Source Code Versioning

A git beginners workshop

Agenda

- Version Control Systems
- What is git
- Git Workflow
- Workspace, Staging, commit
- .gitignore
- Branching and merging
- Work with remote repositories

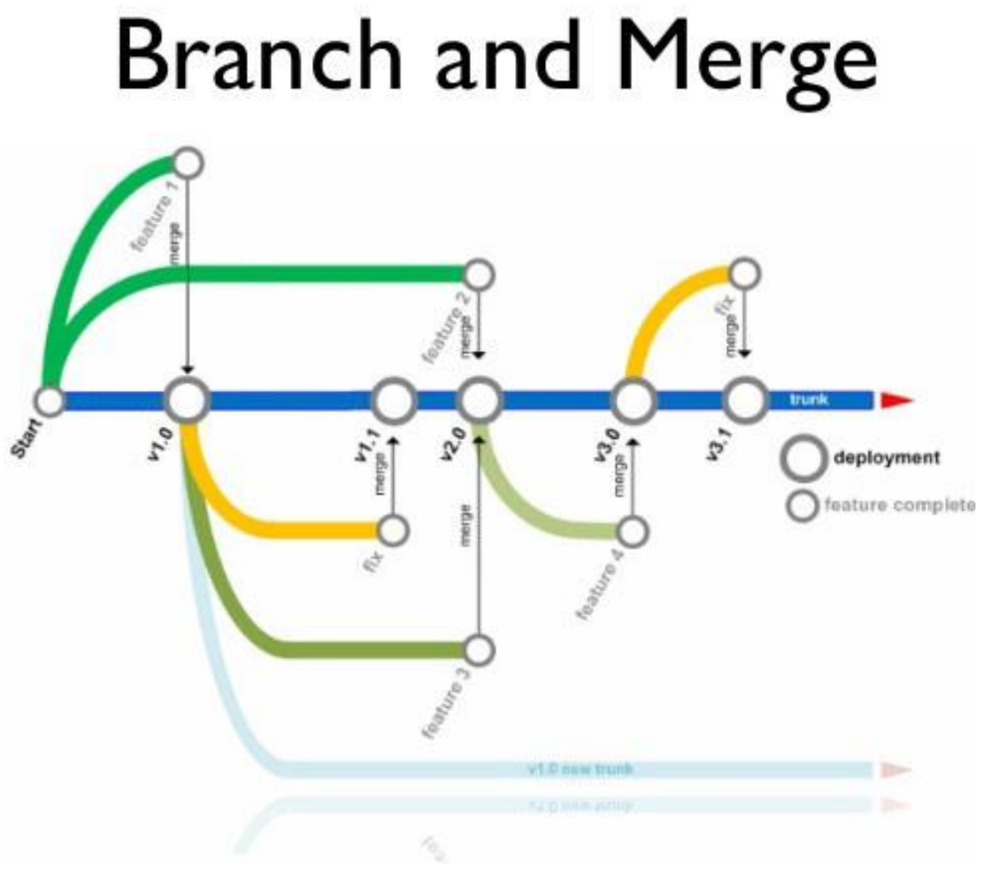
THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

https://xkcd.com/1597/

Using Version Control (Systems)

A **Version Control System (VCS)** records changes to a file or set of files over time so that you can recall specific versions later.

- Retains, and provides access to, **every version** of **every file** that has ever been stored in it.
- Provides **metadata**, like commit messages, attached to single files or collections of files.
- Allows teams that may be **distributed** across space and time to **collaborate**.



Practices of using Version Control (Systems)

- **Keep absolute everything in the version control**
- Check in regularly
- Use **meaningful** commit messages
- Track changes

https://devrant.com/search?term=git+commit+fire

There are more than only Source Code Files

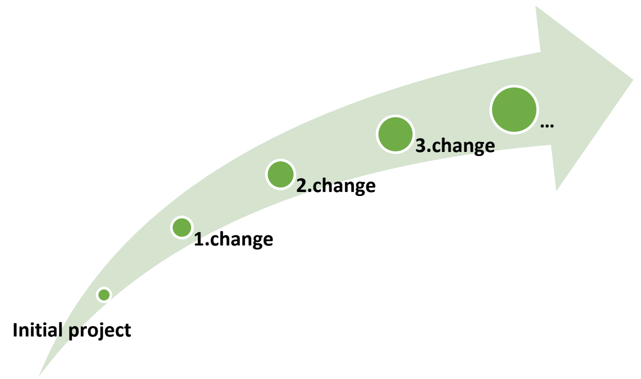
Each project contains different file types, so we need to think about how to create a structure to work with a long time

• **Java source**	• ***.java**
• Java class	• *.class
• Java archive	• *.jar
• Configuration	• *.properties, *.xml, *.json
• Documentation	• *.txt, *.md
• Images	• *.png, *.jpg
• …	• …

Java is used here exclusively as an example, this applies to all common software projects.

Keep track of project changes in software development

- **Records changes** to (all) files, like add, remove, modify, … any content of (nearly) any file.
- Access to **specific version** over time, which you have tracked with Source Code Versioning, e.g. git
- **Revert** to a previous state in case of bugs, problems, … and stop working with "comments" of deprecated sources code
- **Compare changes**, to find differences between version, it's important for debugging, bugfixing, … e.g. compare version 23 with version 54 which lines are different and will lead to a failure
- **Tracks changes** and who have done those changes, in case of questions you are able to identify current and last developers



First touchpoints in Real Life

How have you done versioning …

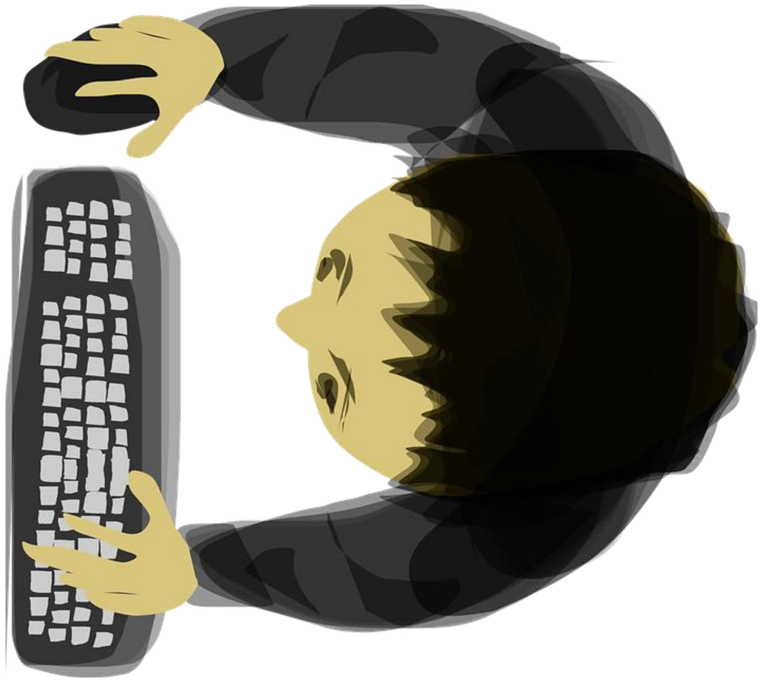
… at your assignments?

… at lab exercises?

… at a previous project?

… at work?

…

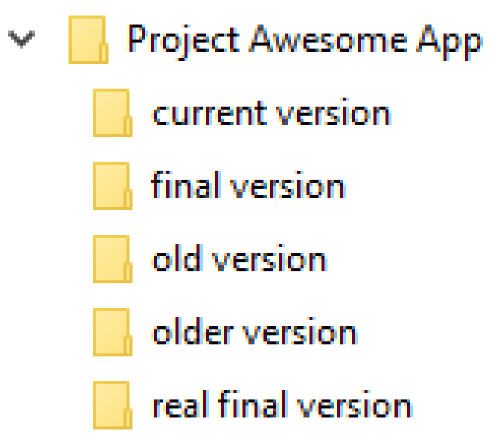


https://pixabay.com/vectors/user-top-view-office-keyboard-154199/

Some people‘s version control method until now

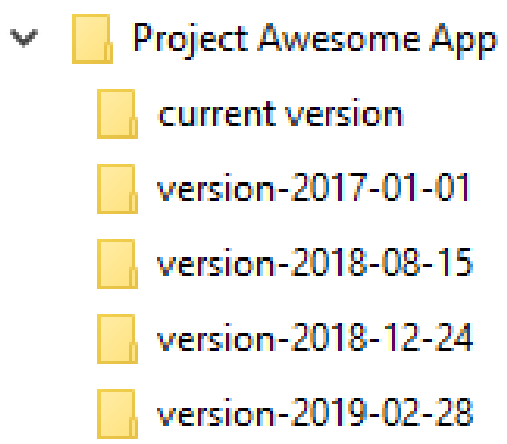
- „Textual version“

- Descriptive
- "Quick"
- Confusing
- …



Project Awesome App
current version
final version
old version
older version
real final version

- „Timestamp version“



Project Awesome App
current version
version-2017-01-01
version-2018-08-15
version-2018-12-24
version-2019-02-28

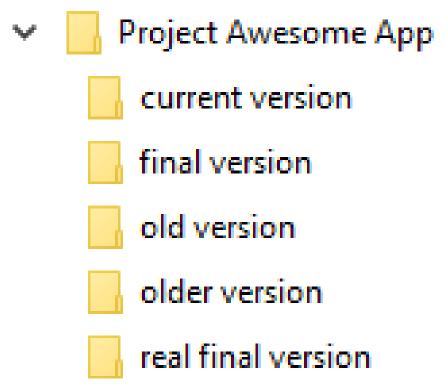
- Early approach
- "Improved"
- Confusing
- …

Both versions <u>are not</u> the best way for source code versioning (!)

Possible reasons „directory-versioning“ is not a good practice

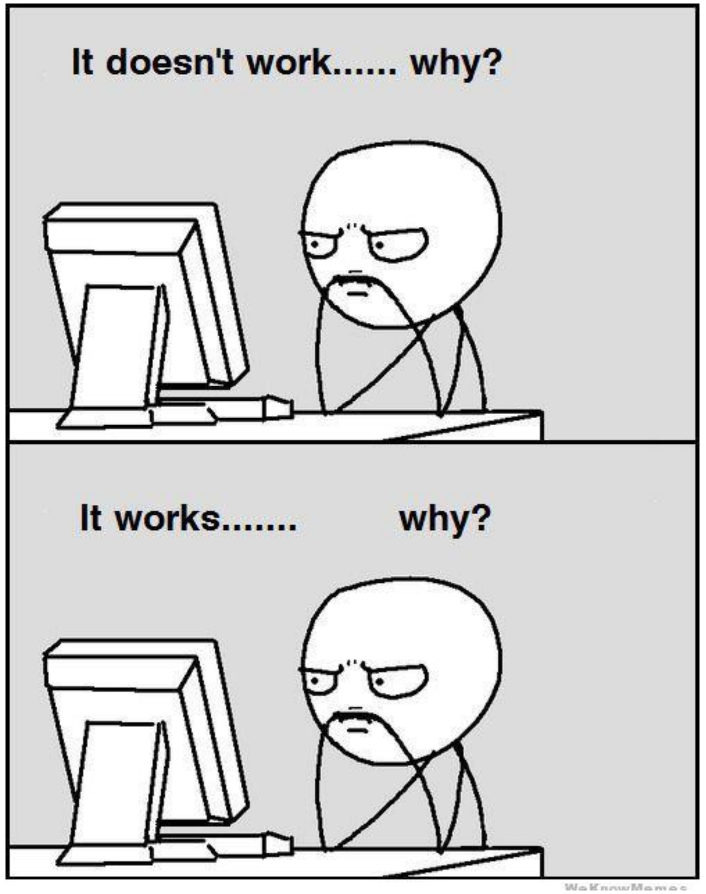
- Really **difficult to compare** old versions
- **Manual comparison** is needed and is defective
- Many other manual work needed
- No easy way for commenting your changes
- **Getting more and more complex** at longer „life-time“
  - Think about developing a software over months or years
  - 2-4 "versions" per week
  - Up to approximately 200 versions a year
  - … in a single folder with manual comparison

  "finally the newest version of last changes but not fully tested"


Project Awesome App
current version
final version
old version
older version
real final version

Already started?

- First lines of code has been already developed
- First challenges of changes
  - It works now
  - It doesn't work, but what I've changed
  - Oh... I'm not sure, there's a backup
  - **Oh... there is no backup***
  - ...
- After Configuration Management
  - **It doesn't work, I will take a look on what I've changed!**



* Software Versioning is no replacement of traditional backup strategies. It's possible to go back to older versions.

Overview about Local, Centralized and Distributed VCS

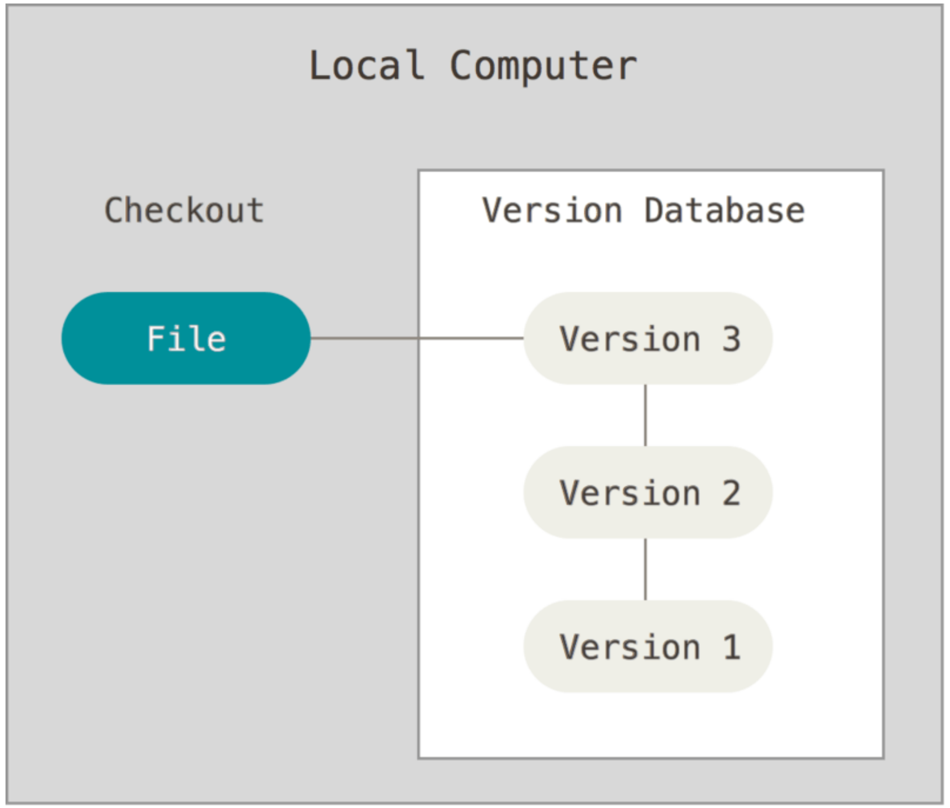
- There are basically three different Types of VCS
- **Local**
  - which is only on your computer
  - no access of any other person
- **Centralized**
  - which is only on one single server
  - any authorized person is able to read and write changes
- **Distributed**
  - where each server and computer has a full copy
  - any changes will be shared and stored at developers pc

Git is a distributed versioning control system!

Local Version Control System

You are developing on your local computer.

- **Version Database** is stored on your PC.
- Working Directory / **Checkout** is located on your computer too.
- Every change you are transmitting to your local Version Database.



Local Version Control System

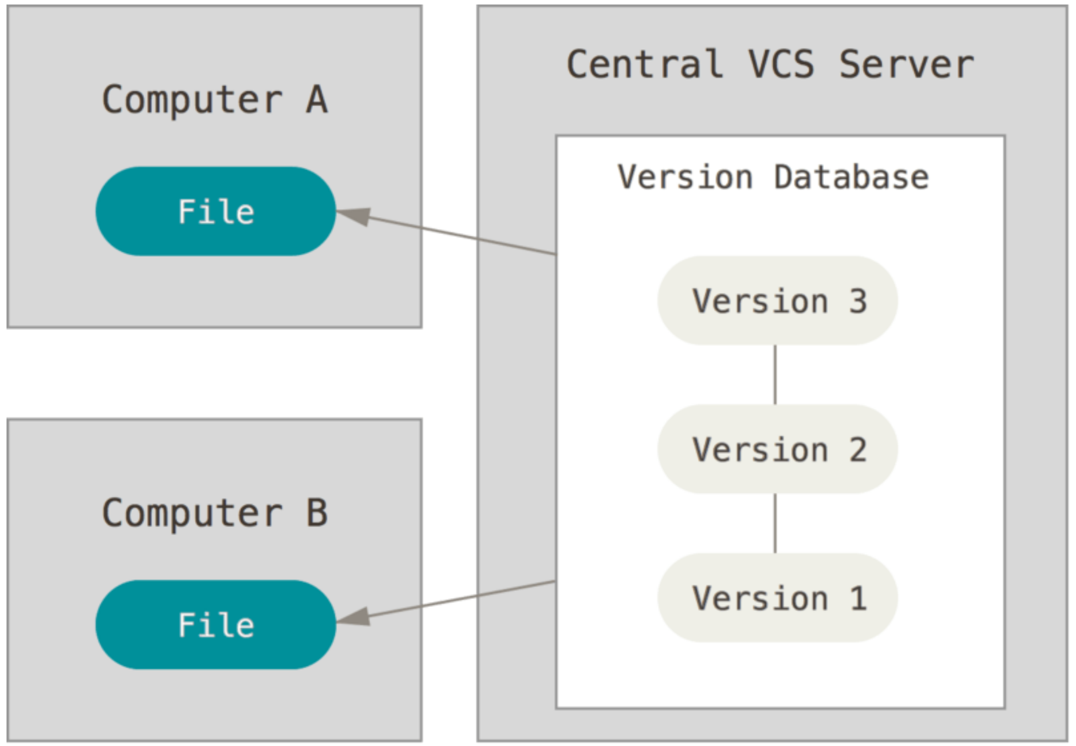
- **Advantages**
  - Simple
  - No access to internet needed
  - Another early approach to compare changes
- **Disadvantages**
  - No collaboration possible
  - Single point of failure
  - Not state-of-the-art

Keep in mind: also possible with Distributed Version Control System (!)

Centralized Version Control System

You are working again on your local computer, and another developer is working on his computer.

- Each developer has only the current version in his **working directory**.
- Changes will be transferred to a centralized server **repository**.
- Each developer has the possibility to keep on track of latest changes of other developers.



Centralized Version Control System

- **Advantages**
  - Central overview about current project state
  - everyone knows to a certain degree what everyone else on the project is doing
  - Collaboration is possible

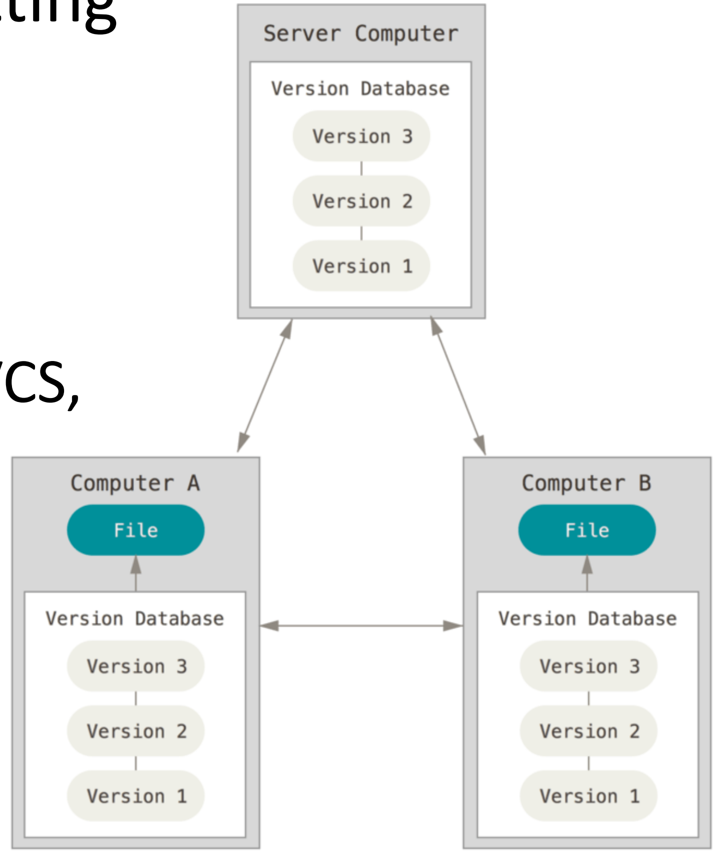
- **Disadvantages**
  - Single point of failure
  - Connection needed to use Version Control System
  - Current development has to be transmitted to create a new trackable version

Distributed Version Control System

Advantages of previous Version Control Systems are getting combined to a distributed VCS.

- Each developer is able and has to
  - **create new versions** through commiting changes to the VCS,
  - **synchronized changes** to a or multiple servers,
  - **collaborate with other team members** through a well known Version Control System,

  - and **keep on track to all changes** within his team.



Distributed Version Control System

- **Advantages**
  - Full „Backups“ with every clone
  - Defined workflows
  - Independent usage
  - Better team collaboration

- **Disadvantages**
  - Needs training and practice
  - Steep learning curve
  - Many commands
  - Complicated workflows (for beginners)
  - Indispensable for professional software development

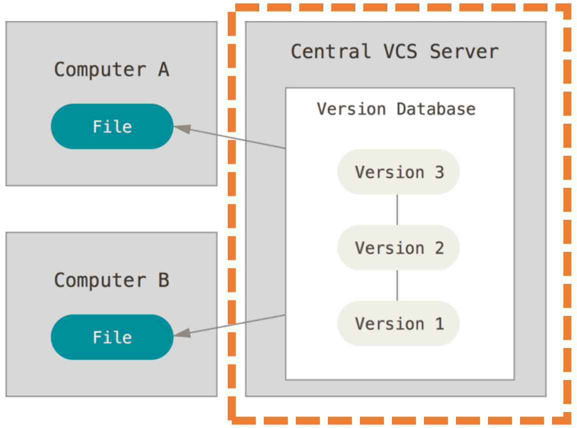
Quick Comparison

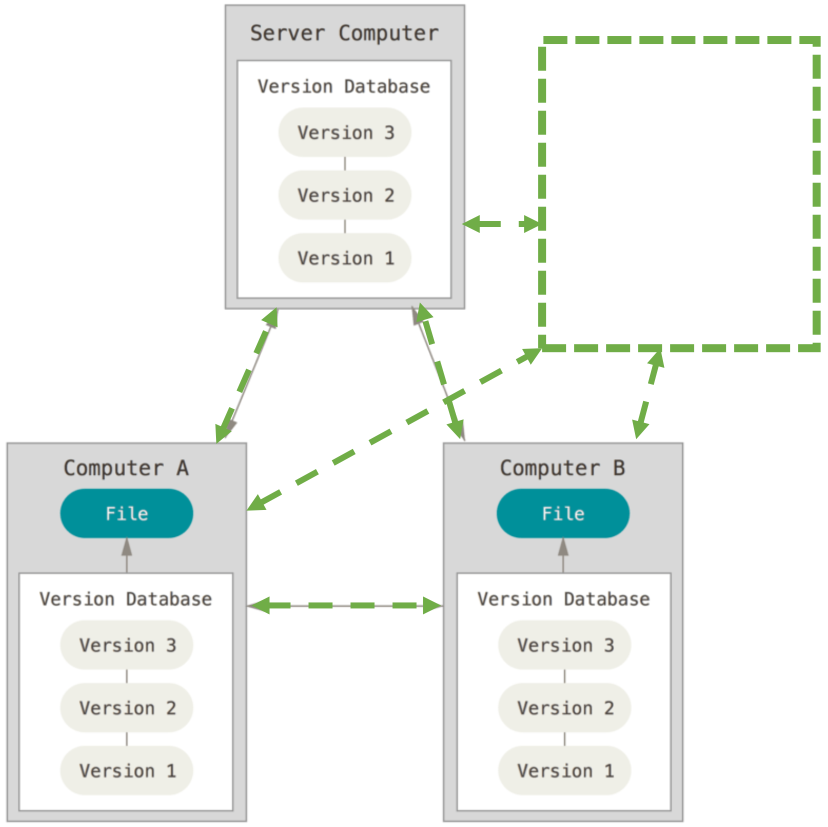
Advantages

- (Centralized) Server is easy to
  - recover after system crash, or
  - change to another system
- Current state-of-the-art VCS

Disadvantages

- Single point of failure
- Outdated





Build software better, together

- **Branching** and **Merging**
- Small and fast
- **Distributed**
- Data assurance
- **Staging area**
- Free and open source

First steps with Git

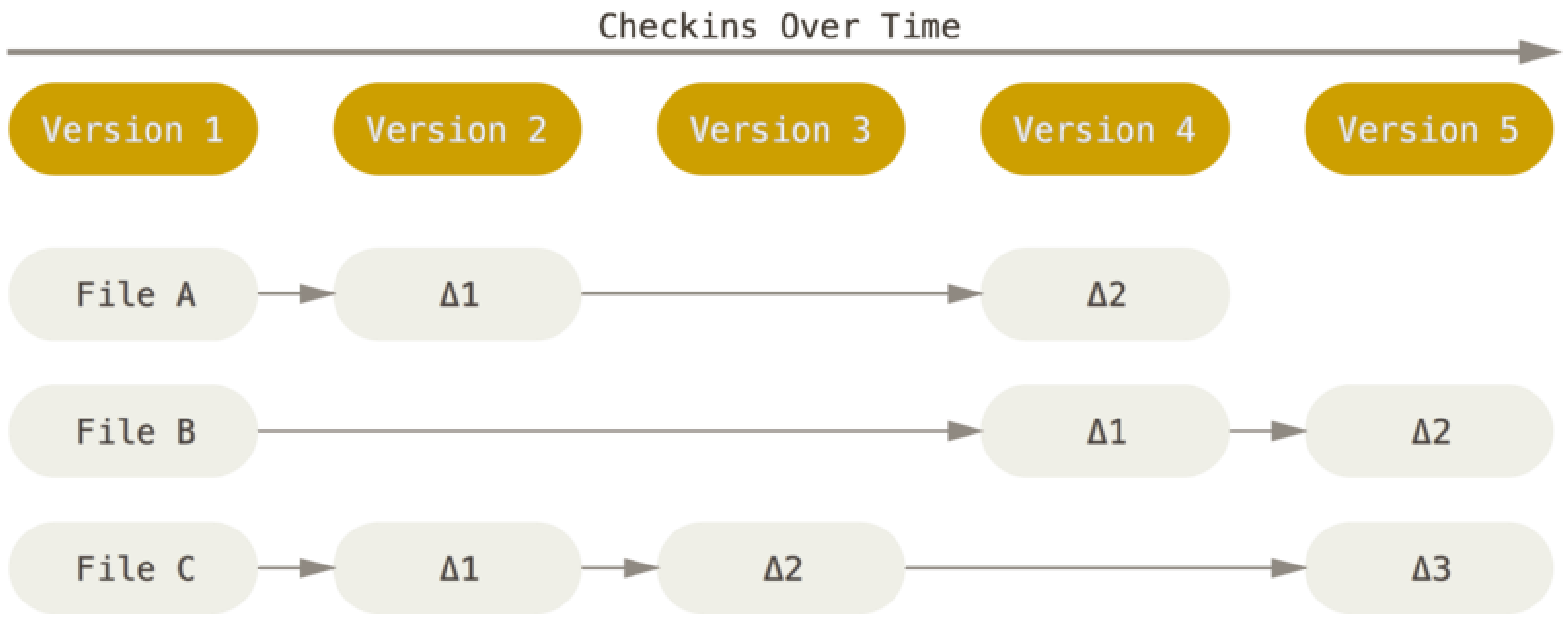
We have started with **local git operations and commands**!

- `git init`

- `git add <filename>`
- `git add .`
- `git commit -m "<your git commit message>"`
- `git status`
- `git log --graph`

Major Difference to other VCS…

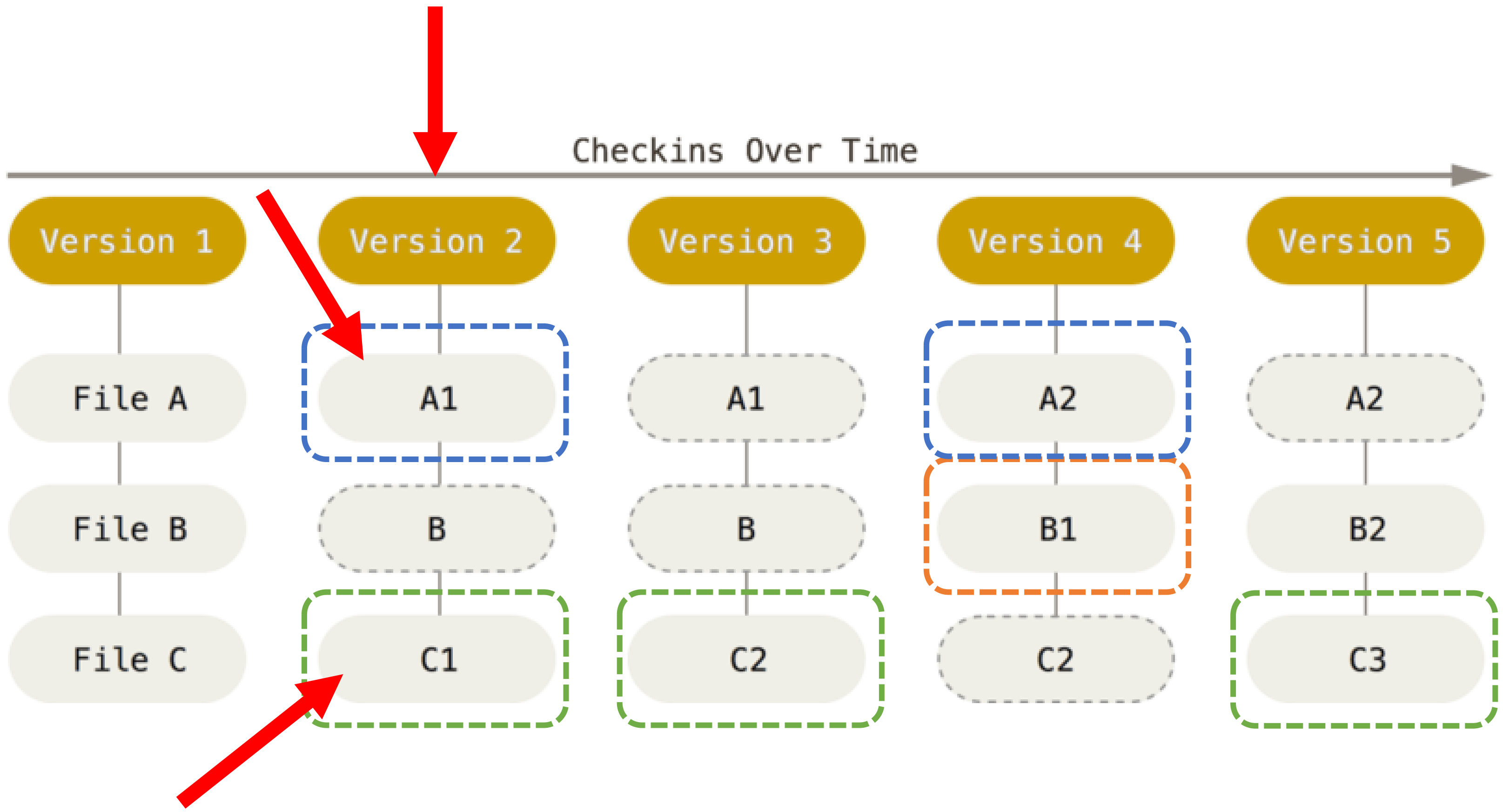
Other VCS are storing versions with Delta-Information. Only the changes are stored and to get the current version all versions has to be combined.



https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

… is the way of storing each version

Git is storing **each new version of a file**, creates a new version "number" of all files, and refers to existing and not changed files.



https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Concept of local operations

- **Most operations** with local files and resources
- No information from another computer / server is needed
- **Offline working** nearly every time is possible
- Also **complete history is available**, because of complete clone

```
git add
git commit
git status
git log
git diff
```

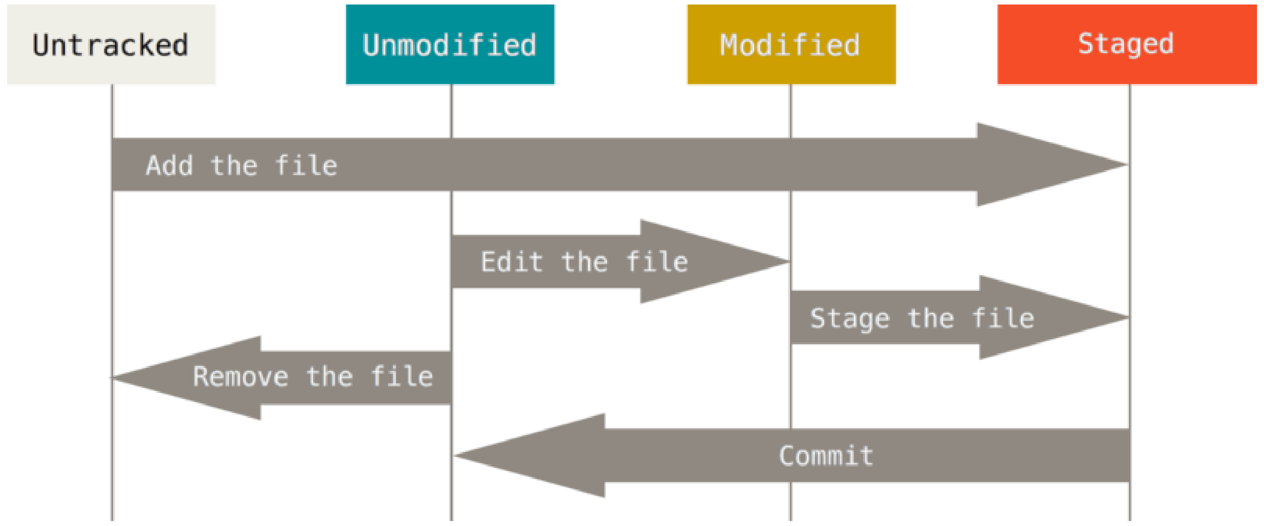
Data assurance – everytime!

- Check-summed before it is stored
  - Every commit has its own checksum
  - impossible to change any file, date, commit message, or any other data
  - Data manipulation is nearly impossible
- SHA-1 hash
  - Example: 24b9da6552252987aa493b52f8696cd6d3b00373
- Most centralized VCS **not** provide such integrity

It's impossible to get anything out of Git other than the exact bits you put in!

Recording Changes to the Repository

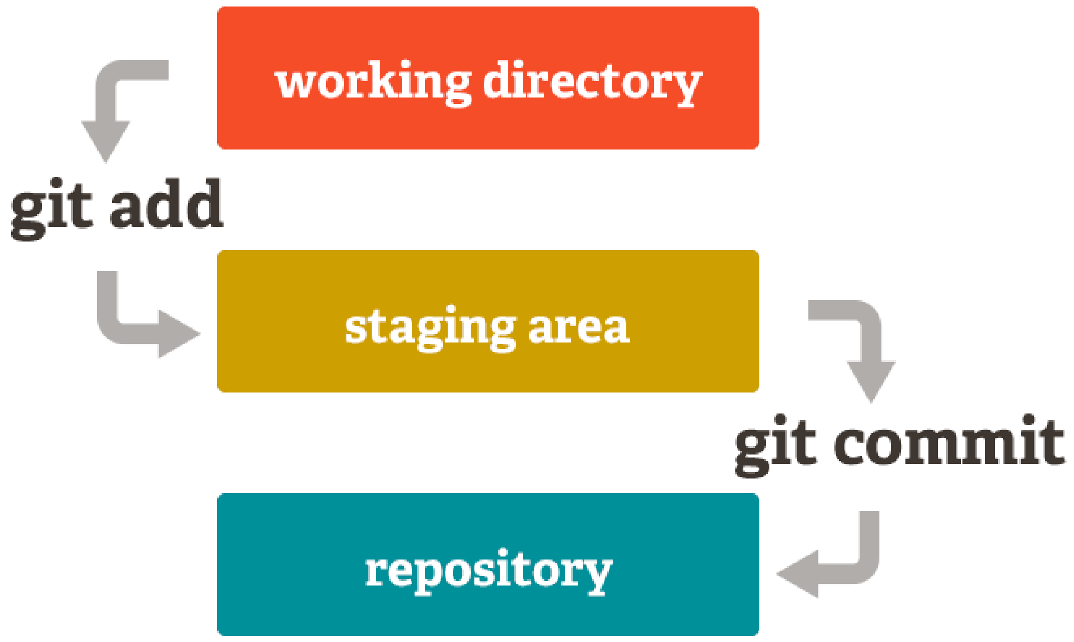
- New Files are **untracked**
- **Tracked** files are already versioned
- Changes to files leads to "**modified**"
- Modified files become "**staged**"
- A group of staged files become unmodified state through a "**Commit**"



https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Three main States of files

- **Modified**
  - Editing files
- **Staged**
  - Finished editing, transfered to **staging area** and ready to create a new version
- **Committed**
  - The **repository** has a new version



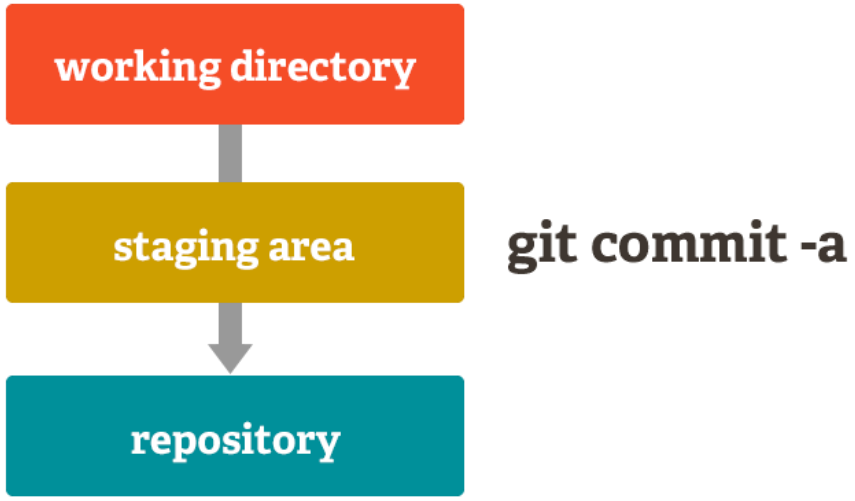
Important note: it's also possible to stage multiple files and commit them all by one single commit!

https://git-scm.com/about/staging-area

Quick way to commit your changes

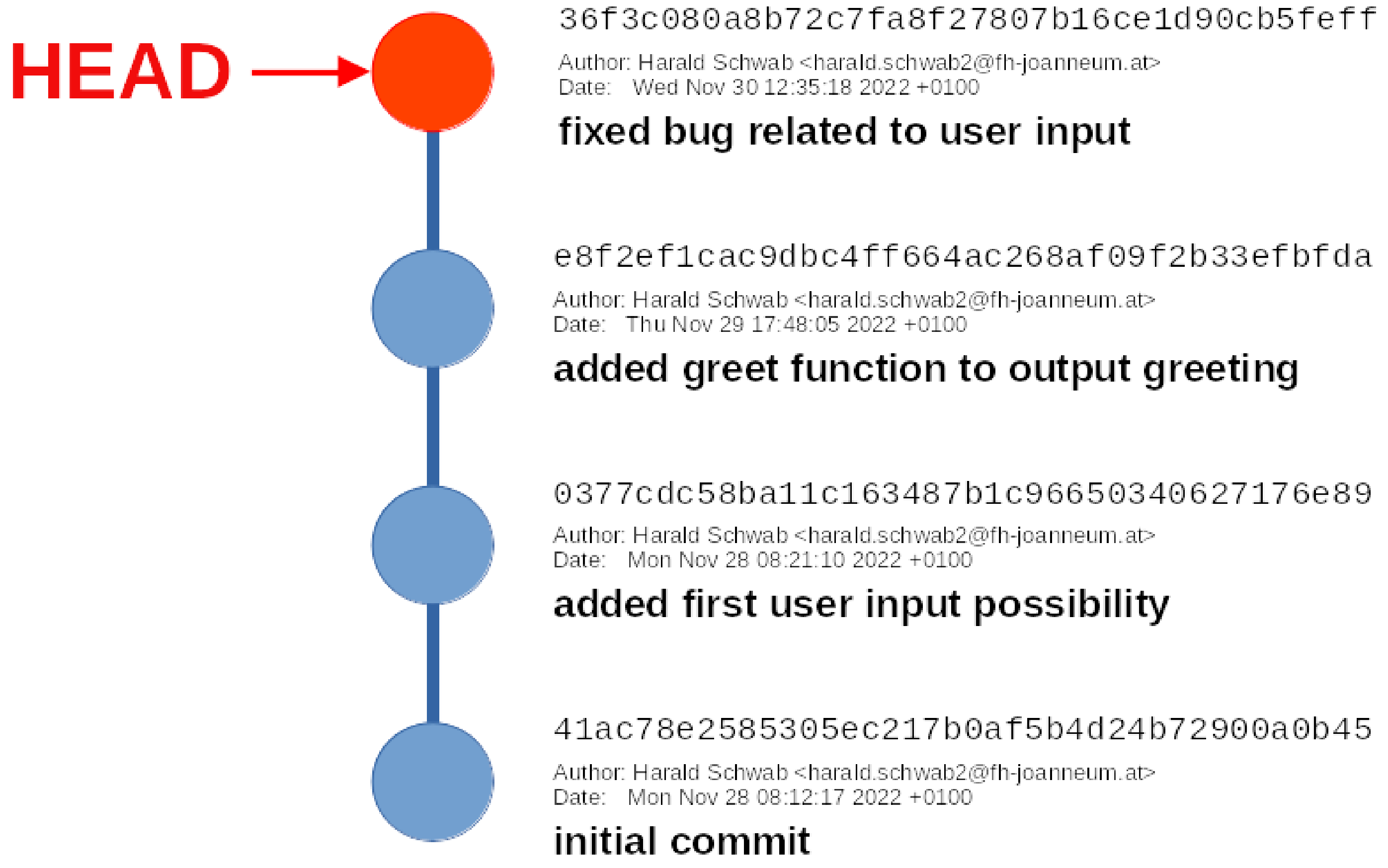
- Possible with one command
- **<u>Not recommended</u>** for the beginning
- **Only commit working version!**

- **<u>Why this could be lead to problems?</u>**



Commits

- Each commit has its own – **unique** – “id” (=hash)
- **HEAD** refers to the current commit
- Components of a commit:
  - WHO: Author (name + email)
  - WHEN: Date + Time
  - WHAT: commit message

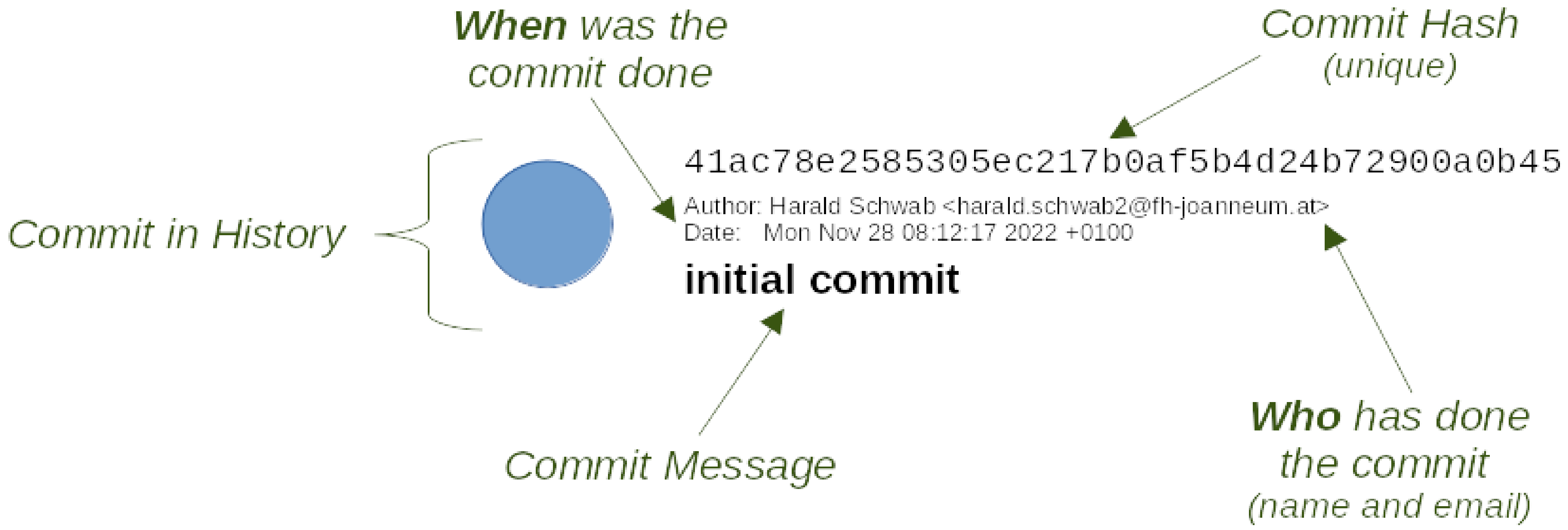


```
36f3c080a8b72c7fa8f27807b16ce1d90cb5feff
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Wed Nov 30 12:35:18 2022 +0100
fixed bug related to user input

e8f2ef1cac9dbc4ff664ac268af09f2b33efbfda
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Thu Nov 29 17:48:05 2022 +0100
added greet function to output greeting

0377cdc58ba11c163487b1c96650340627176e89
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:21:10 2022 +0100
added first user input possibility

41ac78e2585305ec217b0af5b4d24b72900a0b45
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:12:17 2022 +0100
initial commit
```

DIHSÜD

FH JOANNEUM University of Applied Sciences

Components of a commit


When was the commit done
Commit Hash (unique)
Commit in History
Commit Message
Who has done the commit (name and email)

```
41ac78e2585305ec217b0af5b4d24b72900a0b45
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:12:17 2022 +0100
```

initial commit

Basic Git Workflow for every developer

1. **Modify/Create** files

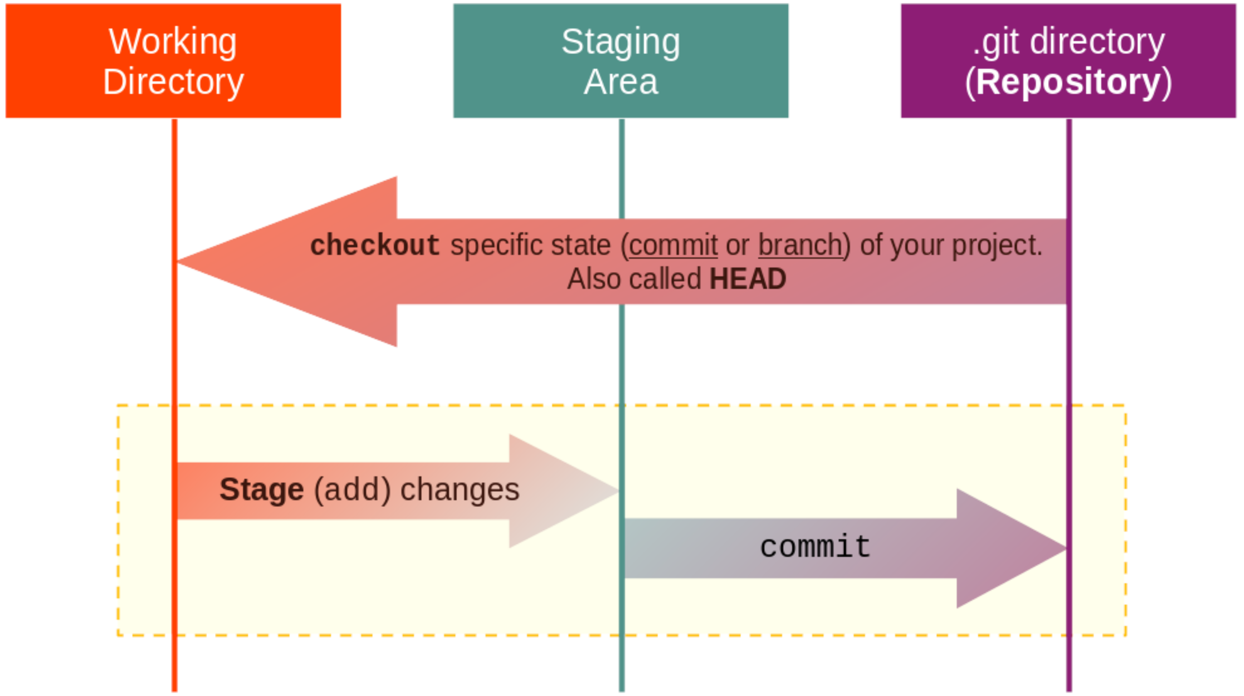
   in **working directory***

2. **Stage** (`git add`) files

   to the **staging area**

3. **Commit** changes (`git commit`)

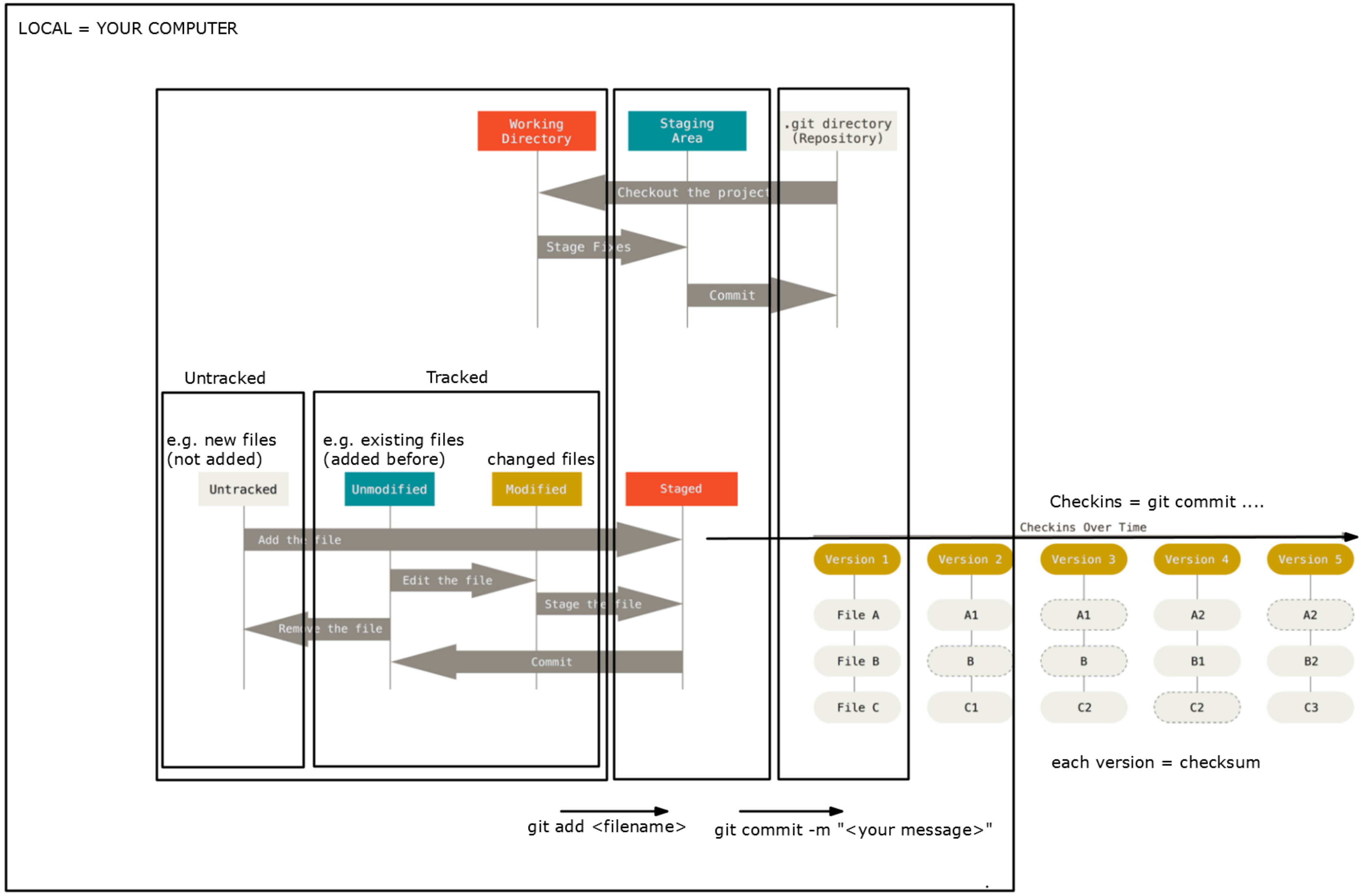
   and store a snapshot permanently in **repository**


Working Directory
Staging Area
.git directory (Repository)
checkout specific state (commit or branch) of your project. Also called HEAD
Stage (add) changes
commit

* When talking about working directory the local git repository is meant. The directory where the .git directory is located, typically your project root directory.

https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Working with Git Local – „Big Picture“



Git commands – Configuration and Initialisation

- `git config`: configure git*
  - `git config --global user.name`: check/set the name that should be used for your commits. Has to be set once!
  - `git config --global user.email`: check/set the email that should be used for your commits. Has to be set once!
    - `git config --local`: use `--local` instead of `--global` to set specific configurations individual for the current local repository
  - `git config --global init.defaultbranch`: What should be the name of the default branch? (today typically main)

- `git init`: initialise a new git repository in the current directory. Only needed once per repo/project at the beginning. Will create the .git-directory.

*Global configurations could be found in the file ~/.gitconfig

DIHSÜD

FH | JOANNEUM University of Applied Sciences

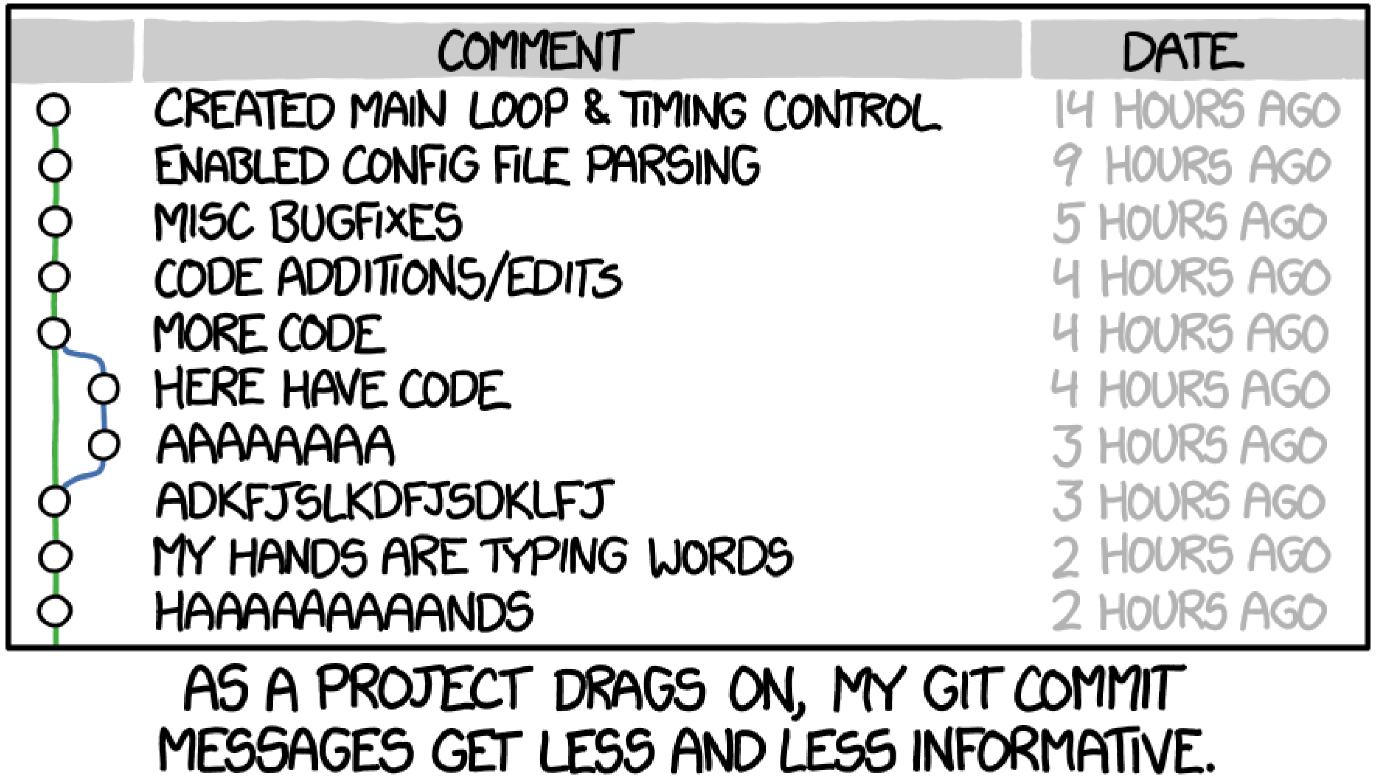
Git commands – Staging

- `git status`: show current state of working directory and staging area
  - `git status -s`: will print a shorter output
- `git add`: add new/changed files to the staging area
  - `git add .` or `git add *` will add all files at once. Use `git add <path/to/file>` to add a specific file
- `git diff`: Show the difference between the working directory and the staging area
- `git reset`: remove files from staging
  - `git reset HEAD` will remove all already staged files from the staging area. *It will not affect any changes since last commit in the files itself.*
  - `git reset <filename>` will remove the file from staging. *It will not affect any changes since last commit in the file itself.*
  - `git reset <commit-hash>` will revert all changes until this commit.

Git commands – Commits and Git-History

- `git commit`: commit all staged changes (will open the set editor)
  - `git commit -m "<commit message>"` allow you to directly provide the message
  - `git commit -am "<message>"`: `-a` will add all changed (not new!) files automatically to this commit
- `git log`: show the history of your (local) repository
- `git checkout`: switch to a specific commit/reset all uncommitted changes of a file
  - `git checkout <filename>` will **REVERT** all changes in the file since last commit! *Use it with care, this operation could not be undone!*
  - `git checkout <commit-hash>` will switch to the specific commit. You can switch back to the "latest" commit with `git checkout main` (*as long your branch is called main!*)

Commit messages,… which you should not write!

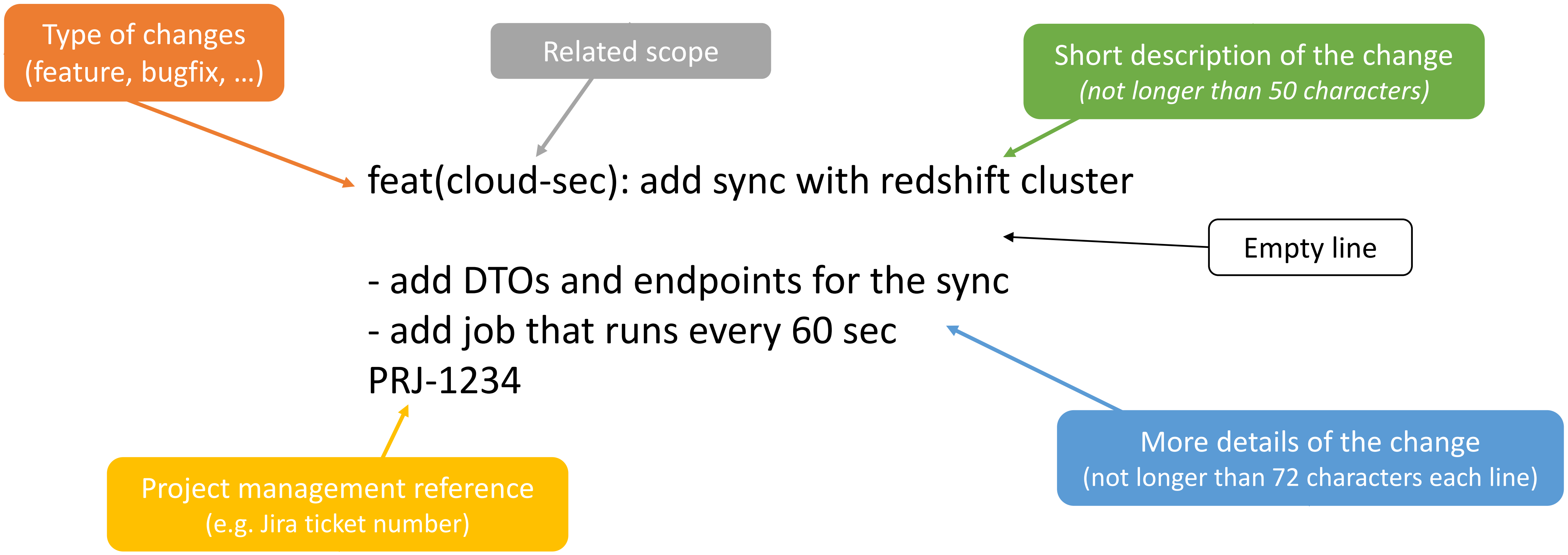


	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

https://chris.beams.io/posts/git-commit/

Meaningful commit messages



```
feat(cloud-sec): add sync with redshift cluster

- add DTOs and endpoints for the sync
- add job that runs every 60 sec
PRJ-1234
```

Type of changes (feature, bugfix, …)
Related scope
Short description of the change *(not longer than 50 characters)*
Empty line
More details of the change (not longer than 72 characters each line)
Project management reference (e.g. Jira ticket number)

https://www.freecodecamp.org/news/how-to-write-better-git-commit-messages/

Not every file has to be tracked

- Files can be tracked to be ignored by list of files
  - create **.gitignore** file
  - .gitignore file will be include a list of files and directories, which will be **NOT** tracked anymore
- Files to ignore, e.g.:

– Executables	*.exe
– Generated files	*.class
– Images	*.iso, *.dmg
– Log files	*.log

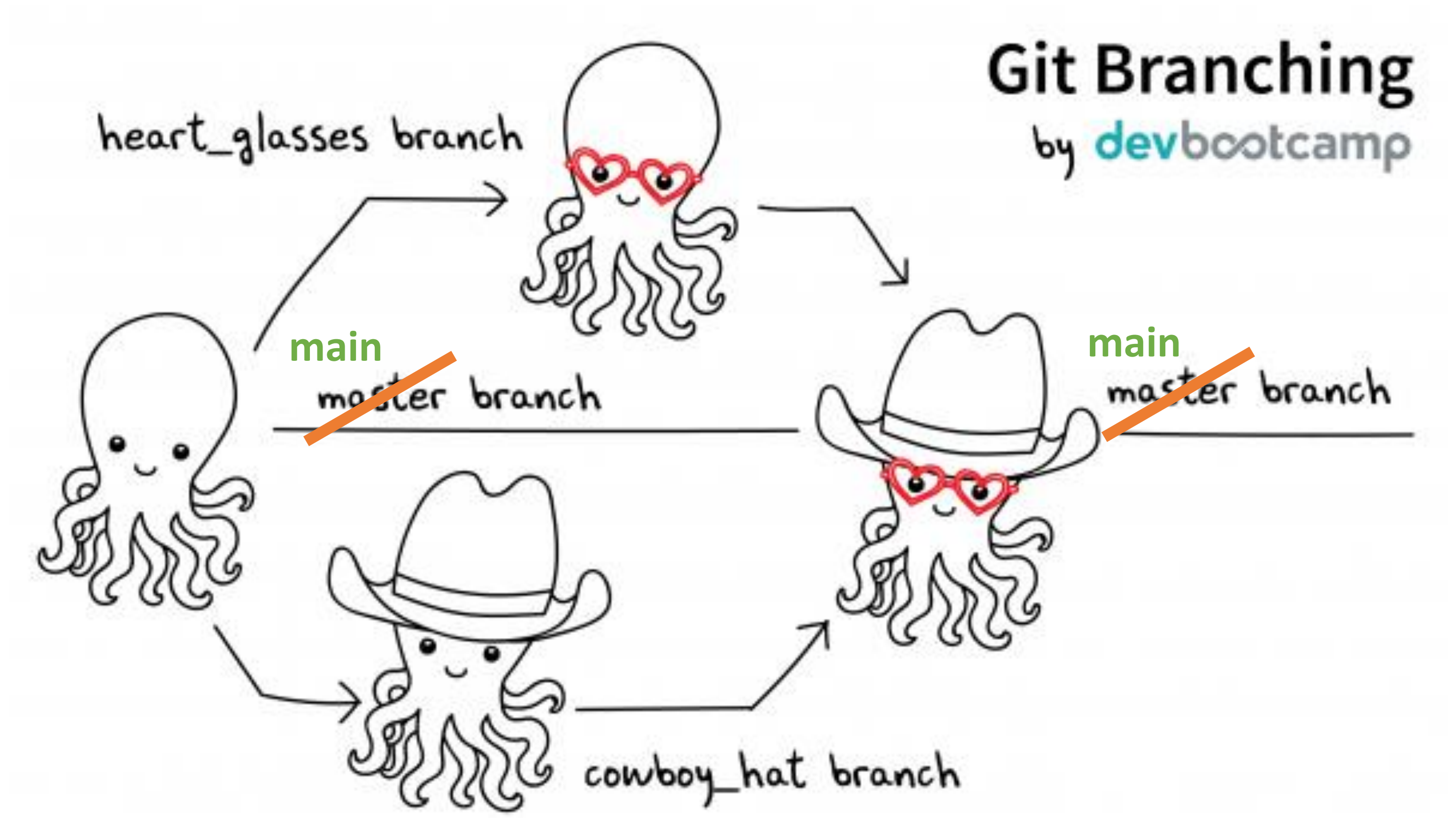
  - 3rd party libs (use a package manager)
  - Secrets (passwords, logins, api keys, …)
  - Backup files *(automatically)* created by your editor/ide
  - Other examples https://gist.github.com/octocat/9257657

.gitignore

NEVER share secrets in a git repository!

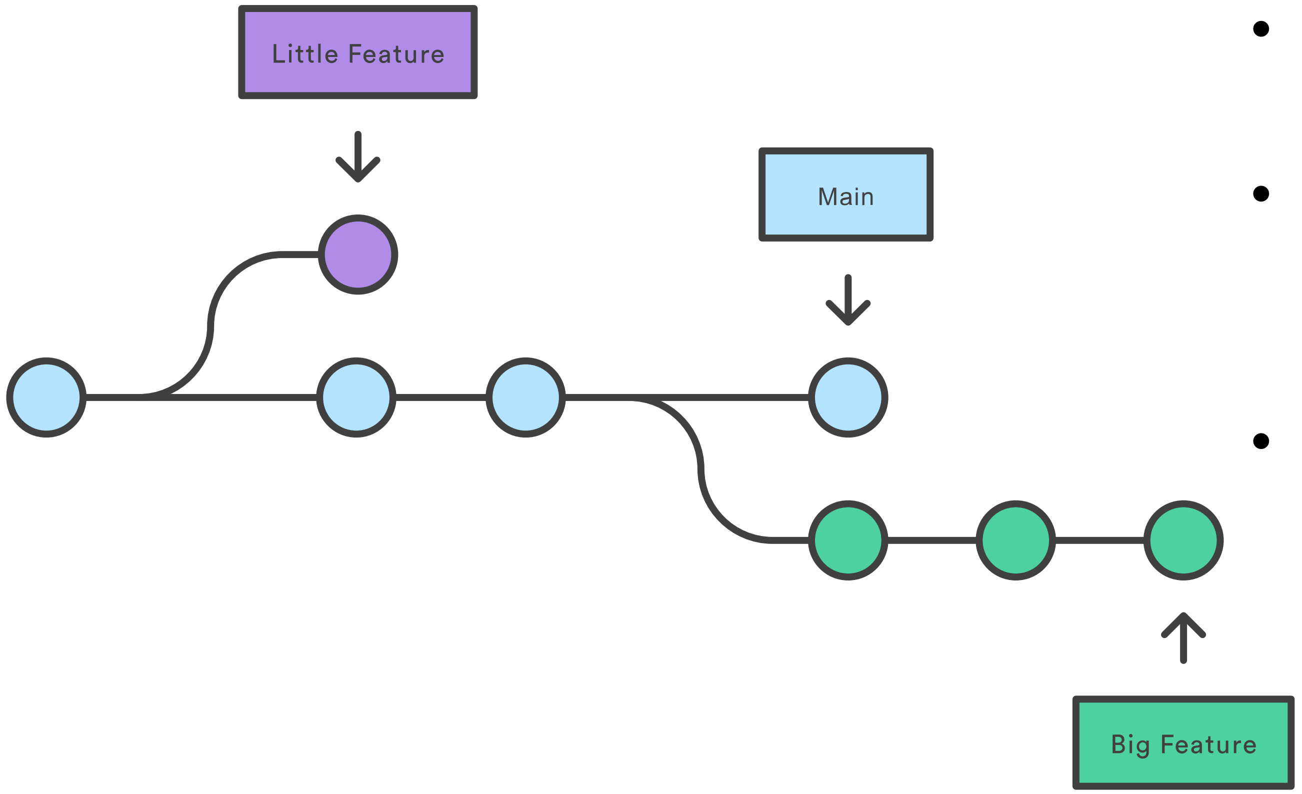
https://git-scm.com/docs/gitignore

Branches in a nutshell



https://gerardnico.com/code/version/git/branch

Branches in git



Branching can be found in many modern version control systems, however often time and space consuming.

In git:

- Branches are **lightweight** and **heavily used** in daily developement activities
- A branch is a **reference to a commit** (nothing is copied)
  - Branch is tip of a series of commits
- Basically in Git a **"main" branch** exists (formally known as "master"), where every commit leads to a new Version like C0, C1, C2, …

A new branch …

… illlustrates independent line of development.

We create it when

- **Testing** first ideas for improvement
- **Work** on feature(s) or any other issues
- **Fixing** bugs

Think of it like an independent brand-new:

- working directory
- staging area
- project history

Branching commands

On the following slides we will get to know the most important commands for working with branches, so we are able to:

Choose in which branch we want to work at one moment -> **checkout**

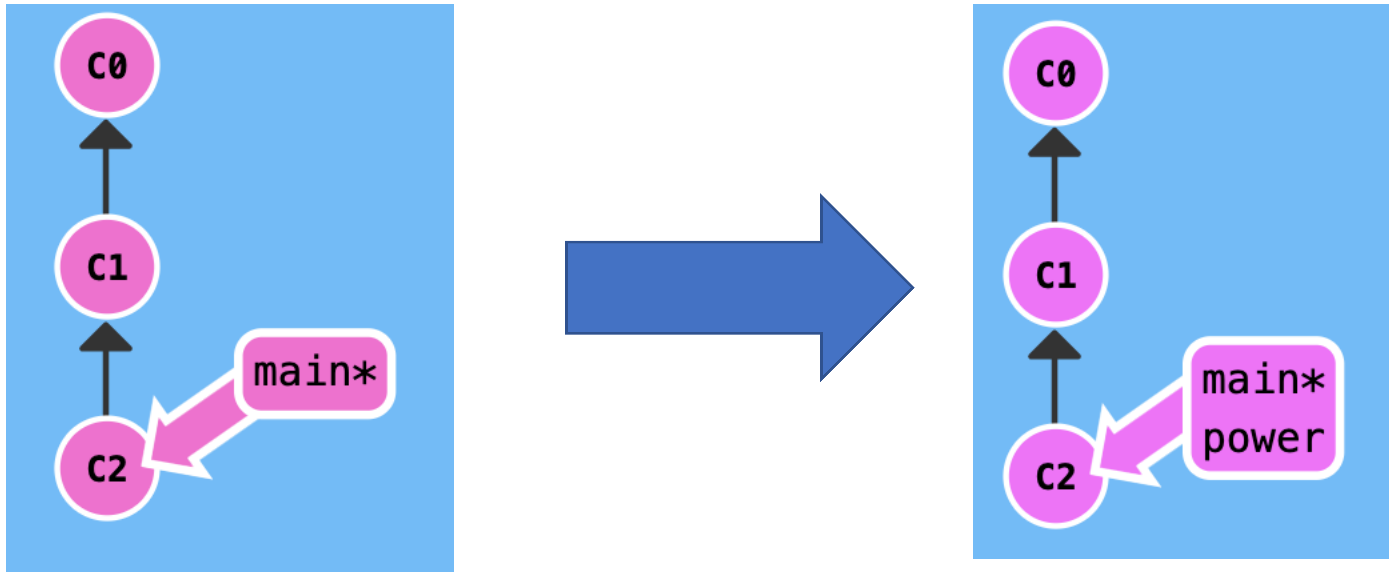
Create new branches -> **branch**

Merge multiple branches together -> **merge**

git branch <name>

Allows us to create a **new branch** based on the commit we are currently working on:

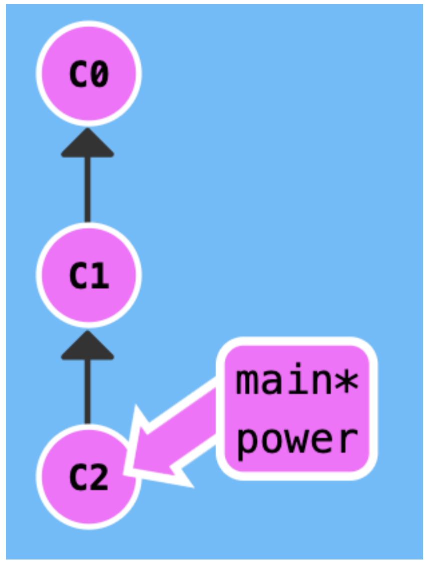
- The repository history has not changed
- We just get a new pointer



git branch (or git branch --list)

Get a list of all branches.

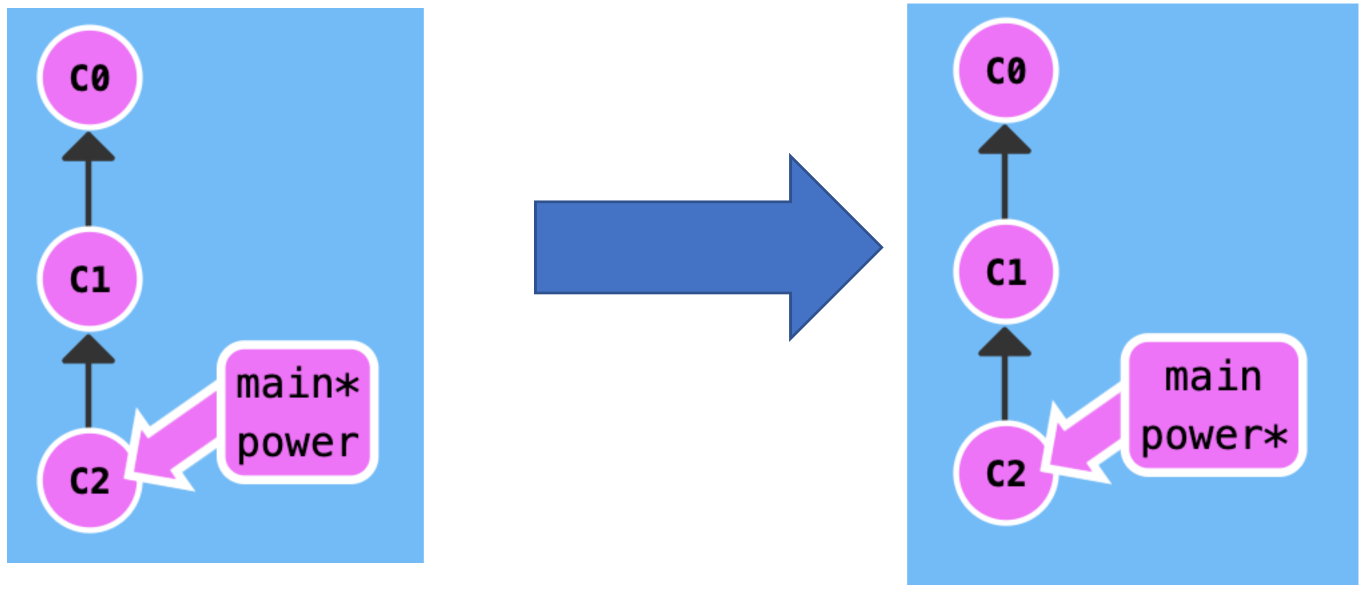
- We are still working on the main branch (* signals active branch)



git checkout <name>

Switch to the specified branch

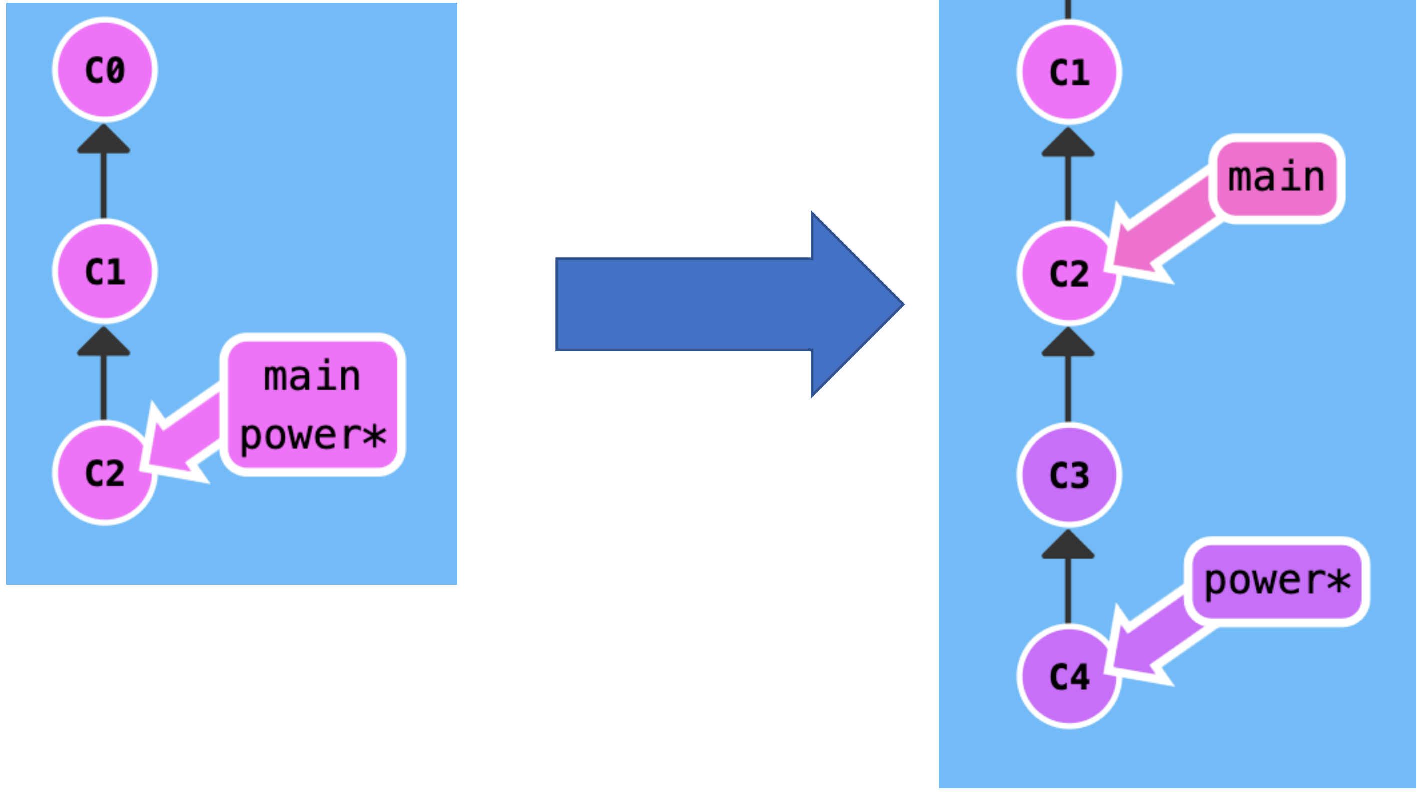
- Now we can start working



git checkout –b <name> creates a new branch and swichtes to it

Coding – Adding - Commiting

Our usual git workflow – two commits later

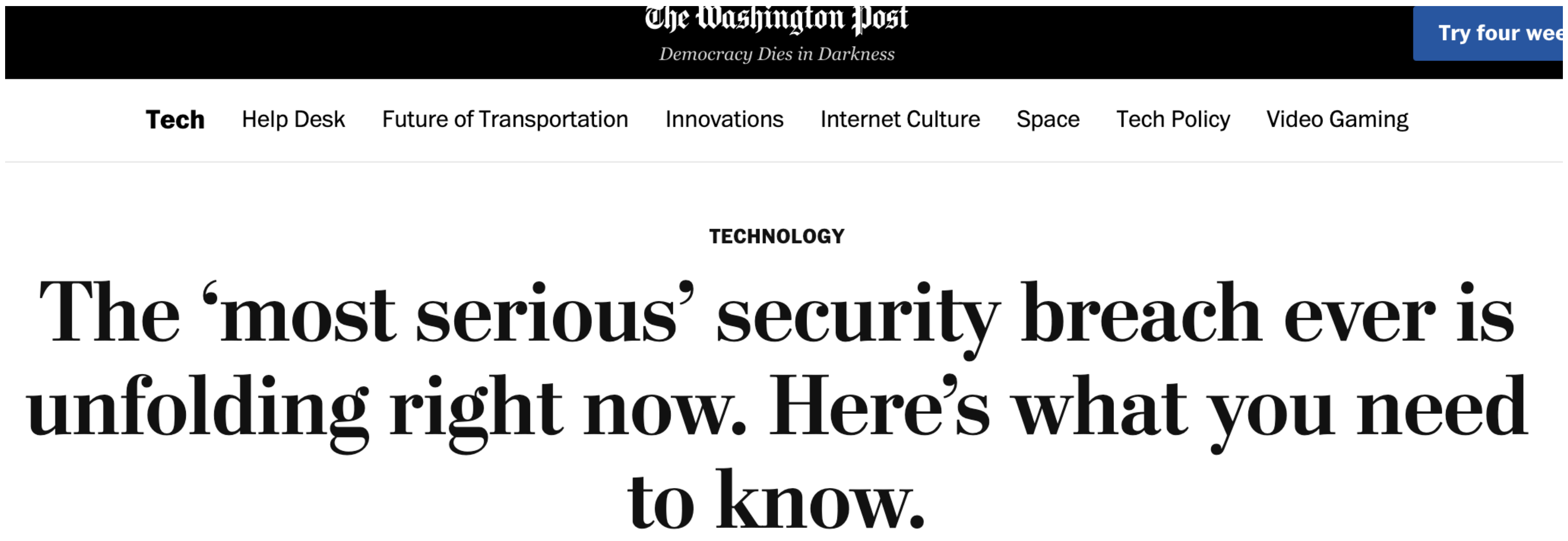


National Cyber Security Centre

Log4j is used worldwide across software applications and online services, and the vulnerability requires very little expertise to exploit. This makes Log4shell potentially the most severe computer vulnerability in years.

Log4j software bug is 'severe risk' to the entire internet

A flaw in a commonly used piece of software has left millions of web servers vulnerable to exploitation by hackers



TECHNOLOGY

The 'most serious' security breach ever is unfolding right now. Here's what you need to know.

We need a fix ASAP

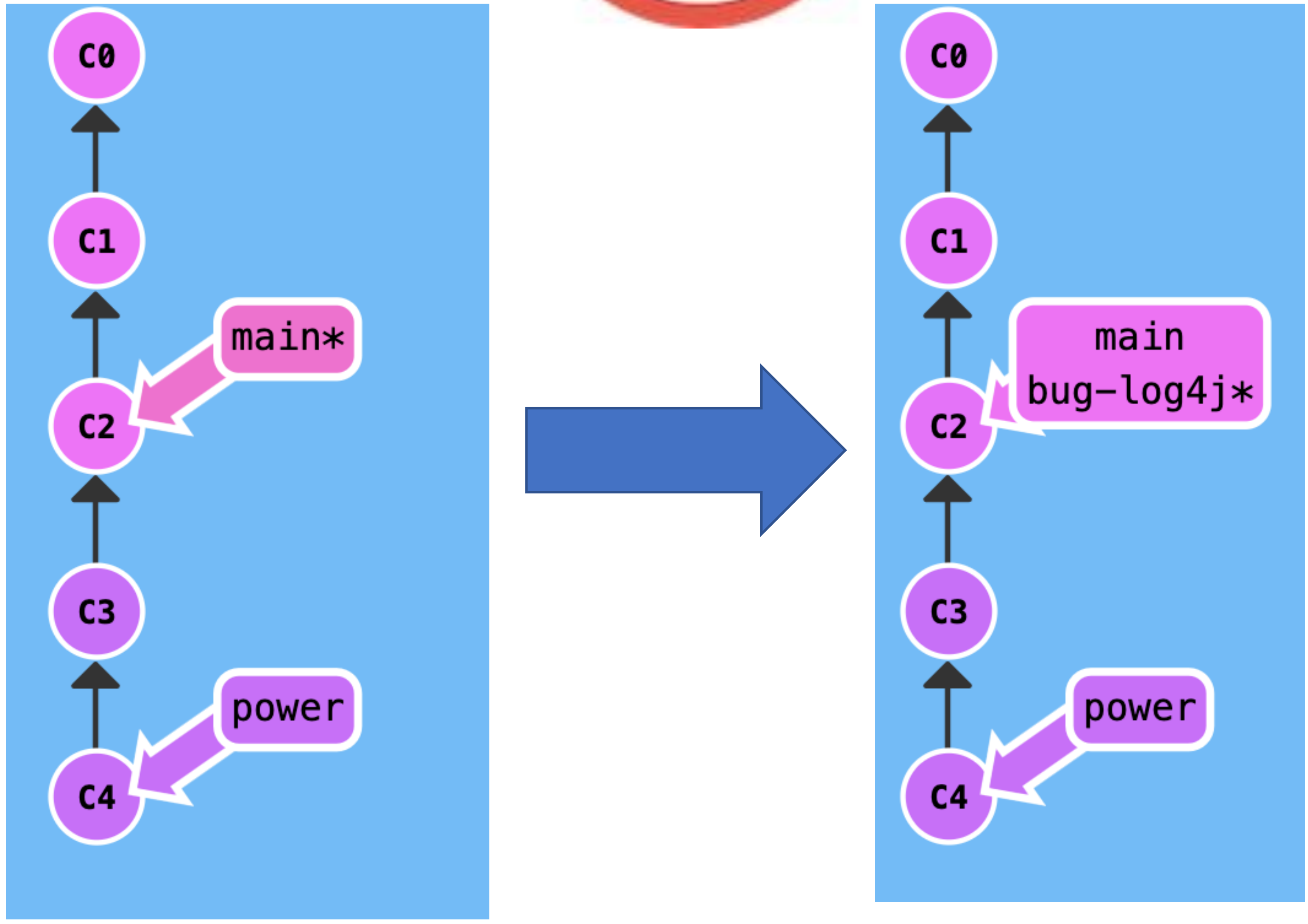
We can switch back to our main development branch

git checkout main

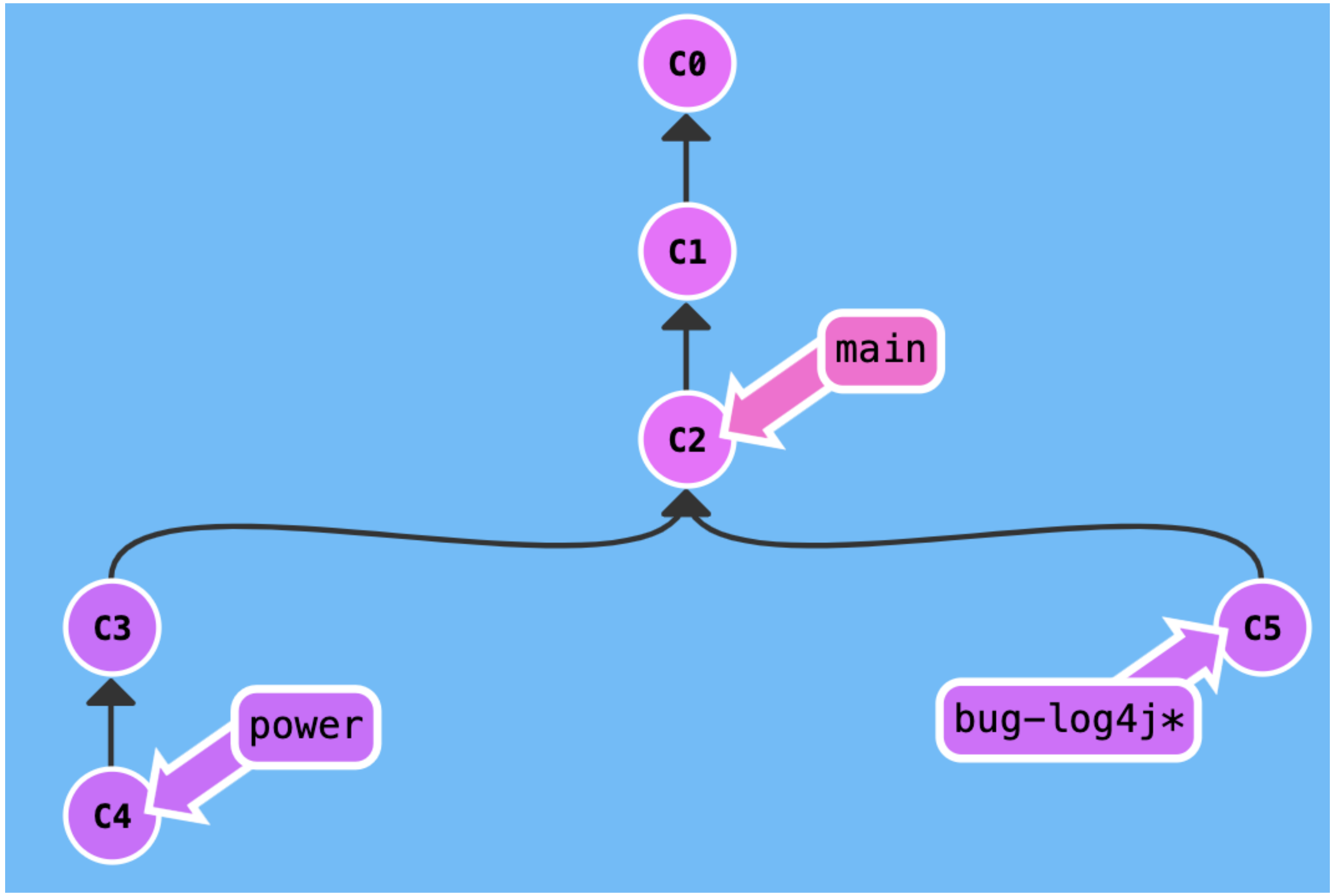
and start there a new branch for the fix:

git branch bug-log4j

git checkout bug-log4j



We fix the bug and commit it



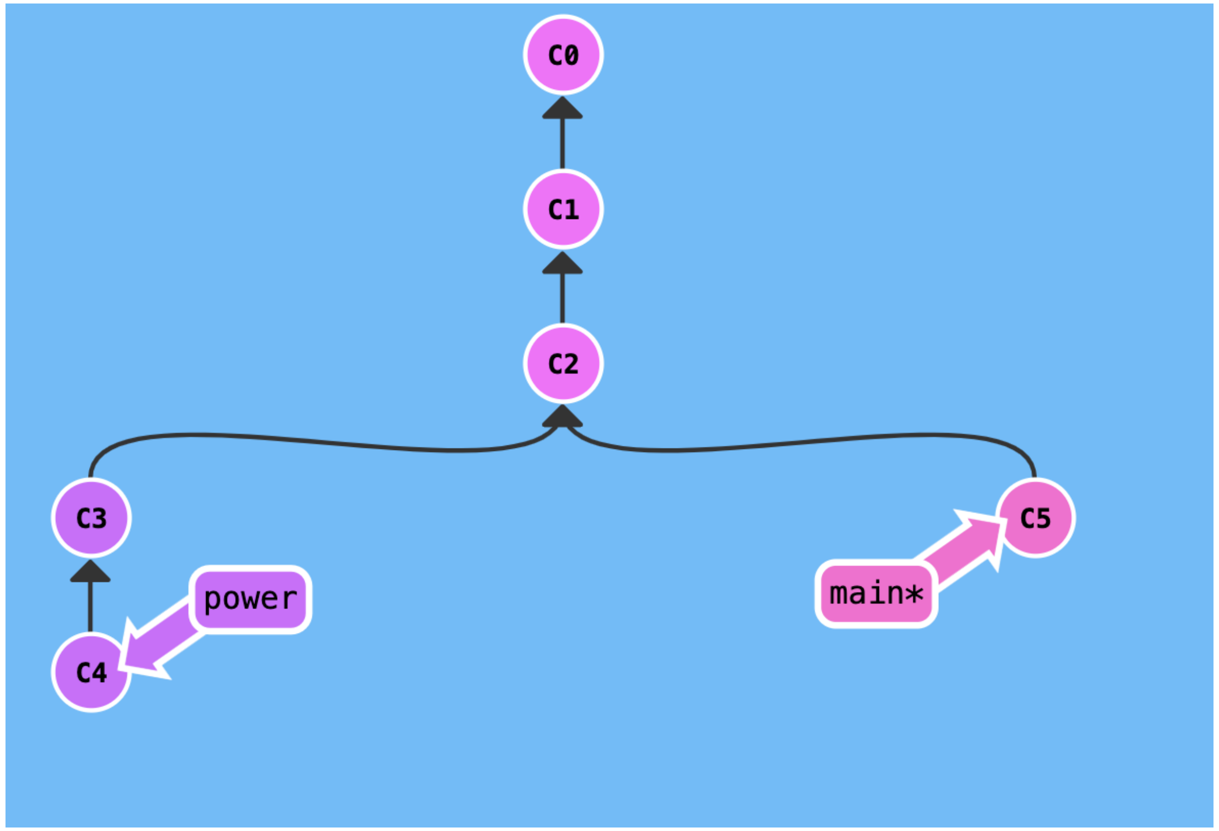
DIHSÜD

FH JOANNEUM University of Applied Sciences

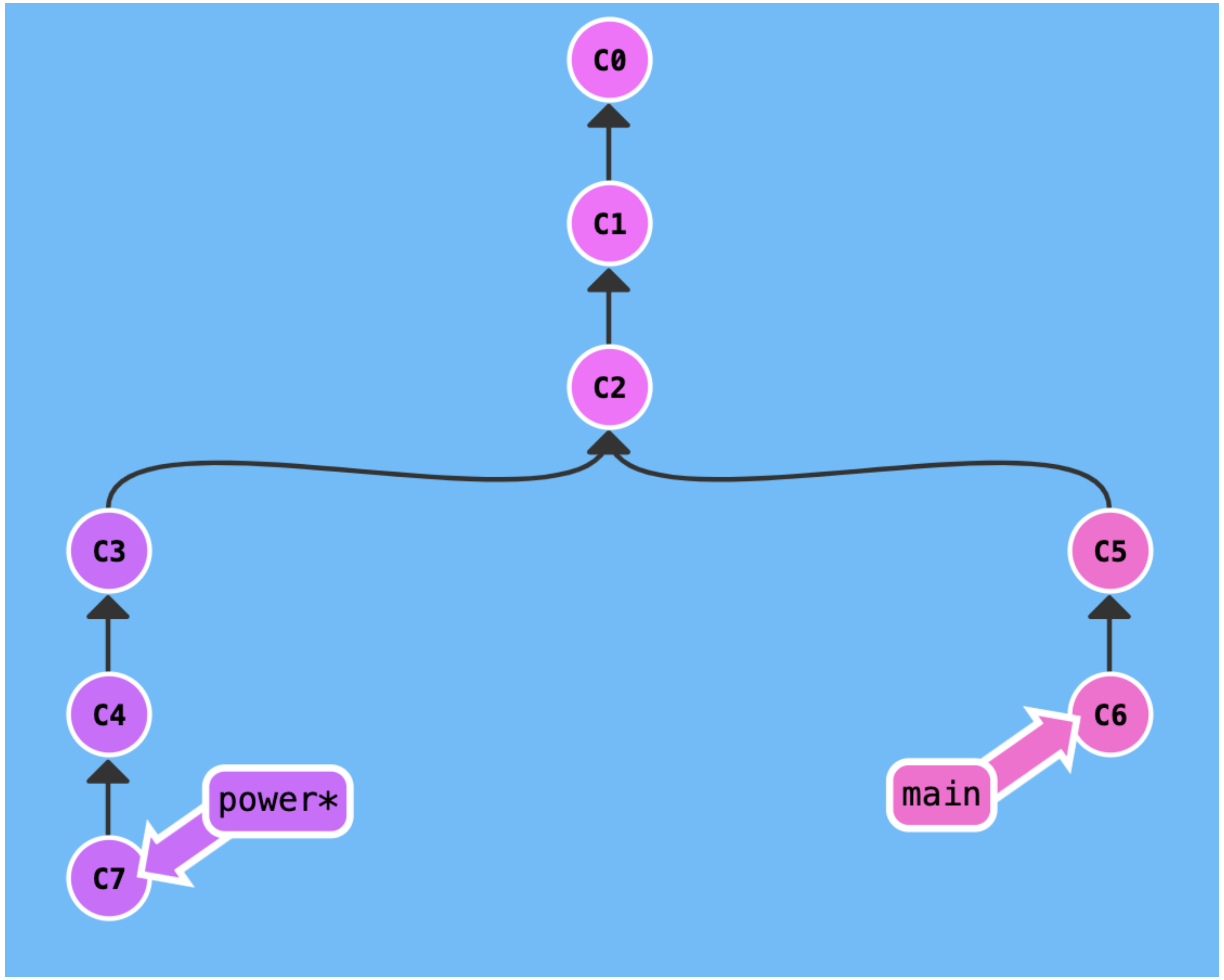
git branch -d <name>

Allows us to delete a branch (like bug-log4j) we no longer need

- Only allows safes operations so we can loose nothing
- We would not be able to delete the power branch, because we would loose all the changes in it
    - **Danger zone:** -D is force delete and would also throw away all commits associtated with branch



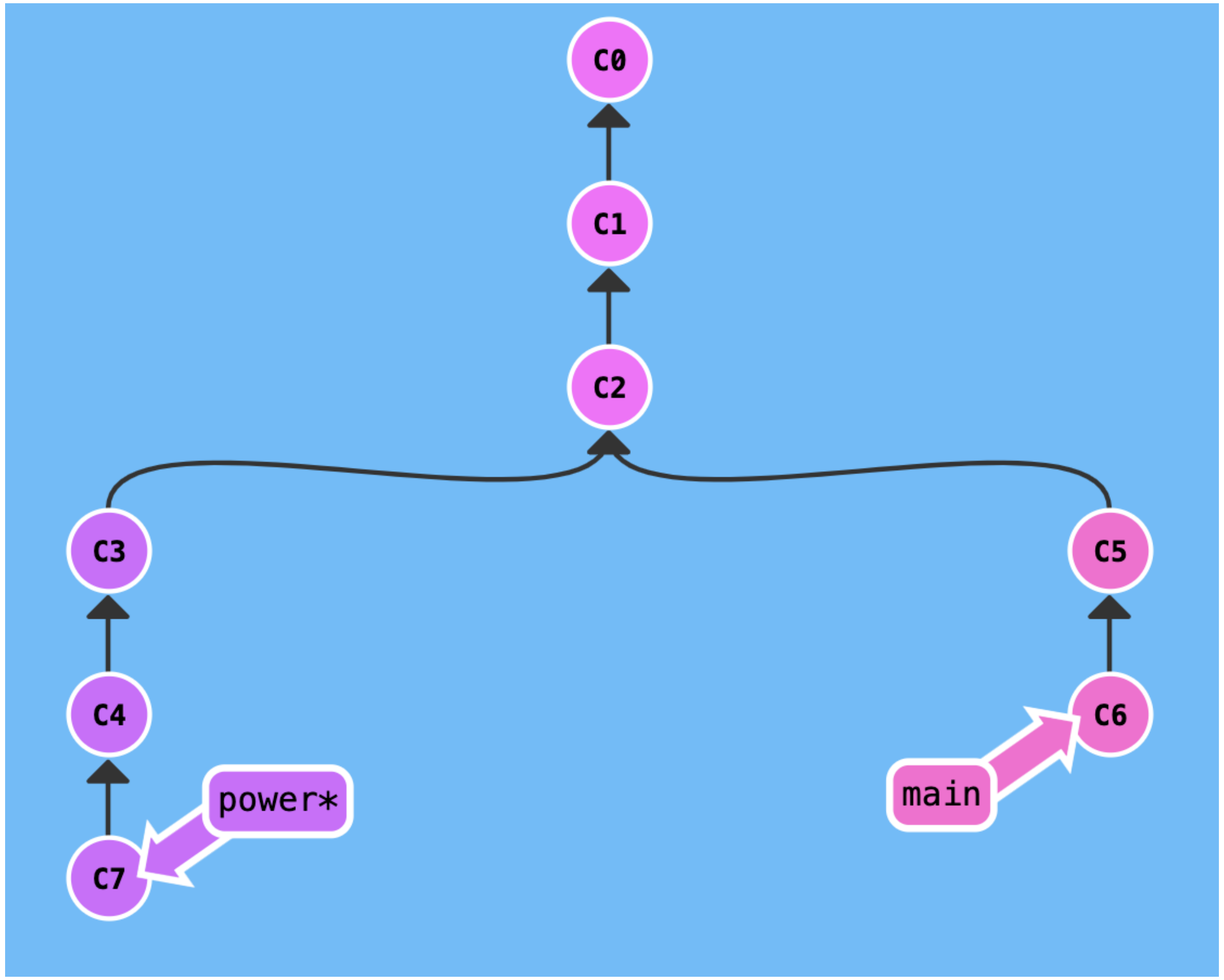
Back to power branch and work continues



When finished

We can now merge back our new functionality to main

- Before we have to checkout main, then merge

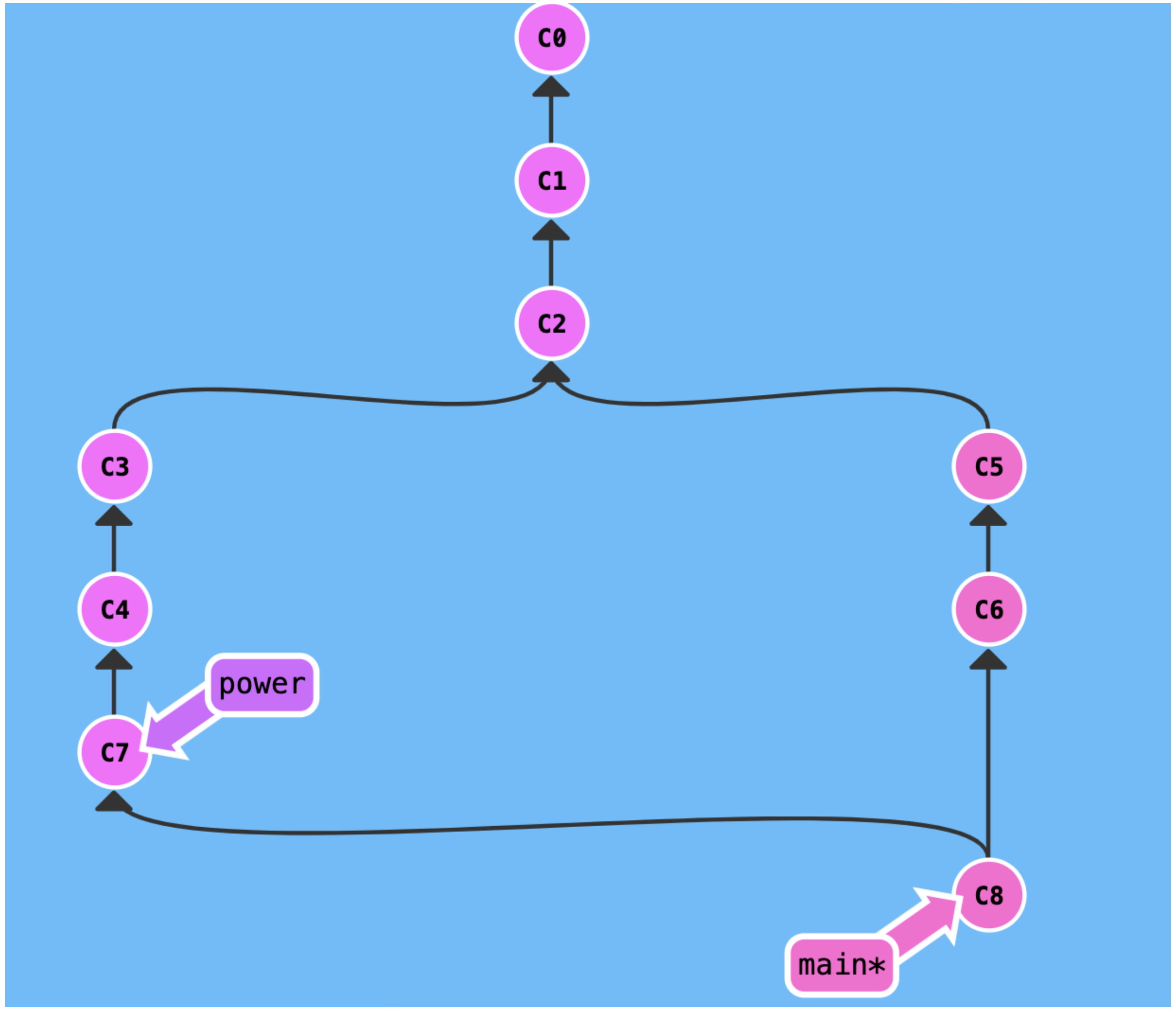


Three-way merge

Merge creates a new commit as both branches came from different paths

- Makes a three-way merge

(Afterwards we want to remove our branch)



Sometimes you need to solve conflicts

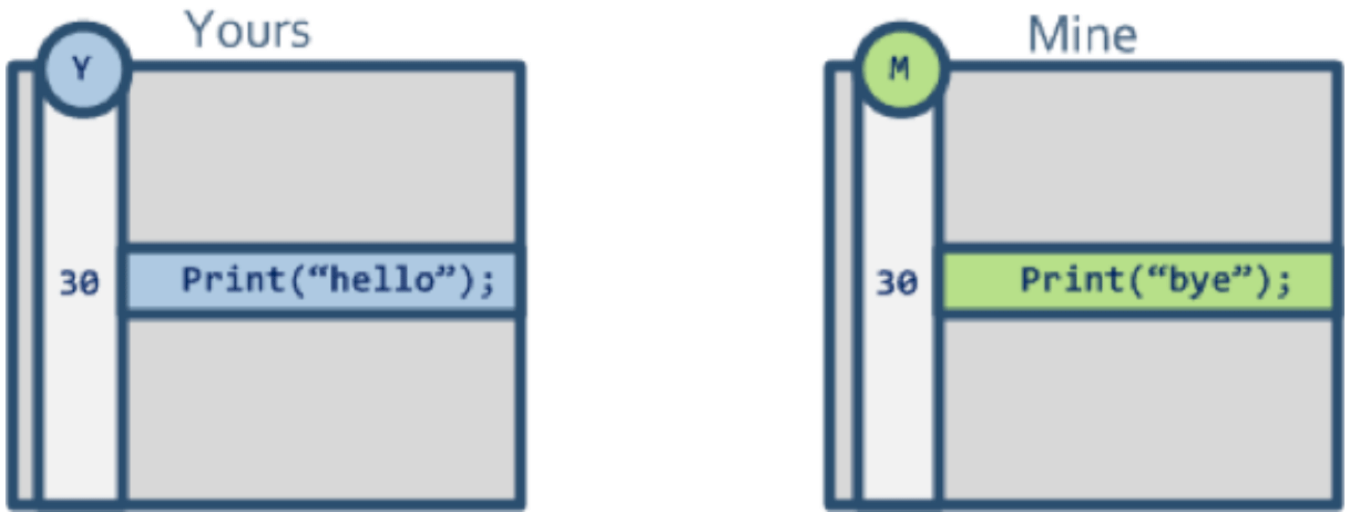
If you are developing software within a team, there will happen sometimes conflicts, which you have to solve before you can continue your work.

Approaches to solve conflicts:

- Two-way merge
- Three-way merge
  - Git is using three-way merge

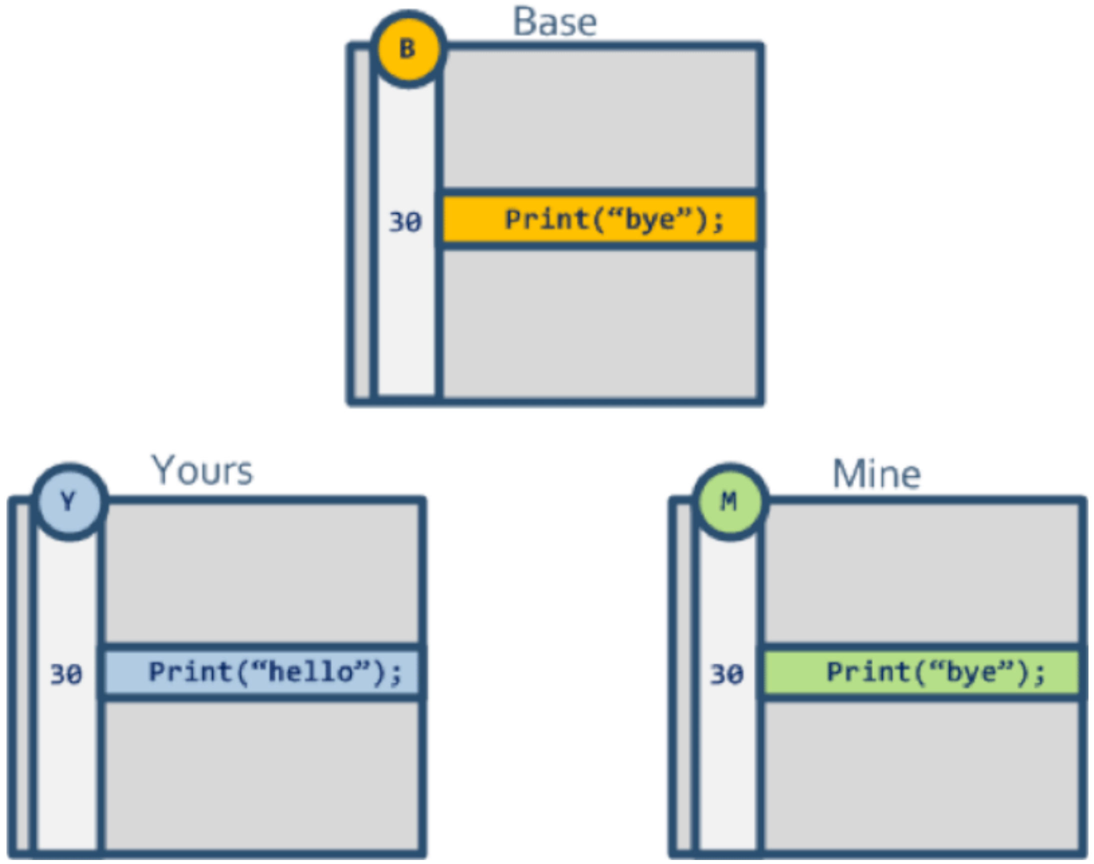
What is a two-way merge?

- Two developers
- Third Person is looking at those two files
- Are both files modified?

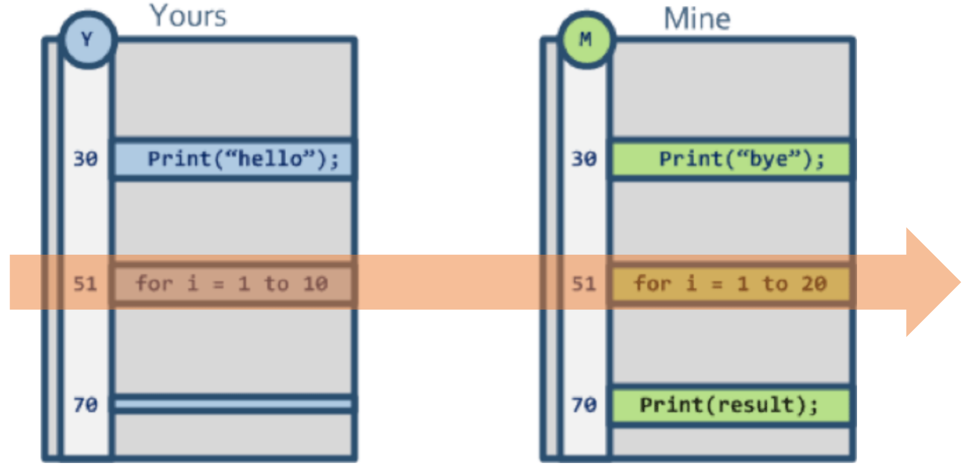


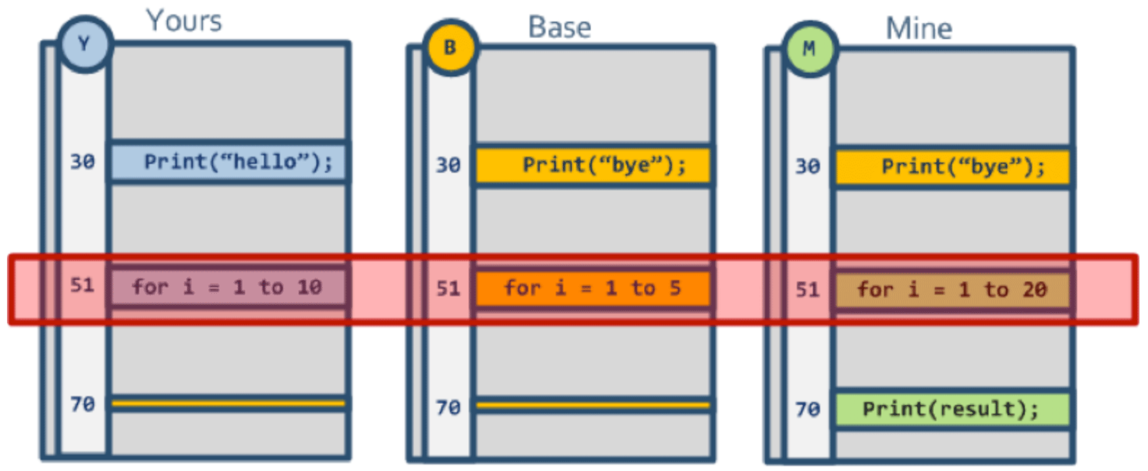
Is the third person able to answer the question?

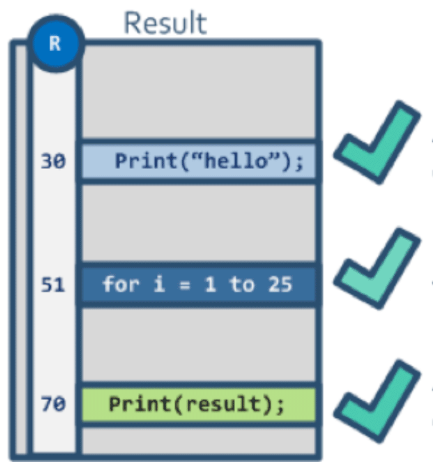
Three-way merging a (possible) solution



Sometime manual merge (szenarios) happens







Automatic – just keep "yours" (no changes on "mine")

Manual – resolved manually because there was a conflict!

Automatic – just keep "mine" (no changes on "yours") (I added the code)

Conflict: manual resolve necessary

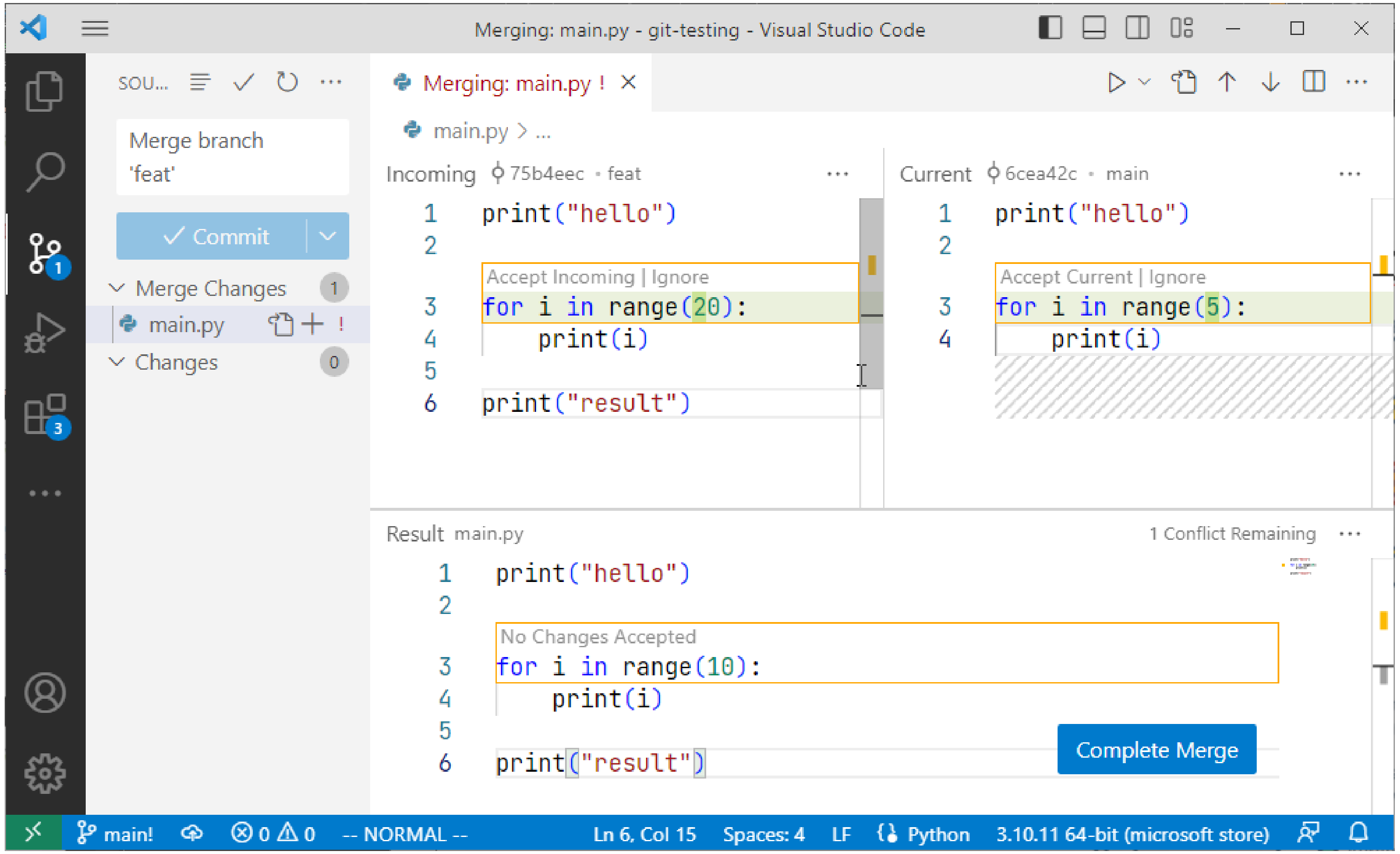
- Choose the correct soluton between „conflict dividers“

```
<<<<<<< HEAD
[Content current branch]
=======
[Content merging branch]
>>>>>>> new_feature
```

- Add and commit the changes to resolve the conflict (and end the merge process)

Conflict: manual resolve necessary

To resolve conflicts as best as possible it is recommended to use better tools, modern IDEs with git support for example offer a good overview for resolving merge conflicts.



Incoming:
```
print("hello")

for i in range(20):
    print(i)

print("result")
```

Current:
```
print("hello")

for i in range(5):
    print(i)
```

Result:
```
print("hello")

for i in range(10):
    print(i)

print("result")
```

Example of merging in visual studio code

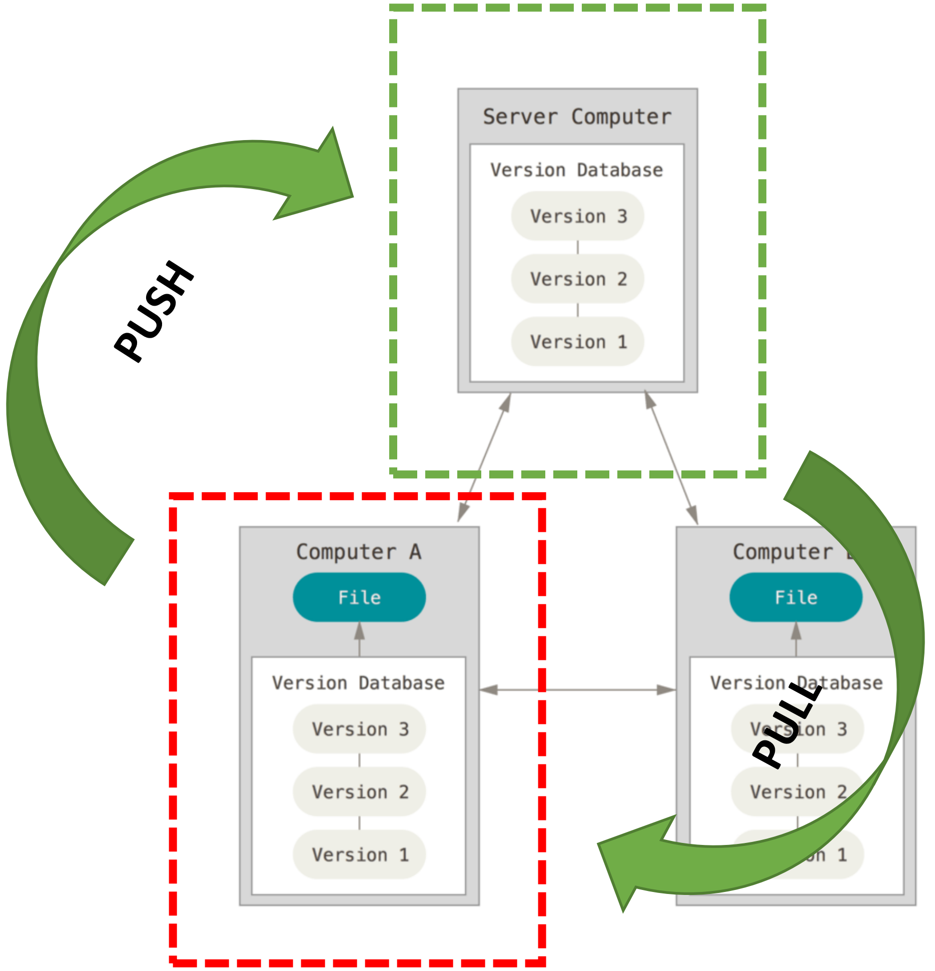
Work with remote operations

At the next few slides we will cover important parts of developing software within a team

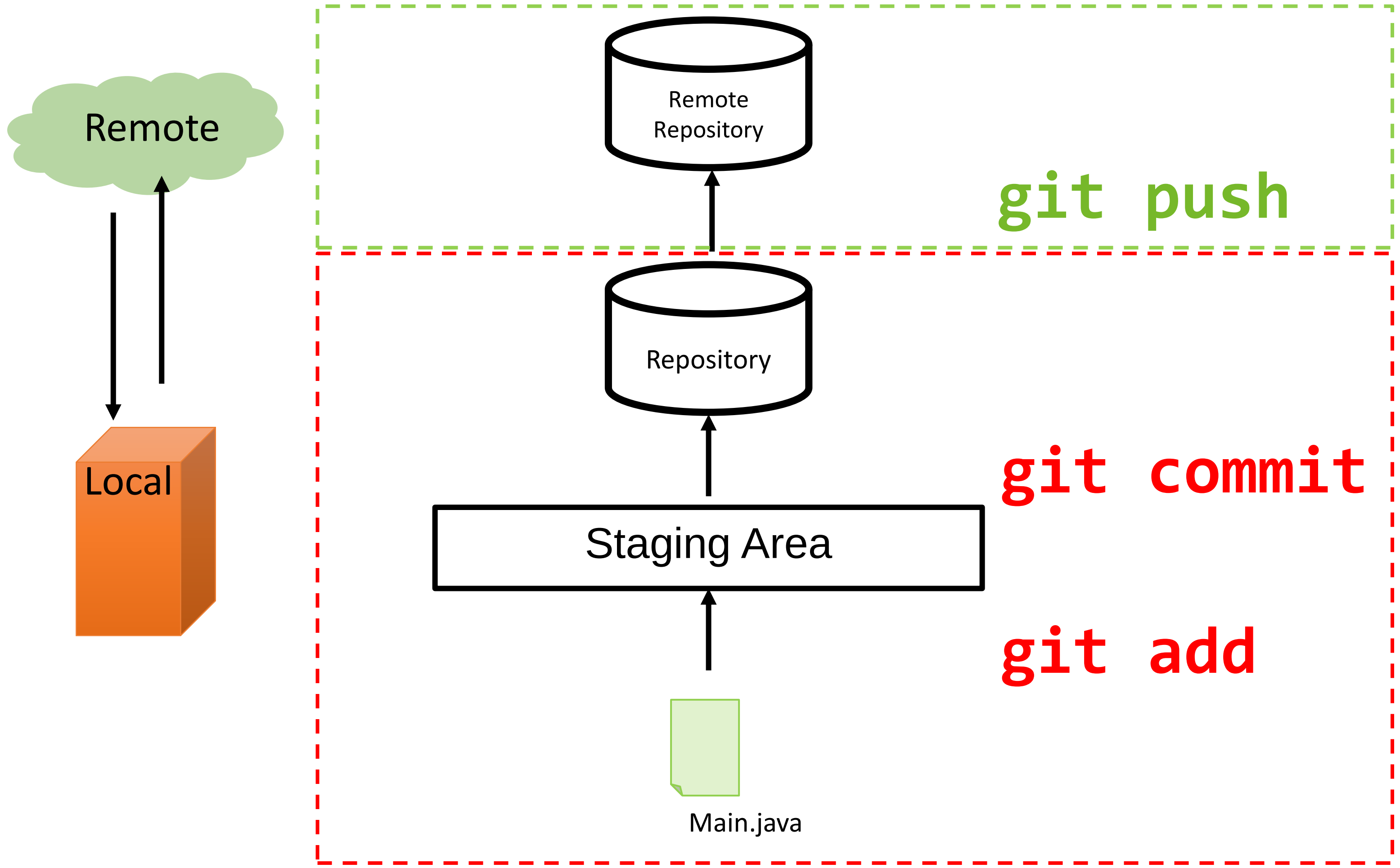
- Difference between **local** and **remote**
- Remote **Repository**
- **Push/Pull**
- Major commands

Local vs. Remote with Distributed Version Control System

- Developer‘s Computer A
  - **„Local“**
  - Local Repository
  - Working Directory
- Server Computer
  - **Remote Repository**
  - **„Remote“**
  - E.g. GitHub, GitLab, …

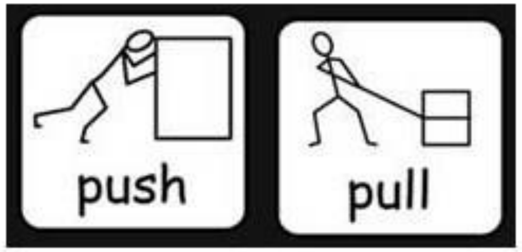


Working with a remote repository



Working with other developers (preview)

- Version of local repository hosted on the internet or local network (e.g. GitLab)
- Collaborate with other developers
  - The remote repository is necessary to simplify team collaboration.
  - A developer share latest commits / versions with **push** (transfer data to server) and another developer will receive latest commits / versions wih **pull** (transfer data from server)
- „It‘s about pushing and pulling“



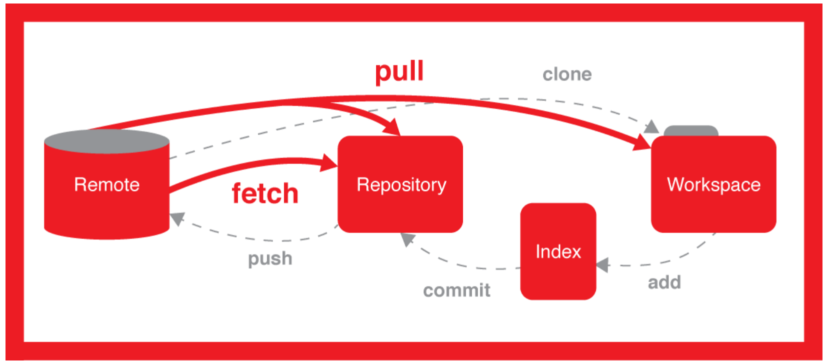
https://vijaysangamworld.files.wordpress.com/2010/07/push-vs-pull.jpg

Major "first" / "init" commands working with remotes

- **`git init`**
- `git add <filename>`
- `git commit -m "<commit message>"`
- **`git remote add origin <url>`**
- `git push -u origin main`
- **`git push --set-upstream origin main`**

OR

- **`git clone <url>`**



highlighted commands are the "same" procedure (first commands local -> remote, second part remote -> local)

https://stackoverflow.com/questions/292357/what-is-the-difference-between-git-pull-and-git-fetch

DIHSÜD

FH JOANNEUM
University of Applied Sciences

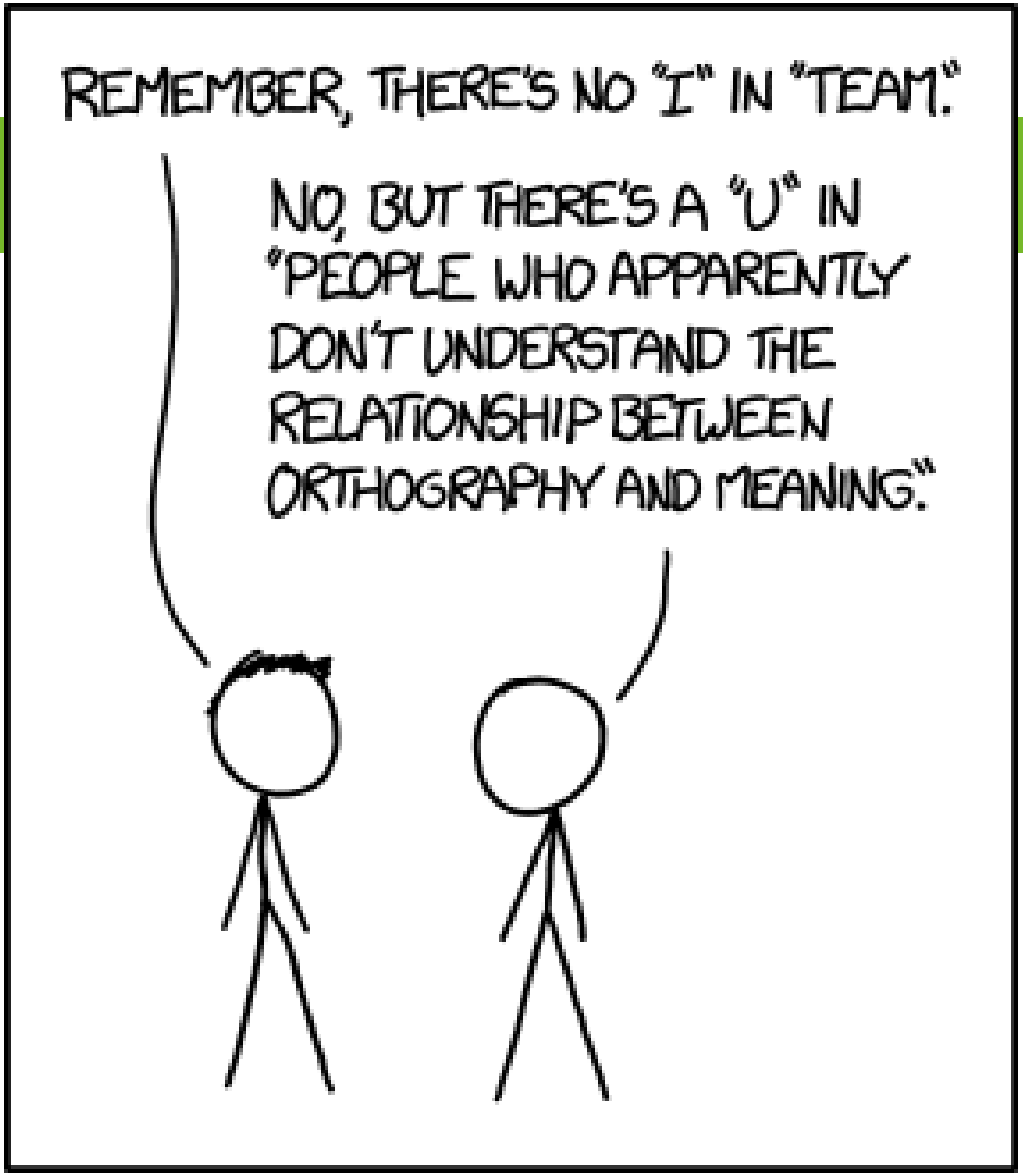
Git “Remote” Server

GitLab

GitHub

Bitbucket

Team work


REMEMBER, THERE'S NO "I" IN "TEAM."
NO, BUT THERE'S A "U" IN "PEOPLE WHO APPARENTLY DON'T UNDERSTAND THE RELATIONSHIP BETWEEN ORTHOGRAPHY AND MEANING."

Today

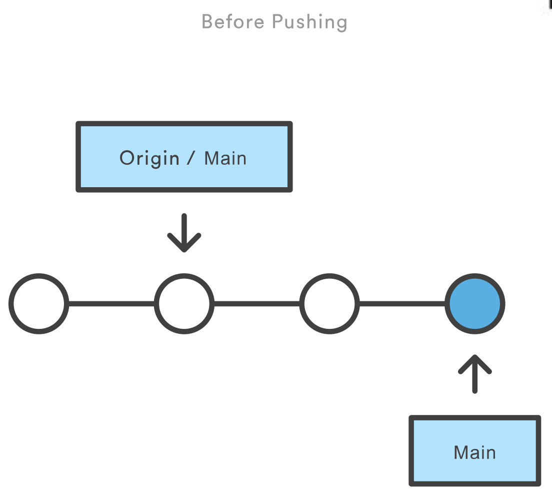
- Deadline for last exercise
- First experience/problems?
- Team work with git

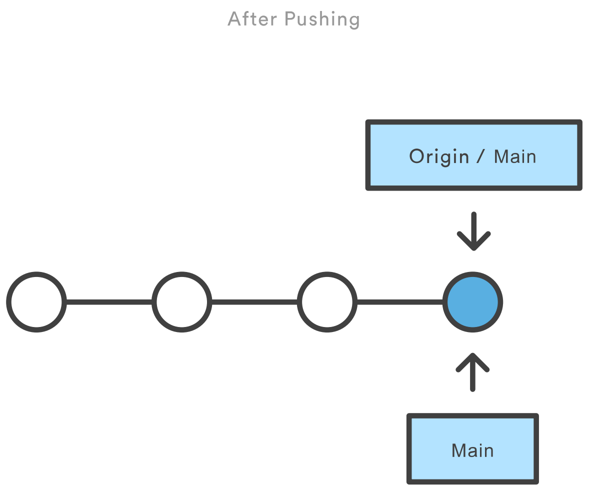
First experience or problems?

git push

- After changes in local repo **push** is normally used to share modifications of current branch with team by uploading changes to remote repository

- git push -u origin *branch_name*
  - Push local branch to remote repo
    - Only once necessary



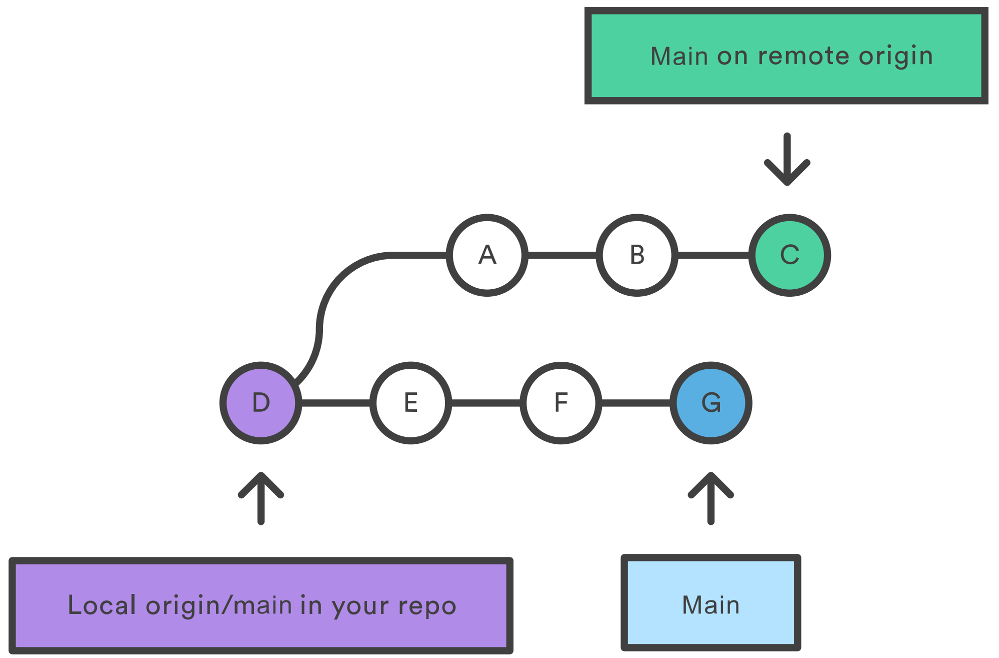


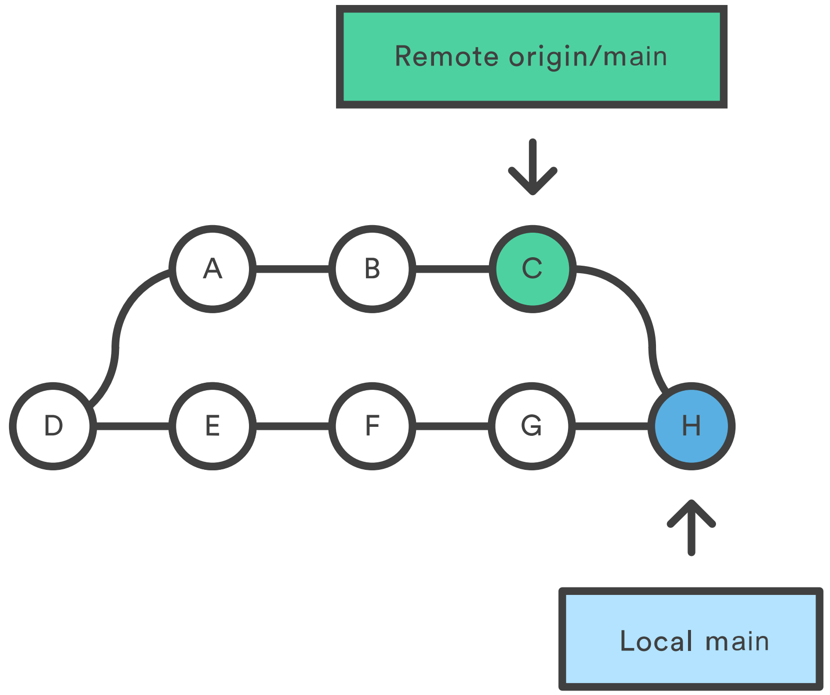
git fetch

- Downloads all branches with their commits from the central repository
  - Changes are not merged
    - Merge afterwards manually
    - Or change to new branch from remote

git pull

- Fetch the remote version of the current branch and merge it into local repo





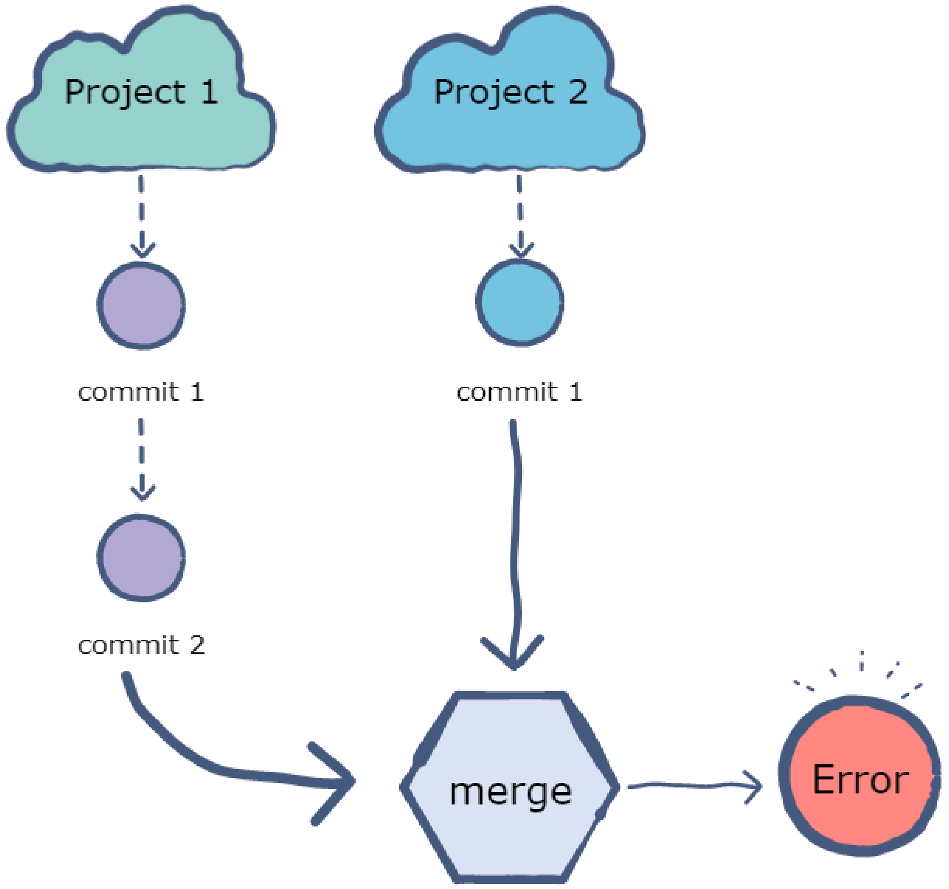
git branch

- If we have never used a remote branch we do not have a local version of it
  - Only after checkouts

- git branch –r
  - Shows remote branches
- git branch –a
  - Shows local and remote branches

git problem: merge unrelated histories

- If multiple team members clone an empty re this error can occur when the second person pulling (and therefore merging)
  - “fatal: refusing to merge unrelated histories”
- Fix it with
  - git pull origin master --allow-unrelated-histories



git connect local repo to remote

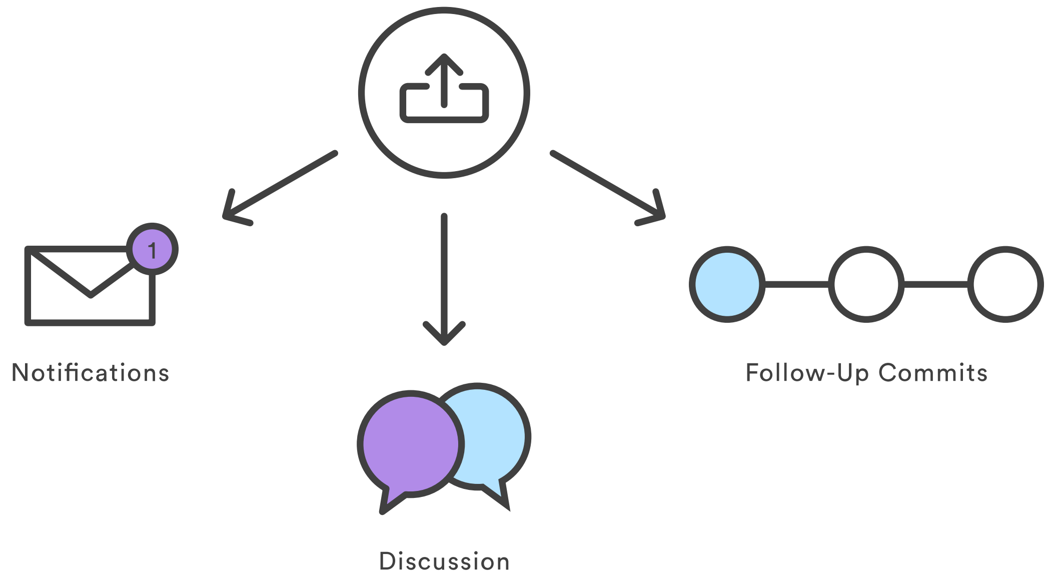
- Connect a locally initalized git repo with a remote repo (like Github)

- 1. Add remote git server
  - git remote add origin <URL>
- 2. Verify URL
  - git remote -v
- 3. Push changes
  - git push origin main

git Pull Request

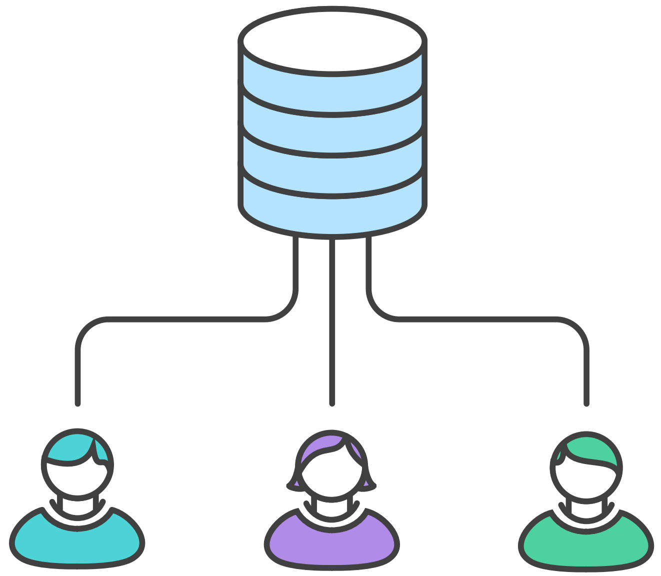
- A way to collaborate between developers
- Developer notifies team members that feature is complete and everybody is informed

- Review code and merge it into *main*
    - Changes can be discussed
    - Additional commits can be made



git Centralized Workflow

- One central repo for reading and writing files
    - Often adapted from older less flexible alternatives
- Changes are stored local and they are published by pushing
- Fetch before publishing necessary
    - Perfectly linear history

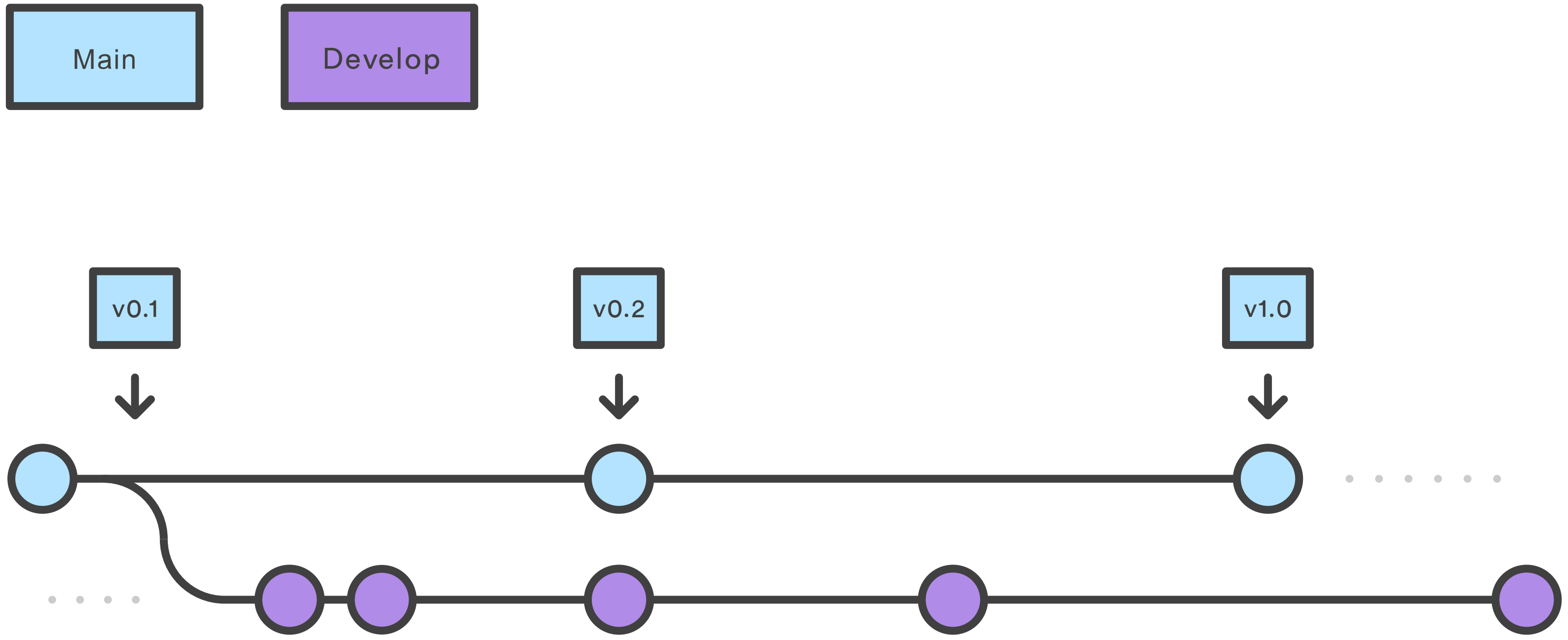


git Feature Branch Workflow

- All development should be realised in dedicated branch
  - main will never contain broken code
- Branch names should be descriptive
  - issue-#1234 or new-menu
- Feature branches can be pushed to central repo
- Discuss changes via pull request

git Gitflow Workflow

- Legacy flow with git branches
- Main branch stores official release history
- Develop branch as integration branch for features



git Forking Workflow

- Fundamentally different – every developer has own server-side repository
  - Not only one central repo
  - Often used in open-source software projects
- Each contributor has
  - Private local repo
  - Public server-side one
- Developer push to their own server-side repo
  - Open pull request to „official“ repo from maintainer
- Project maintainer can acccept contributions without giving write access to project

References

- *Günther Popp*
  Konfigurationsmanagement
  dpunkt.verlag, 2008

- *Scott Chacon, Ben Straub*
  Pro Git
  Apress, 2nd Edition, 2014

- *Jon Loeliger, Matthew McCullough*
  Version Control with Git
  O'REILLY, 2012

- *Bernd Öggl, Michael Kofler*
  Git – Projektverwaltung für Entwickler und DevOps-Teams
  Rheinwerk Verlag, 2020

Links

- *Atlassian*
  Become a git guru
  https://atlassian.com/git/tutorials

- *Peter Cottle*
  Learning Git Branching
  https://learngitbranching.js.org/

- *Github*
  https://github.com/

- *GitLab*
  https://gitlab.com/

- BitBucket
  https://bitbucket.com/

- *Git*
  Git Source Code Management
  https://git-scm.com/

  git Book
  https://git-scm.com/book/en/v2

  Git – Getting Started
  https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control