CT417

Software Engineering III

Name: Andrew Hayes
E-mail: a.hayes18@universityofgalway.ie
Student ID: 21321503

2024–09–21

Contents

1 Introduction **1**
1.1 Lecturer Contact Details . . . 1
1.2 Grading . . . 1

2 Revision **1**
2.1 What is Software? . . . 1
2.2 Functional vs Non-Functional Requirements . . . 1
2.3 What is Software Engineering? . . . 2
2.4 What are Software Development Life Cycles? . . . 2
2.5 What is a Framework? . . . 2
2.6 Agile & DevOps . . . 2
2.6.1 What is Agile? . . . 2
2.6.2 Agile Principles . . . 2
2.6.3 Agile Frameworks . . . 2
2.6.4 What is DevOps? . . . 3
2.6.5 DevOps Core Practices . . . 3
2.6.6 Key Differences between Agile & DevOps . . . 3
2.6.7 Why DevOps Complements Agile . . . 3
2.6.8 Benefits of Agile & DevOps . . . 3

3 Version Control **4**
3.1 What is Version Control? . . . 4
3.2 What files should be checked in to Version Control? . . . 4
3.3 Centralised Version Control – Subversion . . . 4
3.4 Distributed Version Control – Git . . . 7
3.4.1 GitHub . . . 7
3.4.2 Git Commands . . . 7
3.4.3 Pull Requests . . . 7

4 Build Tools **8**
4.1 Build Tools in CI/CD Pipelines . . . 9
4.2 Maven . . . 9
4.2.1 Plugins . . . 10
4.2.2 Build Lifecycle . . . 11
4.2.3 Dependency Management . . . 12
4.2.4 Local & Remote Repository . . . 12
4.3 Spring Boot . . . 12
4.3.1 Spring vs Spring Boot . . . 12
4.4 GitHub Actions . . . 13

1 Introduction

1.1 Lecturer Contact Details

- Dr. Effirul Ramlan.
- Email: `effirul.ramlan@universityofgalway.ie`.
- Will attempt to reply to emails immediately between the hours of 09:00 & 20:00 from Week 01 to Week 12.
- Discord server: `https://discord.gg/CRAtHv9uNg`.

1.2 Grading

- Continuous Assessment: 40%.
  - You will work in pairs on a software project with three key submissions across the 12 weeks. Each deliverable will align with the topics covered in the course up to that point, allowing for continuous progress assessment.
  - AS-01: Set up musicFinder and configure the CI/CD pipeline (Week 4).
  - AS-02: Testing, Security, & Expanded Application (Week 8).
  - AS-03: Refactoring & Application Deployment.
- Final Exam: 60%.
  - Typical 2-hour exam paper covering materials from Week 1 to Week 12, with nothing out of the ordinary – "You can be sure of that".
  - "There is a question on Agile on your final" – key differences between agile & DevOps table.

2 Revision

2.1 What is Software?

Software consists of:

i. Instruction (computer programs) that when executed provide desired features, function, & performance.

ii. Data structures (Arrays, Objects, Lists, Dictionaries, Maps, etc.) that enable programs to manipulate information.

iii. Descriptive information in both hard copy & virtual format describing the operation & use.

2.2 Functional vs Non-Functional Requirements

Functional Requirement	**Non-Functional Requirement**
Describes the actions with which the user's work is concerned	Describes the experience of the user while doing the work
A feature or function that can be captured in use-cases	A global constraint (and therefore difficult to capture in use-cases)
A behaviour that can be analysed via sequence diagrams or state machines	A software quality
can be usually traced back to a single module / class / function	Usually cannot be implemented in a single module or even program

Table 1: Functional vs Non-Functional Requirements

Typical non-functional requirements include: availability, maintainability, performance, privacy, reliablility, scalability, & security.

2.3 What is Software Engineering?

Software Engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers. Software Engineering encompasses a process, a collection of methods, & an array of tools that allow professionals to build high-quality software.

DevOps outlines a software development process that increases the delivery of higher quality software by integrating the efforts of the development & IT operation teams.

$$\text{DevOps} = \text{Software Engineering} + \text{IT Operations}$$

2.4 What are Software Development Life Cycles?

Software Development Life Cycles (SDLC) refers to a process used by software engineers to design, develop, & test software. Each approach focuses on a different aspect of development, from planning to continuous improvement.

2.5 What is a Framework?

A **software framework** is an abstraction in which common code providing generic functionality can be selectively overridden or specialised by user providing specific functionality.

Low-code is a method of designing & developing applications using an intuitive GUI & embedded functionality that reduce traditional professional code writing requirements. **No-code** is similar to low-code, but for non-technical business users as it allows them to develop software / applications without having to write a single line of code.

2.6 Agile & DevOps

2.6.1 What is Agile?

Agile is a method of software development consisting of:

- **Iterative & Incremental Development:** Software is developed in small, workable increments.
- **Customer-Centric:** Constant feedback from customers to refine requirements.
- **Frequent Delivery:** Rapid releases of smaller, functional product versions.
- **Adaptability:** Agile responds to change quickly

2.6.2 Agile Principles

- **Individuals & Interactions:** over processes & tools.
- **Working Software:** over comprehensive documentation.
- **Customer Collaboration:** over contract negotiation.
- **Responding to Change:** over following a plan.
- **Quote:** "The highest priority is to satisfy the customer through early & continuous delivery of valuable software."

2.6.3 Agile Frameworks

Agile methodologies & frameworks include:

- **Scrum:** Divides work into sprints (2-4 weeks) with regular stand-ups & reviews.
- **Kanban:** Focuses on visualising workflow & limiting Work-In-Progress (WIP).
- **XP (eXtreme Programming):** Emphasises technical excellence & frequent releases.
- **Lean Development:** Focuses on minimising waste & maximising value.

2.6.4 What is DevOps?

DevOps is a culture & set of practices that integrated development (Dev) & operations (Ops). It involves collaboration & automation between developers & IT operations for faster delivery of high-quality software. It also involves continuous integration/continuous delivery (CI/CD) to automate code testing & deployment.

$$\text{DevOps} = \text{Development} + \text{Operations}$$

2.6.5 DevOps Core Practices

DevOps core practices include:

- **CI/CD Pipelines:** Automating the building, testing, & deployment of code.
- **Infrastructure as Code (IaC):** Managing infrastructure through code (e.g., Terraform, Ansible).
- **Monitoring & Logging:** Ensures system reliability through real-time tracking & analysis.
- **Collaboration & Communication:** Cross-functional teams sharing ownership of development & operations tasks.

2.6.6 Key Differences between Agile & DevOps

Agile	**DevOps**
Focus on frequent customer feedback	Focus on collaboration between Dev & Ops teams
Iteration done through iterative cycles	Iteration done through rapid feedback loops
Scope of smaller, incremental changes	Scope of large-scale projects
Uses task management software (e.g. Jira)	Uses automation tools (e.g. Jenkins)
Scrum, XP frameworks	Kanban, DevOps lifecycle frameworks

Table 2: Key Differences between Agile & DevOps

Agile focuses on iterative development & customer feedback, with a short feedback loop. **DevOps** focuses on automating delivery, collaboration, & integration between Dev & Ops teams, integrating the entire process for faster releases.

2.6.7 Why DevOps Complements Agile

Agile improvements development velocity, but DevOps extends the concept to deployment & maintenance. Both are customer-focused, but DevOps ensures rapid & reliable deployment in addition to development. DevOps fills gaps Agile doesn't cover, like operations, infrastructure, & automation. Agile helps development teams iterate & adapt to changing requirements, while DevOps bridges the gap between developers & IT operations.

2.6.8 Benefits of Agile & DevOps

- Faster, more frequent delivery of features.
- Improved communication & collaboration between teams.
- Reduced risk of deployment errors.
- Ability to adapt to customer feedback & market changes rapidly.
- Higher-quality software & reduced time-to-market.

3 Version Control

3.1 What is Version Control?

Version Control is a system that records changes to a file or set of files over time, allowing you to recall or access specific versions at a later date. It is also known as *revision control* or *source control*. It allows you to keep track of changes, by whom, & when they occurred. Some of the popular version control programs include Git, CVS, Subversion, Team Foundation Server, & Mercurial.

It allows us to:

- Backup the source code and be able to rollback to a previous version.
- Keep a record of who did what and when (know who to praise & who to fire).
- Collaborate with the team (know who to praise & who to fire).
- Troubleshoot issues by analysing the change history to figure out what caused the problem.
- Analyse statistics such as who is being the most productive etc.

3.2 What files should be checked in to Version Control?

Any file that influences the software build should be checked into version control. This includes configuration files, file encodings, binary settings, etc. Furthermore, anything that is needed to setup the project from a clean checkout / fork should also be included in the version control, such as source code, documentation, manuals, image files, datasets, etc.

You should not check in any binary files such as JAR files or any other "build" files, any intermediate files from build / compilation such as `.pyc` or `.o` files, any files which contain an absolute path, or personal preference / personal settings files.

3.3 Centralised Version Control – Subversion

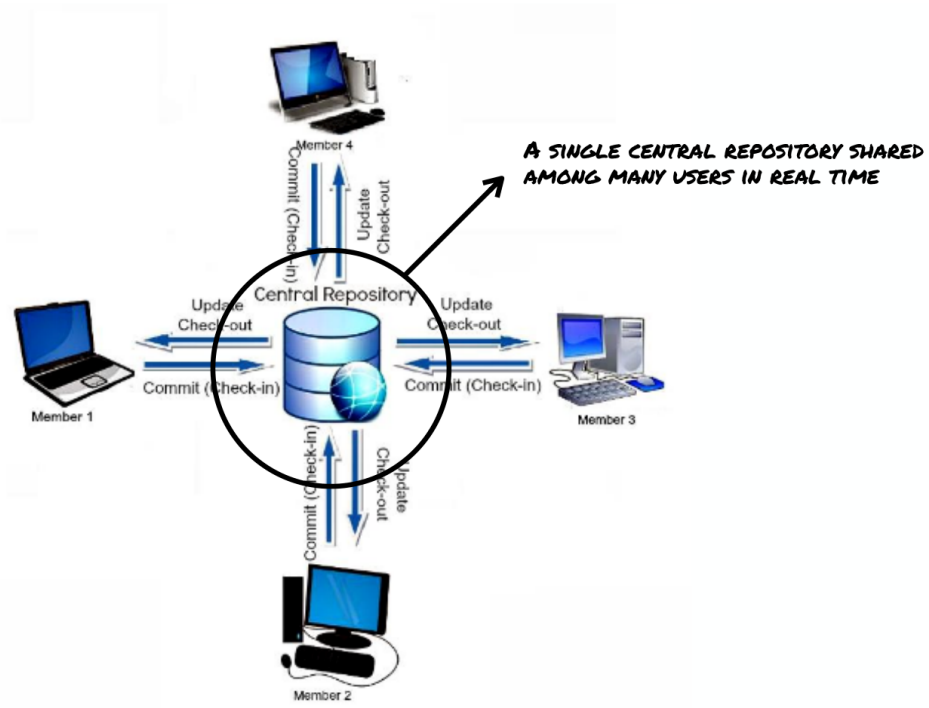

Figure 1: Centralised Version Control System Diagram

Subversion is a centralised version control system in which code is centralised in a repository which can be checked out to get a working copy on your local machine. In general, you don't have the entire repository checked out in Subversion, just a specific branch. Changes are committed back to the central repository, "normally" with useful comments, and a change log is maintained of who did what & when.

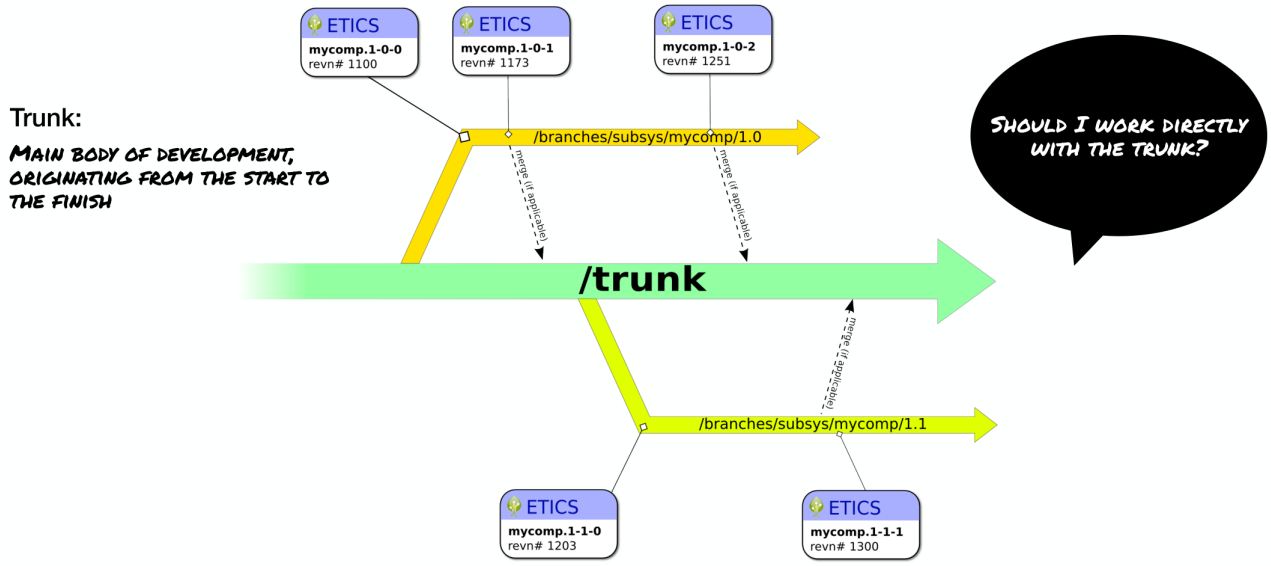

Figure 2: Trunk in Subversion

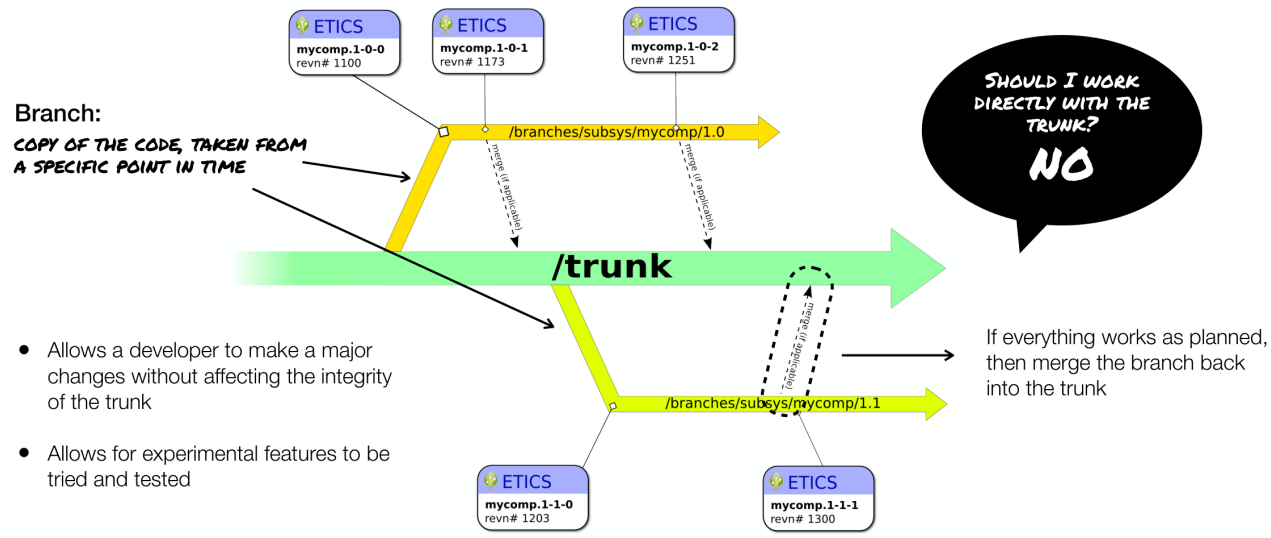

Figure 3: Branching in Subversion

When you check out the project in subversion, you will get the `HEAD` revision. When you invoke the command `svn update`, you are updating your local copy to the `HEAD` version as well. Branches should eventually be merged back into the trunk with `svn commit`. The trunk must build afterwards. The commit is a process of storing changes from your private workplace to the central server. After the commit, your changes are made available to the rest of the team; other developers can retrieve these changes by updating their working copy. Committing is an **atomic operation**: either the whole commit succeeds, or it is entirely rolled back – users never see a half-finished commit.

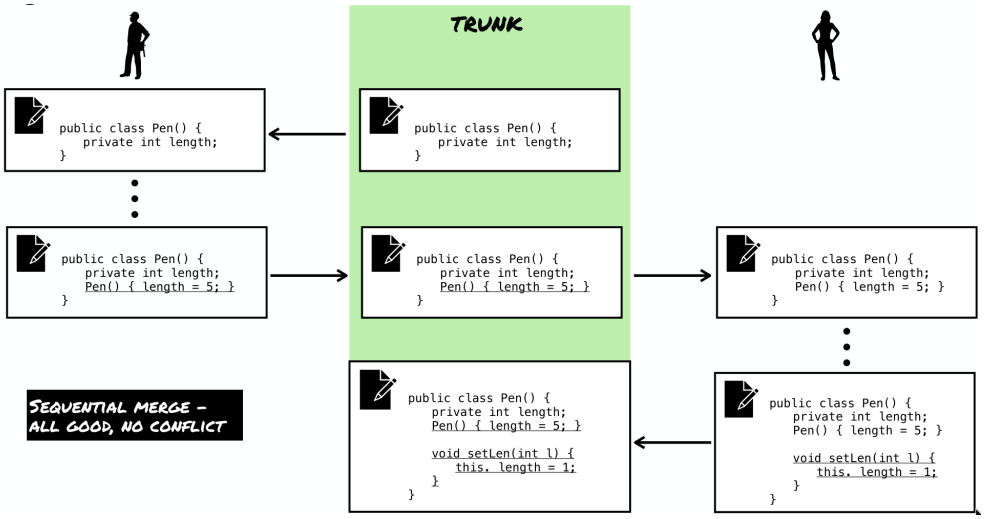

Figure 4: Sequential Merge in Subversion

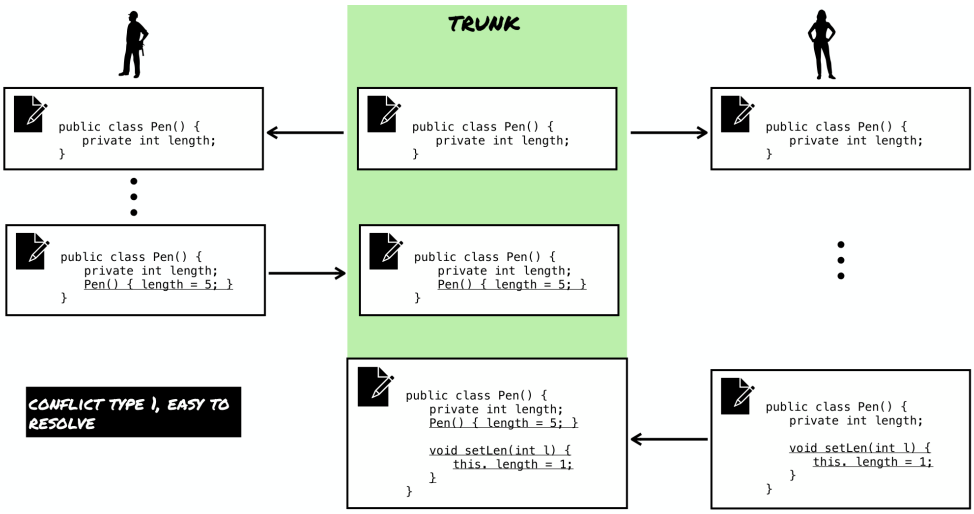

Figure 5: Type 1 Merge Conflict in Subversion

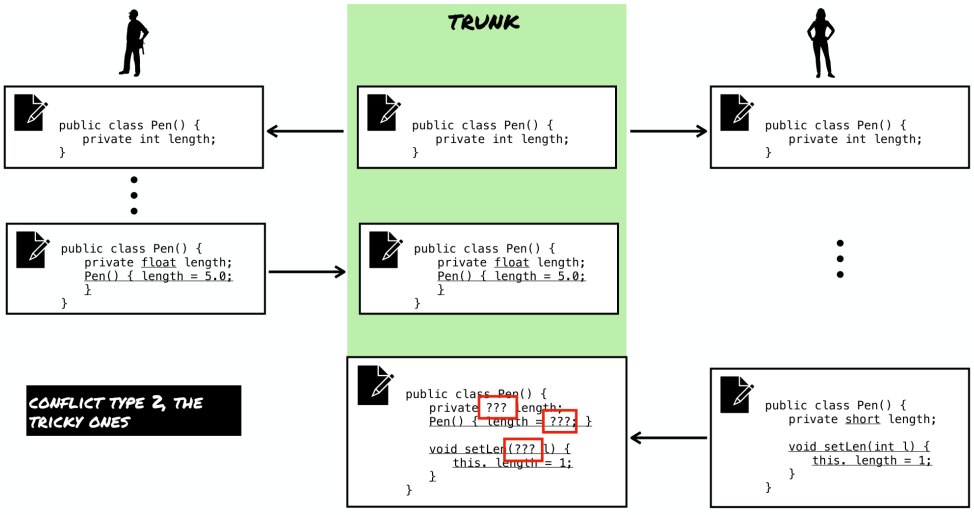

Figure 6: Type 2 Merge Conflict in Subversion

3.4 Distributed Version Control – Git

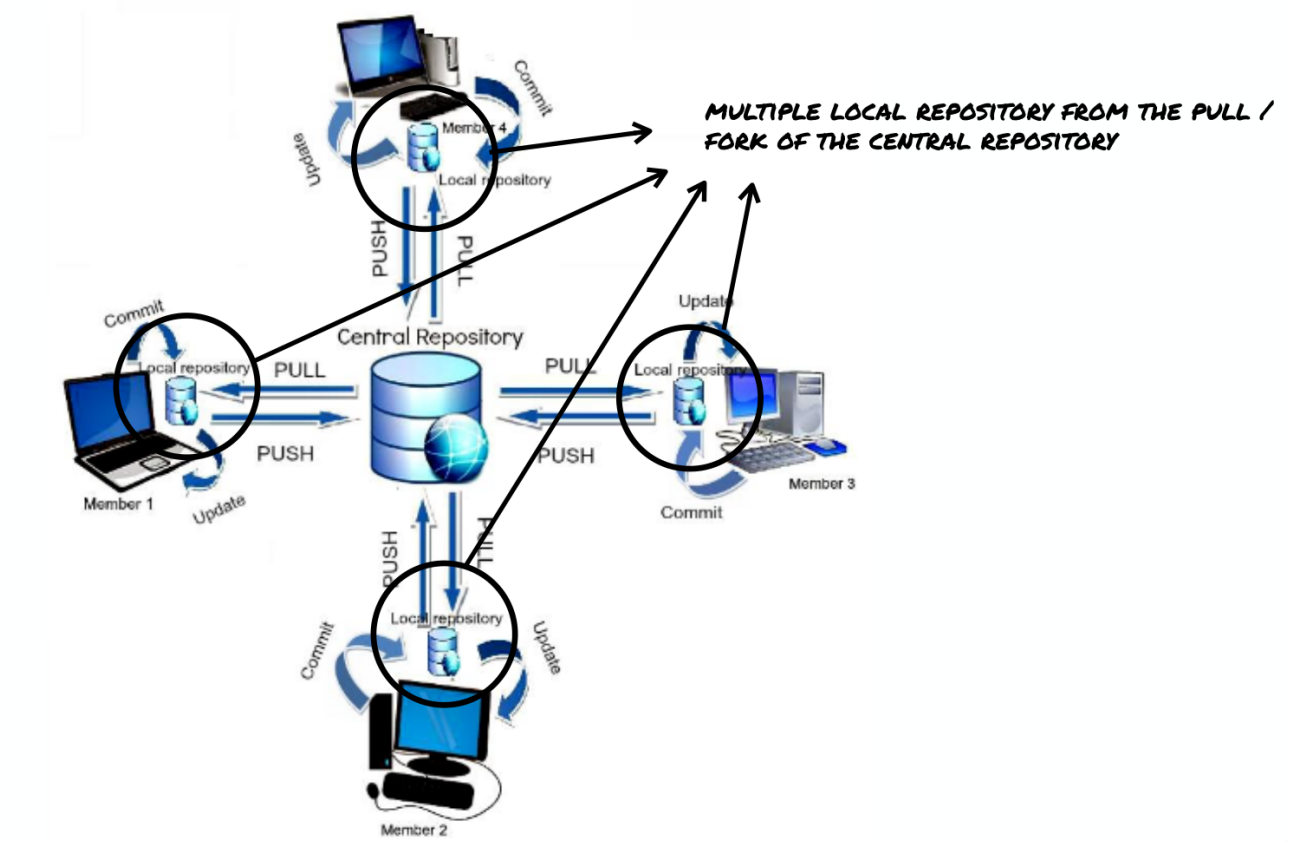

Figure 7: Distributed Version Control System Diagram

Git encourages branching for every feature, regardless of size. After successful completion of the new feature, the branch is merged into the trunk. Each developer gets their own local repository, meaning that developers don't need a network connection to commit changes, inspect previous version, or compare `diffs`. If the production trunk / branch is broken, developers can continue working uninhibited.

3.4.1 GitHub

GitHub is a web-based hosted service for Git repositories. It allows you to host remote Git repositories and has a wealth of community-based services that makes it ideal for open-source projects. It is a publishing tool, a version control system, & a collaboration platform.

3.4.2 Git Commands

- `git clone`: download ("clone") the source code from the remote repository.
- `git fetch`: fetches the latest version from the repository that you've cloned but doesn't synchronise with all commits in the repository.
- `git pull`: pulls the latest version from the repository that you've cloned and synchronises with all commits in the repository. Equivalent to running `git fetch` & `git merge`.
- `git push`: pushes the changes that you have committed to your local branch to the remote repository.

3.4.3 Pull Requests

A **pull request** is when you ask another developer to merge your feature branch into their repository. Everyone can review the code & decide whether or not it should be included in the master branch. The pull request is an invitation to discuss pulling your code into the master branch, i.e. it is a forum for discussing changes.

4 Build Tools

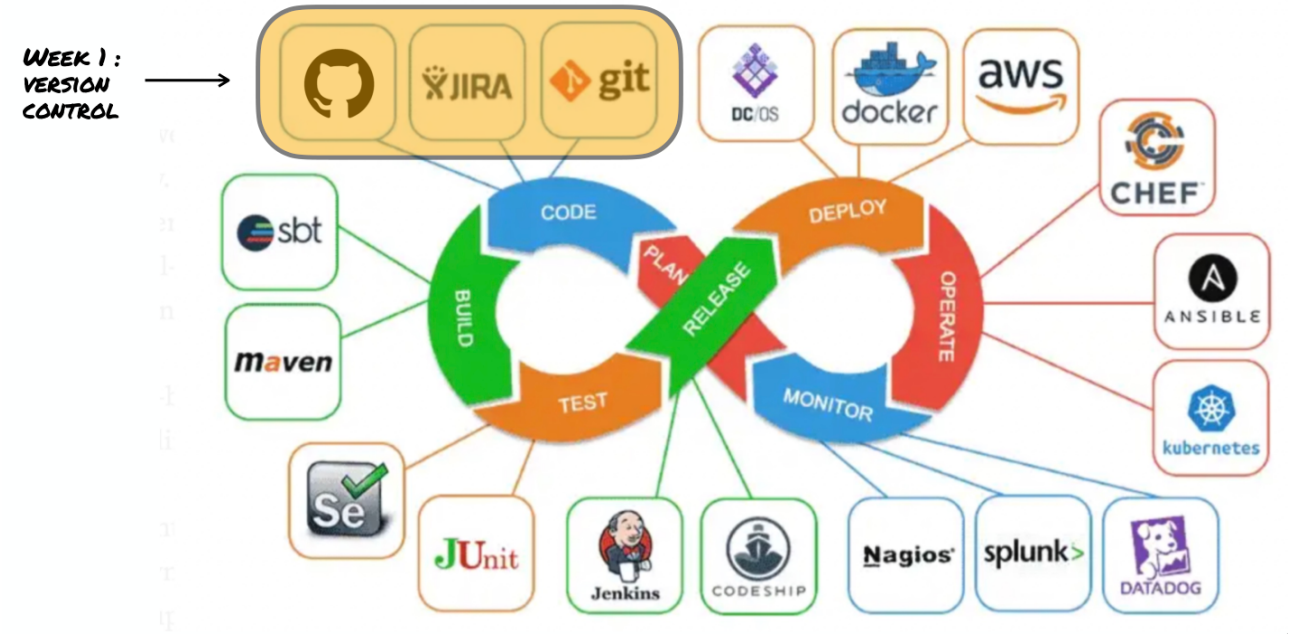

Figure 8: Example of a Continuous Software Development System

The **build** is a process which covers all the steps required to create a deliverable of your software. E.g., in Java:

1. Generating sources.
2. Compiling sources.
3. Compiling test sources.
4. Executing tests.
5. Packaging (`.JAR`, `.WAR`, `EJB`).
6. Running health checks.
7. Generating reports.

Build tools are software utilities that automate the tasks of compiling, linking, & packaging source code into executable application or libraries. Build tools help with:

- **Automation:** build tools help to automate repetitive tasks such as compiling code, running tests, packaging binaries, & even deploying applications.
- **Consistency:** build tools ensure that every build process (e.g., dev, test, prod) is identical, minimising human error.
- **Efficiency:** build tools speed up development by automating builds whenever code is pushed or merged into a repository.

Popular build tools include:

- **Maven** is a software build tool which can manage the project build, reporting, & documentation, primarily used for Java development and supported by most Java IDEs. It features dependency management, project structure standardisation, & automatic builds.

- **Gradle** is a build tool that supports multiple languages including Java, Kotlin, & Groovy. Its features include being highly customisable and being faster than Maven due to its incremental builds & caching. It's preferred for modern Java-based CI/CD pipelines and supports both Android & general Java applications.
- **Node Package Manager (NPM)** is a build tool for JavaScript / Node.js that features dependency management & building for JavaScript applications. It is used to build web-based frontend applications or backend applications in a CI/CD pipeline.

Other popular build tools include Yarn for JavaScript, PyBuilder or tox for Python, MSBuild for C#/.NET, & Rake for Ruby.

4.1 Build Tools in CI/CD Pipelines

Continuous Integration (CI) automatically integrates & tests code on each commit. **Continuous Deployment/Delivery (CD)** automatically deploys tested code to production or staging. Build tools serve many roles in CI/CD pipelines:

- **Integration:** when changes are pushed to the repository, the CI tool (e.g., GitHub Actions) triggers the build tool to compile & package the application.
- **Build Automation:** the build tool automatically handles downloading dependencies, compiling the code, & running tests. It ensures that the same version of the application is built every time.
- **Testing:** many build tools, such as Maven, integrate with testing frameworks (e.g., JUnit, Selenium) to run automated tests after the build.
- **Deployment:** the packaged application can be deployed to a server, containerised (e.g., with Docker), or distributed using CD tools.

Build tools automate the process of building & testing code with each integration to the repository, i.e. **Continuous Integration**. They ensure new changes don't break existing code by running automated tests as part of the build process and exhibit *fail-fast* behaviour: if a test or build fails, the developer is notified immediately.

After a successful build & testing, the build tools package the code, ready for deployment, i.e. **Continuous Deployment**. **Artifact creation** is when the builds create deployable artifacts (e.g., JAR files, WAR files, Docker images). The pipeline can then engage in **automated deployment** by deploying the artifact to a server, cloud, or container.

An example build tool workflow in a CI/CD pipeline with GitHub Actions may look like the following:

1. **Code Push:** a developer pushes new code to the GitHub repository.
2. **CI Tool Trigger:** GitHub Actions detects the change and triggers the pipeline.
3. **Dependency Resolution:** the build tool (e.g., Maven) fetches dependencies from repositories.
4. **Compile & Build:** the build tool compiles & packages the code into executable binaries (e.g., JAR, WAR).
5. **Testing:** run unit & integration tests automatically.
6. **Package & Deploy:** the build tool creates the package, and the CI/CD pipeline deploys it to staging or production.

4.2 Maven

Maven is a software build tool which can manage the project build, reporting, & documentation, primarily used for Java development and supported by most Java IDEs. It is widely used in Spring Boot projects.

1. Compile source code.
2. Copy resources.

3. Compile & run tests.
4. Package projects.
5. Deploy project.
6. Cleanup files.

Developers wanted:

- to make the build process easy.
- a standard way to build projects.
- a clear definition of what the project consists of.
- an easy way to publish project information and a way to share JARs across several projects.

The result is a tool that developers can use to build & manage any Java-based project. It embraces the idea of "convention over configurations".

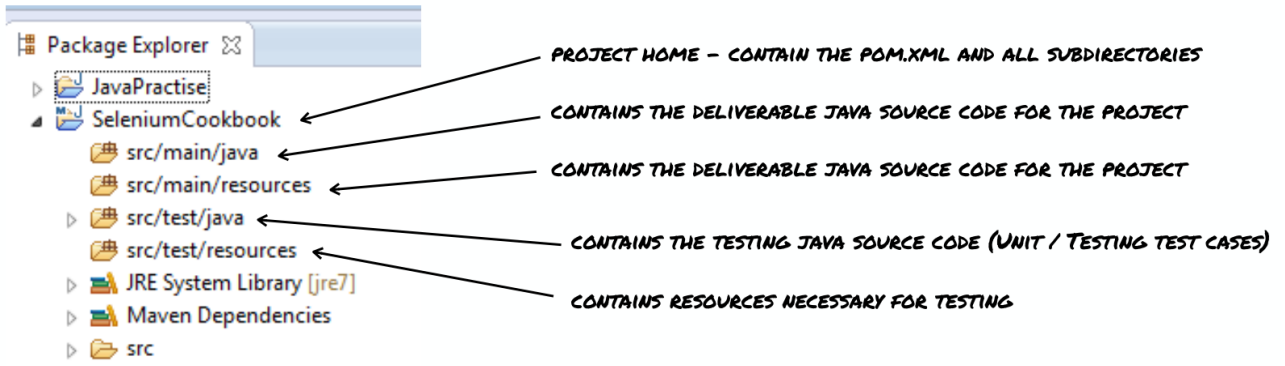

Figure 9: Maven Default Directory Structure

The command `mvn package` compiles all the Java files, runs any tests, and packages the deliverable code & resources into `target/my-app-1.0.jar**`.

The `pom.xml` file is an XML document that contains all the information that Maven requires to automate a build of your software. The `pom.xml` is automatically updated on demand, but can be manually configured as well. The POM provides all the configuration for a single project:

- General configuration covers the project's name, its owner, & its dependencies to other projects.
- One can also configure individual phases of the build process, which are implemented as plugins. E.g., one can configure the `compiler-plugin` to use Java 1.5 for compilation, or specify packaging the project even if some unit tests fail.

Larger projects should be divided into several modules, or sub-projects each with its own POM. All root POM can compile all the modules with a single command. POMs can also inherit configuration from other POMs; all POMs inherit from the super-POM by default. The super-POM provides default configuration, such as default source, default directories, default plugins, etc.

4.2.1 Plugins

Maven build projects based on convention; it expects files to be in a certain place, which is very useful when developing in teams.

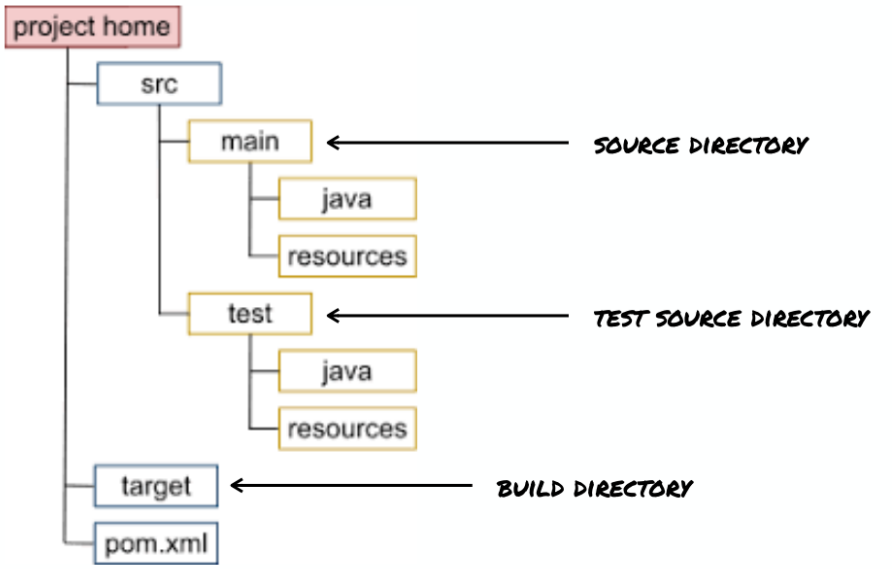

Figure 10: Maven Conventional Directory Structure

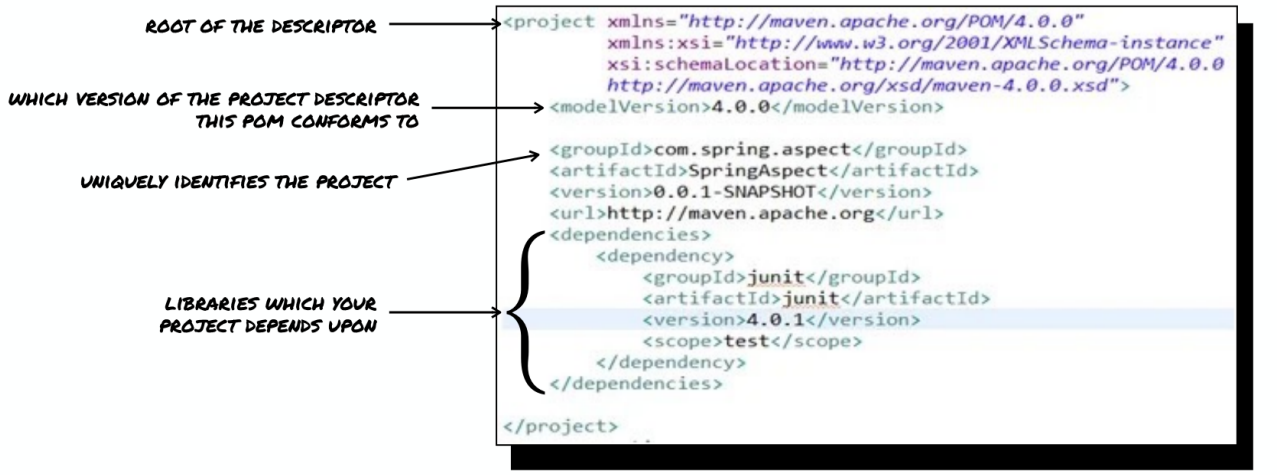

Figure 11: Maven `pom.xml`

A Maven **plugin** is an extension or add-on module that enhances the functionality of Apache Maven. Maven plugins provide additional capabilities & tasks that can be executed during the build process or as part of project lifecycle management. These plugins are typically packages as JAR (Java Archive) files and can be easily added to a Maven project's configuration.

The **core plugins** are plugins corresponding to default core phases (e.g., clean, compile). They may also have multiple goals.

4.2.2 Build Lifecycle

The process for building & distributing a particular project is clearly defined. It comprises a list of named phases that can be used to give order to goal execution. Goals provided by plugins can be associated with different phases of the lifecycle. e.g., by default, the goal `compiler:compile` is associated with the compile phase, while the goal `surefire:test` is associated with the test phase. `mvn test` will cause Maven to run all goals associated with each of the phases up to & including the test phase.

4.2.3 Dependency Management

Dependency management is a central feature in Maven. The dependency mechanism is organised around a co-ordinate system identifying individual artefacts such as software libraries or modules (e.g., JUnit). If your project depends on a JAR file, Maven will automatically retrieve it for you, and store it in the user's local repository. If the JAR file depends on other libraries, Maven will ensure these are also included: these are known as **transitive dependencies**. This wasn't always a part of Maven, so it's a huge benefit.

Dependency management features supported include:

- Management: you can specify library version that transitive dependencies should use.
- Scope: include dependencies appropriate for the current stage of the build, i.e., compile, test, run, etc.
- Exclude dependencies: if project X depends on Project Y, and Project Y depends on Project Z, you can choose to exclude Project Z from your build.

4.2.4 Local & Remote Repository

The remote repository `https://mvnrepository.com/` contains libraries for almost everything: cloud computing, date & time utilities, HTML parsers, mail clients, etc. Once you specify the correct details in the POM file, Maven will automatically fetch the library for you on build.

The local Maven repository can be found at `~/.m2/`. Maven will search the local repository first and then search third-party repositories if it does not find what it's looking for. You can create your own repository and share it within your organisation.

4.3 Spring Boot

Spring Boot is a framework for building standalone Java applications with embedded servers that streamlines Java application development. It provides pre-configured, out-of-the-box functionality to avoid boilerplate code. It reduces configuration & setup, focuses on *convention over configuration*, and is compatible with microservices architecture, REST APIs, & monolithic apps.

4.3.1 Spring vs Spring Boot

Spring is a comprehensive framework for building any Java application requiring more manual configuration & management of dependencies & application context. Spring Boot is an extension of the Spring framework aimed at simplifying development, configuration, & deployment, especially for microservices & cloud-based applications.

Spring	Spring Boot
Requires an external embedded server (e.g, Tomcat, Jetty, etc.)	Comes with an embedded server (Tomcat/Jetty)
Highly flexible but requires more setup effort	Simplifies Spring projects, reducing setup time
Best for complex, large-scale applications	Ideal for microservices & fast prototypes
Requires WAR file and deployment on external server	Packaged as JAR with an embedded server for easy deployment

Table 3: Technical Differences between Spring & Spring Boot

Choose Spring when:

- Your project requires extensive customisations.
- You're building a complex enterprise application where flexibility & modularity are necessary.
- You have a team experiences in managing detailed configurations.

Choose Spring Boot when:

- You're building microservices or need quick iterations in development.
- You want an all-in-one solution with auto-configuration.
- Your focus is on simplicity & speed without worrying about configuration details.

Spring	**Spring Boot**
Framework for building complex enterprise-level Java applications	Simplified framework to quickly build microservices or standalone apps
Slower to set up due to configuration	Faster development with minimal setup
Provides maximum flexibility & customisation	Less flexibility, focuses on ease of use
Suitable for large-scale, complex, highly customised apps	Suitable for small/medium projects, microservices, & rapid development

Table 4: High-Level Differences between Spring & Spring Boot

4.4 GitHub Actions

GitHub Actions automates your workflow by triggering events such as `push`, `pull_request`, & `release`. It easily integrates with other tools like Docker, AWS, Heroku, etc. Key components of GitHub Actions include:

- **Workflows** are defined in YAML in the `.github/workflows/` folder and are triggered by events such as `push`, `pull_request`, etc.
- **Jobs** define units of work that run on a **runner** (e.g., `ubuntu_latest`, `macos-latest`). They can be run sequentially or in parallel.
- Each job consists of a series of **steps**, e.g., checking out the code, building, testing, etc.
- **Runners** include GitHub-hosted runners (e.g., Ubuntu, macOS) but you can also have self-hosted runners to run on your own infrastructure.
- **GitHub Secrets** can be used to securely store sensitive information (e.g., API keys) and are accessible in workflows as `secrets.MY_SECRET_KEY`.

Composite actions can be used to define reusable workflows.