CONFIGURATION MANAGEMENT IN SOFTWARE ENGINEERING

Yulduz Erkiniy
Turin Polytechnic University in Tashkent,
yerkiniy@gmail.com

ABSTRACT

Configuration management plays a crucial role in software engineering, ensuring the effective management of software artifacts throughout the development lifecycle. This article explores the significance of configuration management, its key principles, and the various techniques and tools employed in the field. By maintaining version control, facilitating collaboration, and ensuring traceability, configuration management enables teams to achieve greater software quality, reliability, and maintainability. This abstract provides a concise overview of configuration management in software engineering, highlighting its importance and offering insights into its implementation and benefits.

Keywords*: configuration management, software engineering, version control, collaboration, traceability, software quality, reliability, maintainability.*

1. Introduction

One of the management activities of the software process is configuration management. It is one of the most crucial tasks that must be finished in order to deploy software. Tracking a software's development release by release is the core objective of configuration management. All software components must be identified and managed, along with their evolution. Change management (CM) must also ensure that prior software versions may be rebuilt and regulate changes made to any component, as well as access to and modification rights.

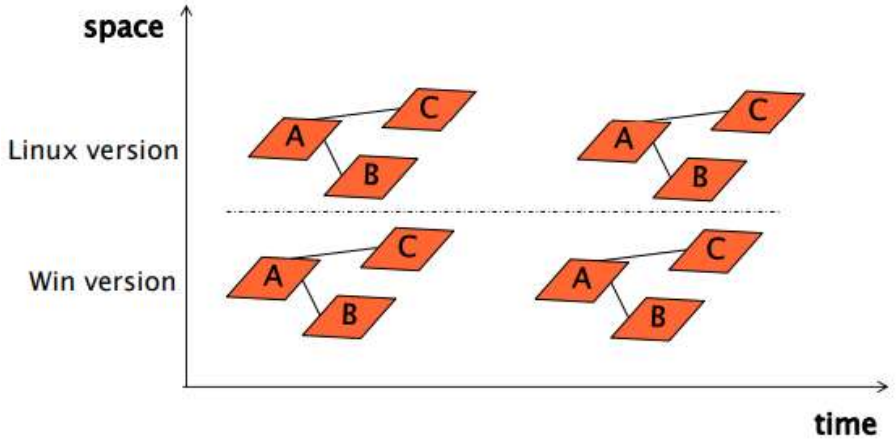

Figure 1 - Growth of a software system in space and time

According to Bersoff et al. (1980), *"No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle"[1].* Software can develop and change in two separate ways:

- **Time:** The components of the software system are susceptible to change throughout time. Different iterations of the same software are released (for instance, to add new features or address bugs);
- **Space**: To adapt software to various contexts and circumstances, several deployments of the same program (for example, a Windows and Linux version of the same software) can be made. Additionally, a software system generally generates hundreds of different documents on its own because it is made up of many different components (such as test logs, code, papers, and so on).

The existence of many software system components and their simultaneous evolution throughout time are dealt with by configuration management. The practice of configuration management is based on four key ideas:

- **Versioning:** What does versioning tell you about a source file's past?
- **Configuration:** Which collection of papers are appropriate for a given need?
- **Change Control:** Who has access to what and how is it controlled?
- **Build:** How is the entire system obtained?

2. Versioning

Software Versioning is the process of monitoring various software releases. Developers and analysts can use it to determine when and what modifications have been made to software code and related documents. The straightforward progressive naming of a software's files and packages is a simple, primitive method of versioning. Tools are able to maintain track of versions for more sophisticated version management, giving users the option to decide whether new versions of a file (with the same name or a different version name) must be created through a commit. With the aid of versioning technologies, it is always feasible to restore an earlier version of the software as a whole or of a single file.[2]

2.1 Configuration Item

An element placed under configuration control is known as a Configuration Item (CI). It is the fundamental building block of the configuration management system and is equivalent to a line of code or any other type of work product (such as a set of specifications or design documents) associated with the program. A Configuration Item may be made up of one or more documents or files, depending on the type of element (for example, a Configuration Item for a C++ class can be made up of two files, the 'hpp' header and the 'cpp' class file). Every CI has a name, a version number, and all previous iterations of the file (its history) are preserved.

Not every document in a project has to be handled as a configuration item; each project gets to decide which documents to treat as CIs. If no document is treated as a CI, the system has no history and no configuration information available for it, which may result in excessive overhead for the Configuration Management activity. For instance, since they are readily available for each version without needing to keep track of all alterations made to them, it is typically a good idea to exclude auto-generated files from the Configuration Items.

2.2 Version

A Version is an instance of a Configuration Item at a specific time (for example, the same Req document on 2015/07/11 and 2015/07/12). Each occurrence of a CI has a special number that serves as the CI's temporal identification.[3]

Versions keep track of the modifications made to Configuration Items over time. The Derivation History is a record of all modifications made to a document or piece of code, including the justification, the performer, the date, and the time of each modification.

A standard prologue style can be used for this purpose, allowing ad-hoc tools (like svn) to handle the derivation history automatically. This can be done, for example, by including it as a header in every new version of a document or source file.

2.3 Configuration

A Configuration is a collection of various Configuration Items in a certain version that are dependent on one another. Configurations typically contain not just code but a variety of project-related documents as well.

Although some dependencies among configuration items (such as those between test cases, requirements, and tested code fragments) may be syntactically declared (e.g., via include instructions in C# or import statements in Java), the bulk of them are not (problem of traceability).

Configurations have their own version as well. Every CI contained in the Configuration is in a certain version. The same version of a CI may be present in multiple configurations (see figure 112, where two classes may be present in three different configurations, each with two versions).

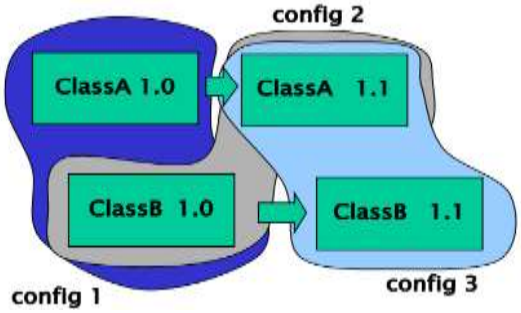

Figure 2 - Configurations and Configuration Items

The two basic methods for identifying Configurations are as follows:

- Keep a configuration ID and a list of the CIs that belong to it (much like in CVS). In this instance, if a CI changes, the configuration also does, even though unmodified CIs never alter their names. The Configuration Items (CI) names and configuration names must be kept separately in this technique, but it is simple to recognize when the CIs change.[4]

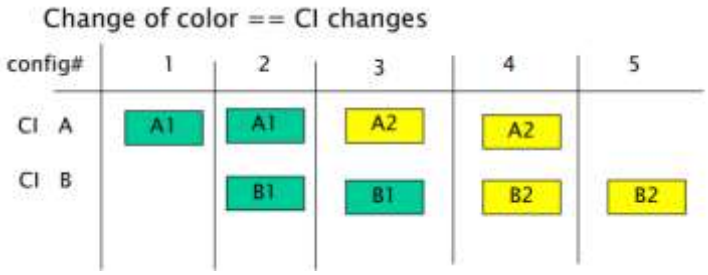

Figure 3 - Different IDs for Configurations and CIs

- Keep a configuration ID and utilize that ID for all the CIs that are a part of it (much like Subversion and GIT do). In this instance, each CI within a configuration that is modified causes all other CIs within it to also change names (even if they are unaffected). This method combines the management of CIs and Configurations, making it simpler to comprehend which Configuration Items are present in a certain Configuration but more challenging to pinpoint the precise moments when a Configuration Item is modified.

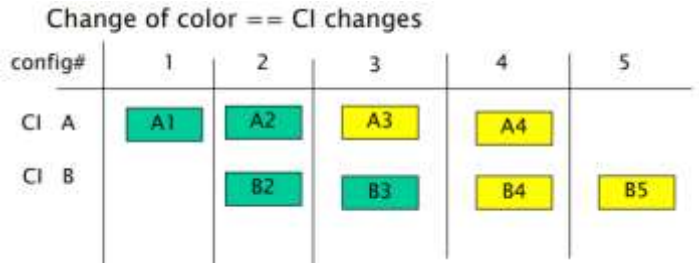

Figure 4 - Same IDs for Configurations and CIs

A baseline is a unique configuration that exists in a consistent state. Every setup is not a baseline. The baseline is frequently delivered and frozen in its current condition; adjustments begin with it, and if something goes wrong, the project is rolled back to the baseline. There are two different kinds of baselines: a product baseline is a stable version of the program supplied to the user or customer, whilst a development baseline is for internal use only and serves as a secure rollback point for development.

The Data Management Model can also be used to describe versioning practices. There are two techniques to save the details of modifications to configurations and CIs:

- Keep **Differences:** Only changes made since the last commit are saved.

- Keep **snapshots:** Each time you commit, a copy of all your CIs is saved. A link to the prior version can be retained if files are not changed from the previous version.[5]

3. Change Control

Usually, teams of developers working on various software components must be shared among them when developing software. Common repositories (shared folders), where all developers can read and write documents and program, are used to share portions of software.

Instead of working directly in the repository, where Configuration Items and Configurations are kept, each developer instead works in his workspace, where copies of the CIs are kept. As a result, there are numerous workspaces (one for each developer) and frequently just one repository for each project. To export the changes produced locally by the developer and import the modifications made remotely by other developers, workspaces must be synchronized with the repository.

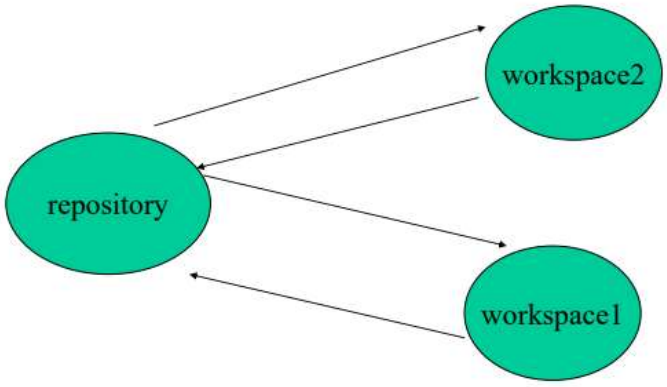

Figure 5 - Repository and Workspaces

The synchronization of changes made by several developers is the main problem that needs to be solved in change control. Each developer updates the repository with their local version after finishing it locally by using the commit command. If another developer is working on the same file, his workspace's local copy may not be instantly updated to reflect the newly committed version. The implementation of change control can take many various forms, from shared files on file servers to CMS solutions with checkin/checkout processes.[6]

3.1 Shared Files

The simplest approach has a file server where developers can upload and download files without using versioning or change control.

Changes made to shared files and folders are not disciplined since there is no oversight of file modifications. Because a developer might duplicate his adjustments on a shared file and overwrite those made by others, this could result in data loss (see figure 117, where John's changes are lost as an example).

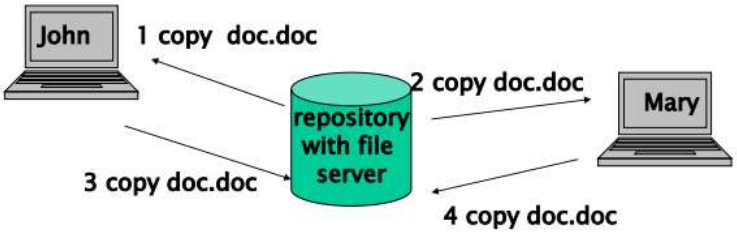

Figure 6 - Repository - shared files

Additionally, a shared folder lacks configuration management tools like automated versioning; all versioning must be done manually using unique file names. The strategy typically only works for extremely small projects.

3.2 Configuration Manager Server (CMS)

The goal of the implementation using Configuration Manager Server is to solve the shared folder issues. On files that have been extracted from the repository, there are two major activities carried out: check-in (commit), which updates the repository with the local modifications made by the CIs, and check-out (pull), which updates the workspace by obtaining the CIs stored in the repository. These two fundamental procedures are employed to control document alterations.

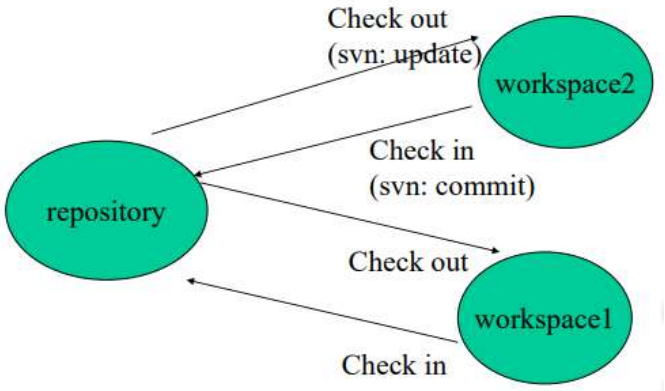

Figure 7 - Check-in and Check-out operations

When opposed to using a shared folder, using a Configuration Manager Server is safer. When using a CMS, in order to change something, the user must check it out first, informing any subsequent users that the file is being changed. Files must be checked back in to the repository after adjustments have been made, and the changes made are compared to those already made by other users. There is no such control over the files in the repository with shared folders; anyone can access and alter any file, and the other users won't even be aware of it.[7]

CMS are useful for more than just Change Control:

- Revert files to a former state as a Configuration Management system;
- Restore the project's complete configuration (or baseline) to a previous state;
- Comparing and tracking alterations over time;

- Keep track of who last edited something, and see if that modification caused any problems.

Check-ins and Check-outs are regulated, and there are a number of options available for them. For example, they can be blocked for a group of users, which would grant them rights within the system. If a checked-out CI is locked, only one person at a time can make changes to it, but several people can still read it. A checked-in CI can choose whether to increase its version at each modification; if not, the previous versions are lost at each change; otherwise, the CI's history is kept for potential rollbacks.

Two primary control tactics can be determined based on these three main options for check-ins and check-outs: Lock Modify Unlock, Copy Modify Merge, etc.[8]

- Modify Lock A serialization of the changes is similar to unlock. Using the check-out process, a developer attempts to obtain a lock over the CI; if no other developer has the lock on the CI, the developer who requested the lock may alter it; otherwise, the developer is unable to checkout the CI and must wait for lock release. Only when the developers holding the lock check in is the lock released. The primary issue with the Lock edit Unlock technique is that no other developer can edit a CI if the locker forgets to unlock it. Additionally, there is no way to work in tandem; only one developer may work on a CI at a time. As a result, this method may be overly inflexible for large development teams and projects.

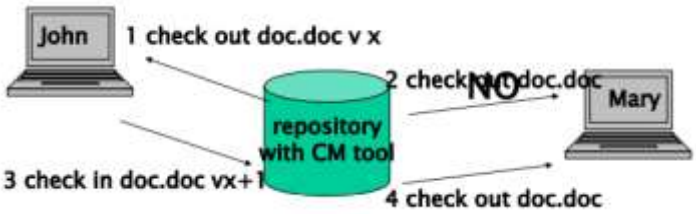

Figure 8 - Lock Modify Unlock

- A less rigid technique is the Copy Modify Merge, which enables many developers to check out the same file and work concurrently on it. The major problem is that conflicts that may result from modifications made by two or more developers to the same file must be addressed and merged (although there are solutions that do automatic merge on changed CIs).

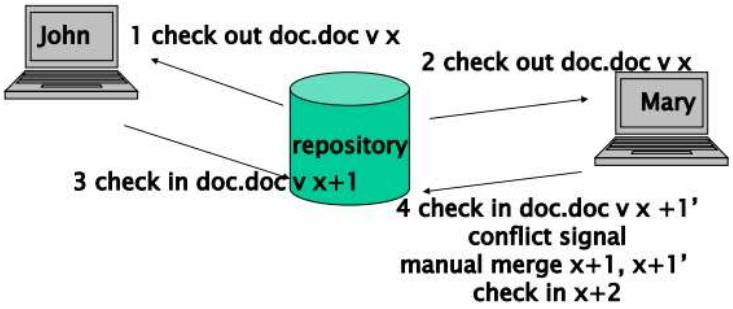

Figure 9 - Copy Modify Merge

Where to store the CMS is another decision that needs to be made. There are three basic approaches available:

- On the same computer as the user's workspace is a local repository (like RCS, for example). A local repository only enables version control; it does not permit file sharing among numerous developers. Therefore, change control methods are not required.

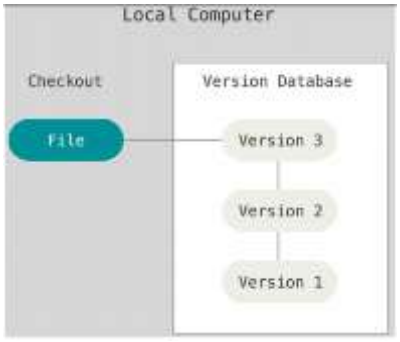

Figure 10 - Local Repository

- The repository is on a single server that maintains the history of files in the centralized way (e.g., Subversion, Perforce, CVS), and numerous users (clients) can access it and check out CIs. A centralized solution has two key drawbacks: a developer cannot function without a connection, and the central server is the system's single point of failure.

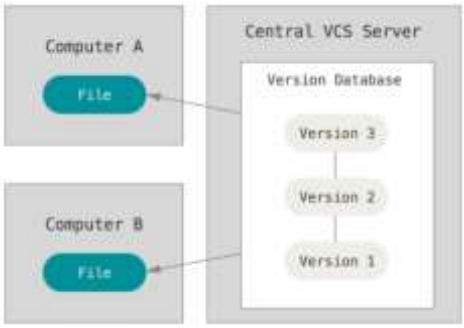

Figure 11 - Centralized Repository

- The repository is duplicated across all servers and clients when using a distributed technique (like GIT). By preserving the history of configurations in the clients as well, this method attempts to solve the issues with the centralized one. Anyhow, it is thought that a central server would simplify the architecture (each alteration would be submitted to the server, and each client would be synchronized with the server).[9]

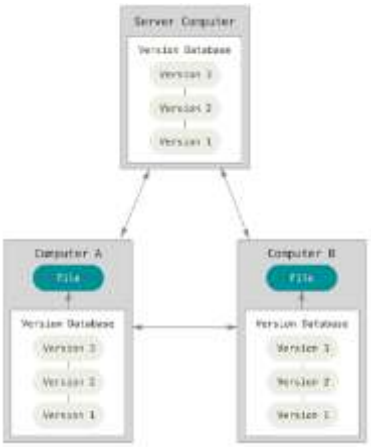

Figure 10 - Distributed Repository

3.3 Branches and Merges

A branch is similar to a development thread; it is a duplicate of an object (or of the entire project) that is under change control and allows for concurrent updates. Versioning also applies to branches, and it is essential to maintain a record of the shared ancestry of related branches.

Every project typically includes two branches, one for developers to utilize to resolve bugs and one for them to add new features.

It is frequently essential to merge separate branches that have undergone parallel operations on code in order to create a new branch that contains all of the parallel modifications (or a subset of them after some have been removed).[10]

3.4 CM Planning

In order to prevent problems with the activities carried out on various Configuration Items and Branches of the project, Configuration Management should be thoroughly planned and documented. If the Change Management operation wasn't documented, the development team can end up doing conflicting or pointless labour.

Key Change Management related decisions and policies for a project are contained in a Configuration Management Plan, which can be developed in accordance with many current templates:

- Which Configuration Management tool is utilized, if any (such as IBM ClearCase, Microsoft BitKeeper, CVS, RCS, Subversion, and Git);
- Which papers ought to be considered Configuration Items, and which shouldn't;
- The project's organization in workspaces and repositories;
- The policy for controlling changes to specified configuration items;
- Who the CM Manager is and what their roles and responsibilities are are particularly important.

One responsible can be assigned to each module and subsystem, and one repository can be designated for any subsystem, with check-in/check-out procedures and

dedicated workspaces for developers, for a product that is a hierarchy of various subsystems (each one an executable and several modules of source files).[11]

4. Build

The process of compiling and connecting software components into a standalone, executable form, starting with the various possible component combinations that can be selected, is known as software building. The building process can be straightforward or quite difficult, depending on the project. Builds for large projects are often automated and guided by build scripts because they can be tedious and error-prone when done manually.

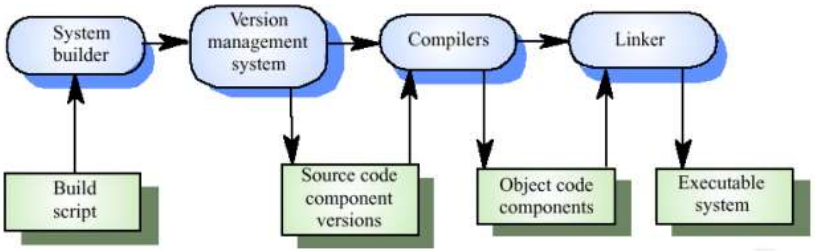

Figure 11 - System building steps

The right versions of the source code are first pulled from the Version Management system using the build script, then sent to the compiler for compilation. The Linker then uses the Object Code components that it has received from the Compiler to build the executable system. The component dependencies (i.e., the connections between various parts of code, such as those described by C includes) must be checked as part of the building process, which is the most crucial step. System Modeling Languages utilize logical system models to solve the problem of users of build tools losing track of which objects are kept in which files, which can lead to mistakes. This is a rather difficult activity, and particularly in projects with numerous components it is easy that linking problems are found and alerted to the developers. In order to create the system correctly, the linker must ensure that all dependencies are consistent and that all files are present.[12]

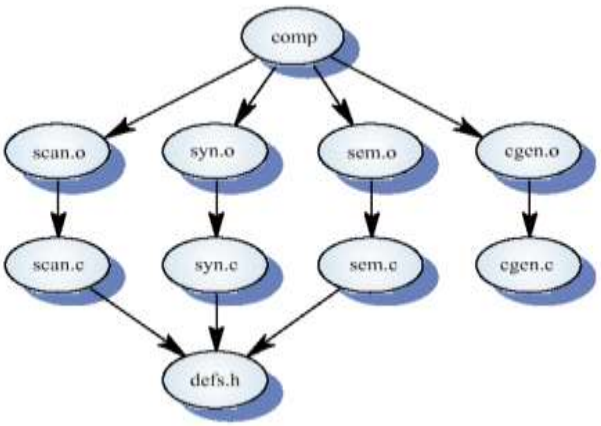

Figure 12 - Component dependencies

The following are the typical issues that arise during the building process:

- Components missing from the build instructions: In really complicated systems, it is simple to overlook one out of hundreds of components. As previously stated, the linker typically handles and signals these issues;
- Wrong component versions specified: The build script contains incorrect versions of specific components. Although a system constructed with the incorrect parts may initially function, faults may occur after delivery;
- Inability to obtain data files: builds should never rely on "standard" data files, which may be absent or different depending on the location;
- Incorrect data file references within components: as naming conventions might vary from place to place, utilizing absolute names in code always leads to issues;
- Incorrect platform was chosen for building: various OSs or hardware configurations should call for particular build settings;
- The wrong version of the supplied compiler (or other build tool) may actually produce different code, and the produced component may behave differently than the expected one.

Automatic build also offers the option of only rebuilding the components that have changed. For example, if only one leaf in the system tree of dependents is modified, only that leaf and the components that depend on it are rebuilt. This tool can help you cut down on building times, which may be important for big projects.

The automated building tools Make, Ant, Apache Maven, and Gradle are a few examples.

5. Configuration Management with Git

Git is a system that was developed in 2005 for the Linux kernel’s development and is mostly used for source code management in the software development industry. According to a survey done in the Eclipse community, Git has firmly established itself as the most popular code management system among developers in recent years, surpassing Subversion in developers’ choices as early as 2014.

Git is a distributed configuration management system, and each computer’s Git directories are taken into account as complete repositories. On snapshots, Git’s data management model is based. Git is mostly based on local operations: all necessary data is stored on the present computer; no data from other machines in the network is required. Integrity characteristics offered by Git include computed checksums at each commit, prior to anything being stored, and the absence of unrecorded modifications to any directory or file.

5.1 Git States

Git-managed documents may be in one of three conditions:

- Committed: Data has been securely stored in the local database (with checksums);
- Modified: the file has undergone local changes, but the local database has not yet been updated;
- Staged files have undergone local changes and have been noted in their current state for inclusion in the upcoming commit.

5.2 Git Project Sections

A Git project's files can be arranged in one of three categories:

- Git Directory: It houses the project's entire object database and any associated metadata. When a repository is copied from another computer or from an online hosting service (like GitHub), it is what is copied. Each commit permanently stores updated and staged files in the Git Directory.
- Working Directory, also known as Working Tree, is a version of the project that has been checked out. It includes all of the files that are extracted from the compressed Git Folder and saved locally for the developer to use or modify.
- Index is another name for the staging section, which holds details about all the files that will be included in the following commit. Git's data management model is built on snapshots, therefore when files are staged, snapshots of those files are also added to the staging area.

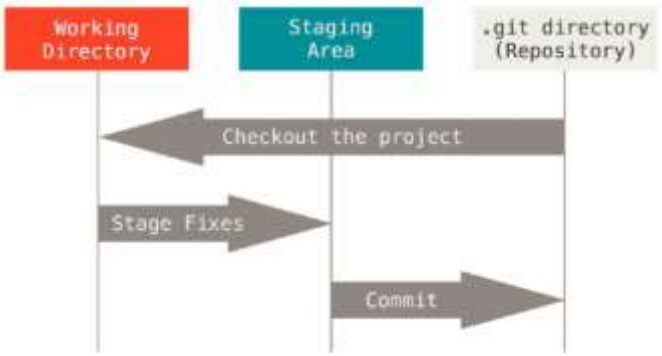

Figure 13 - Typical Git Workflow

5.3 File states

Git requires one of four states for files it manages:

- Files that have been added to the working directory but weren't in the most recent snapshot (from which the project was cloned) or the staging area are classified as untracked. They may also be original files that have been deleted from Git using the rm command.
- Unmodified: When a repository is cloned, all of its files are exactly as they were during the last commit.
- Files marked as modified have undergone changes since the last commit.

- Staged files are those that have been chosen for the upcoming commit.

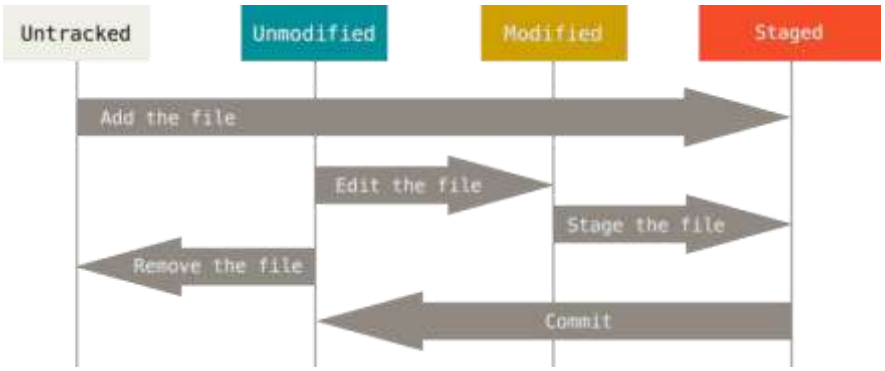

Figure 14 - Lifecycle of files

Conclusion

In conclusion, configuration management is a crucial aspect of software engineering that facilitates effective management and control of software artifacts throughout the development process. By implementing configuration management practices, such as version control, collaboration, and traceability, software teams can achieve improved software quality, reliability, and maintainability. The use of appropriate tools and techniques, combined with adherence to key principles, enables seamless coordination and synchronization among team members, leading to optimized development workflows and enhanced productivity. Configuration management serves as a foundation for successful software development, enabling teams to efficiently handle changes, mitigate risks, and deliver high-quality software products. Embracing configuration management practices can significantly contribute to the success of software engineering projects, ensuring the delivery of robust and reliable software solutions to meet the evolving needs of users and stakeholders.

References:

1. "Software Configuration Management Handbook" by Alexis Leon and Mathews Leon, 3rd edition, 2015.
2. "Configuration Management in Agile Development: A Systematic Literature Review" by Timo Männistö and Jouni Markkula, 2016.
3. "Software Configuration Management Patterns: Effective Teamwork, Practical Integration" by Stephen P. Berczuk and Brad Appleton, 2002.
4. "Configuration Management in Large-Scale Agile Development: A Case Study" by S-P. Krishna and D. Janzen, 2012.
5. "Configuration Management Principles and Practice" by Anne Mette Jonassen Hass, 2011.
6. "Configuration Management in Continuous Delivery: A Systematic Literature

Review" by Tero Paivarinta and Jouni Markkula, 2017.
7. "Software Configuration Management Strategies and IBM Rational ClearCase: A Practical Introduction" by Ahmed Alhagry, 2010.
8. "Configuration Management in Distributed Agile Development: A Systematic Literature Review" by Timo Männistö and Jouni Markkula, 2018.
9. "Software Configuration Management Implementation Roadmap" by Mario E. Moreira, 2004.
10. "Configuration Management Challenges in Large-Scale Agile Development: A Case Study" by S-P. Krishna and D. Janzen, 2011.
11. "Software Configuration Management: A Clear Case for ClearCase" by Anne Mette Jonassen Hass, 2003.
12. "Configuration Management Best Practices: Practical Methods that Work in the Real World" by Robert Aiello and Leslie Sachs, 2010.