Software Requirements: Analysis and Specification

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team (system analysts) carry out this job.

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purposes. Requirements describe the "what" of a system not the 'how".

Requirement engineering

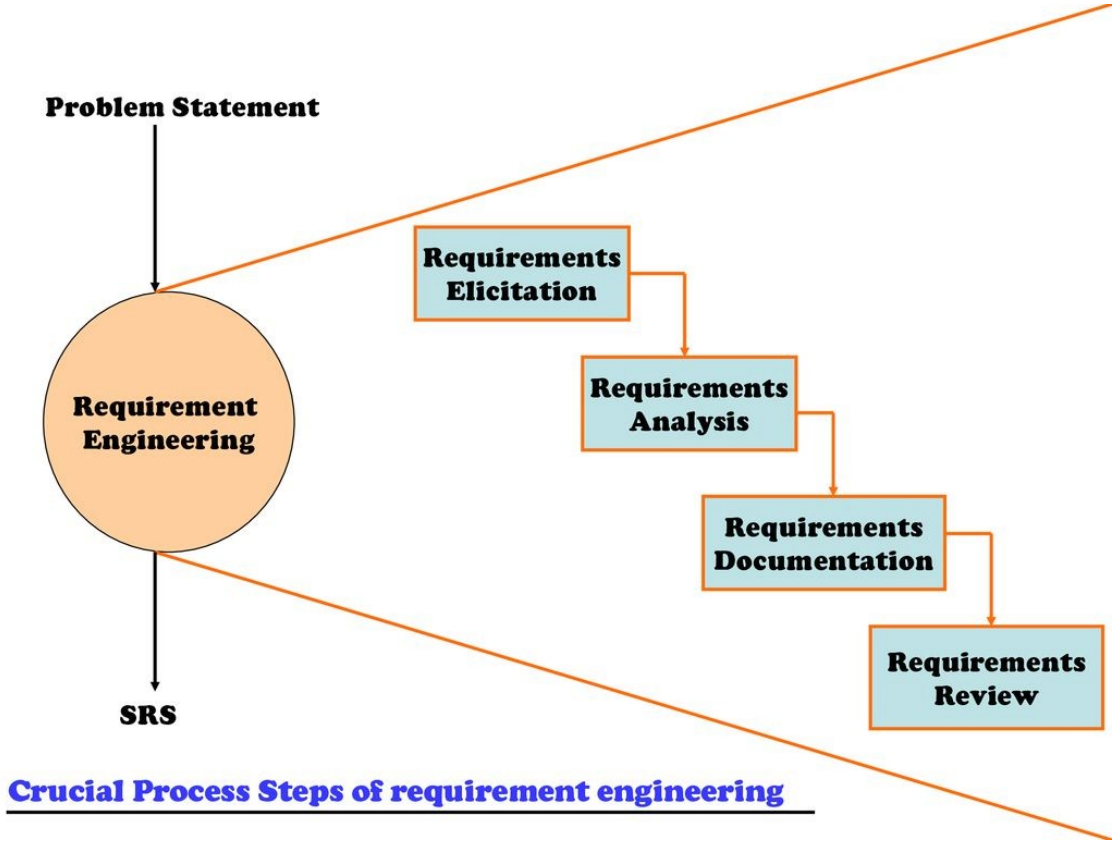
Requirement Engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behaviour and its associated constraints. Requirement engineering is the process of defining, documenting and maintaining the requirements. The input to requirements engineering is the problem statement prepared by the customer. The problem statement may give an overview of the existing system along with broad expectations of the new system. It produces SRS. SRS has three main uses

It acts as a contract between customer and developer

It is used as the reference document for designers

It is used as the reference document for validation at the end.

Crucial process steps:- Requirement engineering process consists of the following main activities:



Crucial Process Steps of requirement engineering

- **Requirements elicitation:** Requirements are gathered with the help of customers and
- existing systems.
- **Requirements analysis:** The collected requirements are analysed to identify defects, omissions and inconsistencies etc.
- **Requirements documentation:** SRS is prepared in this step.
- **Requirements review:** SRS is verified to ensure quality.

Present state of practice:- There are several reasons which make it difficult to improve requirement engineering practices. Some are

1. **Requirements are difficult to uncover:** Today we are automating every kind of work and it is very difficult to identify every requirement in the beginning.
2. **Requirements change:** Because no one can prepare complete list of requirements in the beginning, requirements get added and modified as project proceeds.
3. **Over-reliance of CASE tools:** The exaggerated claims of tools promote unrealistic expectations from them. Moreover, developers depend on them without understanding requirement engineering principles and techniques.
4. **Tight project schedule:** Most projects get insufficient time due to lack of planning and unreasonable customer demand.
5. **Communication barriers:** Requirement engineering is a communication intensive activity and communication gap may occur between customers and developers due to difference in vocabularies, professional background and tastes.
6. **Market driven software development:** Many softwares are developed for general marketing and thus has to satisfy anonymous customers.
7. **Lack of resources:** There may not be enough resources to build complete software. Then suitable process models must be selected.

Types of Requirements

There are different types of requirements such as

i. Known requirements: Something a stakeholder believes to be implemented.
ii. Unknown requirements: Forgotten by one stakeholder because they are not needed now or needed only by another stakeholder.
iii. Undreamt requirements: Stakeholder may not be able to think of new requirements due to limited domain knowledge.

Stakeholder: Anyone who should have some direct or indirect influence on the system requirements.

A known, Unknown, Undreamt requirement may be functional or non-functional.

Functional and Non-functional Requirements

Software requirements are broadly classified as functional and non-functional requirements

- **Functional requirements:** Functional requirements describe what the software has to do. They are often called product features. These are directly related to customer's expectations and are essential for the acceptance of the product.
- **Non Functional requirements:** Non Functional requirements are mostly quality requirements that stipulate how well the software does, what it has to do. These requirements make customers happy and satisfied. Requirements important to users include availability, reliability, usability and flexibility and important to developers include maintainability, portability and testability.

User and system requirements:

- **User requirements** are written for the users and include functional and non functional requirements. Users may not be the experts of the software field; hence simple language should be used. The software terminologies, notations etc. should be avoided. User requirements should specify the external behaviour of the system with some constraints and quality parameters. It highlight only the overview of system without design characteristics.
- **System requirements** are derived from user requirements. They are expanded form of user requirements. They may be used as input to the designers for the preparation of software design document hence uses technical terms. The user and system requirements are the parts of software requirement and specification (SRS) document.

Interface Specification:- Important for the customers.

The types of interfaces are

i. Procedural interfaces (also called Application Programming Interfaces (APIs)).
ii. Data structures
iii. Representation of data(different ordering of bits).

Feasibility study

Feasibility study is defined as “evaluation or analysis of the potential impact of a proposed project or program.”
When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software. Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Main objectives of feasibility study are listed below.

- To analyze whether the software will meet organizational requirements

- To determine whether the software can be implemented using the current technology and within the specified budget and schedule
- To determine whether the software can be integrated with other existing software.

Types of Feasibility

Various types of feasibility that are commonly considered include technical feasibility, operational feasibility, and economic feasibility.

Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, the software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility also performs the following tasks.

- Analyzes the technical skills and capabilities of the software development team members
- Determines whether the relevant technology is stable and established
- Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether the software will operate after it is developed and be operative once it is installed. Operational feasibility also performs the following tasks.

- Determines whether the problems anticipated in user requirements are of high priority
- Determines whether the solution suggested by the software development team is acceptable
- Analyzes whether users will adapt to a new software
- Determines whether the organization is satisfied by the alternative solutions proposed by the software development team.

Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below.

- Cost incurred on software development to produce long-term gains for an organization
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
- Cost of hardware, software, development team, and training.

Requirements elicitation

Requirements elicitation is perhaps the most difficult, most error-prone and most communication intensive software development. It can be successful only through an effective customer-developer partnership. It is needed to know what the users really need. There are a number of requirements elicitation methods. Few of them are listed below -

- Interviews
- Brainstorming Sessions
- Facilitated Application Specification Technique (FAST)
- Quality Function Deployment (QFD)
- Use Case Approach

The success of an elicitation technique used depends on the maturity of the analyst, developers, users and the customer involved.

Interviews:

Objective of conducting an interview is to understand the customer's expectations from the software. Normally, requirement engineers arrange interview with the customer with an understanding that both want project to be a success though their feelings, goals, opinions and knowledge are different. Requirement engineers must be open minded and should not approach the interview with pre-conceived ideas about what is required.

Interviews may be be open ended or structured. In open ended interviews there is no pre-set agenda. Context free questions may be asked to understand the problem. In structured interview, agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview.

It is impossible to interview every stakeholder hence representatives from groups are selected based on their expertise, domain knowledge, accessibility and credibility. There are several groups to be considered for conducting interviews.

Entry level personnel:- They may not have sufficient domain knowledge and experience, but may be very useful for fresh ideas and different views.

Mid-level stakeholders:- They have better domain knowledge and experience of the project. They can answer complex and critical questions about project.

Managers: Higher level management like MD, CEO should also be interviewed. They can give information in a different view point.

Users of the software: Most useful information may get from them as they spend more time with the system.

Types of questions should be simple and short. Compound requirements statement should be avoided

Brainstorming Sessions:

- It is a group discussion technique

- It may lead to lots of new ideas quickly and help to promote creative thinking.
- It provide a platform to share views and all participants are encouraged to say whatever come into their mind. No one will be criticized for any idea.
- Discussions may be conducted with specialized groups like actual users
- A highly trained facilitator is required to handle group bias and group conflicts.
- Every idea is documented so that everyone can see it using projectors or boards etc.
- Finally a document is prepared which consists of the list of requirements and their priority if possible.

Facilitated Application Specification Technique (FAST):

Its objective is to bridge the expectation gap—difference between what the developers think they are supposed to build and what customers think they are going to get. A team oriented approach is developed for requirements gathering.

The basic guidelines for FAST are given below:

- Arrange a meeting at a neutral site for customers and developers.
- Establish rules for preparation and participation.
- Informal agenda to encourage free flow of ideas.
- Appoint a facilitator.
- Prepare definition mechanism board, worksheets, wall stickier etc.
- Participants should not criticize or debate.

Fast session preparations

Each FAST attendee is asked to make a list of objects that are-

1. Part of the environment that surrounds the system
2. Produced by the system
3. Used by the system

In addition, list of services, constrains and performance criteria are also prepared.

Activities of FAST session may have the following steps:

- Every participant presents his/her list.
- Combined list for each topic is prepared (redundant entries are eliminated).
- Discussion is conducted and consensus list is prepared.
- Team is divided into smaller sub-teams to develop mini-specifications for entries in list.
- Sub teams present mini specifications to all attendees. After discussion list is revised.
- An issue list is prepared recording issues not solved.
- A consensus list of validation criteria is created.

- A sub team prepare a draft of specifications using all the inputs from the meeting.

Quality Function Deployment:

In this technique customer satisfaction is of prime concern, hence it emphasizes on the requirements which are valuable to the customer.

Three types of requirements are identified -

Normal requirements- These requirements are explicitly stated by the customer and are essential for product acceptance. Example- normal requirements for a result management system may be entry of marks, calculation of results etc.

Expected requirements- These requirements are so obvious that the customer need not explicitly state them. Example- protection from unauthorised access.

Exciting requirements- It includes features that are beyond customer's expectations and prove to be very satisfying when present. Example- when an unauthorised access is detected, it should backup and shutdown all processes.

The major steps involved in this procedure are -

1. Identify all the stakeholders e.g., Users, developers, customers etc.
2. List out all requirements from customer.
3. A value indicating degree of importance is assigned to each requirement.

   5 Points: Very Important

   4 Points: Important

   3 Points: Not Important but nice to have

   2 Points: Not important

   1 Points: Unrealistic, required further exploration
4. In the end the final list of requirements is categorised as -

- It is possible to achieve
- It should be deferred and the reason for it
- It is impossible to achieve and should be dropped off

The first category requirement will be implemented as per priority (Importance value) assigned with every requirement. If time and effort permits, second category requirements may be reviewed and few of them may be transferred to category first for implementation.

Use Case Approach:

Ivar Jacobson and others introduced Use Case approach for elicitation and modeling, This technique combines text and pictures to provide a better understanding of the requirements. The use cases describe the 'what', of a system and not 'how'. Hence they only give a functional view of the system.

The terms Use Case, Use Case Scenario and Use Case Diagram are often interchanged but they are different. Use Cases are structured outline or template for the description of user requirements modeled in a structured language like English. Use Case Scenarios are unstructured description of user requirements. Use Case Diagrams are graphical representation of functional aspect of system.

The components of the use case design approach are – **Actor, Use cases, use case diagram.**

Actor- It is the external agent that lies outside the system but interacts with it in some way. An actor may be a person, machine etc. It is represented as a stick figure. Customers, users, external devices, or any external entity interacting with the system are treated as actors. Actors can be primary actors or secondary actors.

Primary actors- It requires assistance from the system to achieve a goal.

Secondary actor- It is an actor from which the system needs assistance.

Use cases- They describe the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal. They capture who(actors) do what(interaction) with the system, for what purpose(goal), without dealing with system internals. A complete set of use cases specifies all possible ways to use the system. Use cases are written in an easy to understand structured narrative.

There is no standard use case template for writing use cases. Jacobson proposed a template for writing use cases and is given below.

1. Introduction: Describe a quick background of the use case.

2. Actors:- List the actors that interact and participate in the use cases.

3. Flow of events

3.1. Basic Flow:- List the primary events that will occur when this use case is executed.

3.2. Alternative Flows:- Any Subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow. A use case can have many alternative flows as required.

4.Special Requirements:- Business rules should be listed for basic & information flows as special requirements in the use case narration. These rules will also be used for writing test cases. Both success and failures scenarios should be described.

5. Pre Conditions:-Pre conditions that need to be satisfied for the use case to perform.

6. Post Conditions:-Define the different states in which we expect the system to be in, after the use case executes.

7. Extension Points.

<u>An example of use case</u>

<u>Login</u>

1.Introduction: This use case describes how a user logs into the Result Management System.

2 Actors: Data Entry Operator, Administrator, Marks entry clerk and Co-ordinator.

3.Flow of events

3.1 Basic Flow: This use case starts when the actor wishes to login to the Result Management system.

(i) System requests that the actor enter his/her name and password.

(ii) The actor enters his/her name & password.

(iii) System validates name & password, and if finds correct allow the actor to logs into the system.

3.2 Alternate Flows: Invalid name & password- If in the basic flow, the actor enters an invalid name and/or password, the system displays an error message. The actor can choose to either return to the beginning of the basic flow or cancel the login.

4 Special Requirements: None

5. Pre Conditions: All users must have a user account in the system.

6. Post Conditions: If the use case is successful, the actor is logged into the system. If not, the system state is unchanged.

7. Extension Points: None

Use Case Guidelines

The following provides an outline of a process for creating use cases:

- Identify all users.
- Create a user profile for each category of users including all roles of the users play that are relevant to the system. For each role, identify all goals of users.
- Create a use case for each goal, following the use case template.
- Structure the use case.
- Review and validate with users.

Use case diagram- A use case diagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system. The system is shown as a rectangle with the name of the system inside.

- A stick figure is used to represent an actor.
- An oval is used to represent a use case.
- A line is used to represent a relationship between an actor and a use case.

Components of use case diagram

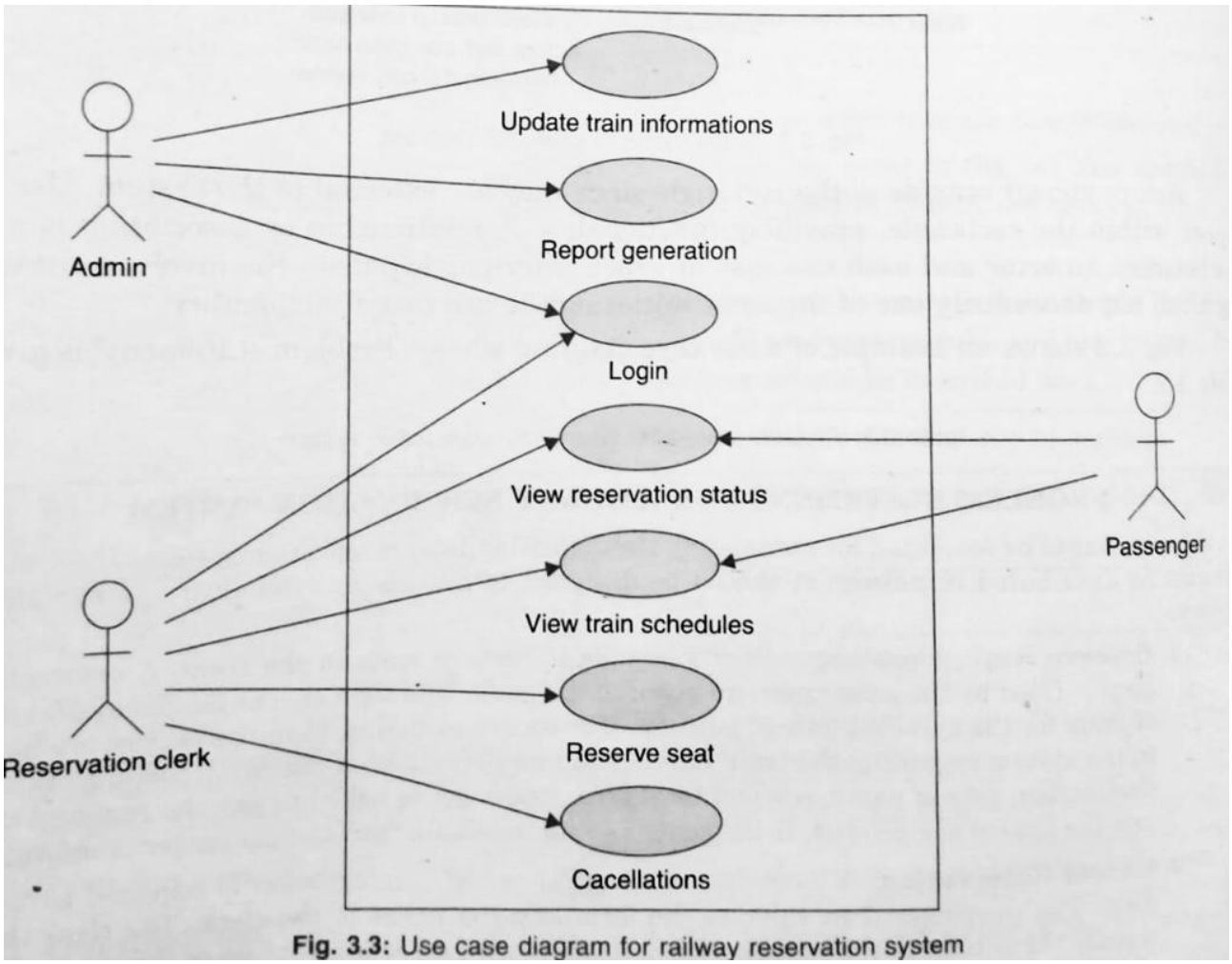
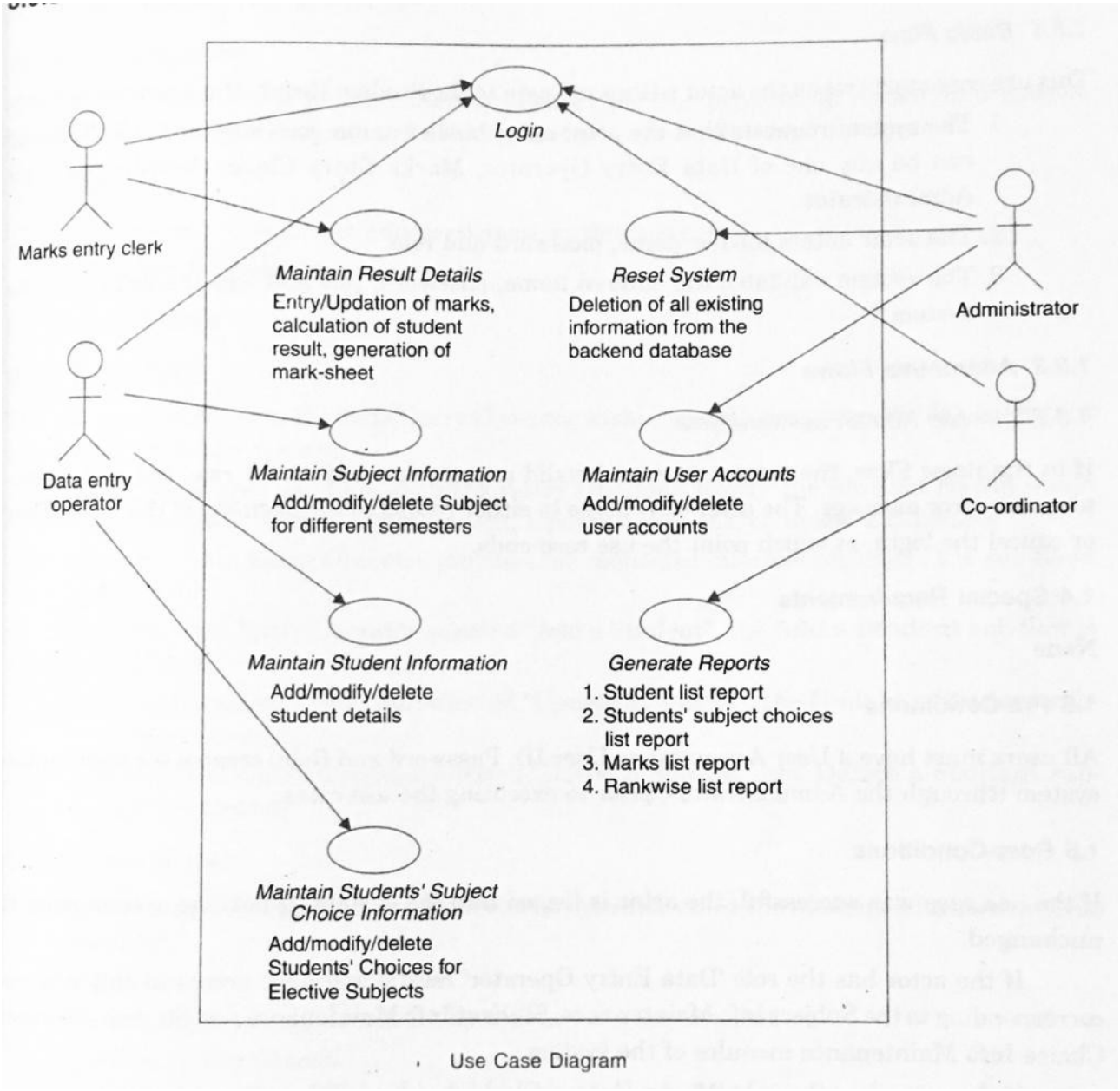


Fig. 3.3: Use case diagram for railway reservation system



Use Case Diagram

Requirements Analysis

We analyze, refine and scrutinize gathered requirements to make consistent and unambiguous requirements. The various steps of requirements Analysis are shown in Fig. 3.4.

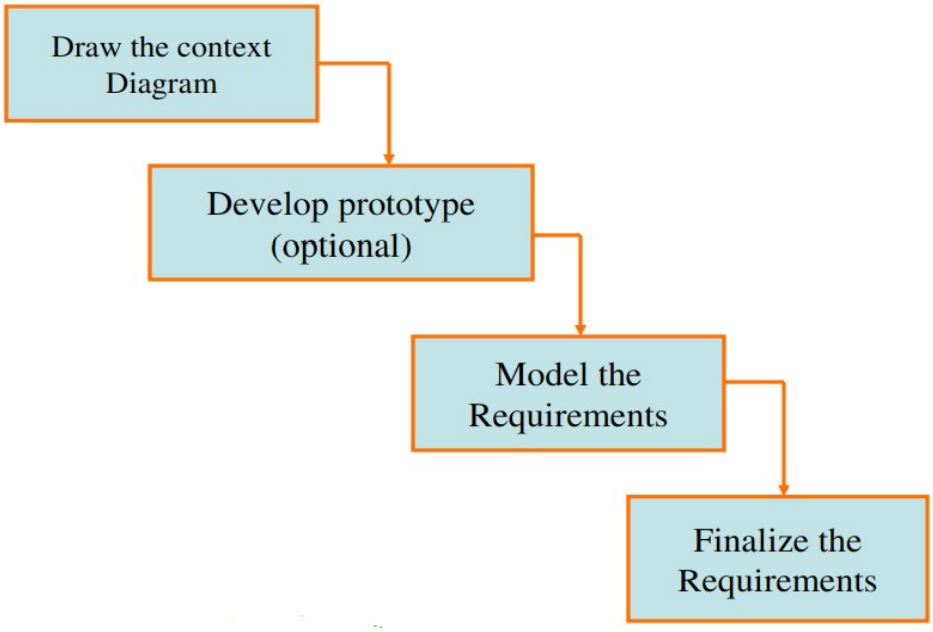


Fig. 3.4: Requirements analysis steps

Draw the context Diagram:- The context diagram (level 0 DFD) is a simple model that represent the entire system as a single bubble with input and output data as shown below.

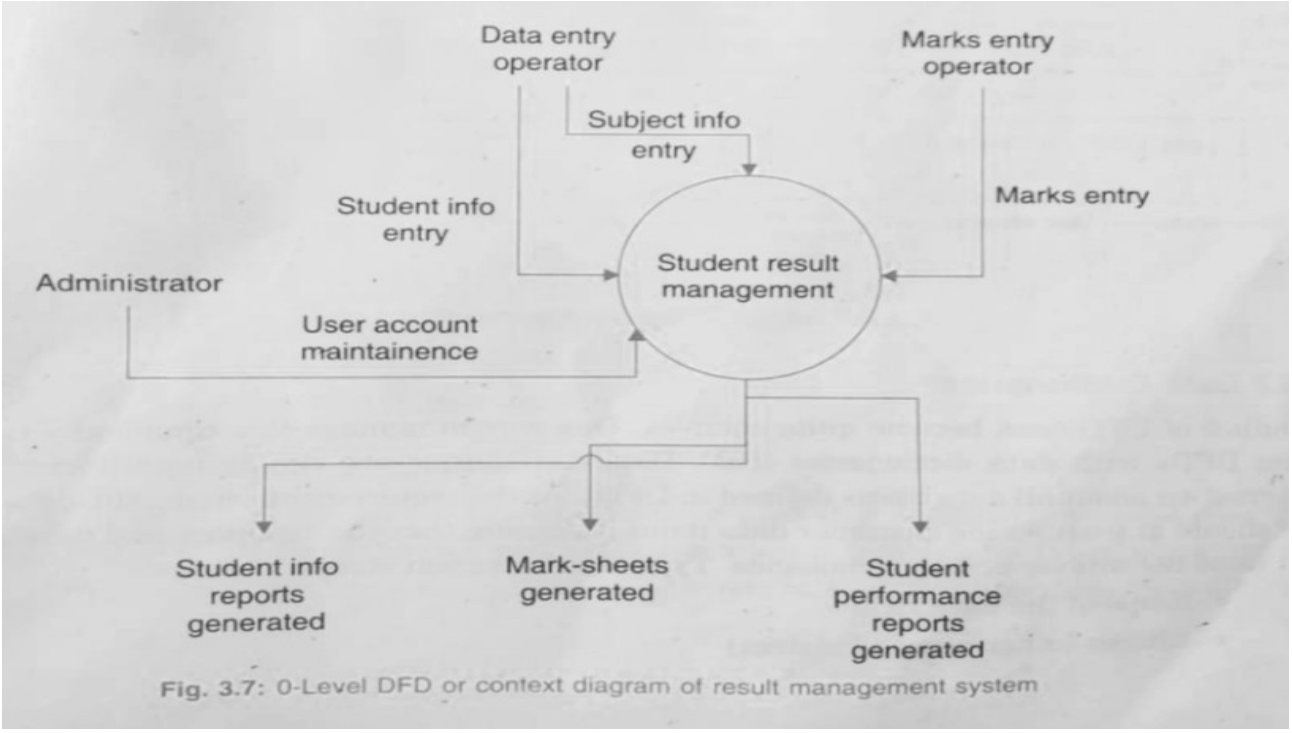


Fig. 3.7: 0-Level DFD or context diagram of result management system

Develop prototype (optional):- A prototype helps the client to visualize the proposed system and increases the understanding of requirements. When developers and users are not certain about some of the requirements, a prototype may help both parties to take a final decision. Prototypes are continuously modified based on the feedback from the customer until they are satisfied.

Model the requirements:- Different types of models like data flow diagrams, ER diagrams and data dictionaries, state transition diagrams are developed in this step which help for verification.

Finalize the Requirements:- The verified requirements are finalized and next step is to document these requirements.

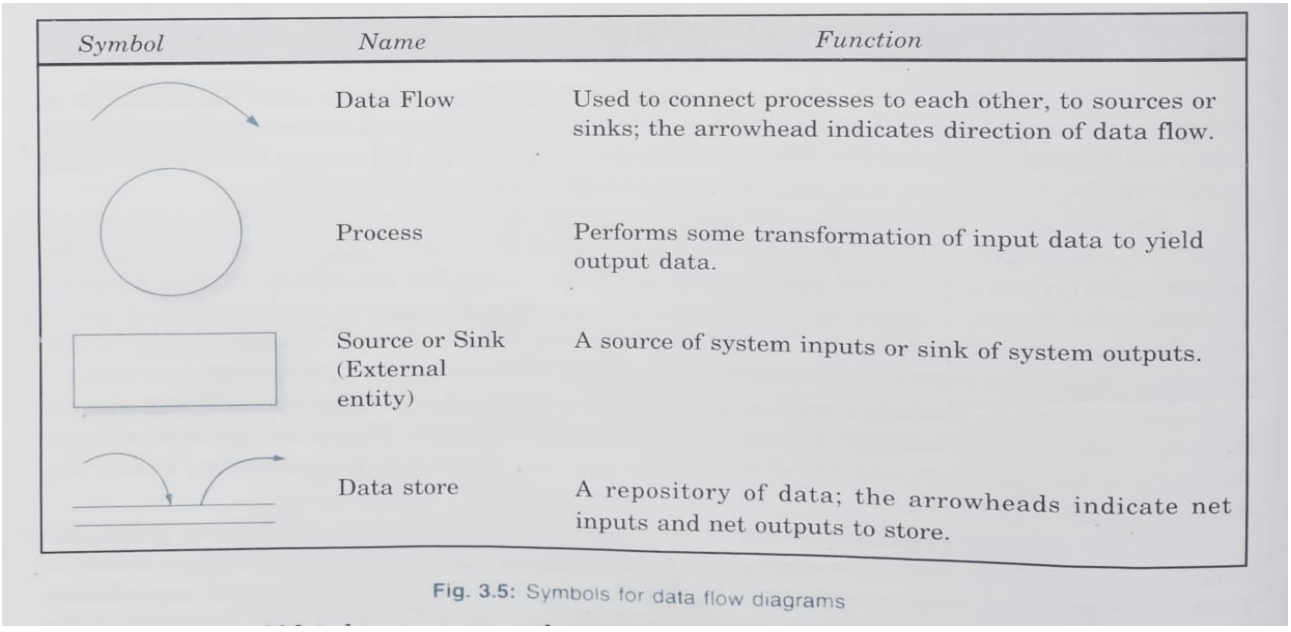
Data Flow Diagrams

Data flow diagrams (DFD) are used widely for modeling the requirements. DFDs show the flow of data through a system. The system may be a company, an organisation, 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 a bubble chart.

The following factors are important for DFDs :

- All names should be unique. This makes it easier to refer to items in the DFD.
- Remember that a DFD is not a flow chart. Arrows in a flow chart represent the order of events; arrows in DFD represent flowing data. A DFD does not imply any order of events.
- Suppress logical decisions.
- Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

Standard symbols for DFDs are given below:



Symbol	*Name*	*Function*
	Data Flow	Used to connect processes to each other, to sources or sinks; the arrowhead indicates direction of data flow.
	Process	Performs some transformation of input data to yield output data.
	Source or Sink (External entity)	A source of system inputs or sink of system outputs.
	Data store	A repository of data; the arrowheads indicate net inputs and net outputs to store.

Fig. 3.5: Symbols for data flow diagrams

Leveling

The DFD may be used to represent a system or software at any level of abstraction. A level-0 DFD, also called a fundamental system model or context diagram represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows. Then the system is decomposed and represented as a DFD with multiple bubbles. The each bubble is again decomposed for more detailed DFDs. This is repeated as many levels as necessary until the problem at hand is well understood. It is important to preserve the number of inputs and outputs between levels; this concept is called leveling by DeMacro. Thus, if bubble "A" has two inputs, x_1, and x_2, 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. 3.6.

Fig. 3.6: Level-0 DFD

Level 0, Level 1, and Level 2 DFDs of result management system are shown below.

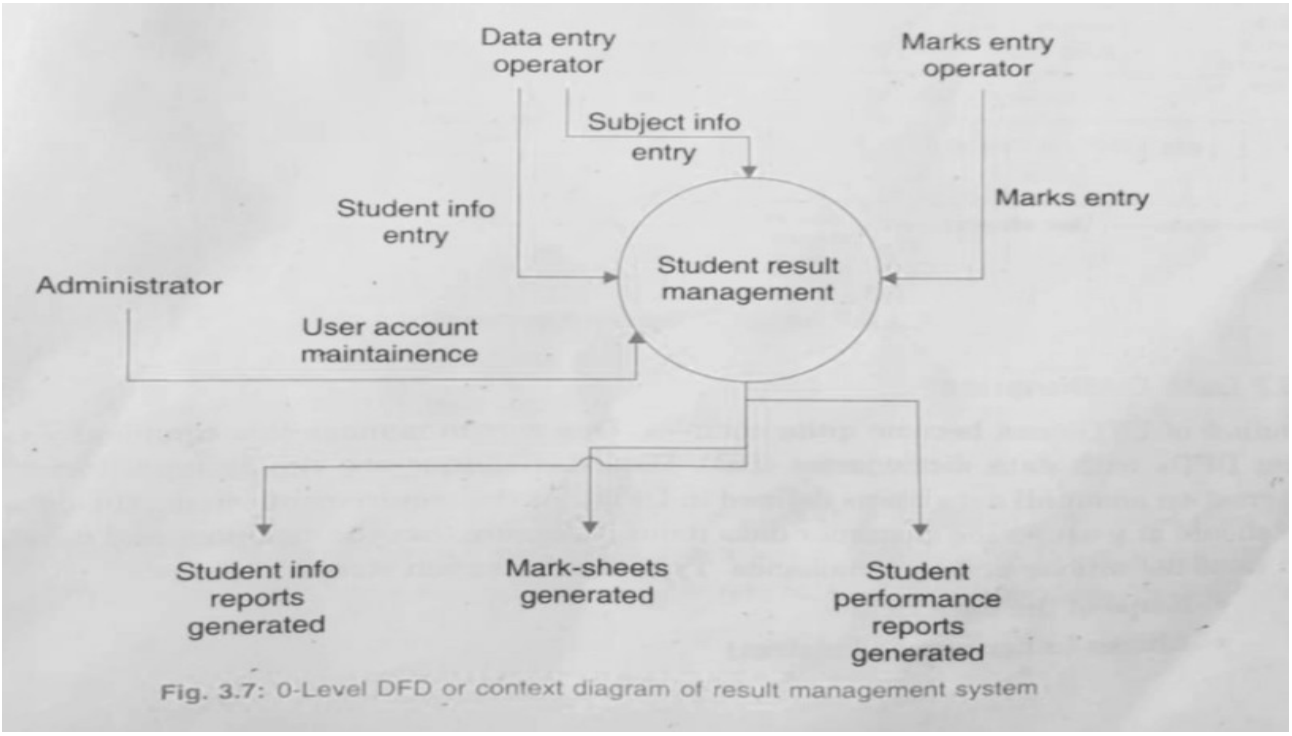
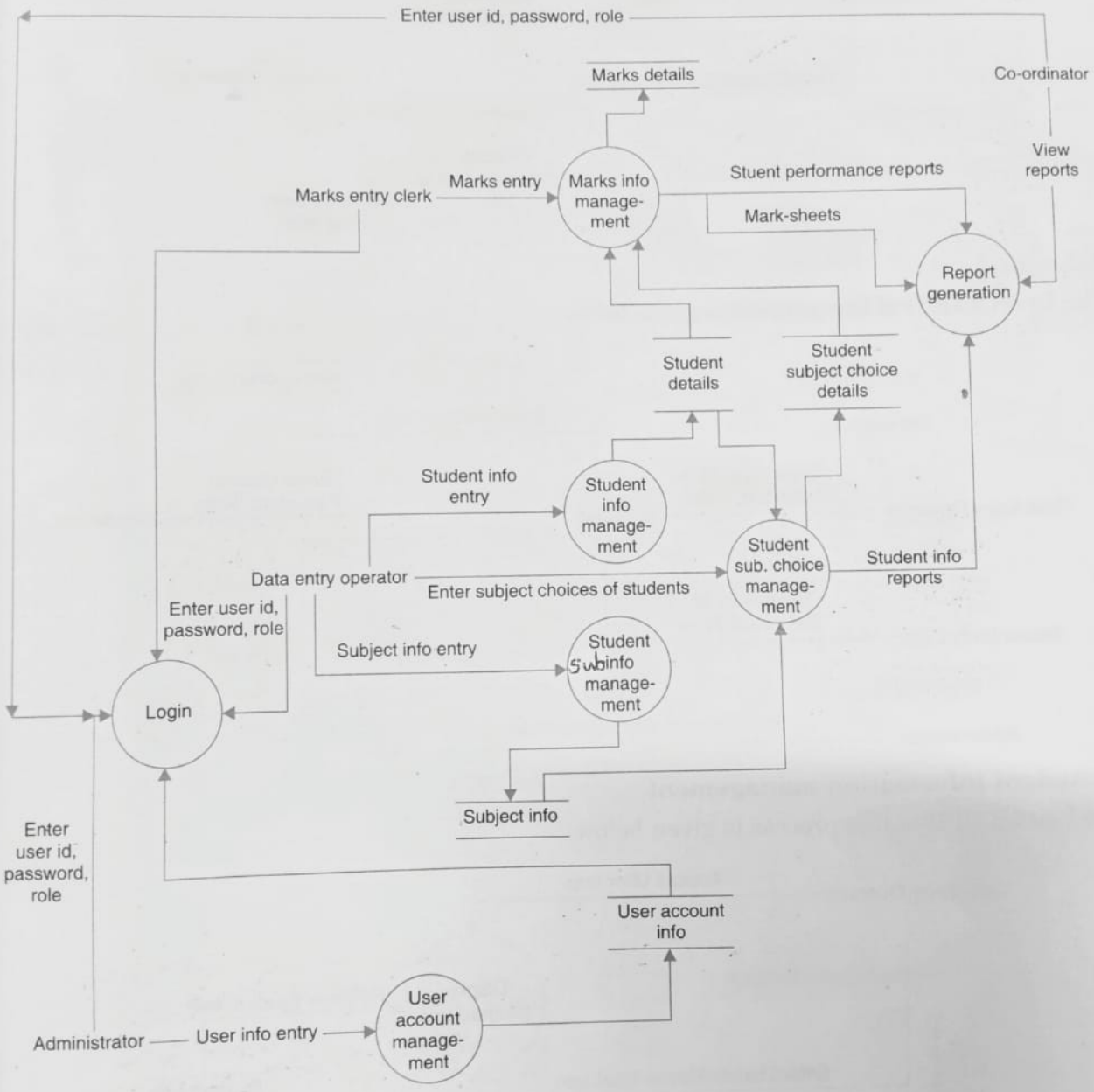


Fig. 3.7: 0-Level DFD or context diagram of result management system

3.9.3 Level-n DFD

Level-1 DFD

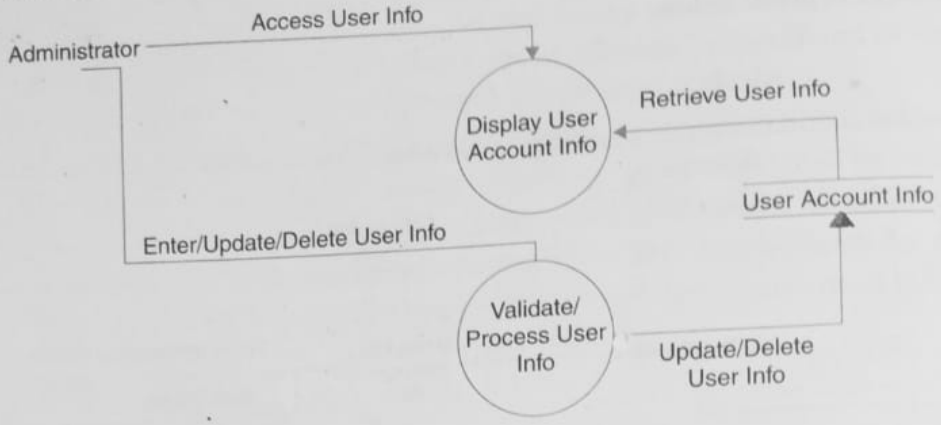
The Level-1 DFD is given below:



Level 1 DFD of result management system

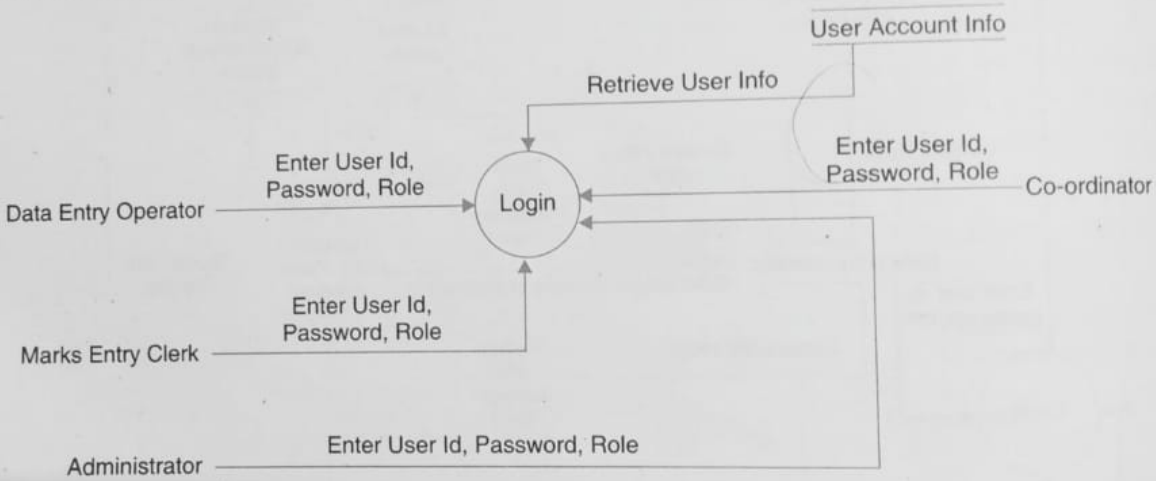
Level-2 DFDs

1. User account maintenance



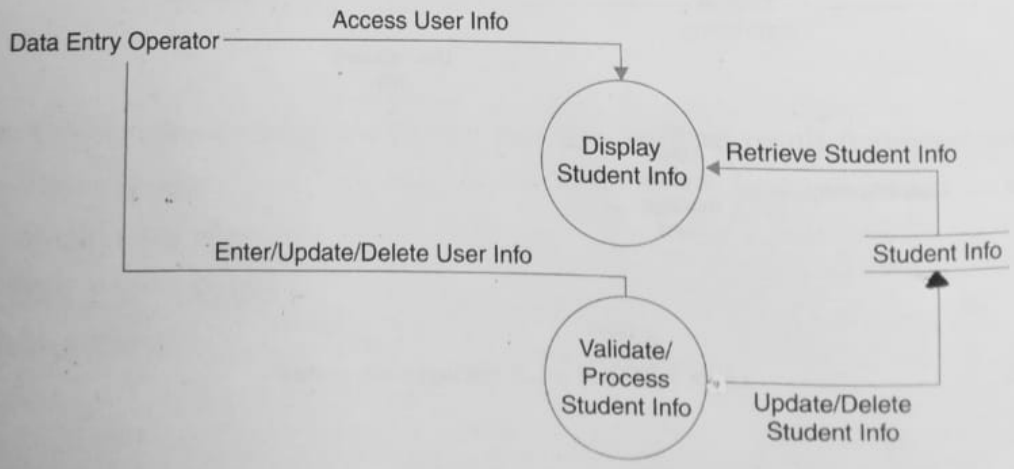
2. Login

The Level 2 DFD of this process is given below:



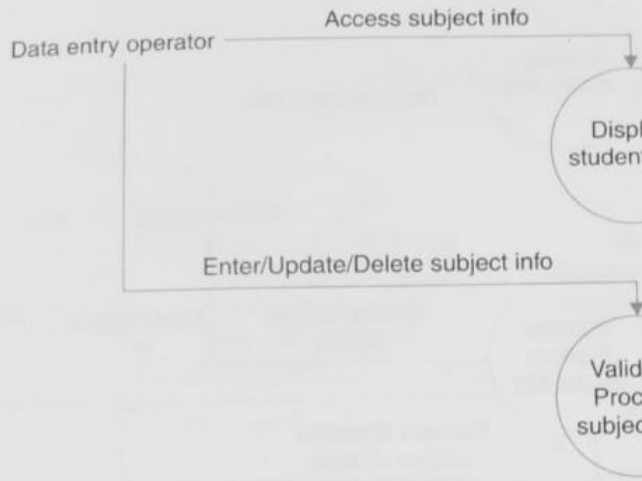
3. Student information management

The Level 2 DFD of this process is given below:



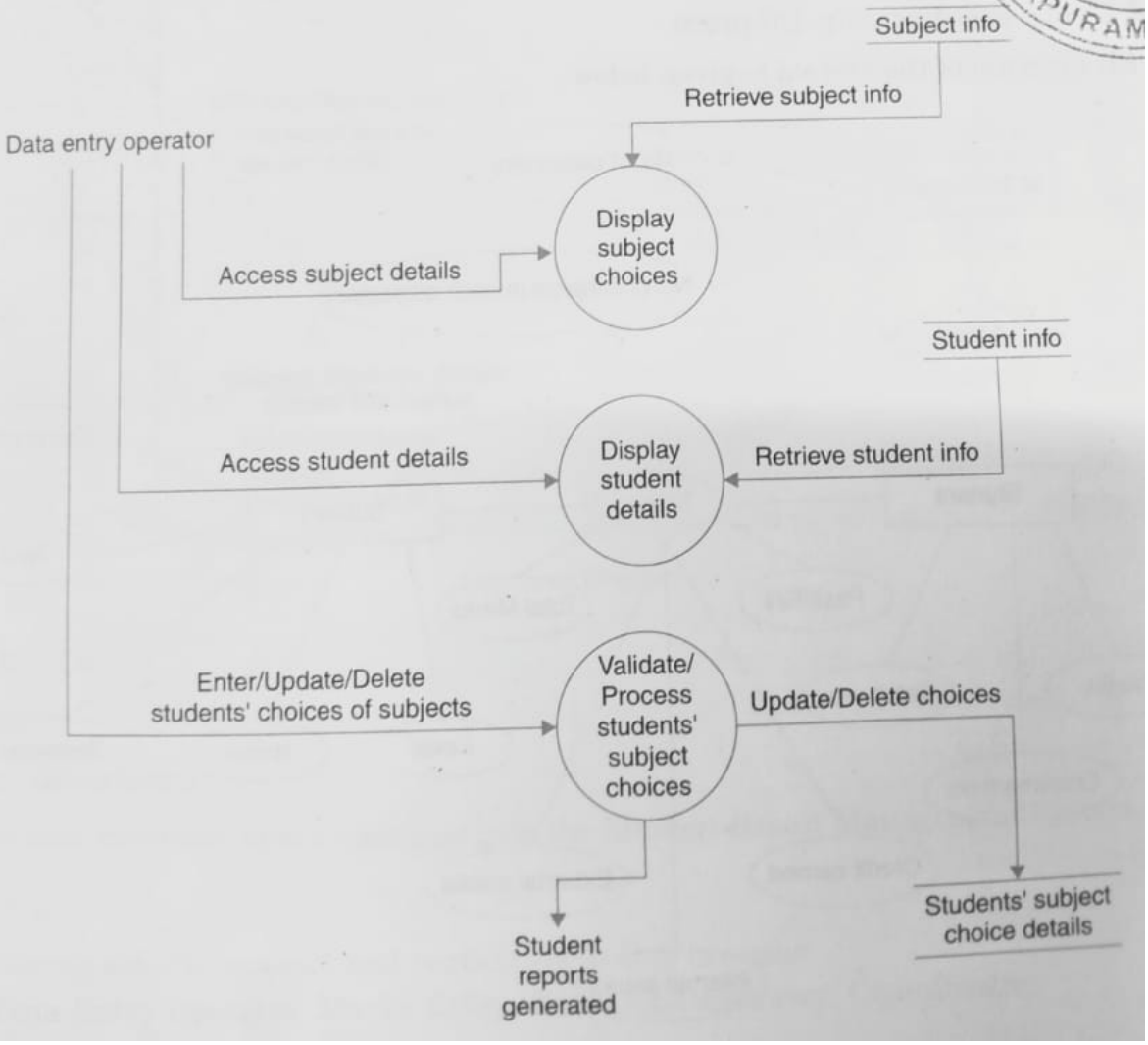
4. Subject information management

The Level 2 DFD of this process is given below:



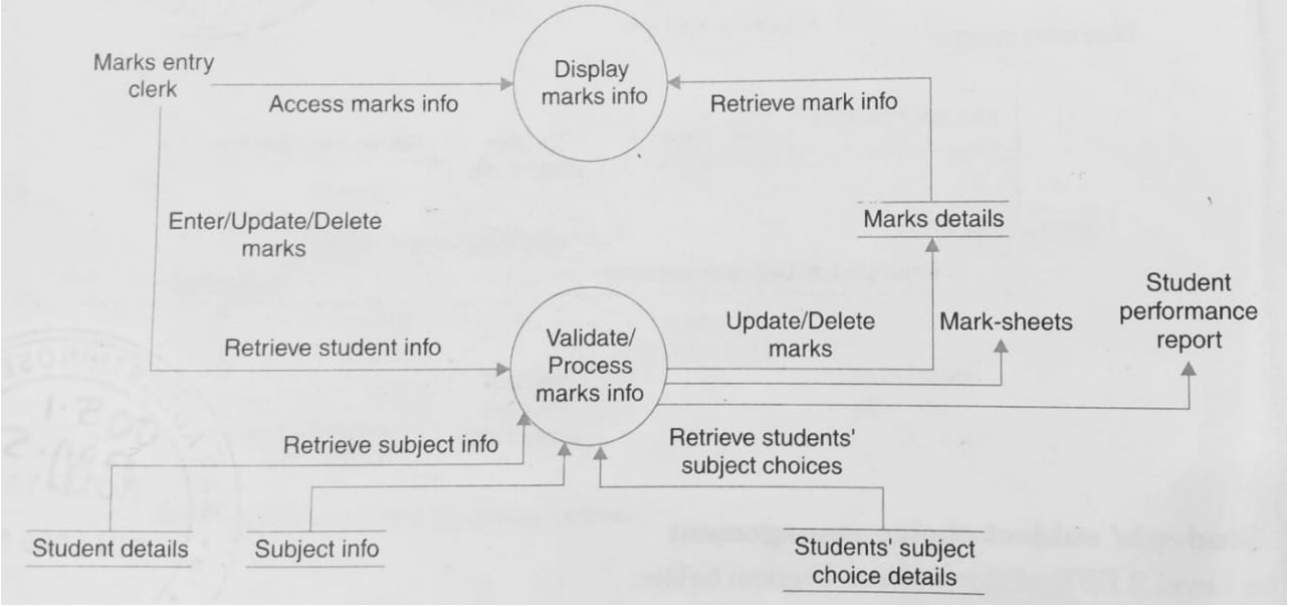
5. Students' subject choice management

The Level 2 DFD of this process is given below:



6. Marks information management

The Level 2 DFD of this process is given below:



Data Dictionaries:-Data Dictionaries are simply repositories to store information about all data items defined in DFD. It is a central repository that defines and describes all of the data structures (i.e., data elements, data flows, data stores) within a system. It contains

- Name of the data item.
- Aliases (other names for items) - DEO for data entry operator
- Description/Purpose -It is a textual description of what the data items is used for or why it exists.
- Related data items - It represent relationship between data items.(Total mark = int + ext)
- Range of values -All possible values(Mark must be between 0 and 100)
- Data structure definition- If the data structure is primitive, it represent physical structure of data item and if the data structure is aggregate, it represent composition of data items.

The mathematical operators used in dictionary are shown below.

Table 3.3: Data dictionary notation and mathematical operators

Notation	*Meaning*
$x = a + b$	x consists of data elements a and b
$x = [a/b]$	x consists of either data element a or b
$x = a$	x consists of an optional data element a
$x = y\{a\}$	x consists of y or more occurrences of data element a
$x = \{a\}z$	x consists of z or fewer occurrences of data element a
$x = y\{a\}z$	x consists of some occurrences of data element a which are between y and z.

The data dictionary can be used to:

- Create an ordered listing of all data items.
- Create an ordered listing of a subset of data items.
- Find a data item name from a description.
- Design the software and test cases.

Entity-Relationship Diagrams

It is a detailed logical representation of data for an organization and uses three main constructs- Entities, Relationships and their attributes.

Entities

Fundamental thing about which data may be maintained. Each entity has its own identity. The entity type is a collection of the entity having similar attributes.

We use capital letters in naming an entity type and in an ER diagram the name is placed inside a rectangle representing that entity as shown below.

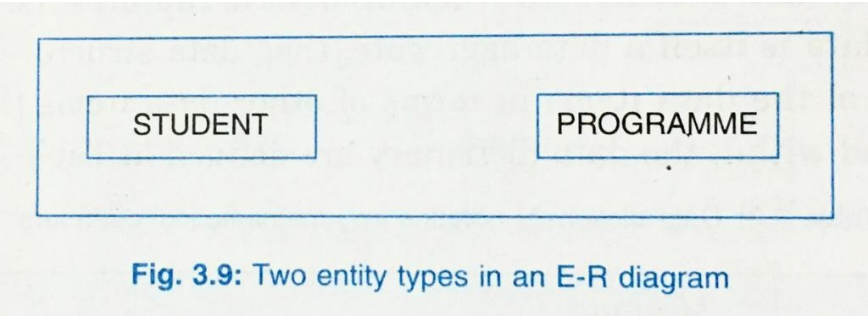


Fig. 3.9: Two entity types in an E-R diagram

Relationships : A relationship is a reason for associating two entity types. Binary relationships involve two entity types. A STUDENT is registered for a PROGRAMME. Relationships are represented by diamond notation in a ER diagram.

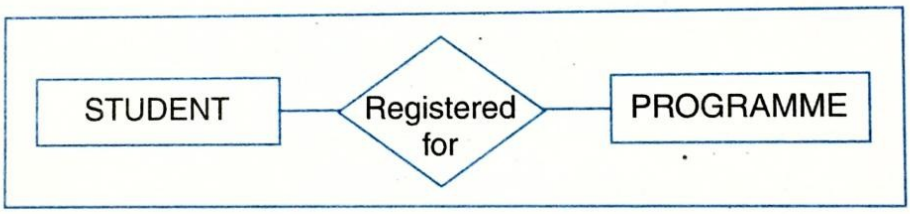


Fig. 3.10: Relationships added to ERD

Degree of relationship - It is the number of entity types that participates in that relationship. Three most common are unary, binary and ternary. Unary relationship(recursive relationship) :- It is a relationship between the instances of one entity type. Examples below.

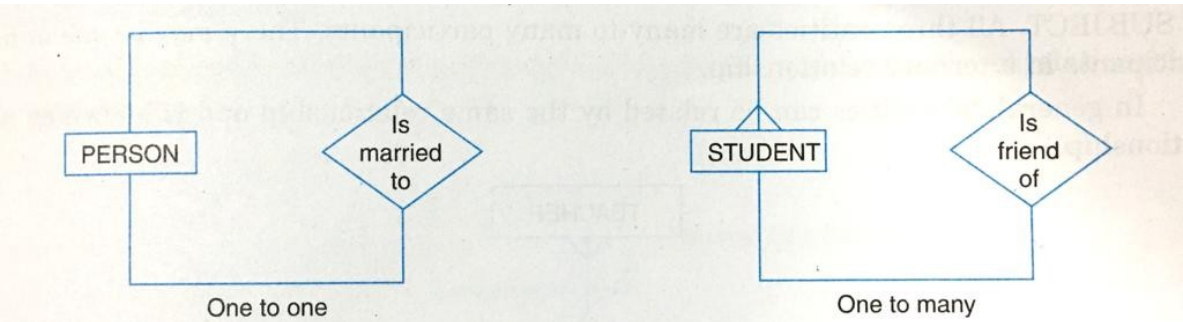


Fig. 3.12: Unary relationships

Binary relationship :- It is a relationship between instances of two entity types. Examples

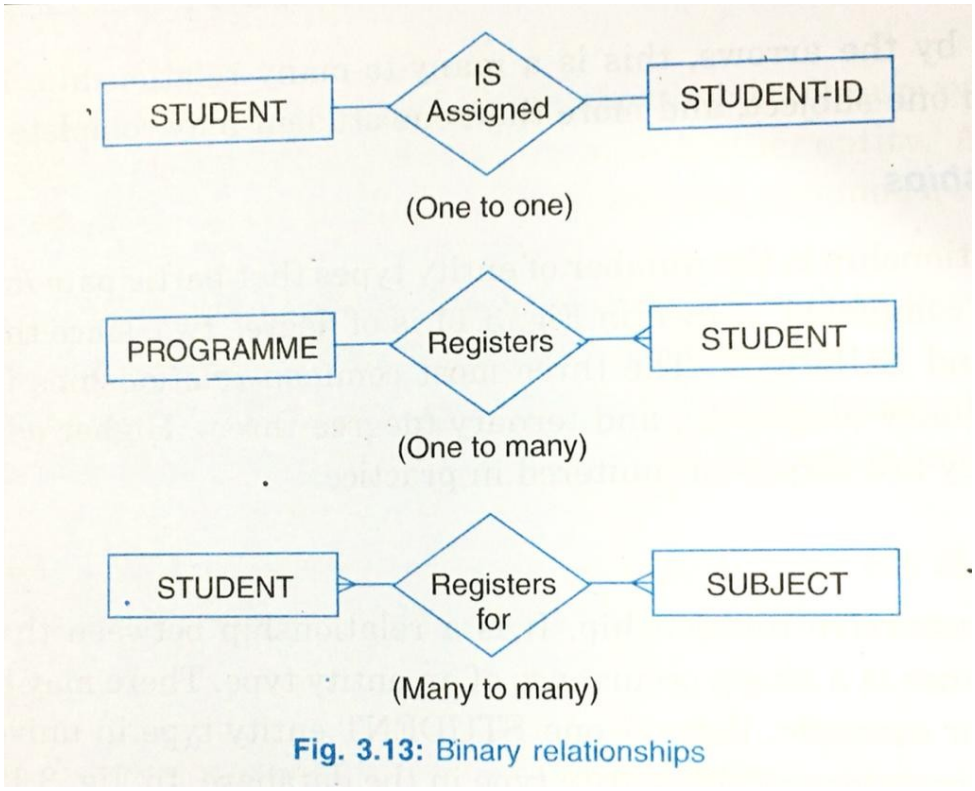


Fig. 3.13: Binary relationships

Ternary relationship :- It is a simultaneous relationship among the instances of three entity types. Examples

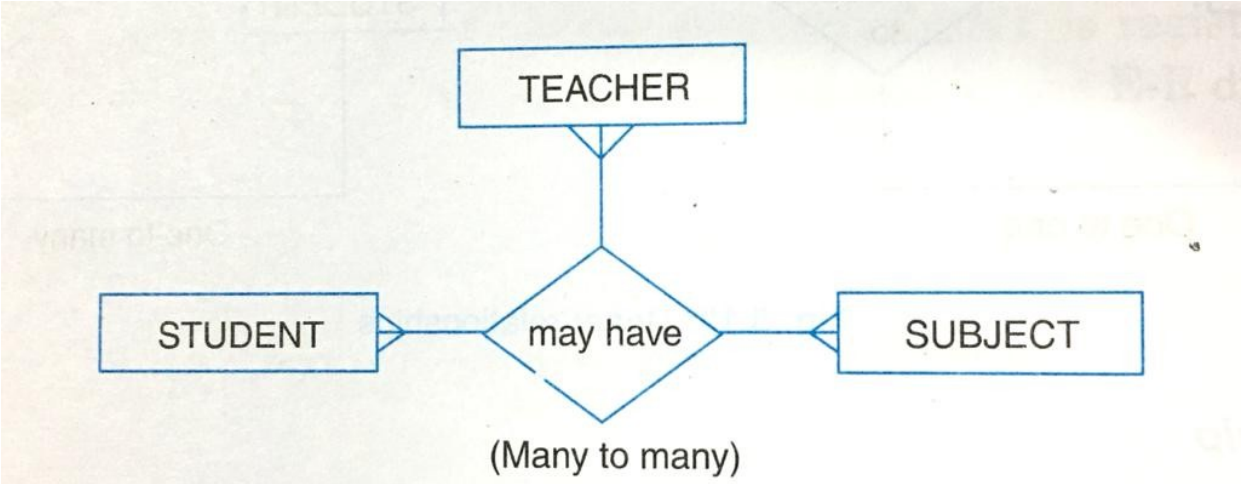


Fig. 3.14: Ternary relationship

Cardinalities and optionality :- Two entity types A and B, connected by a relationship. The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A.

Minimum cardinality is the minimum number of instances of entity B that may be associated with each instance of entity A. In the eg below, if the minimum number of students registered for a subject is zero, then subject is an optional participant in the relationship and if minimum number is 1 then subject is an mandatory participant in the relationship.

Maximum cardinality is the maximum number of instances. The zero through the line near the SUBJECT entity means a minimum cardinality of zero, while the crow's foot notation means a many maximum cardinality.

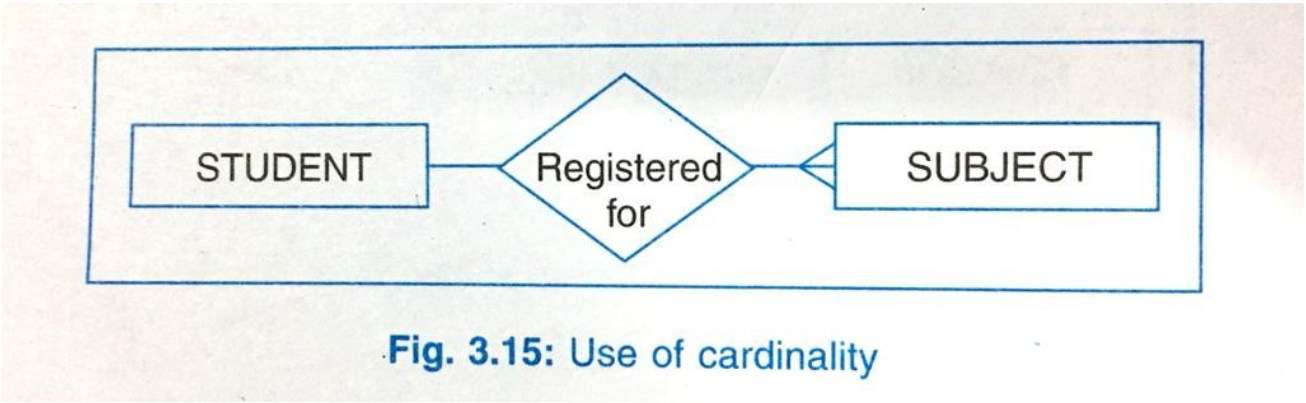


Fig. 3.15: Use of cardinality

The cardinality of relationship is given below.

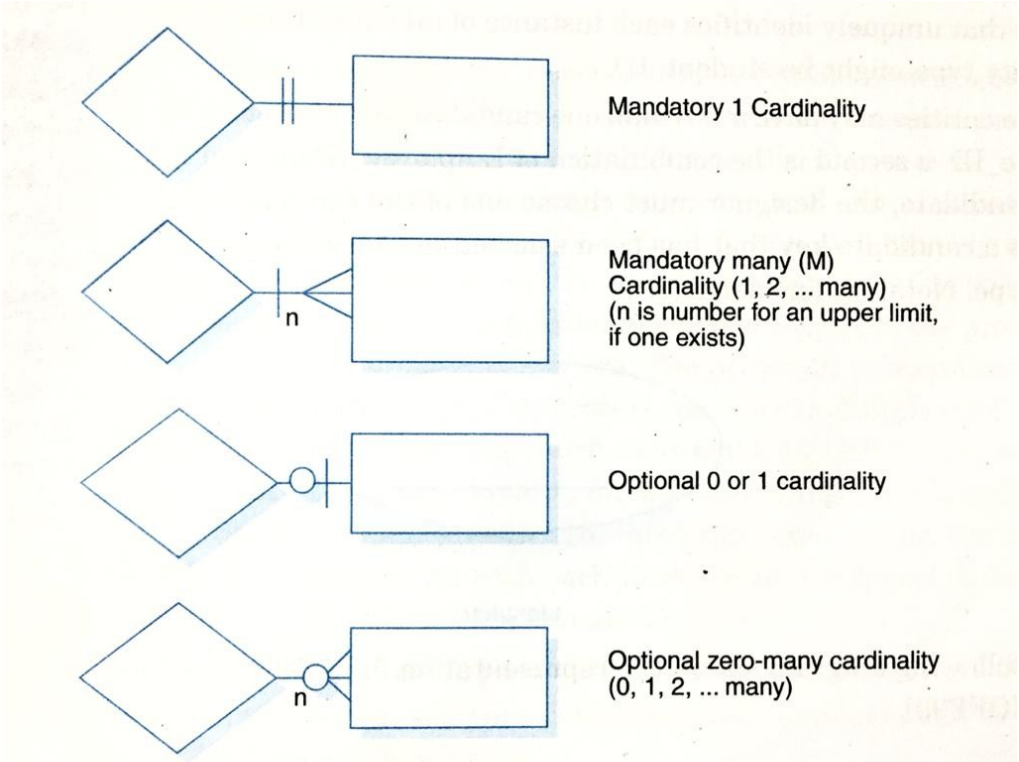
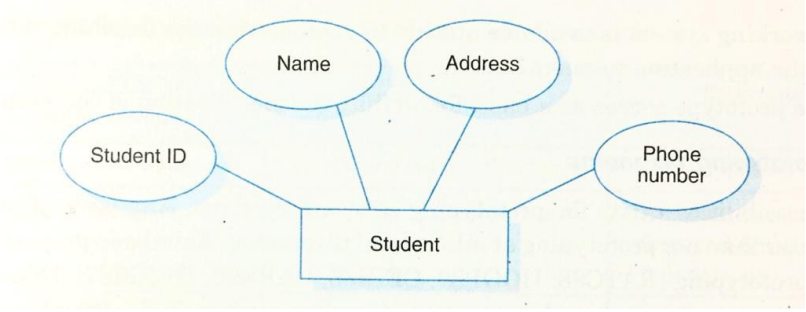


Fig. 3.17: Relationship cardinality [HOFF99]

Attributes :- Each entity type has a set of attributes associated with it. An attribute is a property or characteristic of an entity that is of interest to organization. A candidate key is an attribute or combination of attributes that uniquely identifies each instance of an entity type. If there are more candidate keys, one of the key may be chosen as the Identifier. It is used as unique characteristic for an entity type. Notations for attribute and identifier is given below. An example also.



REQUIREMENTS DOCUMENTATION

Requirements documentation is very important activity after the requirements elicitation and analysis. This is the way to represent requirements in a consistent format. Requirements document is called Software Requirements Specification (SRS).

Nature of the SRS

The basic issues that SRS writer (s) shall address are the following:

1. Functionality: What the software is supposed to do?

2. External interfaces: How does the software interact with people, the system's hardware, other hardware, and other software?

3. Performance: What is the speed, availability, response time, recovery time, etc. of various software functions?

4. Attributes: What are the considerations for portability, correctness, maintainability, security, reliability etc.?

5. Design constraints imposed on an implementation: Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment (s) etc.?

SRS should

1. Correctly define all requirements.

2. Not describe any design details.

3. Not impose any additional constraints.

Characteristics of a good SRS

The SRS should be:

► Correct ► Unambiguous ► Complete ► Consistent ► Ranked for importance and/or stability ► Verifiable ► Modifiable ► Traceable

Correct: The SRS is correct if, and only if; every requirement stated therein is one that the software shall meet. There is no tool or procedure that assures correctness. The customer can determine if the SRS satisfies the actual needs.

Unambiguous: The SRS is unambiguous if, and only if every requirement stated therein has only one interpretation. Each sentence in the SRS should have the unique interpretation. The SRS should be unambiguous both to those who create it and to those who use it.

Complete: The SRS is complete if, and only if; it includes the following elements:

1. All the requirements, related to functionality, performance, design constraints, attributes or external interfaces.

2. Responses of the software to all classes of input data. It is important to specify the responses to both valid and invalid values.

3. Full labels and references to all figures, tables, and diagrams in the SRS.

Consistent: The SRS is consistent if, and only if, no subset of individual requirements described in it conflict. There are three types of conflicts in the SRS:

1. The specified characteristics of real-world objects may conflict. For example.

(a) The format of an output report may be described in one requirement as tabular but in another as textual.

(b) One requirement may state that all lights shall be green while another states that all lights shall be blue.

2. There may be logical or temporal conflict between two specified actions, for example,

(a) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.

(b) One requirement may state that "A" must always follow "B", while another requires that “A and B" occur simultaneously.

3. Two or more requirements may describe the same real world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue” in another. The use of standard terminology and definitions promotes consistency.

Ranked for importance and/or stability: Every requirement has its own importance. Some are urgent and could be on the critical path and must be fulfilled prior to other, some could be delayed. It is better to rank every requirement according to its importance and stability.

The SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Verifiable: The SRS is verifiable, if and only if, every requirement stated therein is verifiable. A requirement is verifiable, if and only if there exists some finite cost-effective process with which a person of machine can check that the software meets the requirements.

Modifiable: The SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.

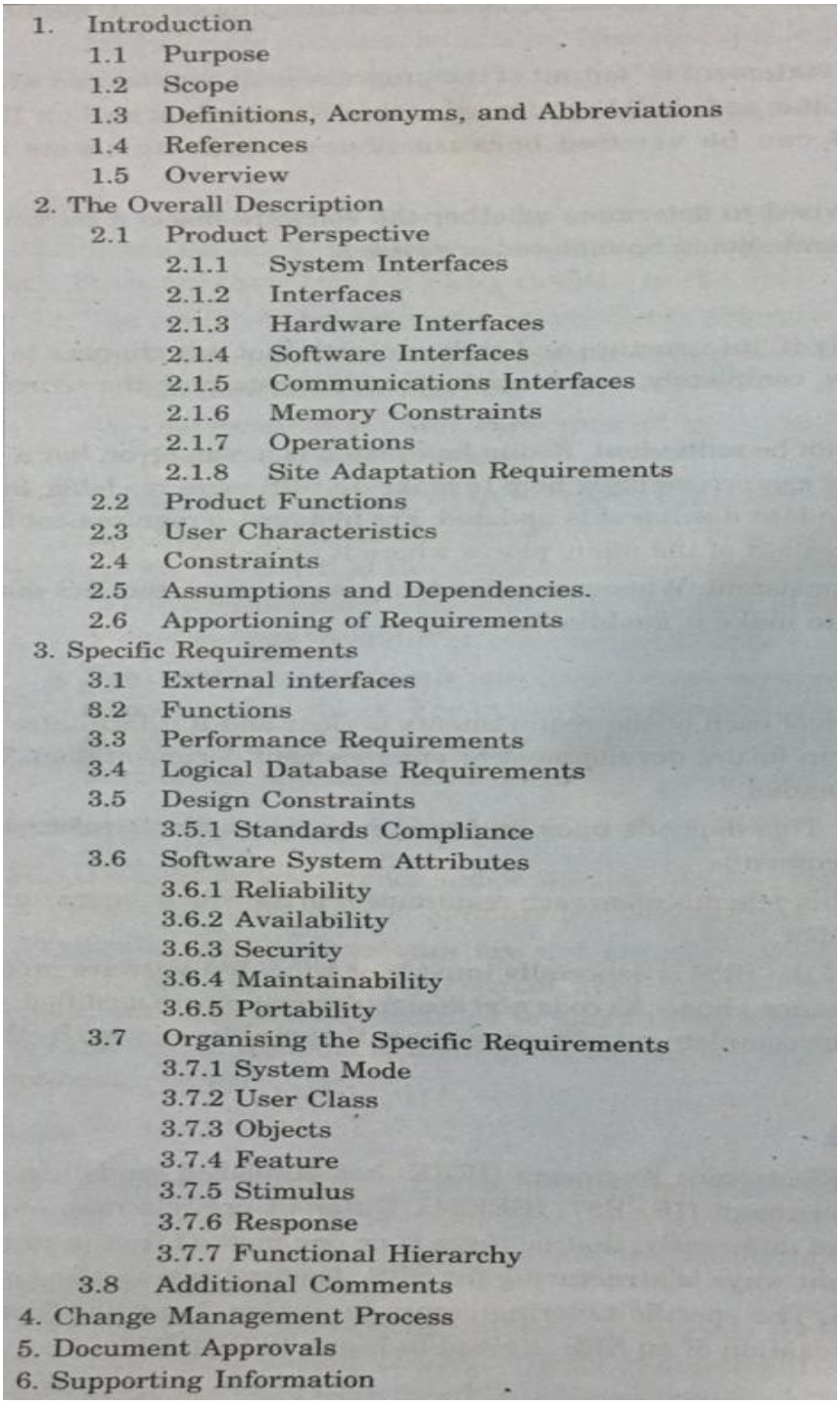
Traceable: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended.

1. Backward traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

2. Forward traceability: This depends upon each requirement in the SRS having a unique name or reference number.

Organization of the SRS

IEEE has published guidelines and standards to organize an SRS. There are different ways of organizing an SRS document. First two sections are same in all. The specific tailoring occurs in section-3.



1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. The Overall Description
   - 2.1 Product Perspective
     - 2.1.1 System Interfaces
     - 2.1.2 Interfaces
     - 2.1.3 Hardware Interfaces
     - 2.1.4 Software Interfaces
     - 2.1.5 Communications Interfaces
     - 2.1.6 Memory Constraints
     - 2.1.7 Operations
     - 2.1.8 Site Adaptation Requirements
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and Dependencies.
   - 2.6 Apportioning of Requirements
3. Specific Requirements
   - 3.1 External interfaces
   - 8.2 Functions
   - 3.3 Performance Requirements
   - 3.4 Logical Database Requirements
   - 3.5 Design Constraints
     - 3.5.1 Standards Compliance
   - 3.6 Software System Attributes
     - 3.6.1 Reliability
     - 3.6.2 Availability
     - 3.6.3 Security
     - 3.6.4 Maintainability
     - 3.6.5 Portability
   - 3.7 Organising the Specific Requirements
     - 3.7.1 System Mode
     - 3.7.2 User Class
     - 3.7.3 Objects
     - 3.7.4 Feature
     - 3.7.5 Stimulus
     - 3.7.6 Response
     - 3.7.7 Functional Hierarchy
   - 3.8 Additional Comments
4. Change Management Process
5. Document Approvals
6. Supporting Information

Organisation of SRS [IEEE-std. 830-1993]

SRS document comprises the following sections.

1. Introduction: This provides an overview of the entire information described in SRS. This involves purpose and the scope of SRS, which states the functions to be performed by the system. In addition, it describes definitions, abbreviations, and the acronyms used. The references used in SRS provide a list of documents that is referenced in the document.

2. Overall description: It determines the factors which affect the requirements of the system. It provides a brief description of the requirements to be defined in the next section called 'specific requirement'. It comprises the following sub-sections.

2.1 Product perspective: It determines whether the product is an independent product or an integral part of the larger product. It determines the interface with hardware, software, system, and communication. It also defines memory constraints and operations utilized by the user.

2.2 Product functions: It provides a summary of the functions to be performed by the software. The functions are organized in a list so that they are easily understandable by the user:

2.3 User characteristics: It determines general characteristics of the users.

2.4 Constraints: It provides the genera1 description of the constraints such as regulatory policies, audit functions, reliability requirements, and so on.

2.5 Assumption and dependency: List each of the factors that affect the requirements stated in the SRS.

2.6 Apportioning of requirements: It determines the requirements that can be delayed until release of future versions of the system.

3. Specific requirements: This section contains the software requirements to a level of detail sufficient to enable designers to design the system, and testers to test that system.

3.1 External interface: This contains a detailed description of all inputs into and outputs from the software system.

3.2 Functions: Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. This includes validity checks on inputs, exact sequence of operations, responses to abnormal situation, and so on.

3.3 Performance requirements: It determines the performance constraints of the software system. Performance requirement is of two types: static requirements and dynamic requirements. Static requirements (also known as capacity requirements) do not impose constraints on the execution characteristics of the system. These include requirements like number of terminals and users to be supported. Dynamic requirements determine the constraints on the execution of the behaviour of the system, which includes response time (the time between the start and ending of an operation under specified conditions) and throughput (total amount of work done in a given time).

3.4 Logical database of requirements: It determines logical requirements to be stored in the database. This includes type of information used, frequency of usage, data entities and relationships among them, and so on.

3.5 Design constraints: It determines all design constraints that are imposed by standards, hardware limitations, and so on.

3.6 Software system attributes: It provide attributes such as reliability, availability, maintainability and portability. It is essential to describe all these attributes to verify that they are achieved in the final system.

3.7 Organizing Specific Requirements: It determines the requirements so that they can be properly organized for optimal understanding. The requirements can be organized on the basis of mode of operation, user classes, objects, feature, response, and functional hierarchy.

4. Change management process: It determines the change management process in order to identify, evaluate, and update SRS to reflect changes in the project scope and requirements.

5. Document approvals: These provide information about the approvers of the SRS document with the details such as approve 's name, signature, date, and so on.

6. Supporting information: It provides information such as table of contents, index, and so on. This is necessary especially when SRS is prepared for large and complex project

Requirements Validation

The objective of requirement validation is to certify that the SRS document is an acceptable document of the system. The SRS document is mainly checked for:

►Completeness & consistency

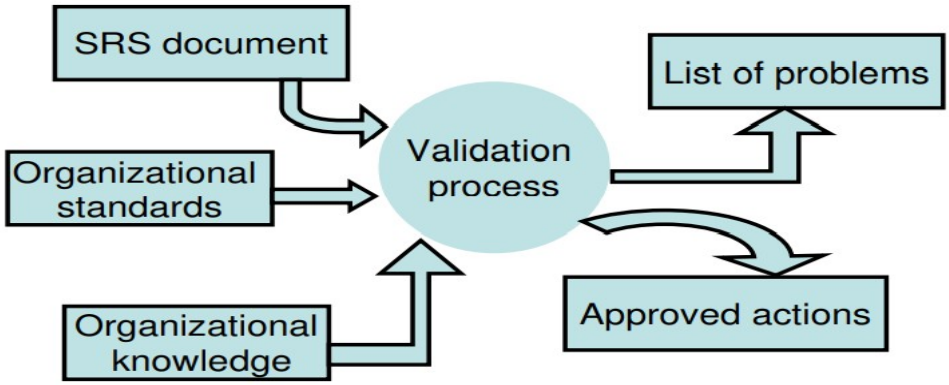
►Conformance to standards

►Requirements conflicts

►Technical errors

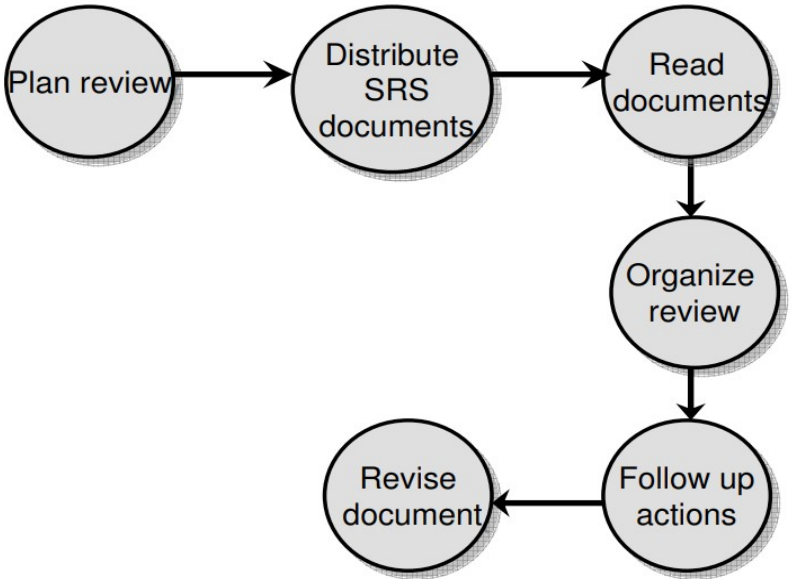
►Ambiguous requirements

The input and output of validation process are shown below.



Validation process with inputs and outputs

Requirement reviews(one type of technical reviews) : This is a common requirement validation technique where a group read the SRS document and check for possible errors. The process is shown below.



Requirements review process

1) The review team is selected and time and place for review meeting is fixed.
2) The SRS is distributed to all members.
3) Everyone read the document.
4) Each member presents his/her views and identified problems. The problems are discussed and a set of actions for problem solving are approved.
5) The chairperson checks that the approved actions have been carried out.
6) Document is revised to reflect the approved actions.

Problem actions

- Requirements clarification: Find omitted or badly expressed requirements
- Missing information: Find this information from stakeholders
- Requirements conflicts: Stakeholders must negotiate to resolve this conflict
- Unrealistic requirements: Stakeholders must be consulted
- Security issue: Review the system in accordance to security standards

Prototyping: Validation prototype should be reasonably complete and should be used as the required system.

Software Project Planning

After the finalization of SRS (sometimes even prior to finalization), customers would like to estimate size, cost and development time of the project which are the key issues during planning.

Software managers are responsible for planning. They supervise the work to ensure that it follows project plan. In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired. Activities during software project planning are illustrated below.

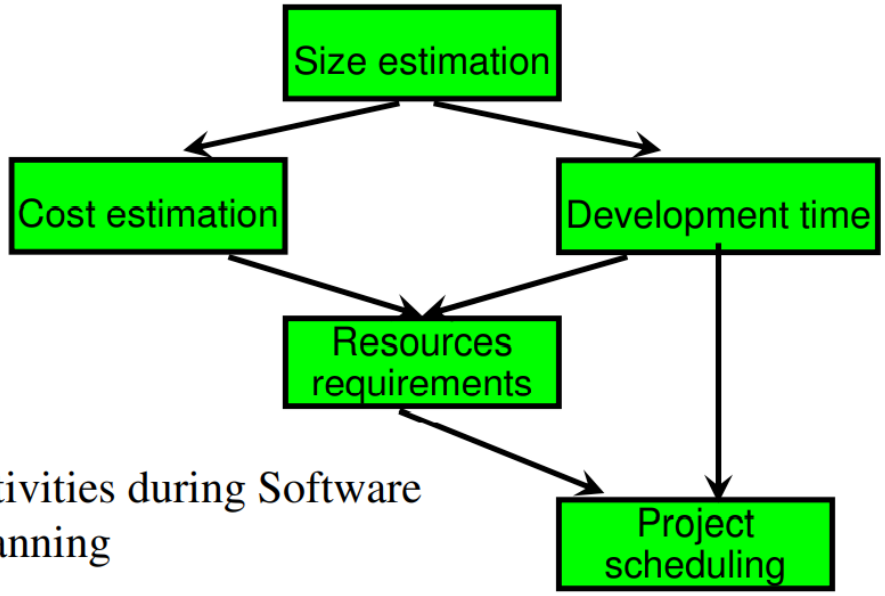


Fig. 1: Activities during Software Project Planning

Size Estimation

Estimation of the size of software is an essential part of Software Project Management. It helps the project manager to further predict the effort and time which will be needed to build the project. Various measures are used in project size estimation. Most used are: Lines of Code (LOC) and Function points.

Lines of Code (LOC) :- As the name suggest, LOC count the total number of lines of source code in a project. "A line of code is any line of program text that is not a comment or blank line,

regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declarations, and executable and non-executable statements."

It is a simple metric and can be calculated easily. But it has many drawbacks

- Programming language dependent. Same program in two languages vary in LOC.
- No common agreement in what constitute a line. Some include comments, blank lines and non executable statements but some others not. Including comments may encourage adding more comments to create the illusion of high productivity.
- No proper industry standard exist for this technique.
- It is difficult to estimate the size using this technique in early stages of project.

For eg, in the program below, If LOC is simply a count of the number of lines then it contains 18 LOC. If comments ignored 17 LOC. If only executable statements are counted then 13LOC.

```
1.  int. sort (int x[ ], int n)
2.  {
3.  int i, j, save, im1;
4.  /*This function sorts array x in ascending order */
5.  If (n<2) return 1;
6.  for (i=2; i<=n; i++)
7.  {
8.  im1=i-1;
9.  for (j=1; j<=im; j++)
10. if (x[i] < x[j])
11. {
12. Save = x[i];
13. x[i] = x[j];
14. x[j] = save;
15. }
16. }
17. return 0;
18. }
```

Function Count:- Alan Albrecht in 1970s developed a technique called Function Point Analysis(FPA) for size estimation. Function point measures functionality from the user's point of view, that is, on the basis of what the user requests and receives in return from the system.

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

- Inputs : information entering the system
- Outputs : information leaving the system
- Enquiries : requests for instant access to information
- Internal logical files : information held within the system
- External interface files : information held by other system that is used by the system being analyzed.

The FPA functional units are shown in figure given below:

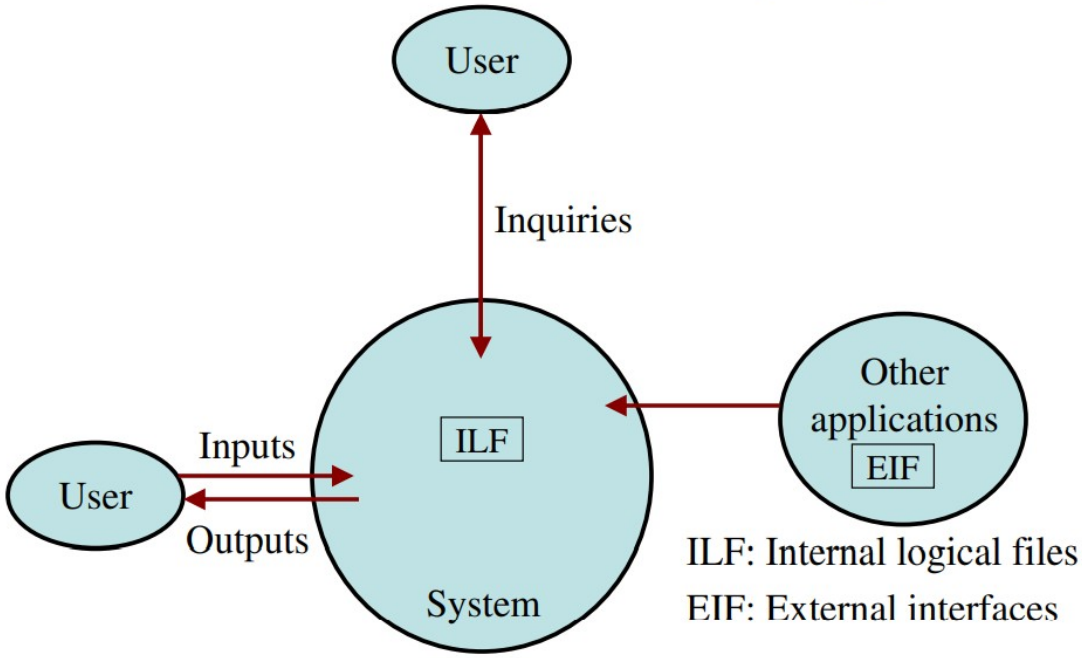


Fig. 3: FPAs functional units System

The five functional units are divided in two categories:

(i) Data function types

- Internal Logical Files (ILF): A user identifiable group of logical related data or control information maintained within the system.

- External Interface files (EIF): A user identifiable group of logically related data or control information referenced by the system, but maintained within another system

(ii) Transactional function types

- External Input (EI): An El processes data or control information that comes from outside the system.
- External Output (EO): An EO is an elementary process that generate data or control information to be sent outside the system.
- External Inquiry (EQ): An EQ is an elementary process that is made up to an input-output combination that results in data retrieval.

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; i.e. they do not take into consideration programming languages, database management systems, processing hardware or any other database technology.
- Function points can be estimated from requirement specification or design specification, thus making it possible to estimate development efforts in early phases of development.
- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

  Counting function points

  The five functional units are ranked according to their complexity ie low, average or high using a set of prescriptive standards. After classifying each of the five function types, the Unadjusted Function Points(UFP) are calculated using predefined weights for each function types as given below.

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 1 : Functional units with weighting factors

The procedure for the calculation of UFP is given below.

Table 2: UFP calculation table

Functional Units	Count Complexity			Complexity Totals	Functional Unit Totals
External Inputs (EIs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External Outputs (EOs)		Low x 4	=		
		Average x 5	=		
		High x 7	=		
External Inquiries (EQs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External logical Files (ILFs)		Low x 7	=		
		Average x 10	=		
		High x 15	=		
External Interface Files (EIFs)		Low x 5	=		
		Average x 7	=		
		High x 10	=		
Total Unadjusted Function Point Count					

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

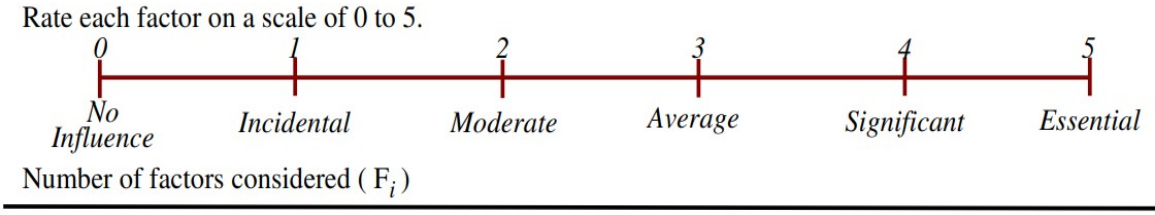
W_{ij} : It is the entry of the i^{th} row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

The final FP is calculated using the formula:

$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to [0.65 + 0.01 x ΣF_i]. The F_i (*i*=1 to 14) are the degree of influence and are based on responses to questions noted in table 3.



1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Functions points may compute the following important metrics:

Productivity= FP / persons-months

Quality= Defects / FP

Cost= Rupees / FP

Documentation= Pages of documentation per FP

Example: 4.1

Consider a project with the following functional units:

Number of user inputs	= 50
Number of user outputs	= 40
Number of user enquiries	= 35
Number of user files	= 06
Number of external interfaces	= 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

Solution

We know

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

UFP = 50 x 4 + 40 x 5 + 35 x 4 + 6 x 10 + 4 x 7
= 200 + 200 + 140 + 60 + 28 = 628

CAF = $(0.65 + 0.01\ \Sigma F_i)$
= (0.65 + 0.01 (14 x 3)) = 0.65 + 0.42 = 1.07

FP = UFP x CAF
= 628 x 1.07 = 672

Example:4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

= 10 x 3 + 12 x 7 + 20 x 7 + 15 + 10 + 12 x 4
= 30 + 84 +140 + 150 + 48
= 452

FP = UFP x CAF
= 452 x 1.10 = 497.2.

Example: 4.3

Consider a project with the following parameters.

(*i*) External Inputs:
- (a) 10 with low complexity
- (b) 15 with average complexity
- (c) 17 with high complexity

(*ii*) External Outputs:
- (a) 6 with low complexity
- (b) 13 with high complexity

(*iii*) External Inquiries:
- (a) 3 with low complexity
- (b) 4 with average complexity
- (c) 2 high complexity

(*iv*) Internal logical files:
- (a) 2 with average complexity
- (b) 1 with high complexity

(v) External Interface files:
- (a) 9 with low complexity

In addition to above, system requires

i. Significant data communication
ii. Performance is very critical
iii. Designed code may be moderately reusable
iv. System is not designed for multiple installation in different organizations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Solution: Unadjusted function points may be counted using table 2

Functional Units	Count	Complexity		Complexity Totals	Functional Unit Totals
External Inputs (EIs)	10	Low x 3	=	30	
	15	Average x 4	=	60	
	17	High x 6	=	102	192
External Outputs (EOs)	6	Low x 4	=	24	
	0	Average x 5	=	0	
	13	High x 7	=	91	115
External Inquiries (EQs)	3	Low x 3	=	9	
	4	Average x 4	=	16	
	2	High x 6	=	12	37
External logical Files (ILFs)	0	Low x 7	=	0	
	2	Average x 10	=	20	
	1	High x 15	=	15	35
External Interface Files (EIFs)	9	Low x 5	=	45	
	0	Average x 7	=	0	
	0	High x 10	=	0	45
Total Unadjusted Function Point Count					424

$$\sum_{i=1}^{14} F_i = 3+4+3+5+3+3+3+3+3+3+2+3+0+3=41$$

$$\begin{aligned} CAF &= (0.65 + 0.01 \times \Sigma F_i) \\ &= (0.65 + 0.01 \times 41) \\ &= 1.06 \\ FP &= UFP \times CAF \\ &= 424 \times 1.06 \\ &= 449.44 \end{aligned}$$

Hence $\boxed{FP = 449}$

Cost Estimation

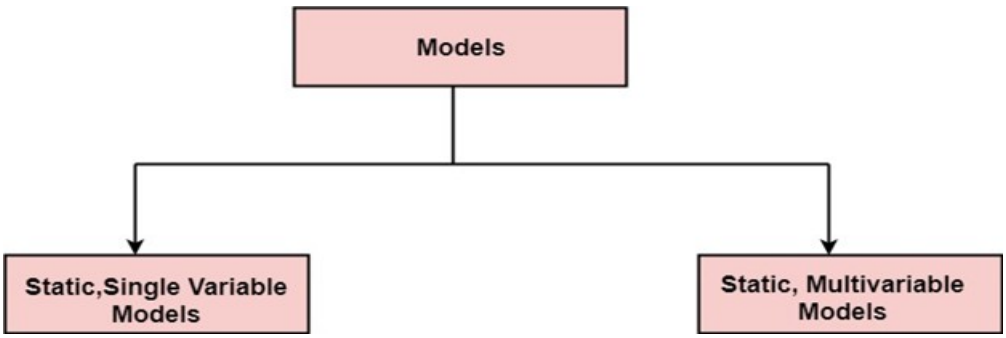
For any new software project, it is necessary to know how much it will cost to develop and how much development time will it take. These estimates are needed before development is initiated. Several estimation techniques have been developed and are having the following attributes in common:

- Project scope must be established in advanced.
- Software metrics are used as a support from which evaluation is made.
- The project is broken into small pieces which are estimated individually.

To achieve true cost & schedule estimate, several option arise.

- Delay estimation until late in project.
- Used simple decomposition techniques to generate project cost and schedule estimates.
- Acquire one or more automated estimation tools.

Cost Estimation Models



Static, Single Variable Models: When a model makes use of single variables to calculate desired values such as cost, time, efforts, etc. is said to be a single variable model. The most common equation is:

$$C=a\ L^b$$

Where C = Costs

L= size

a and b are constants

The Software Engineering Laboratory established a model called SEL model, for estimating its software production. This model is an example of the static, single variable model.

$$E=1.4\ L^{0.93}$$

$$DOC=30.4\ L^{0.90}$$

$D=4.6\ L^{0.26}$

Where E= Efforts (Person Per Month)

DOC=Documentation (Number of Pages)

D = Duration (D, in months)

L = Number of Lines per code

Static, Multivariable Models: These models are based on equation $C = a\ L^b$, they depend on several variables describing various aspects of the software development environment, for example, methods used, user participation, customer oriented changes, memory constraints, etc.

WALSTON and FELIX develop the models at IBM provide the following equation gives a relationship between lines of source code and effort:

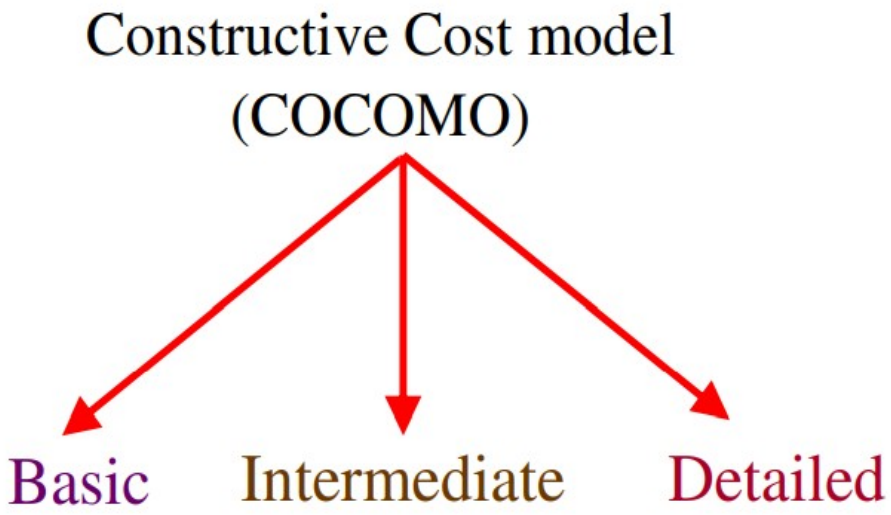
$E=5.2\ L^{0.91}$

In the same manner duration of development is given by

$D=4.1\ L^{0.36}$

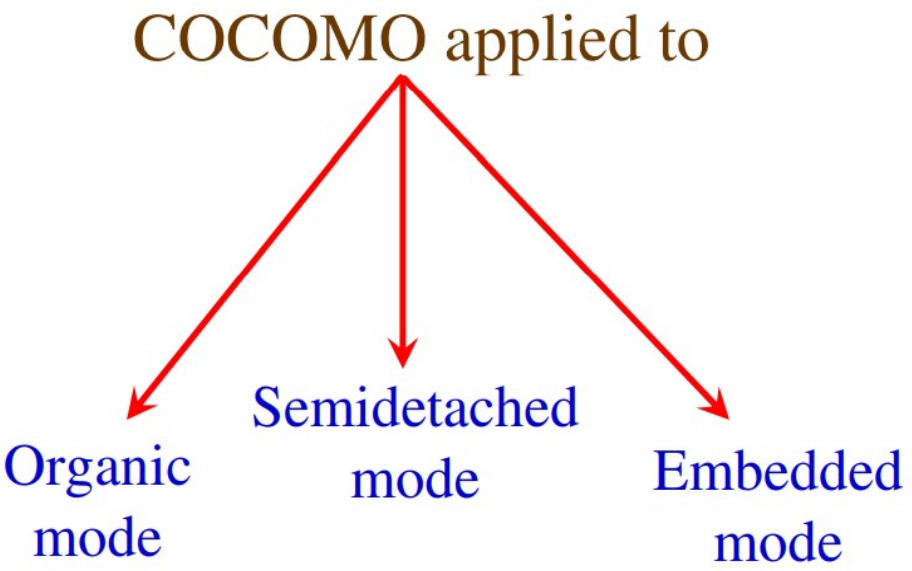
The Constructive Cost Model (COCOMO)

COCOMO is a hierarchy of software cost estimation models, which include basic, intermediate, and detailed sub models.



Basic Model

This model aims at estimating in a quick and rough fashion. Three modes of development are considered in this model: organic, semi-detached and embedded.



Comparison of three modes are given below.

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Table 4: The comparison of three COCOMO modes

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} \; Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} \; KLOC / PM$$

Example: 1

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

Solution

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (KLOC)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$

Example: 2

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{0.35} = 29.3$ PM

$$\text{Average staff size } (SS) = \frac{E}{D} Persons$$

$$= \frac{1133.12}{29.3} = 38.67 Persons$$

$$\text{Productivity } = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765 KLOC / PM$$

$$P = 176 LOC / PM$$

Intermediate Model

The basic model allowed for a rough and quick estimate, but it resulted in lack of accuracy. In this model, 15 cost drivers are used to adjust the nominal cost of a project to the actual project environment, hence increasing the accuracy of the estimate.

The cost drivers are grouped into four categories:

Intermediate Model

Cost drivers

(*i*) Product Attributes
- Required s/w reliability
- Size of application database
- Complexity of the product

(*ii*) Hardware Attributes
- Run time performance constraints
- Memory constraints
- Virtual machine volatility
- Turnaround time

(*iii*) Personal Attributes
- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(*iv*) Project Attributes
- Modern programming practices
- Use of software tools
- Required development Schedule

Each cost driver is rated for a given project environment. The rating uses a scale very low, low, nominal, high, very high, extra high which describes to what extent the cost driver applies to the project being estimated. Table 5 gives the multiplier values for the 15 cost drivers and their rating as provided by Boehm .

Multipliers of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Table 5: Multiplier values for effort calculations

The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF).

Intermediate COCOMO equations

$$E = a_i (KLOC)^{b_i} * EAF$$
$$D = c_i (E)^{d_i}$$

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

Detailed COCOMO Model

This model offers a means for processing all the project characteristics to construct a software estimate. This model introduces two more capabilities.

1. Phase-sensitive effort multipliers:

   The detailed model provides a set of phase sensitive effort multipliers for each cost driver.

2. Three-level product hierarchy:

   Three product levels are defined. These are module, subsystem and system levels. The ratings of the cost drivers are done at appropriate level

Development phases

A software development is carried out in four successive phases: plans/requirements, product design, programming and integration/test.

1. **Plan/requirements:** This is the first phase of the development cycle. The requirement is analyzed, the product plan is set up and a full product specification is generated. This phase consumes from 6% to 8% of the effort and 10% to 40% of the development time.
2. **Product design:** The second phase of the COCOMO development cycle is concerned with the determination of the product architecture and the specification of the subsystem. This phase requires from 16% to 18% the nominal effort and can last from 19% to 38% of the development time.
3. **Programming:** The third phase of the COCOMO development cycle is divided into two sub phases: detailed design and code/unit test. This phase requires from 48% to 68% of the effort and lasts from 24% to 64% of the development time.

4. **Integration/Test:** This phase of the COCOMO development cycle occurs before delivery. This mainly consists of putting the tested parts together and then testing the final product. This phase requires from 16% to 34% of the nominal effort and can last from 18% to 34% of the development time.

Principle of the effort estimate

Size equivalent

As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I$$

The size equivalent is obtained by

$$S\ (equivalent) = (S \times A) / 100$$

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Lifecycle Phase Values of μ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded extra large S≈320	0.08	0.18	0.24	0.24	0.34

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Lifecycle Phase Values of τ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded extra large S≈320	0.40	0.38	0.16	0.16	0.30

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Example: 4.8

Consider a project to develop a full screen editor. The major components identified are:

I. Screen edit

II. Command Language Interpreter

III. File Input & Output

IV. Cursor Movement

V. Screen Movement

The size of these are estimated to be 4k, 2k, 1k, 2k and 3k delivered source code lines. Use COCOMO to determine

1. Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0)
2. Cost & Schedule estimates for different phases.

Solution

Size of five modules are:

Screen edit	= 4 KLOC
Command language interpreter	= 2 KLOC
File input and output	= 1 KLOC
Cursor movement	= 2 KLOC
Screen movement	= 3 KLOC
Total	**= 12 KLOC**

Let us assume that significant cost drivers are

i. Required software reliability is high, i.e.,1.15

ii. Product complexity is high, i.e.,1.15

iii. Analyst capability is high, i.e.,0.86

iv. Programming language experience is low,i.e.,1.07

v. All other drivers are nominal

EAF = 1.15x1.15x0.86x1.07 = 1.2169

(a) The initial effort estimate for the project is obtained from the following equation

$$E = a_i (KLOC)^{bi} \times EAF$$

$$= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM}$$

Development time $D = C_i(E)^{di}$

$$= 2.5(52.91)^{0.38} = 11.29 \text{ M}$$

(b) Using the following equations and referring Table 7, phase wise cost and schedule estimates can be calculated.

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design	= 0.16 x 52.91 = 8.465 PM
Detailed Design	= 0.26 x 52.91 = 13.756 PM
Module Code & Test	= 0.42 x 52.91 = 22.222 PM
Integration & Test	= 0.16 x 52.91 = 8.465 Pm

Now Phase wise development time duration is

System Design	= 0.19 x 11.29 = 2.145 M
Detailed Design	= 0.24 x 11.29 = 2.709 M
Module Code & Test	= 0.39 x 11.29 = 4.403 M
Integration & Test	= 0.18 x 11.29 = 2.032 M