SOFTWARE ENGINEERING

UNIT – 2

CHAPTER – 1

SOFTWARE REQUIREMENTS

I. Functional Requirements:

Functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behaviour, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

Functionality or system services are of two types:

i. Functional *user* requirements - high-level statements of what the system should do

ii. Functional *system* requirements - describe the system services in detail

These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

Examples of functional requirements:

a. The user shall be able to search either all of the initial set of databases or select a subset from it.

b. The system shall provide appropriate viewers for the user to read documents in the document store.

c. Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

The functional requirements definition of a system should be both,

a. Complete [i.e., It means that all services required by the user should be defined]

b. Consistent [i.e., it means that requirements should not have contradictory definitions]

Requirements imprecision

- Problems arise when requirements are not precisely stated.

- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term 'appropriate viewers'
- User intention - special purpose viewer for each different document type;
- Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements completeness and consistency:

- In principle, requirements should be both complete and consistent.
- Complete: They should include descriptions of all facilities required.
- Consistent: There should be no conflicts or contradictions in the descriptions of the system facilities.

  In practice, it is impossible to produce a complete and consistent requirements document.

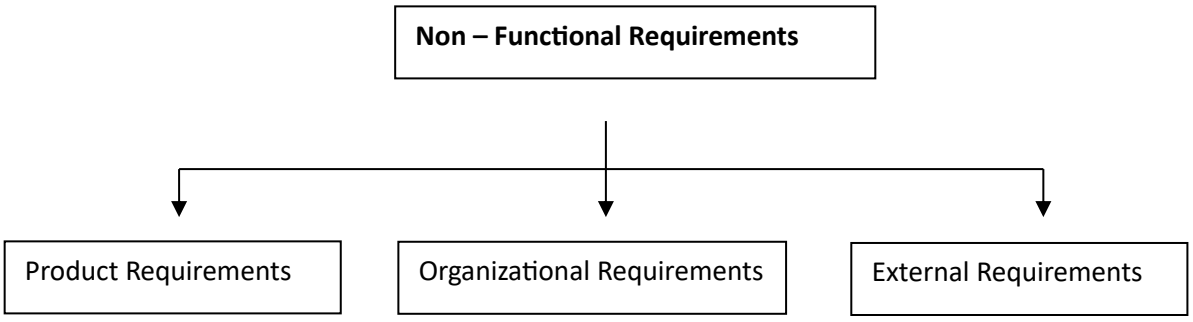
II. Non – Functional Requirements:

Non-functional requirements in software engineering refer to the characteristics of a software system that are not related to specific functionality or behaviour. They describe how the system should perform, rather than what it should do.

Example: Reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non – Functional Requirements Types:



(i) Product Requirements: These requirements result from the need for the delivered product, to behave in a particular way

Example:

- Requirements on how fast the system must execute and how much memory it requires
- Reliability Requirements [i.e., acceptable failure rate]
- Portability Requirements

(ii) Organizational Requirements:

These requirements are consequence of organizational policies and procedures

Example:

- Implementation requirements such as programming language (Or) design method used
- Delivery Requirements which specify when the product and its documentation to be delivered

(iii) External Requirements:

These requirements arise from factors external to the system and its development process

Example:

- Interoperability Requirements which specify how the system interacts with systems in other organizations
- Legislative Requirements, which ensure that the system operates within the law

III. Difference between the functional and non-functional requirements.

Parameters	Functional Requirement	Non-Functional Requirement
What it is	Verb	Attributes
Requirement	It is mandatory Non-Functional Testing like Performance,	It is non-mandatory
Capturing type	It is captured in use case.	It is captured as a quality attribute.
End-result	Product feature	Product properties
Capturing	Easy to capture	Hard to capture
Objective	Helps you verify the functionality of the software.	Helps you to verify the performance of the software.

Parameters	Functional Requirement	Non-Functional Requirement
Area of focus	Focus on user requirement	Concentrates on the user's expectation.
Documentation	Describe what the product does	Describes how the product works
Type of Testing	Functional Testing like System, Integration, End to End, API testing, etc.	Stress, Usability, Security testing, etc.
Test Execution	Test Execution is done before non-functional testing.	After the functional testing
Product Info	Product Features	Product Properties

IV. SOFTWARE REQUIREMENTS

Software requirements are necessary for the following:

- To introduce the concepts of **user and system requirements**
- To describe functional and non-functional requirements
- To explain how software requirements may be organized in a requirements document

V. Definition of requirement

- The requirements for the system are the description of the services provided by the system and its operational constraints.
- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
- May be the basis for a bid for a contract - therefore must be open to interpretation;
- May be the basis for the contract itself - therefore must be defined in detail; Both these statements may be called requirements

VI. Requirements engineering:

- The process of finding out, analysing, documenting and checking these services and constraints is called requirement engineering.

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

VII. Types of requirements:

1. User Requirements
2. System Requirements.

I. USER REQUIREMENTS

User requirements describe what the end-user wants from the software system. User requirements are usually expressed in natural language and are typically gathered through interviews, surveys, or user feedback as these can be understood by all users. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.

User requirements describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.

User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

II. SYSTEM REQUIREMENTS

System requirements are the building blocks developers use to build the system. These are the traditional "shall" statements that describe what the system "shall do." System requirements are classified as either functional or supplemental requirements.

These requirements specify the technical characteristics of the software system, such as its architecture, hardware requirements, software components, and interfaces. System requirements are typically expressed in technical terms and are often used as a basis for system design.

i. They are intended to be a basis for designing the system.

ii. They may be incorporated into the system contract.

iii. System requirements may be defined or illustrated using system models.

I. Software Requirement Specification (SRS)

Software Requirements Document, also known as SRS, is the term used to describe an in-depth description of a software product to be developed. It's considered one of the initial stages of the software development lifecycle (SDLC).

- Software Requirements Document (SRS) used to describe the behavior of the s/w system, functional, non-functional requirements of the s/w system.
- The SRS document is known as black-box specification, the system is considered as a black box whose internal details are not known, only its visible external (i.e., input/output) behaviour is documented.

II. IEEE Software Requirement Specification (SRS) Format

Software Requirements Specification for

<Project>

Version 1.0 approved

Prepared by <author>

<organization>

<date created>

Table of Contents

1. Introduction

1.1 Purpose <Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

1.2 Document Conventions <Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

1.3 Intended Audience and Reading Suggestions <Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

1.4 Product Scope <Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here.>

1.5 References <List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

2. Overall Description

2.1 Product Perspective <Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

2.2 Product Functions <Summarize the major functions the product must perform or must let the user perform. Details will be provided in Section 3, so only a high-level

summary (such as a bullet list) is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top-level data flow diagram or object class diagram, is often effective.>

2.3 User Classes and Characteristics <Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the most important user classes for this product from those who are less important to satisfy.>

2.4 Operating Environment <Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

2.5 Design and Implementation Constraints <Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

2.6 User Documentation <List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

2.7 Assumptions and Dependencies <List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

3. External Interface Requirements

3.1 User Interfaces <Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

3.2 Hardware Interfaces <Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

3.3 Software Interfaces <Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

3.4 Communications Interfaces <Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.>

4. System Features <This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional

hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

4.1 System Feature 1 <Don't really say "System Feature 1." State the feature name in just a few words.>

4.1.1 Description and Priority <Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

4.1.2 Stimulus/Response Sequences <List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

4.1.3 Functional Requirements <Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate, when necessary, information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1:

REQ-2:

4.2 System Feature 2 (and so on)

5. Other Nonfunctional Requirements

5.1 Performance Requirements <If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing

relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

5.2 Safety Requirements <Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product's design or use. Define any safety certifications that must be satisfied.>

5.3 Security Requirements <Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

5.4 Software Quality Attributes <Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

5.5 Business Rules <List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they may imply certain functional requirements to enforce the rules.>

6. Other Requirements <Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

Appendix A: Glossary <Define all the terms necessary to properly interpret the SRS, including acronyms and abbreviations. You may wish to build a separate glossary that spans multiple projects or the entire organization, and just include terms specific to a single project in each SRS.>

Appendix B: Analysis Models <Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>

Appendix C: To Be Determined List <Collect a numbered list of the TBD (to be determined) references that remain in the SRS so they can be tracked to closure.>

III. Characteristics of a Good SRS

1. **Correctness:** Each requirement accurately represents some desired feature in the final system
2. **Completeness:** All desired features/characteristics specified are satisfied, Completeness and correctness strongly related
3. **Unambiguous:** Each requirement has exactly one meaning, without these, errors will creep in, it is important as natural languages often used
4. **Verifiability:** There must exist a cost-effective way of checking if software satisfies requirements
5. **Consistent:** two requirements don't contradict each other
6. **Modifiable:** Any necessary change can be made easily while preserving completeness and consistency
7. **Ranked for importance/stability:** Needed for prioritizing in construction. To reduce risks due to changing requirements
8. **Right Level of Abstraction:** Details in SRS must be represented at right level of abstraction
9. **Traceability:** The origin of the requirement, and how the requirement relates to software elements can be determined
10. **Design Independent:** Provision for multiple design alternatives
11. **Testability:** It should be easy to generate test cases & test plans from the document
12. **Understandable By Customer:** Avoid using formal notations to represent requirements

I. REQUIREMENTS ENGINEERING PROCESSES

The goal of requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirement engineering sub-processes. These are concerned with:

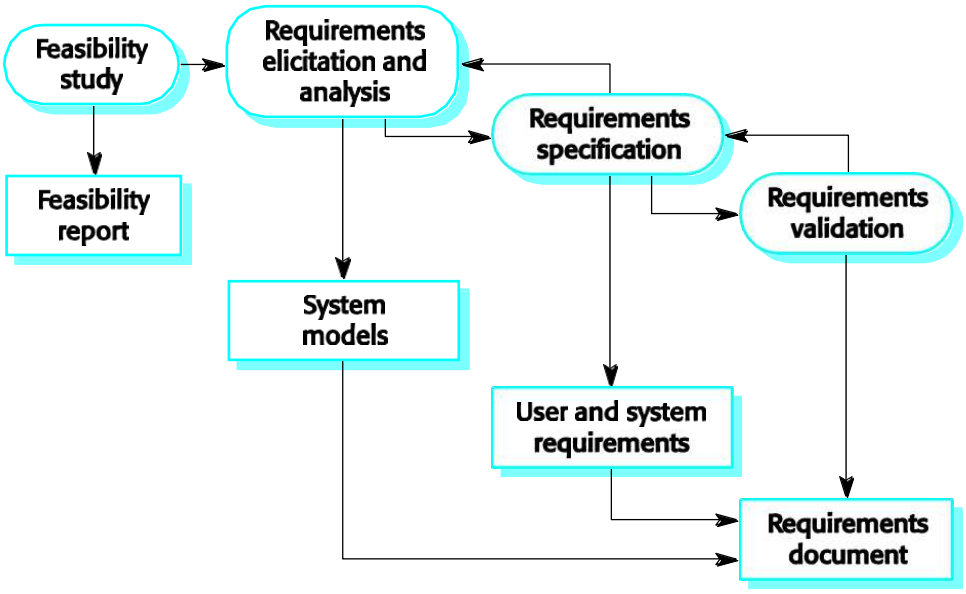
i. Assessing whether the system is useful to the business (feasibility study)

ii. Discovering requirements (elicitation and analysis)

iii. Converting these requirements into some standard form(specification)

iv. Checking that the requirements actually define the system that the customer wants(validation) The process of managing the changes in the requirements is called requirement management.

The requirements engineering process



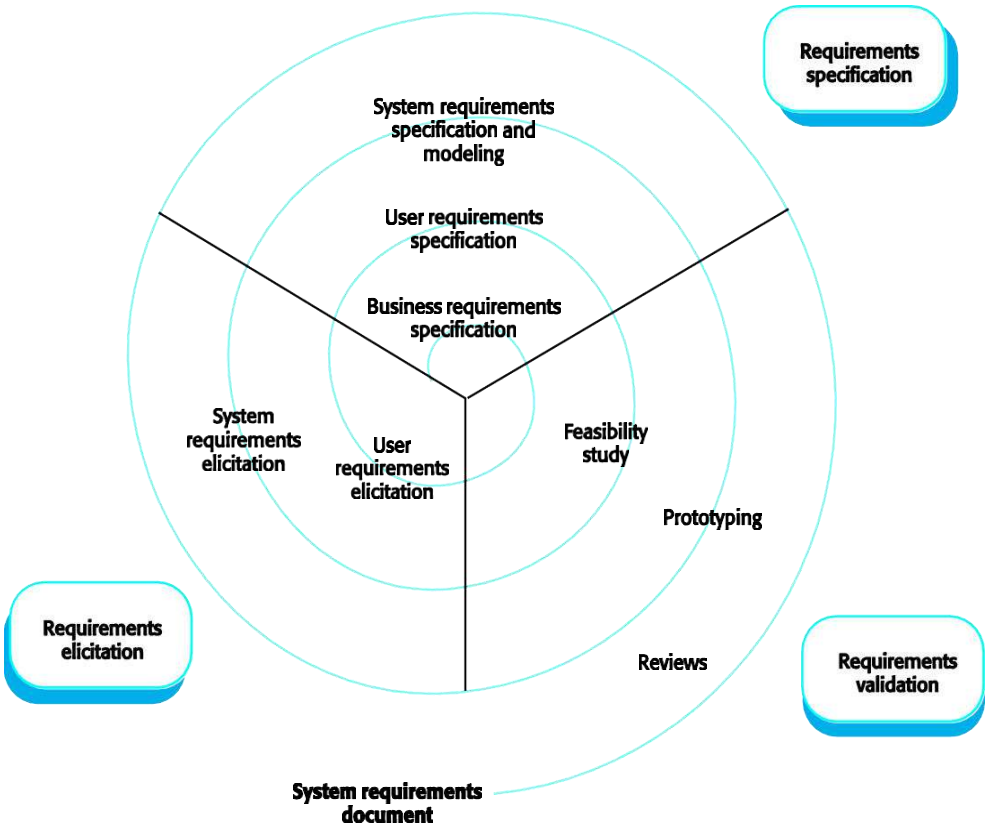
Requirements engineering: The alternative perspective on the requirements engineering process presents the process as a three-stage activity where the activities are organized as an iterative process around a spiral. The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements and the user requirements. Later in the process, in the outer rings of the spiral, more effort will be devoted to system requirements engineering and system modeling.

- This spiral model accommodates approaches to development in which the requirements are

developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.

Some people consider requirements engineering to be the process of applying a structured analysis method such as object-oriented analysis. This involves analysing the system and developing a set of graphical system models, such as use-case models, that then serve as a system specification. The set of models describes the behaviour of the system and are annotated with additional information describing, for example, its required performance or reliability.

Spiral model of requirements engineering processes



Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analysing need, assessing feasibility, negotiating a reasonable solution, specifying the solution un ambiguously, validating the specification.

Requirement engineering consists of seven different tasks as follow:

1. Inception: Inception is a task where the requirement engineering asks a set of questions to establish a software process. In this task, it understands the problem and evaluates with the proper solution. It collaborates with the relationship between the customer and the developer. The developer and customer decide the overall scope and the nature of the question.

2. Elicitation: Elicitation means to find the requirements from anybody. The requirements are difficult because the following problems occur in elicitation:

i. Problem of scope: The customer gives the unnecessary technical detail rather than clarity of the overall system objective.

ii. Problem of understanding: Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.

iii. Problem of volatility: In this problem, the requirements change from time to time and it is difficult while developing the project.

3. Elaboration: In this task, the information taken from user during inception and elaboration are expanded and refined in elaboration. Its main task is developing pure model of software using functions, feature and constraints of a software.

4. Negotiation: In negotiation task, a software engineer decides, how will the project be achieved with limited business resources. To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

5. Specification: In this task, the requirement engineer constructs a final work product. The work product is in the form of software requirement specification. In this task, formalize the requirement of the proposed software such as informative, functional and behavioural. The requirement is formalized in both graphical and textual formats.

6. Validation: The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step. The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

7. Requirement management: It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project. These tasks start with the identification and assign a unique identifier to each of the requirement. After finalizing the requirement traceability table is developed

II. FEASIBILITY STUDIES

A feasibility study decides whether or not the proposed system is worthwhile. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it worth carrying on with the requirements engineering and system development process.

A short-focused study that checks:

i. If the system contributes to organizational objectives;

ii. If the system can be engineered using current technology and within budget;

iii. If the system can be integrated with other systems that are used.

Types of Feasibility Study: The feasibility study mainly concentrates on below five mentioned areas. Among these Economic Feasibility Study is most important part of the feasibility analysis and Legal Feasibility Study is less considered feasibility analysis.

a. Technical Feasibility –

In Technical Feasibility current resources both hardware software along with required technology are analyzed/assessed to develop project. This technical feasibility study gives report whether there exists correct required resources and technologies which will be used for project development. Along with this, feasibility study also analyzes technical skills and capabilities of technical team, existing technology can be used or not, maintenance and up-gradation is easy or not for chosen technology etc.

b. Operational Feasibility –

In Operational Feasibility degree of providing service to requirements is analyzed along with how much easy product will be to operate and maintenance after deployment. Along with these other operational scopes are determining usability of product, determining suggested solution by software development team is acceptable or not etc.

c. Economic Feasibility –

In Economic Feasibility study cost and benefit of the project is analyzed. Means under this feasibility study a detail analysis is carried out what will be cost of the project for development which includes all required cost for final development like hardware and software resource required, design and development cost and operational cost and so on. After that it is analyzed whether project will be beneficial in terms of finance for organization or not.

d. Legal Feasibility –

In Legal Feasibility study project is analyzed in legality point of view. This includes analyzing barriers of legal implementation of project, data protection acts or social media laws, project certificate, license, copyright etc. Overall, it can be said that Legal Feasibility Study is study to know if proposed project conforms legal and ethical requirements.

e. Schedule Feasibility –

In Schedule Feasibility Study mainly timelines/deadlines is analyzed for proposed project which includes how many times teams will take to complete final project which has a great impact on the organization as purpose of project may fail if it can't be completed on time.

III. Feasibility study implementation:

A feasibility study involves information assessment, information collection and report writing. Questions for people in the organization are:

i. What if the system wasn't implemented?
ii. What are current process problems?
iii. How will the proposed system help?
iv. What will be the integration problems?
v. Is new technology needed? What skills?

vi. What facilities must be supported by the proposed system?

In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system. They should try to complete a feasibility study in two or three weeks.

Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

SOFTWARE ENGINEERING

UNIT – 2

CHAPTER – 2

REQUIREMENT ENGINEERING

I. REQUIREMENT ELICITATION AND ANALYSIS:

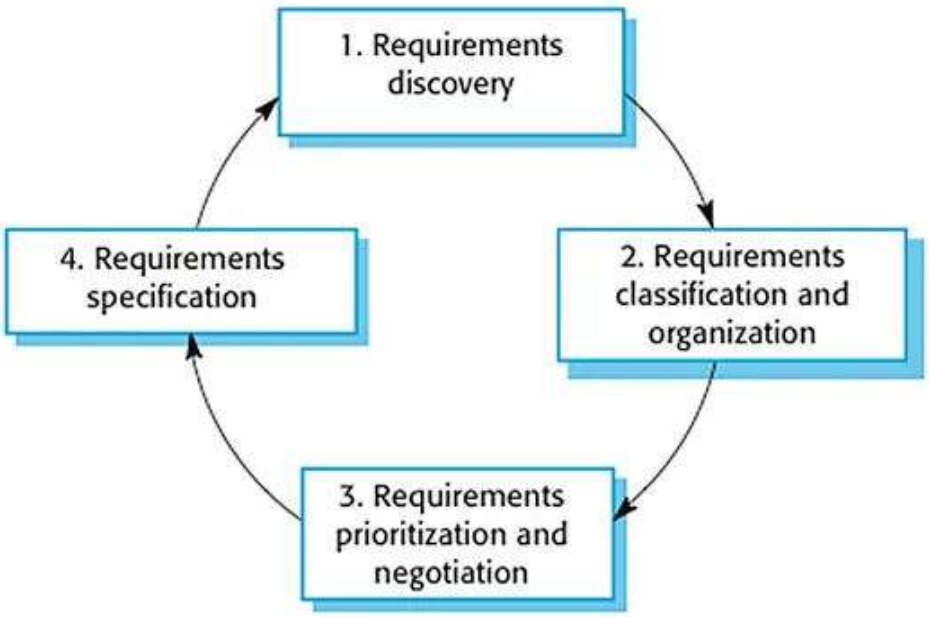
- The requirement engineering process is requirements elicitation and analysis sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called stakeholders.
- We typically start by gathering the requirements, this could be done through a general discussion or interviews with your stakeholders, also it may involve some graphical notation.
- Then you organize the related requirements into sub-components and prioritize them, and finally, you refine them by removing any ambiguous requirements that may arise from some conflicts.

Features of requirements elicitation:

- Stakeholder engagement: Requirements elicitation involves engaging with stakeholders such as customers, end-users, project sponsors, and subject matter experts to understand their needs and requirements.
- Gathering information: Requirements elicitation involves gathering information about the system to be developed, the business processes it will support, and the end-users who will be using it.
- Requirement prioritization: Requirements elicitation involves prioritizing requirements based on their importance to the project's success.
- Requirements documentation: Requirements elicitation involves documenting the requirements in a clear and concise manner so that they can be easily understood and communicated to the development team.
- Validation and verification: Requirements elicitation involves validating and verifying the requirements with the stakeholders to ensure that they accurately represent their needs and requirements.

- Iterative process: Requirements elicitation is an iterative process that involves continuously refining and updating the requirements based on feedback from stakeholders.
- Communication and collaboration: Requirements elicitation involves effective communication and collaboration with stakeholders, project team members, and other relevant parties to ensure that the requirements are clearly understood and implemented.
- Flexibility: Requirements elicitation requires flexibility to adapt to changing requirements, stakeholder needs, and project constraints.

The requirements spiral process activities:



- Requirements discovery
- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organization
- Groups related requirements and organizes them into coherent clusters.
- Prioritization and negotiation
- Prioritizing requirements and resolving requirements conflicts.
- Requirements documentation
- Requirements are documented and input into the next round of the spiral.
- The process cycle starts with requirements discovery and ends with requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle.

- Requirements classification and organization is primarily concerned with identifying overlapping requirements from different stakeholders and grouping related requirements. The most common way of grouping requirements is to use a model of the system architecture to identify subsystems and to associate requirements with each sub-system.
- Inevitably, stakeholders have different views on the importance and priority of requirements, and sometimes these view conflict. During the process, you should organize regular stakeholder negotiations so that compromises can be reached.
- In the requirement documenting stage, the requirements that have been elicited are documented in such a way that they can be used to help with further requirements discovery.

II. REQUIREMENTS DISCOVERY:

- Requirement discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems.
- They interact with stakeholders through interview and observation and may use scenarios and prototypes to help with the requirements discovery.
- Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system.

  For example, system stakeholder for a bank ATM includes
- Bank customers
- Representatives of other banks
- Bank managers
- Counter staff
- Database administrators
- Security managers
- Marketing department
- Hardware and software maintenance engineers
- Banking regulators
- Requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoint, where each viewpoint presents a sub-set of the requirements for the system.

There are a number of requirements elicitation methods. Few of them are listed below –

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

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

1. Interviews:

- Objective of conducting an interview is to understand the customer's expectations from the software.
- It is impossible to interview every stakeholder hence representatives from groups are selected based on their expertise and credibility.
- Interviews maybe be open-ended or structured.
- In open-ended interviews there is no pre-set agenda. Context free questions may be asked to understand the problem.
- In structured interview, agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview.

2. Brainstorming Sessions:

- It is a group technique
- It is intended to generate lots of new ideas hence providing a platform to share views
- A highly trained facilitator is required to handle group bias and group conflicts.
- Every idea is documented so that everyone can see it.
- Finally, a document is prepared which consists of the list of requirements and their priority if possible.

3. Facilitated Application Specification Technique:

objective is to bridge the expectation gap – the difference between what the developers think they

are supposed to build and what customers think they are going to get. A team-oriented approach is developed for requirements gathering.

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

- Part of the environment that surrounds the system
- Produced by the system
- Used by the system
- Each participant prepares his/her list, different lists are then combined, redundant entries are eliminated, team is divided into smaller sub-teams to develop mini-specifications and finally a draft of specifications is written down using all the inputs from the meeting.

4. Quality Function Deployment:

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

3 types of requirements are identified –

Normal requirements –

In this the objective and goals of the proposed software are discussed with the customer. Example – normal requirements for a result management system may be entry of marks, calculation of results, etc

Expected requirements –

These requirements are so obvious that the customer need not explicitly state them. Example – protection from unauthorized access.

Exciting requirements –

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

The major steps involved in this procedure are –

- Identify all the stakeholders, eg. Users, developers, customers etc
- List out all requirements from customer.
- A value indicating degree of importance is assigned to each requirement.
- In the end the final list of requirements is categorized as –
- It is possible to achieve

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

5. Use Case Approach:

This technique combines text and pictures to provide a better understanding of the requirements.

The use cases describe the 'what', of a system and not 'how'. Hence, they only give a functional view of the system.

The components of the use case design includes three major things – Actor, Use cases, use case diagram.

Actor –

It is the external agent that lies outside the system but interacts with it in some way. An actor maybe a person, machine etc. It is represented as a stick figure. Actors can be primary actors or secondary actors.

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

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

Use cases –

They describe the sequence of interactions between actors and the system. They capture who(actors) do what(interaction) with the system. A complete set of use cases specifies all possible ways to use the system.

Use case diagram –

A use case diagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system.

A stick figure is used to represent an actor.

An oval is used to represent a use case.

A line is used to represent a relationship between an actor and a use case.

ETHNOGRAPHY:

- A social scientist spends a considerable time observing and analyzing how people actually work.
- People do not have to explain or articulate their work.
- Social and organizational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Scope of ethnography

Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work. Requirements that are derived from cooperation and awareness of other people's activities.

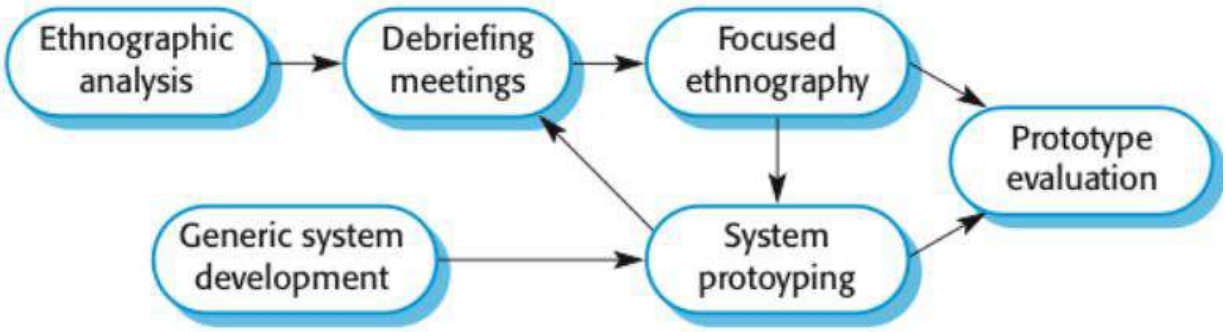
Awareness of what other people are doing leads to changes in the ways in which we do things.

Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system

Focused ethnography: Developed in a project studying the air traffic control process

- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping for requirement analysis



III. Requirements Classification & Organization

- It's very important to organize the overall structure of the system.
- Putting related requirements together, and decomposing the system into sub-components of related requirements. Then, we define the relationship between these components.
- What we do here will help us in the decision of identifying the most suitable architectural design patterns.

3. Requirements Prioritization & Negotiation

- We previously explained why eliciting and understanding the requirements is not an easy process.
- One of the reasons is the conflicts that may arise as a result of having different stakeholders involved. Why? because it's hard to satisfy all parties if it's not impossible.

- This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiations until you reach a situation where some of the stakeholders can compromise.
- We shouldn't reach a situation where a stakeholder is not satisfied because his requirements are not taken into consideration.
- Prioritizing your requirements will help you later to focus on the essentials and core features of the system, so you can meet the user expectations. It can be achieved by giving every piece of function a priority level. So, functions with higher priorities need higher attention and focus.

IV. Requirements Specification

- The requirements are then documented.

Advantages of Requirements Elicitation:

- Helps to clarify and refine customer requirements.
- Improves communication and collaboration between stakeholders.
- Increases the chances of developing a software system that meets customer needs.
- Avoids misunderstandings and helps to manage expectations.
- Supports the identification of potential risks and problems early in the development cycle.
- Facilitates the development of a comprehensive and accurate project plan.
- Increases user and stakeholder confidence in the software development process.
- Supports the identification of new business opportunities and revenue streams.

Disadvantages of Requirements Elicitation:

- Can be time-consuming and expensive.
- Requires specialized skills and expertise.
- May be impacted by changing business needs and requirements.
- Can be impacted by political and organizational factors.
- Can result in a lack of buy-in and commitment from stakeholders.
- Can be impacted by conflicting priorities and competing interests.
- May result in incomplete or inaccurate requirements if not properly managed.
- Can lead to increased development costs and decreased efficiency if requirements are not

Requirements Analysis

- We talk about priority of requirements
- Requirements are analyzed to
- Find inconsistencies, contradictions, defects, etc.
- Check whether two or more requirements can work together or check whether the same requirement is asked in different ways twice. E.g. 1st the user asked for C++ later Java
- Analyze the relationship between requirements
- Omissions
- If later we forgot to ask few requirements in some cases.
- Main purpose:
- Clearly understand the users requirements
- Detect inconsistency, ambiguity (words like good, bad, high, low, etc. for e.g. speed of software should be very high) and incompleteness
- Further discussion with end user and customer to remove above mentioned points.

REQUIREMENTS VALIDATION

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking:

- ***Validity***: Does the system provide the functions which best support the customer's needs?
- ***Consistency***: Are there any requirements conflicts?
- ***Completeness***: Are all functions required by the customer included?
- ***Realism***: Can the requirements be implemented given available budget and technology
- ***Verifiability***: Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
– Systematic manual analysis of the requirements.
- Prototyping
– Using an executable model of the system to check requirements.
- Test-case generation
- Developing tests for requirements to check testability.

Requirements reviews:

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks:

- ***Verifiability*:** Is the requirement realistically testable?
- ***Comprehensibility*:** Is the requirement properly understood?
- ***Traceability*:** Is the origin of the requirement clearly stated?
- ***Adaptability*:** Can the requirement be changed without a large impact on other requirements?

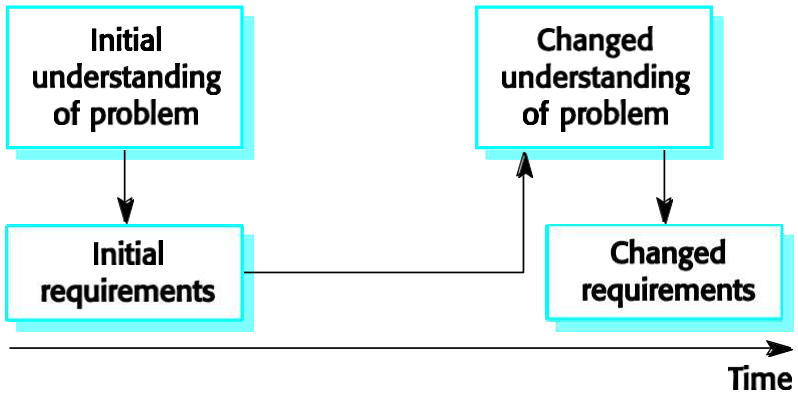
II REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably in complete and inconsistent
- New requirements emerge during the process as business needs change and a better understanding of the system is developed;
- Different viewpoints have different requirements and these are often

Requirements change

- The priority of requirements from different viewpoints changes during the development process.
- Systemcustomersmayspecifyrequirementsfromabusinessperspectivethatconflictwithend-userrequirements.
- Thebusinessandtechnicalenvironmentofthesystemchangesduringitsdevelopment.

Requirements evolution:



Enduring and volatile requirements:

- **Enduring requirements**: Stable requirements derived from the core activity of the customer organization. E.g., a hospital will always have doctors, nurses, etc. May be derived from domain models
- **Volatile requirements**: Requirements which change during development or when the system is in use. Ina hospital, requirements derived from health-care policy

Requirements classification:

Requirement Type	Description
Mutable requirements	Requirements that change because of changes to the environment in which the Organization is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.

Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organizations processes and open up new ways of working which generate new system requirements
Compatibility requirements	Requirements that depend on the particular systems or business processes within an organization. As this change, the compatibility requirements on the commissioned Or delivered system may also have to evolve.

Requirements management planning:

- During the requirements engineering process, you have to plan:

– Requirements identification

- How requirements are individually identified;

– A change management process

- The process followed when analyzing requirements change;

– Traceability policies

- The amount of information about requirements relationships that is maintained;

– CASE tool support

- The tool support required to help manage requirements change;

Traceability:

Traceability is concerned with the relationships between requirements, their sources and the system design

- Source traceability: Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability: Links between dependent requirements;
- Design traceability: Links from the requirements to the design;

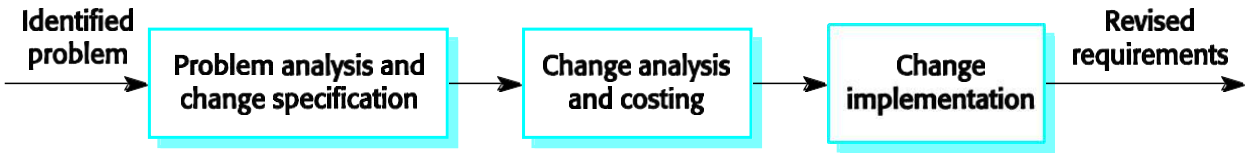
CASE tool support:

- Requirements storage: Requirements should be managed in a secure, managed data store.
- Change management: The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management: Automated retrieval of the links between requirements.

Requirements change management:

- Should apply to all proposed changes to the requirements.
- Principal stages
- Problem analysis. Discuss requirements problem and propose change;

- Change analysis and costing. Assess effects of change on other requirements;
- Changeimplementation.Modifyrequirementsdocumentandotherdocumentstoreflectchange.



Change management:

input to change management is: identify problem

output is revised requirements

steps in change management

- problem analysis and change specification
- change analysis and costing
- change implementation

III. Requirements Monitoring

Especially needs in incremental development

- **Distributed debugging-** uncovers errors and determines their cause
- **Run-time verification-**Determine whether software matches its specification
- **Run-time validation -**assesses whether evolving software meets user goal
- **Business activity monitoring-** evaluates whether a system satisfies business goals
- **Evolution and code design-** Provides information to stakeholders as the system evolves

SOFTWARE ENGINEERING

UNIT – 2

CHAPTER – 3

INTRODUCTION TO GIT AND GITHUB

I. Version Control System (VCS)

Version Control Systems are the software tools for tracking or managing all the changes made to the source code during the project development. It keeps a record of every single change made to the code. It also allows us to turn back to the previous version of the code if any mistake is made in the current version. Without a VCS in place, it would not be possible to monitor the development of the project.

II. Types of VCS

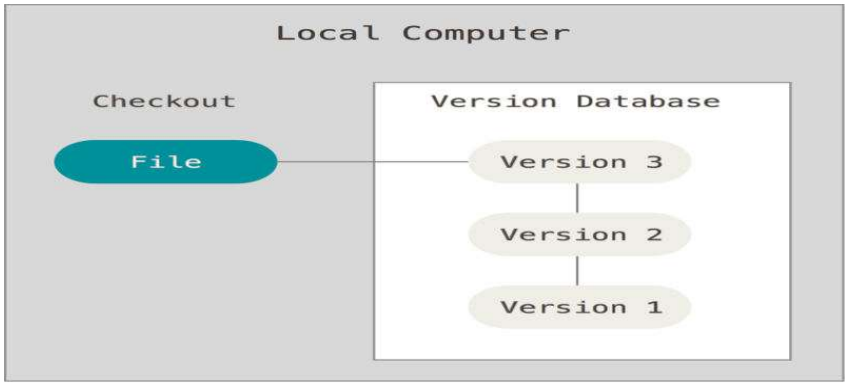
The three types of VCS are:

1. Local Version Control System
2. Centralized Version Control System
3. Distributed Version Control System

Local Version Control System

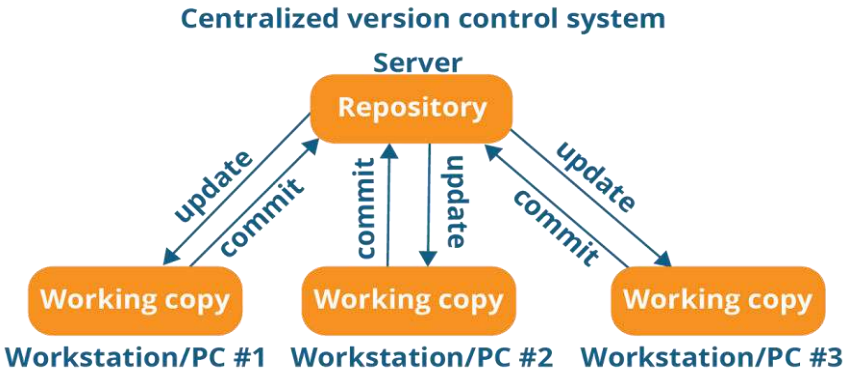
Local Version Control System is located in your local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

Also, with the Local Version Control System, it is not possible to collaborate with other collaborators.



Centralized Version Control System

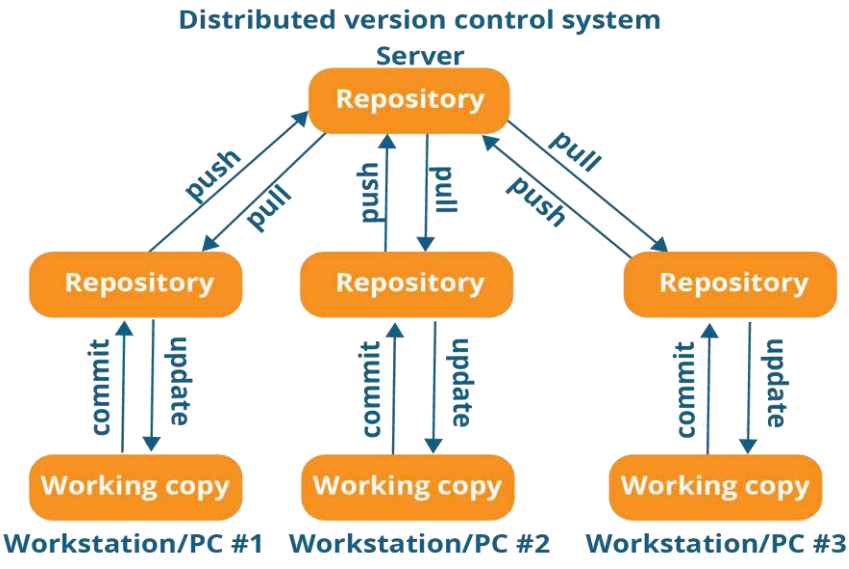
In the Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server (you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.



Distributed Version Control System

In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy (mirroring) of the main repository (including its entire history) on their local machines.

Each user has their own repository and a working copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.



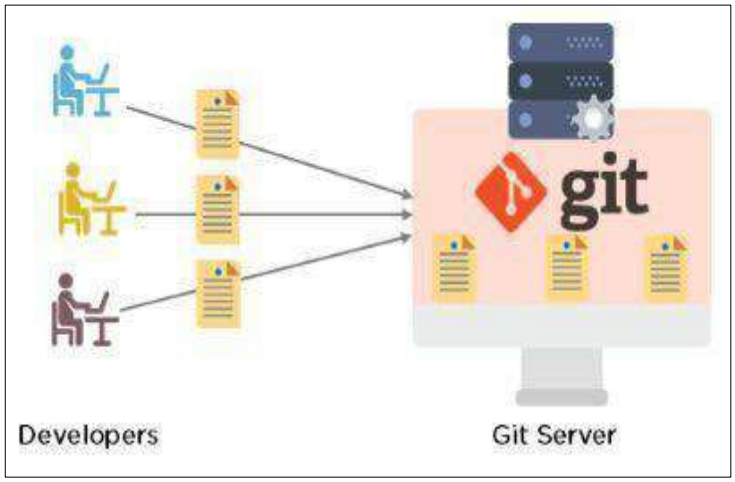
III. What is Git?

Git is a version control system used for tracking changes in computer files. It is a DevOps tool used for source code management in software development. **Linus Torvalds** created Git in 2005 for the development of the Linux kernel.

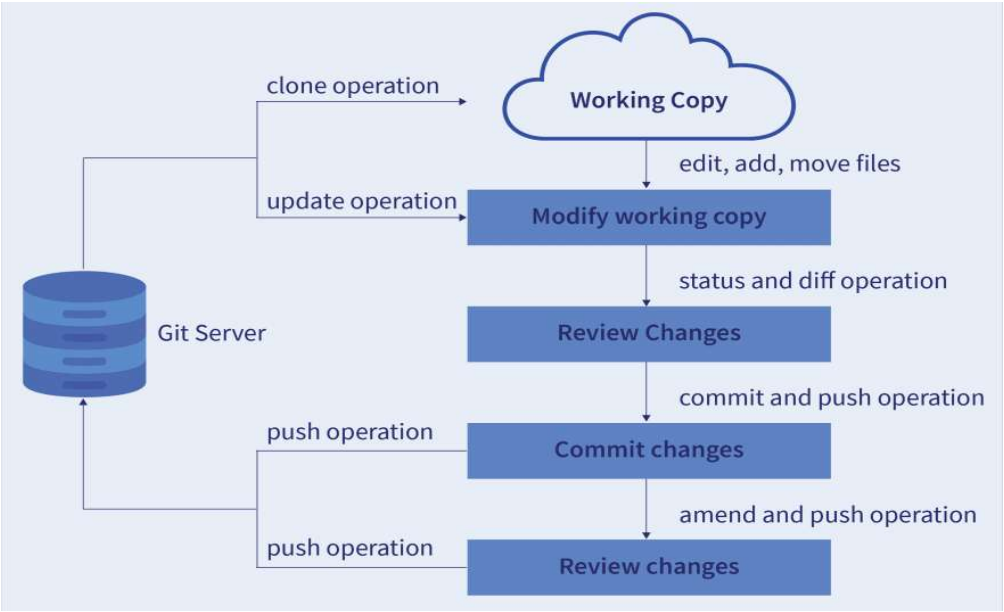
i. Git is used to tracking changes in the source code
ii. The distributed version control tool is used for source code management
iii. It allows multiple developers to work together
iv. It supports non-linear development through its thousands of parallel branches

Features of Git

i. Tracks history
ii. Free and open source
iii. Supports non-linear development
iv. Creates backups
v. Scalable
vi. Supports collaboration
vii. Branching is easier
viii. Distributed development



IV. Git Workflow:

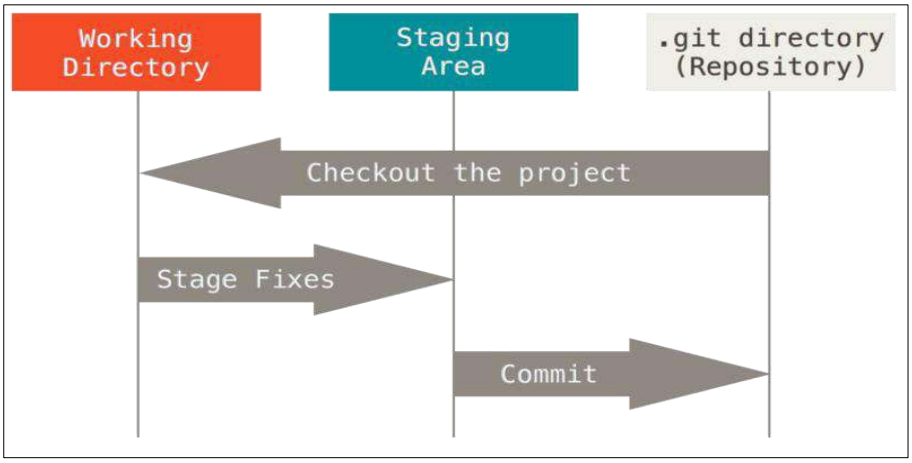


The Git workflow is divided into three states:

a. **Working directory** - Modify files in your working directory

b. **Staging area (Index)** - Stage the files and add snapshots of them to your staging area

c. **Git directory (Repository)** - Perform a commit that stores the snapshots permanently to your Git directory. Checkout any existing version, make changes, stage them and commit.



V. What is GitHub?

GitHub is a Git repository hosting service that provides a web-based graphical interface. It is the world's largest coding community. Putting a code or a project into GitHub brings it increased, widespread exposure. Programmers can find source codes in many different languages and use the command-line interface, Git, to make and keep track of any changes.

GitHub helps every team member work together on a project from any location while facilitating collaboration. You can also review previous versions created at an earlier point in time.

GitHub's Features:

1. Easy Project Management

GitHub is a place where project managers and developers come together to coordinate, track, and update their work so that projects are transparent and stay on schedule.

2. Increased Safety with Packages

Packages can be published privately, within the team, or publicly to the open-source community. The packages can be used or reused by downloading them from GitHub.

3. Effective Team Management

GitHub helps all the team members stay on the same page and organized. Moderation tools like Issue and Pull Request Locking help the team to focus on the code.

4. Improved Code Writing

Pull requests help the organizations to review, develop, and propose new code. Team members can discuss any implementations and proposals through these before changing the source code.

5. Increased Code Safety

GitHub uses dedicated tools to identify and analyze vulnerabilities to the code that other tools tend to miss. Development teams everywhere work together to secure the software supply chain, from start to finish.

6. Easy Code Hosting

All the code and documentation are in one place. There are millions of repositories on GitHub, and each repository has its own tools to help you host and release code.

VI. Alternatives for GitHub:

The market provides many alternatives and competitors for GitHub. Few of them are

1. Bitbucket
2. Google Cloud Source Repositories
3. Phabricator
4. GitLab
5. Gog's
6. Giteau
7. Source Forge
8. Apache Allura
9. Launchpad
10. AWS Code Commit

VII. Difference between Git and GitHub

S.No	Git	GitHub
1.	Git is a software.	GitHub is a service.
2.	Git is a command-line tool	GitHub is a graphical user interface
3.	Git is installed locally on the system	GitHub is hosted on the web
4.	Git is maintained by linux.	GitHub is maintained by Microsoft.
5.	Git is focused on version control and code sharing.	GitHub is focused on centralized source code hosting.
6.	Git is a version control system to manage source code history.	GitHub is a hosting service for Git repositories.
7.	Git was first released in 2005.	GitHub was launched in 2008.
8.	Git has no user management feature.	GitHub has a built-in user management feature.
9.	Git is open-source licensed.	GitHub includes a free-tier and pay-for-use tier.
10.	Git has minimal external tool configuration.	GitHub has an active marketplace for tool integration.
11.	Git provides a Desktop interface named Git Gui.	GitHub provides a Desktop interface named GitHub Desktop.
12.	Git competes with CVS, Azure DevOps Server, Subversion, Mercurial, etc.	GitHub competes with GitLab, Git Bucket, AWS Code Commit, etc.
13	Git is a version control system that lets you manage and keep track of your source code history	GitHub is a cloud-based hosting service that lets you manage Git repositories

SOFTWARE ENGINEERING

UNIT – 2

CHAPTER – 4

GIT COMMANDS: WORKING WITH LOCAL AND REMOTE REPOSITORIES

I. Git Commands

git version

The command git init is used to check the version of git.

git --version

git config

i. The git config command is used initially to configure the user.name and user.email. This specifies what email id and username will be used from a local repository.

ii. When git config is used with --global flag, it writes the settings to all repositories on the computer.

git config --global user.name “any user name”

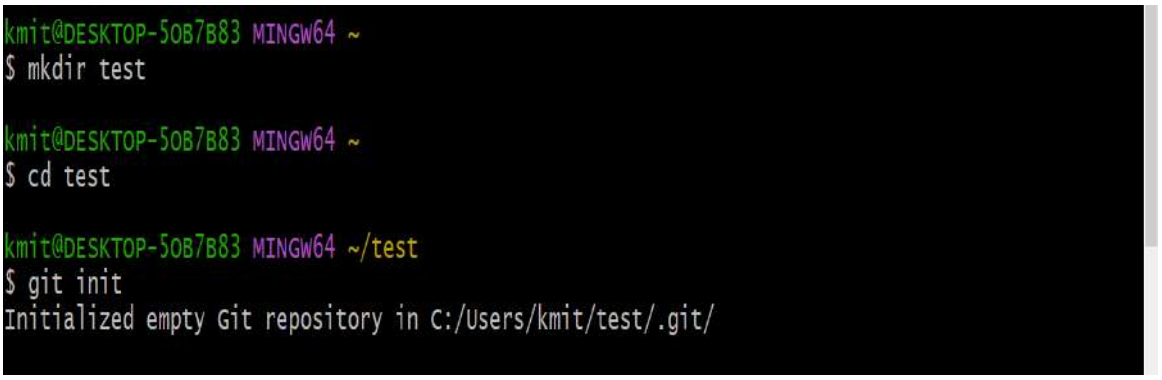
git config --global user.email <email id>

git init

i. The command git init is used to create an empty Git repository.

ii. After the git init command is used, a .git folder is created in the directory with some subdirectories. Once the repository is initialized, the process of creating other files begins.

git init



git status

i. The git status command tells the current state of the repository.

ii. The command provides the current working branch. If the files are in the staging area, but not committed, it will be shown by the git status. Also, if there are no changes, it will show the message no changes to commit, working directory clean.

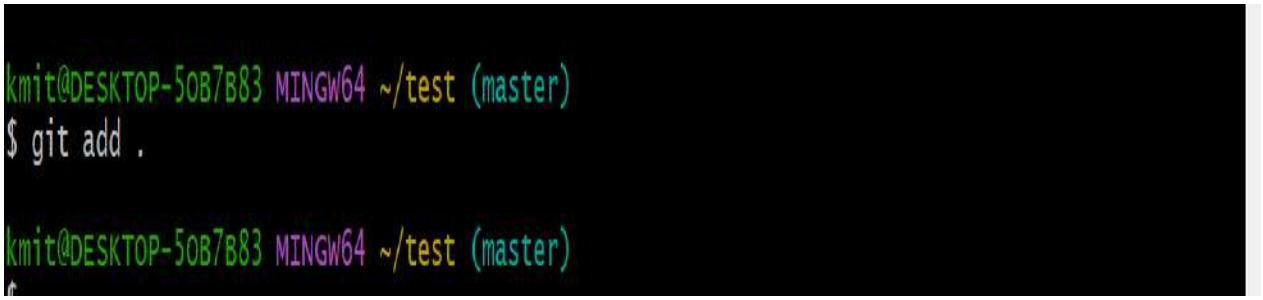
git status

git add

i. New files of the project are created and the Add command is used to add those files to the staging area.

ii. Before running the commit command, "git add" is used to add any new or modified files.

git add .



git commit

i. The commit command makes sure that the changes are saved to the local repository.

ii. The command "git commit –m <message>" allows you to describe everyone and help them understand what has happened.

git commit -m “commit message”

```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git add .

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git commit -m "info file added"
[master (root-commit) cf73846] info file added
 1 file changed, 3 insertions(+)
 create mode 100644 info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$
```

git diff

i. The git diff command is used to track the changes made to the file after the last commit

ii. Apply changes to one of the previously created file and check the diff command

```
MINGW64:/c/Users/kmit/test
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git commit -m "info file added"
[master (root-commit) cf73846] info file added
 1 file changed, 3 insertions(+)
 create mode 100644 info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ notepad info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git diff
diff --git a/info.txt b/info.txt
index 31c4db7..4111fbe 100644
--- a/info.txt
+++ b/info.txt
@@ -1,3 +1,6 @@
 This is my new file
 username : savram674
-email : savitharamesh674@gmail.com
\ No newline at end of file
+email : savitharamesh674@gmail.com
+
+Department :Information Technology
+Designation: Assistant Professor
\ No newline at end of file

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$
```

I Branches in Git

One of the key features of Git is its ability to manage branches. A branch in Git is a separate line of development that allows you to work on different features, bug fixes, or experiments without affecting the main codebase.

Overview of Git Branches:

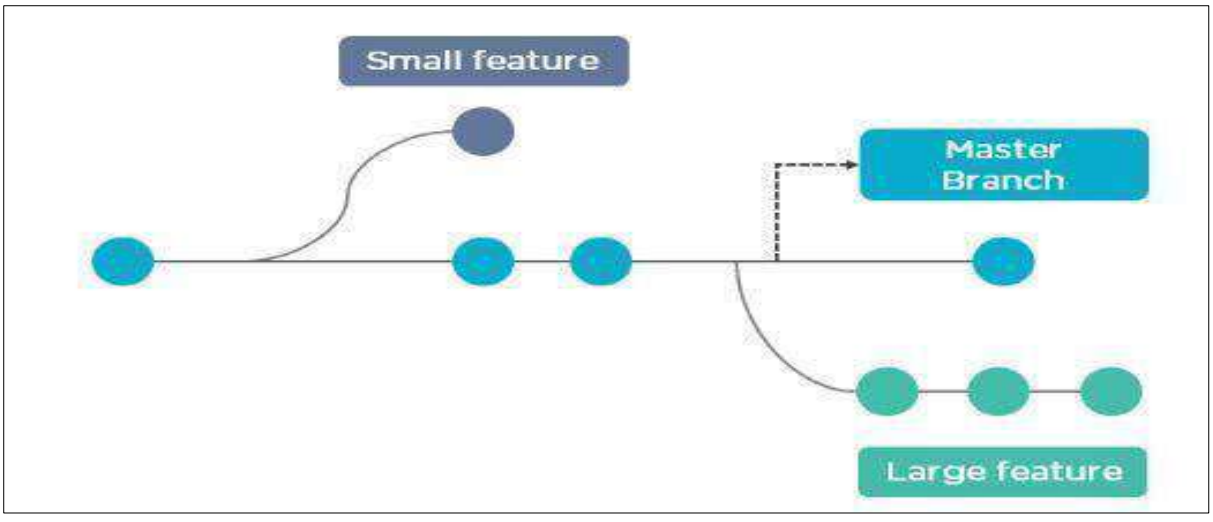
1. **Main/Branch (often referred to as 'master'):** This is the default branch where the stable and production-ready code resides. It typically represents the latest released version of the software

2. **Feature Branches**: When working on a new feature or functionality, developers create a new branch from the main branch. This branch is dedicated to developing and testing that specific feature. Once the feature is complete, the branch can be merged back into the main branch

3. **Bug Fix Branches:** Similar to feature branches, bug fix branches are created to address specific issues or bugs in the codebase. Developers fix the issue in this branch and then merge it back to the main branch.

4. **Release Branches:** These branches are used when preparing for a new release. They allow for final testing and bug fixing before the new version is deployed. Once everything is ready, the release branch is merged into the main branch, and often into other active development branches

Branch in Git is used to keep your changes until they are ready. You can do your work on a branch while the main branch (master) remains stable. After you are done with your work, you can merge it with the main office.



There are two separate branches called “small feature” and “large feature.” Once you are finished working with the two separate branches, you can merge them and create a master branch.

II. Workflow of Git Branches

i. Create a new branch from the main branch
ii. Make changes, develop features, or fix bugs in the branch
iii. Regularly commit changes to the branch
iv. Test the changes thoroughly in the branch
v. Once the work is complete, merge the branch back into the main branch
vi. Resolve any merge conflicts if they arise during the merge process
vii. Delete the branch if it's no longer needed

Branches provide a structured way to work collaboratively on different parts of a project without interfering with each other's work. They also enable versioning and history tracking, making it easier to understand the evolution of the codebase over time.

III. Git commands to deal with branch:

git branch

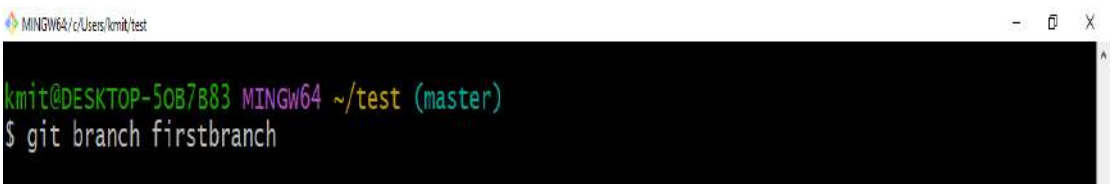
i. Git branch provides an isolated environment for change to the codebase (when working on new features or bug fixing)

ii. The git branch command is used to determine what branch the local repository is on.

iii. The command enables adding and deleting a branch.

Create a new branch

git branch <branch_name>



List all remote or local branches

git branch –a

```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (firstbranch)
$ git branch -a
* firstbranch
  master

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (firstbranch)
$
```

Delete a branch

git branch -d <branch_name>

git checkout

i. The git checkout command is used to switch branches, whenever the work is to be started on a different branch.

ii. The command works on three separate entities: files, commits, and branches.

Checkout an existing branch

git checkout <branch_name>

```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git branch firstbranch

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git checkout firstbranch
Switched to branch 'firstbranch'
M       info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (firstbranch)
$ ls
info.txt  info1.txt  info2.txt  info3.txt
```

Checkout and create a new branch with that name

git checkout -b <new_branch>

```
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git branch -a
  firstbranch
* master

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git checkout -b secondbranch
Switched to a new branch 'secondbranch'

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (secondbranch)
$ git branch -a
  firstbranch
  master
* secondbranch

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (secondbranch)
$
```

git merge

In Git, the **merge** command is used to integrate changes from one branch into another. Merging combines the changes made in one branch with another, allowing you to bring in new features, bug fixes, or updates from one branch to another.

It is used to merge the changes in the staging branch to the stable branch.

git merge <branch_name>

```
MINGW64:/c/Users/kmit/test
kmit@DESKTOP-5OB7B83 MINGW64 ~/test (firstbranch)
$ ls
info.txt  info1.txt  info2.txt  info3.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (firstbranch)
$ git checkout master
Switched to branch 'master'
M       info.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ ls
info.txt  info1.txt  info2.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ git merge firstbranch
Updating cf73846..08aedce
Fast-forward
 info3.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 info3.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$ ls
info.txt  info1.txt  info2.txt  info3.txt

kmit@DESKTOP-5OB7B83 MINGW64 ~/test (master)
$
```

git revert:

In Git, the **revert** command is used to create a new commit that undoes the changes made in a previous commit. This is useful when you want to reverse the effects of a specific commit without removing it from the commit history entirely. Unlike **git reset**, which can be used to discard commits, **git revert** maintains a clear history of changes by adding new commits that undo the changes from the selected commit.

1. Identify the Commit: First, you need to identify the commit you want to revert. You can use commands like **git log** to view the commit history and find the commit hash of the target commit.

2. Perform the Revert: Once you have the commit hash, use the **git revert** command followed by the commit hash.

git revert <commit_hash>

Example: git revert abcdef123

This will create a new commit that undoes the changes introduced in the specified commit.

3. Edit the Revert Message: Git will open an editor to let you modify the commit message for the new revert commit. The default message usually explains that you're reverting the specified commit.

4. Save and Close the Commit Message: After editing the commit message, save and close the editor to complete the revert commit.

5. Push the Revert Commit: If you're working in a shared repository, you should push the new revert commit to the remote repository to ensure that other team members see the change.

Fork:

In Git, a "fork" refers to a copy of a repository from one user's account to another user's account on a platform like GitHub. Forking is commonly used to contribute to open-source projects or collaborate on projects where you don't have direct write access to the original repository. Forking allows you to work independently.

Creating a Fork:

a. Visit the original repository's page on the Git hosting platform (like GitHub)
b. Click the "Fork" button usually located at the top right corner of the repository page
c. Choose your user account or the organization where you want to fork the repository

Result of Forking:

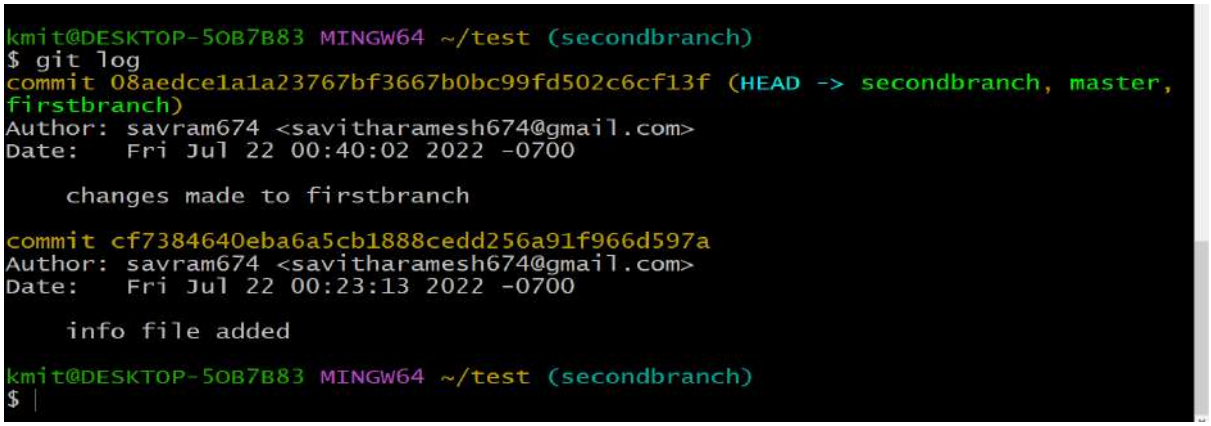
a. Choose your user account or the organization where you want to fork the repository
b. You'll have read and write access to your forked repository
c. Changes you make in your fork won't affect the original repository until you explicitly create a pull request (PR) to merge your changes

git log

a. The git log command shows the order of the commit history for a repository.

b. The command helps in understanding the state of the current branch by showing the commits that lead to this state.

git log



git help

The git help command is used to display the manual page for the specified command

git help add or git add –help


```
SSPL-LP-DNS-YT0+Simplilearn@SSPL-LP-DNS-YT01 MINGW64 ~
$ git help add
```

GIT COMMANDS: WORKING WITH REMOTE REPOSITORIES

I. Git Commands to work with remote repository:

git remote

i. The git remote command is used to create, view, and delete connections to other repositories.

ii. The connections here are not like direct links into other repositories, but as bookmarks that serve as convenient names to be used as a reference.

git remote add origin <address>

MINGW64:/c/Users/Madhu/Desktop/sample

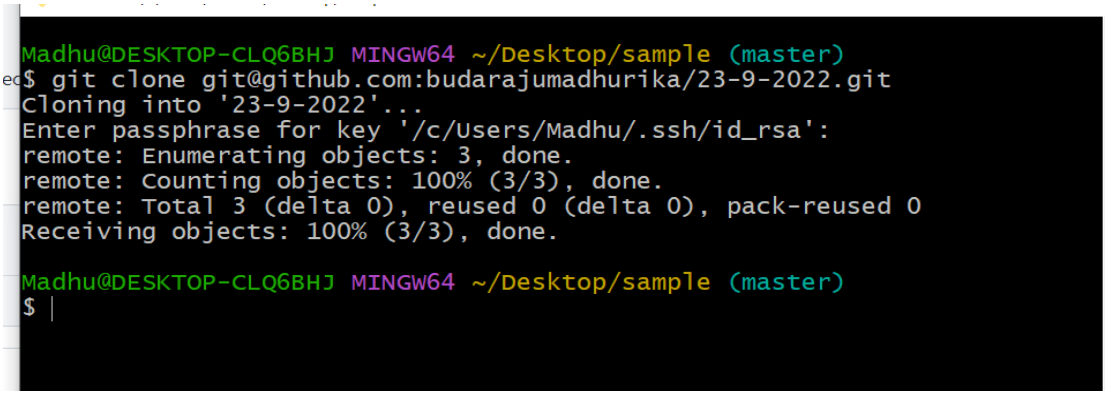
```
Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/sample (master)
$ git remote add origin https://github.com/budarajumadhurika/9-15.git

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/sample (master)
$ git remote -v
origin  https://github.com/budarajumadhurika/9-15.git (fetch)
origin  https://github.com/budarajumadhurika/9-15.git (push)

Madhu@DESKTOP-CLQ6BHJ MINGW64 ~/Desktop/sample (master)
$
```

git clone

i. The git clone command is used to create a local working copy of an existing remote repository.

ii. The command downloads the remote repository to the computer. It is equivalent to the Git init command when working with a remote repository.

git clone <remote URL>

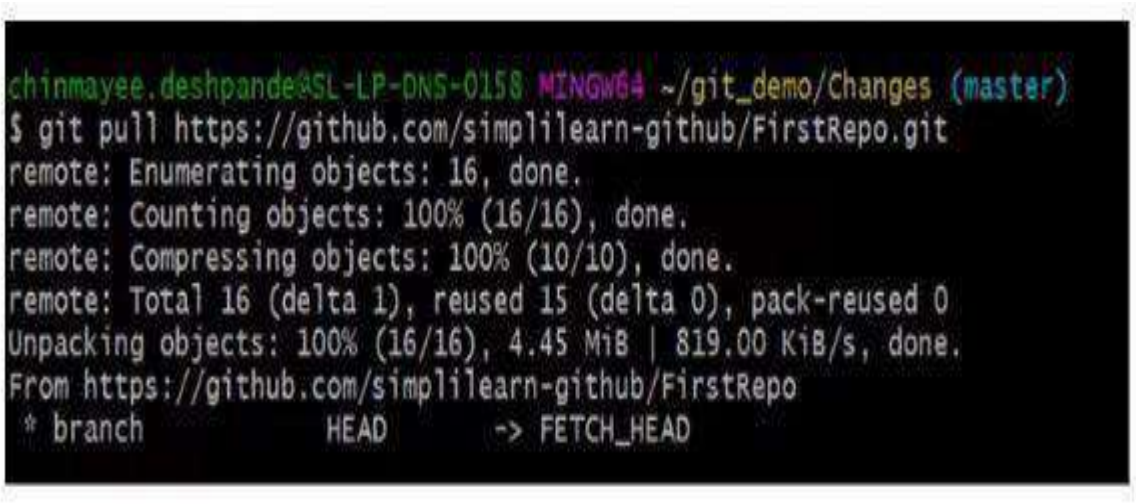


git pull

i. The git pull command is used to fetch and merge changes from the remote repository to the local repository.

ii. The command "git pull origin master" copies all the files from the master branch of the remote repository to the local repository.

git pull <branch_name> <remote URL>

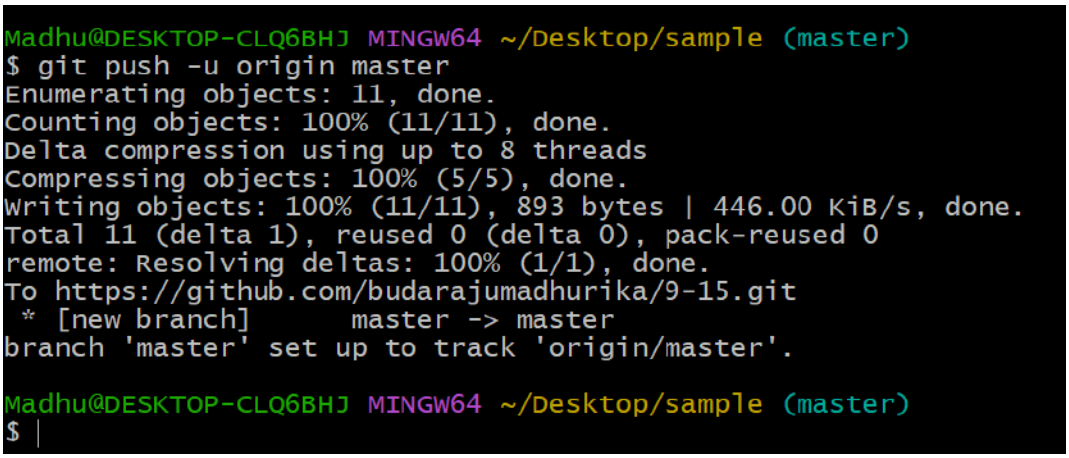


git push

i. The command git push is used to transfer the commits or pushing the content from the local repository to the remote repository.

ii. The command is used after a local repository has been modified, and the modifications are to be shared with the remote team members.

git push -u origin master



Collaborative Work:

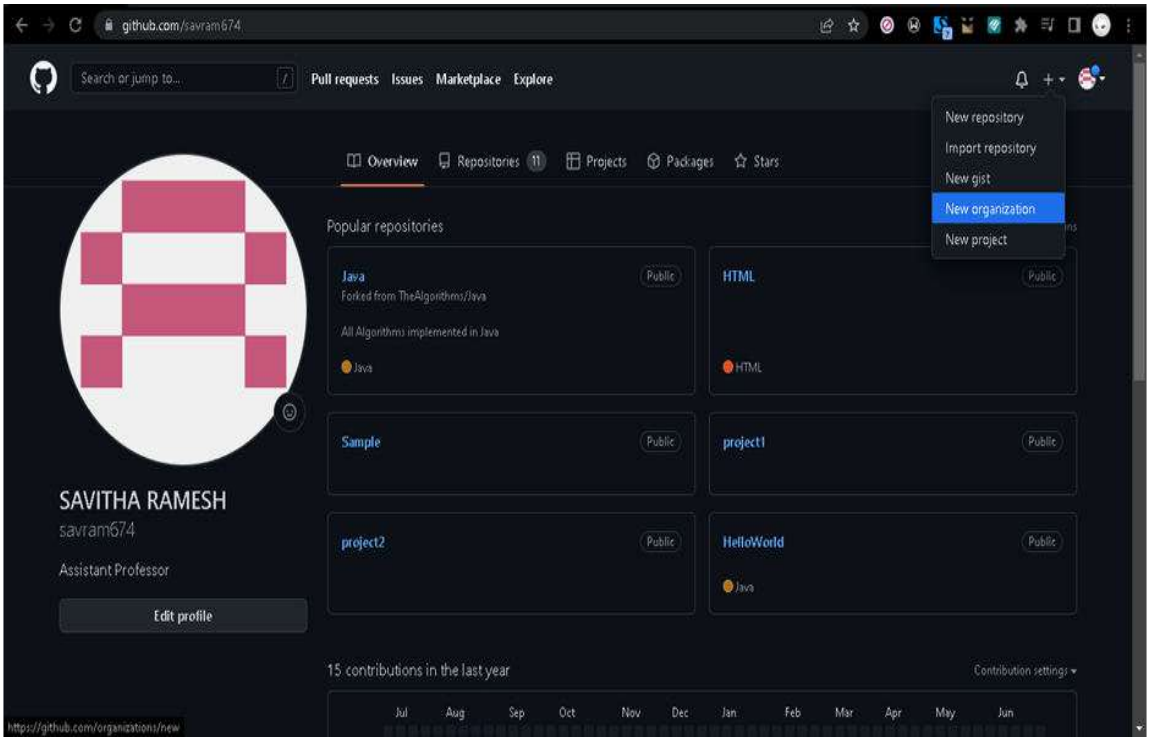
Collaboration is the way different people can work on the same project together. It is like creating a group in GitHub just like Groups in other social media. The people added to the collaborator's list can be able to push, merge, and do other kinds of similar things on the project.

We can collaborate in two ways:

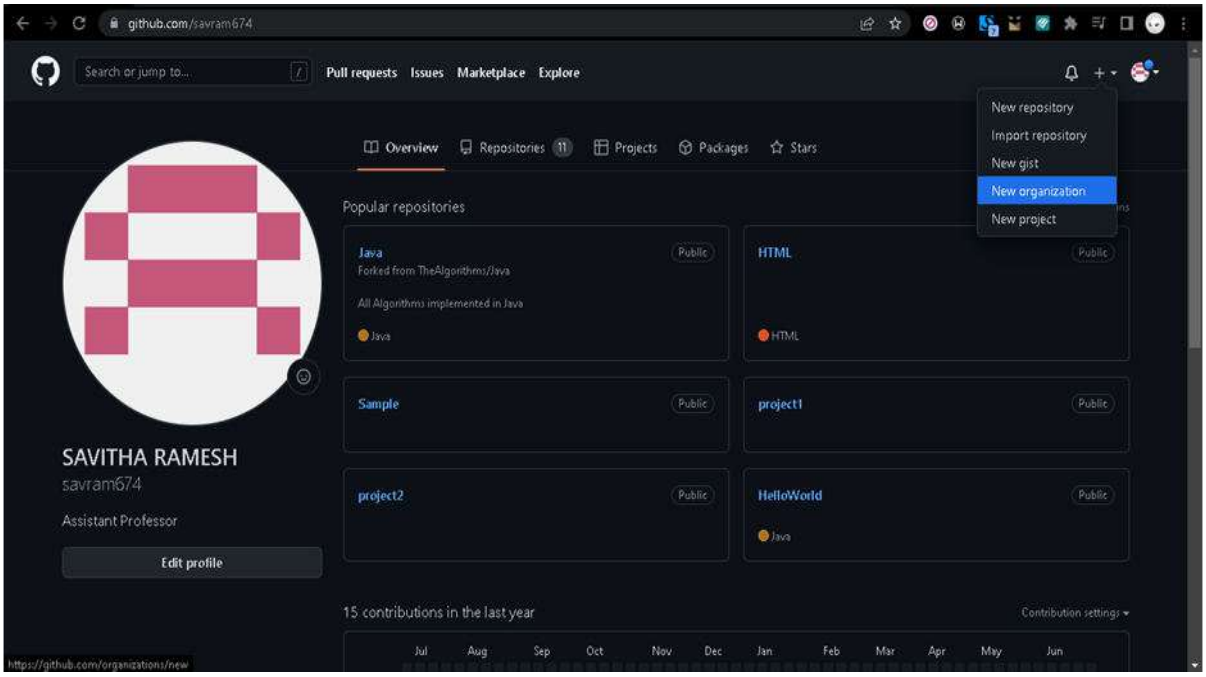
1. **Create a new repository – Invite Collaborators**
2. **Create a new organization – Add team members**

Note: Organization can allow multiple repositories to be included with different people having access to different (even multiple) repositories as described by the owner.

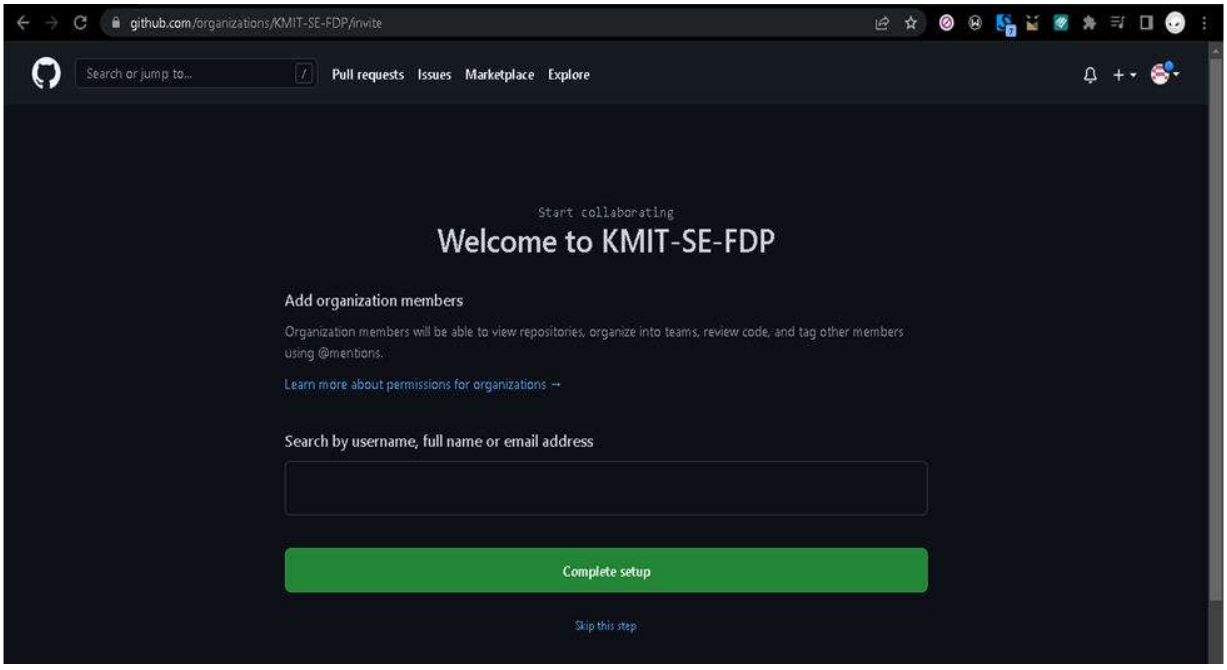
Step 1: Create a new Organization - Organizations are shared accounts where businesses and open-source projects can collaborate across many projects at once. Owners and administrators can manage member access to the organization's data and projects with sophisticated security and administrative features.



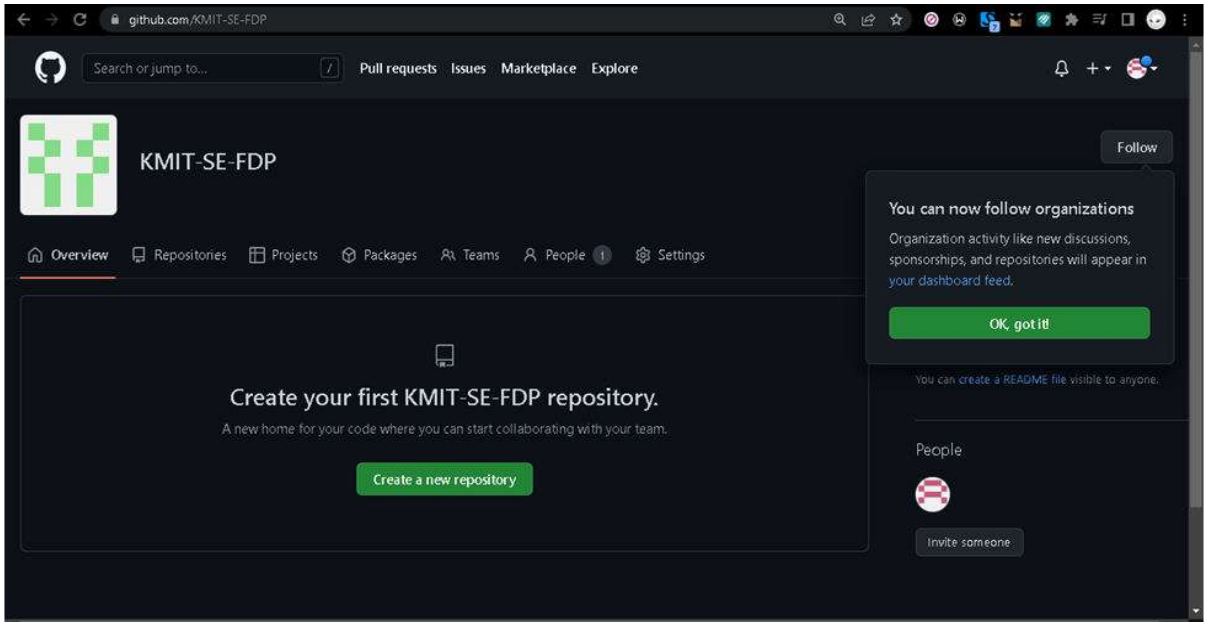
Step 2: Set up your organization by entering the details like name, associated email, account verification, inviting or adding members to the organization etc.



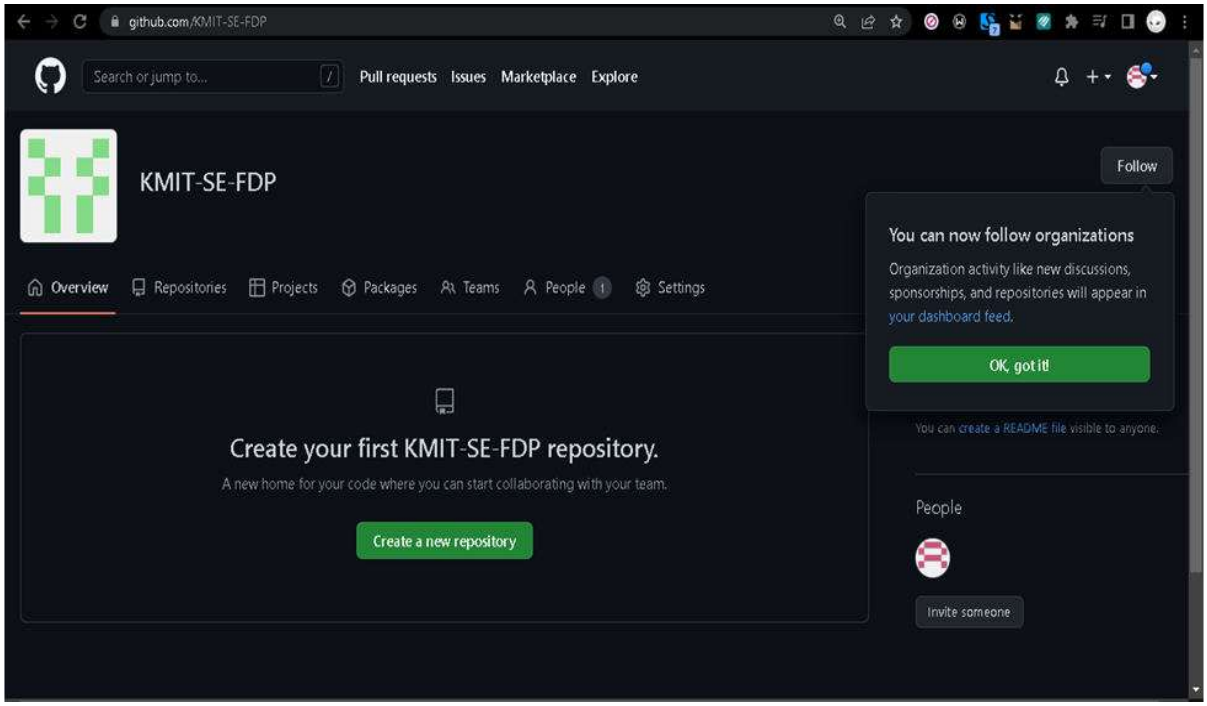
Step 3: Complete the setup by adding members to the group



Step 4: Create the remote repository for storing the project related files. This repository is accessible to every member of the team as per the permissions given.



The repository can be made private so that it is accessible only to the group members rather than being in a public domain.



Now all the members of the team can contribute to the development of the project and the different files with all the versions and modification notices will be available in the respective repositories and is accessible to all the members.