a common problem, no need to create new solution.

2. **Reusable:** Design patterns can be modified to solve many kinds of problems – they are not just tied to a single problem.

3. **Expressive:** Design patterns are an well-designed solution.

4. **Prevent the Need for Refactoring Code:** Since the design pattern is already the optimal solution for the problem, this can avoid refactoring.

5. **Lower the Size of the Codebase:** Each pattern helps software developers change how the system works without a full redesign.

Types of design patterns

There are about 26 Patterns currently discovered.

These 26 can be classified into 3 types:

1.Creational: These patterns are designed for class instantiation. They can be either class-creation patterns or object-creational patterns.

2. **Structural:** These patterns are designed with regard to a class's structure and composition. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition.

3. **Behavioral:** These patterns are designed depending on how one class communicates with others.

UNIT-4

SOFTWARE TESTING

Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.
Early software testing uncovers problems before a product implementation.

The earlier development teams receive test feedback, the sooner they can address issues such as:

✓Architectural flaws
✓Poor design decisions
✓Invalid or incorrect functionality
✓Security vulnerabilities
✓Scalability issues

Types Of Software Testing

There are many different types of software tests, each with specific objectives and strategies:

•**Acceptance testing:** Verifying whether the complete system works as intended.

•**Integration testing:** Ensuring that software components or functions operate together.

•**Unit testing:** Validating that each software unit performs as expected.
•A unit is the smallest testable component of an application.

•**Functional testing:** Checking functions by emulating business scenarios, based on functional requirements.
•Black-box testing is a common way to verify functions.

- **Performance testing:** Testing how the software performs under different workloads.
- Load testing, for example, is used to evaluate performance under real-life load conditions.
- **Regression testing:** Checking whether new features break or degrade functionality.
- Normal/Common testing can be used to verify menus, functions and commands at the surface level, when there is no time for a full regression test.
- Change in hardware and run the software on it also comes in this testing.
- **Stress testing:** Testing how much strain the system can take before it fails.
- Considered to be a type of non-functional testing.
- **Usability testing:** Validating how well a customer can use a system or web application to complete a task.

Testing Methods

- ➢Static Testing
- ➢Dynamic Testing

#1. Static Testing

It is also known as Verification in Software Testing.
This method checks documents and files.
To verify the requirements and to verify product is developing the accordingly or not.

Activities involved here are:

- •Inspections
- •Reviews
- •Walkthroughs

#2. Dynamic Testing

It is also known as Validation in Software Testing.

Validation is a dynamic process of testing the real product.

Validation is the process to validate the product with test cases.

Testing Approaches

There are three types of software testing approaches.

- ✓White Box Testing
- ✓Black Box Testing
- ✓Grey Box Testing

#1. White Box Testing

- It is also called Glass Box, Clear Box, Structural Testing.
- White Box Testing is based on the **application's internal code structure**.
- In white-box testing, an internal perspective of the system, as well as programming skills, are used to design test cases.
- This testing is usually **done at the unit level.**

#2. Black Box Testing

•It is also called Behavioral/Specification-Based/Input-Output Testing.

•Black Box Testing is a software testing method in which testers evaluate the functionality of the software under test without looking at the internal code structure.

#3. Grey Box Testing

Grey box is the combination of both White Box and Black Box Testing.
The tester who works on this type of testing needs to have access to design documents. This helps to create better test cases in this process.

Testing Levels

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

#1. Unit Testing

It is also known as Module Testing or Component Testing.

Unit Testing is done to check whether the individual modules of the source code are working properly. i.e. testing each and every unit of the application separately by the developer in the developer’s environment.

#2. Integration Testing

•Integration Testing is the process of testing the connectivity or data transfer between a couple of unit tested modules.

•It is also called I&T Testing or String Testing.

•It is subdivided into the Top-Down Approach, Bottom-Up Approach, and Sandwich Approach (Combination of Top-Down and Bottom-Up).

#3. System Testing (End to End Testing)

It's a black box testing.

Testing the fully integrated application is also called as an end to end scenario testing.

To ensure that the software works in all intended target systems.

Verify thorough testing of every input in the application to check for desired outputs. Testing of the user's experiences with the application.

#4. Acceptance Testing

To obtain customer sign-off so that software can be delivered and payments received.

Types of Acceptance Testing are Alpha, Beta & Gamma Testing.

Software Testing Strategies Issues.

- Specify the requirement before testing starts in a quantifiable manner.
- According to the categories of the user generate profiles for each category of user.
- Produce a robust software and it's designed to test itself.
- Should use the Formal Technical Reviews (FTR) for the effective testing.
- To access the test strategy and test cases FTR should be conducted.
- To improve the quality level of testing generate test plans from the users feedback.

System Testing Test Case Design

The test case design is a document that **Quality Analysts** prepare to define the testing strategy and scope and provide other details such as testing prerequisites, test steps, post-conditions, and expected results.

Plan and Design Test Cases

Prepare the Test Environment

To run the tests in an environment, the QA engineers must prepare the environment.

This means they would require testing software like test data generators, debuggers, emulators and stimulators, and stubs and drivers.

Plan and Design Test Cases

Determine the Scope of the Test Case

•Test case design begins with determining the scope.

•The stakeholders need to provide the requirements for testing.

•These requirements could be design specifications, use cases, the usability of the software, etc.

Design the Test Case

•Once the scope is defined, and the test environment prepared, the QA engineers determine the best testing strategy for the test cases.

•They start creating test cases for each software requirement to ensure that it meets all the specifications.

The test case document must be as detailed as possible. It could be in any format but must contain information such as:

- ✓**Test case ID:** It is a unique id for the test case.
- ✓**Test description:** Describes the units, features, and functionalities being tested and verified.
- ✓**Testing conditions:** Describes the purpose and conditions under which the test was executed.
- ✓**The assumptions and preconditions:** Describe the conditions to meet before test execution.
- ✓**Testing steps:** Describe the engineer's steps to execute the test.
- ✓**The input test data:** Describes the variables and values used for testing.

✓**The expected results:** Describes the results expected from testing.

✓**The actual and Post-condition results:** Describes the output that was derived during testing. Post conditions explain what happens when the steps are executed.

✓**Pass/Fail status:** If the expected and actual results match, marks the status as ‘Pass.’ Otherwise, the engineer marks it as ‘Fail.’

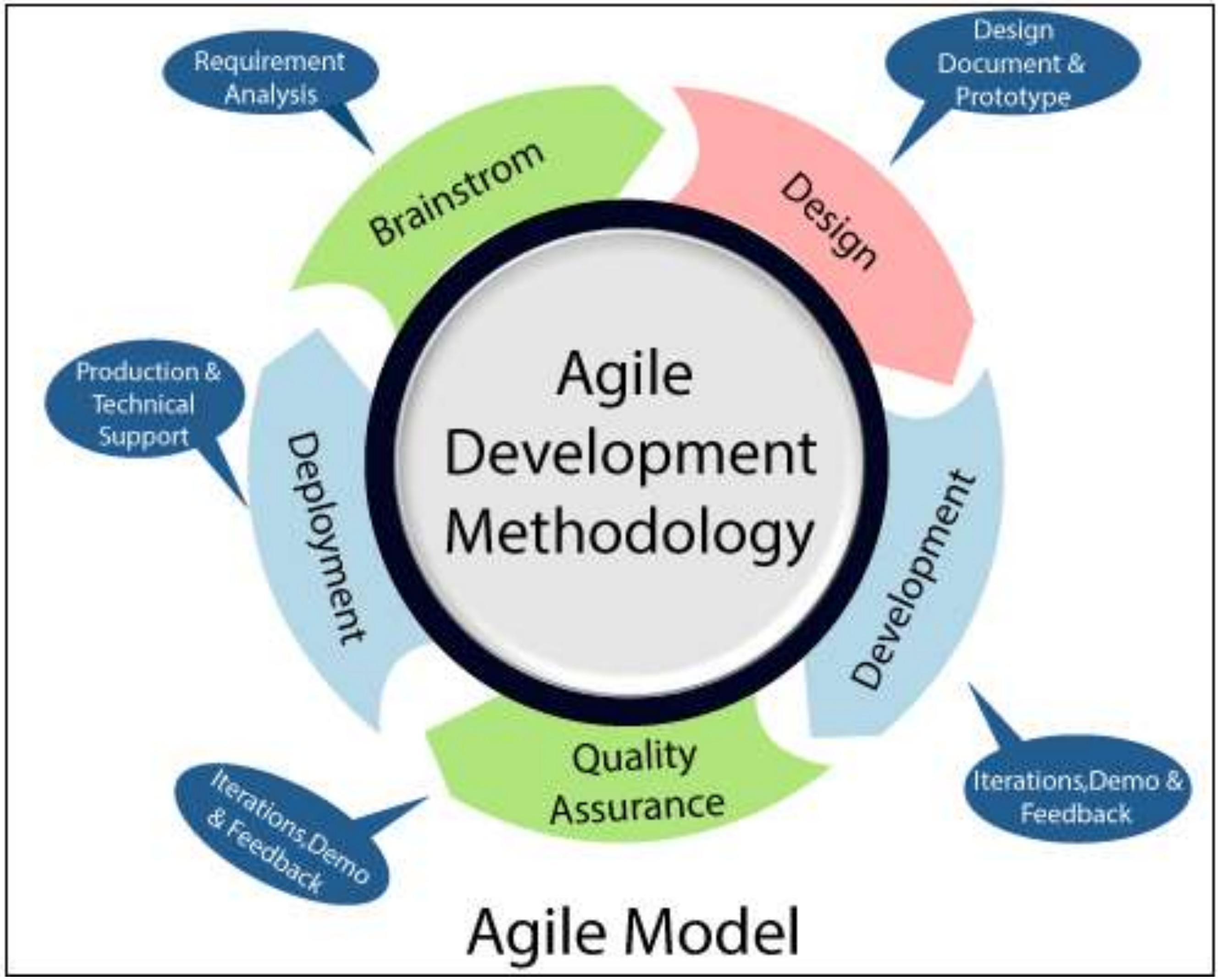
UNIT-5

What is Agile?

Agile is the ability to create and respond to change

Agile Software Development Methodology

- It is an approach that is used to design a disciplined software management process which also allows some frequent alteration in the development project.
- One thing that separates Agile from other approaches to software development is the focus on the people doing the work and how they work together.
- Solutions evolve through collaboration between self-organizing cross-functional teams utilizing the appropriate practices for their context.
- This is a type of software development methodology is used to minimize risk by developing software in short time boxes which are called iterations that generally last for one week to one month.


Requirement Analysis
Design Document & Prototype
Brainstrom
Design
Agile Development Methodology
Production & Technical Support
Deployment
Development
Quality Assurance
Iterations,Demo & Feedback
Iterations,Demo & Feedback
Agile Model

Advantages

•Customer satisfaction by rapid, continuous delivery of useful software.

•People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.

•Agile methodology has an adaptive approach that is able to respond to the changing requirements of the clients.

•Direct communication and constant feedback from customer representatives leave no space for any guesswork in the system.

Disadvantages

- In the case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- This methodology focuses on working software rather than documentation, hence it may result in a lack of documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers unless combined with experienced resources.

Key Agile Concepts

Below are a few key Agile concepts.

User Stories: In consultation with the customer or product owner, the team divides up the work to be done into functional increments called "user stories."
Each user story is expected to yield a contribution to the value of the overall product.

Daily Meeting: Each day the team meets at the **same time**: each team members briefly describes any **"completed"** contributions and **any obstacles** that stand in their way.

Personas: When the project calls for it – for instance when user experience is a major factor in project outcomes – the team crafts detailed, fake biographies of fictitious users of the future product: these are called "personas."

Team: It is a small group of people, assigned to the same project or effort, nearly all of them on a full-time basis.
A small minority of team members may be part-time contributors, or may have competing responsibilities.

Incremental Development: this means that each successive version of the product is usable, and each builds upon the previous version by adding user-visible functionality.

Iterative Development: Agile projects are iterative insofar as they intentionally allow for “repeating” software development activities, and for potentially “revisiting” the same work products.

Milestone Retrospective: Once a project has been underway for some time, or at the end of the project, all of the team’s permanent members (not just the developers) invests from one to three days in a detailed analysis of the project’s significant events.

4 Agile Testing Methods

1. Behavior Driven Development (BDD)

BDD encourages communication between project stakeholders so all members understand each feature, prior to the development process.
In BDD, testers, developers, and business analysts create "scenarios", which facilitate example-focused communication.
The idea of BDD is that the team creates scenarios, builds tests around those scenarios which initially fail, and then builds the software functionality that makes the scenarios pass

2. Acceptance Test Driven Development (ATDD)

ATDD involves the customer, developer, and tester.

The acceptance tests represent a user's perspective, and specify how the system will function. They also ensure that the system functions as intended.

3. Exploratory Testing

In exploratory testing, the test execution and the test design phase go together.

This type of testing focuses on interacting with working software rather than separately planning, building and running tests.

4. Session-Based Testing

This method is similar to exploratory testing, but is more orderly, aiming to ensure the software is tested comprehensively.

It adds test charters, which helps testers know what to test, and test reports which allow testers to document what they discover during a test.

Scrum Development Methodology

- It is used in nearly all types of projects. For companies where the requirements are highly emerging and rapid changes are easily adhere to.
- This model begins with brief planning, meeting, and concludes with a final review.
- This method allows a series of iterations in a single go.
- It is an ideal methodology because it easily brings on track even the slowest progressing projects.

Advantages of Scrum Development

•It is use for fast-moving, rapid coding and testing mistakes that can be easily rectified.

•Decision-making is entirely in the hands of the teams.

•It enables projects with the business requirements documentation and other signs that contribute to success.

•A daily meeting easily helps the developer to make it possible to measure individual productivity.

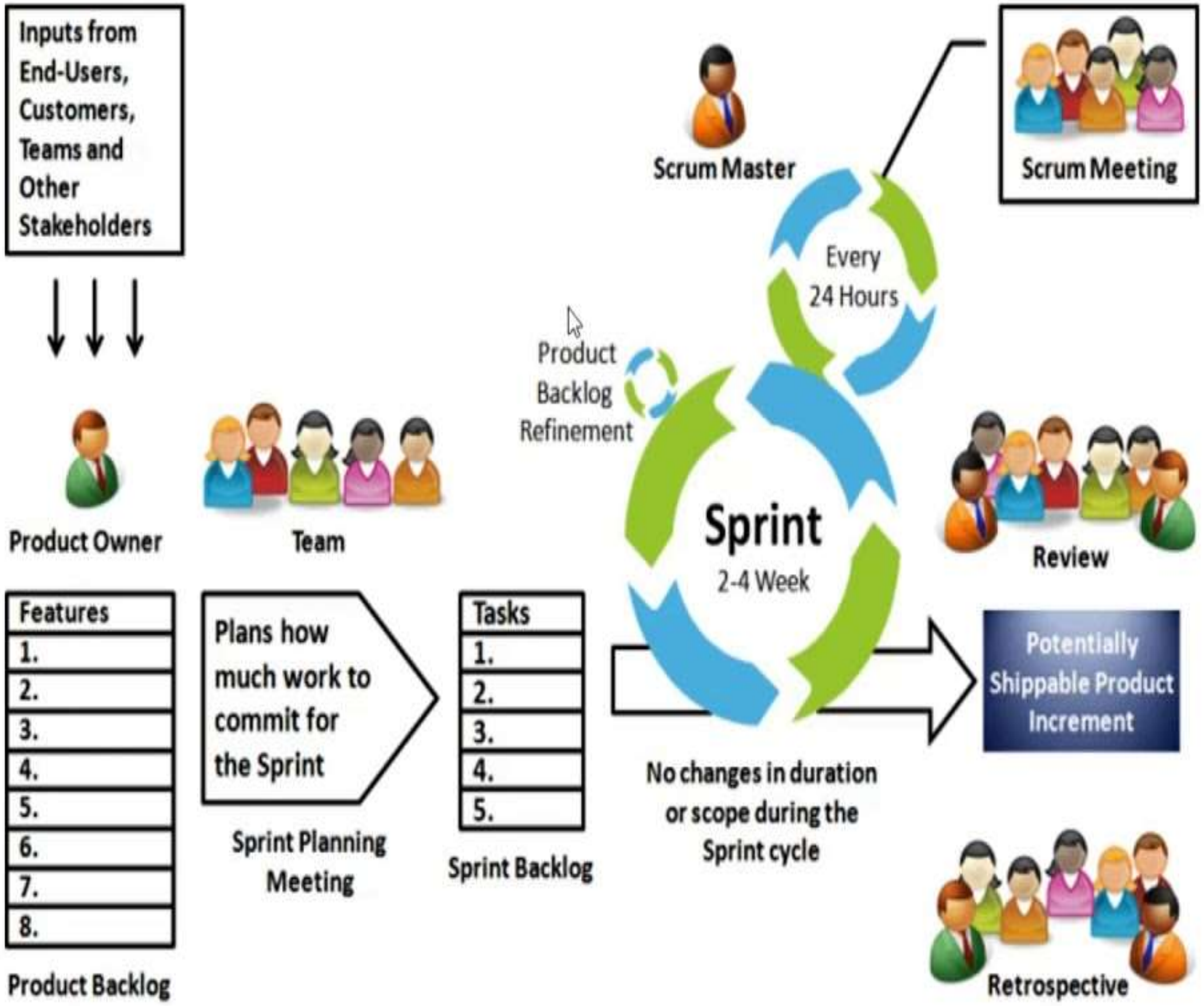
•Due to short sprints and constant feedback, it becomes easier to cope with the changes.

•It is easier to deliver a quality product at a scheduled time.

Disadvantages of Scrum Development

- In Scrum **there is no definite end date,** the project management stakeholders will be tempted to keep demanding that new functionality be delivered.
- **Keep the estimation of project costs and accurate time.** if not then this kind of development model will suffer.
- It is **not suitable for large size projects**.
- This methodology **needs experienced team** members only.
- It works well when the Scrum Master trusts the team. If they have strict control over the team members, it can be frustrating for them, leading to demoralization and the failure of the project.
- Project quality assessment is hard to implement and quantify, unless the test team is able to conduct regression testing after each sprint.

SCRUM

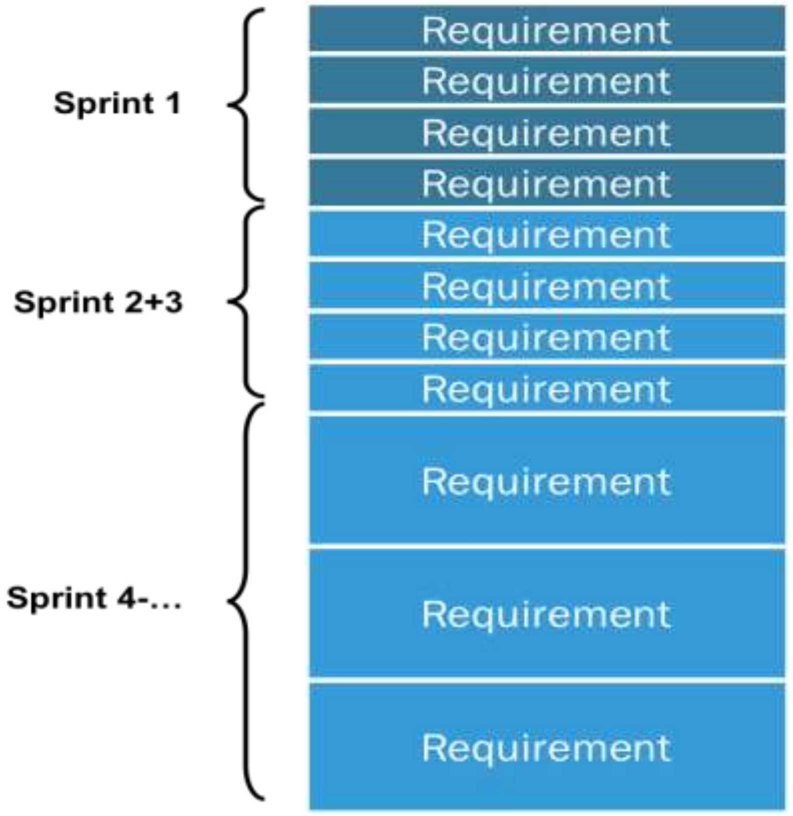

Inputs from End-Users, Customers, Teams and Other Stakeholders
Scrum Master
Scrum Meeting
Every 24 Hours
Product Backlog Refinement
Product Owner
Team
Sprint
2-4 Week
Review
Features
1.
2.
3.
4.
5.
6.
7.
8.
Plans how much work to commit for the Sprint
Tasks
1.
2.
3.
4.
5.
Potentially Shippable Product Increment
No changes in duration or scope during the Sprint cycle
Sprint Planning Meeting
Sprint Backlog
Product Backlog
Retrospective

Product Backlog

The Product Backlog is an emergent, ordered list of what is needed to improve the product.

It is the single source of work undertaken by the Scrum Team.

Product Backlog items that can be Done by the Scrum Team within one Sprint are deemed ready for selection in a Sprint Planning event.



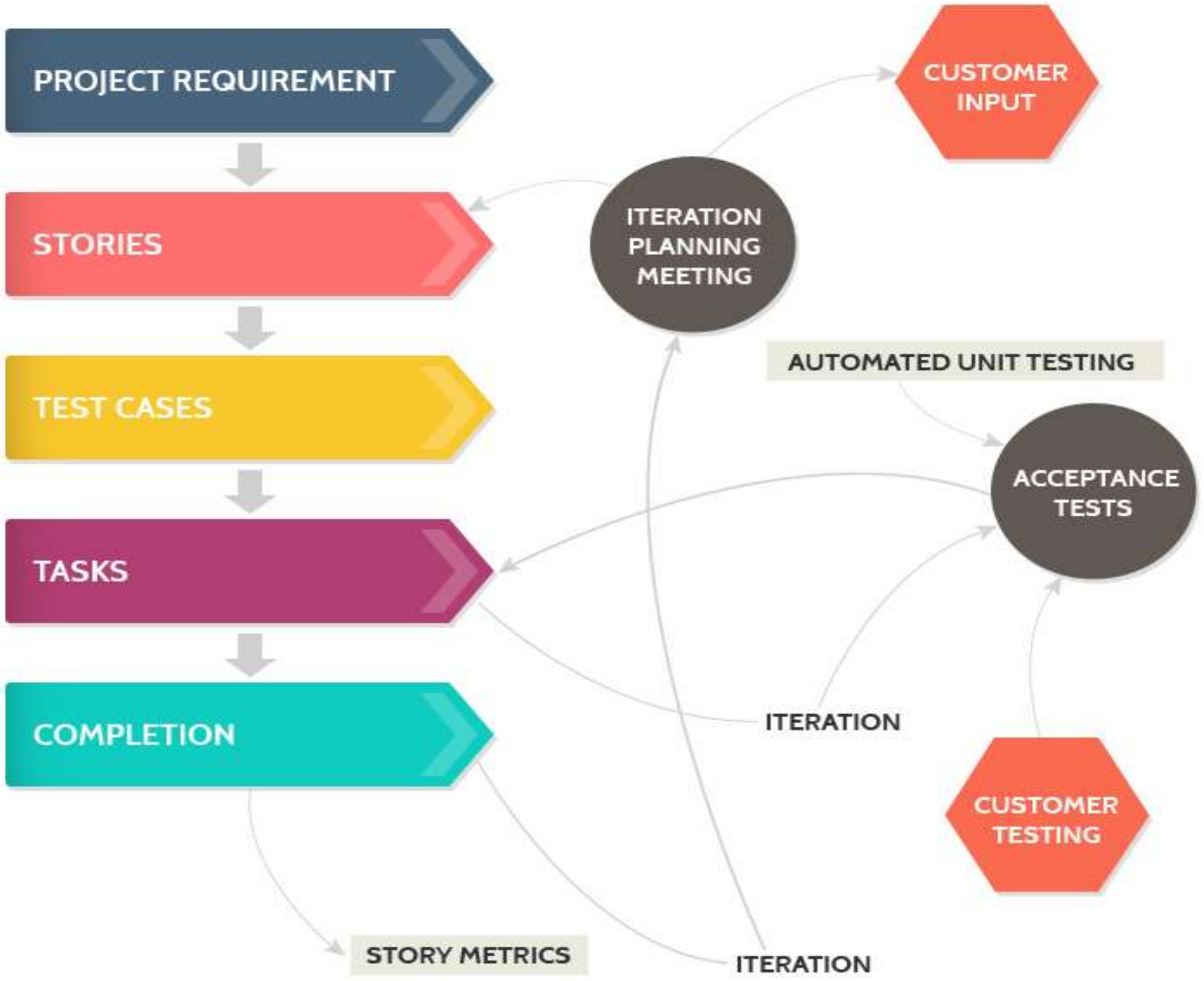
Product Backlog refinement is the act of breaking down and further defining Product Backlog items into smaller more precise items.

This is an ongoing activity to add details, such as a description, order, and size.

Attributes often vary with the domain of work.

Extreme Programming

- **Extreme Programming** is an agile software engineering methodology.
- This methodology, which is shortly known as XP methodology is mainly used for creating software within a very unstable environment.
- It allows **greater flexibility** within the modeling process. The main goal of this XP model is to **lower the cost** of software requirements.
- It is quite common in the XP model that the cost of changing the requirements at later stages in the project can be very high.


PROJECT REQUIREMENT
STORIES
TEST CASES
TASKS
COMPLETION
CUSTOMER INPUT
ITERATION PLANNING MEETING
AUTOMATED UNIT TESTING
ACCEPTANCE TESTS
ITERATION
CUSTOMER TESTING
STORY METRICS
ITERATION

Advantages of Extreme Programming Methodology

•It allows software development companies to **save costs and time** required for project realization. Time savings are available because of the fact that XP focuses on the timely delivery of final products.

•XP teams save lots of money because they **don’t use too much documentation**. They usually solve problems through discussions inside of the team.

•XP methodologies emphasize **customer involvement**.

•This model helps to establish rational (based on reason) plans and schedules and to get the developers personally committed to their schedules.

Disadvantages of Extreme Programming Methodology

- Some specialists say that XP focused on the code rather than on design. That may be a problem because good design is extremely important for software applications.
- In XP projects the defect documentation is not always good. Lack of defect documentation may lead to the occurrence of similar bugs in the future.
- This methodology is only as effective as the people involved, Agile does not solve this issue.
- It requires meetings at frequent intervals at enormous expense to customers.
- It requires too many development changes which are very difficult to adopt every time for the software developer.
- In this methodology, it tends to impossible to be known exact estimates of work effort needed to provide a quote.

Most specialists define five stages of XP software development life cycle: They are

•Planning

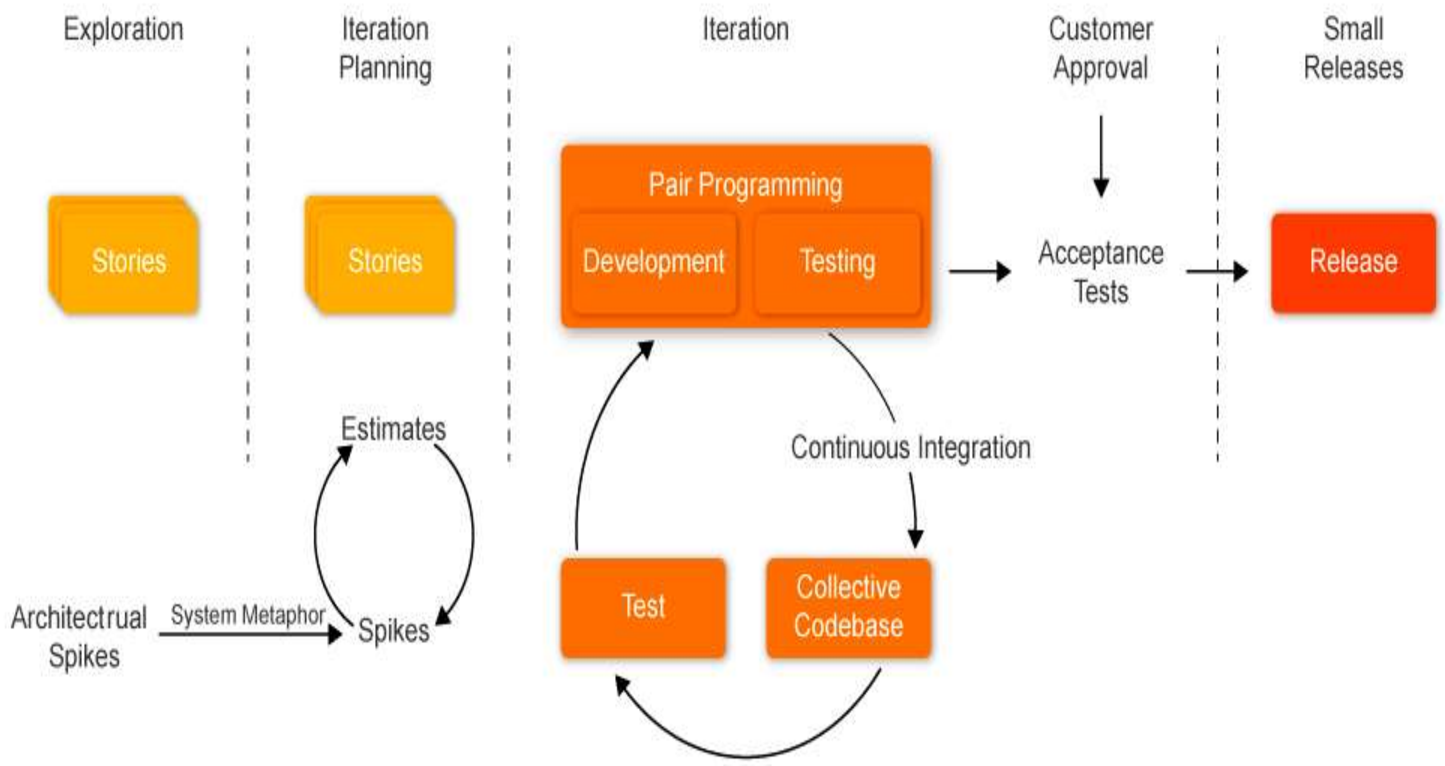
•Designing

•Coding

•Testing

•Listening: At the final stage of the life cycle the XP team must get a feedback from the customer. He is the only person who estimates the final and intermediate products.

Extreme Programming (XP)


Exploration
Iteration Planning
Iteration
Customer Approval
Small Releases
Stories
Stories
Pair Programming
Development
Testing
Acceptance Tests
Release
Estimates
Continuous Integration
Architectrual Spikes
System Metaphor
Spikes
Test
Collective Codebase