

The basics of git and a bit more...

Jean-Claude Passy

ZWE Software Workshop

IMPRS-IS S4 Workshop, October 28, 2021

MAX PLANCK INSTITUTE
FOR INTELLIGENT SYSTEMS



Outline

- 1 Standards for Software Development
- 2 What is git?
- 3 Tutorial
- 4 Advanced topics
- 5 Conclusion

Outline

1 Standards for Software Development

2 What is git?

3 Tutorial

4 Advanced topics

5 Conclusion

Academia vs. Industry

We often try to compare/oppose academia and industry:

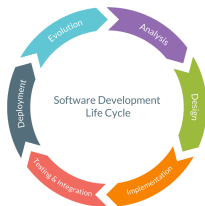
	Academia	Industry
End-goals	Paper, code, start-up	Product (software, car, rocket...), \$
Pressure	Supervisor, deadline,	N+1, N+2,..., customer, credibility
Tasks	Read papers, write code, write paper...	<i>idem</i> + talk to customer, business analyst, activity report
Keywords	Freedom, flexibility, unpredictability, discovery	Process, structure, validation, traceability

The Software Workshop believes that academia can benefit from some of the standards/principles used in the industry, for instance:

- Process comes with some overhead but helps you to stay focused
- Software should be verified and validated (see session about testing)
- During development, one should keep track of who has done what and when

SDLC

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



Software Development Life Cycle (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, modular, *working*)
- for the **lowest cost** (money, time, man power, ...).

The goal of this workshop is to introduce you to these standards and techniques.

Outline

- 1 Standards for Software Development
- 2 What is git?
- 3 Tutorial
- 4 Advanced topics
- 5 Conclusion

How people talk about git

A lot of the internet if about git related questions:



	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
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

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



Version control

Version control is the process of managing and organizing information changes. It is done using **Version Control Systems (VCS)**. The most popular (and really the only option) is [git](#).

It's main advantages are:

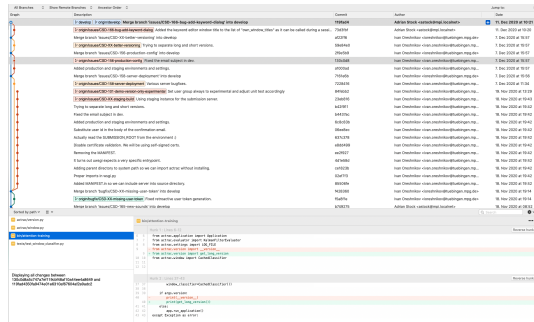
- complete code base is stored on everyone's computer
- work collaboratively
- work simultaneously on several files
- work simultaneously on several tasks
- traceability
- rollback
- reproducibility

More about git

git (unlike svn) is **distributed** (DVCS):

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

git is **complex**, we advise to use a **client**:
SourceTree (macOS/Win), **GitKraken**, or
solutions integrated to your IDE.



Outline

- 1 Standards for Software Development
- 2 What is git?
- 3 Tutorial**
- 4 Advanced topics
- 5 Conclusion

Create a git repository

Git repositories are usually created on the server using the web interface, then cloned:

Cloning

```
$ git clone https://github.com/MPI-IS/workshops.git  
$ git clone git@github.com:MPI-IS/workshops.git
```

You can also create a repo from an existing folder and update the remote:

Init and push

```
$ git init  
$ git branch -M master  
$ git add/commit ...  
$ git remote add origin https://github.com/MPI-IS/workshops.git  
$ git push -u origin master
```

Vocabulary

Local and Remotes

- Two distinct environments: your local repo and the remote repo
- They can evolve independently and the remote is shared among users
- `origin` denotes what is on the remote repo

Commit

- A commit is a snapshot of your repository
- Properties: datetime, author, UID, description
- At least one parent commit (except for the first commit)

Branch

- Branches are simply pointers to commits
- They can be created, moved, reset

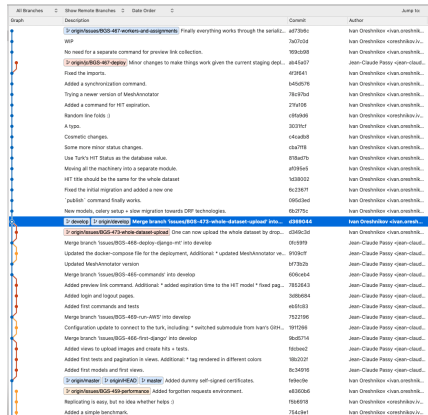
Philosophy

Branches

- Isolate implementation changes
- Switch contexts

Workflow

- New dev. starts on feature branch
- Usually start on reference stable state
- When *ready* merge the feature branch
- Always merge from stable into less stable
- Develop almost always stable
- Master always stable (releases)

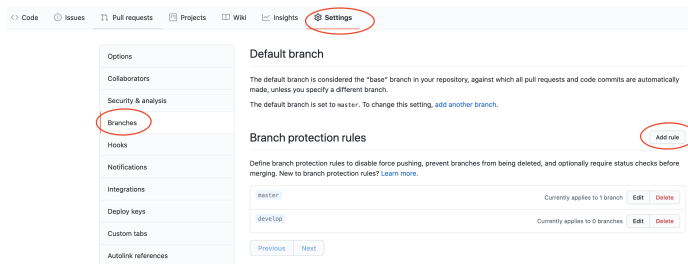


The image shows a Git commit history graph. The left side is a vertical timeline with colored dots representing commits. The right side is a table with columns: All branches, Show Remote Branches, Date Order, Commit, Author, and Jump to. The table lists commits with their descriptions and authors. The commits are organized into branches: master (stable releases), develop (integration branch), and feature branches (used for developing new features). The commits show a clear workflow: new features are developed on feature branches, merged back to develop when ready, and then merged to master for production releases.

All branches	Show Remote Branches	Date Order	Commit	Author	Jump to
Graph					
			origin/issue805-487-workers-and-assignments Finally everything works through the serial...	ad7396c	Jan Oreschke <jan.oreschke@...
			WIP	7a070d4	Jan Oreschke <jan.oreschke@...
			No need for a separate command for preview link collection.	903d098	Jan Oreschke <jan.oreschke@...
			origin/805-487-deploy Minor changes to make things work given the current staging dep...	8b46a57	Jean-Claude Passy <jcp@cloud...
			Fixed the imports.	42f8f41	Jan Oreschke <jan.oreschke@...
			Added a synchronization command.	1a05d78	Jan Oreschke <jan.oreschke@...
			Trying a newer version of MeshKnotator	78c67bd	Jan Oreschke <jan.oreschke@...
			Added a command for HT expiration.	29f406	Jan Oreschke <jan.oreschke@...
			Random line fields ()	c8f9a98	Jan Oreschke <jan.oreschke@...
			A types.	30371cf	Jan Oreschke <jan.oreschke@...
			Cosmetic changes.	c4eadd8	Jan Oreschke <jan.oreschke@...
			Some more minor status changes.	cb47f18	Jan Oreschke <jan.oreschke@...
			Use Turk's HT Status as the database value.	878ad7b	Jan Oreschke <jan.oreschke@...
			Moving all the machinery into a separate module.	a039561	Jan Oreschke <jan.oreschke@...
			HT title should be the same for the whole dataset	1a05d78	Jan Oreschke <jan.oreschke@...
			Fixed the initial migration and added a new one	8c23a7f	Jan Oreschke <jan.oreschke@...
			'publish' command finally works.	095d3ed	Jan Oreschke <jan.oreschke@...
			New models, colary setup + size migration towards DDF technologies.	6b2757c	Jan Oreschke <jan.oreschke@...
			develop origin/develop Merge branch 'issue805-473-whole-dataset-upload' into...	d388014	Jan Oreschke <jan.oreschke@...
			origin/issue805-473-whole-dataset-upload One can now upload the whole dataset by drop...	d388014	Jan Oreschke <jan.oreschke@...
			Merge branch 'issue805-488-deploy-django-ent' into develop	0f03999	Jean-Claude Passy <jcp@cloud...
			Updated the docker-compose file for the deployment. Additional: * updated MeshKnotator ve...	9f03a7f	Jean-Claude Passy <jcp@cloud...
			Updated MeshKnotator version	8f73b2a	Jean-Claude Passy <jcp@cloud...
			Merge branch 'issue805-488-command' into develop	600e4b4	Jean-Claude Passy <jcp@cloud...
			Added preview link command. Additional: * added expiration time to the HT model * fixed pag...	7832443	Jean-Claude Passy <jcp@cloud...
			Added login and logout pages	3a06484	Jean-Claude Passy <jcp@cloud...
			Added first commands and tests	4b161d3	Jean-Claude Passy <jcp@cloud...
			Merge branch 'issue805-489-run-AWS' into develop	7322196	Jean-Claude Passy <jcp@cloud...
			Configuration update to connect to the turk, including: * switched submodule from huan's GH...	09d7794	Jean-Claude Passy <jcp@cloud...
			Merge branch 'issue805-488-first-django' into develop	8b0d708	Jean-Claude Passy <jcp@cloud...
			Added views to upload images and create hits + tests.	f03b6c2	Jean-Claude Passy <jcp@cloud...
			Added first tests and pagination in views. Additional: * tag rendered in different colors	18b2021	Jean-Claude Passy <jcp@cloud...
			Added first models and first views.	8c34916	Jean-Claude Passy <jcp@cloud...
			origin/master origin/HEAD origin/develop origin/issue805-489-performance Added dummy self-signed certificates.	febf0e6	Jan Oreschke <jan.oreschke@...
			origin/issue805-489-performance Added forgotten requests environment.	4336c36	Jan Oreschke <jan.oreschke@...
			Refactoring is slow, but is it worth whether helps ()	19a6879	Jan Oreschke <jan.oreschke@...
			Added a simple benchmark.	754c8e1	Jan Oreschke <jan.oreschke@...

Protected branches

First thing to do is to create the *basic* branches **master/main** and **develop**, and to protect them against human errors. On GitHub:



If you are using Gitlab or a GitHub repo owned by an organization, you can **restrict who can push to matching branches** (user, role...).

Configure your identity

It is now time to set up some global settings

Configuration

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

On SourceTree, these parameters (and more) can be set under Preferences/General. Additionally, names/emails can be set for each individually for each repository:

- use the same command without the `--global` option
- on SourceTree, go to Settings/Advanced for the given repository

In case you committed with the wrong information, you can still fix the history afterwards by using the `git filter-branch` command.

Ignore rules

Some files should not be part of the repository: temporary files, by-products...
These files should be specified in a file named `.gitignore` at the root of the repository.

```
haptipedia > << .gitignore
1  # Django
2  *.log
3  *.pot
4  *.pyc
5  __pycache__/
6  local_settings.py
7  db.sqlite3
8  media
9
10 # Dependency directories
11 node_modules/
12
13 # Wordpress
14 wordpress/
15
16 virtualenvs/
17
18 # Images and raw data
19 #static/data/devices
20 #static/data/commercial_devices
21
22 **/.DS_Store
23
24 # Compiled documentation
25 doc/build/
26
```

Warning

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

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

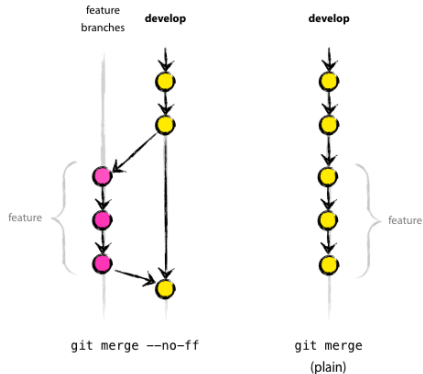
Task: add your name to the README on a dedicated branch

- 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 Uncommitted 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 once they have been merged



Task: let's do some merging!

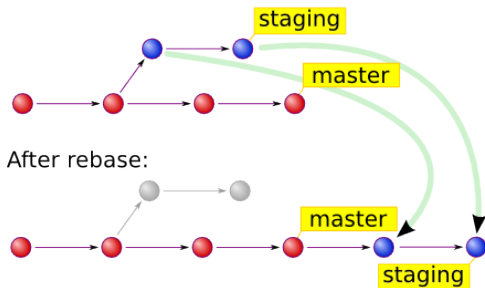
Demo merging

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

- 1 Checkout the base branch 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, they will appear in the lower part with a warning sign.
- 5 To resolve conflicts, right-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.



≈ 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

By default, SourceTree does not allow you to force push. The option must be first enabled in Settings > Advanced.

Warning

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

```
$ 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.

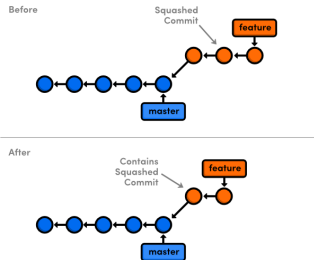
- 1 Checkout the feature branch you want to rebase.
- 2 In the left panel, right-click on the base branch you want to rebase onto (develop) and select `Rebase current changes onto ...`
- 3 If there are conflicts, resolve them as you did when merging.
- 4 At any point, you can stop the rebasing by selecting `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.

Outline

- 1 Standards for Software Development
- 2 What is git?
- 3 Tutorial
- 4 Advanced topics
- 5 Conclusion

Interactive Rebasing

Squashes the commits and rewrites history



- Takes the union of changes (opposite action to branching)
- Conflict resolution is part of the operation, so don't worry!
- Only merge when the branch is ready (green)
- Create new commit to keep topology (no fast-forward)
- Delete branches after some time once they have been merged

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!

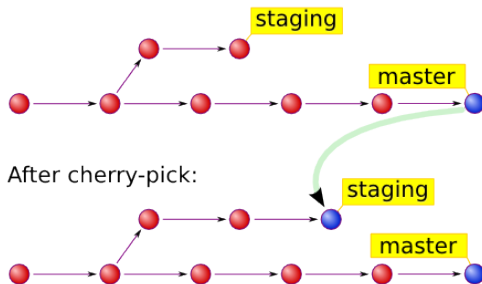
Demo interactive rebasing

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 introduced 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.

Outline

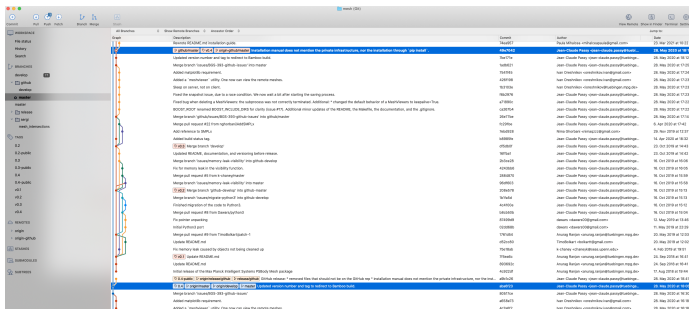
- 1 Standards for Software Development
- 2 What is git?
- 3 Tutorial
- 4 Advanced topics
- 5 Conclusion

Multiple origins

You can work privately on your git repository (just do not push!). As soon as you push, you work will be visible to anyone having access to repo (no branch restriction).

However, you may sometimes want to restrict some part of your repository (e.g. personal data, dev vs. public repo). The solution is to use **multiple origins**:

- You can compare commits between origins
- Many commits can have the same content



Last few tips

- Empty folders are not tracked. Good practice is to use a `.keep` file.
- User `git rm` to remove a file from the repo (a simple `rm` is usually not enough).
- Push as often as possible on feature branches
- Release often using a consistent versioning scheme, for instance [PEP 440](#):
1.0-dev1, 1.0-a1, 1.0-rc2, 1.0, 1.0.1,...
- Use tags and protect commits if needed
- These concepts are sufficient for $\approx 90\%$ of what you will need to do.

Copyright and Licensing

Disclaimer: I am not a legal professional!

Some definitions:

- Copyright: clarifies who owns the code
- License: clarifies who can use the code and how
- Authors: clarifies who wrote the code

It is a good idea to make these clear in the README.md (see [this example](#)).

If your code is public, it should have a copyright and a license. You can choose a permissive (BSD, MIT, Apache) or protective/copyleft license (e.g. GPL).

If a code does not have a license, the owner retains all rights and nobody else can use it.

Copyright and Licensing: GitHub specificities

Disclaimer: I am still not a legal professional!

GitHub has some specific rules (see [here](#)).

By publishing your code, you allow other users to view your work and fork your repository. Changing the visibility of your project does **NOT** delete forks.

There is also the concept of Contributor License Agreement (**CLA**). On GitHub, as long as the repository has a license, any contribution is licensed under the same terms (no need for a CLA).

However, the PR issuer remains the copyright owner of the contribution, which means:

- the project license cannot be changed without consent (e.g. Linux kernel)
- the contributor can re-used the contributed code