Assessment Rubric
Assignment (which architecture): HW06

Which team are you reviewing: T34

What is your team: T24

Topic	Strengths and suggestions for the architects
Diagrams	Very Good
Clarity	Good. The logic diagram's lines are not as clear as they could be, although the labels are appreciated. As for the flowcharts, the lack of a legend for symbols used in the typical scenarios might cause some confusion for those not familiar with such notation.
Consistency	Excellent. There is a high level of consistency in the use of shapes and lines within each diagram. The flowcharts maintain a consistent use of symbols for decisions, processes, and data flow, which aids understanding.
Completeness	Excellent. The different diagrams clearly expose the foundation for the system architecture covering the modules, the action flows and the physical system design. Kudos for the nicely detailed physical system diagram.
Sufficient Level of detail	Very Good. The diagrams generally provide an excellent level of detail for the context it represents. However, the logic diagram should be more detailed regarding the interactions between each module.
Text Description	Excellent
Clarity	Excellent. The textual descriptions of the modules are well-articulated, offering clear insights into the purpose and functionality of each component within the system.
Consistency	Excellent. There is a strong alignment between the text descriptions and the diagrams. The descriptions accurately reflect the functions and interactions displayed in the diagrams.
Sufficient?	Excellent. The text is sufficiently informative, detailing the role of each module and component.
Correctness	Excellent
Anything missed?	Very Good. It's all great but there could be a mention of security measures beyond authentication, such as data encryption, which is critical for a system handling sensitive user data such as credit card information.
Will it work?	Excellent. The system design covers the primary functionalities expected of a library system. The document is well-structured and would likely serve as a solid foundation for implementing the system. It will work.
Presentation	N/A
Summary	Excellent. Overall, the design exhibits a comprehensive approach to creating a modular library system with an emphasis on user experience and system integration. Further improvements could involve areas concerning error handling, security beyond authentication, and system scaling.

Explanation:

Diagrams

- Clarity: Are all the figures clear? Is it easy to understand what they mean? Are they laid out in an intuitive way? Is it clear what colors mean? If necessary, is there a legend?
- Consistency: Are the shapes and lines used consistently? Is there anything (a shape or line) that means two different things? Are colors consistent, if used?
- Complete: Is anything missing that would make the diagrams more clear?
- Sufficient level of detail: Do the diagrams give enough detail? For example, are there some components that should be broken down more?

Text Description

- Clarity: How clear is the description?
- Consistency: How consistent is the description with the diagrams?
- Sufficient: Does the text give enough information?

Correctness:

- Did the architects miss anything, such as did they forget a component?
- Will anything not work?

Summary:

- How confident are you that you could implement the design as it is described?

Faculdade de Engenharia da Universidade do Porto

Understand Someone Else's Architecture

Software Systems Architecture

Team T24:

Henrique Oliveira Silva up202007242@up.pt

João Pedro Matos Araújo up202007855@up.pt

Rui Filipe Cunha Pires up202008252@up.pt

Pedro Miguel Nunes up202004714@up.pt

Diogo Miguel Ferreira Costa up202007770@up.pt

1. Architecture Summary

For this assignment we've decided to look into Git's architecture. At its core, Git is a distributed version control system that enables versioned maintenance of any digital body of work, although mostly popular among software developers, through a peer-to-peer network of repositories. Understanding its architecture will help us gain a better understanding of how these capabilities are achieved and better explain its adoption in the software industry.

Content Storage

For content storage, there are two main concepts to acknowledge: an object oriented design and the usage of directed acyclic graphs (DAG) for content representation. This representation means that objects form a hierarchy similar to the filesystem tree and allows the reuse of unchanged objects whenever possible. Git leverages four primary types of objects: **blobs** (content of a file stored in the repository), **trees** (directory structures, its elements can be trees or blobs), **commits** (snapshot - point to a tree representing the top-level directory for that commit and relevant commit info), and **tags** (annotated pointers to a commit). These objects, stored in a content-addressable manner, facilitate efficient storage, retrieval, and manipulation of project data.

Cryptographic Integrity

All objects are referenced by a SHA-1 identifier which guarantees some properties across every object: identical objects have the same SHA; different objects have different SHAs; if the object's data has been corrupted the recalculation of the SHA will reflect that corruption. Besides some benefits mentioned late, the third property helps assure some safety which Git aimed to provide.

History

Again, Git makes use of DAGs for managing history instead of a linear history. Commits store metadata on its ancestors, ranging from 0 to any number (theoretically) of parent commits. This design choice explains how Git is able to offer branching that records merge history, which is a key aspect of why Git is suitable for large software projects with many collaborators. Additionally, the structure described so far affords Git two properties mentioned in the chapter that contribute to its performance goals: efficient content diffing, since content nodes (even in different commits) with the same reference identity are guaranteed to have the same content; efficient determination of common ancestors (at least compared to previous VCSs), as merging branches is as simple as merging the contents of the respective nodes.

Distribution

Some existing VCSs opted for either local-only (wouldn't fulfill Git's functional requirements) or central-server content distribution models. Git opts for a distributed

model, where commits can be made locally (allowing offline work) and pushed to any accessible repositories later. Compared to a central-server model, Git's workflow has some added complexity, requiring collaborators to explicitly show intent of sharing local changes with remote repositories. In turn, this offers teams greater flexibility regarding their workflow and pushing capabilities, for example, establishing certain requirements to push for a remote repository or branch with no effect on local repositories.

Repository, Staging and Working Areas

When initializing a Git repository in a local directory, a .git subdirectory is created, which will be the local repository. The working directory is typically the parent directory of .git, containing the working set of files. The repository includes many different files and directories serving multiple purposes, including a staging area for the working directory (.git/index). This is used as an intermediary, to stage specific changes within the working directory to be committed together.

Storage

As some of these design decisions are not as efficient regarding storage as some alternatives that don't provide the same functionality, there is a need for Git to tackle this issue. This is done by packing objects using compression techniques. Essentially, Git uses index files with pointers to offsets identifying objects inside a packed file.

Merge Histories

It has been mentioned that Git supports branching, and it's also crucial to look into how different branches (and histories) can be merged together. When merging branches, Git records a merge commit that consolidates the changes from divergent branches, serving as a pivotal point in the project's history. This merge commit, along with its parent commits, establishes a clear lineage facilitating easy navigation through the project's evolution. Beyond just version control, Git's merge histories foster transparency, accountability, and collaboration within development teams, aiding in issue diagnosis, change reverting, and root cause analysis, while also supporting advanced branching strategies like feature branching and release management.

Future Work and Addressing Current Issues

Git, at its core, is based on a toolkit design deriving from the Unix world, which is not particularly suitable for integration with Integrated Development Environments (IDEs) although integrations exist for popular IDEs. Some solutions are being worked on by Google and Eclipse Foundation engineers.

2. Architecture Patterns

Wrapper Facade

- Git's architecture employs a wrapper facade to simplify complex interfaces. The .git directory acts as a unified interface, encapsulating lower-level components such as configuration settings and metadata.
- By providing a cohesive API, Git shields users from the intricacies of its internal data structures, making it easier to interact with repositories.

Active Object

- Git's asynchronous behavior aligns with the Active Object pattern. Git queues requests (commits, merges, etc.) and processes them asynchronously.
- Active Object ensures responsiveness and non-blocking behavior, crucial for handling concurrent operations in Git.

Monitor Object

- Git uses monitor objects to synchronize access to shared resources (e.g., repository data). Locks ensure thread-safe interactions.
- Monitor Object pattern prevents race conditions and coordinates concurrent access, maintaining data consistency.

Thread-Specific Storage

- Git employs thread-specific storage to maintain per-thread state. Each thread has its own context for efficient resource management.
- This pattern ensures that Git's operations (e.g., commits, pulls) don't interfere with each other across threads.

3. Quality Attributes

Distributed Workflows

Git supports distributed workflows, allowing a body of work to either converge or temporarily diverge. The system enables full branching capability using directed acyclic graphs (DAGs) for content storage, reference pointers for heads, and an object model representation.

Content Integrity

Git offers safeguards against content corruption, ensuring reliability.

Flexibility

Git provides flexibility for collaborators to work offline and commit incrementally, with the ability to publish managed work to multiple repositories, potentially with different branches or granularity of changes visible.

4. Relevant Diagrams

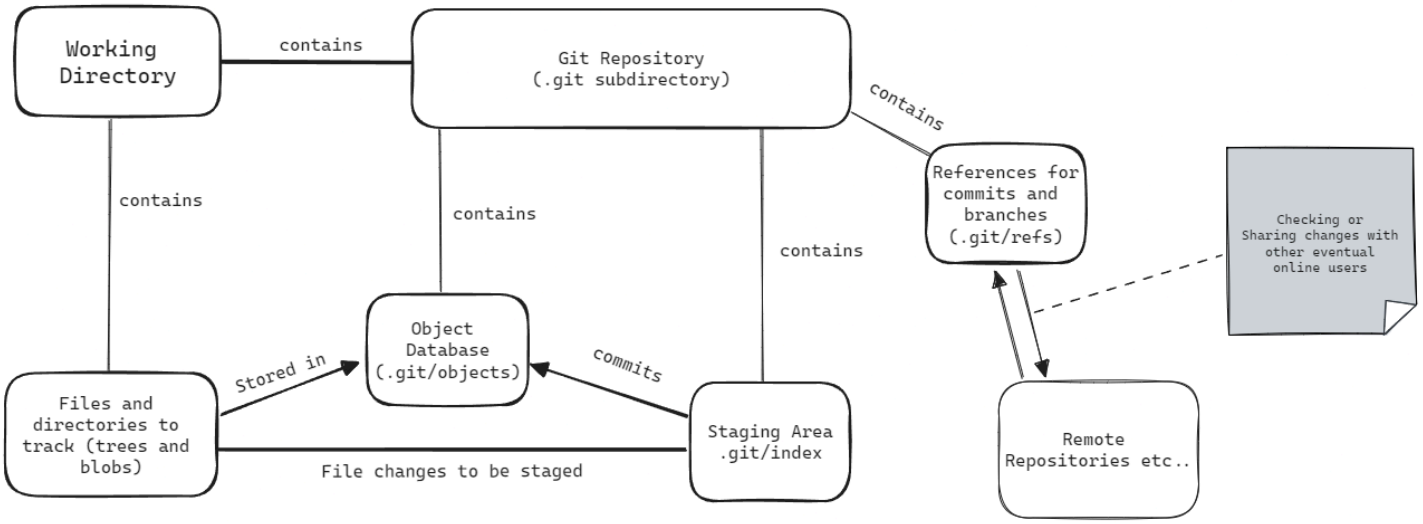

Fig.1 Git Repository High Level Architecture

The above diagram depicts the high-level architecture of Git, a distributed version control system (DVCS). Here's a breakdown of the key components:

- **Working Directory:** This folder contains the files you're currently working on.
- **.git Directory:** This hidden folder stores Git's metadata for the repository, including the following:
    - **Objects Database:** This stores different versions of your project's files and snapshots.
        - **Blobs:** The raw content chunks of your files.
        - **Trees:** Organize blobs into a hierarchical structure, representing your project's directory structure at a specific point.
        - **Commits:** Snapshots capturing the project state at a particular time, referencing a tree and having an optional message describing the

changes made. Some commits may also reference parent commits, creating a history.

- **Refs (References):** Symbolic names like "master" or "branch_name" that point to specific commits in the Objects Database.

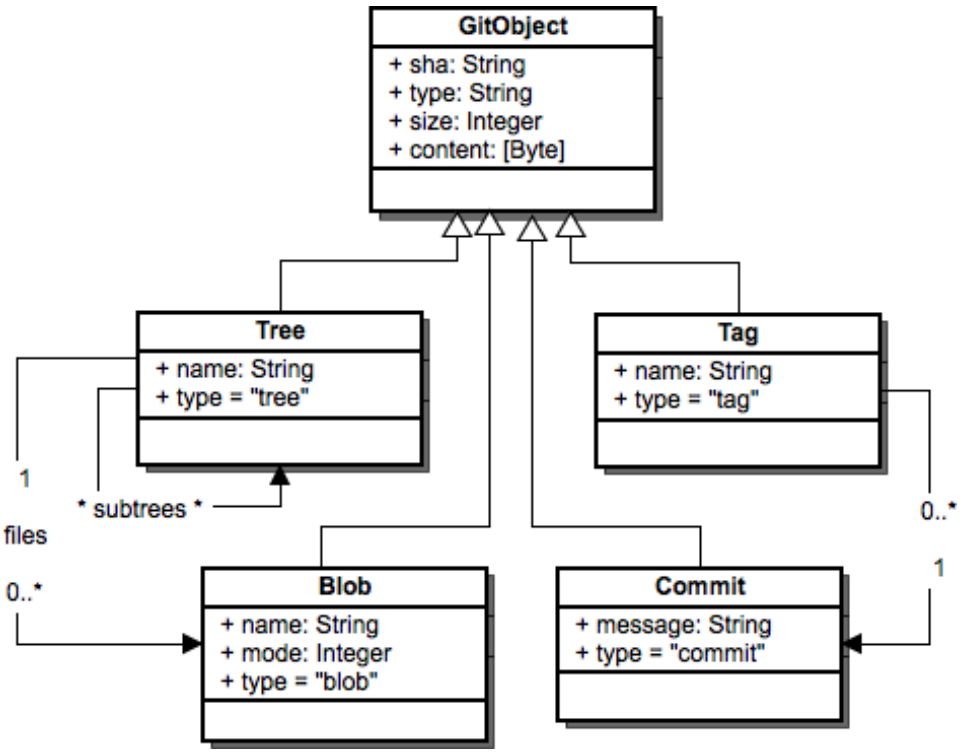

Fig.2 Git's Objects
(The Architecture of Open Source Applications (Volume 2) Susan Potter)

In the book's chapter about Git, we can find this diagram, which could be encapsulated within the first diagram designed by us, namely in the Object Database module (.git/objects). We still decided to include this in this document because we considered this diagram to be very insightful regarding the data model Git uses to store all of the data.

Git stores information about files and directories as a collection of interconnected objects. These objects can be trees, blobs, commits, and tags.

- **Trees** represent directory structures. A tree object stores the names of entries, and each entry can be a blob or another tree.
- **Blobs** store the actual content of files.

- **Commits** capture the state of the project at a specific point in time. A commit object stores a reference to the tree object that holds the snapshot of the project's files and directories, the commit message, and the author and committer information.
- **Tags** are lightweight references to specific commits. They provide a way to name and refer to specific versions of your project.

A SHA string is present in every object for the following reasons.

- **Uniqueness:** Imagine fingerprints for your data. Each Git object has a unique 40-character SHA identifier, just like a fingerprint is unique to an individual. This ensures that identical content always has the same SHA, regardless of how many times it's stored.
- **Integrity Check:** Think of the SHA as a checksum for your data. Any change, even a tiny one, will result in a completely different SHA. This acts like a tamper-proof seal, revealing any data corruption or modification since the object was created.
- **Content Verification:** If you ever suspect your data might be messed up, recalculating the SHA and comparing it to the original is like re-scanning a fingerprint. If they match, the data is intact. If not, something has changed.

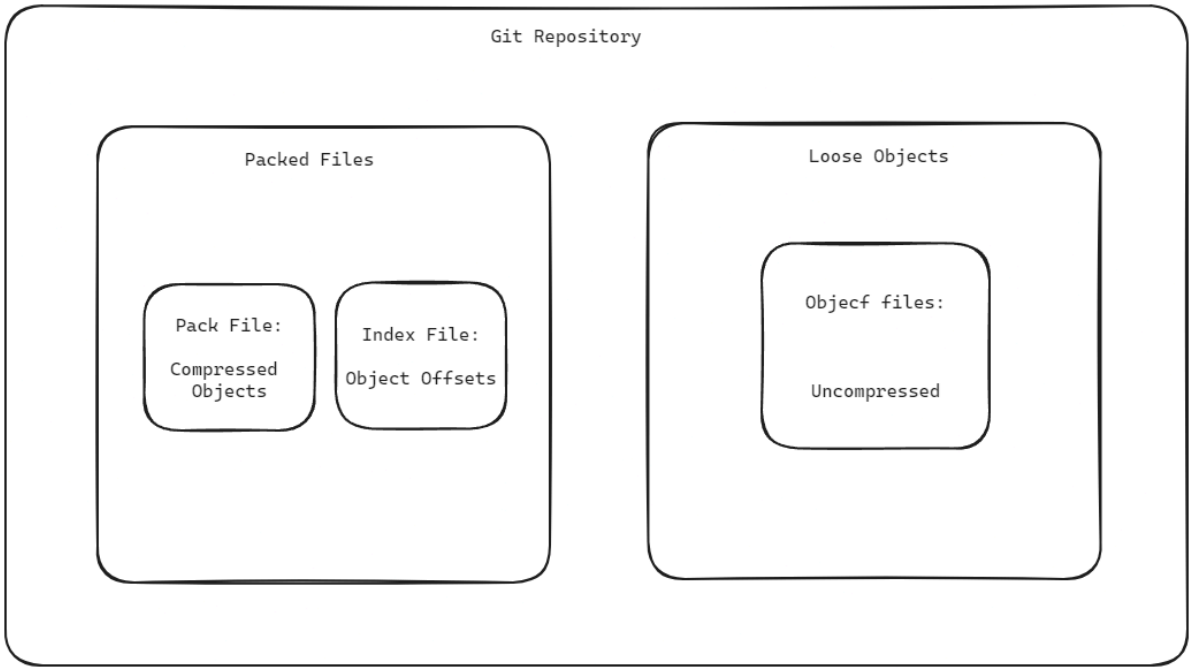

Fig.3 Git's Storage and Compression Techniques

In Git, objects are initially stored as loose objects when first created or modified. To optimize storage space, Git periodically packs loose objects into a single pack file using compression techniques. This pack file, along with its corresponding index file, provides efficient storage and retrieval of versioned content. Loose objects may still exist alongside packed objects, especially during active development phases. However, packing objects helps reduce storage overhead and enhances repository performance.

- **Packed Objects**: Objects that have been compressed and packed into a single pack file for efficient storage.
    - Pack File: A single file containing compressed objects, optimized for storage and transmission.
        - Compressed Objects: Objects (blobs, trees, commits) compressed using compression algorithms.
    - Index File: An index file containing offsets to locate specific objects within the pack file.
        - Object Offsets: Offsets pointing to the location of objects within the pack file.
- **Loose Objects**: Individual objects stored as separate files in their uncompressed form.
    - Object Files: Separate files containing uncompressed objects (blobs, trees, commits).