Software Project Management with Maven and git

Luigi Libero Lucio Starace

luigiliberolucio.starace@unina.it

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**

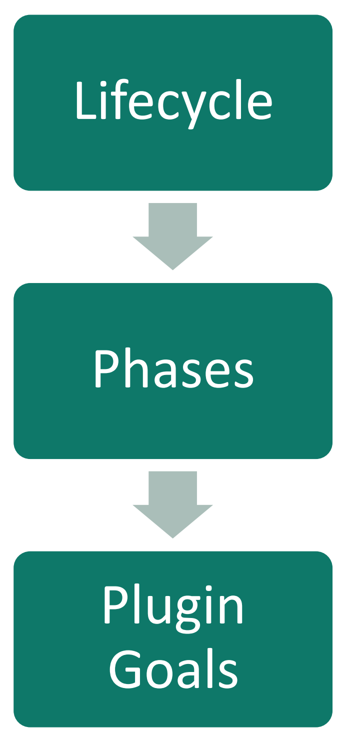
Build phases

- 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**

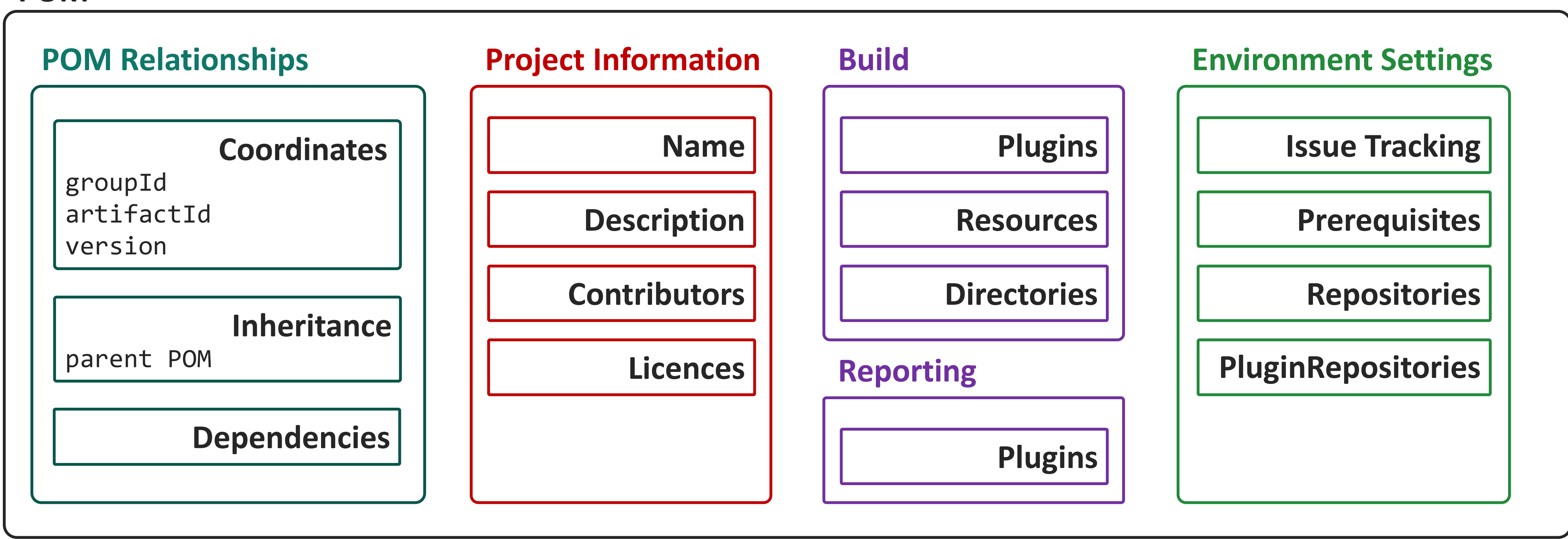


The Project Object Model (POM)

- 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.spme</groupId>
  <artifactId>project</artifactId>
  <version>1.0.0</version>
</project>
<!-- the fully qualified name for the artifact is it.unina.spme:project: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>30.1.1-jre</version>
  </dependency>

  <dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>2.2</version>
    <scope>test</scope> <!- dependency scope reference -->
  </dependency>
  <!- ... -->
</dependencies>
```

The Maven help plugin

- Used to get information about a project or the system
- Useful to understand what's going on
- Include 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
```

Our running example

- Let’s consider a very simple project

```
project
├── src/main/java
│   └── it.unina.spme.project
│       └── Utilities.java
│           ├── mean(Set<Integer>) : double
│           └── numberOfSubsets(Set) : int
├── src/test/java
│   └── it.unina.spme.project
│       └── UtilitiesTest.java
│           ├── meanShouldThrowExceptionOnEmptySet() : void
│           ├── meanShouldWorkWithSets() : void
│           ├── meanShouldWorkWithSingletons(int) : void
│           └── other tests ...
└── pom.xml
```

Running tests with Maven

When we run our default lifecycle up to the test phase, out of the box we get:

```
>> mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< it.unina.spme:project >------------------------
[INFO] Building Project 1.0.0
[INFO] --------------------------------[ jar ]---------------------------------
[....]
-------------------------------------------------------
 T E S T S
-------------------------------------------------------

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0
```

Running tests with Maven

- Let’s start by adding the maven-surefire-plugin
- Defines only one goal: surefire:test, which binds by default to the `test` phase

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.0</version>
    </plugin>
  </plugin>
</build>
```

Running tests with Maven

```
>> mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< it.unina.spme:project >------------------------
[INFO] Building Project 1.0.0
[INFO] --------------------------------[ jar ]---------------------------------
[....]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running it.unina.spme.project.UtilitiesTest
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.091 s - in
it.unina.spme.project.UtilitiesTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[....]
```

Reporting

When dealing with large software projects, **reporting tools** are essentials

- to monitor code quality (metrics)
- to ensure everything is properly tested (coverage/mutation score)

Reporting tools that can be integrated in Maven include Clover, SonarQube, JaCoCo (computes only coverage)

Statistics and Coverage Reports with Clover

- Documentation, quick start, basic usage on the website.

```
<plugin> <!-- main part, note that this snippet alone is not enough! See docs! -->
  <groupId>org.openclover</groupId>
  <artifactId>clover-maven-plugin</artifactId>
  <version>4.4.1</version>
  <executions>
    <execution>
      <id>clover-instrumentation</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>instrument</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

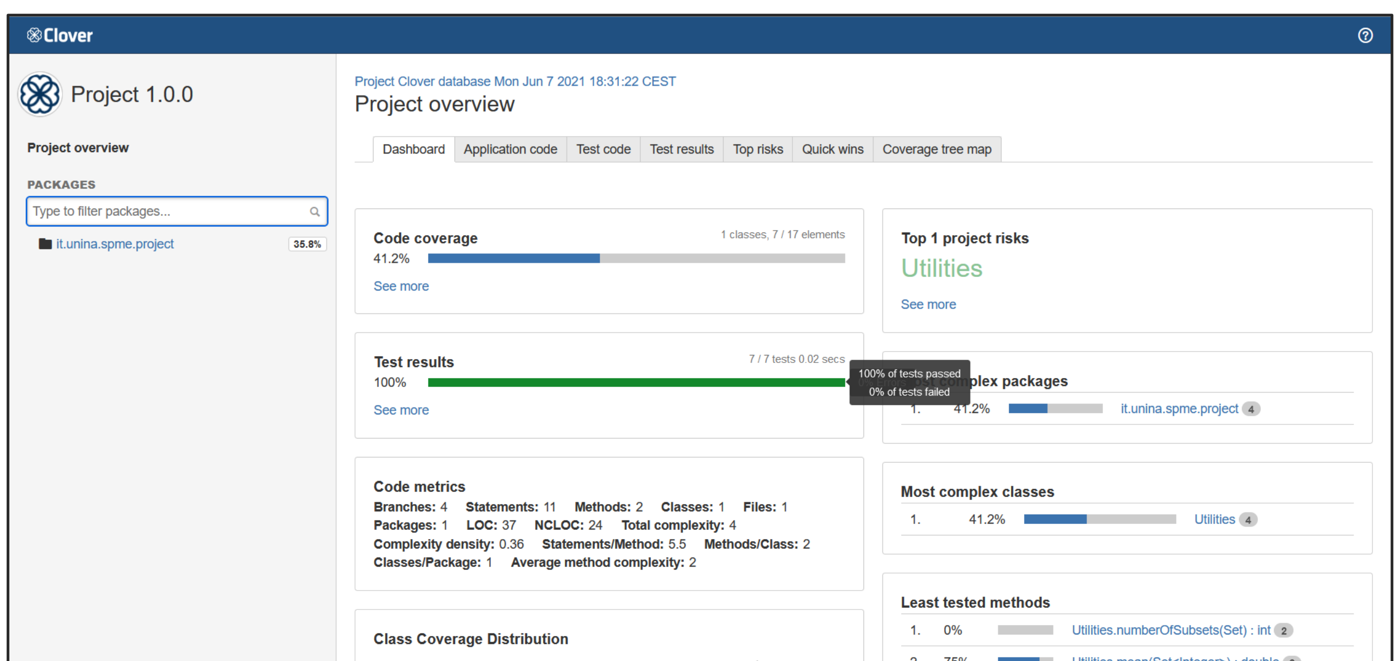
Using Clover

One can run `>> mvn clean test site` to generate the report

- `clean` is necessary to clean Clover’s temp. files
- `test` so we run our default lifecycle up to the test phase
- `site` is used to generate an html report

The Clover report

- Available in `/target/site/clover/index.html`
- Includes very detailed (test-method level) coverage reports and metrics



Failing a build based on a coverage target

```xml
<plugin>
  <groupId>org.openclover</groupId>
  <artifactId>clover-maven-plugin</artifactId>
  <version>4.4.1</version>
  <configuration>
    <generateXml>true</generateXml>
    <!-- define target coverage percentage -->
    <targetPercentage>50%</targetPercentage>
  </configuration>
  <executions>
    <!-- first execution, instrumentation -->
    <execution>
      <id>clover-instrumentation</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>instrument</goal>
      </goals>
    </execution>
```

```xml
    <!-- then we bind the check goal -->
    <execution>
      <id>clover-check-coverage</id>
      <phase>verify</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Failing a build based on a coverage target

- Run up to the verify phase, to which we bounded the clover:check goal

```
>> mvn verify
[INFO]
[INFO] -----------------------< it.unina.spme:project >------------------------
[INFO] Building Project 1.0.0
[INFO] --------------------------------[ jar ]---------------------------------
[....]
[INFO] Coverage check FAILED
[ERROR] Total coverage of 41,2% did not meet target of 50%
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.589 s
[INFO] Finished at: 2021-06-07T18:52:51+02:00
[INFO] ------------------------------------------------------------------------
```

git

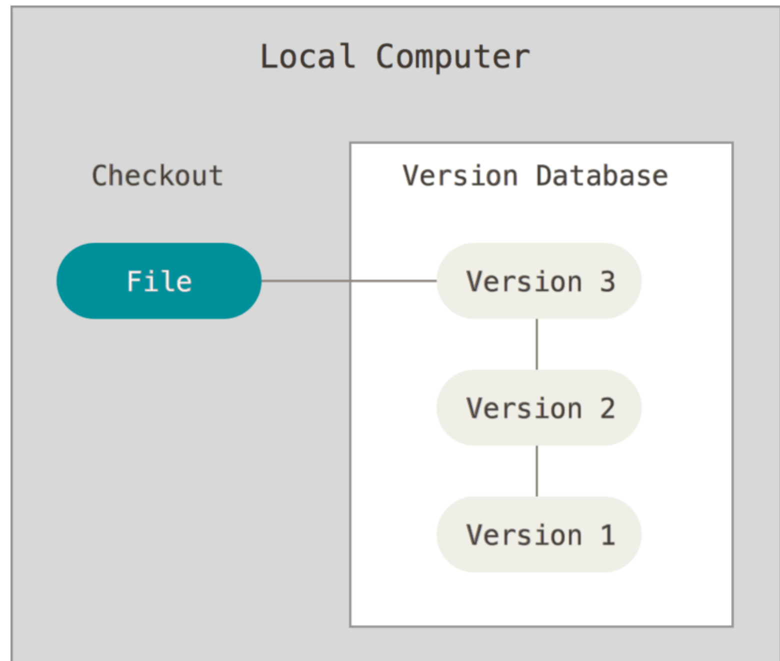
Version Control Systems (VCS)

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

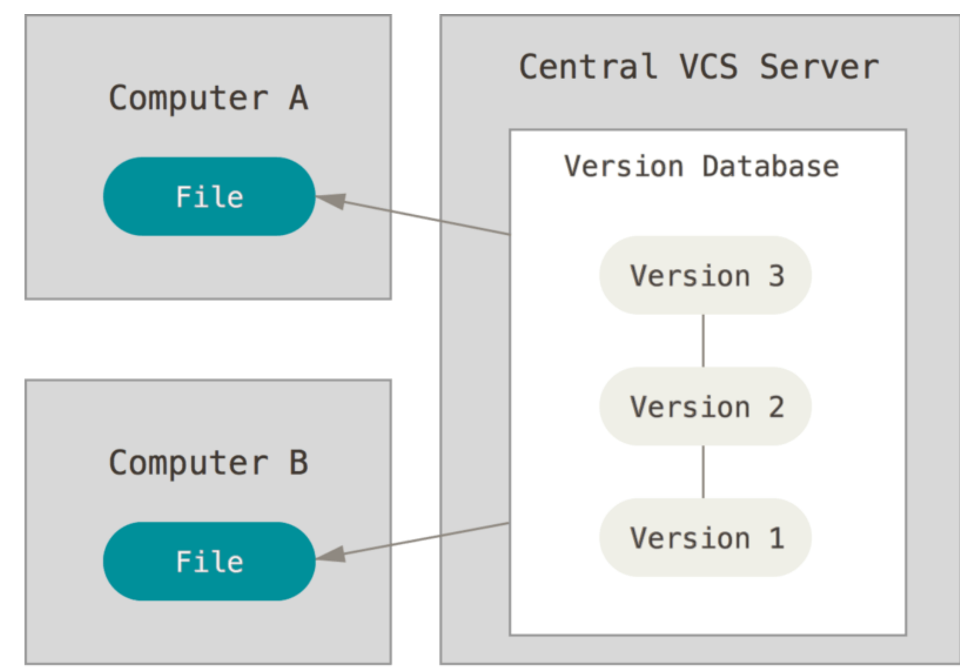
- Copy files in (hopefully timestamped!) directories
  - Error prone!
- Use tools like RCS
- Difficult to collaborate with other people!



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

Centralized version control

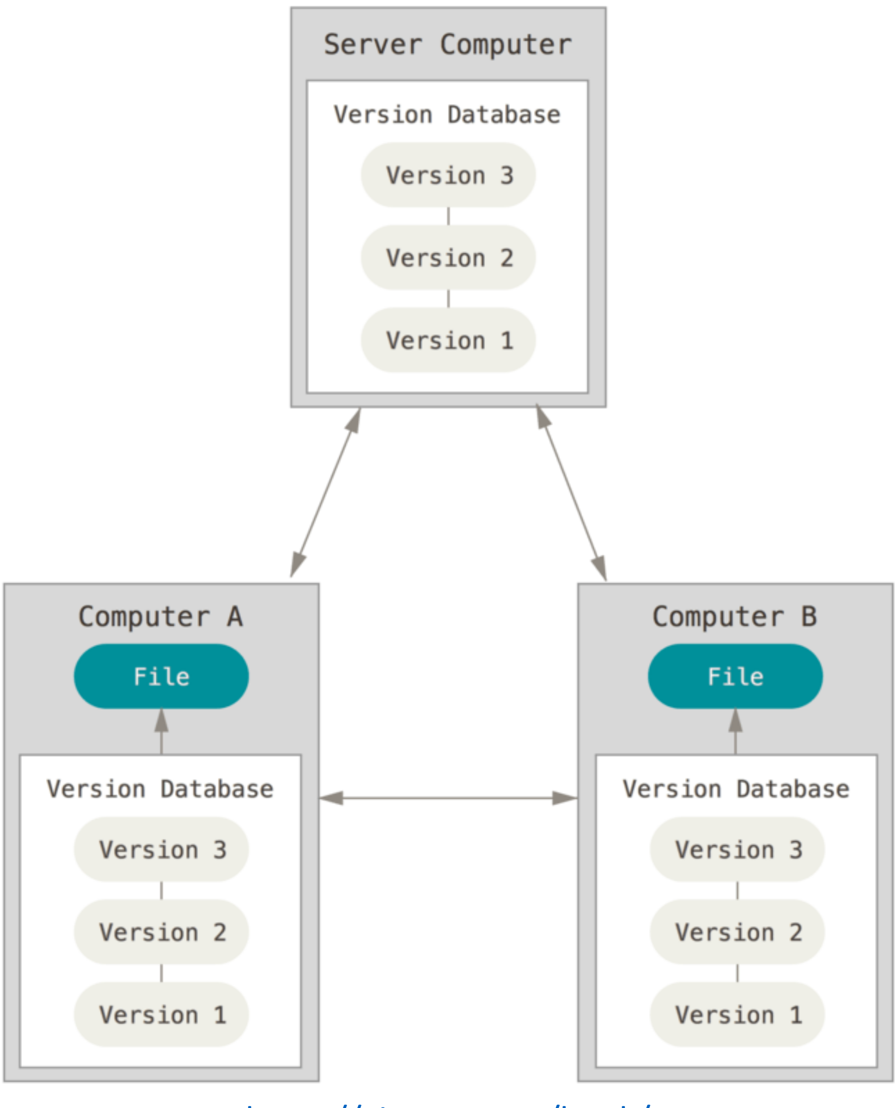
- A centralized server contains all the files
- A number of clients check out files
- They modify their local copies, then "check in" their changes back to the server
- Tools like [Subversion](), [CVS]()
- Server is a **single point of failure**



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

Distributed version control

- Local repository 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

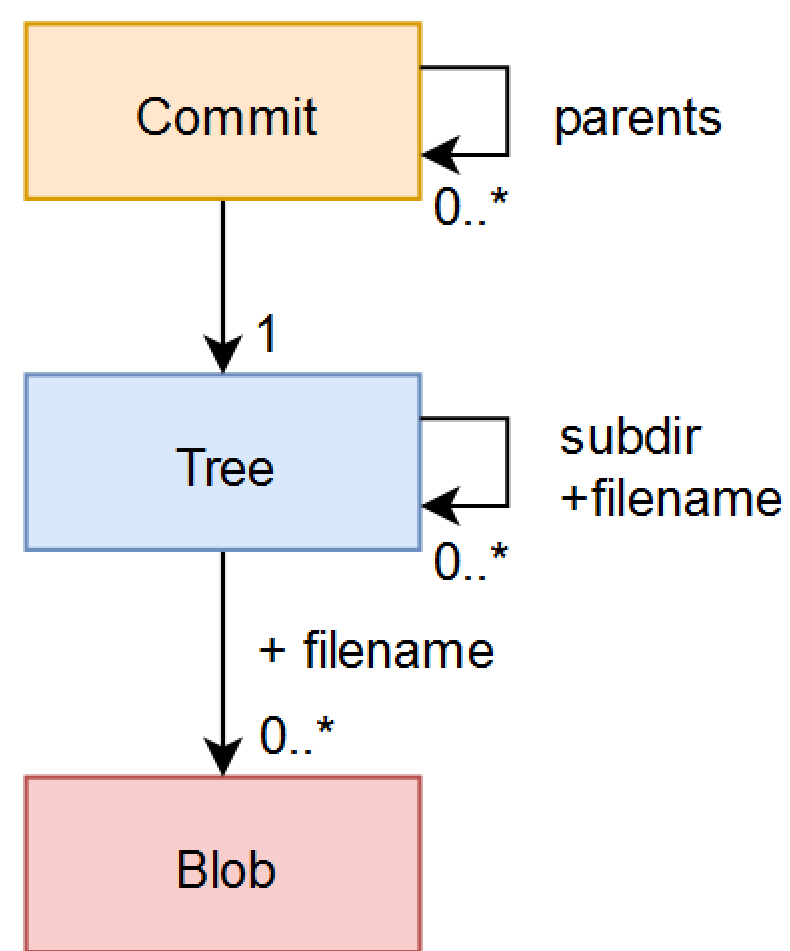
- Official website
- Created by Linus Torvalds in 2005
- Fast, fully distributed, non-linear
- Very popular

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

https://xkcd.com/1597/

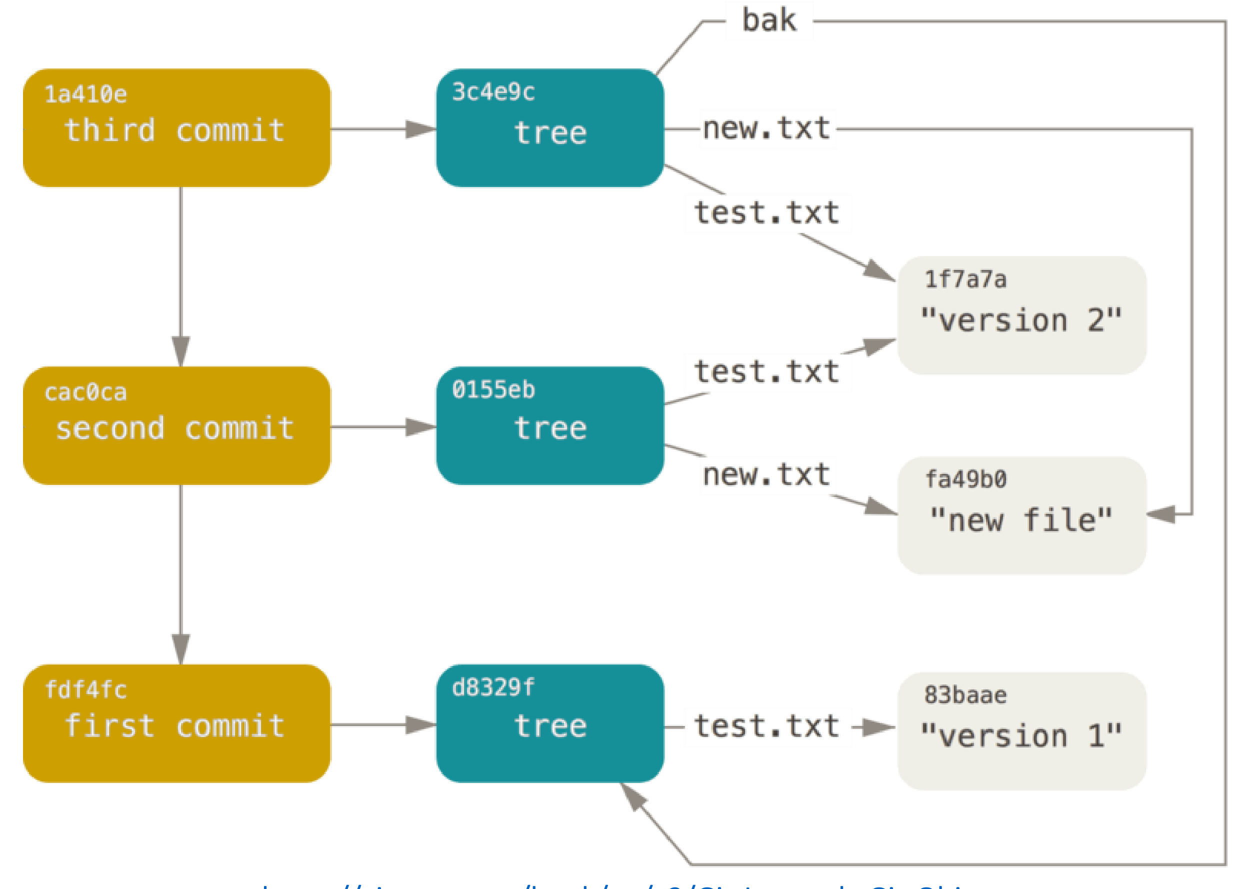
git internals: basics

- 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

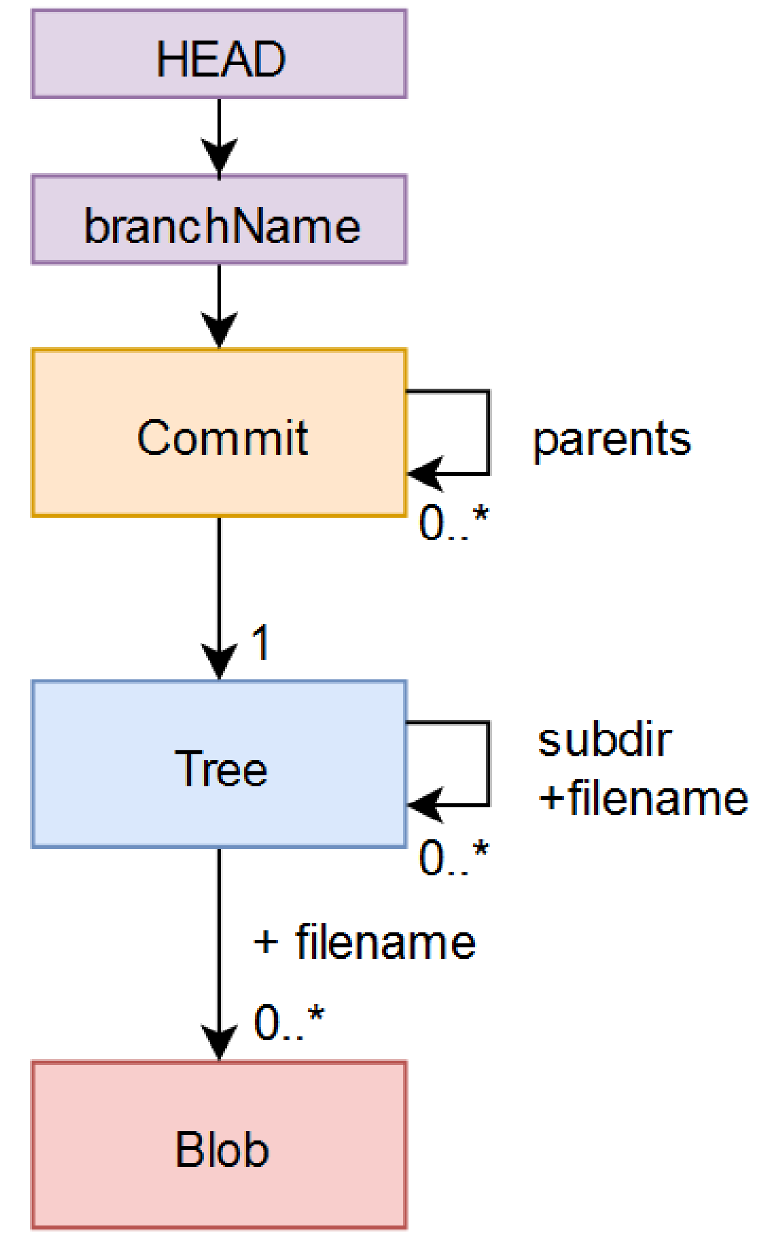


https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

git internals: refs

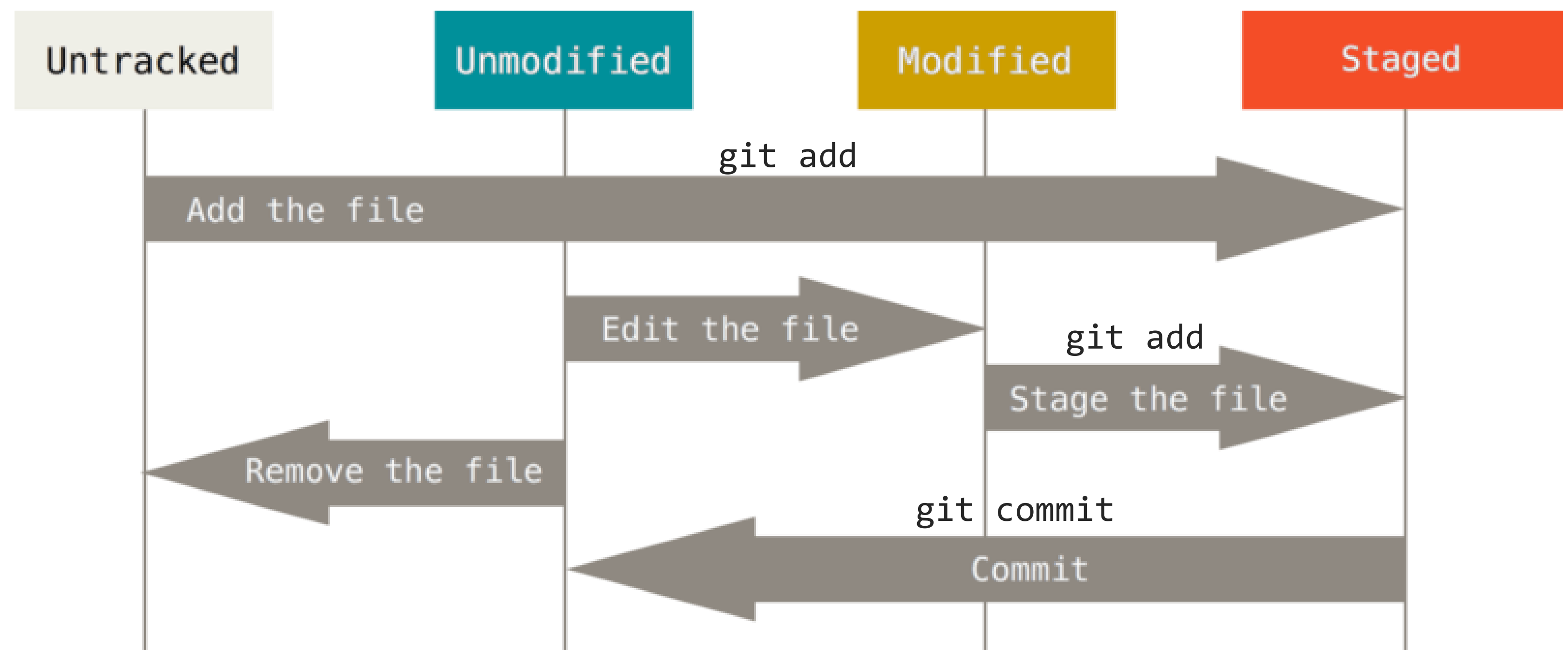
- References, or heads or branches, are **pointers** a node in the DAG.
- Unlike DAG nodes that cannot be changed, these pointers can be moved around freely.
- The **HEAD** ref is a pointer to the currently active branch.

More on git internals: [here]



Tracking changes with git

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



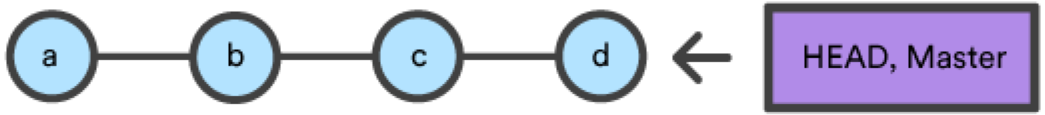
Using git status

```
>> 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
```

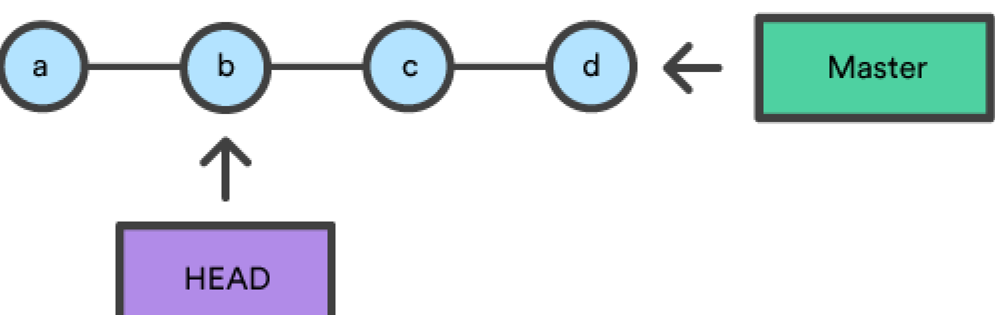
Undoing changes

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

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



>> `git checkout b`

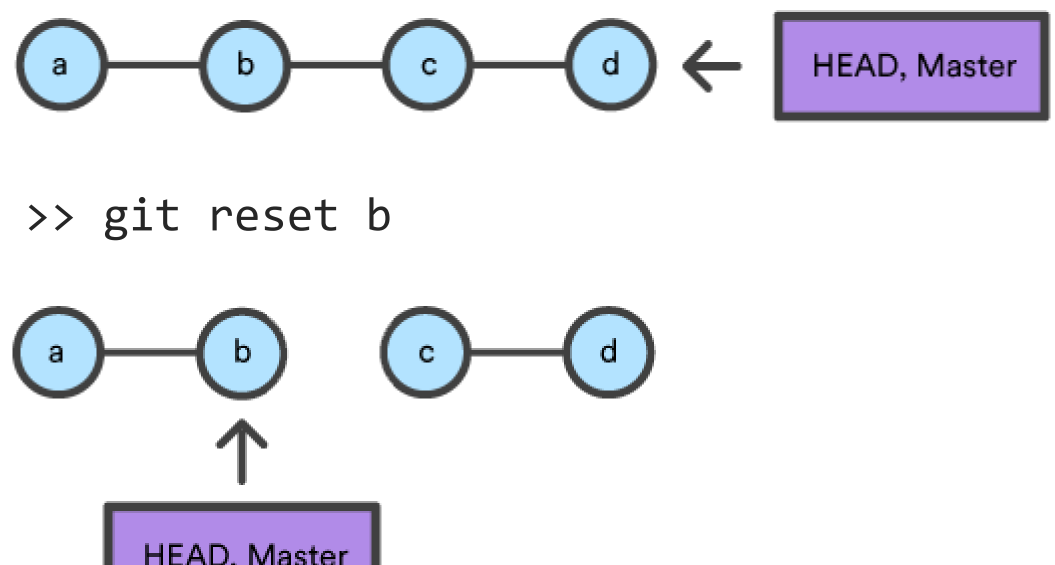


https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

Undoing changes

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

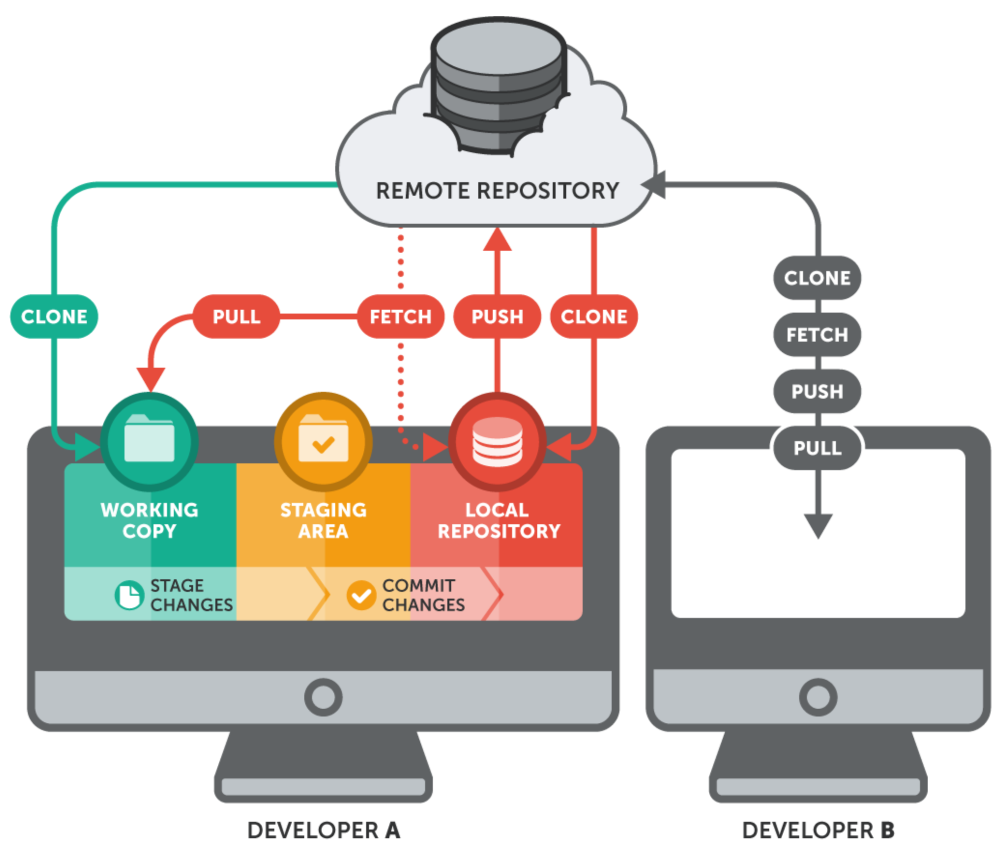
>> `git reset` moves both HEAD and current branch ref



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



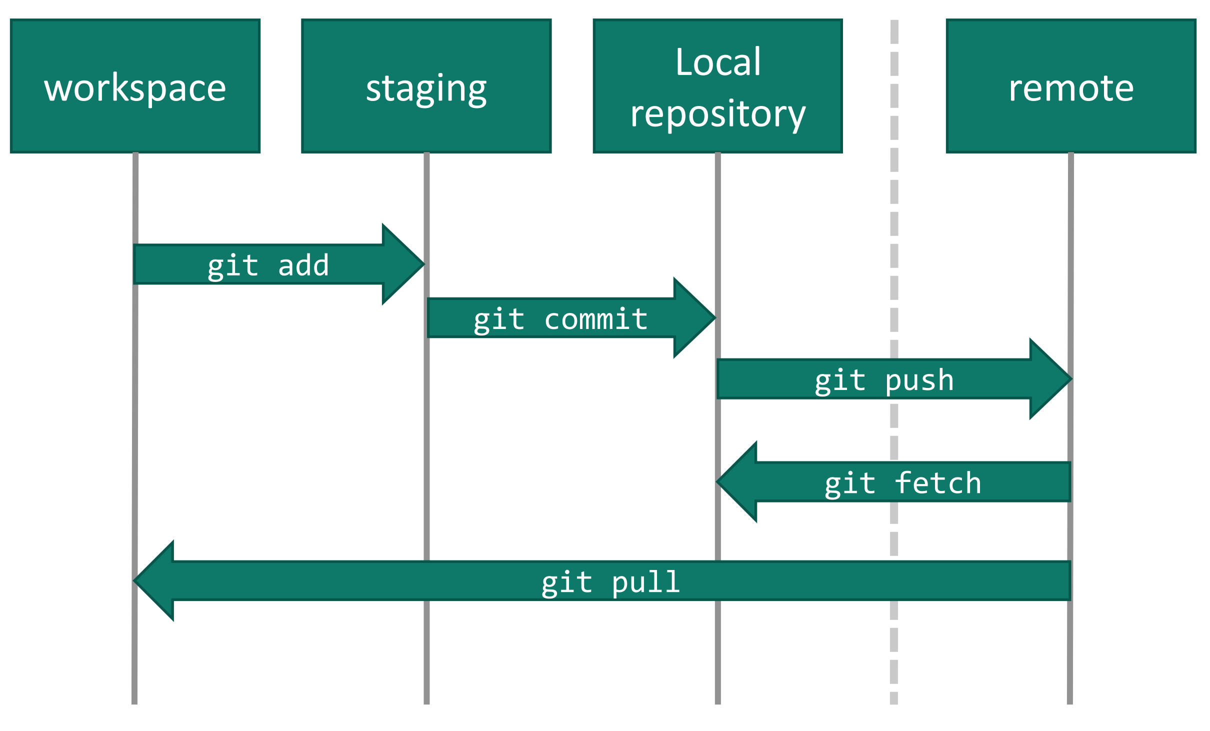
https://blog.netsons.com/git-software-guida-facile/

Listing and adding remotes

```
>> git remote
origin

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

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

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

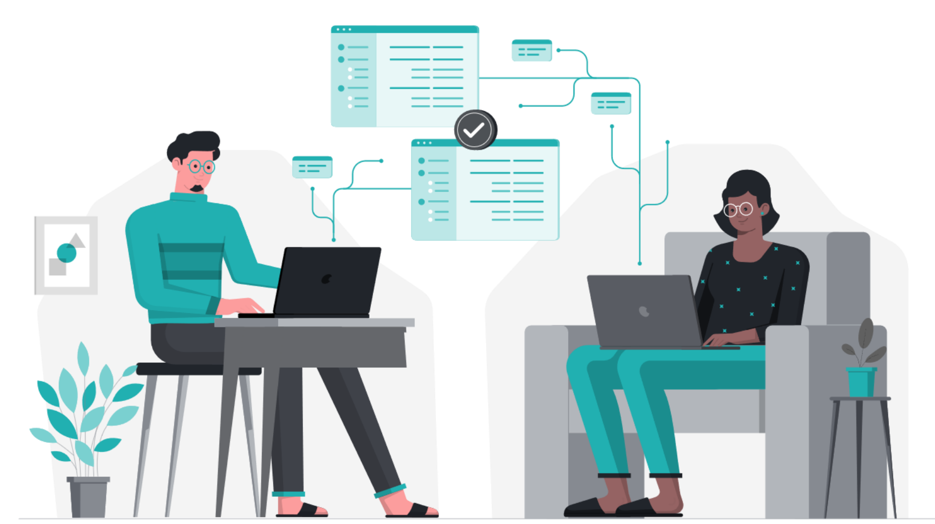
Syncing with remotes

- **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



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
- With CI/CD, the main branch should be always buildable
- **Branches** allow developers to isolate their work



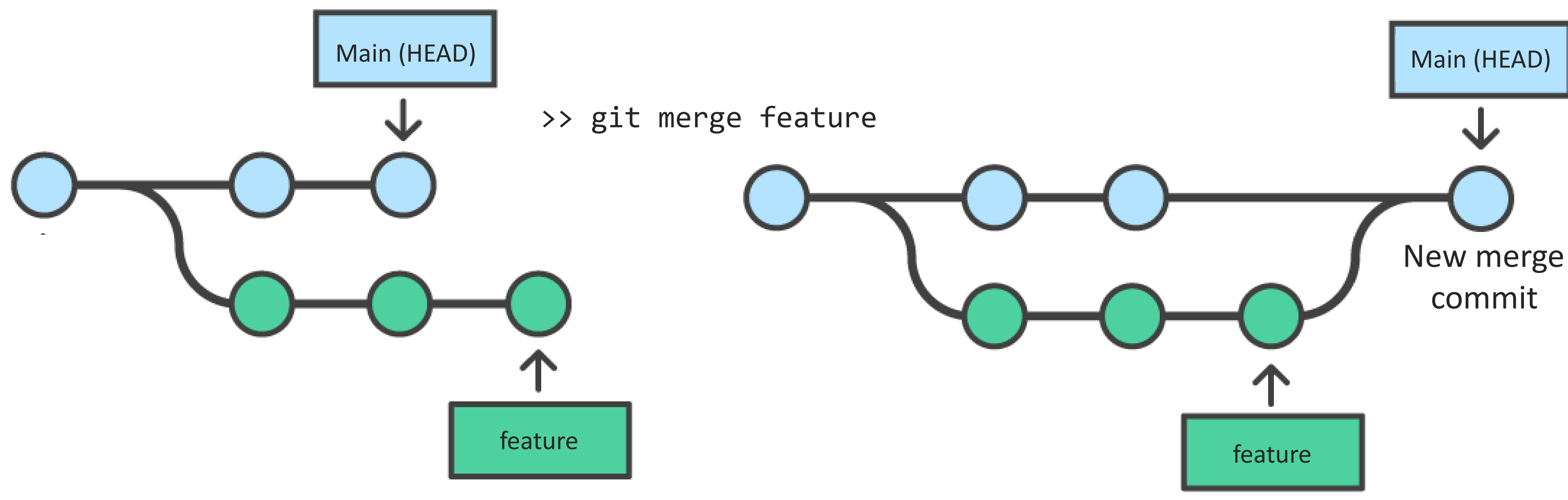
Creating a new branch

- 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

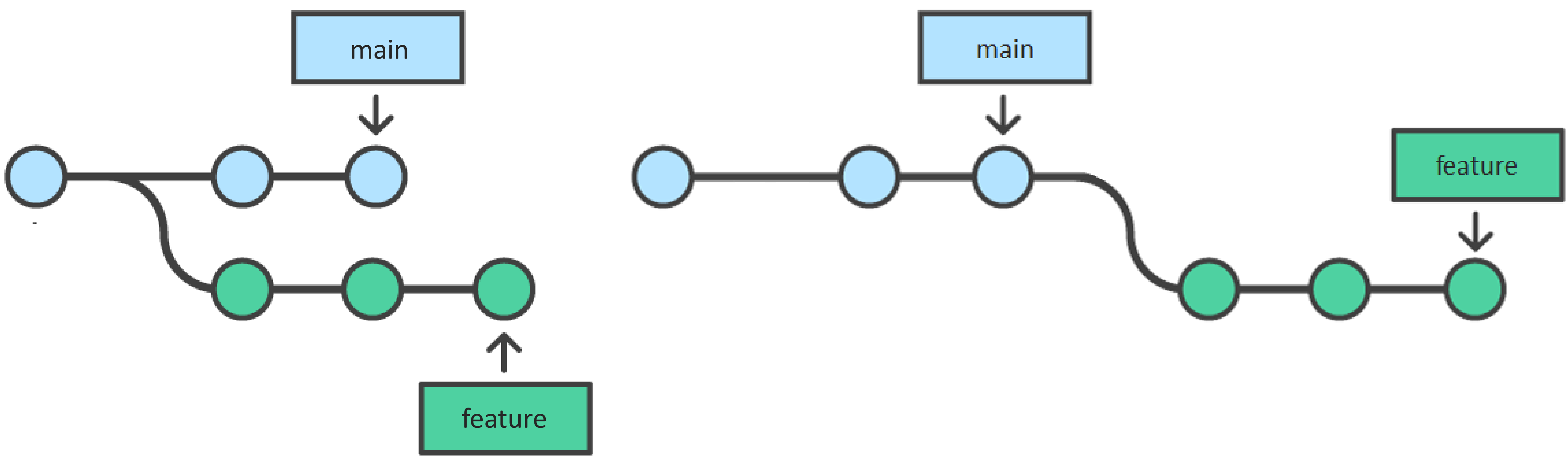


Integrating branched history: git merge

- **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