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) 5
- 1.1. Application Lifecycle Management (ALM) . . . 5
- 1.2. Software Development Lifecycle (SDLC) . . . 6
  - 1.2.1. Waterfall Model . . . 6
  - 1.2.2. Agile Model . . . 7

2. Version Control System (VCS) 11
- 2.1. Types of VCS . . . 11
- 2.2. Git . . . 11
  - 2.2.1. Git Bisect . . . 13
  - 2.2.2. Git LSF . . . 14
  - 2.2.3. Branching . . . 14
  - 2.2.4. Branch Integration . . . 15
  - 2.2.5. Git Hooks . . . 16
- 2.3. Distributed Git . . . 17
  - 2.3.1. Remote Repositories . . . 17
  - 2.3.2. Collaboration Workflows . . . 17
- 2.4. Git Internals . . . 19
  - 2.4.1. Objects . . . 19
  - 2.4.2. Git Filesystem-Check . . . 20

3. Build Engineering und Continuous Integration 21
- 3.1. Build Engineering . . . 21
  - 3.1.1. GitHub Actions . . . 22
- 3.2. Continuous Integration (CI) . . . 22
  - 3.2.1. CI and Branches . . . 24
  - 3.2.2. Testing . . . 24

4. Deployment Strategies and DevOps 27
- 4.1. Disaster Planning . . . 27
  - 4.1.1. Zero-Downtime Releases . . . 28
  - 4.1.2. Emergency fixes . . . 28
- 4.2. Deployment Pipeline . . . 29
  - 4.2.1. Guidelines for a Deployment Pipeline . . . 29
  - 4.2.2. Phases of a Deployment Pipeline . . . 29
  - 4.2.3. Deployment of User-Installed Software . . . 30
  - 4.2.4. Modern Deployment Practices . . . 30
- 4.3. Continuous Deployment (CD) . . . 31

4.3.1. Continuous Delivery 31
4.4. DevOps 32
4.5. Deployment with Containers Technology 33
4.5.1. Isolation Measures 33
4.5.2. Docker 33

5. Secure Deployment and CA Case Study 35
5.1. Binary Provenance 36
5.2. Certificate Authorities (CA) 36
5.2.1. CA Creation 37
5.3. Human Factors in Secure Deployment 37
5.3.1. Vulnerability Management 38
5.3.2. Security Champion 38

6. Software Testing 41
6.1. Test Types 41
6.1.1. Unit Tests 41
6.1.2. Acceptance Tests 42
6.2. Program Analysis 42
6.2.1. Adress Sanitizer 42
6.3. The Quest for Coverage 43
6.3.1. Symbolic Execution 43
6.3.2. Fuzzing 44

7. Übungen 45
7.1. Application Lifecycle Management (ALM) 45
7.2. Version Control System (VCS) 46
7.3. Distributed Git und Internals 49
7.4. Continuous Integration 56
7.5. Docker 60
7.6. Deployment 61
7.7. Secure Deployment 62
7.8. Secure Deployment 2 63
7.9. Secure Development 64
7.10. Fuzzing & Z3 65
7.11. Fuzzing & Z3 2 66

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 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 know *what should they do at any moment*. This leads to overcome the typical "controled chaos" that usually happens in large projects.

Some aspects that are usually included in ALM are:

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

Its main goals are the following ones:

- Create fast 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 is 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 & 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: Define the goals and tasks for the Sprint.
2. Daily Scrum: A short daily meeting to discuss progress and obstacles.
3. Sprint Review: Review the work completed and the work not completed.
4. Sprint Retrospective: Reflect on the past Sprint and identify improvements.

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 commit --amend`: It modifies the last commit by adding the currently staged changes to it. It is useful when you forget to stage some changes before committing, or when you want to fix a mistake in the last commit (e.g., a typo in the code, a missing file, etc). It should be noted that the hash of the last commit will change after amending it. It should be used with the option `--no-edit` if you want to keep the same commit message, or with the option `-m "new message"` if you want to change the commit 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.
- `git checkout <commit_hash>`: Switches to a specific commit, allowing to view the state of the repository at that point in time.

In addition, when referencing a commit, Git allows to use some special notations:

- `<commit_hash>^[<n>]` : Refers to the n-th parent of the specified commit (for instance, when merging, more than a parent may appear). If n is not provided, it defaults to 1, referring to the first parent.
- `<commit_hash>~[<n>]` : Refers to the n-th ancestor of the specified commit. It follows the first parent of each commit, so it is useful to refer to commits in a linear history. If n is not provided, it defaults to 1, referring to the immediate parent.

2.2.1. Git Bisect

In this section, we introduce a new command, `git bisect`, which is used to find the specific commit that introduced a bug or issue in the codebase. A whole section is needed, as it is a really useful command that can save a lot of time when debugging. It uses a binary search algorithm ($O(\log n)$) to efficiently narrow down the range of commits that may have introduced the bug. The workflow for using `git bisect` is as follows:

1. Initialize the bisect process.
   1.1 `git bisect start`: Initializes the bisect process.
   1.2 `git bisect bad`: Marks the current commit as bad (the commit where the bug is present).
   1.3 `git bisect good <commit_hash>`: Marks a specific commit as good (a commit where the bug is not present).
2. Locate the commit that introduced the bug.
   a) Git will automatically check out a commit in the middle of the range between the good and bad commits. The developer needs to test this commit to determine if it is good or bad.
   b) Based on the test result, the developer will mark the commit:
      - `git bisect good`: If the commit is good, it will narrow down the search to the range between this commit and the bad commit.
      - `git bisect bad`: If the commit is bad, it will narrow down the search to the range between this commit and the good commit.
   c) This process is repeated until Git identifies the specific commit that introduced the bug.

During the whole process, the following command are useful:

- `git bisect log`: Shows the log of the bisect process, including the commits that have been marked as good or bad.
- `git bisect visualize --oneline`: Visualizes the bisect process, showing the commits that are still not tested, and where the `HEAD` is currently located.

3. End the bisect process.

   - `git bisect reset`: After finding the bad commit, this command is used to end the bisect process and return to the original state of the repository (the branch and commit that were checked out before starting the bisect). It is important to use this command to clean up the bisect state and avoid any confusion in future operations.
   - `git bisect run <script>`: This command automates the bisecting process by running a specified script that tests each commit. The script should return a zero exit code if the commit is good and a non-zero exit code (normally 1) if the commit is bad. This allows to quickly identify the bad commit without manual intervention.

This command is used in the Exercise 7.4.1, so we refer to that exercise for a practical example of how to use `git bisect`.

2.2.2. 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.

2.2.3. 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 (therefore, it allows modifiers such as `HEAD^` or `HEAD~`). It indicates the current position in the repository's history. Some important aspects about the `HEAD` pointer are the following ones:

- `@` alone is a shorthand for `HEAD`, so it can be used interchangeably.
- `HEAD@{n}` refers to the position of `HEAD` n moves ago. For instance, `HEAD@{1}` refers to the previous position of `HEAD` before the last change (e.g., before the last checkout, merge, rebase, etc).

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). With the option `-b`, it creates a new branch and switches to it in a single command.

As explained with the `HEAD` pointer, the names of branches are just pointers to specific commits (the latest commit of that branch). Therefore, modifiers such as `<branch_name>^` or `<branch_name>~` can be used to refer to commits in the history of that branch. `<branch_name>@{n}` can also be used to refer to the position of the branch pointer n moves ago.

2.2.4. 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.

Observación. Given that understanding how the git tree will be after the integration is difficult, this tool can be used to visualize the effects of 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.

2.2.5. Git Hooks

Git hooks are scripts that are automatically executed by Git before or after certain events, such as committing changes, merging branches, or pushing to a remote repository. They allow developers to customize and automate various aspects of their Git workflow. Git hooks are stored in the `.git/hooks` directory of a Git repository. Each hook is a separate script file that corresponds to a specific event. Some common Git hooks include:

- pre-commit: Executed before a commit is created. It can be used to perform checks on the code (e.g., run tests, check code style, etc) and prevent the commit if any issues are found.
- post-commit: Executed after a commit is created. It can be used to send notifications, update documentation, or trigger other actions.

- pre-push: Executed before changes are pushed to a remote repository. It can be used to run tests or perform other checks to ensure that the code being pushed meets certain criteria.

As a useful aspect, if any of the scripts ends with a non-zero exit code (denoting an error), if it was a pre- hook, the action that triggered the hook will be aborted. It garantees that certain conditions are met before proceeding with the action.

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 (GitHub, GitLab, Bitbucket, etc.). 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.

2.4.2. Git Filesystem-Check

In the following section, a new command is presented, `git fsck`, which is used to verify the integrity of the Git repository. It checks the connectivity and validity of the objects in the repository, ensuring that there are no corrupted or missing objects. An important option is the following:

- `git fsck --lost-found`: Sometimes, files may be accidentally deleted or become unreachable due to various reasons (e.g., a commit is removed, a branch is deleted, etc). However, if those files were sometime staged, they are not completely lost, as their blob is still located in the `.git/objects` directory. However, detecting them can be difficult (there may be many objects in that folder).

  This option allows to find those lost objects and recover them. It creates two new directories, `.git/lost-found/commit` and `.git/lost-found/other`, where it places the lost commits and other objects (e.g., blobs), respectively. The files in those directories are named with their SHA-1 hash, and they can be inspected using the commands described in the previous section.

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.

- **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. The triggers are specified using the `on` keyword, and include events such as `push`, `pull_request`, etc. 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.

It should be noted that GitHub Actions and Git Hooks are different concepts. While Git Hooks are scripts that run locally in a developer's machine before or after certain Git events, GitHub Actions are workflows that run in the cloud on GitHub's servers in response to events in a GitHub repository. Git Hooks are not suitable for CI/CD processes, as they are not shared among team members and cannot be easily integrated with other tools and services.

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.

```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.

- 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 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:

[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.

Unit Tests : These tests focus on small units of code, such as functions or methods, to ensure they work correctly in isolation, and are usually from a developer's perspective. 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, and are usually from an end-user's perspective. 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. Deployment Strategies and DevOps

To deploy an application means to make it available for use. This process is critical, as it ensures that the application is accessible to users in a reliable and efficient manner.

Afterwards we will explore the Deployment Pipeline, but first it should be considered the risky *first deployment*, because it has some particularities that will not be present in the rest of deployments. As it has already been explained in previous chapters, only a small prototype should be deployed at this stage, to show the basic functionality of the application to the users. An IT-environment should be prepared for this deployment, being as similar as possible to the final production environment (same operating system, same installed software, similar hardware, etc.). This will help to identify potential issues that may arise during the deployment process.

As it was with the integration phase, *automatization* is key to ensure a smooth and efficient deployment process. Every step should be scripted and with self-testing capabilities, to minimize human errors and ensure consistency across deployments. Documentation and verification that the deployment process is working as expected is also crucial. Therefore, some aspects to avoid are:

- Manually performing deployment steps.
- Deploying only when the entire development is complete.
- Manual configuration management of production environments.

4.1. Disaster Planning

Despite all the precautions taken, there is always a risk of failure during deployment. Therefore, it is essential to be able to roll back to a previous stable version of the application in case of issues (as debugging in production is not a good practice). It is important to firstly backup the status (database, data systems, etc.) that the application has changed before rolling back, in order to avoid data loss. To minimize the impact of failures and ensure service continuity, modern deployment strategies aim to reduce or completely eliminate downtime while enabling fast rollback mechanisms.

4.1.1. Zero-Downtime Releases

Zero-downtime releases, also known as Hot Deployment, should change instantly between application versions without interrupting the service for users. Easyly changing the resources (Databases, Servers, etc.) that the application is using is key to achieve this, which can be achieved by changing the URI (Uniform Resource Identifier) that the application is pointing to. Some strategies to achieve zero-downtime releases are the following.

Blue-Green Deployment

There are two versions of the application: the current production (green) and the new version to be deployed (blue). These can be hosted on separate environments (e.g., different servers or cloud instances) or in the same environment (e.g. two different processes running on the same server). Switching between versions is done by simply switching in the router (in less than a second).

Problems can however be caused by databases, because the new version may require a different database schema. To avoid this, during the migration the application is set to read-only.

Canary Releasing

In this strategy, the new version of the application is gradually rolled out to a small subset of users (the canary group) while the majority of users continue to use the old version. This allows for monitoring the performance and stability of the new version in a real-world environment before fully deploying it to all users. If any issues are detected, the deployment can be halted or rolled back without affecting the entire user base. It also allows to gather information about the new version from real users (e.g. if it generates more revenue).

4.1.2. Emergency fixes

Despite the amount of testing and precautions taken, it is possible that some bugs or vulnerabilities are discovered after deployment. In such cases, the fixes should also go through the deployment pipeline to ensure that they are properly tested and validated before being released to production. This is usually not done, just fixing the issue directly in production, but this can lead to:

- Introducing new bugs while fixing the issue, known as Regression Bugs.
- The system could be in an unknown state after the fix (e.g., inconsistent data), as it was not tested or committed properly.

Therefore, having short deployment times is even more important. In addition, when a bug is detected the severity of the issue should be evaluated, and rolling back to a previous version should also be considered.

4.2. Deployment Pipeline

A Deployment Pipeline is an automated process that takes code changes from development to production. Before analysing its phases, it is important to explain some good practices that should be followed when implementing a deployment pipeline.

4.2.1. Guidelines for a Deployment Pipeline

Some guidelines to follow when implementing a deployment pipeline are the following:

1. Binary files should only be built once and then promoted through the different stages of the pipeline (e.g., from testing to staging to production). They should be secured with hashes.

2. The deployment process should be similar in every environment (development, testing, staging, production).

3. Smoke-Tests which verify that the application, database, and external services are running correctly should be performed after each deployment.

4. Testing environments should closely resemble the production environment.

5. Each code change should go through the entire pipeline to avoid regression bugs.

6. If any phase in the pipeline fails, the entire process should stop and the team should address the issue immediately.

4.2.2. Phases of a Deployment Pipeline

A typical deployment pipeline consists of the following phases:

1. Commit Stage.

2. Automated Acceptance Test Stage.

3. Manual Test Stage.

4. Release Stage.

Commit Stage

In this phase, where the code is builded and some automated tests are performed (usually unit tests and some acceptance tests), the principles of Continuous Integration (CI) are applied. It should ideally last 5-10 minutes. This phase is crucial, and its implementation leads to significant improvements in software quality and team productivity.

Automated Acceptance Test Stage

The unit tests are usually not enough, and therefore more extensive automated acceptance tests should be performed in this phase. They should be described without technical details or terms, but from the user's perspective. They have an specific structure:

- **Given** some initial context (the preconditions).
- **When** an event occurs (the user action).
- **Then** ensure some outcomes (the postconditions).

Manual Test Stage

Some tests cannot be automated, and therefore they should be performed manually in this phase. Examples of such tests are the following:

- Look & Feel Testing: Ensures that the application meets the desired aesthetic and usability standards.
- Worst Case Testing: Evaluates the application's performance and stability under extreme conditions (e.g., for instance, the application is closed while performing a critical operation).

Release Stage

In this final phase, the application is deployed to the production environment. It should be as automated and as easy as possible, to minimize human errors and ensure consistency across deployments. Monitoring and alerting mechanisms should be in place to detect any issues that may arise after deployment.

4.2.3. Deployment of User-Installed Software

This process differs from the one described above, as the software is installed and updated by the users themselves. Some important aspects to consider are the following:

- Crash Reporting from the users should be implemented.
- Roll back should be also possible.
- Mantaining old versions is time-consuming, so ideally everyone should have the same version. In order to achieve that, updates should be downloaded and installed in the background automatically.

4.2.4. Modern Deployment Practices

In these last years, new practices have emerged to improve the deployment process even further. Some of these practices are the following:

- Progressive Delivery: This practice involves gradually (1 %, 5 %, 25 %, 50 %, 100 %) rolling out new features to users, allowing for monitoring and feedback at each stage before a full release. Canary Releasing and Blue-Green Deployment are combined, and the deployment is automatically paused or rolled back according to the monitoring results.

- Feature Flags: This technique allows developers to enable or disable specific features in an application without deploying new code. It allows dark launches (where a feature is deployed but not yet visible to users) and A/B testing (where different users see different versions of a feature to evaluate its performance) while minimizing risks. The *Flag debt* (the accumulation of unused or outdated feature flags) should be managed properly to avoid code complexity. There are some variations:

  - Release Flags: Used to control the release of new features.
  - Kill Switches: Used to quickly disable a feature in case of issues.
  - Permission Flags: Used to enable features for specific user groups.
  - Experiment Flags: Used for A/B testing.

- GitOps: This practice is based on the fact that the entire system's desired state is stored in a Git repository (*Git is the single source of truth*). Instead of deploying (manually or automatically) the application, the server directly pulls the changes from the Git repository and deploys them.

- Monitory and Observability: Appart from the typical monitoring (tracking predefined metrics as CPU, RAM, etc.), observability focuses on understanding the internal state of the system based on the data it produces (logs, metrics, traces). This allows not only to understand that an error has occurred, but also why it has occurred and, hopefully, how to fix it.

4.3. Continuous Deployment (CD)

Continuous Deployment (CD) is a software development practice that let's the software to be constantly deployed to production automatically, without human intervention. In order to achieve CD, CI is required, and a robust deployment pipeline with extensive automated testing is essential to ensure that only high-quality code reaches production. As happened with CI, CD prefers the changes to be small and incremental, as they are easier to test and deploy.

Some of its benefits include an increased reliability, fast deployment times and major competitiveness, as time is usually a critical factor in the software industry.

4.3.1. Continuous Delivery

Although CD is useful, it focuses on deploying every change to production, which may not be suitable or desired in all scenarios. Therefore, Continuous Delivery is often preferred, where software sould always be in a deployable state, but the actual

deployment to production is a manual decision. This allows for more control over when and how changes are released, allowing Feature Toggles to be used to enable or disable features as needed.

Observación. Both CD and Continuous Delivery focus on automating the deployment process on the production environment, while CI focuses on automating the build and testing processes in the earlier stages of development, in a test environment.

4.4. DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the development lifecycle and provide continuous delivery with high software quality. It aims to improve collaboration and communication between development and operations teams[1].

This approach is needed, because both teams have different goals and viewpoints.

- **Development Team**: Focuses on delivering new features and updates quickly to meet user needs and stay competitive in the market.
- **Operations Team**: Prioritizes system stability, reliability, security...

DevOps practices are based on the following principles:

- **Two Pizza Theory**: Teams should be small enough to be fed with two pizzas (8-10 people), to enhance communication and collaboration. This is not always possible, and it should also be taken into account that too many small teams can lead to coordination issues.
- **Experts Silos are eliminated**: Instead of having separate teams for development, testing, deployment, and operations, cross-functional teams are formed where members have diverse skills and responsibilities. This promotes collaboration and shared ownership of the entire software lifecycle. This way, bottlenecks caused by waiting for expert teams to perform specific tasks are avoided.
- **Avoiding Volleyball Games**: In traditional development processes, tasks are often passed between the development and operations teams, blaming each other for issues. In DevOps, the focus is on collaboration and shared responsibility, avoiding this back-and-forth blaming.
- **Employees are trusted and empowered**: Team members are given the autonomy to make decisions and take ownership of their work, not having to ask for permission for every little change and avoiding delays.

[1]This is usually used to justify bad practices, as letting developers manage the production environment.

4.5. Deployment with Containers Technology

During the last years, containers have become a popular technology for deploying applications. A *container* is a process that runs in a host operating system, isolated from other processes and containers, with its own filesystem, network interfaces, and resource limits. In a container, the application and its dependencies are packaged together, ensuring that it runs consistently across different environments. This increases the portability of applications and helps developping software, as no Virtual Machines (VMs) are needed. Some of the most popular containerization platforms are Docker and Kubernetes.

4.5.1. Isolation Measures

In this section, topics as `chroot`, namespaces, or `cgroups` will be briefly explained, as they are the basis of containerization technology.

`chroot`

The Unix `chroot` command changes the apparent root directory for the current running process and its children. This creates a confined space (a *chroot jail*) where the process can operate, isolating it from the rest of the file system. However, it is not a complete isolation mechanism, as they can still see all the processes running in the host system, no network isolation is provided, and if the process has root privileges, it can escape the jail.

Namespaces

Namespaces are a feature of the Linux kernel that provides isolation for various system resources. There are several types of namespaces, each isolating a specific resource:

- **PID Namespace**: Isolates process IDs, allowing processes in different namespaces to have the same PID. A child namespace will have its PID in its father namespace and a different one in its own namespace.

Control groups - `cgroups`

Control Groups (`cgroups`) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. It is needed in a containerization environment to ensure that containers do not consume more resources than allocated, which could lead to performance degradation or system instability.

4.5.2. Docker

Docker is a popular containerization platform that simplifies the process of creating, deploying, and managing containers. It has become a standard tool in the DevOps world due to its ease of use and powerful features. Some aspects are different in the integration and deployment phases when using Docker:

- After the CI server, an app would normally just be deployed as explained in this chapter.
- With Docker, after the CI server builds and tests the application, it creates a Docker image that is pushed to a Docker registry (e.g., Docker Hub, Amazon ECR). In the deployment phase, the Docker image is pulled from the registry and run as a container in the target environment.

5. Secure Deployment and CA Case Study

As it has already been explained in the previous chapters, testing is a key part of the software development lifecycle. However, it is useless if an atacker can easily compromise the deployed application. Therefore, it is essential to ensure that the deployment process is secure and that the application is protected against common threats. There are two main types of threats that need to be considered:

- **Malicious Adversaries:** They are whether malicious insiders or external attackers that impersonate legitimate users to gain unauthorized access to the system. They are indeed malicious.
- **Benign Insiders:** They are legitimate users that may unintentionally compromise the system's security due to lack of knowledge or carelessness. They are not malicious, but their actions can still pose a threat to the system.

There are some best practices that can be followed to ensure a secure deployment:

- Code Reviews: The code should be reviewed by multiple developers to ensure that it is secure and free of vulnerabilities. It also helps to share the knowledge and improve the overall quality of the code. Moreover, the concept "treat configuration as code" should be applied, so configuration files are also reviewed.
- Secrets: Secrets such as passwords, cryptographic keys, and authorization tokens should be stored securely using Key Management Systems (KMS), and should never be hardcoded in the source code or pushed to version control systems.
- Automatization: It should be done in order to remove human error from the deployment process. It also improves security, as helps avoiding attackers to introduced malicious code during the deployment.
- Builds: The process of building the application should have three main properties:
    - Hermetic: All the inputs (source code, compiler, libraries, etc.) should be specified and controlled. The external dependencies should be fetched from trusted sources and should be versioned and hashed to ensure their integrity.

- Reproducible: The same inputs should always produce the same (bit by bit)outputs. Hermetic builds help achieving this property, and it helps verifying the origins of the deployed code.
- Verifiable: The origin of the build should be verifiable, ensuring that it was built from the intended source code.

5.1. Binary Provenance

In order to achieve builds with these properties, the concept of "binary provenance" is used. It lets to trace back the binary to its source code, build process, and environment. A first aproach to achieve this is:

- There is a build system that, after building the application, produces a *build recipe* that contains all the information needed to reproduce the build, including the source code version, compiler version, build flags, and dependencies. The build recipe is then signed with a private key to ensure its authenticity and integrity, and the output is both the binary and the signed build recipe.

However, in some organizations the build system may execute all types of commands, so attackers could exploit this to introduce malicious code during the build process. To mitigate this risk, user-commands should be executed in an environment with limited privileges (no access to the keys) and a secured HTTP connection should be used to avoid man-in-the-middle attacks. Therefore, a more robust aproach to achieve binary provenance is:

- The build system is divided into two parts: a *orchestrator* and a *worker*. The orchestrator is responsible for issuing a top-level command to the worker, which is in charge of executing the build commands. The worker outputs the binary data and returns the artifact identifier to the orchestrator, which then produces the signed build recipe (if the artifact identifier matches the expected one).

5.2. Certificate Authorities (CA)

For asymetric cryptography, a proof of the authenticity of the public key is needed. This is done through certificates, which are mode of:

- The Public key of the entity with its identity information.
- All that signed by a trusted third party, called Certificate Authority (CA).

The public key of the CA is widely distributed and trusted by all parties, and are usually pre-installed in web browsers and operating systems. A PKI (Public Key Infrastructure) is a system that manages the issuance, revocation, and validation of digital certificates.

5.2.1. CA Creation

Google wanted to have their own CA to issue certificates for their internal services. That way, they did not have to rely on external CAs, which could be compromised or unavailable. They also decided to use their own software because of the flexibility and control it provided. Even though they obviously had to use third-party libraries for some parts, they tested them thoroughly and made sure they were secure.

Programming Languages

They used two programming languages:

- Go: It is memory-safe, important to work with unknown inputs such as certificate signing requests.
- C++: offers more interoperability with existing Google infrastructure, and offers a sandboxed environment to run untrusted code.
- They were both used because of performance reasons and the amount of good and safe libraries available.

Complexity vs Understandability and Ease of Use

Most comercial CA are really complex, as they have to offer a lot of features and support a wide range of use cases. However, Google wanted to have a CA with the minimum set of features needed for their internal use cases, so they could have a better understanding of the system and make it easier to use and maintain. It is continously improved, as they realised that they initially used too many microservices.

Security of their Private Keys

A great risk for a CA is the compromise of its private keys, as it would allow an attacker to issue fraudulent certificates. To mitigate this risk, Google uses Hardware Security Modules (HSM) to store their private keys. HSMs are tamper-resistant devices that provide a secure environment for key storage and cryptographic operations. They also use multi-factor authentication and strict access controls to limit access to the HSMs, which are offline most of the time to prevent remote attacks. Intermediate Keys are also used, so the root key is only used to sign the intermediate keys, which are then used to sign the certificates. This way, if an intermediate key is compromised, the root key remains secure.

5.3. Human Factors in Secure Deployment

In order to avoid human errors during the deployment process, there are some general guidelines available in internet:

- **NIST**: National Institute of Standards and Technology (NIST) provides the "Secure Software Development Framework", which includes guidelines for secure deployment and references to other relevant standards.
- **OWASP SAMM**: Open Web Application Security Project (OWASP) provides the "Software Assurance Maturity Model" (SAMM), which are some open-source guidelines for secure software development, including deployment. They propose that a company should be divided into 5 business functions:
  - Governance: Strategy and metrics.
  - Design: Threat modeling and secure architecture.
  - Implementation: Secure coding and code review.
  - Verification: Security testing and vulnerability management.
  - Operations: Incident detection and response.
- **BSIMM**: Building Security In Maturity Model (BSIMM) provides anual reports of security activities and trends. It is based on the observation of around 130 companies.

5.3.1. Vulnerability Management

Vulnerability management is the process of identifying, assessing, and mitigating vulnerabilities in software applications. To do so, it should be noted that it is impossible to fix all vulnerabilities at once, so they should be prioritized based on their severity and impact. This can be beasured by several metrics (or even combining them):

- **CVSS**, Common Vulnerability Scoring System.
- **EPSS**, Exploit Prediction Scoring System.
- Known Exploited Vulnerabilities (KEV) catalog from CISA.
- Business impact / Asset criticality.

5.3.2. Security Champion

When developing secure software, a common problem is the lack of organization. A common solution to this problem is to have a *security champion* in each development team. In addition, all security champions should form a community to share knowledge and best practices, so the whole company is improved. A security champion should at least have the following responsibilities:

- Being the source of security knowledge in the team. They should increase security awareness and promote best practices.
- Identify security risks and vulnerabilities in the team's code and processes.
- Review and escalation.

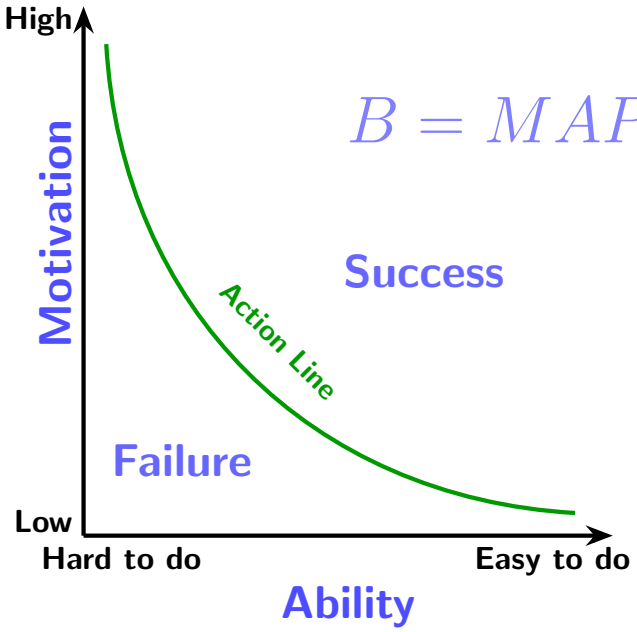

Figura 5.1: Fogg Behavior Model

However, security champions must also be organized and supported by the company, as they cannot do everything alone. Some of the problems that may arise are:

- Shift in responsibilities: Developers may think that the security champion is responsible for all security-related tasks, leading to a lack of ownership and accountability among team members.
- Lack of time: Security champions may struggle to balance their security responsibilities with their regular development tasks, leading to burnout and decreased effectiveness.
- Insufficient training: Security champions may not have the necessary skills or knowledge to effectively identify and mitigate security risks. They all wish they had a security team backing them up.
- Selection: Choosing the right person for the role is crucial, and sometimes no one from the team wants to take the responsibility.
- Security Champion Skills: They need to have both technical and soft skills, such as communication and leadership. The Fogg Behavior Model (figure 5.1) explains that only when the three factors (motivation, ability and trigger) are present, a behavior will occur.
- Communication with PM: They need to effectively communicate security issues and risks to project managers and other stakeholders, as security is not always a priority for them.

In order to address these challenges, in the OWASP Security Champions Guide there is a manifesto that outlines the principles and values that security champions should uphold:

- Be passionate about security.
- Start with a clear vision.
- Secure management support.
- Nominate a dedicated captain.
- Trust your champions.
- Create a community.
- Promote knowledge sharing.
- Reward responsibility.
- Invest in your champions.
- Anticipate personnel changes.

As previously explained, Security Champions should promote knowledge sharing within the organization. With that aim is the OWASP Juice Shop, which "is probably the most modern and sophisticated insecure web application". It is an intentionally insecure web application for security training purposes, as it can be used in security trainings or awareness demos.

6. Software Testing

As discussed in previous chapters, testing is a crucial aspect of software development and maintenance. It is known that in complex systems, usually more time is needed to write the tests than to write the actual code. However, even with the best coding practices, bugs and issues are inevitable. Unexpected inputs may lead to:

- Confidentiality Violations: Unauthorized access to sensitive data.
- Integrity Violations: Unauthorized modification of data.
- Availability Violations: Disruption of service.

6.1. Test Types

There are two main categories of tests: unit tests and acceptance tests. In this section, we will explore these types of tests in detail.

6.1.1. Unit Tests

An unit test is a type of software test that focuses on verifying the functionality of a specific section of code, typically at the function or method level. The main goal of unit testing is to ensure that individual components of the software work as intended in isolation, without dependencies on other parts of the system. There are a lot of tools for unit testing, as `xUnit` or `GoogleTest`. There are two main approaches to develop the unit tests:

- Classic Approach: Write the code first, then create unit tests to verify its functionality. Code and tests should be commited together, and when the code is reviewed, the tests should be reviewed as well.
- Test-Driven Development (TDD): Write the unit tests before writing the actual code. The tests will fail until the code is implemented correctly. This approach encourages developers to think about the requirements and design of the code before implementation.

It should also be noted that unit tests may sometimes require refactoring of the code to make it more testable. This can lead to better code quality and maintainability.

6.1.2. Acceptance Tests

Acceptance tests, also known as end-to-end tests or functional tests, are designed to verify that a software application meets the specified requirements and behaves as expected from the user's perspective. These tests focus on validating the overall functionality of the system, ensuring that all components work together seamlessly to deliver the desired user experience. They usually need much more time to be developed and run than unit tests, but they are crucial to ensure that the software meets the user's needs. If an acceptance test fails but all of the unit tests pass, the error is usually difficult to locate.

Regarding the acceptance tests, it should be noted that *flakiness* is more common than in unit tests. A flaky test is a test that can pass or fail non-deterministically, without any changes to the code being tested. This can be due to various factors, such as timing issues, external dependencies, or environmental factors. Flaky tests can lead to false positives or false negatives, making it difficult to determine the actual state of the software. Therefore, it is important to identify and address flaky tests to ensure the reliability of the testing process.

6.2. Program Analysis

Analyzing code can help to identify potential issues and improve code quality. There are two main types of program analysis: static analysis (analyzing code without executing it) and dynamic analysis (analyzing code during execution).

When analyzing code, both the source and the binary code can be considered. There are two aspects that should be taken into account:

- Dynamic Binary Instrumentation (DBI): Using kind of a virtual machine to analyze the binary code during execution. Examples of DBI tools are `Valgrind` or `Intel Pin`.
- Dynamic Analysis based on compiler support: The compiler inserts additional code to perform the analysis during execution. They are often required to detect memory errors.

In order to analyze code during execution, it is common to use `sanitizers`, which are tools that detect various types of errors at runtime. A really common sanitizer is `Address Sanitizer`, which is described in the next section.

6.2.1. Adress Sanitizer

`Address Sanitizer (ASan)` is a fast memory error detector. It usually detects:

- Use-after-free: Accessing memory after it has been freed.
- Out-of-bounds access: Accessing memory outside the allocated bounds.

ASan uses a technique called *shadow memory* to keep track of the state of each byte of memory. For each 8 bytes of application memory, ASan maintains 1 byte of

shadow memory. The shadow memory is used to store metadata about the state of the corresponding application memory. When a programm is compiled with ASan, additional instrumentation code is added with two main aims:

- Before every memory access, ASan checks the corresponding shadow memory to determine if that memory adress is "poisoned" (i.e., invalid or unsafe to access).
- When memory is allocated, 32 bytes of "red zones" are added before and after the allocated memory to detect out-of-bounds accesses.

An important aspect to consider when using ASan is that it increases both memory usage and execution time. Typically, ASan increases memory usage by about 2-3 times and slows down program execution by a factor of 2-3. Given that overhead, ASan is primarily used during development and testing phases rather than in production environments.

6.3. The Quest for Coverage

The goal of testing is to cover as much code as possible, in order to detect potential bugs and issues. This should be done with the minimum number of test cases, to reduce the time and effort needed to run the tests. Typically, one test case explores one path through the program. In order to achieve this goal, several techniques can be used, such as symbolic execution and fuzzing.

6.3.1. Symbolic Execution

Symbolic execution is a program analysis technique that explores program paths by treating input values as symbolic variables rather than concrete values. This allows the analysis to reason about multiple execution paths simultaneously, enabling the detection of potential bugs and vulnerabilities that may not be easily discovered through traditional testing methods.

When the symbolic execution engine finishes, all the possible paths through the program have been explored, and a set of path constraints has been generated for each path. These path constraints can be used to generate test inputs that will exercise specific paths through the program, helping to achieve better code coverage and identify potential issues.

However, symbolic execution has some limitations in practice, as systems can be really complex:

- Path Explosion: The number of possible execution paths can grow exponentially with the size of the program, making it infeasible to explore all paths.
- Handling of External Dependencies: Symbolic execution may struggle to accurately model interactions with external libraries, system calls, hardware components, or user inputs.

Therefore, the scope is usually limited to small and critical parts of the code, or it is combined with other techniques, such as fuzzing.

6.3.2. Fuzzing

Fuzzing is an automated software testing technique that involves providing different types of inputs to a program in order to identify potential bugs, vulnerabilities, or unexpected behavior. The main goal of fuzzing is to explore the program's input space and uncover edge cases that may not have been considered during development. There are several types of fuzzing techniques, including the following:

- Random Fuzzing: Randomly generates inputs without any specific knowledge of the program's structure or behavior. The inputs for the following test cases are also generated randomly. This technique is simple to implement but may not be very effective in finding deep or complex bugs.
- Mutation-based Fuzzing: Starts with a set of valid inputs, and the inputs for the following test cases are generated by making small modifications (mutations) to these valid inputs.
- Cover-guided Fuzzing: Starts with a set of initial inputs and, after running each test case, it analyzes the code coverage achieved. If new paths are discovered, the inputs that led to those paths are mutated to generate new test cases, and if no new paths are found, that input is discarded. This technique is more effective in exploring the program's input space and finding bugs.

7. Übungen

7.1. Application Lifecycle Management (ALM)

Ejercicio 7.1.1.

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

1. Agile Entwicklung

2. Scrum

3. Wasserfall-Modell

4. DevOps-Ansatz

Ejercicio 7.1.3. 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 7.1.4. Ihr Team entwickelt weiterhin die Echtzeit-Messaging-App. Beschreiben Sie, wie Scrum den Entwicklungsprozess strukturiert.

```
$ git status
On branch cool_stuff
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   text.py

no changes added to commit (use "git add" and/or "git commit -a")
$ git add text.py
$ git commit -m "New Feature"
[cool_stuff 00c5b2a] New Feature
 1 file changed, 3 insertions(+), 2 deletions(-)
```

Código fuente 2: Lösung zu Übung 7.2.2.1

7.2. Version Control System (VCS)

Ejercicio 7.2.1. 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.

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.2.2.

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.

   The code needed to solve this exercise is shown in Listing 2.

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.

   The code needed to solve this exercise is shown in Listing 3.

Ejercicio 7.2.3. 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.

At first it is needed to execute the code shown in Listing 4. After manually solving the conflict, the code shown in Listing 5 is executed.

```
$ git branch
* cool_stuff
  master
$ git checkout master
Switched to branch 'master'
$ git merge cool_stuff
Merge made by the 'ort' strategy.
 text.py | 12 ++++++++++-
 1 file changed, 11 insertions(+), 1 deletion(-)
```

Código fuente 3: Lösung zu Übung 7.2.2.2

```
$ git branch
  master
* other_cool_stuff
$ git checkout master
Switched to branch 'master'
$ git merge other_cool_stuff
Auto-merging text.py
CONFLICT (content): Merge conflict in text.py
Automatic merge failed; fix conflicts and then commit the result.
```

Código fuente 4: Erster Teil der Lösung zu Übung 7.2.3.

```
$ git add text.py
$ git commit
[master 9abac85] Merge branch 'other_cool_stuff'
```

Código fuente 5: Zweiter Teil der Lösung zu Übung 7.2.3.

```
#!/bin/sh

git diff --cached --name-only --diff-filter=A \
| while read -r name; do
    git rm --cached "$name" > /dev/null 2> /dev/null
done

exit 0
```

Código fuente 6: `pre-commit` Hook that prevents committing new files.

Ejercicio 7.2.4. 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.

The problem here lies in the fact that there is a `pre-commit` hook, the shown in Listing 6, that prevents committing new files. The solution is just to delete that hook from the `.git/hooks` directory.

```
$ git log --graph --oneline --all
* e777451 (HEAD -> cool_stuff) New Commit
* fbdb062 new motivating phrases
| * a5fdf5d (master) updated main.py
|/
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 7: Git-Graph nach dem Committen der Änderungen im `cool_stuff`-Branch.

```
$ git rebase master
Successfully rebased and updated refs/heads/cool_stuff.
$ git checkout master ; git merge cool_stuff
Switched to branch 'master'
Updating a5fdf5d..baae187
Fast-forward
 text.py | 12 ++++++++++-
 1 file changed, 11 insertions(+), 1 deletion(-)
```

Código fuente 8: Lösung zu Übung 7.2.2.2 mit Rebase.

7.3. Distributed Git und Internals

Ejercicio 7.3.1. In Aufgaben 7.2.2.2 und 7.2.3 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.

1. Aufgabe 7.2.2.2

   After committing the new changes to the `cool_stuff`-Branch, the git graph is the one shown in Listing 7. The other proposed solution is to rebase the `cool_stuff`-Branch on top of the `master`-Branch. The code needed to do that is shown in Listing 8. The result of the rebase is a linear history, which can be seen in the git graph shown in Listing 9.

```
$ git log --graph --oneline --all
* baae187 (HEAD -> master, cool_stuff) New Commit
* 9c087cd new motivating phrases
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 9: Git-Graph nach dem Rebase des `cool_stuff`-Branches auf den `master`-Branch.

```
$ git log --graph --oneline --all
* ff02d9a (HEAD -> other_cool_stuff) fixed unterminated string literal
and added text2 to main
* 13061d3 new text2
| * 07216d8 (master) finalized cool stuff
| * 36e8466 new motivating phrases
|/
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 10: Git-Graph vor dem Rebase des `other_cool_stuff`-Branches auf den `master`-Branch.

```
$ git add text.py
$ git rebase --continue
[detached HEAD fe644de] new text2
 1 file changed, 4 insertions(+), 9 deletions(-)
Successfully rebased and updated refs/heads/other_cool_stuff.
$ git checkout master ; git merge other_cool_stuff
Switched to branch 'master'
Updating 07216d8..7027614
Fast-forward
 main.py |  4 ++--
 text.py | 13 ++++---------
 2 files changed, 6 insertions(+), 11 deletions(-)
```

Código fuente 11: Lösung zu Übung 7.2.3 mit Rebase.

2. Aufgabe 7.2.3

   The initial git graph is the one shown in Listing 10. When trying to rebase the `other_cool_stuff`-Branch on top of the `master`-Branch, a conflict is generated. After manually solving the conflict, the code shown in Listing 11 is executed. The result of the rebase is a linear history, which can be seen in the git graph shown in Listing 12.

Ejercicio 7.3.2. 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 7.3.3. Der Chef des Teams, zuständig für die Entwicklung der Echtzeit-Messaging-App hat wenig Ahnung von Softwareentwicklung. Er hat damals einfach

```
$ git log --graph --oneline --all
* 7027614 (HEAD -> master, other_cool_stuff) fixed unterminated string
literal and added text2 to main
* fe644de new text2
* 07216d8 finalized cool stuff
* 36e8466 new motivating phrases
* a5fdf5d updated main.py
* 41da045 added new text generation
* 87c476b initial commit
```

Código fuente 12: Git-Graph nach dem Rebase des `other_cool_stuff`-Branches auf den `master`-Branch.

```
$ git fsck --lost-found
Checking object directories: 100% (256/256), done.
Checking objects: 100% (28/28), done.
dangling blob acbc45bdb82b84a3df80a69659ad672c2791f632
Verifying commits in commit graph: 100% (8/8), done.
$ git cat-file -p acbc > lost.py
$ git add lost.py ; git commit -m "File recovered"
[master c00c6ec] File recovered
 1 file changed, 16 insertions(+)
 create mode 100644 lost.py
```

Código fuente 13: Lösung zu Übung 7.3.4 mit dem Befehl `git fsck --lost-found`.

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.

Zu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.3.4. 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.

The solution to this exercise involves using the Git command `git fsck --lost-found` to recover lost objects. The solution is shown in Listing 13.

Ejercicio 7.3.5. Nehmen Sie an, Sie haben ein Git-Repository, welches die in Abbildung 7.1 dargestellte Commit-Graphen hat. Nehmen Sie nun an, Sie führen die

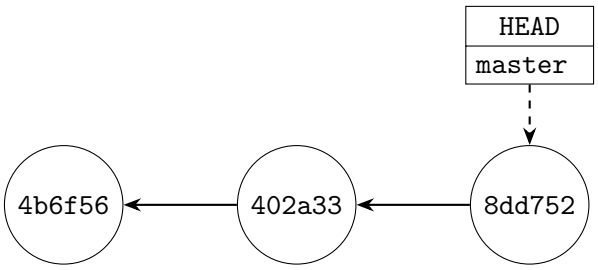

Figura 7.1: Git-Repository mit drei Commits und einem Branch `master`.

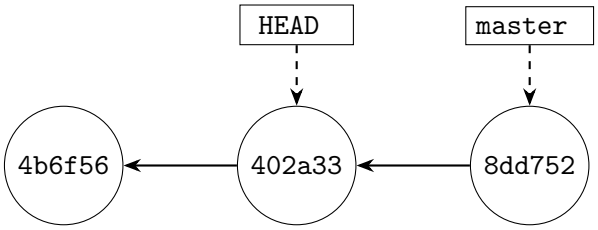

Figura 7.2: Git-Graph after the command of the Exercise 7.3.5.1.

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`

   The result of this command is shown in the git graph in Figure 7.2.

2. `git checkout -b 'feature_branch'`

   The result of this command is shown in the git graph in Figure 7.3.

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

   The result of this command is shown in the git graph in Figure 7.4.

4. Zeichnen Sie bitte beide Graphen

   a) `git merge master`
   The result of this command is shown in the git graph in Figure 7.5.

   b) `git rebase master`
   The result of this command is shown in the git graph in Figure 7.6.

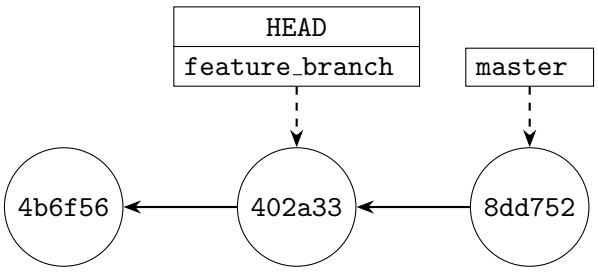

Figura 7.3: Git-Graph after the command of the Exercise 7.3.5.2.

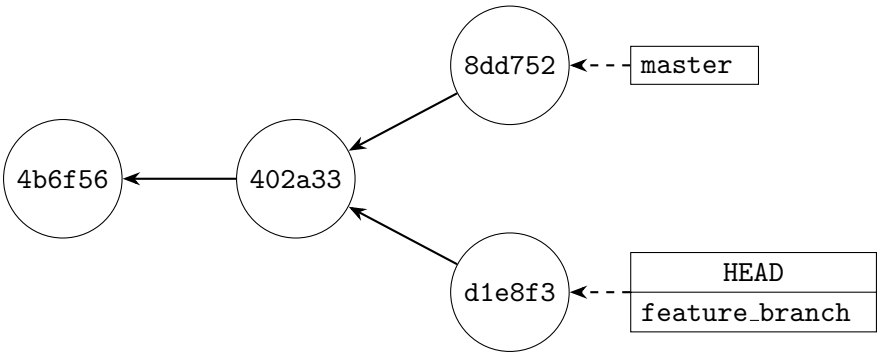

Figura 7.4: Git-Graph after the command of the Exercise 7.3.5.3.

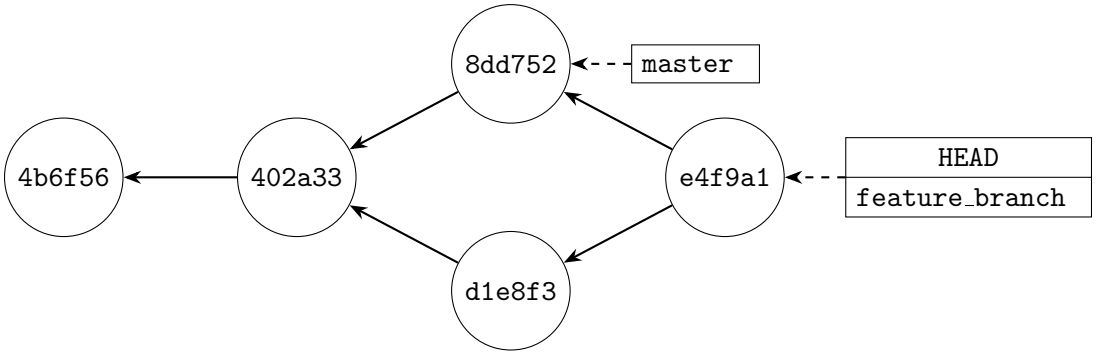

Figura 7.5: Git-Graph after the command of the Exercise 7.3.5.4*a*.

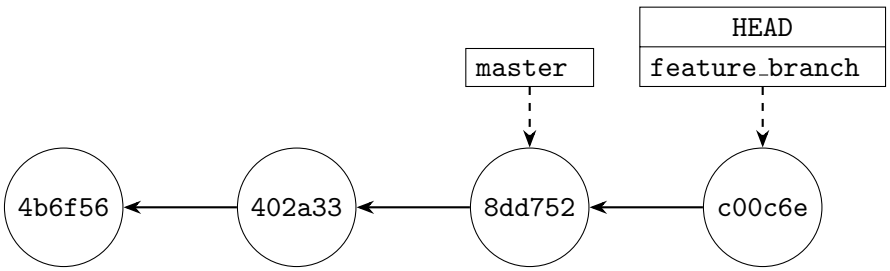

Figura 7.6: Git-Graph after the command of the Exercise 7.3.5.4*b*.

```
1 $ git checkout master ; git merge feature_branch ; git branch -d
  feature_branch
```

Código fuente 14: Command to merge the `feature_branch` into the `master` branch and delete the `feature_branch` for the graph in Figure 7.5.

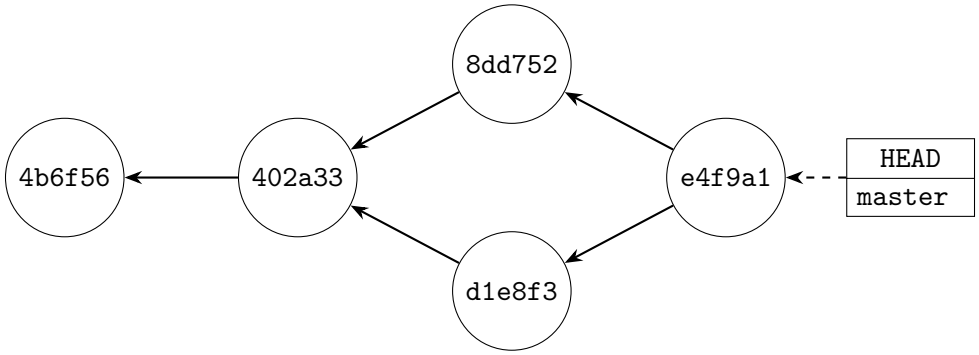

Figura 7.7: Git-Graph after merging the `feature_branch` into the `master` branch and deleting the `feature_branch` for the graph in Figure 7.5.

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

   In both cases, the command needed to merge the `feature_branch` into the `master` branch and delete the `feature_branch` is shown in Listing 14.

   a) For the graph in Figure 7.5, after applying the command in Listing 14, the resulting graph is the one shown in Figure 7.7.

   b) For the graph in Figure 7.6, after applying the command in Listing 14, the resulting graph is the one shown in Figure 7.8.

Ejercicio 7.3.6.

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.

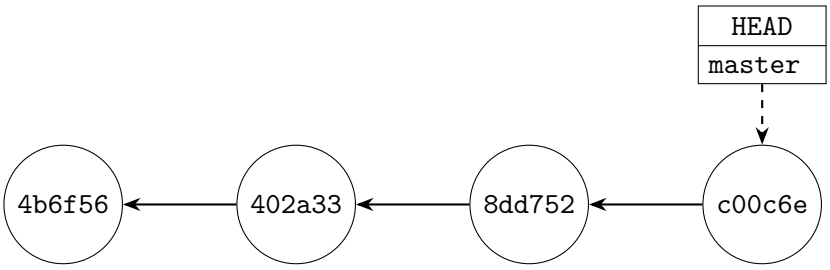

Figura 7.8: Git-Graph after merging the `feature_branch` into the `master` branch and deleting the `feature_branch` for the graph in Figure 7.6.

```
$ git add text.py
$ git commit --amend --no-edit
[other_cool_stuff e9274fd] new text2
 Date: Thu Oct 26 14:38:02 2023 +0200
 2 files changed, 8 insertions(+), 3 deletions(-)
```

Código fuente 15: Lösung zu Übung 7.3.6.1.

```
$ git commit --amend -m "new text2 and added it to main"
[other_cool_stuff 9c087cd] new text2 and added it to main
 Date: Thu Oct 26 14:38:02 2023 +0200
 2 files changed, 8 insertions(+), 3 deletions(-)
```

Código fuente 16: Lösung zu Übung 7.3.6.2.

The command needed to add the change to the last commit is shown in Listing 15.

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.

   The command needed to change the commit message of the last commit is shown in Listing 16.

```
$ git bisect start
status: waiting for both good and bad commits
$ git bisect bad
status: waiting for good commit(s), bad commit known
$ git bisect good b6763c2
Bisecting: 184 revisions left to test after this (roughly 8 steps)
[45d0499cacc9082b426292f53b63e24f5cb87a1e] Commit 215
```

Código fuente 17: Starting the bisecting process

7.4. Continuous Integration

ZuZu den folgenden Aufgaben finden Sie hier ein `zip`-Datei mit den notwendigen Ressourcen.

Ejercicio 7.4.1. Ihre Tests zeigen, dass eine Funktion nicht mehr korrekt funktioniert. Identifizieren Sie den Commit, ab dem der Fehler eingeführt wurde.

As we can see when executing `get_pi.py`, the output is not correct. We will use `git bisect` to find the commit that introduced the error. First of all we need to detect a commit where the output is correct (a good commit). We will check out to some of the first commits and execute the script to check that the output is correct. In this case, commit `b6763c2` is a good commit. Then, the Listing 17 shows how to start the bisecting process by marking the current commit as bad and the good commit as good.

At this point, Git has automatically checked out to a commit in the middle of the history (in this case, commit `45d0499`). The checking process is shown in the Listing 18. After the checking process, we can see that the first bad commit is `46917b5` (commit 173), which is the commit that introduced the error.

Finally, we can see that the first bad commit is `46917b5` (commit 173), which is the commit that introduced the error. In the Listing 19 we can see the details of the bad commit.

The checking process can be automated using the `git bisect run <script>` command, where `<script>` is the shown in the Listing 20. That way, Git will automatically execute the script in each commit and determine if it is good or bad based on the exit code of the script (0 for good, 1 for bad), simplifying the bisecting process.

Ejercicio 7.4.2. 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 7.2 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?

```
            $ python3 get_pi.py
2.77
$ git bisect bad
Bisecting: 91 revisions left to test after this (roughly 7 steps)
[0ed8e90afea45388b98e7caa74475cf1da7ba614] Commit 123
$ python3 get_pi.py
3.14
$ git bisect good
Bisecting: 45 revisions left to test after this (roughly 6 steps)
[534ee36e8acfa1e8112b21c559cc4e390455113d] Commit 169
$ ...   # We continue the process by checking out to the next commit suggested
by Git and executing the script to check the output. We repeat this process
until we find the first bad commit.
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[53b336b7d7cbfc0069951d7cf1988b3c44c603f9] Commit 172
$ python3 get_pi.py
3.14
$ git bisect good
46917b552f8df592e2d86becbba6a26d7be1da36 is the first bad commit
commit 46917b552f8df592e2d86becbba6a26d7be1da36
Author: Alice <alice@example.com>
Date:   Mon Dec 9 18:25:28 2024 +0100

    Commit 173

 get_pi.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Código fuente 18: Bisecting process

```
$ git bisect reset
Previous HEAD position was 53b336b Commit 172
Switched to branch 'master'
$ git show 46917b5
commit 46917b552f8df592e2d86becbba6a26d7be1da36
Author: Alice <alice@example.com>
Date:   Mon Dec 9 18:25:28 2024 +0100

    Commit 173

diff --git a/get_pi.py b/get_pi.py
index 3325ae8..9b12bce 100644
--- a/get_pi.py
+++ b/get_pi.py
@@ -5,7 +5,7 @@ def berechne_pi(n_terms):
     for i in range(2, 2 + 2 * n_terms, 2):
         term = 4.0 / (i * (i + 1) * (i + 2))
         if add:
-            pi += term  # Berechnungsschritt 172
+            pi -= term  # Berechnungsschritt 172
         else:
             pi -= term  # Berechnungsschritt 172
         add = not add
```

Código fuente 19: Details of the bad commit

```
#!/bin/bash

RESULT=$(python3 get_pi.py)

if [ "$RESULT" == "3.14" ]; then
  exit 0
else
  exit 1
fi
```

Código fuente 20: Script to automate the bisecting process

4. Welche Unit-, Component- und Acceptance-Tests würden zur Messaging-App passen?

7.5. Docker

Ejercicio 7.5.1. In Moodle finden Sie eine ZIP-Datei, die ausführbare Dateien für verschiedene Architekturen enthält. Wenn Sie die Ubuntu-VM nutzen, ist die Datei `server_linux_x86_64` die richtige. Alle anderen Dateien sind ungetestet und nicht garantiert zu funktionieren. Sie können die Datei mit `$ ./server_linux_x86_64` ausführen und den Webserver im Browser unter `localhost:8080` erreichen. Ihre Aufgabe ist es, zwei Instanzen des Webservers parallel auf Ihrem System auszuführen.

Ejercicio 7.5.2. Sie sind Teil des Entwicklerteams für die Echtzeit-Messaging-App der „Universität der Zukunft“. Da Ihr Team in einer heterogenen Umgebung arbeitet und sicherstellen muss, dass das entwickelte Python-Skript unter verschiedenen Python-Versionen ordnungsgemäß ausgeführt wird, ist es Ihre Aufgabe, Docker-Container für diese Tests zu erstellen. Ihr Manager hat Sie gebeten, zu testen, ob die App mit allen offiziell unterstützten Python-Versionen ausführbar ist.

Ejercicio 7.5.3.

1. Was ist Docker und wie unterscheidet es sich von Hypervisor-basierten Virtualisierungstechnologien?
2. Erläutern Sie den Begriff “Container” im Kontext von Docker.
3. Wie kann Docker in einer Continuous-Integration/Continuous-Deployment-(CI/CD)-Pipeline eingesetzt werden?
4. Erläutern Sie den Begriff “Docker Registry” und erläutern Sie, warum er für CI/CD wichtig ist.

Ejercicio 7.5.4. In dieser Aufgabe lernen Sie einen der Grundmechanismen der *historischen* Containerisolierung kennen. Dazu verwenden Sie das UNIX-Werkzeug `chroot`, das den sichtbaren Root-Ordner eines Prozesses ändert. In der ZIP-Datei aus Moodle finden Sie die Datei `alpine-rootfs.tar`. Diese Datei enthält ein minimales Linux-Dateisystem, das ursprünglich aus einem Docker-Container (`alpine`) exportiert wurde.

1. Entpacken Sie das Root-Dateisystem in ein Verzeichnis Ihrer Wahl (z.B. `$HOME/alpine-rootfs`)
2. Starten Sie eine Shell innerhalb dieses Dateisystems mithilfe von `chroot`.
3. Untersuchen Sie das Verhalten innerhalb der Umgebung:
   - Führen Sie Befehle wie `ls`, `pwd` und `ps` aus. Was fällt Ihnen auf?
   - Starten Sie in einem separaten Terminal `ps -ef` aus. Können Sie den Prozess aus der chroot-Umgebung sehen?
4. Schreiben Sie ein kleines Programm/Script, das die Schritte des Entpackens sowie das Starten eines Befehls in der Chroot-Umgebung automatisiert.
5. Begründen Sie, warum `chroot` *keine vollständige Isolation* bietet.

7.6. Deployment

Ejercicio 7.6.1. Die *Echtzeit-Messaging-App* für den Campus der "Universität der Zukunft" soll bereitgestellt werden. Die App soll zunächst auf Android-Telefonen verfügbar gemacht werden. Es gibt außerdem einen Server, der Anfragen der App verarbeitet und Nachrichten speichert.

1. Wie sollte man das erste Deployment der App gestalten?
2. Eine neue Version der App ist fertig entwickelt. Lohnt sich ein Zero-Downtime-Release?
3. Was sollte man bei diesem Release beachten?
4. Wann sind Blue-Green Deployments sinnvoll?
5. Wann sind Canary Releases sinnvoll?
6. Was passiert in unserer App während der Commit Stage?
7. Was passiert in der Automated Acceptance Stage?
8. Was passiert in der manual Test Stage?
9. Was passiert in der Release Stage?
10. Würden Sie eher Continuous Deployment oder Continuous Delivery für das Projekt nutzen? Argumentieren Sie.

Ejercicio 7.6.2.

1. Warum lohnt es sich, Container in der Entwicklung und in der Deployment-Pipeline zu verwenden?
2. Welche Eigenschaften von cgroups sind bei der Containerization nützlich?
3. In ihrer Ubuntu-VM sollten sie unter `/sys/fs/cgroup/system.slice/docker.service` die Dateien finden die die "docker.service"-`cgroup` definieren. Schauen Sie sich `cpu.max` und `memory.max` an. Was sagen sie über diese `cgroup` aus?
4. Welche Eigenschaften haben Namespaces, die bei der Containerisierung nützlich sind?
5. Welche Eigenschaften hat `chroot` bzw. `privot_root`, die bei der Containerization nützlich ist, und was unterscheidet die beiden?
6. Was unterscheidet Containerization von Virtual Machines?

7.7. Secure Deployment

Ejercicio 7.7.1. Die *Echtzeit-Messaging-App* für den Campus der "Universität der Zukunft" soll regelmäßig aktualisiert werden, dabei aber sichere Deployments garantiert werden.

1. Erklären Sie kurz die Begriffe hermetic build, reproducible build und verifiable build.

2. Nennen Sie zwei typische Fallstricke, die reproducible Builds verhindern.

Ejercicio 7.7.2. Die Universität überlegt, eine eigene CA für interne Services einzurichten.

1. Erklären Sie, wofür eine CA benötigt wird, und welche Rolle Zertifikate beim sicheren Schlüsselaustausch spielen.

2. Skizzieren Sie den Ausstellungsprozess eines Zertifikats.

3. Nennen Sie drei Risiken beim Betrieb einer CA und mögliche Gegenmaßnahmen.

Ejercicio 7.7.3. Angenommen, ein Angreifer hat Zugriff auf einen CI-Runner erlangt, der Builds ausführt.

1. Welche zwei Angriffe sind dadurch besonders naheliegend?

2. Welche Sofortmaßnahme (erste 24h) würden Sie einleiten?

Ejercicio 7.7.4. Schauen Sie sich SLSA an. Worum handelt es sich dabei?

7.8. Secure Deployment 2

Ejercicio 7.8.1. Erläutern Sie, inwiefern die folgenden Maßnahmen vor einem Benign Insider oder einem Malicious Adversary absichern. Gegen welche der beiden Arten von Akteuren sind die Maßnahmen effektiver und warum?

1. Code Reviews
2. Geheimnisse schützen (durch Key-Management-Systeme, Zugriffskontrollen etc.)
3. Automatisierung der CI/CD-Pipeline
4. Verifiable Builds

Ejercicio 7.8.2.
1. Welche Daten sollten in einer Binary Provenance für unsere Echtzeit-Messaging-App des Campus der „Universität der Zukunft“ enthalten sein?
2. Welche Verbindung besteht zwischen Hermetic, Reproducible und Verifiable Builds und Binary Provenance?

7.9. Secure Development

Ejercicio 7.9.1.

1. Erläutern Sie, was ein Security Champion ist.
2. Welche Vor- und Nachteile hat dieses Konzept?

Ejercicio 7.9.2. Nehmen Sie wieder an, Sie seien Entwickler im Team der Echtzeit-Messaging-App der "Universität der Zukunft". Ihr Team denkt über das Thema Sicherheit nach und möchte das `OWASP SAMM` einführen.

1. Schauen Sie sich das `OWASP SAMM` an.
2. Überlegen Sie sich verschiedene Maßnahmen auf unterschiedlichen Ebenen, um eine sichere Benutzerauthentifizierung sicherzustellen. (Mögliche Hilfestellung kann das `OWASP CheatSheet` bieten.)
3. Überlegen Sie, zu welcher Domäne des `OWASP SAMM`s die Maßnahmen passen und inwiefern Sie den Reifegrad verbessern können.

```
def func(a, b):
    x = 0
    y = 0
    if a > 0:
        x = 1
    if b == 0:
        y = 2
    assert x + y != 3
```

Código fuente 21: Beispielcode für Übung 7.10.4

7.10. Fuzzing & Z3

Ejercicio 7.10.1. Schauen Sie sich das Fuzzingbook an. Sie können den benötigten Code mittels `$ pip install fuzzingbook` installieren. Beschreiben Sie die folgenden Ansätze und erklären Sie die Unterschiede sowie die Vor- und Nachteile.

1. Random Fuzzing
2. Mutation Fuzzing
3. Coverage-guided Fuzzing

Ejercicio 7.10.2.

1. Schauen Sie sich an, welche SMT-Solver es gibt.
2. Nennen Sie einige Anwendungsgebiete, in denen SMT-Solver eingesetzt werden können.
3. Welche Herausforderungen können bei der Anwendung von SMT-Solvern auftreten und wie können sie bewältigt werden?
4. Wie können SMT-Solver zur Sicherheitsanalyse von Softwareanwendungen beitragen?

Ejercicio 7.10.3.

1. Warum sind AddressSanitizer (ASan) nicht für den Produktivbetrieb geeignet?
2. Was sind Redzones?
3. Warum wird Shadow Memory benötigt?

Ejercicio 7.10.4. Schauen Sie sich das folgende Python-Programm an (Abbildung 21).

1. Zeichnen Sie den Kontrollflussgraphen.
2. Leite alle möglichen Pfadbedingungen her.
3. Gibt es eine Belegung von `a` und `b`, die das Assert verletzt? Wenn ja: Welche?

7.11. Fuzzing & Z3 2

Fur die nachsten Aufgaben müssen Sie ein paar Python-Pakete installieren. Sie finden `z3`-Beispiele in Moodle.

1. Öffnen Sie das Terminal (`ctrl + Alt + T`)
2. Installieren Sie den Package Installer for Python (`$ sudo apt install python3-pip python3`)
3. Erstellen Sie ein neues Virtual Environment (`$ python3 -m venv ./venv`)
4. Konfigurieren Sie die aktuelle Shell, damit sie das `venv` verwendet. (`$ source ./venv/bin/act`)
5. Installieren Sie `z3` (`$ pip3 install z3-solver`)
6. Atheris können Sie in einem Ubuntu 24.04 Container installieren (`$ pip3 install atheris`)

Ejercicio 7.11.1.

1. Nutzen Sie `atheris` und instrumentieren Sie die `validate`-Funktion in `fuzz.py`.
2. Zeichnen Sie einen Kontrollflussgraphen für die Funktion. Nutzen Sie als Knotennamen die Zeilen-nummern.
3. Bestimmen Sie die prozentuale Coverage, wenn der Code mit `[246,63,103,121]` aufgerufen wird. Markieren Sie außerdem die erreichten Knoten im Kontrollflussgraphen.
4. Instrumentieren Sie die `validate`-Funktion in `fuzz2.py`
5. Suchen Sie mittels `z3` nach einer Lösung für `validate`.

Ejercicio 7.11.2. Manche Programmierer nutzen Bit-Tricks, um auf spezifische Hardware zu optimieren. Nutzen Sie `z3`, um zu zeigen, dass die Tricks auf einem 64-Bit-System dasselbe Verhalten haben wie eine naive Implementierung.

1. Tauschen zweier Variablen mit XOR (Swapping Values with XOR)
2. Tauschen zweier Variablen mit Subtraktion und Addition (Swapping values with subtraction and addition)
3. Die Bitreihenfolge innerhalb eines Bytes umkehren (Reverse the bits in a byte with 4 operations)

Ejercicio 7.11.3. Nutzen Sie `z3`, um generische Sudokus zu lösen.

Observación. Sollten Sie nicht weiterkommen, kann eine Internetsuche weiterhelfen.

Ejercicio 7.11.4. Ein Freund von Ihnen behauptet, dass er sein eigenes sicheres Krypto-System entwickelt hat. Zum System gehören zwei Komponenten: eine asymmetrische und eine symmetrische Chiffre. Zeigen Sie, dass beide Verfahren nicht sicher sind.