Assessment Rubric
Assignment (which architecture): HW05

Which team are you reviewing: T33

What is your team: T23

Team: T33

Topic	Strengths and suggestions for the architects
Diagrams	Excellent
Clarity	Very good. Missing explanation of architecture arrows and the component Media Storage Bucket. Good use of the yellow notes, providing more context to the connections
Consistency	Excellent
Completeness	Very good. The collaboration part could have been better explained by adding a new diagram.
Sufficient Level of detail	Excellent
Text Description	Excellent
Clarity	Excellent
Consistency	Excellent
Sufficient?	Very good
Correctness	Excellent
Anything missed?	Excellent. They didn’t miss anything
Will it work?	Excellent. Yes, we think that the system will work very well.
Presentation	Excellent
Summary	Excellent

Faculdade de Engenharia da Universidade do Porto

Software Systems Architecture

Homework #06 - Understand Someone Else's Architecture

Team T23:
Anete Pereira (up202008856)
Bárbara Carvalho (up202004695)
David Fang (up202004179)
Milena Gouveia (up202008862)
Pedro Correia (up202006199)

Introduction

Git is a distributed version control system that is designed for maintaining digital projects, mostly code, with multiple collaborators using a peer-to-peer network of repositories. It supports distributed workflows, allowing a project to either eventually converge or temporarily diverge.

This paper summarizes the chapter about Git from “The Architecture of Open-Source Systems”, identifying and complementing the overall architecture of this well-known and open-source system.

Architecture

At its core, Git uses a three-tree architecture that forms the foundation for its functionality. Understanding this architecture is pivotal for comprehending how Git manages changes and facilitates collaboration within software development projects. The three layers are:

Workspace (working directory): Represents the state of the files on the local machine. It's where developers make modifications, add new files, or delete existing ones.

Staging Area: Acts as an intermediary between the working directory and the repository. It serves as an area for developers to review and organize their modifications before committing them to the repository. The "git add" command is used to move modifications from the working directory to the staging area.

Local Repository (Commit History): Stores the committed changes along with their associated metadata. It maintains the complete history of changes made to the project, including commit messages, timestamps, and parent commits. It forms the backbone of version control, enabling developers to track the evolution of the project over time. Changes are committed here using the “git commit“ command.

Git's three-layer architecture primarily focuses on local version control but it can easily be integrated with a remote repository to enable collaboration and centralized management. By adding a remote repository, developers can push their local commits to a shared repository hosted on a remote server, facilitating teamwork, backup, and centralized control over the project. Users can push their local commits to the remote repository using the "git push" command and fetch changes made by others using the "git pull" command, as seen in the image below.

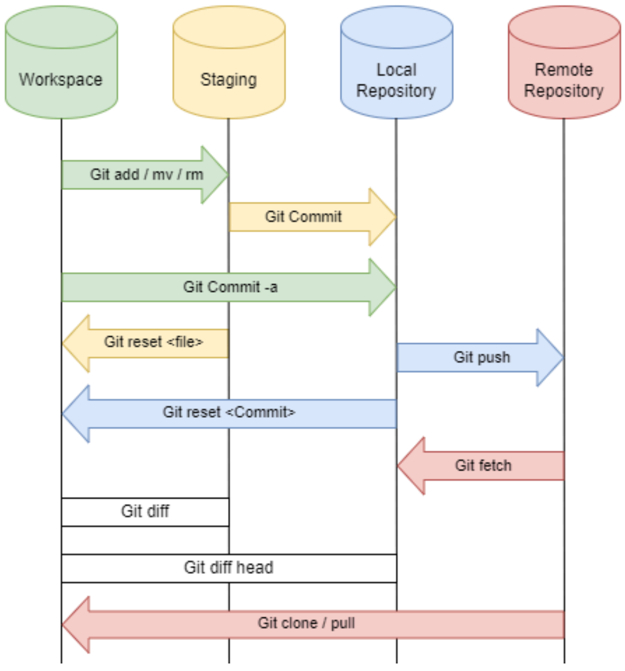

Figure 1: Overview context

Git also shares similarities with the Repository pattern. Git repositories serve as centralized storage for versioned files, akin to repositories in the Repository Pattern, allowing efficient management and access to files. Through high-level commands Git abstracts away the complexity of versioning, similar to how the Repository Pattern abstracts data access. It also embodies the principle of isolating data access by segregating version control logic from application code. Finally, Git encapsulates version control operations within a standardized set of commands, akin to the encapsulation of data access operations within the repository interface in the Repository Pattern.

Objects & Data Model

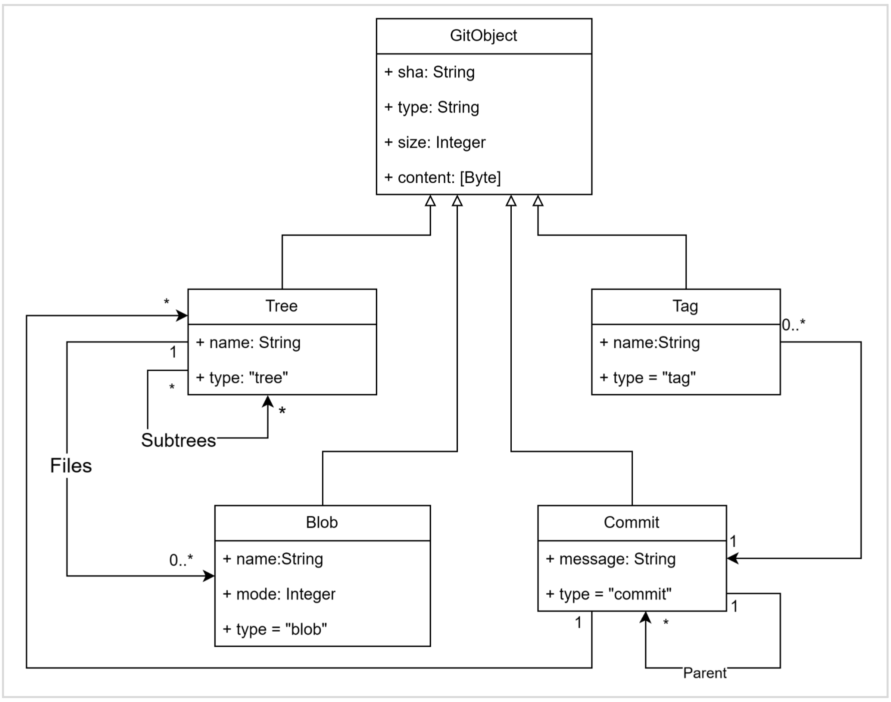

Figure 2: Git's Data Model

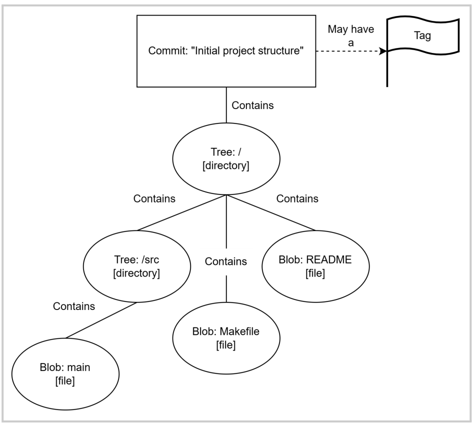

Figure 3: Git's direct acyclic graph

As seen in Figure 2, Git has 4 basic primitive objects: **Blob** represents a file in the repository; **Tree** acts as a directory, organizing the structure of a project by referencing blobs and other trees; **Commit** captures the state of the project at a given moment, referencing the project's top-level tree for that snapshot and connects to previous commits to maintain a continuous history; **Tag** as a custom name to attribute to a commit at the point in the repository history that the tag represents. All of the objects are referenced by a SHA, a 40 bit object identifier, that serves to identify if two objects are identical or not.

Git stores content as a directed acyclic graph, which involves the objects forming an hierarchy that mirrors the content's file system tree as a snapshot of the commit, as seen in Figure 2. To make sure these objects don't occupy a lot of space, Git stores them in a compressed format, using an index file which points to offsets to locate specific objects in the corresponding packed file.

The DAGs also enable full branching capabilities. Each file's history is traced back through its directory structure to the root directory, linked to a commit node, as seen in Figure 3. These commit nodes may have multiple parents, providing Git with two key properties:

• Content Identity: Nodes with the same reference identity (the SHA) contain identical content, enabling Git to efficiently identify unchanged content without needing to diff.

• Merge Efficiency: Merging branches involves merging content nodes in a DAG, allowing Git to determine common ancestors more efficiently compared to other version control systems like RCS.

With all of this in mind, it’s possible to say Git design aligns with principles of Data-Oriented Design (DOD) regarding the goals of optimizing data structures and ensuring reliability, although indirectly. It employs efficient data structures (DAGs) and content-addressable storage, optimizing storage and retrieval operations. It also ensures data integrity through immutability, with commits being immutable once made, and uses cryptographic hashes for integrity verification.

Quality Attributes

Within the provided documentation for Git, various aspects of quality attributes are evidenced, which are crucial for ensuring that Git meets the diverse needs of developers and organizations, facilitating efficient collaboration and reliable code management. Each quality attribute plays a distinct role in shaping Git's capabilities and usability. Let's delve into each one to understand its significance within the Git ecosystem:

Reliability:The core features of Git rely on constant, accurate, and dependable operations. This includes the version control mechanism, commit tracking, branching, merging, and data integrity checks.

Usability: The user-friendliness of Git tools, commands, and interfaces ensures that developers can easily learn and use Git effectively, leading to increased productivity and reduced errors.

Performance: Refers to the speed and efficiency of Git operations, such as committing changes, branching, merging, and pulling/pushing code. Optimizing performance ensures that developers can work efficiently without delays.

Modularity: Git's design fosters flexibility and extensibility. Its architecture consists of modular components, providing integration of third-party tools. This modularity enhances maintainability and allows for easy updates and enhancements.

Security: Focuses on protecting Git repositories, commits, and branches from unauthorized access, data breaches, and malicious activities. Security measures include authentication, access control, encryption, and secure communication protocols.

Conclusion

In conclusion, delving into the architecture of Git provides valuable insights not only into the intricacies of this widely-used version control system but also into the broader concept of understanding someone else's architecture. By dissecting Git's three-tree architecture, data model, and emphasis on quality attributes, our team gained a deeper insight for the rationale behind its design.

In the case of Git, its architecture reflects a careful balance between simplicity and robustness, enabling seamless collaboration and efficient version control. By studying Git's architecture, we not only gain insights into its functionality but also learn valuable lessons in software design, data management, and system reliability.

Moreover, the exercise of understanding Git's architecture underscores the importance of effective documentation, clear communication, and collaborative problem-solving in deciphering complex systems. It highlights the need for interdisciplinary collaboration and knowledge sharing to navigate intricate architectures and leverage them effectively in real-world scenarios.