

Version Controlling and Cooperation

HUSZERL Gábor
huszerl@mit.bme.hu



Méréstechnika és
Információs Rendszerek
Tanszék



**Critical Systems
Research Group**

Learning Outcomes

- At the end of the lecture the students are expected to be able to
- (K2) summarize the basic terms of version control systems,
- (K1) utilize the Git version control system,
- (K3) differentiate typical source code integration patterns.

Further Topics of the Subject

I. Software development practices

Steps of the development

Version controlling

Requirements management

Planning and architecture

High quality source code

Testing and test development

II. Modelling

Why to model, what to model?

Unified Modeling Language

Modelling languages

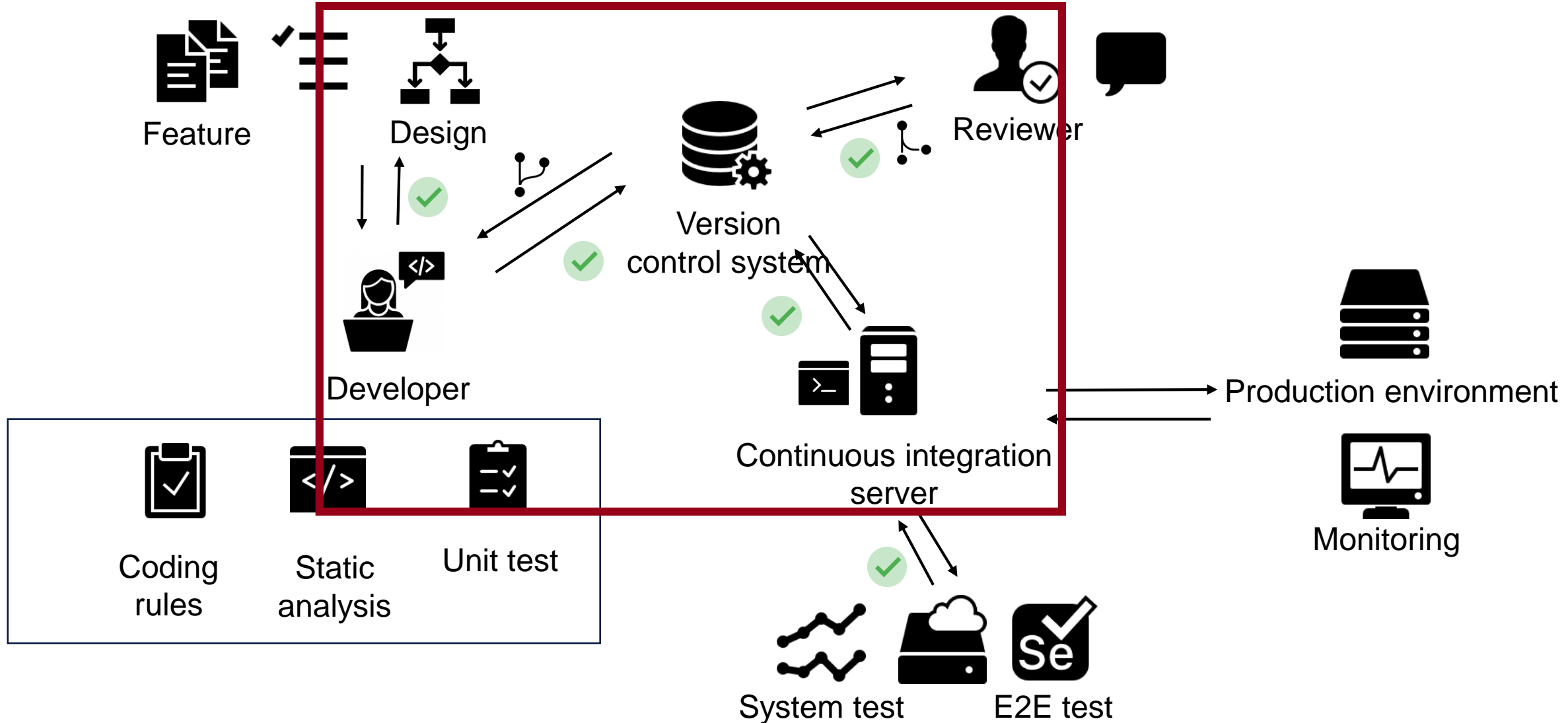
III. Processes and projects

Methods

Project management

Measurement and analysis

Typical Development Workflow

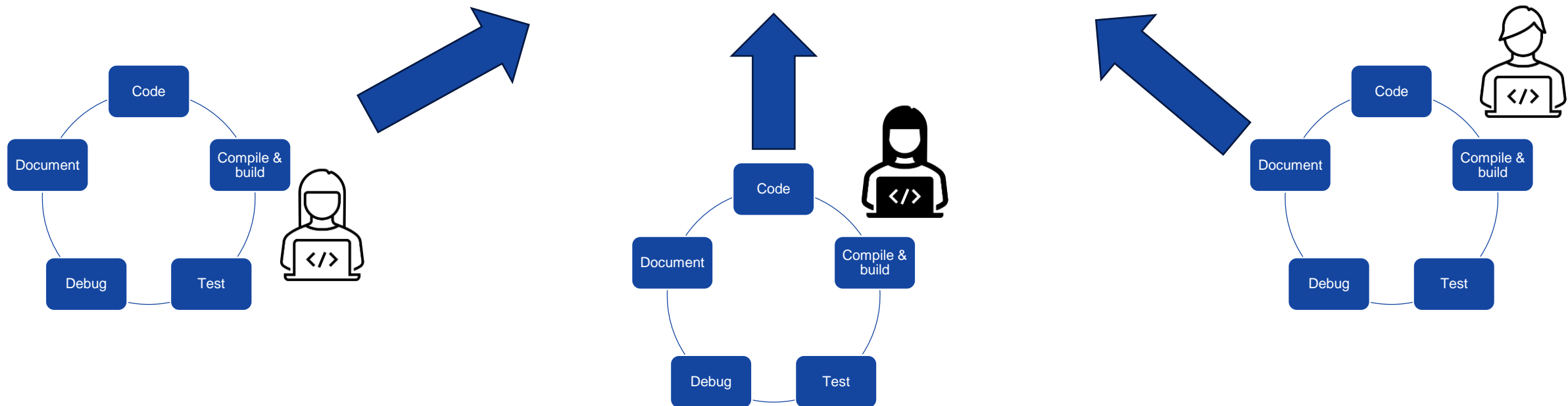


Icons: icons8.com

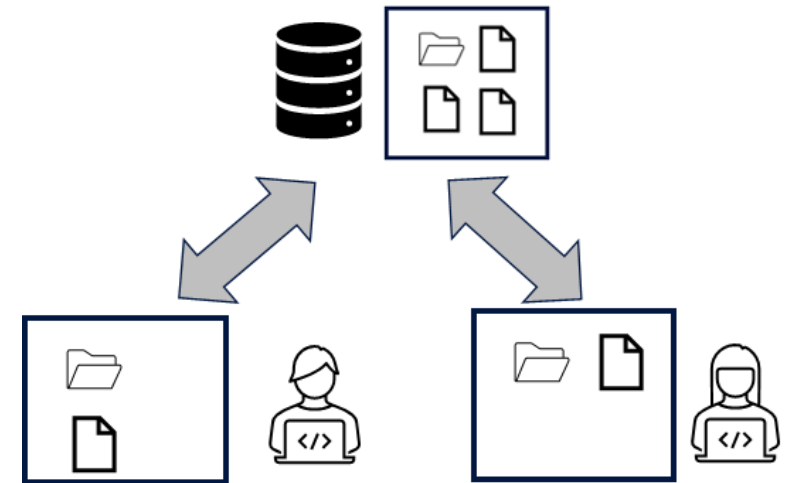
The Code Integration Problem

Until now, there was one single developer, ...

..., but in the practice multiple developers work in parallel



Basic Terms of Version Controlling

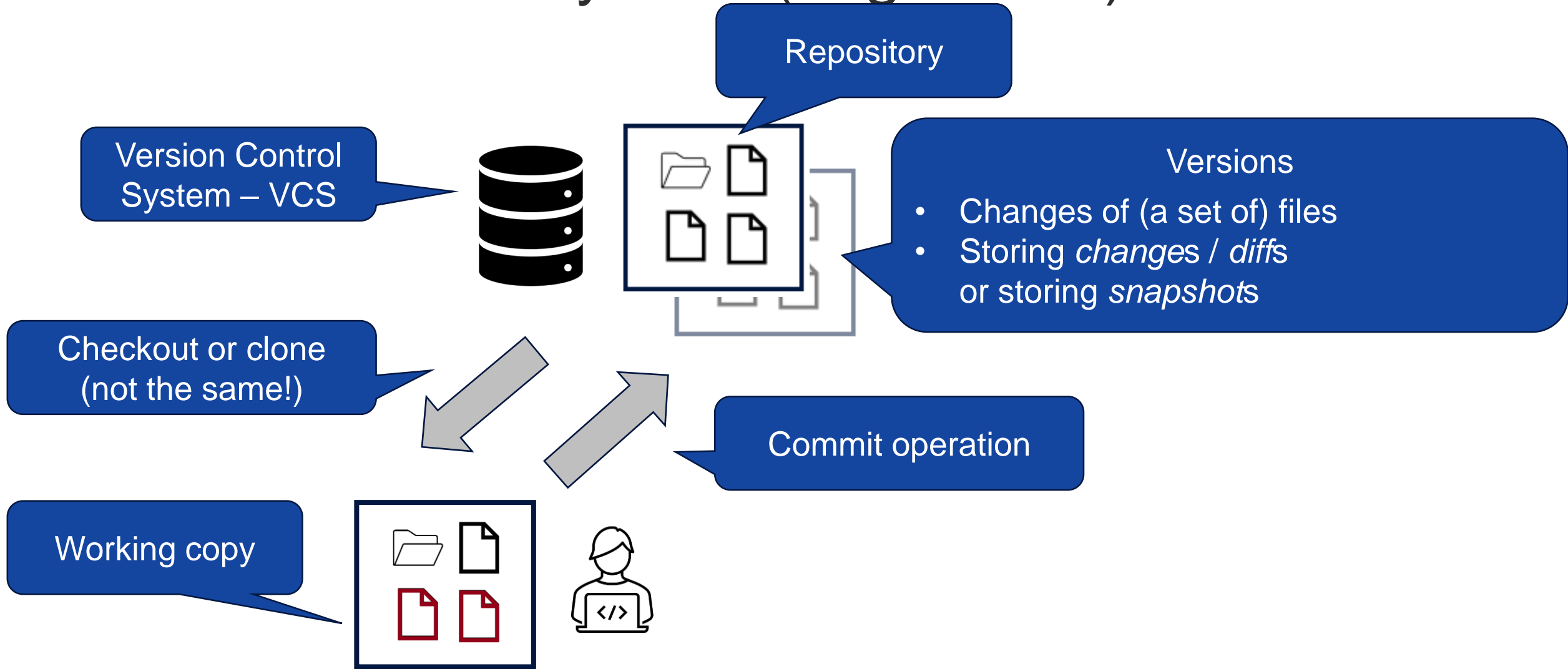


Commit, branch, merge, ...

Warning!

- There is no universal naming
- Each version control system has its own understanding of these terms
 - A “commit” in Git means something different than a “commit” in SVN
- Always read carefully the definition of terms of the system you use

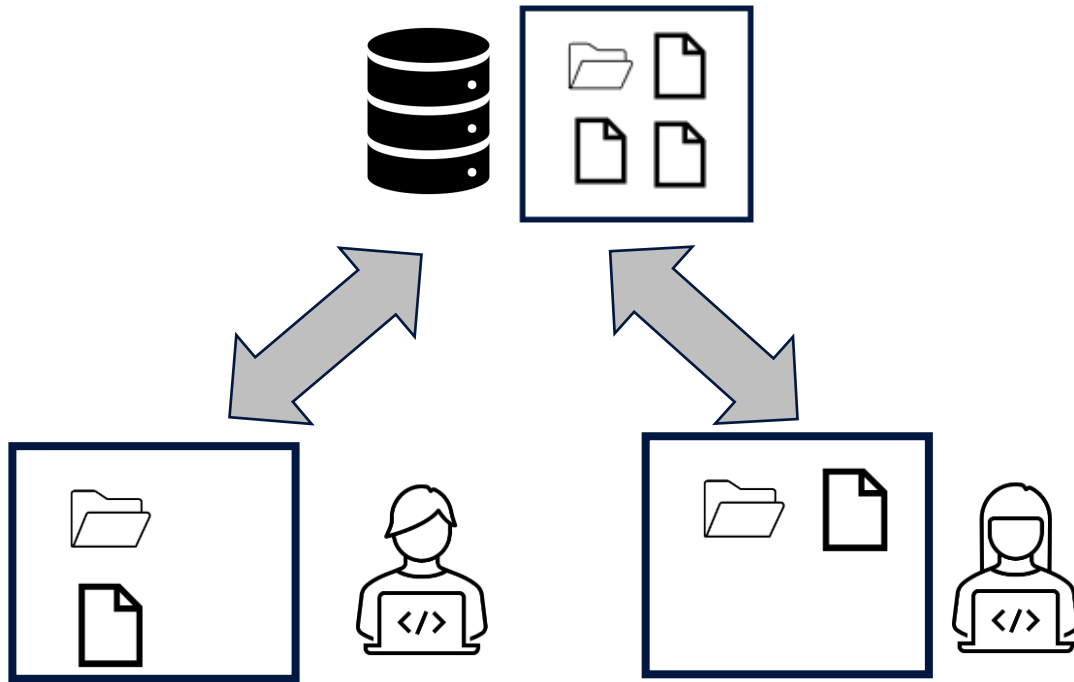
Version Control System (in general!)



Centralized and Distributed VCSs

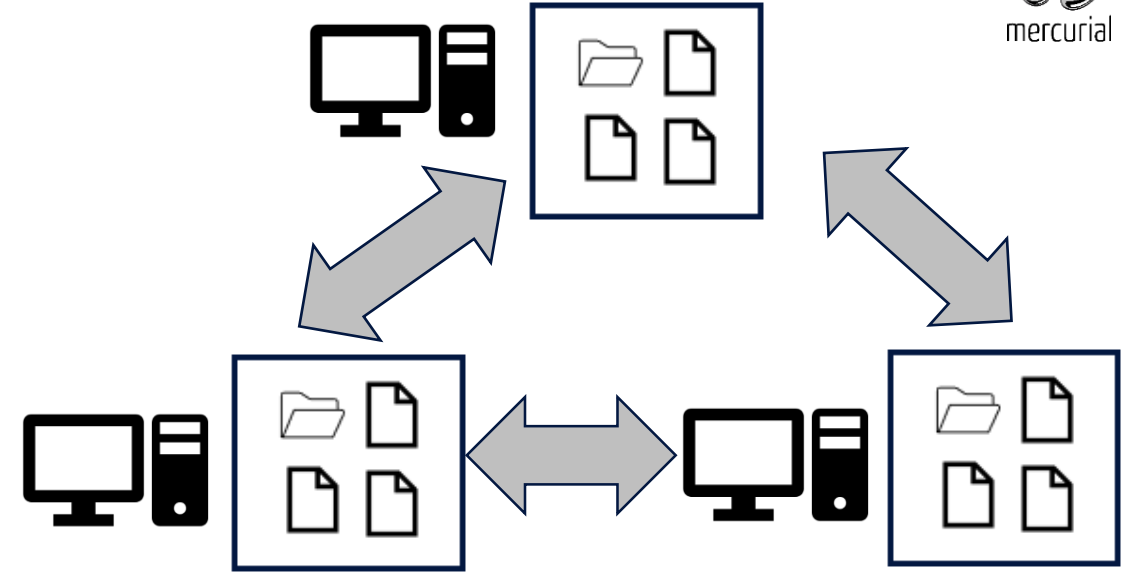


centralized



Complete history only on the central server!
Conflict management: e.g. locking

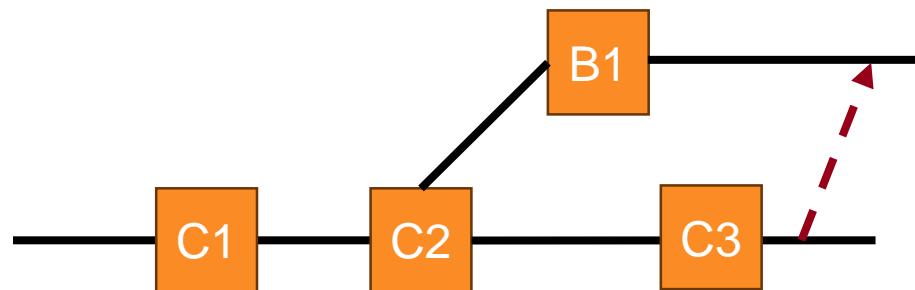
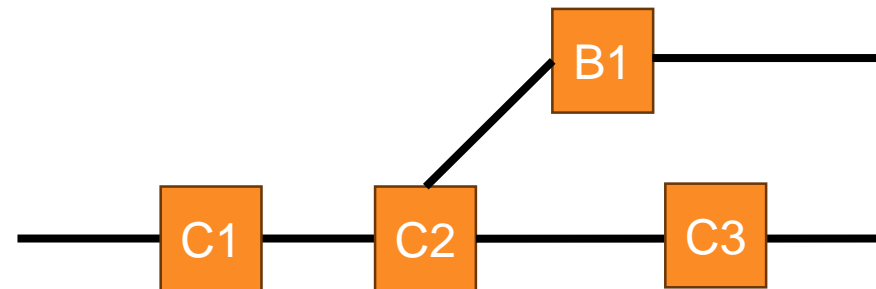
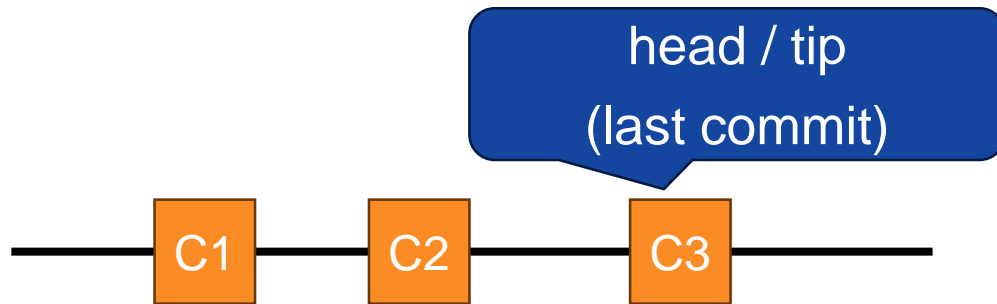
distributed



Everyone has the complete history and metadata.
We can quickly change anything locally.
Conflict management only when they occur.



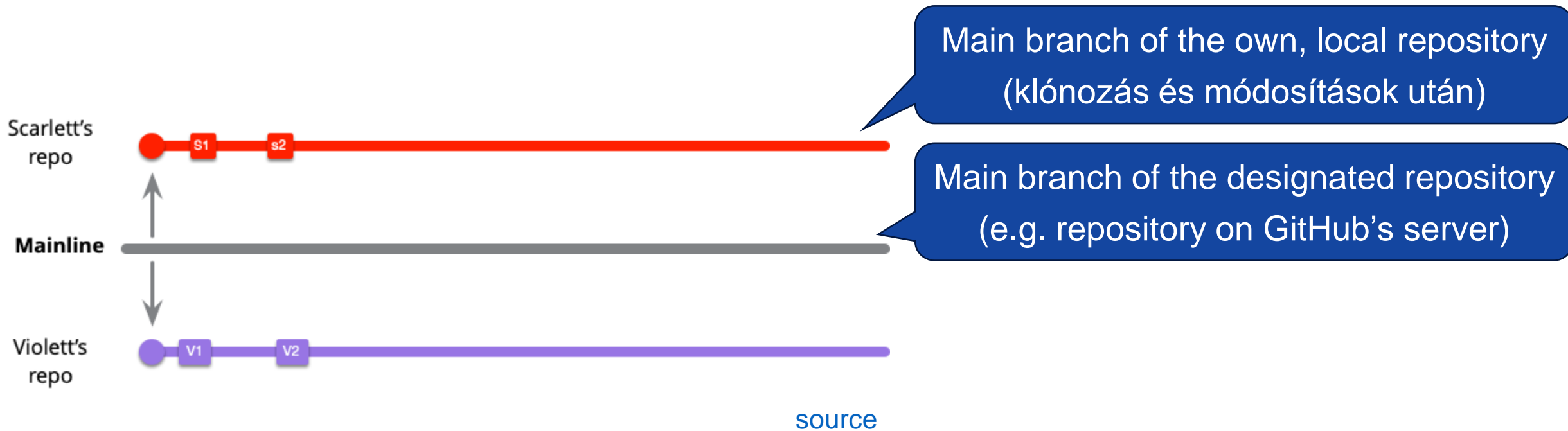
Basic Terms: Codeline, Commit, and Branch



- **commit**: related changes with metadata; defines the difference
- **codeline**: ordered series of commits
- **branch**: a new codeline, typically created when we start to work on something in parallel to others
- **trunk/mainline**: branch starting from the initial state, considered to be the common state
- **integration**: “merging” the changes from different branches
- **conflict**: if two branches have changed the same files, or other semantic collisions

Distributed VCS and Branching

A distributed version control system always has several parallel branches!
(by default, even if I do not branch!)



Configuration Management (An Outlook)

configuration management (CM): discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.

Source: „IEEE Standard for Configuration Management in Systems and Software Engineering”, IEEE 828-2012

Configuration item: any part of the system (HW, SW, documentation, configuration, ...)

Software version management is a part of this activity

Git Version Control System



Source: S. Chacon, B. Straub. „Pro Git”, 2nd edition, 2014

Git Version Control System

- New VCS specifically to support the Linux kernel (2005-)
- Design goals
 - Distributed operation
 - Cheap and comfortable branch management
 - Fast operation even on a large amount of data
 - Data and integrity protection

Git: Storing Snapshots

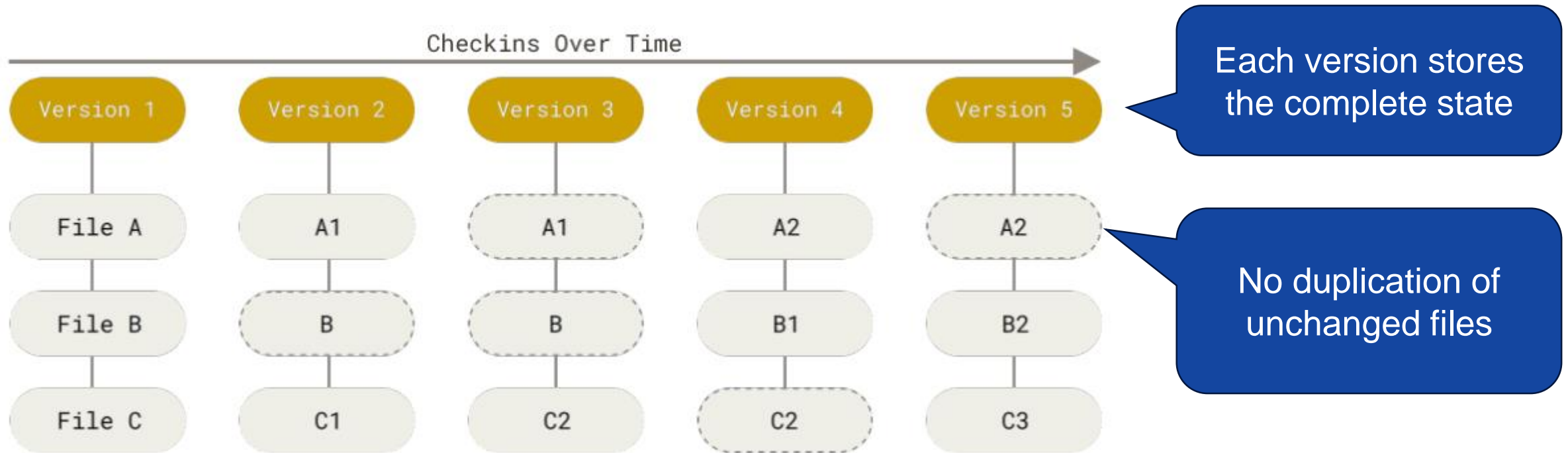


Figure 5. Storing data as snapshots of the project over time

Components of the Local Rep

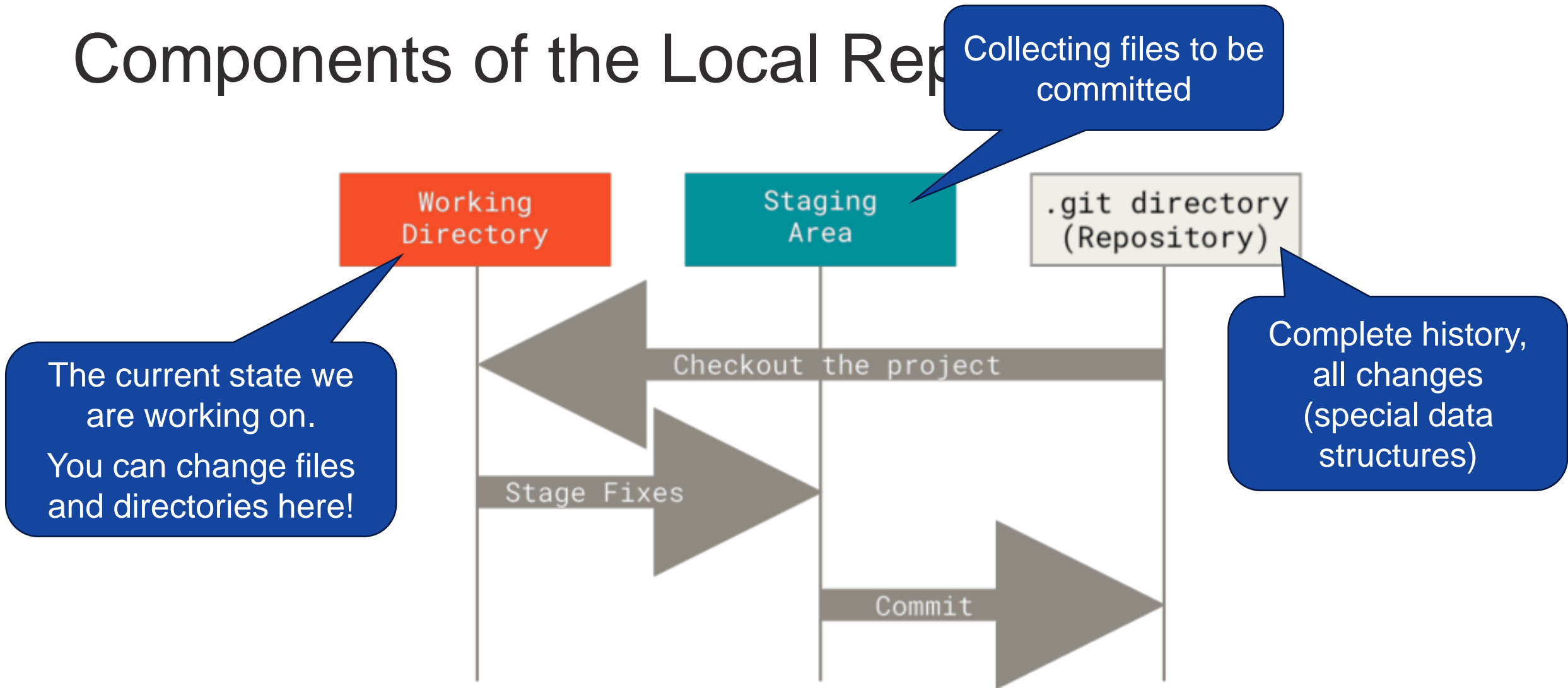


Figure 6. Working tree, staging area, and Git directory

[Source](#)

The Possible States of Files

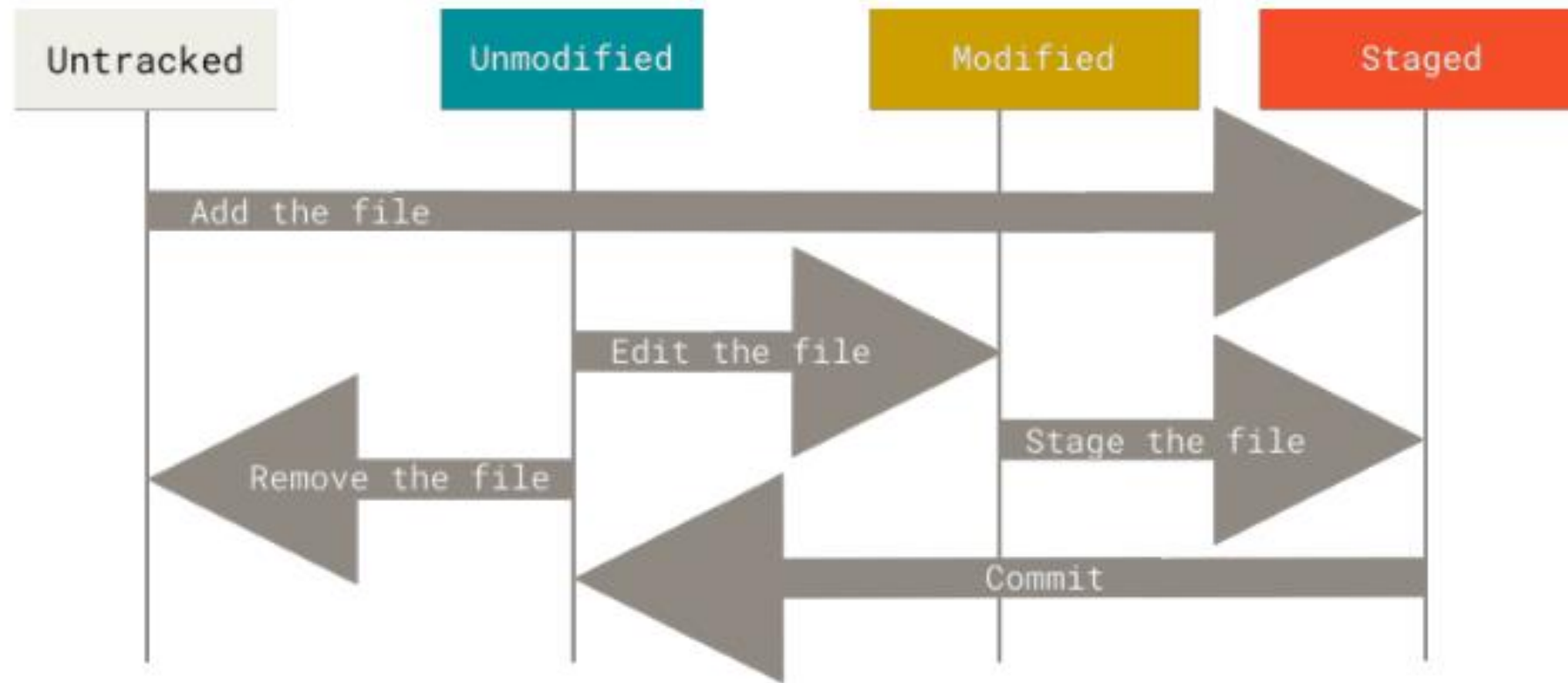


Figure 8. The lifecycle of the status of your files

[Source](#)

Commit: A Snapshot and its Metadata Together

Initial commit

Identifier of the commit:
SHA-1 hash checksum

Parents: defining the order

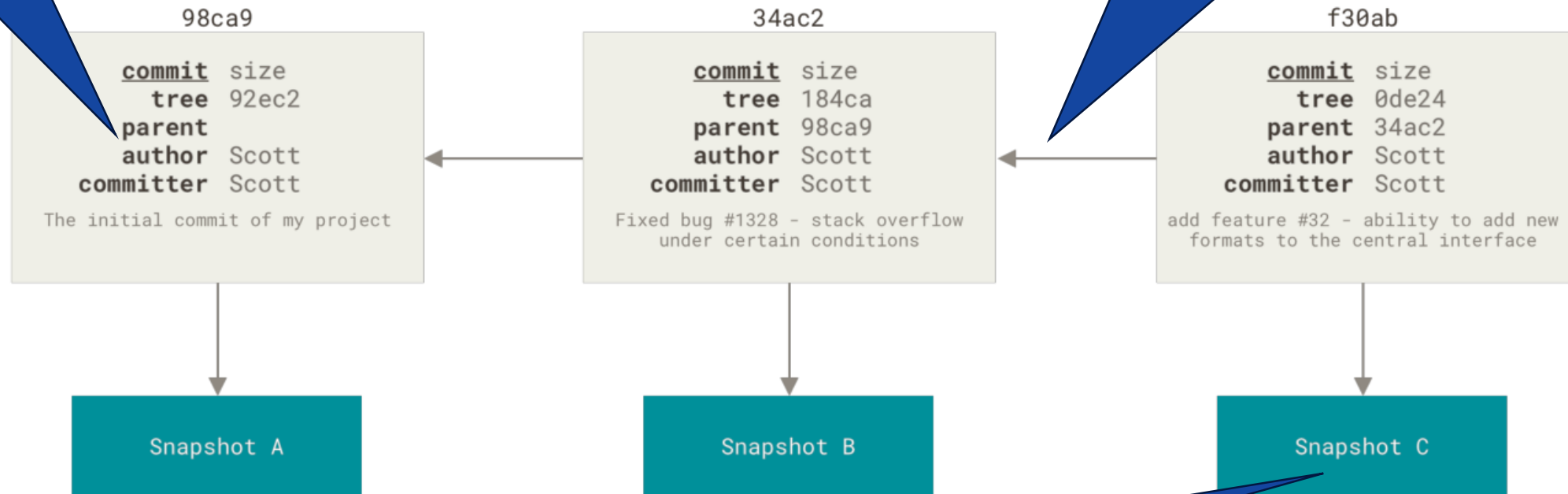


Figure 10. Commits and their parents

Internal data structure:
object, tree, ...

Git: Commit and Branch

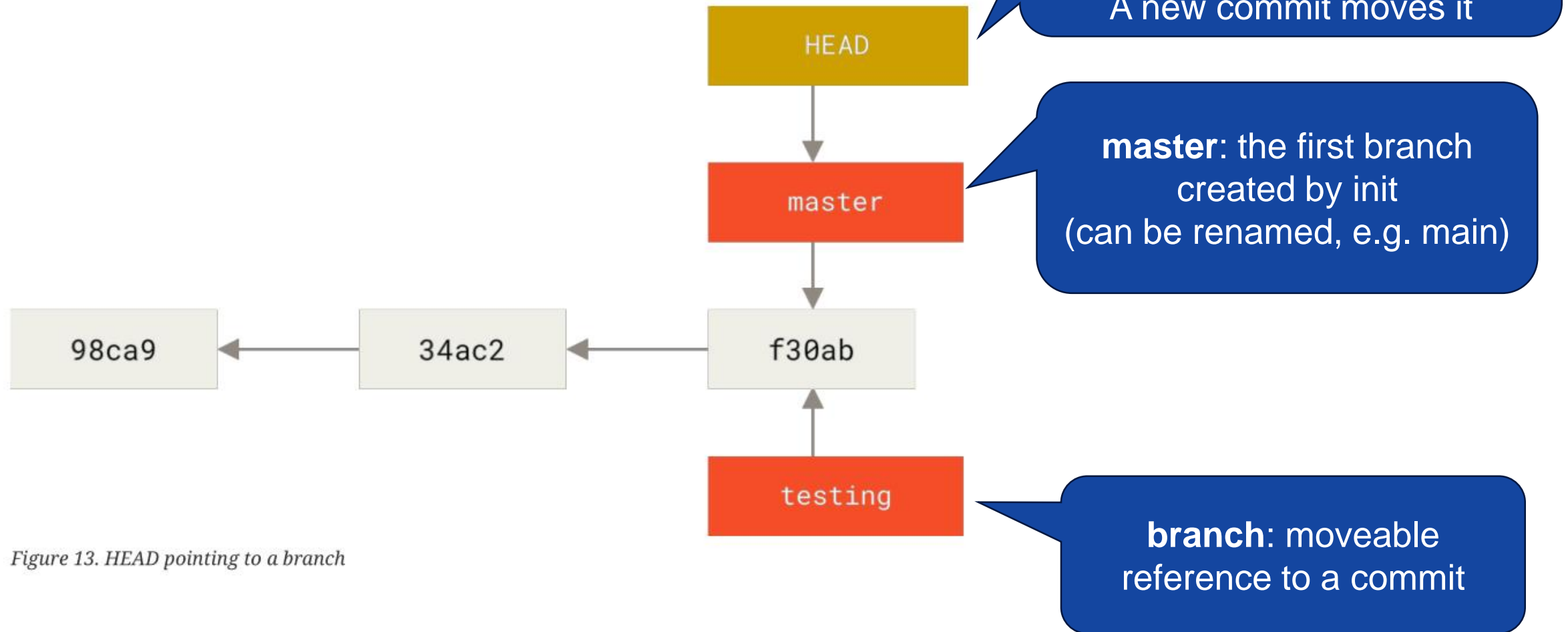


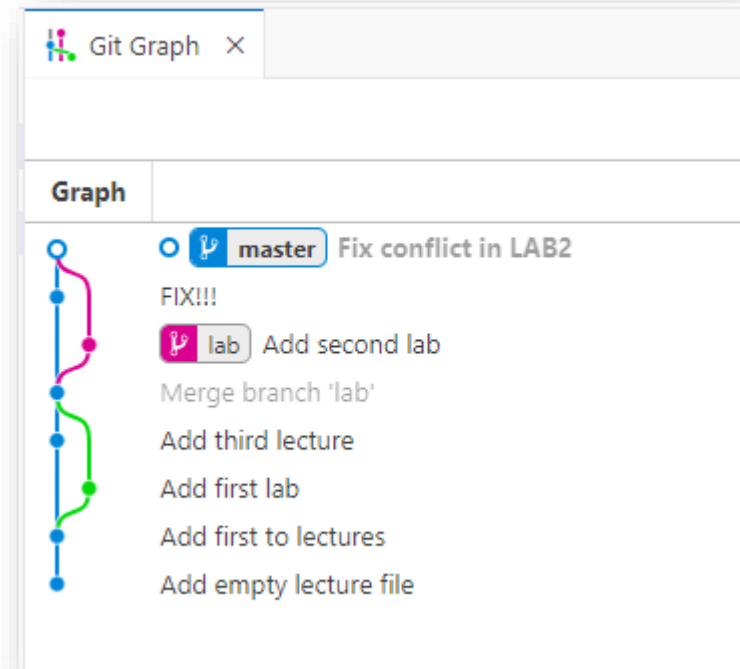
Figure 13. HEAD pointing to a branch

DEMO: The Basics of Git

```
PS C:\files\code\git-softeng> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   lab.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\files\code\git-softeng> cat .\lab.txt
LAB1 Complex
<<<<<< HEAD
LAB2 Git and GitHub
=====
LAB2 Git
>>>>>> lab
PS C:\files\code\git-softeng> git add .\lab.txt
PS C:\files\code\git-softeng> git commit -m "Fix conflict in LAB2"
[master d26f2da] Fix conflict in LAB2
PS C:\files\code\git-softeng> git status
On branch master
nothing to commit, working tree clean
```



The Most Important Git Commands (1)

Initializing an empty repository

```
$ git init
```

Tracking and committing a file

```
$ git add *.c  
$ git add LICENSE  
$ git commit -m 'Initial project version'
```

Cloning a (complete) remote repository

```
$ git clone https://github.com/libgit2/libgit2
```

State of the local repository

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
nothing to commit, working tree clean
```

The Most Important Git Commands (2)

The state if there are some not traced and some changed files

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

The Most Important Git Commands (3)

Checking commit history

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
```

Change version number

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
```

Remove unnecessary test

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700
```

Initial commit

Commit hash

Commit message

The Most Important Git Commands (4)

Querying the remote repository

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

Adding a remote repository

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

Querying the changes in a remote repository

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch]      master    -> pb/master
* [new branch]      ticgit     -> pb/ticgit
```

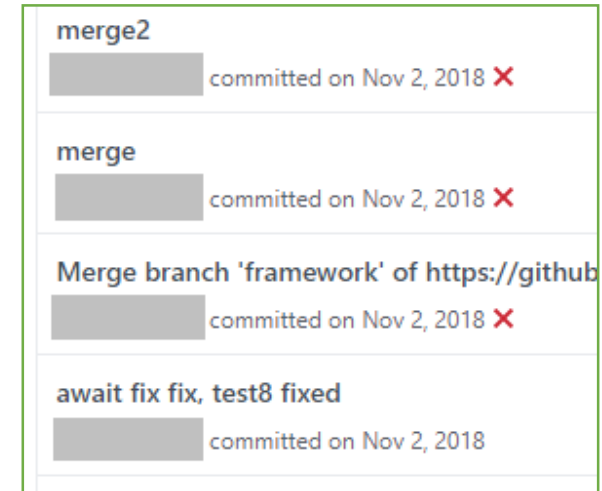
Sending the changes in the local repository

```
$ git push origin master
```

Commit Advices

Bad example: what has happened here?

commit == a related set of changes
→ rather many, smaller commits



Advice on Git commit message

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); some git tools can get confused if you run the two together.

Further paragraphs come after blank lines.


- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

Good example: a story to follow


var: avoid a segmentation fault when HOME is unset ...

 dscho authored and gitster committed last week

sequencer: fix error message on failure to copy SQUASH_MSG ...

 ossilator authored and gitster committed last week


parse-options: mark unused parameters in noop callback ...

 peff authored and gitster committed last week

interpret-trailers: mark unused "unset" parameters in option callbacks ...

 peff authored and gitster committed last week

parse-options: add more BUG_ON() annotations ...

 peff authored and gitster committed last week

Further Git Functions

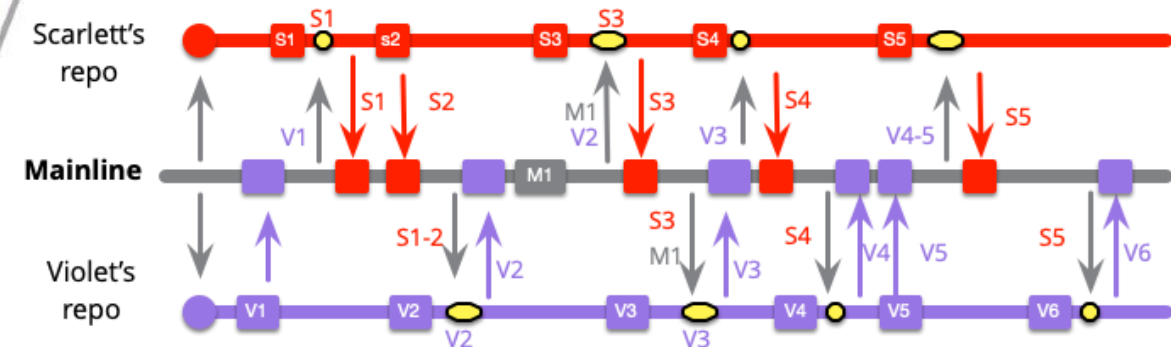
- Tag

- Labelling a commit
- Annotated tag: storing extra metadata
taggers name and email, a tag message, the date, a new hash, ...
- E.g.: tagging a release (1.0.0)

- Git hook

- Running scripts for specific events
- E.g.: pre-commit checks – testing, checking code style, ...
(if the check fails, the changes cannot be committed!)

Version Controlling and Cooperation Patterns



[Source](#): M. Fowler. „Patterns for Managing Source Code Branches”

Basic Pattern: Mainline

A single, shared, branch that acts as the current state of the product

- „Single Source of Truth”
- Everyone starts from here, all accepted changes will be merged here
- Distributed VCS: usually one repository is selected to be the mainline (E.g. central Git server of the company, repository on GitHub, ...)

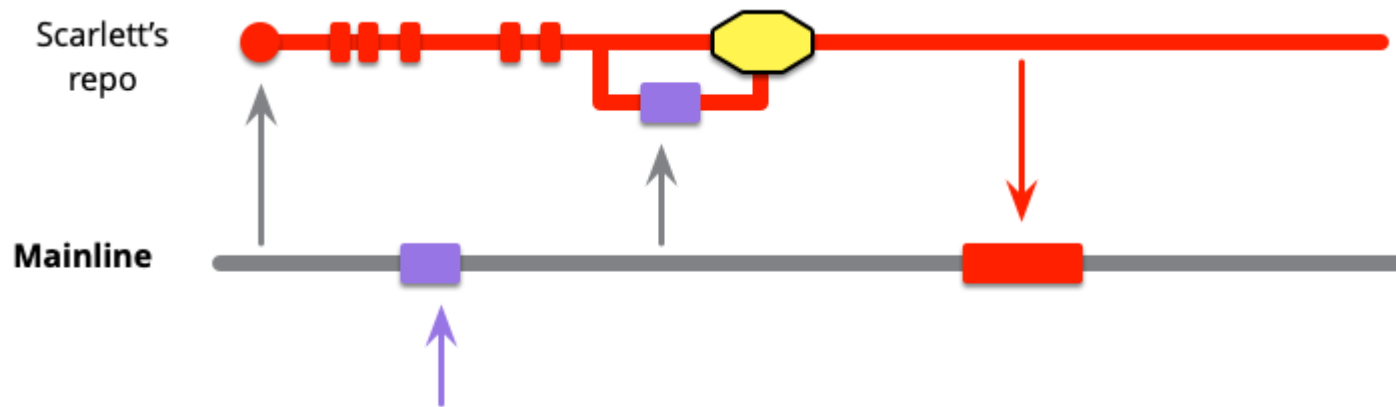
Basic Pattern: Healthy Branch

On each commit, perform automated checks, usually building and running tests, to ensure there are no defects on the branch

- It is extremely important, if there is a mainline
- If there is an error, the branch must be repaired as soon as possible
- Precision \Leftrightarrow speed of checking

Integration Pattern: Mainline Integration

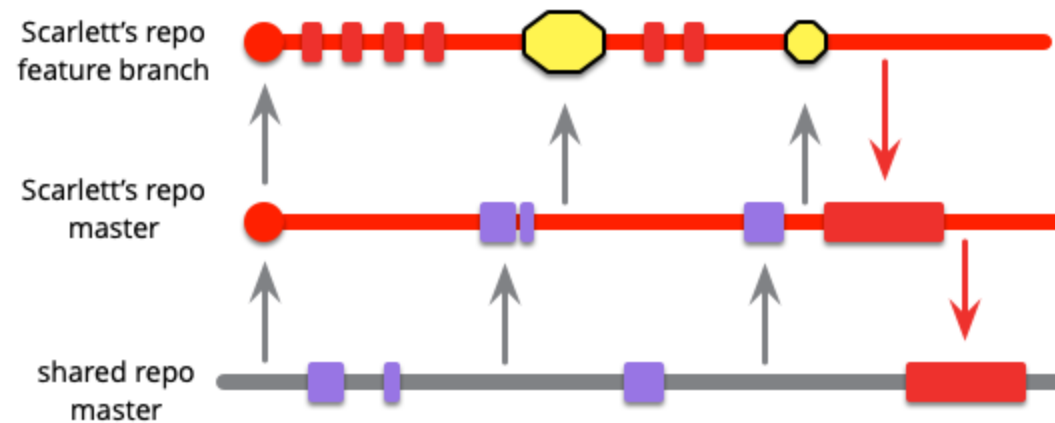
Developers integrate their work by pulling from mainline, merging, and - if healthy - pushing back into mainline



- During the development, integrate from the mainline (pull)
- At the end of development, integrate to the mainline (push)

Integration Pattern: Feature Branching

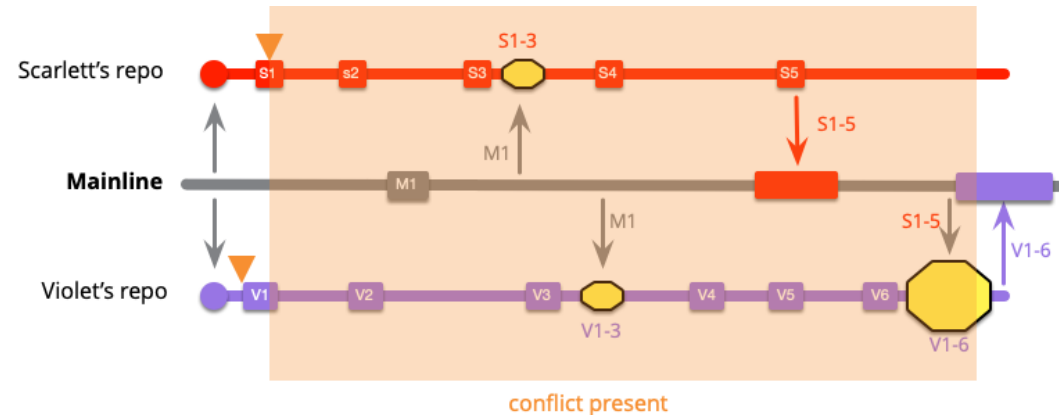
Put all work for a feature on its own branch, integrate into mainline when the feature is complete.



- New branch for each development tasks (new feature, bugfix, ...)

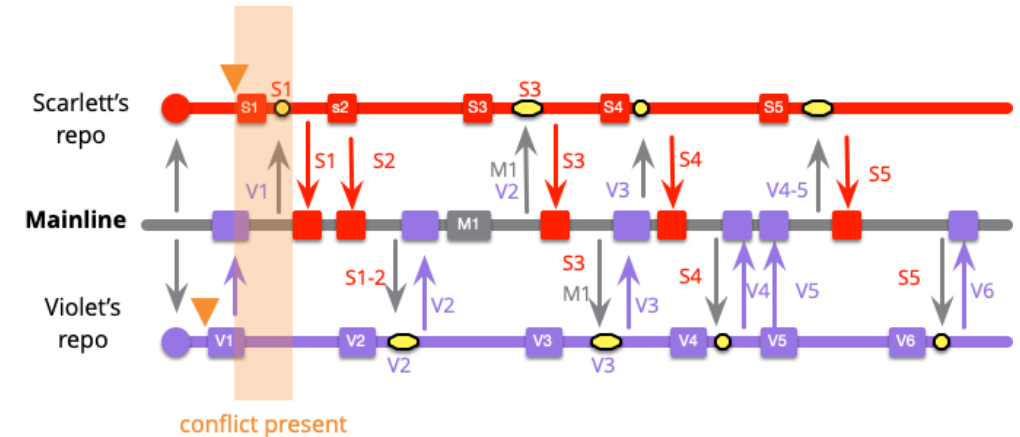
Frequency of Integration

Rare



- Long life feature or development branches
- Collisions will be revealed later, merging is more complicated

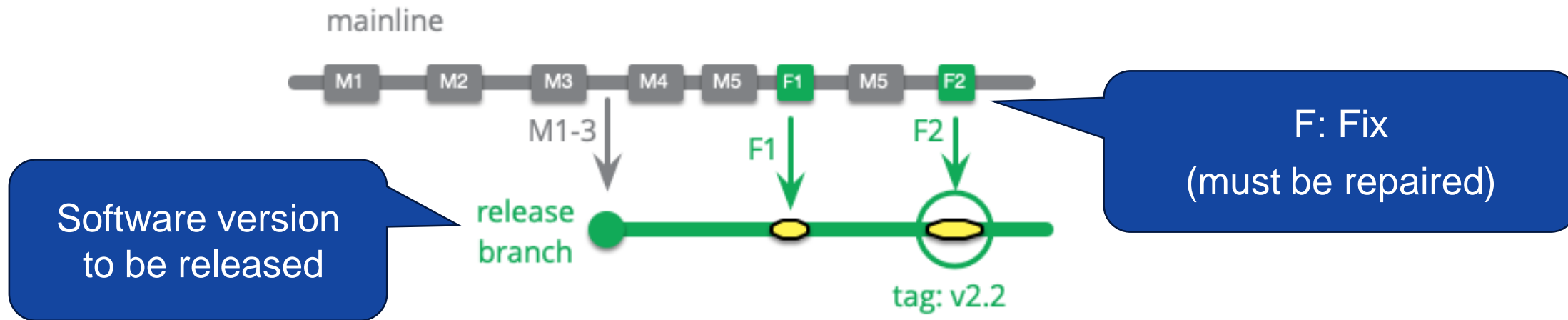
Frequent



- Frequent mainline integration
- Collisions will be revealed early
- Helps one to see what the others are working on

Release Pattern: Release Branch

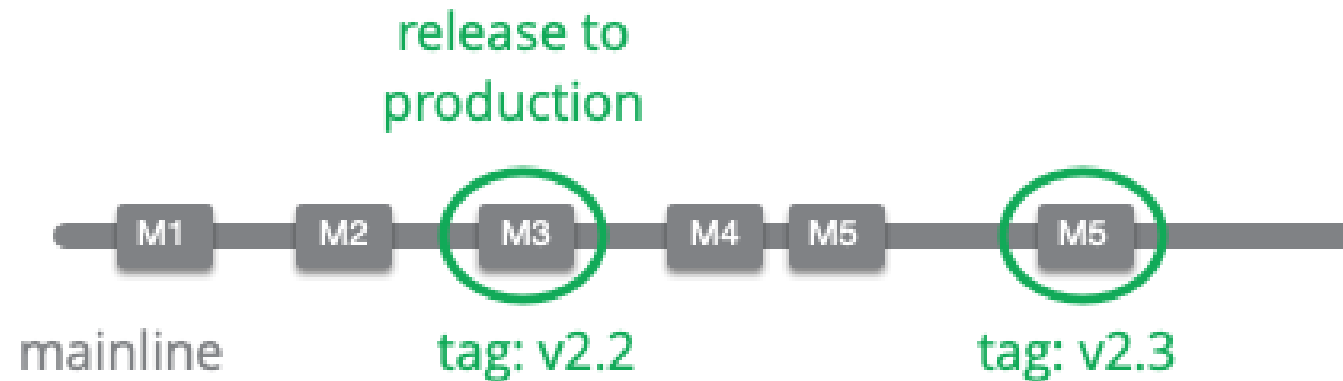
A branch that only accepts commits accepted to stabilize a version of the product ready for release.



- There are no new features on the release branch, just necessary repairs
- Tag: individual versions can be marked

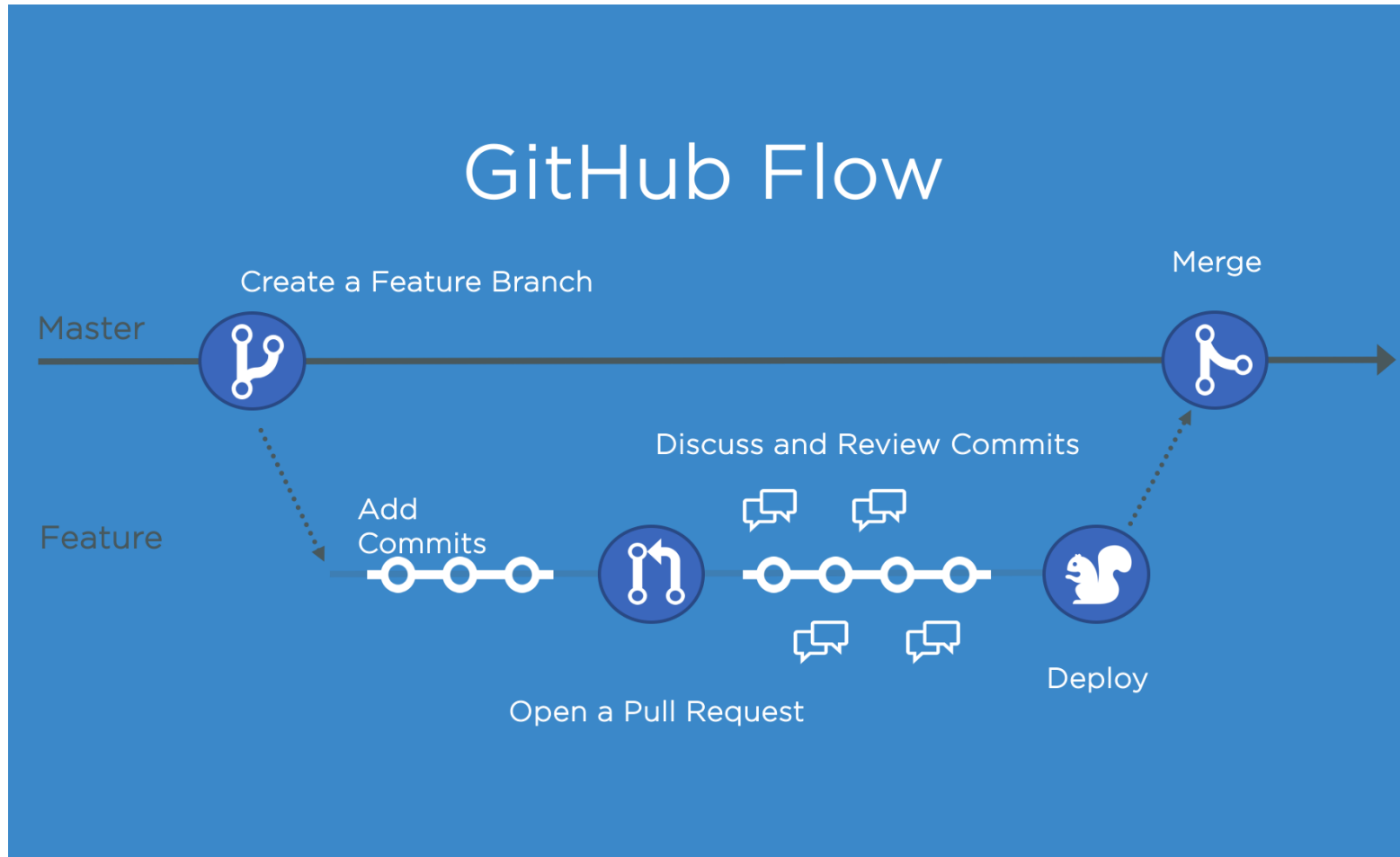
Release Pattern: Release-Ready Mainline

Keep mainline sufficiently healthy that the head of mainline can always be put directly into production



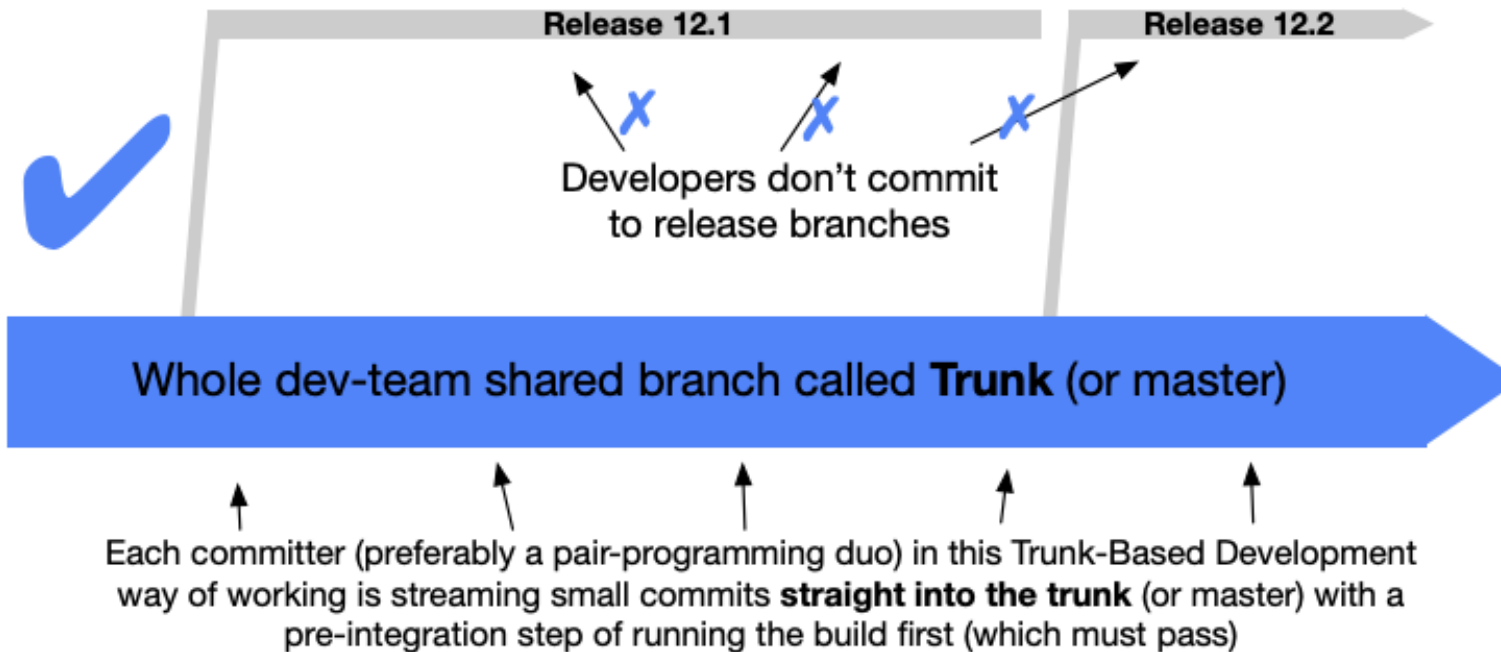
- See: Continuous Delivery and Continuous Deployment
- (based on experience, it results in frequent, successful releases)

Workflow: GitHub Flow



- Feature branch + Release-ready main
- Pull request: GitHub function; it combines integration and review
- 🍷 Open source development (independent contributions)
- (Do not confuse with: Git-flow)

Workflow: Trunk-based Development



[source](#)

- Mainline integration + continuous integration
- For a larger team:
1-2 days feature branch
- 👍 Team members working closely together



Summary

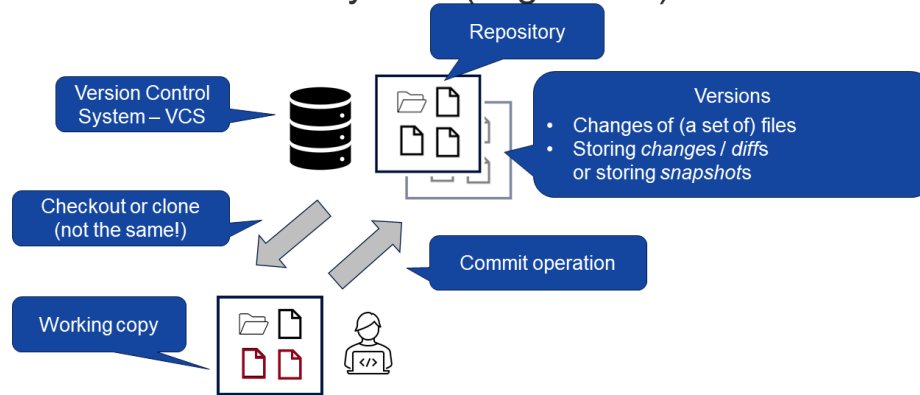
Version Controlling and Cooperation

How To Learn Using Git?

- LAB2
- Practice, practice, practice 😊
- Git CheatSheet (<https://training.github.com/>)
- Git Reference (<http://git.github.io/git-reference>)
- Pro Git - Book (<https://git-scm.com/book>)

Summary

Version Control System (in general!)



Software Engineering (VIMIAB04)



Git: Commit and Branch

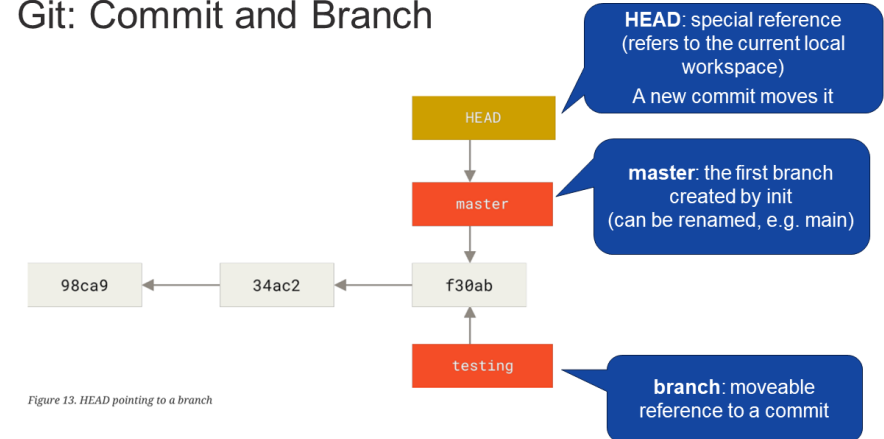


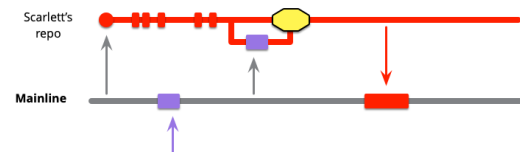
Figure 13. HEAD pointing to a branch

Software Engineering (VIMIAB04)



Integration Pattern: Mainline Integration

Developers integrate their work by pulling from mainline, merging, and - if healthy - pushing back into mainline

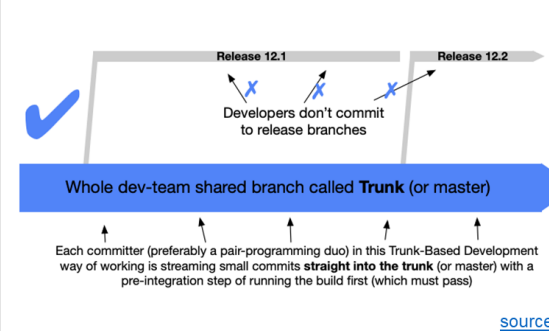


- During the development, integrate from the mainline (pull)
- At the end of development, integrate to the mainline (push)

Software Engineering (VIMIAB04)



Workflow: Trunk-based Development



- Mainline integration + continuous integration
- For a larger team: 1-2 days feature branch
- Team members working closely together

[source](#)

Software Engineering (VIMIAB04)

