# The basics of git

ZWE Software Workshop

Max-Planck-Institut für Intelligente Systeme

Python Introductory Workshop, Stuttgart, December 14, 2020

# Outline

# Outline

# SDLC

One of the **most important mission** of the Software Worksop is to **teach** researchers about **good software development practices and new technologies**.

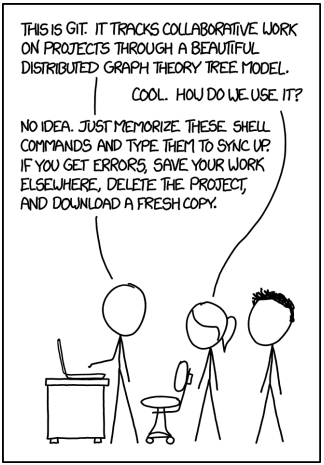

**S**ofware **D**evelopment **L**ife **C**ycle (**SDLC**) is the process followed for the development of a software product. Its goal: is to produce a software:

- with the **highest quality** (bug-free, stable, robust, extensible, following the customer's requirements, ...)
- for the **lowest cost** (money, time, man power, ...).

# Outline

# How people talk about git

**A lot** of the internet if about git related questions





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

## But what is it really?

git is a **V**ersion **C**ontrol **S**ystem (**VCS**).
**VCS** are tools to manage and organize information changes in a code. They allow you to:

- work collaboratively
- work on several files
- work on several tasks
- keep track of your development

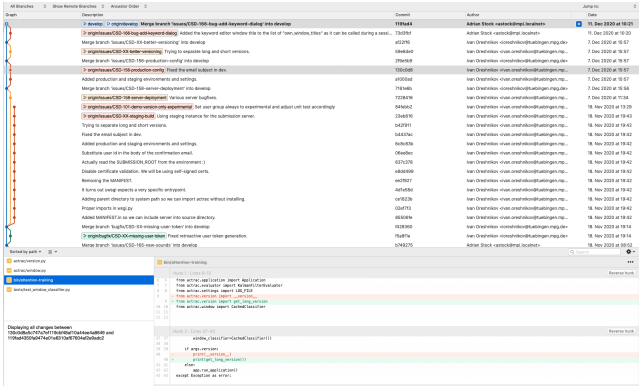
git is **distributed** (DVCS), $\neq$ SVN which is centralized:

- complete code base is stored on everyone's computer
- not dependent on network connection
- faster
- allows private work

# Example

Git is **big** and can be **complex** and **complicated**:

$\Longrightarrow$ **Use a proper client**



# Outline

# Identity

## Global settings

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john.doe@tuebingen.mpg.de"
```

## Note

- Name/emails can be set for each individual repository.
- Name/emails can be fixed afterwards by history rewrite (`git filter-branch`)

# Ignore rules

Some files should not be part of the repository (temporary files, by-products, ...).
Use a `.gitignore` to ignore files **locally** or **globally**.

**Global .gitignore**

```
$ git config --global core.excludesfile $HOME/.gitignore_global
```

**Warning**

.gitignore is shared with other developers. Think carefully before making changes!

# Outline

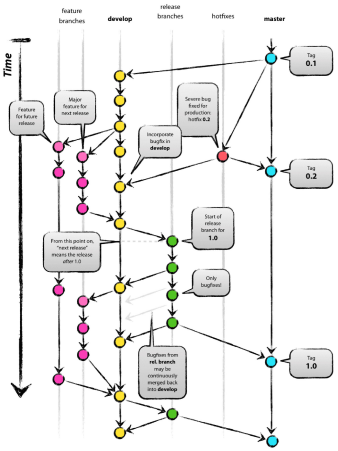
# Philosophy

## Branches

- Branches are just pointers to commits
- Isolate implementation changes
- Switch contexts

## Workflow

- New dev. starts on feature branch
- Usually start on reference stable state
- Always merge from stable into less stable
- Develop almost always stable
- Master always stable



## Basic commands

| | |
|---|---|
| Clone repository | `git clone url` |
| Request changes from remote | `git fetch --all` |
| Work with branches | `git checkout my_branch` |
| Pull changes from the remote | `git pull` |
| Show commit log | `git log` |
| Display the difference between commits | `git diff` |
| See the status of the repository | `git status` |
| Add files to the staging area | `git add --update/--all` |
| Place non-committed changes to a shelf | `git stash apply/pop/drop` |
| Merge last commit with the current one | `git commit -m "Your message"` |
| Push changes onto the remote | `git push` |

## Demo

**Task**: add your name to the README on a dedicated branch

1. Checkout the base branch on which you want to create you branch (double-click on the commit).The current branch appears in bold.
2. Create a new branch by clicking on the `Branch` button. You are now on this branch.
3. Edit the files.
4. `Uncommited changes` appears in the history, click on it. Details appear in the lower part.
5. Stage the changes you want to commit (checkbox or drag-and-drop).
6. Click on the `Commit` button, write your message, and commit.
7. Push onto the remote.

# Merging

Takes the union of changes (opposite action to branching).

- **Conflict resolution is part of life!**
- Only merge when the branch is **ready** (stable)
- Create new commit to keep topology (**no fast-forward**)
- Delete branches after some time once they have been merged

**Task**: let's do some merging!

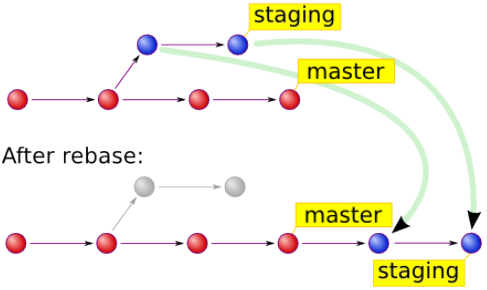
## Demo merging

**Task**: merge your branch onto develop and resolve conflicts if needed.

1. Checkout the base branch (develop) onto which you want to merge your feature branch.
2. Click on the `Merge` button.
3. Select your feature branch and merge.
4. If there are conflicts, the conflicted files will appear in the lower part with a warning sign.
5. To resolve conflicts, righ-click on the file and choose `Resolve Conflicts > Launch External Merge Tool`.
6. A window with 3 panels should appear: content on the base branch (left), content on the feature branch (right), and content that you want after the merge (bottom). This last panel can be edited directly.
7. Save, close the merge tool, and stage the file that has been resolved.
8. Do so for all the files containing conflicts, and commit.
9. Push to the remote.

# Rebasing

Moves the changes made on a branch onto another commit.




$\approx$ projects a branch implementation to the space **orthogonal** to other feature branches.

## Rebasing

Advantages:

- Keeps branch up-to-date with latest stable
- Keeps branch focused
- Make history shorter and therefore clearer

Workflow:

- Rebase your changes locally
- Make a diff between the local and remote branches: should be **orthogonal** to the feature implemented
- Force push to remote:

**Force push**

```
$ git push -f
```

# Dangers of rebasing

**Warning**

- History is changed, so notify your colleagues working on this branch.
- All local copies should be synchronized:

```
$ git fetch --all
$ git checkout my_branch
$ git reset --hard origin/my_branch
```

**Task**: let's do some rebasing!

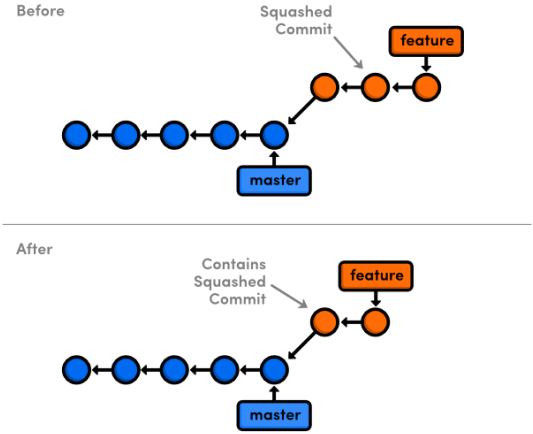
## Demo rebasing

**Task**: rebase your branch on top of develop.

1. Checkout the feature branch you want to rebase.
2. In the left panel, right-click on the base branch you want to rebase onto (develop) and select `Rebase current changes onto ...`
3. If there are conflicts, resolve them as you did when merging.
4. At any point, you can stop the rebasing by selecting `Actions > Abort Rebasing`.
5. Once the conflicts have been resolved, select `Actions > Continue Rebasing`.
6. **Sanity check:** confirm that differences between the rebased local branch and the remote are **orthogonal** to the feature implemented.
7. Force push to the remote.

# Interactive Rebasing

Squashes the commits and rewrites history



# Advantages of interactive rebasing

- Increases $S/N$ by diminishing history
- Makes branch scope easier to understand
- Makes branch easier to handle (rebase, merge, conflicts)

**Task**: let's do some interactive rebasing!

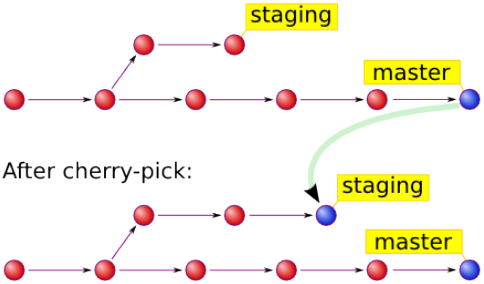
## Demo interactive rebasing

**Task**: squash the commits on your branch.

1. Checkout the feature branch you want to rebase.
2. Select the branch root, i.e. parent commit of the first commit you want to squash.
3. Right-click and select `Rebase children of ... interactively`.
4. Squash the commits, edit the commit messages, and confirm.
5. **Sanity check:** confirm that there is **no difference** between the rebased local branch and the remote.
6. Force push to the remote.

# Cherry-picking

Apply changes introduces by some commit onto the current branch.




**Task**: let's do some cherry-picking!

## Demo cherry-picking

**Task**: apply a change from another branch onto your branch.

1. Checkout your feature branch.
2. Select the commit you want to cherry-pick.
3. Right-click and select `Cherry Pick`.
4. If there are conflicts, resolve them as you did when merging.
5. Push to the remote.

## Last few tips

- **Protect** master/develop against force push/developers!
- Empty folders are not tracked. Good practice is to use a `.keep` file.
- By default, Sourcetree does not allow you to force push. The option must be first enabled in `Settings > Advanced`.
- These concepts are sufficient for $\approx$ 90% of what you will need to do.