UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURE 24

SOFTWARE IMPLEMENTATION: BUILD AUTOMATION AND VERSION CONTROL

Prof. Luigi Libero Lucio Starace

luigiliberolucio.starace@unina.it

https://www.docenti.unina.it/luigiliberolucio.starace

THE WATERFALL SOFTWARE PROCESS MODEL

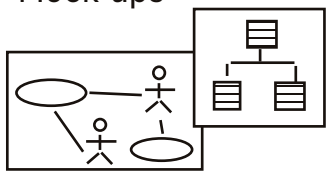
Requirements Engineering → **System Design** → **Software and UI Design** → **Implemen-tation** → **Testing** → **Operation and Maintenance**

Requirements collected via:

- Interviews with Stakeholders
- Personas
- Stories and Scenarios

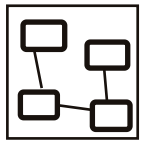
Specified using:

- Use Cases
- Natural Language
- Domain Models
- Mock-ups



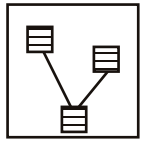
Define System Architecture

- Requirements are allocated to software sub-systems
- Sub-systems are allocated to hardware resources
- Architectural Patterns



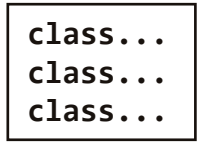
Define Subsystems

- Objects required to realize each subsystem are defined.
- Software Design Patterns
- Usability Engineering
- High-fidelity Wireframing



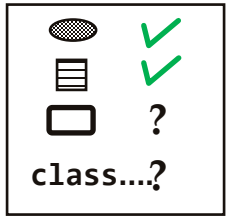
Each Subsystem is implemented

- Source code and other artifacts
- Clean Code
- Frameworks and ORMs
- Focus on Software Quality



Ensure the Software satisfies customers

- Code inspections
- Functional Testing (unit, integration, system testing)
- Usability Testing



System is put into practical use

Maintance will be required at some point

- To fix errors that were not discovered in previous phases
- To adapt the software to changes in requirements on in its environment

Building Software Projects

As the size of a software project grows, **packaging** the **product** in a **distributable package** becomes increasingly complex:

Many steps might be required:

- Preliminary checks;
- Download/Update dependencies;
- Compile;
- Test;
- Generate documentation;
- Code coverage instrumentation, mutation testing, ...

Build automation

- Manually handling all these steps is not very fun…
- Building custom scripts from scratch to orchestrate these steps also does not sound like a very good idea… **Why?**
  - Reinventing the wheel;
  - Maintenance overhead;
  - Limited scalability;
  - Lack of standardization.
- Dedicated build automation solutions are typically employed to handle **most (or even all)** steps in the build/packaging process
- Well-known build automation tools include: CMAKE, Ant, Gradle, **Maven**

Apache Maven

What is Maven?

Maven is a **Project Management** and **comprehension** tool.

It provides ways to manage:

- Builds
- Documentation
- Reporting
- Dependencies
- Releases
- Distribution

Build lifecycle

Maven is based on the concept of **build lifecycles**, i.e., processes for building and distributing a particular artifact

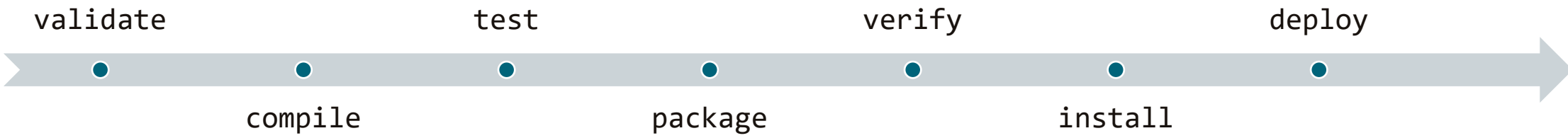
Three built-in build lifecycles:

- **default:** handles the deployment of the entire project
- **clean:** handles project cleaning (remove temporary files)
- **site:** handles the creation of the project site documentation

A build lifecycle is defined by a sequence of **build phases**

Build phases

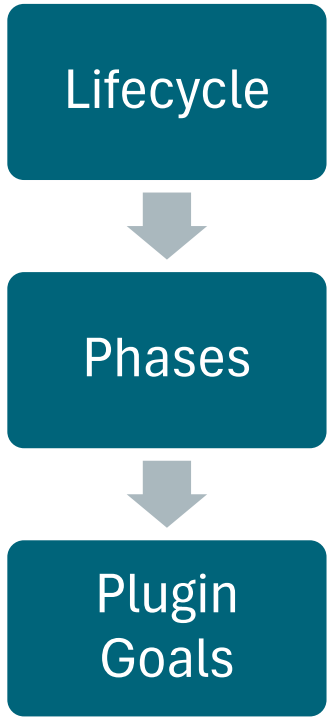
- The **default** lifecycle includes the following phases (and some more!)



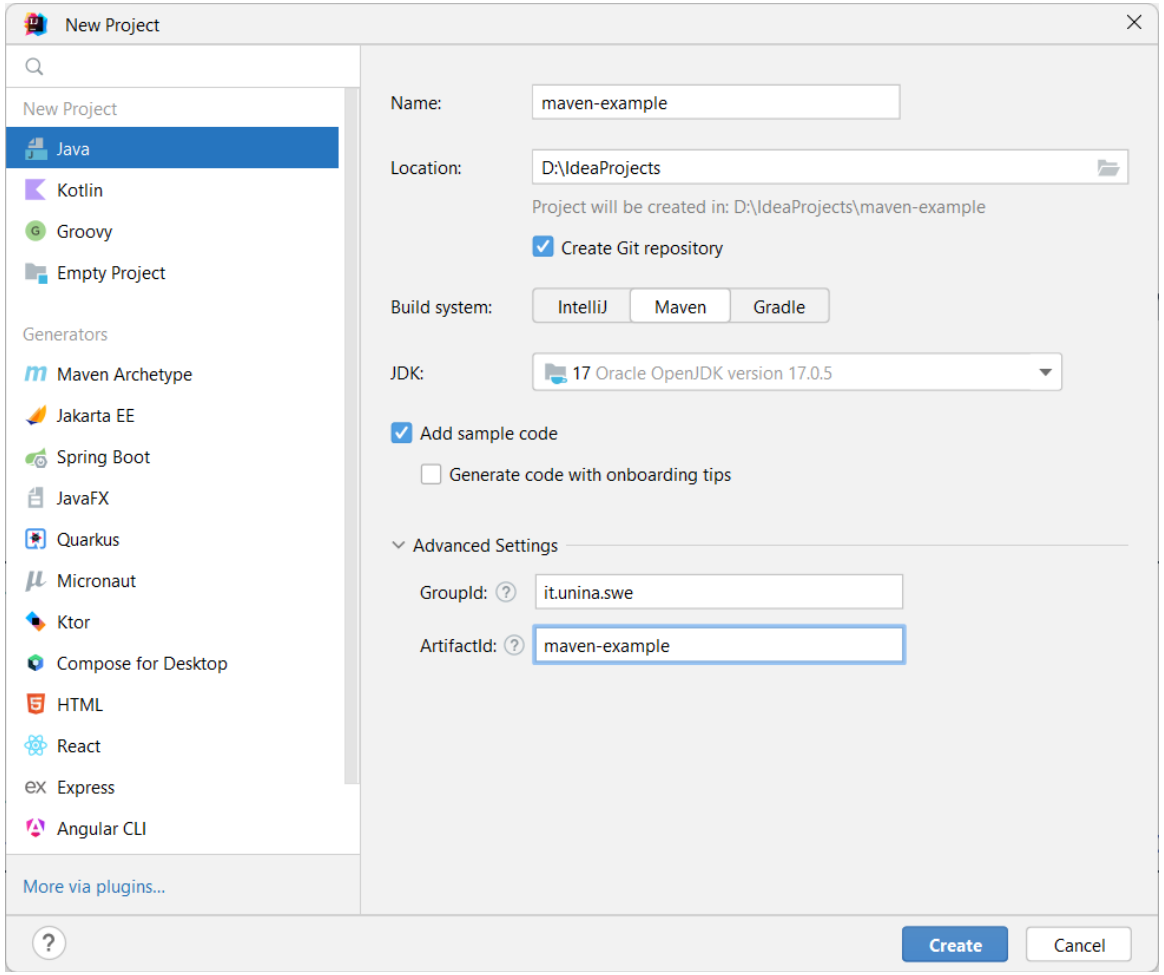
- For more details on the phases in the built-in lifecycles: Reference
- You can also run only some of the phases
- E.g., if you run the command `mvn package` only the `validate`, `compile`, `test` and `package` phases will be executed.

Plugin Goals

- A **build phase** is responsible for a specific step in the build lifecycle, but different project may implement a phase differently. This is done by binding plugin goals to the lifecycle phase.
- A build phase consists of zero or more **plugin goals**



SWE's FIRST (?) MAVEN PROJECT

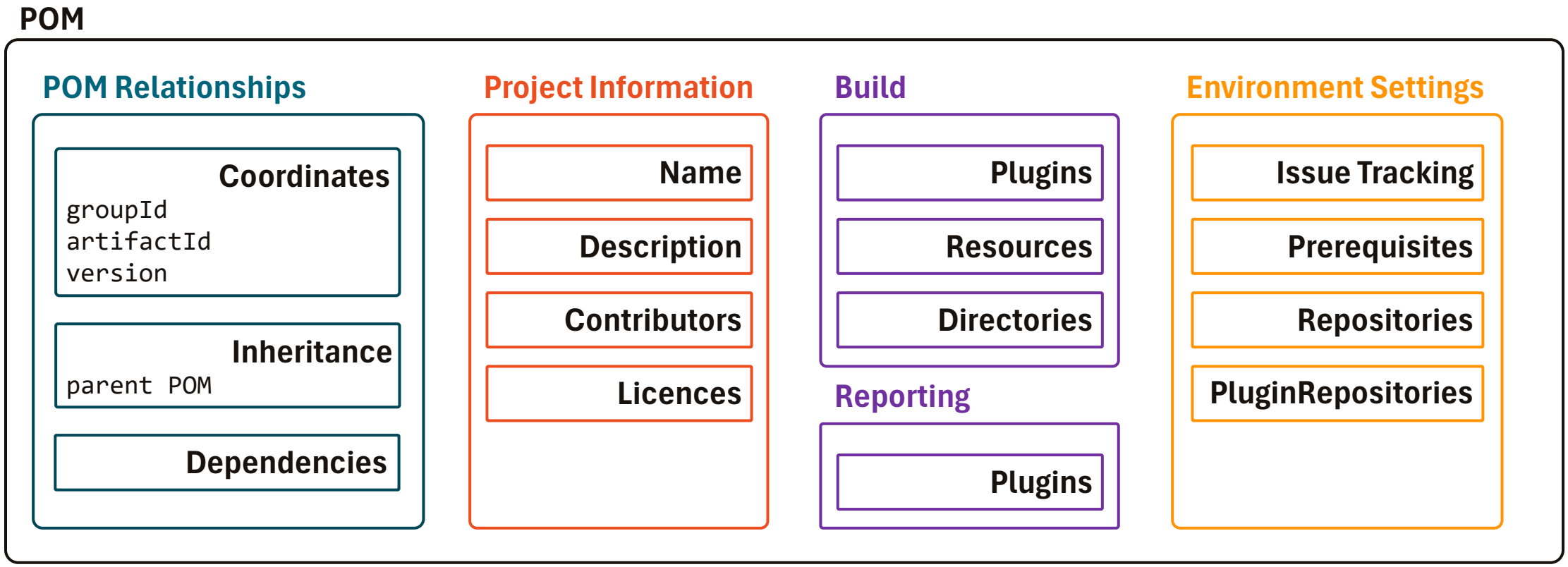


The Project Object Model (POM)

- A **POM** is the fundamental unit of work in Maven.
- It's an XML file information about the project and configuration details
- A minimal POM is as simple as the one below

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>it.unina.swe</groupId>
  <artifactId>maven-example</artifactId>
  <version>1.0.0</version>
</project>
<!-- fully qualified name for the artifact is it.unina.swe:maven-example:1.0.0 -->
```

The Project Object Model (POM)



Managing dependencies

```
<!- in the pom.xml file -->
<dependencies>
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>33.5.0-jre</version>
  </dependency>

  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.18.0</version>
  </dependency>
  <!- ... -->
</dependencies>
```

The Maven help plugin

- Used to get information about a project or the system
- Useful to understand what's going on
- Includes 7 goals, including `help:describe`
- For example, to list the goals in a given phase, one can issue:

```
>> mvn help:describe -Dcmd=<phaseName>

>> mvn help:describe -Dcmd=test

[INFO] 'test' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
```

Built-in plugin goals

- Some default plugin goals are bounded to the built-in phases
- E.g.: the `maven-compiler-plugin` goals `compile` and `testCompile` are bound, respectively, to the `compile` and `test-compile` phases of the default lifecycle.
- To see more details on Maven does by default one can use the `help:describe` of the `help:effective-pom`
- A nice alternative is the `buildplan-maven-plugin`

Using the Maven help plugin

```
>> mvn help:describe -Dcmd=test
[...]
It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle includes the
following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: org.apache.maven.plugins:maven-resources-plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: org.apache.maven.plugins:maven-resources-plugin:2.6:testResources
* test-compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile
[...]
```

Using the buildplan-maven-plugin

To list all the plugin execution within a project:

```
>> mvn fr.jcgay.maven.plugins:buildplan-maven-plugin:list
[INFO] Build Plan for Project:
-------------------------------------------------------------------------------
PLUGIN                  | PHASE                  | ID                     | GOAL
-------------------------------------------------------------------------------
maven-clean-plugin      | clean                  | default-clean          | clean
maven-resources-plugin  | process-resources      | default-resources      | resources
maven-compiler-plugin   | compile                | default-compile        | compile
maven-resources-plugin  | process-test-resources | default-testResources  | testResources
maven-compiler-plugin   | test-compile           | default-testCompile    | testCompile
maven-surefire-plugin   | test                   | default-test           | test
maven-jar-plugin        | package                | default-jar            | jar
maven-install-plugin    | install                | default-install        | install
maven-deploy-plugin     | deploy                 | default-deploy         | deploy
```

Examples

- Generating JAR files
- Generating documentation with Maven and Javadoc
- Generating OpenAPI specs for our JAX-RS Rest API

Version Control with git

Need for version control

- Programming Software Products takes more than few hours and involves multiple software engineers
- Many changes are made over time by possibly multiple engineers to the same files in the codebase

Need for version control

- It’s not nice to deal with files like
  - `Main.java`
  - `Main_final.java`
  - `Main_final_2025_11_03_10_45.java`
  - `Main_final_final_v2_revised_final_I_promise.java`
- Why?
  - Not easy to keep track of **who** changed **what** and **why**
  - Not easy to go back to a previous version if something breaks
  - Not easy to manage conflicts

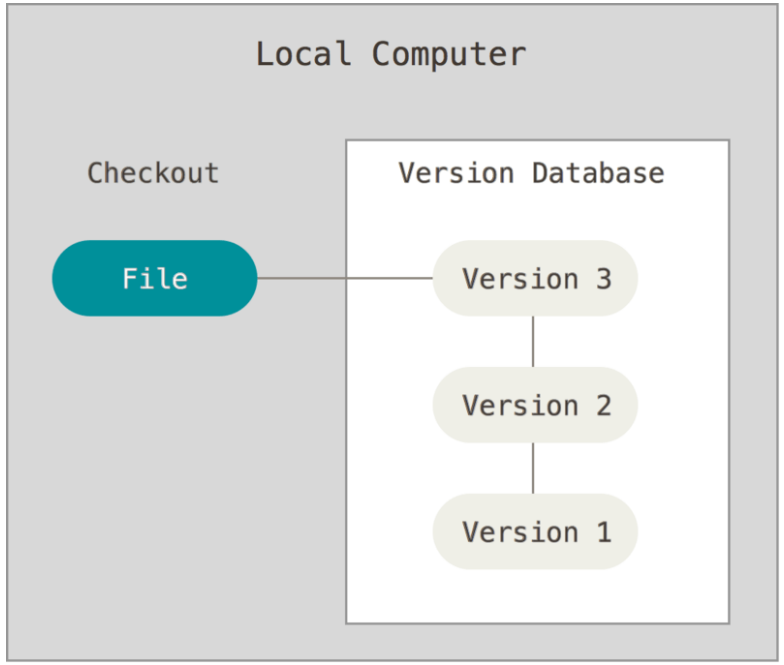
Version Control Systems (VCS)

Tools to record changes to a set of files over time, so you can:

- Revert files back to a previous state
- Revert the entire project back to a previous state
- Compare changes over time

Local version control

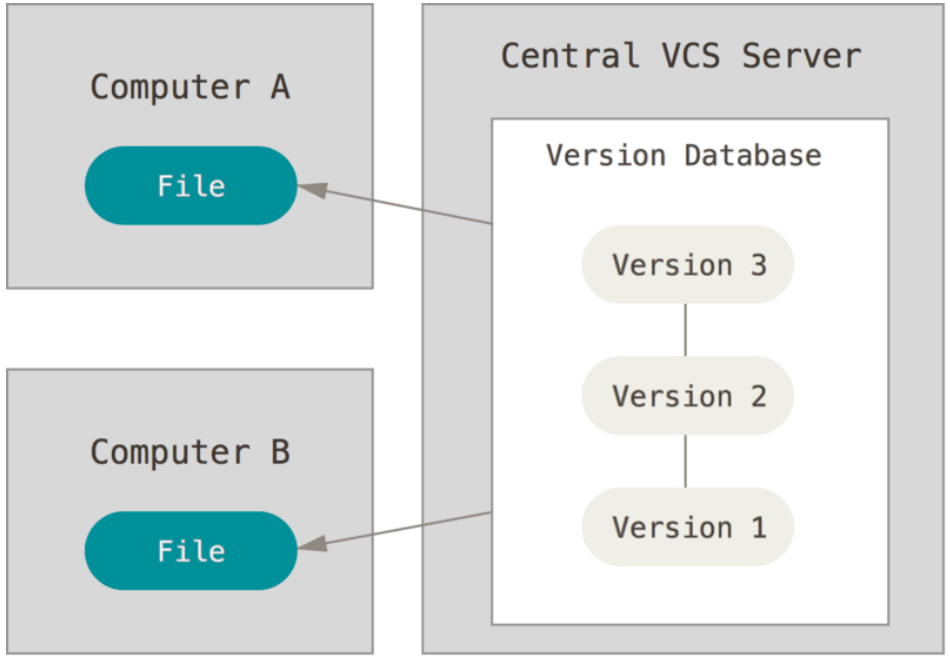
- Copy files in (hopefully timestamped!) directories
  - Error prone!
- Use tools like RCS
- Difficult to collaborate with other people!



https://git-scm.com/book/

Centralized version control

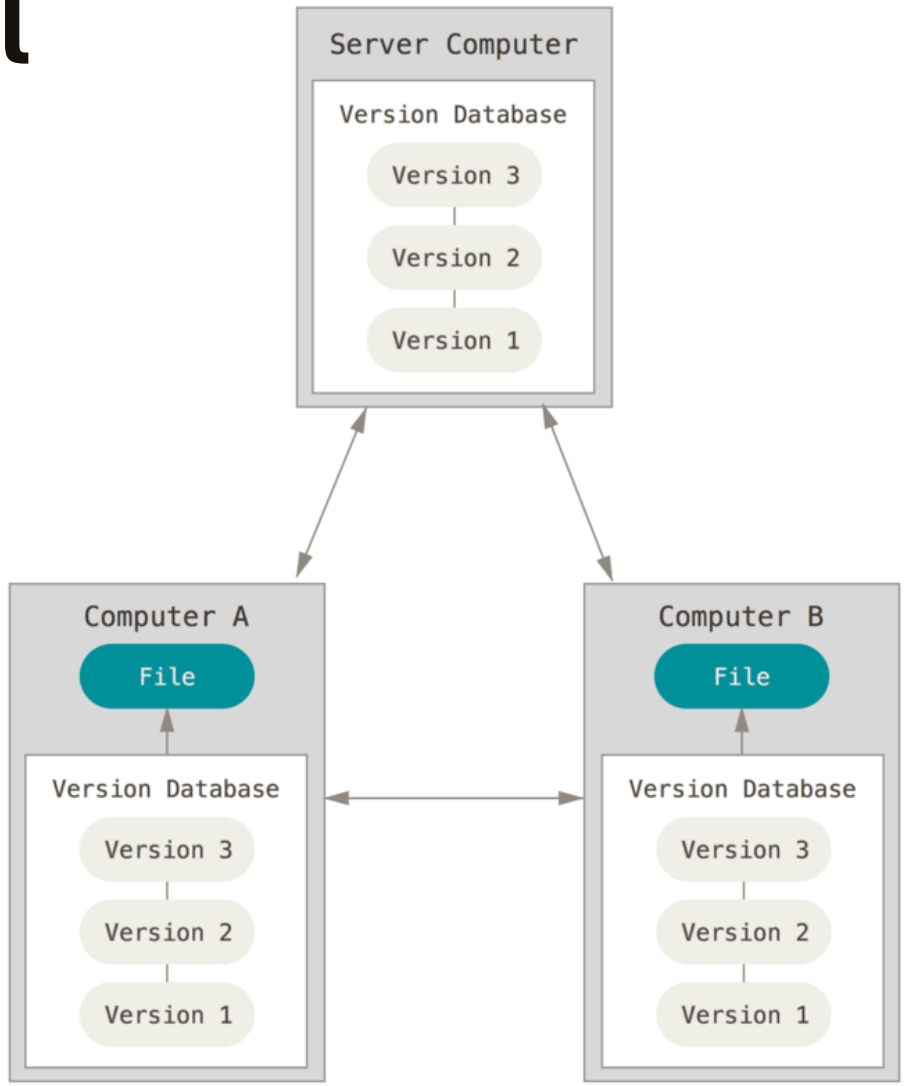
- A centralized server contains all the files
- A number of clients check out files
- They modify their local copies, then “check in” their changes back to the server
- Tools like Subversion, CVS
- Server is a **single point of failure**



https://git-scm.com/book/

Distributed version control

- Local repositories are a complete copy of everything on the remote server
- A number of clients “clone” and “pull” changes from the remote repository
- They modify their local copies, then “push” their changes to the remote server for synchronization with others
- Tools like git, Mercurial

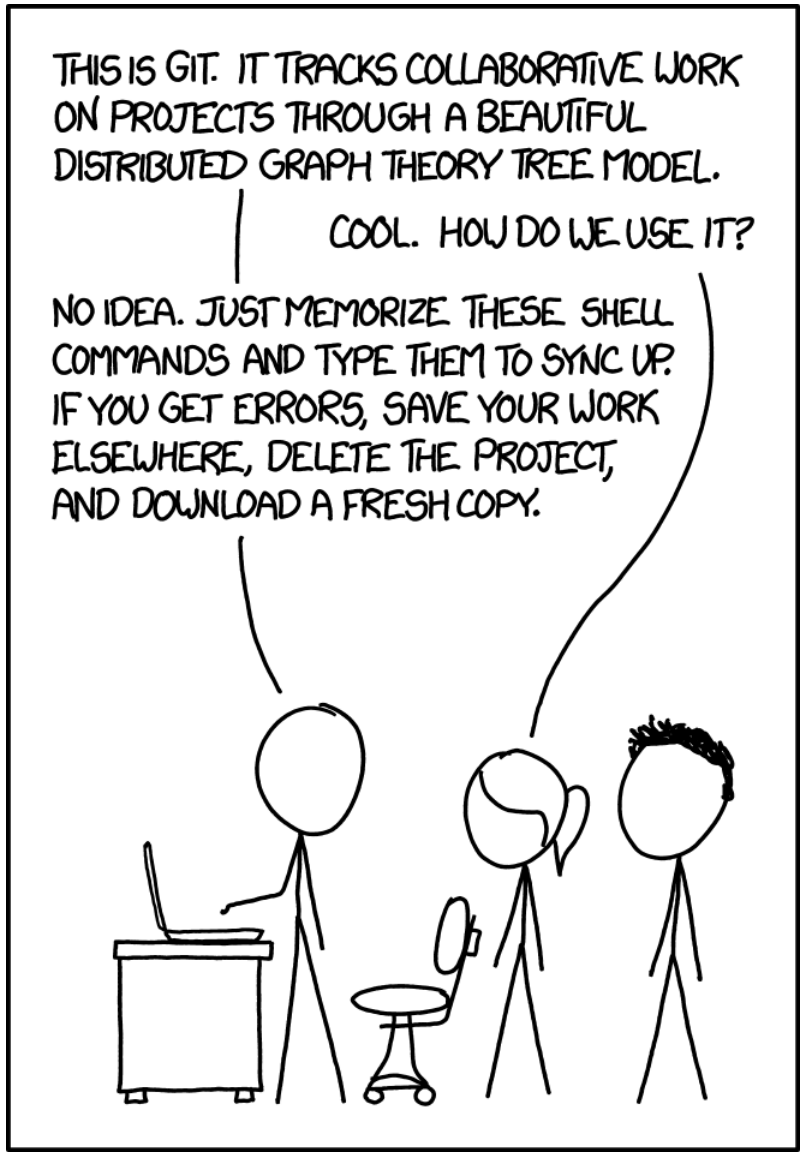


https://git-scm.com/book/

git

- Official website
- Created by Linus Torvalds in 2005
- Fast, fully distributed, non-linear
- Very popular

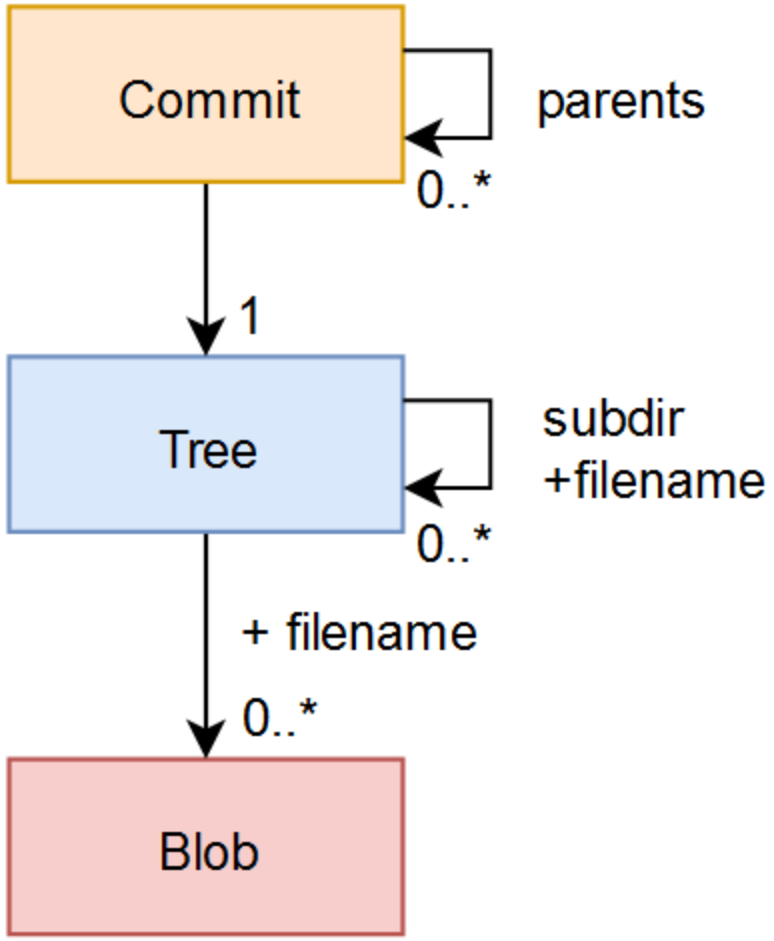
- A «git» is a cranky old man
  (Linus meant himself!)



https://xkcd.com/1597/

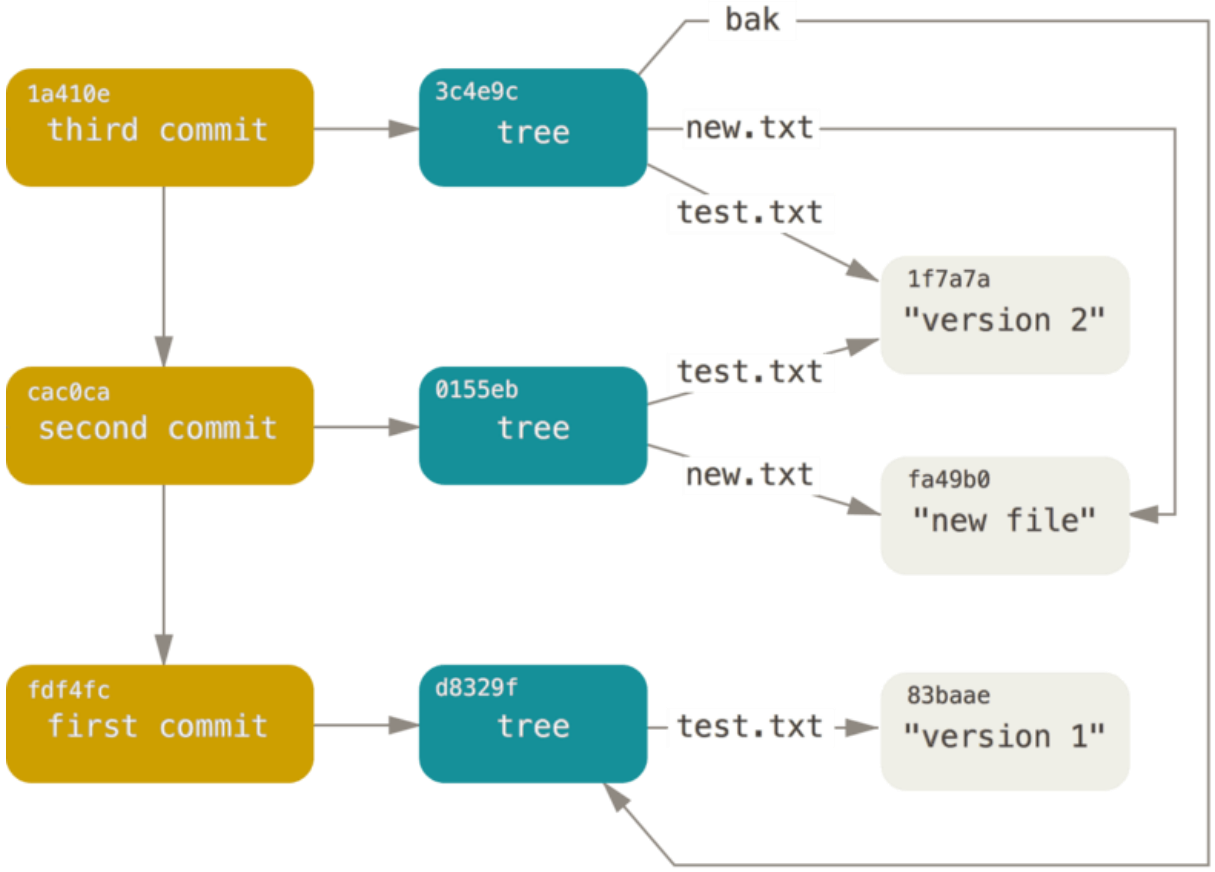
git internals: basics

- git object storage is a DAG of objects, identified by its SHA-1 hash
- A **blob** is the simplest object, a bunch of bytes corresponding to a file
- A **tree** is an object representing directories
- A **commit** refers to a tree representing the state of the files at the time of the commit, and to 0..n **parent** commits
- Nice introduction to Git internals



git internals: example

- In first commit, only **test.txt**
- In second commit, **new.txt** is added and **test.txt** is updated
- In third commit, a new directory **bak** is added, containing the original **test.txt** file

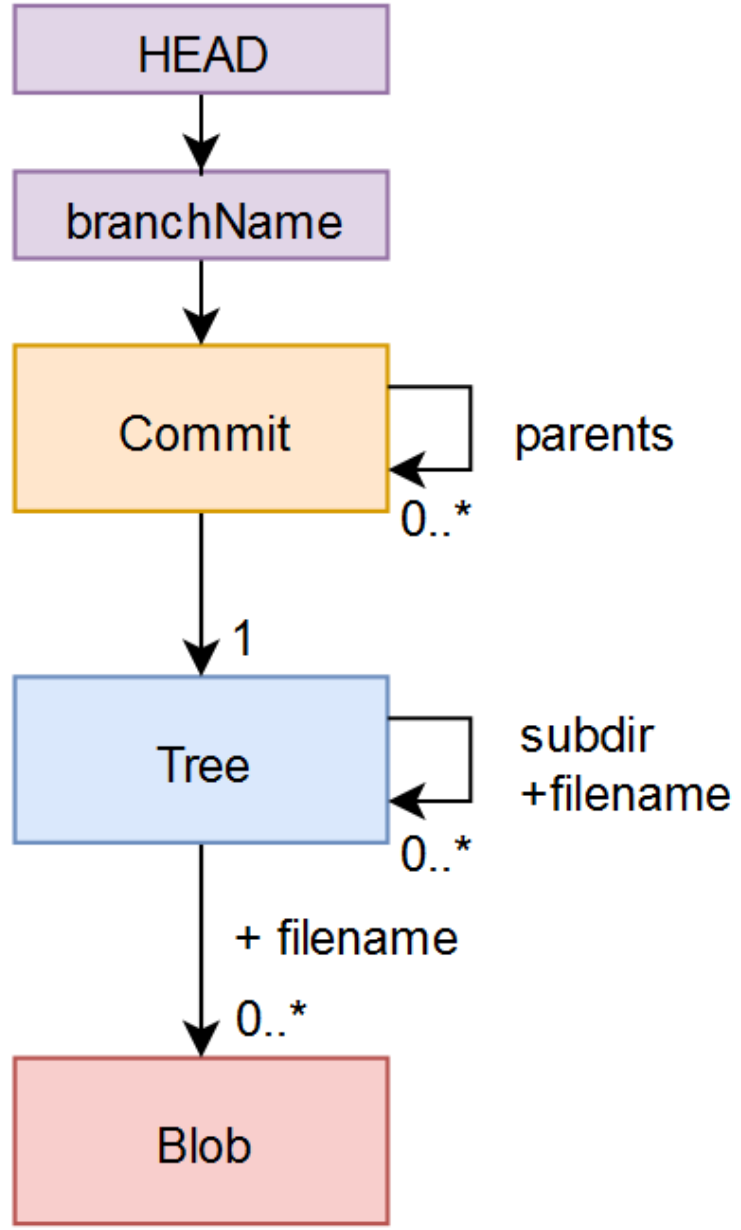


https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

git internals: refs

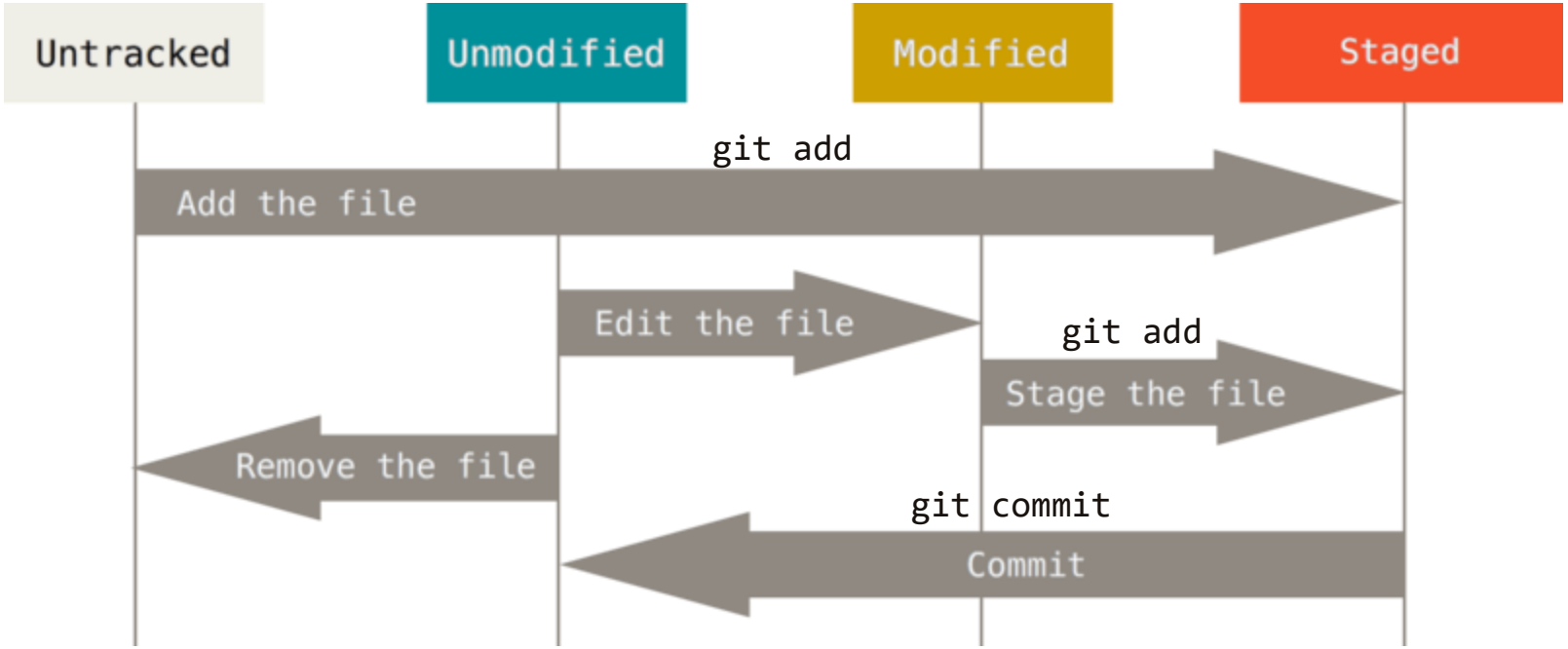
- References, or heads or branches, are **pointers** to a node in the DAG.
- Unlike DAG nodes that cannot be changed, these pointers can be moved around freely.
- The **HEAD** ref is a pointer to the currently active branch.

More on git internals: here



Tracking changes with git

- Lifecycle of your files under git
- `>> git status` prints information about each file



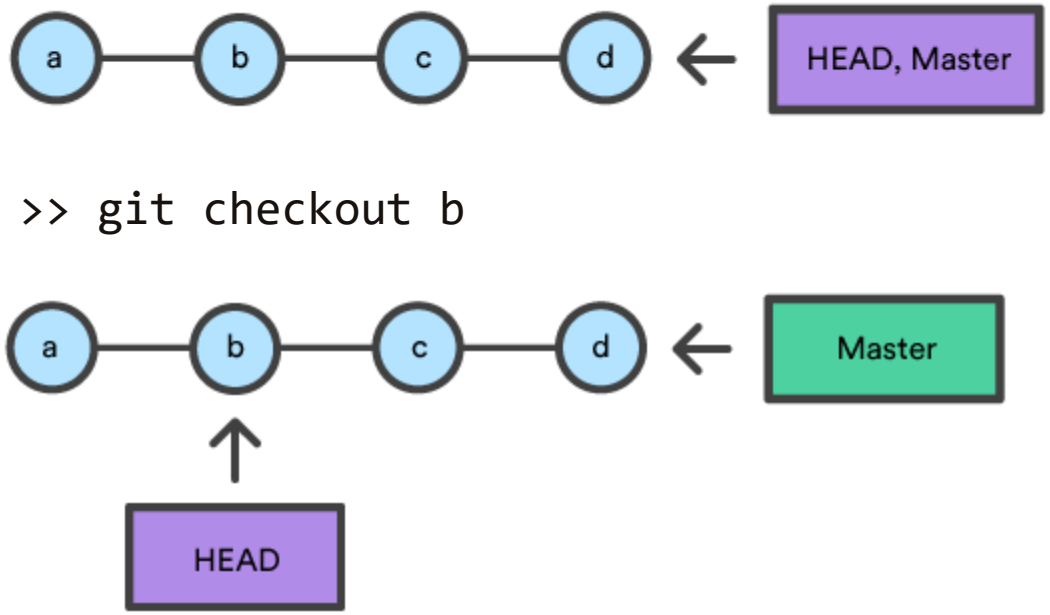
Using git status

```
>> git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits. (use "git push" to publish
your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

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:   bar.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        foo.txt
```

Undoing changes

`>> git commit --amend` is useful to redo the last commit

`>> git checkout` moves the HEAD label to a given commit/branch

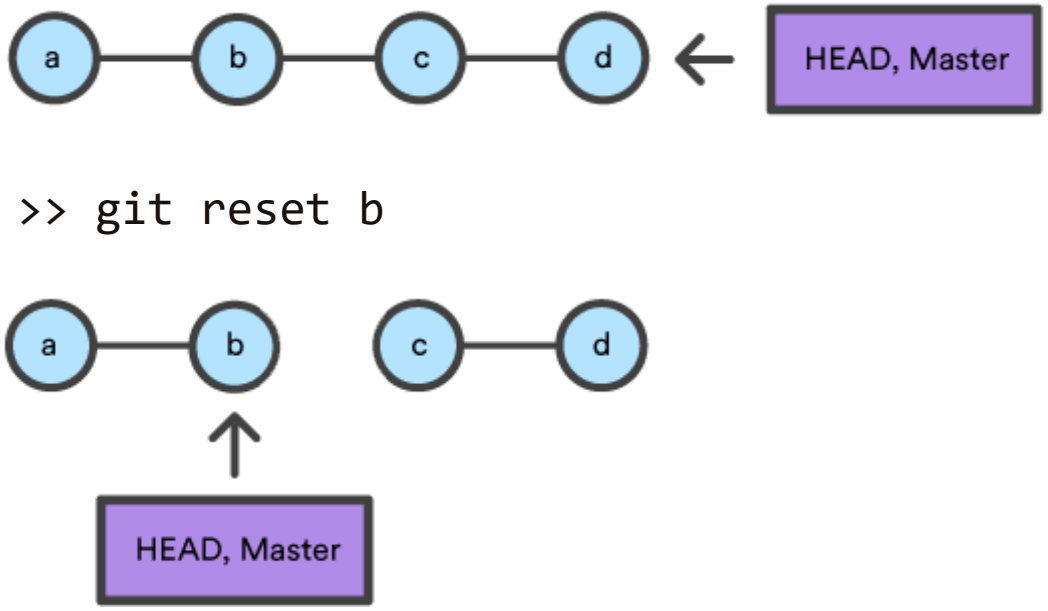


https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

Undoing changes

`>> git commit --amend` is useful to redo the last commit

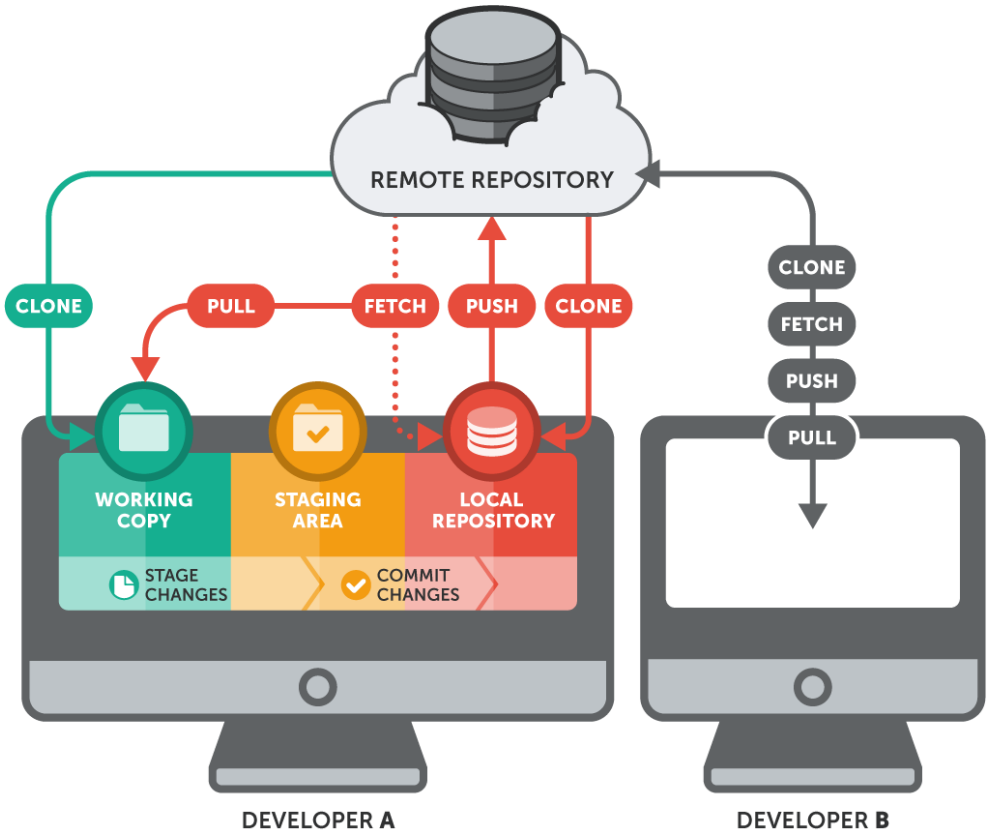
`>> git reset` moves both HEAD and current branch ref



https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

git remotes

- **Remote** repositories are used to collaborate with others
- They are versions of your project hosted somewhere else
- Collaborating means to push/pull data from remotes when you need to share work
- There can up to many remotes



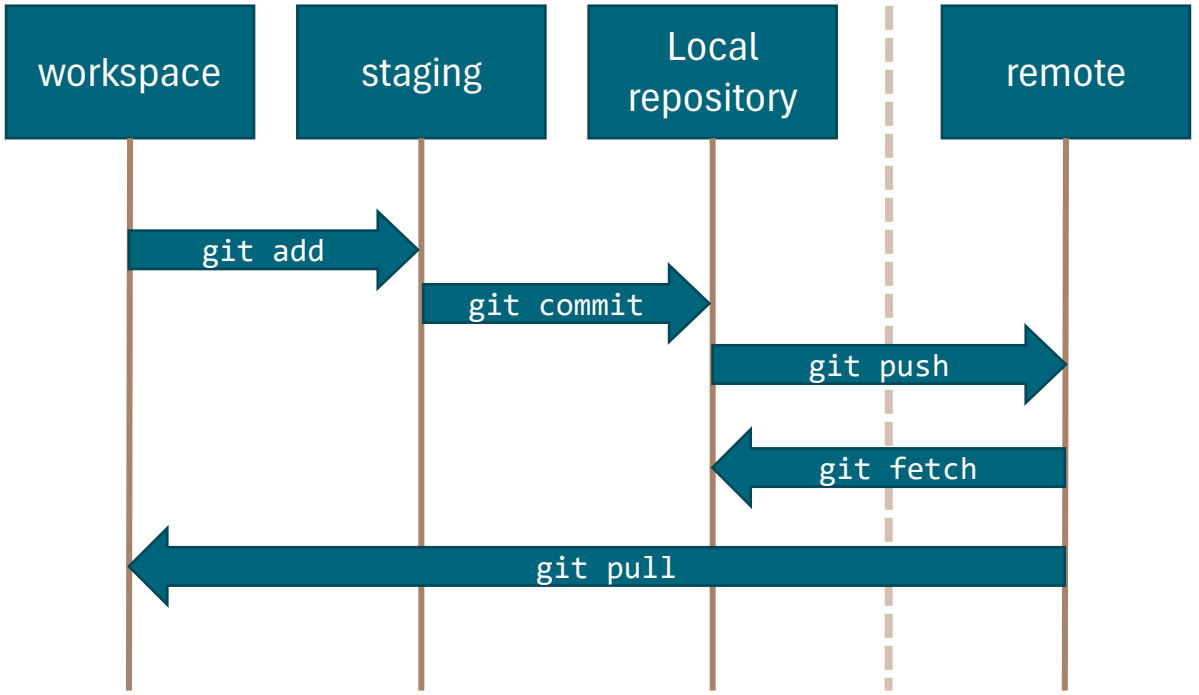
https://blog.netsons.com/git-software-guida-facile/

Listing and adding remotes

```
>> git remote
origin

>> git remote -v
origin  https://github.com/luistar/git-demo-swe.git (fetch)
origin  https://github.com/luistar/git-demo-swe.git (push)

>> git remote add myremote https://github.com/coworker/repo

>> git remote -v
origin  https://github.com/luistar/git-demo-swe.git (fetch)
origin  https://github.com/luistar/git-demo-swe.git (push)
myremote https://github.com/coworker/repo (fetch)
myremote https://github.com/coworker/repo (push)
```

Syncing with remotes

- **git push** is used to upload local repository content to a remote
- **git fetch** is used to download data from the given remote
- **git pull** is used to download data from the given remote, and immediately update the local repository to match that content

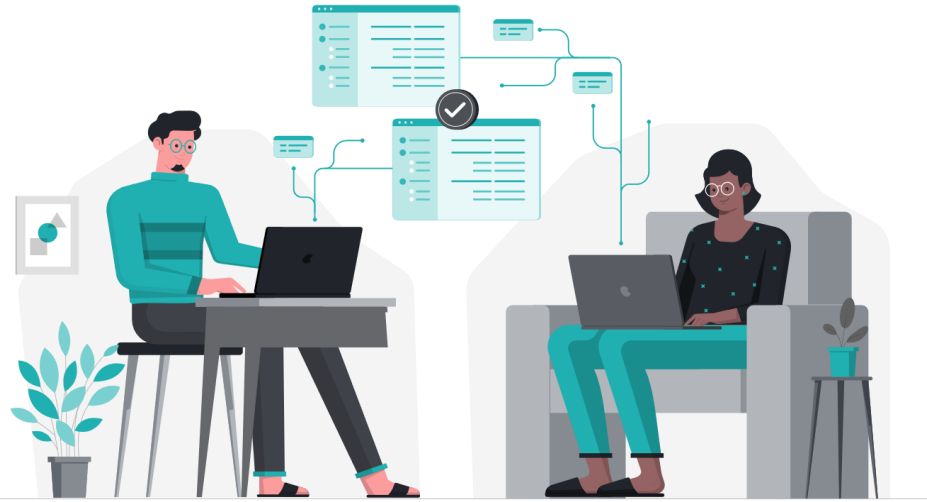


Resolving conflicts

- Sometimes multiple devs might edit the same content
- Git does a good job at automatically merging changes, but it’s not always possible!
- In some cases, we need to manually resolve the conflicts
- **Let’s create and solve a conflict! (demo)**

git branching

- In a collaborative environment, many developers work on the same source code
- Some fix bugs, others add new features
- If they all worked on the main branch, they might conflict often with eachother
- In some settings (e.g.: CI/CD) the main branch should be always buildable
- **Branches** allow developers to isolate their work



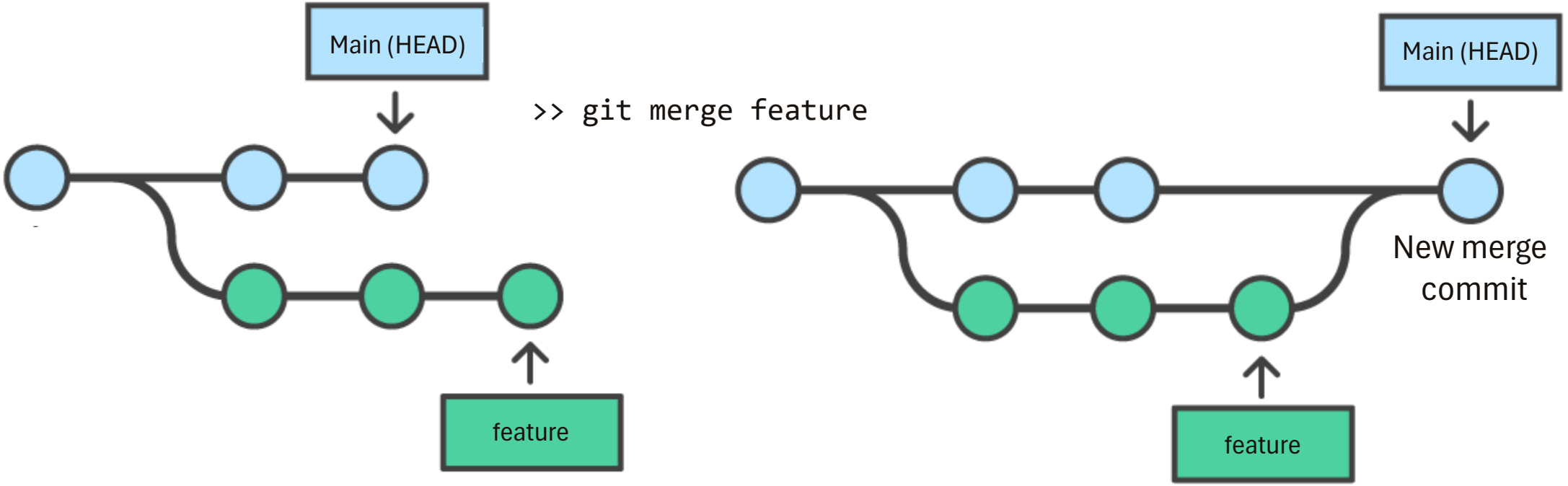
Creating a new branch

- A branch is basically a pointer to a commit
- **git branch** lists all the branches
- **git branch <name>** creates a new <name> branch
- **git checkout** or **git switch** can be used to switch (i.e., move HEAD) to a different branch

```
>> git branch
* main

>> git branch feature

>> git branch
  feature
* main

>> git checkout feature
Switched to branch 'feature’

>> git branch
* feature
  main
```

Integrating branched history: git merge

- **git merge** allows us to put forked history back together again

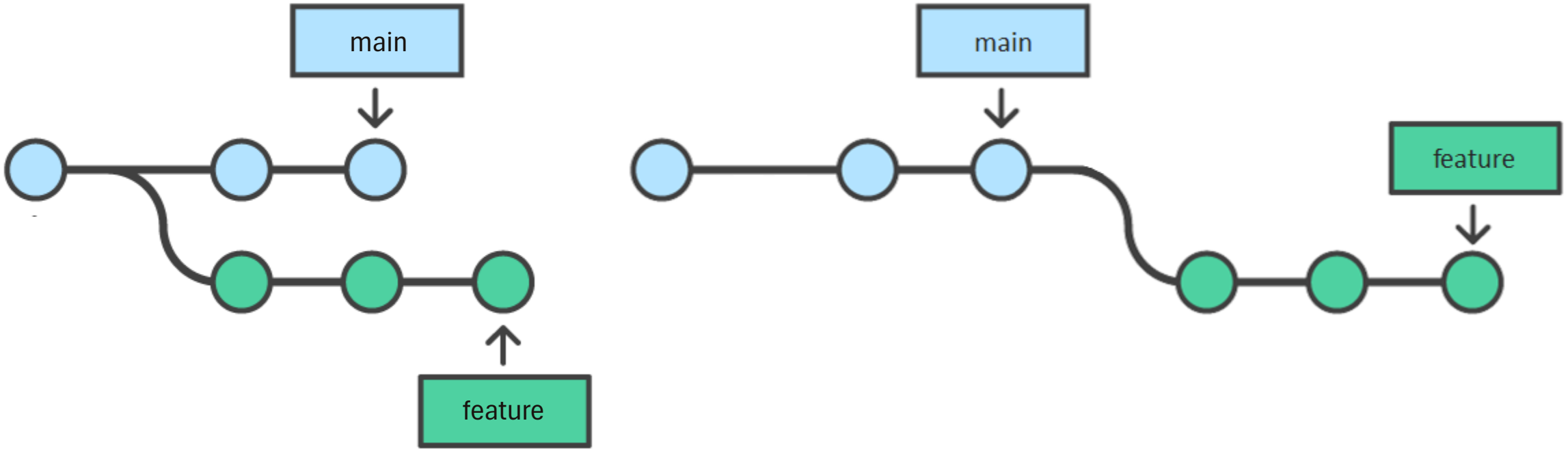


Integrating branched history: git merge

- **git merge** allows us to put forked history back together again
- A new merge commit is added, having as parents the commits referenced by the merged branches
- Conflicts might arise (read more here)

Integrating branched history: git rebase

- **git rebase** solves the same problem as git merge.



Integrating branched history: git rebase

- **git rebase** solves the same problem as git merge.
- The target branch is copied «on top» of the current one
- No new merge commit is created (cleaner history)
- More on merge vs rebase here