SOFT2412 – Agile Software Development Practices

ASSESSMENTS

Assessment Detail	Type	Value
Weekly Lab Activities	Individual	5%
Group Project 1	Group	15%
Group Project 2	Group	25%
Quiz	Individual	5%
Exam	Individual	50%

WEEK 1

Introduction

Software engineering is a systematic and disciplined approach to designing, developing, and maintaining software. It involves the application of engineering principles, methodologies, and best practices to create high-quality software systems that meet specific requirements, are reliable, maintainable, and cost-effective. Software engineering encompasses a wide range of activities, including requirements analysis, design, coding, testing, maintenance, and project management. Following are some reasons why Software Engineering is necessary:

- Structured Development:
  Software engineering provides a structured and organized way to develop software. It involves following a set of well-defined processes, methodologies, and best practices to ensure that software projects are managed efficiently and produce reliable results.
- Quality Assurance:
  Software engineering emphasizes quality throughout the software development lifecycle. This includes techniques for testing and verification to identify and fix defects early in the development process, ultimately leading to more reliable and robust software.
- Efficiency:
  By following established engineering principles and practices, software engineering helps optimize the use of resources, both human and technological. This efficiency is crucial in managing costs and meeting project deadlines.
- Requirements Management:
  Software engineering places a strong emphasis on understanding and managing requirements. This ensures that the final software product meets the needs and expectations of the end-users or stakeholders.
- Scalability and Maintainability:
  Properly engineered software is designed to be scalable and maintainable. It can be adapted to changing requirements and can be extended or modified without significant disruptions or additional costs.
- Risk Management:
  Software engineering techniques help in identifying, analysing, and mitigating risks associated with software development projects. This can prevent costly delays or failures.
- Reusability:
  Software engineering encourages the reuse of components and code libraries, reducing the need to reinvent the wheel for every project. This not only saves time but also enhances software quality and consistency.

- Team Collaboration:

  Software engineering promotes collaboration within development teams. It encourages documentation, clear communication, and the use of standardized processes and tools, which are vital for large, complex projects.
- Customer Satisfaction:

  By focusing on meeting user needs and delivering high-quality software, software engineering contributes to customer satisfaction and the success of the software product.

Software is present in virtually every aspect of modern life. It plays a crucial role in various industries and aspects of daily living. Softwares are present in computers, mobile devices, web and internet, embedded systems, gaming, business and finance, healthcare, aerospace and defence, transportation, entertainment, education, smart home devices, energy and utilities, scientific research, and much more.

In a world where software is increasingly pervasive and critical in various industries, such as healthcare, finance, transportation, and entertainment, the need for software engineering becomes even more apparent. Without the discipline of software engineering, the risk of developing unreliable, insecure, and inefficient software is much higher. To ensure that software systems meet their intended goals and function effectively, software engineering principles and practices are essential. It is also required to satisfy user needs, reduce costs, open new opportunities, etc.

Software Failure – Ariane 5 Disaster

The Ariane 5 disaster, also known as the "Ariane 501" incident, is one of the most well-known and significant failures in the history of space exploration. It occurred on June 4, 1996, just 40 seconds after the launch of the Ariane 5 rocket from the Guiana Space Centre in French Guiana. The rocket experienced a catastrophic failure and had to be destroyed, resulting in the loss of the payload and a cost of approximately $500 million. The primary cause of the Ariane 5 disaster was a software error in the rocket's inertial reference system. Here's an elaboration of the key factors that led to the failure:

- Incompatibility of Software:

  The Ariane 5 rocket was a new and significantly more powerful version of its predecessor, the Ariane 4. The guidance and control software of the Ariane 5 was inherited from the Ariane 4, which used a 64-bit floating-point number to represent horizontal velocity. However, during the rocket's maiden flight, the software tried to convert a 64-bit floating-point number to a 16-bit signed integer, which resulted in an overflow error. This error was due to a lack of compatibility between the two versions of the rocket.
- Inadequate Error Handling:

  The software did not handle this exception properly. When the overflow occurred, it caused a fault in the Inertial Reference System (SRI), which was one of the rocket's critical subsystems responsible for providing data on the rocket's orientation and velocity. This fault subsequently led to the rocket's guidance system being disabled.
- The Lack of a Software Safeguard:

  The original software design did not include a safeguard or a software fail-safe mechanism to prevent this particular error from causing a catastrophic failure. The older Ariane 4 rockets had a different flight profile and were less susceptible to this specific issue.

- Short Flight Duration:

  The failure occurred just 40 seconds into the flight, during a phase when the rocket was subjected to extreme aerodynamic forces and high velocities, making it impossible to recover from the anomaly.

As a result of these software-related issues, the rocket lost its guidance and control capabilities, deviated from its intended flight path, and had to be destroyed in mid-air to avoid posing a threat to populated areas.

The Ariane 5 disaster highlighted the critical importance of thorough software testing and validation in complex systems, especially in high-stakes domains like aerospace. It also led to significant changes in the way software is developed and verified for use in safety-critical applications. This incident emphasized the need for rigorous software engineering practices, robust error handling, and redundancy in critical systems to prevent similar catastrophic failures in the future.

Software Developers vs Software Engineers

The terms "software developer" and "software engineer" are often used interchangeably, and there is some overlap in their roles and responsibilities. However, in practice, there are distinctions that can vary depending on the context, the organization, and the specific tasks involved. Here are the key differences between software developers and software engineers:

- Education and Training:
  - Software Developer:

    Software developers may have varying levels of formal education, ranging from self-taught programmers to individuals with formal degrees in computer science or related fields. They often focus on writing code to meet specific project requirements.
  - Software Engineer:

    Software engineers typically have formal training in computer science, software engineering, or a related discipline. They are more likely to have a structured education in engineering principles and best practices.
- Responsibilities:
  - Software Developer:

    Software developers primarily focus on coding and implementing software solutions based on specifications provided by others. They often work on specific tasks within a larger project, such as creating features or modules.
  - Software Engineer:

    Software engineers take a broader view of the software development process. They are involved in designing the architecture, making high-level decisions about the technology stack, and ensuring that the software aligns with best engineering practices and standards. They may also be responsible for system design, testing, and quality assurance.
- Design and Architecture:
  - Software Developer:

    Developers typically work with predefined system architectures and designs. They implement solutions based on the specifications provided to them.
  - Software Engineer:

Software engineers are more likely to be involved in system and software architecture. They may participate in the design process and help shape the overall structure of the software to meet business and technical goals.

- Coding vs. Systems Thinking:
  - Software Developer:
    Developers often focus on the coding aspect, ensuring that the code they write meets functional requirements and is efficient. They may not always consider the broader system context.
  - Software Engineer:
    Software engineers have a system thinking approach. They consider the entire software development lifecycle, including scalability, maintainability, and long-term system health.
- Quality Assurance and Testing:
  - Software Developer:
    Developers typically write unit tests and perform some level of testing to verify that their code works as intended.
  - Software Engineer:
    Software engineers may be more involved in defining testing strategies, coordinating integration testing, and ensuring the overall quality of the software.
- Documentation and Communication:
  - Software Developer:
    Developers usually provide documentation for their code and communicate with team members about specific coding tasks.
  - Software Engineer:
    Software engineers often need to create and maintain comprehensive system documentation, communicate with various stakeholders, and ensure that the software aligns with business and technical requirements.

Software Processes

A software process, often referred to as a software development process or software engineering process, is a structured set of activities, tasks, and steps that organizations and teams follow to design, develop, test, deploy, and maintain software systems and applications. The purpose of a software process is to provide a systematic and organized approach to software development to ensure the delivery of high-quality, reliable, and maintainable software within time and budget constraints. Below is a simplified breakdown of these phases, which are present in most software engineering operating models:

- Planning (Requirement Analysis)
  During this phase, the software delivery and business teams collect information to outline the product's business objectives. The team creates a Software Requirements Specification (SRS) — a document that includes the objectives, goals, and system requirements for the product. This document should also document key agreements between stakeholders. The requirements-gathering process helps to align business objectives with the product being developed.
- Prototype design
  During this stage, the software delivery team defines how the project will be developed and reflects this information in the Design Document Specification (DDS). This document covers the frameworks, platform specifications, limitations, and delivery estimates. Ideally, this documentation is made available to all parties involved in the development process (developers, testers, managers, and the client).
- Development (Implementation)

This is the core phase of all software process models, during which engineers develop the solution according to the requirements. This step involves a lot of code-writing and fine-tuning.

- Quality Assurance and testing

  This phase consists of code reviews and module assessment. It's a continuous process, especially for the Agile life cycle model. When people talk about testing, they usually refer to unit and integration testing. Unit testing verifies specific sections of the app to ensure it works as planned, while integration testing ensures that all modules work as one system.

- Delivery

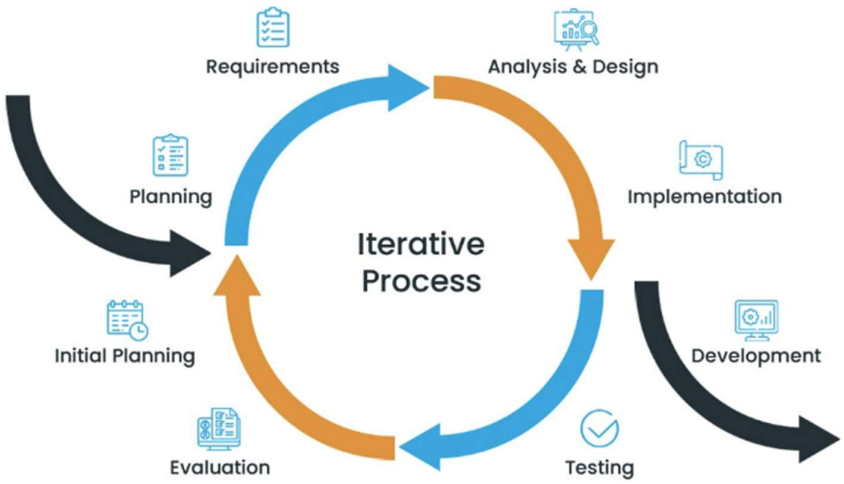
  Once engineers and testers are done with their work, the project manager, the client, and the stakeholders must approve the project. After they do, it's time for the product to hit the market.

- Maintenance

  Many models include the post-deployment software development modelling phase. In fact, most applications require regular maintenance to remain relevant. This phase covers compatibility upgrades, performance optimization, and monitoring, as well as bug fixes.

The SDLC process of software development will help in delivering the project on schedule and in budget. It helps the project stay on track, maximises the productivity, reduces the development time and provides better management. There are different types of SDLC models, a few given below:

- The iterative model



Iterative software process models begin with a smaller set of requirements. Developers use these requirements to analyse and gradually implement features. Afterward, these features are evaluated and tested to identify new requirements. This cycle is called an "iteration". This life cycle repeats numerous times, introducing new design choices, coding, and requirements changes. It is suitable for large scale projects with multiple modules and for projects with clearly defined objectives and tasks.

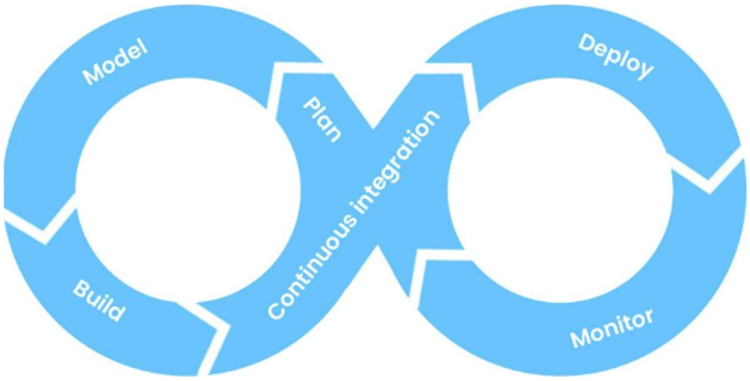
Advantages:

- Coding begins at the start.
- Cost effective way of implementing requirement changes.
- Streamlined management as production is divided into smaller pieces.
- Bugs and defects are easier to spot at earlier phases.

Disadvantages:

- Difficult to analyse risk.
- Potential design issues in later stages
- Too reliant on baseline plan
- A resource heavy model

- The DevOps models.



DevOps is built around the principles of automation and collaboration. This model's primary goal is to enhance cooperation between the operations team and developers thanks to continuous feedback. It is suitable for complex projects which require a lot of QA and testing. It is also recommended for large teams with multiple departments.

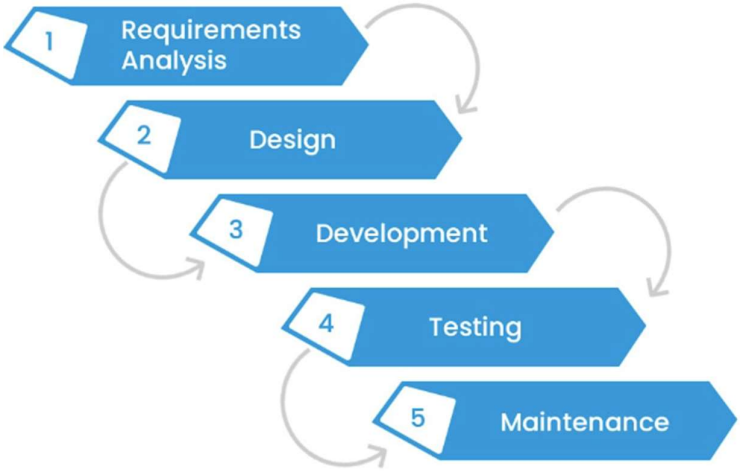
Advantages:

- Automation and optimization of processes
- Continuous feedback cycle between engineers and testers
- Streamlined product delivery.
- Productivity improvements in house
- Early error and defect detection

Disadvantages:

- Lack of focus on documentation
- Difficult to manage emerging product features.
- A challenging adoptive curve

- The waterfall models.



Waterfall is one of the oldest types of software development life cycle methodologies. It focuses on a linear approach with SDLC phases following one another sequentially. The outcome of each phase affects the input of the next one.

Therefore, it's much more difficult and costly to return to earlier phases. It is suitable for the smaller and mid-sized projects with clearly defined requirements.

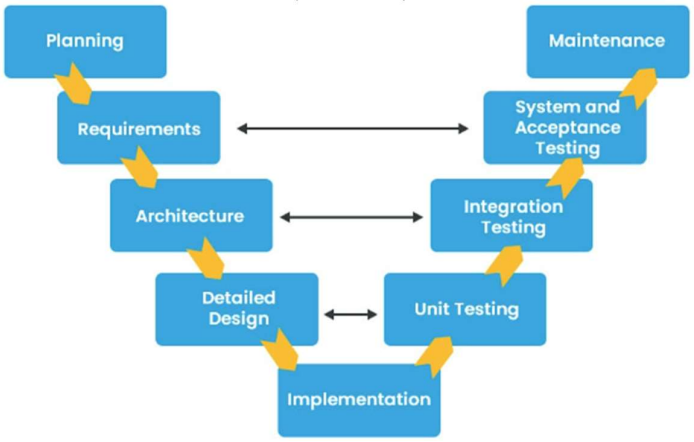
Advantages:

- Straightforward and easily manageable
- Clearly defined tasks and milestones
- Easy to prioritize tasks.

Disadvantages:

- Lacks the versatile nature of agile development.
- Phases can't overlap.
- It is time consuming.
- Too costly to return to prior phases.

- The verification and validation model (V-model)



Verification and Validation is a sequential methodology, just like Waterfall. But there's a difference — in this methodology, every step goes in parallel with testing. It is useful for mid-sized and large projects with explicit objectives and requirements.

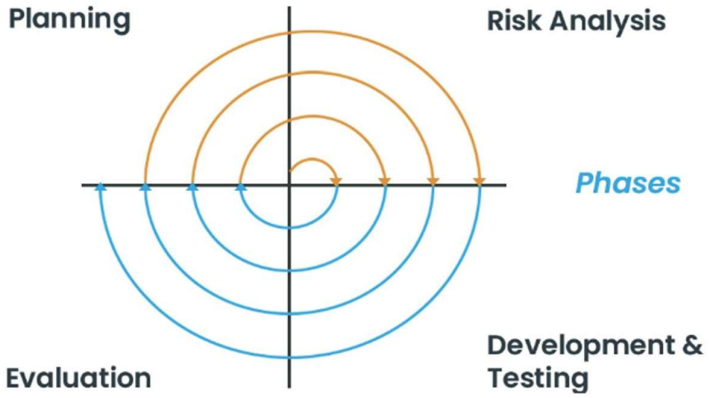
Advantages:

- Simple and straightforward
- Critical issues are taken care of in the initial phases.
- In depth requirement documentation

Disadvantages:

- Lack of flexibility
- Too costly and time consuming

- The Spiral model



Spiral combines the Iterative and Waterfall methodologies with an emphasis on risk assessment. It consists of the usual SDLC phases built on the baseline spiral. Each cycle repeats in "spirals" that allow teams to evaluate risks and get precise estimates.

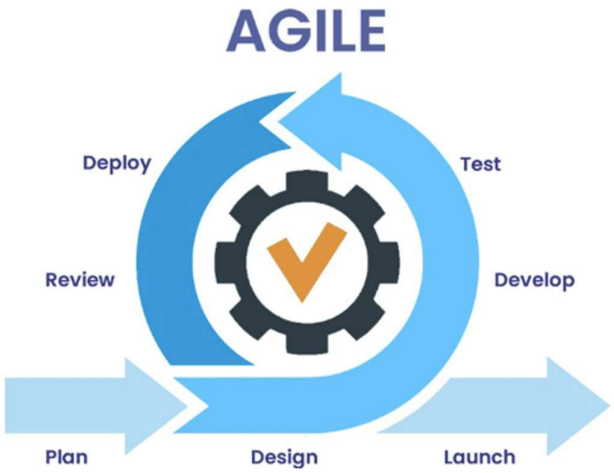
The loop repeats until the product complies with requirements. It is suitable for larger projects with complex requirements and new products with multiple testing stages.
Advantages:

- Precisely documented.
- Accurate time and budget estimates
- Excellent risk assessment
- Can apply new changes to new iterations.

Disadvantages:

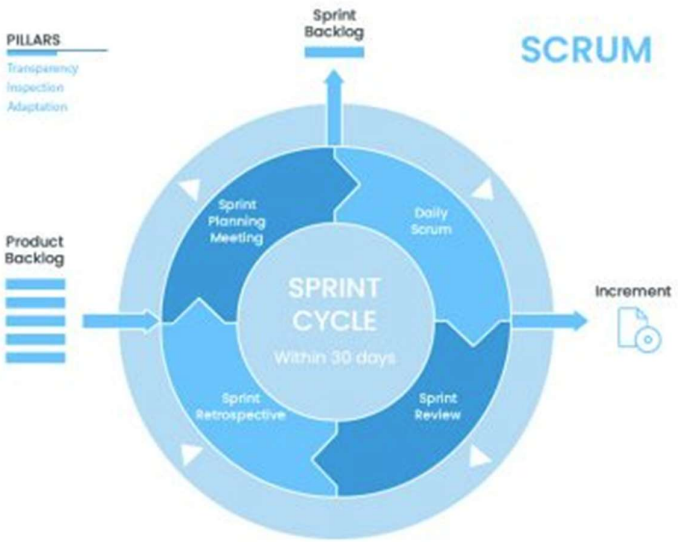
- Success depends on skilled risk managers.
- Requires a large resource pool.
- Time consuming

The Agile Model

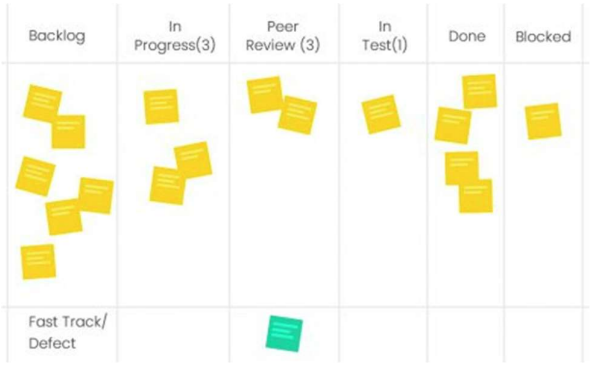


Agile focuses on continuous release cycles and cross-functional development. It's one of the most flexible and popular types of SDLC models. Agile is quite similar to Iterative, where teams work in repeated increments. However, unlike Iterative, Agile doesn't rely on the baseline plan. The plan, along with the requirements, is continuously modified throughout the life cycle. It is suitable for large scale or smaller projects. Best for outsourcing and managed IT services and for adding new features to a working prototype. There are 2 subdivisions to Agile:

- Scrum



- Kanban



Advantages:

- Easy to change requirements and baseline plan.
- Fast release of prototype
- Focus on communication between developer and client.
- Client's engagement integrated to the life cycle.
- Continuous evaluation and feedback

Disadvantages:

- Difficult to estimate final production costs.
- Possible architectural conflicts due to constant requirement changes.

Agile Manifesto

The Agile Manifesto is a set of guiding values and principles for Agile software development. It was created by a group of prominent software developers and thought leaders in 2001 at a gathering in Snowbird, Utah. The Agile Manifesto emphasizes flexibility, collaboration, customer-centricity, and the delivery of high-quality software. It has had a significant impact on the software development industry and has been influential in various other fields as well. The Agile Manifesto consists of four core values and twelve principles:

- Agile Manifesto Values
    - Individuals and interactions over processes and tools:

      Agile places a strong emphasis on the importance of people working together and communicating effectively. While processes and tools are necessary, the primary focus is on the individuals involved in the project.
    - Working software over comprehensive documentation:

      Agile values working, functional software as the primary measure of progress. While documentation is important, it should be concise and serve the needs of the project.
    - Customer collaboration over contract negotiation:

      Agile promotes close collaboration with customers and end-users to understand their needs, gather feedback, and adapt the software to their requirements. This is favoured over rigid contract negotiations.
    - Responding to change over following a plan:

      Agile acknowledges that change is inevitable in software development. It encourages teams to be adaptive and responsive to changing requirements, rather than rigidly adhering to a fixed plan.
- Agile Manifesto Principles
    - Satisfy the customer through early and continuous delivery of valuable software.

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, with a preference for shorter timescales.
- Collaborate with business stakeholders and developers throughout the project.
- Build projects around motivated individuals and give them the environment and support they need. Trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

WEEK 2

Software Artifacts

A "software artifact" refers to any tangible document or product created during the software development process. These artifacts serve various purposes, including communication, documentation, analysis, and as a basis for further development and testing. Software artifacts are essential for ensuring that the software is designed, developed, and maintained effectively and efficiently. Artifacts go through evolution; hence it is usually tracked. The artifacts have value and need to be preserved, communicated, maintained, protected from unauthorized access, etc. A few common software artefacts are:

- Software Requirement Specification

  A Software Requirements Specification (SRS), often referred to simply as a "requirements document," is a critical software artifact that serves as a formal and detailed description of the functional and non-functional requirements of a software system. It plays a fundamental role in the software development process, serving as a key reference document for all stakeholders involved in the project, including software developers, testers, project managers, and clients. In Agile, it is usually in the form of user stories.
- Code

  A "code" refers to the set of programming instructions and statements written in a specific programming language to create software applications. Code is one of the most fundamental and central software artifacts, and it serves as the blueprint for how a software system operates. A code is usually spread over different files and is in a directory structure. It usually requires all the files to work in collaboration with each other for the product to work as required. It can be written in many languages such as Java, Python, R, C, etc. but each language has its own conventions and requirements. Documentations may be derived from the source code.
- Version Control

  It is a tool, system, or process used to manage and track code artifacts, among other software artifacts. In other words, version control is a critical part of the development and maintenance of code artifacts.

Version Control System

A Version Control System (VCS), also known as a Source Code Management (SCM) system, is a software tool or platform that facilitates the management of changes to source code and other digital assets in a collaborative development environment. The primary purpose of a version control system is to track, record, and manage revisions of files and documents, making it easier for multiple developers to work on a project simultaneously while keeping track of changes and ensuring code and project history is well-organized. Key features of VCS include:

- Keep track of every modification to code in a repository.
- Revert selected files back to a previous state.
- Compare changes over time.
- See who last modified something that might be causing a problem.
- Who introduced an issue and when?
- Compare earlier versions of the code to help fix bugs while minimizing disruption to all team members.

There are two main types of Version Control Systems (VCS): centralized version control and distributed version control. Each type has its own characteristics and use cases.

- Centralized Version Control System (CVCS)

  In a centralized version control system, there is a central server that hosts the repository containing all the project's files and their versions. Developers check out copies of the files from the central repository to work on them, and when they make changes, those changes are committed back to the central server. A few characteristics of these include:
  - Central Repository: All project files and their versions are stored in a central repository on a server.
  - Check-Out and Check-In: Developers check out files from the central repository to work on them and then check them back in to commit changes.
  - Collaboration: Multiple developers can collaborate on the same project, but the central server is a single point of truth.
  - History and Tracking: The centralized server tracks changes, so it is possible to see who made changes and when.
  - Examples: Common centralized VCS systems include Subversion (SVN) and Perforce.
- Distributed Version Control System (DVCS)

  In a distributed version control system, every developer has a full copy of the repository, including the complete project history, on their local machine. This approach allows developers to work independently and commit changes to their local repository. Changes can then be synchronized and merged with other developers' repositories. Key characteristics of DVCS include:
  - Local Repositories: Each developer has a complete copy of the repository on their local machine.
  - Commit to Local Repository: Developers commit changes to their local repository.
  - Synchronization: Developers can synchronize changes with other repositories, often hosted on a central server or other team members' machines.
  - Flexibility: Developers can work offline, branch and experiment freely, and commit changes at their own pace.
  - Examples: The most popular distributed VCS system is Git, but others include Mercurial and Bazaar.

GIT fundamentals

Git is a distributed version control system (DVCS) that has become the most widely used system for tracking changes in source code and managing collaborative software development projects. Here are some key features and concepts associated with Git:

- Distributed Version Control:

  Git is a distributed version control system, which means that every developer working on a project has a complete copy of the repository on their local machine. This decentralization enables developers to work independently and even offline.
- Repositories:

  In Git, a repository is a collection of files and directories, along with the entire history of changes. There are often two types of repositories: the local repository on a developer's machine and the remote repository hosted on a server or platform like GitHub, GitLab, or Bitbucket.
- Commits:

  A commit in Git represents a specific set of changes to the repository. Each commit has a unique identifier (hash) and is associated with a commit message that explains the purpose of the change.
- Branches:

  Git allows developers to work on separate lines of development using branches. Branches can be created for specific features, bug fixes, or experiments. Merging or rebasing branches is a common operation in Git.
- Merge and Rebase:

  Merging and rebasing are two techniques for combining changes from one branch into another. Merging combines changes as a new commit, while rebasing replays changes on top of another branch's history.
- Pull Requests (GitHub/GitLab/Bitbucket):

  Platforms like GitHub, GitLab, and Bitbucket introduce the concept of pull requests (PRs) or merge requests (MRs). These are mechanisms for proposing and reviewing changes before they are merged into the main branch.
- Staging Area (Index):

  Git has a staging area where developers can select which changes to include in the next commit. This allows for fine-grained control over what is committed.
- Submodules and Subtrees:

  Git supports submodules and subtrees, which allow you to include other Git repositories within your repository. This is useful for managing dependencies.
- Git Hooks:
- Git hooks are scripts that can be triggered at specific points in the Git workflow, allowing you to automate tasks or enforce policies.
- Extensive Ecosystem:

  Git has a rich ecosystem of tools and services, including hosting platforms (GitHub, GitLab, Bitbucket), graphical user interfaces (like SourceTree), and integration with continuous integration and deployment (CI/CD) systems.
- Strong Cryptographic Hashing:

  Git uses cryptographic hashing to ensure the integrity of data. Every file and commit are identified by a SHA-1 hash, which makes it highly resistant to tampering.

The structure of GIT

Git has a structured architecture that enables it to manage source code and project history efficiently. Knowing the following three are very important:

- Working Directory (Tree)

  The working directory is a local directory on your machine where you make changes to project files. It is essentially a snapshot of the project's current state. It is a single checkout of one version of the project. These files are pulled out of the compressed database and placed on disk for you to use or modify.
- Staging Area (Index)

  The staging area, sometimes referred to as the index, is an intermediate area where you prepare changes before committing them. You can selectively choose which changes to include in the next commit. This is basically a file which stores information about what will go into the next commit.
- Git Directory (Repository)

  A Git repository is a directory that contains all the files, directories, and historical data for a project. It serves as the central store for project assets and history. It is the metadata and object database. This is what is copied when you clone a repository from another computer.

GIT Basic workflow

1. Initialize a Repository (*"git init"*):

   When you run git init, Git initializes a new repository in the current directory. The working directory is created and the .git directory is created inside it, which is your Git repository. The .git directory contains all the configuration and version control data.
2. Add and Edit Files (Working Directory):

   As you add or edit files in your project directory (the working directory), Git tracks these changes. However, they are not yet staged for a commit.
3. Add Changes to Staging Area (*"git add"*):

   To prepare changes for a commit, you use git add. This command stages specific changes or entire files from the working directory into the staging area. The staging area is like a temporary holding area for changes you want to include in your next commit.
4. Commit Changes (*"git commit"*):

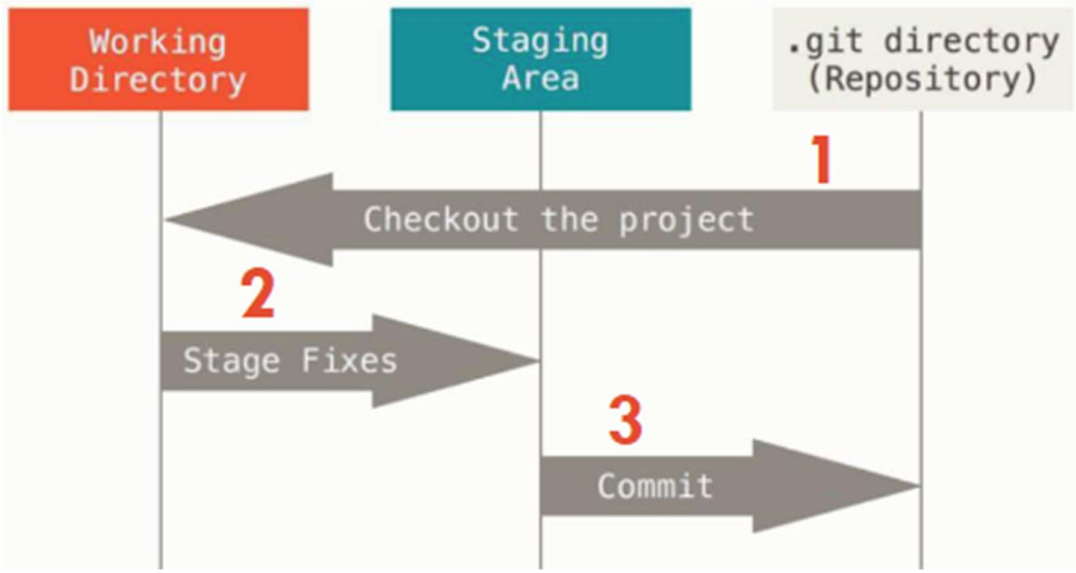
   After you've added changes to the staging area, you use git commit to create a new commit. The commit contains the changes you've staged, along with a commit message describing what you've done. Commits are stored in your local repository, creating a new snapshot of your project's state.
5. Create Branches (Branches in Git Repository):

   When you create branches using git branch, you are essentially creating pointers in the Git repository to specific commits. Branches allow you to work on different lines of development without affecting the main branch.
6. Add Remotes (*"git remote", "git clone"*):

   When you add a remote repository using git remote, this information is stored in your Git repository's configuration. It defines a connection to a remote server that

hosts a copy of the project. In the context of the working directory, these remotes are not directly related to it.

7. Push Changes (*"git push"*):
   When you push changes using git push, you are sending commits from your local repository to a remote repository. The working directory is not directly involved in this operation.
8. Pull Changes (*"git pull"*):
   To pull changes from a remote repository into your local repository, you use git pull. This operation updates the commits in your local repository, but the working directory remains unchanged.
9. Merge Changes (Working Directory):
   When you merge or rebase branches using git merge or git rebase, Git updates the working directory to reflect the changes from the merged or rebased branch.
10. Review and Collaborate (Working Directory):
    You review and collaborate in the working directory by using git log to view the commit history, git diff to see the differences between versions, and making changes in response to code reviews.
11. Resolve Conflicts (Working Directory):
    When resolving conflicts, you make changes directly in the working directory by editing conflicted files. You use git add and git commit to finalize the resolution. The staging area helps in resolving conflicts.
12. Tagging and Releases (Git Repository):
    You create tags using *"git tag"*. Tags are added to the Git repository and point to specific commits, marking important points in your project's history. The working directory is not directly involved in this process.
13. Cleanup and Maintenance (Git Repository):
    Commands like *"git clean"* and *"git gc"* are used for repository cleanup and maintenance. These commands affect the Git repository, not the working directory.



The working directory is where you make changes to project files and perform day-to-day development tasks. The staging area serves as an intermediate step to prepare changes for a commit. The Git repository stores the complete history of your project and is accessed by various Git commands to manage commits, branches, tags, and configuration settings.

GIT Commands

git init
Initializes a new Git repository in the current directory, creating a .git directory to store version control data.
git clone [repository_url]
Clones an existing Git repository from a remote server, creating a local copy of the repository.
git add [file_name]
Stages changes from the working directory to the staging area in preparation for a commit. Use git add . to stage all changes.
git commit – "[commit message]"
Creates a new commit with the staged changes, along with a descriptive commit message.
git status
Displays the current state of the working directory, showing untracked, modified, and staged files.
git log
Lists the commit history, displaying the commit messages, authors, and commit hashes.
git diff
Shows the differences between the working directory and the last committed version of the project.
git branch
Lists all local branches and indicates the current branch with an asterisk. Use git branch [branch_name] to create a new branch.
git checkout [branch_name]
Switches to a different branch, updating the working directory to match the selected branch.
git merge [branch_name]
Combines changes from the specified branch into the current branch. Use --no-ff for a non-fast-forward merge.
git pull
Fetches changes from a remote repository and merges them into the current branch. Equivalent to git fetch followed by git merge.
git push
Pushes local commits to a remote repository, updating the remote branch with your changes.
git tag [tag_name]
Creates a new tag (a reference to a specific commit), often used for marking releases or milestones.
git fetch
Fetches changes from a remote repository, updating your local repository with new commits, branches, and tags.
git rebase [branch_name]
Replays changes from the current branch on top of the specified branch. Useful for keeping a linear project history.
git reset [commit]
Resets the current branch to a specific commit, discarding changes or moving the branch pointer.
git rm [file_name]
Removes files from both the working directory and the version control system.
git checkout -b [branch_name]
Creates a new branch and checks it out in a single step.

Git Conflict Resolution

Git conflict resolution is the process of addressing conflicts that occur when multiple contributors make changes to the same part of a file in a Git repository. Conflicts arise because Git is unable to automatically determine which changes should take precedence. Conflict resolution is a crucial step in collaborative development to ensure that code remains consistent and functional. Following is a step-by-step guide for GIT conflict resolution:

- Identifying Conflicts:

  Git identifies conflicts when you attempt to merge or pull changes from a branch that has conflicting changes with your current branch. Conflicts are typically marked within the affected file with conflict markers like <<<<<<<, =======, and >>>>>>>.
- Open the Conflicted File:

  To begin conflict resolution, open the conflicted file in a text editor or integrated development environment (IDE).
- Understanding Conflict Markers:

  The conflict markers <<<<<<<, =======, and >>>>>>> are used to delimit the conflicting sections:
  - The <<<<<<< marker indicates the beginning of the changes from the incoming branch.
  - The ======= marker separates the incoming changes from the current changes.
  - The >>>>>>> marker indicates the end of the incoming changes.
- Manually Resolve the Conflict:

  Review the conflicting sections of the file and decide which changes to keep and which to discard. Remove the conflict markers and integrate the desired changes. You can make manual edits or choose one side's changes over the other.
- Save the File:

  After resolving the conflicts, save the file with your changes.
- Mark the Conflict as Resolved:

  Use the git add command to mark the conflicted file as resolved and stage it for the next commit. For example: git add [conflicted_file].
- Commit the Changes:

  Create a new commit to confirm that you've resolved the conflict. Ensure your commit message clearly states that it resolves a conflict. For example: git commit -m "Resolve conflict in [file_name]".
- Continue the Merge or Rebase:

  If you were in the process of merging or rebasing when the conflict occurred, continue the operation using git merge --continue or git rebase --continue. This finishes the operation with the resolved conflict.
- Push or Share Your Changes:

  If you were working on a collaborative project, push your changes to the remote repository so that others can access your resolved code.
- Verify and Test:

  After conflict resolution, carefully test your code to ensure that it remains functional and hasn't introduced new issues.

Note: For more GIT commands, please see:

https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

WEEK 3

Distributed Git

Distributed Git refers to the use of Git, a version control system, in a distributed manner where each participant in a project has a complete copy of the entire project's version history. Git is designed to be distributed from the ground up, and this offers several advantages:

- Local Repositories:
  In a distributed Git setup, each developer has a local copy of the entire project's history. This means they can work independently without needing a constant connection to a central server. This is especially helpful when working offline or in low-connectivity situations.
- Branching and Merging:
  Git makes branching and merging easier and more efficient. Developers can create branches for new features, bug fixes, or experiments, and merge them back into the main project when they are ready. Each developer can maintain their own set of branches without interfering with others.
- Collaboration:
  Developers can collaborate by sharing changes with each other using repositories. Git repositories can be hosted on various platforms, such as GitHub, GitLab, Bitbucket, or on a local server. This decentralization of repositories makes it easy to collaborate with contributors from around the world.
- Forking:
  Forking is a concept where someone can create their own copy of a Git repository, make changes independently, and then propose these changes to be merged back into the original repository. This is a common practice in open-source development and is facilitated by Git's distributed nature.
- Data Integrity:
  Each local repository is a complete copy of the project's history, making it highly resilient to data loss. If one repository becomes corrupted or is lost, another developer can easily restore it from their copy.
- Scalability:
  Distributed Git systems can scale to accommodate large teams and complex projects, as the burden of maintaining the version history and tracking changes is distributed among all participants.

Remote Branches

Remote branches are a crucial part of the distributed nature of Git, as they allow multiple developers to collaborate on a project by sharing their changes and coordinating their work. Here are some key points to understand about remote branches:

- Remote Repositories:
  In a Git workflow, there is typically a central or shared repository where the project's code is stored. Remote branches exist in this central repository, which is often hosted on a platform like GitHub, GitLab, Bitbucket, or a company's internal server.
- Tracking Branches:
  When a developer clones a Git repository, they create a local copy of the remote branches. These local copies, which are linked to the remote branches, are known as "tracking branches." Tracking branches serve as references to the state of the remote branch.
- Synchronization:
  Developers can fetch and push changes to and from remote repositories. Fetching updates, the local tracking branches to reflect the state of the remote branches,

and pushing allows developers to share their local branch changes with the remote repository.

- Collaboration:

  Multiple developers can work on the same project by each having their own local branches. They can push their changes to remote branches to share their work with others. Remote branches provide a common ground where developers can see each other's changes and coordinate their work.

- Naming Convention:

  Remote branches are often prefixed with the name of the remote repository they belong to, followed by a slash.

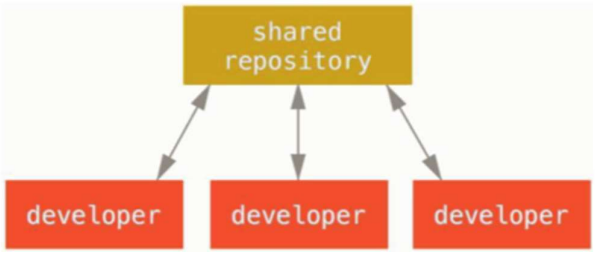
Here's a basic workflow involving remote branches:

- You clone a remote repository, creating a local copy of all its branches, including remote branches.
- You typically work on your own local branches, making changes and commits.
- When you want to share your changes, you push your local branch to a corresponding remote branch.
- You can also fetch changes from the remote repository to update your local tracking branches and see what others have done.
- Collaboration and coordination among team members happen by pushing and fetching changes to and from the remote branches.

Centralized Workflows

A centralized workflow is a software development process that revolves around a central repository for version control, where all project files and code are stored. A particular model which must be noted is the Single Collaboration Model.

Single Collaboration Model



A single collaboration model in a centralized workflow refers to a straightforward and linear approach to collaborating on a software development project, where there is typically one central or "master" branch in the version control system, and developers work together on this branch. Here's how a single collaboration model works in a centralized workflow:

- Central Repository:

  In a centralized workflow, there is a single, central repository where the project's source code is stored. This repository is often referred to as the "master" repository.

- Single Main Branch:

  In this model, there is typically one primary branch, such as "main" or "master," which contains the latest, stable version of the project. This is the branch that all developers collaborate on.

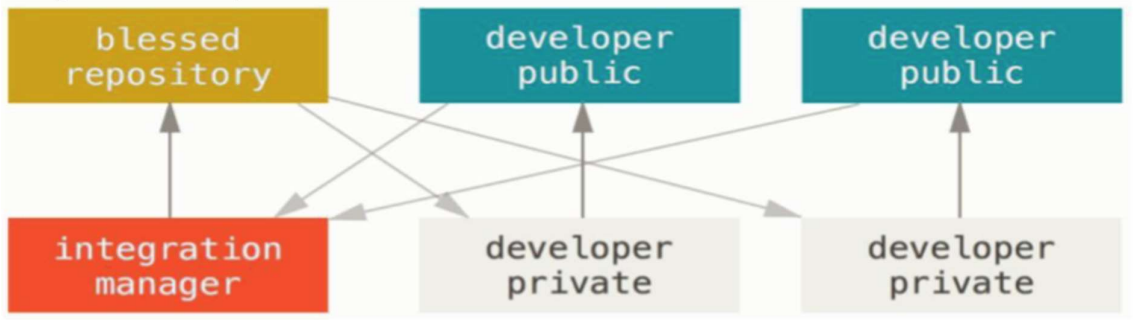
- Checkout and Commit:
  Developers clone the central repository to create their local working copies. They typically work on the main branch directly by checking it out. When they make changes, they commit them to the same branch.
- Synchronization:
  To share their changes and collaborate, developers must frequently update their local working copies from the central repository. They do this to ensure they have the latest changes made by other team members. This process is typically known as "updating" or "updating to the latest revision."
- Conflict Resolution:
  If multiple developers make changes to the same part of the code simultaneously, conflicts can occur when they try to update or commit their changes. Resolving these conflicts can be a challenging aspect of the centralized model.
- Releases and Tags:
  When a stable version of the software is ready, a release can be created from the main branch. In a centralized workflow, this often involves creating a tag to mark a specific revision as a release point.

Advantages of a single collaboration model in a centralized workflow include simplicity and a clear, linear history of changes in the central repository. However, it has limitations when it comes to parallel development, scalability, and robust conflict resolution. If multiple developers need to work on different features or bug fixes simultaneously, they may encounter conflicts and synchronization issues more frequently.

Distributed Workflows

It refers to the processes and strategies used by development teams to work on projects collaboratively and efficiently in a distributed or decentralized manner. These workflows are designed to facilitate cooperation among team members who may be geographically dispersed, while still ensuring effective version control and project management. We will discuss a few models ahead.

Integration Manager Model



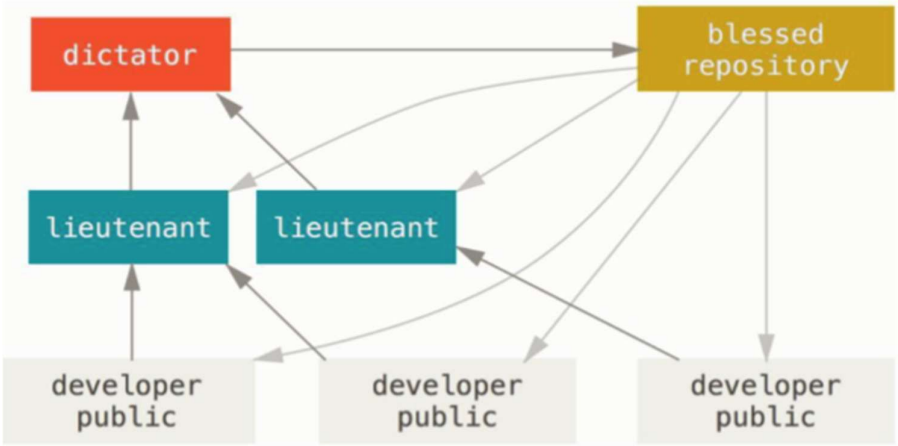
The integration manager model is one of the collaborative strategies used in distributed version control workflows, primarily associated with Git. In this model, an integration manager or a centralized figure plays a key role in managing the integration of changes from multiple contributors. The integration manager is responsible for reviewing, testing, and integrating code changes into the main or stable branch of the project. Here's how the integration manager model works:

- Contributors:
  The development team consists of several contributors, each with their own local repositories. Contributors work on their individual branches, implementing features, bug fixes, or other changes.
- Integration Manager:
  The integration manager is a designated team member, often with a higher level of trust or experience. This person's role is to oversee the integration of changes into the main project branch. The integration manager may be responsible for the stability of the main branch and ensuring that only thoroughly tested and approved code is merged into it.
- Branches:
  Contributors create feature branches in their local repositories to work on specific tasks. These branches contain their work in progress.
- Pull Requests or Merge Requests:
  When contributors complete their work on a feature or bug fix, they create pull requests (in the case of Git) or merge requests (in the case of GitLab) to propose their changes for integration. The integration manager is responsible for reviewing these requests.
- Review and Testing:
  The integration manager reviews the code changes submitted in the pull requests. They may evaluate the code for quality, adherence to coding standards, and potential issues. Additionally, they may run tests to ensure that the changes do not introduce regressions or break existing functionality.
- Integration:
  If the integration manager approves the changes, they are responsible for merging the pull request into the main branch (e.g., "main" or "master"). This process may involve resolving any conflicts that arise when multiple pull requests modify the same parts of the codebase.
- Testing and Continuous Integration:
  After the integration manager merges the changes, the project may undergo additional testing, including automated tests and continuous integration pipelines, to ensure that the integration does not introduce new issues.
- Stability and Release Management:
  The integration manager's role is crucial for maintaining the stability of the main branch, as this branch represents the most stable version of the project. They are often involved in release management, tagging releases, and ensuring that only well-tested code is included in each release.

The integration manager model is particularly useful for projects with strict quality control requirements and the need for a clear gatekeeper to maintain the stability of the main branch. It can help ensure that only high-quality and fully tested code is integrated into the project, minimizing the risk of introducing regressions or bugs. This model is commonly used in open-source projects and commercial software development, where multiple contributors work on the same codebase. The main workflow is as follows:

- The project maintainer pushes to their public repository.
- A contributor clones that repository and makes changes.
- The contributor pushes to their own public copy.
- The contributor sends the maintainer an email asking them to pull changes.
- The maintainer adds the contributor's repository as a remote and merges locally.
- The maintainer pushes merged changes to the main (blessed) repository.

Dictator and Lieutenants Model



The "Dictator and Lieutenants" model is a collaborative strategy used in distributed version control workflows, particularly in the context of Git. In this model, development teams are organized hierarchically, with a central figure (the "Dictator") and one or more subordinates (the "Lieutenants") who manage the integration of code changes from contributors. This model is often used in open-source projects and large-scale development efforts. Here's how the Dictator and Lieutenants model works:

- Contributors:
  The development team consists of several contributors, each with their own local repositories. These contributors work on their individual branches, implementing features, bug fixes, or other changes.
- Dictator:
  The "Dictator" is a designated, authoritative figure in the project. They have the ultimate say in which code changes are integrated into the main branch (e.g., "main" or "master"). The Dictator is responsible for maintaining the overall vision and stability of the project.
- Lieutenants:
  The "Lieutenants" are trusted team members chosen by the Dictator. They are responsible for overseeing specific areas or components of the project. Each Lieutenant manages the integration of code changes from contributors in their area of responsibility.
- Branches:
  Contributors create feature branches in their local repositories to work on specific tasks. These branches contain their work in progress.
- Pull Requests or Merge Requests:
  When contributors complete their work on a feature or bug fix, they create pull requests (in the case of Git) or merge requests (in the case of GitLab) to propose their changes for integration. Lieutenants are responsible for reviewing pull requests related to their area of expertise.
- Review and Testing:
  Lieutenants review the code changes submitted in the pull requests within their area of responsibility. They evaluate the code for quality, adherence to coding standards, and potential issues. Additionally, they may run tests specific to their area to ensure the changes do not introduce regressions or break existing functionality.
- Integration:
  If a Lieutenant approves the changes within their area, they are responsible for merging the pull request into the main branch within their area of responsibility. This

process may involve resolving any conflicts that arise when multiple pull requests modify the same parts of the codebase.

- Dictator's Role:

  The Dictator reviews the work of the Lieutenants and has the final say on which code changes are integrated into the main branch. The Dictator is responsible for maintaining the overall stability and vision of the project and ensuring that code changes do not conflict with the project's overarching goals.
- Testing and Continuous Integration:

  After integration by Lieutenants, the project may undergo additional testing, including automated tests and continuous integration pipelines, to ensure that the integration does not introduce new issues.
- Stability and Release Management:

  The Dictator and Lieutenants play crucial roles in maintaining the stability of their respective areas. They are often involved in release management, tagging releases, and ensuring that only well-tested code is included in each release.

The main workflow of this model is as follows:

- Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the reference repository to which the dictator pushes.
- Lieutenants merge the developers' topic branches into their master branch.
- The dictator merges the lieutenants' master branches into the dictator’s master branch.
- Finally, the dictator pushes that master branch to the reference repository so the other developers can rebase on it.

Contributing to a project

There are several factors which affect how one can contribute effectively to a project. A few such factors are:

- Active Contributor Count

  How many users are actively contributing to code the project and how often do they contribute? It is ideal to figure out the right number of people contributing with how many commits they have per day as there is always a risk of potential conflict and merge issues.
- Project Workflow

  The project workflow model which has been used is a very important factor. Is it a centralized with equal write access to the main code line granted to every user or does the project have a dictator? It is important to have a good grasp on the model and the relationship created through this model.
- Commit Access

  It is important to know how the type of commit access would affect the contribution workflow. Do all users have the write access or just a chosen few? What is the policy for accepting any work? And how much work a developer can contribute at a time and how often can they do it? It is important to know there are certain guidelines to be followed when committing to the main branch. These guidelines are as follows:
  - Make sure to not introduce whitespace errors.
  - Commit logically and do not work on many different issues in one code and submitting them as one commit.
  - Use concise description of the change followed by a detailed explanation of the change in commit messages.

Remote Repository

A remote repository, in the context of version control systems like Git, is a repository that is not located on your local machine but exists on a remote server or a different location. Remote repositories serve as a central point for collaborating with others and for sharing and synchronizing code changes in a distributed version control system. Here's what you need to know about remote repositories:

- Collaboration:

  Remote repositories facilitate collaboration among multiple developers working on a project. Instead of each developer working in isolation, they can push their changes to a remote repository and pull changes from it, allowing for coordinated development.
- Backup and Redundancy:

  Remote repositories provide a backup of the project's code. If your local machine experiences data loss or hardware failure, you can retrieve the code from the remote repository. This redundancy is a key benefit of distributed version control systems like Git.
- Access Control:

  Access to a remote repository is typically controlled through authentication and authorization mechanisms. Administrators can set permissions to determine who can read from and write to the repository, which is crucial for managing contributions and maintaining code integrity.
- Branches:

  Remote repositories can store multiple branches of a project, each representing a different line of development. Developers can push their local branches to the remote repository and collaborate on specific features or fixes.
- Pull and Push:

  Developers use commands like "git push" to send their local changes to the remote repository and "git pull" to retrieve changes from the remote repository to their local copy. These actions help keep local and remote repositories in sync.
- Platform Hosting:

  Many remote repositories are hosted on online platforms such as GitHub, GitLab, Bitbucket, and others. These platforms provide web-based interfaces for managing and collaborating on repositories and offer additional features like issue tracking, pull requests, and code review tools.
- Forks:

  In some distributed version control systems, like Git, developers can create their own fork or copy of a remote repository. They can work on changes independently in their fork and then submit pull requests to have their changes incorporated into the original repository.
- Centralization of Code:

  Remote repositories centralize the project's codebase, making it easier for multiple developers to access and work with the same codebase. This centralization is especially important in collaborative software development.

GitHub

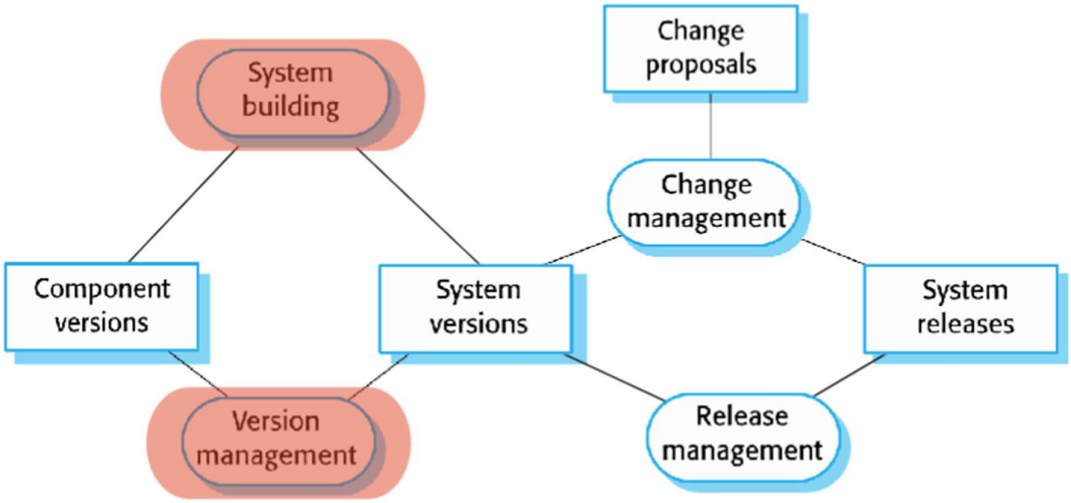
GitHub is a web-based platform and a popular service for version control and collaborative software development. It offers a wide range of tools and features that facilitate code hosting, collaboration, and project management for both open-source and private software development. GitHub allows you to create one user personal account within which you can create public or private repositories.

GitHub Organizations are a feature of GitHub that enable the management of repositories, access control, and collaboration for groups of individuals or teams, whether they are working on open-source projects, private projects, or within an enterprise environment. GitHub Organizations provide a structured way to organize and manage repositories, members, and permissions. The organization members can have an owner, and members. The organisations owner has complete administrative access to the organization. The members are the default role for everyone else. The owner manages the member's access to the organisation's repositories and projects with fine grained permission controls.

WEEK 4

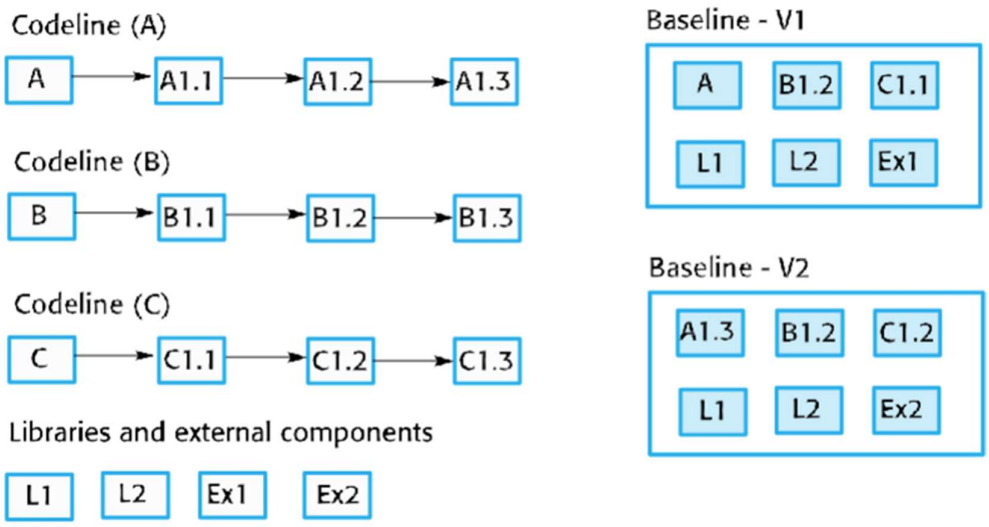
Configuration Management

Configuration management (CM) is a discipline and a set of practices and processes used in software development, IT operations, and various engineering fields to manage and control the configuration of software, systems, hardware, and other components. The primary goal of configuration management is to ensure that systems and components are consistent, well-documented, and traceable throughout their lifecycle. This discipline is crucial for maintaining the integrity, reliability, and quality of complex systems. Following are the configuration management activities:



A few more details about the activities are given below:

- System building: assembling program components, data, and libraries, then compiling these to create an executable system.
- Version management: 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. This activity can be thought of as the process of managing codelines and baselines. Codelines are a sequence of versions of source code with later versions in the sequence derived from earlier versions. Baseline on the other hand is a definition of a specific system and may be specified using a configuration language, to define what components are included in a version of a particular system.
- Change management: keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- Release management: preparing software for external release and keeping track of the system versions that have been released for customers.



Agile development and Configuration Management (CM) can be integrated to enhance software development processes by ensuring that configurations are managed effectively and efficiently throughout the Agile development lifecycle. Agile development methodologies, such as Scrum and Kanban, prioritize collaboration, flexibility, and rapid iteration, which can be complemented by appropriate CM practices.

Semantic Versioning

Semantic Versioning, often abbreviated as SemVer, is a versioning scheme for software that aims to communicate meaning about the underlying changes in a release. It provides a structured and standardized way of versioning software to make it easier for developers to understand what has changed and to manage dependencies. Semantic Versioning is based on a three-part version number, in the format of "MAJOR.MINOR.PATCH," and it follows these guidelines:

- MAJOR Version (X.0.0):
  Increment the major version when you make incompatible changes to the public API or introduce significant changes that require developers to make modifications to their code. This signifies that there are breaking changes.
- MINOR Version (X.Y.0):
  Increment the minor version when you add new features or functionalities in a backward-compatible manner. This indicates that there are new, non-breaking features.
- PATCH Version (X.Y.Z):
  Increment the patch version when you make backward-compatible bug fixes or minor improvements. This is for small, non-breaking changes.

System Building

System building, in the context of software and technology, refers to the process of designing, developing, and assembling a complete software system or a technology solution to meet specific objectives and requirements. This encompasses not only the creation of software applications but also the integration of hardware, software components, data, and infrastructure to form a cohesive and functional system. System building tools and version management tools must communicate as the build process involves checking out component versions from the repo managed by the version management system.

Software Build Automation Tools

Software build automation tools are software utilities that streamline and automate the process of compiling, building, and packaging software applications or components from their source code. These tools are essential in software development and release management to improve efficiency, consistency, and reliability in building software. Build automation tools are especially valuable in large-scale projects with complex codebases and multiple dependencies. A few examples of build tools are given below:

- Apache Ant

  Apache Ant, commonly referred to as Ant, is an open-source, Java-based build automation tool and software project management tool. It is a versatile and highly extensible tool used primarily for automating the build process of Java applications, although it can be adapted for other tasks and projects as well. Apache Ant is part of the Apache Software Foundation and is widely used in Java development projects. Drawbacks of Apache ANT are basically its immense flexibility, complexity, and absence of a standard structure.

- Apache Maven

  Apache Maven is an open-source build automation and project management tool used primarily for Java projects, though it can be adapted for other programming languages and project types. Maven is part of the Apache Software Foundation and provides a structured and convention-based approach to project configuration, build automation, dependency management, and project lifecycle management. Drawbacks of Apache Maven are basically the complexity and rigid conventions as the developers are required to understand follow and follow the conventions.

- Gradle

  Gradle is an open-source build automation tool and project management system that is designed to automate the building, testing, and deployment of software projects. Gradle is known for its flexibility, extensibility, and powerful build-by-convention approach. It is widely used in the software development industry for a variety of project types, including Java, Android, and web applications. Gradle is a build automation tool which builds upon the concept of ANT and Maven. Gradle build files are groovy scripts which is a dynamic language of the Java Virtual Machine. It can be added as a plug-in, allows developers to write general programming tasks in the build files, and relieves developers from the lacking control flow in ANT. Gradle also presents a DSL tailored to the task of building code which is not a programming language. Gradle DSL is extensible through plug ins as well which makes it extensible. Gradle plug-ins allow adding new task definitions, changing behaviour of existing tasks, adding new objects, creating keywords to describe tasks that depart from the standard Gradle categories, and much more.

Gradle

In Gradle, tasks are the fundamental units of work that can be executed during the build process. Tasks define the specific actions to be performed, such as compiling source code, running tests, generating documentation, and many other activities related to building and managing a software project. Gradle provides a powerful and flexible task model that allows developers to define, configure, and execute tasks as part of the build process. A project on the other hand is a composition of several such tasks. Each task has a name, which can be used to refer to the task within its owning project, and a fully qualified path, which is unique across all tasks in all projects.

Task actions are the code blocks or scripts that define the specific work to be done by the task. Task actions can include executing shell commands, invoking Java classes, or running any other custom code needed for the task. Following is a list of all commands associated Task actions:

Task Action	**Action Command**
Executing a task	Action.execute(T)
Add actions to a task	Task.doFirst() Task.doLast()
Abort execution of the action	StopActionException
Abort execution of task and continue to the next task	StopExecutionException

The Gradle build lifecycle represents the sequence of tasks and phases that are executed when you run a Gradle build. Gradle follows a convention-over-configuration approach, and it defines a set of standard lifecycle tasks that are commonly used in most projects. These standard lifecycle tasks are executed in a specific order to represent the build process's progression. Here are the key phases and standard tasks in the Gradle build lifecycle:

- Initialization Phase:
  - Init: This is the first phase of the Gradle build lifecycle. It involves project initialization and setting up the build environment. Gradle determines the project structure, evaluates settings and build scripts, and prepares for task execution.
- Configuration Phase:
  - Settings Configuration: In this phase, the settings.gradle script is evaluated, and project settings are configured.
  - Projects Configuration: Configuration of the root and subprojects is performed, including defining project dependencies and applying plugins.
- Execution Phase:
  - Init Task: Gradle evaluates the init task to set up the build environment.
  - Default Tasks: Gradle executes default tasks that correspond to the build lifecycle phases. These tasks are commonly used and include:
    - Initialization Tasks: Tasks that prepare the build environment.
    - Build Tasks: Tasks that compile, test, package, and build the project.
    - Check Tasks: Tasks that perform code quality checks, tests, and other verifications.
    - Assemble Tasks: Tasks that create artifacts, such as JAR or WAR files.
    - Publish Tasks: Tasks that publish artifacts to a repository, if applicable.
  - Custom Tasks: Developers can also define custom tasks that are executed during this phase based on the project's requirements.
- Finalization Phase:
  - Finalize Task: Gradle executes the finalize task to perform cleanup and finalization tasks.
  - Build Finalization: The build is finalized, and any necessary cleanup or post-build activities are performed.
- Build Completion:
  - Build Finished: Gradle reports the build's completion status, including success or failure.
- Plugin Application: Plugins may apply additional tasks and configurations to the build lifecycle. Plugin-specific tasks can be integrated into the build process.

In Gradle, task configuration involves specifying how a task should behave, what actions it should perform, and what dependencies it should have. Task configuration is a fundamental aspect of a Gradle build script, and it allows you to customize and control various aspects of the build process. You configure a task by defining a configuration block within the task declaration. This block specifies properties, dependencies, actions, and other settings related to the task. Every task is represented internally as an object. Following are methods of default Task:

Method	Description
dependsOn(task)	Adds a task as a dependency of the calling task. A depended-on task will always run before the task that depends on it
doFirst(closure)	Adds a block of executable code to the beginning of a task's action. During the execution phase, the action block of every relevant task is executed.
doLast(closure)	Appends behavior to the end of an action
onlyIf(closure)	Expresses a predicate which determines whether a task should be executed. The value of the predicate is the value returned by the closure.

Following are the Default task's properties:

Method	Description
didWork	A Boolean property indicating whether the task completed successfully
enabled	A Boolean property indicating whether the task will execute.
path	A string property containing the fully qualified path of a task (levels; DEBUG, INFO, LIFECYCLE, WARN, QUIET, ERROR)
logger	A reference to the internal Gradle logger object
logging	The logging property gives us access to the log level
temporaryDir	Returns a File object pointing to a temporary directory belonging to this build file. It is generally available to a task needing a temporary place in to store intermediate results of any work, or to stage files for processing inside the task
description	a small piece of human-readable metadata to document the purpose of a task

Dynamic properties in Gradle are properties that are added to objects or elements at runtime rather than being statically defined in the build script. These dynamic properties can be particularly useful when you need to extend or modify the behaviour of objects, tasks, or other elements in your Gradle build script.

Gradle supports a wide variety of built-in task types, each designed to perform specific actions or operations as part of the build process. These task types are commonly used in Gradle build scripts to automate different aspects of the software development process. Here are some of the most common types of Gradle tasks:

- Copy

  The Copy task type allows you to copy files and directories from one location to another. It is used for various file copying operations.
- Jar

  The Jar task type is used to create JAR (Java Archive) files, which package compiled code and resources.

- JavaExec

  The Exec task type lets you execute external commands or scripts. It is often used for running command-line tools or performing custom build actions.

In Gradle, plugins are a way to extend the build and add specific functionality or tasks to a project. Gradle provides a powerful plugin system that allows you to apply and configure plugins to automate various tasks and integrate with third-party tools or libraries. Gradle plugins are typically packaged as JAR files and can be easily added to your build script. ava plug-in adds some tasks to your project which will compile, and unit test your Java source code, and bundle into a JAR.

WEEK 5

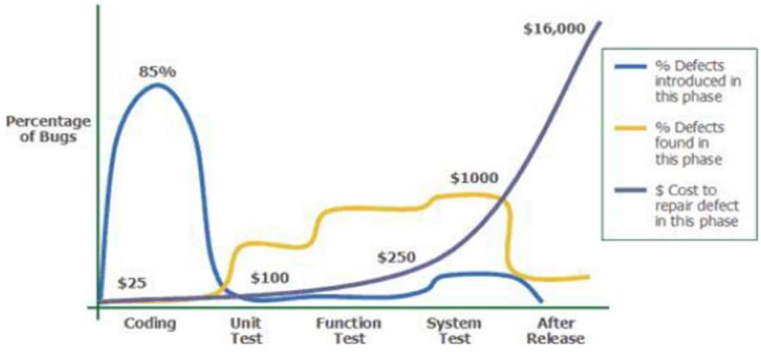
Software Quality Assurance

Software quality refers to the degree to which a software product or system meets specified requirements, fulfills user expectations, and satisfies its intended purpose. It encompasses a set of characteristics and attributes that collectively determine the level of excellence or adequacy of a software application. Software quality is essential for ensuring that the software functions correctly, performs reliably, and is fit for its intended use.

Software Quality Assurance (SQA) is a systematic and comprehensive process that ensures the quality of software throughout its development life cycle. The primary goal of SQA is to provide confidence that the software being developed or maintained meets specified quality standards and functional requirements. SQA encompasses a wide range of activities, methods, and practices aimed at preventing defects, identifying, and fixing issues, and improving overall software quality. Software Quality Assurance plays a crucial role in delivering high-quality software products that are reliable, secure, and meet user expectations. By implementing SQA practices, organizations can minimize the risk of defects, reduce development costs, and enhance their software's reputation and reliability.

Software Testing

Software testing is a systematic and organized process of evaluating and verifying a software application to identify whether it behaves as expected and meets its specified requirements. The primary purpose of software testing is to detect defects, errors, and issues in the software to ensure its quality, reliability, and functionality. Software testing encompasses various activities and techniques to assess the software's correctness, performance, security, and usability. The cost of software testing can vary significantly depending on several factors. These factors include the complexity of the software, the type and depth of testing required, the size of the testing team, the tools and technologies used, and the location of the testing team. Given below is a screenshot of the costs of software testing based on the percentage of bugs:



There are different types of software testing. A few of them are given below:

- Unit Testing:
  Unit testing involves testing individual components or functions of the software in isolation. It ensures that each unit of code (e.g., functions, methods, or classes) performs correctly. Developers often conduct unit testing during the coding phase.
- Integration Testing:
  Integration testing evaluates the interactions and interfaces between different units or components of the software. It aims to identify issues related to the integration of various parts of the system.
- System Testing:
  System testing assesses the entire software system to ensure that it functions. It verifies that the software meets its specified requirements and behaves correctly in different scenarios.
- Acceptance Testing:
  Acceptance testing is performed to determine whether the software meets the user's acceptance criteria and business requirements. It includes User Acceptance Testing (UAT), where end-users validate that the software meets their needs.
- Functional Testing:
  Functional testing evaluates the software's functionality to ensure that it performs its intended tasks and features correctly. It includes positive and negative testing scenarios.
- Non-Functional Testing:
  Non-functional testing focuses on qualities like performance, security, usability, and reliability. Common non-functional testing types include:
  - Performance Testing: Assessing the software's speed, responsiveness, and scalability.
  - Security Testing: Identifying vulnerabilities and weaknesses in the software's security.
  - Usability Testing: Evaluating the user-friendliness and overall user experience.
  - Reliability Testing: Ensuring that the software is stable and reliable under different conditions.
- Regression Testing:
  Regression testing involves retesting previously tested functionalities to ensure that new changes or updates do not break existing features or introduce defects.
- Smoke Testing:
  Smoke testing is a preliminary test to check if the software is stable enough for further, more in-depth testing. It often focuses on major functionalities or critical paths.
- Exploratory Testing:
  Exploratory testing is an unscripted testing approach where testers actively explore the software to find defects and issues without predefined test cases.
- Alpha and Beta Testing:
  Alpha testing is performed by the software development team to uncover issues before the software is released to a limited group of external users, known as beta testers. Beta testing gathers user feedback from a wider audience.
- Ad Hoc Testing:
  Ad hoc testing is an informal testing approach where testers execute test cases without predefined plans or scripts. It is often used for quick, unplanned testing.
- Compatibility Testing:

Compatibility testing ensures that the software functions correctly across different devices, browsers, operating systems, and configurations.

- Localization and Internationalization Testing:
  Localization testing checks if the software is adapted to different languages, regions, and cultural conventions. Internationalization testing assesses the software's readiness for localization.
- Accessibility Testing:
  Accessibility testing evaluates the software's accessibility features to ensure it can be used by people with disabilities. It checks compliance with accessibility standards like WCAG.
- Concurrency and Multi-Threading Testing:
  This type of testing verifies the software's behaviour in multi-threaded or concurrent environments, ensuring it handles concurrent processes correctly.
- Installation and Uninstallation Testing:
  Installation testing confirms that the software installs correctly on target systems, while uninstallation testing checks for a clean and complete removal of the software.
- Data Migration Testing:
  Data migration testing verifies the accuracy and completeness of data transferred from one system to another during data migration processes.
- Boundary Testing:
  Boundary testing evaluates the software's behaviour at boundary values of input domains to check for issues related to boundary conditions.
- White Box Testing:
  White box testing examines the internal structures and code of the software. Testers have knowledge of the internal logic, which helps identify issues related to code paths and data flows.
- Black Box Testing:
  Black box testing assesses the software's functionality without knowledge of its internal code or structure. It focuses on testing from a user's perspective.

The objectives of software testing are to ensure that a software application or system meets specified quality standards, functions correctly, and delivers value to its users and stakeholders. Testing serves various goals and purposes throughout the software development life cycle, and the specific objectives can vary based on the testing phase and the type of testing being conducted. Objectives should be stated precisely and quantitatively to measure and control the test process. Test cases are usually exhaustive and expensive, so a proper risk driven approach is required to increase confidence. Furthermore, only specific test cases need to be selected which are sufficient for a specific purpose and covers the required area.

Software testing can be conducted by various individuals and teams throughout the software development life cycle. The roles and responsibilities of testers can vary depending on the organization's structure, the specific testing phase, and the type of testing being performed. A few roles involved in testing are:

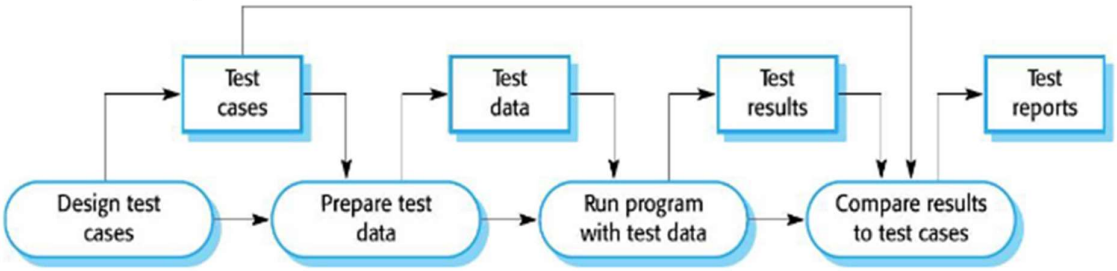
- Software Testers:
  Testers are responsible for executing test cases and test scenarios to evaluate the software's functionality, performance, and other attributes. They report defects and work closely with developers to address issues.
- Automation Testers:

Automation testers specialize in automating test cases using testing tools and scripting languages. They create automated tests to improve testing efficiency and coverage.

- Performance Testers:
  Performance testers focus on assessing the software's performance, scalability, and response times. They conduct load testing, stress testing, and other performance-related assessments.
- Security Testers:
  Security testers concentrate on identifying security vulnerabilities and weaknesses in the software. They perform penetration testing, security assessments, and vulnerability scans.
- User Acceptance Testers:
  User acceptance testers, often end-users or stakeholders, validate that the software meets their business requirements and expectations. They ensure the software fulfills its intended purpose.
- Development Team:
  Developers often participate in testing activities, particularly during unit testing and integration testing. They are responsible for fixing defects and issues identified during testing.

To develop an effective test, one must have detailed understanding of the system, application and solution domain knowledge, and knowledge of the different testing techniques. Testing is done best by independent testers who are not related to the project from the beginning.

Software Testing Process



- Design Test Cases:
  In this phase, testers create detailed test cases, specifying the test conditions, input data, and expected outcomes for the software's functionalities.
- Test Cases:
  The created test cases are used for reference and guidance during the testing process. These test cases specify what to test, the expected outcomes, and any specific instructions.
- Prepare Test Data:
  Test data is gathered or generated to serve as input for the test cases. It should cover a range of scenarios, including normal and boundary cases.
- Test Data:
  The prepared test data is used as input for the test cases, providing the necessary input values for testing.
- Run Program with Test Data:
  Testers execute the software with the prepared test data, following the test cases' instructions and inputs.

- Test Results:

  The actual results of the software's execution are recorded as a result of running the program with the test data.
- Compare Results to Test Cases:

  After the program execution, the actual results are recorded. Testers compare these results to the expected outcomes defined in the test cases.
- Test Results:

  Test results are documented, and any discrepancies between actual and expected results are reported as defects or issues. The purpose is to identify areas where the software does not meet the specified requirements or criteria.

Types of defects in software

Defects in software, also known as software bugs or issues, are anomalies, errors, or unintended behaviours that deviate from the expected or specified functionality. These defects can manifest in various ways and impact different aspects of software quality. Here are some common types of defects in software:

- Functional Defects:
  - Incorrect Output:

    The software produces results that do not match the expected or specified outcomes. This can be caused by coding errors, logical flaws, or incorrect calculations.
  - Incomplete Functionality:

    Parts of the software's functionality are missing or not fully implemented as per the requirements.
  - Functional Misbehaviour:

    The software behaves differently from what was intended or specified, often leading to unintended consequences.
- Non-Functional Defects:
  - Performance Issues:

    The software may run slowly, consume excessive resources, or fail to meet performance requirements, leading to delays and inefficiencies.
  - Security Vulnerabilities:

    Security defects can include vulnerabilities, weaknesses, or loopholes that expose the software to potential threats and unauthorized access.
  - Usability Problems:

    Issues related to user interface design, accessibility, or user experience can affect the software's ease of use and user satisfaction.
  - Compatibility Problems:

    Software may not function correctly or as expected on certain devices, browsers, or operating systems, leading to compatibility defects.
- Documentation Defects:
  - Incomplete or Inaccurate Documentation:

    Inadequate or incorrect documentation, including user manuals, help guides, or system documentation, can lead to user confusion and errors.
  - Misaligned Documentation:

    When the documentation does not align with the actual software behaviour, it can lead to misunderstandings and mistakes during usage.
- Integration Defects:
  - Integration Failures:

Defects may occur when different software components or systems fail to integrate seamlessly, leading to issues in data exchange or communication.

- Data Inconsistencies:

  Inconsistent or incorrect data exchanges between integrated systems can result in data defects.

- Data Defects:
  - Data Corruption:

    Data defects can lead to the corruption or loss of critical data, impacting the software's functionality and reliability.
  - Data Quality Issues:

    Issues with data quality, such as inaccuracies, duplicates, or missing data, can affect the software's performance and decision-making.
- Concurrency and Timing Defects:
  - Race Conditions:

    In multi-threaded or concurrent software, race conditions can lead to defects where the timing of events and data access results in unintended behaviour.
  - Deadlocks: Software can enter a state where multiple processes or threads are waiting for resources, causing the software to become unresponsive.
- Boundary Value Defects:
  - Boundary Errors: Defects related to the handling of boundary values, such as failing to account for the minimum or maximum values in input ranges.
- Environmental Defects:
  - Issues in Different Environments: Software may work correctly in one environment but fail in another, due to differences in configurations, setups, or dependencies.
- Regression Defects:
  - Regression Bugs: Changes or updates to the software can introduce new defects or cause previously fixed issues to reappear, often due to unintended consequences.
- Performance Degradation:

  Over time, software performance may degrade, causing it to become slower or less responsive due to factors like memory leaks or resource consumption.

Software Testing Levels

Testing level	Description
Unit / Functional Testing	The process of verifying functionality of software components (functional units, subprograms) independently from the whole system
Integration Testing	The process of verifying interactions/communications among software components. Incremental integration testing vs. “Big Bang” testing
System Testing	The process of verifying the functionality and behaviour of the entire software system including security, performance, reliability, and external interfaces to other applications
Acceptance Testing	The process of verifying desired acceptance criteria are met in the system (functional and non-functional) from the user point of view

Different types of testing

There are many different types of software testing, each with its own specific goals, methods, and areas of focus. The choice of testing type depends on the nature of the software, its requirements, and the objectives of the testing process. Here are some of the most common types of software testing:

- Unit Testing:
  - Aim: To test individual units or components of the software, such as functions, methods, or classes.
  - Scope: Focused on verifying the correctness of code at the smallest level.
  - Method: Developers write unit tests to assess the functionality of code segments.
  - Tools: Unit testing frameworks like JUnit, pytest, and NUnit are commonly used.
- Integration Testing:
  - Aim: To evaluate the interactions and interfaces between different units or components when integrated.
  - Scope: Tests how integrated parts work together to identify issues related to data flow, communication, and dependencies.
  - Method: Combines individual units and tests their interactions.
  - Tools: Can involve using testing frameworks, tools, or custom scripts.
- System Testing:
  - Aim: To assess the entire software system as a whole to ensure it meets its specified requirements.
  - Scope: Comprehensive testing to validate the system's functionalities, performance, and behaviours.
  - Method: Evaluates the software from an end-user perspective.
  - Tools: May involve manual testing or automated testing tools.
- Acceptance Testing:
  - Aim: To confirm that the software meets user and business requirements.
  - Scope: Ensures that the software is acceptable to end-users or stakeholders.
  - Method: End-users or stakeholders perform acceptance testing to validate that the software fulfills their needs.
  - Tools: User acceptance testing (UAT) may be done manually, often without specific testing tools.
- Functional Testing:
  - Aim: To evaluate whether the software's functions perform as expected.
  - Scope: Emphasizes verifying that the software's features work correctly.
  - Method: Tests the software against its functional requirements, using test cases to validate functionality.
  - Tools: May involve manual testing or automation tools like Selenium or TestNG.
- Non-Functional Testing:
  - Aim: To assess non-functional attributes of the software, such as performance, security, usability, and reliability.
  - Scope: Focuses on qualities other than core functionality.
  - Types: Includes performance testing, security testing, usability testing, and reliability testing, among others.
  - Tools: Utilizes specialized testing tools for each non-functional aspect.

- Regression Testing:
    - Aim: To confirm that recent changes or updates have not introduced new defects or broken existing functionality.
    - Scope: Repeating test cases for areas that have been modified.
    - Method: Reruns existing test cases that are relevant to the changes.
    - Tools: Automated testing is often used to streamline regression testing.
- Usability Testing:
    - Aim: To evaluate the user-friendliness and user experience of the software.
    - Scope: Focuses on assessing the software's design, navigation, and ease of use.
    - Method: Usability experts and end-users test the software through real-world scenarios.
    - Tools: Often conducted with observation, surveys, and usability testing tools.
- Security Testing:
    - Aim: To identify vulnerabilities and weaknesses in the software that could lead to security breaches.
    - Scope: Evaluates the software's resistance to security threats and unauthorized access.
    - Method: Employs techniques like penetration testing and vulnerability scanning.
    - Tools: Specialized security testing tools and manual testing are used.
- Compatibility Testing:
    - Aim: To ensure that the software functions correctly across various platforms, devices, browsers, and configurations.
    - Scope: Focuses on cross-browser testing, cross-platform testing, and device testing.
    - Method: Tests the software on different environments to detect compatibility issues.
    - Tools: Cross-browser testing tools, virtual machines, and emulators are common.
- Black Box Testing:
    - Definition: Black box testing is a testing approach that treats the software as a "black box," where the internal workings, code, and structure are not known to the tester. Testers focus on evaluating the software's external behaviours and functionality.
    - Purpose: Black box testing assesses whether the software performs its intended functions correctly and meets its requirements. It is mainly concerned with validating that the software behaves as expected from the user's perspective.
    - Key Features:
        - Testers do not have access to the software's source code or internal logic.
        - Test cases are designed based on the software's specifications, requirements, and user stories.
        - Testing focuses on inputs and expected outputs, as well as the software's response to various scenarios and conditions.
        - It helps identify issues related to incorrect functionality, boundary conditions, usability, and performance from an end-user's viewpoint.
        - Testers do not need programming knowledge to conduct black box testing.
    - Examples: Functional testing, acceptance testing, usability testing, system testing, and integration testing are common examples of black box testing.
- White Box Testing:

- Definition: White box testing is a testing approach that examines the internal structure, code, and logic of the software. Testers have knowledge of the software's source code and use this knowledge to design test cases.
- Purpose: White box testing is aimed at evaluating the internal workings of the software, ensuring that the code is implemented correctly, and assessing how well it adheres to coding standards and design principles.
- Key Features:
  - Testers have access to the software's source code and design documents.
  - Test cases are created to test specific code paths, branches, and decision points within the software.
  - Testing often involves code reviews, static analysis, and techniques like statement coverage, branch coverage, and path coverage to assess code coverage and quality.
  - White box testing can uncover issues like logic errors, coding mistakes, and vulnerabilities within the code.
  - Testers typically need programming and technical knowledge to conduct white box testing effectively.
- Examples: Unit testing, integration testing (when combined with black box testing), code reviews, code inspections, and static code analysis are common examples of white box testing.

Techniques to choose test cases.

Choosing the right test cases is a critical aspect of effective software testing. Several techniques and methods can be employed to select test cases based on specific goals and testing objectives. Here are some commonly used techniques to choose test cases:

- Equivalence Partitioning:
  - Principle: Equivalence partitioning divides the input domain into equivalence classes or partitions. Test cases are then chosen to represent each partition.
  - Use Cases: This technique is particularly useful for functional testing, where input values can be grouped into categories with similar behaviours.
- Boundary Value Analysis (BVA):
  - Principle: BVA focuses on testing input values at or near the boundaries of equivalence classes. Test cases are designed to assess how the software handles edge or limit conditions.
  - Use Cases: It is applicable for uncovering defects related to boundary conditions and is often used in both functional and non-functional testing.
- Code Coverage Analysis:
  - Principle: Code coverage tools track which portions of the source code have been executed during testing. Test cases are designed to increase code coverage.
  - Use Cases: Code coverage analysis is essential for white box testing and ensuring that the code is adequately exercised.
  - Coverage criteria:

Coverage Criteria	Description
Method	How many of the methods are called, during the tests
Statement	How many statements are exercised, during the tests
Branch	How many of the branches have been exercised during the tests
Condition	Has each separate condition within each branch been evaluated to both true and false
Condition/decision coverage	Requires both decision and condition coverage be satisfied
Loop	Each loop executed zero times, once and more than once

Tools for Agile Development

Agile development relies on a variety of tools and software to facilitate collaboration, project management, communication, and the overall Agile development process. These tools are designed to help Agile teams plan, track progress, and deliver software efficiently. Here are some of the common tools used in Agile development:

- Project Management Tools:
  - Jira: A popular project and issue tracking tool by Atlassian. Jira supports Agile methodologies with features like Scrum and Kanban boards.
  - Trello: A user-friendly, visual project management tool that supports Agile practices with boards and cards.
  - Asana: A flexible project management tool that allows teams to organize and manage work using boards and lists.
- Communication and Collaboration Tools:
  - Slack: A team messaging and collaboration platform that supports real-time communication among team members.
  - Microsoft Teams: A collaboration platform that integrates with other Microsoft products and provides chat, video conferencing, and file sharing.
  - Zoom: A video conferencing tool commonly used for Agile team meetings and remote collaboration.
- Version Control and Code Collaboration:
  - Git: A distributed version control system widely used for managing source code and facilitating collaboration among developers.
  - GitHub: A web-based platform for hosting and collaborating on Git repositories. It offers features for code review, issue tracking, and project management.
  - Bitbucket: Another platform for hosting Git repositories that integrates with Jira and other Atlassian tools.
- Continuous Integration and Continuous Delivery (CI/CD) Tools:
  - Jenkins: An open-source automation server that facilitates continuous integration and continuous delivery processes.
  - CircleCI: A cloud-based CI/CD platform that automates software builds, testing, and deployment.
  - Travis CI: A CI/CD service that integrates with GitHub repositories to automate the testing and deployment pipeline.
- Build Automation Tools:
  - Jenkins: Jenkins is an open-source automation server that is highly popular for build automation and continuous integration (CI). It supports building, deploying, and automating tasks through a wide range of plugins.
  - GitLab CI/CD: GitLab offers built-in CI/CD capabilities, allowing you to automate the build and test processes as part of your GitLab-based development workflow.
  - Travis CI: Travis CI is a cloud-based CI/CD service that is easy to set up and integrate with version control systems like GitHub. It's often used for automating builds and tests in open-source projects.
- Automated Testing:
  - Selenium: Selenium is a popular open-source tool for automating web application testing. It supports multiple programming languages and browsers, making it a versatile choice for web testing.
  - JUnit: JUnit is a widely used testing framework for Java applications. It supports the creation of unit tests and is commonly used for test-driven development (TDD).

Junit

JUnit is a widely used open-source testing framework for Java applications. It is primarily used for unit testing, which involves testing individual components or units of code in isolation to ensure they work correctly. JUnit is a powerful tool that helps Java developers automate and simplify the testing process, making it an essential part of the software development life cycle. JUnit provides several important constructs and annotations that developers can use to create and run tests in their Java applications. These constructs help define test methods, set up test fixtures, and manage the test execution process. Here are the key constructs and annotations in JUnit:

- Test Class
  The test class is a Java class that contains test methods.
- Test Methods
  Test methods are regular Java methods within the test class, and they are annotated with @Test. Each test method represents a single test case and is responsible for asserting specific conditions.
- Test Fixtures
  Test fixtures include methods for setting up and tearing down the test environment.
- Assertions
  Assertions are used within test methods to check whether the actual results match expected outcomes.
- Test Suites
  Test suites are created by using the @RunWith(Suite.class) annotation to group multiple test classes together and execute them as a single unit.
- Parametrized Tests (Junit 4)
  JUnit 4 supports parameterized tests using the @Parameters and @RunWith(Parameterized.class) annotations. These allow you to run the same test method with different sets of input data.
- Timeout (Junit 4)
  The @Test(timeout) annotation allows you to specify a timeout in milliseconds for a test method, ensuring that the test completes within the specified time.

A few annotations of Junit are given below:

JUnit 4*	Description
import org.junit.*	Import statement for using the following annotations
@Test	Identifies a method as a test method
@Before	Executed before each test to prepare the test environment (e.g., read input data, initialize the class)
@After	Executed after each test to cleanup the test environment (e.g., delete temporary data, restore defaults) and save memory
@BeforeClass	Executed once, before the start of all tests to perform time intensive activities, e.g., to connect to a database
@AfterClass	Executed once, after all tests have been finished to perform clean-up activities, e.g., to disconnect from a database

A few methods to assert test results:

Method / statement	Description
assertTrue(Boolean condition [,message])	Checks that the Boolean condition is true.
assertFalse(Boolean condition [,message])	Checks that the Boolean condition is false.
assertEquals(expected, actual [,message])	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals(expected, actual, delta [,message])	Test that float values are equal within a given delta.
assertNull(object [,message])	Checks that the object is null.
assertNotNull(object, [,message])	Checks that the object is not null.

Junit rules:

Rule	Description
TemporaryFolder	Creates files and folders that are deleted when the test finishes
ErrorCollector	Let execution of test to continue after first problem is found
ExpectedException	Allows in-test specification of expected exception types and messages
TimeOut	Applies the same timeout to all test methods in a class
ExternalResources	Base class for rules that setup an external resource before a test (a file, socket, database connection)
RuleChain	Allows ordering of TestRules

JUnit can be integrated with Gradle, a popular build automation tool, to simplify the process of building, testing, and managing Java projects. Gradle provides a convenient way to set up and execute JUnit tests as part of your project's build process. To use Junit in the Gradle build, add a testCompile dependency to the build file.

EclEmma

EclEmma, also known as the EclEmma Java Code Coverage plugin, is a popular open-source code coverage tool for Java developers. It is based on the JaCoCo code coverage library and it integrates with the Eclipse Integrated Development Environment (IDE) to provide code coverage analysis for Java projects. Code coverage tools help developers determine which parts of their code have been executed during testing, helping to identify untested or insufficiently tested code. Here are some key features and aspects of EclEmma:

- Integration with Eclipse:
  EclEmma is a plugin for the Eclipse IDE, making it easy to install and use within your development environment.
- Code Coverage Metrics:
  EclEmma provides code coverage metrics, including line coverage, branch coverage, and method coverage, which help you understand how thoroughly your code has been tested.
- Coverage Colouring:
  EclEmma highlights source code within the Eclipse editor, using different colours to indicate which lines of code have been covered by tests and which have not. This visual feedback is useful for identifying untested code paths.
- Counters
  EclEmma supports different types of counters to be summarized in code coverage overview.

WEEK 6

Continuous Integration

Continuous Integration (CI) is a software development practice and approach that focuses on regularly and automatically integrating code changes from multiple contributors into a shared repository. The primary goal of CI is to detect and address integration issues early in the development process, leading to higher software quality, faster development cycles, and improved collaboration within development teams. The main objectives of CI are:

- Minimize the duration and effort required by each integration.
- Be able to deliver product version suitable for release at any moment.

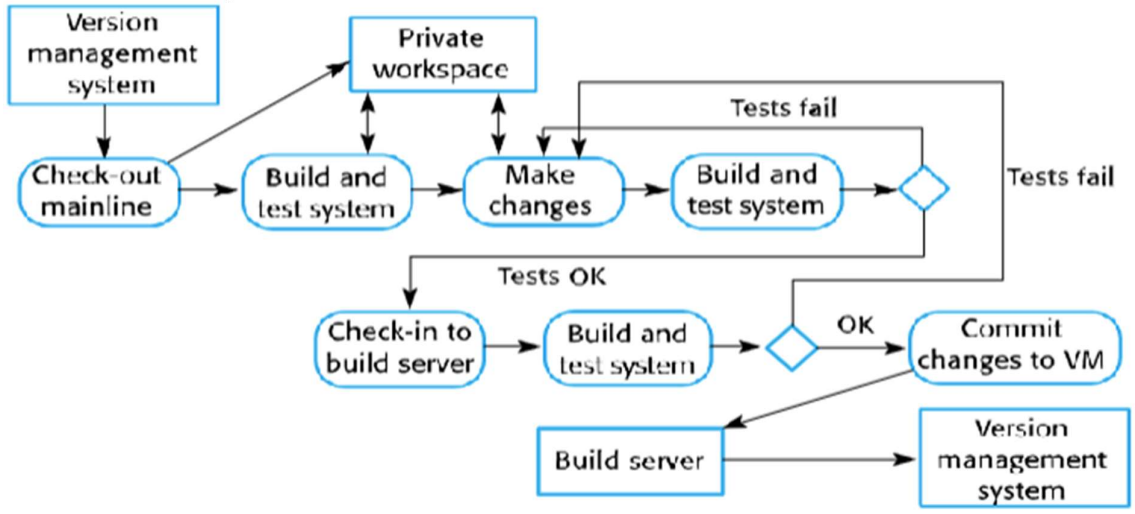
- To achieve the Integration procedure which is reproducible.
- To achieve Largely automated integration.

CI should not be confused with the tools that assist it. CI is a development practice not tools. Implementing Continuous Integration (CI) in a software development project involves several steps and best practices to ensure that code changes are continuously integrated, built, and tested. Following are a few steps to integrate CI practices:

- Set Up a Version Control System:
  Ensure that your project uses a version control system (e.g., Git) to manage and track code changes. All developers should commit their code to a shared repository.
- Choose a CI/CD Tool:
  Select a CI/CD tool that fits your project's requirements. Popular options include Jenkins, Travis CI, CircleCI, GitLab CI/CD, and others. The choice may depend on your programming language, technology stack, and integration needs.
- Configure the CI/CD Environment:
  Set up and configure the CI/CD environment in your chosen tool. This includes defining the build process, specifying the testing framework, and configuring the deployment pipeline if you plan to implement Continuous Delivery (CD).
- Define Automated Build and Testing Procedures:
  Create automated build and testing scripts that define how your code should be built and tested. This includes running unit tests, integration tests, and other types of tests that are relevant to your project.
- Create a CI/CD Pipeline:
  Define a CI/CD pipeline that outlines the stages and steps your code changes will go through. A typical pipeline includes stages for building, testing, and deploying to staging or production environments.

Continuous Integration Workflow

The Continuous Integration (CI) workflow is a development practice that automates the integration and testing of code changes, promoting collaboration, code quality, and efficiency in software development.



Below is a description of the typical CI workflow, which outlines the key stages and steps involved in the process:

- Code Development:
  Developers work on code changes, including new features, bug fixes, or improvements, individually or in teams.
- Version Control:
  Developers commit their code changes to a version control system (e.g., Git), ensuring that changes are tracked, documented, and easily accessible to other team members.
- Automated Build:
  When code changes are committed, an automated build process is triggered by the CI server. This process compiles the code, resolves dependencies, and generates executable artifacts or libraries.
- Automated Testing:
  After the build is successful, the CI server runs a suite of automated tests. These tests may include unit tests, integration tests, functional tests, and other relevant test cases. The goal is to verify that the code changes do not introduce defects or regressions.
- Test Results Analysis:
  The CI server analyses the test results. If all tests pass, the code changes are considered valid, and the build is marked as successful. If any tests fail, the build is marked as failed, and team members are notified.
- Isolation of Failed Changes:
  Failed code changes are isolated from the main codebase to prevent destabilizing the shared repository. Isolation may involve creating a feature branch specifically for the failed changes.
- Immediate Feedback:
  Developers receive immediate feedback on their code changes. This feedback includes the status of the build and test results, allowing them to address issues promptly.
- Notification and Alerts:
  Notifications and alerts are sent to team members to inform them of the CI results. These notifications can be delivered through various channels, such as email, chat, or a CI dashboard.
- Correction of Failed Tests:
  Developers work on fixing issues identified in the failed tests. They make corrections and commit the changes to the version control system.
- Re-Integration:
  The corrected code changes are integrated into the shared repository, and the CI process is triggered again. This process continues until all tests pass, and the code changes are considered stable.
- Merge to Main Branch:
  Once the code changes pass all tests and are reviewed, they are merged into the main branch or trunk of the version control repository.

Effective CI Automation practices

Effective CI (Continuous Integration) automation practices are crucial for streamlining the development process, maintaining code quality, and ensuring that CI pipelines run smoothly. Here are some best practices for automating CI:

- Regular check-ins.
  Developers to check in their code into the mainline regularly (at least couple of times a day). Small changes are less likely to break the build and easier to revert to the last

committed version. And it is even easier when working on a branch as that means the code is not integrated with other developer's code.

- Create a comprehensive automated test suite.

  There are three types of automated tests that need to be considered in a CI build. These are unit tests, component tests, and acceptance tests. These sets of tests should provide high level of confidence that any introduced change has not broken existing functionality.
- Keep the build and test process short.
- Developers to manage their own development workspace.

  Developers should be able to build, run automated tests and deploy on their local machines using the same processes used in the CI.
- Use CI Software
- Don't check in on a broken build.

  If the build fails, responsible developers will need to find the cause of breakage and fix it quickly. CI process will always identify such breakage and will likely lead to interrupting other developers in the team.
- Always run all commit tests locally

  Running tests locally allows developers to receive immediate feedback on their code changes. If there are any issues, it helps them identify and isolate issues within their code changes.
- Wait for commit tests to pass before moving on.

  Commit tests serve as an initial quality assurance step to ensure that the code changes meet basic functional and quality criteria. Waiting for these tests to pass before proceeding helps maintain the overall code quality.
- Never go home on a broken build.
- Always be prepared to revert to the previous revision.

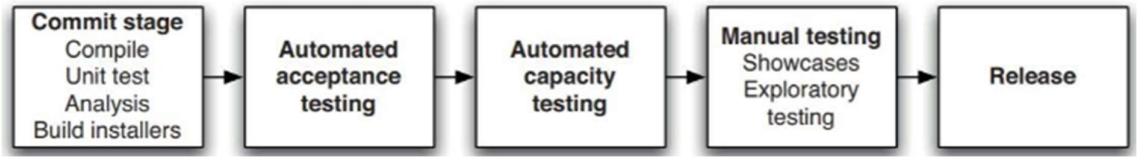
  Software development can involve uncertainty and complexities. Even with the best practices and testing, unexpected issues can arise. Being prepared to revert to the previous revision provides a safety net to mitigate the risk of introducing severe bugs or defects into the production environment.
- Time box fixing before reverting.

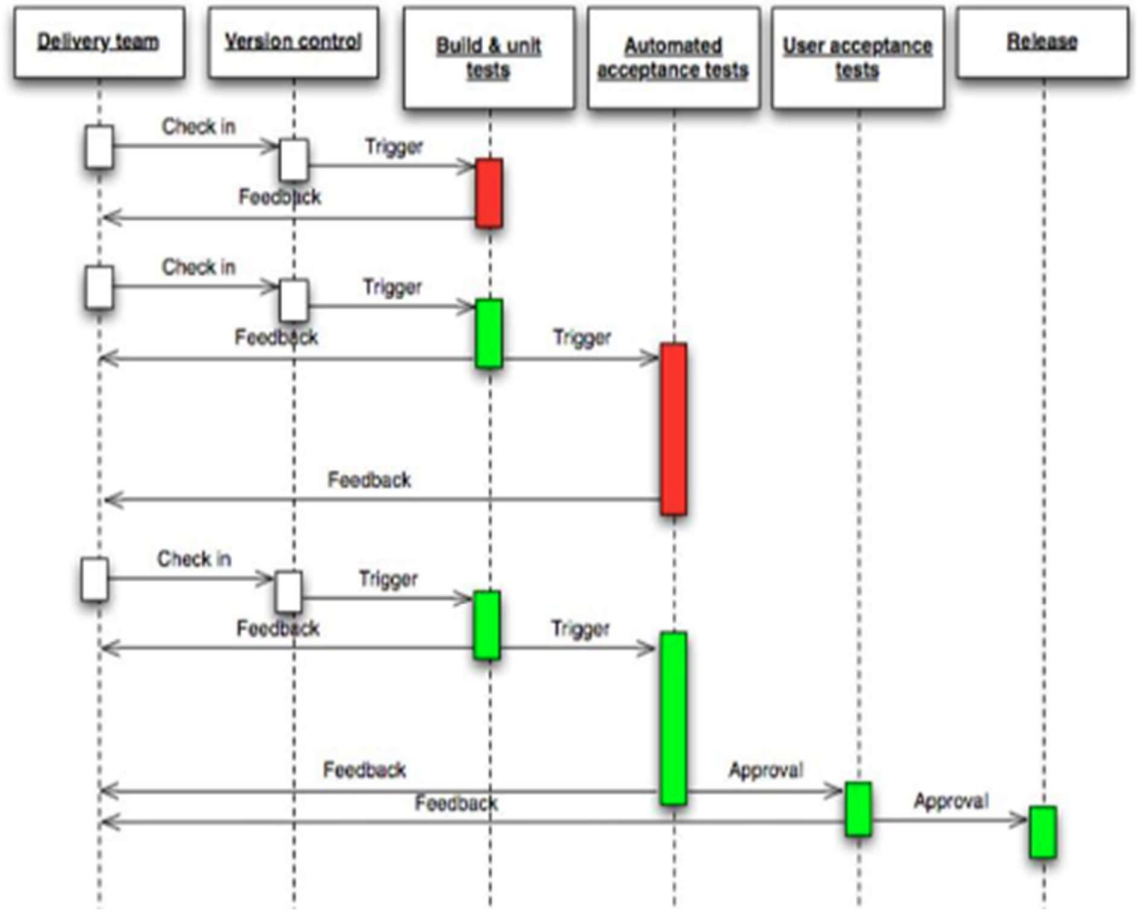
  Setting up a time frame for fixing a bug is one of the best practices as one would not waste a lot of time on fixing the bug. Once the time frame elapses, revert to the last saved checkpoint.
- Do not comment out failing tests.
- Take responsibility for all breakages that result from your changes.

Continuous Delivery

Continuous Delivery (CD) is a software development practice and methodology that focuses on automating the process of delivering software to production or staging environments in a rapid, reliable, and consistent manner. CD builds upon the principles of Continuous Integration (CI) and extends them to include automated deployment, testing, and delivery of code changes. Through continuous delivery practices, the cycle time is in hour and not months. This means that the CD practices enable rapid and frequent releases, reducing the time it takes to deliver new features and improvements to end-users. This can give organizations a competitive edge by responding to market demands more swiftly. In addition to this the rapid feedback loop created by CD helps teams identify issues early, making it easier and less costly to fix problems. This feedback loop supports the principles of continuous improvement.

A Continuous Delivery (CD) deployment pipeline is a sequence of automated steps and stages that code changes go through as they move from development to production. The primary goal of a deployment pipeline is to ensure that code changes are systematically and consistently built, tested, and deployed, leading to the rapid and reliable delivery of software. An example of a deployment practice is given below:

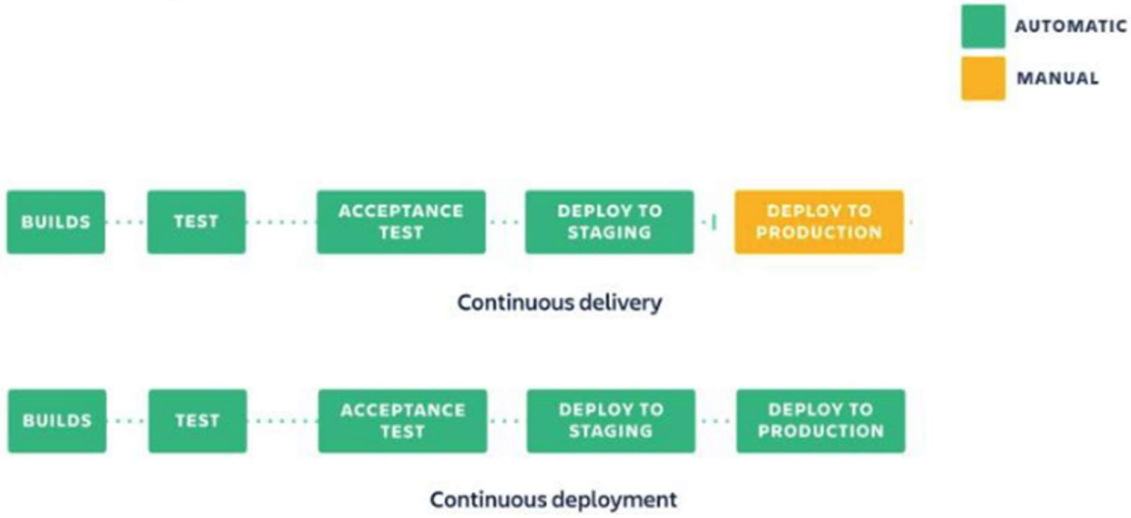




Continuous Deployment

Continuous Deployment (CD) is a software development practice closely related to Continuous Delivery (CD), but with one key distinction: in Continuous Deployment, every code change that passes automated tests and quality checks is automatically and immediately deployed to the production environment without manual intervention. This means that code changes are automatically released to users as soon as they are ready, without any delays or human approval. Continuous Deployment is a practice commonly associated with organizations that prioritize rapid delivery, innovation, and the ability to respond quickly to changing user needs and market conditions. It requires a high degree of confidence in automated testing, a robust deployment pipeline, and a strong culture of collaboration and communication between development and operations teams. While it offers significant benefits

in terms of speed and responsiveness, it also requires careful attention to quality, monitoring, and risk management.



Jenkins

Jenkins is an open-source automation server that is widely used for building, testing, and deploying software. It is a powerful and flexible tool that automates various aspects of the software development lifecycle, making it an integral part of Continuous Integration (CI) and Continuous Delivery (CD) processes. Jenkins is designed to help development and operations teams streamline the development process, improve software quality, and speed up the delivery of applications.

Jenkins, as an automation server, runs in its own execution environment, typically on a dedicated server or cloud instance. The Jenkins server is the core of the execution environment. It hosts the Jenkins application, user interface, and manages all automation tasks. Jenkins can be installed on a physical or virtual machine, or in the cloud. Jenkins's functionality can be extended through a wide range of plugins. The execution environment includes the plugins required for specific automation tasks, such as version control system integrations, build tools, deployment tools, and more. Furthermore, Jenkins can distribute build and test tasks to build agents or nodes, which can be separate physical or virtual machines. These agents can run on various operating systems and architectures. Agents can be set up to execute tasks in parallel, increasing the efficiency of the pipeline.

WEEK 7

Teams

Software development models can significantly influence team structure and interactions in software development. Different development models and methodologies dictate how teams are organized, how they work together, and how they interact with other stakeholders. Some keyways in which software development models can impact team structure are team size and composition, roles and responsibilities, communication and collaboration, project planning and management, feedback loops, client and stakeholder involvement, decision making process, adaptability and flexibility, documentations, and artifacts, etc.

Teams in Waterfall Model

In the Waterfall model, the team structure is typically organized in a sequential and linear fashion, with distinct roles and responsibilities for each team member or group. The Waterfall model follows a rigid, phased approach, where each phase must be completed before moving on to the next one. Here's a typical team structure in the Waterfall model:

- Project Manager:
  The Project Manager is responsible for overall project coordination, planning, and execution. They ensure that the project adheres to the defined schedule and budget. The Project Manager often acts as a point of contact between the development team and stakeholders.
- Business Analyst:
  The Business Analyst works closely with stakeholders to gather and document project requirements. They create detailed requirement documents that serve as the foundation for the entire project.
- System or Software Architects:
  Architects are responsible for designing the system or software. They define the overall system architecture, which includes high-level design and technology stack selection.
- Designers:
  The Designers take the architectural design and create detailed designs for the user interface, database, and other system components. This phase often includes graphic designers, UI/UX designers, and database designers.
- Developers:
  Developers are responsible for coding the software based on the design specifications. In the Waterfall model, development typically starts after the design phase is complete.
- Testers:
  Testers verify that the software meets the specified requirements. They design test cases, perform testing, and report defects. Testing occurs after the development phase.
- Technical Writers:
  Technical Writers create user manuals, system documentation, and other technical documentation necessary for the end-users and maintainers of the software.
- Quality Assurance (QA) Team:
  In some Waterfall projects, a dedicated QA team may exist to oversee quality and compliance with established standards and processes.
- Stakeholders or Clients:
  Stakeholders or clients have an essential role in the Waterfall model. They provide feedback, review documentation and designs, and give approval at various stages of the project.
- Change Control Board (CCB):
  In more formal Waterfall implementations, a Change Control Board may be responsible for evaluating and approving changes to the project scope, requirements, or design. This board ensures that changes do not disrupt the project's defined phases and schedule.

In the Waterfall model, each team or role typically works in isolation within their designated phase. There's little overlap between phases, and progression to the next phase occurs only after the preceding phase is complete. This sequential approach can make it

challenging to accommodate changes once a phase has begun, as the model prioritizes a fixed and well-defined project plan.

Team Structure in Agile Methodology

Agile methodologies promote a different team structure compared to the traditional Waterfall model. Agile focuses on flexibility, collaboration, and iterative development. In Agile, teams are typically smaller and cross-functional, with the aim of delivering working software in short iterations. The most used Agile methodology is Scrum, which has a well-defined team structure. Here's a typical team structure in Agile:

- Product Owner:
  The Product Owner is responsible for defining and prioritizing the product backlog, which is a prioritized list of features and user stories. They act as the liaison between the development team and stakeholders, ensuring that the team works on the most valuable items first.
- Scrum Master:
  The Scrum Master is a servant-leader who facilitates the Scrum process and ensures that the team follows Agile principles. They help the team remove impediments and maintain a productive and collaborative environment.
- Development Team:
  The Development Team is a cross-functional group of professionals who are responsible for delivering the working product. This team includes roles such as developers, testers, designers, and any other skillsets needed to complete the work. Agile promotes self-organization, and team members collectively decide how to complete the work in each sprint.
- Stakeholders or Customers:
  Agile encourages active involvement of stakeholders or customers throughout the development process. They are engaged in sprint reviews and provide feedback on the incrementally developed product.

The Agile team structure is characterized by the following key principles:

- Cross-Functionality:
  Agile teams are cross-functional, meaning they have all the skills and expertise required to take user stories from idea to a potentially shippable product increment. This reduces dependencies and allows for faster decision-making.
- Self-Organization:
  Agile teams are encouraged to self-organize and make decisions about how to achieve their goals. The team collectively decides how to plan and execute the work.
- Iterative and Incremental Development:
  Agile teams work in short timeframes called sprints (usually 2-4 weeks) and deliver potentially shippable increments at the end of each sprint. This allows for continuous improvement and adaptation based on feedback.
- Customer-Centric:
  Agile emphasizes delivering value to customers by focusing on user stories and features that provide the most business value.
- Continuous Collaboration:
  Agile teams have ongoing collaboration with stakeholders and prioritize open communication to gather feedback and adapt the product as needed.

- Emphasis on Individuals and Interactions:
  Agile values individuals and their interactions over processes and tools, fostering a people-centric approach to development.
- Adaptability:
  Agile teams are highly adaptable and responsive to change, which is a fundamental Agile principle.

Overall, Agile methodologies aim to create a dynamic, collaborative, and responsive environment that allows teams to deliver high-quality software that meets the evolving needs of stakeholders. The specific roles and practices may vary between Agile methodologies like Scrum, Kanban, or Extreme Programming (XP), but the core principles of flexibility and collaboration remain consistent. Agile methodologies promote a set of individual practices that are essential for fostering a collaborative and adaptive development environment. These practices are designed to enhance individual productivity, communication, and flexibility. Here are some individual practices commonly associated with Agile:

- Self-Organization:
  Team members are encouraged to take ownership of their work and make decisions related to how they will achieve their goals. This practice empowers individuals to organize their tasks and responsibilities effectively.
- Time-Boxing:
  Agile practices often involve time-boxed activities, such as time-boxed meetings (e.g., daily stand-ups, sprint planning, and retrospectives) and time-boxed work periods (e.g., sprints or iterations). Time-boxing helps individuals focus on tasks within a specific timeframe and promotes discipline and productivity.
- Pair Programming:
  In Agile software development, pair programming is a practice where two developers work together at a single computer, collaborating on the same piece of code. This practice can lead to higher code quality, knowledge sharing, and immediate feedback.
- Test-Driven Development (TDD):
  TDD is a practice where developers write automated tests for a piece of functionality before writing the actual code. This practice ensures that the code is testable and meets the specified requirements.
- Continuous Integration (CI):
  CI is a practice where code changes from multiple team members are frequently integrated into a shared repository. Automated tests are run to detect integration issues early. This practice helps identify and fix problems quickly.
- Refactoring:
  Refactoring is the practice of improving the internal structure and design of the code without changing its external behaviours. Agile teams often refactor their code to maintain and enhance its quality.
- Single-Tasking:
  Agile encourages individuals to focus on one task at a time rather than multitasking. Single tasking can lead to better concentration, reduced context switching, and higher productivity.
- Regular Self-Reflection:
  Agile teams often practice self-reflection through techniques like sprint retrospectives. Team members reflect on what went well, what didn't, and what improvements can be made. This practice promotes continuous learning and adaptation.

- Open and Transparent Communication:
  Agile encourages open and transparent communication among team members. Individuals should be willing to share information, issues, and feedback openly to promote collaboration and problem-solving.
- Incremental Learning:
  Agile values incremental learning and improvement. Team members are encouraged to learn from their experiences and continuously seek opportunities to enhance their skills and knowledge.
- Visual Management:
  Visual management tools like Kanban boards and burndown charts are used to help individuals track progress, identify bottlenecks, and visualize work. This practice aids in transparency and collaboration.
- Empathy and Teamwork:
  Agile places a strong emphasis on empathy and collaboration among team members. Individuals are encouraged to understand each other's perspectives and work together effectively.

Team Dynamics

Team dynamics refer to the psychological and social interactions among members of a group or team as they work together to achieve common goals and objectives. It encompasses the patterns of communication, collaboration, cooperation, and conflict resolution within the team. Team dynamics can significantly impact a team's performance, effectiveness, and overall working environment. Here are some key aspects of team dynamics:

- Communication:
  Effective communication is a cornerstone of good team dynamics. It involves how team members share information, ideas, and feedback. Clear, open, and regular communication fosters understanding and helps prevent misunderstandings or conflicts.
- Collaboration:
  Collaboration is the act of working together to achieve a common goal. In a team, collaboration involves combining the skills, knowledge, and efforts of team members to produce better results collectively than they could individually.
- Roles and Responsibilities:
  Team dynamics are influenced by the distribution of roles and responsibilities among team members. Clarifying and understanding each team member's role is crucial to ensure that work is well-coordinated and that everyone knows their contributions.
- Leadership:
  Leadership dynamics within a team can greatly affect its performance. Effective leaders inspire, guide, and motivate team members, and their leadership style can influence the team's culture and productivity.
- Conflict Resolution:
  Conflicts are a natural part of team dynamics. How a team handles conflicts, whether through constructive dialogue and problem-solving or by allowing conflicts to escalate, can impact the team's cohesion and effectiveness.
- Trust and Psychological Safety:
  Trust is essential for positive team dynamics. Team members need to trust each other to collaborate effectively and share their ideas and concerns. Psychological safety, which is the feeling that it's safe to take risks and be vulnerable in the team, plays a crucial role in fostering trust and open communication.

- Motivation and Morale:

  Team dynamics can influence team members' motivation and morale. A supportive and positive team environment can boost individual and collective motivation, while negative dynamics can have the opposite effect.
- Decision-Making:

  The process of making decisions within a team can shape team dynamics. Decision-making methods, such as consensus, majority vote, or relying on a designated leader, affect how team members engage in the decision-making process.
- Cohesion and Group Identity:

  Cohesion refers to the level of unity and solidarity within a team. A strong sense of group identity and cohesion can enhance team dynamics by promoting a shared commitment to the team's goals.
- Diversity and Inclusion:

  Teams with diverse members bring different perspectives and skills. Managing diversity and fostering inclusion are vital for creating a positive and productive team dynamic that leverages these differences.
- Feedback and Learning:

  Providing and receiving feedback is a key component of team dynamics. Teams that encourage constructive feedback and a culture of continuous learning tend to adapt and improve more effectively.

Tuckman Team Development Model

Tuckman's Team Development Model, also known as "Tuckman's Stages of Group Development," was developed by psychologist Bruce Tuckman in 1965. It describes the different stages that teams go through as they form, develop, and mature.

- Forming:

  In the forming stage, team members come together, often with a sense of excitement and anticipation. They are polite and cautious in their interactions as they get to know each other and understand the team's goals and objectives. Team members may be unclear about their roles, and there's a reliance on the team leader for guidance and direction.
- Storming:

  During the storming stage, team members start to express their individual opinions, ideas, and preferences. Conflicts and disagreements may arise as team

members vie for influence and ownership of tasks. This stage can be challenging as it tests the team's ability to handle differences and conflicts. A team leader or facilitator can play a crucial role in mediating disputes and helping the team find common ground.

- Norming:

  In the norming stage, team members begin to resolve their differences and establish shared norms, values, and procedures. They develop a sense of unity and camaraderie as they agree on how to work together more effectively. Roles and responsibilities become clearer, and team members work collaboratively to achieve common goals.

- Performing:

  The performing stage is characterized by high levels of cooperation and productivity. Team members have successfully resolved their conflicts, developed strong working relationships, and had a clear understanding of their roles. They can work together efficiently and effectively to achieve the team's objectives. The team is at its most productive during this stage.

- Adjourning:

  Tuckman later added the adjourning stage to the model, which addresses the process of winding down the team after its objectives have been achieved. In this stage, team members may experience a sense of loss as the team disbands. They reflect on their achievements and the experiences they've had together.

Team Culture

Team culture, often referred to as group culture or team climate, refers to the shared values, norms, beliefs, attitudes, and behaviours that characterize a particular group of people working together as a team. Team culture plays a critical role in shaping how team members interact, communicate, collaborate, and achieve their goals. It reflects the collective identity and personality of the team.

Effective Teamwork

Effective teamwork is the collaborative effort of a group of individuals working together to achieve a common goal or objective. It involves a combination of strong communication, cooperation, shared goals, trust, and a supportive team culture. Here are key principles and elements of effective teamwork:

- Clear Goals and Objectives:

  Team members should have a clear understanding of the team's goals and what needs to be accomplished. Well-defined objectives help keep the team focused and aligned.

- Role Clarity:

  Each team member should have a clearly defined role and responsibilities. When everyone knows their role, it reduces confusion and overlaps, and ensures that tasks are distributed effectively.

- Open and Effective Communication:

  Effective communication is crucial for team success. Team members should be able to express their ideas, concerns, and feedback openly. This includes active listening and the ability to convey information clearly and concisely.

- Trust and Respect:

  Trust is the foundation of effective teamwork. Team members should trust each other's abilities, integrity, and commitment. Respect for each other's perspectives and contributions is equally important.

- Collaboration:
  Team members should work together to achieve common goals. This means sharing information, knowledge, and resources to solve problems and complete tasks. Collaboration promotes a sense of unity within the team.
- Conflict Resolution:
  Conflicts are a natural part of teamwork. Effective teams have mechanisms in place to address conflicts constructively. This may involve open dialogue, compromise, and seeking win-win solutions.
- Feedback and Continuous Improvement:
  Team members should be comfortable giving and receiving constructive feedback. This promotes ongoing learning and improvement within the team.
- Appropriate Leadership:
  Leadership within a team is important, but it should be facilitative and supportive rather than autocratic. A good team leader helps guide the team, provides support, and fosters an environment where team members can contribute their best.

Team Building

Team building is a process or set of activities designed to enhance the effectiveness, cohesiveness, and collaboration of a group of individuals working together as a team. The primary goal of team building is to improve communication, trust, and problem-solving skills among team members. Here are some common team-building activities and strategies:

- Icebreakers:
  Icebreakers are short, informal activities or games designed to help team members get to know each other and feel more comfortable working together. They are often used at the beginning of team-building sessions or meetings.
- Trust-Building Exercises:
  Trust-building activities aim to build trust among team members. Examples include trust falls, where team members catch a falling colleague, and blindfolded activities that require trust in a partner to navigate.
- Problem-Solving Challenges:
  Problem-solving challenges involve tasks or puzzles that require the team to work together to find solutions. These challenges encourage critical thinking and collaboration.
- Role Clarification:
  Sometimes, teams need to clarify roles and responsibilities to avoid confusion and conflicts. Role-playing or scenario-based discussions can help with this.
- Interpersonal relations:
  Focus on teamwork skills such as giving and receiving support, communication and sharing information.

High performing Agile teams

High-performing Agile teams exhibit several characteristics that set them apart in terms of productivity, collaboration, and their ability to deliver value. These characteristics are essential for achieving success in Agile software development. Agile teams are typically cross-functional, meaning they have a diverse set of skills needed to deliver a complete product. This diversity reduces dependencies on external teams and enables end-to-end development. High-performing Agile teams have a culture of continuous improvement. They regularly reflect on their processes and practices, seeking ways to enhance their efficiency and effectiveness. If the team follows a specific Agile methodology like Scrum, they adhere to its practices and ceremonies, such as daily stand-ups, sprint planning, sprint reviews, and retrospectives.

Tools and technologies for Teamwork

There are numerous tools and technologies available to support teamwork, collaboration, and project management in various professional settings. These tools help teams communicate, share information, track tasks, and work together efficiently. A few of them are:

- Issue tracking systems

  This is a software that manages and maintains lists of issues and is used to create, update, and resolve reported issues internally or externally. GitHub is one of the most common tools used for issue tracking. One can use the project boards for issue tracking as well on GitHub.
- Bug tracking system

  This is a software that keeps track of reported software bugs in software development projects. One of the most used tools is Bugzilla which is an open-sourced web-based bug tracker and testing tool by Mozilla Project.

WEEK 8

Extreme Programming

Extreme Programming (XP) is a software development methodology that emphasizes a set of values, principles, and practices to deliver high-quality software rapidly and efficiently. XP was created by Kent Beck in the late 1990s and gained popularity for its focus on adaptability, customer collaboration, and a strong emphasis on engineering and technical practices. It is often associated with Agile software development and shares many Agile principles.

XP programming focusses on development and delivery of very small increments of functionality and it relies on constant code improvements, user involvement in the development team, and pair wise programming. It emphasizes on Test Driven Development (TDD) as part of the small development iterations and many more. Given below are some key practices of XP programming:

- Pair Programming:

  Developers work in pairs, with one writing code (the "driver") and the other reviewing the code and suggesting improvements (the "navigator"). This practice enhances code quality and knowledge sharing.
- Test-Driven Development (TDD):

  Developers write tests before writing code, ensuring that code meets requirements and can be tested automatically. This practice helps prevent defects and supports continuous integration.
- Continuous Integration (CI):

  Code changes are integrated into the main codebase frequently (often multiple times a day). Automated tests are run to detect integration issues immediately.
- Small Releases:

  XP encourages frequent and small releases, allowing the team to deliver value to customers more frequently.
- Customer Involvement:

  Customers and stakeholders are actively involved throughout the development process. They participate in planning, provide feedback, and prioritize features.
- Sustainable Pace:

  XP emphasizes maintaining a sustainable work pace to prevent burnout and support long-term productivity.

- Collective Code Ownership:
  All team members are responsible for the entire codebase and can make changes or improvements as needed. This fosters a sense of shared responsibility.
- Coding Standards:
  Teams follow coding standards to ensure consistency in code quality and readability.
- Metaphor:
  Teams use a common metaphor or shared understanding of the system to guide development decisions.
- On-Site Customer:
  Ideally, a customer or a customer representative is available to the team daily to answer questions and provide clarifications.
- Refactoring:
  Code is continuously improved through refactoring to maintain its quality and adapt to changing requirements.
- Planning Game:
  XP uses a "planning game" to estimate and prioritize tasks, features, and user stories. This helps in setting clear development goals for each iteration.

SCRUM Methodology

Scrum is an Agile framework for managing and delivering complex projects. It was originally developed for software development but is now applied to various fields, including product development, marketing, and more. Scrum provides a structured approach to iterative and incremental development, emphasizing collaboration, adaptability, and customer satisfaction. It was first introduced by Ken Schwaber and Jeff Sutherland in the early 1990s. Here are the key components and principles of the Scrum methodology:

- Scrum Artifacts:
  - Product Backlog:
    The product backlog is a prioritized list of all the features, enhancements, and bug fixes needed for a product. It is maintained and ordered by the Product Owner.
  - Sprint Backlog:
    The sprint backlog is a subset of the product backlog items selected for a specific sprint. It is the work the team commits to completing in that sprint.
  - Increment:
    An increment is a potentially shippable product version created at the end of each sprint. It includes all completed and tested product backlog items.
- Scrum Events:
  - Sprint:
    A sprint is a time-boxed development iteration, usually lasting two to four weeks. During a sprint, the team works to deliver a set of prioritized backlog items.
  - Sprint Planning:
    At the beginning of each sprint, the team holds a sprint planning meeting to select backlog items and determine how to complete them.
  - Daily Scrum:
    Daily Scrum, also known as the daily stand-up, is a brief daily meeting where team members share progress updates, discuss challenges, and plan the day's work.

- Sprint Review:
  At the end of each sprint, a sprint review meeting is held to demonstrate the work completed, gather feedback, and review the product increment.
- Sprint Retrospective:
  After the sprint review, the team holds a sprint retrospective to reflect on what went well, what didn't, and what can be improved in the next sprint.

- Scrum Values:
  Scrum is guided by five values: commitment, courage, focus, openness, and respect. These values help teams work effectively, make decisions, and foster a positive and productive work environment.

In Scrum, there are several roles and responsibilities that define the members of the Scrum team. These roles collaborate to deliver a product increment during each sprint. The main roles in a Scrum team are:

- Scrum Master:
  - Responsibilities:
    - Facilitate and coach the team in Scrum practices.
    - Ensure that the Scrum framework is understood and followed.
    - Remove impediments that hinder the team's progress.
    - Help the team continuously improve their processes and practices.
    - Organize and facilitate Scrum events (sprint planning, daily Scrum, sprint review, sprint retrospective).
    - Act as a servant-leader, promoting a productive and collaborative team environment.
- Product Owner:
  - Responsibilities:
    - Define and prioritize the product backlog, which is a prioritized list of all work items needed for the product.
    - Ensure that the team works on the most valuable and high-priority items first.
    - Make product-related decisions, answer team questions, and provide clarification during development.
    - Accept or reject work results during the sprint review.
    - Continuously engage with stakeholders to understand their needs and ensure that the product aligns with business goals.
- Development Team:
  - Responsibilities:
    - Cross-functional group of individuals responsible for delivering a potentially shippable product increment by the end of each sprint.
    - Self-organize to determine how to accomplish the sprint backlog items.
    - Estimate the effort required for backlog items and commit to completing a set of them during the sprint.
    - Collaborate to develop, test, and deliver high-quality work.
    - Hold each other accountable for achieving the sprint goal.
    - Participate in sprint planning, daily Scrum, sprint review, and sprint retrospective.

Scrum estimation, also known as Agile estimation, is the process of assigning relative values or sizes to work items in a Scrum project. The primary purpose of estimation in Scrum is to help the team understand the effort required to complete tasks, user stories, or features,

and to assist in making informed decisions regarding priorities and commitments. Scrum progress monitoring, also known as Agile progress monitoring, is the practice of regularly and systematically tracking and evaluating the progress of work in a Scrum project. It involves measuring and assessing how the team is performing in terms of completing tasks, meeting sprint goals, and delivering value to the customer. Scrum progress monitoring is an essential aspect of the Agile framework as it allows teams to adapt to changes and continuously improve their processes. Here are key aspects of Scrum progress monitoring:

- Burndown Charts:

  Burndown charts are graphical representations of the work remaining in a sprint. They provide a visual way to track progress over time, showing whether the team is on track to complete the sprint's scope. Burndown charts can be used to identify potential issues early in the sprint.
- Velocity Tracking:

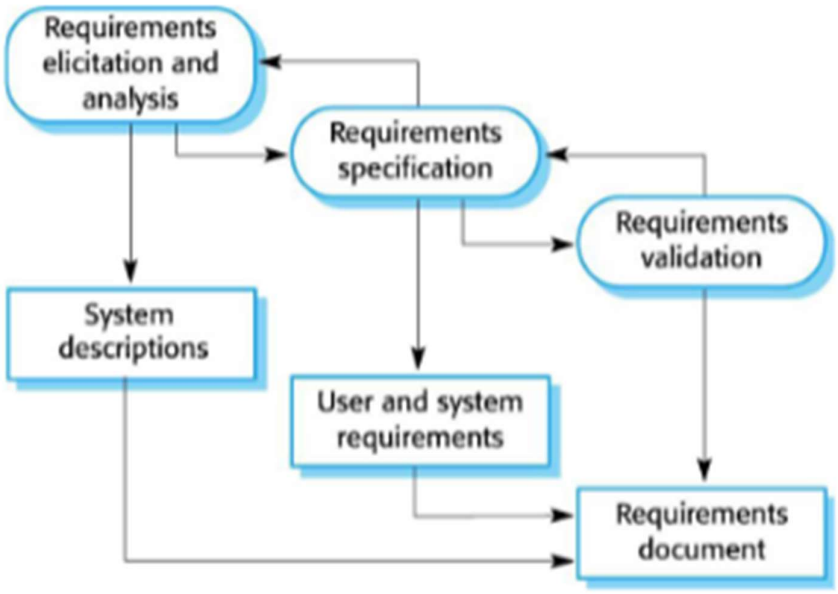
  Velocity is a measure of the amount of work a team can complete in a sprint based on past performance. Teams track their velocity to help plan future sprints and understand how much work they can commit to.
- Sprint Reviews:

  At the end of each sprint, a sprint review is conducted to demonstrate the work completed to stakeholders. This provides an opportunity to gather feedback and make any necessary adjustments based on stakeholder input.

WEEK 9

Requirement Engineering Process

Requirement engineering is a crucial phase in software development and system engineering. It involves gathering, documenting, and managing requirements for a system or software project. The goal of requirement engineering is to define what the system or software needs to do, how it should behave, and what constraints and quality attributes it must satisfy.



The requirement engineering process typically consists of several stages:

- Requirements Elicitation:

  This is the initial phase where requirements are gathered from stakeholders, including customers, users, and domain experts. Techniques such as interviews, surveys, workshops, and observations are used to elicit requirements. The aim is to understand the needs and expectations of all stakeholders.

- Requirements Analysis:
  In this stage, gathered requirements are analysed to ensure they are complete, consistent, and clear. Conflicting or ambiguous requirements are identified and resolved. The analysis phase also involves prioritizing and categorizing requirements.
- Requirements Specification:
  Once the requirements are analysed, they are documented in a formal and structured way. This typically involves creating requirement documents, such as Software Requirements Specifications (SRS), that detail what the system should do, its interfaces, and any constraints or quality attributes.
- Requirements Validation:
  In this phase, the documented requirements are reviewed and validated with stakeholders. Validation ensures that the documented requirements accurately represent the stakeholders' needs and that they are feasible to implement. This often involves reviews, walkthroughs, and inspections.
- Requirements Management:
  As the project progresses, changes to requirements may occur due to evolving stakeholder needs, new information, or other factors. Requirement management involves tracking and controlling changes to requirements, maintaining their traceability, and ensuring that all stakeholders are aware of changes and their implications.
- Requirements Traceability:
  This involves establishing and maintaining relationships between requirements and other project artifacts, such as design documents, test cases, and source code. Traceability helps ensure that each requirement is properly implemented, tested, and validated.
- Requirements Communication:
  Throughout the project, effective communication of requirements to all team members and stakeholders is crucial. This helps ensure that everyone has a common understanding of the project's objectives and requirements.
- Requirements Documentation:
  Keeping accurate and up-to-date documentation is essential throughout the requirement engineering process. Documentation serves as a reference for the development team, testers, and other stakeholders.
- Requirements Verification and Validation:
  Verification ensures that the implemented system or software meets the specified requirements, while validation ensures that it satisfies the actual needs of the users and stakeholders. Testing and quality assurance activities play a significant role in this phase.
- Requirements Change Management:
  Handling changes to requirements is an ongoing process. Changes should be carefully evaluated, documented, and communicated to all relevant parties. The impact of changes on the project schedule and budget should be assessed as well.

There are multiple techniques associated with requirement elicitation such as interviewing stakeholders, cooperating with the clients to create scenarios, creating use cases, etc. Requirement elicitation is mainly done to figure out the functional and non-functional requirements of the project. It is necessary to note that a project has multiple stakeholders and different stakeholders will have different thoughts on the requirements. It is the responsibility of the Requirement Specification Document to list out the requirements that everyone is happy with.

Behaviour Driven Development

Behaviour-Driven Development (BDD) is a software development methodology that extends the principles of Test-Driven Development (TDD) and focuses on improving communication and collaboration among developers, testers, and non-technical stakeholders. BDD aims to ensure that the software being developed meets the desired behaviour and functionality while providing clear and easily understandable documentation. It is a conceptual approach for specifying application's behaviour and communicating them clearly among stakeholders. Considering BDD starts with a focus on expected behaviour of the software from the user's perspective, BDD clearly has a User Centric Approach. BDD scenarios are typically structured using the "Given-When-Then" format, where:

- Given: Describes the initial context or preconditions.
- When: Describes the specific action or event that triggers the behaviour.
- Then: Describes the expected outcomes or results.

The requirements for BDD are quite simple. It requires user stories to elicit functional requirements, Low fidelity User Interfaces and story boards to elicit the UIs and transfer user stories into acceptance tests. User stories are short and concise description of how the application is expected to be used from the user's point of view. These user stories help the stakeholders to plan and prioritize development, improve requirements clarity, document functional requirements as executable scenarios, etc. User stories need to be Specific, Measurable, Achievable, Relevant, and Time boxed. In short user stories need to be SMART.

Sprint Planning

Sprint planning is a critical ceremony in Agile software development methodologies, such as Scrum. It is a collaborative meeting where the development team, including developers, testers, and sometimes designers, work with the product owner to plan and commit to a set of work for an upcoming sprint, which is typically a time-boxed period (e.g., two weeks) during which the team will work on a specific set of features or user stories. The primary goals of sprint planning are to ensure that the team understands the work to be done, agrees on the scope for the sprint, and commits to delivering a potentially shippable product increment at the end of the sprint. The steps are usually:

- Break the defined stories down into tasks.
    - The stories need to be broken down by team members.
    - Each task should be placed on a separate card.
    - And uncompleted user stories should be put on a single card to plan out the story.
- Group stories and their tasks together.
    - Once the stories and tasks are grouped together, they are added to the To-Do section of the Sprint Backlog (Task board).
- Team members work on the tasks.
    - Each team member works on one task.
    - As soon as the task is being worked on, it is moved into the In-Progress section of the Sprint Backlog (Task Board).
    - Team members can add additional tasks to the board to finish stories and communicate it in the daily scrum.
- Finishing the task/story.
    - Once the task has been completed, the card is placed under the Done section of the Sprint Backlog (Task Board).

- The team member to finish the final task of a story needs to verify that all conditions of satisfaction have been met with the Product Owner before moving it to the Done Section.

User Interfaces, Scenarios, and Storyboard

In Scrum, User Interface (UI) sketches are informal, low-fidelity drawings or diagrams that help visualize and communicate the user interface design and layout of a software product or feature. They are a helpful tool during the early stages of product development, especially in sprint planning and backlog grooming. The UI Sketches can be of two types:

- Low Fidelity UIs
  - Purpose:
    Low-fidelity UI design is used primarily in the early stages of the design process, such as brainstorming, concept exploration, and initial idea validation. It is ideal for quickly sketching out ideas and concepts.
  - Level of Detail:
    Low-fidelity UI designs are simple, abstract, and lack fine-grained details. They focus on the basic layout and structure of the user interface.
  - Medium:
    Low-fidelity UI can be created using paper and pencil, whiteboards, or digital tools that simulate hand-drawn sketches. They are intentionally rough and not meant to resemble the final product.
  - Realism:
    Low-fidelity UI designs are not realistic and do not attempt to replicate the exact appearance of the final product. They may consist of basic shapes, lines, and text.
  - Functionality:
    They do not include interactive elements, animations, or detailed visual assets. Low-fidelity designs are static and serve to convey the general layout and flow.
  - Iterative:
    Low-fidelity designs are easy to create and modify, making them suitable for quick iterations and feedback gathering.
  - Usage:
    These are often used for concept validation, early brainstorming sessions, and initial discussions with stakeholders.
- High Fidelity UIs
  - Purpose:
    High-fidelity UI design is used in later stages of the design process when the overall layout and functionality of the user interface have been finalized. It aims to create a highly realistic representation of the final product.
  - Level of Detail:
    High-fidelity UI designs are rich in detail, including accurate typography, colour schemes, graphics, and interactive elements. They represent the final product as closely as possible.
  - Medium:
    High-fidelity UI designs are typically created using design software, such as Adobe XD, Sketch, Figma, or similar tools. Designers use these tools to create pixel-perfect representations of the user interface.

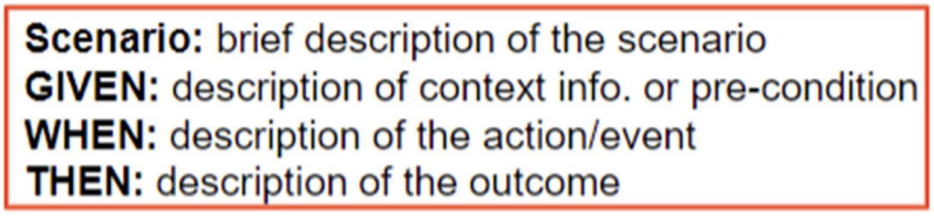
- Realism:
  High-fidelity UI designs aim to be highly realistic and visually accurate. They may incorporate real content and images to provide a true-to-life representation.
- Functionality:
  High-fidelity designs often include interactive elements, transitions, and animations, making them more closely resemble the actual user experience.
- Final Validation:
  They are used for final validation, usability testing, and as references for development teams to implement the user interface as designed.
- Usage:
  High-fidelity designs are employed to create a final, polished, and production-ready look and feel for the user interface. They are used to guide developers in the implementation process and provide a clear vision for stakeholders.

"Scenarios" and "storyboards" are both terms commonly used in the fields of user experience (UX) design and user interface (UI) design to create and communicate design ideas and concepts. More is explained about them below:

- Scenarios
  - Purpose:
    Scenarios are used to describe how a user interacts with a product or system in a specific situation or context. They focus on the user's goals, actions, and the system's responses.
  - Content:
    Scenarios typically include a narrative that explains the user's objectives, the actions they take to achieve those objectives, and the system's responses to those actions.
  - Format:
    Scenarios are usually presented as written narratives or descriptions. They can be textual descriptions of a user's journey, often including details about their motivations, the environment, and the steps they take to accomplish their goals.
  - Visualization:
    Scenarios may also incorporate visuals like images, diagrams, or sketches to help illustrate the user's interactions or the context in which the interactions occur.
  - Usage:
    Scenarios are valuable for understanding user needs, pain points, and desired interactions. They help designers and stakeholders empathize with users and make design decisions that align with user goals.
- Story board
  - Purpose:
    Storyboards are used to visually illustrate a sequence of events or interactions in a user's journey. They provide a step-by-step depiction of a user's experience with a product or system.
  - Content:
    Storyboards consist of a series of frames or panels, each representing a specific moment in the user's journey. These frames depict what the user sees, does, and experiences.

- Format:
  Storyboards are typically presented as a series of sketches, drawings, or digital illustrations arranged in a sequential order.
- Visualization:
  Storyboards are highly visual and help convey the user's experience more tangibly. They often include depictions of user interfaces, user actions, and system responses.
- Usage:
  Storyboards are commonly used in the early stages of design to visualize user flows and interactions. They help designers and teams understand the user's path through a system and identify potential usability issues or design improvements.

A typical scenario format is as follows:



Different types of scenarios are often used in the context of scenario planning and scenario analysis. They refer to different approaches for considering and preparing for possible future events, developments, or situations. Here's what each of these terms means:

- Explicit Scenario
  - Definition:
    Explicit scenarios are future scenarios that are carefully and clearly defined, typically through a structured and systematic process. They are developed based on a thorough analysis of known data, trends, and events.
  - Characteristics:
    Explicit scenarios are characterized by being well-documented, specific, and often quantifiable. They may involve precise data, metrics, and assumptions to create a clear picture of possible future outcomes.
  - Usage:
    Explicit scenarios are used when there is a relatively high degree of certainty or predictability in the available data and information. They are common in strategic planning, financial forecasting, and risk management where historical and current data can be used to project potential future states.
  - Example:
    In the context of financial planning, an explicit scenario might involve projecting future revenue and expenses based on historical financial data and known market trends.
- Implicit Scenario
  - Definition:
    Implicit scenarios are less formally defined and structured than explicit scenarios. They often involve more speculative or less well-understood elements of the future, relying on assumptions, educated guesses, or qualitative information.

- Characteristics:

  Implicit scenarios are characterized by their subjectivity and lack of precision. They may rely on expert opinions, qualitative information, or less quantifiable data.
- Usage:

  Implicit scenarios are employed when there is a higher degree of uncertainty, complexity, or ambiguity about the future. They are used to explore a wider range of possible outcomes, especially when there is limited historical data or when dealing with novel and unpredictable situations.
- Example:

  In the context of technology innovation, an implicit scenario might involve speculating on the potential impact of emerging technologies on a specific industry, even when there is limited historical data to support precise predictions.

- Imperative Scenario
  - Definition:

    Imperative scenarios focus on specifying step-by-step instructions for how a task should be executed. In imperative programming, the emphasis is on detailing the sequence of actions to achieve a specific outcome.
  - Characteristics:

    Imperative scenarios are characterized by explicit and detailed instructions that define the control flow, data manipulation, and the order in which actions are performed. It tells the computer “How" to accomplish a task.
  - Usage:

    Imperative scenarios are commonly used in languages like C, C++, and Java. They are well-suited for situations where fine-grained control over program execution is required, such as low-level system programming and performance optimization.
  - Example:

    In an imperative scenario for sorting an array of numbers, you would specify the algorithm step by step, including the loop structure, conditionals, and data movement operations.
- Declarative Scenario
  - Definition:

    Declarative scenarios focus on specifying what needs to be accomplished rather than the specific steps to achieve it. In declarative programming, the emphasis is on describing the desired outcome or state.
  - Characteristics:

    Declarative scenarios are characterized by a higher level of abstraction. Instead of detailing the process, they specify the properties, relationships, or constraints that should hold true. It tells the computer “What" should be achieved.
  - Usage:

    Declarative scenarios are commonly used in languages like SQL (for querying databases), HTML (for web page structure), and certain aspects of functional programming. They are well-suited for situations where you want to express the desired results without getting bogged down in implementation details.

- Example:

  In a declarative scenario for sorting, you would specify that you want the array sorted in ascending order without specifying the exact sorting algorithm to use.

WEEK 10

Agile Methods for estimating Size

Agile methods for estimating size refer to techniques and approaches used in Agile software development to estimate the size or effort required to complete various tasks or user stories. Estimation is a crucial aspect of Agile development as it helps in planning, prioritizing, and managing work during the development process. Agile methods for estimating size are typically characterized by their collaborative, iterative, and relative nature, emphasizing team involvement and adaptability. Some common Agile methods for estimating size include:

- Story Points

  Story points are a relative estimation technique commonly used in Agile frameworks like Scrum. Each user story or task is assigned a story point value, representing its perceived complexity or size compared to other stories. Story points are typically determined during team discussions and can be based on factors like effort, complexity, risk, and uncertainty. They are a fundamental component of Agile estimation, especially in Scrum, and play a significant role in sprint planning, backlog prioritization, and tracking progress. In this, a point value is assigned to each item and the number of story points is the overall size of the story. There are different estimation scales associated with this such as the Fibonacci sequence or subsequent number as twice as the number that precedes it. There are 2 approaches to story points:
  - The smallest story is estimated to be at 1 story point.
  - Medium sized story is assigned a number somewhere in the middle of the range you expect to use.

  Estimating in story points completely separates the estimation of effort from the estimation of duration.
- Ideal days vs Elapsed Days

  "Ideal days" and "elapsed days" are two different measures used in project management and software development to track and estimate time. They are often used in similar contexts but represent distinct concepts:
  - Ideal Days:
    - Definition:

      Ideal days, also known as “workdays” or “effort days,” are a measure of how long a task or user story is expected to take when considering only the actual working time required to complete it. In other words, ideal days represent the amount of time a task would take if there were no interruptions, distractions, or other non-productive activities during the workday.
    - Assumptions:

      Ideal days assume that the individual or team working on the task can work at their full productivity capacity without any external factors that might affect their efficiency.
    - Use:

      Ideal days are commonly used for estimating the effort or duration of a task during sprint planning or backlog grooming in Agile methodologies. They are often used to set expectations for how much work can be completed within a specific time frame.

- Elapsed Days:
  - Definition:

    Elapsed days, also known as "calendar days" or simply "days," represent the actual passage of time, including weekends, holidays, and non-working days. Elapsed days account for the total amount of time that passes from the start to the completion of a task or project.
  - Assumptions:

    Elapsed days consider the real-world conditions that can affect the progress of a project, such as non-working days, waiting for external approvals, and other delays.
  - Use:

    Elapsed days are used to measure the actual time it takes to complete tasks, milestones, or the entire project. They are often used for tracking project schedules, deadlines, and overall project duration.

Estimation Techniques

Estimation techniques are methods and approaches used to assess and predict the time, effort, or other resources required to complete tasks, projects, or activities. Estimation is a crucial aspect of project management, software development, and many other fields. There are several different estimation techniques, each with its own strengths, weaknesses, and best-use cases. Here are some of the most common estimation techniques:

- Expert Judgment:

  This technique involves consulting experts or experienced individuals in the field to provide estimates based on their knowledge and expertise. Expert judgment is particularly useful when there is a lack of historical data.
- Analogous Estimation:

  Analogous estimation, also known as top-down estimation, relies on historical data from similar projects to estimate the effort and duration of the current project. It is often based on the assumption that the current project is comparable to past projects.
- Parametric Estimation:

  Parametric estimation uses statistical relationships and data from previous projects to make estimations. It involves creating mathematical models that consider various factors, such as the size of the project, complexity, and productivity rates.
- Three-Point Estimation:

  This technique uses a weighted average of three estimates: the most optimistic (best-case), the most pessimistic (worst-case), and the most likely estimates. It provides a range of potential outcomes, which can be used for risk analysis.
- Bottom-Up Estimation:

  In bottom-up estimation, tasks or components are estimated individually, and these estimates are then rolled up to create an overall estimate for the project. This approach provides a detailed estimate but can be time-consuming.
- Delphi Technique:

  The Delphi technique is a consensus-based estimation method that involves multiple experts providing estimates independently. These estimates are then anonymized and discussed until a consensus is reached.
- PERT (Program Evaluation and Review Technique):

  PERT is a variation of three-point estimation that places more weight on the most likely estimate. It is commonly used in project management to estimate task durations and calculate project completion times.
- Historical Data Estimation:

Historical data estimation relies on past project data to estimate future projects. It is particularly useful when there is a repository of data from completed projects that can be analysed for estimating similar projects.

- Story Points and Planning Poker:

  These techniques are commonly used in Agile software development. Story points are used to estimate the relative size and complexity of user stories, while Planning Poker involves team members discussing and voting on estimates collaboratively.

Estimating Progress

Estimating progress is the process of assessing and quantifying how much work has been completed in a project or task relative to the total work that needs to be done. It's a fundamental aspect of project management and is crucial for tracking a project's status, meeting deadlines, and making informed decisions. There are several methods and techniques for estimating progress:

- Percentage of Completion:

  This method involves expressing progress as a percentage. For example, if you've completed 30 out of 60 tasks, you've completed 50% of the work. This method is straightforward and easy to understand.

- Work Breakdown Structure (WBS):

  In a WBS, you break down the project into smaller, more manageable tasks or work packages. As each task is completed, you can track progress based on how many work packages are finished compared to the total number.

- Milestone Tracking:

  Milestones are significant project events or achievements. You can track progress by monitoring how many milestones have been reached. For example, you might have completed four out of eight milestones, indicating 50% progress.

- Story Points:

  In Agile methodologies like Scrum, story points are used to estimate the size and complexity of user stories or tasks. As user stories are completed, you can track progress based on the sum of story points completed versus the total story points planned for the sprint.

- Burn-Down Charts:

  Burn-down charts are common in Agile project management. They visually represent how work decreases over time. The x-axis represents time, while the y-axis represents the remaining work. The slope of the burn-down line provides insight into the rate of progress.

- Daily Stand-Up Meetings:

  In daily stand-up meetings, team members provide updates on what they've completed since the previous meeting and what they plan to work on next. This method helps teams track daily progress.

In the context of Agile software development, "velocity" refers to a measure of a team's capacity to complete work in a specific time frame, typically within a sprint or iteration. Velocity is a key metric used to predict and plan the amount of work a team can reasonably accomplish in future sprints. It provides a basis for setting expectations and managing the scope of work in Agile projects. Velocity is usually expressed as the total number of story points or other estimation units (e.g., ideal days) that a team completes in a single sprint. It represents the "output" or productivity of the team within a fixed time frame. At the end of each sprint, the team calculates its velocity by summing up the story points of all the user stories and tasks

that were successfully completed and delivered within that sprint. This becomes the team's velocity for that sprint. For example:

$$3\ stories * 5\ Story\ points\ (each) = Velocity = 15$$

Using this, the number of sprints can also be estimated easily. Calculate the sum of story point estimates for all desired features and then divide the sum with the velocity to get the estimated number of sprints (iterations).

Tools to tracking project progress.

Tracking project progress is a crucial aspect of project management, regardless of the project's size or complexity. It involves monitoring and measuring how well a project is advancing toward its goals, staying on schedule, and adhering to the budget. Effective progress tracking allows project managers and teams to identify and address issues early, make informed decisions, and ensure successful project completion. An excellent tool to use is a burndown chart.

In Agile and Scrum methodologies, burndown charts are a visual tool used to track project progress, specifically the progress of work within a sprint or iteration. Burndown charts provide a clear and real-time representation of how well a team is performing and whether they are likely to complete the planned work by the end of the sprint. Following is the step-by-step process of creating burndown charts:

Burndown Charts based on Story Points

- x-axis: first and end dates of the sprint
- y-axis: story points (0 to 20% more than the total no. of points in the Sprint)
- The plot: a user story is "Done", plot the number of points left in the sprint, at the current day
  - Fill in more days in the chart as more stories are finished
- More work needs to be added (discovered during daily scrum)
  - Estimate amount of points to remove to balance out the Sprint
  - Add card(s) to the task board, follow-up team meeting to estimate added work
  - Add notes to the chart
- As you get close to the end of the Sprint, more points burn off the chart

WEEK 11

Ethics

Ethics refers to the study and practice of principles or standards of right and wrong conduct. It is a branch of philosophy that explores concepts such as morality, fairness, justice, and what constitutes good and bad behaviour in various contexts. Ethics provides a framework for individuals, organizations, and societies to make decisions and judgments about what is considered morally acceptable or unacceptable. Ethics and morals are related concepts that deal with principles of right and wrong conduct, but they are distinct in terms of their scope, origin, and application. Here are the key differences between ethics and morals:

- Scope:
  - Morals:

    Morals are personal, individual beliefs and values that guide an individual's behaviour. They are often deeply rooted in an individual's

upbringing, culture, religion, and personal experiences. Morals represent an individual's inner sense of what is right or wrong.

- Ethics:

  Ethics is a broader, more systematic framework that encompasses the principles and rules of right and wrong behaviour that apply to a group, organization, profession, or society as a whole. Ethics provides a shared standard of conduct for a collective entity.

- Origin:
  - Morals:

    Morals are often shaped by an individual's upbringing, family, culture, religion, and personal experiences. They can be highly subjective and vary from person to person.
  - Ethics:

    Ethics can be influenced by a variety of sources, including legal systems, professional codes of conduct, cultural norms, and societal values. Ethical principles are more objective and have a broader consensus within a particular group or context.
- Application:
  - Morals:

    Morals guide an individual's personal decisions and behaviour. They are subjective and may not always align with the ethical standards of a larger group or society.
  - Ethics:

    Ethics provides a framework for making decisions and judgments about what is considered morally acceptable or unacceptable in a specific context. It applies to a group, organization, or society and helps standardize behaviour within that context.
- Flexibility:
  - Morals:

    Morals are highly subjective and may be more resistant to change. They are deeply ingrained in an individual's belief system and can vary widely between people.
  - Ethics:

    Ethics can be more adaptable and flexible because they are determined by consensus and can evolve to reflect changing social norms and values. Ethical standards within a profession or organization may be updated or revised over time.
- Enforcement:
  - Morals:

    Morals are not typically enforceable by external authorities or legal systems. Violating one's personal moral code may result in feelings of guilt or internal conflict.
  - Ethics:

    Ethical standards are often codified in codes of conduct, professional regulations, and legal frameworks. Violating ethical standards can result in disciplinary actions, legal consequences, or societal disapproval.
- Relativity:
  - Morals:

    Morals can be highly relative and may differ from one person to another. What is considered morally right by one person may not be the same for another.

- Ethics:

  Ethical standards can also vary between different groups and contexts, but they are generally more standardized and subject to collective consensus within that specific context.

Professional Frameworks

Professional frameworks, often referred to as professional frameworks or professional bodies, are organizations or associations that represent and support professionals in specific fields or industries. These frameworks play a crucial role in establishing and maintaining standards, promoting best practices, providing education, and training, and advocating for the interests of professionals within their respective domains. Professional frameworks serve as valuable resources for individuals seeking to excel in their careers and for industries to maintain quality and integrity. There are different Frameworks such as:

- ACS Values
  - The Primacy of the Public Interest
    - You will place the interests of the public above those of personal, business or sectional interests.
  - The Enhancement of Quality of Life
    - You will strive to enhance the quality of life of those affected by your work.
  - Honesty
    - You will be honest in your representation of skills, knowledge, services and products.
  - Competence
    - You will work competently and diligently for your stakeholders.
  - Professional Development
    - You will enhance your own professional development, and that of your staff.
  - Professionalism
    - You will enhance the integrity of the ACS and the respect of its members for each other.
- IEEE CS
  - **Public:** Software engineers shall act consistently with the public interest.
  - **Client and employer:** Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.
  - **Product:** Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
  - **Judgment:** Software engineers shall maintain integrity and independence in their professional judgment.
  - **Management:** Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
  - **Profession:** Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
  - **Colleagues:** Software engineers shall be fair to and supportive of their colleagues.
  - **Self:** Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.
- ACM
  - General Ethical Principles
    - Society and human well-being, avoid hard, be honest and trustworthy, be fair, respect work, respect privacy, honour confidentiality
  - Professional Responsibilities
    - More specific aspects including quality of work and processes, quality of reviews and judgement, competence
  - Professional Leadership Principles
    - Those have leadership roles including quality of work, training and competence,
      public interest decisions to use or retire systems

Ethical Responsibility

Ethical responsibility, often referred to as ethical or moral responsibility, is the obligation or duty of individuals, organizations, or institutions to act in ways that are morally right, just, and principled. It involves adhering to a set of ethical principles, values, and norms that guide behaviour and decision-making, with a focus on doing what is considered morally acceptable and avoiding actions that are perceived as unethical. Ethical responsibility in software development is shared among various stakeholders and entities involved in the software development process. The exact distribution of ethical responsibility can vary depending on the specific context and roles.

Privacy

Another important concept which comes in view is privacy. Privacy is the ability of an individual or group to seclude information, personal data, and activities from public scrutiny or unauthorized access. It encompasses the right to keep certain aspects of one's life, identity, and communications confidential and free from intrusion. Privacy is a fundamental human right and plays a critical role in various aspects of life, including personal autonomy, security, and the protection of personal information. Furthermore, there is another concept called as Product Liability. Product liability in the context of software development refers to the legal responsibility and potential liability of software developers, manufacturers, or distributors for harm or damages caused by defects, errors, or vulnerabilities in the software they have produced or distributed. This concept extends the principles of product liability, which are typically applied to physical products, to the realm of software and digital products.

Product Liability and Software Liability

In terms of software development, product liability can also be called as software liability. Like product liability, Software liability, also known as software product liability, refers to the legal and financial responsibility of software developers, manufacturers, vendors, or distributors for any harm, damages, or losses caused by defects, errors, or vulnerabilities in their software products. It is a subset of product liability specifically applied to software and digital products. Software liability encompasses a range of issues, including security breaches, data loss, system failures, and other types of harm that may result from the use of software.

Intellectual Property

Another important concept to note is Intellectual Property (IP). Intellectual property (IP) refers to legal rights that are granted to individuals or entities for their creations or inventions. These creations can be in various forms, including inventions, original works of authorship, trade secrets, and brand names. Intellectual property rights protect the creations and allow their creators or owners to control how they are used, shared, and monetized. IP rights are essential for fostering innovation and creativity while providing incentives for individuals and organizations to invest in new ideas and developments. These are usually a category of property that includes intangible creations of the human intellect. IP is intangible and hence is challenging to protect. There are various types of IP:

- Copyright
  - Grant a creator of an original work exclusive right it
  - Source code, executable code, database, artistic work
  - Granted automatically in Australia, not all countries though
  - Does not protect ideas or methods employed
- Trademarks
  - Sign, design or expression which distinguishes products or services
  - Formal registration and require meeting certain criteria
- Trades secretes
  - Process, formula, practice, design, pattern which is unknown and provide a competitive advantage

Licensing

In the context of software, licensing refers to the legal permission or authorization granted by the software copyright holder (licensor) to another party (licensee) to use, distribute, or otherwise interact with the software. A software license defines the terms and conditions under which the software can be used and the rights and restrictions that apply to the licensee. Software licensing plays a critical role in regulating the distribution and usage of software products while protecting the intellectual property rights of the software developer or publisher. There are different types of licenses such as:

MIT License

- A short and simple permissive license
- Require preservation of copyright and license notices
- License works, modifications, and larger works may be distributed under different terms and without source code.

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Private use	•License and copyright notice	•Liability •Warranty

- Example: Ruby

Apache License 2.0

- Require preservation of copyright and license notices.
- Contributors provide an express grant of patent rights.
- Licensed works, modifications, and larger works may be distributed under different terms and without source code

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use	•License and copyright notice •State changes	•Liability •Trademark use •Warranty

- Example: PDF.JS

Mozilla Public License 2.0

- Conditioned on making available source code of licensed files and modifications of those files under the same license
- Copyright and license notices must be preserved.
- Contributors provide an express grant of patent rights
- a larger work using the licensed work may be distributed under different terms and without source code for files added in the larger work.

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use	•Disclose source •License and copyright notice •Same license (file)	•Liability •Trademark use •Warranty

GNU Affero General Public License(AGPL) v3.0

- Conditioned on making available complete source code of licensed works and modifications
- Copyright and license notices must be preserved.
- Contributors provide an express grant of patent rights.

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use	•Disclose source •License and copyright notice •Network use is distribution •Same license •State changes	•Liability •Warranty

GNU General Public License GPLv3.0

- Conditioned on making available complete source code of licensed works and modifications
- include larger works using a licensed work, under the same license
- Copyright and license notices must be preserved
- Contributors provide an express grant of patent rights.

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use	•Disclose source •License and copyright notice •Same license •State changes	•Liability •Warranty

GNU Lesser General Public License (GNU LGPLv3)

- Conditioned on making available complete source code of licensed works and modifications under the same license or the GNU GPLv3.
- Copyright and license notices must be preserved.
- Contributors provide an express grant of patent rights.

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use	•Disclose source •License and copyright notice •Same license (library) •State changes	•Liability •Warranty

The Unlicense

- Unlicensed works, modifications, and larger works may be distributed under different terms and without source code
- Example : Youtube-dl

Permissions	Conditions	Limitations
•Commercial use •Distribution •Modification •Patent use •Private use		•Liability •Warranty

Open-Source Software

Open-source software refers to computer software with a specific licensing model that grants users the freedom to access, use, modify, and distribute the software's source code. This means that the source code, which is the human-readable code that software developers write, is made available to the public and can be freely examined, modified, and shared. Open-source software is typically distributed under licenses that encourage collaboration and the principles of openness, transparency, and community-driven development.