Version Control with Git

Eileen Kühn, David Kunz, Sarah Müller, Robin Roth
Plead guilty!

It’s easy to copy digital content, so why not re-create it over and over again?
Plead guilty!

It’s easy to copy digital content, so why not re-create it over and over again?

```
* 1. Mar 16:42 Kopie (4) von x-KIT_g/
* 17. Jun 13:35 Kopie (5) von x-KIT_g/
  8. Feb 12:35 Kopie (5) von x-KIT_g_OK_app/
* 17. Jul 10:26 Kopie (6) von x-KIT_g/
* 18. Sep 2012 Kopie von x-KIT_f/
* 22. Jan 2013 Kopie von x-KIT_g/
* 21. Jan 2013 Versionen.txt
* 17. Jul 11:06 current_version/
* 22. Jan 2013 etc/
* 14. Sep 2012 old/
* 21. Jan 2013 tmp/
* 29. Jun 2011 x-KIT_c_4/
* 17. Jan 2012 x-KIT_e/
* 14. Sep 2012 x-KIT_f/
```

“One of these folders *must* contain the latest version . . .”
Plead guilty!

It’s easy to copy digital content, so why not re-create it over and over again?

"Here is the latest version of the proposal/paper/report." — “Thanks.”

“One of these folders must contain the latest version . . .”
Obvious disadvantages

- No meta data about *what* was changed *when* by *whom*
- You lose track of what’s going on
- You cannot easily roll-back to a working state
- Poor solution for collaboration
Version control

- Track files
- Record (commit) changes
- Share changes with others
- Roll-back to an earlier state
- Implicit backup
Why Git?

- De-facto standard for open source software
- Probably the fastest version control system out there
- GitHub: web based collaboration platform
- Works well both with central and distributed repositories
- Easy to learn
Git Basics
Configuration

- Tell git who you are
  
  ```
  $ git config --global user.name <name>
  $ git config --global user.email <email>
  ```

- Configure auto correct for git commands
  
  ```
  $ git config --global help.autocorrect 1
  ```

- Use colors to show git information
  
  ```
  $ git config --global color.ui auto
  ```
Single User Workflow

1. Create a repository and a branch “master”
   
   $ git init

2. Create a commit
   2.1 Add something to the commit
   
   $ git add README.txt

   2.2 Perform the commit
   
   $ git commit -m "Added a README file"
Commits

Everytime you make a change, you create a commit containing:

- added/removed lines in files
- a comment summarizing what was changed
- an author
- a date
- a checksum (SHA-hash) to identify the commit
- a reference to the previous state of your files (parent(s))
Single User Workflow

1. Change something, and inspect the difference to the last commit
   
   $ vi README.txt
   $ git diff

2. Create a commit (as before)
   2.1 Add some changes to the commit
      
      $ git add README.txt
   2.2 Perform the commit
      
      $ git commit -m "Added project description"
Single User Workflow

1. Change something, and inspect the difference to the last commit
   $ vi README.txt
   $ git diff

2. Create a commit (as before)
   2.1 Add some changes to the commit
   $ git add README.txt
   2.2 Perform the commit
   $ git commit -m "Added project description"

Local changes checked in to index but not committed
master Added a README file
Single User Workflow

1. Change something, and inspect the difference to the last commit

   $ vi README.txt
   $ git diff

2. Create a commit (as before)
   2.1 Add some changes to the commit

      $ git add README.txt

   2.2 Perform the commit

      $ git commit -m "Added project description"
How to commit

- Small logical units
- Several times an hour
- Check the status before committing
- Write descriptive commit messages and keep 50/72 limits
  ⇒ Allows you to retrace your steps
Branching

- Keep master branch free from “questionable” code
  - Working on independent features at the same time
  - Trying incompatible changes
  - Quick and dirty work without changing the master branch
- Cheap, instant and easy
- Create and destroy often
- Integral part of a typical Git workflow
Branching

Create two branches from master

$ git checkout master
$ git checkout -b featureA
$ ...change & commit something
$ git checkout master
$ git checkout -b featureB
$ ...change & commit something
Branching

- Create two branches from master

```
$ git checkout master
$ git checkout -b featureA
$ ...change & commit something
$ git checkout master
$ git checkout -b featureB
$ ...change & commit something
```

- FeatureA: Place project under CC BY
- Added license
- Master: Added project description
- Added a README file

*Sarah Mueller*  |  2014-07-06 20:54:08
Sarah Mueller   |  2014-07-06 20:50:21
Sarah Mueller   |  2014-07-06 20:46:14
Sarah Mueller   |  2014-07-06 20:31:44
Branching

- Create two branches from master

```
$ git checkout master
$ git checkout -b featureA
$ ...change & commit something
$ git checkout master
$ git checkout -b featureB
$ ...change & commit something
```
Branching

Create two branches from master

- $ git checkout master
- $ git checkout -b featureA
- $ ...change & commit something
- $ git checkout master
- $ git checkout -b featureB
- $ ...change & commit something
Create two branches from master

```bash
$ git checkout master
$ git checkout -b featureA
$ ... change & commit something
$ git checkout master
$ git checkout -b featureB
$ ... change & commit something
```
Branching

- Switch back to master branch
  
  ```
  $ git checkout master
  ```

- Merge your changes into master
  
  ```
  $ git merge featureA # fast forward
  $ git merge --no-ff featureA #
  $ git merge featureB # merge
  ```

- Delete merged branches
  
  ```
  $ git branch -d featureA featureB
  ```
Branching

- Switch back to master branch
  
  ```
  $ git checkout master
  ```

- Merge your changes into master
  
  ```
  $ git merge featureA # fast forward
  $ git merge --no-ff featureA #
  $ git merge featureB # merge
  ```

- Delete merged branches
  
  ```
  $ git branch -d featureA featureB
  ```
Branching

- Switch back to master branch
  
  $ git checkout master

- Merge your changes into master
  
  $ git merge featureA # fast forward
  $ git merge --no-ff featureA #
  $ git merge featureB # merge

- Delete merged branches
  
  $ git branch -d featureA featureB
Branching

- Switch back to master branch
  
  ```
  $ git checkout master
  ```

- Merge your changes into master
  
  ```
  $ git merge featureA # fast forward
  $ git merge --no-ff featureA #
  $ git merge featureB # merge
  ```

- Delete merged branches
  
  ```
  $ git branch -d featureA featureB
  ```

![Branching diagram](image-url)
Branching

- Switch back to master branch
  
  $ git checkout master

- Merge your changes into master
  
  $ git merge featureA # fast forward
  $ git merge --no-ff featureA #
  $ git merge featureB # merge

- Delete merged branches
  
  $ git branch -d featureA featureB
Retracing Your Steps

1. Check the log
   
   
   $ \text{git log} \# \text{copy the SHA-key}$

2. Show changes to current version
   
   
   $ \text{git diff <paste SHA key>}$

3. Check out old version
   
   
   $ \text{git checkout <paste SHA key>}$
Collaboration
Group Exercises

- Clone a repository, possible protocols: https, ssh, git, file, ...
  
  ```
  $ git clone https://github.com/ksetagit/groupproject.git
  ```

- Copies the complete history of all branches to your disk
- Stores the cloning source as the *remote* “origin”

  ```
  $ git remote show
  $ git remote show origin
  ```

- ... now work as described before
Incorporate Changes of Collaborators

1. Fetch what others have done

   $ git fetch

   Downloads all commits and labels (e.g. “origin/master”) from the server, but leaves local labels unchanged.

2. Decide what to do:
   - Fast-forward your branch if you did not make changes
   - Merge a remote branch into your branch
   - Rebase your branch on top of a remote branch
   - Cherry-pick a commit from a different branch
Merge Other Branch Into Yours

- Trivial merge: fast-forward
- Non-trivial: creates new commit which includes both changes
  
  ```
  $ git merge origin/master
  ```

- Almost always works, but may result in *conflicts* if same lines changed in both branch heads
- Note that you can also do
  
  ```
  $ git pull
  ```

  which is the same as a *fetch* and a consecutive *merge*
Distributing Your Changes

- Upload changes in your branch “featureA” to origin

  $ git push origin featureA

- Does not work if featureA is changed on origin, in this case fetch and merge first

- Does not work if you deleted commits which were on origin, in this case force the update (be careful!):

  $ git push -f origin featureA
Group Tasks

- https://github.com/ksetagit/groupproject.git
- Group tasks in Readme.md
If Something Goes Wrong

Things go wrong if changes conflict. You can then:

1. Fix the conflicts, then
   
   ```
   $ git add <changed files>
   $ git merge --continue
   ```

2. Stop the operation
   
   ```
   $ git merge --abort
   ```

3. Undo broken merges:
   
   ```
   $ git reflog
   $ git checkout HEAD@{1}
   ```
How it works

**Stash**
- contains changes of a dirty working directory
- `git stash` for stacking

**Working Directory**
- holds files
- can freely be edited
- `git init` turns any directory into new repository

**Index**
- contains files included in next commit
- `git add` puts files to index

**Local Repository**
- history
- most recent commit is `HEAD`
- `git commit` creates commit which is `HEAD`

**Remote Repository**
- contains shared history of all commits
- `git clone` copies it
- `git push` for sending
- `git pull` for receiving
Best Practice Workflow

feature branches

develop

bugfixes

master

Time

New features

Severe bug fixed

Incorporates bug fix

Tag 0.1

Tag 0.2

Tag 1.0
Best Practice

- Do commit early and often
- Do not panic (as long as you committed [or even added] your work)
- Do not change published history (reset/rebase can be evil)
- Do divide your work into different repositories
- Do useful commit messages
- Do keep up to date
Further reads

- `$ man git`
- Different aspects from beginners to pros: http://gitready.com
- Interactive git tutorial: https://try.github.io
- Get these slides from: http://github.com/ksetagit/kseta-dvcs-talk
Advanced Git Operations
Stashing Your Work

- Get rid of uncommitted changes temporarily
  
  $ git stash

- Resets your working copy to the last committed version $C$
- Creates a “stash commit” whose parent is $C$
- Puts the stash commit on a stack
- Top-most stash commit can be applied again using

  $ git stash pop
Most complex operation in git:

$ git rebase origin/master

- Detach a commit from its parent and attach it to another commit
- Pre-condition is that changes can be applied to new parent
- Pro: Does not result in a merge-commit
- Contra: May create cascades of conflicts during rebase
Cherry-Picking

- Take a commit from another branch and apply it to yours as well
  
  $ git cherry-pick <SHA>

- Pre-condition is that you did not change same lines
- Git keeps track of commits by SHA and can ignore double commits
Other Interesting Commands

Append some changes to the last commit (use only if not pushed):

```
$ git commit --amend
```

Select only some of the changes to a file for a commit:

```
$ git add --patch/-p
```

Graphical tool to select changes to include in a commit:

```
$ git gui
```

Rewrite the history: reorder commits, combine them, . . . :

```
$ git rebase -i
```