# Syllabus ...

| Unit No. | Contents | Weightage in % | Number of Sessions |
|---|---|---|---|
| 1. | **Overview of Software Engineering**<br>1.1 Overview of Software Engineering<br>1.2 SDLC Models<br>1.3 Requirement Engineering<br>1.3.1 Types of Requirements: Functional and Non-functional<br>1.3.2 Four Phases of Requirement Engineering<br>1.4 Software requirement Specification (SRS)<br>1.4.1 Structure and Contents of SRS<br>1.4.2 IEEE SRS Format<br>Case Studies : Based on SRS | 15 | 16 |
| 2. | **System Analysis and Modeling**<br>2.1 Use case diagrams<br>2.2 Class Diagram<br>2.3 Activity Diagram<br>2.4 Interaction Diagram<br>2.5 Package, component and deployment Diagrams<br>Case Studies Based on Diagrams | 20 | 08 |
| 3. | **Fundamentals of Project Management**<br>3.1 Overview of Project MANAGEMENT<br>3.2 Project Management Life Cycle-IEEE Life Cycle<br>3.3 Quality Metrics<br>3.4 Risk Management Process<br>3.5 Linear Software Project Cost Estimation<br>3.5.1 COCOMO-I (Problem Statement)<br>3.5.2 Function Point Analysis (Problem Statement)<br>3.5.3 The SEI Capability Maturity Model CMM<br>3.5.4. Software Configuration management<br>Case Studies/Numerical Problems Based on Risk management , COCOMO-I and FPA | 25 | 12 |

| | | | |
|---|---|---|---|
| 4. | **Agile Project Management Framework**<br>4.1 Introduction and Definition Agile, Agile Project Life Cycle<br>4.2 Agile Manifesto: History of Agile and Agile Principles<br>4.3 Team and roles of an Agile Team: Scrum Master Product Owner, Development Team<br>4.4 Key Agile Concepts<br>4.5 User Stories, Story Points<br>4.6 Techniques for estimating Story Points<br>4.7 Product Backlog<br>4.8 Sprint Backlog,<br>4.9 Product Vision and Product Roadmap<br>4.10 Sprint Velocity<br>4.11 Swim lanes<br>4.12 Minimum Viable Product (MVP)<br>4.13 Version and Release<br>4.14 Agile Project Management v/s Traditional Project Management<br>4.15 Agile Reports: Daily Reports, Sprint Burn Down Chart and Reports<br>User Stories Scenarios and Writing User Stories | **30** | **14** |
| 5. | **Implementation with Agile Tools**<br>5.1 MS Project Tool<br>5.2 Agile Tools: Open Source<br>5.3 Hands on GitHub<br>5.4 Create Project using Kanban<br>5.5 Project Repositories<br>5.6 Continuous Integration<br>5.7 Project Backlog<br>5.8 Team Management | **10** | **05** |

# Contents ...



***

UNIT 1

# Overview of Software Engineering

## Objectives...

After reading this chapter, you will be able:

- To learn basic concepts of software engineering.
- To learn about basic system development cycle.
- To study basics of requirement engineering.
- To know about different approaches and models for system Development.
- To describe functional and non-functional requirements.
- To understand the software requirement specifications.

## 1.1 OVERVIEW OF SOFTWARE ENGINEERING

- The nature of the software is changing continually. Complexities are increasing day by day. Huge amount of data need to be handled by the system in efficient manner. To tackle all these requirement of software different development methodologies are used.
- These are majorly categorized in two approaches as structured system analysis & design (SSAD) and object oriented analysis & design (OOAD).
- Structured Systems Analysis and Design Methodology (SSADM) is a set of standards for systems analysis and application design. It uses a formal methodical approach to the analysis and design of information systems. It was developed by **Learmonth Burchett Management Systems** (LBMS) and the Central Computer Telecommunications Agency (CCTA) in 1980-1981 as a standard for developing British database projects.
- It has been used by many commercial businesses, consultants, educational establishments and CASE tool developers.
- SSAD emphasizes on the identification of the functionalities related with the system. Systematic approach is used for investigating the functional aspects of the system. Relationship between functionalities within system or outside the system will be identified. This phase is called as **System Analysis** phase.
- The design that includes overall hardware, software architecture, components, processes involved in the system is called as the **System Design.** The complete structure of the system is identified by system design.

- A system is an orderly grouping of interdependent components linked together according to a plan to achieve a specific objective.
- The system will be divided in to its **subsystem** depending upon their functionalities. Each system will be divided into subsystems & further each subsystem is again divided into smaller subsystem. This division of the system will go until the last individual task or functions are identified.
- The approach for division is in **Top-Down manner**. Every time the top module will be subdivided into smaller modules. The structure of the system will be in hierarchical manner. All the functionalities of the system will be at the bottom of the structure as the approach used is in top down manner.

**Characteristics of a System:**

- According to definition of system some characteristics that are present in all systems. Following are the some important characteristics of a system:

1. **Organization:**
   - Organization implies structure and order.
   - Basically, organization means the arrangement of components that helps to achieve objectives or goals.
   - A computer system is designed around an input device, a central processing unit, an output device and one or more storage units. When linked together they work as a whole system for producing information.
2. **Interaction:**
   - Interaction refers to the manner in which each component interacts with other components of the system.

     Thus, interaction is the media of which every component interacts or communicates with other for proper functioning.
   - In a computer system, the central processing unit must interact with the input device to solve a problem. In turn, the main memory holds programs and data that the arithmetic unit uses for computation. The interrelationship between these components enables the computer to perform.
3. **Interdependence:**
   - Interdependence means that parts of the organization depend on one another. They are co-ordinate and linked together according to plan.
   - One subsystem depends on the input of another subsystem for proper functioning i.e. the output of one subsystem is the required input for another
   - subsystem.
4. **Integration:**
   - Integration is concerned with how a system is tied together in order to achieve common goal.

- Integration refers to holism of system.
- Synthesis follows analysis to achieve the central objective of organization. It means that parts of system work together within system even though each part performs unique function.

5. **Central Objective:**
   - It may be real or stated.
   - The stated objective and real objective of the system could differ based on the policy of the company.
   - The user should develop a central objective by taking into consideration real objective and stated objective.
   - The important point is that the users must know the central objective of computer application in the analysis for successful design and conversion.
   - This means that the analyst must work around the obstacles to identify real objective of proposed change.

## 1.1.1 Elements of System

- To construct a system the following key elements of a system must be considered:

1. **Outputs and Inputs:**
   - A major objective of a system is to produce the output as per the user's requirement.
   - The output could be in the form of goods (finished products), information as services. It must be done with expectations of intended user.
   - Inputs are elements that make the system to work in order to produce required output. Input could be material, human resources or information. Inputs are elements that enter the system for processing.
   - Output is outcome of processing. Determining the output is a first step in specifying the nature, amount, and regularity of the input needed to operate a system.
   - For example, in Systems Analysis, the first concern is to determine the user's requirements of a proposed computer system.
2. **Processors:**
   - The processor is the operational element of a system that involves the actual transformation of input into output.
   - The processor should be designed of such type that it can accept the input in the given format and can give output in desired format.
3. **Control:**
   - The control elements guide the system. It is decision making subsystem that controls the working of the system at all stages.

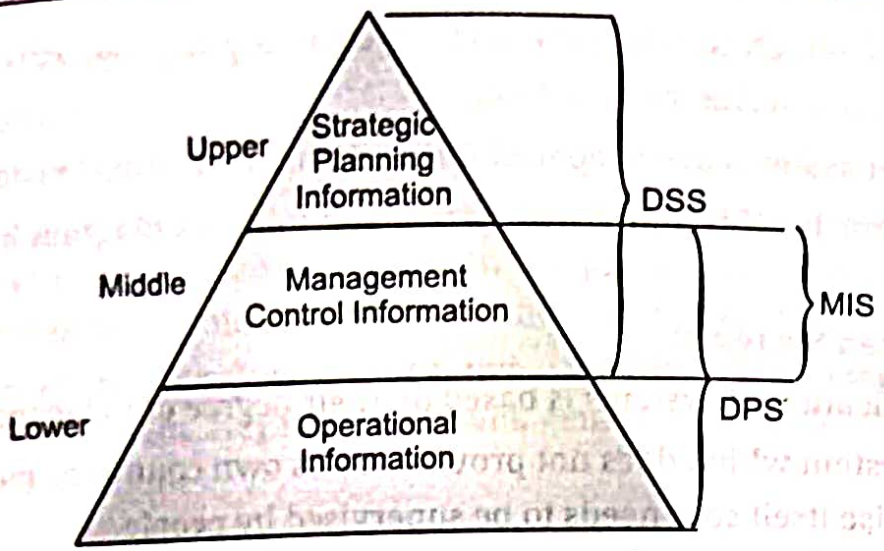



**Fig. 1.1: Organization Chart of FIS**

- There are three categories of information related to management levels and decision managers make.

(i) **Strategic Information:** The first level relates to long range planning policies that are of direct interest to upper management.

(ii) **Management Control Information:** The second level is of direct use of middle management and department heads for implementation and control.

(iii) **Operational Information:** The third level is short term, daily information is used to operate department and enforce day-to-day rules and regulations of information.

5. **Informal Information System:**
   - A formal information system is one which is shown on the chart, on the other hand, the informal information system is one which is working to meet requirements of employees.
   - Thus, informal information system is related with what is happening practically rather than what is shown on paper.
   - It is an employee based system designed to meet personnel and vocational needs and help to solve work–related problems.
   - It also guides information upward through indirect channels. In this respect, it is a useful system because it works within the framework of the business and its stated policies.
6. **Computer-based Information System:**
   - In man-made (manual) information system papers were used to hold the information. But today entire world depends on the computer.
   - Computer based information systems are faster, more accurate, more neat and attractive. It is possible to perform different operations easily. In this system, security of data is essential.
   - The system analyst must be familiar with computer technology and have experience in handling people in an organizational context.

7. **Management Information System (MIS):**
   - A MIS is a system that provides historical information and information on the current status. It is a communication process in which data are recorded and processed for further operational uses.
   - A MIS is a system that collects, processes, stores and distributes information to help in decision making for the managerial functions such as planning, organizing, directing, controlling and staffing a business organization.
   - Within a MIS, a single transaction can simultaneously update all related data files in the system. In so doing, data redundancy (duplication) and the time it takes to duplicate data are kept to a minimum, thus insuring that data are kept current at all times.
8. **Decision Support Systems (DSS):**
   - DSS systems make use of analytical planning modules (operation research model).
   - DSS mostly used for assisting top-level management in decision making. Better decision can be taken using DSS. DSS reduces clerical work and overtime.
   - DSS also saves cost and time. It consists of decision making with support of other lower level systems (MIS).
   - DSS systems used organizational data as well as external data collected from environment of the organization.
9. **Expert System (ES):**
   - ES operates with few rules. Effectiveness is a major goal of these types of systems. Human beings are experts in specific areas.
   - ES are more flexible than other systems. ES increases output and productivity of the system.
   - ES gives effective manipulation of large knowledge based system. The output is selected with the opinion of many experts.
   - Expert system consists of following components:
     (i) User interface. (ii) Explanation facility.
     (iii) Knowledge acquisition. (iv) Knowledge-based Facts rules.
     (v) Knowledge refining system. (vi) Explanation facility.
10. **Execution Information Systems (EIS):**
    - An EIS system operates continuously to look into what is happening in all major areas.
    - EIS structured tracking system. It provides rapid access to timely information and direct access to management reports.
    - EIS contains extensive graphics capabilities. It serves the information needs of top executives. EIS gives quick and easy access to detailed information.

## 1.1.3 Overview of Software and Software Engineering

- Software is set of instructions, (computer programs) that when executed provid desire function and performance.
- IEEE defines Software engineering as, "the application of a systematic, disciplined quantifiable approach to the development, operation and maintenance of software".
- The objective of software engineering is to produce software products. Software products are software systems delivered to a customer with the documentation which describes how to install and use the system.
- The main characteristics of software are software is developed or engineered, software does not wear out and most software is custom-built, rather than being assembled from existing components.
- Today, software can be applied in various fields like, education, business, social sectors, etc. Software is designed to suite some specific goal or task like data processing, communication, information sharing etc.
- Software engineering is needed for develop complex and critical software systems in a timely manner, with high quality and low cost.

## 1.2 SDLC MODELS

- Software development Life Cycle (SDLC) is a set of steps, which are used for building software.
- This is an organizational process of developing and maintaining software systems.
- It helps in establishing a system project plan, because it gives overall list of processes and sub-processes required for developing a system.
- SDLC means combination of various activities. In other words, we can say that various activities put together are referred as SDLC.
- The SDLC has been widely used in the design of system that can be easily understood and which have well defined workflows.
- In information systems and software engineering, the SDLC also referred to as the application development life-cycle, is a process for planning, creating, testing, and deploying an information system.
- The steps/phases involved in SDLC are shown below:
  1. Recognition of need / Understanding the user requirements
  2. Feasibility study.
  3. Analysis.
  4. Design.
  5. Implementation.
  6. Post-implementation and Maintenance.

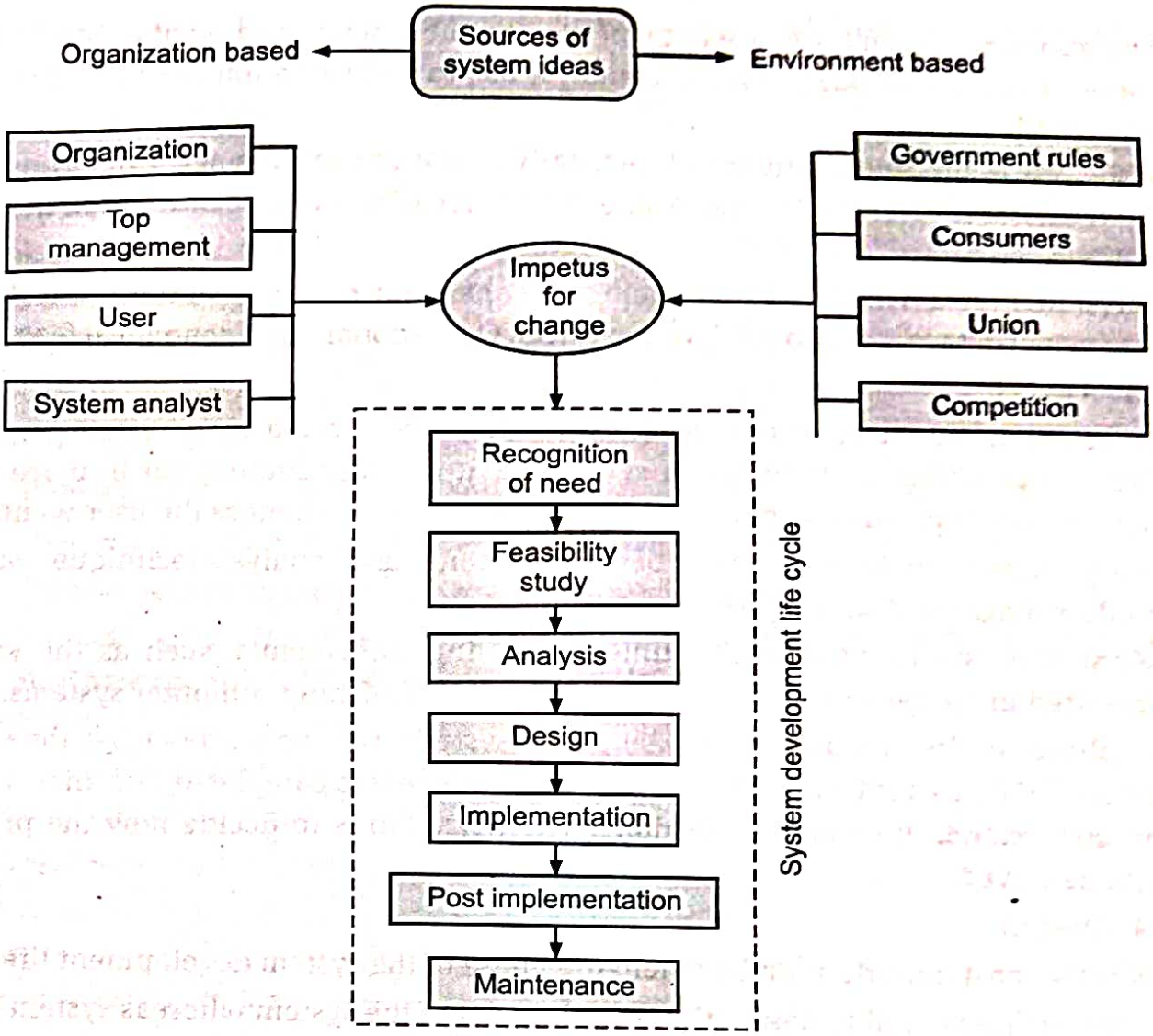



**Fig. 1.2: Software Development Life Cycle**

**Phase 1 - Recognition of Need:**

- This is the first step in SDLC. In this step, system analyst has to find out why a new system is required.
- In this stage, he/she should understand what are the present methods of solving the problem for which we are designing the system? What new the user wants in the new system?
- In short, system analysts describe user requirement, (system) project objectives, system and resource limits.

**Phase 2 - Feasibility Study:**

- The Feasibility Study proposes one or more conceptual solutions to the problem st for the project. The conceptual solutions give an idea of what the new system will look like.
- They define what will be done on the computer and what will remain manual.

- They also indicate what input will be needed by the systems and what outputs will be produced i.e. in feasibility study the analyst has to do evaluation of existing systems and procedures.
- He/she has to present a number of alternative solutions to the user. After consulting with user, the analyst has to finalize one alternative which will be best for all the given solutions.
- Analyst identifies whether it is possible to go ahead for development of the system or not considering various aspects like Technical, Operational and Economical feasibility.

**Phase 3 - Analysis:**

- This is a detailed study of the various operations performed by a system and their relationships within and outside of system i.e. it includes finding out in more detail what the system problems are and what the different new changes the user wants.
- During analysis, analysts use many of the commonly used analysis techniques, such as data flow diagram, decision table and so on.
- Analyst must spend considerable time examining components, such as the various forms used in the system, as well as the operation of finishing computer systems.

This phase results in a detailed model of the system. The model describes the system functions, data and information flows i.e. once analysis is completed, the analyst has a firm understanding what is to be done. The next step is to decide how the problem might be solved.

**Phase 4 - Design:**

- This is the most important and challenging phase of the system development life cycle.
- Analysis phase is used to design the logical model of the system whereas system design phase is used to design the physical model of the system.
- The design phase produces a design for the new system. There are many things to be done here.
- Designers must select the equipment needed to implement the system. They must specify new programs or changes to existing programs as well as a new database or changes to the existing database.
- Designers must also produce detailed documents that describe how users will use the system. Thus, in this phase, the designer/analyst designs:
  1. **Output:** Output design means that what should be the format for presenting the results obtained. It should be in most convenient, attractive format for the user.
  2. **Input:** In input design phase, which is a part of system design phase the system analyst has to decide what inputs are required for the system and prepare input format to have input to system according to user requirement.
  3. **File:** File design deals with how the data has to be stored on physical devices.

4. **Processing:** Finally process design includes the description of the procedure for carrying out operations on the given data.

**Phase 5 - Implementation:**

- During implementation, the components built during development are put into operational use.
- Following are the activities which takes place during System Implementation phase:
  1. Writing, testing, debugging and documenting programs.
  2. Converting data from the old to the new system.
  3. Giving training to the users about how to operate the system.
  4. Ordering and installing any new hardware.
  5. Developing operating procedures for the computer centre staff.
  6. Establishing a maintenance procedure to repair and enhance the system.
  7. Completing system documentation.
  8. Evaluating the final system to make sure that it is fulfilling original needs and that it began operation on time and within budget.

**Phase 6 - Post Implementation and Maintenance:**

- The system is considered to be working when Phase 5 has been completed. However, there are still a number of activities that take place after a system is completed.
- The two main activities are: 1.Post implementation and 2. Maintenance.
- The post-implementation activity take place after about a year to determine that whether the system is perfectly working or not, whether the system is satisfying the user needs or not, if not, changes are made to the system to make it perfect.
- Maintenance is necessary to eliminate errors in the system during its working life and to tune the system to any variations in its working environments.
- Maintenance has been seen that there are always some errors found in the systems that must be noted and corrected.
- Maintenance includes following tasks:
  1. Correcting errors.
  2. Resolving necessary changes.
  3. Enhances or modifies the system.
  4. Assigns staff to perform maintenance activities.
  5. Provides for scheduled maintenance.
- **Example:** If Corporation changes Octroi rate from one value to another, corresponding changes will have to be done in the programs which uses this Octroi rate.

## 1.2.1 Types of SDLC Model

- A Software Life Cycle Model is either a descriptive or prescriptive characterization of how software is or should be developed.

1. **Descriptive Model:**
   - A descriptive model describes the history of how a particular software system was developed. A descriptive process model is defined as, "a model that describes 'what to do' according to a certain software process system".
   - Descriptive models may be used as the basis for understanding and improving software development process or for building empirically grounded prescriptive models.
2. **Prescriptive Model:**
   - Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in which order.
   - Prescriptive model describes what should be done during software development, including responses to error situations.

**Types of Prescriptive Model:**

- There are three types of prescriptive process models as following:

**(i) Waterfall Model:**

- In waterfall model, a work flow is in a linear (sequential) fashion. The waterfall model often used with well-defined adaptations or enhancements to current software. In a Waterfall model, each phase must be completed fully before the next phase can begin.

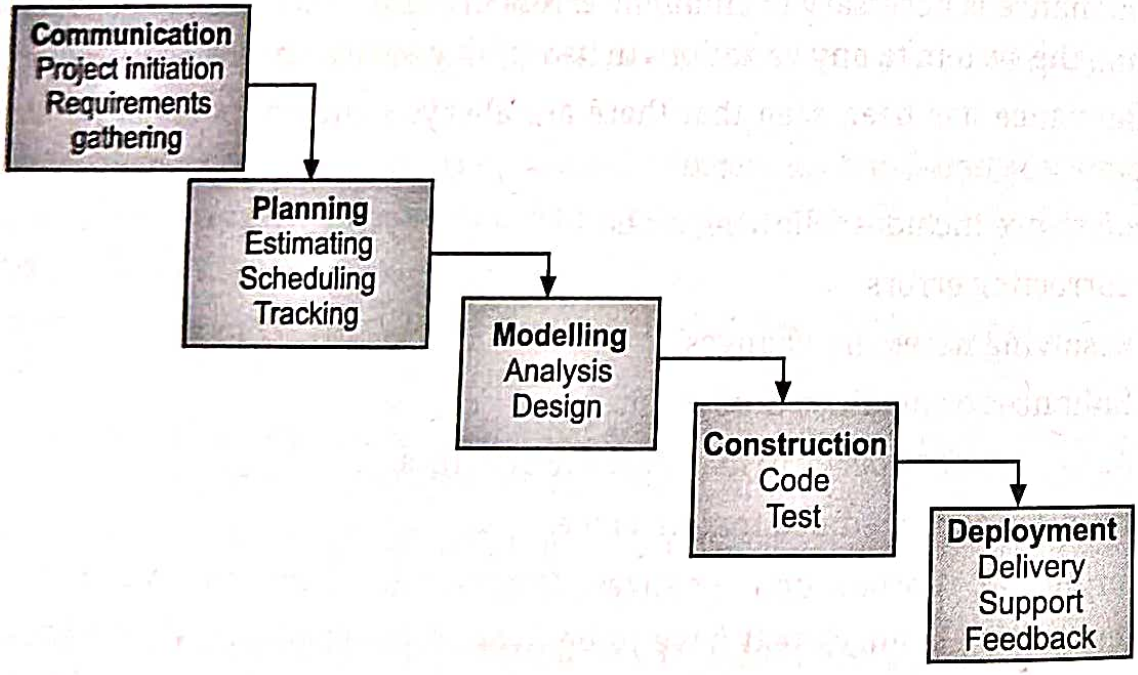



*Fig. 1.3: Waterfall Model*

**Advantages:**

1. Waterfall model is a linear model and of course, linear models are the most simple to be implemented.
2. The amount of resources required to implement this model is very minimal.
3. One great advantage of the waterfall model is that documentation is produced at every stage of the waterfall model development. This makes the understanding of the product designing procedure simpler.
4. After every major stage of software coding, testing is done to check the correct running of the code.
5. In waterfall model progress of system is measurable.

**Disadvantages:**

1. In waterfall model, there is the difficulty of accepting change after the process is in progress. It does not support iteration, so changes can cause confusion.
2. It has high amount of risks and uncertainty.
3. It requires customer patience because a working version of the program does not occur until the final phase.
4. It is a poor model for long and ongoing projects.

**(ii) Incremental Process Model:**

- The incremental model is defined as, "a model of software development where the product is designed, implemented and tested incrementally (a little more is added each time) until the product is finished."
- In incremental model, multiple development cycles take place that making the software life cycle a multi-waterfall. In this model, cycles are divided up into smaller, more easily managed modules.
- The work flow is in a linear (sequential) fashion within an increment and is staggered between increments. Iterative nature of this model focuses on an operational product with each increment.

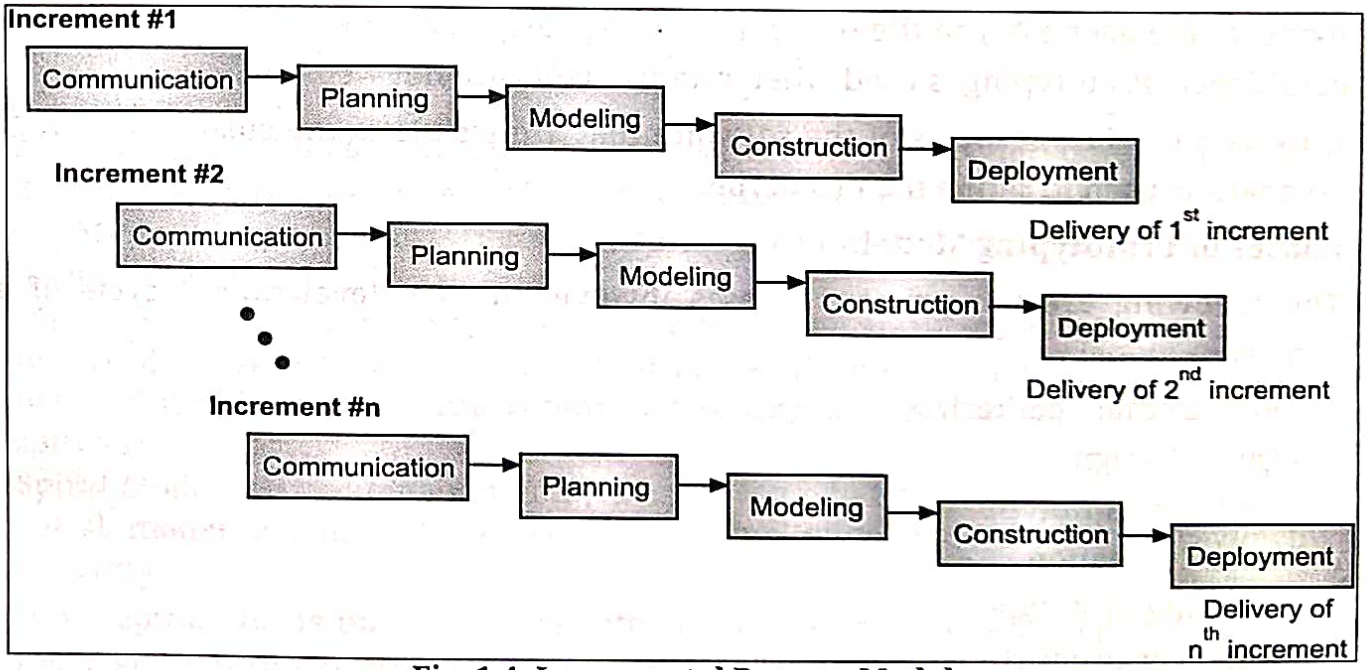



**Fig. 1.4: Incremental Process Model**

**Advantages:**

1. This model generates working software quickly and early during the software life cycle.
2. This model is more flexible to change scope and requirements.
3. It is easier to test and debug during a smaller iteration.
4. In this model, customer can respond to each built.
5. Lowers initial delivery cost.
6. Easier to manage risk because risky pieces are identified and handled during iteration.

**Disadvantages:**

1. Needs good planning and design.
2. Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
3. Total cost is higher than waterfall model.

**(iii) Evolutionary Process Model:**

- Evolutionary model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.
- It has basic two models i.e. Prototyping and Spiral Model.

**(a) Prototyping Model:**

- Prototyping follows an evolutionary and iterative approach. It provides a working model to the user early in the process, enabling early assessment and increasing user's confidence. Prototyping is used when requirements are not well understood.
- It focuses on those aspects of the software that are visible to the customer/user. User feedback is used to refine the prototype.

**Phases of Prototyping Model:**

The following are the primary phases involved in the development cycle of any prototype model:

- Requirement gathering
- Quick Design
- Building Prototype
- User Evaluation
- Refining prototype
- Engineer product

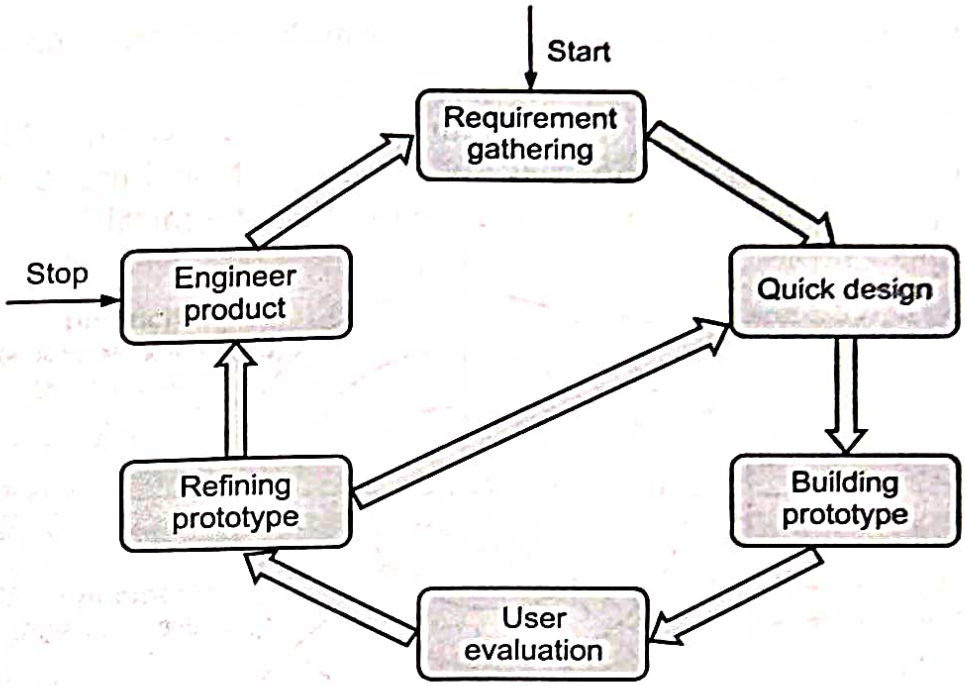



**Fig. 1.5: Prototype Model**

**Advantages:**

1. Prototype model need not know the detailed input, output, processes, adaptability of operating system and full machine interaction.
2. In development process of this model users are actively involved.
3. The development process is the best platform to understand the system by the user.
4. Errors are detected much earlier.
5. It gives quick user feedback for better solutions.
6. In this model, missing functionality can be easily identified.
7. It also identifies confusing or difficult functions.

**Disadvantages:**

1. The client involvement is more and it is not always considered by the developer.
2. It is a slow process because it takes more time for development.
3. Many changes can disturb the rhythm of the development team.

**(b) Spiral Model:**

- IEEE defines, the spiral model as, "a model of the software development process in which the constituent activities, typical requirements analysis, preliminary and detailed design, coding, integration and testing are performed iteratively until the software is complete".
- Spiral Model is a combination of a waterfall model and iterative model. Each phase in spiral model begins with a design goal and ends with the client reviewing the progress.
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping.

- Quiet spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software.
- The baseline spiral, starting in Communication phase, the Planning phase requirements are gathered and risk is assessed, Modeling, Construction and Deployment. Each subsequent spiral builds on the baseline spiral.

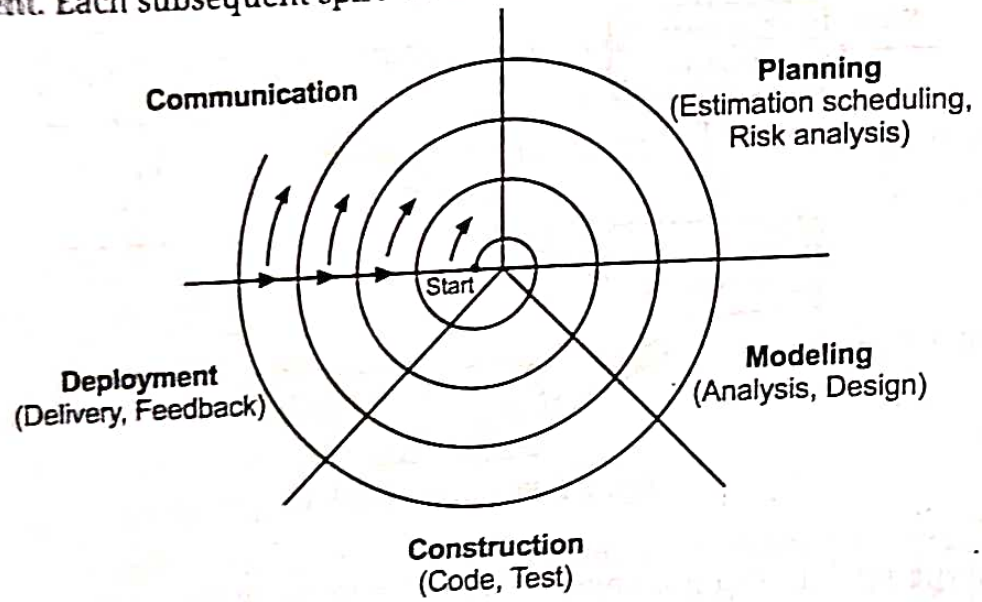



**Fig. 1.6: Spiral Model**

**Advantages:**

1. The spiral model accommodates life-cycle evaluation, growth and requirement changes.
2. It focuses on early error detection and design flaws.
3. It incorporates prototyping as a risk-reduction strategy.
4. The spiral model incorporates software quality objectives into the product.

**Disadvantages:**

1. Spiral model can be a costly model to use.
2. It is not suitable for low risk projects.
3. Risk analysis in this model requires highly specific expertise.

## 1.2.2 Different Users and their Role in SDLC

- Different people involve in development process of software plays different and unique role in SDLC. Some people are directly involved in development process where some people indirectly or directly support them. People involved in the development process are known as **stakeholders**. These stakeholders plays very important role at their level. Following are the users and their role in SDLC:

1. **Sponsoring Agency:** It includes those people which are going to provide funding for the software development. The role of funding agency is very important as various expenses required for development phases are managed by funding agencies.

2. **End Users:** These people are directly or indirectly going to use the software. These people involves in SDLC even before starting the Inception Phase. In SDLC, the primary role of Users is to provide basic information required for development of software which is also known as **Software specification**. Generally these people are domain experts and in collaboration with domain expert/End users Analyst collect/gather required information.
3. **Analyst:** These people are directly involved in gathering the information from end user. In order to collect information from user, analyst can use various fact finding techniques. There primary role is to analyze the provided information to them. This analysis process includes finalizing technical and non-technical requirements, checking the feasibility, determining the approximate cost in the initial phase and time duration required to deliver the project.
4. **Steering Ccommittees** : These are advisory bodies that are made up of senior stakeholders or experts that provide guidance on a lot of different issues that could face companies such as budgets, new endeavours, company policy, marketing strategies, and project management concerns.
5. **Designer:** Designers are those people who design the system. It includes logical development of the system. Architectural plan of the system is created by designer on the basis of which actual implementation of the system starts. This process includes drawing various diagrams which represents the system virtually.
6. **Developer:** Developers are those people who transforms design model into the coding. Various factors need to be considered before coding like, target environment, maximum memory uses etc.
7. **Testers:** These people are responsible for ensuring the quality of the software. The role of tester is to execute the program with the intent of finding an error. Tester has to write various test cases on the basis of which quality of the software can be determine. He/she has to also prepare reports of testing on the basis of which developer can remove the errors from the software.
8. **Configuration Manager**: This person is responsible for various software/ hardware configuring management.
9. **Marketing Manager**: This person is responsible for promotion and marketing management of the software.

## 1.3 REQUIREMENT ENGINEERING

- Basically, a requirement is a feature that must be included in new system.
- A requirement may include a way of capturing or processing data producing information, controlling a business activity or supporting management.

- IEEE defines a requirement as:
  - *A condition that must be possessed by a system to satisfy a contract specification, standard or other formally imposed document.*

**OR**

  - *A condition or situation of capability needed by a user to solve a problem achieve an objective or to accomplish predefined goal or task.*
- SRS contains a contract between the customer and the developer. This SRS document is used for verifying whether all the functional and non-functional requirements specified in the SRS are implemented in the product.
- The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.
- Three main activities involved in the process of requirements determination are Requirements Anticipation, Requirements Investigation and Requirements Specification.

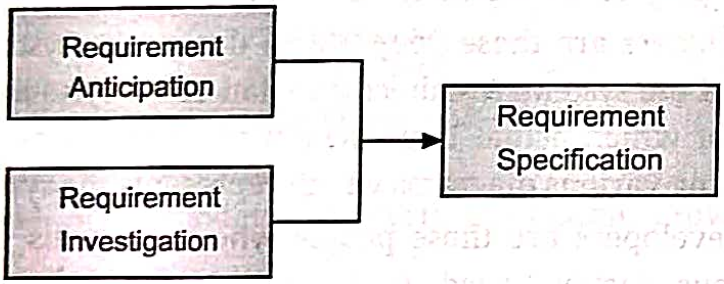



**Fig. 1.7: Activities in the process of Requirements**

- The requirement specification document will be produced or developed which states the requirement or needs of the customer for the new information system.

**Requirement Anticipation:**

- Requirement anticipation activities include the past experience of the analysis, which influence the study.
- This also includes certain problems or features and requirements for a new system.
- Requirement anticipation has following two approaches:
  1. First approach is, experience from previous studies can lead to the investigation of areas that would not noticed by an inexperience analyst. Having the background of the analysts to know what to ask or which aspects to investigate can be useful in the system investigation.
  2. Second approach is, if a biased is introduced or shortcuts are taken in conducting the investigation then requirement anticipation is not fully developed.

**Requirement Investigation:**

- Requirement investigation activity is the core of the system analysis.
- Requirement investigation includes methods for documentation and describing system features.

- An important and reliable method of gathering requirements is Requirement Investigation.
- Fact-finding techniques are employed to carry out the requirement investigation. All these investigation will be documented.
- Using the varieties of tools and schemes, analyst studies the current system and documents its features for further analysis.

## 1.3.1 Types of Requirements – Functional and Non-Functional

- Requirements of software system are classified as Functional requirements and Non-functional requirements.

**Functional Requirements:**

- The requirement that defines the system's functionality and its associated components is known as functional requirements. It defines those functions which system must perform.
- Function is nothing but the tasks that user want that system should perform which may accept some input, process it and generate desired output. Functional software requirements help you to capture the intended behavior of the system.
- Some of the typical functional requirements are:
  - Business Rules
  - Transaction corrections, adjustments and cancellations
  - Administrative functions
  - Authentication
  - Authorization levels
  - Audit Tracking
  - External Interfaces
  - Certification Requirements
  - Reporting Requirements
  - Historical Data
  - Legal or Regulatory Requirements

**Non-Functional Requirements:**

- The requirements that define the system's quality attributes are known as non-functional requirements. For example, the amount of time required by the system to generate desired output.
- Non-functional Requirements allows you to impose constraints or restrictions on the design of the system.
- Some typical Non-functional requirements are:
  - **Performance:** Performance is the responsiveness of the application to perform specific actions in a given time span. For example: Response Time, Throughput, Utilization, Static Volumetric.

- **Scalability:** Scalability is the ability to handle an increase in the work load without impacting the performance, or the ability to quickly expand the architecture.
- **Capacity:** This defines the ways in which the system is expected to scale-up by increasing capacity, hardware or adding machines based on business objectives. Capacity is delivering enough functionality required for the end users.
- **Availability:** Availability is measured as the percentage of total application downtime over a defined time period. Availability is affected by failures, exceptions, infrastructure issues, malicious attacks, and maintenance and upgrades.
- **Reliability:** Reliability is the ability of the application to maintain its integrity and veracity over a time span and also in the event of faults or exceptions. It is measured as the probability that the software will not fail and that it will continue functioning for a defined time interval.
- **Recoverability:** As part of disaster recovery, electronic backups of data and procedures must be maintained at the recovery location and be retrievable within the appropriate time frames for system function restoration. In the case of high criticality, real-time mirroring to a mirror site should be deployed.
- **Maintainability:** Maintainability is the ability of any application to go through modifications and updates with a degree of ease. This is the degree of flexibility with which the application can be modified, whether for bug fixes or to update functionality.
- **Extensibility:** Extensibility is the ability of a system to cater to future changes through flexible architecture, design, or implementation. Extensible applications have excellent endurance, which prevents the expensive processes of producing large inflexible applications and retiring them due to changes in business needs. Extensibility enables organizations to take advantage of opportunities and respond.
- **Security:** Security is the ability of an application to avoid malicious incidences and events outside of the designed system usage, and prevent disclosure or loss of information.
- **Manageability:** Manageability is the ease with which the administrators can manage the application, through useful instrumentation exposed for monitoring.
- **Usability:** Usability measures characteristics such as consistency and aesthetics in the user interface. Consistency is the constant use of mechanisms employed in the user interface while Aesthetics refers to the artistic, visual quality of the user interface.
- **Interoperability:** Interoperability is the ability to exchange information and communicate with internal and external applications and systems.

**Table 1.1: Difference between Functional and Non-functional Requirements**

| Functional Requirements | Non-functional Requirements |
|---|---|
| 1. It defines what a system is supposed to do. | 1. It defines how a system is supposed to be. |
| 2. It is specified by the User. | 2. It is specified by technical people like architects, Technical leaders, and software developers. |
| 3. It describes the functionality of the software. | 3. It describes the performance of the software. |
| 4. It is mandatory. | 4. It is not mandatory. |
| 5. Functional Testing such as System, Integration, API testing, etc. are done. | 5. Non-Functional Testing such as Performance, Usability, Stress, Security testing, etc. are done. |

## 1.3.2 Four Phases of Requirement Engineering

- Requirement engineering is the process of defining, documenting and maintaining the requirements. It is the process of collecting the information about the services, functions or tasks performed by the software system.

### 1.3.2.1 Feasibility Study

- Ideally, the requirements engineering process begins with feasibility study activity, which produces a feasibility report. It is possible that the feasibility may lead to a decision not to continue with the development of the software product. If the feasibility study suggests that the product should he developed, then requirements analysis can begin.

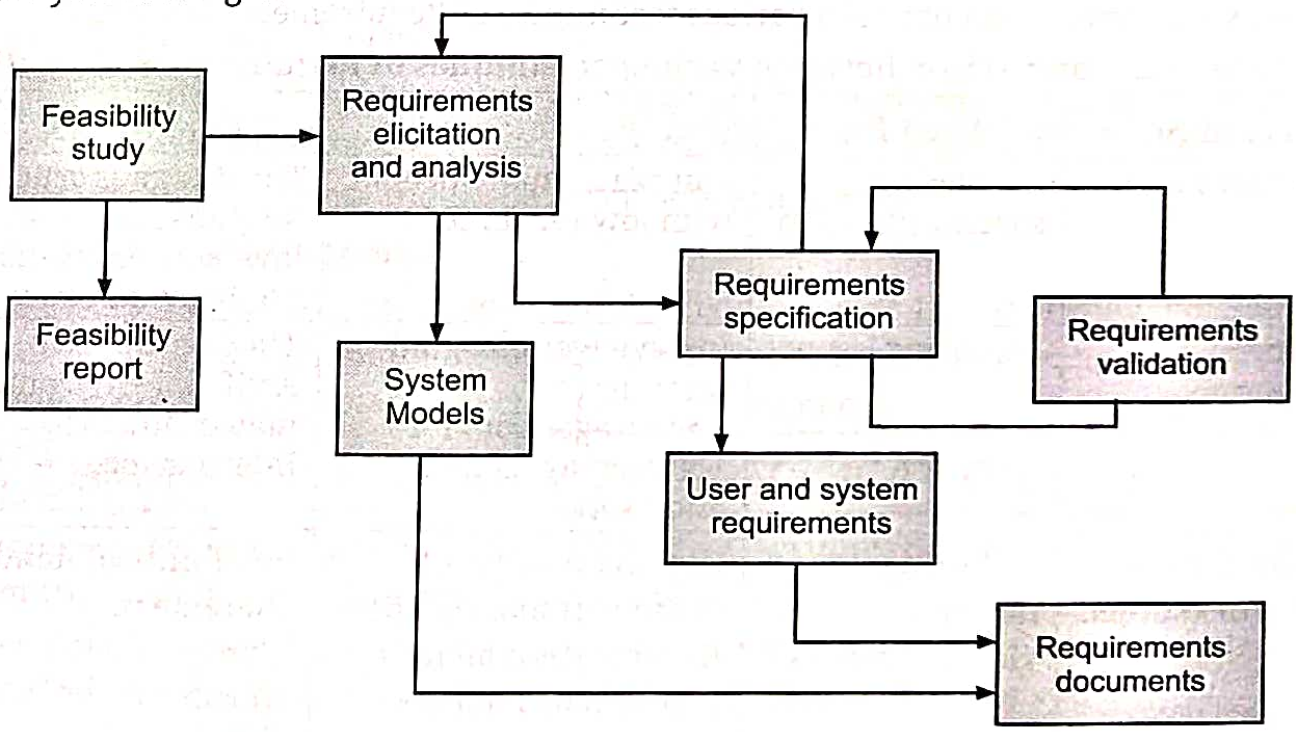



**Fig. 1.8: Requirement Engineering Process**

**Types of Feasibility:**

- Various types of feasibility that are commonly considered include technical feasibility, operational feasibility, and economic feasibility:
  - **Operational feasibility:** This checks the usability of the proposed software. If the operational scope is high, then the proposed system will be used more ensuring the acceptance from the sponsors of the project.
  - **Technical feasibility:** This checks whether the level of technology required for the development of the system is available with the software firm, including hardware resources, software development platforms and other software tools.
  - **Economic feasibility:** This checks whether there is scope for enough return by investing in this software system. All the costs involved in developing the system, including software licenses, hardware procurement and manpower cost, needs to be considered while doing this analysis. The cost-benefit ratio is derived as a result and based on the justification the stakeholders make a decision of going ahead with the proposed project.

## 1.3.2.2 Phases of Requirement Engineering Process

- Requirement engineering process consists of following phases:

**(1) Requirement Elicitation and Analysis:**

- This phase is about the gaining the domain knowledge and associated requirement. Domain knowledge can be acquired from domain experts or end users of the system. Various techniques can be used for requirement elicitation, which includes questionnaires, document reviews, interviews etc.
- Let us see comparison between various techniques of Requirement Elicitation:

**Table 1.2: Comparison between various techniques of Requirement Elicitation**

| Technique | Used for | Advantages | Disadvantages |
|---|---|---|---|
| **Questionnaire** | Answering specific question | Can reach many people with low resource. | The design is crucial. Response rate may be low. Responses may not be what you want. |
| **Interviews** | Exploring issues | Interviewer can guide interviewee. Encourages contact between developers and users. | Time consuming. Artificial environment may intimidate interviewee. |
| **Focus groups and workshops.** | Collecting multiple viewpoints. | Highlights areas of consensus and conflict. Encourages contact between developers and users. | Possibility of dominant characters. |

| Technique | Used for | Advantages | Disadvantages |
|---|---|---|---|
| **Naturalistic observation** | Understanding context of user activity. | Observing actual work gives insight that other techniques cannot give. | Very time consuming. Huge amounts of data. |
| **Studying documentation** | Learning about procedures, regulations, and standards | No time commitment from users required. | Day-to-day work will differ from documented procedures. |

**(2) Requirement Specification:**

- In this phase of requirement engineering, formal software requirement model is produced. Functional and non-functional requirements are documented along with the constraints. In this phase, if committee has deep domain knowledge then it may also trigger the elicitation process. Various models are used in this phase like ERD, DFD, and functional decomposition diagram and data dictionary.

**(3) Requirement Verification and Validation:**

- **Verification:** It is the process that ensures that the software correctly implements a specific function.
- **Validation:** It is the process that ensures that the correct process is implemented in software.
- In SDLC, rest of phases are dependent on the requirement analysis. If requirements are not properly verified and validated the same errors in the requirement would propagate to the successive phases of SDLC which may result in the defective software.
- To avoid all such problems:
  1. Requirement should not conflict with other requirements.
  2. Requirements should be very clear and complete in all sense.
  3. All the requirements should be practically achievable.

**(4) Requirements Management:**

- It includes the overall management of the requirements like analysing, documenting, tracking, prioritizing, communicating with stakeholders and agreeing on them. In this phase, there is an acceptance in change in requirements.

## 1.4 SOFTWARE REQUIREMENT SPECIFICATION (SRS)

- Requirement Specification is the process of writing down the user and system requirements in a requirement documents.
- The data produced during fact finding investigation are analysed to determine requirement specification i.e. the description of features for new system.
- It is an agreement between the system engineer/developer and the end users.

- Software Requirement specification activity has three inter-related parts.
  1. **Analysis of Factual Data:** The data collected during the fact finding study and included in data flow and decision analysis documentation are examined to determine how well the system is performing and whether it will meet the organization's demands.
  2. **Identification of Essential Requirement:** Features that must be included in a new system, ranging from operational details to performance criteria are specified.
  3. **Selection of Requirement Strategies:** The methods that will be used to achieve the stated requirement are selected. These form the basis for system design, which follows the requirement specification.
- All three activities are important and must be performed correctly.
- The Software Requirement Specification (SRS) is a technical specification of requirements for the software product.
- The term *specification* means different things to different people. A specification can be in a form of a written document, a graphical model, a mathematical mode prototype or any combination of the above.
- Some people say that a standard should be developed for the system requirement specification. So the requirements presented will be consistent and therefore, it will be understood nicely.
- But sometimes it is necessary to remain flexible for the systems which are very large a combination of written document, natural language descriptions and graphical models is the best approach. For smaller systems, usage scenarios are required.
- The system specification is the final output produced by the system and the requirements engineer. It serves the baseline (foundation) for hardware engineering, software engineering, database engineering and human engineering.
- It describes the function and performance of a computer based system and also the constraints which will govern the development. The specification bounds each allocated system element. The system specification also describes the input and output from the system.
- The software requirement specification is produced at the culmination of the analysis task. The National Bureau of standards, IEEE (Standard No. 830 - 1984) and the U.S. Department of Defence have all proposed candidate formats for software requirements specifications.
- The SRS records the outcome of the software requirements definition activity.
- The main and important goal of software requirements definition is to specify completely and consistently the technical requirements for the software product in a concise and unambiguous manner, using formal notations as appropriate.

- Depending on the size and complexity of the software product, the SRS may consist of a few pages and it is based on the system definition.
- The requirements specification will state the "What" of the software product without implying "how".
- Software design is concerned with specifying how the product will provide the required features and goals.

## 1.4.1 Structure and Contents of SRS

- The simplified outline presented in Fig. 1.5 may be used as a framework for the specification.
- SRS document includes the following sections:
  - "**Introduction**" states the goals and objectives of the software, describing it in the content of the computer-based system. Actually, it is nothing but the software scope.
  - The "**Information description**" provides a detailed description of the problems that the software must solve. Information content and relationships, flow and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.
  - A description of each function required to solve the problem is presented in the "**Functional description**". A processing narrative is provided for each function, design constraints are stated and justified; performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.
  - The "**Behavioural description**" section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.
  - Probably the most important and the most often ignored section of a software requirements specification is "**Validation criteria**". How do we identify a successful implementation? What classes of tests must be shown to validate function, performance and constituents?
  - "**Specification of validation criteria**" acts as an implicit review of all other requirements. It is essential that time and attention must be given to this section.
  - Finally, the software requirements specification includes a "**Bibliography**" and "**Appendix**". The **Bibliography** contains references to all documents that relate to the software. These include :
    - Other software engineering documentation.
    - Technical references
    - Vendor literature
    - Standards

- The **Appendix** contains information that supplements the specification. Tabular data, detailed description of algorithms, charts, graphs and other material are presented as appendices.

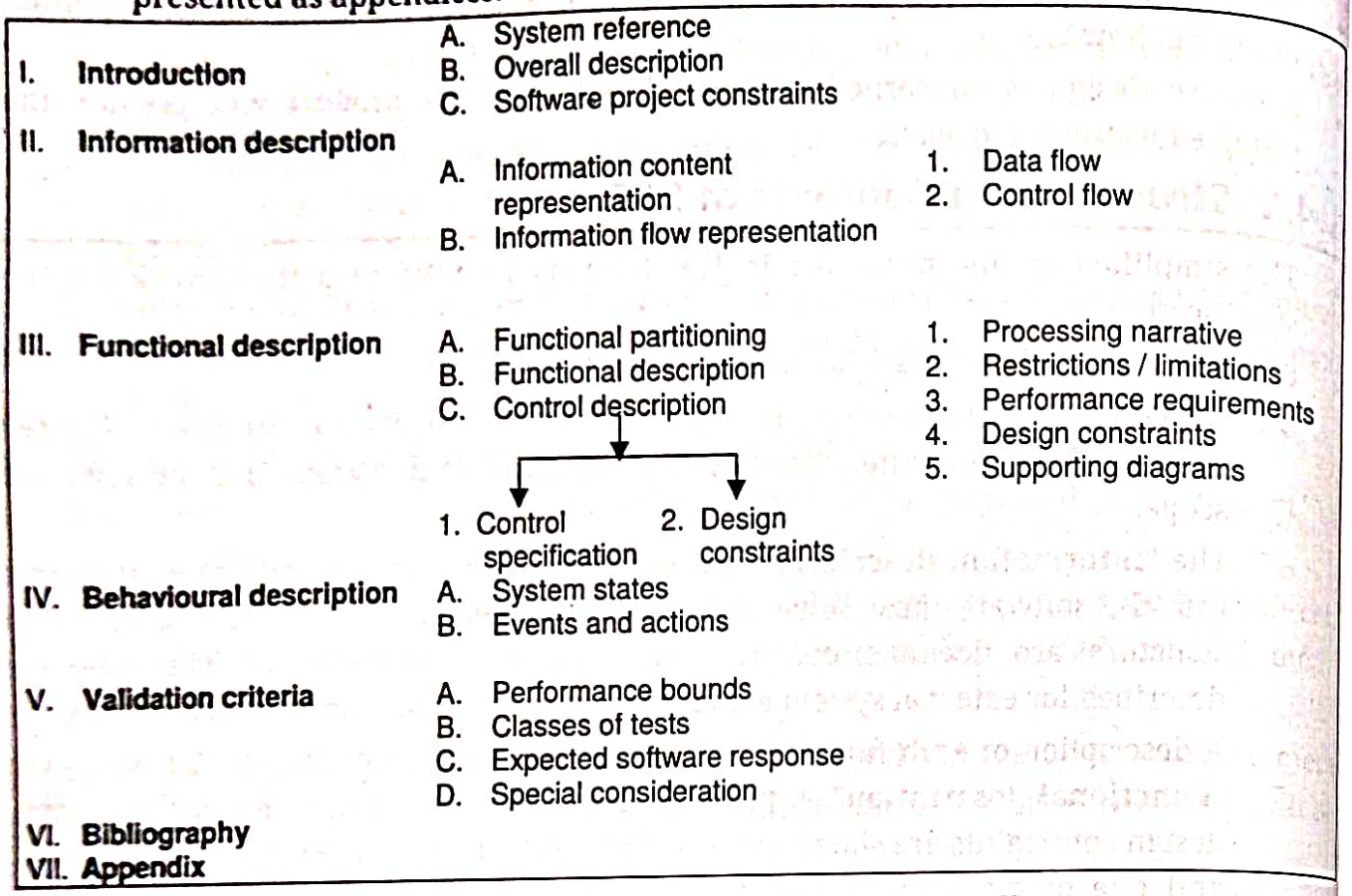



**Fig. 1.9: Software Requirements Specification Outline**

- In some cases, a requirement has seen as a high-level abstract statement of a service that the system should provide or a constraint on the system. In the other cases, it is detailed, mathematically formal definition of a system function.
- Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between the different levels of descriptions such as,
  1. Requirements definition.
  2. Requirements specification.
  3. Software specification.

1. A **"Requirements Definition"** is a statement in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate. It is generated using customer-supplied information.
2. A **"Requirements Specification"** is a structured document which sets out the system services in detail. This document, which is sometimes called a functional specification, should be precise. It may serve as a contract between the system buyer and software developer.
3. A **"Software Specification"** is an abstract description of the software which is a basis for design and implementation. This specification may add further detail to the requirements specification.

- The requirements engineering process should normally involve writing a requirements definition then expanding this into a requirements specification.
- Fig. 1.10 illustrates how a definition of a requirement may be expanded in more detail as a requirements specification.

**Requirements Definition :**

| | |
|---|---|
| 1. | The software must provide a means of representing and accessing external files created by other tools. |

**Requirements Specification :**

| | |
|---|---|
| 1.1 | The user should be provided with facilities to define the type of external files. |
| 1.2 | Each external file type may have an associated tool which may be applied to the file. |
| 1.3 | Each external file type may be represented as a specific icon on the user's display. |
| 1.4 | Facilities should be provided for the icon representing an external file type to be defined by the user. |
| 1.5 | When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon. |

**Fig. 1.10: Requirements Definitions and Specifications**

- Different levels of system specification are useful because they communicate information about the system to different types of reader.
- Fig. 1.11 shows the classes of reader that may be concerned with the different levels of specification.

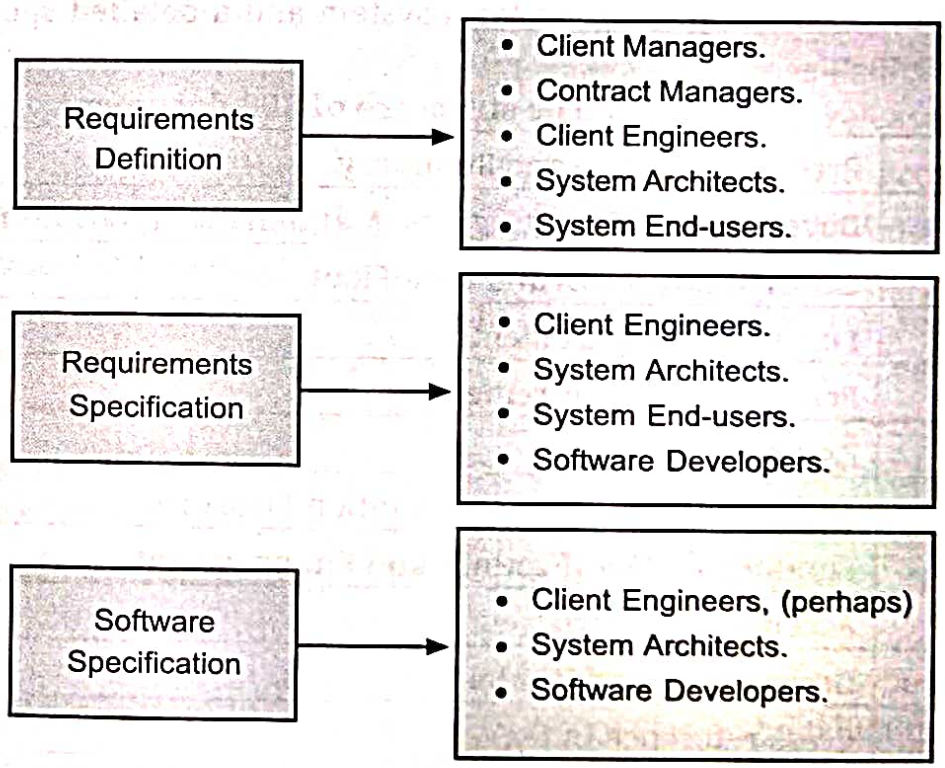



**Fig. 1.11 Readers of different types of Specification**

## 1.4.2 IEEE Standard Format of SRS

- The IEEE Std.830-1998 was created to standardize the software requirements specification document.
- The aim of an SRS document is to capture requirements in an unambiguous manner in order to facilitate communication between stakeholders.
- IEEE Std.830-1998 provides a structure (template) for documenting the software requirements.
- The SRS document described in IEEE Std.830 is divided into a number of recommended sections to ensure that information relevant to stakeholders is captured.
- This specification document serves as a reference point during the development process and captures requirements that need to be met by the software product.
- Basic issues addressed in the SRS include functionality, external interfaces, performance requirements, attributes and design constraints.
- SRS also serves as a contract between the supplier and customer with respect to what the final product would provide and help achieve.
- Although the IEEE Std. 830-1998 specifies the structure it does not choose one representation for requirements over the other. Neither does it specify what techniques should be used to populate the various sections.

**SRS Format:**

- SRS is the standard statement of what the system developers should implement.
- SRS includes the user's requirements for a system and a detailed specification of the system requirement.

**Table 1.3: The structure of SRS document**

| | |
|---|---|
| **Section 1** | Product Overview and Summary |
| **Section 2** | Development, Operating, and Maintenance Environments |
| **Section 3** | External Interfaces and Data Flow |
| **Section 4** | Functional Requirements |
| **Section 5** | Performance Requirements |
| **Section 6** | Exception Handling |
| **Section 7** | Early Subsets and Implementation Priorities |
| **Section 8** | Foreseeable Modifications and Enhancement |
| **Section 9** | Acceptance Criteria |
| **Section 10** | Design Hints and Guidelines |
| **Section 11** | Cross-reference Index |
| **Section 12** | Glossary of Terms |

SRS format's sections are described below:

- **Section 1 and 2** of SRS document present an overview of product features and summarize the processing environments for development, operation, and maintenance of the software product.
- **Section 3** of SRS document specifies the externally observable characteristics of the software product and it includes user displays and report formats, a summary of user commands and report options, data flow diagrams, and a data dictionary.
- **Section 4** of SRS document specifies the functional requirements for the software product. Typically, functional requirements are expressed in relational and state-oriented notations that specify relationships among inputs, actions and outputs etc.
- **Section 5** of SRS document specifies the performance characteristics like response time for various activities, processing time for various processes, throughput, primary and secondary memory constraints, required telecommunication bandwidth, and unusual reliability requirements etc.
- **Section 6** of SRS document specifies the exception handling, including the actions to be taken and the messages to be displayed in response to undesired situations or events or errors. Various categories of possible exceptions include temporary and permanent resource failure, incorrect, inconsistent or out of range input data, violation of capacity limits and violations of restrictions on operators etc.
- **Section 7** of SRS document specifies early subsets and implementation priorities for the system under development and it is important to specify implementation priorities for various system capabilities.
- **Section 8** of SRS document specifies foreseeable modifications and enhancements that may be incorporated into the product following initial product release.
- **Section 9** of SRS document specifies the software product acceptance criteria. Acceptance criteria specify functional and performance tests that must be performed, and the standards to be applied to source code, internal documentation and external documentations like the design specifications, the test plan, the user's manual, the principles of operations, and the installation and maintenance procedures and so on.
- **Section 10** of SRS document specifies design hints and guidelines. The "how to" of product implementation is the topic of software design, and should be deferred to the design phase of product.
- **Section 11** of SRS document specifies product requirements to the source of information used in deriving the requirements.
- **Section 12** of SRS document specifies definition of terms that may be unfamiliar to the customer and the product developers. Proper care should be taken to define standard terms that are used in non-standard ways.

## 1.4.3 Quality Characteristics of SRS

- Various quality characteristics of SRS are given below :

1. **Complete** : SRS defines precisely all the go-live situations that will be encountered and the system's capability to successfully address them.
2. **Consistent** : SRS capability functions and performance levels are compatible, and the required quality features (security, reliability, etc.) do not negate those capability functions.
3. **Accurate** : SRS precisely defines the system's capability in a real-world environment, as well as how it interfaces and interacts with it. This aspect of requirements is a significant problem area for many SRSs.
4. **Modifiable** : The logical, hierarchical structure of the SRS should facilitate any necessary modifications, (grouping related issues together and separating them from unrelated issues makes the SRS easier to modify).
5. **Ranked** : Individual requirements of an SRS are hierarchically arranged according to stability, security, perceived ease/difficulty of implementation, or other parameter that helps in the design of that and subsequent documents.
6. **Testable** : An SRS must be stated in such a manner that unambiguous assessment criteria (pass/fail or some quantitative measure) can be derived from the SRS itself.
7. **Traceable** : Each requirement in an SRS must be uniquely identified to a source (use case, government requirement, industry standard, etc.)
8. **Unambiguous** : SRS must contain requirements statements that can be interpreted in one way only. This is another area that creates significant problems for SRS development because of the use of natural language.
9. **Valid** : A valid SRS is one in which all parties and project participants can understand, analyze, accept, or approve it. This is one of the main reasons SRSs are written using natural language.
10. **Verifiable** : A verifiable SRS is consistent from one level of abstraction to another. Most attributes of a specification are subjective and a conclusive assessment of quality requires a technical review by domain experts. Using indicators of strength and weakness provide some evidence that preferred attributes are or are not present.

**Features of SRS:**

- Various features of SRS are given below :
  1. It forms the basis for software development.
  2. It provides a reference for validation of the final software product.
  3. It is a medium or media through the client and used needs are accurately specified determined.
  4. It helps to clients to understand their own needs and requirements.
  5. It establishes the basis for agreement between the client and the supplier.

## Case Studies for SRS

**Example 1:** Table 1.4 shows what a basic SRS outline might look like. This example is an adaptation and extension of the IEEE Standard 830-1998.

**Table 1.4: A Sample of a Basic SRS Outline**

1. **Introduction:**
   - 1.1 Purpose
   - 1.2 Document conventions
   - 1.3 Intended audience
   - 1.4 Additional information
   - 1.5 Contact information/SRS team members
   - 1.6 References
2. **Overall Description:**
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User classes and characteristics
   - 2.4 Operating environment
   - 2.5 User environment
   - 2.6 Design/implementation constraints
   - 2.7 Assumptions and dependencies
3. **External Interface Requirements:**
   - 3.1 User interfaces
   - 3.2 Hardware interfaces
   - 3.3 Software interfaces
   - 3.4 Communication protocols and interfaces
4. **System Features:**
   - 4.1 System feature A
     - 4.1.1 Description and priority
     - 4.1.2 Action/result
     - 4.1.3 Functional requirements
   - 4.2 System feature B

| |
|---|
| **5. Other Non-functional Requirements:**<br>5.1 Performance requirements<br>5.2 Safety requirements<br>5.3 Security requirements<br>5.4 Software quality attributes<br>5.5 Project documentation<br>5.6 User documentation<br>**6. Other Requirements:**<br>Appendix A: Terminology/Glossary/Definitions list<br>Appendix B: To be determined |

**Example 2:** Table 1.5 shows a more detailed software requirements specifications outline document shows the structure of an SRS template.

**Table 1.5: A Sample of a more detailed SRS outline**

| | |
|---|---|
| **1. Scope** | 1.1 ***Identification***: Identify the system and the software to which this document applies, including, as applicable, identification numbers, titles, abbreviations, version numbers, and release numbers.<br>1.2 ***System overview***: State the purpose of the system or subsystem to which this document applies.<br>1.3 ***Document overview***: Summarize the purpose and contents of this document. |
| **2. Referenced Documents** | 2.1 ***Project documents***: Identify the project management system documents here.<br>2.2 ***Other documents***<br>2.3 ***Precedence***<br>2.4 ***Source of documents*** |
| **3. Requirements** | This section shall be divided into paragraphs to specify the Computer Software Configuration Item (CSCI) requirements, that is, those characteristics of the CSCI that are conditions for its acceptance. CSCI requirements are software requirements generated to satisfy the system requirements allocated to this CSCI. Each requirement shall be assigned a project-unique identifier to support testing and traceability and shall be stated in such a way that an objective test can be defined for it. |

| | |
|---|---|
| | 3.1 *Required states and modes.*<br>3.2 *CSCI capability requirements.*<br>3.3 *CSCI external interface requirements.*<br>3.4 *CSCI internal interface requirements.*<br>3.5 *CSCI internal data requirements.*<br>3.6 *Adaptation requirements.*<br>3.7 *Safety requirements.*<br>3.8 *Security and privacy requirements.*<br>3.9 *CSCI environment requirements.*<br>3.10 *Computer resource requirements.*<br>3.11 *Software quality factors.*<br>3.12 *Design and implementation constraints.*<br>3.13 *Personnel requirements.*<br>3.14 *Training-related requirements.*<br>3.15 *Logistics-related requirements.*<br>3.16 *Other requirements.*<br>3.17 *Packaging requirements.*<br>3.18 *Precedence and criticality requirements.* |
| **4. Qualification Provisions** | To be determined. |
| **5. Requirements Traceability** | To be determined. |
| **6. Notes** | This section contains information of a general or explanatory nature that may be helpful, but is not mandatory.<br>6.1 ***Intended use.*** This Software Requirements specification shall contain a complete and detailed description of the intended use of the process.<br>6.2 ***Definitions used in this document.*** Insert here an alphabetic list of definitions and their source if different from the declared sources specified in the "Documentation standard."<br>6.3 ***Abbreviations used in this document.*** Insert here an alphabetic list of the abbreviations and acronyms if not identified in the declared sources specified in the "Documentation Standard."<br>6.4 ***Changes from previous issue.*** Will be "not applicable" for the initial issue. Revisions shall identify the method used to identify changes from the previous issue. |

## SUMMARY

- A system is an orderly grouping of interdependent components linked together according to a plan to achieve a specific objective.
- Structured Systems Analysis and Design Methodology (SSADM) is a set of standards for systems analysis and application design. It uses a formal methodical approach to the analysis and design of information systems. It was developed by Learmonth Burchett Management Systems (LBMS) and the Central Computer Telecommunications Agency (CCTA) in 1980-1981 as a standard for developing British database projects.
- System Development Life Cycle (SDLC) is a set of steps, which are used for building a system.
- Requirement is a feature that must be included in new system.
- There are two types of requirements: Functional requirements and non-functional requirements.
  - The requirements that defines the system's functionality and its associated components is known as functional requirements
  - The requirements that defines the system's quality attributes are known as non-functional requirements
- Requirement engineering is the process of defining, documenting and maintaining the requirements. It is the process of collecting the information about the services functions or tasks performed by the software system.
  - **Requirement Elicitation:** This phase is about the gaining the domain knowledge and associated requirement.
  - **Requirement Specification:** Formal software requirement model is produce in this phase of requirement engineering.
  - **Requirement Verification and Validation:** to avoid software failure proper software verification and validations must be done
  - **Requirements Management:** It includes the overall management of the requirements like analyzing, documenting, tracking, prioritizing, communicating with stakeholders and agreeing on them
- IEEE defines a requirement as "A condition that must be possessed by a system to satisfy a contract specification, standard or other formally imposed document".
- Software Requirements Specification (SRS) is a perfect detailed description of the behavior of the system to be developed.

## CHECK YOUR UNDERSTANDING

1. Testing objective is different for each phase of life cycle activity. True or false?
   (a) True (b) False
2. What is the first step in the SDLC?
   (a) Implementation
   (b) Testing
   (c) Coding
   (d) Preliminary Investigation and Analysis
3. Requirements analysis is critical to the success of a development project.
   (a) True (b) False
   (c) Depends upon the size of project (d) None of the mentioned
4. _____ and _____ are the two issues of Requirement Analysis.
   (a) Performance, Design (b) Stakeholder, Developer
   (c) Functional, Non-Functional (d) None of the mentioned
5. Which one of the following is not a step of requirement engineering?
   (a) Elicitation (b) Design
   (c) Analysis (d) Documentation
6. The user system requirements are the parts of which document ?
   (a) SDD (b) SRS
   (c) DDD (d) SRD
7. IEEE 830-1993 is a IEEE recommended standard for _____ .
   (a) Software Requirement Specification (b) Software design
   (c) Testing (d) Both (a) and (b)

**Answers**

| | | | |
|---|---|---|---|
| 1. (a) | 2. (d) | 3. (a) | 4. (b) |
| 5. (b) | 6. (b) | 7. (a) | |

## PRACTICE QUESTIONS

**I. Answer the following questions in short.**

1. What is software engineering?
2. What is SDLC? What are its phases?
3. What is SRS?
4. Explain the term Requirement Anticipation in detail.
5. What is Requirement Specification?

**Q.II. Answer the following questions.**

1. Explain SDLC with its limitations.
2. What do you mean by software requirements? How these requirements can be categorized into functional, non-functional and domain requirements?
3. Explain 4 phases of Requirement engineering.
4. With the help of diagram describe SRS format.
5. Explain functional and non-functional requirements in brief.
6. Compare functional and non-functional requirements.
7. Elaborate various methods of Requirement Elicitation.
8. Describe any two SDLC Models.

**Q.III. Write a short note on:**

1. Prototyping Model
2. Spiral Model
3. IEEE SRS Format

***

UNIT 2

# System Analysis and Modeling

## Objectives...

After reading this chapter, you will be able:

- To understand the basic concepts of system analysis and modelling.
- To study system's static aspect using Class diagram.
- To learn system's dynamic aspect using Use case diagram, Activity diagram, Interaction diagram, Package, Component and Deployment diagrams.

## 2.1 INTRODUCTION

- System analysis and modelling is a study of methods used to capture, analyse and model system requirements. Students will acquire practical skills through case study work utilizing techniques and software tools used by industry. The System Analysis and Modeling focuses on the use of models for software-intensive systems at development-time and run-time.
- System modeling may represent a system using graphical notation, e.g. the Unified Modeling Language (UML).

### 2.1.1 UML

- The UML is a standard graphical language for modeling object-oriented software. It can be defined as "UML is a language for visualizing, specifying, constructing and documenting artifacts of a software intensive system".
- This definition shows various goals of UML as:
  - **Visualizing** means UML has the standard diagramming notations for drawing or presenting pictures of software systems.
  - **Specifying** means building models that are exact, unambiguous and complete.
  - **Constructing** is related to actual implementation of design into coding.
  - **Documentation** plays a vital role in any type of system development, which helps during developing a system and after its deployment. UML addresses documentation of system architecture and other details.

- UML is not a programming language, but tools can be used to generate code in various languages using UML diagrams.
- UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. It was developed in the mid-1990s as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson.

**Reasons to use UML:**

- There are various reasons of use of UML. Some of them are listed below:

1. **Helps in Software construction model.** Software development is extremely complex, critical. Once a particular structure is in place, it is often lengthy and difficult job to change. UML helps in constructing a model.
2. **Appropriate for both new and legacy systems:** UML is appropriate for both new system developments and improvements to existing systems. Only the parts will be modelled those are affected by the change.
3. **Inherent traceability:** The Use Case Driven nature of modeling with UML ensures that all levels of model trace back to elements of the original functional requirements.
4. **Parallel development of large systems:** Large complex, critical UML models can be decomposed in a totally user-definable way so that different parts of the model can be developed independently by different people or groups.
5. **Visualize in multiple dimensions and levels:** UML allows software to be visualized in multiple dimension, so that a computer system can be completely understood before construction/development begin.
6. **Incremental development and re-development:** UML models respond well to an incremental development environment.
7. **Unified:** OMG (Object Management Group) an independent organization has made UML as the de-facto (actual) standard modeling language in the software industry.
8. **Documents software assets:** UML documentation removes the company from dependence on key personnel who may leave at any time and improves understanding of the company's critical business processes.
9. **Universal:** UML is a universal language because it can be applied in many areas of software development.

**Diagrams in UML:**

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs/paths (relationships).
- We draw diagrams to visualize a system from different perspective so a diagram is a projection into a system.
- UML diagrams are broadly categorized as Structural and Behavioral diagrams as shown in Fig. 2.1.

1. **Structural Diagrams:** A type of diagram that describes the elements of a specification those are irrespective of time. These diagrams show the things in a system being model.
2. **Behavioral Diagrams:** A type of diagram that describes behavioral features of a system or business process. These diagrams show what should happen in a system. They describe how the objects interact with each other to create a functioning system.

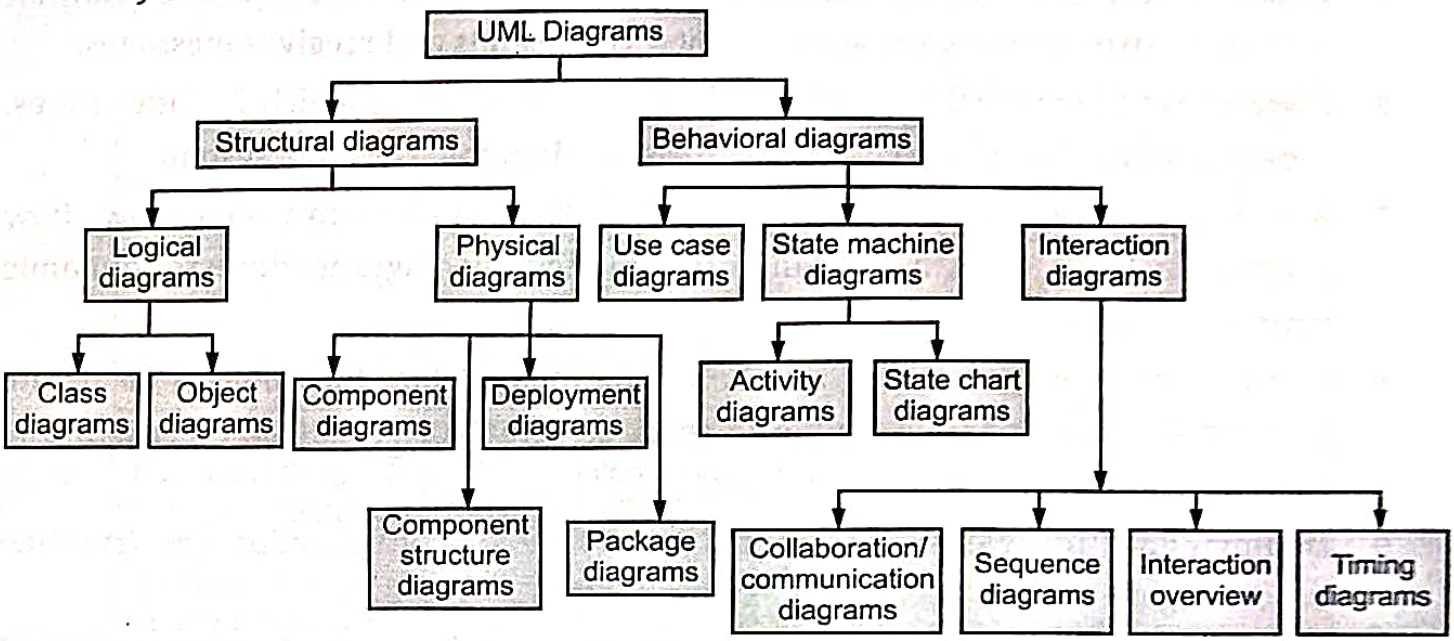



**Fig. 2.1: Classification of UML Diagrams**

- The structural diagrams are further categorized as:
  1. **Logical diagrams** represent the logical structure of the system.
  2. **Physical diagrams** represent the physical structure of the system.
- The behavioral diagrams can be categorized as Use Case diagrams, State Machine diagrams and Interaction diagrams.
- Interaction diagrams include Sequence and Collaboration diagrams, whereas State Machine diagrams include State chart and Activity diagrams.
- To view a system in different perspective, UML included different types of diagrams as follows:
  1. **Class Diagram:** This diagram shows a set of classes, interfaces and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs. A class diagram addresses static design view of a system.
  2. **Object Diagram:** An object diagram shows a set of objects and their relationships at a point in time. Object diagram represents static snapshots of instances of things found in class diagrams. An object diagram addresses static design view of a system.

3. **Use Case Diagram:** A use case diagram represents or shows a set of use cases and actors and their relationships. Use case diagrams address static view of a system.
4. **Interaction Diagram:** An interaction diagram shows an interaction of set of objects and their relationships along with messages that is shared among them. Interaction diagram address dynamic view of a system.
5. **Collaboration Diagram:** A collaboration diagram is an interaction diagram that focuses on structural organization of objects that sends and receives messages.
6. **State Chart Diagram:** This diagram shows a state machine which includes states, events, transitions and activities. It addresses a dynamic view of a system.
7. **Activity Diagram:** This is a special kind of state chart diagram which shows flow from one activity to another within a system. Activity diagram addresses dynamic view of a system.
8. **Component Structure Diagram:** These diagrams explore how objects cooperate to complete a task, document the internal structure of an object, or graphically describe a design structure or strategy.
9. **Deployment Diagrams:** These diagrams show configuration of runtime processing nodes and objects that exist on them.
10. **Package Diagrams:** These diagrams describe the high level organization of software packages. Package diagrams are used to divide the model into logical containers, or 'packages' and describe the interactions between them at a high level.
11. **Component Diagram:** Component diagram shows organizations and dependencies among set of components. Component diagram addresses static implementation view of a system.
12. **Timing Diagrams:** These diagrams combine sequence and state diagrams to provide a view of an object's state over time and messages which modify that state.
13. **Sequence Diagrams:** These diagrams show the time ordering of messages and object lifeline.

## Conceptual Model of UML:

- A conceptual model in UML can be defined as "a model which is made of concepts and their relationships".
- The UML vocabulary has three kinds of building blocks: Things/Elements of UML, Relationships, and Diagrams. Things are the basic elements in a model, while Relationship binds these things together, diagrams binds collection of things.

## 2.1.2 Things in UML / Elements of UML

- Things are the basic building blocks of the UML. We use them to write well formed models.
- Any type of diagram has structure and it also specifies behavior.
- In UML, there are four kinds of things i.e. Structural things, Behavioral things, Grouping things and Annotational things as shown in Fig. 2.2.

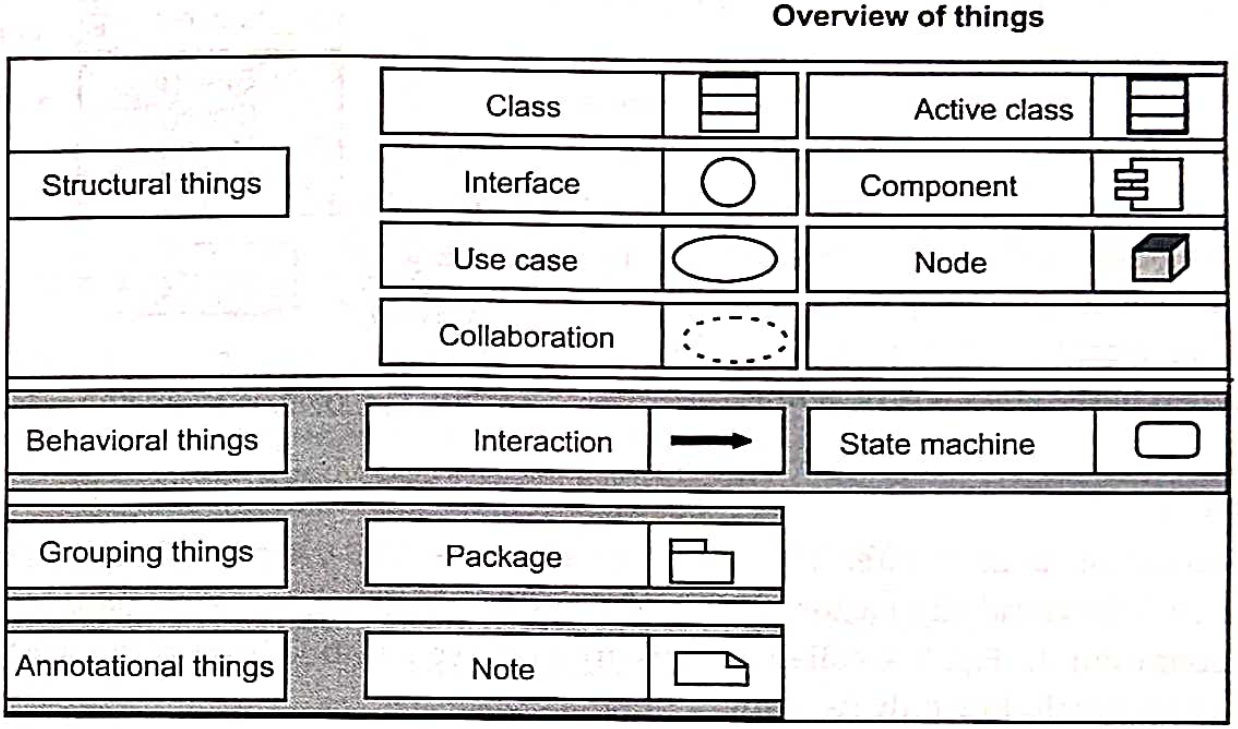



**Fig. 2.2: Things in UML**

**I. Structural Things:**

- Structural things represent a conceptual or physical element. These things are nouns of UML models. Structural things are the static parts of model.
- Some structural things are listed below:

**(i) Class:**

- A class is a description of a set of object that shares the same attributes, operation, relationship and semantics.
- A class implements one or more interfaces. For example, in Fig. 2.3, a class is represented with rectangle.
- The rectangle includes name, attributes and operations.

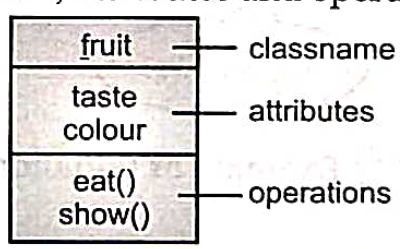


Fig. 2.3: Class

**(ii) Interface:**

- An interface is a collection of operations that specifies a service of a class o component. An interface therefore describes the externally visible behavior of tha element.
- An interface defines a set of operation specification (that is, their sign), but never set of operation implementation.
- The declaration of an interface looks like a class with the keyword <<interface>> above the name; attributes are not relevant, except sometimes to show constant.
- An interface provided by a class to the outside world is shown as a small circle attached to the class box by a line.

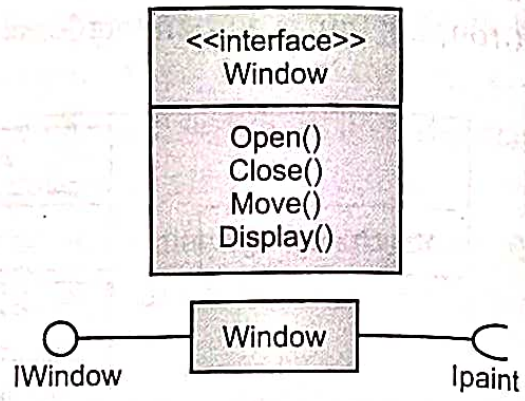



Fig. 2.4: Declaration of Interfaces

**(iii) Collaboration:**

- Collaboration defines interaction between elements. Collaboration has structural a well as behavioral dimensions. A given class or object might participate in severa collaboration. In Fig. 2.5, collaborations are displayed as an ellipse with dashed lines sometimes including only its name.


Fig. 2.5(a): Representation of Collaborations

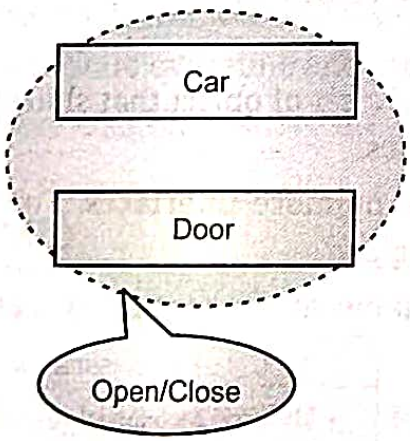



Fig. 2.5(b):Example of Collaborations

**(iv) Use Case:**

- Use case represents a set of actions performed by a system for a specific goal.
- A use case is a description of sequences of action that a system performs that yield observable results of value to a particular actor.
- A use case is used to structure the behavioral things in a model. It is realized by collaboration.
- Graphically, a use case is displayed as an ellipse with solid lines, usually including only its name, as shown in Fig. 2.6.


**Fig. 2.6: Use cases**

**(v) Active Class:**

- An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.
- An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.
- Graphically, it is represented as class only along with dark lines as shown in Fig. 2.7.

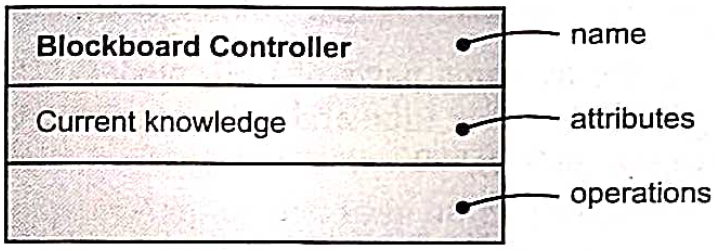



**Fig. 2.7: Representation of Active class**

**(vi) Component:**

- A component is a modular part of the system design that hides its implementation behind a set of external interfaces.
- Within a system, component sharing the same interfaces can be substituted while preserving the same logical behavior.
- The implementation of a component can be expressed by wiring together parts and connectors. The parts can include smaller components.
- Graphically, a component is shown at the upper right corner as shown in Fig. 2.8.

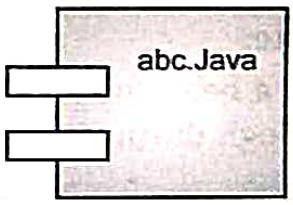



**Fig. 2.8: Component**

**(vii) Artifacts:**

- The artifacts represent physical things whereas the previous five things represent conceptual or logical things.

- An artifact is a physical and replaceable part of a system that contains physical information.
- You can see different kinds of deployment artifacts such as source code files, executables, script etc. in the system.
- Graphically, an artifact is as a rectangle with the keywords <<artifact>> above the name is shown in Fig. 2.9.


**Fig. 2.9: Artifact**

**(viii) Node:**

- A node can be defined as a physical element that exists at run time.
- A node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and often, processing capability.
- A set of component may reside on a node and may also migrate from node to node.
- Graphically, a node is shown as a cube usually including only its name, as in Fig. 2.10.

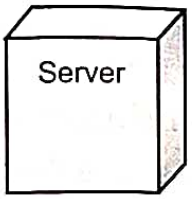



**Fig. 2.10: Node**

**2. Behavioral Things:**

- Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In UML, there are three kinds of behavioral things as given below:

**(i) Messages:**

- Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.
- An interaction involves a number of other elements, including messages, actions and connectors.
- Graphically, a message is displayed as a directed line, almost always including the name of its operation, as in Fig. 2.11.

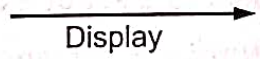



**Fig. 2.11: Messages**

**(ii) State:**

- It is a behavior that specifies the sequences of state, an object or an interaction goes through during its lifetime in response it events, together with its responses to those events.
- The behavior of an individual class or a collaboration of classes may be specified with a state machine.
- A state machine involves a number of other elements including states, transition.
- Graphically, a state is displayed as a rounded rectangle, usually including its name shown in Fig. 2.12.


**Fig. 2.12: States**

**(iii) Activity:**

- It is a behavior that specifies the sequences of steps a computational process perform.
- In an interaction, the focus is on the set of objects that interact whereas in a state machine, the focus is on the life cycle of one object at a time.
- In an activity, the focus is on the flows among steps without regard to which object performs each step. A step of an activity is called an action.
- Graphically, an action is displayed as a rounded rectangle with a name indicating its purpose. States and actions are distinguished by their different contexts.


**Fig. 2.13: Action**

**3. Grouping Things:**

- Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed.
- Grouping things can be defined as a mechanism to group elements of a UML model together.

**(i) Package:**

- Package is the only one grouping thing available for gathering structural and behavioral things.
- A package is a general purpose mechanism for organizing the design itself, as opposed to classes, which organize implementation constructs.
- Structural things, behavioral things and even other grouping things may be placed in a package.
- Graphically, a package is displayed as a tabbed folder, usually including only its name and sometimes, its contents, as in Fig. 2.14.


**Fig. 2.14: Packages**

**4. Annotational Things:**

- Annotational things are the explanatory parts of UML models.
- Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.
- *Note* is the only one Annotational thing available.

**(i) Note:**

- A note is used to show comments, constraints etc of an UML element.
- A note is simply a symbol for displaying constrains and comments attached to an element or a collection of elements.
- Graphically, a note is displayed as a rectangle with a dog eared corner, together with textual or graphical comments, as in Fig. 2.15.


**Fig. 2.15: Note**

## 2.1.3 Relationships in UML

- Relationship is another most important building block of UML.
- A relationship is a connection between things. When we model a system, we are supposed to not only identify the things that form vocabulary of system, but we should model how these things are related to one another.
- Relationship shows how elements are associated with each other and this association (linked) describes the functionality of an application.
- There are four kinds of relationships in the UML, i.e., Dependency, Association, Generalization and Realization. These relationships are explained below:

**1. Dependency:**

- Dependency is a semantic relationship. Dependency states that a change in specification of one thing (independent thing) may affect another thing (depending thing) that uses it, but not the reverse.
- Graphically, dependency relationship is represented with a dashed line with an arrow. (See Fig. 2.16 (a)).

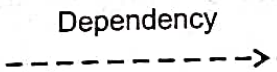



**Fig. 2.16(a)**

- For example, in the Fig. 2.16(b), the FlimClip class is depending on the Channel class. The dependency relationship is shown using the dashed or dotted line starting from FilmClip class and ending in the Channel class with an arrow pointing to it.

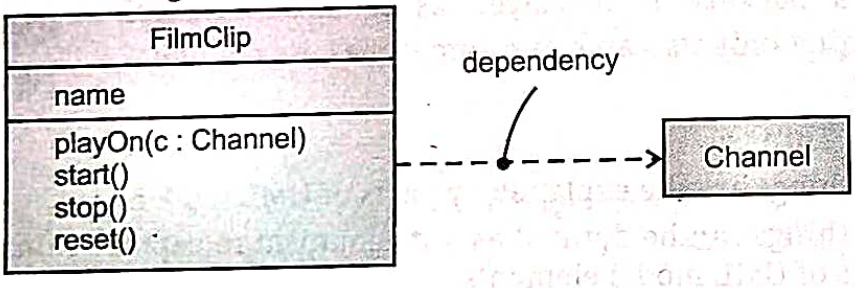



**Fig. 2.16(b): Dependency**

**2. Association:**

- Association is a structural relationship in which objects are dually dependent.
- Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.
- Graphically, association is represented as a solid line. For example, Fig. 2.17 shows linking of Airplane and Passengers.

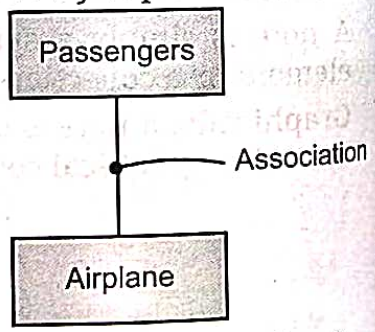



**Fig. 2.17: Dependency**

**There are two types of Association:**

**1) Aggregation:**

- It refers to the formation of a particular class as a result of one class being aggregated or built as a collection.
- In aggregation, the contained classes are not strongly dependent on the life cycle of the container. To show aggregation graphically in a diagram, draw a line from the parent class to the child class with a diamond shape near the parent class.
- For example, the class "Library" is made up of one or more books, among other materials. In the example, Books will remain so even when the Library is destroyed.

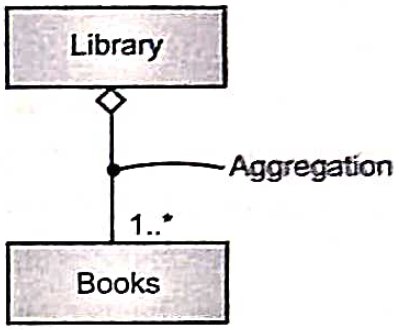



**Fig. 2.18: Aggregation**

**2) Composition:**

- It is very similar to the aggregation relationship, with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class.
- That is, the contained class will be destroyed when the container class is destroyed. For example, House (parent) and Room (child). Room doesn't exist separate from a House.
- To describe a composition relationship in a UML diagram, use a directional line connecting the two classes, with a filled diamond shape (See Fig. 2.19), adjacent to the container class and the directional arrow to the contained class.

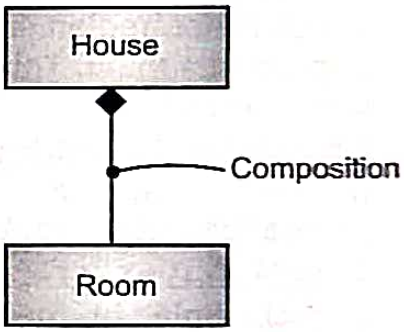



**Fig. 2.19: Composition Relationship**

**3) Generalization:**

- A generalization is a relationship between a general thing i.e. super class or parent class and a specific thing i.e. subclass or child class. It is similar to the concept of inheritance in OOP.
- Generalization can be defined as "a relationship which connects a specialized element with a generalized element."
- It basically describes inheritance relationship in the world of objects.
- It is known as an "is a" relationship since the child class is a type of the parent class.
- Generalization is the ideal type of relationship that is used to showcase reusable elements in the class diagram. Literally, the child classes "inherit" the common functionality defined in the parent class.

- Graphically, generalization relationship is shown as a solid line with a hollo arrowhead pointing towards parent as shown in Fig. 2.20(a).


**Fig. 2.20(a): Representation of Generalization**

- **Example:**

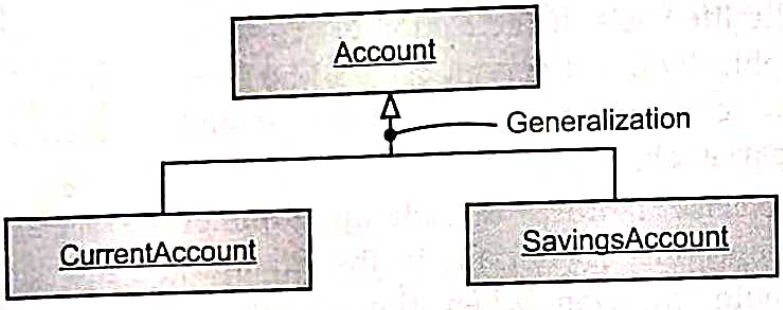



**Fig. 2.20(b): Example of Generalization**

**4. Realization:**

- It is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.
- Realization can be defined as "a relationship in which two elements are connected".
- One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.
- Graphically, a realization relationship is displayed as a cross between a generalization and a dependency relationship, as in Fig. 2.21(a).
- For example, in the Fig. 2.21(b), printing preferences that are set using the printer setup interface are being implemented by the printer.

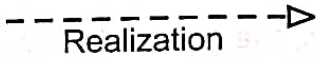



**Fig. 2.21(a): Representation of Realization**

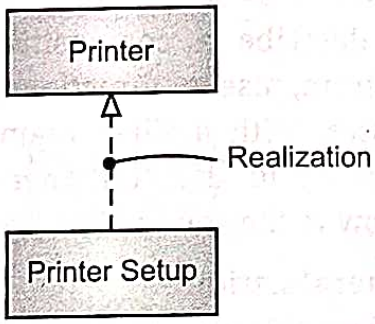



**Fig. 2.21(b): Example of Realization**

**Other Important Term used in UML:**

**Multiplicity:**

- It is the active logical association when the cardinality of a class in relation to another is being depicted. It is also called as cardinality.
- For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers. The notation 0..* in the diagram means "zero to many", (See Fig. 2.22).

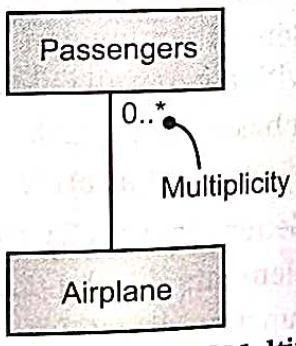



**Fig. 2.22: Example of Multiplici**

## 2.1.4 Rules of UML

- The UML has syntactic and semantic rules for:
  1. **Names** : What you can call things, relationship and diagrams?
  2. **Scope** : The context that gives specific meaning to a name.
  3. **Visibility** : How those names can be seen and used by others?
  4. **Integrity** : How things properly and consistently related to one another?
  5. **Execution** : What it means to run or simulate a dynamic model?
- Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For these reasons, it is common for the development team not only to build models that are well-formed, but also to build models that are:
  1. Elided : Certain elements are hidden to simplify the view.
  2. Incomplete : Certain elements may be missing.
  3. Inconsistent : The integrity of the model is not sure.

## 2.1.5 UML Architecture/Architecture of Software System

- Any real world system is used by different users like developers, testers, business people, analysts and many more. So before designing a system the architecture is made with different perspectives in mind.
- UML plays an important role in defining different perspectives of a system. These perspectives are Logical Design, Implementation, Process and Deployment and Use Case View.

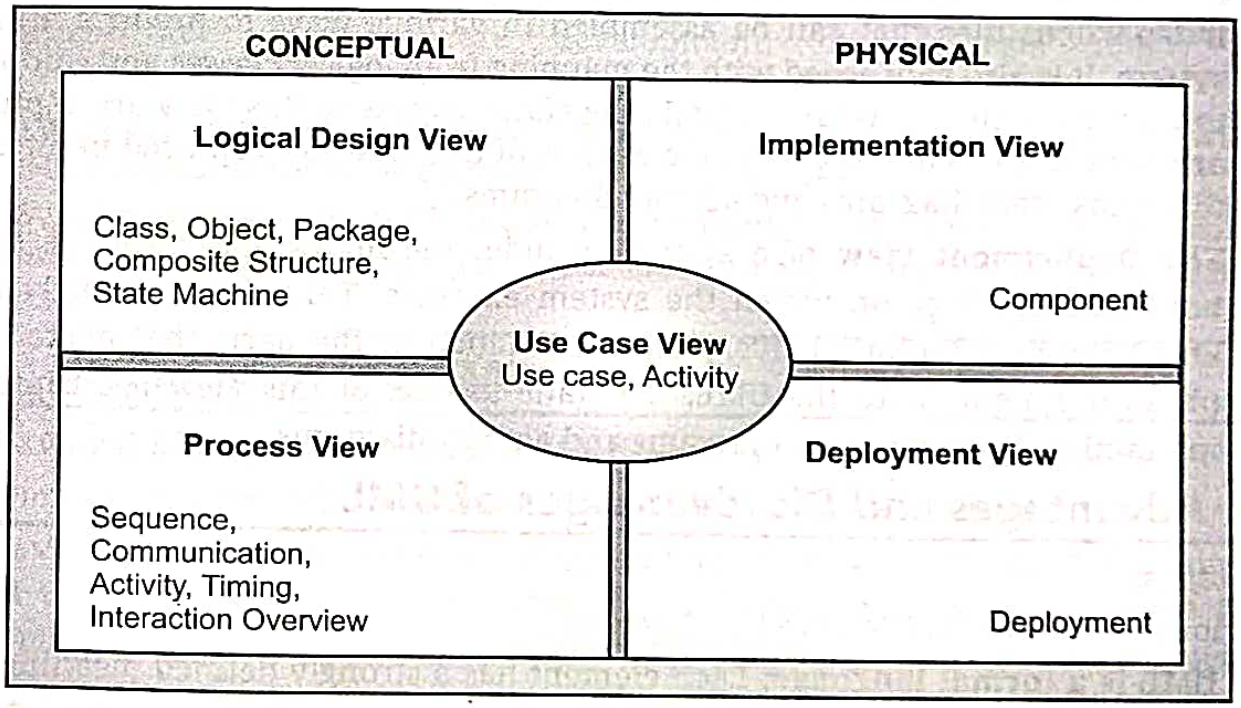



Fig. 2.23: UML Architecture

1. **The Use Case View** of a system includes the use cases that describe the behavior of the system as seen by its end users, analyst and testers. This view does not specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture.

   With the UML, the static aspects of this view are captured in use case diagrams. The dynamic aspects of this view are captured in interaction diagrams, state diagrams and activity diagrams.
2. **The Logical Design view** of a system includes the classes, interfaces and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams, the dynamic aspects of this view are captured in interaction diagrams, state diagrams and activity diagrams. The internal structure diagram of a class is mainly useful.
3. **The Process/Interaction view** of a system shows the flow of control among its various parts, including possible concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that control the system and the messages that flow between them.
4. **The Implementation view** of a system includes the artifacts that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent files that can be assembled in various ways to produce a running system. It is also concerned with the mapping from logical classes and components to physical artifacts. With the UML, the static aspects of this view are captured in artifact diagrams and the dynamic aspects of this view are captured in interaction diagrams, state diagrams and activity diagrams.
5. **The Deployment view** of a system includes the nodes that form the system's hardware topology on which the system executes. This view primarily reports about the distribution, delivery and installation of the parts that makes up the physical system. With the UML, the static aspects of this view are captured in interaction diagrams, state diagrams and activity diagrams.

## 2.1.6 Advantages and Disadvantages of UML

**Advantages:**

- Various advantages of UML are listed below:
  1. **UML is a formal language:** Each element has a strongly defined meaning. When we create a model of a system, we can be assured that the designed model is understandable to all.

2. **UML is concise:** The entire language is made up of simple and straightforward notation.
3. **It is comprehensive:** It describes all important aspects of a system.
4. **It is scalable:** Wherever needed, the language is formal enough to handle massive system modeling projects, but it also scales low to small projects, avoiding overload.
5. **It is built on lessons learned:** UML is the result of best practices in object-oriented community over the past few years.
6. **It is standard:** UML is controlled by open standard group with active contributions from a worldwide group of vendors and academics.
7. **Most-used:** It is a most useful method of visualization and documenting software system design.
8. **It is independent:** It uses object-oriented design concept and it is independent of programming language.

**Disadvantages of UML:**

- Various disadvantages of UML are listed below:

1. **Lack of formality:** UML has still no structure and specification for modeling user interfaces. UML does not define a standard file format.
2. **Time required managing and maintaining diagrams:** Some developers might find a disadvantage when using UML is the time it takes to manage and maintain UML diagrams. To work properly, UML diagrams must be synchronized with the software code, which requires time to set up and maintain, and adds work to a software development project.
3. **Diagrams can get more complicated:** When creating a UML diagram in conjunction with software development, the diagram might become overcomplicated. This can be confusing and frustrating for developers.
4. **Too much emphasis on design:** UML places much emphasis on design, which can be problematic for some developers and companies.
5. **Synchronizing code with models is difficult:** Using multiple models/diagrams makes it difficult to keep them consistent with each other and much code has to be added by hand.

## 2.2 USE CASE DIAGRAMS

- **Behavioral Diagram:** Behavioral model describes the interaction in the system. In short, it represents the interaction among the structural diagrams.
- Behavior modeling represents the simplest type of logical abstraction that aligns well with the way the human brain processes information, enabling someone to grasp complicated systems and, in turn, enhance their capacity to tackle challenging issues.
- Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

## 2.2.1 Concept of Use Case Diagram

- The use case diagram is a behavioral diagram in the Unified Modeling Language (UML).
- A use case diagram at its simplest is a representation of a user's interaction with the system and depicting the specifications of a use case.
- It can represent the different types of users of a system and the various ways that they interact with the system.
- A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams a name and graphical contents that are a projection into a model.

**Purpose of Use Case Diagram:**

1. Used to gather requirements of a system.
2. Used to get an outside view of a system.
3. Identify external and internal factors influencing the system.
4. Show the interacting among the requirements are actors.

- Use case diagrams commonly contain subject, use cases, actors, dependency, generalization and association relationships and so on.

**Use Cases:**

- A use case specifies the behavior of a system or a part of a system.
- Graphically, a use case is displayed as an ellipse .
- Every interesting system interacts with human or automated actors that use the system for some purpose and those actors expect that system to behave in predictable ways.
- A use case represents a functional requirement of the system.
- An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. An actor can be human or they can be automated systems.
- Use case can be applied to whole system. It can be applied to part of a system including subsystem and even individual classes and interfaces.

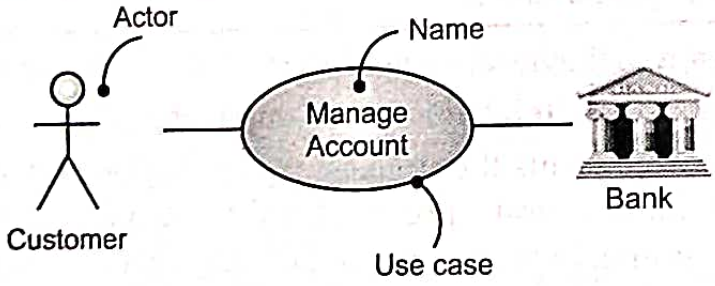



**Fig. 2.24: Use-case Diagram for Manage Account with Class Customer and Bank**

- Every use case must have a name that distinguishes it from other use cases.

- A name is a textual string. That name alone is known as a simple name. A qualified name is the use case name prefixed by the name of the package in which that use case lives.
- A use case is typically drawn showing only its name, as in Fig. 2.25.

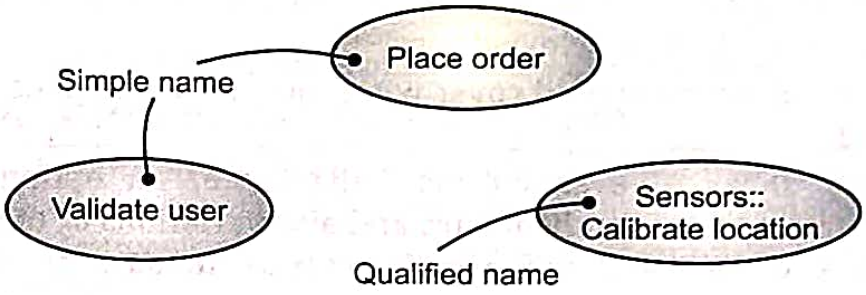



**Fig. 2.25: Examples of Use Cases**

## 2.2.1.1 Identify Actors

- An actor is someone or something that must interact with the system under development.
- An actor represents a role that a human, a hardware device or even another system plays with a system. For example, in Fig. 2.26, Loan Officer is an Actor for a bank and even a Customer is also Actor.

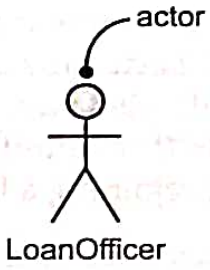



**Fig. 2.26: Actor for a Class bank**

- There are two types of actors are as follows:
  1. **Primary Actor:** System delivers services after getting call from actor.
  2. **Secondary Actor:** System needs help of actors for performing services called by primary actor.
- Actors are used in models which are not actually part of the system but they live outside the system. We can define general kind of actors (like Customer) and specialize them (such as Commercial Customer) using generalization relationships. Actors may be connected to use cases only by association.
- An association between an actor and a use case indicates that actor and the use case communicate with one another by sending and receiving messages as shown in Fig. 2.27.

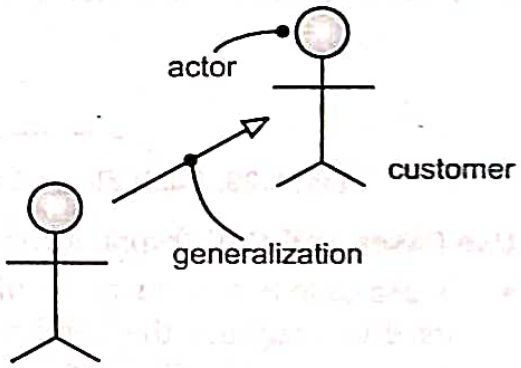



**Fig. 2.27: Association between an actor and a use case**

## 2.2.1.2 Identify Use Cases

- A use case describes what a system, subsystem, class or interface does but it does not specify how it does it.

- Flow of events is a step-by-step description of the interactions between the actor(s) and the system, and the functions that must be performed in the specified sequence to achieve a user goal.
- The most important part of a use case for generating test cases is the flow of events. The two main parts of the flow of events are:
  1. The basic flow of events should cover what "normally" happens when the use case is performed.
  2. The alternate flows of events cover behavior of an optional or exceptional character relative to normal behavior, and also variations of the normal behavior. You can think of the alternate flows of events as "detours" from the basic flow of events.
- In Fig. 2.28, each of the different paths through a use case reflecting the basic and alternate flows are represented with the arrows. The basic flow represented by the straight back-line is the simplest path through the use case. Each alternate flow begins with the basic flow and then dependent upon a specific condition, the alternate flow is executed. Alternate flows may rejoin the basic flow (alternate flows 1 and 3), may originate from another alternate flow (alternate flow 2), or may terminate the use case without rejoining a flow (alternate flows 2 and 4).

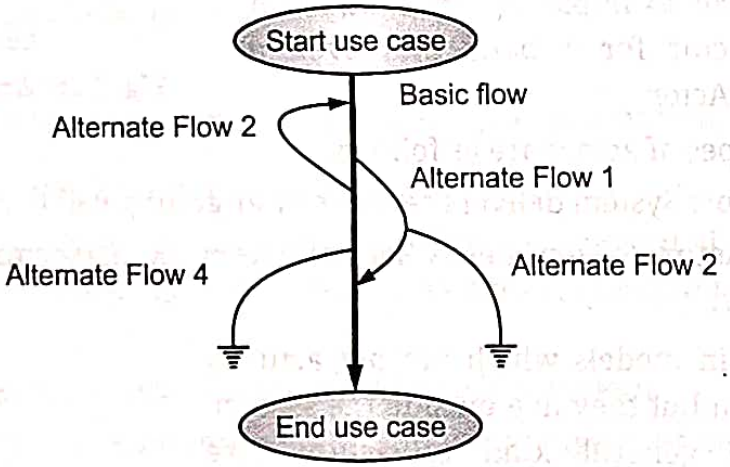



**Fig. 2.28: Basic flow of events and alternate flows of events for a Use Case**

**Use Cases and Collaborations:**

- A use case is a description of sequences of actions that a system performs. A use case is used to structure the behavioral things in a model.
- A use case captures the intended behavior of the system i.e. class, subsystem or interface developing without having to specify how that behavior is implemented.
- Use cases can be implemented by creating a society of classes and other elements that work together to implement the behavior of use case.
- Collaborations are used to implement the behaviour of use cases with society of classes and other elements that work together. It includes static and dynamic structures.
- Realization of a use case can be specified by collaboration, (See Fig. 2.29).

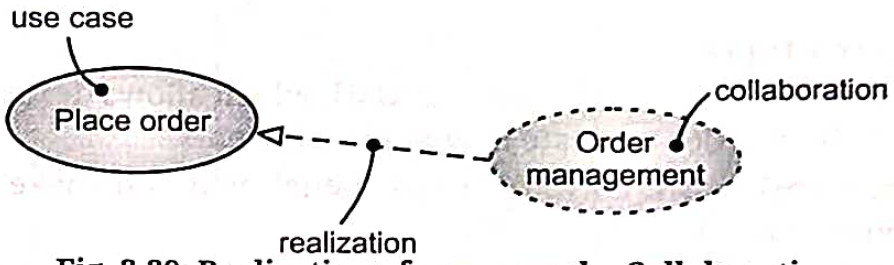



**Fig. 2.29: Realization of a use case by Collaboration**

## 2.2.1.3 Organization of Use case Diagram

- Use cases can be organized by grouping them in packages like classes.
- By using generalization *include* and *extend* relationships, we can organize use cases.
- It is a relationship between actors to support re-use of common properties.

1. **Include Relationship:**

- An *include* relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- Include relationship is shown by *include* keyword having open and close triangles i.e. <<include>>.
- The relationship between a base use case and an included use case (See Fig. 2.30) is shown using a dashed line connecting the base use case to the included use case, with an open arrowhead pointing at the included use case. The line is labeled with the «include» stereotype.

2. **Extend Relationship:**

- An *extend* relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.
- Extend relationship is shown by *extend* keyword having open and close triangles i.e. <<extend>>.
- The relationship between a base use case and a use case that extends it (See Fig. 2.30) is shown using a dashed line connecting the extended use case to the base use case, with an open arrowhead pointing at the base use case. The line is labeled with the «extend» stereotype.

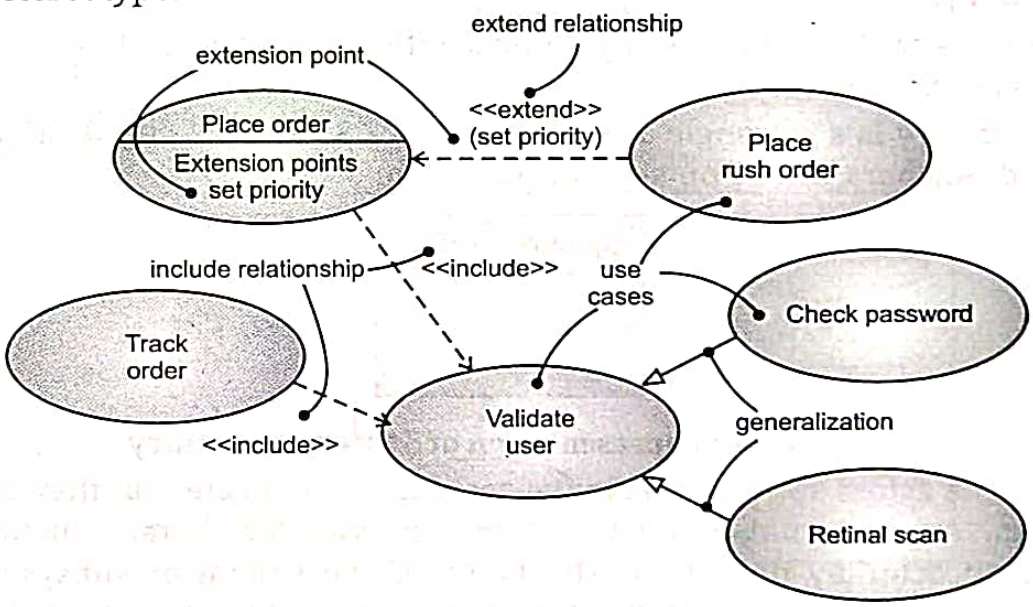



**Fig. 2.30: Relationship between a Base Use Case and a Use Case**

## Use Cases with Stereo Types:

- Stereotypes are extensibility mechanisms in UML which allows designers to extend the vocabulary of UML in order to create new model elements.
- By applying appropriate stereotypes in the model, you can make the specification model comprehensible.
- Graphically, a stereotype is represented by a name enclosed by guillemets << >>.
- As shown in the Fig. 2.31, the stereotype External User is applied to a model element (i.e. actor) called Customer.

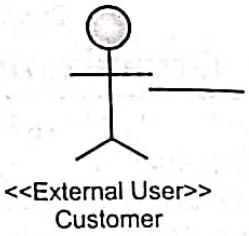



Fig. 2.31: Stereotype External User

**Example:**

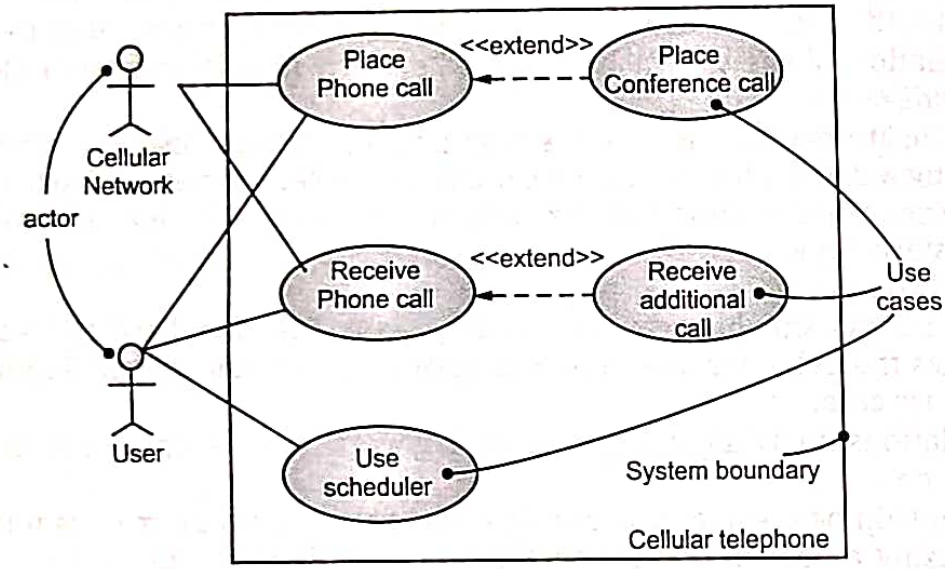



Fig. 2.32: Use case diagram for Cellular telephone

## System Boundary:

- System boundary shows how user interacts with the system. System is class in whic use cases are executed.
- System boundary is a rectangle as shown in Fig. 2.33 and inside this all use cases wi be inserted. Anchors will be outside this rectangle.

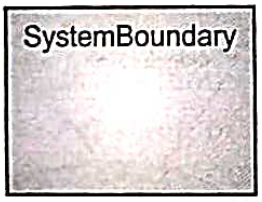



Fig. 2.33 (a): Representation of System boundary

- **Example:** The actors appear outside the rectangle to indicate that they are not part o the system. Any external entity that interacts with the system, including anothe system is an actor by definition. The name of the system or subsystem is usuall shown inside the rectangle at the top, preceded by the stereotype <<system>> o <<subsystem>>.

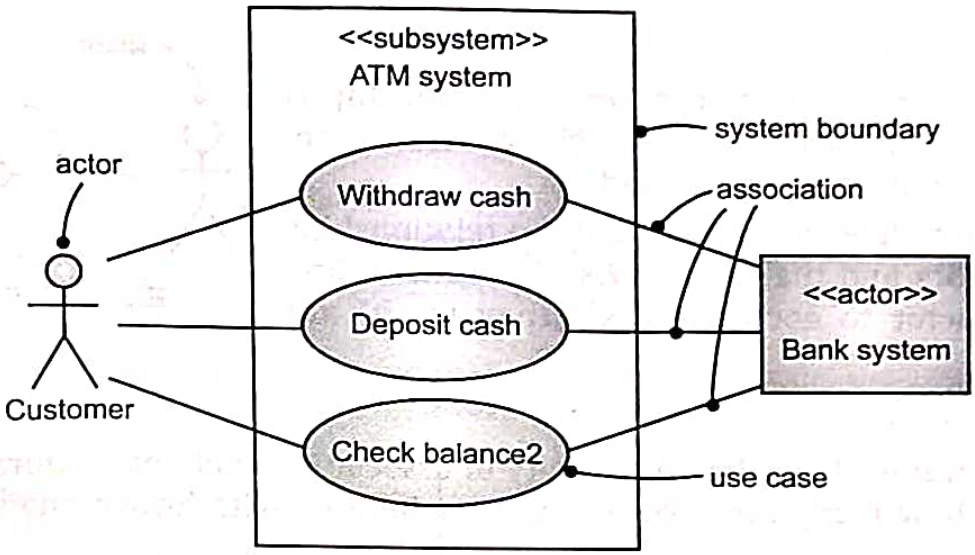



**Fig. 2.33 (b): Example of System Boundary**

## Communication Lines:

- Use case diagrams have communication lines or link for communication between its various components.
- Use cases and actors have a certain relationship with one another. Relationships are modeled with lines.
- An actor is linked to use cases using simple association. This indicates an interaction with the system belonging to the use case, and in the context of that use case. Fig. 2.34 shows various communication lines in use case diagram.
- Communications represents when one use case communicates information with another. Association displayed as a solid line without an arrowhead. Association identifies an interaction between actors and use cases.
- Generalizations are represented by a solid line with a solid arrowhead. It defines a relationship between two actors or two use cases.
- Dependencies represented by dotted line with arrowhead. Dependency identifies a communication relationship between two use cases.

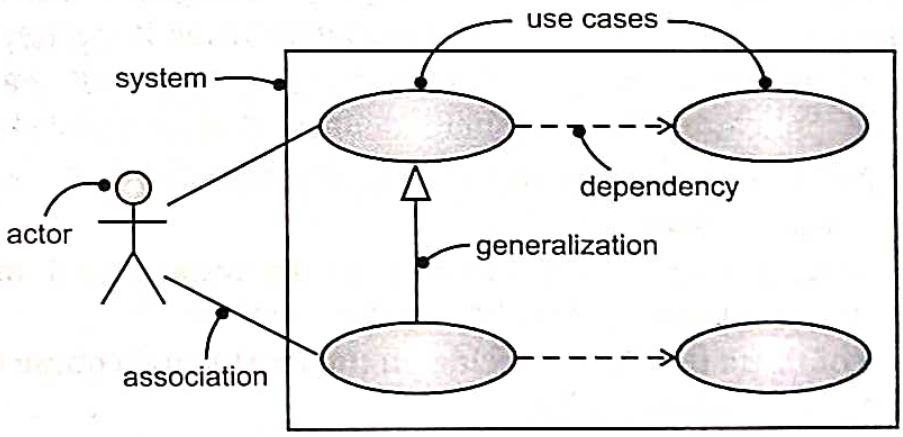



**Fig. 2.34: Relationship of Communication**

**Relationships:**

- In use case diagram, various types of relationship is used. These relationship types include association, generalization, include and extend.
- UML has a simple way of indicating the relationships between actors and their use cases. You draw a line from each actor to each use cases they (or it) are involved in.

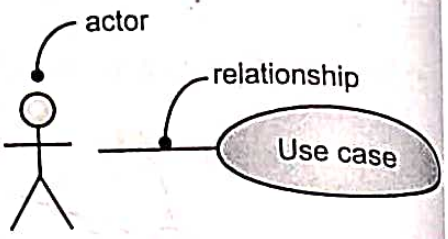



**Fig. 2.35: Relationship**

**Multiplicity:**

- It can be useful to show the multiplicity of associations between actors and use cases. This means how many instances of an actor interact with how many instances of the use case.
- By default, we assume one instance of an actor interacts with one instance of a use case. In other cases, we can label the multiplicity of one end of the association, either with a number to indicate how many instances are involved, or with a range separated by two periods (..). An asterisk (*) is used to indicate an arbitrary number.
- In Fig. 2.36, you can clearly see that an instance of the *Withdraw Cash* use case may be initiated by one instance, and only one instance, of the Customer actor. Conversely, the Customer actor may initiate zero or one instances of the Withdraw Cash use case. Each end of an association may display Multiplicity.

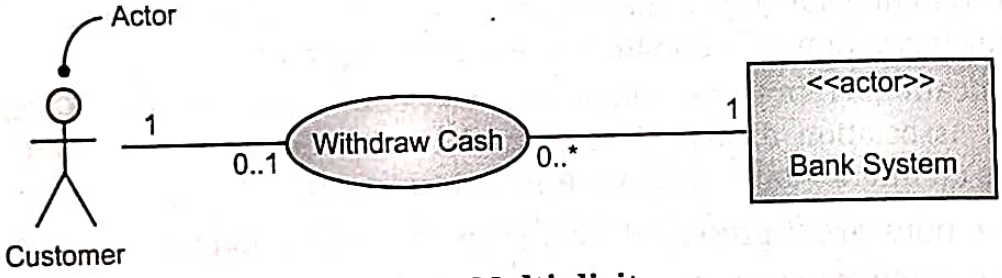



**Fig. 2.36: Multiplicity**

## 2.2.1.4 Develop Use-Case Model

- To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running or operating.
- When developing use cases, you should start with a listing of the major functional categories of the application. This will help identify what areas need to be focused on.

**Step 1:** Identify who is going to be using the system directly. These are the Actors.

**Step 2:** Pick one of those Actors.

**Step 3:** Define what that Actor wants to do with the system. Each of these things that the actor wants to do with the system becomes a Use Case.

**Step 4:** For each of those Use Cases, decide on the most usual course when that Actor is using the system. What normally happens?

**Step 5:** Describe that basic course in the description for the use case.

**Step 6:** Now consider the alternatives and add those as extending use cases.

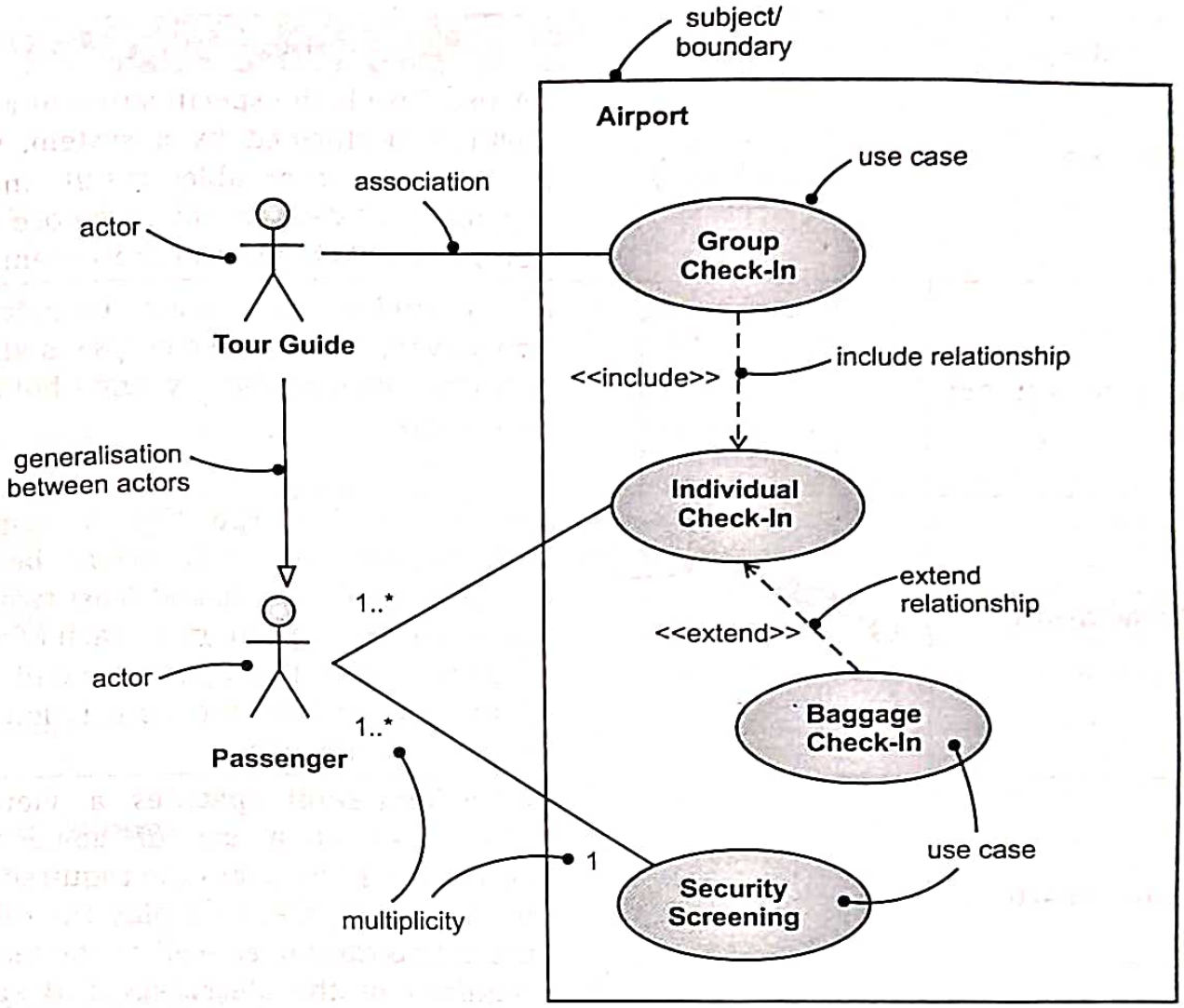
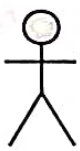


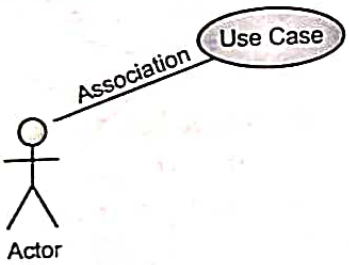
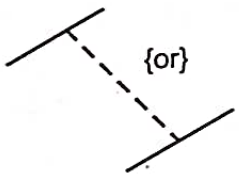
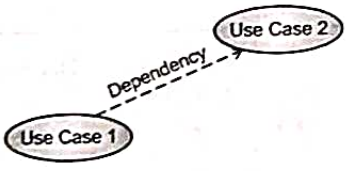
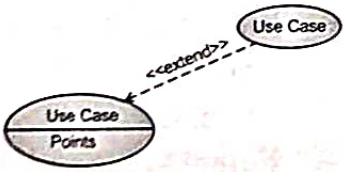

**Fig. 2.37: Use case diagram to model the behavior of Airport Entry System**

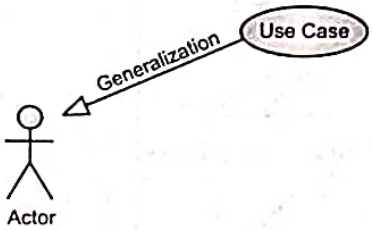
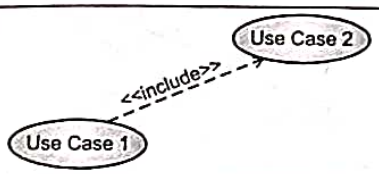
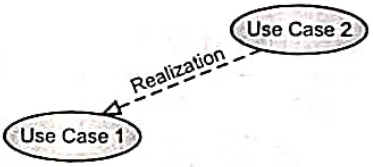
- Once the use cases are complete, it is important to discuss with the customer and make any necessary changes. It is important to have user buy in before proceeding to the next step in the process because these use cases will become the foundation of your user requirements and design specifications.

## 2.2.1.5 Notations

**Table 2.1: Basic Notations of Use Case Diagrams**

| Name | Symbols | Description |
|---|---|---|
| 1. Actor |  Actor | An Actor models a type of role played by an entity that interacts with the subject, but which is external to the subject. Actors may represent roles played by human users, external hardware, or other subjects. |

| Name | Symbols | Description |
|---|---|---|
| 2. Use Case |   | A Use Case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system. |
| 3. System/ Subject |   | If a subject or system boundary is displayed, the use case ellipse is visually located inside the system boundary rectangle. |
| 4. Association |   | An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type. |
| 5. Collaboration |   | A collaboration specifies a view (or projection) of a set of co-operating classifiers. It describes the required links between instances that play the roles of the collaboration, as well as the features required of the classifiers that specify the participating instances. |
| 6. Constraint |   | Constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. |
| 7. Dependency |   | A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. |
| 8. Extend |   | This relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. |

| Name | Symbols | Description |
|---|---|---|
| 9. Generalization |  | A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier. |
| 10. Include |  | It is a directed relationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. |
| 11. Note |  | A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler. |
| 12. Realization |  | Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client). |

## 2.2.1.6 Examples

1. **Use Case Diagram for ATM System:**

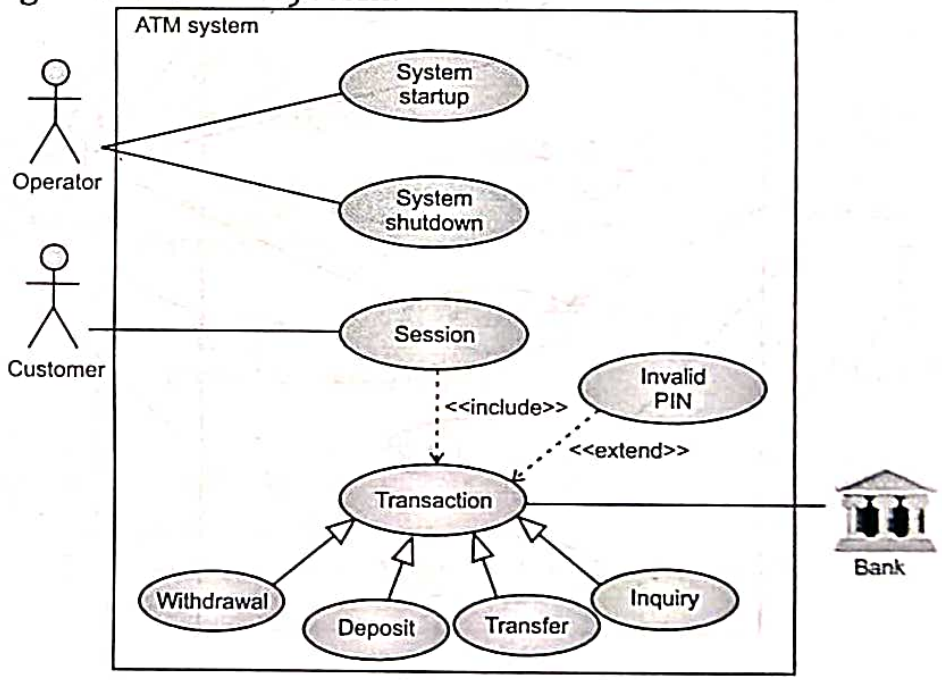



Fig. 2.38

## 2. Use Case Diagram for Online Shopping (Credit Cards Processing):

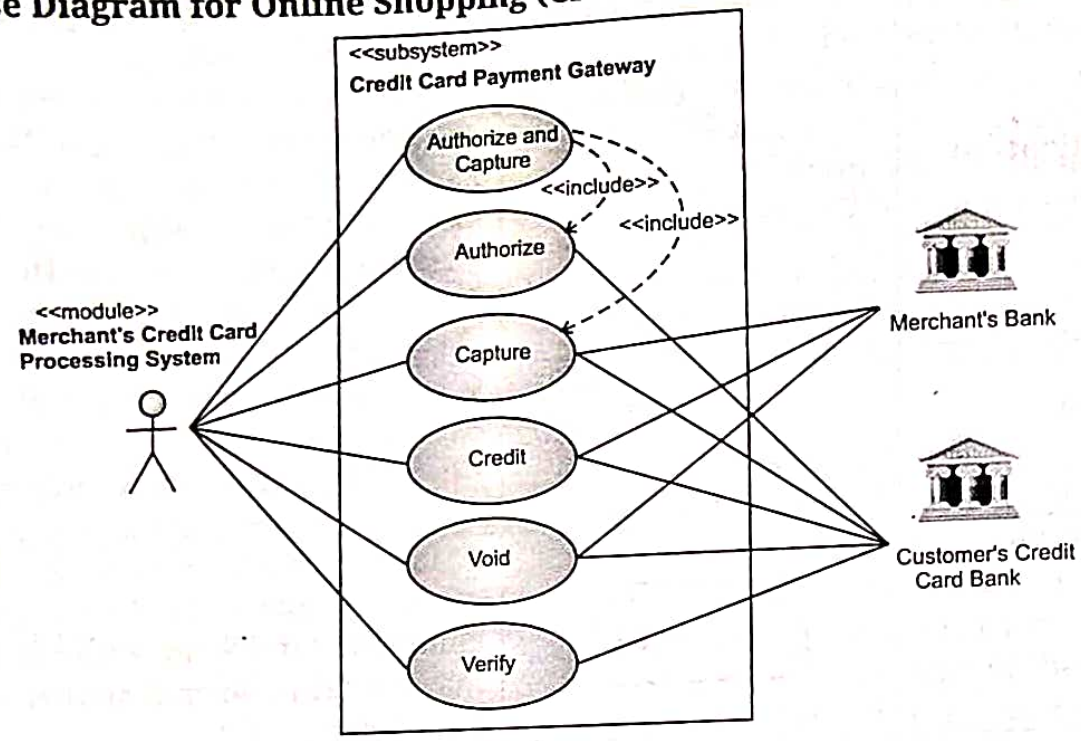



Fig. 2.39

## 3. Use Case Diagram for Library Management System:

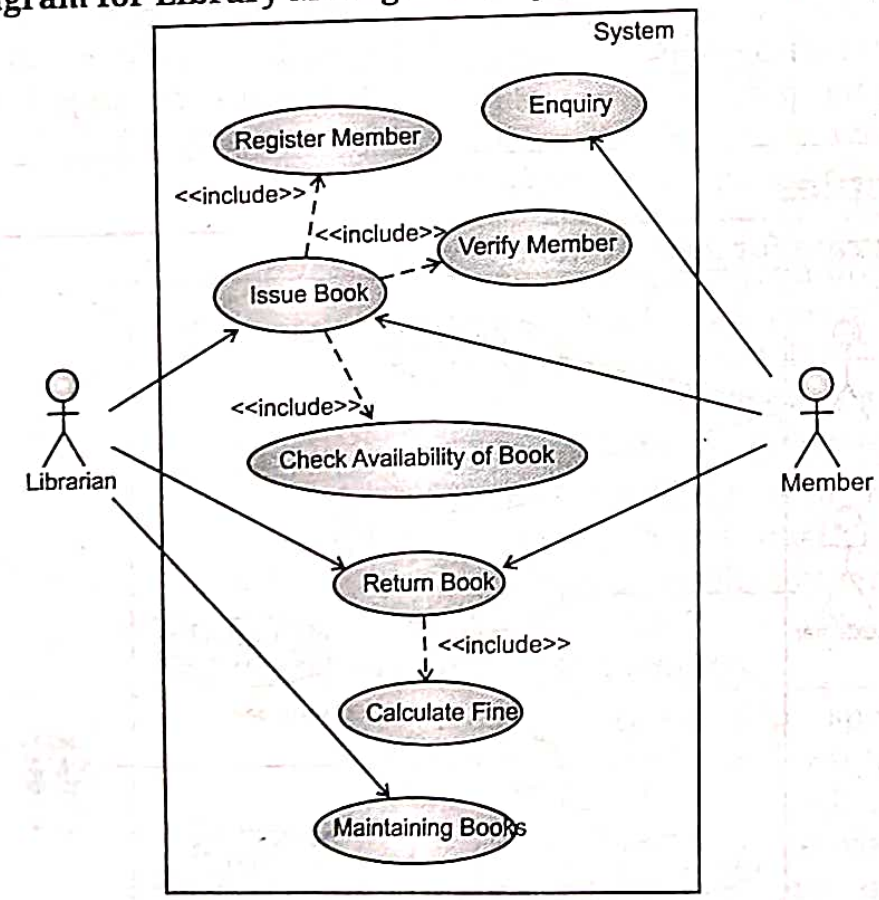



Fig. 2.40

## 4. Use Case Diagram for Railway Reservation System:

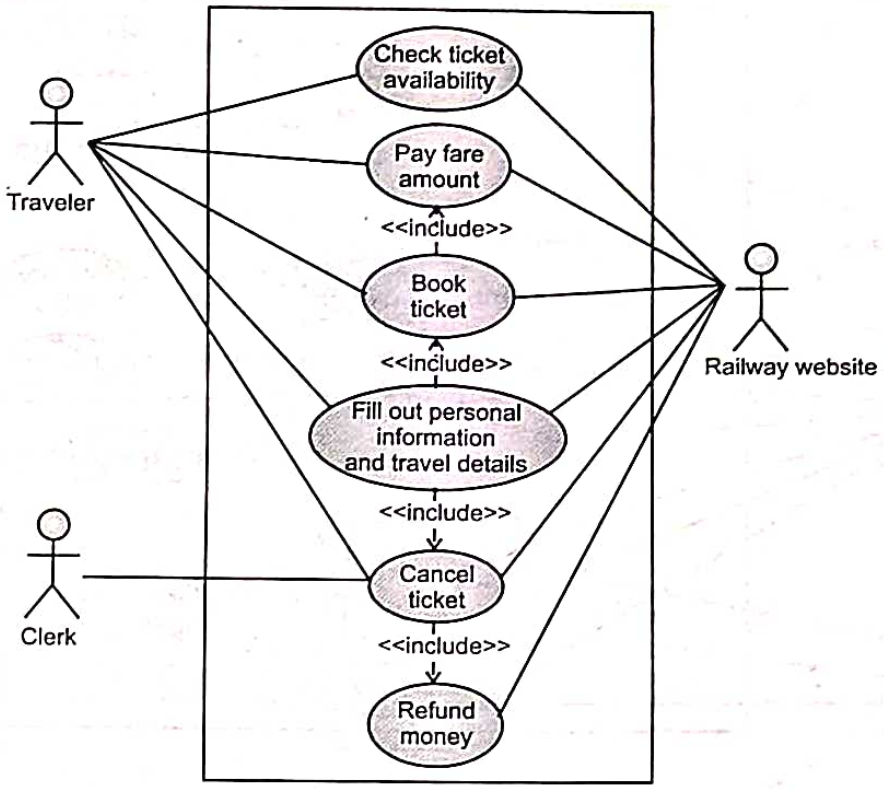



Fig. 2.41

## 5. Use Case diagram for Online Airline Reservation System:

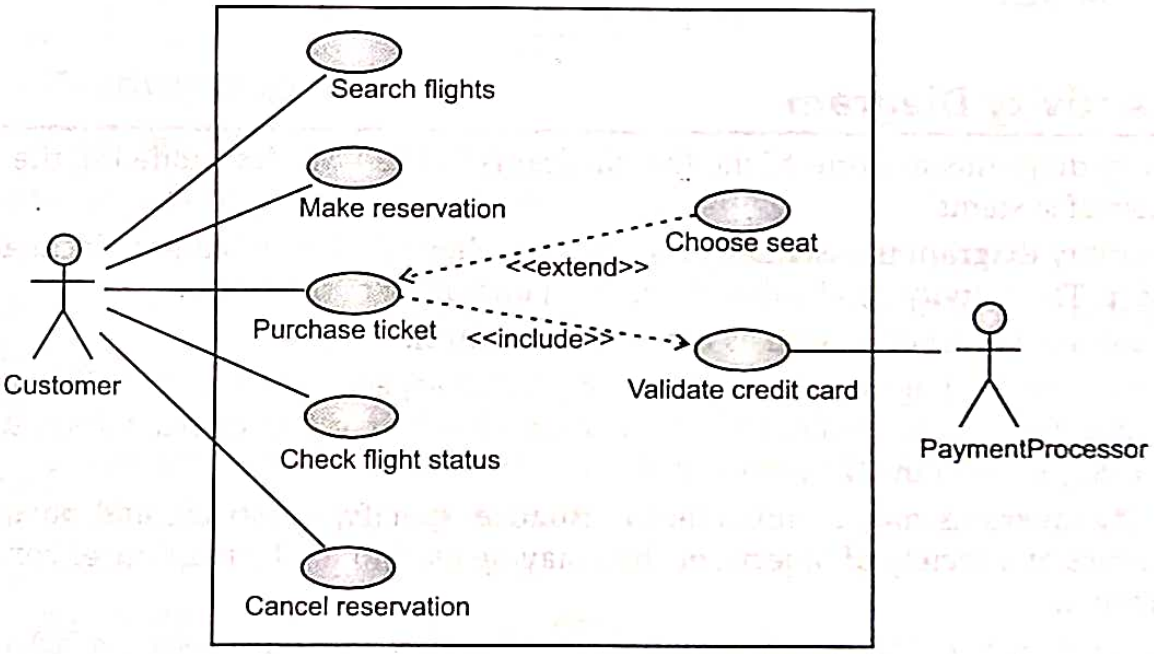



Fig. 2.42

### 6. Use Case Diagram for Hospital Management:

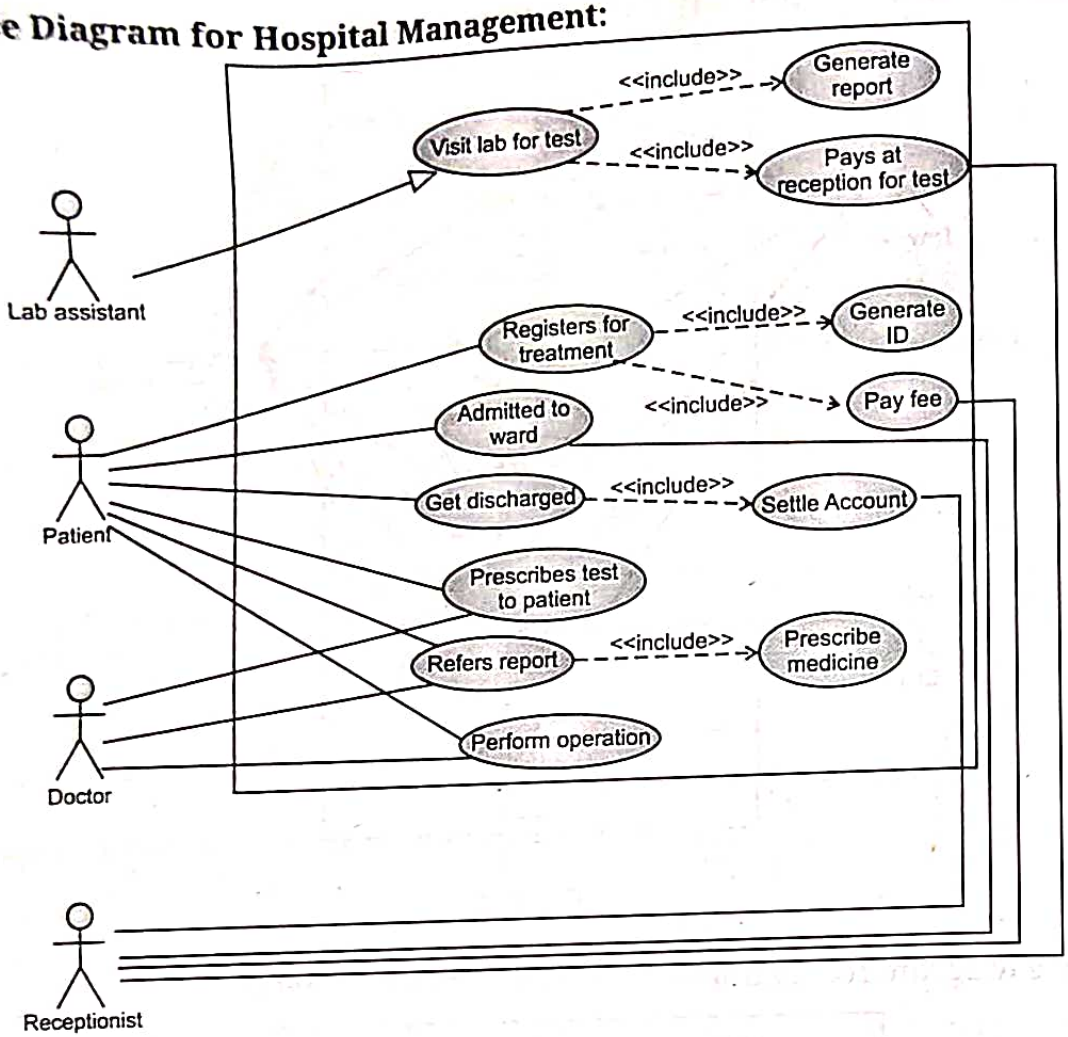



Fig. 2.43

## 2.2.2 Activity Diagram

- Activity diagrams are one of the five diagrams in the UML for modeling the dynami aspects of systems.
- An activity diagram is essentially a flowchart, showing flow of control from activity t activity. The activity can be described as an operation of the system.
- It shows concurrency as well as branches of control.
- We use activity diagrams to model the dynamic aspects of a system. With an activit diagram, we can also model the flow of an object as it moves from state to state a different points in the flow of control.
- Activity diagrams may standalone to visualize, specify, construct, and document th dynamics of a society of objects, or they may be used to model the flow of control of a operation.
- Activity diagrams consist of activities, states and transitions between activities an states. Activities ultimately result in some action, which is made up of executabl atomic computations that result in a change in state of the system or the return of value.

Actions include calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression.

- Graphically, an activity diagram is a collection of vertices and arcs.

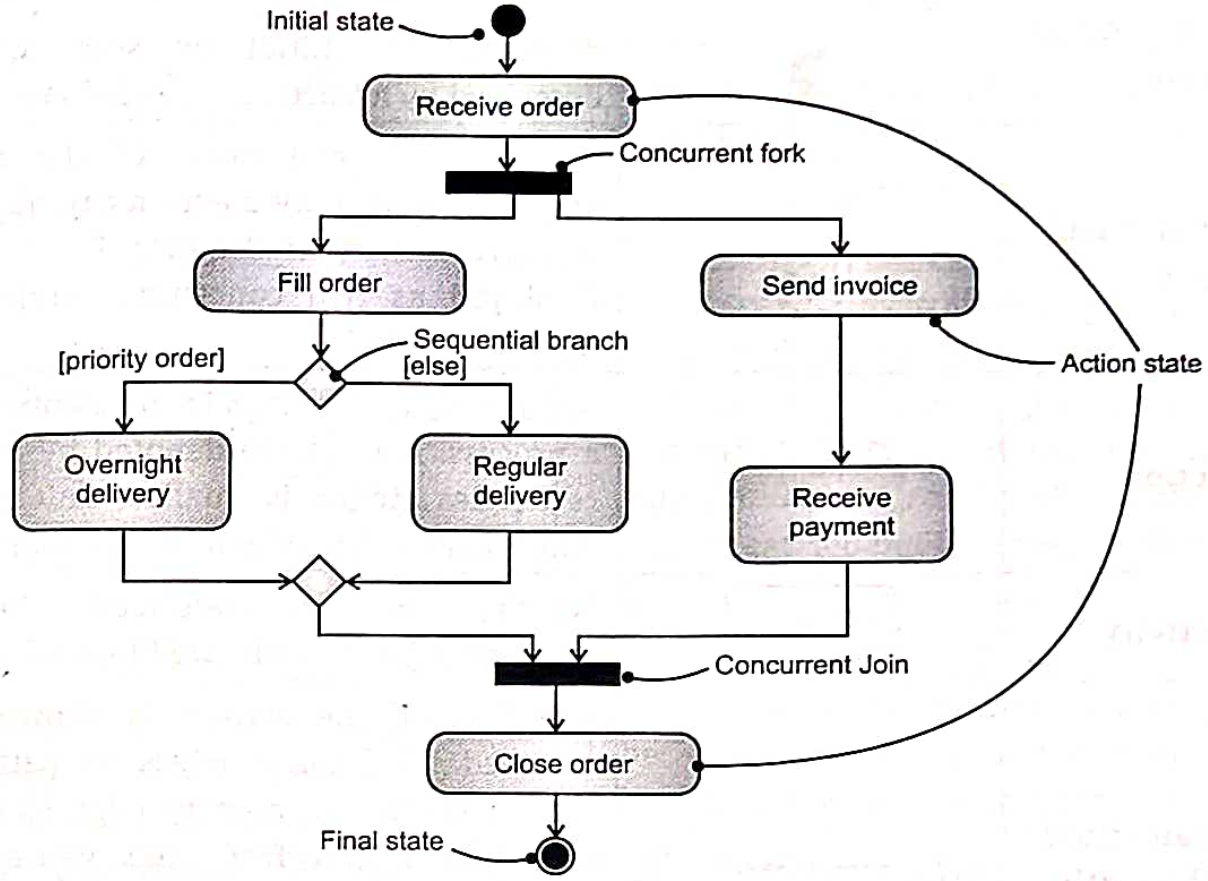



**Fig. 2.44: Activity Diagram**

**Purposes of Activity Diagram:**

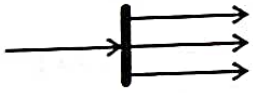
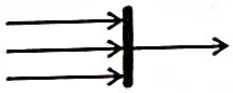
1. Draw the activity flow of a system.
2. Describe the sequence from one activity to another.
3. Describe the parallel, branched and concurrent flow of the system.

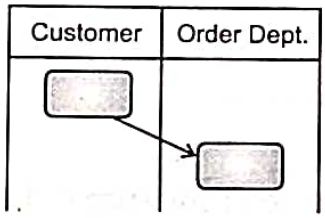
- Activity diagrams commonly contain actions, activity nodes, flows, fork, join etc. as shown in Fig. 2.44.
- Activities ultimately result in some action, which is made up of executable atomic computations those results in a change in state of the system or the return of a value.
- We can model these dynamic aspects using activity diagrams, which focus first on the activities that take place among objects.

## 2.2.2.1 Activity Diagram Notations

- An activity diagram is essentially a flowchart that emphasizes the activity that takes place over time.

**Table 2.3: Basic Notations of Activity Diagram**

| Name | Symbol | Description |
|---|---|---|
| 1. Start/Initial Node |  | It shows the starting point of the activity diagram. An initial or start node is described by a filled circle with black color. |
| 2. Final/Exit Node |  | It shows the exit point of the activity diagram. An activity diagram can have zero or more activity final nodes. Final node is displayed as two concentric circles with filled inner circle, |
| 3. Action |   | Actions are active steps in the completion of a process. Actions are denoted by rounded rectangles. Action is smallest unit of work which cannot be divided into further tasks. |
| 4. Activity |   | Activity is parameterized behavior represented as co-ordinated flow of actions. |
| 5. Transition/ Edge/Path |  | The flow of the activity is shown using arrowed lines called edges or paths. The arrowhead on an activity edge shows the direction of flow from one action to the next. A line going into a node is called an incoming edge, and a line exiting a node is called an outgoing edge. |
| 6. Fork Node |  | It is used to show the parallel or concurrent actions. Steps that occur at the same time are said to occur concurrently or in parallel. Fork has single incoming flow and multiple outing flows. |
| 7. Join Node |  | The join means that all incoming actions must finish before the flow can proceed past the join. Join has multiple incoming flows and single outing flow. |
| 8. Condition | [condition] | Condition text is placed next to a decision marker to let we know under what condition an activity flow should split off in that direction. |

| Name | Symbol | Description |
|---|---|---|
| 9. Decision/ Branch |   | A marker shaped like a diamond is the standard symbol for a decision. There are always at least two paths coming out of a decision and the condition text lets we know which options are mutually exclusive. |
| 10. Note |  | A note is used to display comments, constraints etc. of an UML element. |
| 11. Swimlane |   | We use partitions to show which participant is responsible for which actions. Partitions divide the diagram into columns or rows (depending on the orientation of your activity diagram) and contain actions that are carried out by a responsible group. The columns or rows are sometimes referred to as Swimlanes. |
| 12. Flow Final Node |  | A flow end node terminates its own path not the whole activity. The flow final node is described as a circle with a cross inside. |

## 2.2.2.2 Common Terms used in Activity Diagrams

1. **Action:**

- Activity represents a behavior that is composed of individual elements that are actions.
- An action represents a single step within an activity, for example, a calculation step that is not deconstructed any further.
- Actions are denoted by round-cornered rectangles.

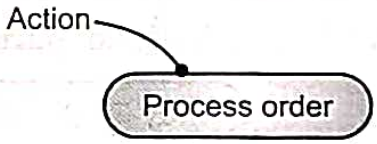



**Fig. 2.45: Process order action**

- In the flow of control modeled by an activity diagram, things happen.
- We might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, we might call an operation on an object, send a signal to an object, or even create or destroy an object. These executable, atomic computations are called actions.
- Fig. 2.46 shows inside the shape we may write an expression. Action states cannot be decomposed.

- Actions are atomic, i.e. events may occur, but the internal behavior of the action sta is not visible. We cannot execute part of an action, either it executes completely or n at all.

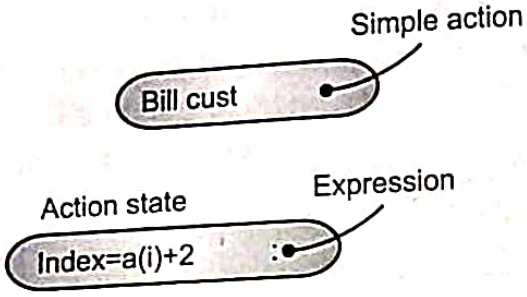



**Fig. 2.46: Actions**

**2. Activity Node:**

- An activity node is an organizational unit within an activity. These are neste groupings of actions or other nested activity nodes.
- Activity nodes have visible substructure.
- An activity is the specification of a parameterized sequence of behaviour.
- An activity diagram illustrates one individual activity.
- An activity is shown as a round-cornered rectangle enclosing all the actions, contr flows and other elements that make up the activity.


**Fig. 2.47: Activity Node**

- We can think of an action state as a special case of an activity node. An action is a activity node that cannot be further decomposed. Similarly, we can think of an activit node as a composite, whose flow of control is made up of other activity nodes an actions.
- Fig. 2.48 shows, there is no notational distinction between action and activity state except that an activity state may have additional parts, such as entry and exit actions.

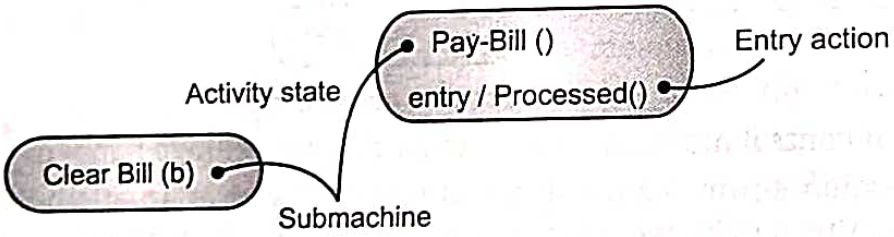



**Fig. 2.48: Activity States**

**3. Forking:**

- A fork represents the splitting of a single flow of control into two or more concurren flows of control.

- A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.
- Fig. 2.49 shows the fork, the activities associated with each of these paths continue in parallel. Conceptually, the activities of each of these flows are truly parallel.
- We represent a branch using diamond shape.

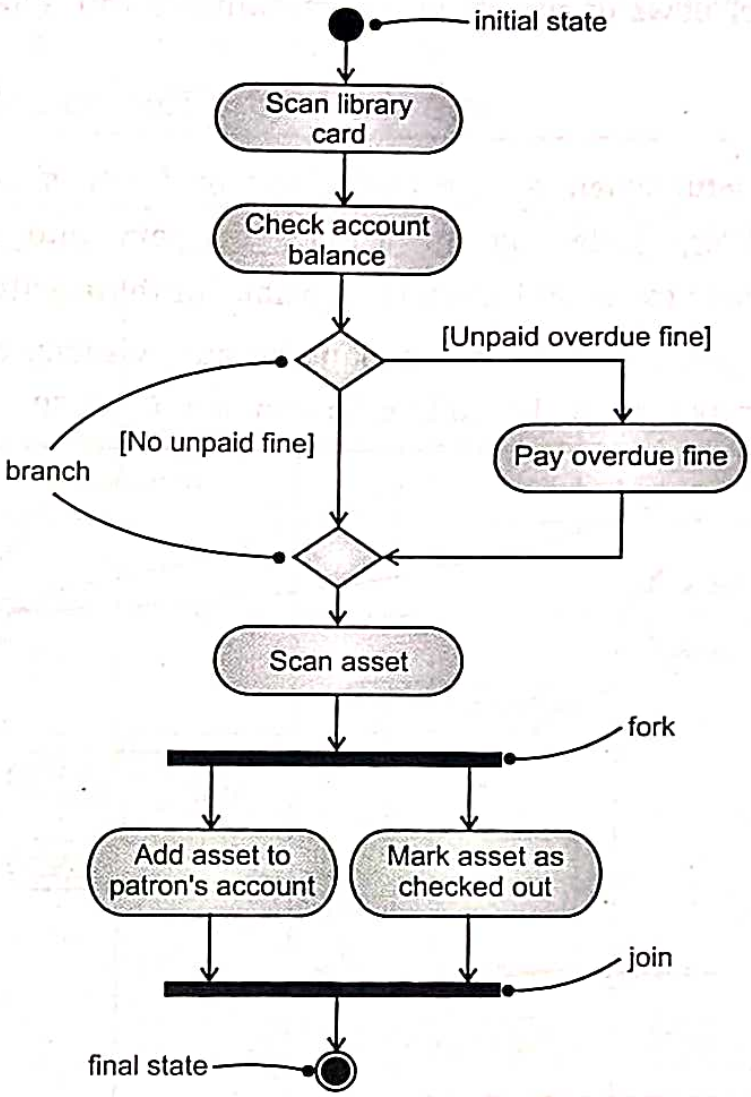



**Fig. 2.49: Forking and Joining Process**

**4. Joining:**

- Fig. 2.49 also shows a join represents the synchronization of two or more concurrent flows of control.
- A join may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continue in parallel.

- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.
- Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join. Also, activities that are in parallel flows of control may communicate with one another by sending signals.

5. **Swimlanes:**

- We will find it useful when we are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.
- In the UML, each group is called a swimlane because visually each group is divided from its neighbour by a vertical solid line, as shown in Fig. 2.50.

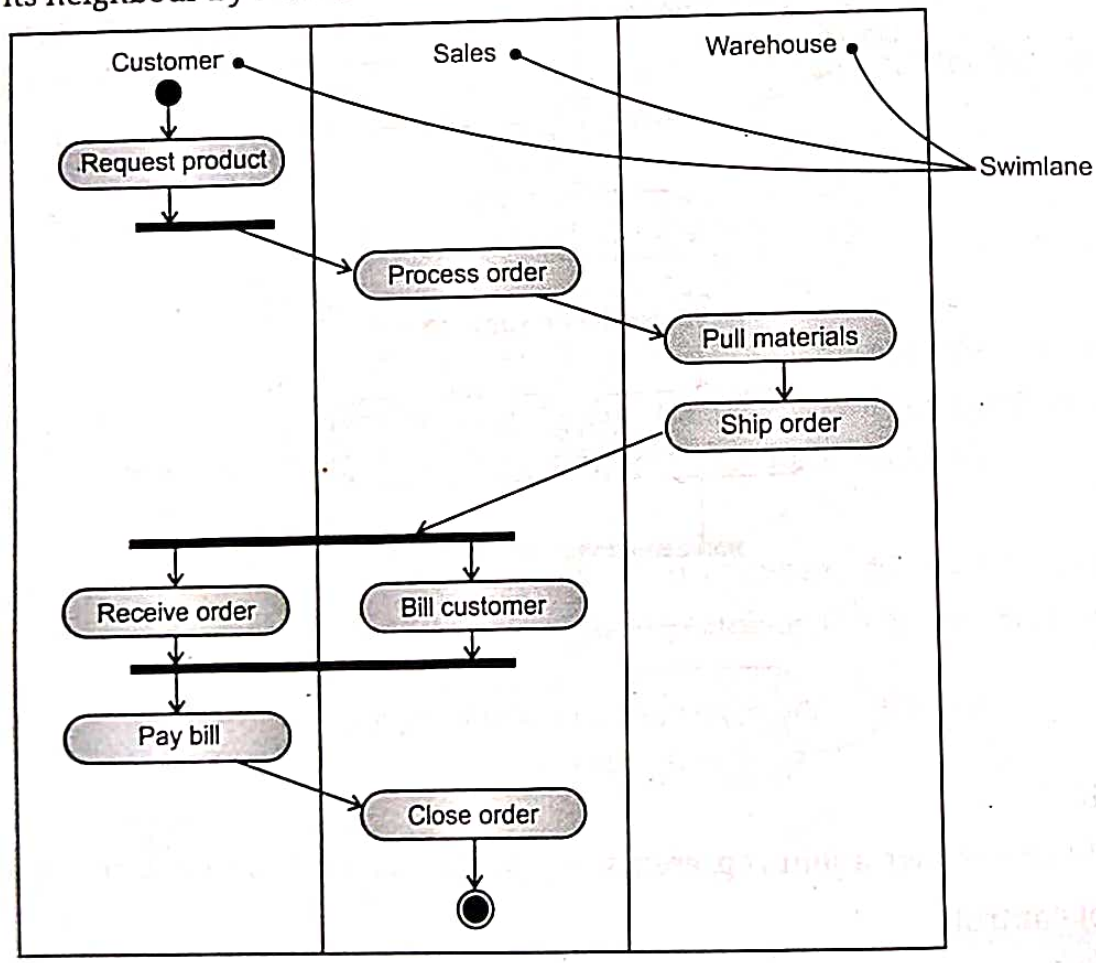



**Fig. 2.50: Swimlanes**

- A swimlane specifies a set of activities that share some organizational property.

- Each swimlane has a name unique within its diagram. It may represent some real-world entity, such as an organizational unit of a company.
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes.
- In an activity diagram (partitioned into Swimlanes), every activity belongs to exactly one swimlane, but transitions may crosslanes.

## 2.2.2.3 Examples of Activity Diagram

**1. Activity Diagram for ATM System:**

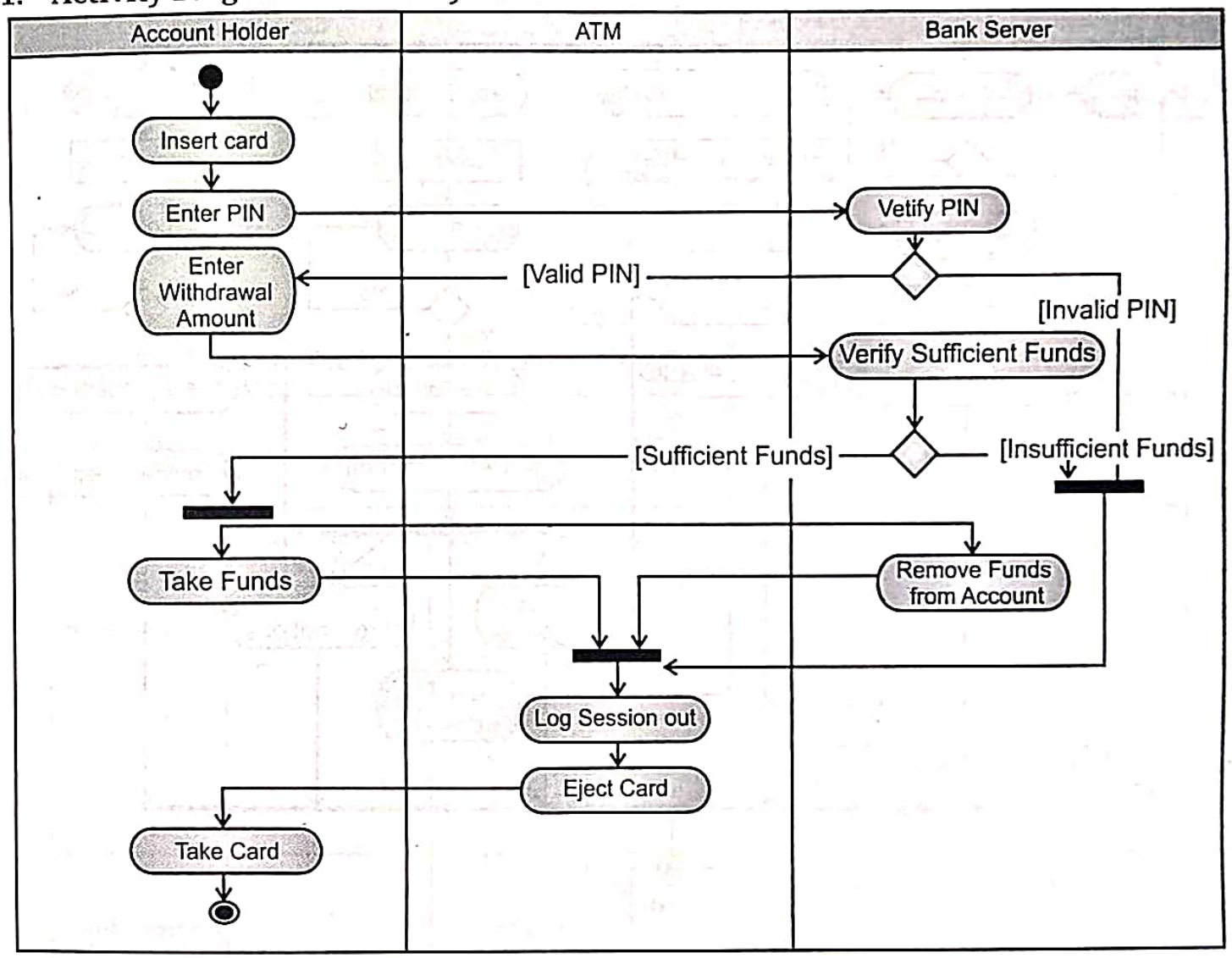



Fig. 2.51

## 2. Activity diagram for Hospital Management System:

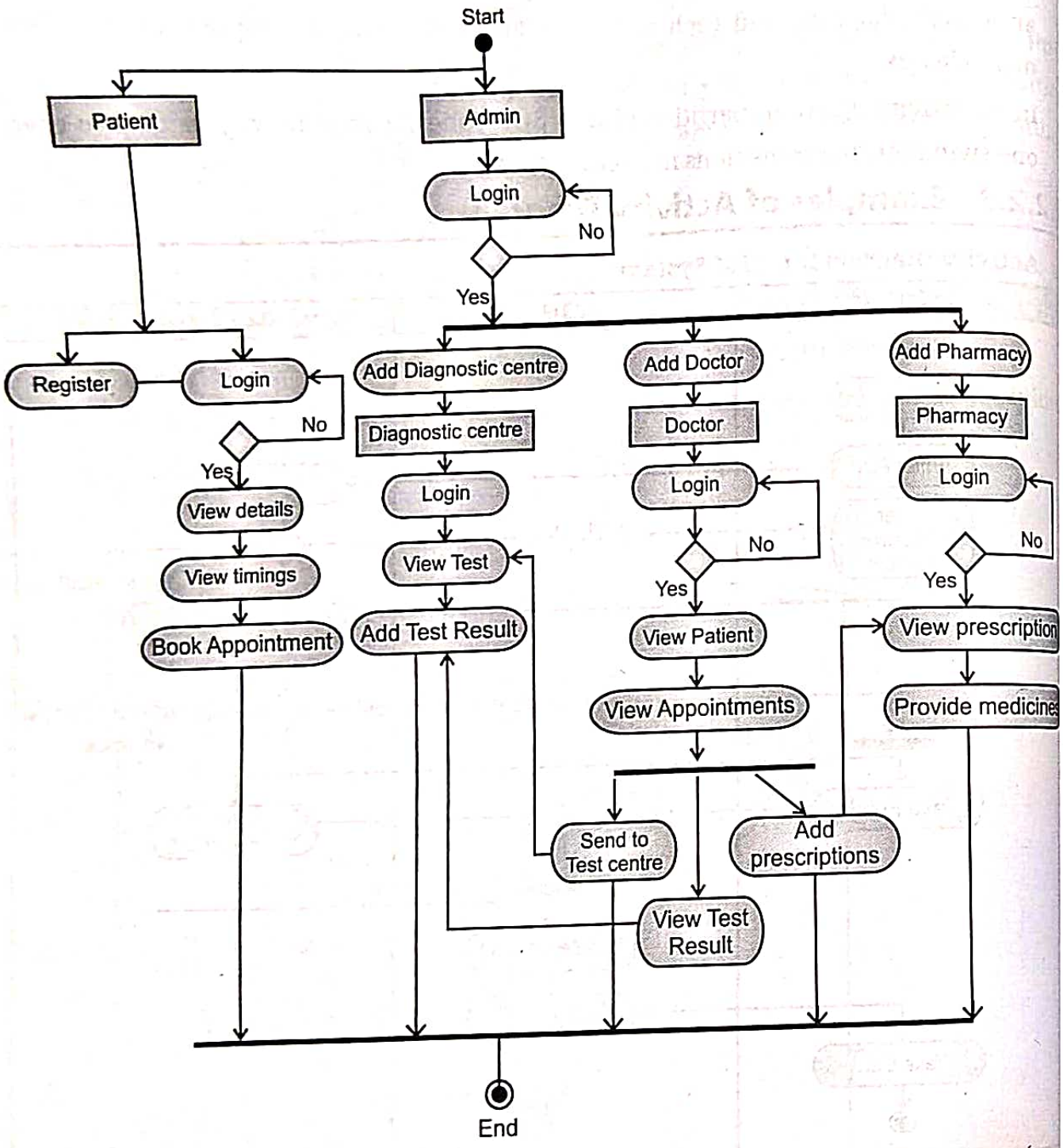



Fig. 2.52

## 3. Activity diagram for Library Management System:

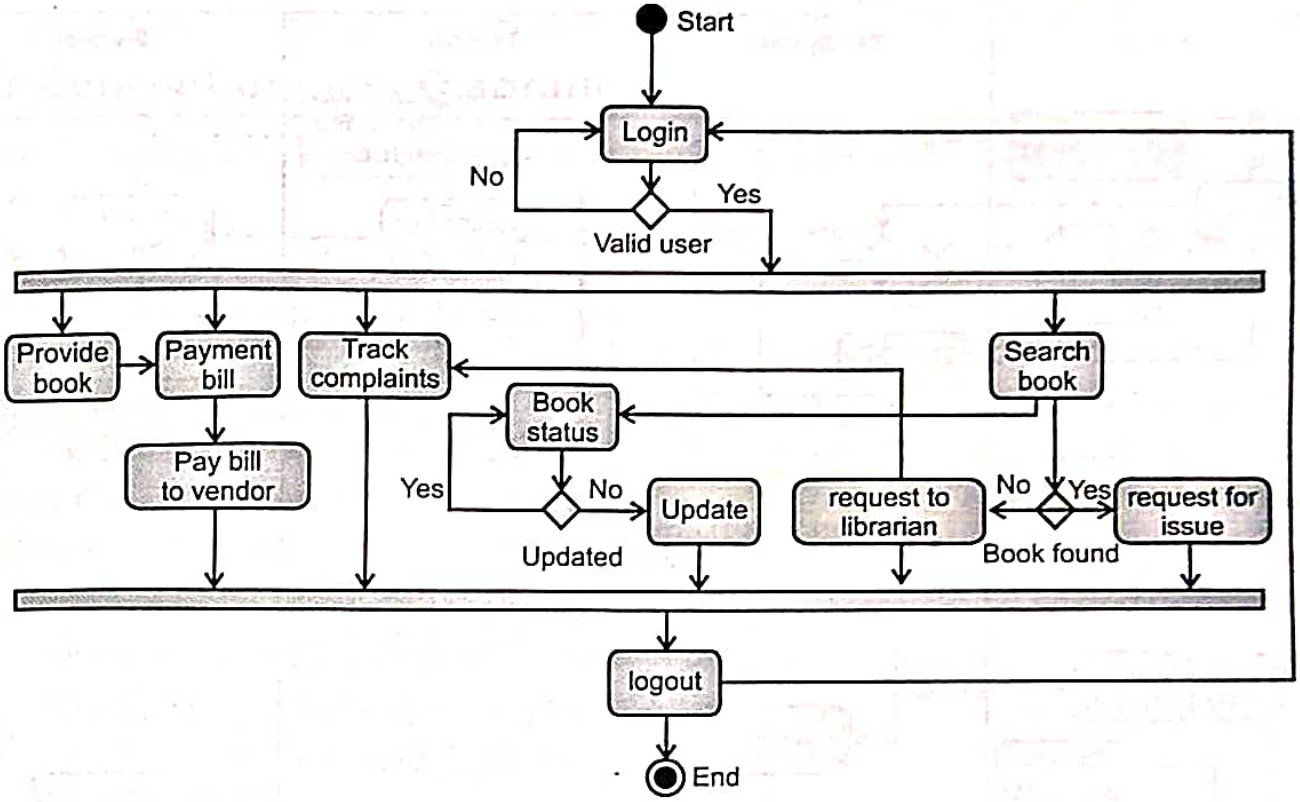



Fig. 2.53

## 4. Activity Diagram for Railway Reservation System

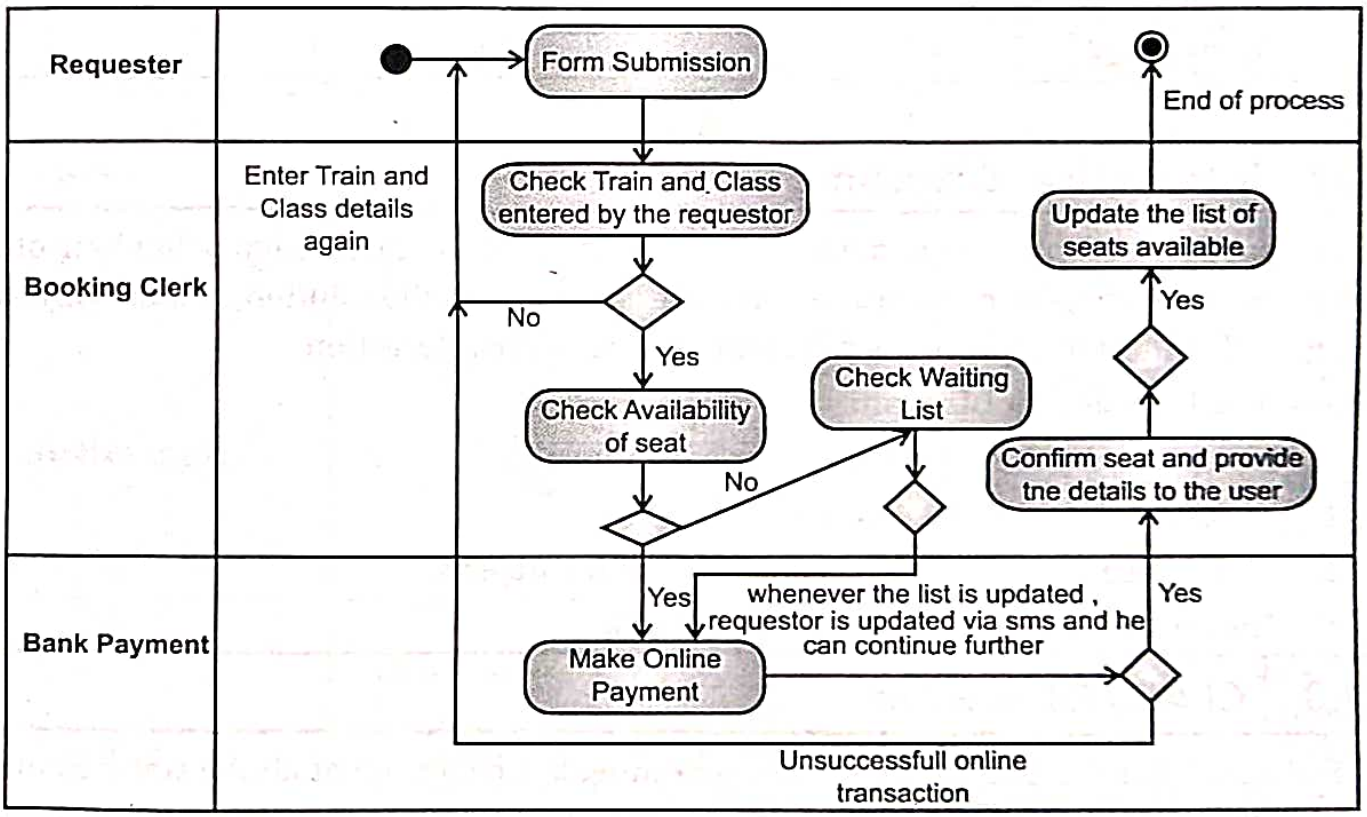



Fig. 2.54

5. **Activity Diagram for College Management System:**

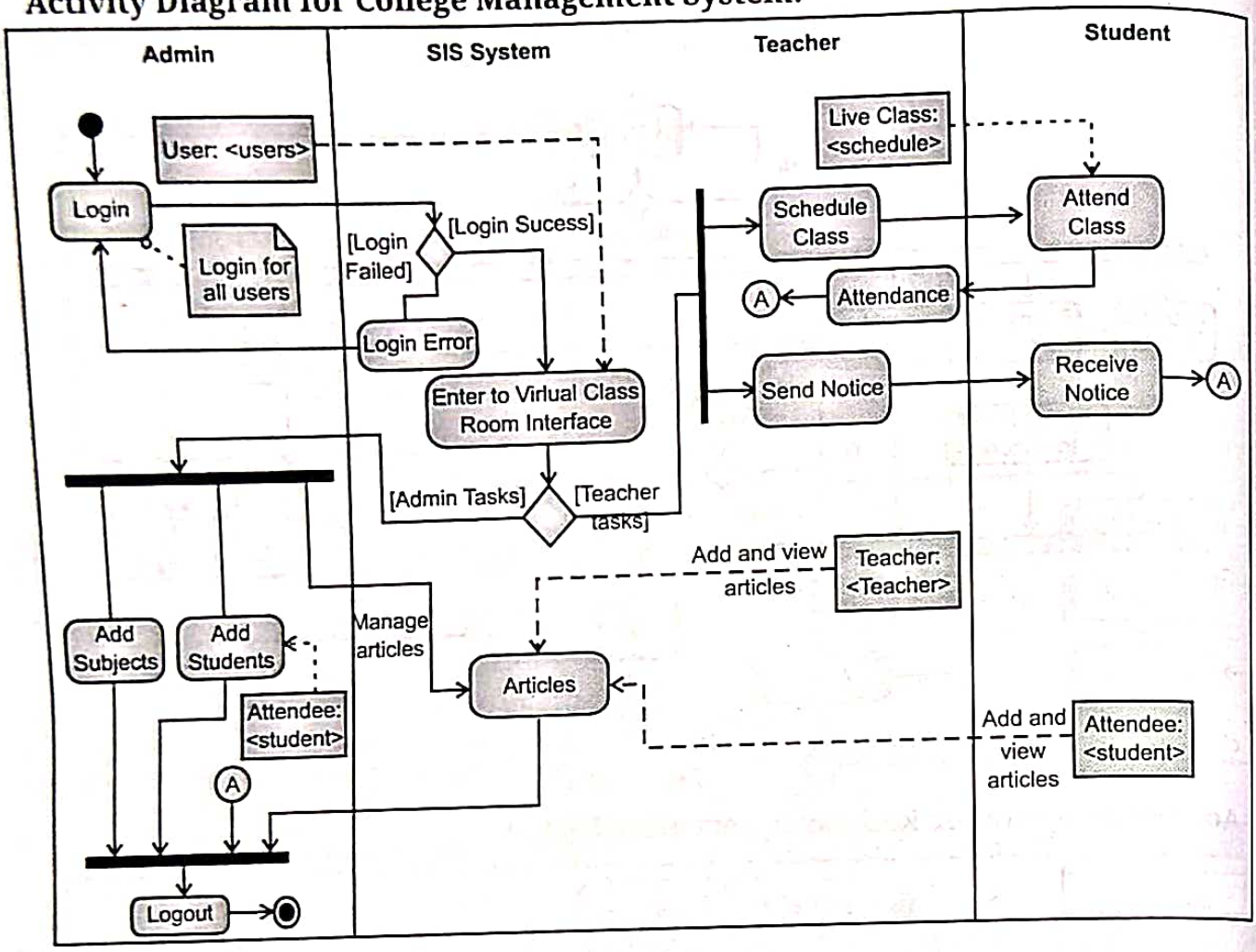



Fig. 2.55

## 2.2.3 Interaction Diagram

- The purposes of interaction diagrams are to visualize the interactive behaviour of th system. Now visualizing interaction is a difficult task. So the solution is to use differer types of models to capture the different aspects of the interaction.

**Purposes of Interaction Diagram:**

1. To capture the dynamic behaviour of a system.
2. To describe the message flow in the system.
3. To describe the structural organization of the objects.
4. To describe the interaction among objects.

## 2.3 CLASS DIAGRAM

- Static/structural aspect of the system is nothing but collection of all the static elemen that makeups the system. Designing of system's static aspect helps developers understand the basic building blocks which makeup the system.

- There are two different diagrams in UML that can be used to represent static structure of the system, namely Class diagram and Object diagram.

## 2.3.1 Concept of Class Diagram

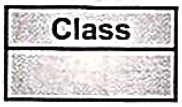
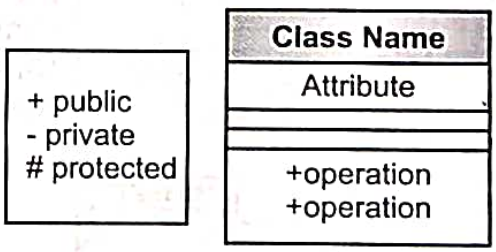
- The class diagram is a static diagram. It represents the static view of an application or a system.
- The class diagram describes the attributes and operations of a class and also the constraints imposed on the system.
- The class diagrams are widely used in the modeling of object-oriented systems.
- Class diagram provides an overview of the target system by describing the objects and classes inside the system and the relationships between them.
- Class diagrams provide a wide variety of usages from modeling the domain-specific data structure to detailed design of the target system.

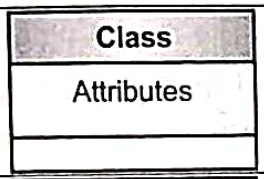
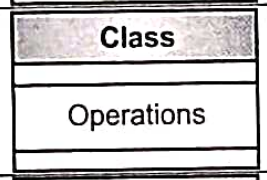
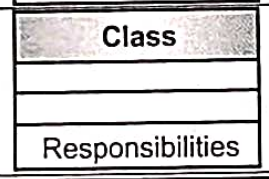
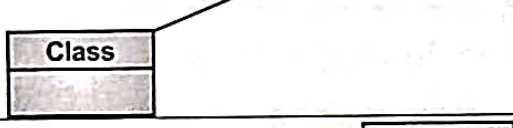
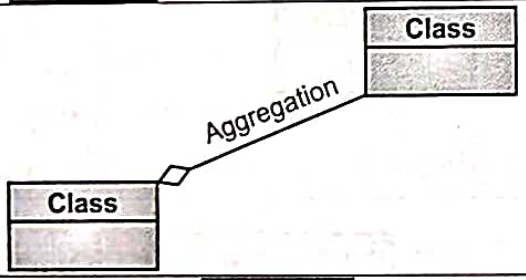
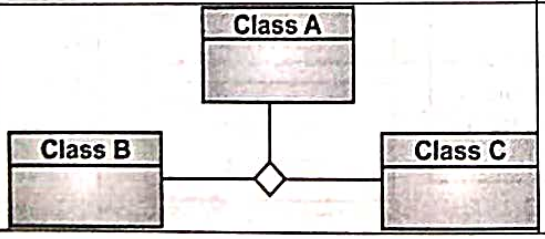
**Purpose of the Class Diagram:**

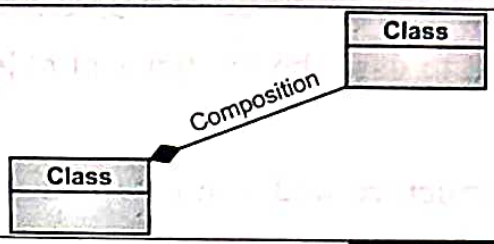
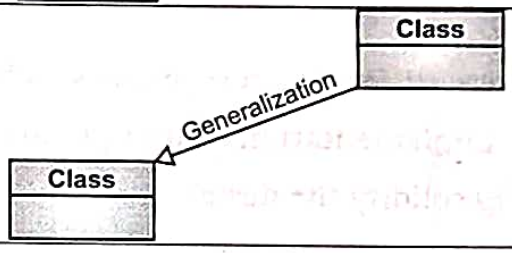
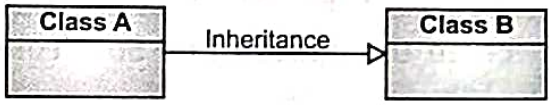
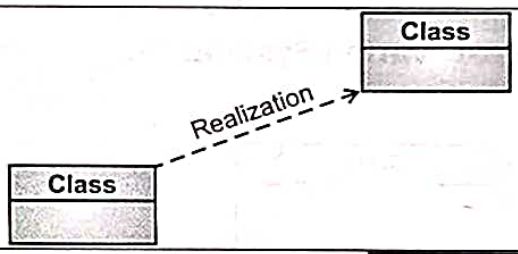
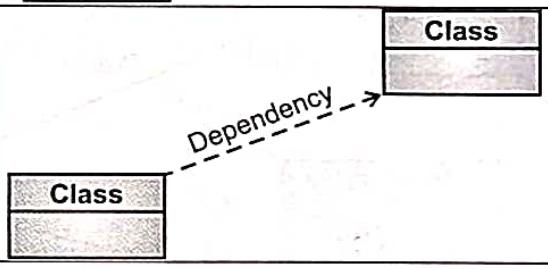
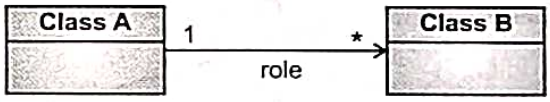
1. Analysis and design of the static view of an application.
2. Describe responsibilities of a system.
3. Base for component and deployment diagrams.
4. Forward and Reverse engineering.

### 2.3.1.1 Notations

**Table 2.3: Basic Notations used for Class Diagrams**

| Name | Symbol | Description |
|---|---|---|
| 1. Class |   | A class describes a set of objects that share the same specifications of features, constraints, and semantics. |
| 2. Active class |   | Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border. |
| 3. Visibility |   | Use visibility markers to signify who can access the information contained within a class. **Private** visibility hides information from anything outside the class partition. **Public** |

| Name | Symbol | Description |
|---|---|---|
| | | visibility allows all othe classes to view the marke information. **Protecte** visibility allows child class to access information the inherited from a pare class. |
| 4. **Attribute** |  | Attribute is a typed val that defines the properti and behavior of the object. |
| 5. **Operation** |  | Operation is a function th can be applied to the obje of a given class. |
| 6. **Responsibility** |  | Responsibility is a contra which the class mu conform. |
| 7. **Interface** |  | Interface is an abstract cl that defines a set operations that the object the class associated with t interface provides to otl objects. |
| 8. **Association** |  | Association is a relations that connects two classes. |
| 9. **Aggregation** |  | Aggregation is an associat with the relation betw the whole and its parts. relation when one class certain entity that inclu the other entities components. |
| 10. **N-ary Association** |  | N-ary association represe two or more aggregations |

| Name | Symbol | Description |
|---|---|---|
| 11. Composition |  | Composition is a strong variant of aggregation when parts cannot be separately of the whole. |
| 12. Generalization |  | Generalization is an association between the more general classifier and the more special classifier. |
| 13. Inheritance |  | Inheritance is a relationship when a child object or class assumes all properties of his parent object or class. |
| 14. Realization |  | Realization is a relationship between interfaces and classes or components that realize them. |
| 15. Dependency |  | Dependency is a relationship when some changes of one element of the model can need the change of another dependent element. |
| 16. Multiplicity |  | Multiplicity shows the number of instances of one class that are linked to one instance of the other class. |
| 17. Package |  | Package groups the classes and other packages. |
| 18. Collaboration |  | A collaboration specifies a view (or projection) of a set of co-operating classifiers. It describes the required links between instances that play the roles of the collaboration, as well as the features required of the classifiers that specify the participating instances. |

## 2.3.1.2 Examples

- Class diagrams are widely used to describe the types of objects in a system and relationships.
- Class diagrams model class structure and contents using design elements su classes, packages and objects.
- Class diagrams describe three different perspectives when designing a s conceptual, specification and implementation. These perspectives become evid the diagram is created and help solidify the design.
- Class diagram is basically a graphical representation of the static view of the s and represents different aspects of the application. So a collection of class dia represent the whole system.

**1. Class Diagram for Railway Reservation System:**

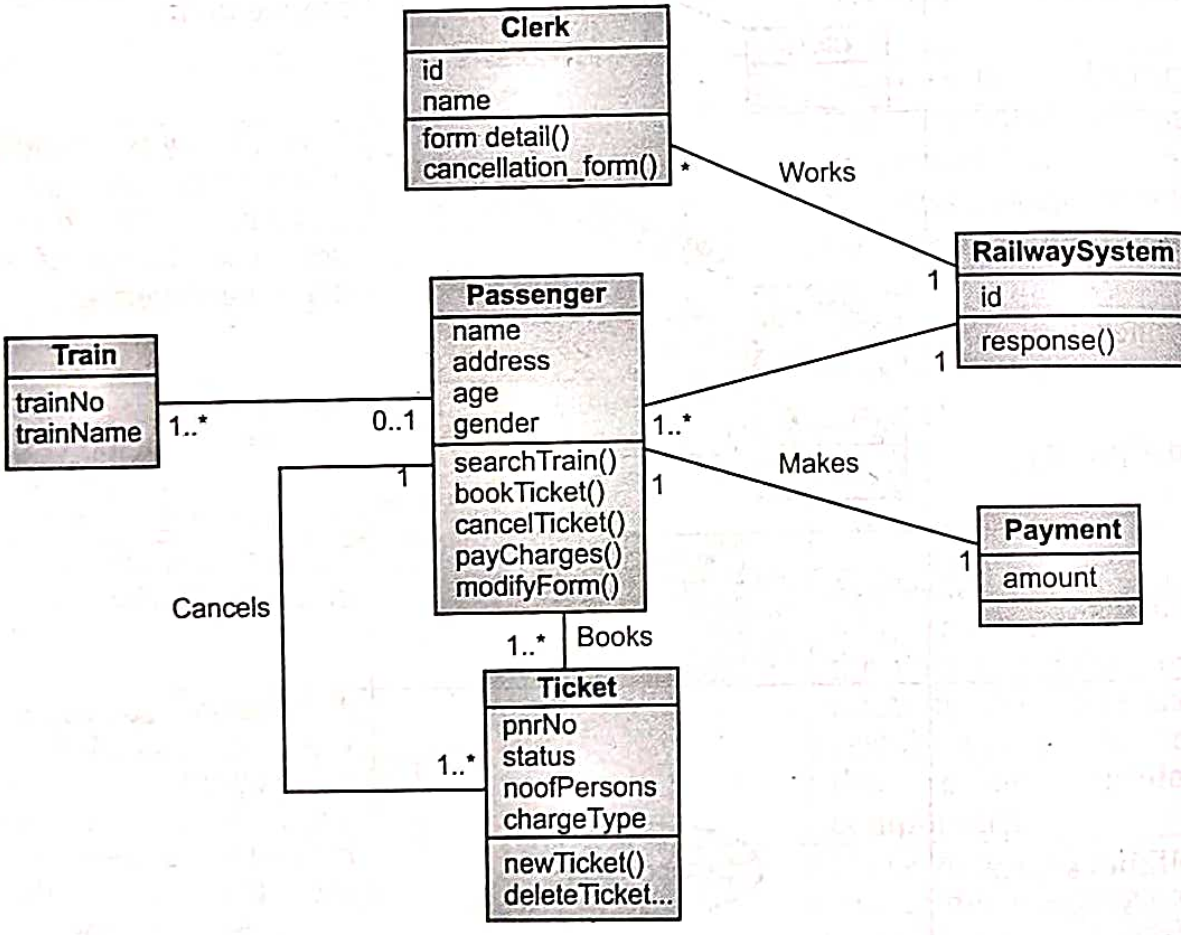



Fig. 2.56

## 2. Class Diagram for Library Management System:

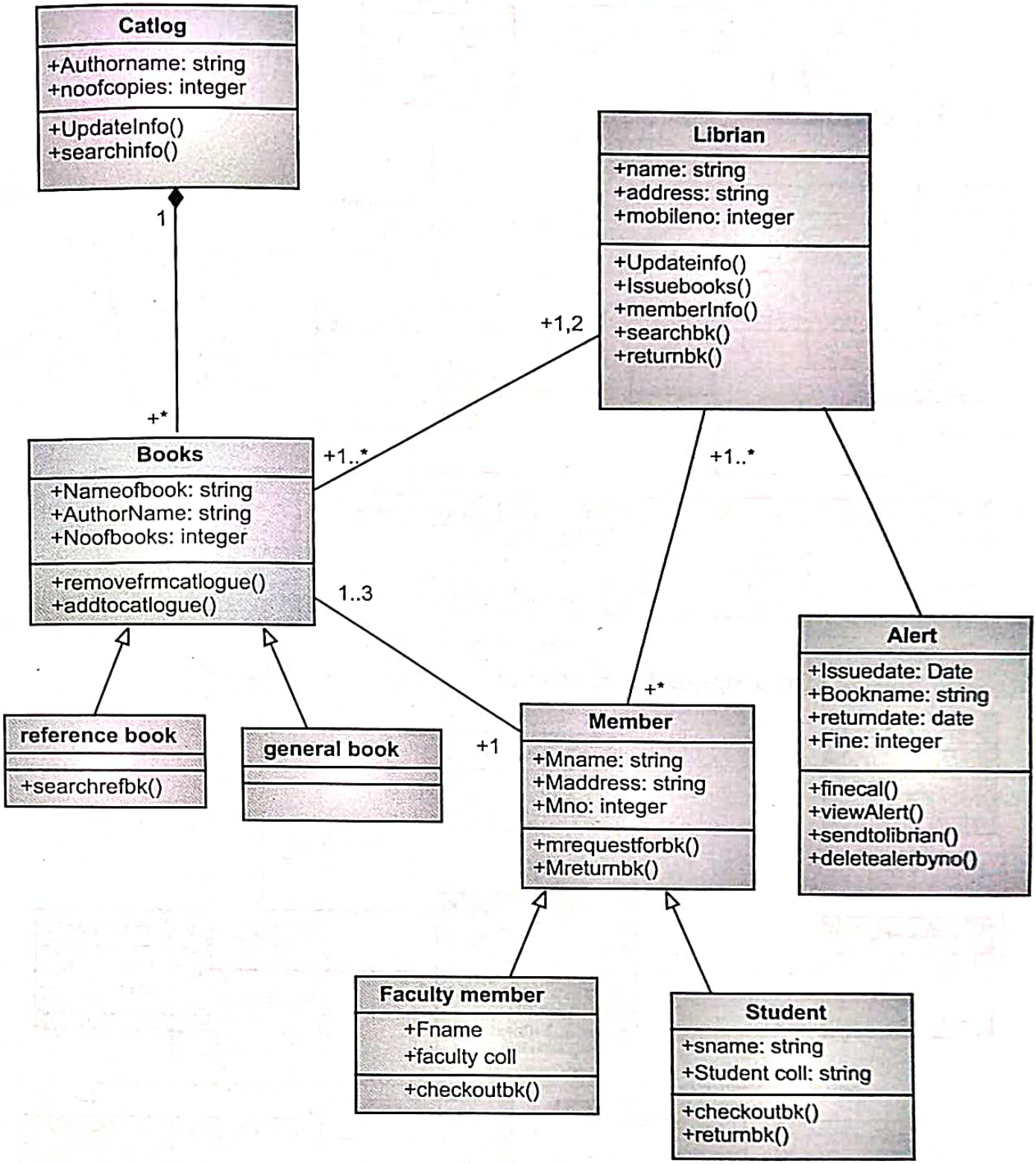



Fig. 2.57

## 3. Class Diagram for ATM System:

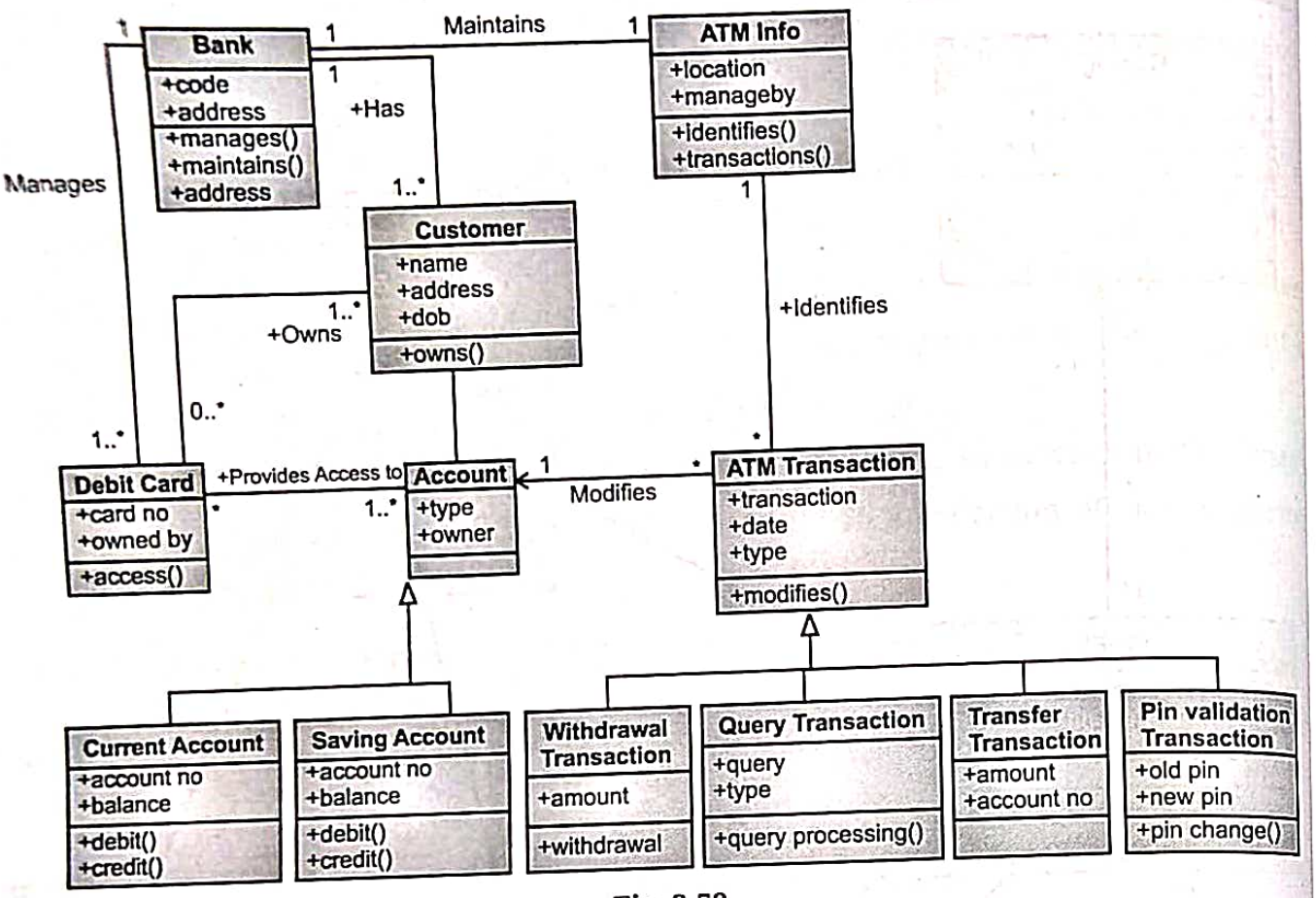



Fig. 2.58

## 4. Class Diagram for a Simple Bank System:

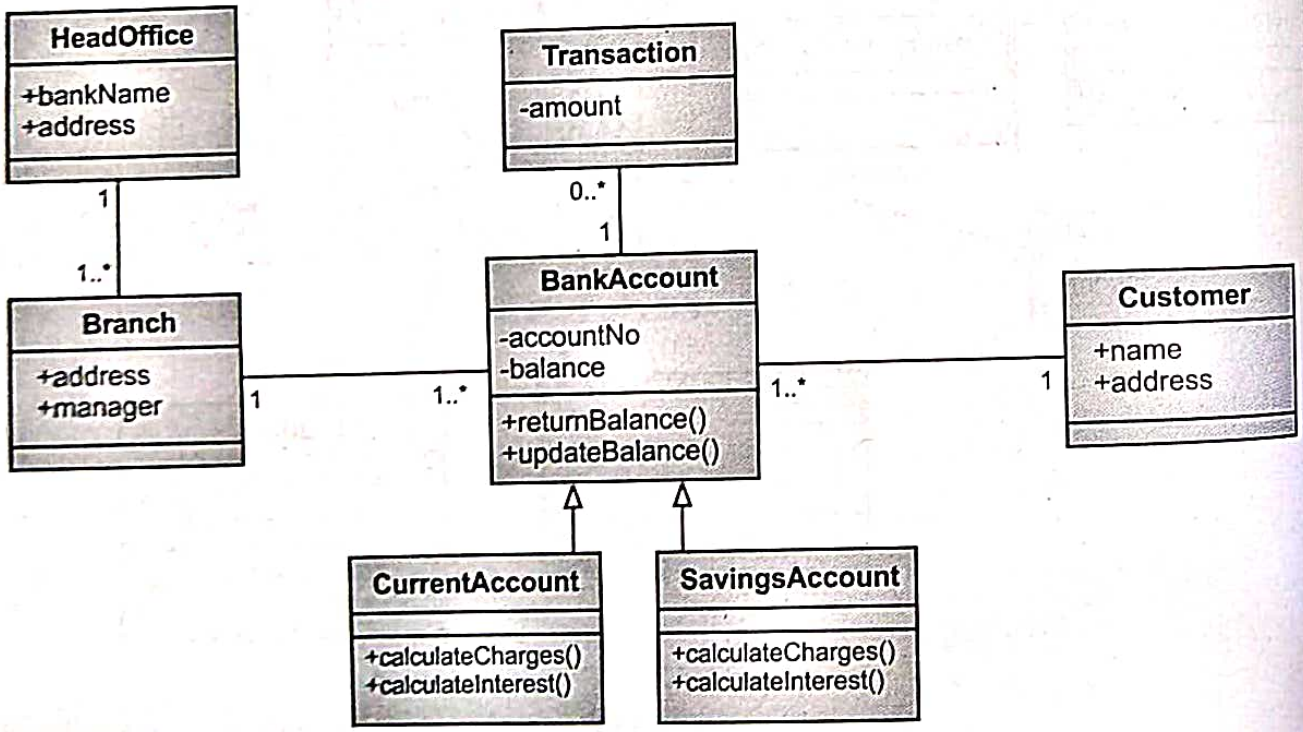



Fig. 2.59

## . Class Diagram for Hospital Management System:

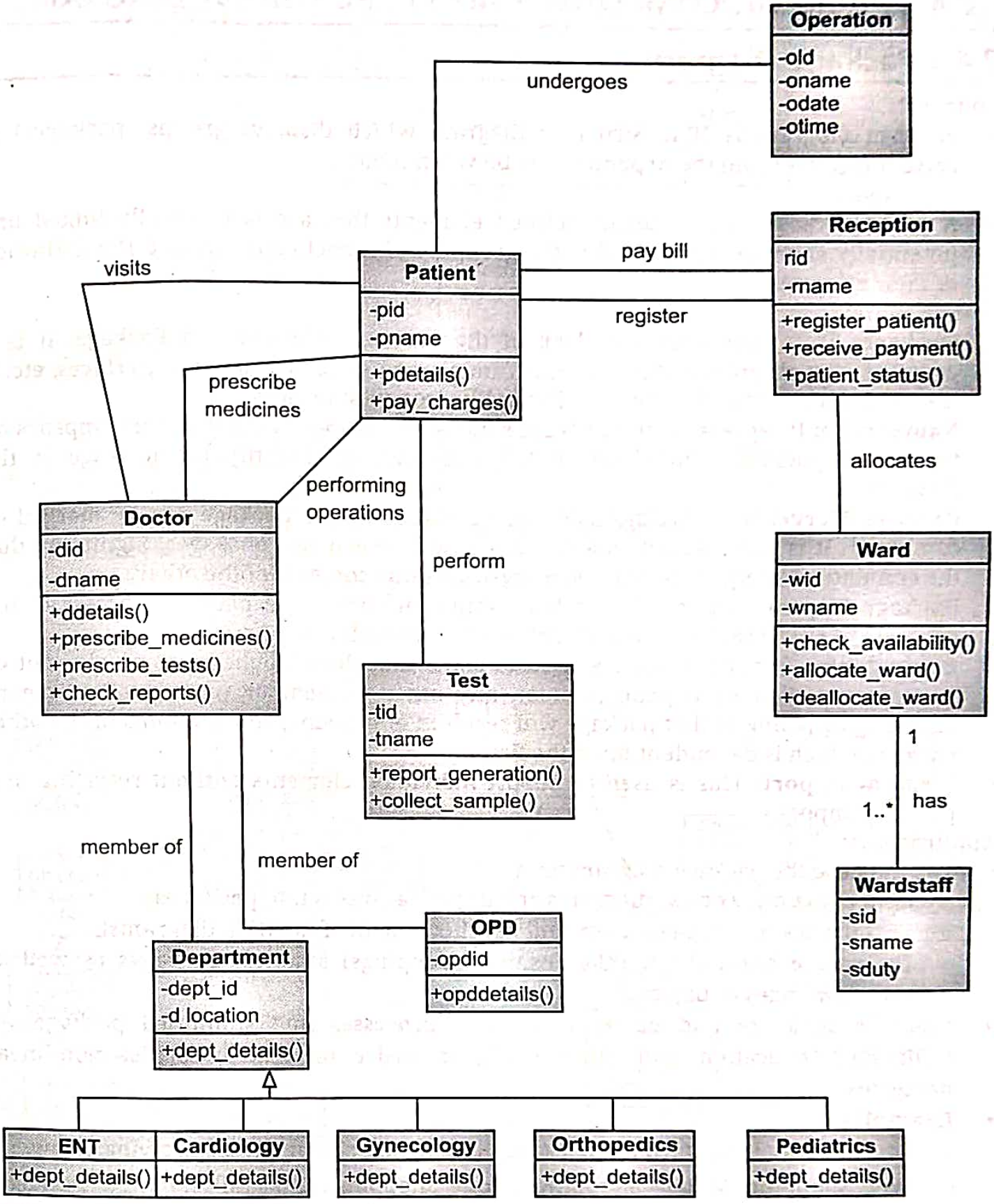



Fig. 2.60

# 2.4 PACKAGE, COMPONENT AND DEPLOYMENT DIAGRAMS

## 2.4.1 Package Diagram

**Concept:**

- Package diagram is UML structure diagram which displays groups (packages) o related elements and the dependencies between them.

Or

- A package diagram is a set of defined elements that are semantically linked an potentially changed together. It is the grouping of model elements and the definitio of their relationships.

**Components:**

- **Package:** This basic building block of the Package Diagram is a Package. It is container for organizing different diagram elements such as classes, interfaces, etc. is represented in the Diagram in a folder-like icon with its name.
- **NameSpace:** It represents the package's name in the diagram. It generally appears o top of the package symbol which helps to uniquely identify the package in th diagram.
- **Package Merge:** It is a relationship that signifies how a package can be merged o combined. It is represented as a direct arrow between two packages. Signifying tha the contents of one package can be merged with the contents of the other.
- **Package Import:** It is another relationship that shows one package's access to th contents of a different package. It is represented as a Dashed Arrow.
- **Dependency:** Dependencies are used to show that there might be some element o package that can be dependent upon any other element or package. This mean changing anything of that package will result in alteration of the contents of the othe package which is dependent upon the first one.
- **Element Import:** This is used to import individual elements without resorting to package import.

**Applications:**

- You may use the package diagram to:
  - Simplify complex class diagrams and organize classes into packages.
  - Define packages as file folders and use them on all of the UML diagrams.
  - Define the hierarchical relationships (groupings) amongst packages as well a other packages or objects.
- Create a structure and visualize complex processes into simplified packages i technology, education and other fields, in order to visually depict non-linea processes.
- **Examples:**
  - Some major elements of the package diagram are shown on the drawing below.
  - Web Shopping, Mobile Shopping, Phone Shopping, and Mail Shopping package merge Shopping Cart package. The same 4 packages use Payment package. Bot Payment and Shopping Cart packages import other packages.

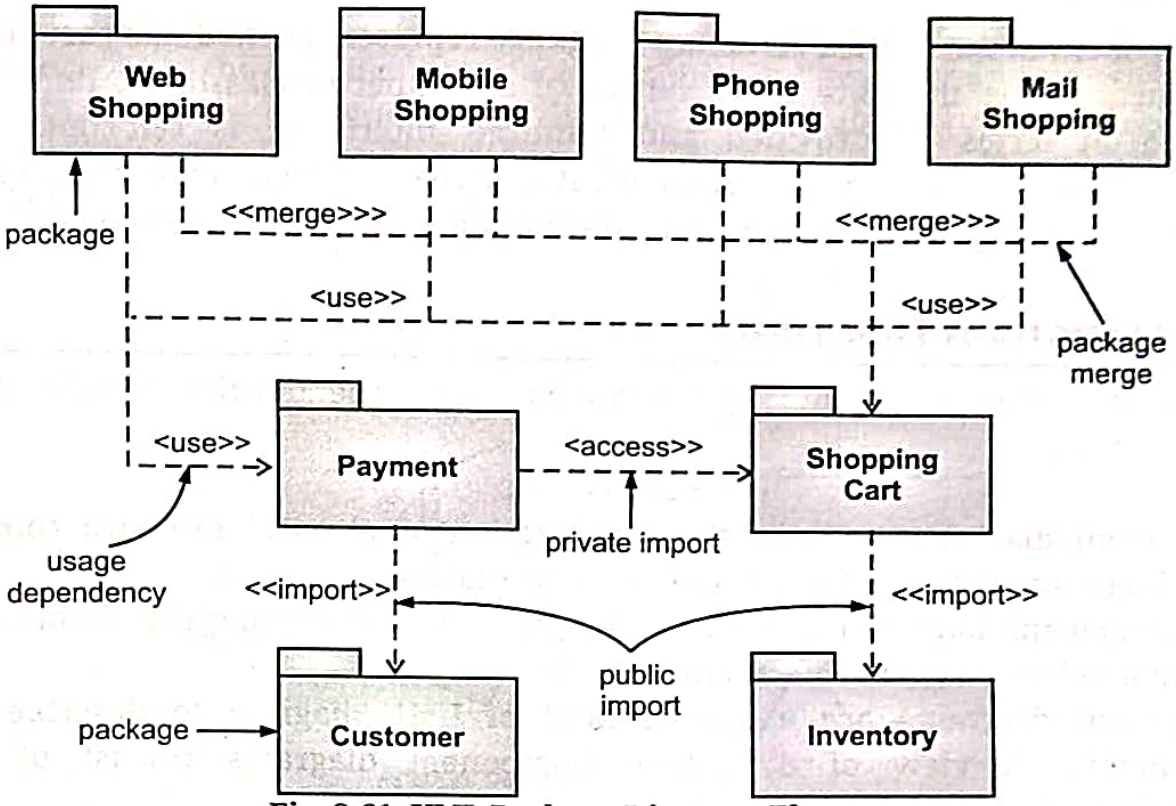



**Fig. 2.61: UML Package Diagram Elements**

**Example:** Purchase Order.

- In this diagram, the "purchase order" process is simplified by a package diagram.

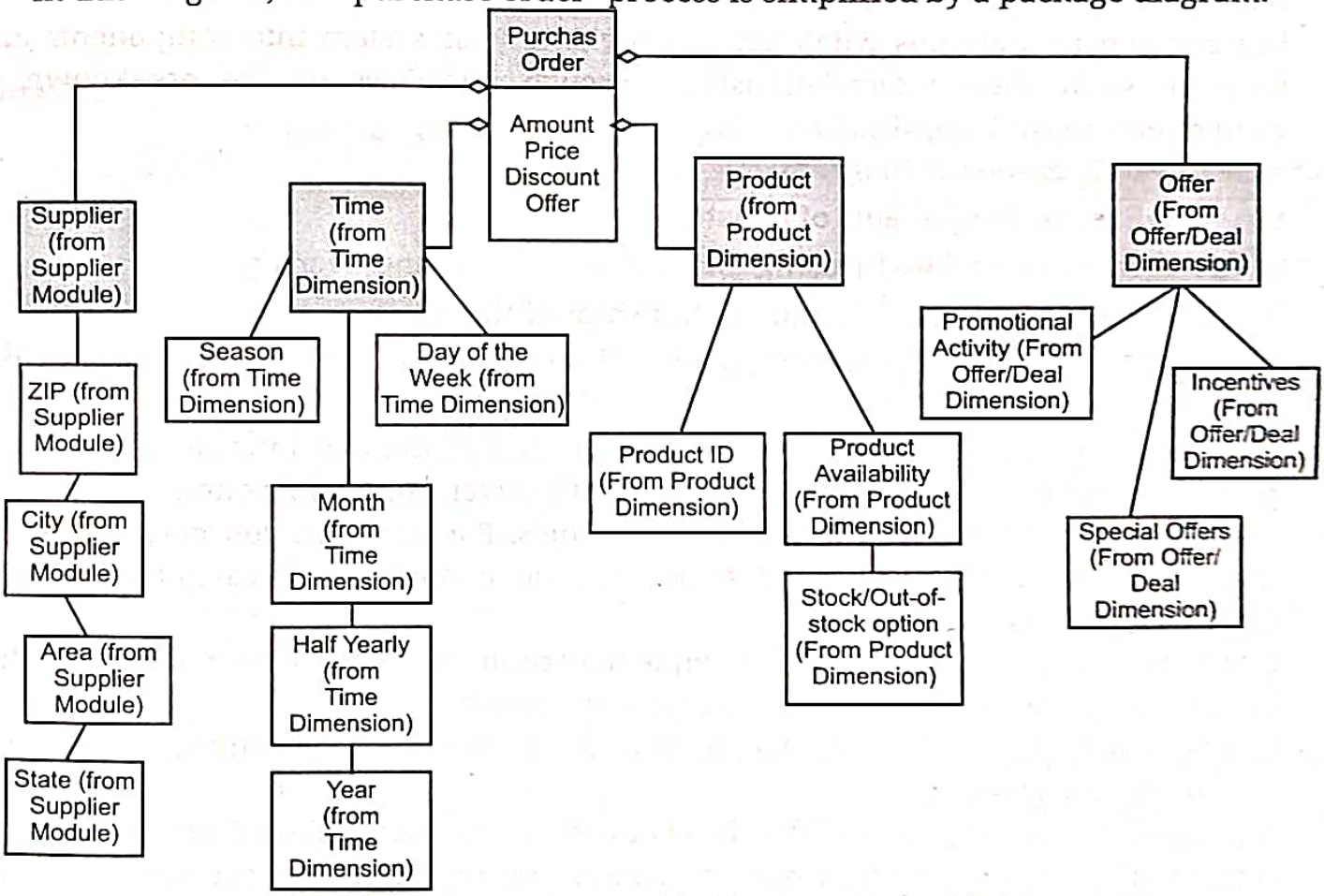



**Fig. 2.62: Package diagram for Purchase Order System**

- A component in the Unified Modeling Language represents a modular part of a syste that encapsulates the state and behavior of a number of classifiers. Its behavior defined in terms of provided and required interfaces, is self-contained, an substitutable. A number of UML standard stereotypes exist that apply to components.
- Component is a physical part of an information system that exists at developme time.

## 2.4.2 Component Diagrams

- Component diagram shows organizations and dependencies among a set components.

**Concept:**

- Component diagrams describe the organization of physical software componen including source code, run-time code and executables.
- The component diagram represents at a high level, what components form part of th system and how they are interrelated.
- Component diagrams are a special kind of UML diagram to describe a stat implementation view of a system. Component diagrams consist of physic components like libraries, files, folders etc.
- Component diagram shows components, provided and required interfaces, ports, an relationships between them.
- Use component diagrams when you are dividing your system into components an want to show their interrelationships through interfaces or the breakdown components into a lower-level structure.

**Purpose of the Component Diagram:**

1. Visualize the components of a system.
2. Construct executables by using forward and reverse engineering.
3. Describe the organization and relationships of the components.

- A component is a logical, replaceable part of a system that conforms to and provid the realization of a set of interfaces.
- Good components define hard abstractions with well-defined interfaces, making possible to easily replace older components with newer, compatible ones.
- Interfaces bridge your logical and design models. For example, you may specify interface for a class in a logical model, and that same interface will carry over to so design component that realizes it.
- Interfaces allow you to build the implementation of a component using small components by wiring ports on the components together.
- Component diagrams illustrate the pieces of software, embedded controllers, etc., th will make up a system.
- A component diagram has a higher level of abstraction than a class diagram; usuall component is implemented by one or more classes (or objects) at run-time. They building blocks so a component can eventually encompass a large portion of a syste

Fig. 2.63 demonstrates some components and their inter-relationships. Assembly connectors "link" the provided interfaces supplied by "Product" and "Customer" to the required interfaces specified by "Order". A dependency relationship maps a customer's associated account details to the required interface; "Payment", indicated by "Order".

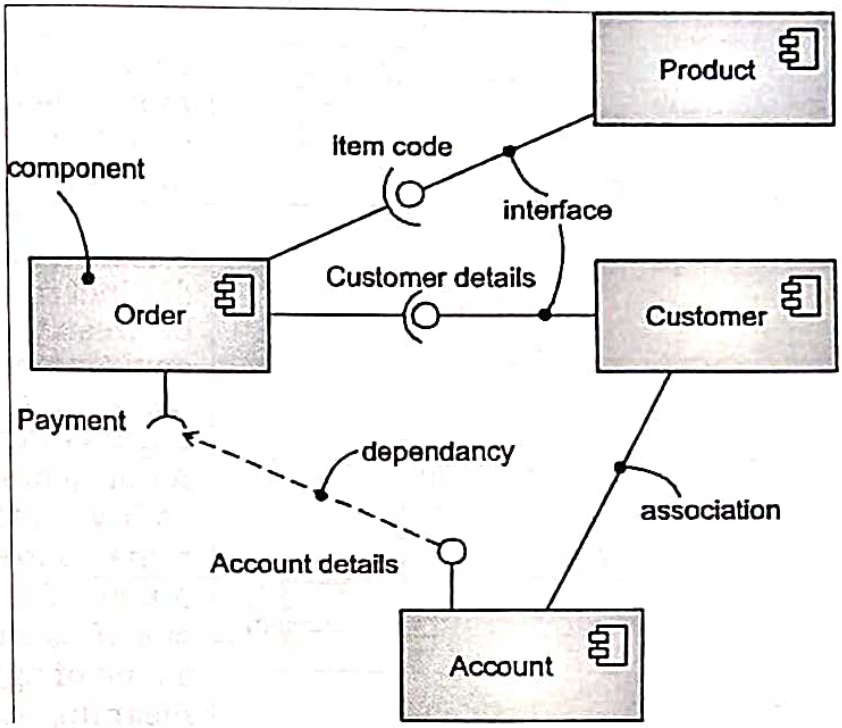
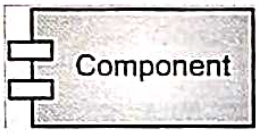
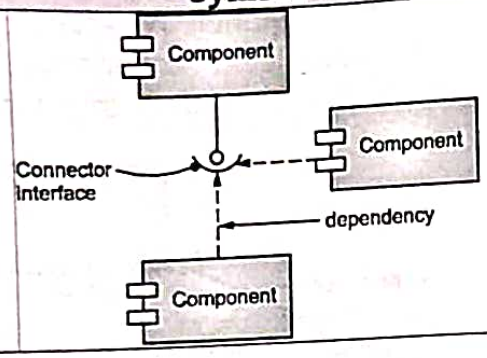
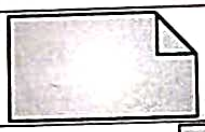
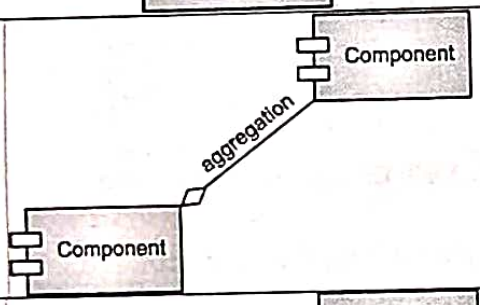
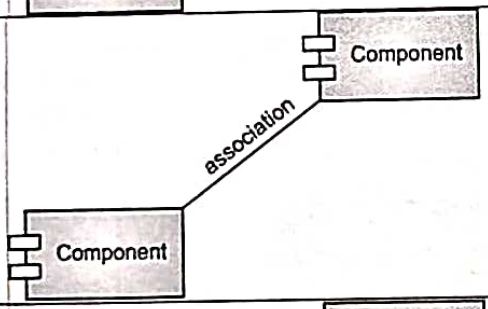
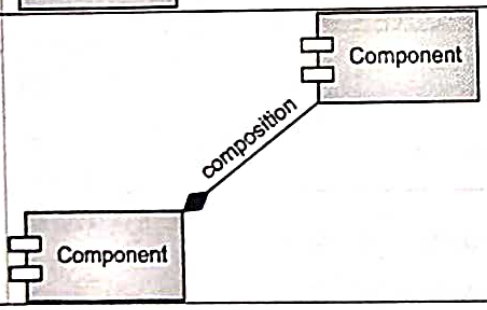
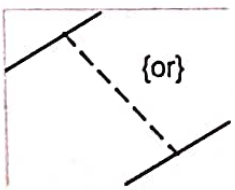


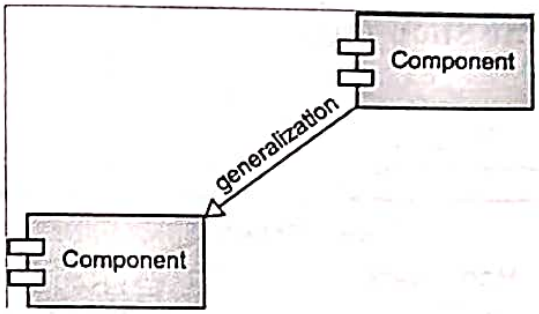
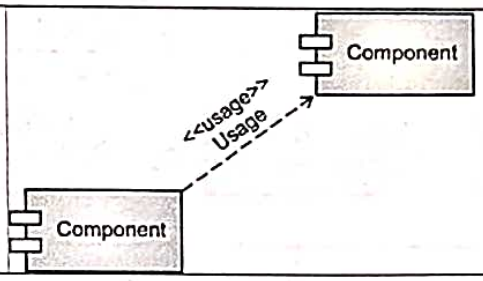

**Fig. 2.63: Component Diagram**

**Notations:**

**Table 2.4: Basic Notations for Components Diagrams**

| Name | Symbol | Description |
|---|---|---|
| **1. Component** |   | A component is a physical building block of the system. It is represented as a rectangle with tabs. A component represents a modular part of a system that encapsulates its contents and whose appearance is replaceable within its environment. |
| **2. Interface** |  | An interface describes a group of operations used or created by components. An interface is a kind of classifier that represents a declaration of a set of clear public features and obligations. |

| Name | Symbol | Description |
|---|---|---|
| 3. Dependencies |  | Draw dependencies among components using dashed arrows. A dependency is a relationship that signifies a single or a set of model elements that requires other model elements for their specification or implementation. |
| 4. Port |  | A port is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. |
| 5. Note |  | A note (comment) gives the ability to attach various remarks to elements. |
| 6. Aggregation |  | A kind of association that has one of its end marks shared as a kind of aggregation, meaning that it has a shared aggregation. |
| 7. Association (Link) |  | An association specifies a semantic relationship (link) that can occur between typed instances. |
| 8. Composition |  | An association may represent a composite aggregation (i.e., a whole/part relationship). |
| 9. Constraint |  | A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. |

| Name | Symbol | Description |
|---|---|---|
| 10. Generalization |  | A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier. |
| 11. Usage |  | A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. |

xamples:

. Component Diagram for Library Management System:

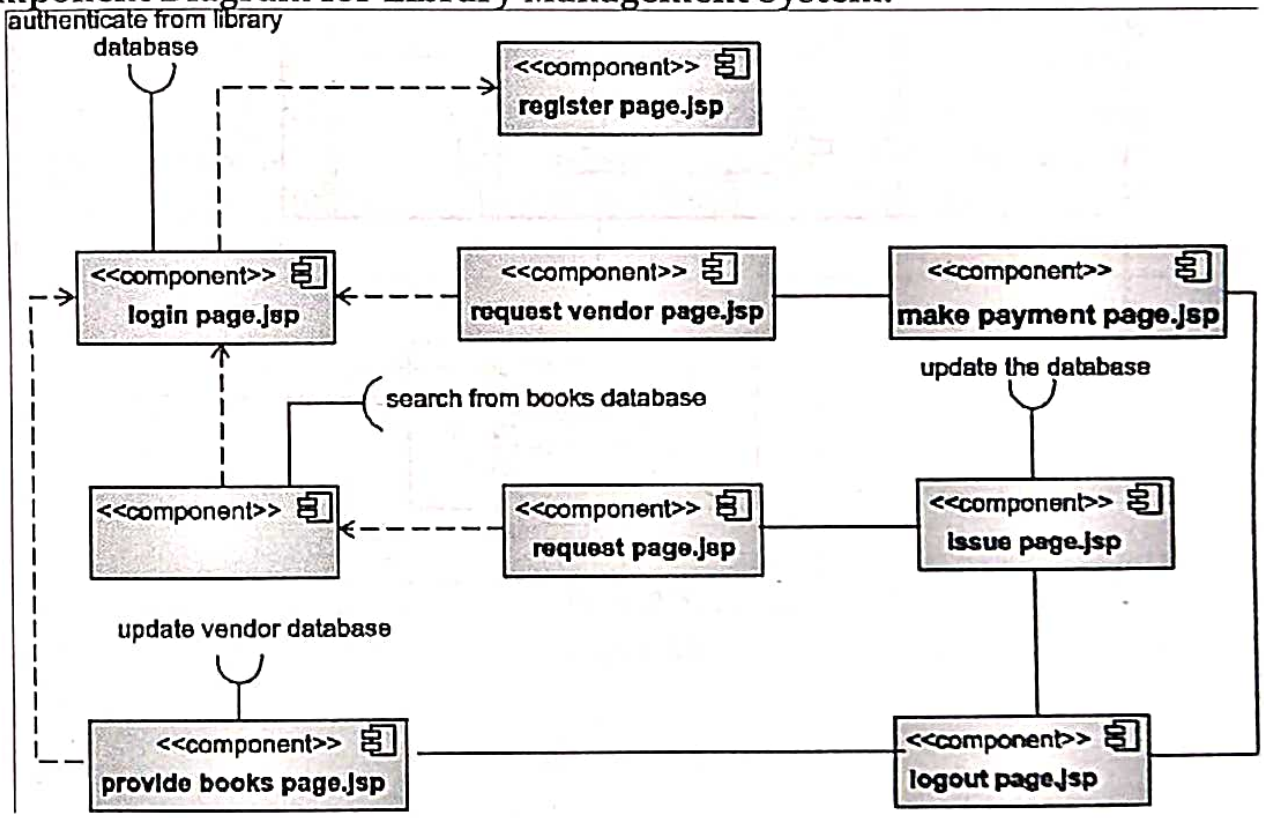



Fig. 2.64

## 2. Component Diagram for Online Shopping:

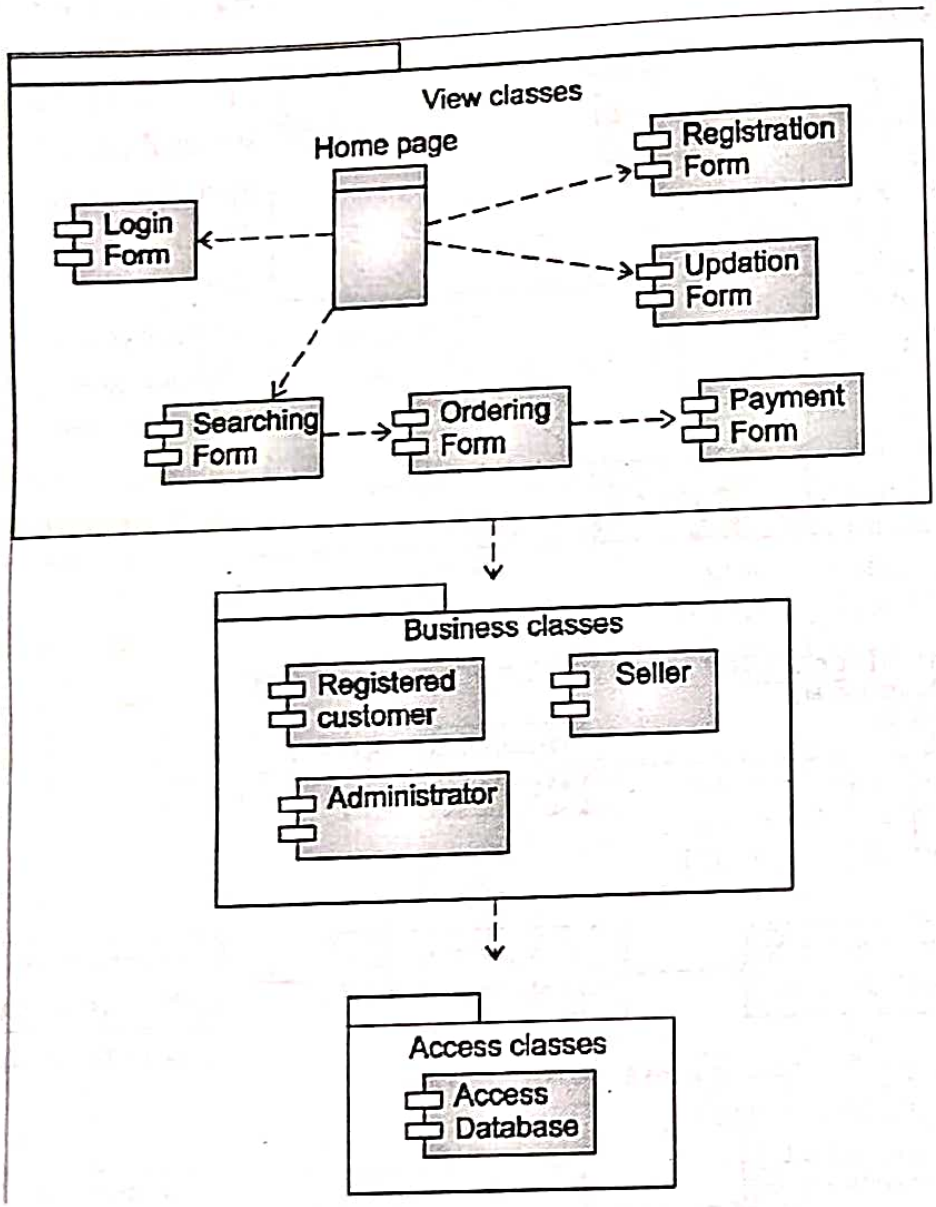



Fig. 2.65

## 3. Component Diagram for Railway Ticket Reservation System:

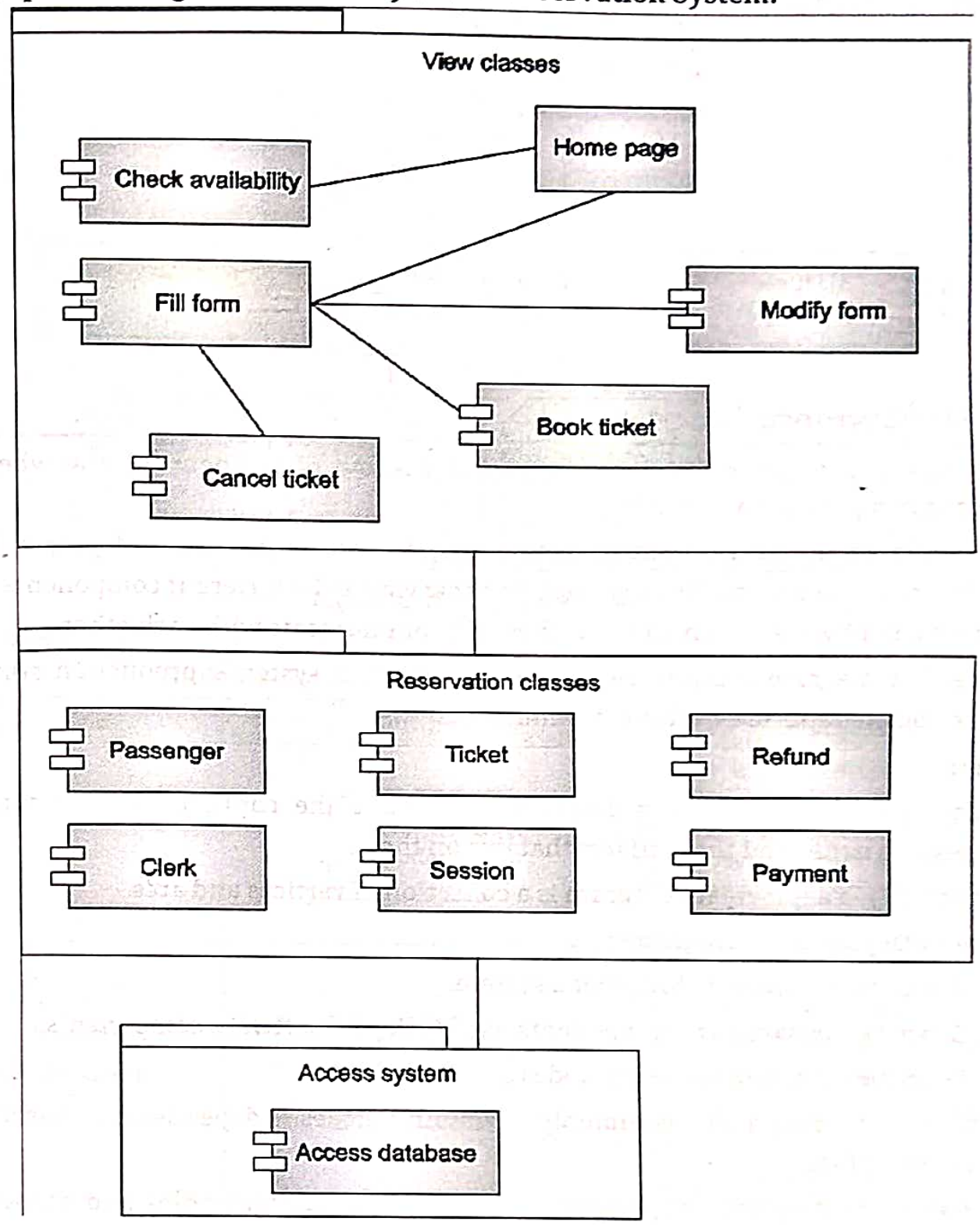



Fig. 2.66

**4. Component Diagram for ATM Transaction System:**

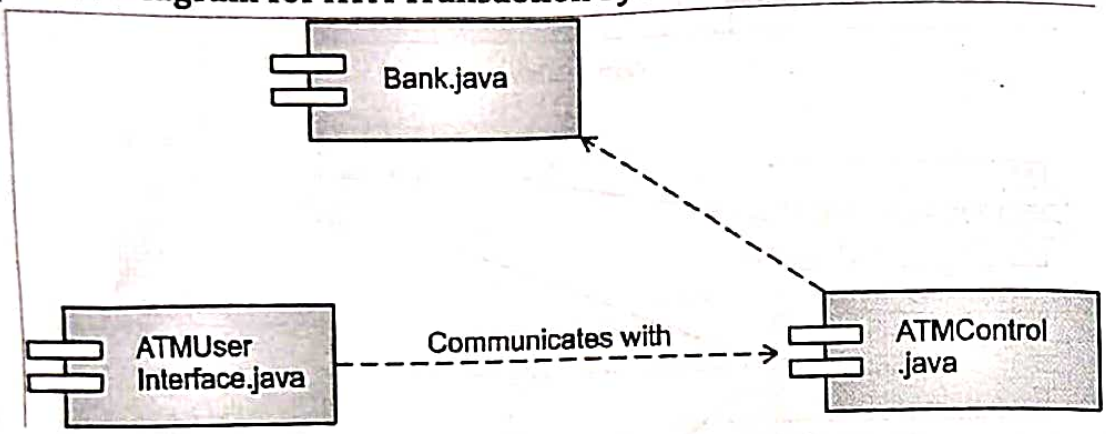



**Fig. 2.67**

## 2.4.3 Deployment Diagrams

- The deployment diagrams show the physical layout of the network and where th various components will reside.
- The deployment diagram shows how a system will be physically deployed in th hardware environment. Its purpose is to show where the different components of th system will physically run and how they will communicate with each other.
- Since, the diagram models the physical runtime; a system's production staff wi make considerable use of this diagram.

**Concept:**

- A deployment diagram is a diagram that shows the configuration of run tim processing nodes and the artifacts that live on them.
- Graphically, a deployment diagram is a collection of vertices and arcs.

**Purpose of Deployment Diagrams:**

1. Visualize hardware topology of a system.
2. Describe the hardware components used to deploy software components.
3. Describe runtime processing nodes.

- Deployment diagrams commonly contain nodes, dependency, associatio relationships etc.
- Like all other diagrams, deployment diagrams may contain notes and constraint Deployment diagrams may also contain artifacts, each of which must live on som node.
- With the UML, you use deployment diagrams to visualize the static aspect of the physical nodes and their relationships and to specify their details for construction, shown in Fig. 2.68.

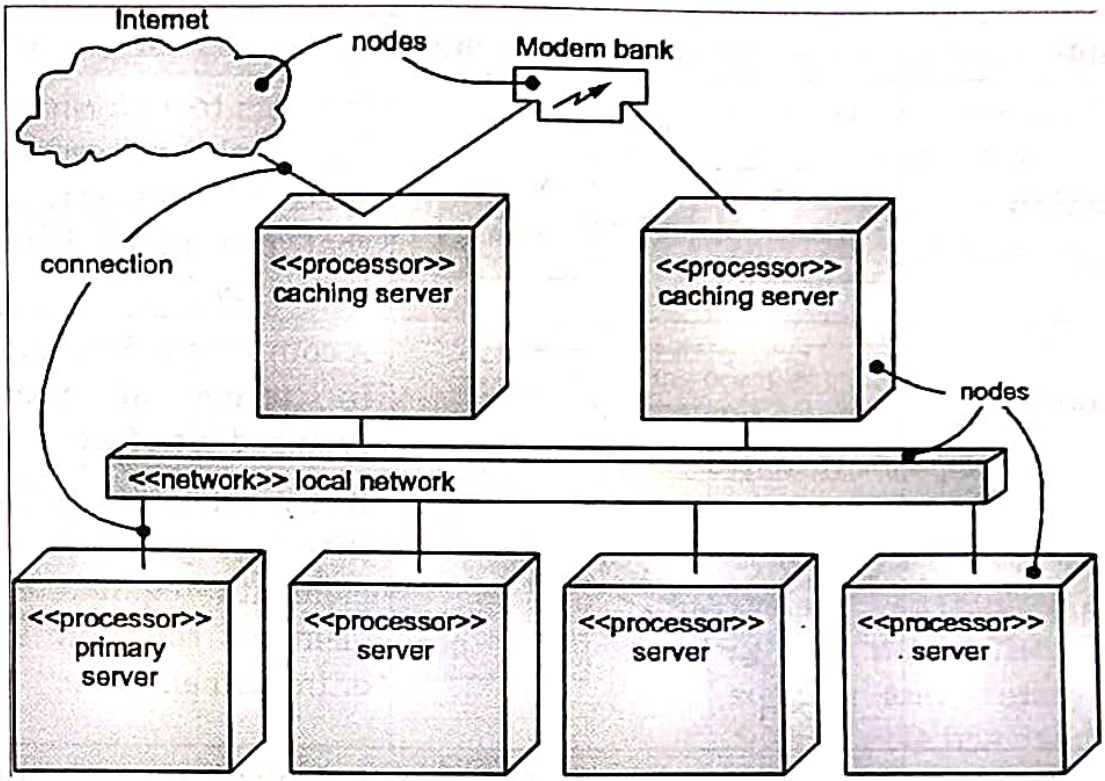
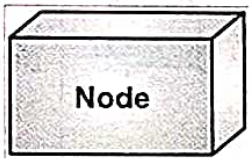
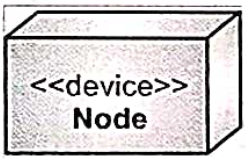
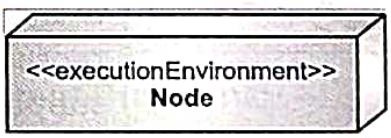
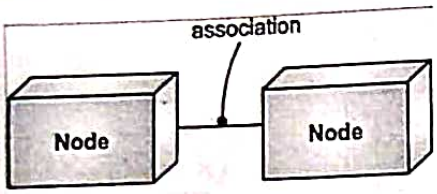
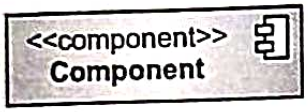
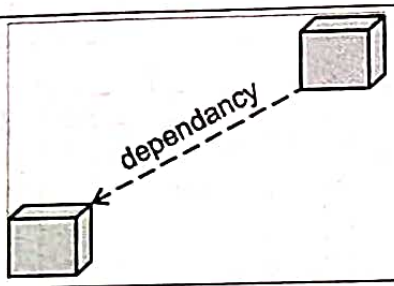
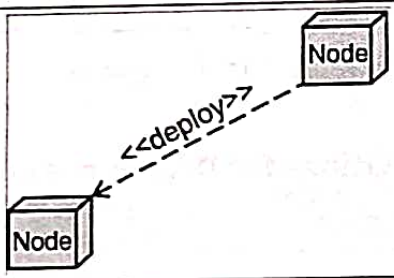
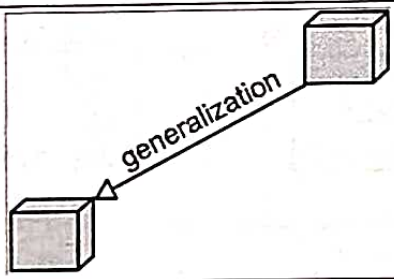
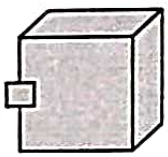
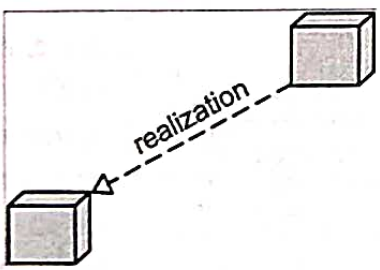


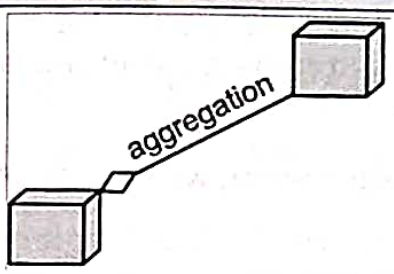
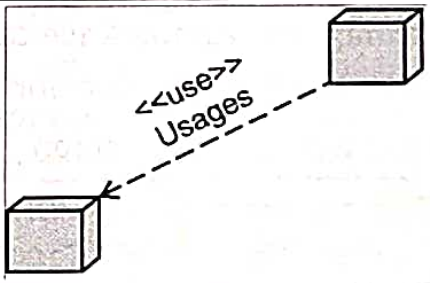

**Fig. 2.68: Deployment diagram**

**Notations:**

**Table 2.5: Basic Notations for Deployment Diagram**

| Name | Symbol | Description |
|---|---|---|
| **1. Node** |   | A node is a physical resource that executes code components. A node is a computational resource upon which artifacts may be deployed for execution. |
| **2. Device node** |   | A device node is a physical computational resource with processing capability upon which artifacts may be deployed for execution. Devices may be complex (i.e., they may consist of other devices). |
| **3. Execution Environment Node** |   | An execution environment node is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts. |

| Name | Symbol | Description |
|---|---|---|
| 4. Association |  | It refers to a physical connectio i.e., link between nodes. specifies a semantic relationshi that can occur between type instances. |
| 5. Component |  | A component defines its behavio in terms of provided an required interfaces. |
| 6. Dependency |  | It is a relationship that signifie that a single or a set of mode elements requires other mode elements for their specificatio or implementation. |
| 7. Deployment |  | A deployment is the allocation o an artifact or artifact instance t a deployment target. |
| 8. Generalization |  | It is a taxonomic relationshi between a more general classifie and a more specific classifier. |
| 9. Port |  | A Port may specify the services classifier provide (offer) to it environment as well as th services that a classifier expect (requires) of its environment. |
| 10. Realization |  | Realization is a specialize abstraction relationship betwee two sets of model elements, on representing a specification (th supplier) and the othe represents an implementation the latter (the client). |

| Name | Symbol | Description |
|---|---|---|
| 11. Aggregation |   | It is a kind of association that has one of its end marks shared as a kind of aggregation, meaning that it has a shared aggregation. |
| 12. Artifact |   | An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, a mail message. |
| 13. Interface |  | It is a kind of classifier that represents a declaration of a set of coherent public features and obligations. It specifies a contract; any instance of a classifier that realizes the interface must fulfill that contract. |
| 14. Note |  | It gives the ability to attach various remarks (comments) to elements. |
| 15. Usage |   | It is a relationship in which one element requires another element or set of elements for its full implementation or operation. |

**Common terms in Deployment Diagrams:**

1. **Nodes and Node Names:**

- Nodes are an important building block in modeling the physical aspects of a system.

- We use nodes to model the topology of the hardware on which our system execute A node typically represents a processor or a device on which artifacts may b deployed. Good nodes represent the vocabulary of the hardware in your solutio domain.
- When we design a software-intensive system, we have to consider both its logical an physical dimensions.
  (i) On the logical side, you will find classes, interfaces, collaborations, interaction and state machines.
  (ii) On the physical side, you will find artifacts (which represent the physic packaging of these logical things) and nodes (which represent the hardware o which these artifacts are deployed and executed).
- Every node must have a name that distinguishes it from other nodes. Name alone known as a simple name; a qualified name is the node name prefixed by the name the package in which that node lives. A node is typically drawn showing only i name.

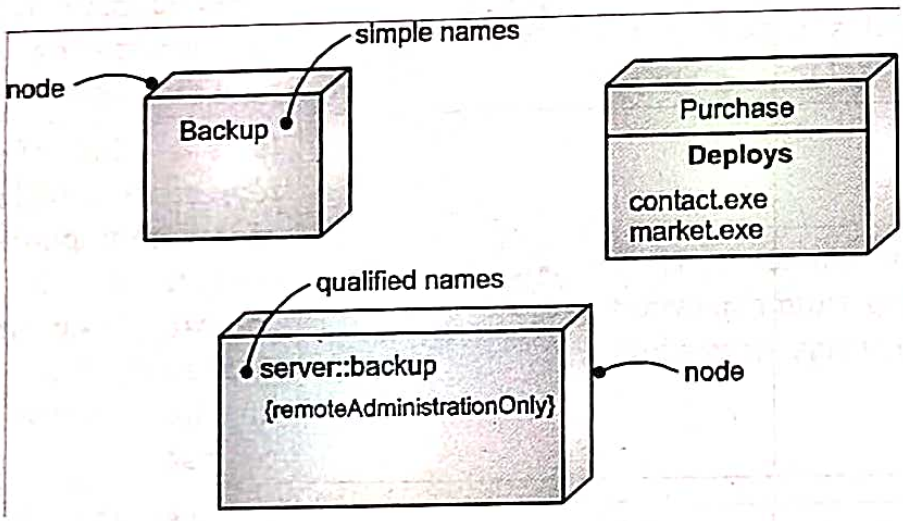



**Fig. 2.69: Nodes with simple and qualified names**

**2. Node Instances:**

- A node instance can be shown on a diagram. An instance can be distinguished from node by the fact that its name is underlined and has a colon before its base node type
- An instance may or may not have a name before the colon.

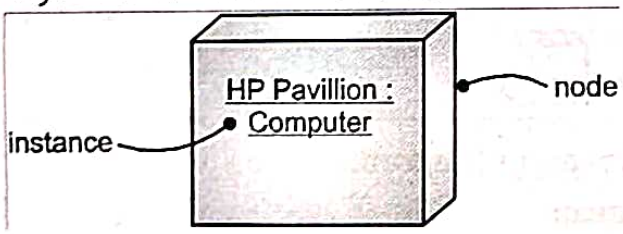



**Fig. 2.70: Named Instances of node**

### 3. Nodes and Components:

- In many ways, nodes are a lot like components i.e. both nodes and components have names. Both nodes and components may participate in dependency, generalization and association relationships. Both nodes and components may be nested.
- Both nodes and components may have instances and may be participants in interactions.
- However, there are some significant differences between Nodes and Components.
  (i) Components are things that participate in the execution of a system, while nodes are things that execute components.
  (ii) Components represent the physical packaging of otherwise logical elements, while modes represent the physical deployment of components.
- Fig. 2.71 shows, the relationship between a node and components it deploys can be shown explicitly by using a dependency relationship.

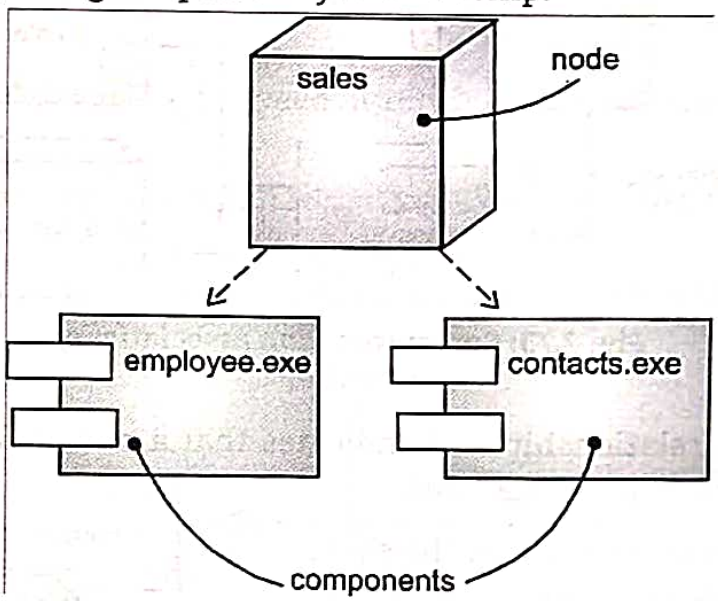



**Fig. 2.71: A node and components**

- A set of objects or components that are allocated to a node as a group is called a 'distribution unit'.

### 4. Node Stereotypes:

- A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc».
- These will display an appropriate icon in the top right corner of the node symbol.


**Fig. 2.72: Node Stereotype**

## 5. Connections (Relationship):

### (i) Association:

- The most common kind of relationship you will use among nodes is an association.
- Association refers to a physical connection or link between the nodes.
- A communication association between nodes indicates a communication pat between the nodes that allows components on the nodes to communicate with on another.
- A communication association is shown as a solid-line between nodes. Fig. 2.73 show that the Business-Processing Server has a communication association with th Desktop Client, Printer, and Database Server nodes.

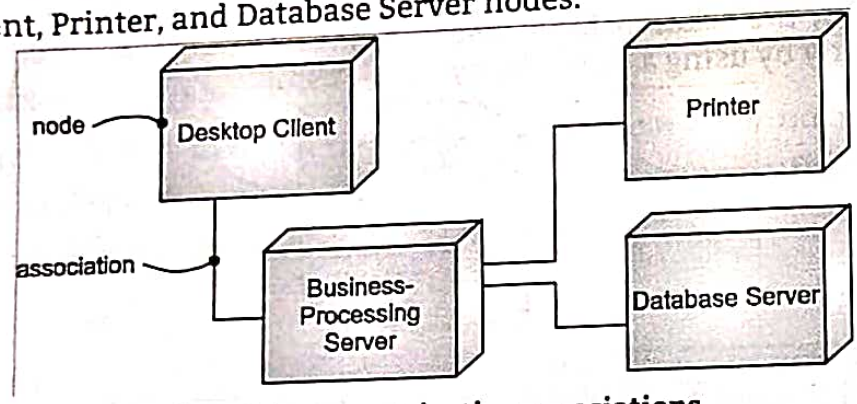



**Fig. 2.73: Communication associations**

### (ii) Dependency:

- A dependency is a relationship that indicates that a model element is in some wa dependent on another model element.
- All model elements must exist on the same level of abstraction or realization. A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing).
- The dependency of a node on components is depicted using a dashed line, This indicates that a node uses the services of the components that are executing on another node.
- For example, you can depict the dependency relationship between an Application Server node and a database server component, as shown in the Fig. 2.74.

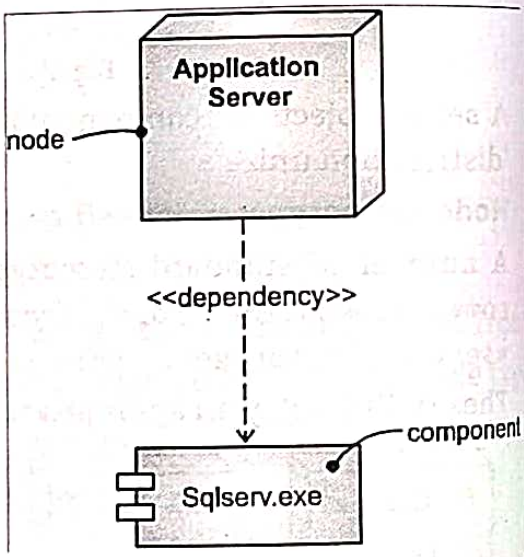



**Fig. 2.74: Dependency between node an component**

### (iii) Generalization:

- A generalization is a relationship between a more general element and a more specific element.
- Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent).

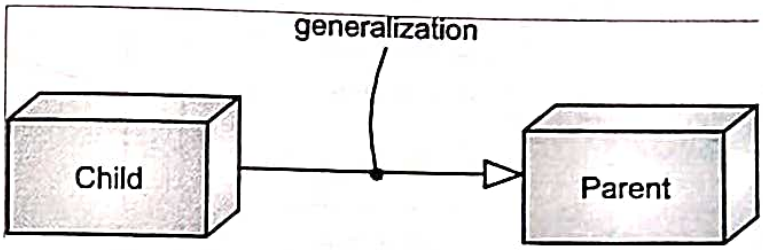



**Fig. 2.75: Example of generalization**

### (iv) Realization:

- Realization is a relationship between interfaces and classes or components that realize them.
- Realization defines a relationship in which one class specifies something that another class will perform.
- Example: The relationship between an interface and the class that realizes or executes that interface.

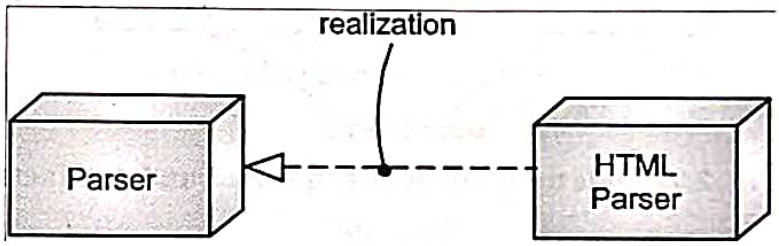



**Fig. 2.76: Realization**

### 6. Nodes and Artifacts:

- In many ways, nodes are a lot like artifacts: both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions.
- However, there are some significant differences between nodes and artifacts.
  - This first difference is the most important. Simply put, nodes execute artifacts; artifacts are things that are executed by nodes.
  - The second difference is a relationship among classes, artifacts, and nodes. In particular, an artifact is the demonstration of a set of logical elements, such as classes and collaborations, and a node is the location upon which artifacts are deployed.
  - A class may be marked by one or more artifacts, and, in turn, an artifact may be deployed on one or more nodes.

- Fig. 2.77 shows, the relationship between a node and the artifacts it deploys can b shown explicitly by using nesting. Most of the time, you won't need to visualize thes relationships graphically but will indicate them as a part of the node's specificatio for example, using a table.

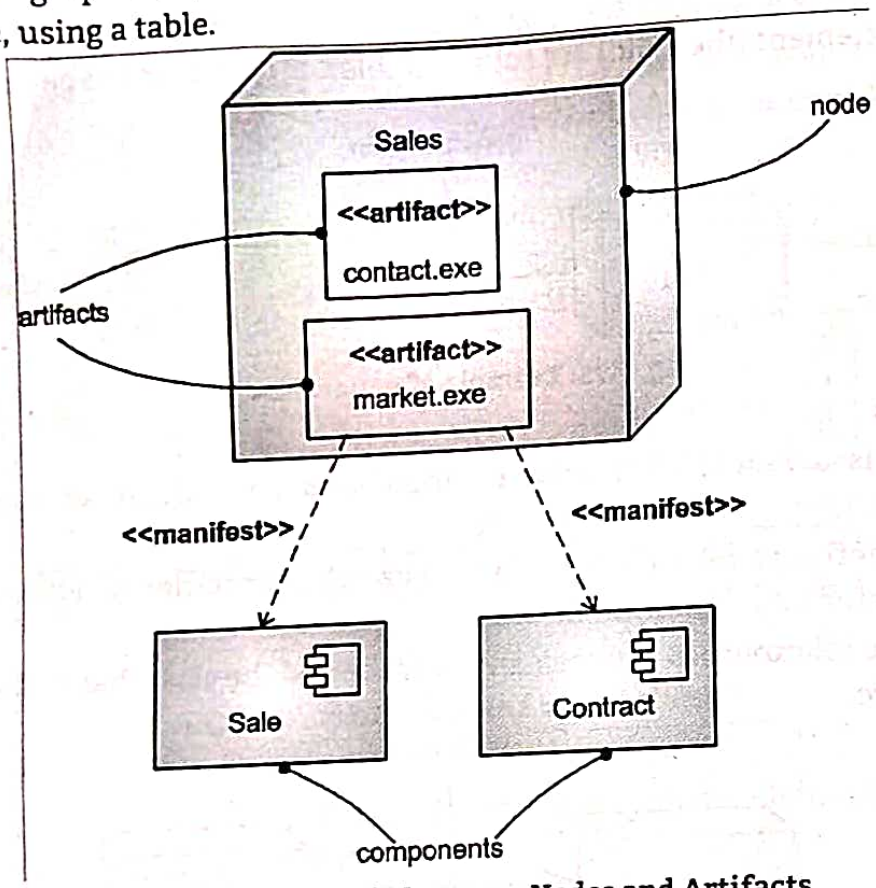



**Fig. 2.77: Relationship between Nodes and Artifacts**

**Examples:**

1. **Deployment Diagrams for Online Hospital Management System:**

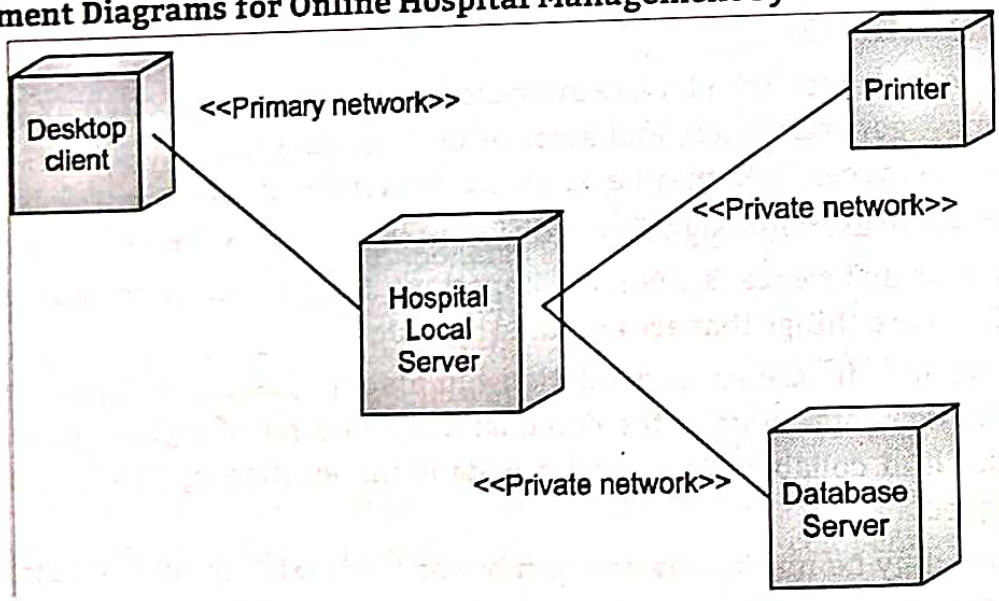



**Fig. 2.78**

. **Deployment Diagram for ATM Machine:**

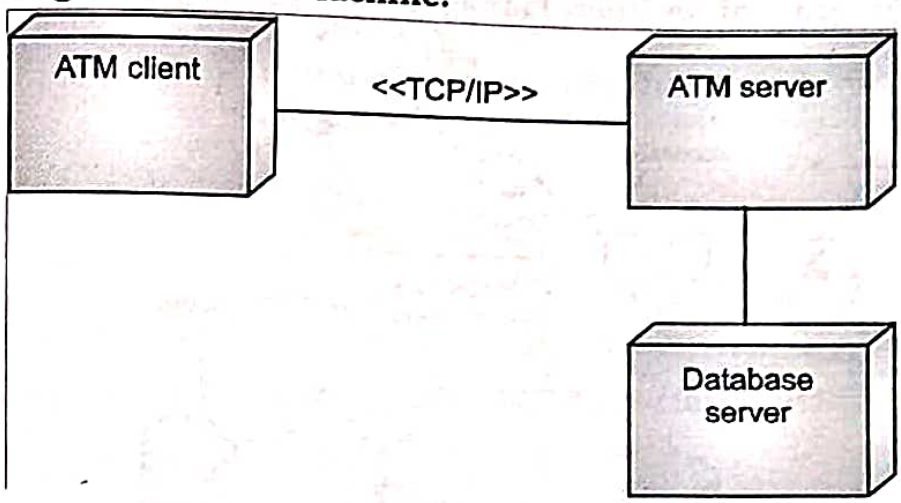



Fig. 2.79

. **Deployment Diagram for Online Shopping:**

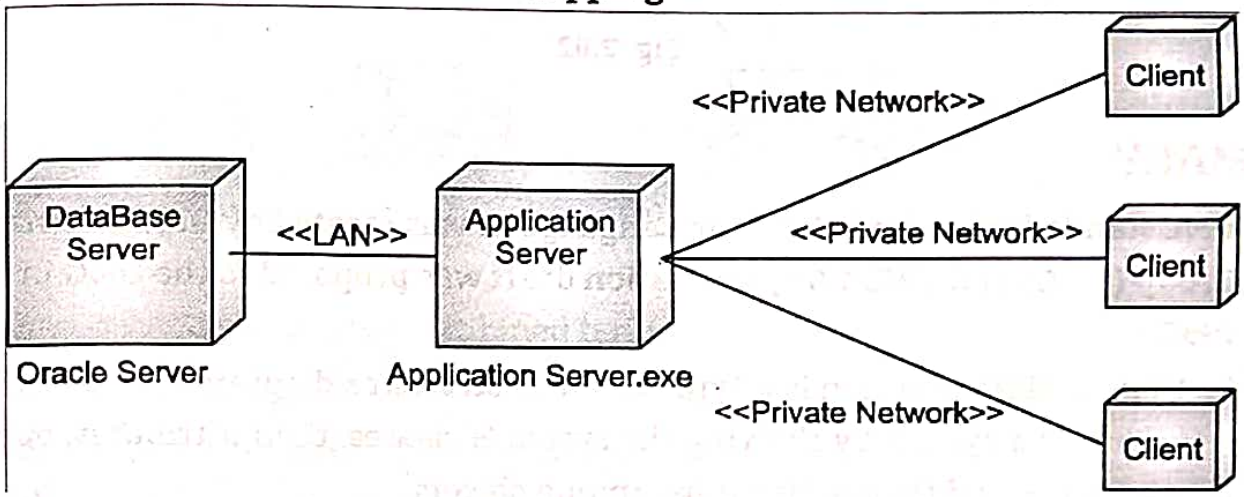



Fig. 2.80

. **Deployment Diagram for Railway Ticket Reservation System:**

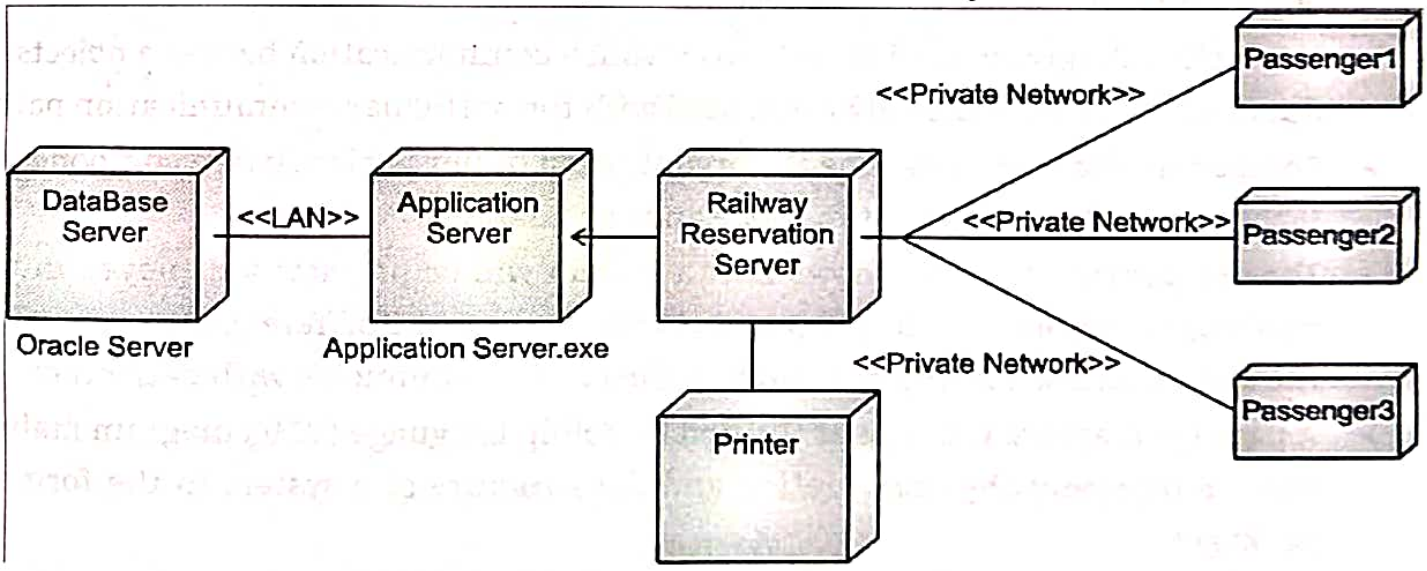



Fig. 2.81

## 5. Deployment Diagram of an Order Management System:

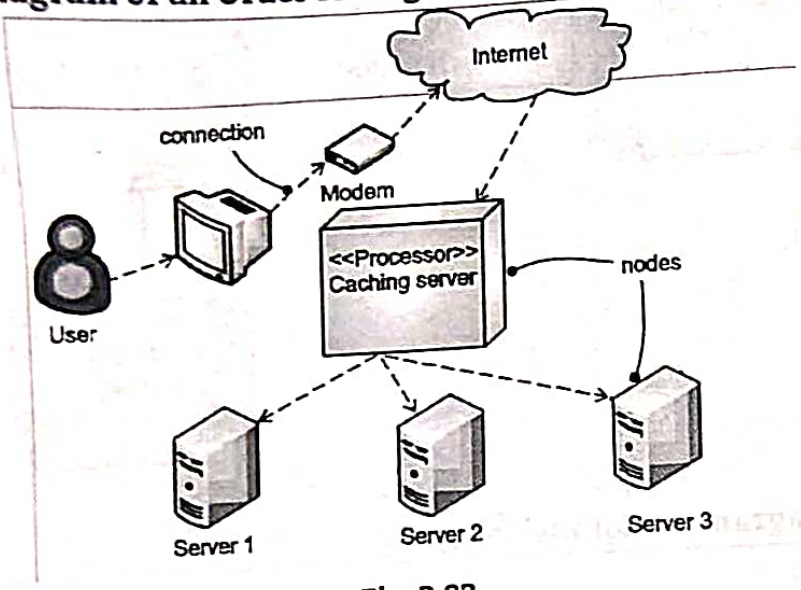



Fig. 2.82

## SUMMARY

- UML stands for Unified Modeling Language. It was created by Object Manageme Group (OMG) and UML 1.0 specification draft was proposed to the OMG in Janua 1997.
- In UML, a class diagram is a type of static structure diagram that describes t structure of a system by showing the system's classes, their attributes, operatio (or methods), and the relationships among objects.
- Activity diagram is a special type of state chart diagram which shows flow fro one activity to another within a system.
- Interaction diagrams used in UML to establish communication between objects. does not manipulate the data associated with the particular communication pat
- Component diagrams describe the organization of physical software componen including source code, run-time code and executables.
- The deployment diagram shows how a system will be physically deployed in t hardware environment. Its purpose is to show where the different components the system will physically run and how they will communicate with each other.
- A package diagram is a type of Unified Modeling Language (UML) diagram mai used to represent the organization and the structure of a system in the form packages.

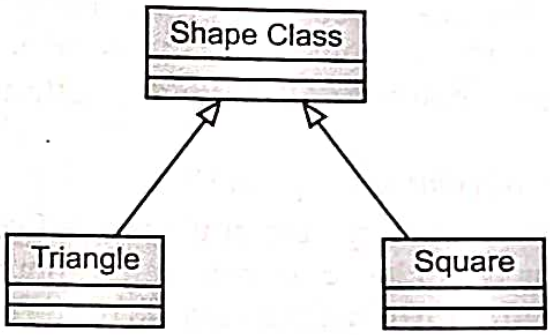
## HECK YOUR UNDERSTANDING

1. What type of core-relationship is represented by the symbols in the figure below?

(a) Aggregation (b) Dependency

(c) Generalization (d) Association

2. What type of relationship is represented by Shape class and Square?




(a) Realization (b) Generalization

(c) Aggregation (d) Dependency

3. Interaction Diagram is a combined term for ___ .

(a) Sequence Diagram + Collaboration Diagram

(b) Activity Diagram + State Chart Diagram

(c) Deployment Diagram + Collaboration Diagram

(d) None of the mentioned

4. Activity diagram, use case diagram, collaboration diagram and sequence diagram are considered as types of _____.

(a) non-behavioral diagrams (b) nonstructural diagrams

(c) structural diagrams (d) behavioral diagrams

5. A package diagram consists of the following?

(a) Package symbol

(b) Groupings of Use cases, classes, component

(c) Interface

(d) Package symbols, Groupings of Use cases, classes and components

**Answers**

| | | | | |
|---|---|---|---|---|
| 1. (a) | 2. (b) | 3. (a) | 4. (d) | 5(d) |

## PRACTICE QUESTIONS

**Q.I. Answer the following questions in short.**

1. Explain activity diagram with an example.
2. Explain common uses of interaction diagram.
3. Explain the use diagram with example.
4. What is UML? Which are characteristics of UML?
5. Define the thing. Explain types of things in UML.
6. Which are components of package diagram?

**Q.II. Answer the following questions.**

1. Explain which diagrams are called as an interaction diagram .Explain how the diagrams are used to model various aspects of system.
2. Explain generalization, include and extend relationship among Use Case wi example.
3. List and explain types of relationships in UML
4. Zoom car, a car rental agency has multiple offices/branches. Authenticat customers search for vehicles and can sent enquiry. Agency checks the availability of the car and intimates the status to the customer. Customer book t car by signing the terms and conditions. The customer can also avail the driv facility if required, by paying additional charges. The billing is done based on t type of car and distance travelled. The customer can pay on line through deb card/credit card.

   (i) Draw use case diagrams for explaining system features.
5. JNU has well – stocked library providing services to various members – stude research scholars, faculty members and visiting staff. The library has boo journals, periodicals, magazines, newspapers and CDs. A member can a book fo period of one week, journals and periodicals for two days and CD for one da A fine of ₹ 5/- per day will be charged for not returning on time.

   The rules for issuing number of books are as follows:

   Research scholars - 3
   Faculty members - 5
   Visiting staff - 3
   Students - 4

   To avail additional books every members has to obtain a special permission fr respective HODs.

   (i) Draw class diagram for the above case.

**Q.III. Write short notes on:**

1. Interaction
2. SwimLanes
3. Component diagram
4. Deployment diagram

**

UNIT 3

# Fundamentals of Project Management

## Objectives...

After reading this chapter, you will be able:

- To learn about Project Management Life cycle and Process.
- To understand the role of Project Manager.
- To get information of Quality Metrics.
- To know about Risk Management Process.
- To get knowledge of Linear Software Project Cost Estimation.
- To learn different methods of Cost Estimation such as COCOMO-I and II model and Delphi Cost Estimation.

## 3.1 OVERVIEW OF PROJECT MANAGEMENT

**Project:**

- A project is a unique venture to produce a set of deliverables within clearly specified time, cost and quality constraints.

**Characteristics of Project:**

- Projects are different from standard business operational activities as they:
  - **Are unique in nature:** They do not involve repetitive processes. Every project undertaken is different from the last, whereas operational activities often involve undertaking repetitive (identical) processes.
  - **Have a defined timescale:** Projects have a clearly specified start and end date within which the deliverables must be produced to meet a specified customer requirement.
  - **Have an approved budget:** Projects are allocated a level of financial expenditure within which the deliverables are produced, to meet a specified customer requirement.
  - **Have limited resources:** At the start of a project an agreed amount of labour, equipment and materials are allocated to the project.

- **Involve an element of risk:** Projects entail a level of uncertainty and therefore carry business risk.
- **Achieve beneficial change:** The purpose of a project is typically to improve an organization through the implementation of business change.
- **Have a Primary Customer or Sponsor:** Most projects have many interested parties or stakeholders, but someone must take the primary role of sponsorship. The project sponsor usually provides the direction and funding for the project.
- **Involves uncertainty:** As every project is unique, it is sometimes difficult to define the project's objectives clearly, estimate exactly how long it will take to complete or determine how much it will cost. External factors also cause uncertainty, such as a supplier going out of business or a project team member needing unplanned time off. This uncertainty is one of the main reasons project management is so challenging.

**Project Management:**

- Projects are organized and staffed by people of varying skills, responsibilities and roles. In order to perform their work, these people use processes and tools.
- Projects are constrained by many factors. The common ones are time, cost, resources, product requirements and quality. The ultimate goal of a project team is to deliver a product on time, within budget, that meets the product requirement and quality constraints.
- To achieve this goal the team must use effective methods to manage the people, processes and tools used for the project.
- Basic components of Project Management are skills, tools and management processes required to undertake a project successfully as shown below:

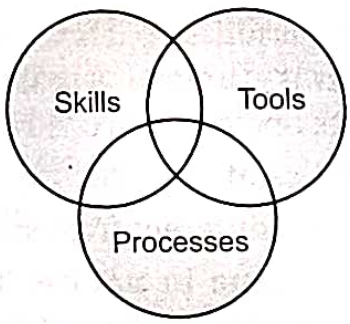



**Fig. 3.1: Project Management Component**

1. **Asset of Skills:** Specialist knowledge, skills and experience are required to reduce the level of risk within a project and thereby enhance its likelihood of success.
2. **A Suite of Tools:** Various types of tools are used by project managers to improve their chances of success. Examples include document templates, register, planning software, modeling software, audit checklists and review forms.
3. **A Series of Processes:** Various processes and techniques are required to monitor and control time, cost, quality and scope on projects. Examples include time management, cost management, quality management, change management, risk management and issue management.

**Project Constraints:**

- Like any human undertaking, projects need to be performed and delivered under certain constraints. Traditionally, these constraints have been listed as scope, time, and cost. These are also referred to as the Project Management Triangle, where each side represents a constraint. One side of the triangle cannot be changed without impacting the others.

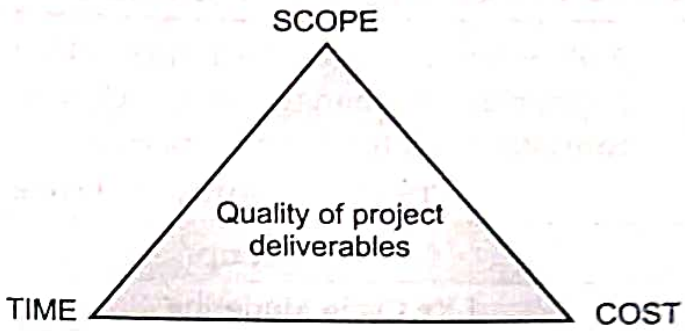



**Fig. 3.2: Project Constraints**

  - **Time:** For analytical purposes, the time required to produce a product or service is estimated using several techniques. One method is to identify tasks needed to produce the deliverables documented in a work breakdown structure or WBS. The work effort for each task is estimated and those estimates are rolled up into the final deliverable estimate.
  - **Cost:** Cost to develop a project depends on several variables including: labour rates,
    material rates, risk management, equipment, and profit.
  - **Scope:** Scope is requirement specified for the end result. The overall definition of what the project is supposed to accomplish, and a specific description of what the end result should be or accomplish can be said to be the scope of the project. A major component of scope is the quality of the final product. The amount of time put into individual tasks determines the overall quality of the project.

## 3.2 PROJECT MANAGEMENT LIFE CYCLE-IEEE LIFE CYCLE

- Project Management is the process by which a project is initiated, planned, controlled, and brought to a conclusion to support the accomplishment of business and system objectives.
- From beginning to end in a project, many activities and deliverables have to be managed. The aggregation of these management methods is called as Project Management Life Cycle (PMLC).
- IEEE 1074 provides a process for creating a Software Life Cycle Process (SLCP). The SLCP is defined as the project-specific description of the process that is based on a project's software life cycle (SLC) and the integral and project management processes used by the organization. These integral processes include configuration management, metrics, quality assurance, risk reduction, and the acts of estimating, planning, and training. It is primarily the responsibility of the project manager and the process architect for a given software project to create the SLCP.
- In the following section, we summarize the main processes and activities introduced by the standard and clarify its fundamental concepts using UML diagrams.

## 3.2.1 Processes and Activities

- A **Process** is a set of activities that is performed towards a specific purpose (e.g. requirements, management, delivery). IEEE 1074 includes six groups of processes consisting of a total of 17 processes (see Table 3.1).

**Table 3.1: Software Processes in the IEEE Standard 1074**

| Process Group | Processes |
|---|---|
| Life Cycle Modeling | Selection of Life Cycle Modeling |
| Project Management | Project Initiation<br>Project Monitoring and Control<br>Software Quality Management |
| Pre-development | Concept Exploration<br>System Allocation |
| Development | Requirements Analysis<br>Design<br>Implementation |
| Post-Development | Installation<br>Operation and Support<br>Maintenance<br>Retirement |
| Cross-Development | Verification and Validation<br>Software Configuration Management<br>Documentation Development<br>Training |

- These include, for example:
  - The *Requirements Process*, during which the client and the developers describe t system.
  - The *Design Process*, during which developers decompose the system in components.
  - The *Implementation Process*, during which developers realize each component.
  - The *Project Monitoring and Control Process*, during which management monito project progress and take corrective actions. Processes are composed of activities
- An **Activity** is a task or group of tasks that are assigned to project participants achieve a specific purpose. For example, the *Requirements Process* is composed three activities:
  1. *Define and Develop Software Requirements*, during which the functionality of t system is precisely defined.
  2. *Define Interface Requirements*, during which the interactions between the syste and the user are precisely defined.
  3. *Prioritize and Integrate Software Requirements*, during which all requirements a integrated for consistency, and prioritized by client preference.

- Tasks consume resources (e.g., personnel, time, money) and produce a work product. During planning, activities are decomposed into project specific tasks, are given a starting and ending date and assigned to a team or a project participant (see Figure 3.3). During the project, actual work is tracked against planned tasks, and resources are reallocated to respond to problems.

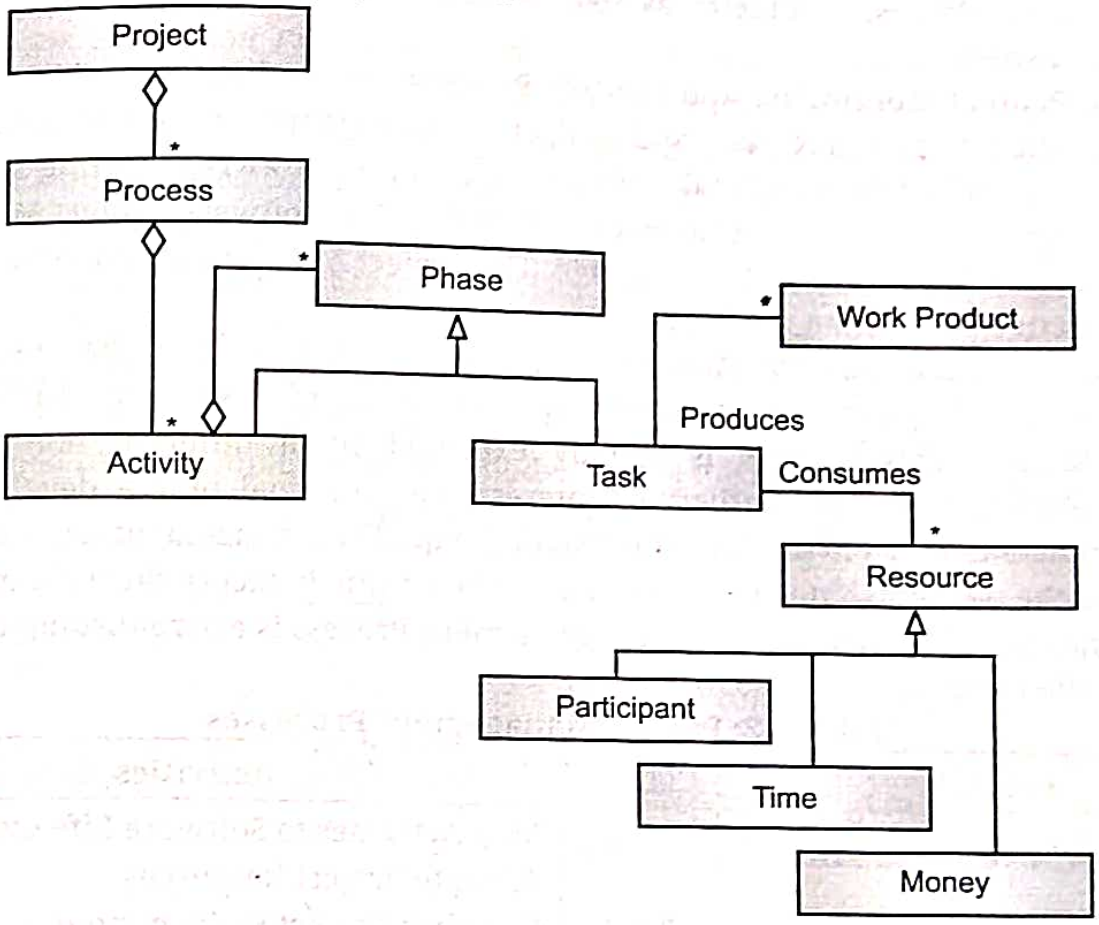



**Fig. 3.3: Processes, Activities, and Tasks (UML Class Diagram)**

## 3.2.2 Life Cycle Modeling

- During Life Cycle Modeling, management selects and orders the activities defined in IEEE 1074 for a specific project. Not all projects require the same activities and the same ordering. For example, projects that do not require a database need not execute the activity *Design Data Base*. Similarly, if developers adopt a prototyping approach, all development activities will occur in parallel. The selected life cycle model serves as input to the Project Initiation Process.

## 3.2.3 Project Management

- During Project Management, management initiates, monitors, and controls the project throughout project development life cycle. Project Management consists of following three processes:
  1. **The Project Initiation Process** creates the infrastructure for the project. During this process the task plan, schedule, budget, organization, and project environment

are defined. The project environment includes project standards, communication infrastructure, meeting and reporting procedures, development methods, and development tools. Most of the information generated during this process is documented in the Software Project Management Plan (SPMP). The Project Initiation Process completes as soon as a stable environment for the project is established.

2. **The Project Monitoring and Control Process** ensure that the project is executed according to the task plan and budget. If management observes any deviation from the schedule, it will take corrective action such as reallocating some of the resources, changing procedures, or replanning. The Software Project Management Plan is updated to reflect any change. The Project Monitoring and Control Process are active throughout the life cycle.
3. **The Software Quality Management Process** ensures that the system under construction meets the required quality standards (selected during Project Initiation). This process is usually executed by a different team than the developers to avoid conflicts of interest (i.e., the goal of the developers is to complete the system on time, the goal of the quality management team is to ensure that the system is not considered complete until it meets the required quality standard). The Software Quality Management Process is active throughout most of the life cycle.

**Table 3.2: Project Management Processes**

| Process | Clause* | Activities |
|---|---|---|
| Project Initiation | 3.1.3 | Map Activities to Software Life Cycle Model |
| | | Allocate Project Resources |
| | 3.1.4 | Establish Project Environment |
| | 3.1.5 | Plan Project Management |
| | 3.1.6 | |
| Project Monitoring and Control | 3.2.3 | Analyze Risks |
| | 3.2.4 | Perform Contingency Planning |
| | 3.2.5 | Manage the Project |
| | 3.2.6 | Retain Records |
| | 3.2.7 | Implement Problem Reporting Model |
| Software Quality Management | 3.3.3 | Plan Software Quality Management |
| | 3.3.4 | Define Metrics |
| | 3.3.5 | Manage Software Quality |
| | 3.3.6 | Identify Quality Improvement Needs |

(*: The 'Clause' column in this table and the other tables in this chapter is a clause number in IEEE 1074. This is a cross reference to the standards document as published in [IEEE 1074, 1995]).

**Pre-development:**

- During *Pre-development*, management (or marketing) and a client identify an idea or a need. This can be a new development effort (Greenfield Engineering), or a change to the interface of an existing system (interface engineering) or software replacement of an existing business process (re-engineering).
- The *System Allocation Process* establishes the system architecture, in particular the subsystem decomposition and identifies the hardware, software and operational requirements. Note that the subsystem decomposition is the foundation of the communication infrastructure among the project members.
- The requirements, subsystem decomposition and communication infrastructure are described in the **Problem Statement** (The statement of need mentioned in the IEEE 1074 is similar to the problem statement, but does not contain any project organization information) which serves as input into the development process. The Problem Statement is written in terms of scenarios and user interface sketches that are refined into a complete use case model during the *Requirements Analysis Process.*

**Table 3.3: Pre-development Processes**

| Process | Clause[a] | Activities |
|---|---|---|
| Concept Exploration | 4.1.3 | Map Activities to Software Life Cycle Model |
| | | Allocate Project Resources |
| | 4.1.4 | Establish Project Environment |
| | 4.1.5 | Plan Project Management |
| | 4.1.6 | Refine and Finalize the Idea or Need |
| | 4.1.7 | |
| System Allocation | 4.2.3 | Analyze Function |
| | 4.2.4 | Develop System Architecture |
| | 4.2.5 | Decompose System Requirements* |

(*. System requirements are decomposed into Hardware Requirements and Software Requirements. Hardware development is not addressed in the IEEE 1074 standard.)

**Development:**

- Development consists of the processes directed toward the construction of the system.
  - The *Requirements Analysis Process* takes the system requirements in terms of high-level functional requirements and produces a complete specification of the system.
  - The *Design Process* takes the architecture produced during the System Allocation Process and produces a coherent and well-organized representation of the system. The high level design is completed (Perform Architectural Design and Design Interfaces activities) and results into the refinement of the subsystem decomposition. This also includes the allocation of requirements to hardware and

software systems, the description of boundary conditions, the selection of off-th shelf components and the definition of design goals. The detailed design of eac subsystem is done during the Perform Detailed Design activity.

In Objectory, the *Design Process* results into the definition of design objects, the attributes and operations, and their organization into packages. By the end of th activity, all methods and their type signatures are defined. New classes a introduced to take into account non-functional requirements and compone specific details.

- The *Implementation Process* takes the design model and produces an equivale executable representation. The Implementation Process includes integrati planning and integration activities. Note that tests performed during this proce are independent of those performed during quality control or Verification a Validation.

**Table 3.4: Development Processes**

| Process | Clause | Activities |
|---|---|---|
| Requirements Analysis | 5.1.3 | Define and Develop Software Requirements |
| | 5.1.4 | Define Interface Requirements |
| | 5.1.5 | Prioritize and Integrate Software Requiremen |
| Design | 5.2.3 | Perform Architectural Design |
| | 5.2.4 | Design Data Base (If Applicable) |
| | 5.2.5 | Design Interfaces |
| | 5.2.6 | Select or Develop Algorithm (If Applicable) |
| | 5.2.7 | Perform Detailed Design |
| Implementation | 5.3.3 | Create Test Data |
| | 5.3.4 | Create Source |
| | 5.3.5 | Generate Object Code |
| | 5.3.6 | Create Operating Documentation |
| | 5.3.7 | Plan Integration |
| | 5.3.8 | Perform Integration |

**Post-development:**

- Post-development consists of Installation, Maintenance, Operation, Support Retirement processes.
- During *Installation*, the system software is distributed and installed at the client The installation terminates in the client acceptance test according to the cri defined in the project agreement.
- *Operation and Support* are concerned with user operation of the system and training.

- *Maintenance* is concerned with the resolution of software errors, faults and failures after the delivery of the system. *Maintenance* requires a ramping of the software life cycle processes and activities into a new project.
- *Retirement* removes an existing system terminating its operations or support. *Retirement* takes place when the system is upgraded or replaced by a new system. To ensure a smooth transition between the two systems, both systems are often run in parallel until the users get used to the new system.

**Table 3.5: Post-development Processes**

| Process | Clause | Activities |
|---|---|---|
| Installation | 6.1.3 | Plan Installation |
| | 6.1.4 | Distribution of Software |
| | 6.1.5 | Installation of Software |
| | 6.1.7 | Accept Software in Operational Environment |
| Operation and Support | 6.2.3 | Operation the System |
| | 6.2.4 | Provide Technical Assistance and Consulting |
| | 6.2.5 | Maintain Support Request Log |
| Maintenance | 6.3.3 | Reapply Software Life Cycle |
| Retirement | 6.4.3 | Notify Users |
| | 6.4.4 | Conduct Parallel Operations (If applicable) |
| | 6.4.5 | Retire System |

**Cross-development (Integral Processes):**

- Several processes take place during the complete duration of the project. These are integral processes, which we call cross-development processes.
- Cross-development process includes *Validation and Verification, Software Configuration Management, Documentation Development, and Training.*

**Table 3.6: Cross-development Processes (called Integral Processes in IEEE 1074)**

| Process | Clause | Activities |
|---|---|---|
| Verification and Validation | 7.1.3 | Plan Verification and Validation |
| | 7.1.4 | Execute Verification and Validation Tasks |
| | 7.1.5 | Collect and Analyze Metric Data |
| | 7.1.6 | Plan Testing |
| | 7.1.7 | Develop Test Requirements |
| | 7.1.8 | Execute the Tests |
| Software Configuration Management | 7.2.3 | Plan Configuration Management |
| | 7.2.4 | Develop Configuration Identification |
| | 7.2.5 | Perform Configuration Control |
| | 7.2.6 | Perform Status Accounting |

| Process | Clause | Activities |
|---|---|---|
| Documentation Development | 7.3.3 | Plan Documentation |
| | 7.3.4 | Implement Documentation |
| | 7.3.5 | Produce and Distribute Documentation |
| Training | 7.4.3 | Plan the Training Program |
| | 7.4.4 | Develop Training Materials |
| | 7.4.5 | Validate the Training Program |
| | 7.4.6 | Implement the Training Program |

- *Verification and Validation* includes verification and validation tasks. Verification tasks focus on showing that the work products and the system comply with the specification. Verification includes reviews, audits, and inspections. Validation tasks focus on ensuring that the system addresses the client's need. Validation includes system testing, beta testing, and client acceptance testing.
- Verification and Validation activities occur throughout the life cycle with the intent of detecting anomalies as early as possible. For example, in Objectory, each model is reviewed against a checklist at the end of the process that generated it. The review of a model say the design model, may result in the modification of models generated in other processes, say the analysis model.
- The activity *Collect and Analyze* Metric Data may also serve for future projects and contribute to the knowledge of the organization.
- The activities *Plan Testing and Develop Test Requirements* can be initiated as early as the completion of the requirements analysis. In large projects, these tasks are performed by different participants than the developers.

## 3.3 QUALITY METRICS

### 3.3.1 Software Quality

- Software quality is the degree of conformance to requirements and expectations. Quality software is reasonably **bug or defect** free, delivered on time and within budget, meets requirements and/or expectations and is maintainable.
- The three aspects of software quality are:
  1. Functional Quality
  2. Structural Quality
  3. Process Quality
- Let us see more detail of each quality.
  1. **Functional Quality** aims to ensure your software operates as intended by checking if the application performs its specified functions correctly. The attributes of functional quality are:

- *Meeting the specified requirements:* Whether they come from the project's sponsors or the software's intended users. Since requirements commonly change throughout the development process, achieving this goal requires the development team to understand and implement the correct requirements throughout, not just those initially defined for the project.
- *Creating software that has few defects:* Among these are bugs that reduce the software's reliability, compromise its security, or limit its functionality. Achieving zero defects is too much to ask for most projects, but users are rarely happy with software they perceive as buggy.
- *Good enough performance:* From a user's point of view, there is no such thing as a good, slow application.
- *Ease of learning and ease of use:* To its users, the software's user interface is the application, and so these attributes of functional quality are most commonly provided by an effective interface and a well-thought-out user workflow. The aesthetics of the interface is important especially in consumer applications.
- Software testing primarily focusses on functional quality. All of the criteria mentioned above can be checked to some extent, hence testing plays an important role in assuring functional quality.

2. **Structural Quality** means that the code itself is well-structured. It is hard to test as compared to functional quality.

- The attributes of this type of quality include:
  - ***Code testability***: Is the code organized in a way that makes testing easy?
  - ***Code maintainability***: How easy is it to add new code or change existing code without introducing bugs?
  - ***Code understandability***: Is the code readable? Is it more complex than it needs to be? These have a large impact on how quickly new developers can begin working with an existing code base.
  - ***Code efficiency***: Especially in resource-constrained situations, writing efficient code can be critically important.
  - ***Code security***: Does the software allow common attacks such as buffer overruns and SQL injection? Is it insecure in other ways?

3. **Process Quality** is also critically important. The quality of the development process significantly affects the value received by users, development teams, and sponsors, and so all three groups have a stake in improving this aspect of software quality.

- The most obvious attributes of process quality include these:
  - ***Meeting delivery dates:*** Was the software delivered on time?
  - ***Meeting budgets:*** Was the software delivered for the expected amount of money?
  - ***A repeatable development process that reliably delivers quality software:*** If a process has the first two attributes, software delivered on time and on budget but causes the development team so much stress that some its best members quit, then it is not a quality process. True process quality means being consistent from one project to the next.

## 3.3.2 Software Quality Attributes

- Much of a software architect's life is spent designing software systems to meet a set of quality attribute requirements. General software quality attributes include:
  - **Correctness:** A software system is expected to meet the explicitly specified functional requirements and the implicitly expected non-functional requirements. If a software system satisfies all the functional requirements, the system is said to be correct.
  - **Reliability:** It is difficult to construct large software systems which are correct. A few functions may not work in all execution scenarios, and, therefore, the software is considered to be incorrect. However, the software may still be acceptable to customers because the execution scenarios causing the system to fail may not frequently occur when the system is deployed. Moreover, customers may accept software failures once in a while. Customers may still consider an incorrect system to be reliable if the failure rate is very small and it does not adversely affect their mission objectives. Reliability is a customer perception, and an incorrect software can still be considered to be reliable.
  - **Efficiency:** Efficiency concerns to what extent a software system utilizes resources, such as computing power, memory, disk space, communication bandwidth, and energy. A software system must utilize as few resources as possible to perform its functionalities.
  - **Integrity:** A system's integrity refers to its ability to withstand attacks on its security. In other words, integrity refers to the extent to which access to software or data by unauthorized persons or programs can be controlled.
  - **Usability:** A software system is considered to be usable if human users find it easy to use. Users put much emphasis on the user interface of software systems. Without a good user interface a software system may fail even if it possesses many desired qualities.
  - **Maintainability** refers to how easily and inexpensively the maintenance tasks can be performed. For software products, there are three categories of maintenance activities: Corrective, Adaptive, and Perfective.
    - **Corrective Maintenance** is a post release activity, and it refers to the removal of defects existing in the in-service software. The existing defects might have been known at the time of release of the product or might have been introduced during maintenance.
    - **Adaptive Maintenance** concerns adjusting software systems to changes in the execution environment.
    - **Perfective Maintenance** concerns modifying a software system to improve some of its qualities.
- **Testability:** It is important to be able to verify every requirement, both explicitly stated and simply expected. Testability means the ability to verify requirements.

every stage of software development, it is necessary to consider the testability aspect of a product.

- **Flexibility:** Flexibility is reflected in the cost of modifying an operational system. As more and more changes are effected in a system throughout its operational phase, subsequent changes may cost more and more.
- **Portability** is important for developers because a minor adaptation of a system can increase its market potential. Moreover, portability gives customers an option to easily move from one execution environment to another to best utilize emerging technologies in furthering their business.
- **Reusability** saves the cost and time to develop and test the component being reused. In the field of scientific computing, mathematical libraries are commonly reused.
- **Interoperability**: Today's software systems are coupled at the input–output level with other software systems. Intuitively, interoperability means whether or not the output of one system is acceptable as input to another system. It is likely that the two systems run on different computers interconnected by a network.

## 3.3.3 Software Quality Metrics and Indicators

- In general, for most software quality assurance systems the common software metrics that are checked for improvement are explained below:

**Software Metric Description:**

1. **Fan-in/Fan-out:** Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
2. **Length of Code:** This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
3. **Cyclomatic Complexity:** This is a measure of the control complexity of a program. This control complexity may be related to program understandability. Cyclomatic complexity counts the number of independent paths through a program. For instance, each "if" statement adds one extra code path. The higher the cyclomatic complexity, the harder it is to understand a program. Moreover, the more paths there are, the more test cases need to be written to achieve decent code coverage. The average cyclomatic complexity per function is an indicator that enables comparisons of complexity between programs. It is part of the "Maintainability" attribute.
4. **Code Duplication**: Sometimes, it is very tempting for a software engineer to copy some piece of code and make some small modifications to it instead of generalizing

functionality. The drawback of code duplication is that if one part of the code mu be changed for whatever reason (solving a bug or adding missing functionality), is very likely that the other parts ought to be changed as well. If nobody notic code duplication will lead to rework in the long term. This has a negative effect "Maintainability".

5. **Compiler Warnings:** In order to execute a software program on a computer it fi must be compiled or interpreted. Compilers/interpreters generate errors a warnings. Errors must be fixed otherwise the program cannot run. Warnings the other hand do not necessarily need to be solved. However, some compil warnings indicate serious program flaws. Leaving these unresolved has probab impact on the "Reliability" of the code. Apart from this, most compilers also wa about portability issues. So this metric can also mapped to "Transferability" most cases.
6. **Length of Identifiers:** This is a measure of the average length of identifie (names for variables, classes, methods, etc.) in a program. The longer identifiers, the more likely they are to be meaningful and hence the mo understandable the program.
7. **Depth of Conditional Nesting:** This is a measure of the depth of nesting of statements in a program. Deeply nested if-statements are hard to understand a potentially error-prone.
8. **Fog Index:** This is a measure of the average length of words and sentences documents. The higher the value of a document's Fog index, the more difficult document is to understand.
9. **Code Coverage:** The code coverage metric indicates how many lines of code executable branches in the code have been touched during the unit test runs. T lower the coverage, the lower the quality of the performed unit tests. Co coverage is an indicator of both "Functional Suitability" and "Reliability".

## 3.4 RISK

**Risk and its Types:**

- A risk is something you would like not occur. Risks may threaten the project, t software that is being developed or the organization.
- Following table is showing different category of risks.

**Table 3.7: General Categories of Risk**

| Generic Risks | Product-Specific Risks | |
|---|---|---|
| Project Risks | Product Risks | Business Risks |

1. **Generic risks** are potential threats to every software project. Some examples generic risks are changing requirements, losing key personnel, or bankruptcy of t software company or of the customer.

2. **Product-specific risks** can be distinguished from generic risks because they can only be identified by those with a clear understanding of the technology, the people, and the environment of the specific product. An example of a product-specific risk is the availability of a complex network necessary for testing.

- Generic and Product-specific risks can be further divided into Project, Product, and Business risks.
  1. **Project Risks:** Risks are threats that affect the project schedule or resources. It is related to loss of personnel or funds allocated to the project. An example of a project risk is the loss of an experienced and talented staff. Finding an equivalent staff with appropriate skills and experience may take a long time and, consequently, the software design will take longer to complete.
  2. **Product Risks:** Risks that affect the quality or performance of the software. An example of a product risk is the failure of a purchased component to perform as expected. This may affect the overall performance of the system so that it is slower than expected.
  3. **Business Risks:** Risks that threaten the viability of the software. It affects the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product will affect the sales of existing software products.
- The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed. However, there are also common risks that are not related to the type of software being developed and these can occur in any project. Some of these common risks are shown in following table:

**Table 3.8: List of Risks**

| Sr. No. | Risk Category | Risk Description |
|---|---|---|
| 1. | Requirements | • The requirements have not been clearly specified.<br>• The requirements specified do not match the customer's needs.<br>• The requirements specified are not measurable. |
| 2. | Benefits | • The business benefits have not been identified.<br>• The business benefits are not quantifiable.<br>• The final solution delivered does not achieve the required benefits. |
| 3. | Schedule | • The schedule does not provide enough time to complete the project.<br>• The schedule does not list all of the activities and tasks required.<br>• The schedule does not provide accurate dependencies. |

| Sr. No. | Risk Category | Risk Description |
|---|---|---|
| 4. | Budget | • The project exceeds the budget allocated.<br>• There is unaccounted expenditure on the project.<br>• There are no resources accountable for recording project spends. |
| 5. | Deliverables | • The deliverables required by the project are not clearly defined.<br>• Clear quality criteria for each deliverable have not been defined.<br>• The deliverable produced does not meet the quality criteria defined. |
| 6. | Scope | • The scope of the project is not clearly outlined.<br>• The project is not undertaken within the agreed scope.<br>• Project changes negatively impact on the project. |
| 7. | Issues | • Project issues are not resolved within an appropriate timescale.<br>• Similar issues continually reappear throughout the project.<br>• Unresolved issues become new risks to the project. |
| 8. | Suppliers | • The expectations for supplier delivery are not defined.<br>• Suppliers do not meet the expectations defined.<br>• Procurement delays impact on the project delivery timescales. |
| 9. | Acceptance | • The criteria for accepting project deliverables are not clearly defined.<br>• Customers do not accept the final deliverables of the project.<br>• The acceptance process leaves the customer dissatisfied. |
| 10. | Communication | • Lack of controlled communication causes project issues.<br>• Key project stakeholders are left in the dark about progress. |
| 11 | Resource | • Staff allocated to the project is not suitably skilled.<br>• There is insufficient equipment to undertake the project.<br>• There is a shortage of materials available when required. |

## 3.4.1 Risk Management Process

- Risk management is a series of steps whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or a major source of expensive rework. (Boehm, 1989).
- Risk management is one of the most important jobs for a project manager. Risk management involves expecting risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks.
- The Risk Management Process is an iterative process that continues throughout the project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes available, you have to reanalyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.
- An outline of the process of risk management is illustrated as follows. It involves several stages:

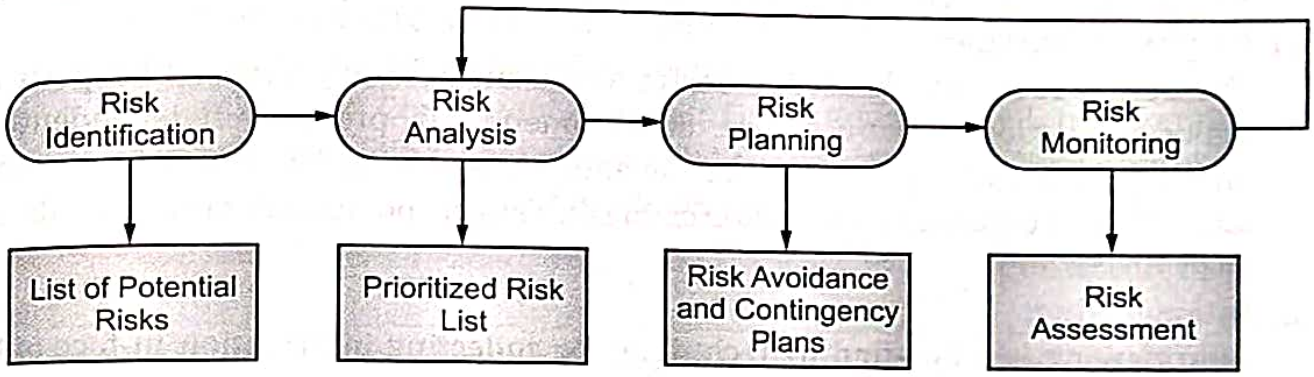



**Fig. 3.4: Risk Management Process**

1. **Risk Identification:** You should identify possible project, product, and business risks.
2. **Risk Analysis:** You should assess the likelihood and consequences of these risks.
3. **Risk Planning:** You should make plans to address the risk, either by avoiding it or minimizing its effects on the project.
4. **Risk Monitoring:** You should regularly assess the risk and your plans for risk mitigation and revise these when you learn more about the risk.

- Let us see more detail of each stage:

**1. Risk Identification:**

- Risk identification is the first stage of the risk management process. Risk identification may be a team process where a team get together to brainstorm possible risks. Alternatively, the project manager may simply use his or her experience to identify the most probable or critical risks.

- It is concerned with:
  - Identifying the risks.
  - What type of risk it is?
  - To which it is associated?
  - From where it is derived?
  - Possible threats of risk.
- There are several other tools and techniques also for identifying risks. Five comm information gathering techniques for risk identification include Brainstorming, Delp technique, Interviewing, Root Cause Analysis, and SWOT Analysis.

**1) Brain Storming:**

It is a technique by which a team attempts to generate ideas or find solutions fo specific by gathering ideas spontaneously and without judgment. This approa can help the group create a comprehensive list of risks to address later in t qualitative and quantitative risk analysis process. An experienced facilita should run the brainstorming session and introduce new categories of potent risks to keep the ideas flowing. After the ideas are collected, the facilitator c group and categorize the ideas to make them more manageable.

**1) Delphi Technique:**

The Delphi Technique is used to derive a consensus among a panel of experts w make predictions about future developments. It provides independent a anonymous input regarding future events. Uses repeated rounds of questioni and written responses and avoids the biasing effects possible in oral methods, su as brainstorming.

**2) Interviewing:**

Interviewing is a fact-finding technique for collecting information in face-to-fa phone, e-mail, or instant messaging discussions. Interviewing people with simi project experience is an important tool for identifying potential risks.

**3) Root Cause Analysis:**

Root causes are determined for the identified risks. These root causes are furth used to identify additional risks.

**4) SWOT Analysis:**

SWOT Analysis (Strengths, Weaknesses, Opportunities, and Threats) can also used during risk identification. It helps identify the broad negative risks that ap to a project. Applying SWOT to specific potential projects can help identify t broad risks and opportunities that apply in that scenario.

- Some other techniques for risk identification are:

**1) Use of Checklists:**

The list of risks that have been encountered in previous projects provide t meaningful template for understanding risks in current projects. It is important analyze project assumptions to make sure that they are valid. Incomple inaccurate or inconsistent assumptions might lead to identifying more risks.

**2) Diagramming Technique:**

This method includes using cause and effect diagrams or fishbone diagrams, flow charts and influence diagrams. Fishbone diagrams help you trace problems back to their root cause. Process flow charts are diagrams that show how different parts of the system interrelate.

**3) The Risk Register:**

The main output of the risk identification process is a list of identified risks and other information needed to begin creating a risk register. Once identified the risks should be logged on the project risk register. There are varying approaches to risk log contents, but general risks logs should contain:

- **Risk ID**: A number or reference uniquely identifying the risk.
- **Risk Description**: A description of the risk including cause and effect.
- **Identification Date:** The date the risk was identified.
- **Risk owner** : The person responsible for managing the risk to close.
- **Probability/Likelihood:** The predicted likelihood that the risk will be realized. This is either expressed as a number or on a scale such as high/medium/low.
- **Impact:** The predicted impact the risk could have on the project if it is realized. This is either expressed as a number or on a scale such as high/medium/low.
- **Risk Status:** Whether the risk is open or closed.
- **Mitigating and Contingent Actions:** The actions that will be taken to respond to the risk including what action will be taken if the risk does occur.

**Table 3.9: Risk Log**

| ID | Date raised | Risk description | Likelihood | Impact | Severity | Owner | Mitigating action | Contingent | Progress on actions | Status |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 12/12/23 | There is a risk that assets may not be completed in time to meet production schedules | Low | High | Amber | S Scott | Agree writing days in advance, reallocate writer's other work. Agree to stagger delivery of chapters so that editing can start earlier. | Increase duration of printing schedules and move from 4 col to 2 col. | Update 13/12/2023 mitigation actions implemented | Open |

**Case Study:**

- Following table gives some examples of possible risks identification in different categories.

**Table 3.10: Possible Risks in Risk Identification**

| Sr. No. | Risk Type | Associated /Derived From | Possible Threats |
|---|---|---|---|
| 1. | Technology Risks | Derive from the software or hardware technologies that are used to develop the system | • The database used in the system cannot process as many transactions per second as expected. |

| Sr. No. | Risk Type | Associated /Derived From | Possible Threats |
|---|---|---|---|
| | | | • Reusable software component contain defects that mean the cannot be reused as planned. |
| 2. | People Risks | Associated with the people in the development team. | • It is impossible to recruit staf with the skills required.<br>• Key staff is ill and unavailabl at critical times.<br>• Required training for staff i not available. |
| 3. | Organizational Risks | Derive from the organizational environment where the software is being developed. | • The organization restructured so that differen management is responsible fo the project.<br>• Organizational financia problems force reductions i the project budget. |
| 4. | Tools Risks | Derive from the software tools and other support software used to develop the system. | • The code generated by softwar code generation tools inefficient.<br>• Software tools cannot wor together in an integrated way. |
| 5. | Requirements Risks | Risks that derive from changes to the customer requirements and the process of managing the requirements change. | • Changes to requirements tha require major design rewor are proposed.<br>• Customers fail to understan the impact of requirement changes. |
| 6. | Estimation Risks | Risks that derive from the management estimates of the resources required to build the system. | • The time required to develo the software is under estimated.<br>• The rate of defect repair i underestimated.<br>• The size of the software i underestimated. |

## 2. Risk Analysis:

- During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk. To do this you have to rely on own judgment and experience of previous projects and the problems that arose in them. It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, you should assign the risk to one of the number of bands:

  1) The probability of the risk might be assessed as:
     - Very low (<10%)
     - Low (10–25%)
     - Moderate (25–50%)
     - High (50–75%)
     - Very high (> 75%).
  2) The effects of the risk might be assessed as:
     - Catastrophic -threaten the survival of the project.
     - Serious -would cause major delays.
     - Tolerable -delays are within allowed contingency.
     - Insignificant- not affect the project.

- You should tabulate the results of this analysis process using a table as shown below which specifying the following things:
  - **Type of risk.**
  - **Description** of the risk itself.
  - **Probability:** It is the likelihood of the risk occurring, using either a numeric or categorical scale.
  - **Impact:** It is the magnitude of the loss if the risk were to occur, using either a numeric or a categorical scale.
- Once the risks have been analyzed and ranked, you should assess which of these risks are most significant. Your judgment must depend on a combination of the probability of the risk arising and the effects of that risk. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.

**Case Study:**

**Table 3.11: Risk Probability Effects**

| Sr. No. | Type of Risk | Risk | Probability | Effects |
|---|---|---|---|---|
| 1. | Technology Risks | The database used in the system cannot process as many transactions per second as expected. | Moderate | Serious |
| 2. | Technology Risks | Faults in reusable software components have to be repaired before these components are reused. | Moderate | Serious |
| 3. | People Risks | It is impossible to recruit staff with the skills required for the project. | High | Catastrophic |
| 4. | People Risks | Key staff is ill at critical times in the project. | Moderate | Serious |
| 5. | People Risks | Required training for staff is not available. | Moderate | Tolerable |
| 6. | Organizational Risks | The organization is restructured so that different management is responsible for the project. | High | Serious |
| 7. | Organizational Risks | Organizational financial problems force reductions in the project budget. | Low | Catastrophic |
| 8. | Tools Risks | Code generated by code generation tools is inefficient. | Moderate | Insignificant |
| 9. | Tools Risks | Software tools cannot be integrated. | High | Tolerable |
| 10. | Requirements Risks | Changes to requirements that require major design rework are proposed. | Moderate | Serious |
| 11. | Requirements Risks | Customers fail to understand the impact of requirements changes. | Moderate | Tolerable |
| 12. | Estimation Risks | The time required to develop the software is under-estimated. | High | Serious |
| 13. | Estimation Risks | The rate of defect repair is underestimated. | Moderate | Tolerable |
| 14. | Estimation Risks | The size of the software is underestimated. | High | Tolerable |

**3. Risk Mitigation Planning:**

- Risk mitigation is defined as the process of reducing risk exposure and minimizi the likelihood of an incident. It requires continually addressing top risks a concerns to ensure the business is fully protected.

- Mitigation often takes the form of controls or processes and procedures that regulate and guide an organization. The four common strategies are:

1) **Risk Mitigation:** Mitigating a risk is also a more specific strategy within the overall practice of mitigating risk. Someone would be best suited to go down the route of mitigating a risk if the risk level is out of tolerance and controls need to be put in place.
2) **Risk Acceptance:** This refers to not putting any controls in place, but rather trusting that the current risk level is within your determined tolerance. This strategy can be used to identify risks in order to ensure certain risks don't materialize. For example, projecting budget, which in turn lowers the risk of going over budget. This ensures that your team is made aware of the risk and possible consequences.
3) **Risk Transfer:** The transference strategy is another of our risk mitigation strategies and this refers to when an organization or team within an organization shifts the strain of a particular risk, as well as the consequences of that risk, onto another party. One of the most common examples of this is buying insurance. It is critical that in this strategy, all parties involved accept the terms.
4) **Risk Avoidance:** In short, this is the act of avoiding the actions that are suspected to bring on a particular risk. One way of implementing avoidance is to avoid scheduling issues. To do this, you would identify issues that could come up that may affect the timeline of a project (important deadlines, due dates and delivery dates). Some risks may include being overly optimistic about a timeline, scheduling conflicts and poor time management. You could counteract those risks by creating a managed schedule that illustrates specific time allowances for every aspect of the project.

**afting a Risk Mitigation Plan:**

When thinking about developing your risk mitigation plan, keep in mind that it should address the following areas of concern:

- **Change Management:** How do you manage change to the activity over time?
- **Compatibility:** Is the activity aligned with other activities?
- **Corporate Objectives:** Are performance goals advanced by this activity?
- **Cost:** Does the cost exceed the benefit derived from it?
- **Dependencies:** Are the relevant resource elements linked to the activity?
- **Effectiveness:** Does it address specific risks?
- **Efficiency:** Is it easy to implement and monitor?
- **Leverage:** Can it provide benefit in other areas?
- **Ownership:** Who is responsible for maintaining this activity?
- **Regulatory:** Does it address compliance readiness standards?

**Case Study 1:**

To better understand risk mitigation, let's examine some real-world examples of controls or processes and procedures that we use in our everyday lives to reduce certain risks from materializing.

**[Note:** The following examples are aimed to provide context to better understand how mitigating activities work; every person has different circumstances and needs, so these are not to be taken as personal advice].

**Mitigating Financial Risk:**

- We need money to survive on a daily basis. We also need it to be prepared for the possibility of a major life event requiring a large sum of money be put forward, and for when old age prevents us from being able to earn money through a job. In order to stay financially secure, we may decide to:
  - Max out our retirement savings.
  - Keep an emergency fund in a liquid savings account.
  - Pay cash for everything to ensure we're not buying anything we cannot afford.

**Risk Mitigation in Personal Relationships:**

- Positive personal relationships bring fulfilment to our lives, and like everything else we need to actively maintain the quality of those relationships to keep them from falling apart. Here are some examples of those nurturing efforts:
  - Treating those we love with kindness and respect.
  - Consistently calling, sending cards and visiting.
  - Cutting out relationships with people who don't treat us well (in order to make more time for those that do).

**Mitigating the Risk of Health Problems:**

- Our health is the foundation of our lives, so it's critical to take proper measures in ensuring it. While there are infinite ways to maximize our health and minimize the risk of serious problems, here are just a few of the most common mitigation activities:
  - Drinking plenty of water (the recommended amount for our body size).
  - Staying away from toxic behaviors like smoking, drinking or eating processed foods.
  - Exercising regularly.
- The risk planning process considers each of the key risks that have been identified and develops strategies to manage these risks. For each of the risks, you have to think about the actions which will minimize the interference to the project if the problem identified in the risk occurs. You should have strategies in place to cope with the risk if it arises. Strategy should be so effective that reduces the overall impact of a risk on the project or product. It relies on the judgment and experience of the project manager.

**Case Study 2:**

- Following table shows possible risk management strategies corresponding to risk that have been identified earlier.

**Table 3.12: Common Risk Mitigation Strategies**

| Sr. No. | Risk | Strategy |
|---|---|---|
| 1. | Organizational financial problems | Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective. |
| 2. | Recruitment problems | Alert customer to potential difficulties and the possibility of delays; investigate buying-in components. |
| 3. | Staff illness | Reorganize team so that there is more overlap of work and people therefore understand each other's jobs. |
| 4. | Defective components | Replace potentially defective components with bought-in components of known reliability. |
| 5. | Requirements changes | Derive traceability information to assess requirements change impact; maximize information hiding in the design. |
| 6. | Organizational restructuring | Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business. |
| 7. | Database performance | Investigate the possibility of buying a higher-performance database. |
| 8. | Underestimated development time | Investigate buying-in components; investigate use of a program generator. |

**4. Risk Monitoring:**

- Risk monitoring control is the process of:
  - Keeping track of the identified risks.
  - Monitoring residual risks and identifying new risks.
  - Ensuring the execution of risk plans.
  - Evaluating their effectiveness in reducing risk.

- Risk monitoring and control is an ongoing process for the life of the project. The risks change as the project matures, new risks develop, or anticipated risks disappear. You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable. You should also think about whether the effects of the risk have changed.
- To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are obviously dependent on the types of risk.
- Following table gives some examples of factors that may be helpful in assessing these risk types.

**Table 3.13: Factors that helpful in assessing Risk Monitoring**

| Sr. No. | Risk Type | Potential Indicators |
|---|---|---|
| 1. | Technology Risks | • Late delivery of hardware or support software.<br>• Many reported technology problems. |
| 2. | People Risks | • Poor staff morale.<br>• Poor relationships amongst team members.<br>• High staff turnover. |
| 3. | Organizational Risks | • Organizational gossip.<br>• Lack of action by senior management. |
| 4. | Tools Risks | • Reluctance by team members to use tools.<br>• Complaints about CASE tools.<br>• Demands for higher-powered workstations. |
| 5. | Requirements Risks | • Any requirements change requests<br>• Customer complaints. |
| 6. | Estimation Risks | • Failure to meet agreed schedule.<br>• Failure to clear reported defects. |

**5. Risk Mitigation, Monitoring, and Management (RMMM):**

- A risk management technique is usually seen in the software Project plan. This can be divided into Risk Mitigation, Monitoring, and Management Plan (RMMM). In this plan, all works are done as part of risk analysis. As part of the overall project plan project manager generally uses this RMMM plan.
- In some software teams, risk is documented with the help of a *Risk Information Sheet* (RIS). This RIS is controlled by using a database system for easier management of information i.e. creation, priority ordering, searching, and other analysis. After documentation of RMMM and start of a project, risk mitigation and monitoring steps will start.

**Risk Mitigation:**

- It is an activity used to avoid problems (Risk Avoidance). Steps for mitigating the risks as follows.
- Finding out the risk.
- Removing causes that are reason for risk creation.
- Controlling the corresponding documents time to time.
- Conducting timely reviews to speed up the work.

**Risk Monitoring:**

- It is an activity used for project tracking. It has following primary objectives as the follows:
  - To check if predicted risks occurs or not.
  - To ensure proper application of risk aversion steps defined for risk.
  - To collect data for future risk analysis.
  - To allocate what problems are caused by which risks throughout the project.

**Risk Management and Planning:**

- It assumes that the mitigation activity failed and the risk is a reality. This task is done by Project manager when risk becomes reality and causes severe problems.
- If the project manager effectively uses project mitigation to remove risks successfully then it is easier to manage the risks. This shows that the response that will be taken for each risk by a manager.
- The main objective of the risk management plan is the risk register. This risk register describes and focuses on the predicted threats to a software project.

## 3.5 OVERVIEW OF PROJECT ESTIMATION

- Accurately estimation of software size, cost, effort, and schedule is probably the biggest challenge in project management today. Project estimation consists of estimation of following entities which are related to a project:
  1. Estimating Project/Software Size.
  2. Estimating Project Cost.
  3. Estimating Effort.
  4. Estimating Project Schedule.
- These Estimations are explained in detail as follows:

### 3.5.1 Estimating Software Size

- An accurate estimate of software size is an essential element in the calculation of estimated project costs and schedules.

- Initial size estimates are typically based on the known system requirements. You must search for every known detail of the proposed system, and use these details to develop and validate the software size estimates.
- In general, you present size estimates as lines of code (KSLOC or SLOC) or as function points. There are constants that you can apply to convert function points to lines of code for specific languages, but not vice versa. If possible, choose and adhere to one unit of measurement, since conversion simply introduces a new margin of error into the final estimate.
- The following are techniques for estimating software size:
  1. **Developer Opinion:** Developer opinion is known as *guessing*. If you are an experienced developer, you can likely make good estimates due to familiarity with the type of software being developed.
  2. **Previous Project Experience:** Looking at previous project experience serves as a more educated guess. By using the data stored in the metrics database for similar projects, you can more accurately predict the size of the new project.
  3. **Count Function Blocks:**

     The technique of counting function blocks relies on the fact that most software systems decompose into roughly the same number of "levels". Using the information obtained about the proposed system, follow these steps:

     1) Decompose the system until the major functional components have been identified (call this a function block, or software component).
     2) Multiply the number of function blocks by the expected size of a function block to get a size estimate.
     3) Decompose each function block into subfunctions.
     4) Multiply the number of subfunctions by the expected size of a subfunction to get a second size estimate.
     5) Compare the two size estimates for consistency.

     Compute the expected size of a function block and/or a subfunction with data from previous projects that use similar technologies and are of similar scope.
  4. **Function Point Analysis (FPA):**

     Function point allows the measurement of software size in standard units independent of the underlying language in which the software is developed. Instead of counting the lines of code that make up a system, count the numbers of externals (inputs, outputs, inquiries, and interfaces) that make up the system.

## 3.5.2 Estimating Software Cost

- The cost of medium and large software projects is determined by the cost of developing the software, plus the cost of equipment and supplies. The cost of equipment and supplies are constant for most projects. The cost of developing the

software is simply the estimated effort, multiplied by presumably fixed labor costs. For this reason, we will concentrate on estimating the development effort.

- A number of methods have been used to estimate software costs. These include:

1. **Algorithmic Models:** These methods provide one or more algorithms which produce a software cost estimate as a function of a number of variables which are considered to be the major cost drivers. COCOMO is an example of an algorithmic model.
2. **Expert Judgment:** This method involves consulting one or more experts, perhaps with the aid of an expert-consensus mechanism such as the Delphi technique.
3. **Analogy:** This method involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project.
4. **Parkinson:** A Parkinson principle ("Work expands to fill the available volume") is invoked to equate the cost estimate to the available resources.
5. **Price to Win:** The cost estimate developed by this method is equated to the price believed necessary to win the job (or the schedule believed necessary to be first in the market with a new product, etc.).
6. **Top-Down:** An overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.
7. **Bottom-Up:** Each component of the software Job is separately estimated, and the results aggregated to produce an estimate for the overall job.

## 3.5.3 Estimating Effort

- There are two basic models for estimating software development effort (or cost): Holistic and Activity-based. The single biggest cost driver in either model is the estimated project size.
- **Holistic Models** are useful for organizations that are new to software development, or that do not have baseline data available from previous projects to determine labor rates for the various development activities.
- Estimates produced with **Activity-based Models** are more likely to be accurate, as they are based on the software development rates common to each organization. Unfortunately, you require related data from previous projects to apply these techniques.
- Popular holistic models include the following:
  - SDM (Software Development Model - Putnam - 1978).
  - SLIM (Software Lifecycle Management - Putnam - 1979).
  - COCOMO (Constructive Cost Model - Boehm - 1981).
  - COPMO (Cooperative Programming Model - Conte, Dunsmuir, Shen- 1986).

### 3.5.4 Estimating Software Schedule

- There are many tools on the market (such as MS Project etc.) which help develop Ga and PERT charts to schedule and track projects. These programs are most effecti when you break the project down into a Work Breakdown Structure (WBS), and assig estimates of effort and staff to each task in the WBS.

## 3.6 LINEAR SOFTWARE PROJECT COST ESTIMATION

- Every year more projects are lost by poor cost and schedule estimates than b technical, political or organizational problems. It's no wonder that so few compani realize that software cost estimating can be a science, not just an art. It has bee proven that it is quite applicable to accurately and consistently predict developme life cycle costs and schedules for a broad array of software projects.
- In early models, complexity means the project size or the program volume, which c be estimated via kilo lines of codes KLOC.
- In late models, complexity is determined firstly by inputs, outputs, interfaces, files a queries that the software system needs. Then this complexity is further adjusted via to 14 different added-complexity factors. Eventually, the final result is converte through a standard conversion table to KLOC.
- In Basic Cost Estimation Model, the calculation is straightforward. By determining t value of only two variables, total efforts in *person-months* can be easily calculat These two variables are :
  1. How many thousands of lines of code (KLOC) your programmers must develop?
  2. The effort required per KLOC (i.e. Linear Productivity Factor).
- Accordingly, multiplying these two variables together will result in the *person-mon* of effort required for the project provided that the project is relatively sm Otherwise, another exponential size penalty factor has to be incorporated for lar project sizes. *Person-months* imply the number of months required to complete entire project if only one person was to carry out this mission. This underlying con is the foundation of all of the software cost estimating models, especially th originated from Barry Boehm's famous COCOMO models.

### 3.6.1 COCOMO-I Model

- The term COCOMO stands for "Constructive Cost Model". This model is use estimate effort, cost and schedule for software projects. Barry W. Boehm proposed model. The fundamental input to COCOMO is the estimated number of lines of so code.
- Boehm proposed three levels of the model: Basic, Intermediate, and Detailed.
  1. The **Basic COCOMO-I model** is a single-valued, static model that comp software development effort (and cost) as a function of program size express estimated thousand delivered source instructions (KDSI). The basic method simple formulae to derive the total number of man months of effort, and the elapsed time of the project, from the estimated number of lines of code.

2. The **Intermediate COCOMO-I model** computes software development effort as a function of program size and a set of fifteen 'effort multipliers/cost drivers" that include subjective assessments of product, hardware, personnel, and project attributes. Wildly inaccurate estimates can result from a poor choice of effort multiplier values. COCOMO supplies objective criteria to help estimator make a sensible choice.
3. The **Advanced or Detailed COCOMO-I model** incorporates all characteristics of the intermediate version with an assessment of the effort multiplier/cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

- When estimating by means of COCOMO, the intermediate method should be used because the detailed method does not appear to perform more accurately than the intermediate method. The basic method provides a level of accuracy that is only adequate for preliminary estimate.
- COCOMO-I models depends on the two main equations:

$$\textbf{Effort} = A \times (\textbf{Size})^{B}$$

$$\textbf{Time}_{dev} = C \times (\textbf{Effort})^{D}$$

- The coefficients A, B, C and D depend on the mode of the development/classes of software projects.
- These classes or types are as follows:

1. **Organic:** In this type, the software is developed by small software teams in a highly familiar environment.
2. **Semi-detached:** This type is seen as a bridge-type between the organic and embedded types. It can either be seen as a mixture of characteristics of both types or an intermediate level of project characteristic.
3. **Embedded:** In this type, the product is operated within a highly coupled complex of hardware, software and operational constraints.

**Table 3.14: Three modes of development and its characteristics**

| Development Mode | Project Characteristics | | | |
|---|---|---|---|---|
| | A (size) | B (innovation) | C (deadline/constraints) | Dev. Environment |
| **Organic** | Small | Little | Not tight | Stable |
| **Semi-detached** | Medium | Medium | Medium | Medium |
| **Embedded** | Large | Greater | Tight | Complex hardware/ customer interfaces |

1. **Equations and Parameters of Basic COCOMO :**

- The basic COCOMO applies the parameterized equation without much details consideration of project characteristics.

$$\text{Effort} = A \times (\text{Size})^{B}$$

Where,

**A:** Constant (based on the software project class).

**B:** Constant (based on the software project class).

**Size:** Size of the software (expressed in KLOCs).

**Effort:** Estimated effort (expressed in units of PMs).

$$\text{Time}_{dev} = a \times (\text{Effort})^{b}$$

Where,

**a:** Constant (based on the software project class).

**b:** Constant (based on the software project class).

**Effort:** Previously calculated estimated effort (expressed in units of PMs).

**$\text{Time}_{dev}$:** Estimated development time (expressed in months).

**Parameters of Basic COCOMO Model:**

**Table 3.15: Parameters**

| Basic COCOMO | Effort | | Schedule | |
|---|---|---|---|---|
| | A | B | a | b |
| **Organic** | 2.4 | 1.05 | 2.5 | 0.38 |
| **Semi-detached** | 3.0 | 1.12 | 2.5 | 0.35 |
| **Embedded** | 3.6 | 1.20 | 2.5 | 0.32 |

2. **Equations and Parameters of Intermediate COCOMO:**

- In the viewpoint of intermediate COCOMO, there are 15 different factors that affec software project. These factors are grouped together in 4 different categories. Th categories and their factors are listed as follows:
- **Software Product:**
  1. Reliability requirement
  2. Database size
  3. Product complexity
- **Computer System:**
  1. Execution time constraints
  2. Main storage constraints
  3. Virtual machine volatility
  4. Computer turnaround time

**Human Resource:**

1. Analyst capability
2. Virtual machine experience
3. Programmer capability
4. Programming language experience
5. Application experience

**Software Project:**

1. Use of modern programming practices
2. Use of software tools
3. Required development schedule

- Each of these properties or factors is rated in 5 levels ranging between "very low" to "extra high". As in basic COCOMO, a numerical value is assigned to each property. This numerical value is called Effort Multiplier (EM).
- The same basic equation for the model is used, but fifteen cost drivers are rated on a scale of 'very low' to 'very high' to calculate the specific effort multiplier and each of them returns an adjustment factor which multiplied yields in the total EAF (Effort Adjustment Factor).
- In addition to the EAF, the model parameter "A" is slightly different in Intermediate COCOMO from the basic model. Other parameter remains the same in both models.

**Table 3.16: Parameters of Intermediate COCOMO Model**

| Intermediate COCOMO | Effort | | Schedule | |
|---|---|---|---|---|
| | A | B | a | b |
| **Organic** | 3.2 | 1.05 | 2.5 | 0.38 |
| **Semi-detached** | 3.0 | 1.12 | 2.5 | 0.35 |
| **Embedded** | 2.8 | 1.20 | 2.5 | 0.32 |

- To calculate EAF, the product of all effort multipliers of factors is used. Hence, the formula of EAF is as follows:

$$EAF = \prod_{i=1}^{15} EM_i$$

Where,

**EAF**: The overall effort adjustment factor (expressed in PMs).

$\mathbf{EM_i}$: The effort multiplier for the factor i (expressed in PMs).

$$\mathbf{Effort_{adj} = Effort_{in} \times EAF}$$

Where,

$\mathbf{Effort_{adj}}$: Adjusted effort for the intermediate COCOMO (expressed in PMs).

$\mathbf{Effort_{in}}$: Initial effort calculated with the basic COCOMO (expressed in PMs).

**EAF**: The overall adjustment factor (expressed in PMs).

### 3. Advanced/Detailed COCOMO:

- The Advanced COCOMO model computes effort as a function of program size and a s of cost drivers weighted according to each phase of the software lifecycle. Th Advanced model applies the intermediate model at the component level, and then phase-based approach is used to consolidate the estimate.
- The four phases used in the detailed COCOMO model are: Requirements Planning an Product Design (RPD), Detailed Design (DD), Code and Unit Test (CUT), and Integratio and Test (IT). Each cost driver is broken down by phases as in the example shown Table 3.17

**Table 3.17: Analyst capability effort multiplier for Detailed COCOMO Model**

| Cost Driver | Rating | RPD | DD | CUT | IT |
|---|---|---|---|---|---|
| ACAP | Very Low | 1.80 | 1.35 | 1.35 | 1.50 |
| | Low | 0.85 | 0.85 | 0.85 | 1.20 |
| | Nominal | 1.00 | 1.00 | 1.00 | 1.00 |
| | High | 0.75 | 0.90 | 0.90 | 0.85 |
| | Very High | 0.55 | 0.75 | 0.75 | 0.70 |

- Estimates for each module are combined into subsystems and eventually an over project estimate. Using the detailed cost drivers, an estimate is determined for ea phase of the lifecycle.

**Advantages of COCOMO-I:**

1. COCOMO is transparent; one can see how it works.
2. Drivers are particularly helpful to the estimator to understand the impact different factors that affect project costs.

**Disadvantages of COCOMO-I:**

1. The number of line of code is only accurately predictable at the end of t architectural design phase of a project, and this is too late.
2. Line of code can vary amongst programming languages and conventions.
3. The concept of a line of code does not apply to some modern programmi techniques, e.g. visual programming.
4. It is hard to accurately estimate KDSI early on in the project, when most effi estimates are required.
5. KDSI, actually, is not a size measure it is a length measure.
6. Extremely vulnerable to misclassification of the development mode.
7. Success depends largely on tuning the model to the needs of the organizati using historical data which is not always available.

**ase Study:**

The case study is *a development of a task manager web application.* The application rovides facilitation to a supervisor to track the progress of job tasks assigned to his/her eam. Team inserts the completed tasks associated with time spent for each task. The upervisor will check these tasks and give some notes on these tasks. This project has hree actors: Administrator, Supervisor and Team Members. Each actor has his/her own esponsibilities as given below:

1. **Team Member:**
   At the end of day, the system allows the member to insert the tasks that have been completed by him association with the time spent to complete them.
   The member can receive a report from supervisor as a feedback of these tasks.
2. **Supervisor:**
   The supervisor can receive the tasks completed by his/her employees. He can reject or pass or give some notes on any task to improve depending on three criteria:
   1) It is on or out the scope.
   2) It has any effect on the project or not.
   3) It is completed on the realistic time or not.
3. **Administrator:**
   The system generates different types of reports for administrator about each employee such as:
   1) Number of task reject and number of task pass.
   2) Calculate the active hours (which are the total number of hours that spent for pass task) and present them as chart (this chart will present active hours for each employee that allows the administrator) to evaluate the employee.

**Numerical problem based on COCOMO-I:**

Java Programming Language is used during the case study to develop Task Manager Application. The number of lines of code will be approx. 6762.1 DSI (6.7621KDSI). How much effort and time is required to complete a project successfully?

**Solution:**

The size of the project is small and the complexity is simple, we categorize this project as Organic and the number of lines of code is 6762.1 DSI (6.7621KDSI).

**Parameters of Basic COCOMO** are given below:

**Table 3.18: Parameters**

| Basic COCOMO | Effort | | Schedule | |
|---|---|---|---|---|
| | A | B | a | b |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

**Effort** = $A \times (Size)^{B}$ = 2.4 *6.7621^1.05 = 17.86

$T_{Dev} = a \times (Effort)^{b}$ = 2.5*17.86^0.38 = 7.5(~8 months)

**People** = 17.86/7.5 = ~2 members

## 3.6.2 Function Point Analysis (Problem Statement)

- Function points allow the measurement of software size in standard uni independent of the underlying language in which the software is developed. Instead counting the lines of code that make up a system, count the number of externa (inputs, outputs, inquiries, and interfaces) that make up the system.
- There are five types of externals to count The *Function Point* (FP) *metric* can be us effectively as a means for measuring the functionality delivered by a system. Usi historical data, the FP metric can then be used to:
  1. Estimate the cost or effort required to design, code, and test the product.
  2. Predict the number of errors that will be encountered during testing.
  3. Forecast the number of components and/or the number of projected source lines the implemented system.

**Domain Values:**

- Function points are derived using an empirical relationship based value of projec information domain and project complexity. Information domain values are defin in the following manner:
  - **Number of External Inputs (EIs):** Each *external input* originates from a user or transmitted from another application and provides distinct application-orient data or control information. Inputs are often used to update *Internal Logical Fi* (ILFs). Inputs should be distinguished from inquiries, which are coun separately.
  - **Number of External Outputs (EOs):** Each *external output* is derived data with the application that provides information to the user. In this context exter output refers to reports, screens, error messages, etc. Individual data items with a report are not counted separately.
  - **Number of External Inquiries (EQs):** An *external inquiry* is defined as an onli input that results in the generation of some immediate software response in t form of an online output.
  - **Number of Internal Logical Files (ILFs):** Each *internal logical file* is a logi grouping of data that resides within the application's boundary and is maintain via external inputs.
  - **Number of External Interface Files (EIFs):** Each *external interface file* is a logi grouping of data that resides external to the application but provides informati that may be of use to the application.
- Once these data have been collected then a complexity value is associated with ea count. Organizations that use function point methods develop criteria for determini whether a particular entry is Low, Medium or High.

**Table 3.19: Computing FPs**

| Information domain Value | Count | Simple | Average | Complex | Total |
|---|---|---|---|---|---|
| External Inputs | - | 3 | 4 | 6 | - |
| External Outputs | - | 4 | 5 | 7 | - |
| External Inquiries | - | 3 | 4 | 6 | - |
| Internal Logical Files | - | 7 | 10 | 15 | - |
| External Interface Files | - | 5 | 7 | 10 | - |

To compute function points (FP), the following relationship is used:

$$\mathbf{FP = Count\ total \times [0.65 \times 0.01 \times \Sigma (F_i)]} \quad \text{... (1)}$$

Where, *Count total* is the sum of all FP entries obtained from above table:

The $F_i$ (i = 1 to 14) are *Value Adjustment Factors* (VAF) is a rating of 0 to 5 for each of the following fourteen factors:

1. Data communications
2. Distributed functions
3. Performance
4. Heavily used operational configuration
5. Transaction rate
6. On-line data entry
7. Design for end user efficiency
8. On-line update of logical internal files
9. Complex processing
10. Reusability of system code
11. Installation ease
12. Operational ease
13. Multiple sites
14. Ease of change

- Each of these factors using a scale that ranges from 0 - 5. Assign the rating of 0 to 5 according to these values:

0 - Factor not present or has no influence

1 - Insignificant influence

2 - Moderate influence

3 - Average influence

4 - Significant influence

5 - Strong influence

- The constant values in Equation-1 and the weighting factors that are applied to information domain counts are determined empirically.

## Numerical Problems based on FPA:

Suppose the requirement specification for the Website Development of the Children Welfare Project has been carefully analyzed and the following estimates have been obtained. There is a need for 11 inputs, 11 outputs, 7 inquiries, 22 files, and 6 external interfaces. Also, assume outputs, queries, internal files function point attributes are of low complexity and all other function points attributes are of medium complexity. Sum of all FP is 40.

What is the Function Points (FP) for the Children Welfare project?

**Solution:**

**Table 3.20: Calculating Function Points**

| Information domain Value | Count | Low | Medium | High | Total |
|---|---|---|---|---|---|
| External Inputs | - | 3 | 4*11 | 6 | 44 |
| External Outputs | - | 4*11 | 5 | 7 | 44 |
| External Inquiries | - | 3*7 | 4 | 6 | 21 |
| Internal Logical Files | - | 7*22 | 10 | 15 | 154 |
| External Interface Files | - | 5 | 7*6 | 10 | 42 |
| | | | | Total Unadjusted Function Points | 305 |

- The count total computed is 305 which must be adjusted using Equation-1. As we ha Sum of All FP is 40. Therefore,

$$FP = 305 * [0.65 + (0.01 * 40)] = 320.25$$

**Case Study 1:**

- For Stock Control application project: Estimate how complex such system can be a after that try to predict how long it would take to develop it.
- At first, we should pay attention to the functionality - what exactly the system shou be able to do. Basically, it should be able to take care about three parts - custome stock, and transactions. Then, let us group functions into five categories:
  1. **External Inputs (EI):** There are four things we need to consider these a customer, order, stock, and payment details.
  2. **External Outputs (EO):** There are four things to consider these are custom order, stock details and credit rating.
  3. **External Inquiries (EQ):** The system is requested for three things, which a customer, order, and stock details.

4. **External Interface Files (EIF):** There are no EIFs to consider.
5. **Internal Logical Files (ILF):** Finally, the four elements belong to the last group. Customer, and good files, customer, and good transaction files.

- Let's predict every function's complexity is low, so the values can be presented in a table:

**Table 3.21: Values of system functions**

| Category | Multiplier | Weight |
|---|---|---|
| EI | 4 | 3 |
| EO | 4 | 4 |
| EQ | 3 | 3 |
| ILF | 4 | 7 |

4 * 3 + 4 * 4 + 3 * 3 + 4 * 7 = 65 [Function Points]

- Let us omit additional technical complexity factors, so the only thing left to do is to check how long it takes to produce 65 function points.
- Some sources prove that one function point is an equivalent of eight hours of work in C++ language. Then, the last thing is:

65 * 8 = 520 [hours]

- The answer is the estimate for developing the application would take about 520 hours of work.

## 3.6.3 The SEI Capability Maturity Model (CMM)

- The Capability Maturity Model (CMM) for software provides software organizations with guidance on how to gain control of their processes for developing and maintaining software and how to evolve toward a culture of software engineering and management excellence.
- The Software Engineering Institute (SEI) Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization.
- The quality of a software development organization is analyses by its position at which level in SEI-CMM levels. The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.
- Following Figure 3.5 shows the five levels of software process maturity defined by CMM.

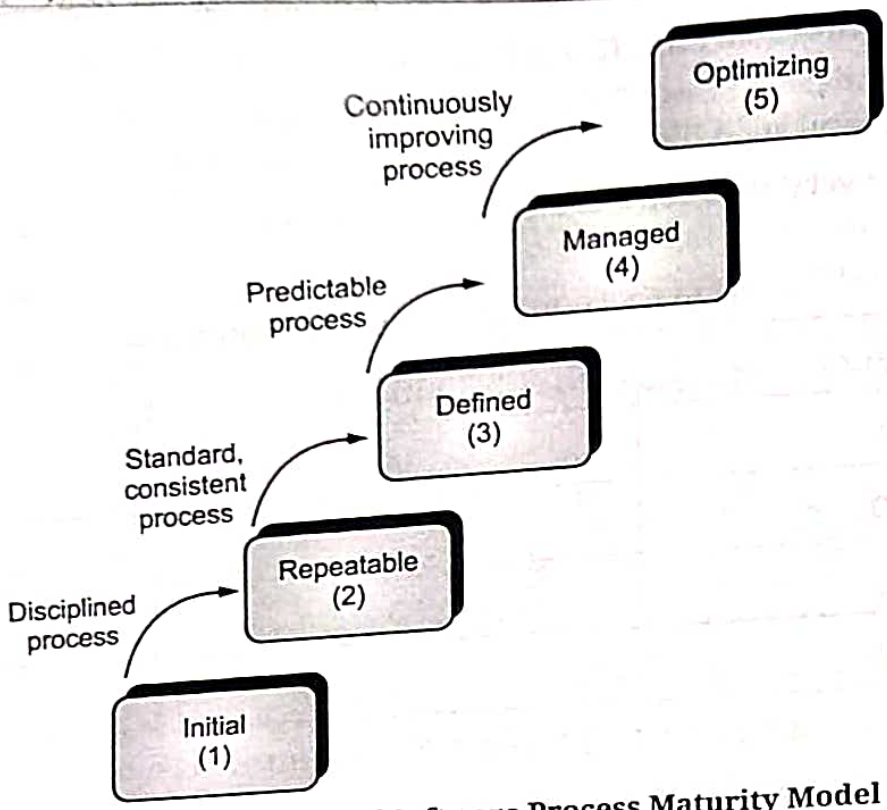



**Fig. 3.5: The Five Levels of Software Process Maturity Model**

- The following characterizations of the five maturity levels highlight the prim process changes made at each level:
  - **Level 1: Initial:** The software process is characterized as ad hoc, and occasiona even chaotic. Few processes are defined, and success depends on individual eff The software process capability of Level 1 organizations is unpredictable beca the software process is constantly changed or modified as the work progresses (i the process is ad hoc). Schedules, budgets, functionality, and product quality generally unpredictable. Performance depends on the capabilities of individu and varies with their innate skills, knowledge, and motivations.
  - **Level 2: Repeatable:** Basic project management processes are established to tra cost, schedule, and functionality. The necessary process discipline is in place repeat earlier successes on projects with similar applications. The software proc capability of Level 2 organizations can be summarized as disciplined becau planning and tracking of the software project is stable and earlier successes can repeated. The project's process is under the effective control of a proj management system, following realistic plans based on the performance previous projects.
  - **Level 3: Defined:** The software process for both management and engineeri activities is documented, standardized, and integrated into a standard softwa process for the organization. All projects use an approved, tailored version of t organization's standard software process for developing and maintaini software. The software process capability of Level 3 organizations can

summarized as standard and consistent because both software engineering and management activities are stable and repeatable. Within established product lines, cost, schedule, and functionality are under control, and software quality is tracked. This process capability is based on a common, organization-wide understanding of the activities, roles, and responsibilities in a defined software process.

- **Level 4: Managed:** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled. The software process capability of Level 4 organizations can be summarized as predictable because the process is measured and operates within measurable limits. This level of process capability allows an organization to predict trends in process and product quality within the quantitative bounds of these limits. When these limits are exceeded, action is taken to correct the situation. Software products are of predictably high quality.
- **Level 5: Optimizing:** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

## 3.6.5 Software Configuration Management

- Configuration Management (CM) is concerned with the policies, processes, and tools for managing changing software systems The configuration management of a software system product involves four activities:
  1. **Change Management:** This involves keeping track of requests for changes to the software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
  2. **Version Management:** This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
  3. **System Building and Defect Management:** This is the process of assembling program components, data, and libraries, and then compiling and linking these to create an executable system. A defect management process focuses on preventing defects, catching defects as early in the process as possible, and minimizing the impact of defects.
  4. **Release Management:** This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

## 3.6.5.1 Change Management Process

- A change management process is a method by which changes to the project scope, deliverables, timescales or resources are identified, evaluated and approved prior to implementation. The process entails completing a variety of control procedures to ensure that if implemented, the change will cause minimal impact to the project.
- This process is undertaken during the execution phase of the project, once the project has been formally defined and planned.

- In theory, any change to the project during the execution phase will need to b formally managed as part of the change process.
- The change management process is terminated only when the execution phase of th project is complete. Following figure 3.6 shows the processes and procedures to b undertaken to initiate, implement and review changes within the project.

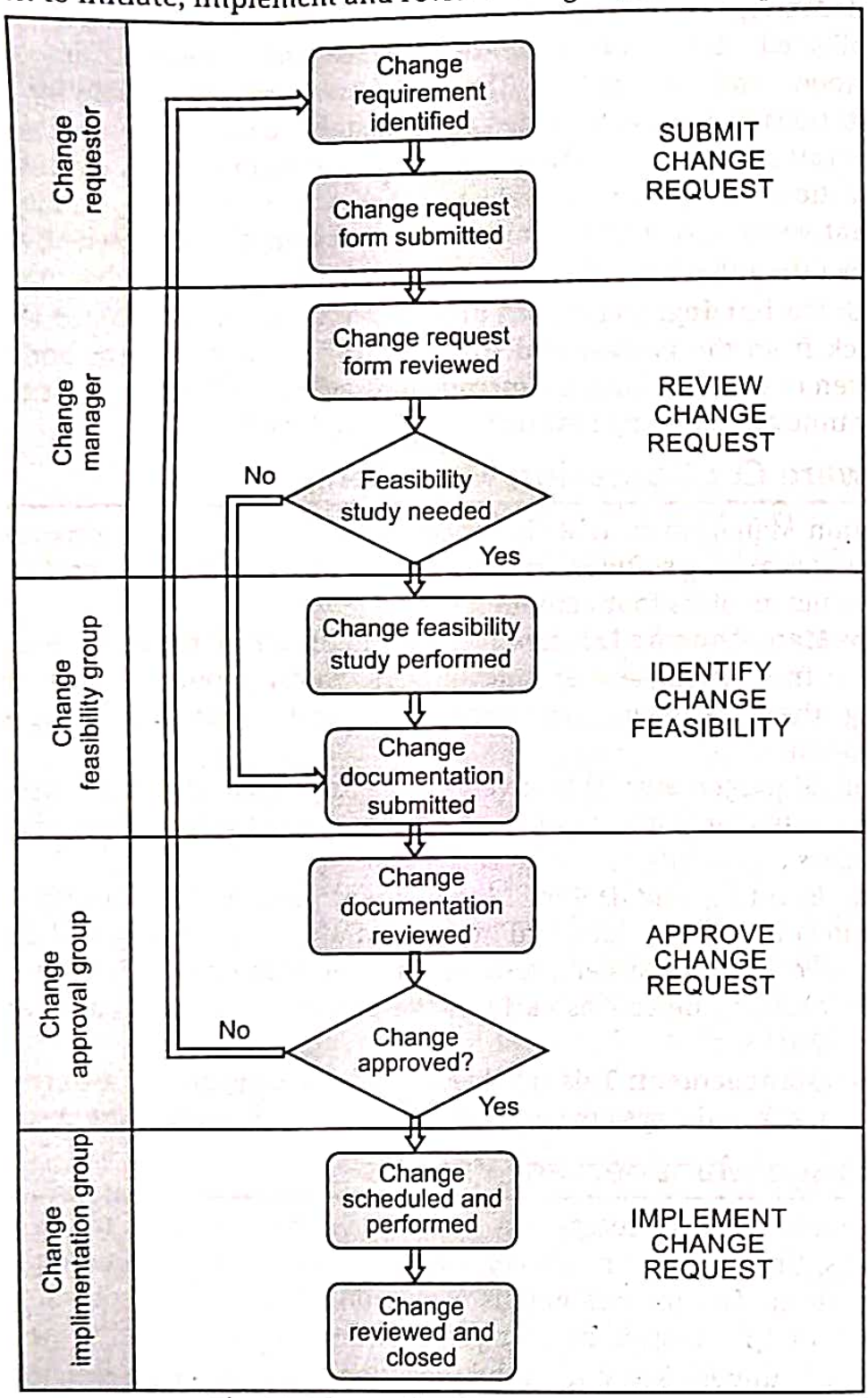



**Fig. 3.6: Change Management Process**

- During change management following activities are taken place. All these activities are carried out in sequence.

1. **Submit Change Request:**

To initiate a change process, you should first allow any member of the project team to submit a request for a change to the project. The person raising the change is called the 'change requester'. The change requester will document the requirement for change to the project by completing a change request form (CRF), summarizing the change description, benefits, costs, impact and approvals required.

2. **Review Change Request:**

The change manager reviews the CRF and determines whether or not a feasibility study is required for the change approval group to assess the full impact of the change. The decision will be based on the size and complexity of the change proposed. The change manager will record the CRF details in the change register.

3. **Identify Change Feasibility:**

If deemed necessary, a change feasibility study is completed to determine the extent to which the change requested is actually feasible. The change feasibility study will define in detail the change requirements, options, costs, benefits, risks, issues, impact, recommendations and plan. All change documentation is then collated by the change manager and submitted to the change approval group for final review. This includes the original CRF, the approved change feasibility study report and any supporting documentation.

4. **Approve Change Request:**

A formal review of the CRF is undertaken by the change approval group. The change approval group will reject the change, request more information related to the change, approve the change as requested or approve the change subject to specified conditions. Their decision will be based on the level of risk and impact to the project resulting from both implementing and not implementing the change.

5. **Implement Change Request:**

Approved changes are then implemented. This involves:

1. Identifying a date for implementation of the change;
2. Implementing the change;
3. Reviewing and communicating the success of the change implementation;
4. Recording all change actions in the change register.

**Roles and Responsibilities:**

- Following are roles and responsibilities which are involved in change management within the project.

**Table 3.22: Roles and Responsibilities of Change Management**

| Sr. No. | Role | Main Responsibility | Sub Responsibilities |
|---|---|---|---|
| 1. | **Change requestor** | The change requestor initially recognizes a need for change to the project and formally communicates this requirement to the change manager. | 1. Identifying the need to make a change to the project.<br>2. Documenting the need for change by completing a CRF.<br>3. Submitting the CRF to the change manager for review. |
| 2. | **Change manager** | The change manager receives, logs, monitors and controls the progress of all changes within a project. | 1. Receiving all CRFs and logging them in the change register.<br>2. Categorizing and prioritizing all change requests.<br>3. Reviewing all CRFs to determine whether additional information is required.<br>4. Determining whether or not a formal change feasibility study is required.<br>5. Forwarding the CRF to the change approval group for approval.<br>6. Escalating all CRF issues and risks to the change approval group.<br>7. Reporting and communicating all decisions made by the change approval group. |
| 3. | **Change feasibility group** | The change feasibility group complete feasibility studies for CRFs | 1. Undertaking research to determine the likely options for change, costs, benefits and impacts of the change.<br>2. Documenting all findings within a feasibility study report. |

| Sr. No. | Role | Main Responsibility | Sub Responsibilities |
|---|---|---|---|
| | | issued by the change manager. | 3. Forwarding the feasibility study report to the change manager for change approval group submission. |
| 4. | **Change approval group** | The change approval group is the principal authority for all CRFs forwarded by the change manager. | 1. Reviewing all CRFs forwarded by the change manager.<br>2. Considering all supporting documentation.<br>3. Approving or rejecting each CRF based on its relevant merits.<br>4. Resolving change conflict, where two or more changes overlap.<br>5. Identifying the implementation timetable for approved changes. |
| 5. | **Change implementation group** | The change implementation group will schedule and implement all changes. | 1. Scheduling all changes within the timeframes provided by the change approval group.<br>2. Testing all changes, prior to implementation.<br>3. Implementing all changes within the project.<br>4. Reviewing the success of each change, following implementation.<br>5. Requesting that the change manager close the change in the change register. |

## 3.6.5.2 Versioning and Version Control

**Versioning:**

- As a project progresses, many versions of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.
- The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents and unique objects like screen and report definitions, object files, test data and results.
- A mature repository tracks versions of objects with arbitrary levels of granularity. For example, a single data definition or a cluster of modules can be tracked.

## Version Control:

- Version control provides for unique identification of documents, whether electronic or hard copy, and assists with the easy identification of each subsequent version of a document. The version number changes as the document is revised allowing released versions of a document to be readily discernable from draft versions.

**How to use Version Control?**

- Documents may be identified by a version number, starting at one and increasing by one for each release. Ideally, the version number should appear on the first page of the document and within the footer of each page.
- Documentation should be identified as follows:
  1. A release number (and a revision letter if in draft).
  2. The original draft shall be Version 0.A subsequent drafts shall be Version 0.B, Version 0.C etc.
  3. The accepted and issued document is Version 1.0 subsequent changes in draft form become Version 1.0A, 1.0B etc.
  4. The accepted and issued second version becomes Version 1.1 or Version 2.0, as determined by the author based on the magnitude of the changes (minor or major).
- On this basis a document, which is presented to a Steering Committee for endorsement, would include a revision letter such as Version 0.A. After receiving endorsement, a copy of the document (with any approved amendments) would be created without a revision letter, in this case as Version 1.0. Wherever the version number appears in the document (i.e. front page, release notice, footer etc.) it should be updated.
- If controlled copies are to be distributed (refer below) the information in the Distribution List, Build Status and Amendments in this Release sections may need to be updated. Where there is provision for signatures, these should also be obtained prior to distribution. Including the corresponding version number in the file name (for example, Project XYZ Business Plan v0.A.doc) can uniquely identify electronic copies of documents.

**Versioning Models:**

- Consider this scenario: Suppose we have two co-workers, Amit and Sunil. They each decide to edit the same repository file at the same time. If Amit saves his changes to the repository first, then it's possible that Sunil could accidentally overwrite them with his own new version of the file. While Amit's version of the file won't be lost forever (because the system remembers every change), any changes Amit made won't be present in Sunil's newer version of the file, because he never saw Amit's changes to begin with. Amit's work is still effectively lost - or at least missing from the latest version of the file - and probably by accident. All version control systems have the same fundamental problem. We have two solutions to avoid these problems.

1. **The Lock-Modify-Unlock Solution:**

- In such a system, the repository allows only one person to change a file at a time. First Amit must lock the file before he can begin making changes to it. If Sunil tries to lock the file, the repository will deny the request. After Amit unlocks the file, his turn is over, and now Sunil can take his turn by locking and editing as shown in Fig. 3.7

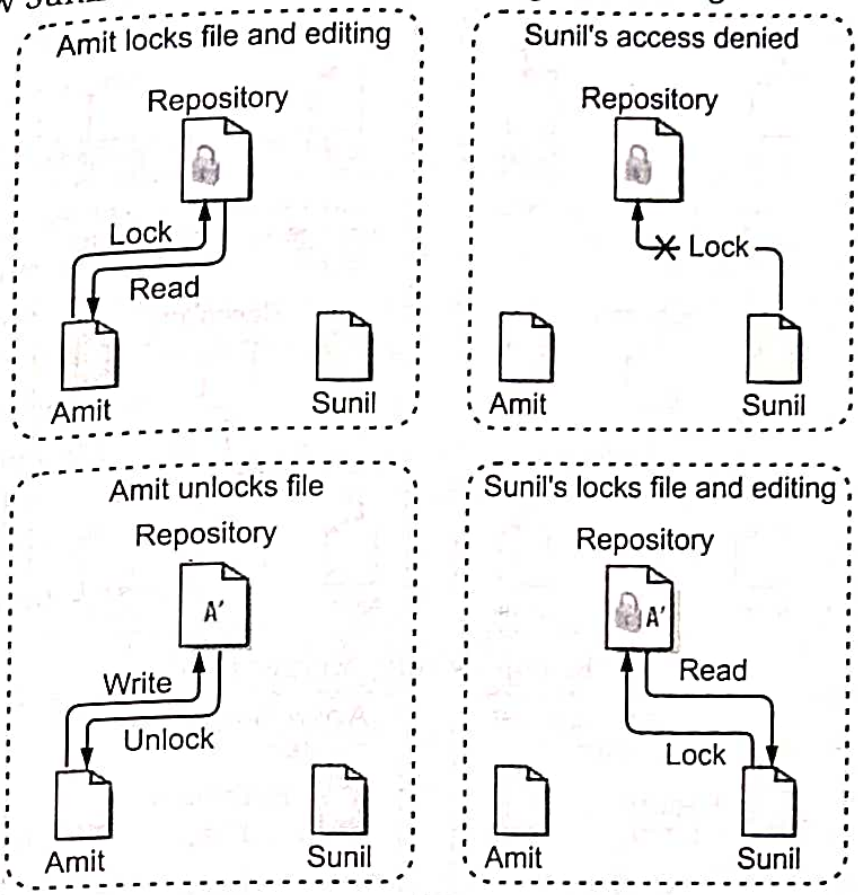



**Fig. 3.7: Lock-Modify-Unlock Solution**

**Limitations:**

1. The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users.
2. Locking may cause administrative problems. Someone will lock a file and then forget about it. The situation ends up causing a lot of unnecessary delay and wasted time.
3. Locking may cause unnecessary serialization.

2. **The Copy-Modify-Merge Solution:**

- Subversion and other version control systems use a copy-modify-merge model as an alternative to locking.
- In this model, each user's client reads the repository and creates a personal working copy of the file or project. Users then work in parallel, modifying their private copies.
- Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.
- The overall process is shown in Fig. 3.8 and Fig. 3.9.

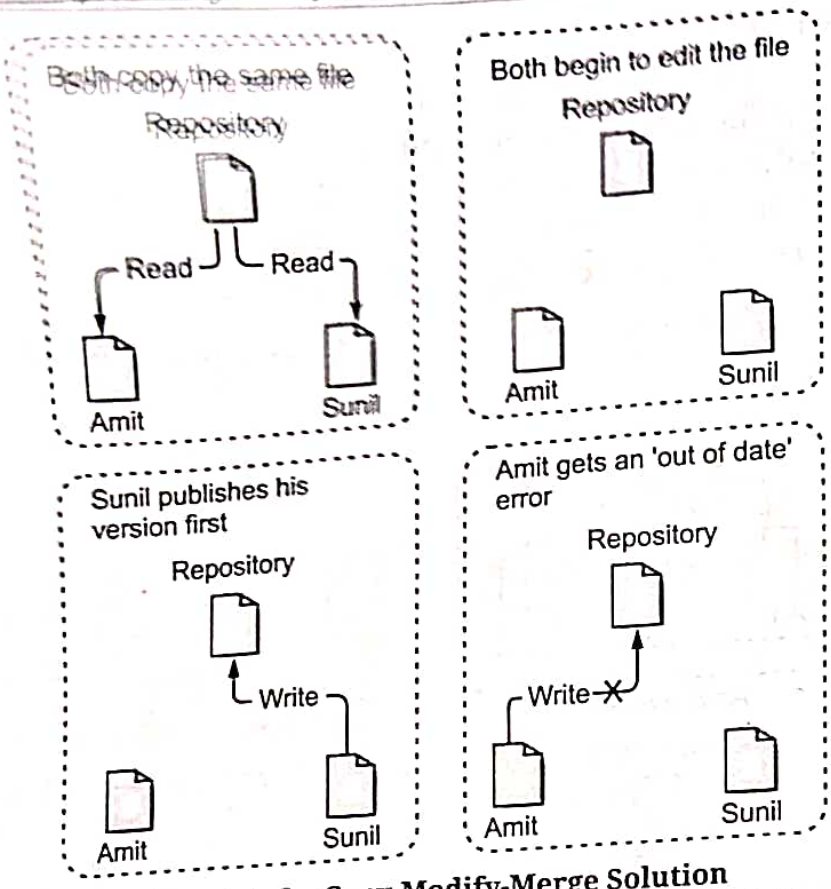



**Fig. 3.8: The Copy-Modify-Merge Solution**

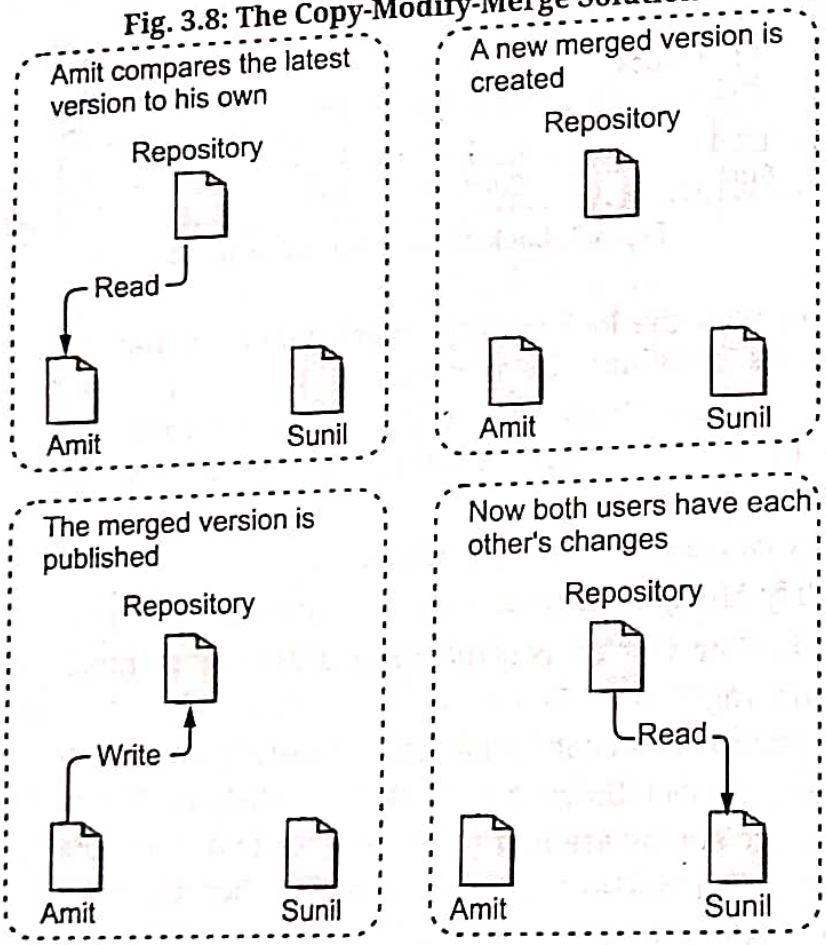



**Fig. 3.9: The Copy-Modify-Merge Solution Continued**

**Limitations:**

1. Need of effective user communication. When users communicate poorly, both syntactic and semantic conflicts increase.
2. If you are working on unmerged files your work will lose. For example, if your repository contains some graphic images, and two people change the image at the same time; there is no way for those changes to be merged together. Either Amit or Sunil will lose their changes.

## 3.6.5.3 Defect Management

- Software defects are expensive. Moreover, the cost of finding and correcting defects represents one of the most expensive software development activities. For the probable future, it will not be possible to eliminate defects. While defects may be unavoidable, we can minimize their number and impact on our projects.
- To do this development teams need to implement a defect management process that focuses on preventing defects, catching defects as early in the process as possible, and minimizing the impact of defects. A little investment in this process can yield significant returns.

**Defect Management Process:**

- The typical defect management process includes the following mentioned steps. When implemented inside of a specific organization, each of these steps would have more detailed standard operating procedures along with policies to carry out the details of the process.

1. **Identification:** This step involves the discovery of a defect. Hopefully, the person discovering the defect is someone on the testing team. In the real world, it can be anyone including the other individuals on the project team, or on rare occasions even the end customer.
2. **Categorization:** When a defect is reported, it is typically assigned to a designated team member to confirm that the defect is actually a defect as opposed to an enhancement, or other appropriate category as defined by the organization. Once categorized, the defect moves on in the process to the next step which is prioritization.
3. **Prioritization:** Prioritization is typically based on a combination of the severity of impact on the user, relative effort to fix, along with a comparison against other open defects. Depending on the size and structure of the organization, the prioritization is often handled by a formal change control board. The priority should be determined with representation from management, the customer, and the project team.
4. **Assignment:** Once a defect has been prioritized, it is then assigned to a developer or other technician to fix.

5. **Resolution:** The developer fixes (resolves) the defect and follows the organization's process to move the fix to the environment where the defect was originally identified.
6. **Verification:** Depending on the environment where the defect was found and the fix was applied, the software testing team or customer typically verifies that the fix actually resolved the defect.
7. **Closure:** Once a defect has been resolved and verified, the defect is marked as closed.
8. **Management Reporting:** Management reports are provided to appropriate individuals at regular intervals as defined reporting requirements. In addition, on demand reports are provided on an as-needed basis.

### 3.6.5.4 Release Management Process

- A system release is a version of a software system that is distributed to customers. Major releases are very important economically to the software vendor as customers have to pay for these. Minor releases are usually distributed free of charge. A system release is not just the executable code of the system.
- The release may also include:
  1. Configuration files defining how the release should be configured for particular installations.
  2. Data files, such as files of error messages that are needed for successful system operation.
  3. An installation program that is used to help install the system on target hardware.
  4. Electronic and paper documentation describing the system.
  5. Packaging and associated publicity that have been designed for that release.
- Release creation is the process of creating the collection of files and documentation that includes all of the components of the system release.
- Release management is the process of managing software releases from development stage to software release. It is rapidly growing discipline within software engineering. As software systems, software development processes, and resources become more distributed, they invariably become more specialized and complex.
- The need therefore exists for dedicated resources to oversee the integration and flow of development, testing, deployment, and support of these systems. Release managers address this need. They must have a general knowledge of every aspect of the software development process, various applicable operating systems and software application or platforms, as well as various business functions and perspectives.

**Challenges facing by a Software Release Manager:**

- Challenges facing by a software release manager during Release Management are:
  - Software defects
  - Issues

- Risks
- Software change requests
- New development requests (additional features and functions)
- Deployment and packaging
- New development tasks

**Technical and Organizational Factors:**

- The various technical and organizational factors (as shown in following table) need to consider when deciding to release a new version of a system.

**Table 3.23: Technical and Organizational Factors of System**

| Factor | Description |
|---|---|
| **Technical quality of the system** | If serious system faults are reported which affect the way in which many customers use the system. It may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system. |
| **Platform changes** | You may have to create a new release of a software application when a new version of the operating system platform is released. |
| **Lehman's fifth law** | This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance. |
| **Competition** | For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers. |
| **Marketing requirements** | The marketing department of an organization may have made a commitment for releases to be available at a particular date. |
| **Customer change proposals** | For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented. |

- When a system release is produced, it must be documented to ensure that it can be recreated exactly in the future. This is important for customized long-lifetime embedded systems as these systems control complex machines. The programs and all associated data files must be identified in the version management system and tagged with the release identifier.

**Release Types:**

- As there are different purposes that releases of new versions are scheduled for, Release is classified in three different types as explained below:
  1. **Patch Release:** Patch releases are designated for bug fixes and small changes without changes to the Application Programming Interface (API). No new features are introduced in patch releases. Only existing features are maintained. They are fully for and backward compatible to the other releases in the same minor version line.
  2. **Minor Release:** In a Minor release, new features can be introduced and new API functionality may be added. Existing API functions must not be changed. Therefore code that was designed to work with a previous version in the same major version still has to work when upgrading to a new minor version. If new functionality is already in use, downgrading to an older minor release will not provide full compatibility anymore.
  3. **Major Release:** Major releases contain the biggest possible change sets. Only in major releases it is allowed to change or remove existing API functions.

## 3.6.5.5 Configuration Management Tools

- Now a day's various SCM tools are available in market which can be used to make configuration tasks easy. Here we will discuss only four most commonly used SCM tools.

1. **SVN:**

- SVN is a free/open source Version Control System tool, which is a subset of SCM. SVN **(Subversion)** is a tool that provides.versioning functionality. Developers used this tool to check in their codes. Using this we can track down the changes made to the files/codes during the software development process. It aims to solve the same sorts of problems, but does so by forcing all activity to go through one location that everyone uses.
- That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine."
- Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit if some incorrect change is made to the data, and then just undo that change.
- Some version control systems are also *Software Configuration Management* (SCM) systems. These systems are specifically tailored to manage trees of source code and

have many features that are specific to software development such as natively understanding programming languages, or supplying tools for building software.

- Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files. For you, those files might be source code but for others, it might be anything from grocery shopping lists to digital video mix-downs and beyond.

**Subversion's Architecture:**

- Figure 3.10, "Subversion's architecture" illustrates a "mile-high" view of Subversion's design.

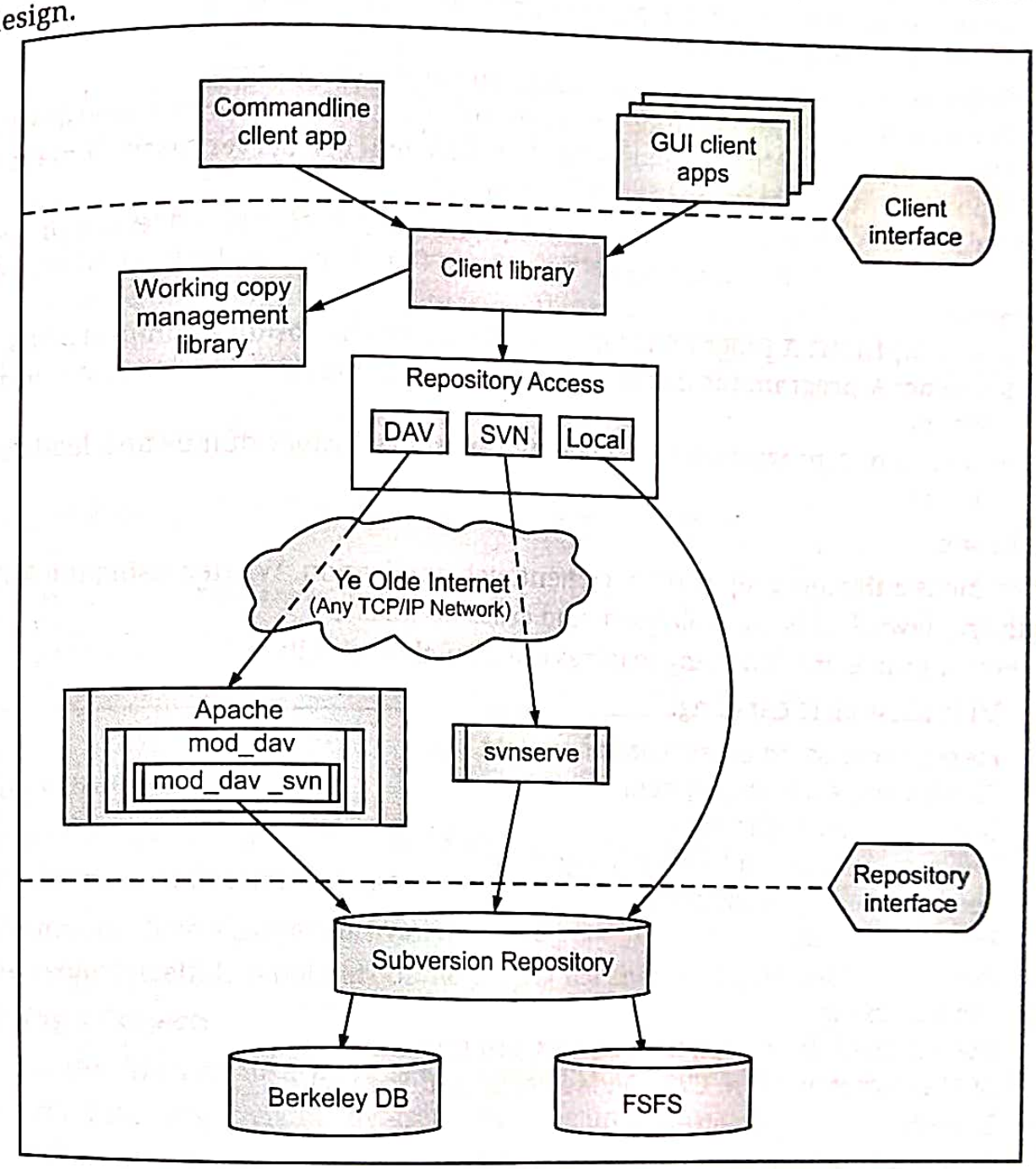



**Fig. 3.10: Subversion's Architecture**

- On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of

portions of that versioned data. Between these extremes are multiple routes through a Repository Access (RA) layer, some of which go across computer networks and through network servers which then access the repository, others of which bypass the network altogether and access the repository directly.

**Subversion's Components:**

- Subversion, once installed, has a number of different pieces. The following is a quick overview of what you get:
  - **Svn:** The command-line client program.
  - **Svnversion:** A program for reporting the state (in terms of revisions of the items present) of a working copy.
  - **Svnlook:** A tool for directly inspecting a Subversion repository.
  - **Svnadmin:** A tool for creating, tweaking, or repairing a Subversion repository.
  - **mod_dav_svn:** A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network.
  - **Svnserve:** A custom standalone server program, runnable as a daemon process or invoked by SSH. Another way to make your repository available to others over a network.
  - **Svndumpfilter:** A program for filtering Subversion repository dump streams.
  - **Svnsync:** A program for incrementally mirroring one repository to another over a network.
  - **Svnrdump:** A program for performing repository history dumps and loads over a network.

2. **Redmine:**

- Redmine is a flexible project management web application. Written using the Ruby on Rails framework, it is cross-platform and cross-database.
- It is open source and following features are available with it:
  - Multiple project handling.
  - Flexible role based access control.
  - Flexible issue tracking system.
  - Gantt chart and calendar.
  - News, documents and files management.
  - Feeds and email notifications.
  - Per project wiki.
  - Per project forums.
  - Time tracking.
  - Custom fields for issues, time-entries, projects and users.
  - SCM integration (SVN, CVS, Git, Mercurial and Bazaar).
  - Issue creation via email.
  - Multiple LDAP authentication support.
  - User self-registration support.
  - Multilanguage support.
  - Multiple databases support.

**(a) Projects List:**

- The Projects list shows all active projects by default. To view all projects (active and archived projects) changes the status filter to all. Note that there is also the project list for regular users that can be accessed by clicking on *Projects* in the top menu. Non-administrator can create new projects in this list (and close projects within a project), but cannot do more administrative operations as it is possible in project list within the administration area that gets discussed here. Depending on the user permissions, here presented project list may be the only place to manage projects at all.
  - **Project:** The project name.
  - **Description:** A short description of the project.
  - **Public:** If the icon ✓ is present the project is public and everyone can see this project and may do things that are allowed by the roles *Non-member* and *Anonymous*. Non-public projects can be viewed only by a user who was given access by an administrator of the project.
  - **Created:** Indicated the date when this project has been created.

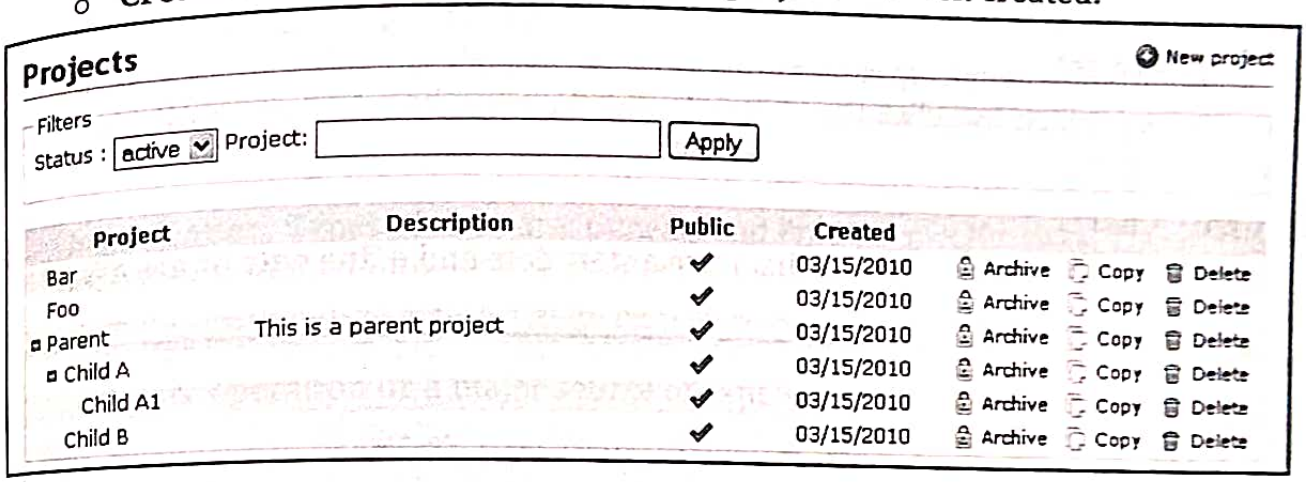



**Fig. 3.11: Project List**

**Closing a Project:**

- This sets a project to a read-only state. A closed project is still being accessible like a regular project but nothing can be changed anymore, thus it is read-only.
- The link to close a project is available on a per Project basis in the **Overview** screen of the project itself. It is not part of the project list shown above.

**Archiving a Project:**

- From the Projects list, click the **Archive** link to archive a project. An archived project is no longer visible by users. It can be un-archived (with its original content) by the administrator. When archiving a project, any subproject is also archived.
- Note that archiving a project does not reduce storage usage. It only hides the project but does not compress it.

**Copying an Existing Project:**

- From the Projects list, click the **Copy** link on the right of the project you want to copy.
- You will get the new project form prefilled with the settings of the copied project (trackers, custom fields...). At the bottom of the form, you can choose what should be also copied to your new project:

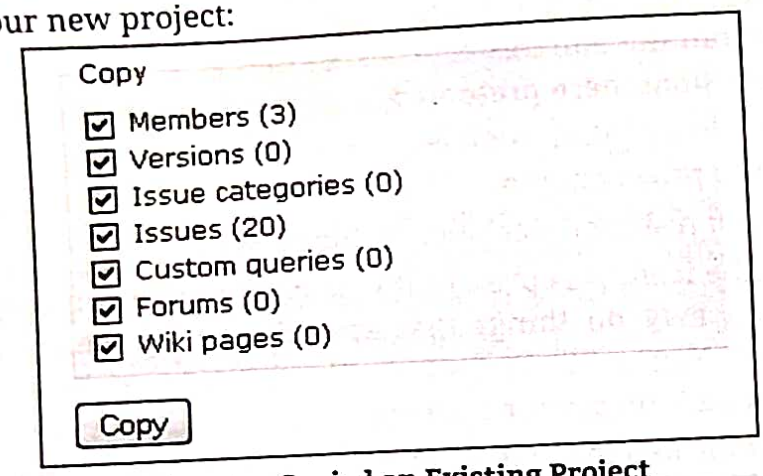



**Fig. 3.12: Copied an Existing Project**

**Deleting a Project:**

- From the projects list, click the **Delete** link. You will be asked for confirmation on a separate screen.

**(b) Gantt Chart :**

- The Gantt chart displays issues that have a start date and a due date or are assigned to a version with a date.

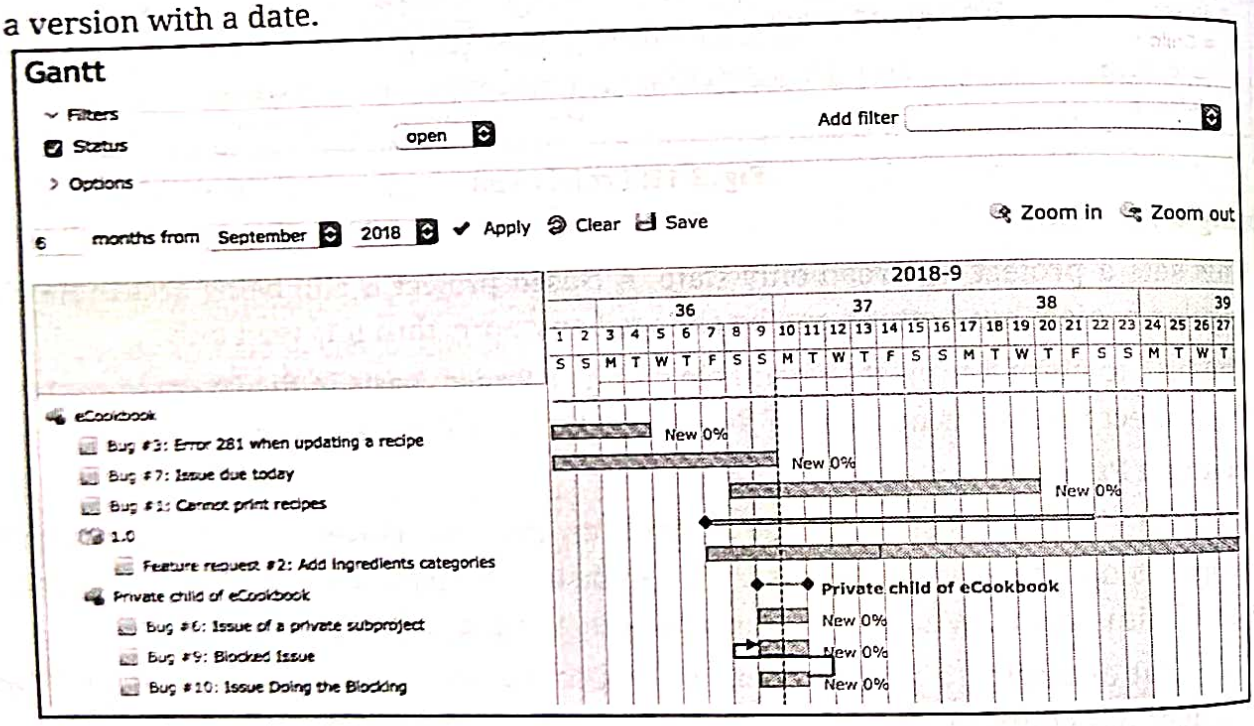



**Fig. 3.13: Gantt chart displays Date Issues**

**c) Time Tracking:**

Time tracking feature allows users to track how many hours of work are logged to a specific issue or project. Each time entry of "spent time" can be categorized by activity and further explained with comments. By filling time estimates, project managers are able to produce better suited project planning as well as follow individual user's progress.

Time tracking is always related to a user, thus it can be used to track how many billable hours this user has accomplished.

## SUMMARY

- A project is a unique venture to produce a set of deliverables within clearly specified time, cost and quality constraints.
- Project Management is the process by which a project is initiated, planned, controlled, and brought to a conclusion to support the accomplishment of business and system objectives.
- From beginning to end in a project numerous activities and deliverables have to be managed, the aggregation of these management methods is called as Project Management Life Cycle (PMLC).
- Software quality is the degree of conformance to requirements and expectations. The three aspects of software quality are: Functional quality, Structural quality, Process quality.
- Risks may threaten the project, the software that is being developed, or the organization. Types of risks are: Generic and Product-specific risks. These risks can be further divided into Project, Product, and Business risks.
- Risk management is a series of steps whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or a major source of expensive rework.
- Stages of Risk Management Process are: Risk identification, Risk analysis, Risk planning and RMMM (Risk Mitigation, Monitoring, and Management) plan.
- The cost of medium and large software projects is determined by the cost of developing the software, plus the cost of equipment and supplies.
- Entities in Project estimation are: Estimating Project/Software Size, Estimating Project Cost, Estimating Effort, and Estimating Project Schedule.
- The term COCOMO stands for "Constructive Cost Model". This model is used to estimate effort, cost and schedule for software projects.
- Boehm proposed three levels of the COCOMO model: Basic, Intermediate and Detailed.
- The most fundamental calculation in the COCOMO model is the use of the Effort Equation to estimate the number of Person-Months required developing a project. Most of the other COCOMO results, including the estimates for Requirements and Maintenance, are derived from this quantity.
- Delphi Cost Estimation method requires several cost estimation experts and a coordinator to direct the process. Under this method of software estimation, the project specifications would be given to a few experts and their opinion taken.

# CHECK YOUR UNDERSTANDING

1. Different activity of a project management is _____.
   (a) Project Planning (b) Project monitoring
   (c) Project control (d) all of the above
2. Which of the following activity is undertaken immediately after feasibility study a before the requirement analysis and specification phase?
   (a) Project Planning (b) Project Monitoring
   (c) Project Control (d) Project Scheduling
3. This activity is undertaken once the development activities start?
   (a) Project Planning (b) Project Monitoring and Control
   (c) Project size estimation (d) Project cost estimation
4. Which of the following activity is not the part of project planning?
   (a) Project estimation (b) Project scheduling
   (c) Project monitoring (d) Risk management
5. In the project planning, which of the following is considered as the most ba parameter based on which all other estimates are made?
   (a) Project size (b) Project effort
   (c) Project duration (d) Project schedule
6. During project estimation, project manager estimates following :
   (a) Project cost (b) Project duration
   (c) Project effort (d) All of the above
7. Which of the following is not project management goal?
   (a) Keeping overall costs within budget.
   (b) Delivering the software to the customer at the agreed time.
   (c) Maintaining a happy and well-functioning development team.
   (d) Avoiding customer complaints.
8. Project managers have to assess the risks that may affect a project. State true/fa
   (a) True
   (b) False
9. Which of the following is not considered as a risk in project management?
   (a) Specification delays (b) Product competition
   (c) Testing (d) Staff turnover
10. The process each manager follows during the life of a project is known as _____
    (a) Project Management (b) Development life cycle
    (c) Project Management Life Cycle (d) All of the mentioned
11. A 66.6% risk is considered as _____.
    (a) very low (b) low
    (c) moderate (d) high
12. Which of the following is/are main parameters that you should use when compu the costs of a software development project?
    (a) Travel and training costs.
    (b) Hardware and software costs.
    (c) Effort costs (the costs of paying software engineers and managers).
    (d) All of the mentioned

13. Quality planning is the process of developing a quality plan for ___.
(a) team (b) project
(c) customers (d) project manager

14. Which of the following is an important factor that can affect the accuracy and efficacy of estimates?
(a) Project size (b) Planning process
(c) Project complexity (d) Degree of structural uncertainty

15. Which of the following uses empirically derived formulas to predict effort as a function of LOC or FP?
(a) FP-Based Estimation
(b) Process-Based Estimation
(c) COCOMO
(d) Both FP-Based Estimation and COCOMO

16. The empirical data that support most estimation models are derived from a vast sample of projects.
(a) True (b) False

17. COCOMO stands for _____.
(a) Constructive cost model (b) Comprehensive cost model
(c) Constructive cost estimation model (d) Complete cost estimation model

18. Which version of COCOMO states that once requirements have been stabilized, the basic software architecture has been established?
(a) Early design stage model (b) Post-architecture-stage model
(c) Application composition model (d) All of the mentioned

19. Which one is not a size measure for software product?
(a) LOC (b) Halstead's program length
(c) Function Count (d) Cyclomatic Complexity

20. COCOMO was developed initially by _____.
(a) B. Beizer (b) Rajiv Gupta
(c) B. W. Bohem (d) Gregg Rothermal

21. Estimation of size for a project is dependent on _____.
(a) Cost (b) Time
(c) Schedule (d) None of the mentioned

**Answers**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1. (d) | 2. (a) | 3. (b) | 4. (c) | 5. (a) | 6. (d) | 7. (d) | 8. (b) | 9. (c) | 10. (c) |
| 11. (d) | 12. (d) | 13. (b) | 14. (a) | 15. (d) | 16. (b) | 17. (a) | 18. (a) | 19. (d) | 20. (c) |
| 21. (d) | | | | | | | | | |

## PRACTICE QUESTIONS

**Q.I Answer the following questions in short.**

1. Write names of Project constraints ?
2. What is PERT?
3. Write the difference between CPM and PERT.
4. Write steps how to create a task list in MS project.
5. Which are Predecessor and dependency types w.r.t. MS project?

6. Write the step to view Gantt chart, network diagram and schedule table in MS Project.
7. What is version control?
8. What is versioning model?
9. Define the release management.
10. Write difference COCOMO-I and COCOMO-II.
11. List technical and organizational factors need to consider during release management.
12. Which variables are used in basic cost estimation model?
13. Write equations and parameters of Basic COCOMO model.

**Q.II Answer the following questions.**

1. What is Project? List characteristics of Project.
2. What is project management? Explain its components.
3. Explain project constraints.
4. Write in detail roles and responsibilities in project organization.
5. Explain project organization structure.
6. Explain Communication steps.
7. Explain communication documents.
8. Describe phases of PMLC.
9. What is risk identification? Explain with risk list.
10. Explain risk analysis and give impact and probability of risk.
11. Explain MS Project environment.
12. What is CPM? Describe with example.
13. Explain software project cost estimation techniques.
14. Explain COCOMO-I with its Model.
15. Explain Delphi cost estimation technique.
16. Explain NPV, ROI and Payback Model.
17. Explain Function Point analysis with example.
18. Explain the use of Rayleigh curve in project management.
19. Write a case study on Function Point Analysis.
20. What is software configuration management? Explain activities of configuration management.
21. Explain change management process.
22. What is defect management? Explain defect management process.
23. Explain different tools used in configuration management.
24. Explain roles and responsibilities which are involved in change management.

**Q.III Write a short note on:**

1. Slack Time
2. Earliest Start Time and Late Start Time
3. Latest Finish and Earliest Finish Time
4. Software Quality
5. Process and Activity
6. Software Size Estimation Techniques.
7. Domain Values in FPA.

***

UNIT 4

# Agile Project Management Framework

**Objectives...**

After reading this chapter, you will be able:

- To understand concept of Agile and Agile project life cycle.
- To know history of Agile and Agile principles.
- To get information of key Agile concepts.
- To understand difference between Agile project management and traditional project management.
- To study about Agile Reports.

## 4.1 INTRODUCTION AND DEFINITION OF AGILE

- Agile is a set of principles that are used to improve the process of project management and software development. To put in simple terms, Agile helps teams in delivering value to customers quickly and effortlessly.
- Agile software development is based on an incremental, iterative approach. Instead of in-depth planning at the beginning of the project, agile methodologies are open to changing requirements over time and encourage constant feedback from the end users. Cross-functional teams work on iterations of a product over a period of time, and this work are organized into a backlog that is prioritized based on business or customer value. The goal of each iteration is to produce a working product.

**Agile Software Development (or System Development) Methodologies:**

- Agile is a framework and there are a number of specific methods within the Agile movement. You can think of these as different features of Agile:
  - **Extreme Programming (XP):** Also known as XP, Extreme Programming is a type of software development intended to improve quality and responsiveness to evolving customer requirements. The principles of XP include feedback, assuming simplicity, and embracing change.

- **Feature-driven development (FDD):** This iterative and incremental software development process blends industry best practices into one approach. There are five basic activities in FDD: develop overall model, build feature list, plan by feature, design by feature and build by feature.
- **Adaptive system development (ASD):** Adaptive system development represents the idea that projects should always be in a state of continuous adaptation. ASD has a cycle of three repeating series: Speculate, Collaborate and Learn.
- **Dynamic Systems Development Method (DSDM):** This Agile project delivery framework is used for developing software and non-IT solutions. It addresses the common failures of IT projects, like going over budget, missing deadlines, and lack of user involvement. The eight principles of DSDM are: focus on the business need, deliver on time, collaborate, never compromise quality, build incrementally from firm foundations, develop iteratively, communicate continuously and clearly, and demonstrate control.
- **Lean Software Development (LSD):** Lean Software Development takes Lean manufacturing and Lean IT principles and applies them to software development. It can be characterized by seven principles: eliminate waste, amplify learning, and decide as late as possible, deliver as fast as possible, empower the team, build integrity in, and see the whole.
- **Kanban:** Kanban, meaning "visual sign" or "card" in Japanese, is a visual framework to implement Agile. It promotes small, continuous changes to your current system. Its principles include: visualize the workflow, limit work in progress, manage and enhance the flow, make policies explicit, and continuously improve.
- **Crystal Clear:** Crystal Clear is part of the Crystal family of methodologies. It can be used with teams of six to eight developers and it focuses on the people, not processes or artifacts. Crystal Clear requires the following: frequent delivery of usable code to users, reflective improvement, and osmotic communication preferably by being co-located.
- **Scrum:** Scrum is one of the most popular ways to implement Agile. It is an iterative software model that follows a set of roles, responsibilities, and meetings that never change. Sprints, usually lasting one to two weeks, allow the team to deliver software on a regular basis.

## 4.1.1 Agile Project Life Cycle

- The overall goal of each agile method is to adapt to change and deliver working software as quickly as possible. However, each methodology has slight variations in the way it defines the phases of software development.

- Furthermore, even though the goal is the same, each team's process flow may vary depending on the specific project or situation. As an example, the full Agile software development lifecycle includes the Concept, Inception, Construction, Release, Production, and Retirement phases.

1. **Concept:** Projects are envisioned and prioritized.
2. **Inception:** Team members are identified, funding is put in place, and initial environments and requirements are discussed.
3. **Iteration/Construction:** The development team works to deliver working software based on iteration requirements and feedback.
4. **Release:** QA (Quality Assurance) testing, internal and external training, documentation development, and final release of the iteration into production.
5. **Production:** Ongoing support of the software.
6. **Retirement:** End-of-life activities, including customer notification and migration.

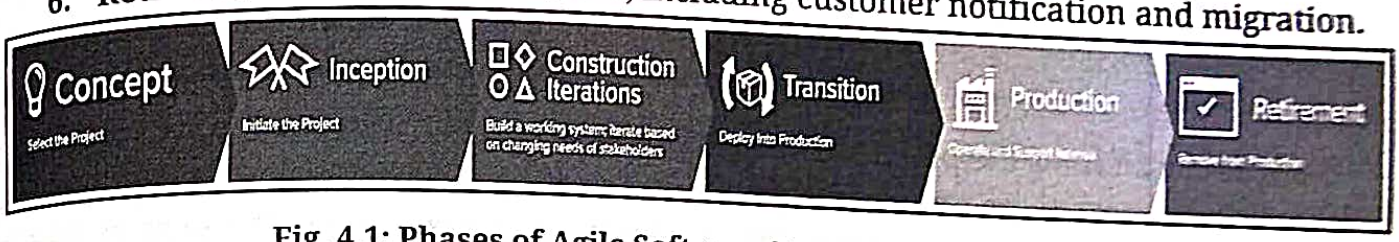



**Fig. 4.1: Phases of Agile Software Development Lifecycle**

- The Agile software development lifecycle is dominated by the iterative process. Multiple iterations will take place during the Agile software development lifecycle and each follows its own workflow. During iteration, it is important that the customers and business stakeholders provide feedback to ensure that the features meet their needs.

**Iteration Process Flow:**

- A typical iteration process flow can be visualized as follows:
  - **Requirements:** Define the requirements for the iteration based on the product backlog, sprint backlog, customer and stakeholder feedback.
  - **Development:** Design and develop software based on defined requirements.
  - **Testing:** QA (Quality Assurance) testing, internal and external training, documentation development. User Acceptance Testing (UAT) is performed after the software has been thoroughly tested. It is sometimes known as End User Testing.
  - **Delivery:** Integrate and deliver the working iteration into production.
  - **Feedback:** Accept customer and stakeholder feedback and work it into the requirements of the next iteration.

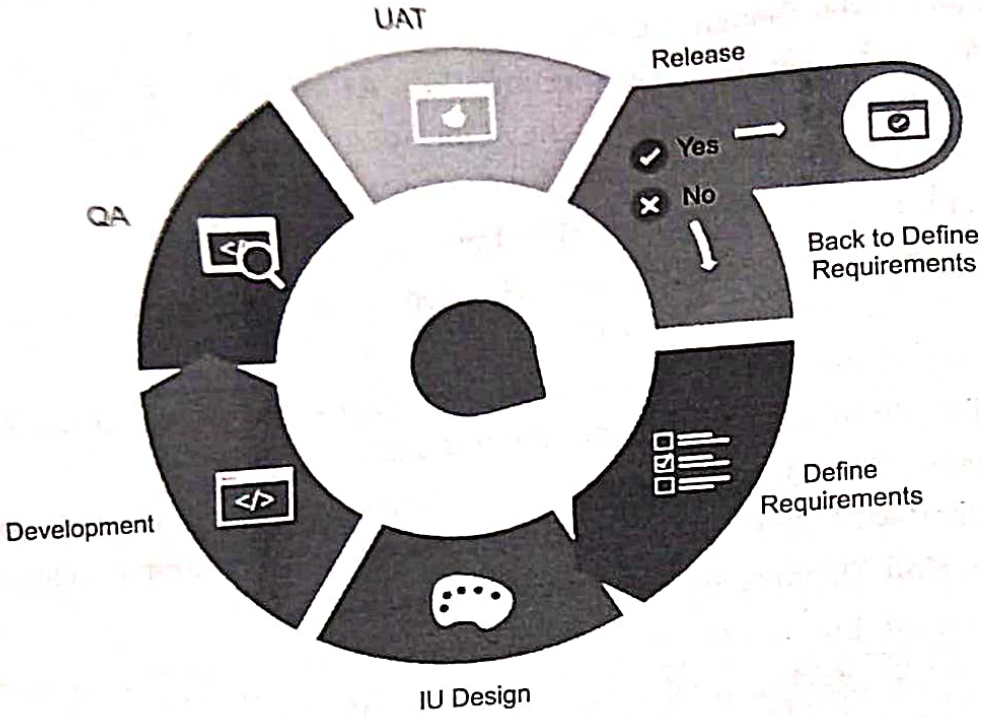



**Fig. 4.2: Iteration Process Flow**

- For the duration of the project, while additional features may be fed into the prod backlog, the rest of the process is a matter of repeating the steps over and over until of the items in the product backlog have been fulfilled. As a result, the process flow more of a loop and not a linear process.

## 4.1.2 Advantages and Disadvantages of Agile

**Advantages of Agile:**

- Here are some of the top advantages of Agile:
  1. **Adapt change more effectively:** With shorter planning cycles, it's easy accommodate and accept changes at any time during the project.
  2. **End-goal can be unknown:** Agile is very beneficial for projects where the end-g is not clearly defined. As the project progresses, the goals will come to light development can easily adapt to these evolving requirements.
  3. **Faster, high-quality delivery:** Breaking down the project into iterat (manageable units) allows the team to focus on high-quality development, tes and collaboration. Conducting testing during each iteration means that bugs identified and solved more quickly. And this high-quality software can delivered faster with consistent, successive iterations.
  4. **Strong team interaction:** Agile highlights the importance of freq communication and face-to-face interactions. Teams work together and people able to take responsibility and own parts of the projects.

5. **Customer satisfaction:** Customers have many opportunities to see the work being delivered, share their input, and have a real impact on the end product.
6. **Continuous improvement:** Agile projects encourage feedback from users and team members throughout the whole project, so lessons learned are used to improve future iterations.

**Disadvantages of Agile:**

- Here are some of the disadvantages of Agile:

1. **Planning can be less concrete:** It can sometimes be hard to pin down a solid delivery date. Because Agile is based on time-boxed delivery and project managers are often reprioritizing tasks, it's possible that some items originally scheduled for delivery may not be complete in time. And, additional sprints may be added at any time in the project, adding to the overall timeline.
2. **Team must be knowledgeable:** Agile teams are usually small, so team members must be highly skilled in a variety of areas. They also must understand and feel comfortable with the chosen Agile methodology.
3. **Time commitment from developers:** Agile is most successful when the development team is completely dedicated to the project. Active involvement and collaboration is required throughout the Agile process, which is more time consuming than a traditional approach.
4. **Documentation can be neglected:** The Agile Manifesto prefers working software over comprehensive documentation, so some team members may feel like it's less important to focus on documentation. While comprehensive documentation on its own does not lead to project success, Agile teams should find the right balance between documentation and discussion.
5. **Final product can be very different:** The initial Agile project might not have a definitive plan, so the final product can look much different than what was initially intended. Because Agile is so flexible, new iterations may be added based on evolving customer feedback, which can lead to a very different final deliverable.

## 4.2 AGILE MANIFESTO - HISTORY OF AGILE

- Iterative and incremental software development methods can be traced back as early as 1957, with evolutionary project management and adaptive software development emerging in the early 1970s.
- During the 1990s, a number of lightweight software development methods evolved in reaction to the prevailing heavyweight methods (often referred to collectively as waterfall) that critics described as overly regulated, planned, and micro-managed. These included: Rapid Application Development (RAD), from 1991; the Unified Process (UP) and Dynamic Systems Development Method (DSDM), both from 1994; Scrum,

from 1995; Crystal Clear and Extreme Programming (XP), both from 1996; and Feature Driven Development, from 1997. Although these all originated before the publication of the Agile Manifesto, they are now collectively referred to as agile software development methods. At the same time, similar changes were underway in manufacturing and management thinking.

- In 2001, these seventeen software developers (Kent Beck, Ward Cunningham, Dave Thomas, Jeff Sutherland, Ken Schwaber, Jim Highsmith, Alistair Cockburn, Robert C Martin, Mike Beedle, Arie van Bennekum, Martin Fowler, James Grenning, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, and Steve Mellor) met at a resort in Snowbird, Utah to discuss these lightweight development methods. Together they published the Manifesto for Agile Software Development.
- In 2005, a group headed by Cockburn and Highsmith wrote an addendum of project management principles, the PM Declaration of Interdependence, to guide software project management according to agile software development methods.
- In 2009, a group working with Martin wrote an extension of software development principles, the Software Craftsmanship Manifesto, to guide agile software development according to professional conduct and mastery.
- In 2011, the Agile Alliance created the Guide to Agile Practices (renamed the Agile Glossary in 2016), an evolving open-source compendium of the working definitions of agile practices, terms, and elements, along with interpretations and experience guidelines from the worldwide community of agile practitioners.

## 4.2.1 Agile Principles

- In 2001, seventeen software developers came together in Snowbird, Utah to propose a new way of developing software "by doing it and helping others do it". Through their work, the signers of the Manifesto understood how much of an impact these principles would help them in the field of software development but they had no idea how quickly their ideas would spread beyond their industry.
- Agile Values the Manifesto creators mentioned were:
  - Individuals and interactions over processes and tools.
  - Working software over comprehensive documentation.
  - Customer collaboration over contract negotiation.
  - Responding to change over following a plan.
- Since that time, the original document has been used by groups as disparate as Girl Scout Troops, from marketing departments to restaurants. Its universal appeal is derived from a group of principles that can be broadly applied, easily learned, and rarely mastered completely. Before spreading to all corners of the globe, here are the key principles for incremental development that have made Agile what it is today.

- The Agile Manifesto lists 12 principles to guide teams on how to execute with agility. These are the principles:

1. Customer Satisfaction through Early and Continuous Software Delivery. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Accommodate changing requirements throughout the development process. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Collaboration between the business stakeholders and developers throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential. You need to focus on the things that are important to add value to the project and customers. Ignore the things that do not add value, such as components, process, etc.
11. The best architectures, requirements, and designs emerge from self- organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

## 4.3 TEAM AND ROLES OF AN AGILE TEAM

- The Agile Software Development Lifecycle (SDLC) was developed with a clear goal: Rapid delivery of software builds through an incremental and iterative process designed to adapt and improve software quality from an end-user perspective. The goal is readily adopted by IT departments and shops, though the process framework is not always adequately adopted.

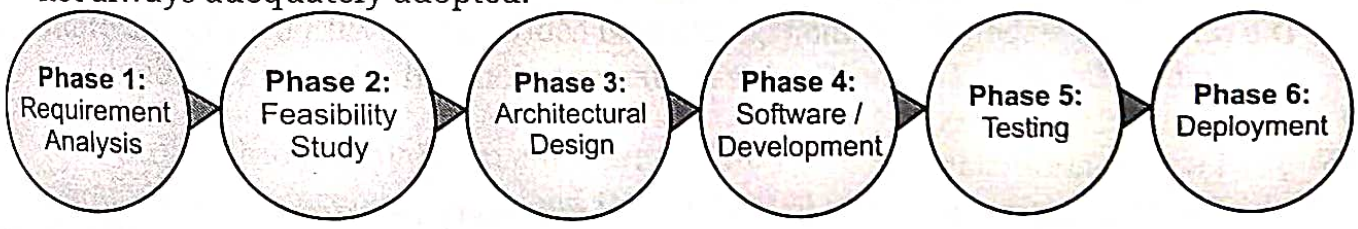



**Fig. 4.3: Phases in the SDLC**

- Organizations adopting an Agile approach may find themselves resorting to traditional/waterfall SDLC practices due to inappropriate distribution of Agile roles and responsibilities. So, let's take a look at the roles that support Agile software development.

**Roles in an Agile Team:**

- This section explores the roles and responsibilities within the Scrum framework for Agile implementation. Some key differences in Agile team building exercise include:
  1. The development of holistic teams with cross-functional expertise.
  2. Domain specialists with a broad knowledge and view of the business aspects associated with their work areas.
  3. Stable team structures that can iterate and improve the SDLC workflows on a continuous basis.
- Agile teams are often comprised of the following key roles and responsibilities:

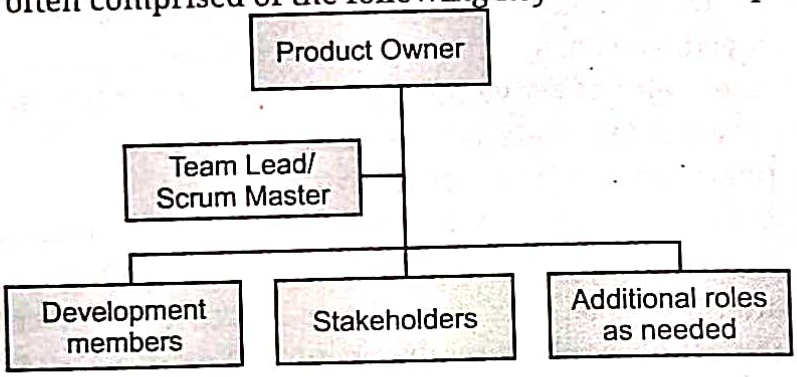



**Fig. 4.4: Roles of Agile Team**

## 4.3.1 Scrum Master

- The Team Lead or Scrum Master ensures team coordination and supports the progress of the project between individual team members. The Scrum Master takes the instructions from the Product Owner and ensures that the tasks are performed accordingly.
- The role may involve:
  - Facilitating the daily Scrum and Sprint initiatives.
  - Communicating between team members about evolving requirements and planning.
  - Coaching team members on delivering results.
  - Handling administrative tasks such as conducting meetings, facilitating collaboration, and eliminating hurdles affecting project progress.
  - Shielding team members from external interferences and distractions.
- The role is also responsible to manage external coordination with the organization and the Product Owner to ensure effective implementation of the Scrum framework.
- The responsibilities may include:
  - Implementing changes.
  - Coordinating between stakeholders to find necessary resources.
  - Helping Product Owners optimize the backlog planning for optimum performance.

- The role of a Scrum Master is focused on attributes such as transparency across the Scrum Team, self-organization, commitment, respect and most importantly, following an empirical process to identify the best approach for product development.

## 4.3.2 Product Owner

- The Product Owner represents the stakeholders of the project. The role is primarily responsible for setting the direction for product development or project progress.
- The Product Owner understands the requirements of the project from a stakeholder perspective and has the necessary soft skills to communicate the requirements to the product development team. The Product Owner also understands the long-term business vision and aligns the project with the needs and expectations of all stakeholders. End-user feedback is taken into account to determine appropriate next-best action plans for the development throughout the project cycle.
- The key responsibilities of a Product Owner include:
  1. Scrum backlog management
  2. Release management
  3. Stakeholder management
- The Product Owner is knowledgeable of the backlog items added to the list as well as items selected for work. The Product Owner changes and sets the priority of backlog item list based on stakeholder feedback and project circumstances. The role also manages the release cycle planning to ensure that the development team can deliver updated project iterations on a continuous basis.
- Finally, the Product Owner ensures that product development translates into value for the stakeholders. Communication with end-users, business executives, partners and the development team is a key responsibility.

## 4.3.3 Development Team

- The team members within the Development Team are comprised of individuals with responsibilities including but not limited to product development. The team takes cross-functional responsibilities necessary to transform an idea or a requirement into a tangible product for the end-users.
- The required skills might be wrapped up in one or more development team members:
  - Product designer
  - Writer
  - Programmer
  - Tester
  - UX specialist
- Not every member may be an engineer, but may be a part of the team if their skills are required for the project to proceed at the required pace.
- In addition to the skills facilitating product development, the team members should also boast soft skills that would enable them to self-organize and get the work done.

This means that when an issue occurs, the team is both capable and empowered to take corrective actions.

- The key responsibilities of the Development Team are to perform work sprints as per the requirements provided by the Product Owner and coordinated by the Scrum Master.
- A regular standup meeting called the Daily Scrum is followed to communicate project progress with the peers and the Scrum Master. This activity ensures transparency and allows the Development Team to incorporate the changes as necessary in future sprints based on feedback from the Product Owner.

## 4.4 KEY AGILE CONCEPTS

### 4.4.1 User Stories

- In software development and product management, a user story is an informal, natural language description of one or more features of a software system. A user story is a tool used in Agile software development to capture a description of a software feature from an end-user perspective. A user story describes the type of user, what they want and why. A user story helps to create a simplified description of a requirement.
- User stories are often recorded on index cards, on Post-it notes, or in project management software. Depending on the project, user stories may be written by various stakeholders such as clients, users, managers or development team members.

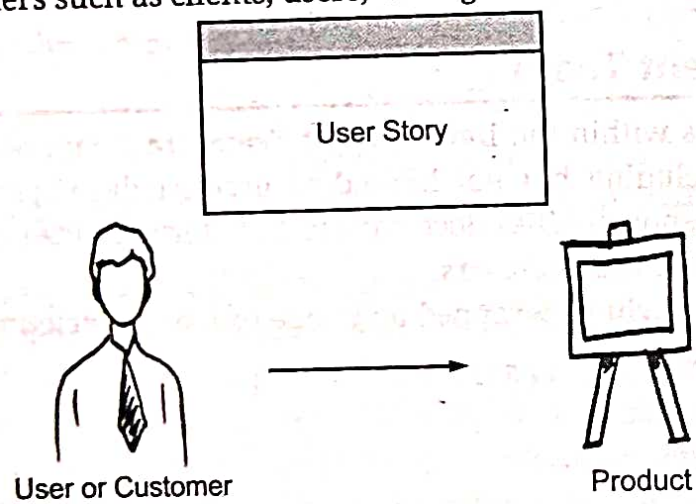



**Fig. 4.5: Concept of User Story**

- Large user stories that are complex to understand when defined using the standard format are called epics. For simplification, the user stories are divided into smaller understandable user stories.

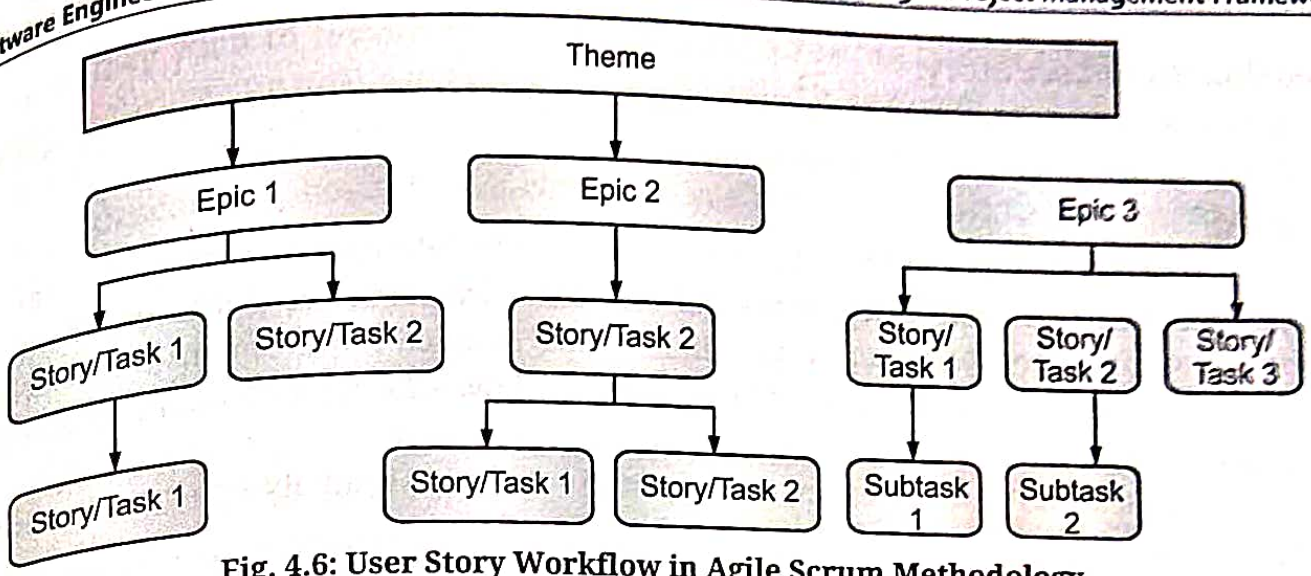



**Fig. 4.6: User Story Workflow in Agile Scrum Methodology**

**Benefits for Adopting User Story Approach in Agile Development:**

- There are a few benefits for adopting user story approach in Agile development such as:
  1. The simple and consistent format saves time when capturing and prioritizing requirements while remaining versatile enough to be used on large and small features alike.
  2. Keep yourself expressing business value by delivering a product that the client really needs.
  3. Avoid introducing detail too early that would prevent design options and inappropriately lock developers into one solution.
  4. Avoid the appearance of false completeness and clarity.
  5. Get to small enough chunks that invite negotiation and movement in the backlog.
  6. Leave the technical functions to the architect, developers, testers, and so on.

**User Story Template:**

- User stories only capture the essential elements of a requirement:
  - Who it is for?(Role)
  - What it expects from the system?(Action)
  - Why it is important (optional?)?(Benefit)
- Here is a simple format of user story used by 70% of practitioners:

**Template** : As a <role> I can <action>, so that <receive benefit>

**Examples:**

- As a **[customer]**, I want **[shopping cart feature]** so that **[I can easily purchase items online]**.
- As an **[manager]**, I want to **[generate a report]** so that **[I can understand which departments need more resources]**.
- As a **[customer]**, I want to **[receive SMS when the item is arrived]** so that **[I can go to pick it up right away]**.

**Components of User Story:**

- A User Story has three primary components, each of which begins with the letter 'C': Card, Conversation, and Confirmation.

1. **Card:**

- Card represents 2-3 sentences used to describe the intent of the story that can be considered as an invitation to conversation. The card serves as a memorable token which summarizes intent and represents a more detailed requirement, whose details remain to be determined. The Card is usually follows the format similar to the one below:
  - As a **(role)** of the product, I can (do **action**) so that I can obtain (some **benefits**/ value).
  - The written text, the invitation to a conversation, must address the "**who (role)**", "**what (action)**" and "**why (benefits)**" of the story.

2. **Conversation:**

- Conversation represents a discussion between the target users, team, product owner, and other stakeholders, which is necessary to determine the more detailed behavior required to implement the intent. In other words, the card also represents a "promise for a conversation" about the intent.
- The collaborative conversation facilitated by the Product Owner which involves all stakeholders and the team.
- The conversation is where the real value of the story lies and the written Card should be adjusted to reflect the current shared understanding of this conversation.
- This conversation is mostly verbal but most often supported by documentation and ideally automated tests of various sorts (e.g. Acceptance Tests).

3. **Confirmation:**

- Confirmation represents the Acceptance Test, which is how the customer or product owner will confirm that the story has been implemented to their satisfaction. In other words, Confirmation represents the conditions of satisfaction that will be applied to determine whether or not the story fulfils the intent as well as the more detailed requirements.
- The Product Owner must confirm that the story is complete before it can be considered "done".
- The team and the Product Owner check the "doneness" of each story in light of the Team's current definition of "done".
- Specific acceptance criteria that are different from the current definition of "done" can be established for individual stories, but the current criteria must be well understood and agreed to by the Team. All associated acceptance tests should be in a passing state

**User Stories – INVEST:**

- The acronym INVEST helps to remember a widely accepted set of criteria or checklist to assess the quality of a user story. If the story fails to meet one of these criteria, the

team may want to reword it, or even consider a rewrite (which often translates into physically tearing up the old story card and writing a new one).

- A good user story should be - **INVEST**:
  - **Independent:** Should be self-contained in a way that allows to be released without depending on one another.
  - **Negotiable:** Only capture the essence of user's need, leaving room for conversation. User story should not be written like contract.
  - **Valuable:** Delivers value to end user.
  - **Estimable:** User stories have to able to be estimated so it can be properly prioritized and fit into sprints.
  - **Small:** A user story is a small chunk of work that allows it to be completed in about 3 to 4 days.
  - **Testable:** A user story has to be confirmed via pre-written acceptance criteria.

**Limitations of User Stories:**

- These are some of the issues that agile development teams have faced when using user stories to capture requirements.

1. **Scale-up problem:** User stories written on small physical cards are hard to maintain, difficult to scale to large projects and troublesome for geographically distributed teams.
2. **Produce vague, informal and incomplete requirements:** User story cards are regarded as conversation starters. Being informal, they are open to many interpretations. Being brief, they do not state all of the details necessary to implement a feature. Therefore, stories are inappropriate for reaching formal agreements or writing legal contracts.
3. **Lack of non-functional requirements:** User stories rarely include performance or non-functional requirement details, so non-functional tests (e.g. response time) may be overlooked.
4. **No need to represent how technology has to be built:** Since user stories are often written from the business perspective, once a technical team begins to implement it may find that technical constraints require effort which may be broader than the scope of an individual story. Sometimes splitting stories into smaller ones can help resolve this. Other times, 'technical-only' stories are most appropriate. These 'technical-only' stories may be challenged by the business stakeholders as not delivering value which can be demonstrated to customers/ stakeholders.

### 4.4.1.1 Story Points

- A story point is a metric used in agile project management and development to estimate the difficulty of implementing a given user story, which is an abstract measure of effort required to implement it. With story points, teams take into account

the effort and complexity to assign each item in a product backlog with a numerical value. Story points are much more comprehensive than looking at only one factor. time to estimate sprint planning.

**Components:**

- Story point estimation includes three main components:
  1. **Risk:** The risk of a particular project or item includes vague demands, dependence on a third party, or changes mid-task.
  2. **Complexity:** This component is determined by how difficult the feature is to develop.
  3. **Repetition:** This component is determined by how familiar the team member is with the feature and how monotonous certain tasks are within development.
- By incorporating above three points, your team can more accurately plan sprints, include cushion for uncertainty, better estimate issues, and avoid leaning too heavily on time commitments. Story points allow for consistency not just in teams, but across departments.

## 4.4.1.2 Techniques for Agile Story Point Estimation

- Follow this process to more accurately plan out your sprints, give realistic expectations, and push projects through faster.

**Step 1. Use Fibonacci Sequence Numbers:**

- It is tempting to assign items with a linear scale, but those integers aren't differentiated enough to clearly define an estimate.
- You've likely encountered this at the doctor's office with a pain scale. If 1 on the pain scale represents "totally fine" and 10 is a pain so severe it feels like you may be dying, what is 4? And, furthermore, how are 4 different from 5? And where does a kidney stone fit in on the scale if you've never experienced severe pain before?
- Fibonacci sequence numbers eliminate those minor jumps. As you might remember, the Fibonacci sequence is a series of numbers where each number is the sum of the two previous numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.
- For Agile, the sequence is typically modified to 0.5, 1, 2, 3, 5, 8, 13, etc. Using these numbers, it's much easier to decide if an item is 3 story points or 5 story points.

**Step 2. Determine a Matrix:**

- After you have decided to use the Fibonacci sequence, it is time to determine a baseline for each story point. For instance:

  1 = Add a new product to a dropdown menu.

  2 = Add order tracking for logged-in users.

  3 = Add a ratings system to the website.

  5 = Add a forum to the site.

  8 = Add GDPR and CCPA compliance across the site.

- Your baseline is included in this matrix as 1, which sets the standard for what the least amount of risk, complexity and repetition looks like in practice. This matrix is a way to more concretely measure effort; keep this in mind instead of defaulting to judging items based only on length of time.

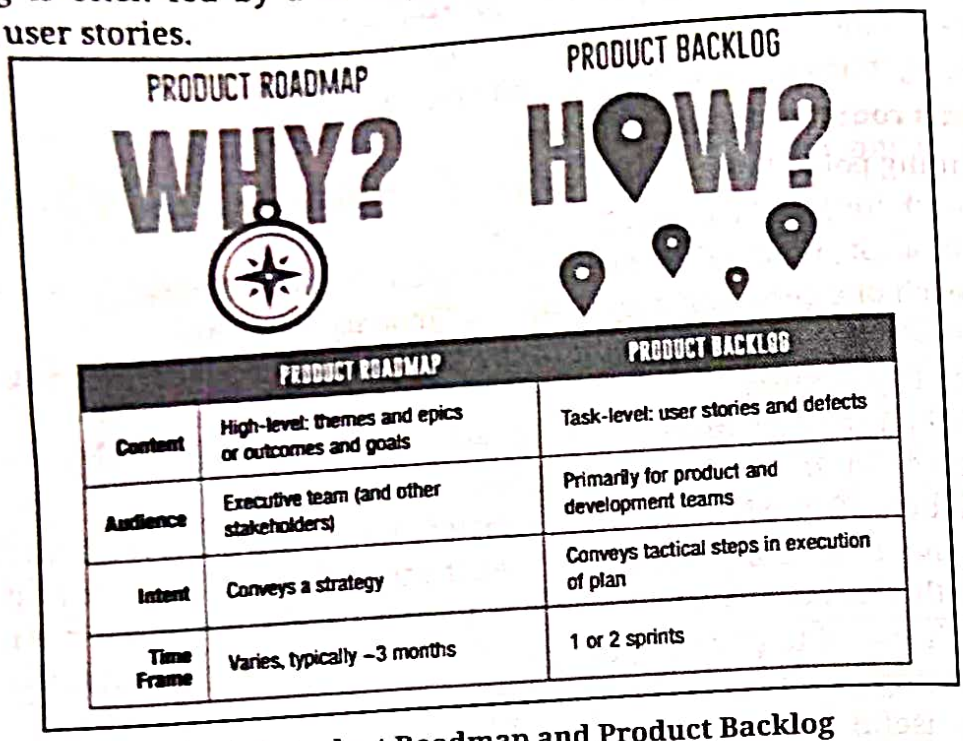
**ep 3. Hold a round of Planning Poker:**

- Planning poker helps a team gain a consensus of correct story point approximation for each item. Here is how it works:
  - In a sprint planning meeting, each developer and tester receives a set of cards, each one depicting a number of a Fibonacci sequence.
  - A backlog item is brought to the table so that the team may ask questions and clarify features.
  - When the discussion is closed, each developer and tester privately selects the card that most accurately reflects their estimate.
  - When all cards have been selected, the estimators reveal their cards at the same time. If a consensus is met, it's time to move on to the next backlog item. If the estimates vary, the leaders discuss until they arrived at a consensus.
  - It is useful to have a completed matrix on hand for the estimators to reference during planning poker, as it allows for greater consistency across tasks. Also, it is useful to set a maximum limit (13, for instance). If a task is estimated to be greater than that limit, it should be split into smaller items. Similarly, if a task is smaller than 1, it should be incorporated into another task.
  - At this point, within your sprint planning meeting, items in the product backlog can be prioritized and divided out amongst the team based on the team's workload capacity.

## 4.4.2 Product Backlog

- A backlog is a list of tasks required to support a larger strategic plan. In a product development context, it contains a prioritized list of items that the team has agreed to work on next. Typical items on a product backlog include user stories, changes to existing functionality, and bug fixes.
- One key component that gives a backlog meaning is that its items are ordered by priority. The items ranked highest on the list represent the most important or urgent items for the team to complete.
- In this context, the purpose of the backlog can be reduced to following three simple goals:
  1. Develop a common ground to align stakeholders and teams, so that teams implement the most valuable user stories.
  2. Provide flexibility to adapt to new needs and realities.
  3. Improve the accuracy of product release forecasts by creating a common denominator across many teams collaborating on one product.

- This backlog is often fed by a strategic roadmap then divided into themes, epics, sprints, and user stories.




| | PRODUCT ROADMAP | PRODUCT BACKLOG |
|---|---|---|
| Content | High-level: themes and epics or outcomes and goals | Task-level: user stories and defects |
| Audience | Executive team (and other stakeholders) | Primarily for product and development teams |
| Intent | Conveys a strategy | Conveys tactical steps in execution of plan |
| Time Frame | Varies, typically ~3 months | 1 or 2 sprints |

**Fig. 4.7: Product Roadmap and Product Backlog**

**Purpose of a Backlog:**

A backlog can serve several important functions for an organization.

1. **Provide a single source of truth for the team's planned work.**

- When a cross-functional team works from a product backlog, the team knows they never need to search for what to work on next, or to wonder in which order they should prioritize their work. It represents an agreed-upon plan for the items the team should tackle next.

2. **Facilitate team discussion.**

- Not every item on a product backlog is fully fleshed out and ready to work on. Sometimes a team will place items on a backlog at the bottom, to indicate they are not yet priority tasks as a springboard for further discussion. This makes them a useful tool to facilitate conversations among a cross-functional team. They help the team discuss how to prioritize work on a product, what (if any) interdependencies or conflicts an item might create, etc.

3. **Make it easier to assign work.**

- When a product team gets together to plan work for a specific upcoming time period, a backlog makes it much easier to assign tasks to each person. The tasks are already written down, ordered according to their priority level, and the team can simply hand out the highest-priority items to the most appropriate members of the team.
- For agile organizations, in particular, this is where a sprint backlog comes in.

**Example of Product Backlog:**

Below is a Product Backlog example for a *social media game*. It is organized by themes, features, and user stories.

| Item name | Status | Product backlog priority | Release tag |
|---|---|---|---|
| ▸ Idea Bucket Delegated to: All project members | ⊖ Not done | | |
| ▾ Social Media Game | ◌ In progress | | |
| ▾ Theme 1 | ◌ In progress | | |
| ▾ Feature 1 | ◌ In progress | | |
| User story 1.1 | ✓ Completed | Very high priority | Release 1 2018-02-19 |
| User story 1.2 | ✓ Completed | High priority | Release 1 2018-02-19 |
| User story 1.3 | ◌ In progress | Medium priority | Release 1 2018-02-19 |
| ▸ Feature 2 | ⊖ Not done | | |
| ▸ Theme 2 Visible to: Outsourced Vendor | ⊖ Not done | | |

**Fig. 4.8: Example of Product Backlog**

Using Agile project management methodologies, projects are broken down into sprints or iterations. These are short, repeatable phases, typically one to four weeks in length. Each sprint should result in a draft, prototype or workable version of the final project deliverable.

The purpose of sprints is to break down a project into bite-sized chunks. This enables the team to plan a single sprint at a time and adapt future sprints based on the outcome of the sprints already completed.

While the planning occurs at the beginning of each sprint, the number of sprints should be determined at the beginning of the project. A sprint in Agile needs to be time boxed, and each sprint must be the same length.

## 4.4.3 Sprint Backlog

- The Sprint Backlog is a container for work the team is committed to doing, either right now as a part of the sprint (typically a one- to four-week period). It is an output of a sprint planning meeting attended by the team.
- The sprint backlog, ideally, doesn't change at all for duration of the sprint. It consists of user stories that the team has committed to delivering within the next sprint's timeframe. But it can also include bugs, refactoring work, and so forth. It is usually more granular, and broken down into tasks, focusing on the technical implementation of a user story. It is the purview of the scrum master and team.
- Sprint Backlog is a set of Product Backlog items selected for the current Sprint, plus plans for delivering product increments for achieving Sprint goals. Sprint Backlog is

the development team's expectation of what functions will be included in the next increment and what work will be required to deliver those functions.

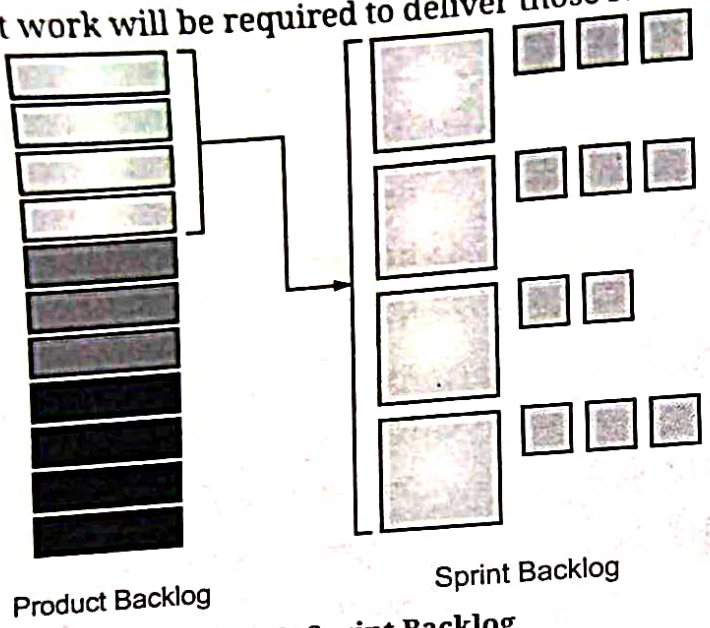



**Fig. 4.9: Sprint Backlog**

- In short, the sprint backlog is the short-term plan for the team's sprint. The product backlog is the long-term plan for the product, where the vision is itemized into concrete deliverable items that make the product more valuable. Many think of the sprint backlog as a subset of the product one. Ideally, this is true; the sprint backlog consists solely of items from the product backlog. In practice, however, a sprint backlog will include other work the team has committed to.

## 4.4.4 Sprint Velocity

- At the end of each iteration, the team adds up effort estimates associated with user stories that were completed during that iteration. This total is called velocity.
- Knowing velocity, the team can compute (or revise) an estimate of how long the project will take to complete, based on the estimates associated with remaining user stories and assuming that velocity over the remaining iterations will remain approximately the same. This is generally an accurate prediction, even though rarely a precise one.
- In order to estimate what work can be completed in the future, you need to measure the work that has previously been done. To get a good average measurement of work that has been done, plan to review the previous sprints.
- In the following example, we will use story points to measure the amount of work completed in each sprint. A story point is a measurement used by Agile development teams to estimate how much effort and time it will take to complete a user story.

**Example 1:**

**Step 1:** Count how many user story points are completed in each sprint. At the end of a sprint, add up how many story points the team completed.

Step 2: Calculate the average of completed story points.

**Table 4.1: The amount of work completed in each sprint**

| ID | Description | Sprint 1 | Sprint2 | Sprint 3 |
|---|---|---|---|---|
| A | Total User Stories | 5 | 7 | 9 |
| B | Value of Story points | 8 | 8 | 8 |
| C | Total story points need to obtain | 40 | 58 | 72 |
| D | Completed User stories | 3 | 4 | 5 |
| E | Total of story points completed (B*D) | 24 | 32 | 40 |
| F | Total completed | 24 + 32 + 40 = 96 | | |
| G | Average Sprint velocity | 96/3 = 32 | | |

- You can now base the amount of work to be done in future sprints on the average of 32 story points. If you have 160 story points remaining to be completed in the project, you can assume that your team will need another five sprints to complete the project.
- However, this is just an estimate and is accurate only if variables such as team size and project complexity and scope remain constant. Your teams will experience fluctuations from sprint to sprint. But the sprint velocity estimation is a good starting point to help you determine how much work your team can do.
- If your team is new to Agile development, you will not have any previous sprints to look at. As part of your sprint velocity estimation to-do list, you will have to complete a couple of sprints while tracking how many story points are completed in each. Then you will have some useful data that will help calculate an average.

## 4.4.5 Product Vision and Product Roadmap

- The main objective of product management is the development of a new product or products. This product should be better than what is currently available, or at least be able to differentiate itself as unique, in order to be of value for the customer. Product planning is all of the planning and strategy that goes into a product, from market research and design all the way through the product launch. During Product planning, we consider Product Vision, Strategy and Roadmap as shown in following Fig. 4.10.


**Fig. 4.10: Product Strategy Pyramid**

1. **Product Vision:**

- Vision is about the future and a hopefully better world. It is the essence of what you hope to achieve and forms the beginning of your strategy. Vision is not a statement that you define once and then forget about. It should be revisited at least once a year. And it should not be overly complex or difficult to parse — everyone in the company needs to know and deeply understand it.
- Once you have your new product idea, the next step is to develop a high-level vision for the product that can be used to pitch it to potential consumers. This high-level vision should outline the following things.
  - What your new product does?
  - What problem your new product solves?
  - Who specifically could benefit from using your product?
  - Why your new product is better than alternative solutions?
- While a product vision describes the essence of your product, it should be more than just that. A good product vision will be aspirational. It should motivate your teams to see this not merely as another job but a mission that fulfills an important need.

2. **Product Strategy:**

- If your product vision is the long-term outcome you hope to deliver with your product, then your product strategy defines how you'll get there. Your strategy is also the link between your roadmap and product vision—the strategy turns the vision into actions you can take to achieve those long-term goals.
- In general, you need to have a few basic elements for good product strategy planning. It includes:
  - **Market and Needs:** Understanding the target audience and the problems they face that you wish to solve in a way your competitors have not done before.
  - **Key Features and Differentiators:** What your competitive differentiation is and how you will deliver that value to customers.
- Your product strategy needs to be flexible and adaptable. Understand that team will always be learning about the users and the product as it evolves and makes its way down the lifecycle.
- Something you can do to make sure you have a solid product strategy is to ask yourself, and your team, the following product strategy questions. The answers should have been the foundation for your strategy back when you first built it from scratch, but they can be used to calibrate it:
  - What are the emotional reactions you are hoping to get from users every time you release a feature?
  - What unique purpose will your product fulfil in the market?
  - What are the valuable aspects need to build into the product?
  - What resources (time, effort, money) do the company have to achieve that value?
  - What are the limitations and possible snags your product might face in the market?

## The Product Strategy Canvas:

The Product Strategy Canvas is a step-by-step guide designed to help product managers define their product strategy. By filling in the different questions and sections of a product strategy canvas, you can be confident that you are on the right path: defining the things that matter the most to product success.

<table>
<tr><td colspan="2">Product Vision<br>Combination of the motivation for creating the product and the key functionalities that make the product stand out.</td><td colspan="2">• Product Name<br>The product name</td></tr>
<tr><td rowspan="2">Personas<br>Create provisional personas of the Target Group users by:<br>– Research<br>– Input from experts<br>– Input from users (interviews)</td><td colspan="2">• User Journeys<br>Create customer journeys, storyboards and user flows to define how the users will realize their tasks.</td><td>• Ready Stories<br>Create a list of ready user stories that cover at least the first sprint.</td></tr>
<tr><td>• Constraints<br>Describe the most important boundaries of the implementation</td><td>• Designs<br>Minimal design of the application; wireframes, sitemap, guidelines etc.</td><td>• Epics<br>Basically big user stories that have yet to be divided into user stories.</td></tr>
</table>

**Fig. 4.11: The Product Strategy Canvas**

## Product Roadmap:

- A roadmap is a visualization of your strategic plan. It captures activities you will complete within a given time frame. It communicates upcoming work in one view. You can use a roadmap to drive conversations. It can be your guide for prioritizing work, allocating resources, and tracking dependencies.
- A roadmap is not static. You can make adjustments as plans change, show progress as you complete work, and create tailored views for different audiences. A roadmap should excite. It is a visual guide that defines the work that is required for the team to be its best.
- A product roadmap is essential to product management because it allows the product manager, and anyone involved in the project, to see the big picture.
- A product roadmap is a bird's-eyeview of the vision and direction of a product over time. What a product roadmap captures is the vision and strategy of the product and it serves as a guide for executing that strategy. It's all laid out in such a way for it to be easily understood by everyone in the product group.
- The product roadmap is the responsibility of the product manager, both to create and to use as a communication tool. This sets the proper expectations, while sharing the product overview and highlighting important parts to come.
- The great importance of the product roadmap is that it speaks to everyone from stakeholders to team members on a high-level. Therefore, it must be clear, easy to understand and communicate critical details.

- Because a product roadmap must be clear and simple, it can't be cluttered with too much detail that will distract from the overall strategy and implementation. Therefore, a product roadmap is not a product plan. It might share elements of a product plan, but not at such a granular level.

**Key Elements of a Product Roadmap:**

- **Measurable Goals and Objectives:** Goals are the high-level statements that give a larger context to the product in terms of what it's trying to achieve. Objectives are more specific and result in tangible deliverables.
- **Collaboration and Communication:** There are likely many teams that are working together to create the product. Therefore, the product roadmap must facilitate clear communication of the overall strategy of the product and keep everyone on the same page.
- **Align with Corporate Objectives:** The high-level view of a product roadmap is designed to dovetail with the overall objectives of the corporation to make sure the product is aligned with where the company is going.
- **Leave Time to Learn and Research:** This graphic view of the schedule, which is laid out from left to right, must also include whatever research and learning curve is needed to make sure everyone on the product team is knowledgeable about what they're doing and how they're doing it.

**Why Use a Product Roadmap?**

- A product roadmap is a great tool to make sure that everyone knows what the product is about, how it will be executed and who is doing what. Those are just a few of the benefits, here are more.
  - **Fast Communication:** The product roadmap is a great way to quickly communicate the product strategy and goals at a high level and with a visual clarity that makes it easy to understand for everyone. This includes managing the expectations of stakeholders as well as communicating with all other important parties.
  - **Keeps it grounded:** As constraints and variables occur, the product roadmap keeps your decision-making tethered to the goals and objectives of the product, so you can prioritize tasks and make decisions quickly and accurately.

**Tools for Making a Product Roadmap:**

- There are many tools that can help one create a product roadmap, from the simple to the more dynamic. Because the product roadmap is a visual tool, though, it is not something that can be slapped together on a Word document or even a spreadsheet.
- A **Gantt chart** is useful to display all the milestones of the product. The tasks will be points on the timeline with bars to indicate the duration of the tasks. If any tasks are dependent on one another, they can be linked. All of which offers a graphic and clear overview of the product.

It is easy to build a product roadmap on our tool by following these steps:

**Step 1: List Milestones**

- Milestones are important dates, like deadlines or when certain documents or requirements are due. They typically reflect the end of one product phase and the start of the next one.
- Set milestones on the Gantt roadmap and they're indicated on the visual timeline by diamond-shaped icons. This is the first step to breaking up your product into feasible bits.

**Step 2: Link Dependencies**

- Dependent tasks are those that cannot start or end until another has started or ended. Identifying these avoids bottlenecks later on in your product execution phase.
- Connect dependent tasks on the Gantt roadmap by simply dragging one to the other. This can be done for all four types of dependencies and is indicated on the timeline by a dotted line.

**Step 3: Update Progress**

- Keeping your product on track and within budget requires monitoring and tracking its progress and performance so you can make necessary adjustments.
- View high-level progress with our real-time dashboard, which captures metrics such as time, tasks, costs and more, automatically does the calculations and then displays them.

**Step 4: Update Stakeholders**

- Keeping your stakeholders updated on how the product is proceeding and that it's staying on target as defined by the product plan is essential.
- Know product details by using one-click reports on progress, costs and more. They can be filtered to show just what you need to see and then easily shared with stakeholders.

## 4.4.6 Swim Lanes

- A Swim Lane diagram is a **process flowchart** that allows you to visually distinguish duties and responsibilities, as well as sub-processes within these business processes. The swim lane diagram first surfaced in the 1960s.
- Like any other flowchart, it visualizes a process from beginning to end, using the metaphorical lanes of an actual swimming pool to place the steps of mapping the lanes either vertically or horizontally.
- A swim lane is typically used for projects that extend over various departments and distinguishes channels according to a specific set of objectives. By organizing the responsibilities in various directions, it can clearly distinguish the objective of each department and individuals inside the team.

**Symbols:**

- The symbols you use in your swim lane diagram will be the same you'd use in a regular flowchart.

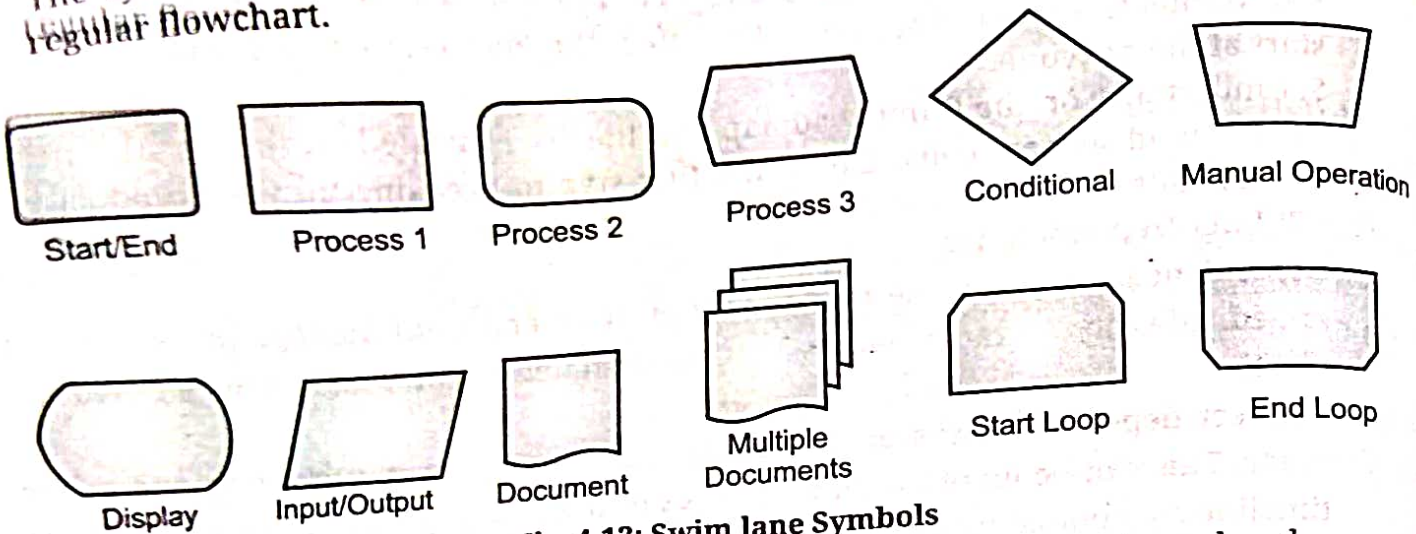



**Fig. 4.12: Swim lane Symbols**

1. **Start/End:** This shows where your flow begins or ends. Within the shape, you should include the words "Start" or "End" to clarify which it is.
2. **Process 1:** This signifies any process, action, or operation. They define an action, such as "write article", "create schedule" or "water plants".
3. **Process 2:** Another shape used for processes is the rounded rectangle. They're usually used to represent automatic events that trigger a subsequent action, such as "receive feedback".
4. **Process 3:** This represents a setup to another step in the process.
5. **Conditional:** Also known as the "decision shape", this symbolizes a question. The answer to that question determines which arrow to follow coming out of the diamond. Arrows should be labeled to avoid confusion.
6. **Manual Operation:** This represents an action where a user is prompted for information that must be manually inputted into a system.
7. **Display:** This symbol indicates a step that displays information.
8. **Input/Output:** Also referred to as the "data" object, this symbol refers to any information the goes into or comes out of your flow.
9. **Document:** This shape represents any document or report that takes part in the process flow.
10. **Multiple Documents:** This shape simply clarifies that there are multiple documents involved.
11. **Start Loop:** This shape represents the beginning of a loop.
12. **End Loop:** This shape indicates the point at which a loop should finish.

**Example:**

- Swim Lane diagrams are generally used in multi-departmental organizations for illustrating cooperative business models between the departments by displaying the departments in a vertical lane and objectives in a horizontal direction, or vice versa.

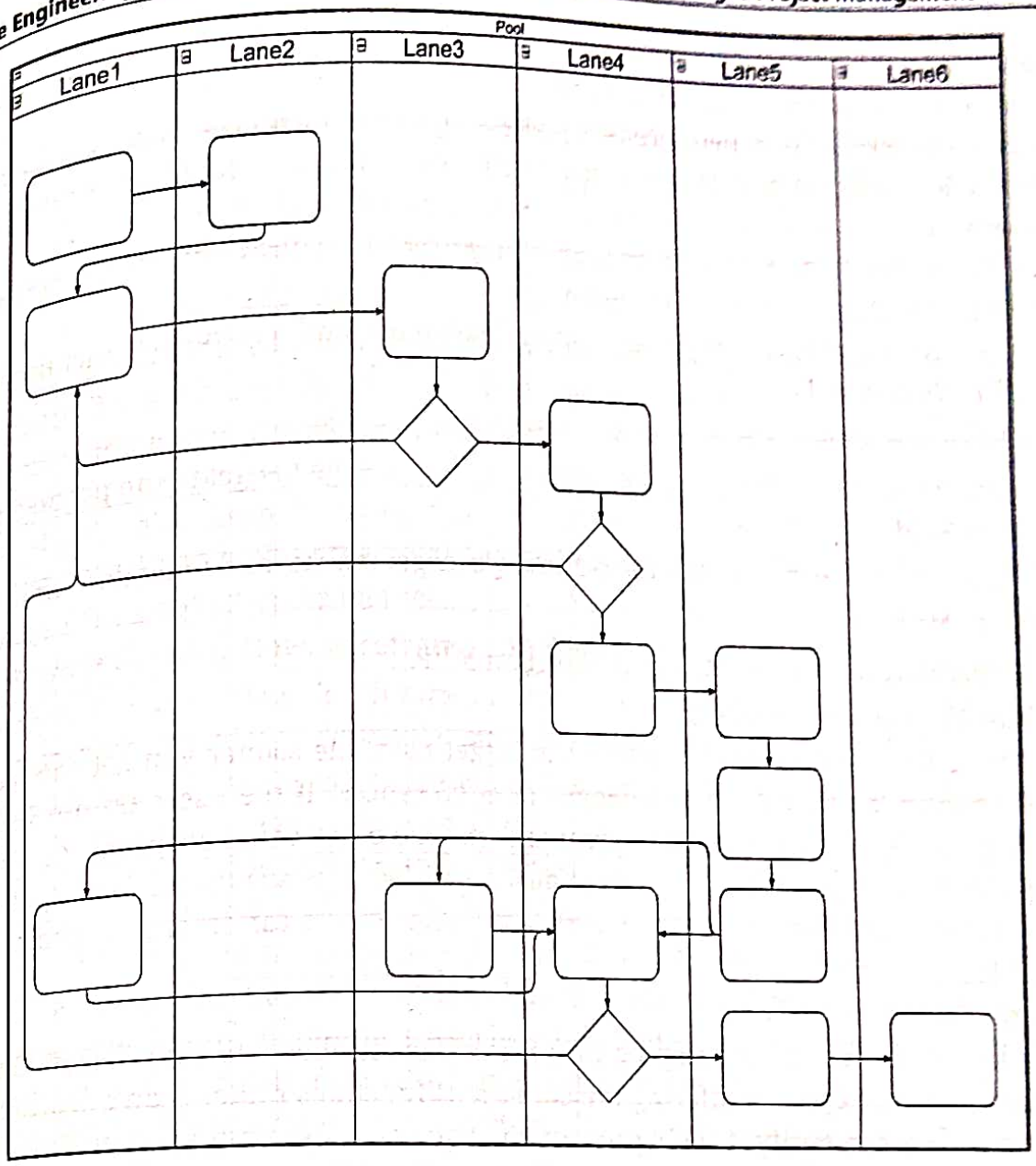



**Fig. 4.13: Swim Lane Diagram**

- Essentially, each department or team is represented by its own lane. You can use as many lanes as needed to illustrate your objective. You can even create loops in diagrams, or multiple "if ... then", or "or" lanes in case of different outcomes of the proposed objective.
- Swim lane diagrams also make it easier for departments to work with each other, as they not only identify bottlenecks but also objectives. Moreover, swim lane diagrams are able to distinguish the teams' limits when it comes to abilities and capacity. It allows departments to know what their counterparts are doing which in turn helps avoid collision and repetition of work by multiple figures.

## 4.4.7 Minimum Viable Product (MVP)

- Minimum Viable Product or MVP is a development technique in which a new product is introduced in the market with basic features, but enough to get the attention of the

consumers. The final product is released in the market only after getting sufficient feedback from the product's initial users.

- It is a 1.0 version of a new product possessing enough features to satisfy customers and collect maximum data about their opinions in order to continue improving the product.
- The main key factors of MVP are "minimum set of features", "customer feedback", "minimum effort" and "early prototype".
- A company might choose to develop and release a minimum viable product because its product team wants to:
  - Release a product to the market as quickly as possible.
  - Test an idea with real users before committing a large budget to the product's full development.
  - Learn what resonates with the company's target market and what does not.

**Benefits of MVP :**

- The following are the benefits of building an MVP:

1. **Initial Consumer Research:**
   As faster the product reaches out to the target user, the sooner you get the feedback and analyze the customer's challenges or preferences. If the users do not find your MVP valuable, you have space to pivot and test the other value propositions. Or, if the opposite is true, you will be sure that developed features are useful for target clients, so you can move forward. In the worst-case scenario, you can freeze the project to cut your losses.
2. **Testing Stage:**
   The biggest benefit of developing an MVP is that it allows testing various business models and concepts. By offering the core set of functions rather than a feature-heavy product, you can verify if their product concept resonates with your business model, providing an opportunity to change a product's direction based on findings.
3. **Cost Efficiency:**
   The quality products are the result of years of development, with an appropriate price tag. But because these products were created iteratively over a longer period, the cost is spread over time.

## 4.4.7.1 The Process of Building the MVP (Agile)

- There are 6 steps required for building an MVP.

**Step 1: Identify what problem you are solving and for whom**

- When developing a version of new products, most people are not sufficiently rigorous in defining the problems they're attempting to solve and articulating why those issues are important. Without this process, the product may miss opportunities and waste resources. That is why you need to become better at asking the right questions so that they tackle the right problems.

- Take a look at the product idea and ask yourself:
  1. Who's your target audience?
  2. Why does your target audience need this product?
  3. In what situation would they use it?
- In the process of identifying the answers to the questions above, it is important to keep everything real and do time and cost estimates in addition.

**Step 2: Analyze your competitors**

- As soon as you figured out what problem you are solving, it is time to see how other companies are solving this issue or at least trying to solve it.
- At this point, it is obvious that you have to conduct a competitor analysis if there are similar products on the market.
- You should also keep in mind that even if you don't think you have any direct competitors. Your faith in the uniqueness of your product will ground for confidently bringing the product to the market.

**Step 3: Define the user flow, wireframe and design**

- Defining the user flow for future product involves you focusing directly on the primary goal. In order to define the main user flow, we should first define the process stages which are actually easy because all you need to do is explain the steps required to reach the primary goal of your product.
- Here one should focus on basic tasks such as types of goals that end users have when they use your product, their expectations, etc.
- Once you have done with defining the user flow, you can move on to wire framing, which is simply an illustration of a webpage or an app. A wireframe is a layout that articulates what kind of interface elements will find a place on important pages.
- Designing the User Interface (UI) brings together concepts from interaction design, visual design, and information architecture.

**Step 4: Analyse features**

- Now that you have created user flow, you can start creating a more detailed list of features for each particular stage, but keeping the statistics in mind.
- Once you have laid out features for each stage, you will need to prioritize them. The main feature is the most important action that you want your users to complete.
- You can use one of the prioritization methods like MoSCoW which is used to decide which functions to complete first, which must come later and which to exclude.

**Step 5: Development & Testing**

- After approval of the Wireframes, you should start working on the setup architecture, database and start developing API, Administration, and all back-end.
- Alpha and Beta testing can help here as some of the most popular ways to test a product's performance in different scenarios. Make sure to align the testing and only make the changes that affect the entire user experience.

- As standard, you should implement the following tests in a controlled environment before the product is launched:
  - Functionality Testing
  - Usability Testing
  - Compatibility Testing
  - Crowd Testing
  - Interface Testing
  - Performance Testing
  - Security Testing

During the development process, you should continuously test all the implemented features.

**Step 6: Iteratively get to product-market fit or fail**

- If you validate the MVP, you can start looking at your product's scope and expand. At this point, you either raise start-up capital to help you get to market faster or fail.
- For a mobile app, we usually do a "soft launch" when we release the MVP to AppStore and Google Play but don't encourage any marketing of the app at this time.
- As the app is getting more users some bugs can pop and we are sure to fix everything ASAP. Usually, we release new builds every other day. Once the app is stable and we have a crash-free users rate over 99.9%, we recommend to start with marketing and acquire more users.
- The development of the product is never finished. The important thing is to ask for the user's feedback and iterate the product. When we have more users, we perform A / B testing (also known as Split testing or Bucket testing) to be able to test various solutions and increase user engagement.

## 4.4.8 Version and Release

- As a project progresses, many versions of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.
- The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents and unique objects like screen and report definitions, object files, test data and results.
- A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

## 4.4.8.1 Version Control

- Version control provides for unique identification of documents, whether electronic or hard copy, and assists with the easy identification of each subsequent version of a document. The version number changes as the document is revised allowing released versions of a document to be readily discernable from draft versions.
- Version control, also known as Source control. It is the practice of tracking and managing changes to software code.

**How to use Version Control?**

- Documents may be identified by a version number, starting at one and increasing by one for each release. Ideally, the version number should appear on the first page of the document and within the footer of each page. Documentation should be identified as follows:
  1. A release number (and a revision letters if in drafts).
  2. The original draft shall be Version 0.A subsequent drafts shall be Version 0.B, Version 0.C etc.
  3. The accepted and issued document is Version 1.0 subsequent changes in draft form become Version 1.0A, 1.0B etc.
  4. The accepted and issued second version becomes Version 1.1 or Version 2.0, as determined by the author based on the magnitude of the changes (minor or major).
- On this basis a document, which is presented to a Steering Committee for endorsement, would include a revision letter such as Version 0.A. After receiving endorsement, a copy of the document (with any approved amendments) would be created without a revision letter, in this case as Version 1.0. Wherever the version number appears in the document (i.e. front page, release notice, footer etc.) it should be updated.
- If controlled copies are to be distributed, the information in the 'Distribution List', 'Build Status' and 'Amendments In This Release' sections may need to be updated. Where there is provision for signatures, these should also be obtained prior to distribution. Including the corresponding version number in the file name (for example, Project XYZ Business Plan v0.A.doc) can uniquely identify electronic copies of documents.

## 4.4.8.2 Release

- A product release is the launch of a new product or a combination of features that will provide value to customers or users. A release is much more to both customers and internal teams. For customers, it is a promise of new value they can look forward to embedding in their everyday work and activities. For internal teams, a release helps them plan their work and when they'll be needed to launch great products.
- When we talk about releases, teams often confuse the roll-out of new features (a deployment) with the total customer experience. A release is not just the act of providing access to new technical functionality. Instead, think of a release as the date when the company is ready to deliver a new customer experience and support every customer interaction point associated with it.
- Releases should take into account all the additional work that must be accomplished, such as updating the public website and training the customer support team. While the supporting teams (like development) may define a release through their lens of

deploying code into production through a series of sprints, the product owner incorporates this perspective (and those of teams and customers) into a complete delivery.

**Release Management Process:**

- A release management process may likely mean different things to a product management team and a development team. Both perspectives are strongly incorporated into a dependable release management process.
- **A Release Management Process** incorporates all of the following:

**1. Planning:**

- An initial release plan takes into account the team's velocity on the previous release (or general capacity to deliver) and the feature prioritization to create a general scope, sequencing, and timeline for the release.
- During release planning, a general expectation on the number of sprints or iterations to deliver the scope is achieved. The accuracy of this expectation and plan depends on whether the team's capacity is well-known as well as the level of detail (or grooming) the scope has been through during estimation. This general plan will also provide expectations of major product changes (or dependencies) for products that may depend on your roadmap or platform.
- Plans will be revisited after each iteration. For this reason, a tracking of an external release target (with quarterly, monthly, or other precision) can be helpful. It will still set the expectation and trust with your customers, but can be refined by you as the plan progresses.

**2. Phased Communication and Supporting Team Engagement:**

- Product managers know best how to deliver their product successfully and that includes a series of non-development tasks of engagement with supporting teams to complete items like documentation, sales training, or marketing campaigns. As a product manager, establishing a launch or release template will enable you to create a "gold standard" for major delivery.
- Use this template to engage your greater team, who may be supporting multiple products in the portfolio only when needed. A standard for launch also sets an expectation for when these teams will be needed internally.

**3. Repeatable Stages to Readiness:**

- Establish standardized status at both the release and feature level to indicate overall health of the plan. Status provides an "Are we good?" pulse point that can be key to seeing around corners and proactive risk mitigation. A release status will enable communication to your internal stakeholders, while feature status workflows enable granular visibility into the readiness of the feature and its current status with respect to development, staging, or QA environments.

**4. Forward Adjustment on Plan:**

- Regardless of whether the release plan is executed in sprints or via more waterfall methodologies, regular check-ins and adjustments to plan are necessary. Use sprint closure to adjust plans as needed, or schedule regular reviews to ensure plans are on track.

# 4.5 AGILE PROJECT MANAGEMENT VS TRADITIONAL PROJECT MANAGEMENT

- The traditional Project Management (waterfall) approach is linear where all the phases of a process occur in sequence. Its concept depends on predictable tools and experience. Each and every project follows the same life cycle which includes the stages such as Feasibility, Planning, Designing, Building, Testing, Production, and Support, as shown in the Figure 4.14.
- The entire project is planned upfront without any scope for changing requirements. This approach assumes that time and cost are variables and requirements are fixed. The rigidity of this method is the reason why it is not meant for large projects and leaves no scope for changing the requirements once the project development starts.

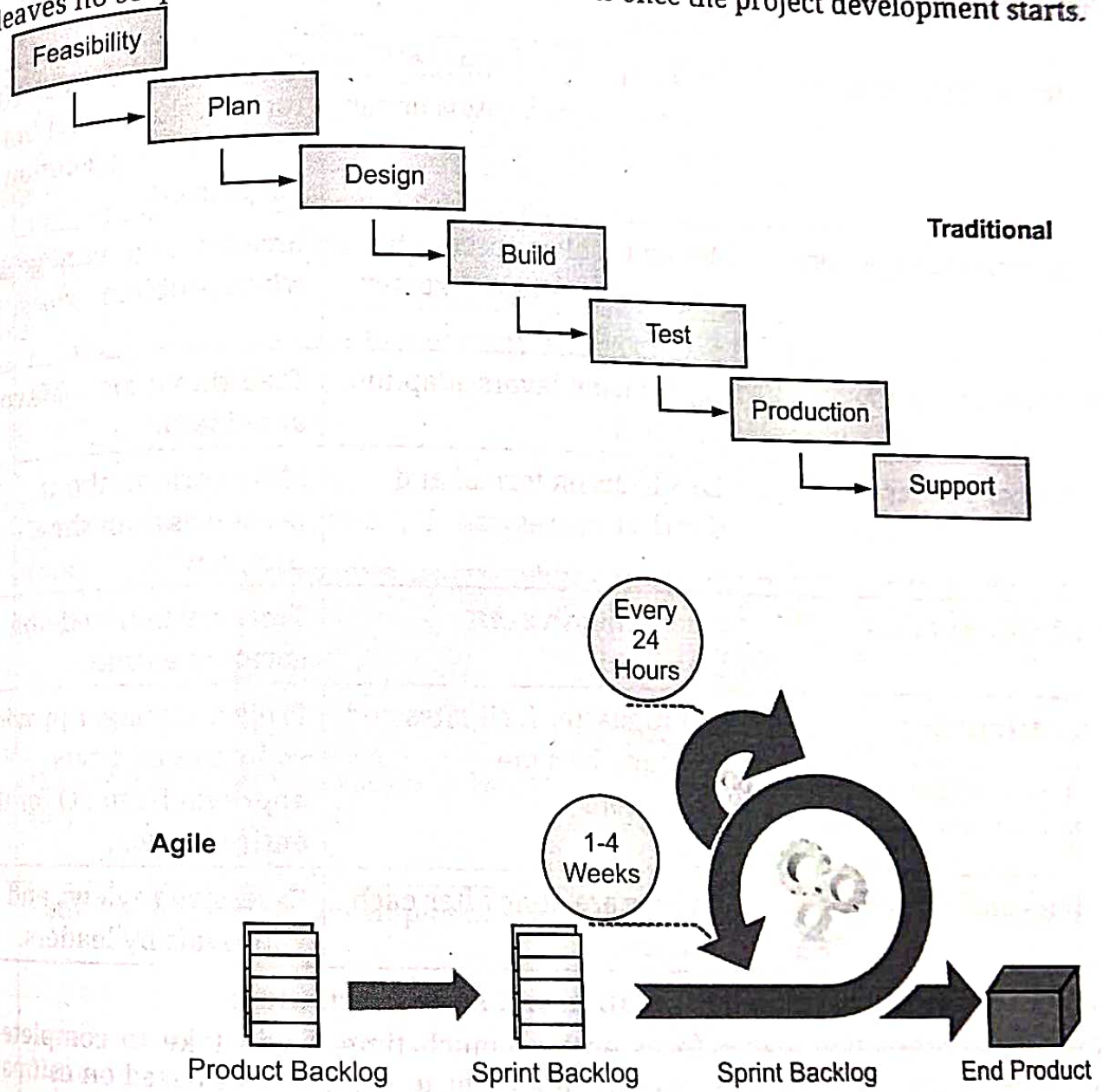



**Fig. 4.14: Traditional Project Management Vs Agile Project Management**

**Table 4.2: Difference between Traditional Project Management and Agile Project Management**

| Characteristic | Agile Project Management | Traditional Project Management |
|---|---|---|
| **Organizational Structure** | Iterative | Linear |
| **Scale of projects** | Small and medium scale | Large-scale |
| **User requirements** | Interactive input | Clearly defined before coding or implementation. |
| **Involvement of clients** | High | Low |
| **Development Model** | Evolutionary delivery Model. | Life cycle Model. |
| **Customer involvement** | Customers are involved from the time work is being performed. | Customers get involved early in the project but not once the execution has started. |
| **Escalation management** | When problems occur, the entire team works together to resolve it. | Escalation to managers when problem arises. |
| **Model preference** | Agile model favors adaption. | Traditional model favors anticipation. |
| **Product or process** | Less focus on formal and directive processes. | More serious about processes than the product. |
| **Test documentation** | Comprehensive test planning. | Tests are planned one sprint at a time. |
| **Effort estimation** | Scrum master facilitates and the team does the estimation. | Project manager provides estimates and gets approval from PO for the entire project. |
| **Reviews and approvals** | Reviews are done after each iteration. | Excessive reviews and approvals by leaders. |

**Difference between Traditional Estimation and Agile Estimation:**

- Traditional estimation always focus on how much time it will take to complete th work hence the scope is always constant and Time and cost varies based on estimatio where in Agile we focus on what are the functionality we can complete with in th fixed time box. This concept explained in a below diagram.

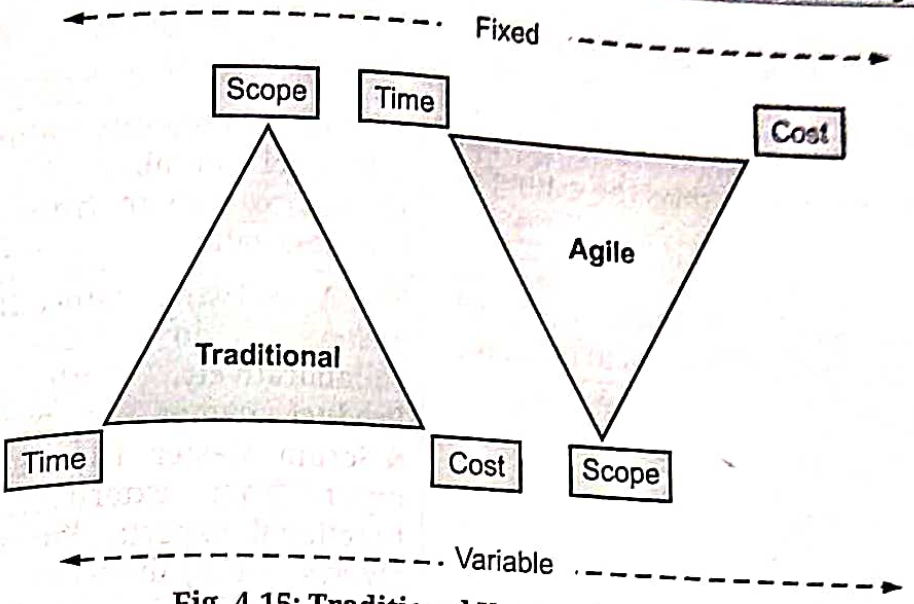



**Fig. 4.15: Traditional Vs Agile Estimation**

**Table 4.3: Difference between Traditional Estimation and Agile Estimation**

| Sr. No. | Traditional Estimation | Agile Estimation |
|---|---|---|
| 1. | Primarily estimation on tasks takes place to get the timeline of the project. | Estimate the size of story for its value and complexity, task estimation is secondary. |
| 2. | Estimate to get the time line to complete the entire product. | Estimate to plan how much business value we can deliver in a single sprint or iteration duration (typically 2 to 3 weeks). |
| 3. | We estimate development, Testing and other effort separately for any functionality. | Estimate the size of a story keeping in mind, its development, testing, or any other activity to complete the functionality. However if required, we can estimate different tasks for independent resource specific efforts. |
| 4. | We estimate absolute values in Hours or Days. | We estimate Relative sizing, learn more about relative sizing in our next section of understanding story point. |
| 5. | Estimate Before the development start | Estimate between the developments for upcoming Stories of future iteration, by applying lessons learned from past sprints. |
| 6. | Estimates to create Plan of schedule and the development is plan driven. | Estimates to Identify Value, and the development is value driven. |

| Sr. No. | Traditional Estimation | Agile Estimation |
|---|---|---|
| 7. | Estimates of absolute number of time have high chances to miss the estimate. | Estimates to Identify Value in a range in Fibonacci number, have very less chance to deviate from the estimated business value. |
| 8. | Panels of technical Experts, Architects and other member involves to estimate. | The Agile Team, mainly Developers and Testers do the estimation collaboratively, with presence of Product owner or business analyst & Scrum Master The Team may seeks input from external Technical or functional experts, but the estimation always own by the team. |

## 4.6 AGILE REPORT

- Reporting is the direct result of the inherent need to measure, digest, and understand key data for decision making. In Agile, that reporting part must be quick and easy to get, read and understand.
- Following reports are typically created at the end of each iteration:
  - The *Product Backlog Report*, a prioritized list of all the user stories that make up the entire product. It captures all desired functionality for a particular release of software.
  - The *Sprint Backlog Report*, which includes the user stories the team is committed to deliver in the next iteration (called a Sprint in Scrum).
  - The *Burndown report or chart*, which illustrates (usually in the form of a trend graph) the work your team has "burned through," giving organizations a real-world view of the team's progress.
  - *The Velocity Report*, which is a report of how much the team gets done in iteration, often measured in "story points" chosen by the team.

**1. Product Backlog Report:**

- The Product Backlog Report is an ordered list of all things that need to be done within a project. Items on the list in Scrum projects are usually user-centric and follow a standard user story format that replaces traditional requirements specifications.
- The most important items are shown at the top of the backlog report so the team knows what to deliver first. A customer representative (known as a product owner in Scrum) reviews (or "grooms") the backlog report before each iteration planning meeting to ensure prioritization is correct and feedback from the last iteration has been incorporated.
- The Product backlog report serves as the main communication device between the development team and the customer/product owner, who can re-prioritize work on the backlog at any time, as well as add or subtract requirements as business conditions change.

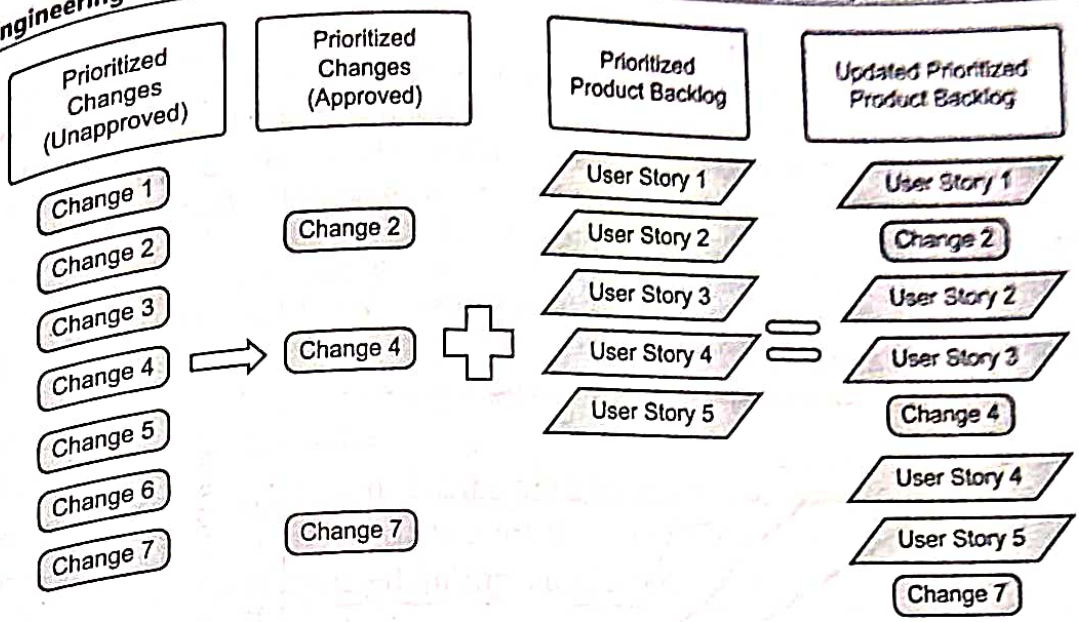



**Fig. 4.16: Updating or Re-prioritizing Process for the Product Backlog Report**

**2. Sprint Backlog Report:**

- The sprint backlog is a list of tasks identified by the Scrum team to be completed during the Scrum sprint. During the sprint planning meeting, the team selects some number of product backlog items, usually in the form of user stories, and identifies the tasks necessary to complete each user story. Most teams also estimate how many hours each task will take someone on the team to complete.
- The sprint backlog is often maintained using issue and project software program like JIRA, or via a group spreadsheet, or with sticky notes on a whiteboard (usually referred to a Kanban Board). An example of a Sprint backlog in a Kanban board looks like this:

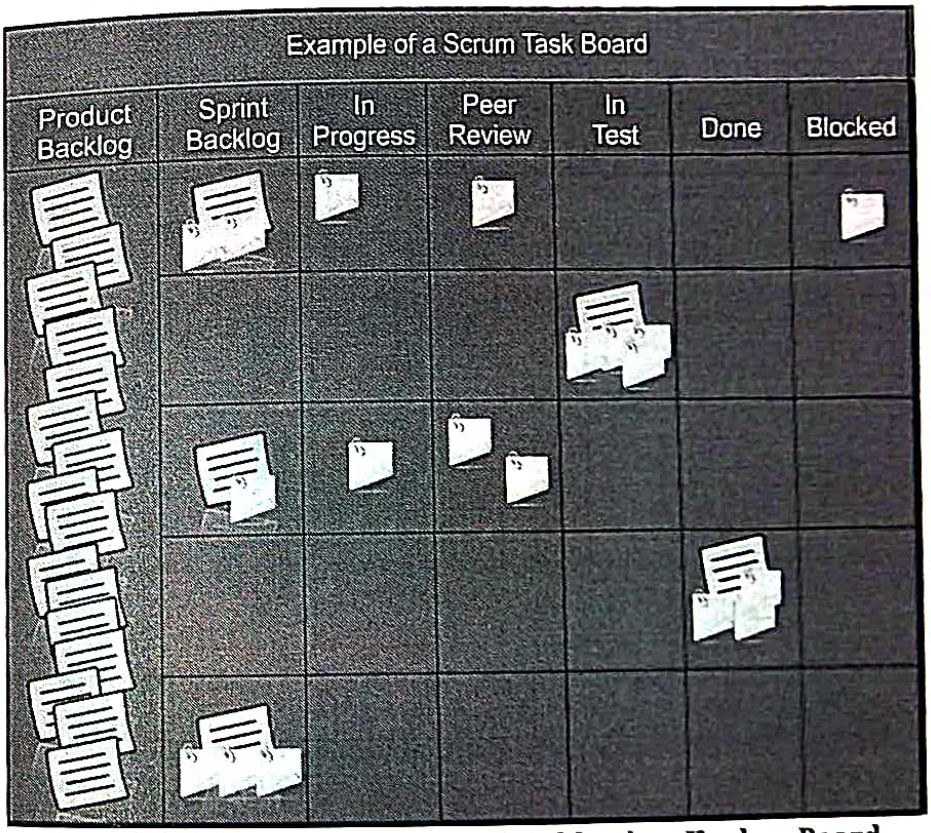



**Fig. 4.17: Example of a Sprint Backlog in a Kanban Board**

### 3. Burndown Report or Chart:

- A burndown chart is a plot of work remaining to reach a given goal on the vertical axis, and time on the horizontal axis. Each point on the chart shows how much work is left to do at the end of that day (or week, month or other time period).
- A burndown chart typically has a line that runs diagonally from the top left to the bottom right corner that represents the estimated rate of work or "burn" needed to reach the goal. If the line that shows the actual work done on the project is above the estimation line the project is behind schedule. If the actual line is below the estimation line the project is ahead of schedule.

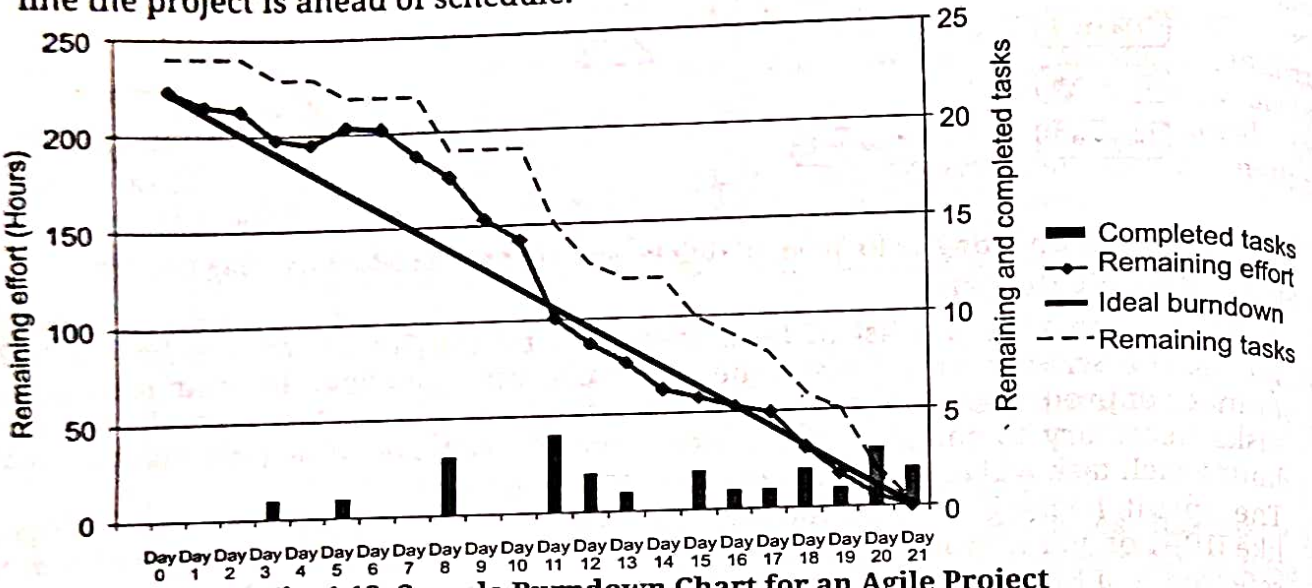



**Fig. 4.18: Sample Burndown Chart for an Agile Project**

### 4. Velocity Chart:

- The Velocity Chart shows the amount of value delivered in each sprint, which enable to predict the amount of work your team can get done in future sprints. It is useful during sprint planning meetings, to help make decision about how much work a particular team should commit to.

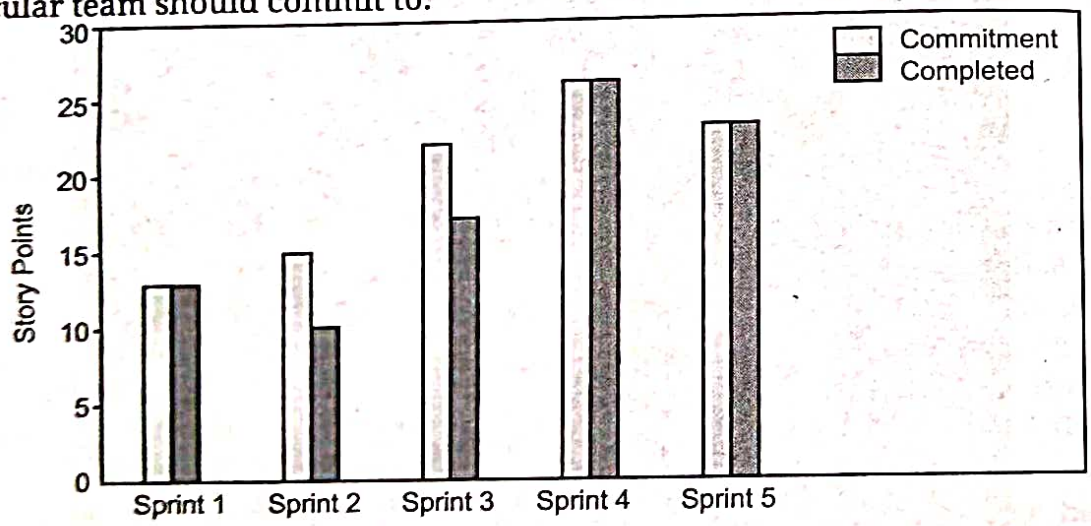



**Fig. 4.19: Velocity Chart**

- Velocity charts can show the sum of estimates of the work delivered across all iterations.

## 4.6.1 Daily Reports

- Each day at the same time, the team meets so as to bring everyone up to date on the information that is vital for coordination: Each team members briefly describes any "completed" contributions and any obstacles that stand in their way.
- The meeting is normally held in front of the task board. In its most basic form, a task board can be drawn on a whiteboard or even a section of wall. The board is divided into three columns labelled "To Do", "In Progress" and "Done". Sticky notes or index cards, one for each task the team is working on, are placed in the columns reflecting the current status of the tasks.
- Different layouts can be used, for instance by rows instead of columns (although the latter is much more common). The number and headings of the columns can vary, further columns are often used for instance to represent an activity, such as "In Test".
- The task board is updated frequently, most commonly during the daily meeting, based on the team's progress since the last update. The board is commonly "reset" at the beginning of each iteration to reflect the iteration plan.
- This is also called "daily stand-up" in Extreme Programming, and "daily scrum" in Scrum framework.
- The daily meeting is structured around some variant of the following three questions:
  1. What have you completed since the last meeting?
  2. What do you plan to complete by the next meeting?
  3. What is getting in your way?
- From these three points, the following can be understood:
  1. The progress of individual parts and how they fit into the overall picture. In other words, you are able to check on the accomplishment of goals your team committed to the previous day. So you can learn if team members are setting themselves achievable goals or perhaps, if they are not stretching themselves enough and thus slowing down progress. Moreover, you are able to quantify the development of a project.
  2. What you are able to forecast for the coming 24 hours. Namely, what targets are team members setting for themselves? By knowing this, you can comprehend how much further the project is likely to develop over the next day based on the predictions of the team members in relation to their past achievements.
  3. Where individual team players have blockers. That is to say, a list of the obstacles faced and ideas on how to solve them. Although blockers should not be dwelled upon and discussed in detail within the Daily Scrum, by highlighting them it allows the Scrum Master in particular the opportunity to delegate and assign problem solving exercises to others.

## 4.6.2 Sprint Burn Down Chart and Reports

- **A burn down chart** is a graphical representation of work left to do versus time. The outstanding work (or backlog) is often on the vertical axis, with time along the horizontal. Burn down charts is a run chart of outstanding work.
- It is useful for predicting when all of the work will be completed. It is often used in agile software development methodologies such as Scrum. However, burn down charts can be applied to any project containing measurable progress over time. Outstanding work can be represented in terms of either time or story points.

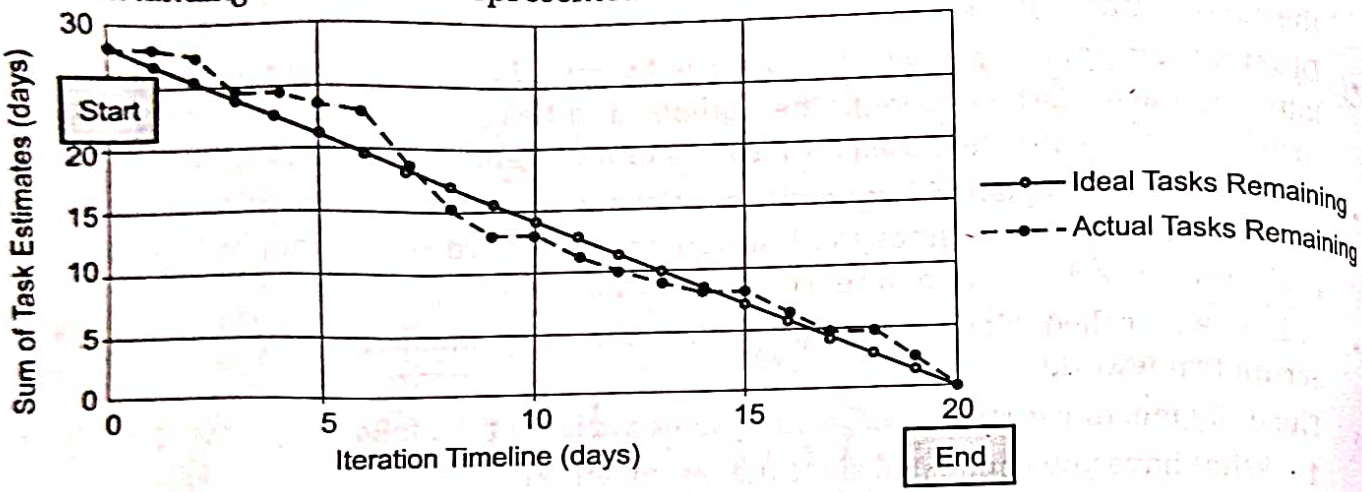



**Fig. 4.20: Project XYZ Iteration 1 Burn Down Chart**

- A burn down chart for a completed iteration is shown above and can be read by knowing the following:

**Table 4.4: Burn Down Chart Elements and its Description**

| Element | Description |
|---|---|
| X-Axis | The project/iteration timeline. |
| Y-Axis | The work that needs to be completed for the project. The time or story point estimates for the work remaining will be represented by this axis. |
| Project Start Point | This is the farthest point to the left of the chart and occurs at day 0 of the project/iteration. |
| Project End Point | This is the point that is farthest to the right of the chart and occurs on the predicted last day of the project/iteration. |
| Number of Workers and Efficiency Factor | In the above example, there is an estimated 28 days of work to be done, and there are two developers working on the project, who work at an efficiency of 70%. Therefore, the work should be completed in (28 ÷ 2) ÷ 0.7 = 20 days. |
| Ideal Work Remaining Line | • This is a straight line that connects the start point to the end point.<br>• At the start point, the ideal line shows the sum of the estimates for all the tasks (work) that needs to be completed. |

| Element | Description |
|---|---|
| | • At the end point, the ideal line intercepts the x-axis showing that there is no work left to be completed. Some people take issue with calling this an "ideal" line, as it's not generally true that the goal is to follow this line. This line is a mathematical calculation based on estimates, and the estimates are more likely to be in error than the work.<br>• The goal of a burn down chart is to display the progress toward completion and give an estimate on the likelihood of timely completion. |
| Actual Work Remaining Line | • This shows the actual work remaining.<br>• At the start point, the actual work remaining is the same as the ideal work remaining but as time progresses; the actual work line fluctuates above and below the ideal line depending on this disparity between estimates and how effective the team is.<br>• In general, a new point is added to this line each day of the project. Each day, the sum of the time or story point estimates for work that was recently completed is subtracted from the last point in the line to determine the next point. |

## 4.7 USER STORIES SCENARIOS AND WRITING USER STORIES

- In the world of product design and development, three terms often come up that can sometimes be confusing for us: user stories, user scenarios, and use cases.

**User Stories:**

User stories only capture the essential elements of a requirement:

- Who it is for? (Role)
- What it expects from the system? (Action)
- Why it is important (optional?)?(Benefit)

Here is a simple format of user story used by 70% of practitioners:

**Template:** As a <role> I can <action>, so that <receive benefit>

**Examples:**

- As a [customer], I want [shopping cart feature] so that [I can easily purchase items online].
- As an [manager], I want to [generate a report] so that [I can understand which departments need more resources].
- As a [customer], I want to [receive SMS when the item is arrived] so that [I can go to pick it up right away].
- As an [online gamer], I want to [have a multiplayer option] so that [I can play online with friends.]
- As a [design team lead], I want to [organize assets], so [I can keep track of multiple creative projects].

**User Scenarios:**

- User scenarios are descriptions in a narrative way that tells us the story of a user's interaction with a product. They provide more detailed context than user stories, including the user's motivations, actions, and the overall flow of interaction with the product.
- User scenarios help teams understand the user's journey and the different steps they might take to achieve their goal, which can inform design decisions to make that journey as smooth as possible.

**Use Case:**

- Use cases, a term originating from the field of systems engineering, are more detailed and technical than both user stories and user scenarios. A use case describes a sequence of actions that a system performs to achieve a specific outcome.
- Each use case is typically represented with a title, a primary actor (usually a user), a goal, preconditions, main success scenario (steps), extensions, and post conditions.

**Example 1:**

- **User story**: User A needs to SIGN IN to access the app. (Who, What, Why).
- **User stories Scenario:**
  - Let say that we encounter a mobile app sign-in. The user story will be: User A needs to SIGN IN to access the app. (Who, What, Why). Then, we developed the use case: First, we need to open the app, verify connectivity in the device, and then present the user the options that they have to accomplish the task. We see that we have multiple choices. We can sign-in using our Facebook account, Twitter account, or email. The use case is sign-in; each of the options to sign in is a use case scenario. All of the situations lead to the same outcome, but as users, we will encounter different interfaces depending on our choice of sign-in.
- **Use Case:**
  - A user story will give us more information regarding the motive and needs of the user; it will also give us a high-level goal vs. the use case. It will provide us with the details of how to accomplish the target and all the scenarios that the user can encounter while performing the task. The use case is more detailed and focuses more on functionality.

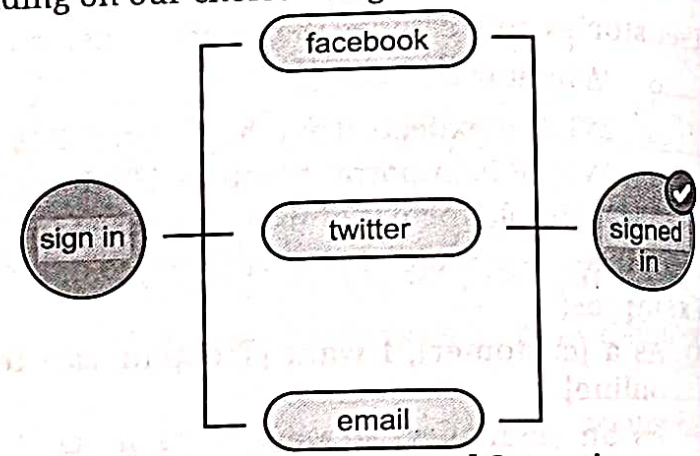



**Fig. 4.21: User Story and Scenarios**

**Examples 2:**

- **User Story:** A web designer wants to create a website so that bakery items can be sel (Who, What, Why).
- **User Stories Scenario:**

A web designer has been contracted to create a new website for a small local bakery He has a clear idea of how he wants the website to look. He starts a new project in hi design tool, names it "Angle Website Project", and sketches out the basic structure. He use

the color palette tool to apply the bakery's brand colors to his design. After he's happy with the design, he previews it on different devices, makes necessary adjustments, and finally, exports the design to present to his client.

- **Example of Use Case:**

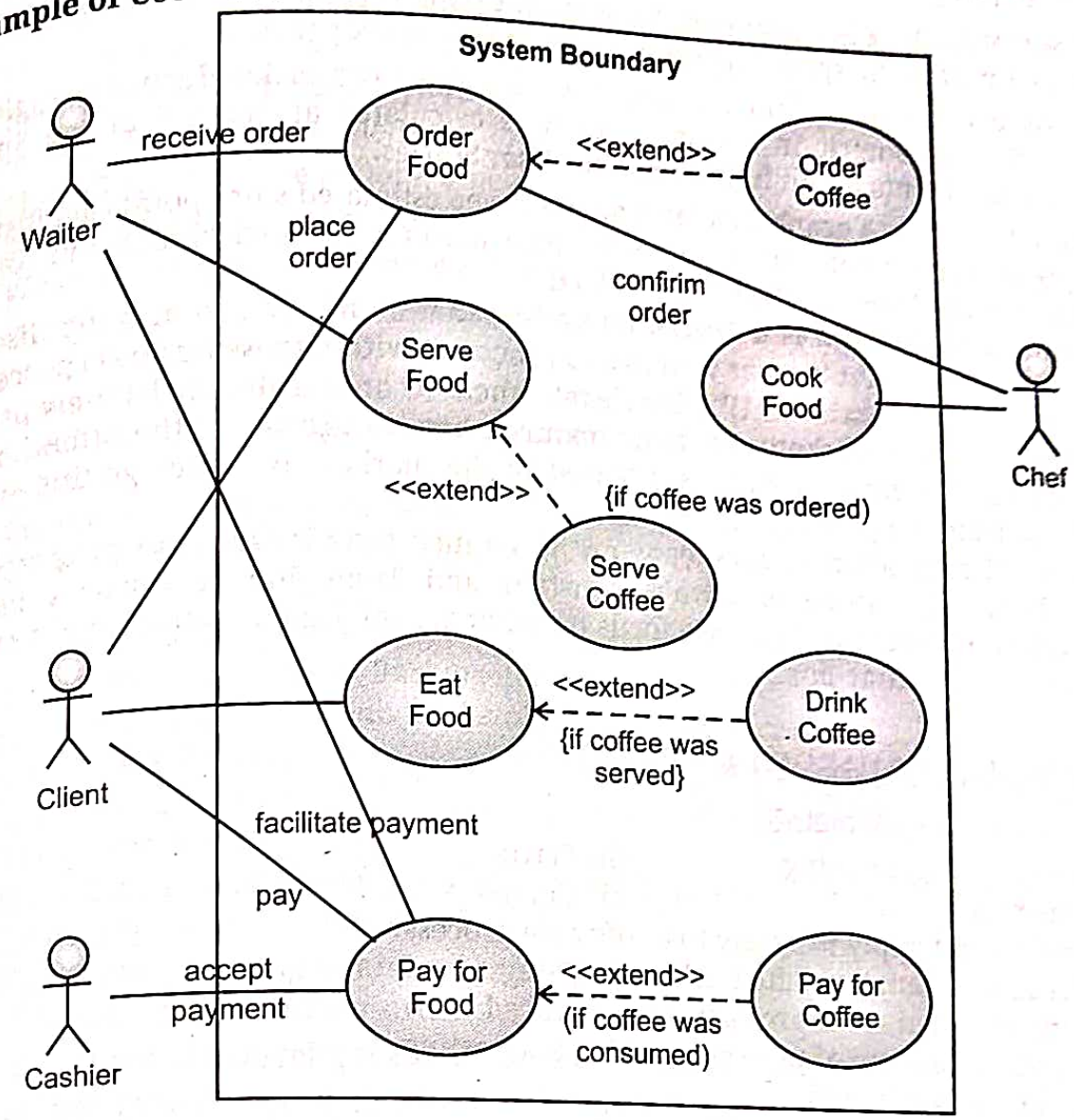



**Fig. 4.22: Use-case Diagram**

## SUMMARY

- Agile software development is based on an incremental, iterative approach.
- Agile software development lifecycle includes the concept, inception, construction, release, production, and retirement phases.
- The Agile Manifesto lists 12 principles to guide teams on how to execute with agility.
- A user story is a tool used in Agile software development to capture a description of a software feature from an end-user perspective.
- A story point is a metric used in agile project management and development to estimate the difficulty of implementing a given user story, which is an abstract measure of effort required to implement it.

- A backlog is a list of tasks required to support a larger strategic plan. In a product development context, it contains a prioritized list of items that the team has agreed to work on next.
- The Sprint Backlog is a container for work the team is committed to doing, either right now as a part of the sprint (typically a one- to four-week period).
- Velocity is a measure of the amount of work a Team can tackle during a single Sprint and is the key metric in Scrum. Velocity is calculated at the end of the Sprint by totalling the Points for all fully completed User Stories.
- A Velocity chart is a graph that lets you easily see estimated story points against actual, completed story points. Story points are measured on the vertical axis and completed sprints are displayed on the horizontal axis.
- A Swim Lane diagram is a Process Flowchart that allows you to visually distinguish duties and responsibilities, as well as sub-processes within these business processes.
- Minimum Viable Product (MVP) is a development technique in which a new product is introduced in the market with basic features, but enough to get the attention of the consumers. The final product is released in the market only after getting sufficient feedback from the product's initial users.
- Traditional estimation always focus on how much time it will take to complete the work hence the scope is always constant and Time and cost varies based on estimation. Whereas in Agile, we focus on what are the functionality we can complete with in the fixed time box.

## CHECK YOUR UNDERSTANDING

1. _____ is not an agile method.
   (a) Extreme Programming (b) Scrum
   (c) Waterfall (d) Crystal
2. Which does not apply to agility to a software process?
   (a) Uses incremental product delivery strategy.
   (b) Only essential work products are produced.
   (c) Eliminate the use of project planning and testing.
   (d) All of the mentioned
3. Agile Software Development is based on _____.
   (a) a call to remove Linear Development
   (b) Iterative Development
   (c) Incremental Development
   (d) Both Iterative and Incremental Development
4. Scrum has ____ phases.
   (a) Two
   (b) Three
   (c) Four
   (d) Scrum is an agile method which means it does not have phases.
5. Which of the following is delivered at the end of the Sprint?
   (a) A document containing test cases for the current sprint.
   (b) An architectural design of the solution.
   (c) An increment of done software.
   (d) Wireframes designs for User Interface.

6. Product Backlog should be ordered on the basis of _____.
   (a) Value of the items being delivered.
   (b) The complexity of the items being delivered.
   (c) Size of the items being delivered.
   (d) The risk associated with the items.
   (e) Based on the Scrum Team choice.
7. When can a Sprint be canceled?
   (a) The Sprint items are no longer needed.
   (b) Sprint can never be canceled.
   (c) When Development is unable to complete the work.
   (d) Information required to start the development is not available.
   (e) Whenever the Product Owner says.
8. What should a Development Team do during a Sprint Planning meeting when they have realized that they have selected more than the items they can complete in a Sprint?
   (a) Get more developers onboard.
   (b) Seek help from the other Scrum Team Members.
   (c) Work overtime.
   (d) Inform the Product Owner.
   (e) Take some of the Sprint Backlog items.
9. Who is responsible to measure the Project's performance?
   (a) The Scrum Master
   (b) The Delivery Manager
   (c) The Product Owner
   (d) The Development Team
   (e) The Scrum Team
10. What are the main responsibilities of a Scrum Master?
   (a) Removing Impediments.
   (b) Facilitating meeting as and when requested.
   (c) Help the Product Owner order the Product Backlog.
   (d) Consulting the Development Team and Product Owner.
   (e) Bridging the Gap between the Team and the Customer.
11. In Scrum, when is a Sprint Over?
   (a) When all the Sprint Backlog Items are completed.
   (b) When the Product Owner suggests.
   (c) When all the Sprint Backlog tasks are completed.
   (d) When the final testing is completed.
   (e) When the time box expires.
12. What is done during a Sprint Review Meeting?
   (a) Demo of the Increment.
   (b) The team discusses the improvements that can be applied for the upcoming sprints.
   (c) Present the Project's performance to the Stakeholders.
   (d) Inspect progress towards the Sprint Goal.
   (e) Discuss the architectural and technical aspects of the project.

13. What is a Sprint Review?
    (a) Activity to Introspect and Adapt.
    (b) Activity to improve Scrum Processes.
    (c) Activity to seek approval for the work done.
    (d) Activity to plan for the next Sprint.
    (e) Activity to plan for the release.
14. What happens when all the Sprint items cannot be completed?
    (a) The Sprint should be extended.
    (b) The Sprint ends with the done items.
    (c) The Sprint should be canceled.
    (d) The unfinished Sprint items should be removed from the Sprint Backlog.
    (e) Start the next Sprint with the unfinished items first.
15. The intersection of a trend line for work remaining (or backlog) and the horizontal axis indicating the most probable completion of work at the point in time would be found in which graphical chart?
    (a) Burn-up chart
    (b) Velocity graph
    (c) Burndown chart
    (d) Execution chart

**Answers**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1. (c) | 2. (c) | 3. (d) | 4. (b) | 5. (c) | 6. (a) | 7. (e) | 8. (d), (e) | 9. (c) |
| 10. (a), (b), (d) | 11. (e) | 12. (a), (c) | 13. (a) | 14. (b) | 15. (c) | | | |

## PRACTICE QUESTIONS

**Q.I Answer the following questions in short.**

1. What is Agile software development?
2. What is product backlog?
3. Explain the concept of version and release.
4. What is Minimum Viable Product (MVP)?
5. What is velocity chart?
6. Which reports are typically created at the end of each iteration?
7. What is sprint Backlog Report?

**Q.II Answer the following questions.**

1. Explain Agile system development Methodologies.
2. Explain 12 agile principles.
3. Explain 4 agile values.
4. What are User stories? Explain the role of user stories in Development.
5. How Sprint backlog is different from Product backlog?
6. How swim lanes are used in project development?
7. Give any 8 differences between Agile project management and Traditional project management.
8. Explain daily reports in detail.
9. Explain burndown chart report with diagram.

**Q.III Write a short note on:**

1. Swim lane
2. Story Point
3. Sprint Backlog
4. Version control
5. Agile Project Life Cycle
6. Product backlog Report
7. Burn down Chart
8. Velocity Report

***

UNIT 5

# Implementation with Agile Tools

**Objectives...**

After reading this chapter, you will be able:

- To learn about Agile tools.
- To know about GitHub tool.
- To study the implementation of Problem Statements with Agile Tools-GitHub.
- To learn how to Create Project using Kanban.

## 5.1 MS PROJECT TOOL

**MS Project:**

- Microsoft Project is a project management software program, developed and sold by Microsoft, which is designed to assist a project manager in developing a plan, assigning resources to tasks, tracking progress, managing the budget, and analyzing workloads.
- It is part of the Microsoft Office family, but never been included in any of the Office suites. It is available currently in two editions, Standard and Professional. Microsoft Project's proprietary file format is .mpp.

**MS Project Environment:**

- Open the Microsoft Project software on your computer, click on Start → Programs→ and look for MS Project software and you will see the following screen which nothing but the MS project environment.

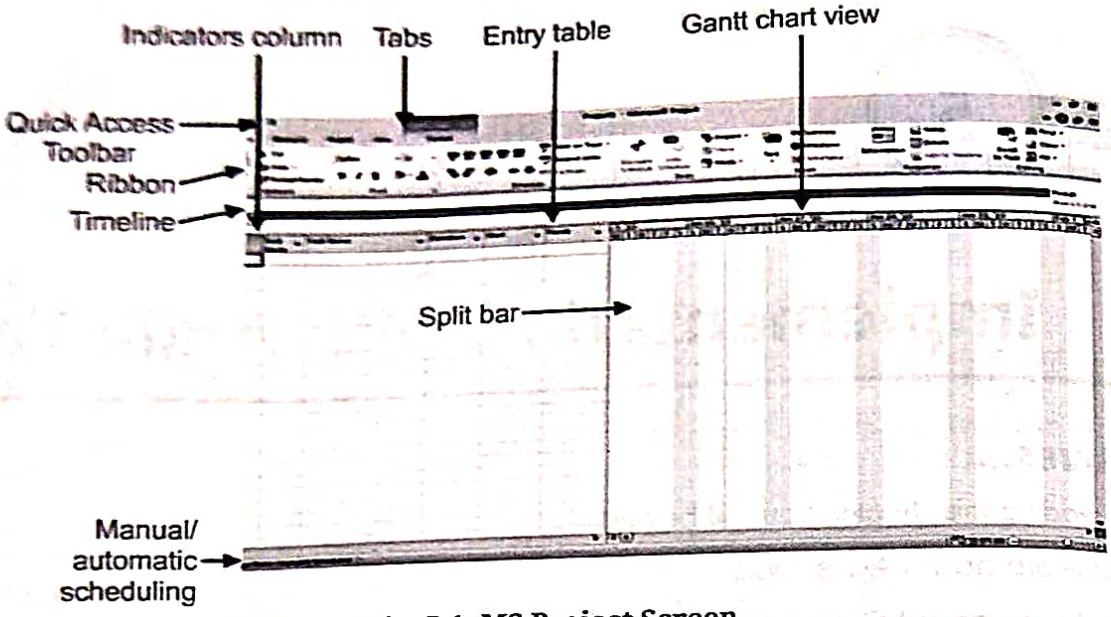



**Fig. 5.1: MS Project Screen**

**Create a Project Plan:**

- There are multiple ways to create a project plan : from a blank project, from a template, from existing project and from Excel.

**To create a project:**

1. From **File** tab, choose **New** option.
2. Select **Blank Project** and click **Create**.
3. The first required step in creating a project is to provide project information. To do this select **Project Information** from **Project** tab.
4. Set the desired date and click OK.
5. Additional project properties can be identified. From File tab, choose **Project Information** then **Advanced Properties**. Complete the fields as desired and click OK.

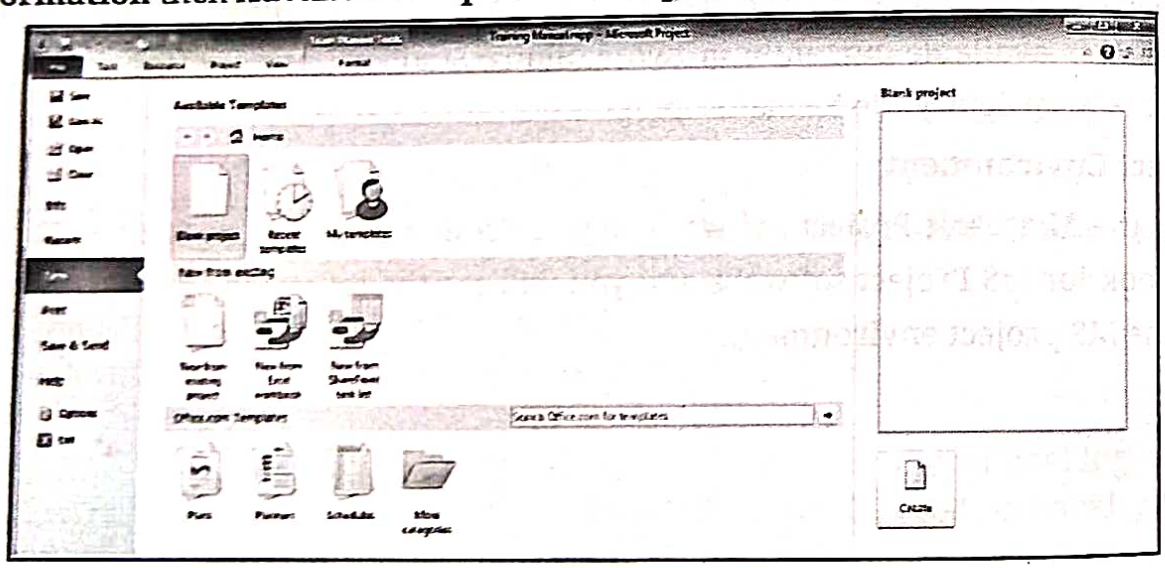



**Fig. 5.2 (a): Create new file**

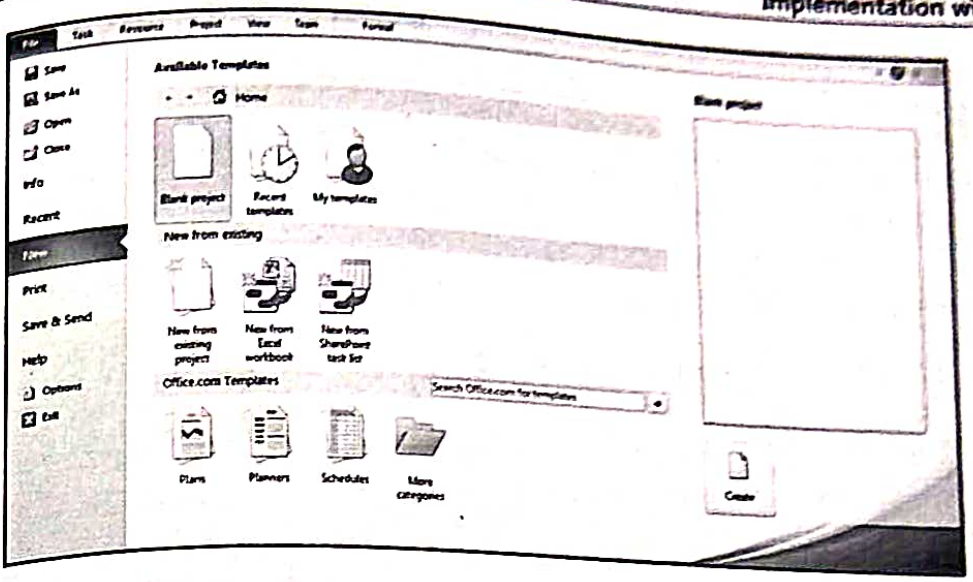



Fig. 5.2 (b): Create Blank Project

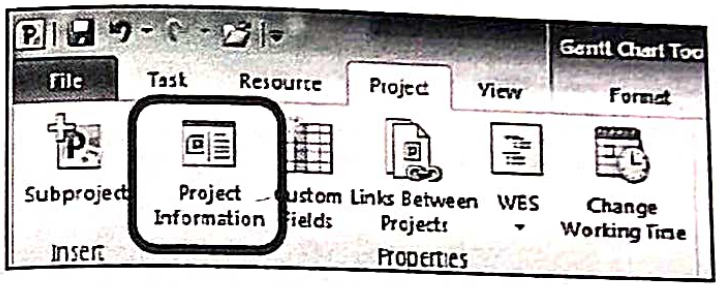



Fig. 5.2 (c): Select Project Information

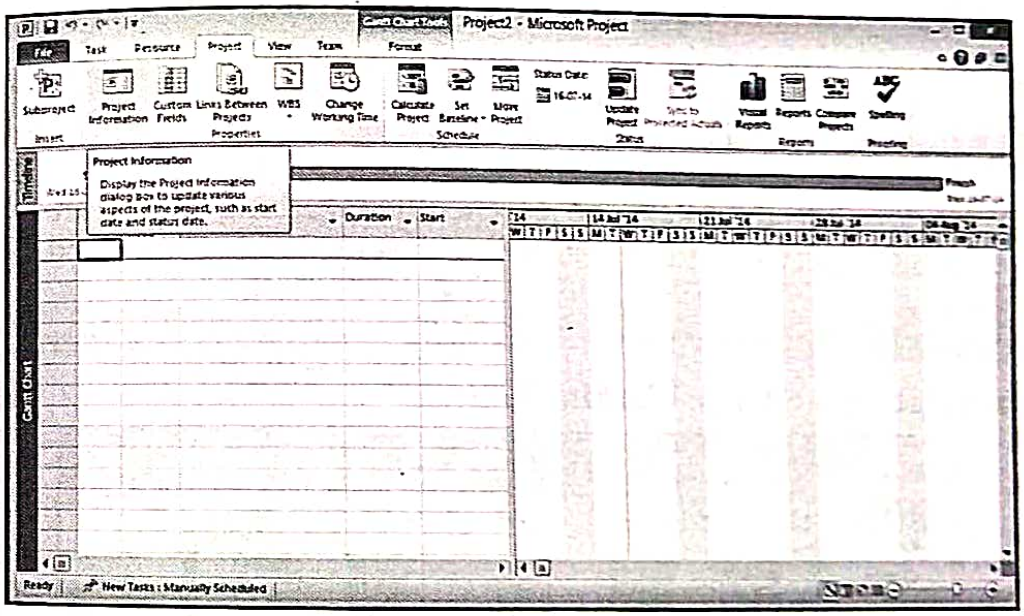



Fig. 5.2 (d): Enter Project information

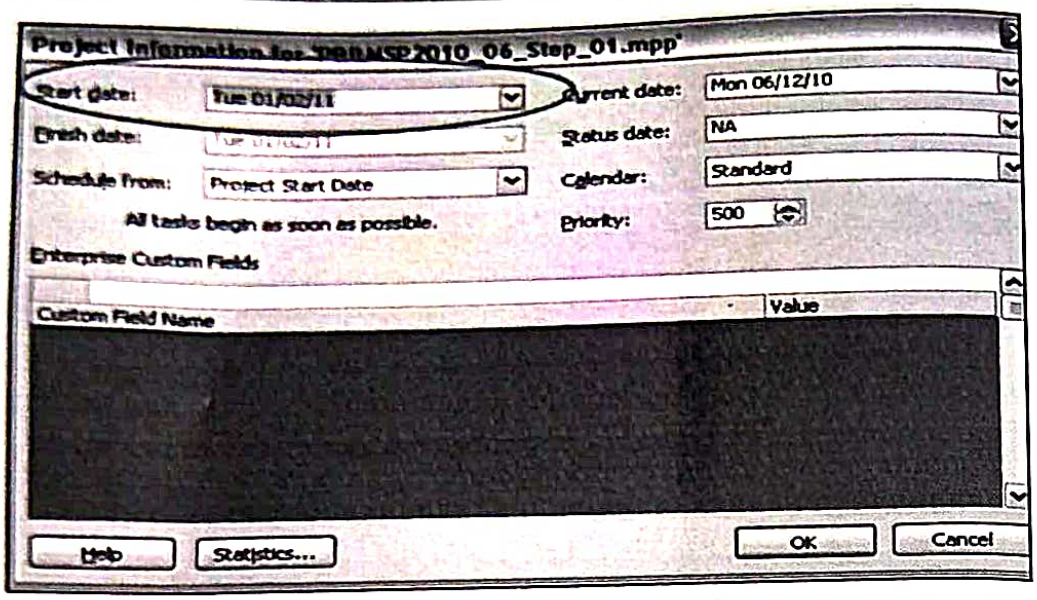



Fig. 5.2 (e): Select Date for Project

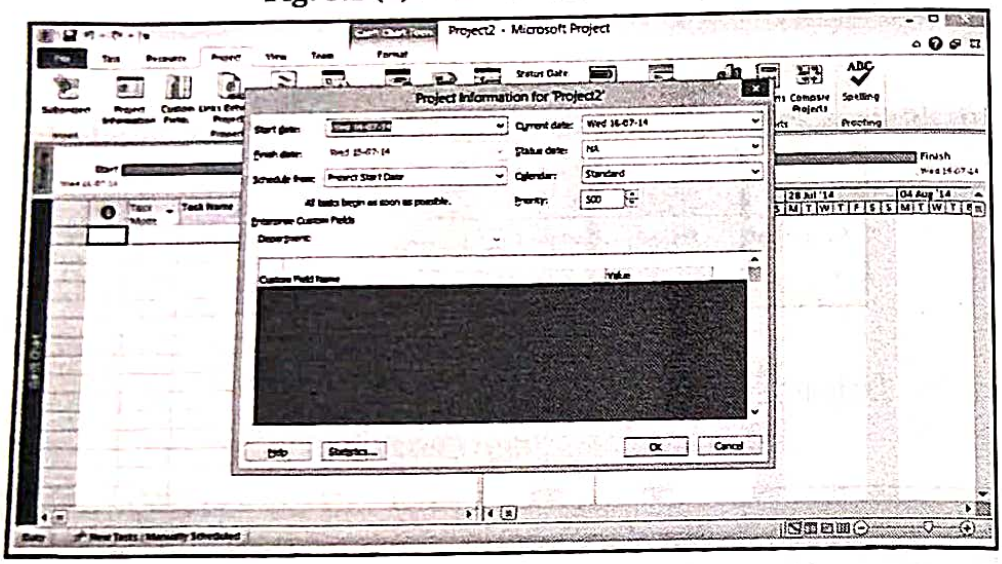



Fig. 5.2 (f): Enter Additional Project Properties

**Assigning a Project Calendar:**

- For a project to correctly determine a schedule, working and non-working times should be included. Project has three default base calendar-standard, 24 hours and night shift.

**To create a Calendar:**

1. From **Project** tab click **Change Working Time.**

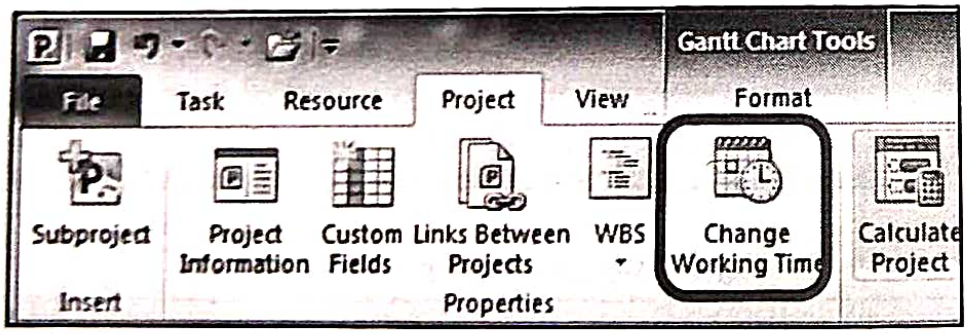



Fig. 5.3 (a): Change Working Time

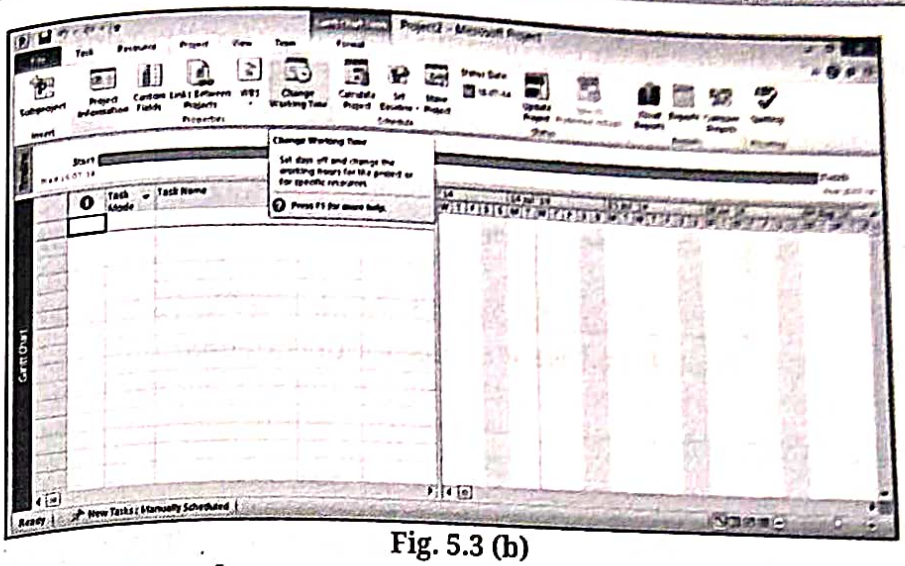


Fig. 5.3 (b)

2. Click **Create New Calendar.**

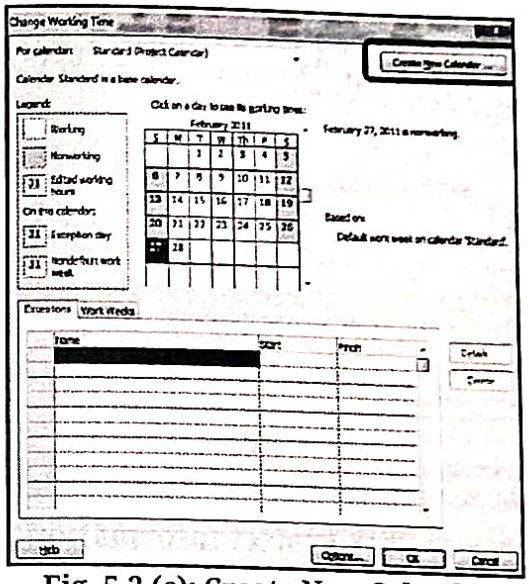



Fig. 5.3 (c): Create New Calendar

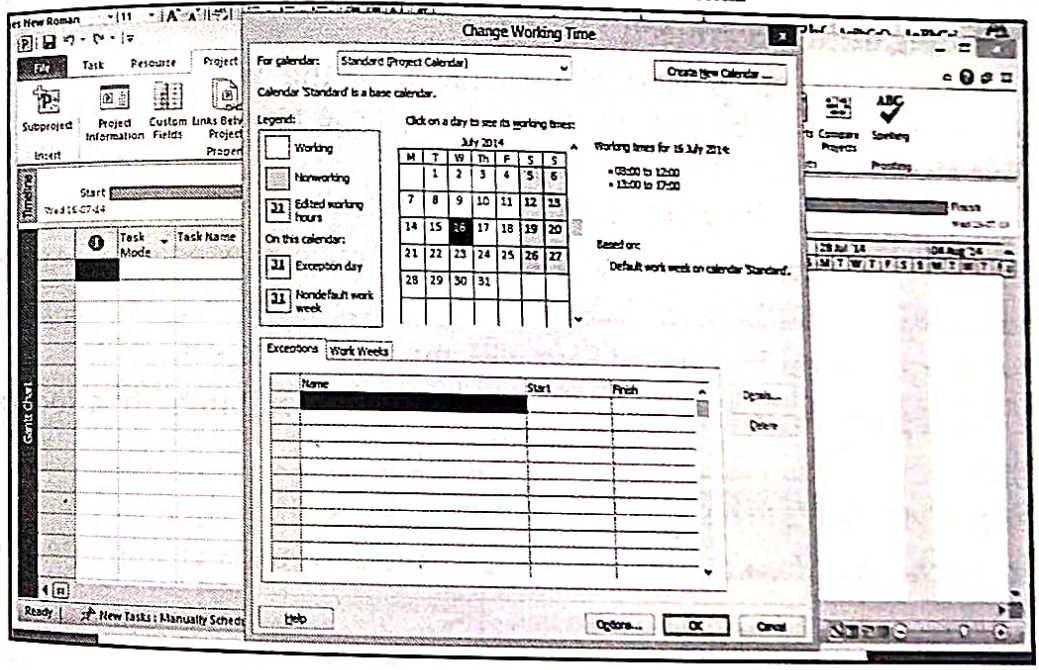



Fig. 5.3 (d)

3. Enter a name for calendar, select a type and click OK.

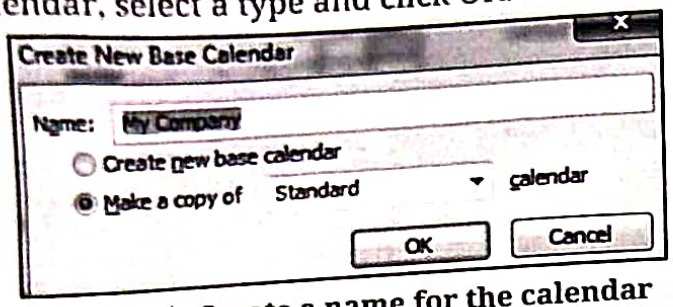



**Fig. 5.3 (e): Create a name for the calendar**

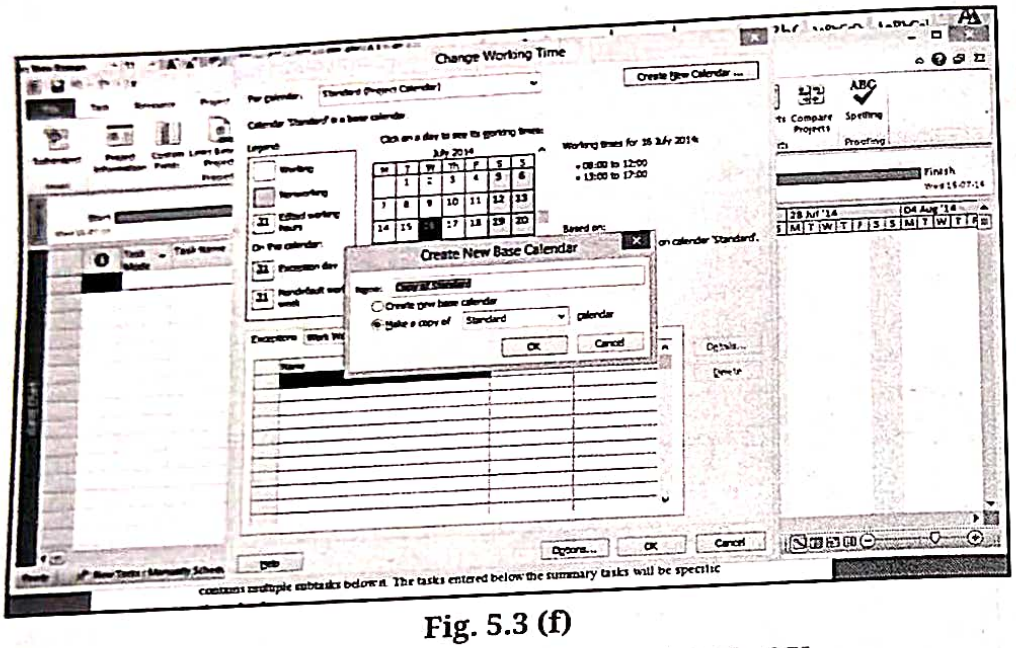



**Fig. 5.3 (f)**

4. Specify exceptions (holidays, non-working days) and click OK.
5. To assign a calendar to project: Click **Project Information** from **Project Tab**. Select the desired calendar from dropdown and click OK.

**Project Scheduling using MS Project:**

- Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these taskswill be executed. Project scheduling also help you do the following:
  - They provide a basis for you to monitor and control project activities.
  - They help you determine how best to allocate resources so you can achieve the project goal.
  - They help you assess how time delays will impact the project.
  - You can figure out where excess resources are available to allocate to other projects.
  - They provide a basis to help you track project progress.
- In order to **schedule the project activities**, a software project manager needs to do the following:
  1. Identify all the tasks needed to complete the project.
  2. Break down large tasks into small activities.

3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

**Create a Task List and Work Breakdown Structure:**

Once you have created a project plan and assigned a calendar, the task can be entered. To add tasks, first we will create summary task. The summary task is the general focus of a task and contains multiple subtasks below it. The tasks entered below the summary tasks will be specific planned tasks.

**To create summary tasks:**

1. Select **View** tab and click **Gantt chart** to view the entry mode.

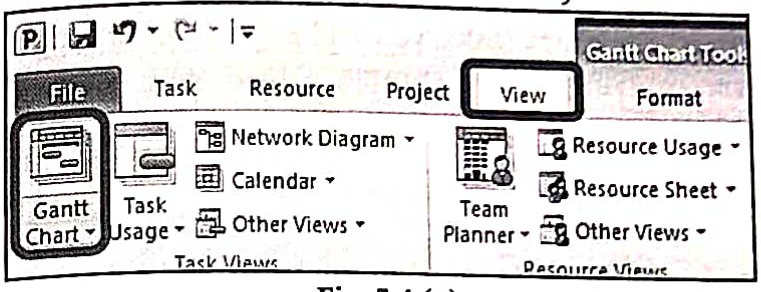



**Fig. 5.4 (a)**

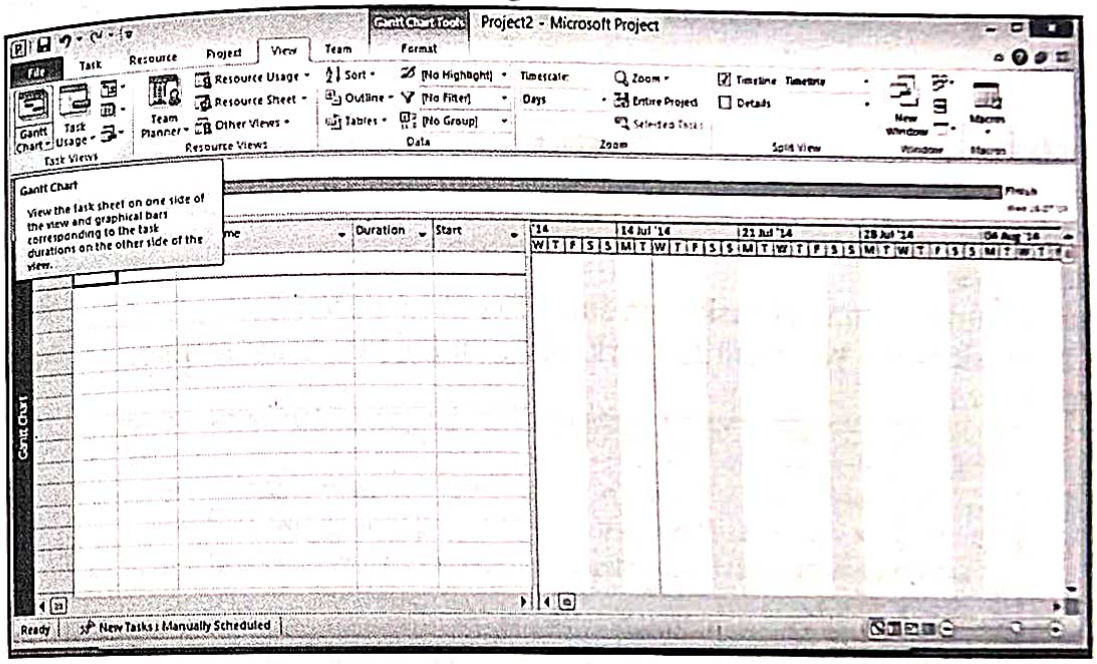



**Fig. 5.4 (b)**

2. In the **task name** field type a description of first major step of your project. Continue to enter all tasks associated with your project.

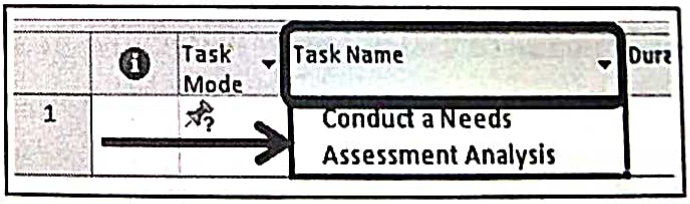



**Fig. 5.4 (c)**

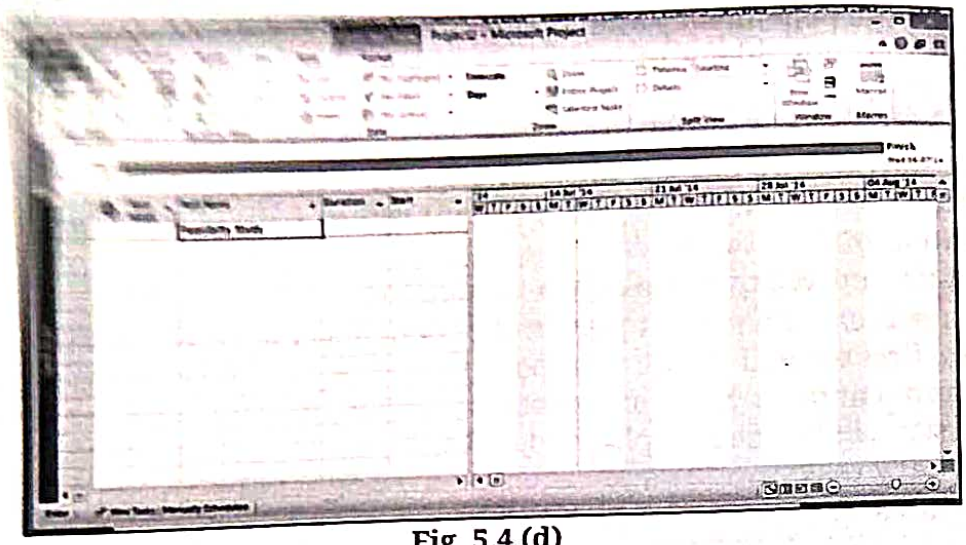


Fig. 5.4 (d)

- After creating all your summary tasks, you will begin to enter the rest of your tasks.
  1. Select the first row under the summary task you are working with, so it is highlighted.
  2. Under **Task** tab select **Task** and then **Task** again .you also can add a task by right click and selecting **Insert Task**. Complete.

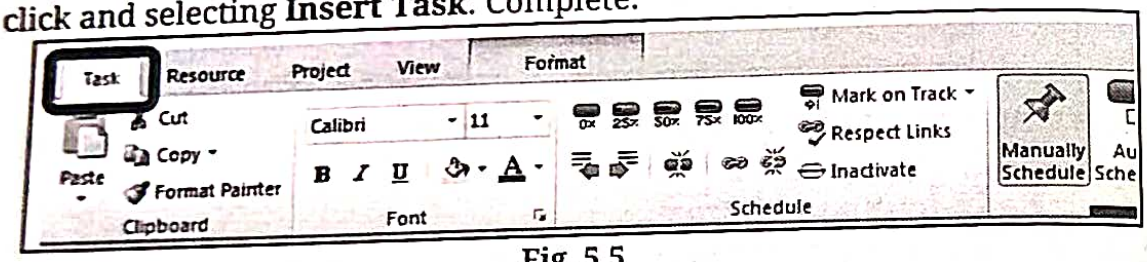



Fig. 5.5

3. Continue to add rows and subtasks until your entire summary task detailed are complete.

**Organizing Tasks:**

- You can organize summary tasks and sub tasks in hierarchy.

1. Select the task name you wish to work with, under the **Task** tab select the **indent** or **outdent** button.

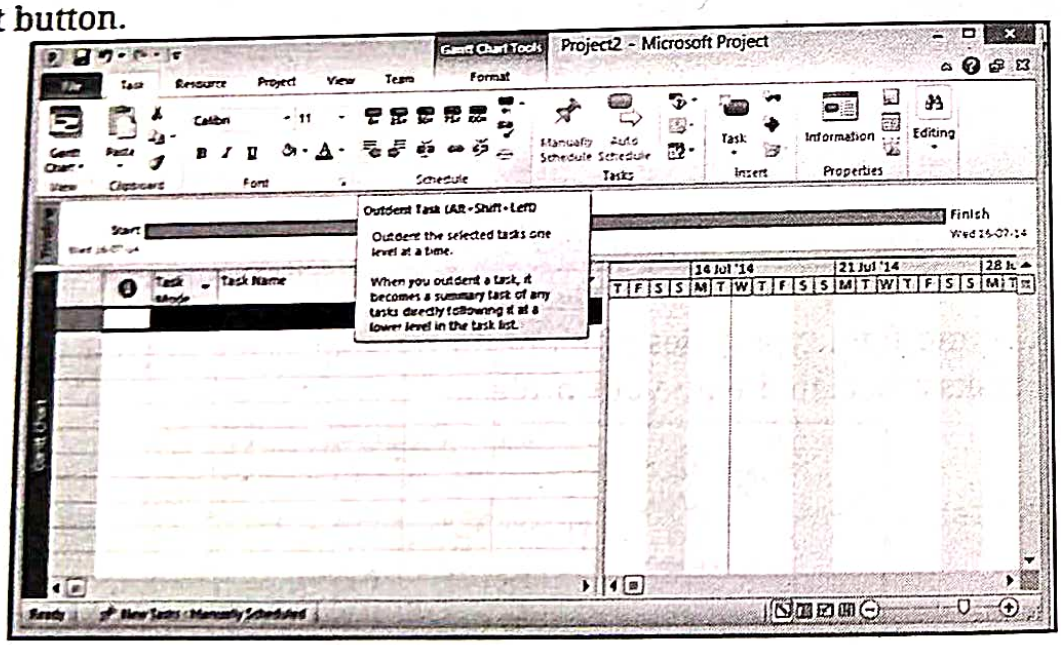



Fig. 5.6 (a)

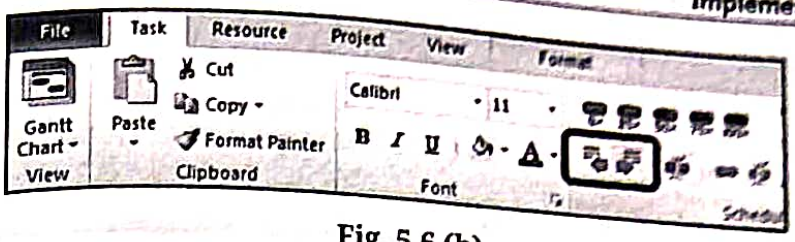



Fig. 5.6 (b)

After completing indentation project is able to identify summary tasks and subtasks based on indentation pattern.

**Work Breakdown Structure:**

A work breakdown structure is a logical hierarchy of tasks in a project represented by alphanumeric codes similar to the outline.

To view outline number on all summary tasks and subtasks, check the **Outline Number Box** under the **Format** tab.

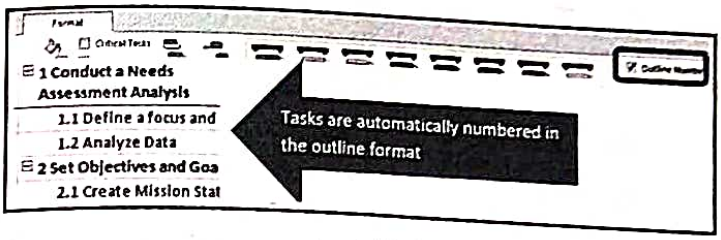



Fig. 5.7 (a)

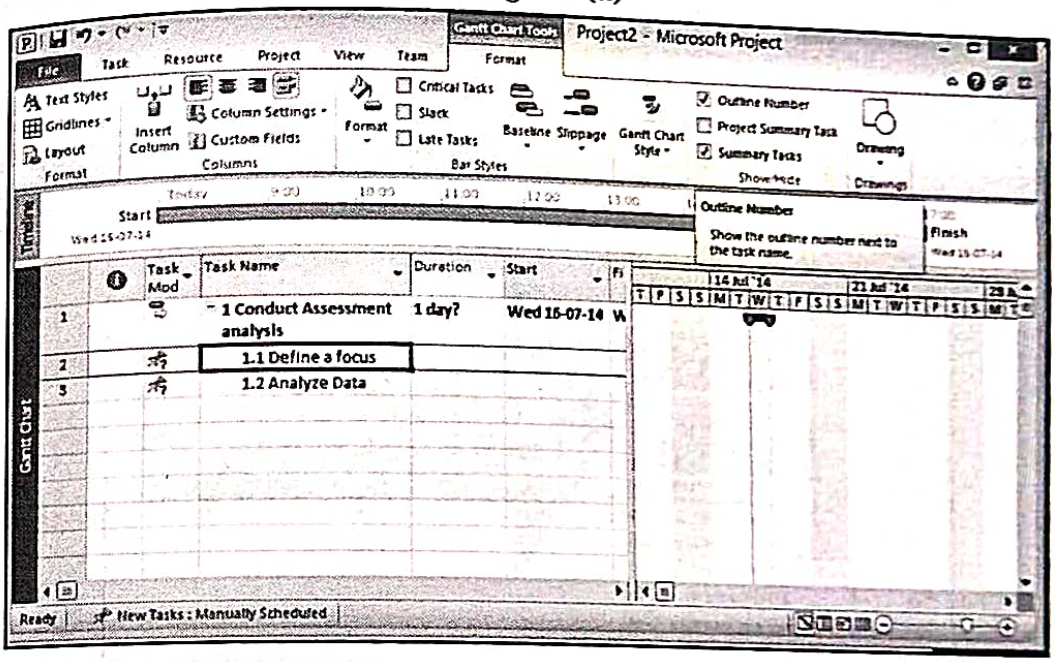



**Fig. 5.7 (b): Outline Number**

Enter **Duration:**

- Work is the number of hours between all contributors spends on the task and duration is the amount of time in which you want the task to be completed.
- When we enter durations,we enter them in subtasks cells.The duration of summary task will automatically update according to subtask.To enter the duration of subtask, click inside the empty cell under **Duration** column.

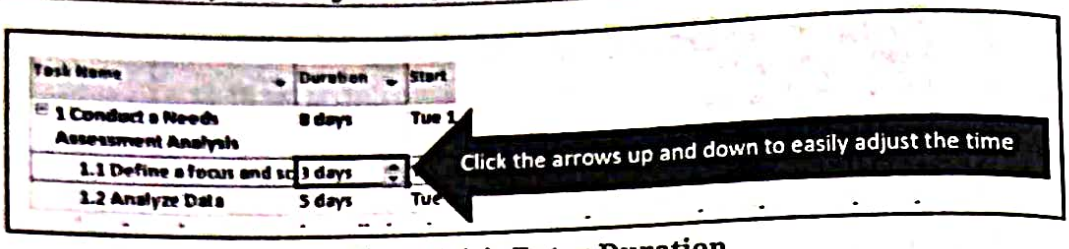



Fig. 5.8 (a): Enter Duration

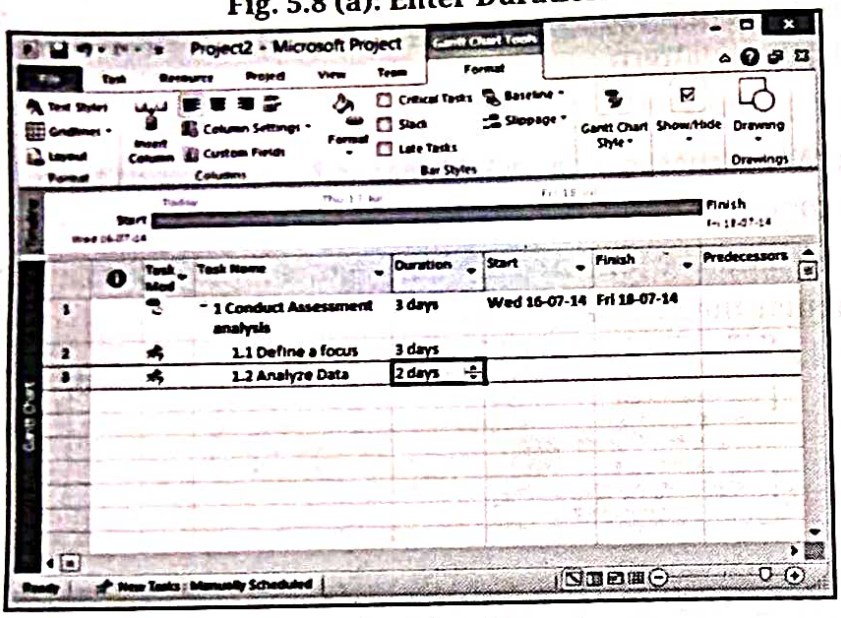



Fig. 5.8 (b)

**Creating Dependencies:**

- To link tasks together, select the first task that need to completed and then the task that can be started. Use this pattern to link as many tasks as desired by holding the ctrl key.

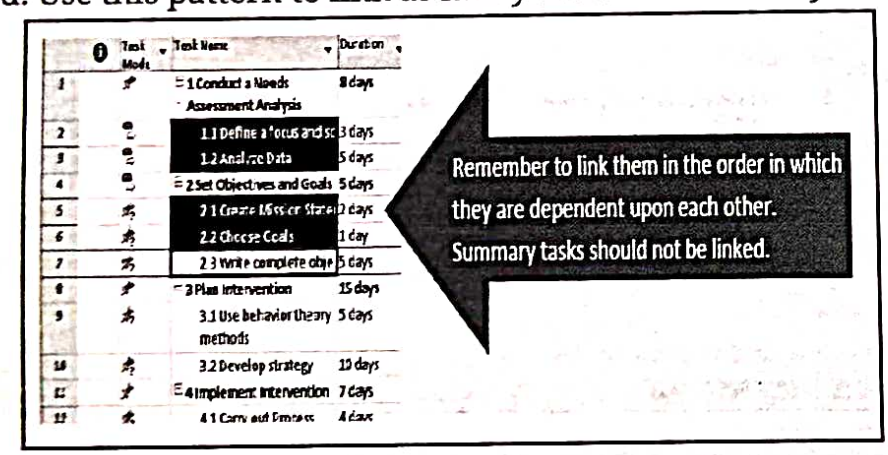



Fig. 5.9 (a)

- Click the **Link Tasks** icon under the **Task** tab.

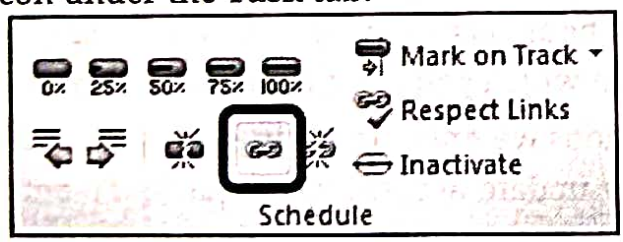



Fig. 5.9 (b)

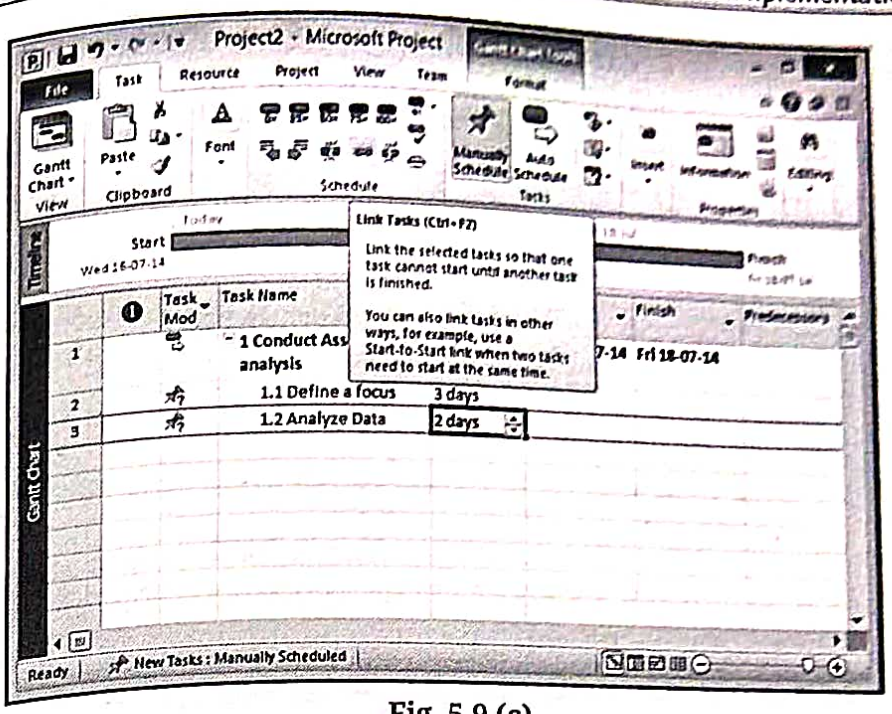



**Fig. 5.9 (c)**

- You will see how each task leads to another on the timeline.
- After linking tasks and creating dependencies, you will see number in **Predecessors** columns as shown in following fig. These numbers represent the ID of the task that must be completed prior to the highlighted task.

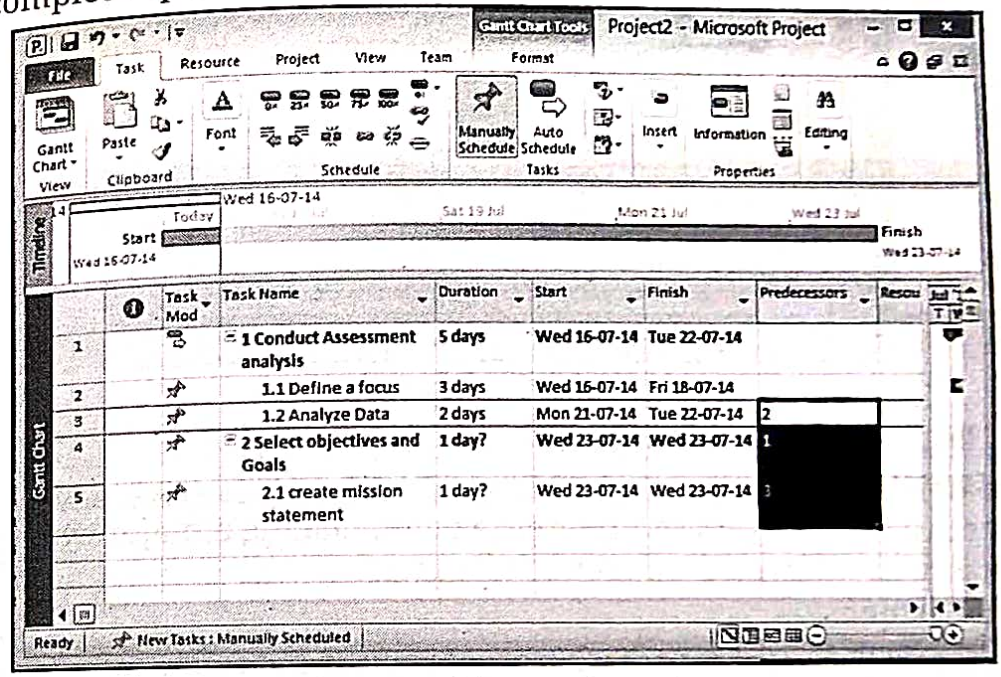



**Fig. 5.9 (d): ID of the task**

## Predecessor and Dependency Types:

- There are four types of dependencies:
  1. **Finish to Start**: The task can't be started until the predecessor task is completed.
  2. **Start to Start**: The task can't be started until the predecessor task is started.
  3. **Finish to Finish**: The task can't be finished until the predecessor task is finished.
  4. **Start to Finish**: The task can't be finished until the predecessor task begins.

- To enter a predecessor, In **Predecessor** column enter the number of the row the task is dependent upon. Double click to select the specific dependency type of task.

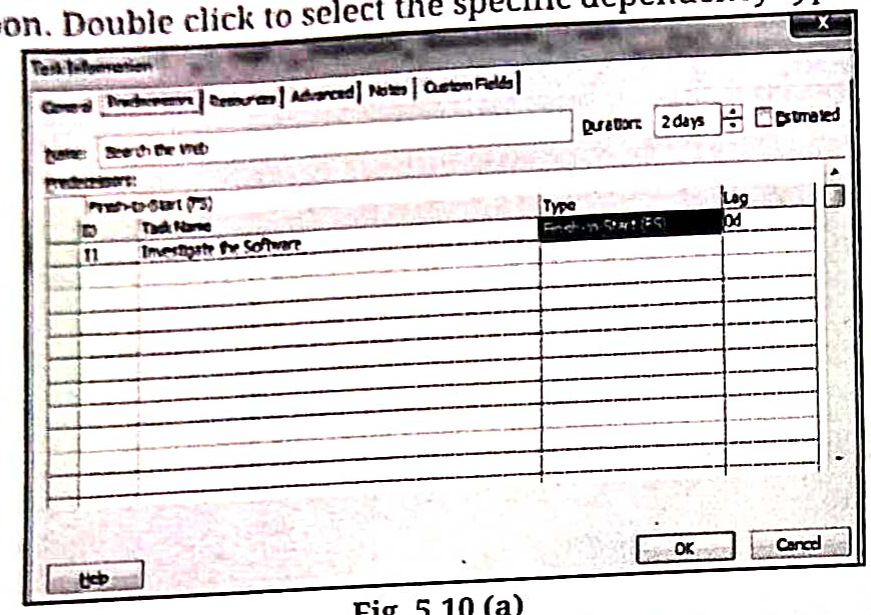



**Fig. 5.10 (a)**

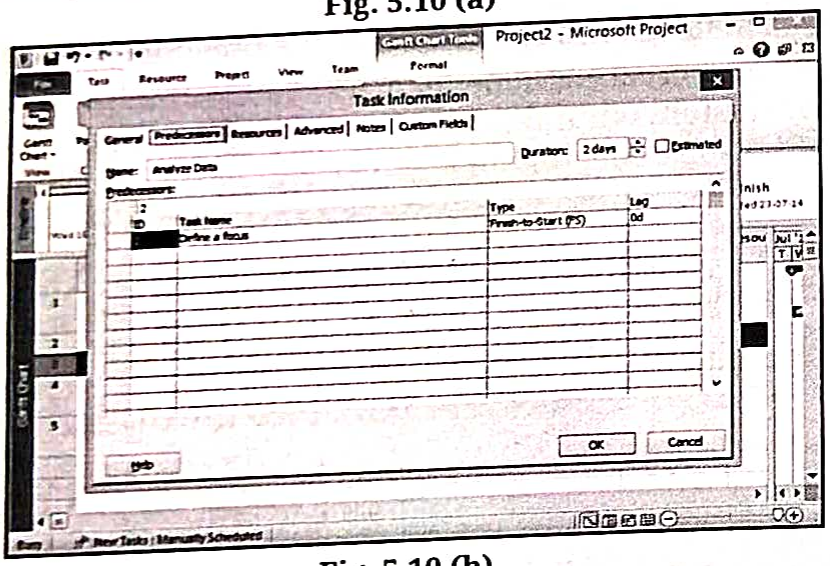



**Fig. 5.10 (b)**

- The link is displayed in the Gantt chart.

**Adding Milestones:**

- A milestone is a task that acts as a reference point, making a major event in the project and which is used to monitor the project's progress.
- **To add a milestone:**
  1. Click the row below the row of where you want to put milestone.
  2. Select the **Milestone** icon under the Task tab.

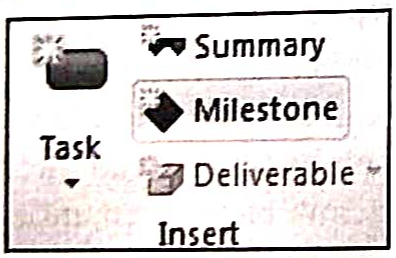



**Fig. 5.10 (c)**

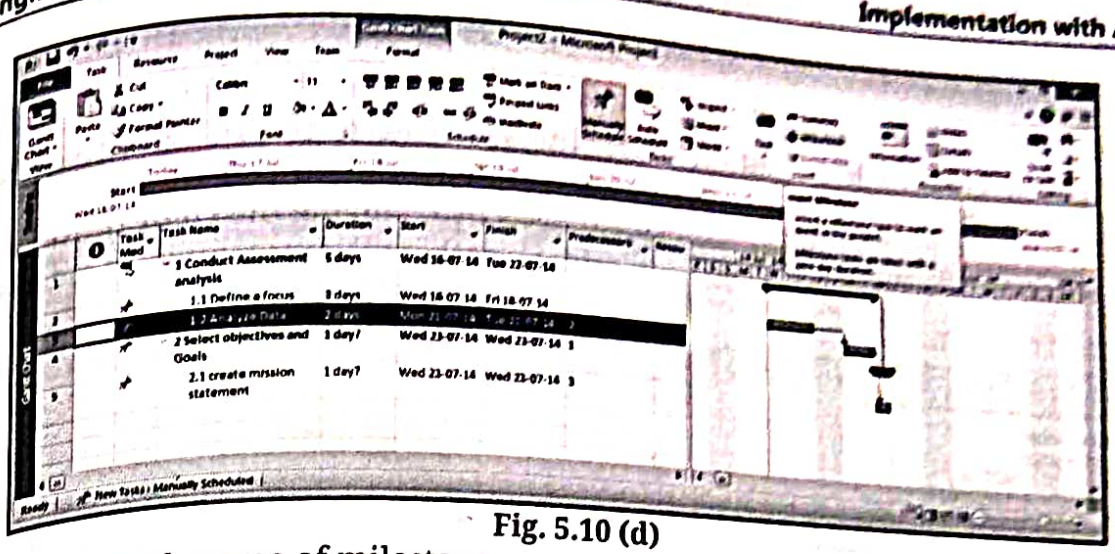



Fig. 5.10 (d)

3. Type the task name of milestone.
4. The milestone will then appear on the calendar timeline with an asterisk. Any task can be identified as milestone by making the duration value 0.

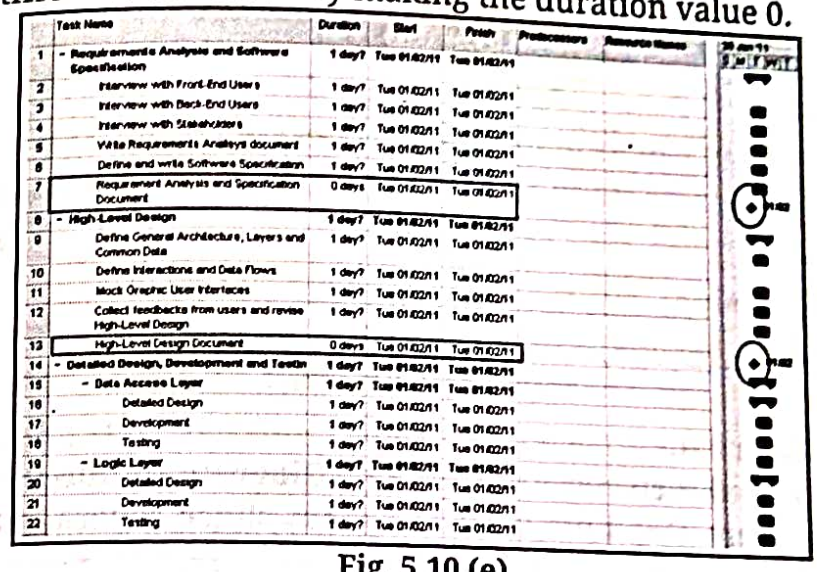



Fig. 5.10 (e)

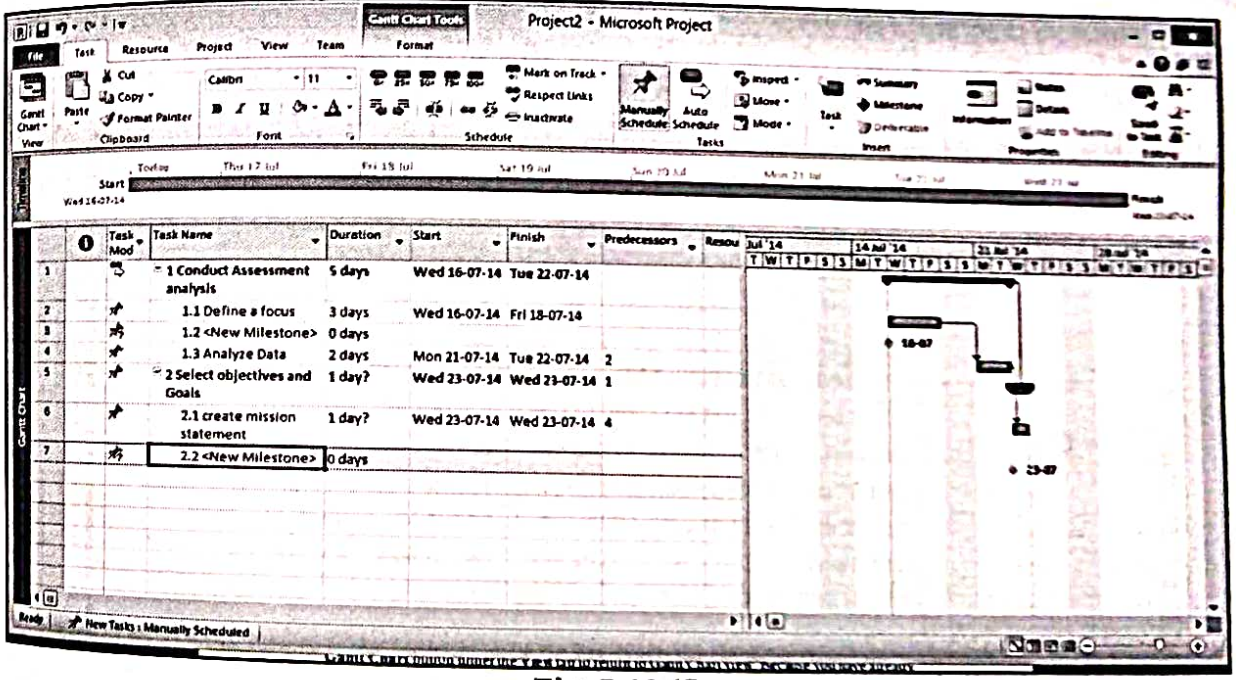



Fig. 5.10 (f)

**View the Gantt Chart:**

- Project 2010 shows a Gantt chart as the default view to the right of the Entry table. Or Click the **Gantt Chart** button under the **View** tab to return to Gantt Chart view. Because you have already created task dependencies, you can now find the critical path for Project. You can view the critical tasks by changing the color of those items in the Gantt Chart view. Tasks on the critical path will automatically be red in the Network Diagram view.

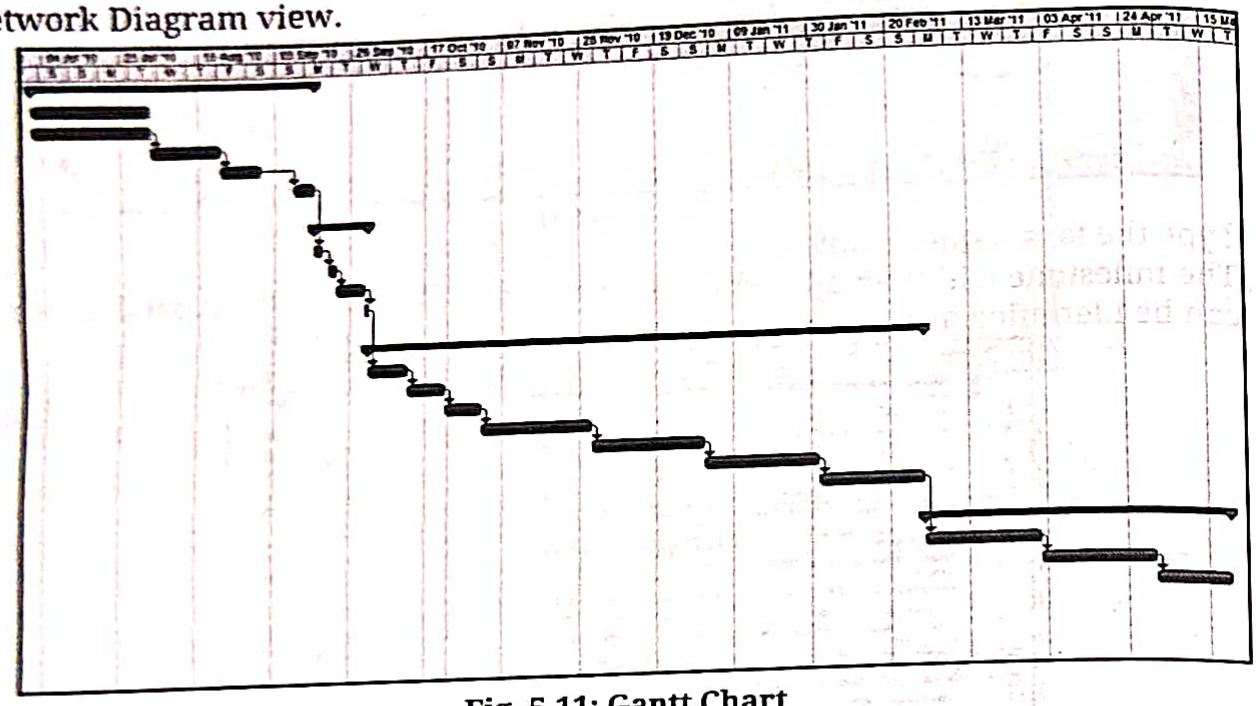


**Fig. 5.11: Gantt Chart**

**View the Network Diagram:**

- Click the **View** tab, and then click the **Network Diagram** button under the **Task Views** group. Click the **Zoom Out** button on the Zoom slider several times and watch the view change.

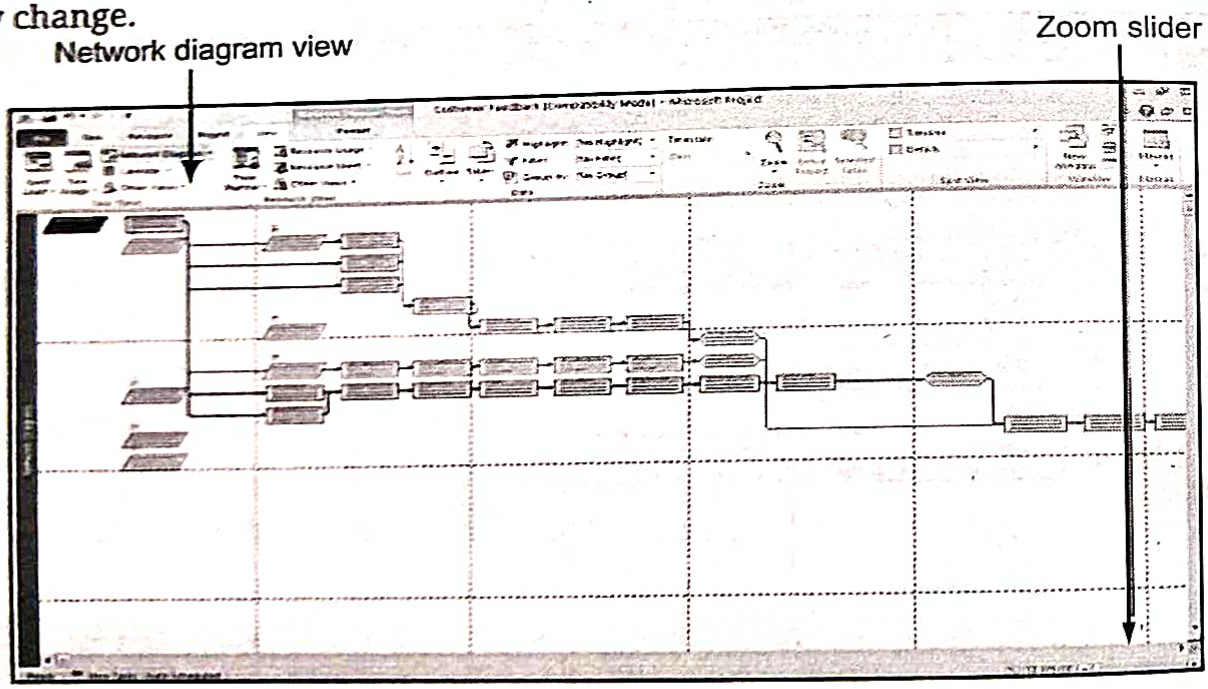



**Fig. 5.12 (a): Network Diagram**

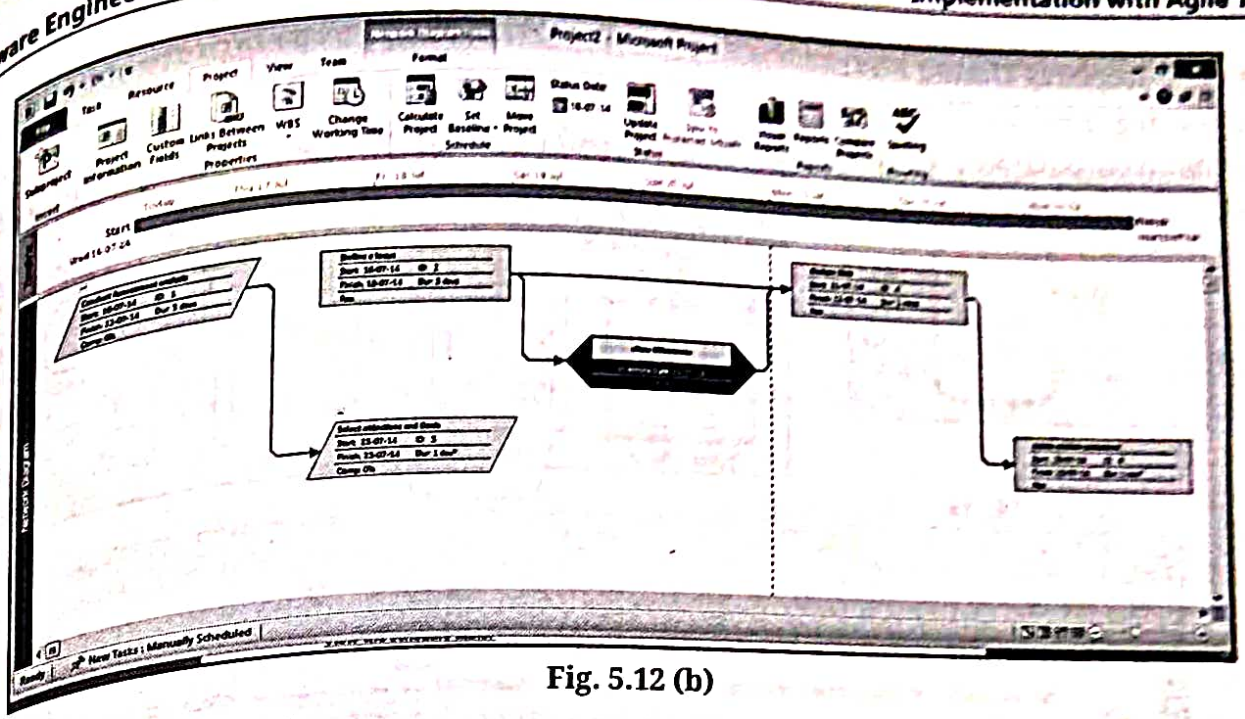


Fig. 5.12 (b)

**View the Schedule Table::**

- Click the **Gantt Chart** button under the **View** tab to return to Gantt Chart view. Right-click the **Select All** button to the left of the Task Mode column heading and select **Schedule**.
- Alternatively, you can click the **View** tab and click the **Tables** button under the Data group and then select **Schedule**. The Schedule table replaces the Entry table to the left of the Gantt Chart.
- This view shows the start and finish (meaning the early start and early finish) and late start and late finish dates for each task, as well as free and total slack. Right-click the **Select All** button and select **Entry** to return to the Entry table view.

| | Task Name | Start | Finish | Late Start | Late Finish | Free Slack | Total Slack |
|---|---|---|---|---|---|---|---|
| 1 | 1 Mobile Reporting | Mon 9/11/09 | Thu 10/12/09 | Mon 9/11/09 | Thu 10/12/09 | 0 days | 0 days |
| 2 | 1.1 Performance | Mon 9/11/09 | Wed 18/11/09 | Mon 9/11/09 | Wed 18/11/09 | 0 days | 0 days |
| 3 | 1.1.1 Interview | Mon 9/11/09 | Tue 17/11/09 | Mon 9/11/09 | Tue 17/11/09 | 0 days | 0 days |
| 4 | 1.1.2 Define th | Wed 18/11/09 | Wed 18/11/09 | Wed 18/11/09 | Wed 18/11/09 | 0 days | 0 days |
| 5 | 1.2 Software | Thu 19/11/09 | Fri 4/12/09 | Thu 19/11/09 | Fri 4/12/09 | 0 days | 0 days |
| 6 | 1.2.1 Design L | Thu 19/11/09 | Mon 23/11/09 | Thu 19/11/09 | Mon 23/11/09 | 0 days | 0 days |
| 7 | 1.2.2 Design D | Tue 24/11/09 | Wed 25/11/09 | Tue 24/11/09 | Wed 25/11/09 | 0 days | 0 days |
| 8 | 1.2.3 Code A | Thu 26/11/09 | Fri 4/12/09 | Thu 26/11/09 | Fri 4/12/09 | 0 days | 0 days |
| 9 | 1.2.4 Code B | Thu 26/11/09 | Wed 2/12/09 | Mon 30/11/09 | Fri 4/12/09 | 2 days | 2 days |
| 10 | 1.3 Hardware | Thu 19/11/09 | Tue 1/12/09 | Tue 24/11/09 | Fri 4/12/09 | 3 days | 3 days |
| 11 | 1.3.1 Purchas | Thu 19/11/09 | Fri 27/11/09 | Tue 24/11/09 | Wed 2/12/09 | 0 days | 3 days |
| 12 | 1.3.2 Assembl | Mon 30/11/09 | Tue 1/12/09 | Thu 3/12/09 | Fri 4/12/09 | 3 days | 3 days |
| 13 | 1.4 Prototype | Fri 4/12/09 | Thu 10/12/09 | Mon 7/12/09 | Thu 10/12/09 | 0 days | 0 days |
| 14 | 1.4.1 Integratic | Fri 4/12/09 | Fri 4/12/09 | Thu 10/12/09 | Thu 10/12/09 | 4 days | 4 days |
| 15 | 1.4.2 Integrate | Mon 7/12/09 | Wed 9/12/09 | Mon 7/12/09 | Wed 9/12/09 | 0 days | 0 days |
| 16 | 1.4.3 Test | Thu 10/12/09 | Thu 10/12/09 | Thu 10/12/09 | Thu 10/12/09 | 0 days | 0 days |

Fig. 5.13: Schedule Table

**Open the Reports Dialog Box:**

- Click the **Project** tab, and then click the **Reports** button under the **Reports** group Double-click **Overview** to open the **Overview Reports** dialog box, and then double click **Critical Tasks**. A Critical Tasks report as of today's date is displayed.


Fig. 5.14 (a)

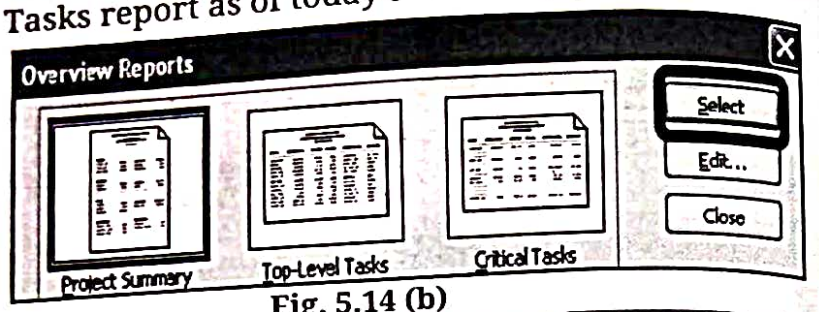



Fig. 5.14 (b)

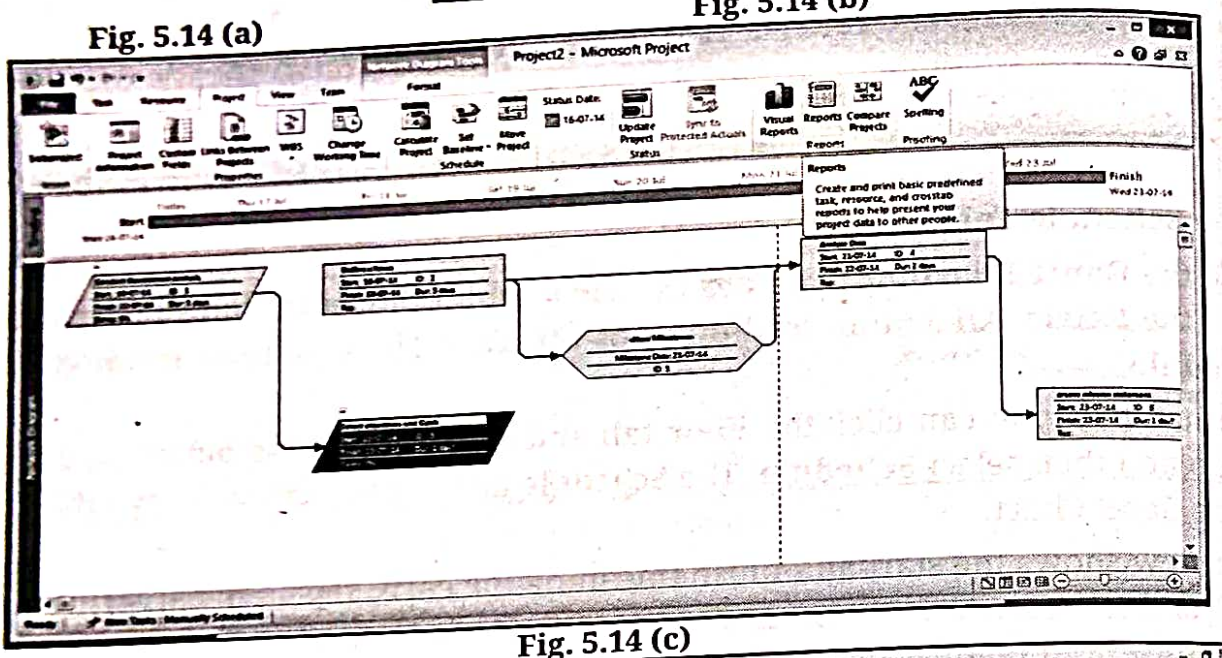



Fig. 5.14 (c)

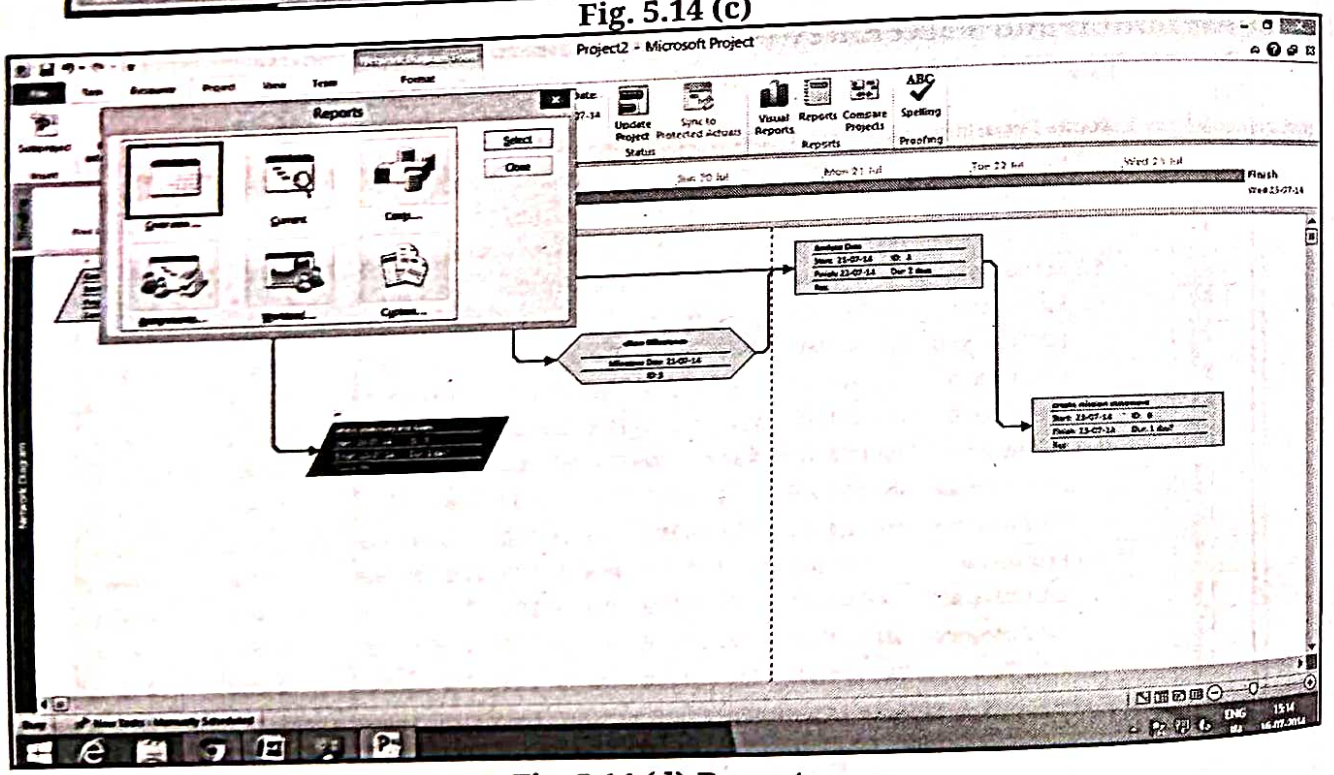



Fig. 5.14 (d):Reports

**Close the report and save your file.**

- When you are finished examining the **Critical Tasks report**, click the **Tasks**. Click the **Save** button on the Quick Access toolbar to save your final schedule.mppfile, showing the Entry table and Gantt chart view.

## 5.2 AGILE TOOLS : OPEN SOURCE

- Today, the popular frameworks for implementing Agile methodology include Scrum, Kanban, Extreme Programming, and the Adaptive Project Framework.

1. **Scrum:**

- Scrum is the most popular Agile development framework that has taken a highly iterative approach focusing on the key features and objectives. The best part about Scrum is that it is adaptable and relatively simple to implement.

2. **Kanban:**

- Kanban was developed by Toyota to help increase productivity in factories. It is another framework for Agile based methodology. This approach gives a quick way to bring code to production. Teams can have a visual representation of their tasks and keep a transparency in the progress of their tasks by moving it through stages and identify where roadblocks occur.

3. **Extreme Programming (XP):**

- Extreme programming is a disciplined approach to improve the quality (and simplicity) of software and deliver high-quality software continuously. In XP, the team estimates, plan and deliver highest priorities user stories and constantly collaborate with high-quality software.

4. **Adaptive Project Framework (APF):**

- Due to changing requirements in most IT projects, teams started using Adaptive Project Framework. APF is basically designed to continuously adapt to the changing situation of a project. Stakeholders also use this approach to change the scope of the project at the start of each stage to produce the most business value.
- Using agile tools and techniques can help to:
  - Self-organize and plan.
  - Communicate within the team and with the rest of your organization.
  - Continuously improve the way you work.
  - Get support from management.
- In Agile development, leading as project management is not the easiest job. Jumping between your daily scrums to your next sprint, it causes hard to focus on the work. The agile development tool fulfills your needs, and does it for you.

## Agile Tools:

- There are several agile tools available in the market. Some of them are listed below:

**Table 5.1: Agile Tools**

| Sr. No. | Tool | Description | Features | Logo |
|---|---|---|---|---|
| 1. | JIRA Agile | **Jira** is a tool developed by Australian Company **Atlassian.** It is used for **issue tracking, bug tracking, and project management.** The bugs and issues are related to your software and Mobile apps. The Jira dashboard consists of many useful functions and features. This function and features make secure handling of issues | ○ Issue tracking<br>○ Bug tracking<br>○ Boards<br>○ Epics<br>○ Custom fields | JIRA |
| 2. | Github | **Github** is one of the largest hosted Git serve where the developers can store all of their codes for a vast number of projects there. The Github provides such a facility of record edits across an entire team in **real time**. Github is also integrated with many other tools so, many people such as developer and product owner can work on the same code at the same time.<br>The project manager can make the Github work for their team. It includes lots of project management tools which help him to inspect what the development team is working on. | ○ Issue tracking<br>○ Mentions<br>○ Labels<br>○ Link issues and pull requests | |

| Sr. No. | Tool | Description | Features | Logo |
|---|---|---|---|---|
| 3. | LeanKit | **LeanKit** is the ultimate management tool for a **Kanban** board on the agile progress for your sprints. It uses cards to represent the work items and live statuses of team member. It work perfect for the remote employees to ensure everyone can see the Kanban board in real time. It prevents the same task to complete twice and make sure the whole team remains on the same page. LeanKit work well for cross-functional team which is benefit for Scrum or Kanban boards. | ○ Board view templates<br>○ Track issues and bugs<br>○ Manage project portfolios<br>○ Lean metrics and reporting | leankit |
| 4. | Backlog | Backlog is an all-in-one project management tool built for developers. Agile Teams use Backlog to work with other teams for enhanced team collaboration and high-quality project delivery. | ○ Easy bug tracking tool<br>○ Project and issues with subtasks<br>○ Git and SVN built-in<br>○ Gantt Charts and Burndown charts<br>○ Wikis<br>○ Watchlists<br>○ Native mobile apps<br>○ Available both in cloud and on-premise | backlog |
| 5 | Zoho Sprints | Sprints is a tool that helps you to manage your team and product with ease. It enables you to track your progress with no hassle. This software can be used to find bottlenecks and discover ways to generate | ○ It is integrated with CI/CD tools.<br>○ This tool helps you to get product feedback with ease.<br>○ Allows you to work on any | ZOHO |

| Sr. No. | Tool | Description | Features | Logo |
|---|---|---|---|---|
| | | business value. | device and place.<br>o Enables the team to comment on code changes. | |
| 6 | Wrike | Wrike is ideal for centralizing and connecting multiple projects and spiking up your team's efficiency. | o Holistic, comprehensive task modeling<br>o Loads of configurable features<br>o Community feedback voting for roadmap features | wrike |
| 7 | Taiga | An open source project management tool to solve the problem of software usability. It is a reliable and popular platform for small developers, designers, project managers for applications like project collaboration, reporting, time tracking, and task management. | o Easily create cards and track progress<br>o Options for adding custom user inputs<br>o Saves time by easily replicating past workflows | TAIGA |

## 5.3 HANDS ON GITHUB

- Git is an open-source, version control tool created in 2005 by developers working on the Linux operating system. GitHub is a company founded in 2008 that makes tools which integrate with Git. We cannot use GitHub without using Git.
- GitHub is highly used software that is typically used for version control. It is helpful when more than just one person is working on a project. For example, a software developer team wants to build a website and everyone has to update their codes simultaneously while working on the project. In this case, GitHub helps them to build a centralized repository where everyone can upload, edit, and manage the code files.

**Creating a Repository:**

- A repository is a storage space where your project lives. It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, images or any kind of a file in a repository.
- You need a GitHub repository when you have done some changes and are ready to be uploaded. This GitHub repository acts as your remote repository. Follow these simple steps to create a GitHub repository:

- Go to the link: https://github.com/. Fill the sign up form and click on "Sign up for GitHub".
- Click on "Start a new project".
- Refer to the below screenshot to get a better understanding.

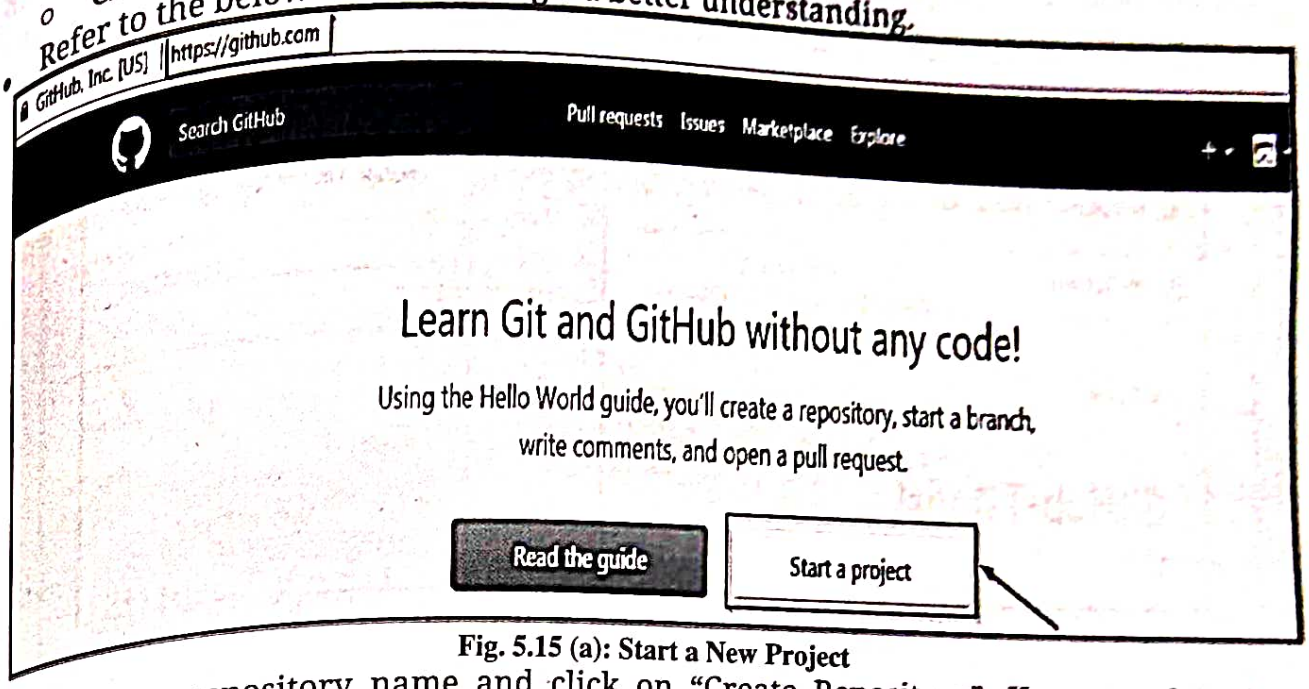



**Fig. 5.15 (a): Start a New Project**

- Enter any repository name and click on "Create Repository". You can also give a description to your repository (optional).

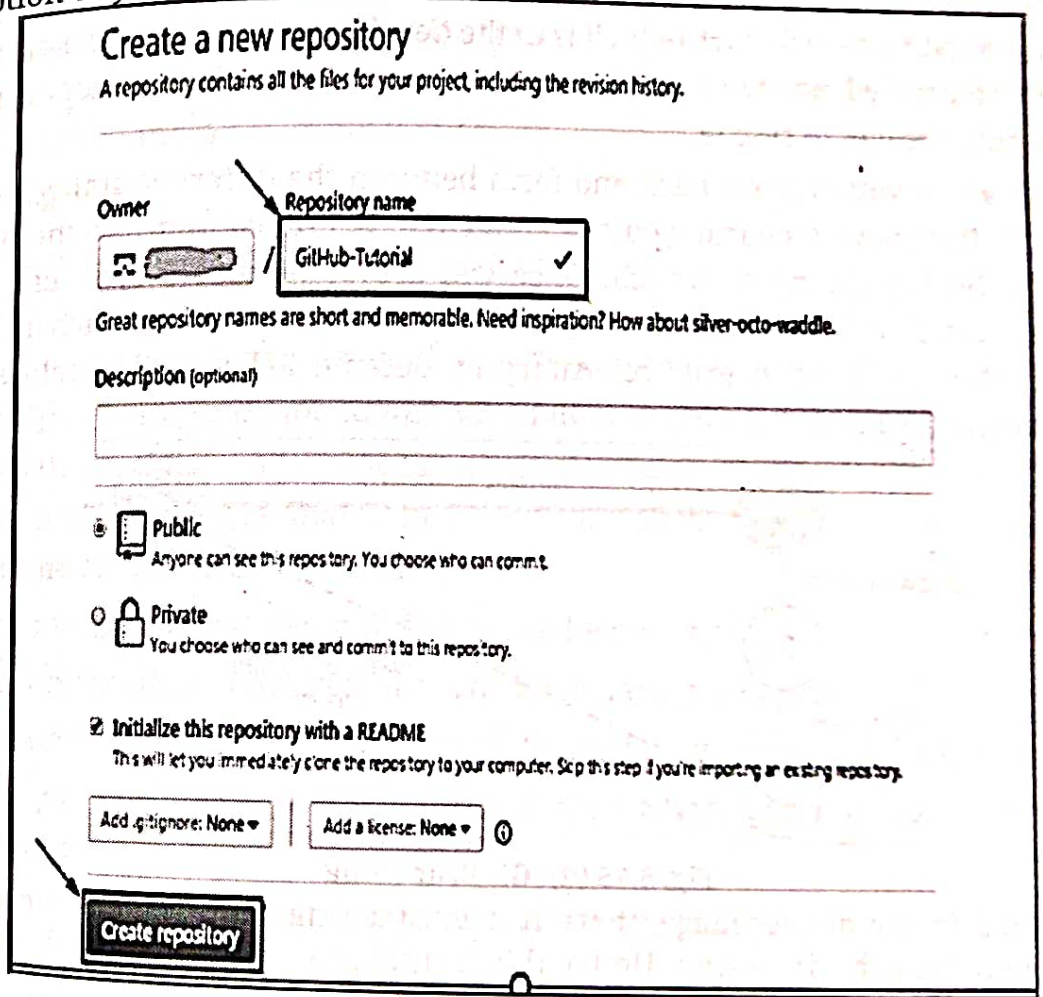



**Fig. 5.15 (b): Create Repository**

- Now, if you noticed by default a GitHub repository is Public this means that anyone can view the contents of this repository whereas in a Private repository, you can choose who can view the content. If your repository is successfully created then you will see below screenshot:

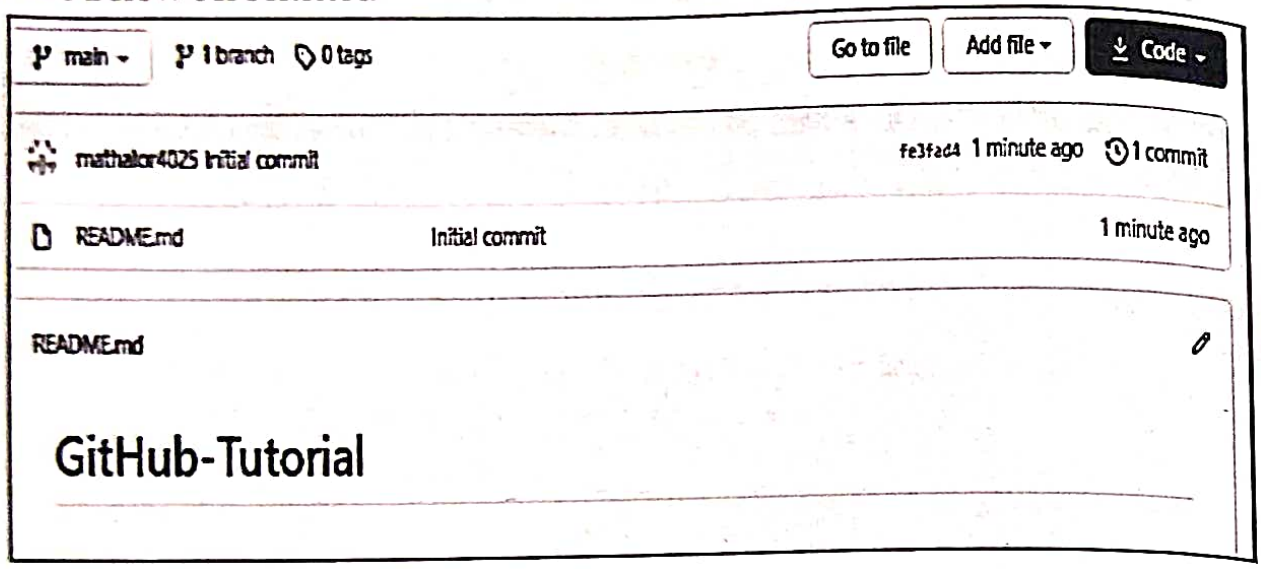



**Fig. 5.15 (c) Contents of Repository**

**Create Branches:**

- Branches help you to work on different versions of a repository at one time. Let's say you want to add a new feature (which is in the development phase), and you are afraid at the same time whether to make changes to your main project or not. This is where Git branching comes to rescue.
- Branches allow you to move back and forth between the different states/versions of a project. In the above scenario, you can create a new branch and test the new feature without affecting the main branch. Once you are done with it, you can merge the changes from new branch to the main branch. Here the main branch is the master branch, which is there in your repository by default. Refer to the below image for better understanding:

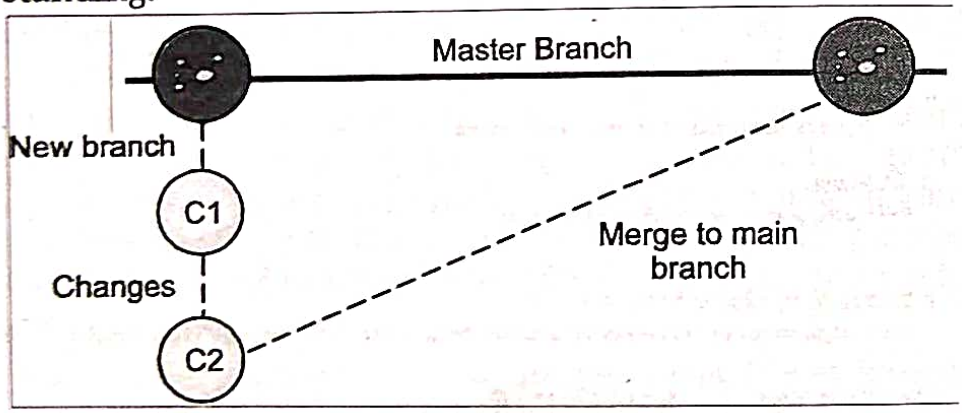



**Fig. 5.15 (d): Git Branching**

- As depicted in the above image, there is a master (main)/ production branch which have a new branch for testing. Under this branch, two set of changes are done and once it completed, it is merged back to the master branch.

- To create a branch in GitHub, follow the below steps:
  - Click on the dropdown "**Branch: main**".
  - As soon as you click on the branch, you can find an existing branch or you can create a new one. In this case, you are creating a new branch with a name "**readme- changes**". Refer to the below screenshot for better understanding.

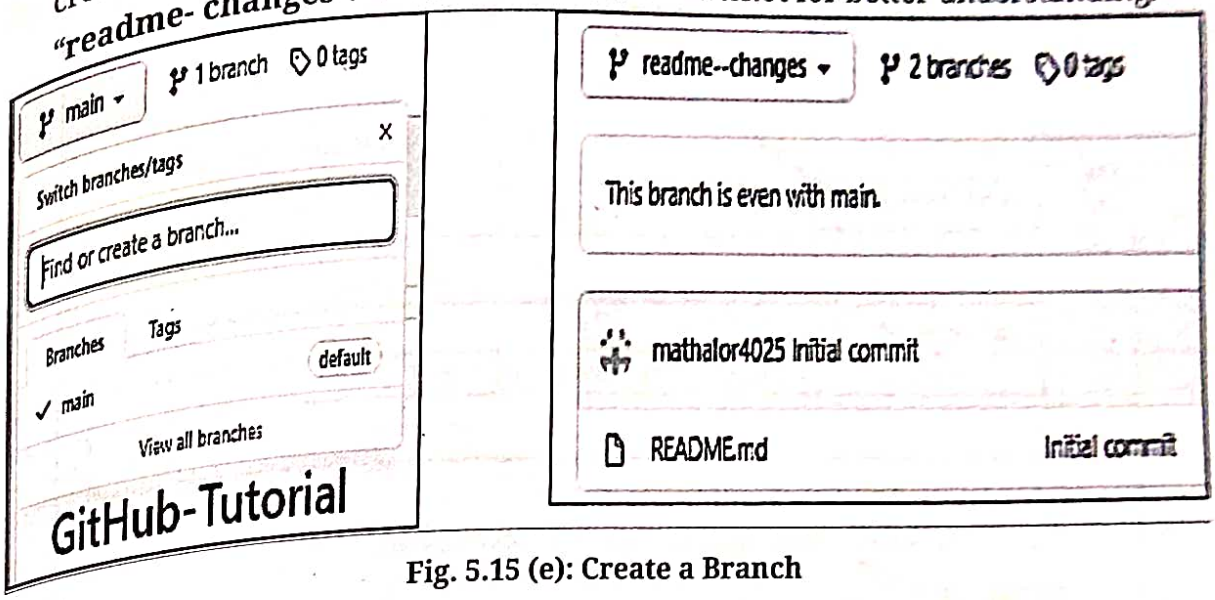



**Fig. 5.15 (e): Create a Branch**

  - Once you have created a new branch, you have two branches in your repository now main and readme- changes. The new branch is just the copy of main branch. So let's perform some changes in our new branch and make it look different from the main branch.

**GitHub Operations:**

1. **Commit Command:**

- This operation helps you to save the changes in your file. When you commit a file, you should always provide the message, just to keep in the mind the changes done by you. Though this message is not compulsory but it is always recommended so that it can differentiate the various versions or commits you have done so far to your repository. These commit messages maintain the history of changes which in turn help other contributors to understand the file better.
- Now let's make our first commit, follow the below steps:
  - Click on "readme- changes" file which we have just created.
  - Click on the "edit" or a pencil icon in the rightmost corner of the file.
  - Once you click on that, an editor will open where you can type in the changes or anything.
  - Write a commit message which identifies your changes.
  - Click Commit changes in the end.

- Refer to the below screenshot for better understanding:

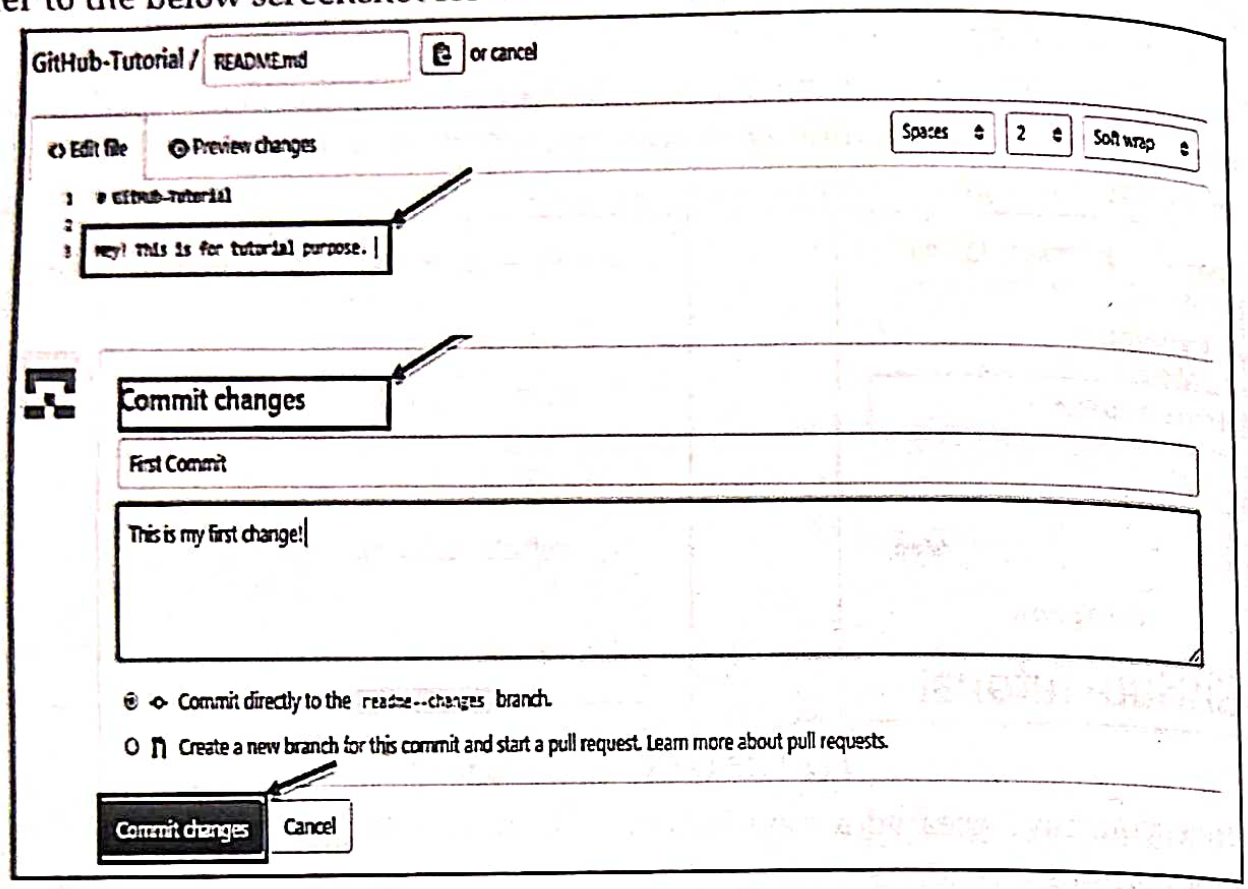



**Fig. 5.15 (f): Commit Changes**

- We have successfully made our first commit. Now this "readme- changes" file is different from the master branch. Next, let us see how we can open a pull request.

**2. Pull Command:**

- Pull command is the most important command in GitHub. It tells the changes done in the file and requests other contributors to view it as well as merge it with the master branch. Once the commit is done, anyone can pull the file and can start a discussion over it. Once it's all done, you can merge the file. Pull command compares the changes which are done in the file and if there are any conflicts, you can manually resolve it.
- Now let us see different steps involved to pull request in GitHub:
  - Click the "Pull requests" tab.
  - Click "New pull request".
  - Once you click on Pull request, select the branch and click "readme- changes" file to view changes between the two files present in our repository.
  - Click "Create pull request".
  - Enter any title, description to your changes and click on "Create pull request". Refer to the below screenshots.

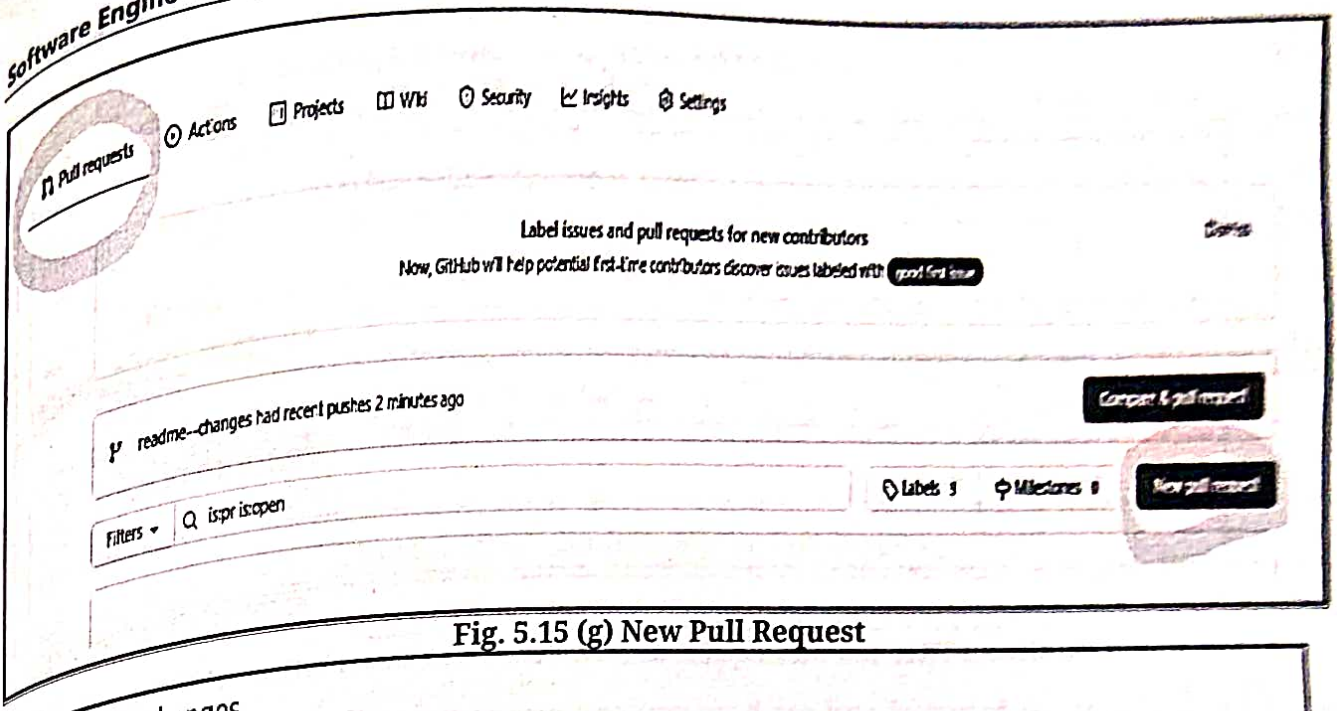



Fig. 5.15 (g) New Pull Request

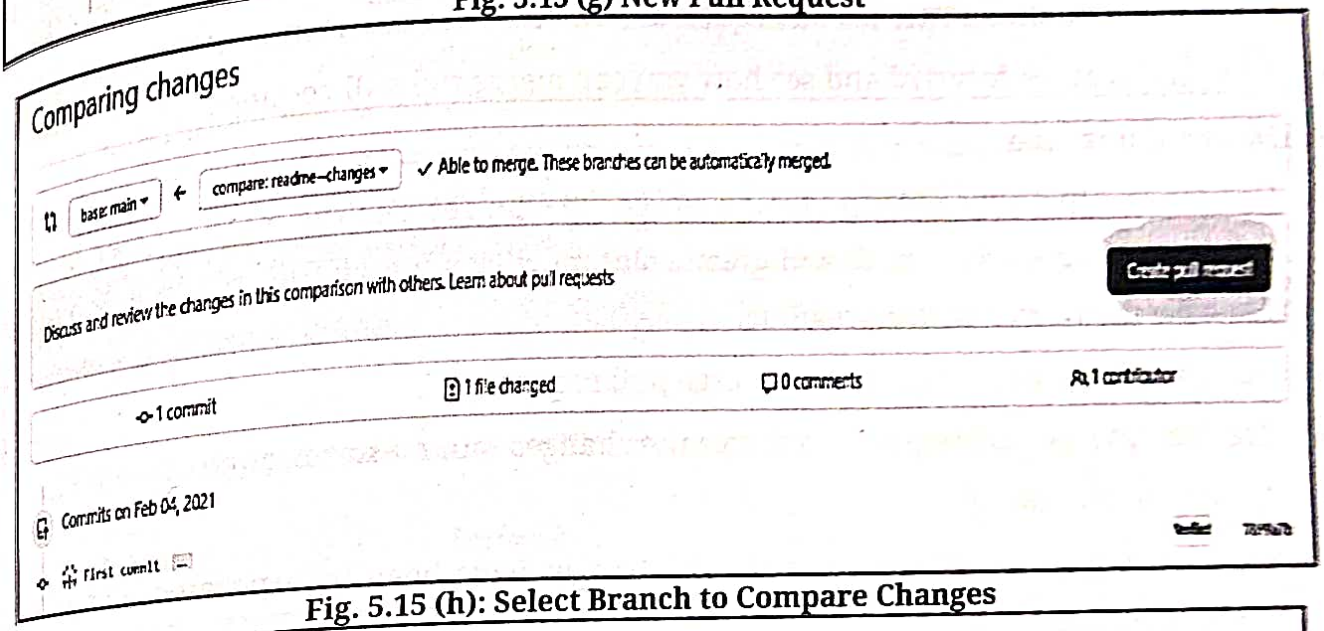



Fig. 5.15 (h): Select Branch to Compare Changes

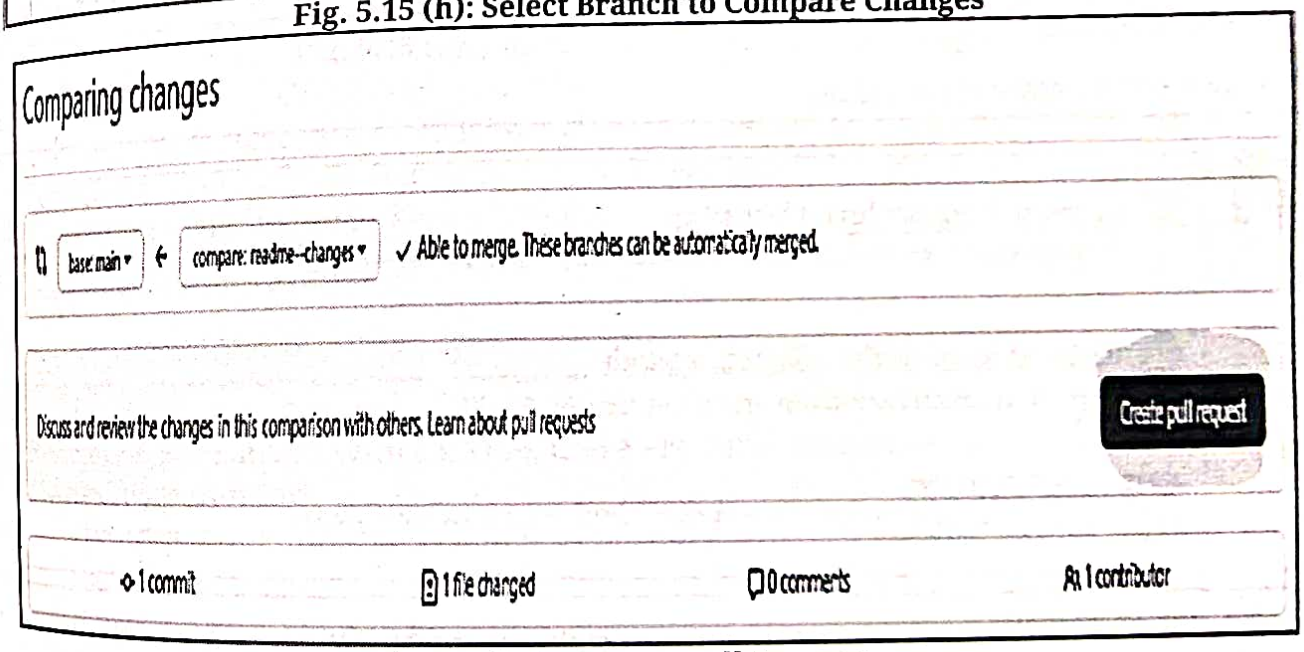



Fig. 5.15 (i): Create Pull Request

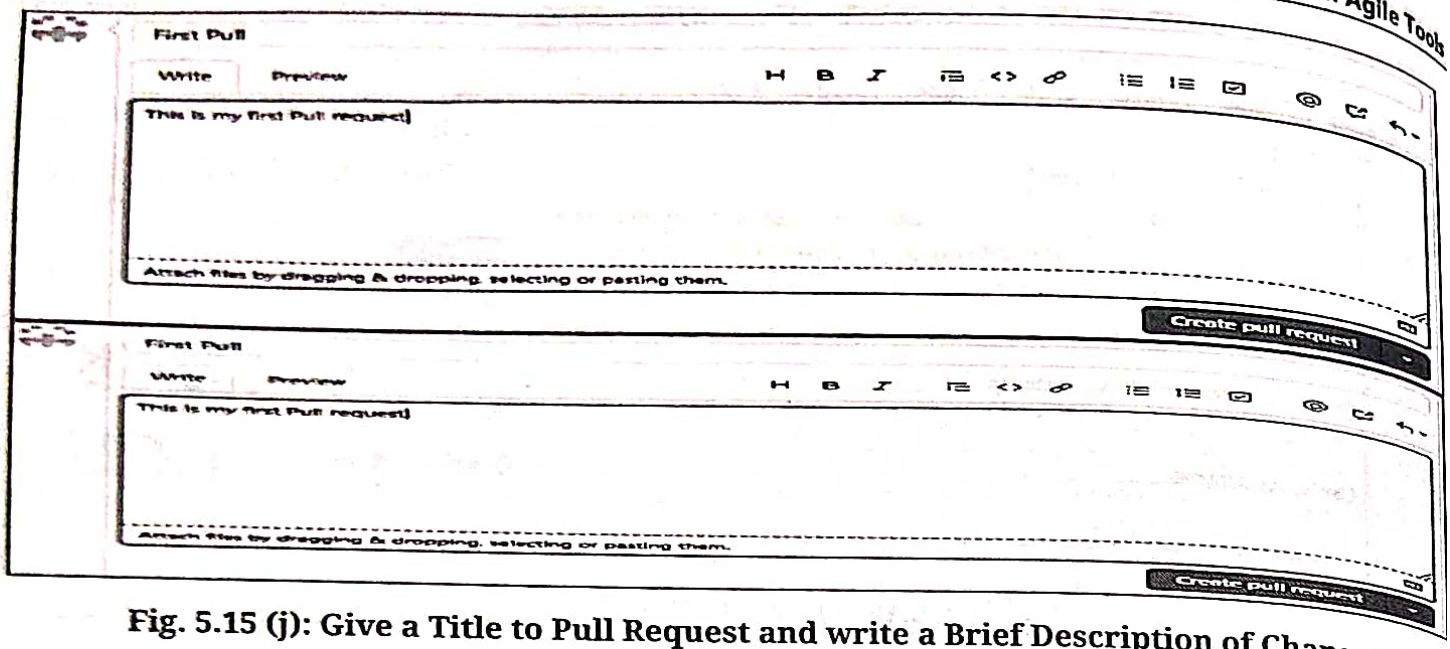



**Fig. 5.15 (j): Give a Title to Pull Request and write a Brief Description of Changes**

- Next, let us move forward and see how you can merge the pull request.

3. **Merge Command:**

- Here comes the last command which merge the changes into the main master branc We saw the changes in pink and green color, now let's merge the "readme - change file with the master branch/readme.
- Go through the following steps to merge pull request:
- Click on "Merge pull request" to merge the changes into master branch.
- Click "Confirm merge".
- You can delete the branch once all the changes have been incorporated and if the are no conflicts.
- Refer to the below screenshots.

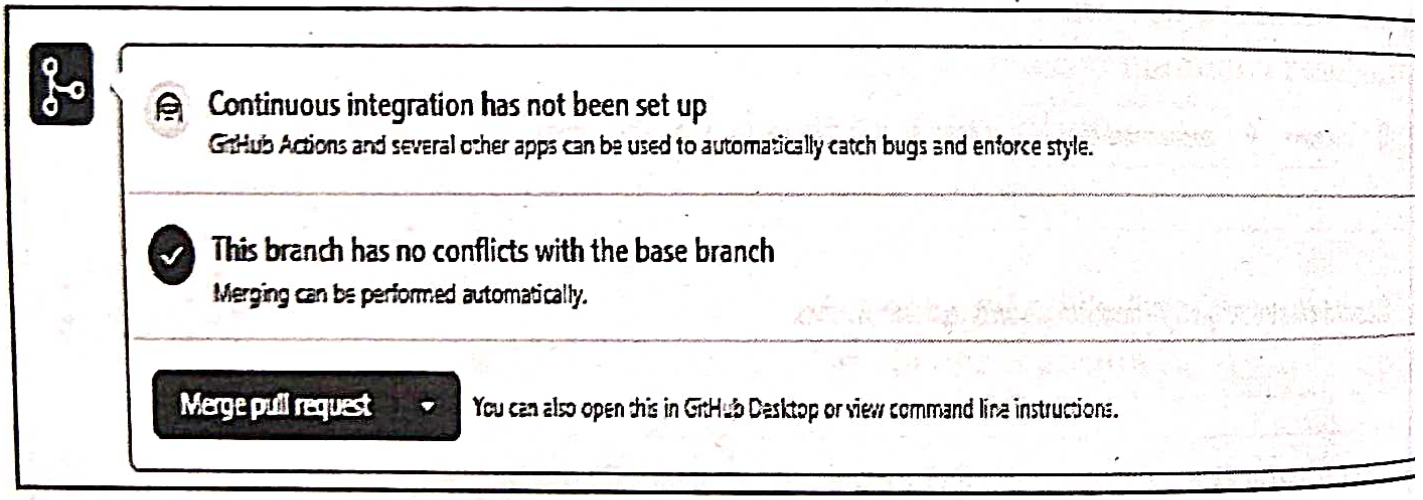



**Fig. 5.15 (k): Merge Pull Request**

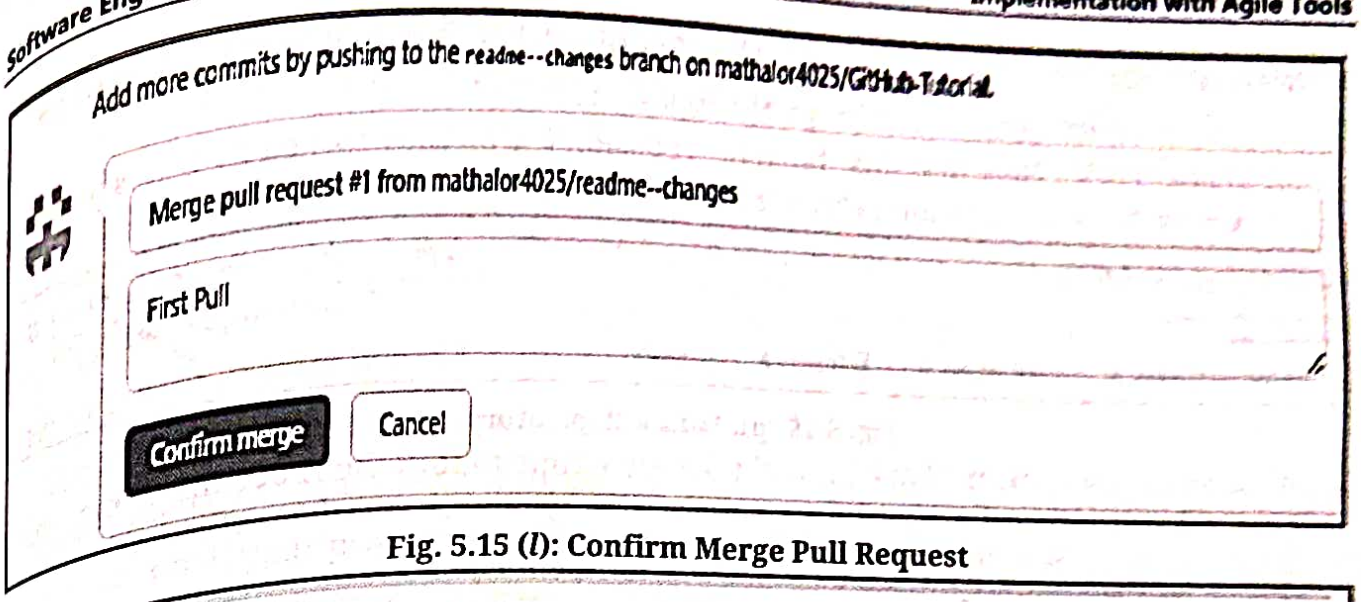



**Fig. 5.15 (*l*): Confirm Merge Pull Request**

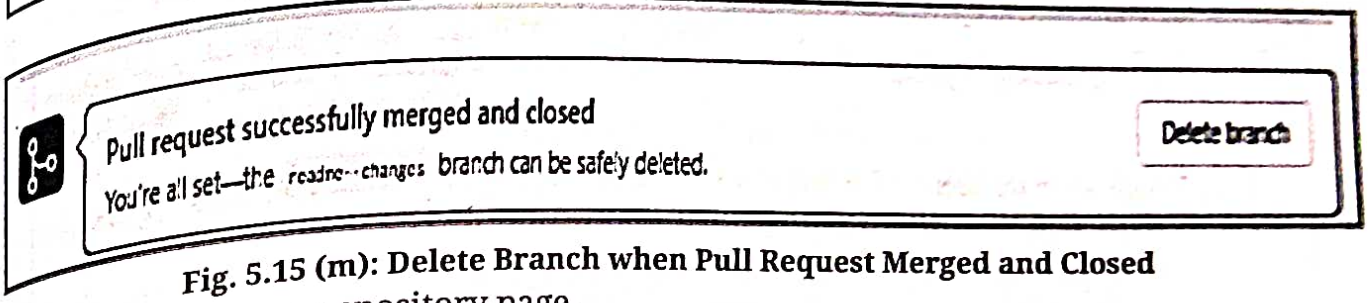



**Fig. 5.15 (m): Delete Branch when Pull Request Merged and Closed**

- Confirm merge on repository page.

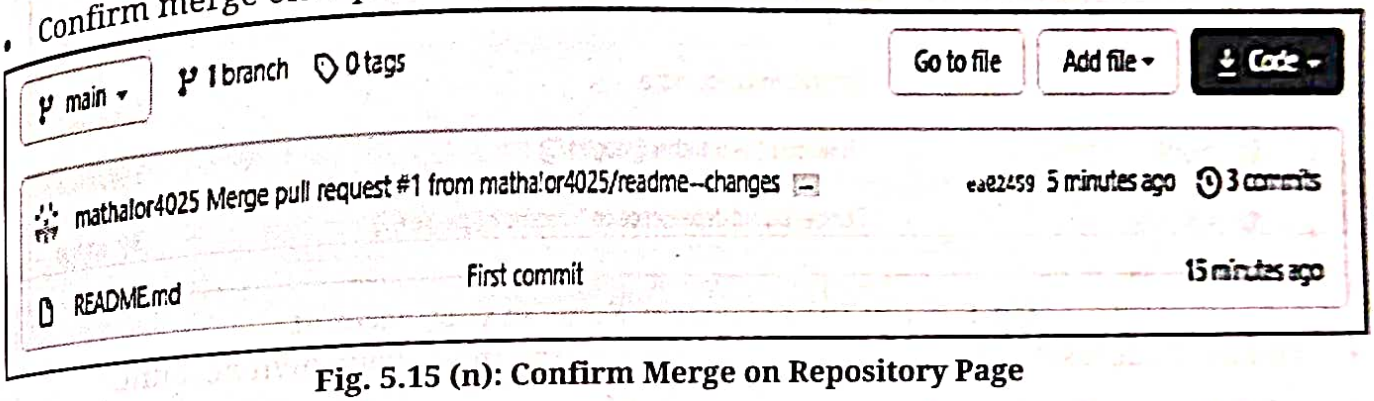



**Fig. 5.15 (n): Confirm Merge on Repository Page**

4. **Forking:**

- Suppose, you need some code which is present in a public repository, under your repository and GitHub account. For this, we need to fork a repository. fork option available on right side as shown below:

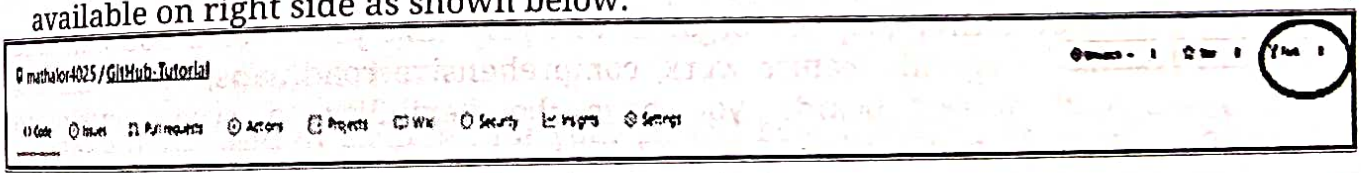



**Fig. 5.15 (o): Fork Option**

- Before we get started with forking, there are some important points which you should always keep in mind:
  - Changes done to the original repository will be reflected back to the forked repository.
  - If you make a change in forked repository, it will not be reflected to the original repository until and unless you have made a pull request.

- Now let's see how you can want to fork a repository. For that, follow the below steps:
  - Go to Explore and search for public repositories.
  - Click "Fork". Note that this "tangent" repository is already forked 27 times and it is under "google" account. Refer the below image for better understanding.

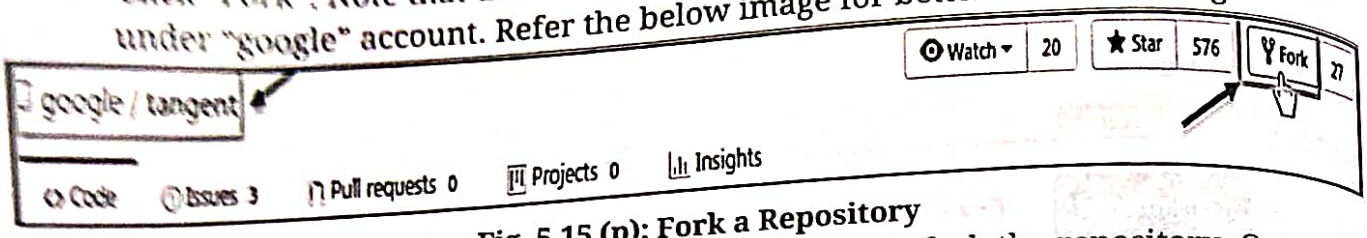



**Fig. 5.15 (p): Fork a Repository**

- As soon as you click on "Fork", it will take some time to fork the repository. Once done you will notice that the repository name is under your account. For reference, you can have a look at the below screenshot.

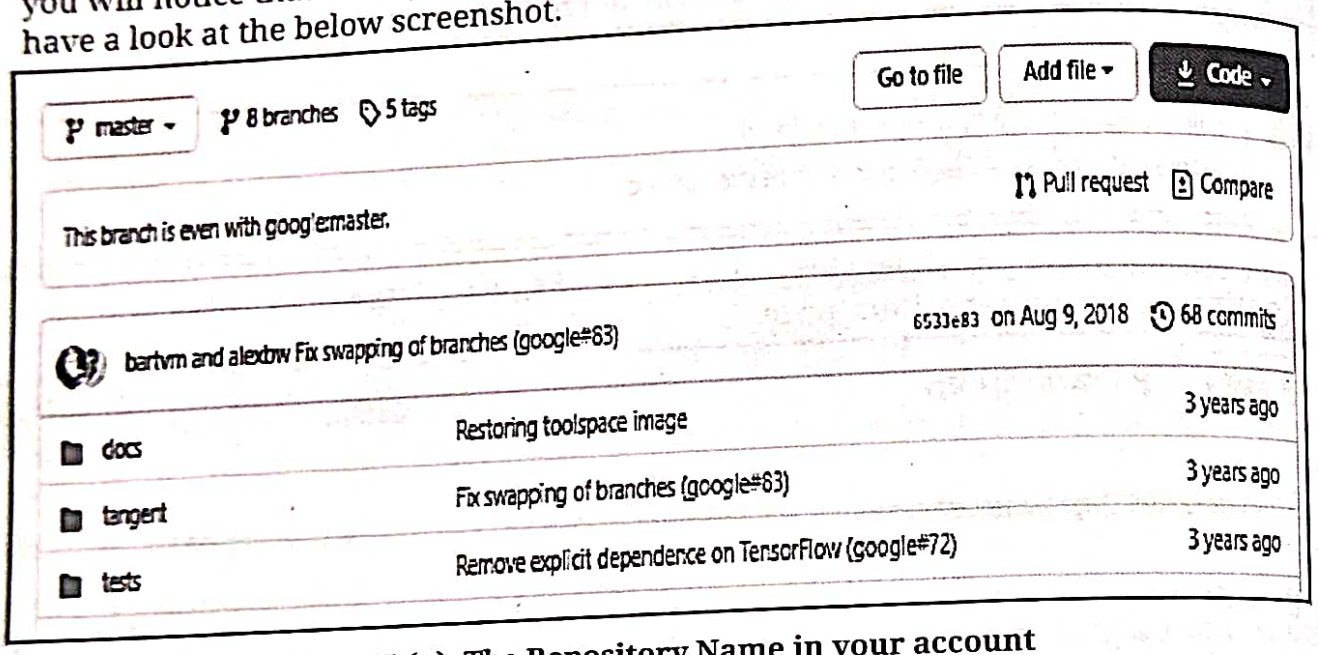



**Fig. 5.15 (q): The Repository Name in your account**

- You have successfully forked an existing repository under your own account.

## 5.3.1 Create Project Using Kanban

- GitHub Project lets you create multiple Kanban-style boards within a repository. Ea board can have custom stages.
- Project boards on GitHub help you organize and prioritize your work. You can crea project boards for specific feature work, comprehensive roadmaps, or even relea checklists. With project boards, you have the flexibility to create customiz workflows that suit your needs.

**Types of Project boards:**

- Following types of project boards can be created:
  1. User-owned project boards can contain issues and pull requests from any perso repository.
  2. Organization-wide project boards can contain issues and pull requests from a repository that belongs to an organization. You can link up to twenty-f

repositories to your organization or user-owned project board. Linking repositories makes it easier to add issues and pull requests from those repositories to your project board using Add cards or from the issue or pull requests sidebar.

3. Repository project boards are scoped to issues and pull requests within a single repository. They can also include notes that reference issues and pull requests in other repositories.

**How to create Project board?**

- Use following steps to create a Project board in GitHub:

1. In the top right corner of GitHub, click your profile photo, and then click your profile.
2. On the top of your profile page, in the main navigation, click Projects.

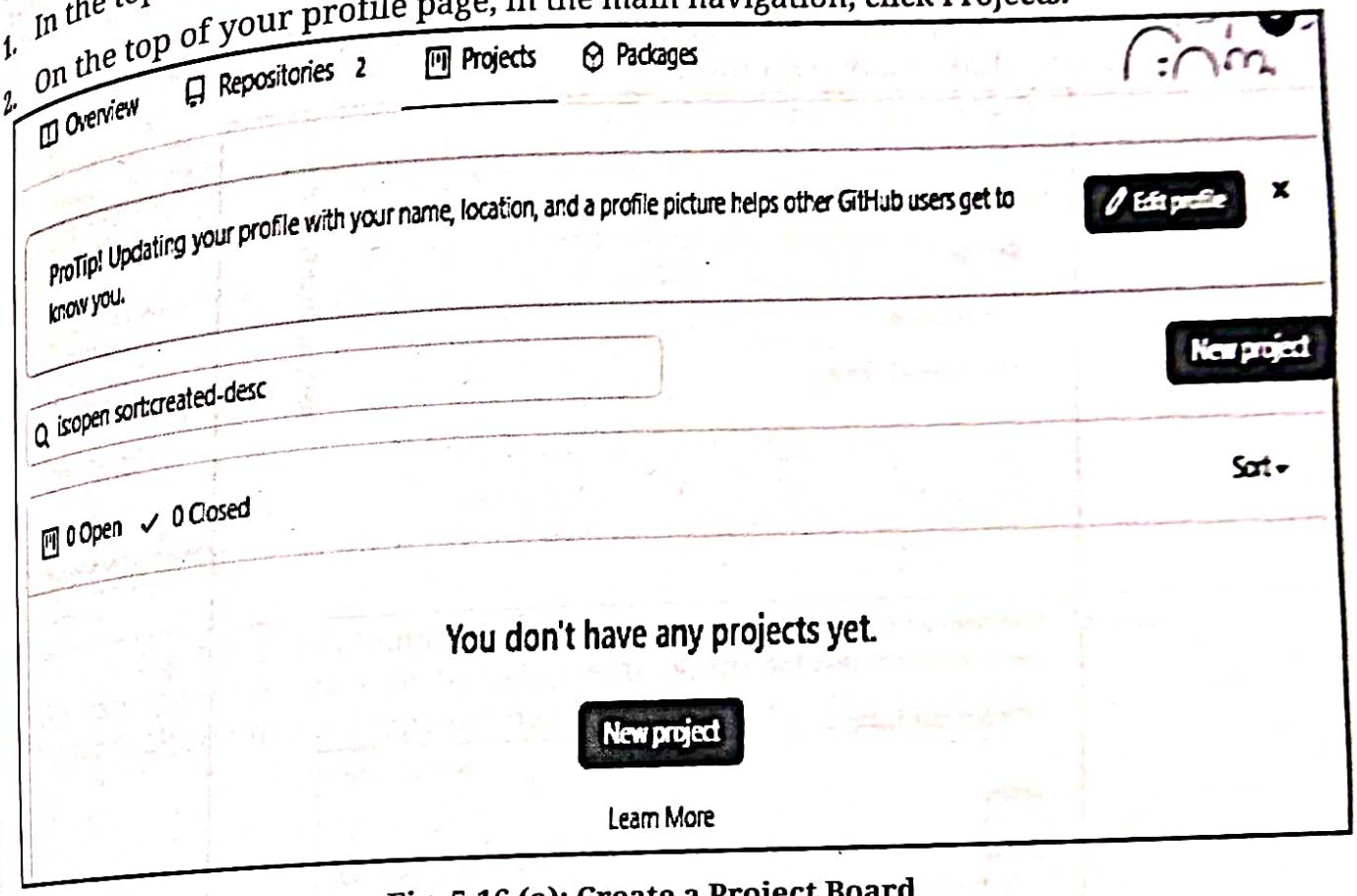



**Fig. 5.16 (a): Create a Project Board**

3. Click on "New project" button.
4. Type a name and description for your project board. Optionally, to add a template to your project board, use the Template: drop-down menu and click a template. You can use templates to quickly set up a new project board. When you use a template to create a project board, your new board will include columns as well as cards with tips for using project boards. You can also choose a template with automation already configured.

**Table 5.2: Project Board Templates**

| Template | Description |
|---|---|
| Basic Kanban | Track your tasks with To do, In progress, and Done columns. |
| Automated Kanban | Cards automatically move between To do, In progress, and Done columns. |
| Automated Kanban with review | Cards automatically move between To do, In progress, and Done columns, with additional triggers for pull request review status. |
| Bug triage | Triage and prioritize bugs with To do, High priority, Low priority, and Closed columns. |

- Under "Visibility", choose to make your project board public or private. Optionally, under Linked repositories, search for a repository you'd like to link to your project board.
- Here , We have taken name as "SPM Project1" and Template as "Basic Kanban". At this stage we have not linked any respositories.

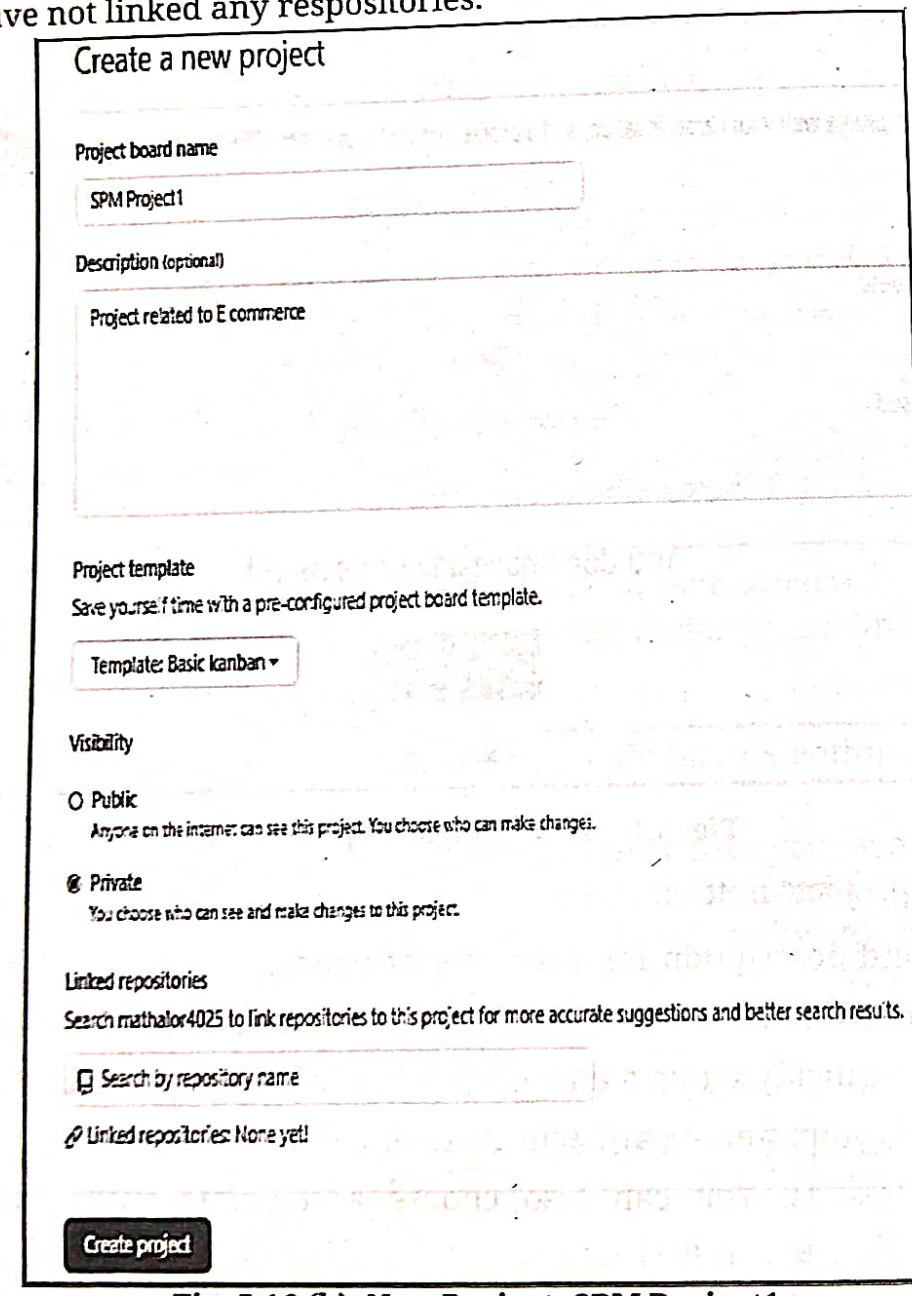



**Fig. 5.16 (b): New Project: SPM Project1**

5. Click **Create project.**
6. On your new project board, you will see three columns named as "To -do", "In progress", "Done" as shown in Fig. 5.16 (c).

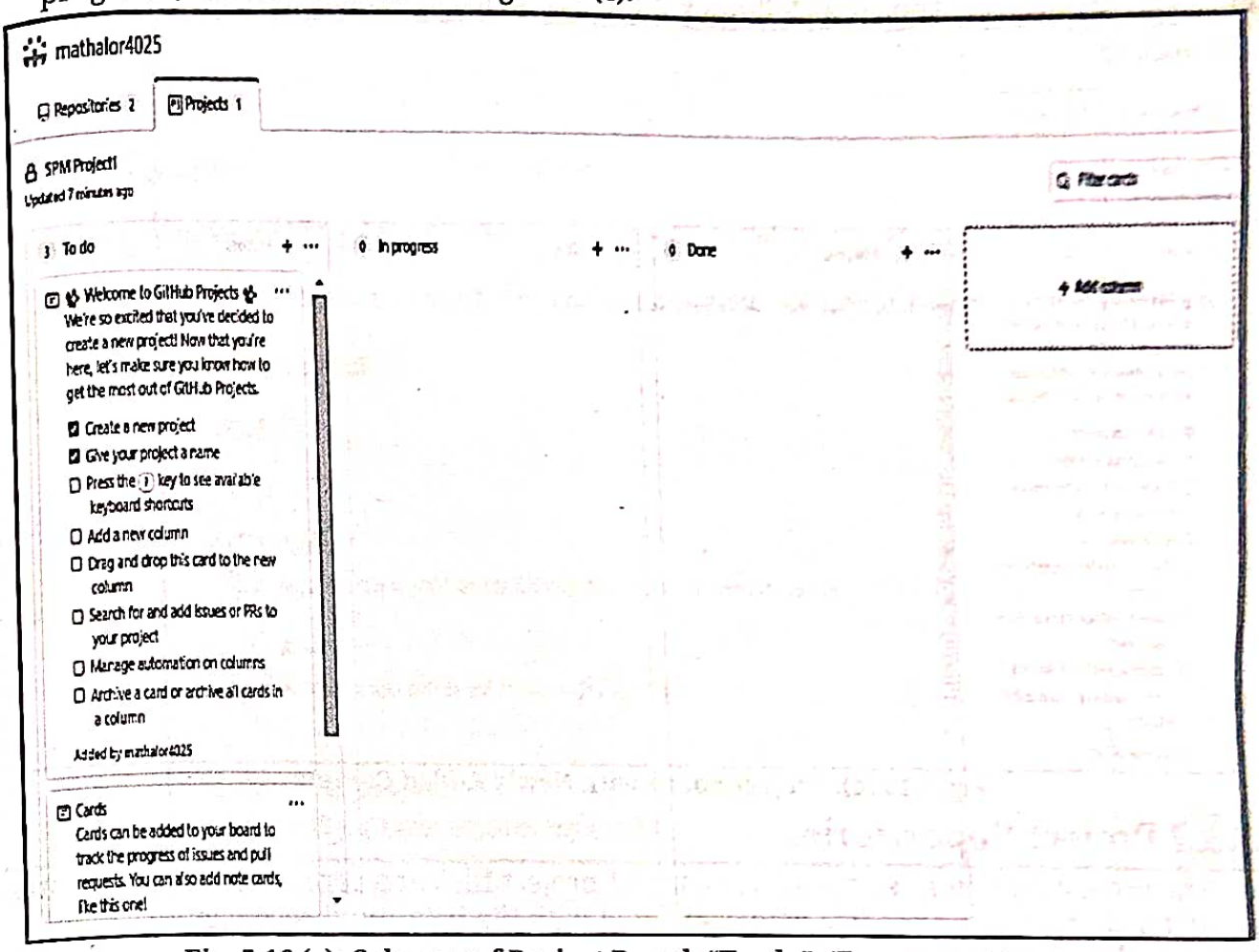



**Fig. 5.16 (c): Columns of Project Board: "To -do", "In progress", "Done"**

7. You can add more columns by using "Add column".
8. Under "Column name", type the name of the column you want to create.

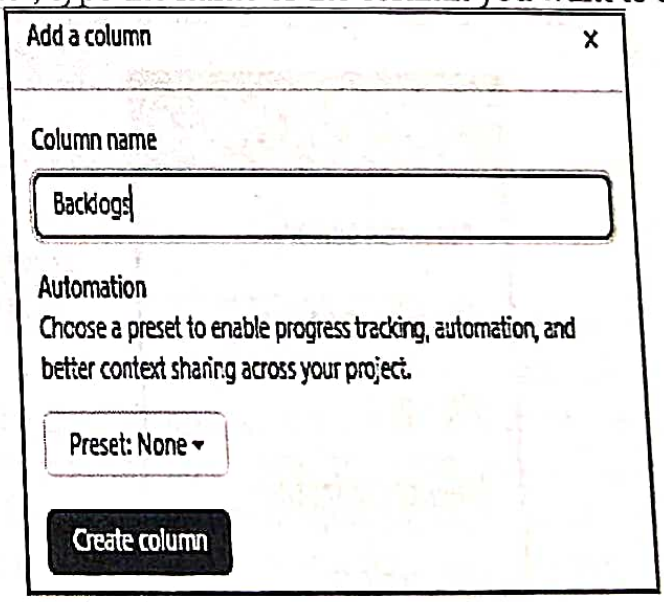



**Fig. 5.16 (d): Add column**

9. Optionally, under "Automation", select the workflow automations you want to configure for the column.
10. Click Create column.
11. Project board will appear as follows with newly added columns.

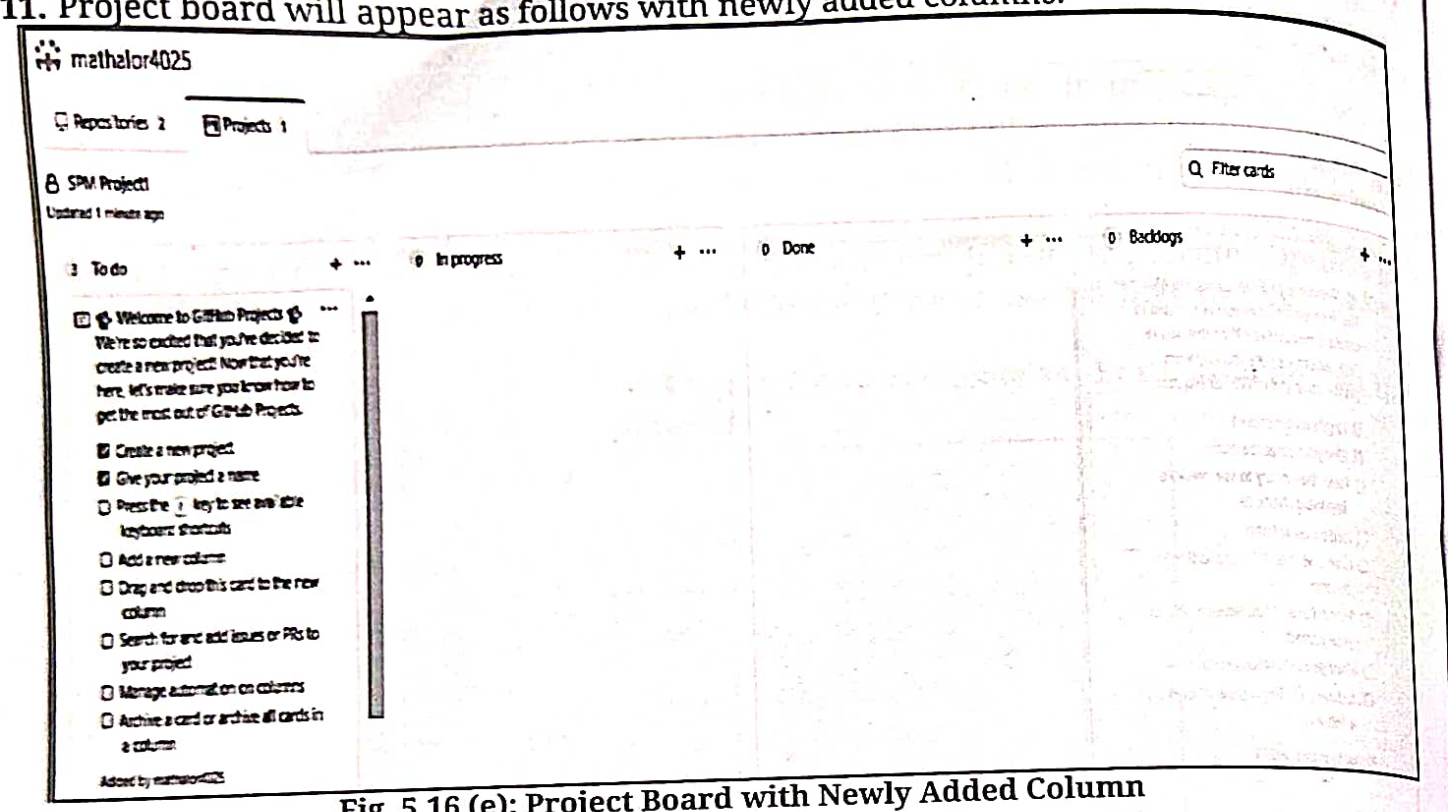



Fig. 5.16 (e): Project Board with Newly Added Column

## 5.3.2 Project Repositories

- The project repository stores all versions of project files and directories. It also stores all the derived data and Meta data associated with the files and directories.

**Create a New Repository:**

- You can create a new repository on your personal account or any organization where you have sufficient permissions.
  1. In the upper-right corner of any page, use the drop-down menu, and select New repository option.

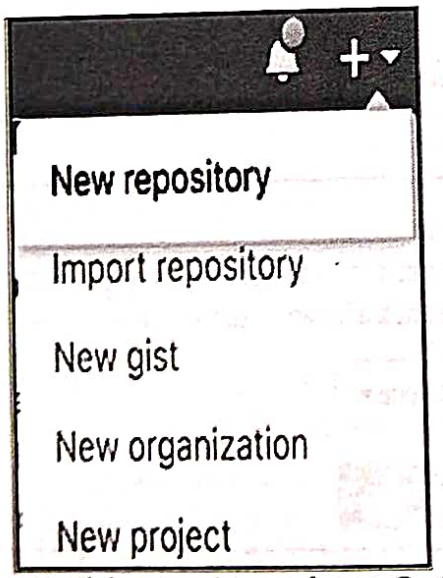



Fig. 5.17 (a): New Repository Option

2. Type a name for your repository, and an optional description. Choose a repository visibility.

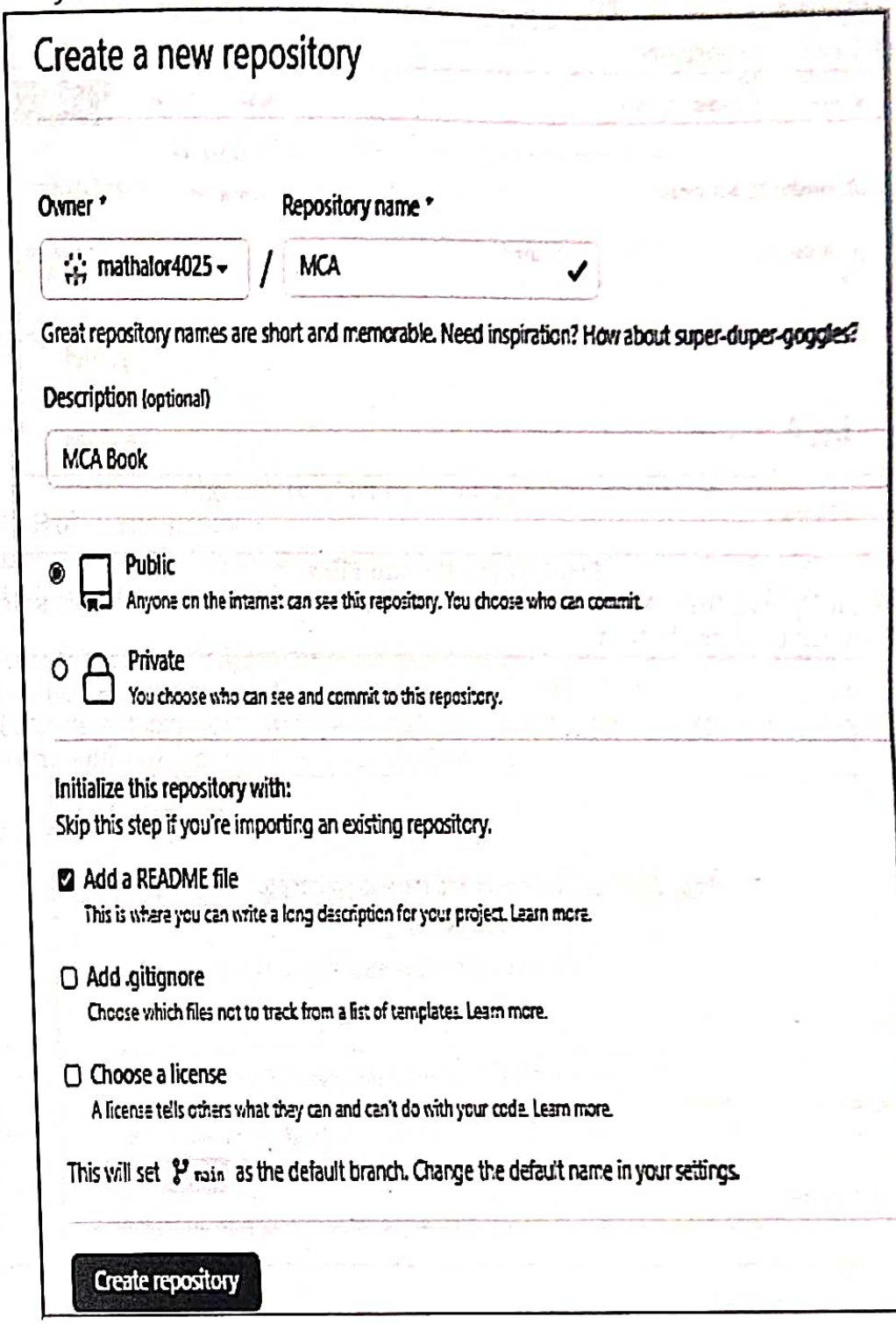



**Fig. 5.17 (b): Create Repository**

3. If you're not using a template, there are a number of optional items you can pre-populate your repository with. If you're importing an existing repository to GitHub, don't choose any of these options, as you may introduce a merge conflict. You can add or create new files using the user interface or choose to add new files using the command line later.

- You can create a README, which is a document describing your project.
- You can create a .gitignore file, which is a set of ignore rules.
- You can choose to add a software license for your project.

4. Click Create repository.

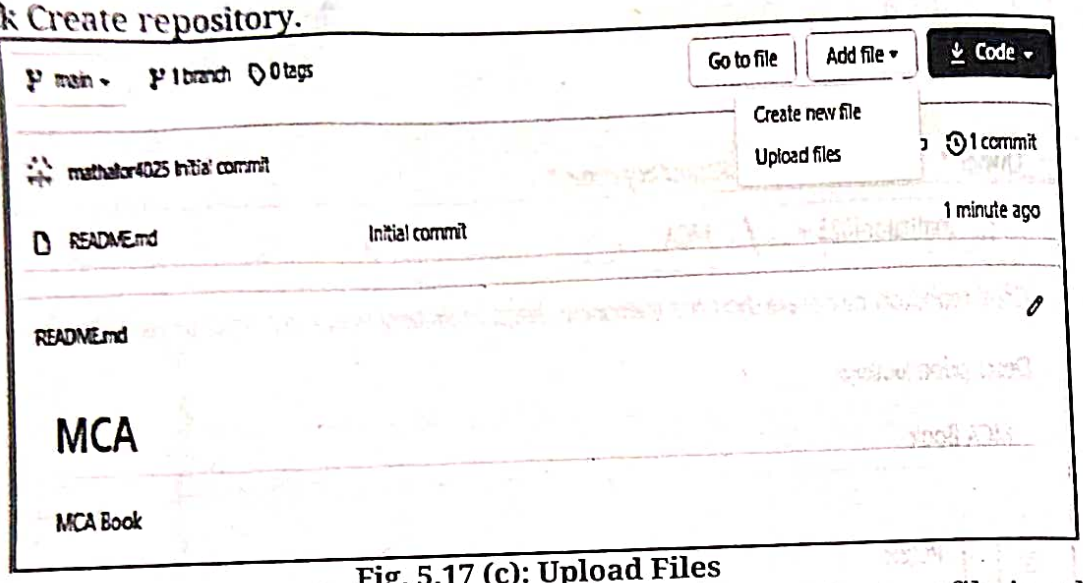



**Fig. 5.17 (c): Upload Files**

5. Click on Upload files. And you can upload some files on it. Once file is selected click on Commit changes button.

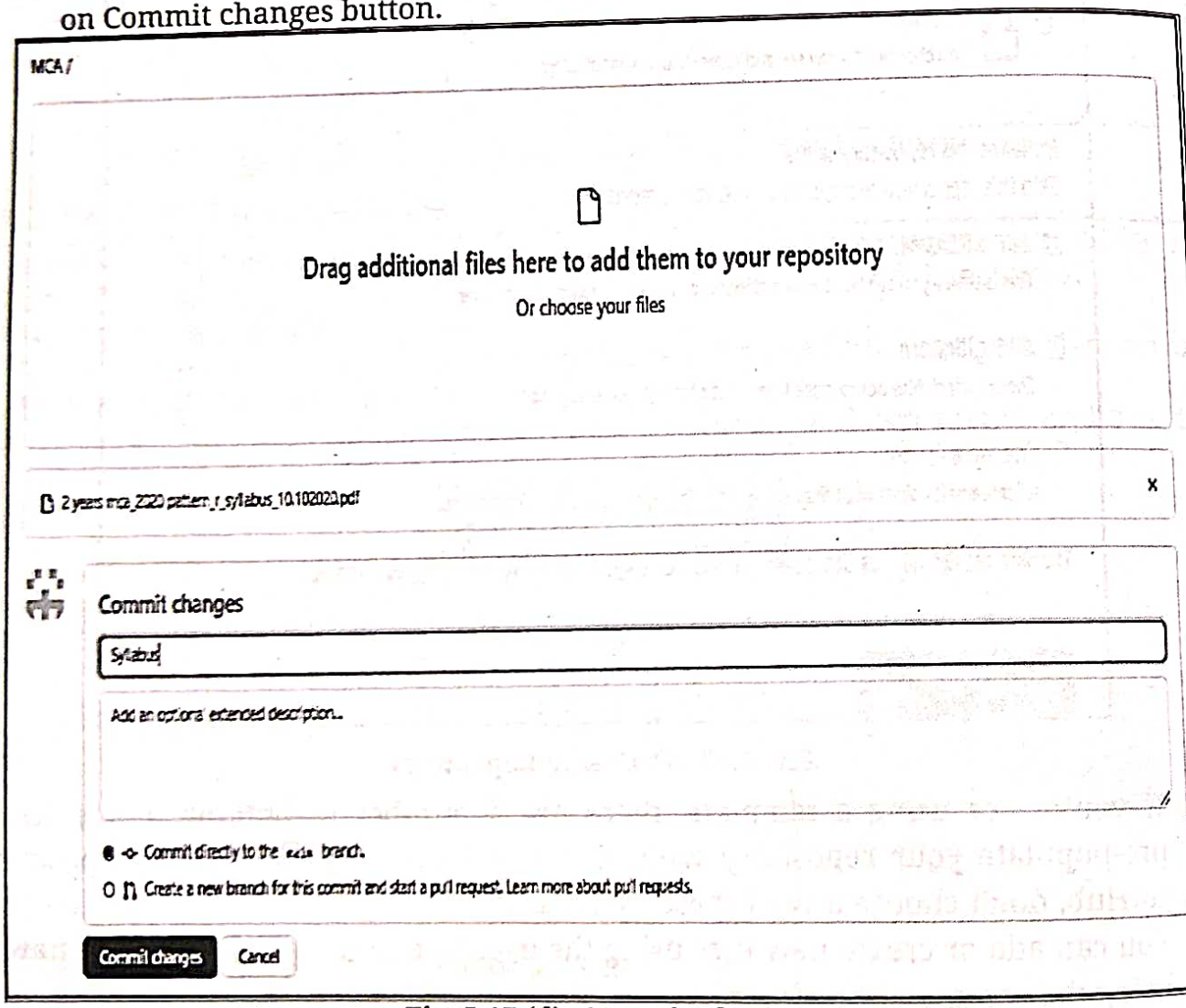



**Fig. 5.17 (d): Commit Changes**

- Now you will see that all of our files uploaded in our GitHub.

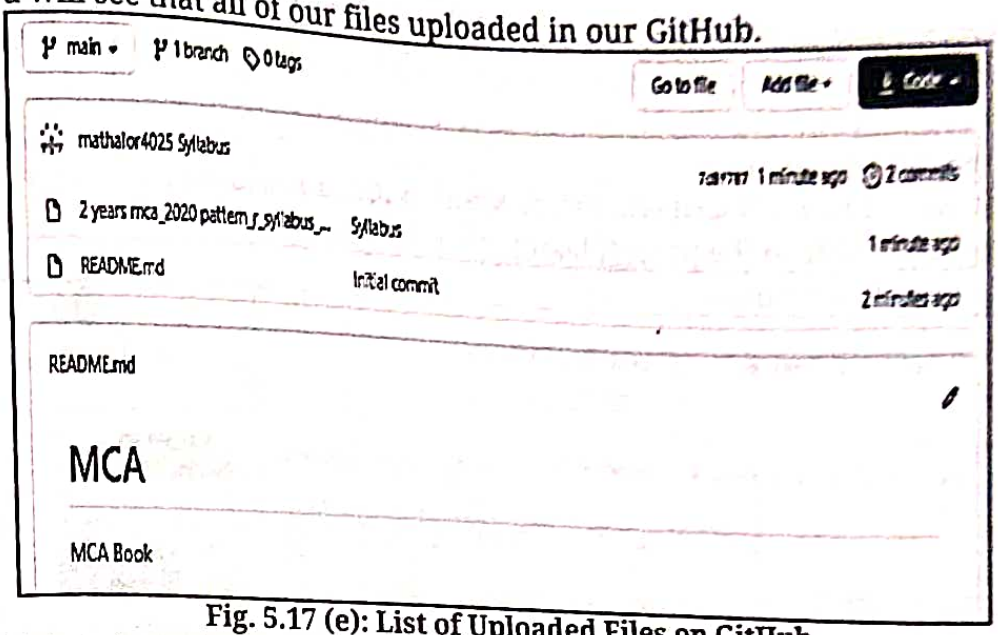



**Fig. 5.17 (e): List of Uploaded Files on GitHub**

**Hosting GitHub Repository:**

- GitHub Pages are designed to host your personal, organization, or project pages from a GitHub repository. As we already have the repository, we only have to activate our pages.
- Go to settings and scroll down to GitHub Pages section.
- If your site URL (https://mathalor4025.github.io/MCA/) is not published as shown below then First select branch: main and select: /root folder and click on Save option. After some time you will get the site URL as shown below:

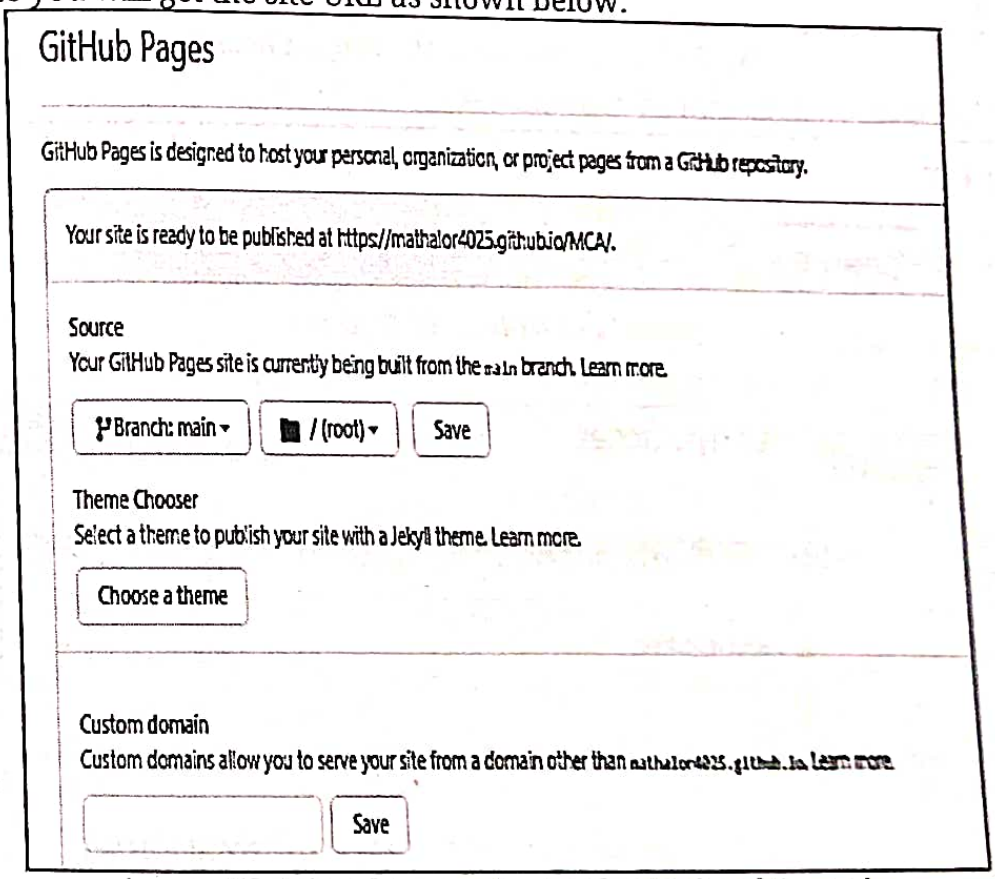



**Fig. 5.17 (f): GitHub Pages to Host from GitHub Repository**

- Now we are done and our project can be accessed worldwide.

**Linking a Repository to a Project Board:**

- As we have created a project board (SPM project1) and repository (MCA). You can link a repository to your organization's or user account's project board. Now it's time to link repository with project.

1. Navigate to the Project board where you want to link a repository.
2. On the top-right side of the project board, click Menu.
3. Click ... , then click Settings.

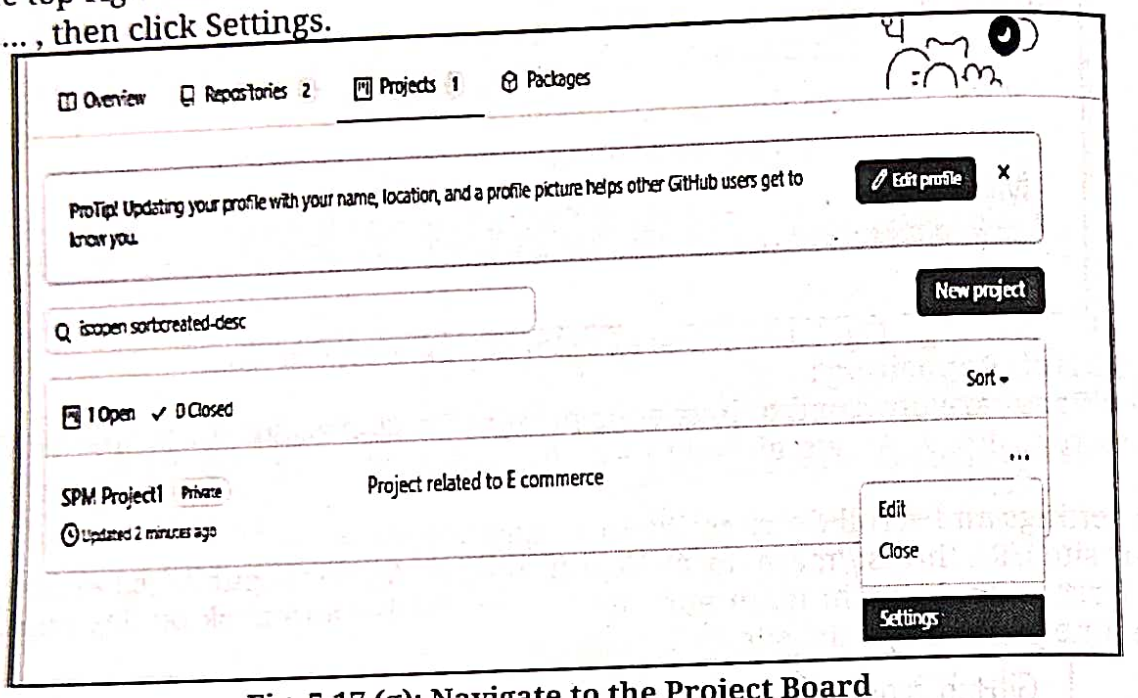



**Fig. 5.17 (g): Navigate to the Project Board**

- In the left sidebar, click Linked repositories.

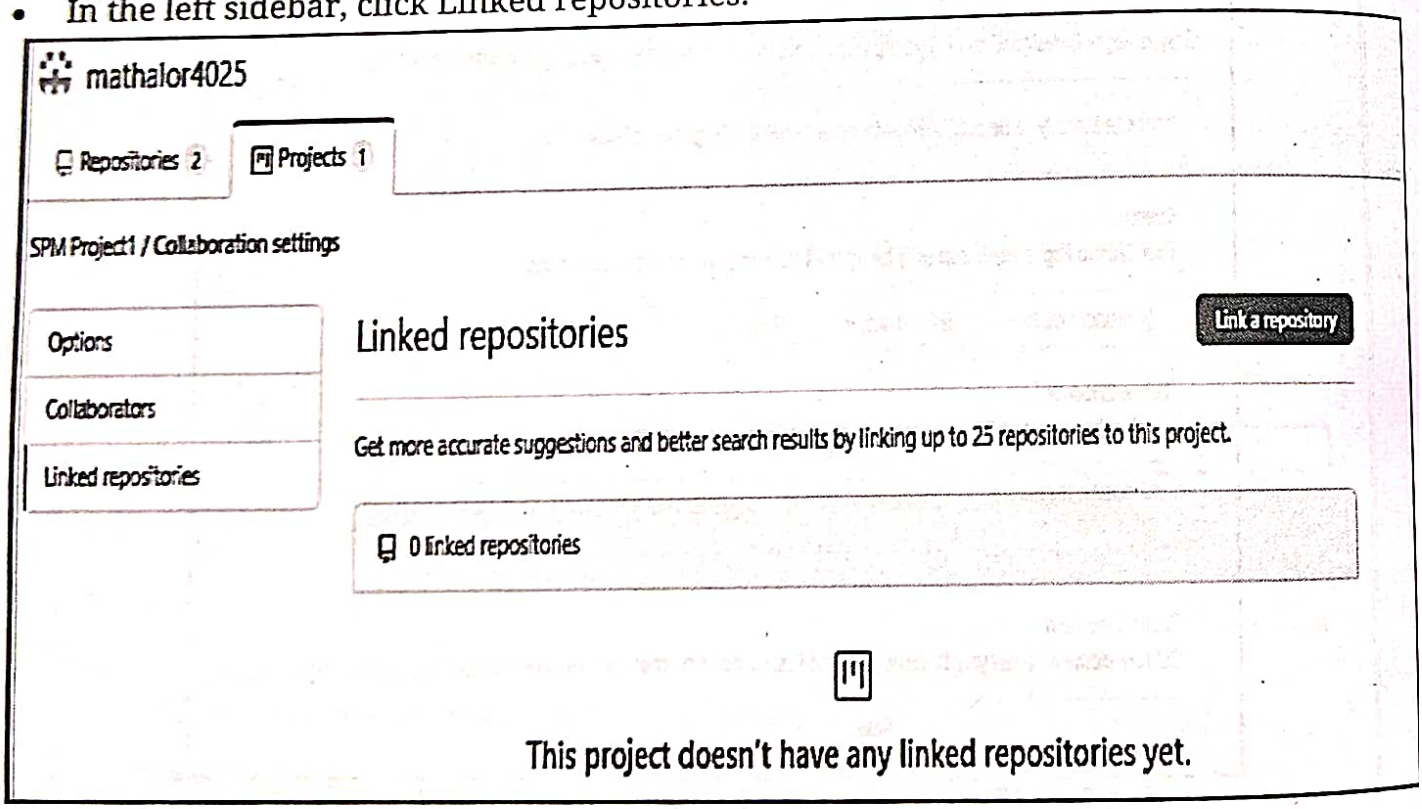



**Fig. 5.17 (h): Search for the Repository you'd like to Link**

- Click Link a repository. Search for the repository you'd like to link.

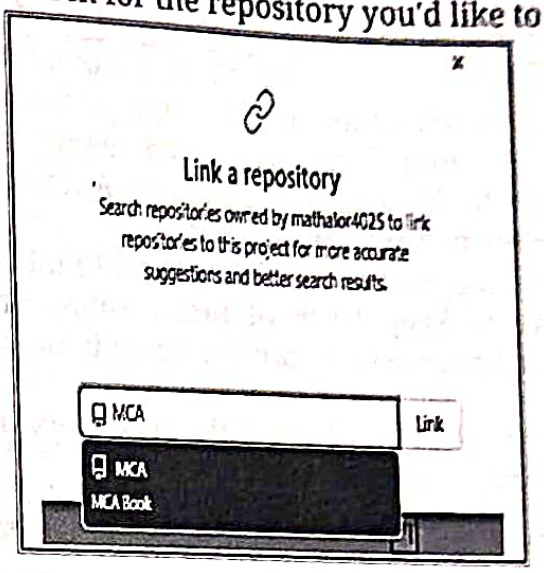



**Fig. 5.17 (i): Select One of Existing Repository**

- Select one of existing repository created by you or someone else and click on Link. If the process is successful you will see next fig.

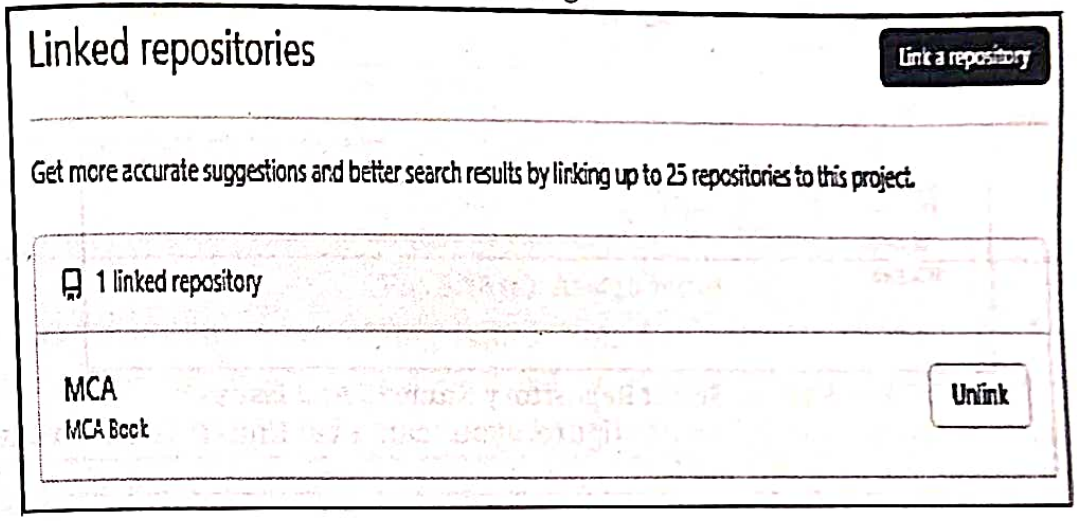



**Fig. 5.17 (j): Linked Repository**

- Once you return to your profile you will see following details with your project.

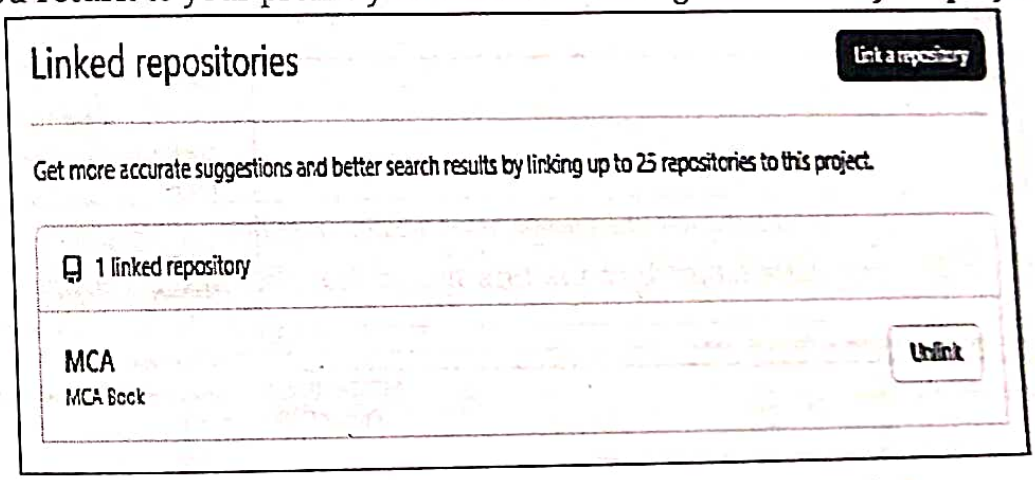



**Fig. 5.17 (k): Details of Linked Repository with Project**

## 5.3.3 Continuous Integration

- Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. It is a primary DevOps best practice, allowing developers to frequently merge code changes into a central repository where builds and tests then run. Automated tools are used to assert the new code's correctness before integration.
- Once your project is created, you can add issues and pull requests to a project board. Issues are a great way to keep track of tasks, enhancements, and bugs for your projects. People with read permissions can create an issue in a repository where issues are enabled.

1. On GitHub, navigate to the main page of the repository. Here MCA is repository.
2. Under your repository name, click Issues.

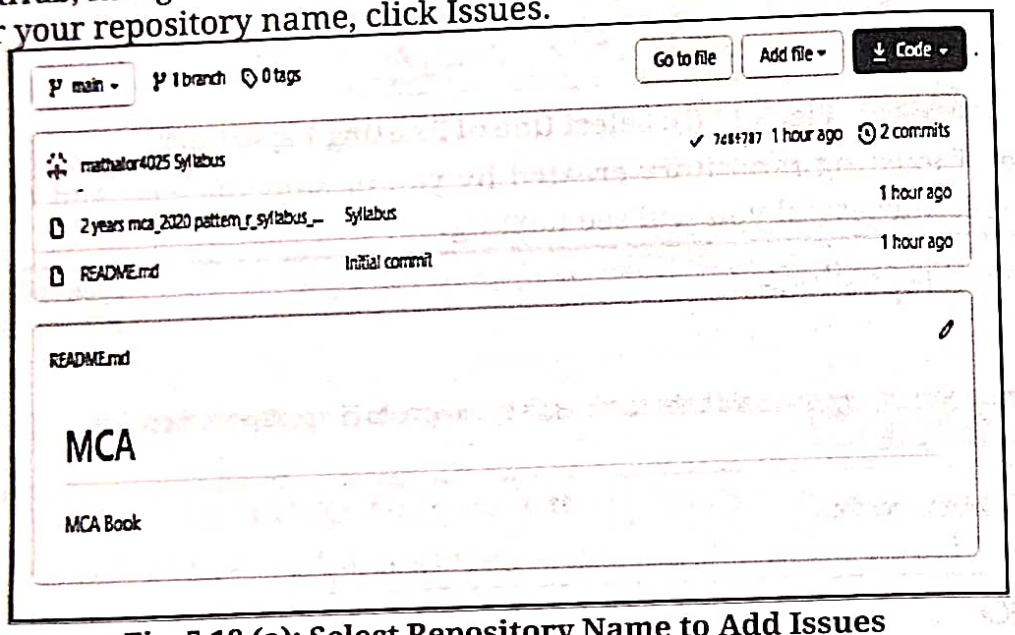



**Fig. 5.18 (a): Select Repository Name to Add Issues**

3. Click New issue. In following figure, you can see Unit-1 issue created with description "spelling mistake".

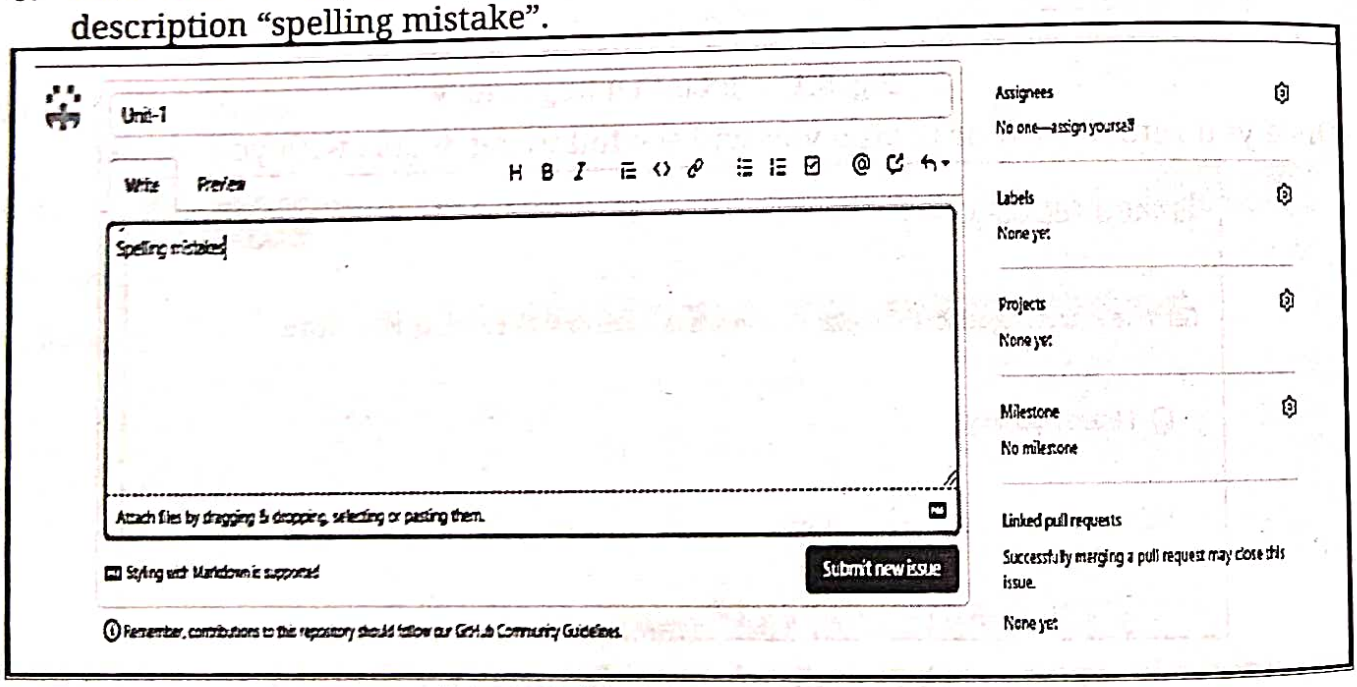



**Fig. 5.18 (b): Create New Issue**

4. If you are a project maintainer, you can assign the issue to someone, add it to a project board, associate it with a milestone, or apply a label then click on submit new issue.

- In following figure, assigned this issue to user and associate this issue with SPM Project 1.

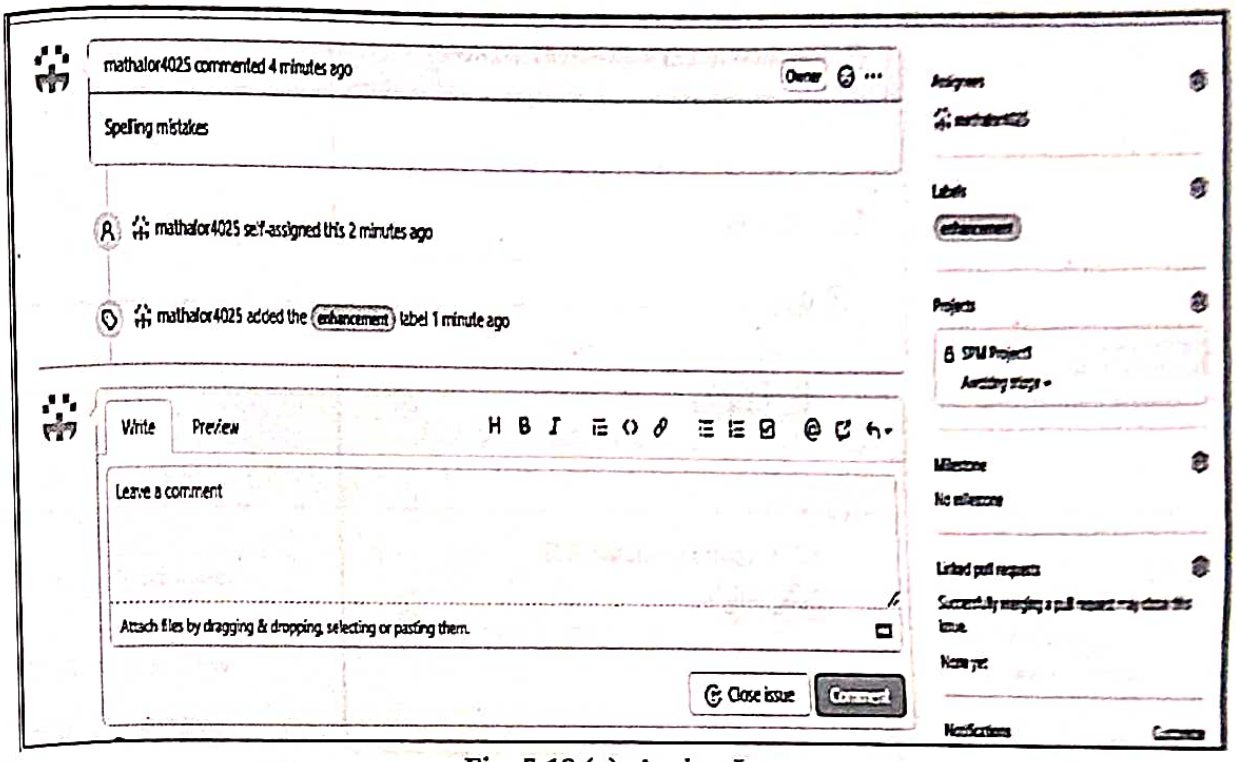



**Fig. 5.18 (c): Assign Issue**

1. Once you are done with adding issues. You can navigate to the main page of the repository for list of issues as shown below. Here under MCA repository, two issues named as Unit-1 and Unit-2 have created and both assigned to SPM Project1.

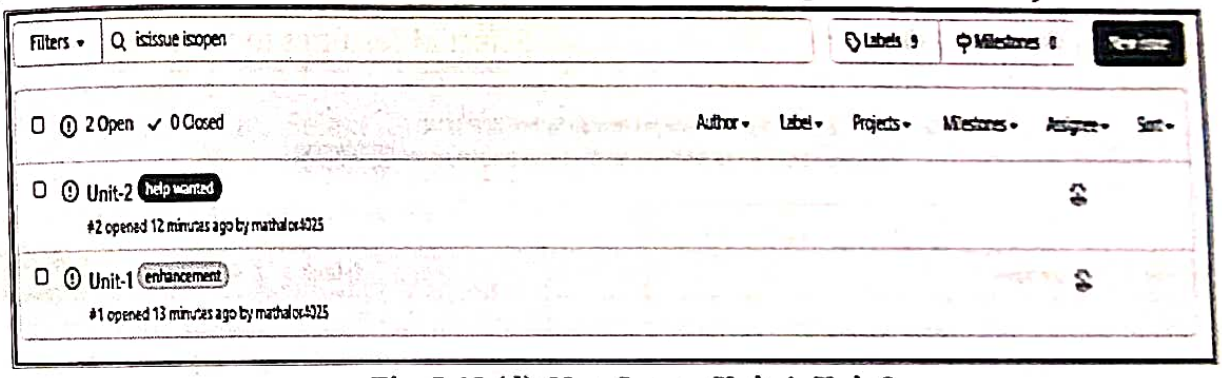



**Fig. 5.18 (d): New Issues Unit-1, Unit-2**

2. Now navigate to SPM Project1 board and see rightmost side, you will notice that two cards are get added into this project.


**Fig. 5.18 (e): Add Card**

3. When you explore + sign, you will get following details:

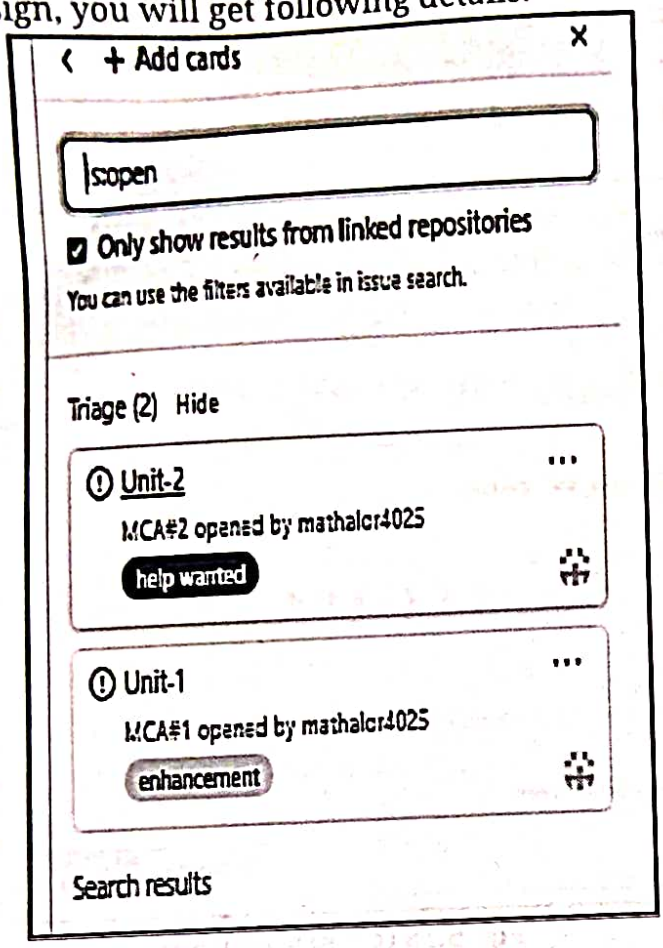



**Fig. 5.18 (f): Details of Issues**

- Using Milestones, you can group issues with certain goals or due dates using milestone option. To start, select Issues or Pull requests in your repository's navigation bar. Here, you will see a search bar at the top of the page. Select Milestones to the right of it.

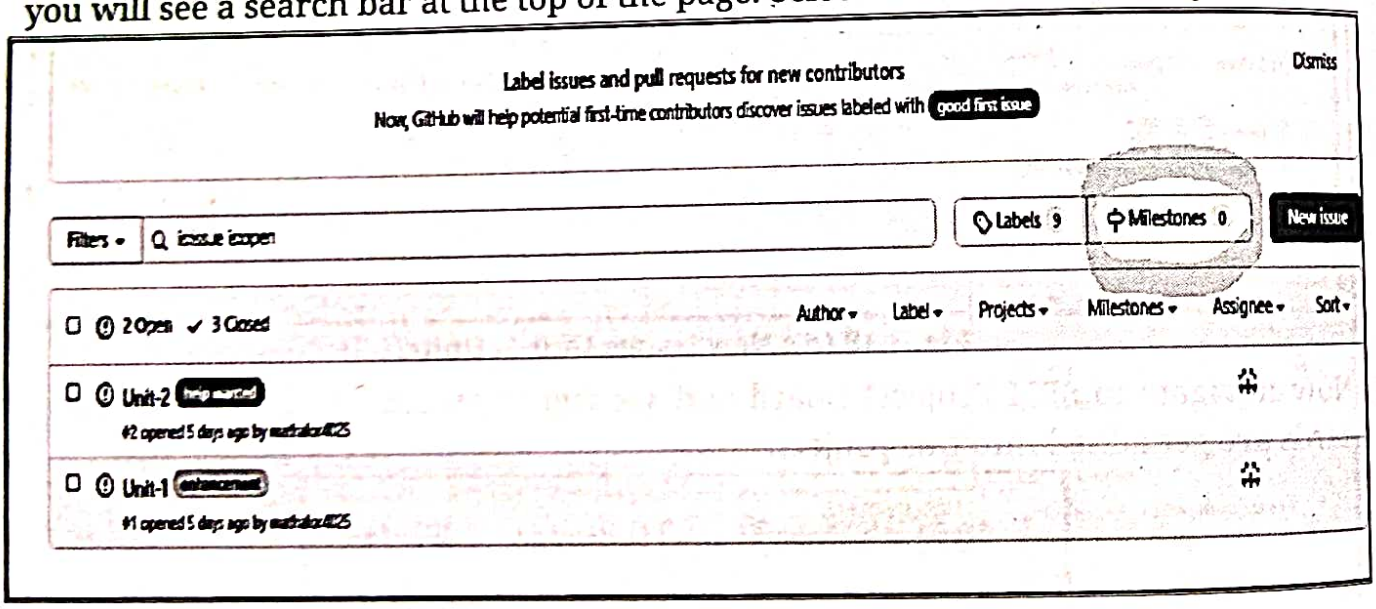



**Fig. 5.18 (g): Milestones Option**

- On next page, click "Create Milestone".

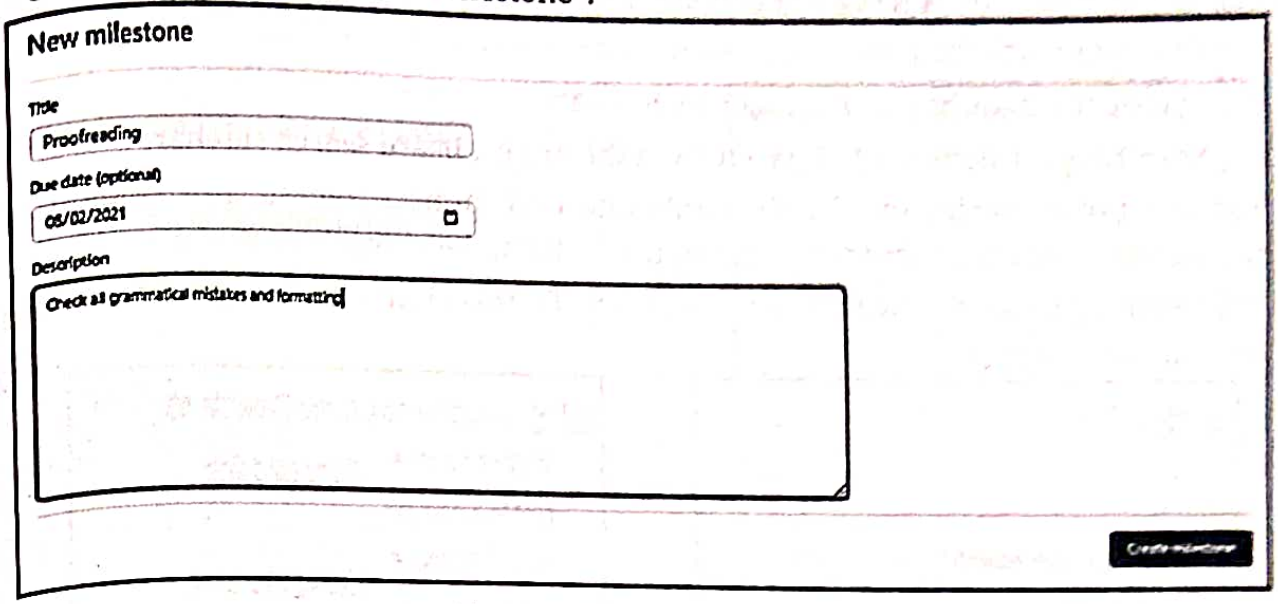



Fig. 5.18 (h): Create Milestone

- Click on Create milestone and confirm on Issue page as shown below:

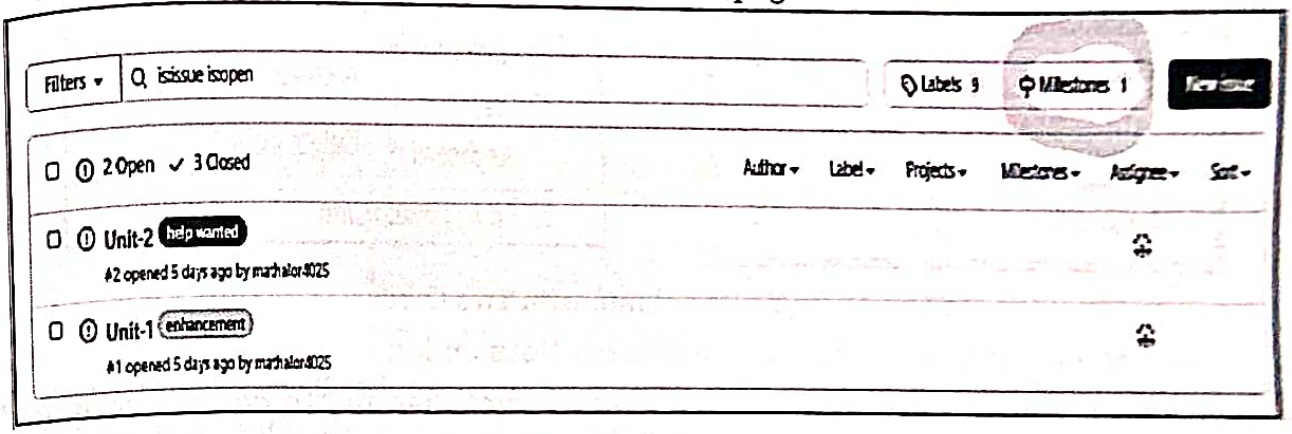



Fig. 5.18 (i): Issue Page

- Now you can group issues with certain goal and deadline. Select the issue and picking a milestone corresponding to it as shown below.

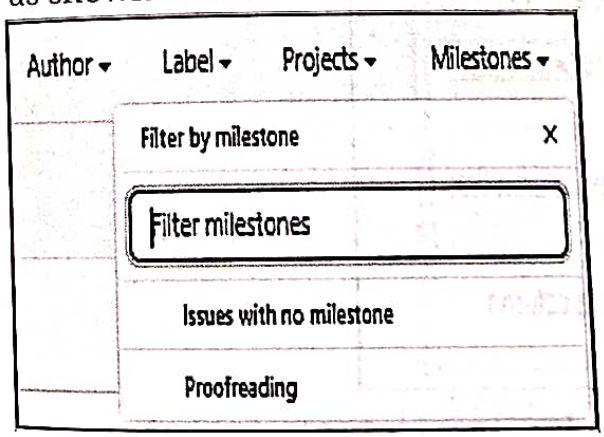



Fig. 5.18 (j): Group Issues by Milestones

- Once a milestone is assigned you, issue will look as follows:

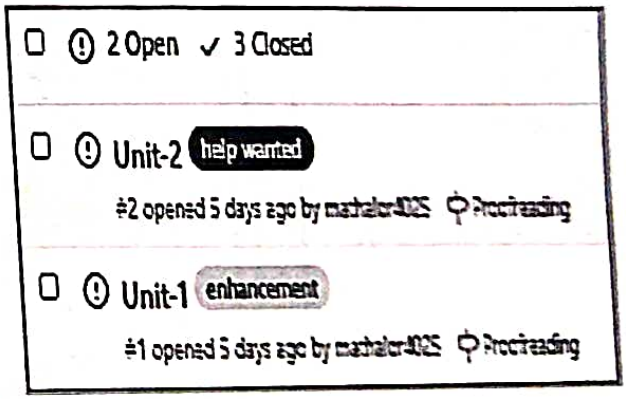



Fig. 5.18 (k): Issue with Assigned Milestones

4. You can add issue or pull request cards to your project board by:
   - Dragging cards from the Triage section in the sidebar.
   - Typing the issue or pull request URL in a card.
   - Searching for issues or pull requests in the project board search sidebar.
5. You can put a maximum of 2,500 cards into each project column. If a column has reached the maximum number of cards, no cards can be moved into that column.
6. Following figure shows two cards are added in **To-Do** column by dragging from Triage section in the sidebar.

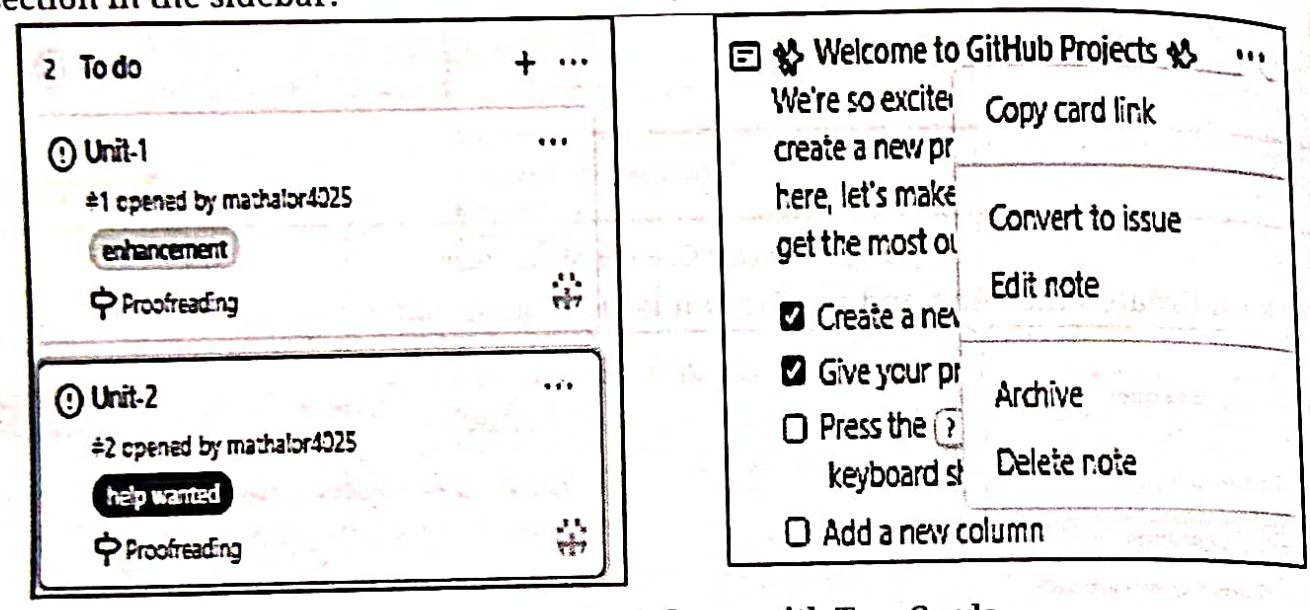



**Fig. 5.18 (l): To-Do Column with Two Cards**

7. Remaining notes you can delete by using **Delete Note** option.
8. If you want to avoid the dragging of individual issues then in "To do", "In progress", "Done" use "Manage automation" option as shown in Fig.

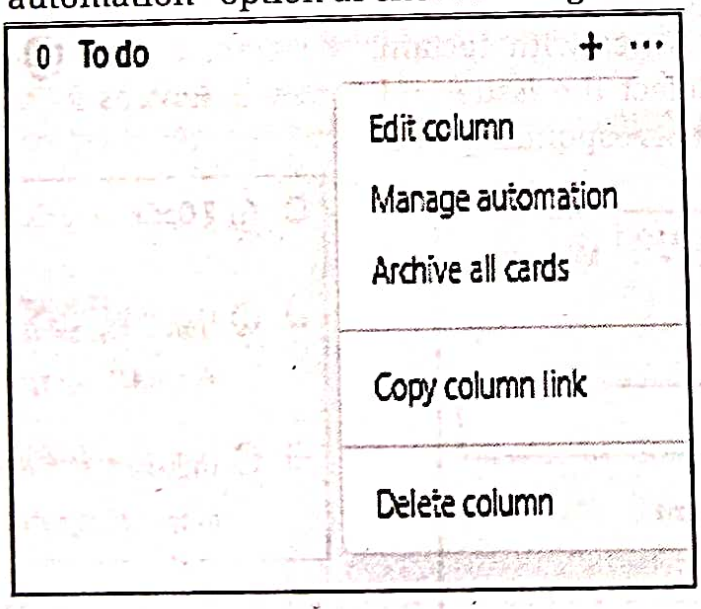



**Fig. 5.18 (m): Manage Automation to avoid dragging of individual option**

9. Using **Manage automation** option, you can set up automatic workflows to move issues and pull requests to a Project board column when a specified event occurs.
10. Click **Update automation** once options selected.

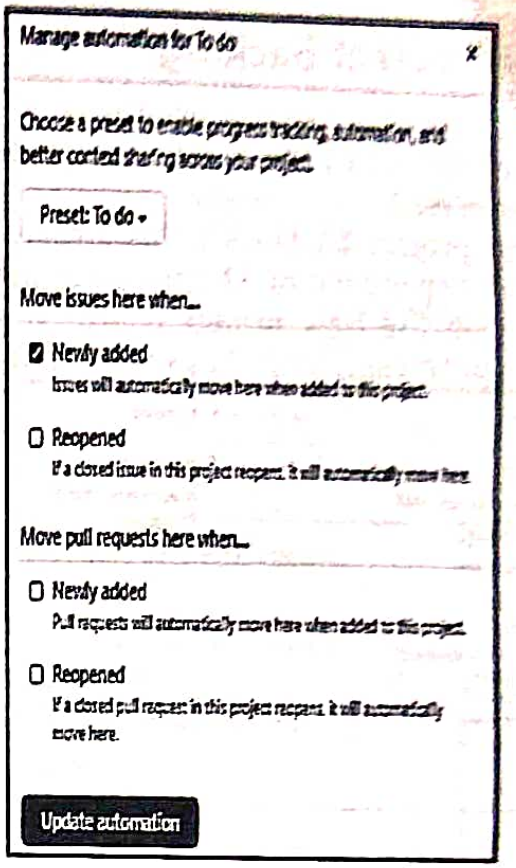



**Fig. 5.18 (n): Set up Automatic Workflows**

11. You can add a card in any of section of Kanban project using + sign.

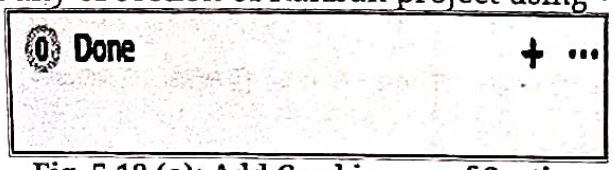



**Fig. 5.18 (o): Add Card in any of Section**

12. You can turn card it into an issue as well. Simply find the card in the project, click the '...' icon, then select "Convert to issue" button.

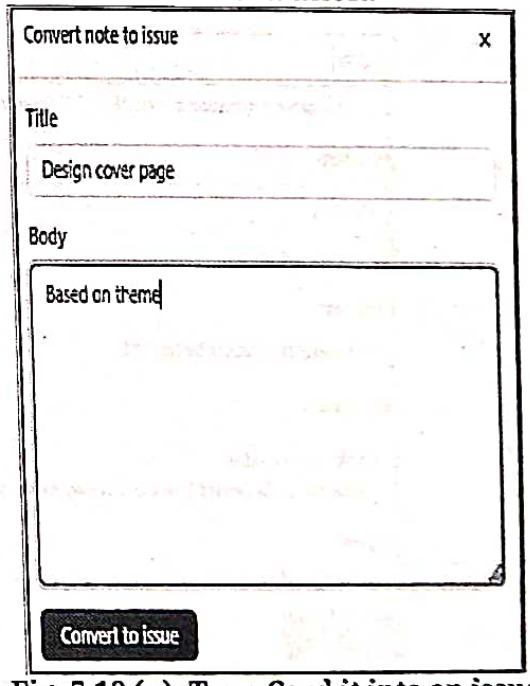



**Fig. 5.18 (p): Turn Card it into an issue**

## 5.3.4 Project Backlog

- A project backlog is a prioritized and structured list of deliverables that are a part of the scope of a project. It is often a complete list that breaks down work that needs to be completed.
- The project backlogs helps structure scope and identify business priorities before teams spend too much time planning the details.

1. On the new project board, you will see three columns named as "To do", "In progress", "Done" as shown in following Fig. 5.19 (a).

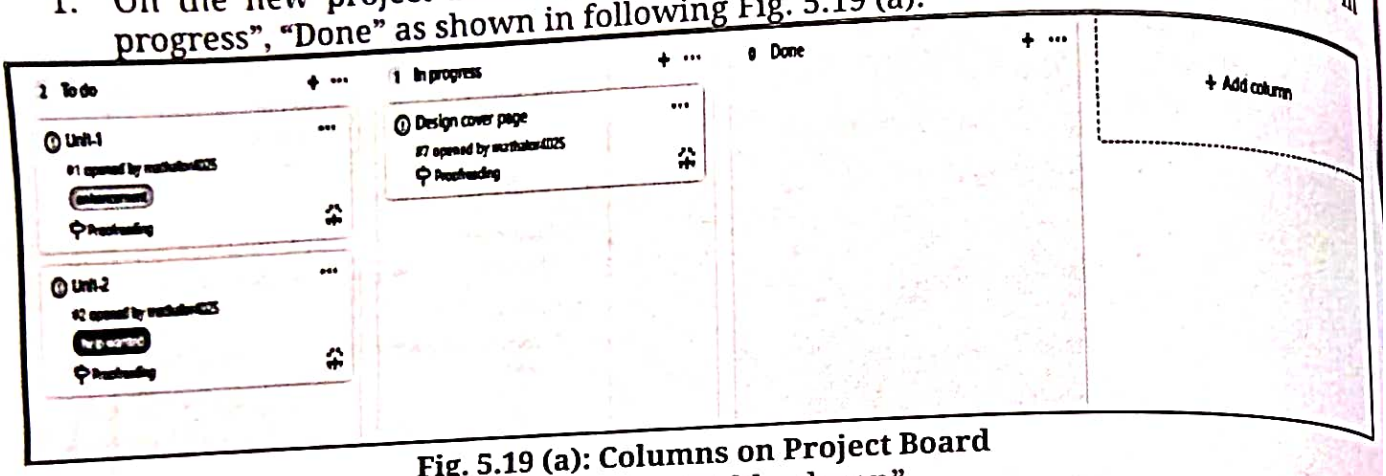



**Fig. 5.19 (a): Columns on Project Board**

2. You can add more columns by using "Add column".
3. Under "Column name", type the name of the column you want to create.

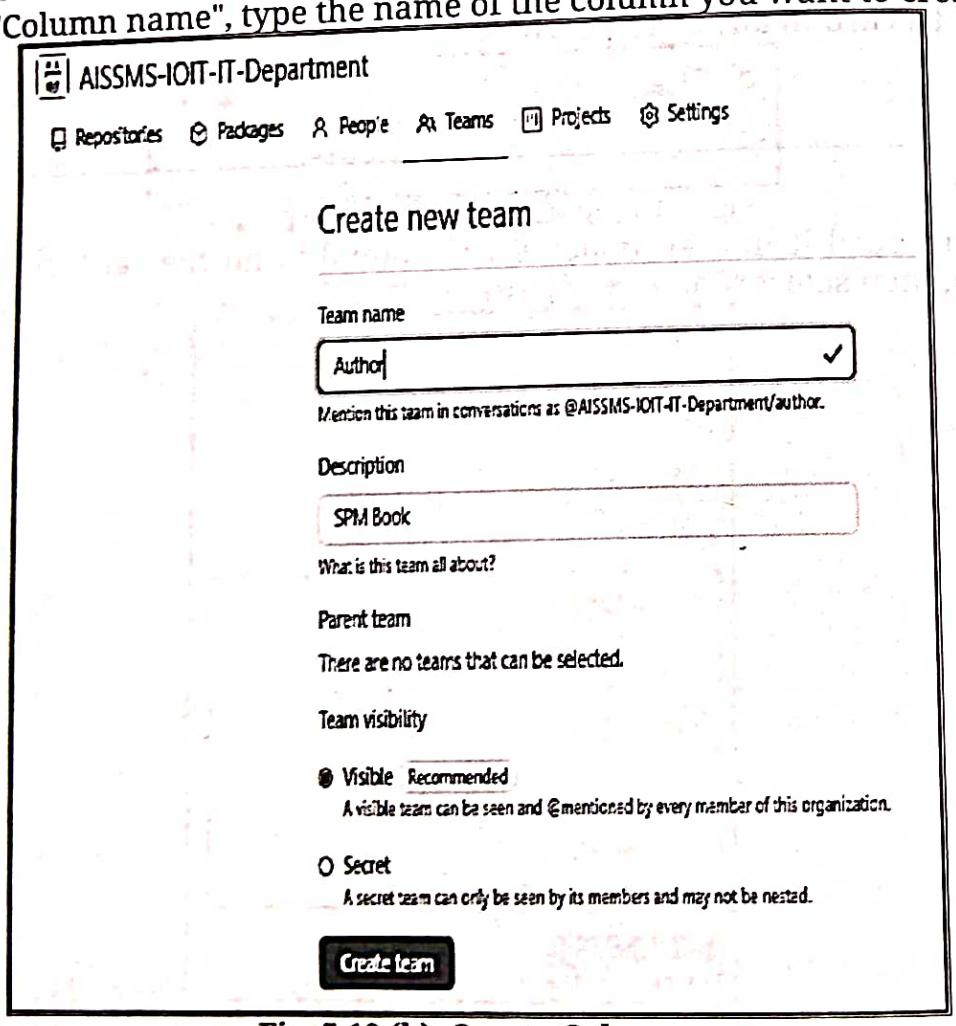



**Fig. 5.19 (b): Create Column**

4. Optionally, under "Automation" select the workflow automations you want to configure for the column.
5. Click **Create column.**

- Project board will appear as follows with newly added columns and you can drag the cards/issues which are at high priority.

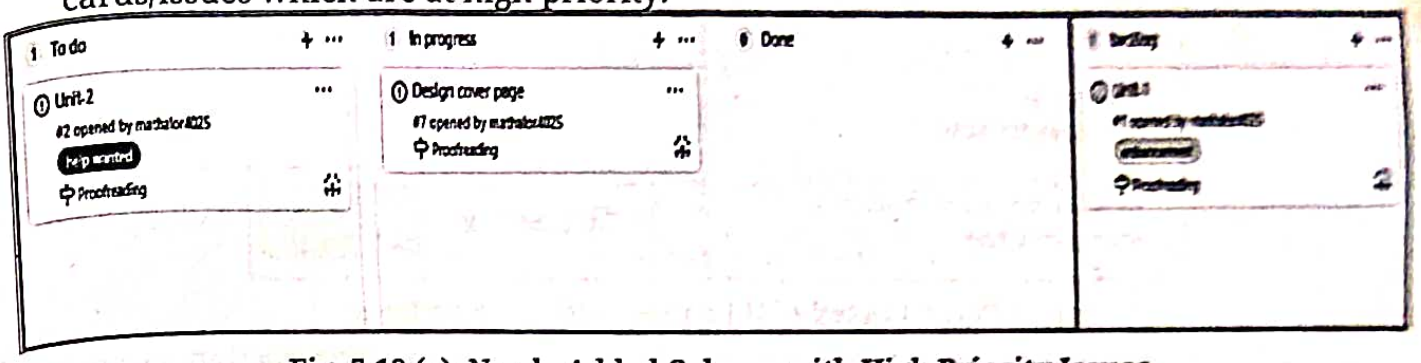



**Fig. 5.19 (c): Newly Added Column with High Priority Issues**

- When you click on one of issue, then you will see detail of that issue line on right side as shown below:

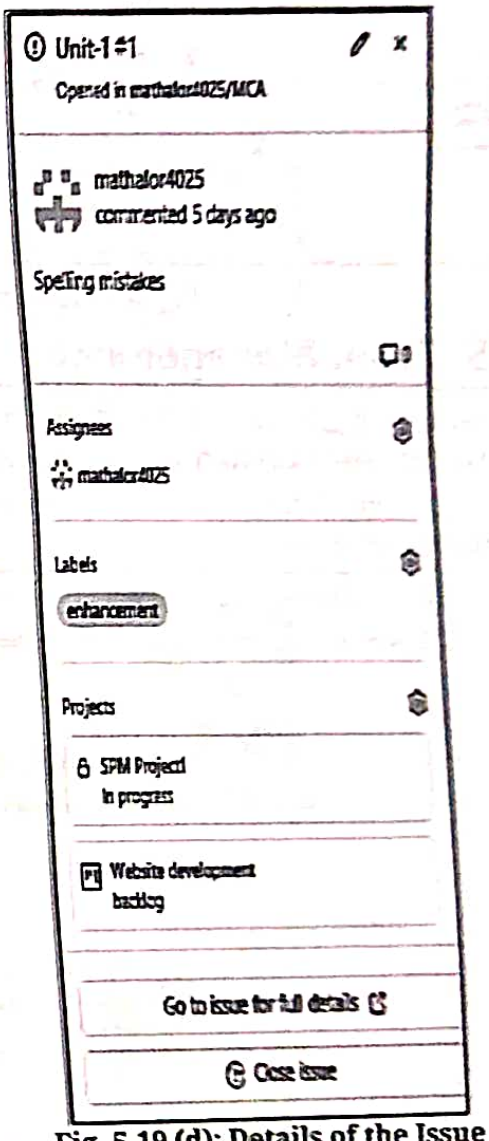



**Fig. 5.19 (d): Details of the Issue**

- Once this issue is commit and pull request is generated then by using "Go to issue for full detail" option, you can use linked pull request there and can close an issue.

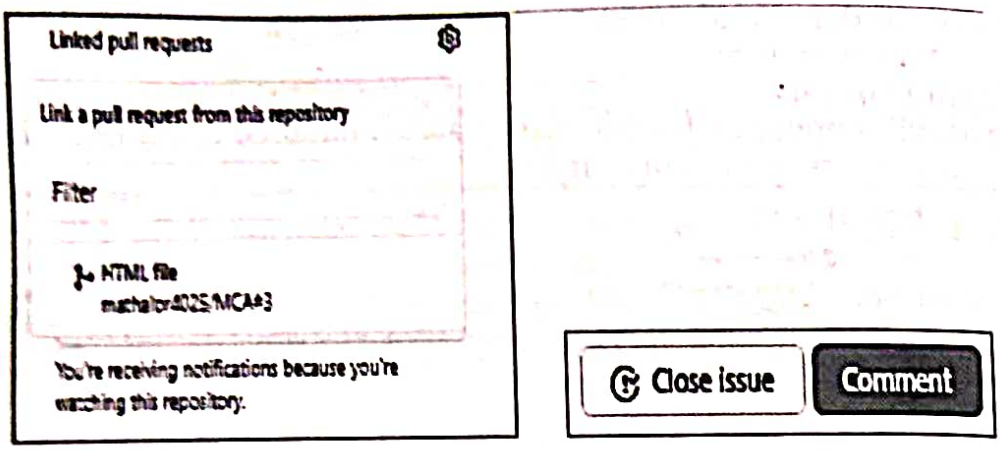



**Fig. 5.19 (e): Linked Pull Request and Close an Issue**

- Once an issue is closed, it will automatically reflected in **Done** section as shown below.

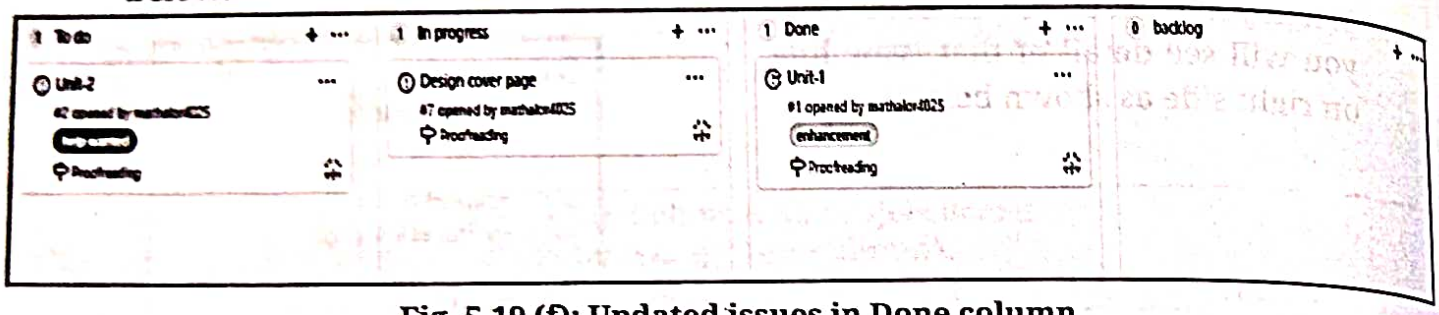



**Fig. 5.19 (f): Updated issues in Done column**

## 5.3.5 Team Management

- Teams are groups of organization members that reflect your company or group's structure with cascading access permissions and mentions.
- If your organization has Github account and no team is added in that then click on Teams option as shown in fig.

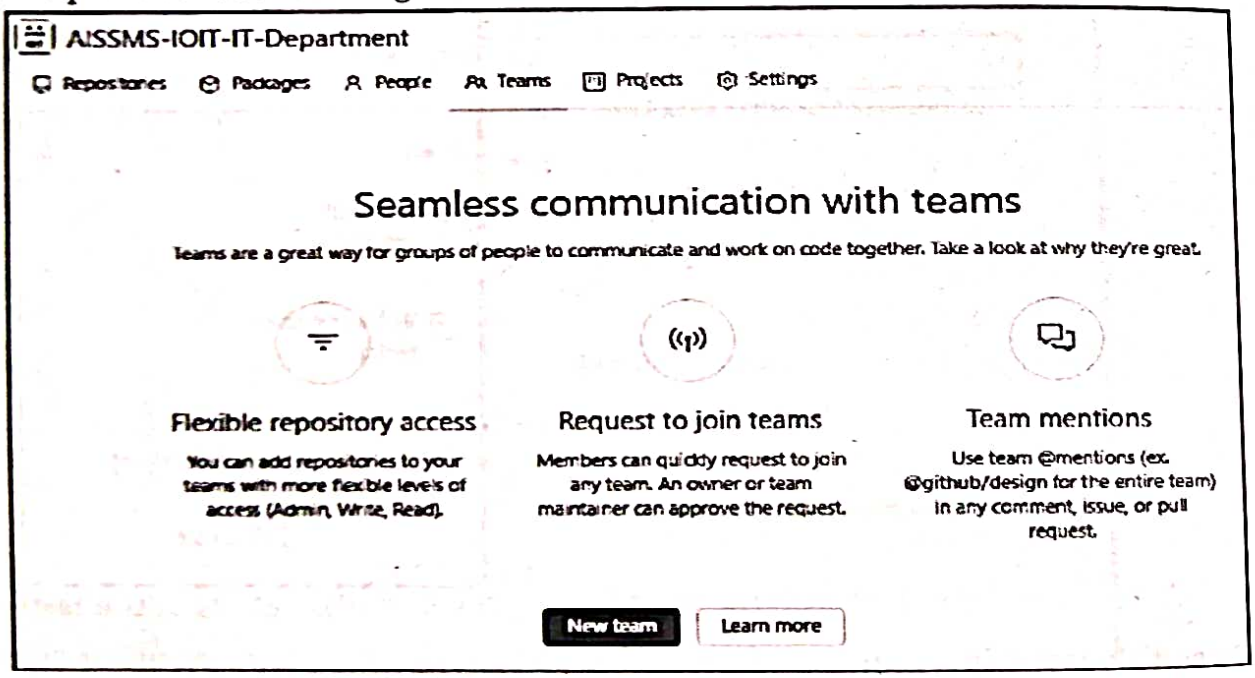



**Fig. 5.20 (a): New Team in GitHub Account**

- Click on **Create new Team** and enter name and description as shown below:

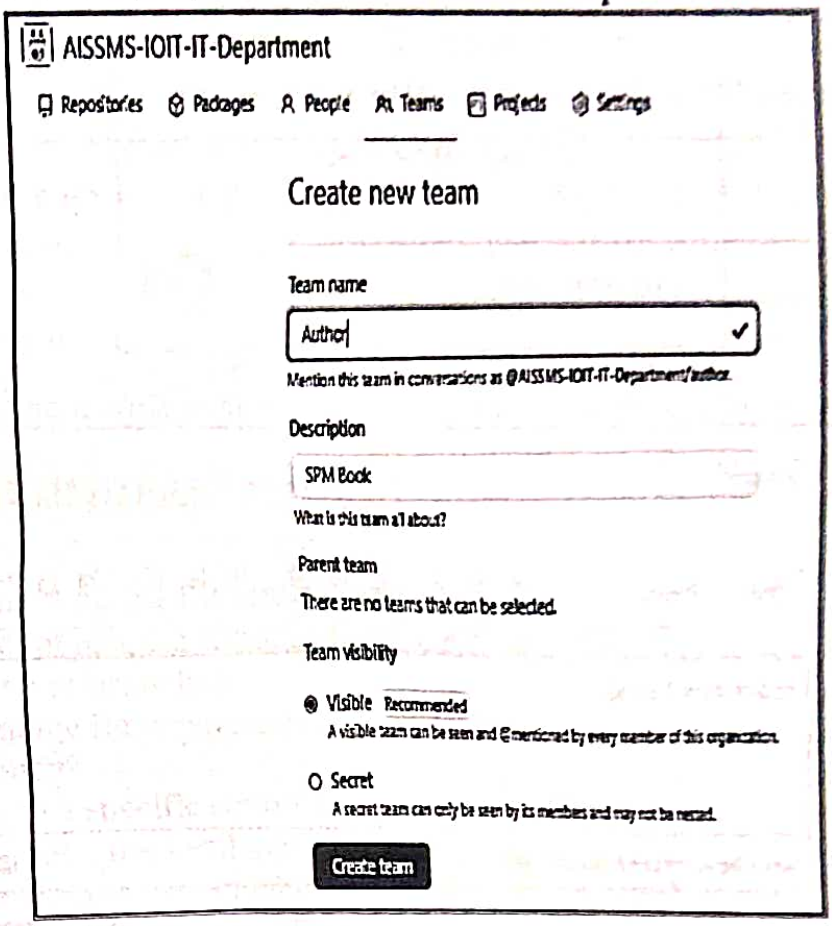



**Fig. 5.20 (b): Create New Team**

- Once you will click on **Create team** button, a team is added under the organization as shown below. In this way you can add multiple teams who working on project.

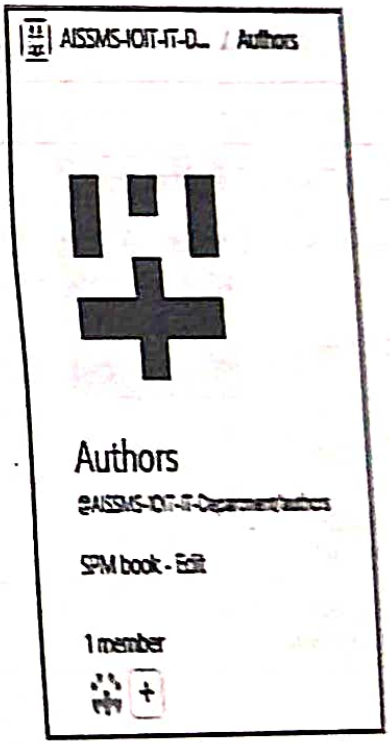



**Fig. 5.20 (c): Team added in Organization**

- In a team further you can add team members by inviting them like below. Once that member had accepted your request, then you can work in a team.

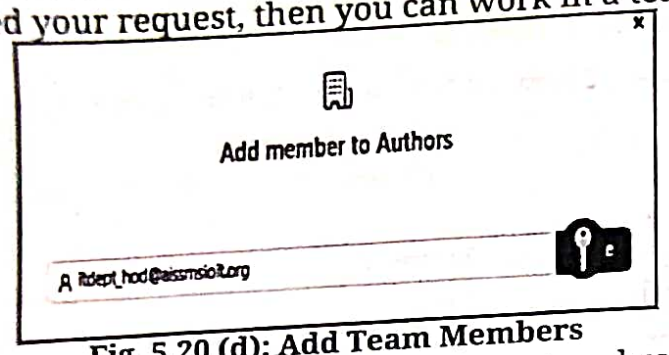



**Fig. 5.20 (d): Add Team Members**

- In a team, you can communicate by using @team name as shown below.

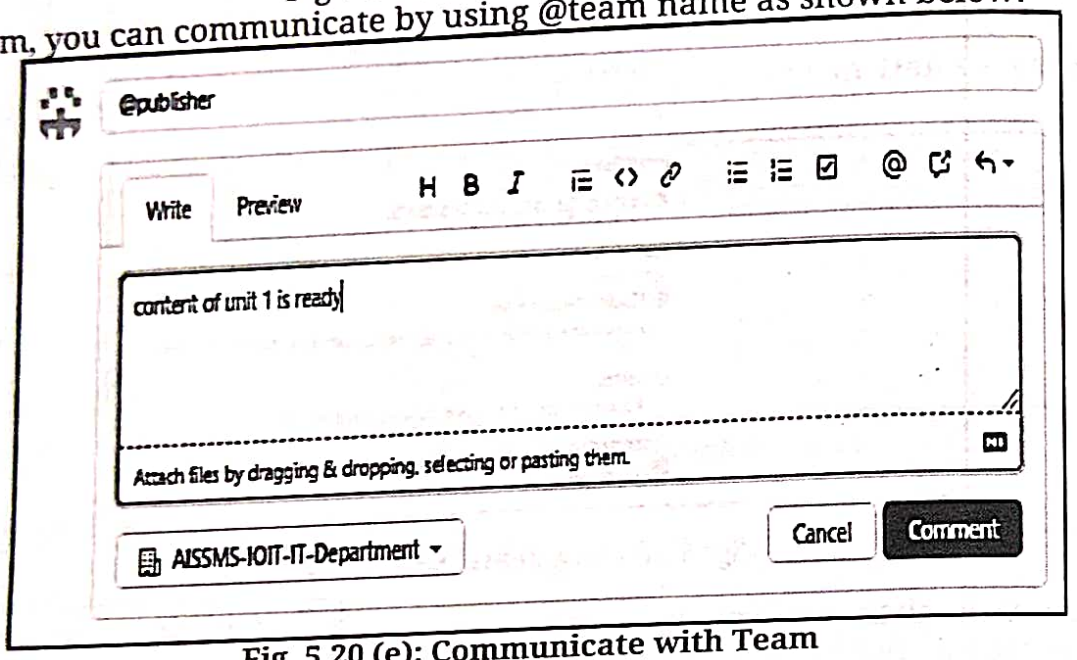



**Fig. 5.20 (e): Communicate with Team**

- On homepage, you can see the team details under **Teams** tab.

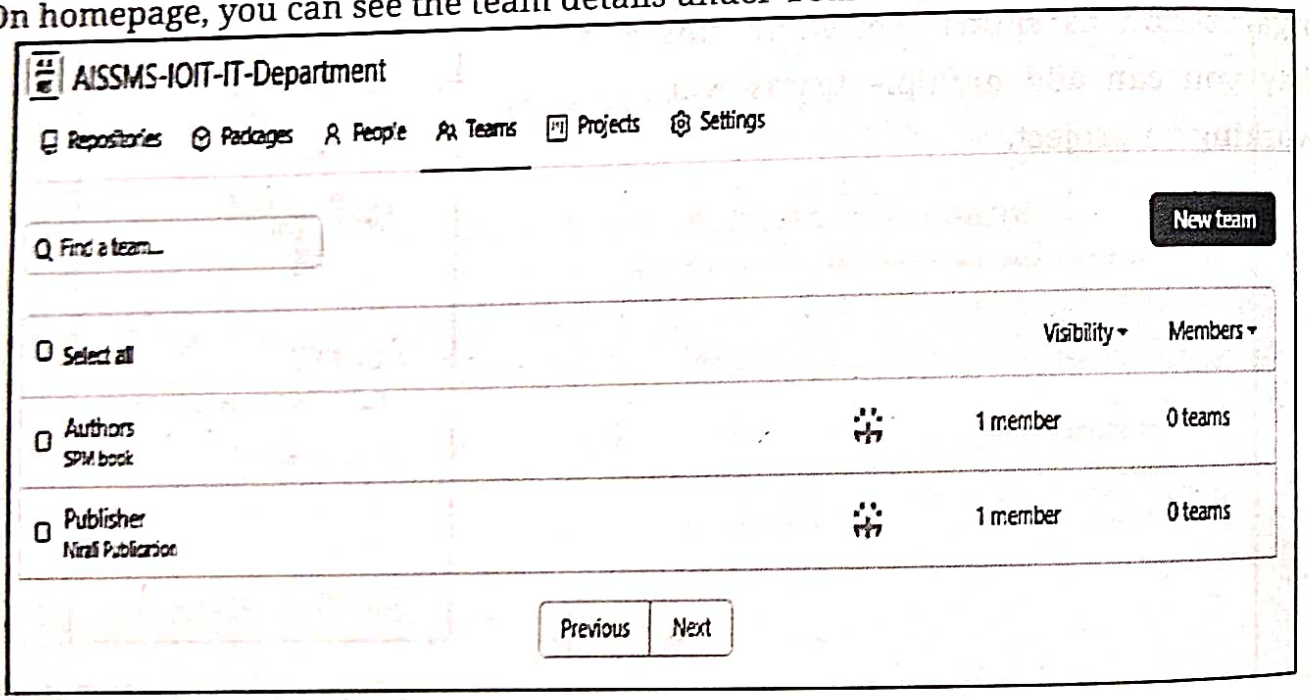



**Fig. 5.20 (f): Team Details**

## SUMMARY

- The popular frameworks for implementing Agile methodology include Scrum, Kanban, Extreme Programming, and the Adaptive Project Framework.
- Git is an open-source, version control tool created in 2005 by developers working on the Linux operating system; GitHub is a company founded in 2008 that makes tools which integrate with Git. We cannot use GitHub without using Git.
- A repository is a storage space where your project lives. It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, images or any kind of a file in a repository.

## CHECK YOUR UNDERSTANDING

1. What is a commit?
   (a) A snapshot of all the files in the repository.
   (b) A snapshot of just the changes from one time to the other.
   (c) A collection of branches.
   (d) Another name for a repository
2. What is a branch?
   (a) A pointer to a specific commit.
   (b) A link between the local and remote histories.
   (c) The centralized location where repositories are stored.
   (d) A version of a file at a specific time.
3. What is GitHub?
   (a) A host for Git repositories
   (b) An integrated development environment (IDE)
   (c) The company that owns Git
   (d) All of the above
4. Git is _____.
   (a) A version control system (b) Centralized
   (c) Distributed (d) The same as GitHub
5. Git is a _____ Version Control tool.
   (a) Decentralized (b) Centralized
   (c) Undistributed (d) Unique
6. Git belongs to the _____ generation of Version Control tools.
   (a) $2^{nd}$ (b) $3^{rd}$
   (c) $4^{th}$ (d) $5^{th}$
7. The main objectives of Git are _____.
   (a) speed
   (b) data integrity
   (c) support for distributed non-linear workflows
   (d) All of the above

8. Which language is used in Git?
   (a) C (b) HTML
   (c) PHP (d) C++
9. Imagine that you just joined a development team that uses Git for version control and collaboration. To start contributing to the project, which Git operation would you most likely invoke first?
   (a) checkout (b) clone
   (c) export (d) revert
   (e) update
10. Imagine that you have a local repository, but other team members have pushed changes into the remote repository. Which Git operation would you use to download those changes into your working copy?
   (a) checkout (b) commit
   (c) export (d) pull
   (e) update

**Answers**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1. (b) | 2. (d) | 3. (d) | 4. (a) | 5. (a) | 6. (b) | 7. (d) | 8. (a) | 9. (b) | 10. (d) |

## PRACTICE QUESTIONS

**Q. I Answer the following questions in short.**

1. What is GitHub?
2. What is Project board?
3. What is the use of Pull command?
4. How to create branch in GitHub?
5. What is Team management?
6. What's the Git command that downloads your repository from GitHub to your computer?

**Q. II. Answer the following questions.**

1. Explain any four agile tools available in market.
2. What is repository? How Repositories created on GitHub?
3. How to create an issue in Github?
4. How to enable project tracking in GitHub?

**Q. III Write a short note on:**

1. Commit Command
2. Branches in GitHub

***