Application Management

Los Del DGIIM, `losdeldgiim.github.io`

Doble Grado en Ingeniería Informática y Matemáticas

Universidad de Granada

Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional (CC BY-NC-ND 4.0).

Eres libre de compartir y redistribuir el contenido de esta obra en cualquier medio o formato, siempre y cuando des el crédito adecuado a los autores originales y no persigas fines comerciales.

Application Management

Los Del DGIIM, losdeldgiim.github.io

Arturo Olivares Martos

Granada, 2025

Índice general

1. Application Lifecycle Management (ALM)

1.1. Application Lifecycle Management (ALM)

Creating an Application is not just installing it and updating it, it should cover much more aspects throughout its whole lifecycle, from the initial idea to its end of life. With that in mind, we define the following.

Definición 1.1 (Application Lifecycle Management (ALM))**.** ALM is the framework that defines the process of managing an Application throughout its whole lifecycle, from the initial idea to its end of life. It integrates people, processes and tools to manage the Application effectively and efficiently.

This framework lets to manage the complexity of modern Applications. Nowadays there are a lot of different people involved in the creation and maintenance of an Application (Developers, Bussiness Analysts, Testers, Final Users, etc). ALM provides a structured approach to coordinate all these people and their tasks. This lets everyone to know *what should they do at any moment*. This leads to overcome the typical "controled chaos" that usually happens in large projects.

The phases of the ALM are the following ones:

- Design & Development
- Continuous Integration
- Source Control & Configuration Management
- Quality Assurance
- Requirement Management

Its main goals are the following ones:

- Create fastly high-quality products.
- Definition of tasks, roles and responsibilities.
- Knowing which tasks are being done, by whom and when.
- It improves the communication between teams.

It takes into account one big problem: *too much planning can have negative consequences*. If every single detail is planned, it can lead to a lack of flexibility and adaptability to changes. Therefore, when new changes are planned, they are usually not taken into account, leading to a worse final product.

In addition, the first, difficult step in ALM is to define the requirements of the Application. It should be taken into account that many organizations are in a hurry to develop and release the Application in order to be competitive in the market. With that in mind, a minimum set of requirements that gives them the competitve advantage should be defined. This firsy prototype should be developped and released as soon as possible. After that, new features can be added in future versions of the Application.

1.2. Software Development Lifecycle (SDLC)

The Software Development Lifecycle (SDLC) is a methodology that defines the process of creating and maintaining software applications. It is a structured approach that covers all the phases of the software development process, from the initial idea to its end of life. It defines some guidelines and best practices to reduce future problems. It also helps to decide the responsibilities of each team member, so that everyone knows what they should do at any moment.

It should not be confused with the following concepts:

VS ALM : SDLC is a part of ALM. While ALM covers the whole lifecycle of an Application, SDLC focuses on the development phase.

VS System Development Lifecycle : System Development Lifecycle also takes into account testing and using softwares from third parties. It is really important to not blindly trust third-party softwares, as they can have security vulnerabilities or other problems that can affect the final product. There are ISO standards for Software and System Development Lifecycles.

The following subsections describe some concretes SDLC models. The number of phases of the SDLC can vary depending on the model used. Regarding documentation (which is usually not included into the phases), it often overlooked (functional software is more important than comprehensive documentation), but it should ideally be complete.

1.2.1. Waterfall Model

The Waterfall Model is a linear and sequential approach to software development. It is the one that has been historically used the most.

Advantages : It is easy to understand and manage. The phases do not overlap.

Disadvantages : It is inflexible to changes. Once a phase is completed, it is difficult to go back to it. In addition, a working product is not available until the end of the process.

It should only be used when the requirements are well understood and unlikely to change during the development process. It is not suitable for complex or large projects where requirements may evolve over time.

Phases

1. Requirements and Analysis *What should the System do?*

   A good requirements list is essential for the success of the project. Test cases should be defined at this stage to ensure that the final product meets the requirements. They should be *relevant, valid and verifiable.* They are divided into Functional Requirements (what the system should do) and Non-Functional Requirements (how the system should be, e.g., performance, security, usability, etc).

2. Design & Architecture *How should the System be designed?* Online/Offline, etc.

   The system architecture is defined at this stage. It should let the requirements be implemented effectively and efficiently. One important aspect is the scalability of the system to a higher number of users or data.

3. Implementation & Coding *How is the System going to be coded?*

   The actual coding of the system is done at this stage. It should be taken into account that the knowledge of the different stakeholders can vary a lot.

4. Testing & Quality Assurance *Does the System meet the requirements?*

   Testing is so important that it should be done in parallel with the coding (Test-Driven Development and Continuous Integration). The final tests are usually done by a different team (QA Team) to ensure the objectivity of the tests. In really critical systems, formal verification techniques can be used to mathematically prove that the system meets its requirements.

5. Deployment & Maintenance *How do the deployment and updates work?*

   The system is deployed. Two aspects should be taken into account:

   - Continuous Deployment: The changes in the code should be automatically considered. Automatic tests should be done to ensure that the new code does not introduce new bugs.
   - Maintenance: A balance between new versions and bug fixing the current version should be found. Releasing new updates can be difficult depending on the type of Application.

1.2.2. Agile Model

Scrum

Scrum is an Agile iterative and incremental framework for managing software development projects. There are three main roles:

1. Product Owner: Responsible for defining the product vision (client representative).
2. Scrum Master: Responsible for ensuring that the Scrum process is followed.
3. Development Team: Responsible for delivering the product increment.

The development process is divided into Sprints (usually 2-4 weeks long). Each Sprint has 4 main events:

1. Sprint Planning
2. Daily Scrum
3. Sprint Review
4. Sprint Retrospective

The main tools (artifacts) used in Scrum are:

- Product Backlog: List of all desired work on the project.
- Sprint Backlog: List of tasks to be completed in the current Sprint.
- Increment: The sum of all the completed products.

DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). Its main goal is to shorten the development lifecycle and provide continuous delivery with high software quality. It emphasizes collaboration, communication, and integration between development and operations teams.

Agile ALM

The agile ALM is a flexible and iterative SDLC approach that focuses on delivering value to the customer through continuous feedback and improvement. It follows the principles of Agile development.

$$\begin{gathered}\text{Individuals and interactions} \succ \text{Processes and tools}\\ \text{Working software} \succ \text{Comprehensive documentation}\\ \text{Customer collaboration} \succ \text{Contract negotiation}\\ \text{Responding to change} \succ \text{Following a plan}\end{gathered}$$

The principles of Agile ALM are:

1. Satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently.

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals.
6. The most efficient method of conveying information is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. Everyone should maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2. Version Control System (VCS)

Definición 2.1 (Version Control System (VCS))**.** A Version Control System (VCS) is a software tool that helps to manage changes to source code over time. It keeps track of every modification made to the code, also allowing to revert to previous versions if needed.

2.1. Types of VCS

There are three main types of VCS:

- Local VCS: It stores all the changes in a local database on the developer's computer. It is simple to use, but it does not provide collaboration features. One example is GNU RCS (Revision Control System).

- Centralized VCS: It uses a central server to store all the changes. Follows a client-server architecture.

  Advantages : Clear control access, correct backup of big files and locking mechanism.

  Disadvantages : Single point of failure, limited offline capabilities and the possibility of forgetting to release the locks.

  Only the original file and then each specific changes (deltas) are stored, not every version of the file. This saves space but depends on the whole set of deltas to reconstruct a specific version. One example is Subversion (SVN).

- Distributed VCS: It allows every developer to have a complete copy (including history) of the repository on their local machine. This provides better collaboration features and allows developers to work offline.

In the following section, we will focus on Distributed VCS, and specially on one of the most popular ones: Git.

2.2. Git

Git is a distributed version control system that is widely used in the software development industry. Opposite to the delta-based approach of the centralized VCS, Git uses a *snapshot-based* approach. It keeps a list of the whole snapshots of the filesystem at specific points in time. Almost every operation in Git is performed *locally*, which makes it really fast. Given that the history is stored locally, it allows to work and to restore previous versions even when offline. It uses *Hashing* (SHA-1) to identify every single commit, ensuring the integrity of the codebase.

Git has some diffeent states for each file, which are stored in three different areas:

- Modified (Stored in the working directory): The file has been changed but not yet staged for commit (not registered in the repository).
- Staged (Stored in the staging area): The file has been modified and is marked to be included in the next commit.
- Committed (Stored in the local repository, `.git` folder): The file has been saved in the local repository.

Some important Git commands are the following ones:

- `git init`: Initializes a new Git repository (creates the `.git` folder).
- `git clone <repo_url>`: Clones an existing repository from a remote server to the local machine.
- `git add <file>`: Stages a file for commit. It also lets tracking new files and indicating when a file conflict has been resolved.

  Using the option `-A` stages all the changes (new, modified and deleted files).
- `git commit -m "message"`: Commits the staged changes to the local repository with a descriptive message.
- `git status`: Shows the current status of the working directory and staging area. Indicates which files are untracked, modified, deleted or staged for commit.

  Observación. All the files that are not tracked will always appear in red when using `git status`. Sometimes they are intended to be untracked (e.g., temporary files, build artifacts, etc). In that case, they can be added to the `.gitignore` file to avoid them appearing in the status. That will make Git ignore them.
- `git diff`: Shows the differences between files in different states (working directory, staging area, last commit). Has different options to compare specific states:
  - `git diff`: Working directory vs Staging area.
  - `git diff --staged`: Staging area vs Last commit.
  - `git diff HEAD`: Working directory vs Last commit.
- `git log`: Displays the commit history of the repository. Using the option `-p` shows the differences introduced in each commit.

2.2.1. Git LSF

Git Large File Storage (Git LFS) is an extension for Git that is designed to handle large files more efficiently. Given that Git is based on snapshots, storing large files directly in the repository can lead to repeated storage of the same file in different commits, which can quickly bloat the repository size. Git LFS solves this problem by replacing large files with lightweight references in the Git repository. Instead of storing the actual file content in the repository, Git LFS stores a pointer to the file in the Git repository and keeps the actual file content in a separate storage location. It is especially useful for binary files (e.g., images, videos, audio files, `pptx`, etc).

Some important Git LFS commands are the following ones:

- `git lfs install`: Installs Git LFS in the local repository.
- `git lfs track "<file_pattern>"`: Tracks files matching the specified pattern with Git LFS.
- `git lfs ls-files`: Lists the files that are being tracked by Git LFS.

2.2.2. Branching

Branching is a powerful feature in Git that allows developers to create separate lines of development within a repository. This enables multiple developers to work on different features or bug fixes simultaneously without interfering with each other's work. Each branch represents an independent line of development, allowing changes to be made in isolation. Once the changes in a branch are complete and tested, they can be merged back into the main branch (usually called `main` or `master`).

In order to explain branching, we need to define the concept of *HEAD*.

Definición 2.2 (*HEAD* Pointer)**.** The `HEAD` pointer is a reference to the current commit that the working directory is based on. It indicates the current position in the repository's history.

Some important commands related to branching are the following ones:

- `git branch <branch_name>`: Creates a new branch with the specified name that points to the current commit. With the option `-d`, it deletes the specified branch (if it has been merged).
- `git checkout <branch_name>`: Switches to the specified branch, updating the working directory to reflect the state of that branch. It then also updates the `HEAD` pointer to point to the new branch (the latest commit of that branch).

2.2.3. Branch Integration

When the development in a branch is complete, it is often necessary to integrate the changes back into another branch (usually the `main` or `master` branch). There are two main methods for integrating branches in Git: merging and rebasing.

Merging

The main command is the following one:

- `git merge <branch_name>`: Merges the specified branch into the current branch. It combines the changes from both branches, creating a new commit that represents the merged state.

There are two main strategies for merging branches:

- Fast-Forward Merge: If the current branch has not diverged from the branch being merged, Git simply moves the `HEAD` pointer forward to the latest commit of the merged branch. No new commit is created in this case.

- Recursive Merge: If the branches have diverged, Git creates a new commit that combines the changes from both branches. This new commit has two parent commits: one from each branch.

In the latter strategy, conflicts may arise if the same lines of code have been modified in both branches. Git will mark these conflicts in the affected files, and it is the developer's responsibility to resolve them manually before completing the merge. They can use `git status` to see which files have conflicts and need to be resolved. After resolving the conflicts, the developer can stage the changes and complete the merge by committing the changes.

Rebasing

This strategy uses the following command:

- `git rebase <branch_name>`: Reapplies the commits from the current branch on top of the specified branch. It effectively moves the entire branch to start from the latest commit of the specified branch. With the option `--continue`, it continues the rebase process after resolving conflicts.

After rebasing, the commit history appears linear, and therefore a simple fast-forward merge can be performed to integrate the changes into the target branch.

When rebasing, the commit history is rewritten, which can make it appear cleaner and more linear. However, the details about the mistakes committed during the development in the feature branch may be lost. In addition, it is dangerous to rebase branches that have already been pushed to a remote repository, as it can lead to confusion and conflicts for other developers working on the same branch, as their local copies will have a different history than the rebased branch.

2.3. Distributed Git

Until this point, we have only talked about local operations. To collaborate with other developers, it is necessary to use remote repositories.

2.3.1. Remote Repositories

A remote repository is a version of the repository that is hosted on a remote server. It allows multiple developers to collaborate on the same codebase. All developers should synchronize their local repositories with the remote repository to share changes and keep their code up to date. When more than one developer is working on the same codebase, conflicts may arise if two developers modify the same lines of code in different ways. Some important commands to interact with remote repositories are the following ones:

- `git remote add <name> <url>`: Adds a new remote repository with a specific name (e.g., `origin`).
- `git push <remote> <branch>`: Pushes the local commits to the specified remote repository and branch.
- `git fetch <remote>`: Fetches the latest changes from the specified remote repository without merging them into the local branch.
- `git pull <remote> <branch>`: Fetches and merges changes from the specified remote repository and branch into the local branch.

2.3.2. Collaboration Workflows

There are different collaboration workflows that teams can follow when using Git, depending on their preferences and project requirements. In the following, we describe three common workflows.

Centralized Workflow

The repository has a single central shared repository. All developers clone this repository, make changes in their local copies, and then push their changes back to the central repository. When more than one developer changes the same lines of code, conflicts may arise during the push operation, which need to be resolved before the changes can be successfully pushed.
This workflow is simple and easy to understand, making it suitable for small teams or projects with straightforward collaboration needs.

Integration Manager Workflow

In this workflow, there are two types of developers:

- Integration Manager: He is the responsible and manteiner of the project.
- Contributors: They suggest the integration managers the changes they want to make to the codebase.

As well as the two types of developers, there are also two types of repositories:

- Blessed Repository: It is maintained by the integration manager. It is the main repository where all the changes are eventually integrated, and it is considered the authoritative source of the codebase.

- Developper Repositories: They are maintained by the contributors. Each contributor has their own repository where they can make changes and experiment with new features. Each developper has their public repository and its local copy.

The workflow works as follows:

1. The integration manager creates the blessed repository, and make it public.
2. Each contributor clones the blessed repository to create their own developper repository (that is, a fork of the blessed repository).
3. Contributors make changes in their local copies and push them to their developper repositories.
4. When a contributor wants to suggest changes to the blessed repository, they email the integration manager asking him to make those changes (this is typically done through a pull request).
5. The integration manager adds the contributor's repository as a remote, fetches the changes, reviews them, and if everything is fine, merges them into their local blessed repository.
6. Finally, the integration manager pushes the updated blessed repository to the remote server, so that all contributors can access the latest changes.

This workflow allows for better control and review of changes, as the integration manager can carefully evaluate each contribution before integrating it into the main codebase. It is suitable for larger projects with multiple contributors.

Dictator and Lieutenants Workflow

This workflow is similar to the Integration Manager Workflow, but with a hierarchical structure. In this case, a new upper role is introduced: there are more than one integration managers, called *lieutenants*, and they report to a single *dictator* (the main integration manager). Each lieutenant is responsible for a specific area of the codebase and manages contributions related to that area.

The workflow works as follows:

1. Each contributor works on their own branch created for their feature or bug fix.
2. When needed, the lieutenants merge the branches of the contributors into their own master's branches.
3. When needed, the dictator merges the master's branches of the lieutenants into his own master's branch.
4. Finally, the dictator pushes the updated blessed repository to the remote server, so that all contributors can access the latest changes.

This workflow allows for better organization and management of contributions, as each lieutenant can focus on their specific area of expertise. It is suitable for large projects with multiple teams working on different aspects of the codebase. For instance, the Linux kernel development follows this workflow.

2.4. Git Internals

Git is built around a few fundamental concepts that enable its powerful version control capabilities. Understanding these concepts can help developers use Git more effectively and troubleshoot issues when they arise.

2.4.1. Objects

Git stores all its data in a simple key-value database, where the key is a SHA-1 hash of the content, and the value is the actual content. There are four main types of objects in Git:

- Blob: It represents the content of a file. It does not contain any metadata (e.g., filename, permissions, etc).
- Tree: It represents a directory. It contains references to blobs (files) and other trees (subdirectories), along with their names and permissions.
- Commit: It represents a snapshot of the repository at a specific point in time. It contains:
  - A reference to a tree object that represents the state of the repository at that commit (the root tree).
  - References to parent commit(s) (the previous commit(s) in the history).
  - Metadata such as the author, committer, timestamp, and commit message.

All of the objects are stored in the `.git/objects` directory. There, you can find subdirectories named with the first two characters of the SHA-1 hash, and inside those subdirectories, you can find files named with the remaining 38 characters of the hash. Git uses a combination of compression and delta encoding to store these objects efficiently, minimizing disk space usage. To see the objects stored in a Git repository, some useful commands are:

- `git cat-file -t <object_hash>`: Displays the type of the specified object (blob, tree, commit, etc).
- `git cat-file -p <object_hash>`: Displays the content of the specified object.
- `git ls-tree <tree_hash>`: Lists the contents of the specified tree object.
- `git show <commit_hash>`: Displays the details of the specified commit, including the commit message, author, date, and the changes introduced in that commit.

In all the cases, the <object_hash> is the SHA-1 hash of the object you want to inspect. Normally the first few characters of the hash are enough to uniquely identify the object.

3. Build Engineering und Continuous Integration

The main aim of this chapter is to introduce the concept of Continuous Integration (CI) and its significance in modern software development. However, before diving into CI, a good understanding of build engineering is essential, as it forms the foundation for effective CI practices.

3.1. Build Engineering

Build engineering refers to the process of compiling source code into executable programs. This process should ideally be automated to ensure consistency, efficiency, and reliability. It is crucial in order to increase productivity and reduce human error during the build process.

To start with the building process, engineering teams typically start from available scripts (e.g., Ant, Maven, Make) that automate the build process. However, these scripts often need to be adapted to support Quality Assurance (QA) and deployment on production systems. This is where the role of a Build Engineer becomes vital.

There are usually two types of methologies for build engineering:

Using IDEs : Integrated Development Environments (IDEs) provide built-in tools for building and managing projects. However, using them can lead to inconsistencies, as different developers may have different IDE configurations.

Command-Line Build : Command-line build is usually preferred in professional environments. It allows for greater control and automation, ensuring that builds are consistent across different environments. It also lets the build scripts be version-controlled alongside the source code.

One important aspect of build engineering is the security of the build process. In order to ensure that the build process is secure, there are three concepts that need to be taken into account:

- **Automation**: The build process should be fully automated to minimize human intervention and reduce the risk of errors. These scripts should also follow the “Failing Fast” principle, which means that if an error occurs during the build process, it should stop immediately and report the error.

```yaml
name: C++ CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up CMake
      uses: jwlawson/actions-setup-cmake@v1
      with:
        cmake-version: '3.18.4'
    - name: Build
      run: |
        mkdir build
        cd build
        cmake ..
        cmake --build .
    - name: Test
        run: |
            cd build
            ctest --output-on-failure
    -
```

Código fuente 1: Example of a GitHub Actions workflow for building and testing a C++ project using CMake.

- **Secure Supply Chain**: The build process should ensure that all dependencies and components used in the build are secure and trustworthy. This includes verifying the integrity of third-party libraries and tools. Isolated build environments (e.g., using containers) can help mitigate risks associated with compromised dependencies.
- **Secure Trusted Base**: In the event of cyberattacks, it is crucial to accurately identify the software that has been compromised. This includes knowing which version of the software was deployed and whether it was deployed correctly. Methods for achieving this include using version numbers, hash functions, and creating a Manifest file that contains all configuration parameters.

3.1.1. GitHub Actions

In thuis building process, CI tools are used to automate the build and testing of code changes. One popular CI tool is GitHub Actions, which allows developers to create custom workflows that are triggered by specific events in a GitHub repository. In a repository, workflows are defined in YAML files located in the `.github/workflows/` directory. These workflows can include various jobs, such as building the code, running tests, and deploying the application. An example of a simple GitHub Actions workflow that builds and tests a `C++` project using `CMake` is shown in the Source Code 1.

3.2. Continuous Integration (CI)

The concept of Continuous Integration (CI) revolves around the idea of frequently integrating code changes into a shared repository. During the normal development process, this integration is not done frequently enough, leading to integration problems and conflicts when multiple developers work on the same codebase. The aim is to continuously integrate code changes with every commit, ensuring that the code is compilable and that the executable passes all tests.

In order to achieve CI, apart from the agreement among developers to follow this practice, the following are required:

- A version control system (e.g., Git) to manage code changes and facilitate collaboration among developers.
- An automated build script that compiles the code and runs tests.
- CI server (e.g., Jenkins, Bamboo) that monitors the version control system for changes, triggers the build process, and stores the results.
- An automated deployment of the software to a test environment.

A lot of the stakeholders in a software project benefit from CI, including developers (who get immediate feedback on their changes), QA teams (who can run automated tests) and project managers (who can get stadistics on build health and code quality).

For a good CI practice, really frequent commits are necessary. With CI, with each commit, the code is integrated, built, and tested automatically[1]. This helps to identify and fix integration issues early, reducing the risk of conflicts and bugs. A good practice is to firstly pull the latest changes from the main branch, resolve any conflicts locally, and then push the changes to the shared repository. In addition, the changes should be small and with a low complexity, making it easier for the other developers to solve the possible conflicts.

All this building process should be carried out in the "Build Farm", which is a dedicated environment for building and testing the software. It should be administrated separately from the development environment to ensure consistency and reliability. The build farm should also be scalable to handle multiple builds simultaneously, especially in large projects with many developers. If desired, developers should also be allowed to build and test their changes locally to reduce the load on the build farm and get faster feedback.

3.2.1. CI and Branches

CI and branching strategies do not fit well together, as branches are by nature changes in the code that should not yet be integrated into the main codebase. In

[1] This can mean an overload in the early stages. A possible solution is "Nightly Builds", where the build process is run once a day, usually at night.

order to mitigate this, the number of branches should be minimized and the changes in the master branch should at least once a day be merged into the feature branches. In addition, the lifetime of branches should be considered:

- Most branches should be short-lived, lasting only a few days to a week. This minimizes the risk of conflicts and integration issues.
- However, sometimes long-lived branches are necessary, for example, when it is not still clear which features will be included in the software release, and one will be later merged into the main branch. It should be clear from the beginning that late-binding always carries risks.

3.2.2. Testing

Without testing, CI can only ensure that the code compiles successfully. To ensure that the code behaves as expected, automated tests should be included in the CI process. Appart from specific types of tests (e.g. code quality tests, security tests), there are three main types of tests that should be considered:

Unit Tests : These tests focus on small units of code, such as functions or methods, to ensure they work correctly in isolation. No database or external systems should be involved. The whole application should not be started. Usually less than 10 minutes are required to run all unit tests.

Component Tests : More than one unit is tested together, possibly involving databases or external systems. The whole application is still not started.

Acceptance Tests : These tests validate the entire application against the requirements. The whole application is started. Usually more than a day is required to run all acceptance tests.

The CI process should also consider the time all this testing takes. If it takes too long, developers may have to wait too much time to get feedback on their changes, or while the tests are running, they may continue working on other tasks, leading to more integration issues later. A possible solution is to have a “Smoke Test Suite” that runs a subset of the tests that can give quick feedback on the most critical functionalities of the application, and only run the full test suite at specific times (e.g., nightly).

4. Übungen

4.1. Application Lifecycle Management (ALM)

Ejercicio 4.1.1.

1. Erklären Sie den Begriff "Application Lifecycle Management" (ALM). Geben Sie an, welche Phasen im ALM typischerweise enthalten sind und warum das Verständnis dieser Phasen für das App Management wichtig ist.

2. Beschreiben Sie die Bedeutung der Sicherheit im Application Management. Nennen Sie mindestens drei Sicherheitsaspekte, die bei der Entwicklung und Verwaltung von Anwendungen zu berücksichtigen sind.

3. Vergleichen Sie die Phasen des Application Lifecycle Management (ALM) mit den Phasen des Software Development Lifecycle (SDLC). Identifizieren Sie mindestens zwei Gemeinsamkeiten und zwei Unterschiede zwischen diesen beiden Ansätzen.

Ejercicio 4.1.2. Erklären Sie die Vor- und Nachteile der folgenden Entwicklungsmethoden:

1. Agile Entwicklung

2. Scrum

3. Wasserfall-Modell

4. DevOps-Ansatz

Ejercicio 4.1.3. Nehmen Sie an, Sie sind der Manager eines kleinen Softwareentwicklungsteams, das eine Echtzeit-Messaging-App für den Campus der "Universität der Zukunft" entwickelt. Diese App ermöglicht Studierenden und Professoren eine einzigartige Kommunikation, die den Alltag auf dem Campus einfacher und unterhaltsamer macht. Erklären Sie, warum es wichtig ist, von Anfang an ein effektives Application Management in Ihre Projekte zu integrieren. Geben Sie konkrete Beispiele für mögliche Probleme, die vermieden werden könnten, wenn Sie sich frühzeitig auf das Application Management konzentrieren, um sicherzustellen, dass Ihre App im Universitätsalltag reibungslos funktioniert.

Ejercicio 4.1.4. Ihr Team entwickelt weiterhin die Echtzeit-Messaging-App. Beschreiben Sie, wie Scrum den Entwicklungsprozess strukturiert.

4.2. Version Control System (VCS)

Ejercicio 4.2.1. Beschreiben Sie die grundlegenden Unterschiede zwischen Git und SVN hinsichtlich ihrer Arbeitsweise und ihres Datenmodells. Erläutern Sie, was ein verteiltes Versionskontrollsystem (Git) und ein zentrales Versionskontrollsystem (SVN) sind.

Ejercicio 4.2.2.

1. Nehmen Sie an, Sie arbeiten mit einem kleinen Team an der Echtzeit-Messaging-App für die „Universität der Zukunft“. Sie haben noch lokale Änderungen, die noch nicht committet sind. Erstellen Sie bitte einen Commit und schließen Sie somit die Entwicklung des neuen Features ab.

2. Da das neue Feature nun fertig implementiert ist, möchten Sie dafür sorgen, dass es auch in den aktuellen master aufgenommen wird. Der übliche Prozess in Ihrem Team ist, einen Merge Request (MR) zu erstellen, der den Feature-Branch in den master übernimmt. Es gibt die Richtlinie, dass MRs nur dann akzeptiert werden, wenn sie mit dem master aktuell sind und keine Konflikte erzeugen. Sorgen Sie dafür, dass der Feature-Branch `cool_stuff` mit dem master-Branch aktuell ist.

Ejercicio 4.2.3. Ihr MR wurde akzeptiert und das neue Feature ist im master. Parallel dazu haben Sie im `other_cool_stuff` -Branch noch an einem ähnlichen Feature gearbeitet. Bereiten Sie auch diesen Branch für den MR in den master vor.

Ejercicio 4.2.4. Ein Kollege von Ihnen ist erst seit Kurzem im Team und bittet Sie um Hilfe, da er lokale Änderungen vorgenommen hat, die er jedoch nicht committen kann. Helfen Sie ihm, alle geänderten und neu hinzugefügten Dateien zu committen.

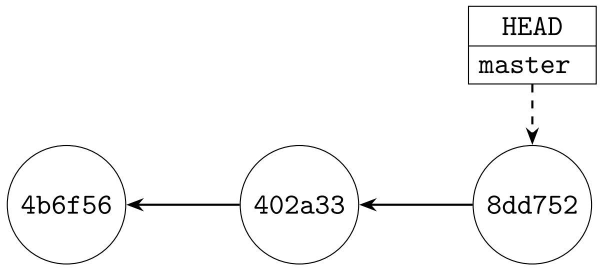

Figura 4.1: Git-Repository mit drei Commits und einem Branch `master`.

4.3. Distributed Git und Internals

Ejercicio 4.3.1. In Aufgaben 4.2.2.2 und 4.2.3 des vorherigen Aufgabenblatts sollten Sie die Änderungen des master-Branches in den aktuellen Feature-Branch übernehmen. Überlegen Sie sich eine weitere Möglichkeit, die Änderungen zu übernehmen.

Ejercicio 4.3.2. Überlegen Sie sich mögliche Vor- und Nachteile der drei vorgestellten Distributed Workflows.

1. Dictator and Lieutenants Workflow
2. Integration-Manager Workflow
3. Centralized Workflow

Ejercicio 4.3.3. Der Chef des Teams, zuständig für die Entwicklung der Echtzeit-Messaging-App hat wenig Ahnung von Softwareentwicklung. Er hat damals einfach irgendwelche Regeln bezüglich des Merge-Prozesses festgelegt, weiß aber nicht wirklich, was diese bedeuten, und bittet Sie, einen sinnvollen Workflow für das Projekt zu wählen. Wählen Sie einen passenden Workflow aus und begründen Sie Ihre Wahl.

Ejercicio 4.3.4. Ihr Kollege bittet Sie erneut um Hilfe. Dieses mal sei es ernst, er hat all seine Arbeit der letzten Wochen verloren. Da er sich nicht gut mit Git auskennt, committet er selten. Als er fertig war, wollte er committen. Dafür hat er seine Änderungen mit `git add -A` in den Staging-Bereich gepackt. Doch da ist ihm eingefallen, dass er vorher noch Änderungen vom Remoteserver herunterladen muss. Also führt er git fetch aus und sieht, dass Remote-Änderungen übernommen wurden. Doch als er die Änderungen dann endlich in seinem lokalen Branch hat, sind alle seine eigenen Änderungen verschwunden. Helfen Sie Ihm die verlorenen Dateien wiederherzustellen, nutzen Sie dafür das im Moodle bereitgestellte Repository.

Ejercicio 4.3.5. Nehmen Sie an, Sie haben ein Git-Repository, welches die in Abbildung 4.1 dargestellte Commit-Graphen hat. Nehmen Sie nun an, Sie führen die untenstehenden Git-Befehle aus. Zeichnen Sie den Commit-Graphen nach jedem Befehl. Wenn ein neuer Commit-Hash berechnet wurde, wählen Sie bitte eine eindeutige zufällige Nummer.

1. `git checkout HEAD~1`
2. `git checkout -b 'feature_branch'`

3. `git commit -m 'new feature'`

4. Zeichnen Sie bitte beide Graphen

   a) `git merge master`

   b) `git rebase master`

5. Führen Sie abschließend für beide Graphen einen Merge von `feature_branch` in den `master` aus und löschen Sie den obsoleten Branch.

Ejercicio 4.3.6. Nutzen Sie das Git-Repository aus Moodle.

1. Angenommen, Sie haben gerade Ihre neuesten Änderungen committet und möchten vor dem Pushen testen, ob alles noch funktioniert. Dabei fällt Ihnen auf, dass ein Anführungszeichen fehlt. Sie haben es bereits hinzugefügt, finden es jedoch unnötig, dafür einen neuen Commit anzulegen. Fügen Sie die Änderung Ihrem lokalen Commit hinzu.

2. Nachdem Sie die Änderung vorgenommen haben, stellen Sie fest, dass die Commit-Nachricht nicht alle Änderungen widerspiegelt. Sie möchten der Nachricht hinzufügen, dass die neue Funktion auch in `main.py` aufgenommen wurde.

4.4. Continuous Integration

Ejercicio 4.4.1. Ihre Tests zeigen, dass eine Funktion nicht mehr korrekt funktioniert. Identifizieren Sie den Commit, ab dem der Fehler eingeführt wurde.

Ejercicio 4.4.2. Für die Echtzeit-Messaging-App der „Universität der Zukunft“ soll eine CI/CD-Pipeline aufgebaut werden.

1. Welche Schritte sollte die Pipeline umfassen und welche Werkzeuge könnte man dafür nutzen?
2. In Section 4.2 haben Sie bereits Git-Hooks kennen gelernt. Wie könnten Sie diese in einer CI- bzw. CD-Pipeline benutzen?
3. Welche Branching-Strategie für die Echtzeit-Messaging-App würden Sie vorschlagen?
4. Welche Unit-, Component- und Acceptance-Tests würden zur Messaging-App passen?