UNIT I: Introduction

Introduction: Software Crisis, Software Processes & Characteristics, Software life cycle models, Waterfall, Prototype, Evolutionary and Spiral Models. Software Requirements Analysis & Specifications: Requirement engineering, requirement elicitation techniques like FAST, QFD, requirements analysis using DFD, Data dictionaries ER Diagrams, Requirements documentation, Nature of SRS, Characteristics & organization of SRS .

# What is Software Engineering?

The term software engineering is the product of two words, software, and engineering. The software is a collection of integrated programs.

Software consists of carefully-organized instructions and code written by developers on any of various particular computer languages.

Computer programs and related documentation such as requirements, design models and user manuals.

Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.

# Software Crisis

Software crisis is a term used in the early days of computing science for the difficulty of writing useful and efficient computer programs in the required time.

1. **Size:** Software is becoming more expensive and more complex with the growing complexity and expectation out of software. For example, the code in the consumer product is doubling every couple of years.
2. **Quality:** Many software products have poor quality, i.e., the software products defects after putting into use due to ineffective testing technique. For example, Software testing typically finds 25 errors per 1000 lines of code.
3. **Cost:** Software development is costly i.e. in terms of time taken to develop and the money involved. For example, Development of the FAA's Advanced Automation System cost over $700 per lines of code.
4. **Delayed Delivery:** Serious schedule overruns are common. Very often the software takes longer than the estimated time to develop, which in turn leads to cost shooting up. For example, one in four large-scale development projects is never completed.

# Software Processes

The term software refers to the set of computer programs, procedures and associated documents (Flowcharts, manuals, etc.) that describe the program and how they are to be used.

A software process is the set of activities and associated outcomes that produce a software product. Software engineers mostly carry out these activities. These are four key process activities, which are common to all software processes. These activities are:

1. **Software specifications:** The functionality of the software and constraints on its operation must be defined.
2. **Software development:** The software to meet the requirement must be produced.
3. **Software validation:** The software must be validated to ensure that it does what the customer wants.
4. **Software evolution:** The software must evolve to meet changing client needs.

# Software Development Life Cycle (SDLC)

A software life cycle model (also termed process model) is a pictorial and diagrammatic representation of the software life cycle. A life cycle model represents all the methods required to make a software product transit through its life cycle stages. It also captures the structure in which these methods are to be undertaken.

In other words, a life cycle model maps the various activities performed on a software product from its inception to retirement. Different life cycle models may plan the necessary development activities to phases in different ways. Thus, no element which life cycle model is followed, the essential activities are contained in all life cycle models though the action may be carried out in distinct orders in different life cycle models. During any life cycle stage, more than one activity may also be carried out.

# Need of SDLC

The development team must determine a suitable life cycle model for a particular plan and then observe it.

Without using an exact life cycle model, the development of a software product would not be in a systematic and disciplined manner. When a team is developing a software product, there must be a clear understanding among team representative about when and what to do. Otherwise, it would point to project failure. This problem can be defined by using an example. Suppose a software development issue is divided into various parts and the parts are assigned to the team members. From then on, suppose the team representative is allowed the freedom to develop the roles

assigned to them in whatever way they like. It is possible that one representative might start writing the code for his part, another might choose to prepare the test documents first, and some other engineer might begin with the design phase of the roles assigned to him. This would be one of the perfect methods for project failure.

A software life cycle model describes entry and exit criteria for each phase. A phase can begin only if its stage-entry criteria have been fulfilled. So without a software life cycle model, the entry and exit criteria for a stage cannot be recognized. Without software life cycle models, it becomes tough for software project managers to monitor the progress of the project

# SDLC Cycle

SDLC Cycle represents the process of developing software. SDLC framework includes the following steps:

## The stages of SDLC are as followsSoftware Development Life Cycle(SDLC)

### Stage1: Planning and requirement analysis

1. Requirement Analysis is the most important and necessary stage in SDLC.

The senior members of the team perform it with inputs from all the stakeholders and domain experts or SMEs in the industry.

Planning for the quality assurance requirements and identifications of the risks associated with the projects is also done at this stage

Business analyst and Project organizer set up a meeting with the client to gather all the data like what the customer wants to build, who will be the end user, what is the objective of the product. Before creating a product, a core understanding or knowledge of the product is very necessary.

# Stage2: Defining Requirements

Once the requirement analysis is done, the next stage is to certainly represent and document the software requirements and get them accepted from the project stakeholders.

This is accomplished through "SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.

# Stage3: Designing the Software

The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project. This phase is the product of the last two, like inputs from the customer and requirement gathering.

# Stage4: Developing the project

In this phase of SDLC, the actual development begins, and the programming is built. The implementation of design begins concerning writing code. Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

# Stage5: Testing

After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed and gathered during the requirements stage.

During this stage, unit testing, integration testing, system testing, acceptance testing are done.

# Stage6: Deployment

Once the software is certified, and no bugs or errors are stated, then it is deployed.

Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.

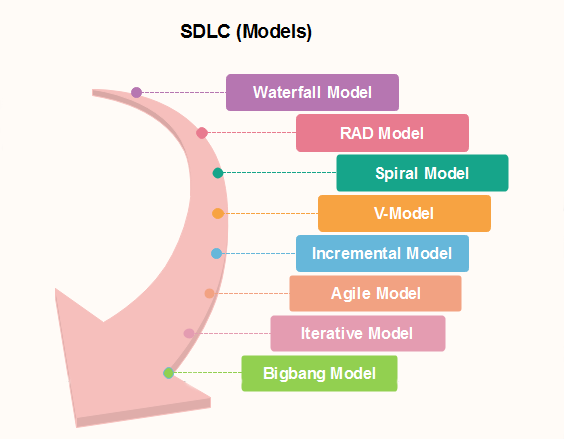
After the software is deployed, then its maintenance begins.

# Stage7: Maintenance

Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time.

This procedure where the care is taken for the developed product is known as maintenance.

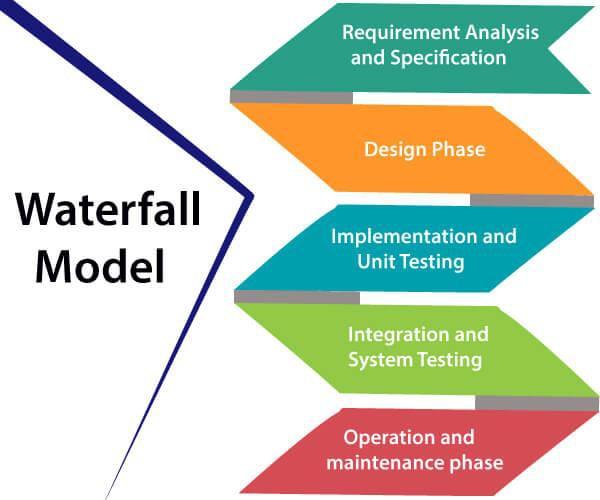
# SDLC Models



**Waterfall model**

Winston Royce introduced the Waterfall Model in 1970.This model has five phases: Requirements analysis and specification, design, implementation, and unit testing, integration and system testing, and operation and maintenance. The steps always follow in this order and do not overlap. The developer must complete every phase before the next phase begins. This model is named "**Waterfall Model**", because its diagrammatic representation resembles a cascade of waterfalls.

1. **Requirements analysis and specification phase:** The aim of this phase is to understand the exact requirements of the customer and to document them properly. Both the customer and the software developer work together so as to document all the functions, performance, and interfacing requirement of the software. It describes the "what" of the system to be produced and not "how." In this phase, a large document called **Software Requirement Specification (SRS)** document is created which contained a detailed description of what the system will do in the common language.



1. **Design Phase:** This phase aims to transform the requirements gathered in the SRS into a suitable form which permits further coding in a programming language. It defines the overall software architecture together with high level and detailed design. All this work is documented as a Software Design Document (SDD).
2. **Implementation and unit testing:** During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by software developers is contained in the SDD.

During testing, the code is thoroughly examined and modified. Small modules are tested in isolation initially. After that these modules are tested by writing some overhead code to check the interaction between these modules and the flow of intermediate output.

1. **Integration and System Testing:** This phase is highly crucial as the quality of the end product is determined by the effectiveness of the testing carried out. The better output will lead to satisfied customers, lower maintenance costs, and accurate results. Unit testing determines the efficiency of individual modules. However, in this phase, the modules are tested for their interactions with each other and with the system.
2. **Operation and maintenance phase:** Maintenance is the task performed by every user once the software has been delivered to the customer, installed, and operational.

# When to use SDLC Waterfall Model?

Some Circumstances where the use of the Waterfall model is most suited are:

* + When the requirements are constant and not changed regularly.
  + A project is short
  + The situation is calm
  + Where the tools and technology used is consistent and is not changing
  + When resources are well prepared and are available to use.

# Advantages of Waterfall model

* + This model is simple to implement also the number of resources that are required for it is minimal.
  + The requirements are simple and explicitly declared; they remain unchanged during the entire project development.
  + The start and end points for each phase is fixed, which makes it easy to cover progress.
  + The release date for the complete product, as well as its final cost, can be determined before development.
  + It gives easy to control and clarity for the customer due to a strict reporting system.

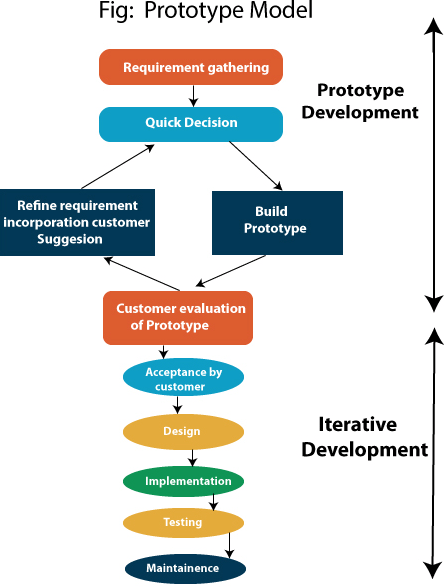
# Disadvantages of Waterfall model

* + In this model, the risk factor is higher, so this model is not suitable for more significant and complex projects.
  + This model cannot accept the changes in requirements during development.
  + It becomes tough to go back to the phase. For example, if the application has now shifted to the coding phase, and there is a change in requirement, It becomes tough to go back and change it.
  + Since the testing done at a later stage, it does not allow identifying the challenges and risks in the earlier phase, so the risk reduction strategy is difficult to prepare.

# Prototype Model

The prototype model requires that before carrying out the development of actual software, a working prototype of the system should be built. A prototype is a toy implementation of the system. A prototype usually turns out to be a very crude version of the actual system, possible exhibiting limited

functional capabilities, low reliability, and inefficient performance as compared to actual software. In many instances, the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs, and the output requirement, the prototyping model may be employed.



# Steps of Prototype Model

1. Requirement Gathering and Analyst
2. Quick Decision
3. Build a Prototype
4. Assessment or User Evaluation
5. Prototype Refinement
6. Engineer Product

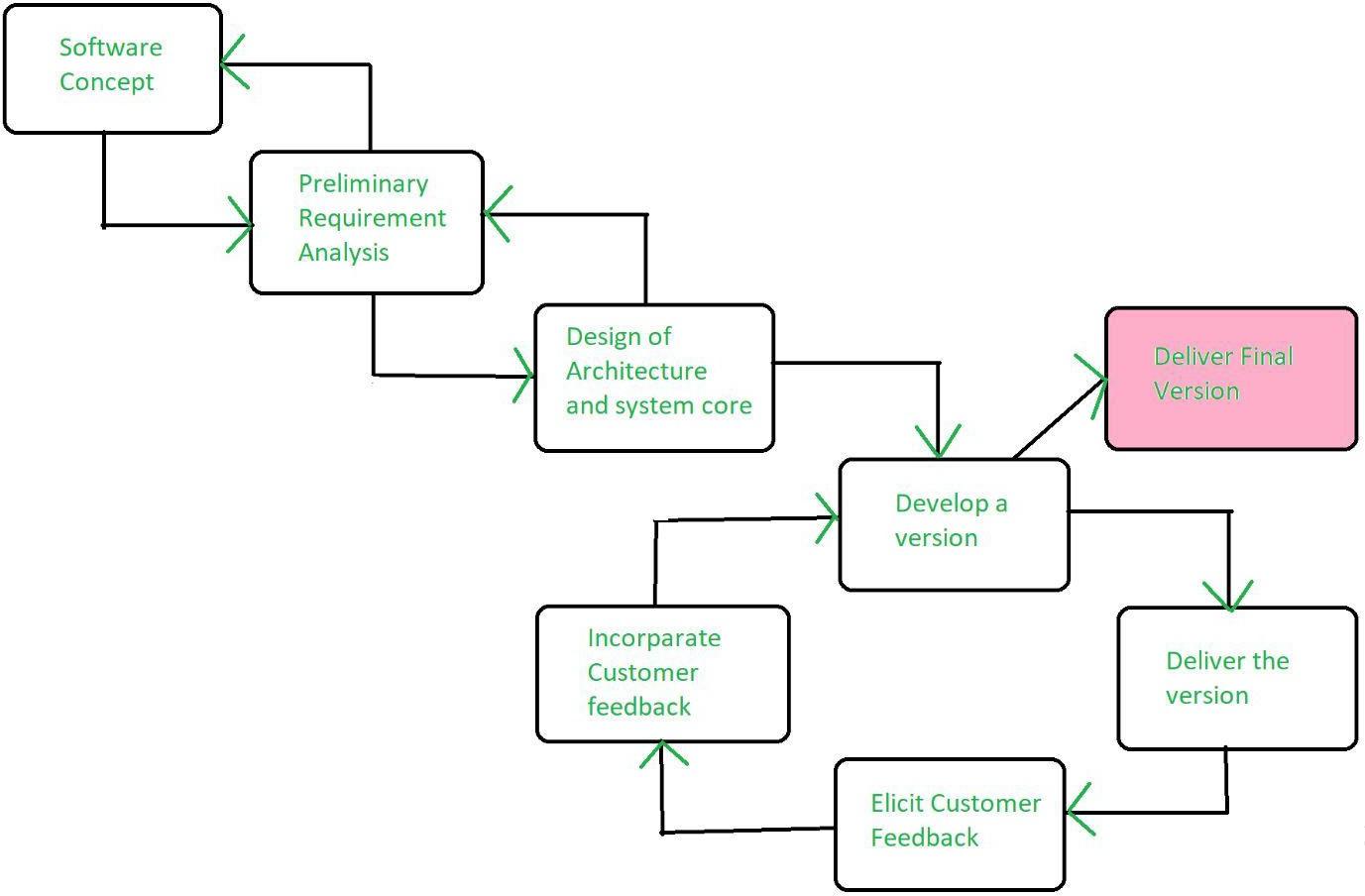
# Advantage of Prototype Model

1. Reduce the risk of incorrect user requirement
2. Good where requirement are changing/uncommitted
3. Regular visible process aids management
4. Support early product marketing
5. Reduce Maintenance cost.
6. Errors can be detected much earlier as the system is made side by side.

# Disadvantage of Prototype Model

1. An unstable/badly implemented prototype often becomes the final product.
2. Require extensive customer collaboration
   * Costs customer money
   * Needs committed customer
   * Difficult to finish if customer withdraw
   * May be too customer specific, no broad market
3. Difficult to know how long the project will last.
4. Easy to fall back into the code and fix without proper requirement analysis, design, customer evaluation, and feedback.
5. Prototyping tools are expensive.
6. Special tools & techniques are required to build a prototype.
7. It is a time-consuming process.

**Evolutionary model** is a combination of [Iterative](https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/) and [Incremental model](https://www.geeksforgeeks.org/software-engineering-incremental-process-model/) of software development life cycle. Delivering your system in a big bang release, delivering it in incremental process over time is the action done in this model. Some initial requirements and architecture envisioning need to be done. It is better for software products that have their feature sets redefined during development because of user feedback and other factors. The Evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. Feedback is provided by the users on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plan or process. Therefore, the software product evolves with time. All the models have the disadvantage that the duration of time from start of the project to the delivery time of a solution is very high. Evolutionary model solves this problem in a different approach.



Evolutionary model suggests breaking down of work into smaller chunks, prioritizing them and then delivering those chunks to the customer one by one. The number of chunks is huge and is the number of deliveries made to the customer. The main advantage is that the customer’s confidence increases as he constantly gets quantifiable goods or services from the beginning of the project to verify and validate his requirements. The model allows for changing requirements as well as all work is broken down into maintainable work chunks.

# Necessary conditions for implementing this model:-

* Customer needs are clear and been explained in deep to the developer team.
* There might be small changes required in separate parts but not a major change.
* As it requires time, so there must be some time left for the market constraints.
* Risk is high and continuous targets to achieve and report to customer repeatedly.
* It is used when working on a technology is new and requires time to learn.

# Advantages:

* In evolutionary model, a user gets a chance to experiment partially developed system.
* It reduces the error because the core modules get tested thoroughly.

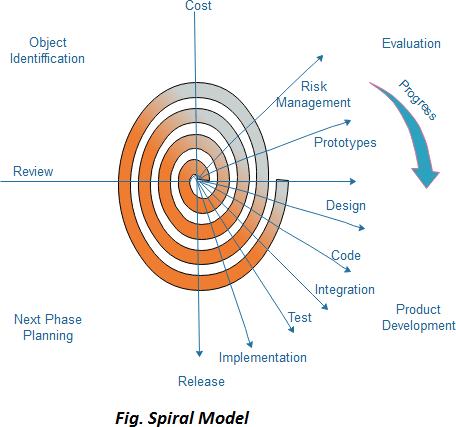
# Disadvantages:

* Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

# Spiral Model

The spiral model, initially proposed by Boehm, is an evolutionary software process model that couples the iterative feature of prototyping with the controlled and systematic aspects of the linear

sequential model. It implements the potential for rapid development of new versions of the software. Using the spiral model, the software is developed in a series of incremental releases. During the early iterations, the additional release may be a paper model or prototype. During later iterations, more and more complete versions of the engineered system are produced.



# Each cycle in the spiral is divided into four parts:

**Objective setting:** Each cycle in the spiral starts with the identification of purpose for that cycle, the various alternatives that are possible for achieving the targets, and the constraints that exists.

**Risk Assessment and reduction:** The next phase in the cycle is to calculate these various alternatives based on the goals and constraints. The focus of evaluation in this stage is located on the risk perception for the project.

**Development and validation:** The next phase is to develop strategies that resolve uncertainties and risks. This process may include activities such as benchmarking, simulation, and prototyping.

**Planning:** Finally, the next step is planned. The project is reviewed, and a choice made whether to continue with a further period of the spiral. If it is determined to keep, plans are drawn up for the next step of the project.

The development phase depends on the remaining risks. For example, if performance or user-interface risks are treated more essential than the program development risks, the next phase may be an evolutionary development that includes developing a more detailed prototype for solving the risks.

The **risk-driven** feature of the spiral model allows it to accommodate any mixture of a specification- oriented, prototype-oriented, simulation-oriented, or another type of approach. An essential element of the model is that each period of the spiral is completed by a review that includes all the products developed during that cycle, including plans for the next cycle. The spiral model works for development as well as enhancement projects.

# When to use Spiral Model ?

* + When deliverance is required to be frequent.
  + When the project is large
  + When requirements are unclear and complex
  + When changes may require at any time
  + Large and high budget projects

# Advantages

* + High amount of risk analysis
  + Useful for large and mission-critical projects.

# Disadvantages

* + Can be a costly model to use.
  + Risk analysis needed highly particular expertise
  + Doesn't work well for smaller projects.

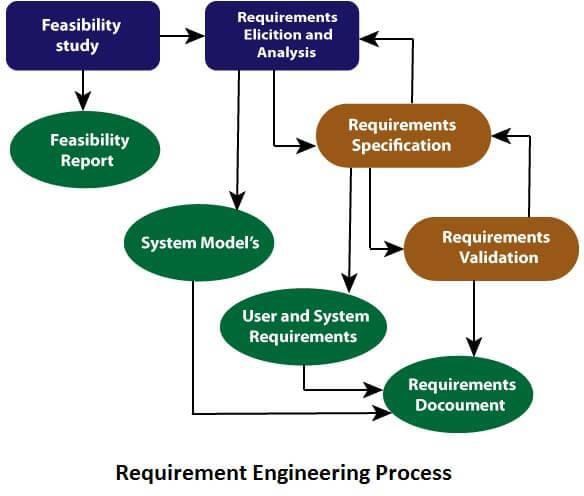
# Requirement Engineering

**Requirements engineering (RE)** refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analysing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirement engineering is the disciplined application of proven principles, methods, tools, and notation to describe a proposed system's intended behaviour and its associated constraints.

# Requirement Engineering Process

It is a four-step process, which includes -

1. Feasibility Study
2. Requirement Elicitation and Analysis
3. Software Requirement Specification
4. Software Requirement Validation
5. Software Requirement Management



# Feasibility Study:

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

# Types of Feasibility:

* 1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
  2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
  3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

# Requirement Elicitation and Analysis:

This is also known as the **gathering of requirements**. Here, requirements are identified with the help of customers and existing systems processes, if available.

Analysis of requirements starts with requirement elicitation. The requirements are analysed to identify inconsistencies, defects, omission, etc. We describe requirements in terms of relationships and also resolve conflicts if any.

# Problems of Elicitation and Analysis

**Quality Function Deployment (QFD)** is process or set of tools used to define the customer requirements for product and convert those requirements into engineering specifications and plans such that the customer requirements for that product are satisfied.

* QFD was developed in late 1960s by Japanese Planning Specialist named Yoji Akao.
* QFD aims at translating Voice of Customer into measurable and detailed design targets and then drives them from the assembly level down through sub-assembly level, component level, and production process levels.
* QFD helps to achieve structured planning of product by enabling development team to clearly specify customer needs and expectations of product and then evaluate each part of product systematically.

# Key steps in QFD :

1. **Product planning :**
   * Translating what customer wants or needs into set of prioritized design requirements.
   * Prioritized design requirements describe looks/design of product.
   * Involves benchmarking – comparing product’s performance with competitor’s products.
   * Setting targets for improvements and for achieving competitive edge.

# Part Planning :

* + Translating product requirement specifications into part of characteristics.
  + For example, if requirement is that product should be portable, then characteristics could be light-weight, small size, compact, etc.

# Process Planning :

* + Translating part characteristics into an effective and efficient process.
  + The ability to deliver six sigma quality should be maximized.

# Production Planning :

* + Translating process into manufacturing or service delivery methods.
  + In this step too, ability to deliver six sigma qualities should be improved.

# Benefits of QFD :

1. **Customer-focused –**

Very first step of QFD is marked by understanding and collecting all user requirements and expectations of product. The company does not focus on what they think customer wants, instead, they ask customers and focus on requirements and expectations put forward by them.

# Voice of Customer Competitor Analysis –

House of Quality is significant tool that is used to compare voice of customer with design specifications.

# Structure and Documentation –

Tools used in Quality Function Deployment are very well structured for capturing decisions made and lessons learned during development of product. This documentation can assist in development of future products.

# Low Development Cost –

Since QFD focuses and pays close attention to customer requirements and expectations in initial steps itself, so the chances of late design changes or modifications are highly reduced, thereby resulting in low product development cost.

# Shorter Development Time –

QFD process prevents wastage of time and resources as enough emphasis is made on customer needs and wants for the product. Since customer requirements are understood and developed in right way, so any development of non-value-added features or unnecessary functions is avoided, resulting in no time waste of product development team.

1. ​



1. Software Requirement Specification:

Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.

The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

* **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.
* **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.
* **Entity-Relationship Diagrams:** Another tool for requirement specification is the entity- relationship diagram, often called an "***E-R diagram***." It is a detailed logical representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes.

1. Software Requirement Validation:

After requirement specifications developed, the requirements discussed in this document are validated. The user might demand illegal, impossible solution or experts may misinterpret the needs. Requirements can be the check against the following conditions -

* If they can practically implement
* If they are correct and as per the functionality and specially of software
* If there are any ambiguities
* If they are full
* If they can describe

# Requirements Validation Techniques

* **Requirements reviews/inspections:** systematic manual analysis of the requirements.
* **Prototyping:** Using an executable model of the system to check requirements.
* **Test-case generation:** Developing tests for requirements to check testability.
* **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

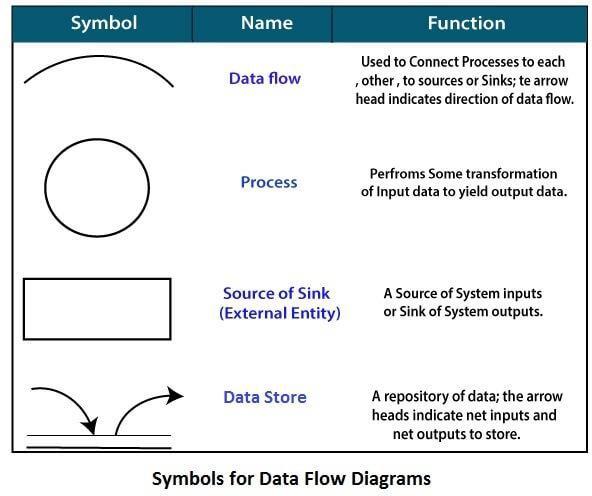
Data Flow Diagrams

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. It can be manual, automated, or a combination of both.

It shows how data enters and leaves the system, what changes the information, and where data is stored.

The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in fig:



**Circle:** A circle (bubble) shows a process that transforms data inputs into data outputs.

**Data Flow:** A curved line shows the flow of data into or out of a process or data store.

**Data Store:** A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have an element or group of elements.

**Source or Sink:** Source or Sink is an external entity and acts as a source of system inputs or sink of system outputs.

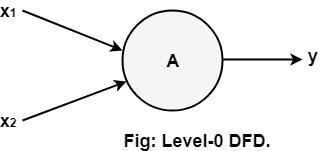
Levels in Data Flow Diagrams (DFD)

The DFD may be used to perform a system or software at any level of abstraction. Infact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

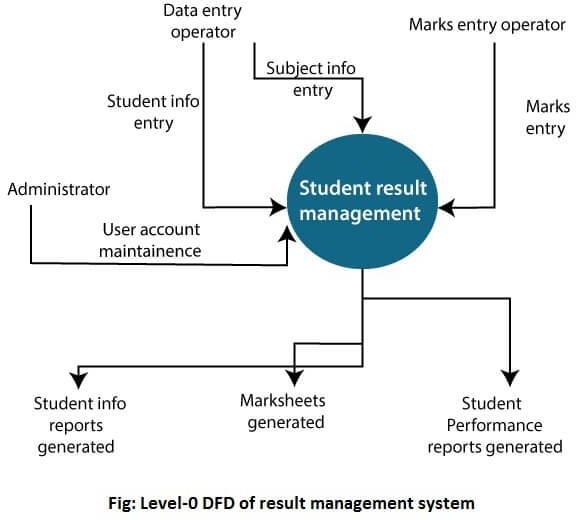
# level DFDM

It is also known as fundamental system model, or context diagram represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows. Then the system is decomposed and described as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the program at hand is well understood. It is essential to preserve the number of inputs and outputs between levels, this concept is

called leveling by DeMacro. Thus, if bubble "A" has two inputs x1 and x2 and one output y, then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in fig:

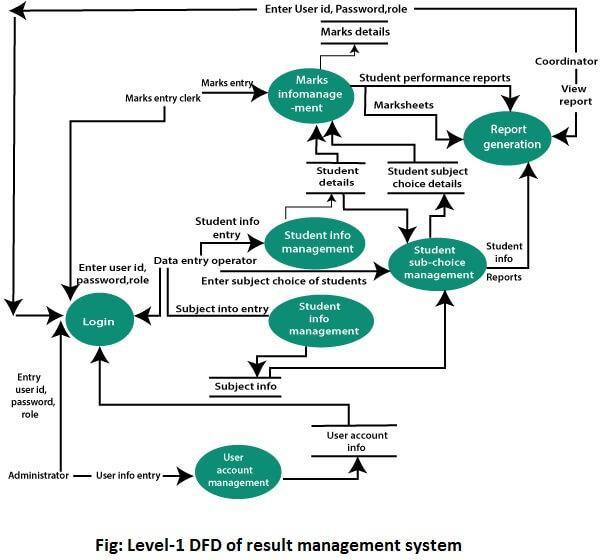


The Level-0 DFD, also called context diagram of the result management system is shown in fig. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flow may also be needed to be decomposed.



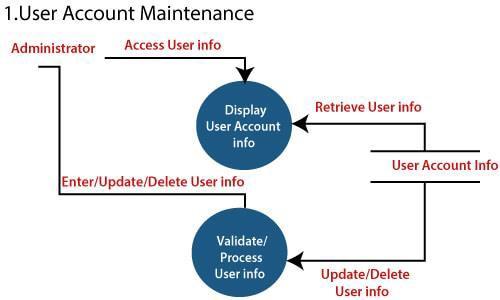
# level DFD

In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main objectives of the system and breakdown the high-level process of 0-level DFD into subprocesses.



# Level DFD

2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.



1. Login

11he levet 2 DFO of this pto,:;ess 1'5,given below:



lllser Account info Retrieve User info

| EnterUser D, try Password,Role |  | |
| --- | --- | --- |
|  | | |
| Enter User D, Password, Role | | , |

Ent r User ID, **Data** Ent ry P\_a\_,ss\_·wo\_r\_d\_.R\_o\_J\_e.. Operator

Enter User 110,

..\_....::P ::a::,,;,s.;,.sw;.;.·.;,.or;.;d;;;;,R;.;.o,;.,;l;..;;e\_--t..oordina,tot

Marks En

**,clerk**

Administrato,r

1. Student lnfo,rmation Managem.ent

Data entry

Operator

Access User info

Retrieve student Info

St\.ldent Info

Entet/1.Jpdate/Del,ete Uset Info

Validate/ Proce5s

student info Update/Delete

Userinfa

1. Subject lnformati,on Management

The leve'I'2 DFD **ofll,js process** i!. **given** :belowi

Data Entry

Operato

Access Subject Info

Retrieve Subject llflfo

SubJect I to

Enter/Update/Delete Subject Info

Valldate:s/ Process

Subject Info

Update/Delete

Subject Info



Data Dictionaries

A data dictionary is a file or a set of files that includes a database's metadata. The data dictionary hold records about other objects in the database, such as data ownership, data relationships to other objects, and other data. The data dictionary is an essential component of any relational database. Ironically, because of its importance, it is invisible to most database users. Typically, only database administrators interact with the data dictionary.

The data dictionary, in general, includes information about the following:

* Name of the data item
* Aliases
* Description/purpose
* Related data items
* Range of values
* Data structure definition/Forms

The **name of the data item** is self-explanatory.

**Aliases** include other names by which this data item is called DEO for Data Entry Operator and DR for Deputy Registrar.

**Description/purpose** is a textual description of what the data item is used for or why it exists.

**Related data items** capture relationships between data items e.g., total\_marks must always equal to internal\_marks plus external\_marks.

**Range of values** records all possible values, e.g. total marks must be positive and between 0 to 100.

**Data structure Forms:** Data flows capture the name of processes that generate or receive the data items. If the data item is primitive, then data structure form captures the physical structures of the data item. If the data is itself a data aggregate, then data structure form capture the composition of the data items in terms of other data items.

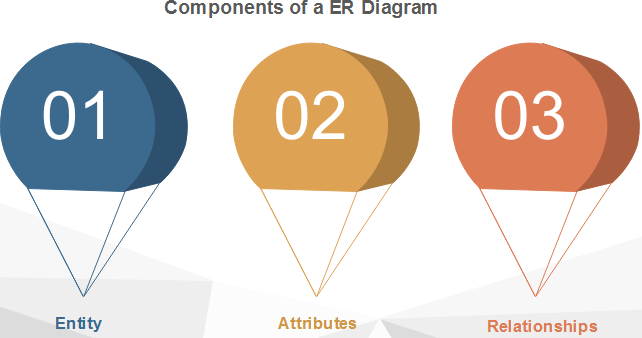
Entity-Relationship Diagrams

ER-modeling is a data modeling method used in software engineering to produce a conceptual data model of an information system. Diagrams created using this ER-modeling method are called Entity- Relationship Diagrams or ER diagrams or ERDs.

Purpose of ERD

* The database analyst gains a better understanding of the data to be contained in the database through the step of constructing the ERD.
* The ERD serves as a documentation tool.
* Finally, the ERD is used to connect the logical structure of the database to users. In particular, the ERD effectively communicates the logic of the database to users.

Components of an ER Diagrams



1. Entity

An entity can be a real-world object, either animate or inanimate, that can be merely identifiable. An entity is denoted as a rectangle in an ER diagram. For example, in a school database, students, teachers, classes, and courses offered can be treated as entities. All these entities have some attributes or properties that give them their identity.

# Entity Set

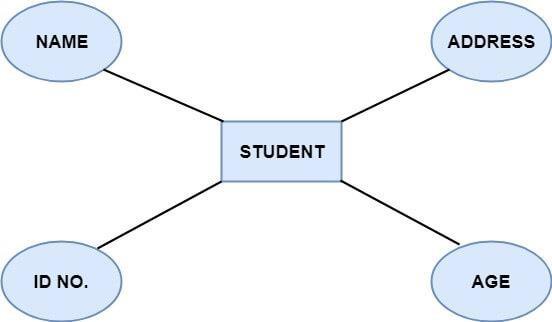
An entity set is a collection of related types of entities. An entity set may include entities with attribute sharing similar values. For example, a Student set may contain all the students of a school; likewise, a Teacher set may include all the teachers of a school from all faculties. Entity set need not be disjoint.



1. Attributes

Entities are denoted utilizing their properties, known as attributes. All attributes have values. For example, a student entity may have name, class, and age as attributes.

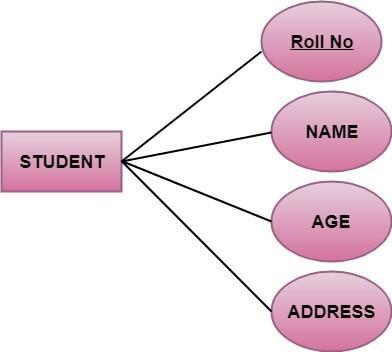
There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.



# There are four types of Attributes:

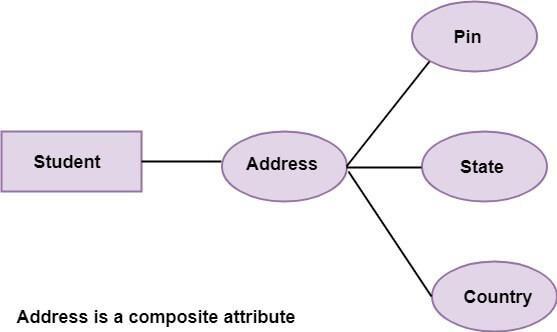
* 1. Key attribute
  2. Composite attribute
  3. Single-valued attribute
  4. Multi-valued attribute
  5. Derived attribute

1. **Key attribute:** Key is an attribute or collection of attributes that uniquely identifies an entity among the entity set. For example, the roll\_number of a student makes him identifiable among students.

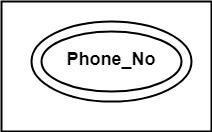


# There are mainly three types of keys:

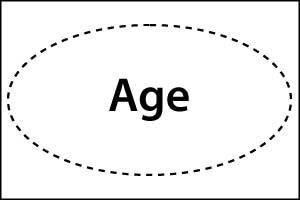
1. **Super key:** A set of attributes that collectively identifies an entity in the entity set.
2. **Candidate key:** A minimal super key is known as a candidate key. An entity set may have more than one candidate key.
3. **Primary key:** A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.
4. **Composite attribute:** An attribute that is a combination of other attributes is called a composite attribute. For example, In student entity, the student address is a composite attribute as an address is composed of other characteristics such as pin code, state, country.

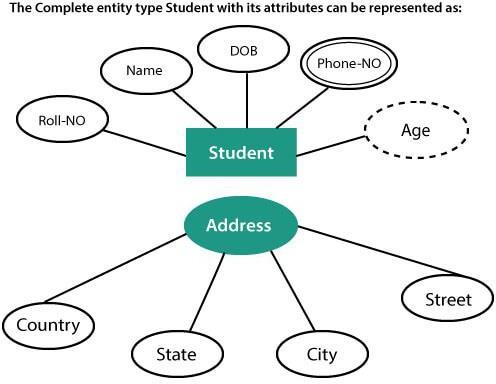


1. **Single-valued attribute:** Single-valued attribute contain a single value. For example, Social\_Security\_Number.
2. **Multi-valued Attribute:** If an attribute can have more than one value, it is known as a multi- valued attribute. Multi-valued attributes are depicted by the double ellipse. For example, a person can have more than one phone number, email-address, etc.



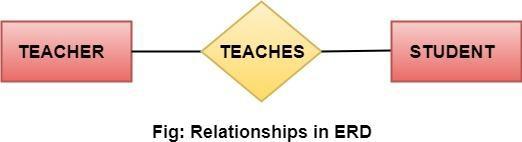
1. **Derived attribute:** Derived attributes are the attribute that does not exist in the physical database, but their values are derived from other attributes present in the database. For example, age can be derived from date\_of\_birth. In the ER diagram, Derived attributes are depicted by the dashed ellipse.





1. Relationships

The association among entities is known as relationship. Relationships are represented by the diamond-shaped box. For example, an employee works\_at a department, a student enrolls in a course. Here, Works\_at and Enrolls are called relationships.

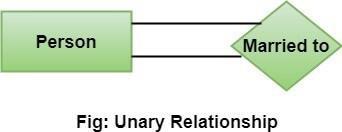


Degree of a relationship set

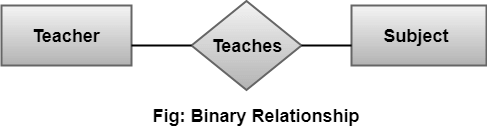
The number of participating entities in a relationship describes the degree of the relationship. The three most common relationships in E-R models are:

* 1. Unary (degree1)
  2. Binary (degree2)
  3. Ternary (degree3)

1. **Unary relationship:** This is also called recursive relationships. It is a relationship between the instances of one entity type. For example, one person is married to only one person.

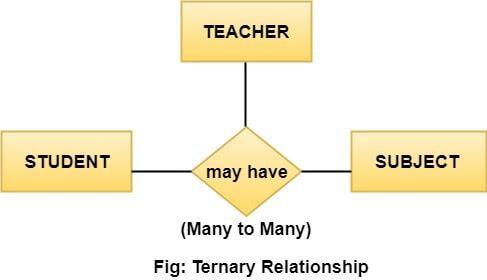


1. **Binary relationship:** It is a relationship between the instances of two entity types. For example, the Teacher teaches the subject.



1. **Ternary relationship:** It is a relationship amongst instances of three entity types. In fig, the relationships "**may have**" provide the association of three entities, i.e., TEACHER, STUDENT, and SUBJECT. All three entities are many-to-many participants. There may be one or many participants in a ternary relationship.

In general, "**n**" entities can be related by the same relationship and is known as **n-ary** relationship.



Cardinality

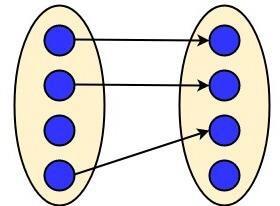
Cardinality describes the number of entities in one entity set, which can be associated with the number of entities of other sets via relationship set.

Types of Cardinalities

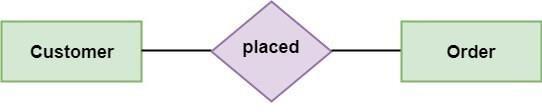
1. **One to One:** One entity from entity set A can be contained with at most one entity of entity set B and vice versa. Let us assume that each student has only one student ID, and each student ID is assigned to only one person. So, the relationship will be one to one.



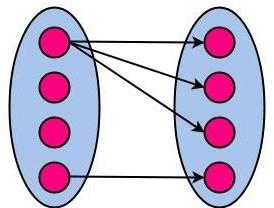
# Using Sets, it can be represented as:



1. **One to many:** When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationships. For example, a client can place many orders; a order cannot be placed by many customers.



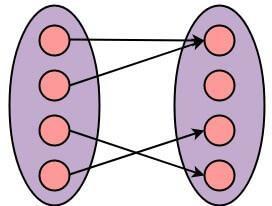
# Using Sets, it can be represented as:



1. **Many to One:** More than one entity from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A. For example - many students can study in a single college, but a student cannot study in many colleges at the same time.



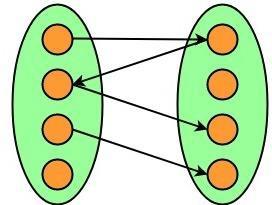
# Using Sets, it can be represented as:



1. **Many to Many:** One entity from A can be associated with more than one entity from B and vice- versa. For example, the student can be assigned to many projects, and a project can be assigned to many students.



# Using Sets, it can be represented as:



**Requirements documentation**

[Requirements](https://en.wikipedia.org/wiki/Requirement) documentation is the description of what a particular software does or shall do. It is used throughout [development](https://en.wikipedia.org/wiki/Software_development) to communicate how the software functions or how it is intended to operate. It is also used as an agreement or as the foundation for agreement on what the software will do. Requirements are produced and consumed by everyone involved in the production of software, including: [end users,](https://en.wikipedia.org/wiki/End_user) [customers,](https://en.wikipedia.org/wiki/Customer) [project managers,](https://en.wikipedia.org/wiki/Project_manager) [sales,](https://en.wikipedia.org/wiki/Sales) [marketing,](https://en.wikipedia.org/wiki/Marketing) [software architects,](https://en.wikipedia.org/wiki/Software_architect) [usability](https://en.wikipedia.org/wiki/Usability_engineering) [engineers,](https://en.wikipedia.org/wiki/Usability_engineering) [interaction designers,](https://en.wikipedia.org/wiki/Interaction_design) [developers,](https://en.wikipedia.org/wiki/Software_developer) and [testers.](https://en.wikipedia.org/wiki/Software_testing)

Requirements come in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started by right-clicking a configuration file and selecting the 'build' function*), and anything in between. They can be specified as statements in [natural language,](https://en.wikipedia.org/wiki/Natural_language) as drawn figures, as detailed [mathematical formulas,](https://en.wikipedia.org/wiki/Mathematical_formula) or as a combination of them all.

The variation and complexity of requirement documentation make it a proven challenge. Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people who shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult — and therefore more error prone (decreased [software quality](https://en.wikipedia.org/wiki/Software_quality)) and time-consuming (expensive).

**Nature of SRS-** An SRS minimizes the time and effort required by developers to achieve desired goals and also minimizes the development cost. A good SRS defines how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations.

Following are the characteristics of a good SRS document:

# Correctness:

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

# Completeness:

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

# Consistency:

Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

# Unambiguousness:

A SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

# Ranking for importance and stability:

There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

# Modifiability:

SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross- referenced.

# Verifiability:

A SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement starting that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

# Traceability:

One should be able to trace a requirement to design component and then to code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

# Design Independence:

There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

# Testability:

A SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

# Understandable by the customer:

An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

# Right level of abstraction:

If the SRS is written for the requirements phase, the details should be explained explicitly.

Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the SRS.