Software Engineering II: **Git**

Philipp Fruck

Introduction

What is a Version Control System (VCS)

- Version Control: Track different versions in history of a file, e.g. source code
- Version Control System (VCS): Software tool that handles version control
- Git is not the first and not the last VCS
  - There have been e.g. Mercurial (distributed) and Subversion (centralized)
  - Jujutsu is a modern, git-compatible VCS

Why should we use a VCS / `git`

- Track history
- Collaborate safely
- Experiment without fear
- Recover from mistakes
- `git`: Industry standard
  - Used by open source & enterprises
  - Mandatory skill for developers

View Linus Torvalds on git: https://www.youtube.com/watch?v=4XpnKHJAok8
Transcript: https://git.wiki.kernel.org/index.php/LinusTalk200705Transcript

Command Overview

Command	Description
`git init`	Initializes the repository
`git branch`	Modifies (`-m`) or deletes (`-d`/`-D`) a branch
`git remote`	Set a remote url, e.g. GitHub, GitLab, etc
`git add`	Add file contents to the index ("staging area")
`git commit`	Create new commit from index ("staging area")
`git push`	Push changes to remote repository
`git pull`	Pull changes from remote repository
`git fetch`	Fetch updates from remote repository without applying changes
`git status`	Show "state" of the repo (changes files, current branch, ...)
`git diff`	Show the current diff (use `--cached` to compare index)
`git restore`	Restore a given diff (use `--staged` to restore index)
`git reset`	Perform a reset to a given state

Something went wrong? **oshitgit.com** (https://ohshitgit.com/)

Index

Description

- Git utilizes an index, also called the staging area
- Buffer between the working directory and the repository history
  - Select which changes go into the next commit
  - Commit changes incrementally
  - Temporary snapshot
- Files are added to the index using `git add` and restored using `git restore --staged`

Gitignore

- To avoid files being added to the index, add them to `.gitignore`
  - Do this for e.g. local dependencies, config (secrets), ...
- Use `/foo` for a top-level file
- Use `foo` to ignore all `foo` files anywhere in the repo
- `.gitignore` files can be located in every subfolder of the repo

Stash

Description

Git allows getting rid of changes while saving them for later

- `git stash` saves local changes away
  - Your working directory is clean again
  - Changes are added to an internal stack
- You can show stashed changes using `git stash list`
- Get back stashed changes using `git stash pop` or `git stash apply`

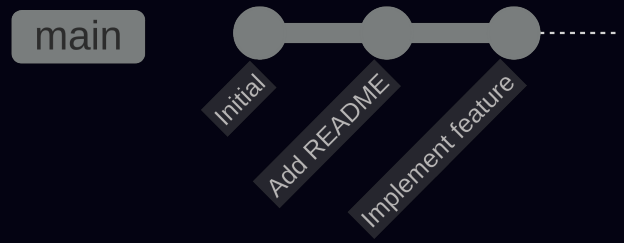
Commit

Description

A **commit** is a snapshot of the project

- Records file changes
  - Represented as diff
  - Stores full file contents
- Identified by a hash
- Contains metadata:
  - Author name and email
  - Commit date and time
  - Commit message describing the changes
- Linked to parent commit
  - Forms a Directed Acyclic Graph (DAG)

Example



⚠ Once in git, always in git. Only because you override a commit does not mean its content will disappear. Once pushed to a remote, the commit will remain there even if not referenced by an active branch.

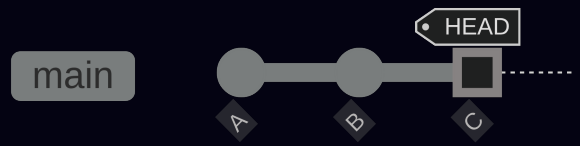
HEAD

Description

HEAD is a symbolic reference

- Pointer to the current commit "you're on"
- Moves when you commit
- Usually points to a branch (uses the latest branch commit)
  - Can also point to commits directly (detached HEAD)

Example



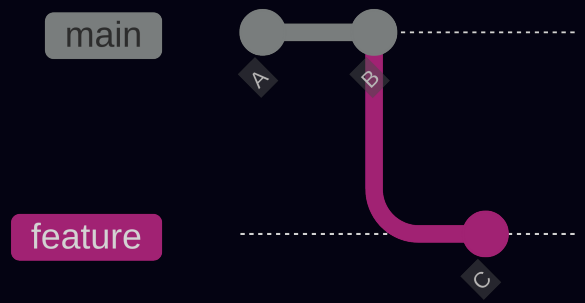
Branch

Description

A **branch** is a movable pointer (label/name) to a commit

- Represents a line of development
- Moves forward automatically as you make new commits
- Separates features
  - Encourages experimentation

Example



Multiple Branches

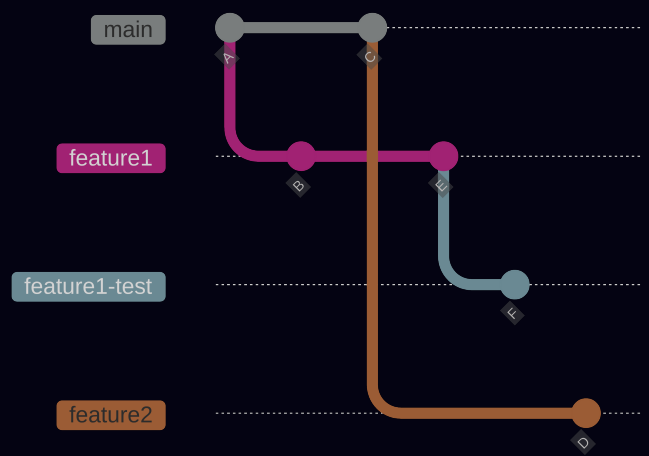
Description

Parallel development without interference

- Create as many branches as you want
- Work on multiple features simultaneously
- Build on features from colleagues without interfering

Question: How do we reunite?

Example



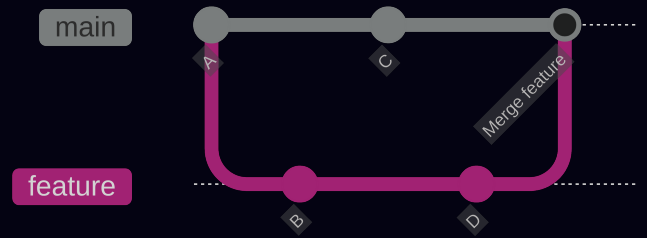
Merging

Description

Merging combines multiple branches

- Preserves history (no commits are lost)
  - Ordered by original timestamp
- Creates a merge commit
  - Special commit with 2 or more parents
  - Non-linear history
    - Use `git log --graph [--oneline]` to show
    - Easy to use, but suboptimal

Before merge



After merge

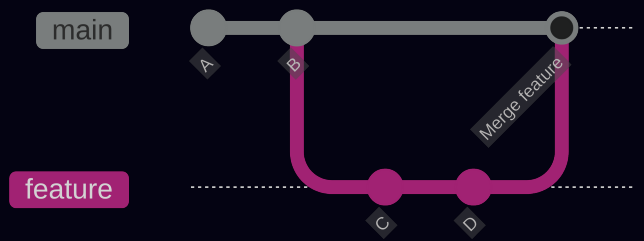
Fast-Forward Merge

Description

Occurs when no divergence exists

- No merge commit
- Just moves the branch pointer
  - Linear history remains

Before merge



After merge

Note: No merge commit after **D**

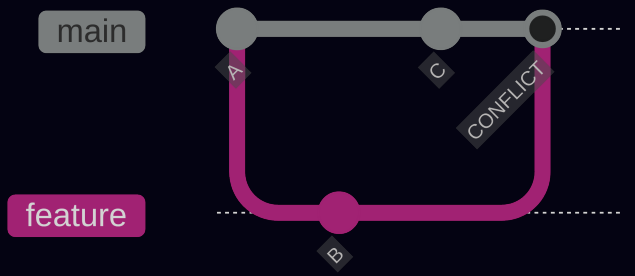
Merge Conflicts

Description

A **merge conflict** happens when Git is unsure

- Same lines changed differently
- Requires manual resolution

Example




```
# git show B
--- a/commit
+++ b/commit
@@ -1 +1 @@
-A
+B
```

```
# git show C
--- a/commit
+++ b/commit
@@ -1 +1 @@
-A
+C
```

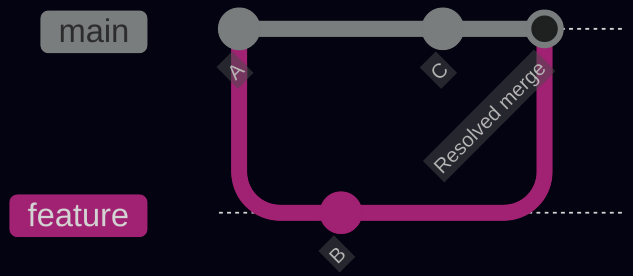
```
<<<<<<< HEAD
C
||||||| 5f1bca4
A
=======
B
>>>>>>> feat
```

Resolving Conflicts

What happens

- Git stops the merge
- You edit conflicted files
- Mark resolved (`git add`)
- Commit the result
  - Becomes the merge commit

Example



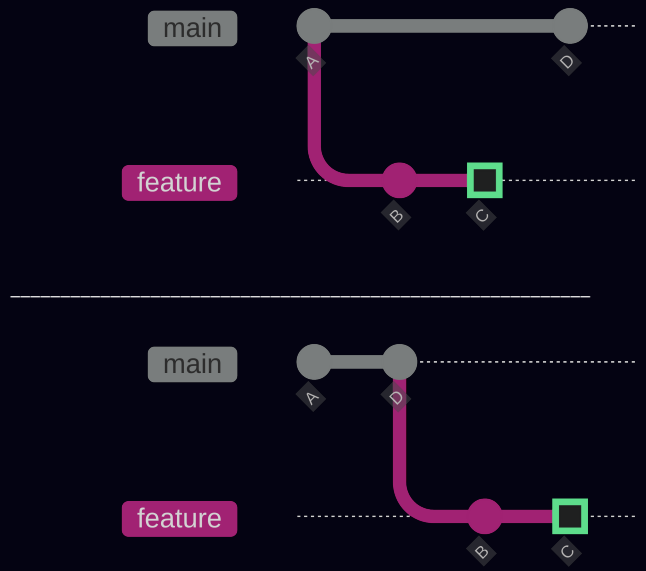
Rebase

Description

Rebasing moves commits to a new base

- Can be from branch to another
- Or just updating the base branch
- Produces a linear history
  - Keeps same logical order
- Rewrites hashes for new commits
- Replays commits one-by-one
  - Potentially multiple conflicts
  - Solve them like a merge conflict
  - Then `git rebase --continue`
- Hint: `git config rerere.enable true`
  - **re**use **re**corded **re**solution
  - Solve repeating conflicts automatically

Example



Rebase

Why?

- Cleaner history
- Easier to understand
- No unnecessary merge commits

Recommendations

⚠ Do not rebase shared branches

⚠ Use `git push --force-with-lease` for "careful" forced update

- Only forces an update if remote branch has not changed

Interactive Rebase

Description

Rewrite commits interactively

- Reorder commits
- Edit messages
- Drop commits
- Squash commits

Example:

- `git rebase -i A`
  - `A` is the last original commit hash
  - Leave `C` as `pick`
  - Set `B` to `delete` or `d`

Before rebase


```
pick 9124c1b # B
pick 00553b4 # C

# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit
message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into
previous commit
# d, drop <commit> = remove commit
# [...]
```

After rebase

Squashing Commits

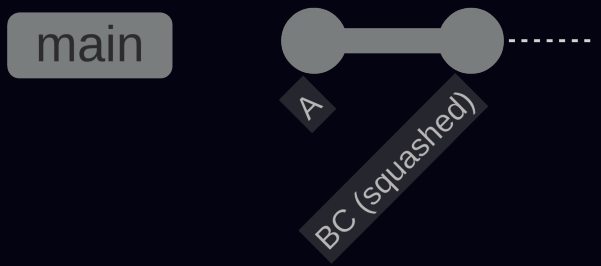
Description

Squashing combines commits

- Reduces noise
- Creates meaningful history
- To squash multiple commits, use an interactive rebase
  - e.g. `git rebase -i HEAD~2` to squash the last two commits
  - Leave the first commit as `pick`
  - Set the second to squash
  - Define the commit message for the squashed commit

Before Squash

After Squash



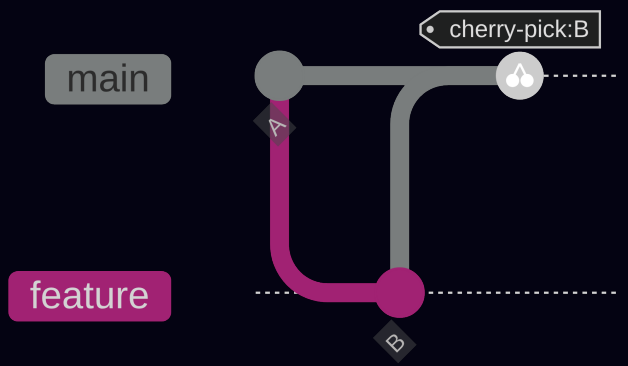
Cherry-Pick

Description

Cherry-picking applies a single commit

- Useful for hotfixes
- Avoids full merges
- Relevant for backporting
- e.g. `git cherry-pick B`

Example



Remote

Explanation

A **remote** is another repository

- Often shared
- Enables collaboration
- Example: origin

Pull Requests

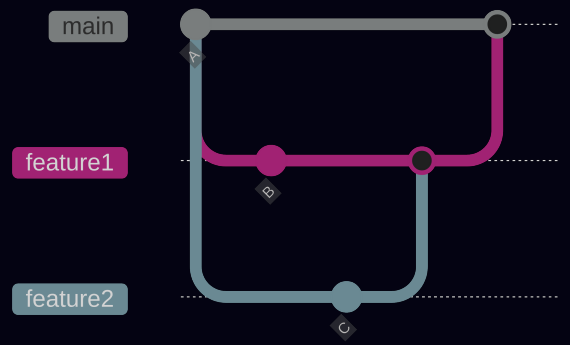
A **pull request** is a review process

- Discuss changes
- Run tests (and more automation)
- Merge safely

Best Practices

- Use feature branches
- Commit small changes
- Write clear messages
- Avoid force-push (except for rebases)
- Enable branch protection
- Rebase, then squash-merge (personal preference)

Example



Commit Messages

- Keep commit messages short and descriptive
  - No "small fixes" -> What did you fix?
- Write a summary on the first line
- More details (if necessary) after one blank line
- Add co-authors after another blank line

```
docs: short commit summary in present tense
# |<----  Using a Maximum Of 50 Characters  ---->|

After one blank line, we can start a more detailled message. If you want
to employ a consistent message style, consider conventional commits:
https://www.conventionalcommits.org/en/v1.0.0/
There is also https://gitmoji.dev, but I don't like it personally
# |<----   Try To Limit Each Line to a Maximum Of 72 Characters   ---->|


# Co-authored-by: Author <author@example.com>
```

Versioning

Tags

- Tags can be used as an alias to a given commit
- `git tag v1.0.0` marks the current commit as version v1.0.0 of the software
  - Can be used later on to see changes between v1.1.0 and v1.0.0
  - Both diff and commit messages -> Changelog
- Tags can be pushed to the remote via `git push --tags`

Releases

- There are no "releases" in git
- Releases are a concept of GitHub, GitLab, etc.
- Releases are bound to a tag and can contain additional description, release binaries, etc.
  - Pipelines can be used to create releases from tags
  - We will do this in a later lecture

Config

There are three types of git configs:

1. System: `git config --system` or `/etc/gitconfig`
2. Global (user): `git config --global` or `~/.gitconfig` / `~/.config/git/config`
3. Local (repo): `git config --local` or `<repo>/.git/config`

Example settings:

User Config

```
[user]
    email = a.b@c.de
    name = Hacker Man
```

Convenience Features

```
[branch]
    sort = -committerdate
[rerere]
    enabled = true
```

For more convenient diffs, start using **delta** (https://github.com/dandavison/delta)

Signing

Impersonation

- In git, you can define your name and email as you want
  - No verification by e.g. GitHub etc
  - Makes it possible to blame other persons for your fault
- Solution: Commit signing
  - Cryptographically proves who created a commit
  - Ensures authenticity and integrity
  - Supports GPG and SSH keys
  - GPG has key expiry and trust levels, but SSH is simpler

```
# GPG example
[user]
    signingkey = ABCDEF1234567890
[commit]
    gpgsign = true
[gpg]
    program = gpg
```

```
# SSH example
[user]
    signingkey = ~/.ssh/id_ed25519.pub
[commit]
    gpgsign = true
[gpg]
    format = ssh
```

Config (bonus)

If you want to share different git configs amongst multiple projects, consider the following config:

```
# Enable per-folder git config
[includeIf "gitdir:~/dev/personal/"]
  path = config.personal
[includeIf "gitdir:~/dev/work/"]
  path = config.work
[includeIf "gitdir:~/dev/dhbw/"]
  path = config.dhbw
```

Git Hooks

Description

Git also has more advanced configuration options that allow automating tasks

- Add files in `.git/hooks/` and make them executable
- Different hooks are called at different times
  - `pre-commit`: Runs before a commit is created
    - Use for linting, formatting, ...
    - Can abort the commit with non-zero exit code
  - `pre-push`: Runs before pushing to a remote
    - Can be used for tests etc.
  - `post-merge` / `post-checkout`: Runs after merging or switching branches
    - Reinstall dependencies, generates files, update $stuff

Tools

- Tools like **pre-commit** (https://pre-commit.com) can be used to run checks before your commit
  - Once configured, add the hook using `pre-commit install`

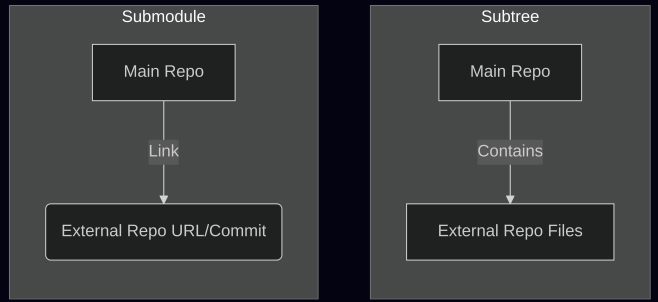
Git Submodules vs. Subtrees

How to include external Git repositories within your project?

Comparison at a Glance

Feature	Git Submodules	Git Subtrees
Storage	Pointer to a specific commit	Entire history merged into repo
Workflow	`git submodule update` required	Standard `git push/pull`
File Size	Lightweight (metadata only)	Heavier (includes full source)
Visibility	Clearly separated	Seamlessly integrated

Visual Workflow



Secrets

How do we handle secrets in git?

- Wo don't.
  - Add local secrets to the gitignore as mentioned in the beginning
- But if we have to?
  - This can happen in GitOps workflows
  - Use SOPS (Secrets OPerationS) to do so
    - Supports GPG, age, KMS
    - Encrypt file for multiple users
    - With correct config, git can produce a plaintext diff

⚠ Recall the definition of cryptographically safe. While breaking the encryption takes an unrealistic amount of time, you should consider only using this in private repos and/or rotating such secrets every no and then :)

Thank you for your attention!

Don't forget the feedback