

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Git

Phillip Raffeck, Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
<https://www4.cs.fau.de>

Wintersemester 2020



1 Versionsverwaltung mit git



Typische Aufgaben eines Versionsverwaltungssystems sind:

- *Sichern* alter Zustände
- *Zusammenführung* paralleler Entwicklung
- *Transportmedium*

Idealerweise zusätzlich:

- *Unabhängige Entwicklung* ohne zentrale Infrastruktur



- wir werden in VEZS git verwenden
- 2005 von Linus Torvalds für den Linux-Kernel geschrieben
- Konsequenz der Erfahrungen mit *bitkeeper*
- Eigenschaften:
 - dezentrale, parallele Entwicklung
 - Koordinierung hunderter Entwickler
 - Visualisierung von Entwicklungszweigen



		Tobias Klaus	May 3 2016, 12:26
master	origin/master	no ascii art needs formatting	
Every repository should contain a README file		Peter Waegemann	Apr 22 2016, 18:26
fix two warnings caused by ignored return values of system		Florian Franzmann	Apr 8 2014, 14:01
change colors to black and white theme		Florian Franzmann	May 27 2013, 18:46
fix warnings		Florian Franzmann	May 27 2013, 18:46
switch to cmake		Florian Franzmann	May 27 2013, 18:45
Anpassung fuer neuen gcc		Florian Franzmann	Jun 2 2008, 20:21
- fixed compiler warnings		Florian Franzmann	Sep 13 2009, 22:35
Sonnenauf/-untergang korrigiert		Florian Franzmann	Jul 18 2007, 22:45
gestrige Aenderung wieder rueckgaengig		Florian Franzmann	Jul 12 2007, 12:14
Monat fuer Sonnenaufgang/Untergang korrigiert		Florian Franzmann	Jul 12 2007, 00:23
Test fuer sunset		Florian Franzmann	Feb 22 2007, 23:05
logarithmische Skalierung fuer tics		siflfran	May 5 2006, 12:56
Outlineblamodi		siflfran	May 5 2006, 12:47
*** empty log message ***		siflfran	May 4 2006, 21:50
Kreiskrams fertig		siflfran	May 4 2006, 21:48
Kreisbla angefangen		siflfran	May 4 2006, 20:43
Neon-Krams		siflfran	May 2 2006, 17:22



The diagram illustrates a complex git repository history with numerous branches and merges. The commits are listed on the right, each with a date and a short log message. Colored lines represent different branches, and arrows indicate merge operations between them.

Merge tag 'as' Takashi Iwai	Mar 30 2017, 20:03
PNFS fix fallb Olga Kornievskaya	Mar 30 2017, 10:10
brcmfmac: usi Arend Van Spriel	Mar 28 2017, 09:11
rtlwifi: Fix sch Larry Finger	Mar 21 2017, 09:24
Merge tag 'iw' Kalle Valo	Mar 30 2017, 19:38
arm64: fi Mark Salter	Mar 24 2017, 09:53
drm/vc4: Al Eric Anholt	Mar 28 2017, 13:13
Merge branch 'i2' Wolfram Sang	Mar 30 2017, 16:53
block: do not put Minchan Kim	Mar 30 2017, 14:20
x86/build: Mostly c Josh Poimboeuf	Mar 16 2017, 14:31
drm/vmwgfx: fix int Li Qiang	Mar 27 2017, 20:10
drm/vmwgfx: Rem Thomas Hellstrom	Mar 27 2017, 13:06
drm/ttm: Avoid call Thomas Hellstrom	Mar 27 2017, 12:38
drm/ttm, drm/vmwgf Thomas Hellstrom	Mar 27 2017, 11:21
drm/vmwgfx: avoid calli Murray McAllister	Mar 27 2017, 11:15
drm/vmwgfx: NULL poi Murray McAllister	Mar 27 2017, 11:12
drm/vmwgfx: Type-che Thomas Hellstrom	Mar 27 2017, 11:09



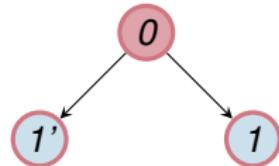
- Was speichert ein Commit?



- Was speichert ein Commit?
 - Wer? \leadsto Autor
 - Warum? \leadsto Commit-Nachricht
 - Was?
 - Vorher/Nachher *Zustände* Arbeitskopie

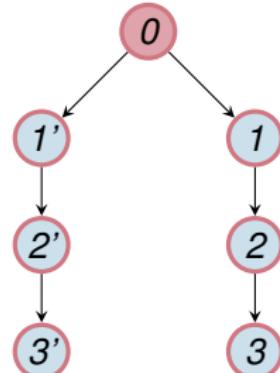


- Was speichert ein Commit?
 - Wer? \leadsto Autor
 - Warum? \leadsto Commit-Nachricht
 - Was?
 - Vorher/Nachher *Zustände* Arbeitskopie
 - Vorgänger Commits, auch *mehrere!*
 - *Keine* Nachfolger
- \leadsto Commit-Id: SHA-1 Hash über Inhalt



git-Commits

- Was speichert ein Commit?
 - Wer? \leadsto Autor
 - Warum? \leadsto Commit-Nachricht
 - Was?
 - Vorher/Nachher *Zustände* Arbeitskopie
 - Vorgänger Commits, auch *mehrere!*
 - *Keine* Nachfolger
- \leadsto Commit-Id: SHA-1 Hash über Inhalt
- \leadsto Gerichteter Azyklischer Graph
(engl.: Directed Acyclic Graph: DAG)
 - \leadsto Sprünge zurück *möglich*
 - \leadsto Sprünge vorwärts *nicht möglich*



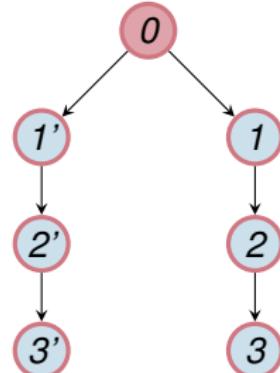
- Was speichert ein Commit?
 - Wer? \leadsto Autor
 - Warum? \leadsto Commit-Nachricht
 - Was?
 - Vorher/Nachher *Zustände* Arbeitskopie
 - Vorgänger Commits, auch *mehrere!*
 - *Keine* Nachfolger

\leadsto Commit-Id: SHA-1 Hash über Inhalt

\leadsto Gerichteter Azyklischer Graph
(engl.: Directed Acyclic Graph: DAG)
 - \leadsto Sprünge zurück *möglich*
 - \leadsto Sprünge vorwärts *nicht möglich*

 - Woher kriegt man "obere" Commits?

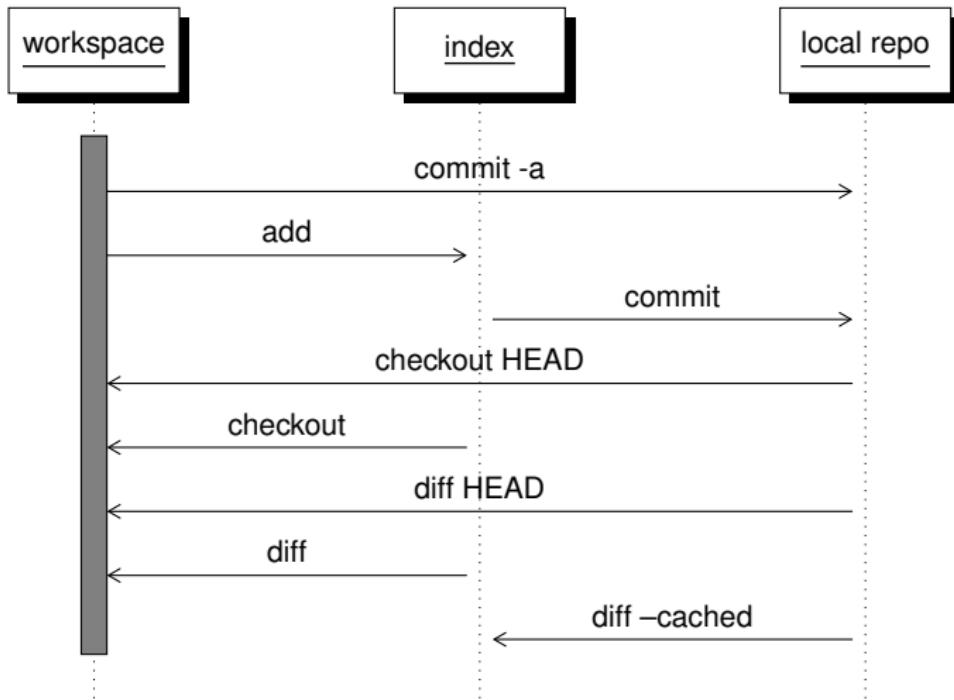
\leadsto Symbolische Namen (Zeiger)
 - HEAD: Aktueller Commit
 - Branch: Zeiger auf Commit



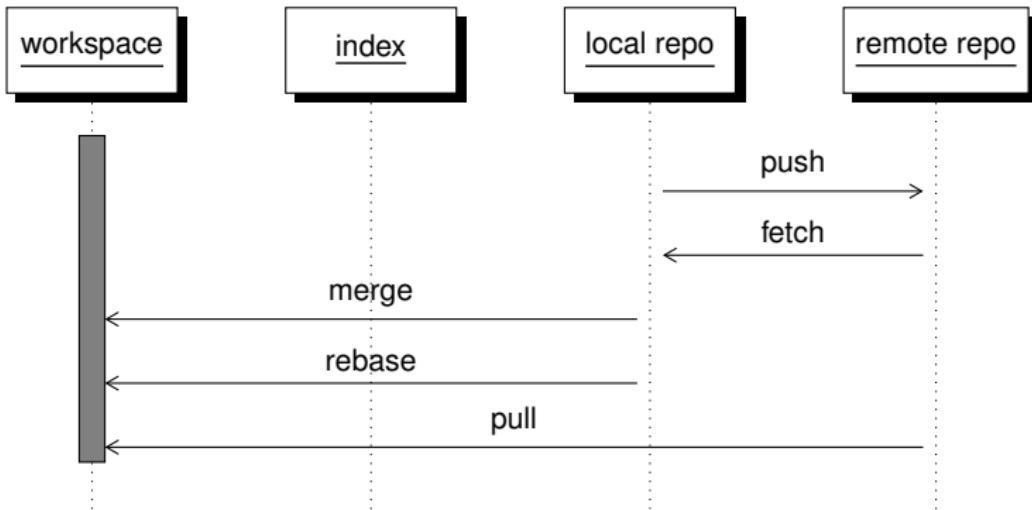
- initiales Repository herunterladen:
% git clone <URL>
- oder anlegen:
% git init
- Commit im Index zusammenbauen (⇒ „*Verladerampe*“):
% git add <Datei1>
% git add <Datei2>
% ...
- anschauen was bei git commit passieren würde:
% git status
oder
% git diff --cached
- anschließend Index an das Repository übergeben:
% git commit (⇒ „*Einladen in den LKW*“)



git-Arbeitsschritte – lokal



git-Arbeitsschritte – entfernt I



git push [<remote> [<branch>]]

- schiebt Commits nach <remote> in den ausgewählten <branch>
- dies geht nur, wenn lokales Repo auf dem aktuellen Stand ist!
- sonst beschwert sich git:

```
% git push origin master
```

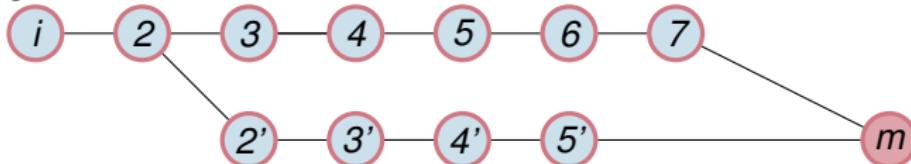
```
To /tmp/test.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to '/tmp/test.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

~ wir müssen das Repository erst auf den aktuellen Stand bringen



git pull [<remote> [<branch>]]

- holt Änderungen aus remote in den aktuellen Branch
- verschmilzt aktuellen Branch mit geholten Änderungen
- gleicher Effekt wie % git fetch && git merge FETCH_HEAD



```
% git pull origin
```

```
remote: Counting objects: 5, done.  
remote: Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From /tmp/test  
    38b95cb..8ec6e93  master      -> origin/master  
Auto-merging test.txt  
CONFLICT (content): Merge conflict in test.txt  
Automatic merge failed; fix conflicts and then commit the result.
```

- Änderungen an gleicher Stelle in der Zwischenzeit
~~~ Konflikte müssen von Hand behoben werden



```
% cat test.txt
hallo
<<<<< HEAD
welt!      meine Version
=====
Welt!      Version in origin/master
>>>>> 8ec6e9309fa37677e2e7ffcf9553a6bebf8827d6
```

- Konflikt auflösen:
  - Manuelles Bearbeiten
  - Übernahme des eigenen, Verwerfen des fremden Standes:  
git checkout --ours `test.txt`
  - Übernahme des fremden, Verwerfen des eigenen Standes:  
git checkout --theirs `test.txt`



- Konfliktlösung an git signalisieren:

```
% git add test.txt && git commit  
[master 4d21871] Merge branch 'master' of /tmp/test
```

```
% git push origin master  
Counting objects: 5, done.  
Writing objects: 100% (3/3), 265 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
To /tmp/test.git  
 8ec6e93..278c740  master -> master
```



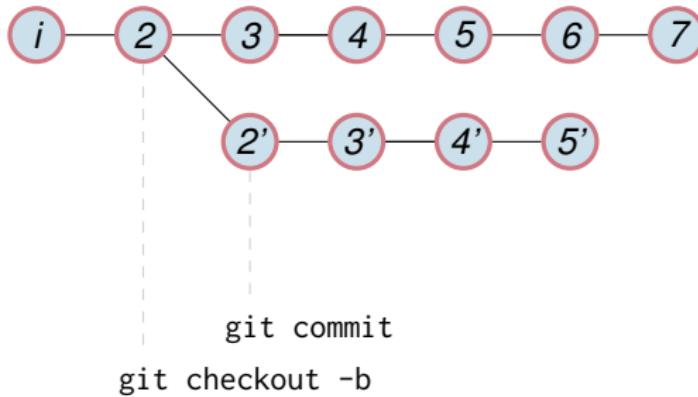
# Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung:



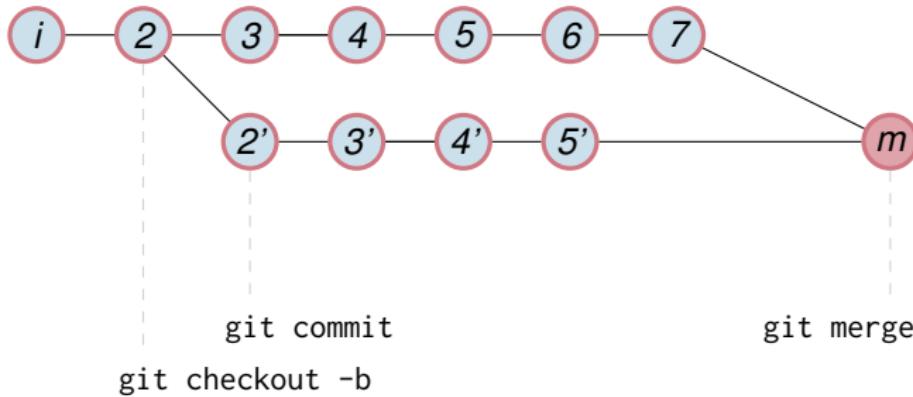
# Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung:



# Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung:



## In den meisten Versionsverwaltungssystemen

1. Featurebranch anlegen
2. Feature im Branch implementieren, testen
3. Featurebranch mit master verschmelzen
4. ggf. Featurebranch löschen



## In den meisten Versionsverwaltungssystemen

1. Featurebranch anlegen
2. Feature im Branch implementieren, testen
3. Featurebranch mit master verschmelzen
4. ggf. Featurebranch löschen

## Naiver Ansatz

~ skaliert nicht!



## Aufgaben von Versionsverwaltung

1. Codeschreiben unterstützen
2. Konfigurationsmanagement/Branches  
~ z. B. Release-Version, HEAD-Version ...



## Aufgaben von Versionsverwaltung

1. Codeschreiben unterstützen
2. Konfigurationsmanagement/Branches  
~ z. B. Release-Version, HEAD-Version ...

### ~ Konflikt

1. braucht Checkpoint-Commits
  - möglichst oft einchecken

~ skaliert nicht
2. braucht Stable-Commits
  - nur einchecken, wenn Commit perfekt

~ nicht praktikabel



## Öffentlicher Branch ~ verbindliche Geschichte

Commits sollen  $\left\{ \begin{array}{l} \text{atomar} \\ \text{gut dokumentiert} \\ \text{linear} \\ \text{unveränderlich} \end{array} \right\}$  sein

## Privater Branch ~ Schmierpapier

- für einzelnen Entwickler
- möglichst lokal
- wenn im zentralen Repo ~ auf Privatheit einigen

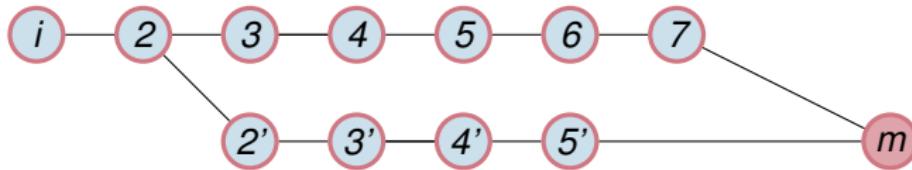


# Aufräumen

- verschmelze nie direkt privaten mit öffentlichem Branch

- Historie wird sonst unübersichtlich

→ nicht einfach git merge im master machen



- vorher immer erst git

- rebase → Commits auf Branch anwenden

- merge --squash → einzelnen Commit aus Branch-Commits  
oder: rebase --interactive → Commits umgruppieren

- commit --amend → letzten Commit überarbeiten

- Ziel: öffentlicher Commit ≡ Kapitel eines Buches

Michael Crichton

*Great books aren't written – they're rewritten.*



# Arbeitsablauf für kleinere Änderungen

- git merge --squash
- ~~ zieht Änderungen aus einem Branch in den aktuellen Index

## Branch

```
% git checkout -b private_feature_branch (Branch anlegen)
% touch file1.txt file2.txt
% git add file1.txt; git commit -am "WIP1" (file1.txt einchecken)
% git add file2.txt; git commit -am "WIP2" (file2.txt einchecken)
```

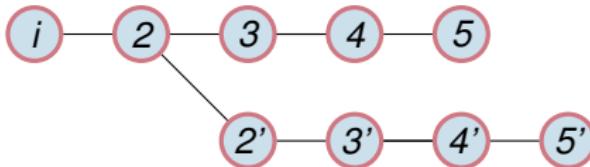
## Merge

```
% git checkout master (nach master wechseln)
% git merge --squash private_feature_branch
(Änderungen auf Index von master anwenden)
% git commit -v (Änderungen einchecken)
```



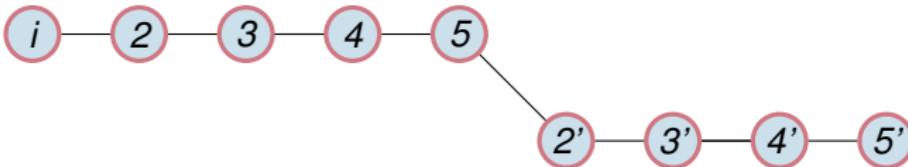
## git rebase <branch>

- Aufsetzen auf bestehenden <branch>



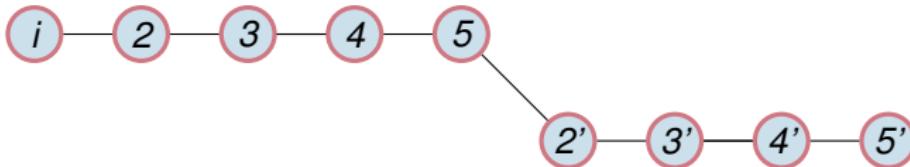
## git rebase <branch>

- Aufsetzen auf bestehenden <branch>



## git rebase <branch>

- Aufsetzen auf bestehenden <branch>



- Patches aus dem „unteren“ Zweig werden auf den „oberen“ aufgespielt
- Die Historie ist nun linear
- Linearisierte Änderungen lassen sich häufig einfacher bewerten
- **Vorsicht!**
  - Verzweigungen vom alten Zweig können nicht mehr zusammengeführt werden
  - Keine gemeinsamen Vorgänger mehr
  - Visualisierung der Historie ist nun bestenfalls verwirrend



## git rebase -interactive <commit>

- schreibt Geschichte um

```
git rebase -interactive ccd6e62^
```

**pick** ~ übernimmt Commit

```
pick ccd6e62 Work on back button
```

```
pick 1c83feb Bug fixes
```

```
pick f9d0c33 Start work on toolbar
```

**fixup** ~ verschmilzt Commit mit Vorgänger

```
pick ccd6e62 Work on back button
```

```
fixup 1c83feb Bug fixes # mit Vorgaenger verschmelzen
```

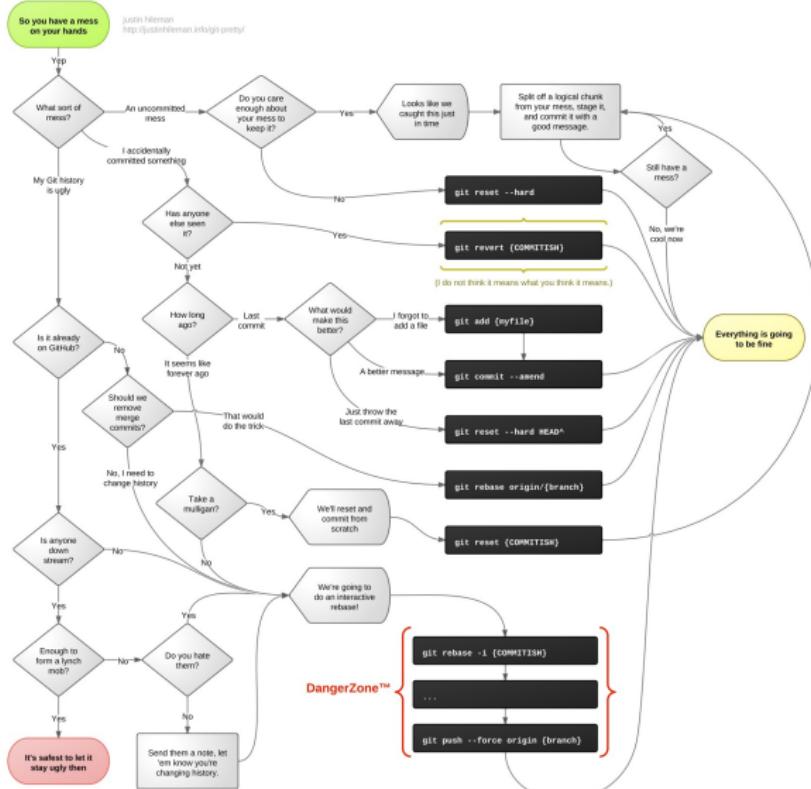
```
pick f9d0c33 Start work on toolbar
```

**reword** ~ Beschreibung editieren

**edit** ~ kompletten Commit editieren



## Geschichte neuschreiben



git push -force



git push -force



~ Wenn wirklich erforderlich: git push --force-with-lease



# git stash [pop]

- Vorübergehendes Sichern von Änderungen
- git stash und git stash pop
- ~ sichert Änderungen an der Working Copy auf Stapel
- rebase braucht saubere Working Copy  
⇒ vorher git stash

## Im Feature-Branch

```
% git stash
```

```
 Saved working directory and index state WIP on master: 81c0895 cmake  
HEAD is now at 81c0895 cmake, git ...
```

```
% ...
```

```
% git stash pop
```



# Wenn der Feature-Branch im Chaos versinkt?

- ~> aufgeräumten Branch anlegen
- 1. auf Branch master wechseln  
% git checkout master
- 2. Branch aus master erzeugen  
% git checkout -b cleaned\_up\_branch
- 3. Branch-Änderungen in den Index und die Working Copy ziehen  
% git merge --squash private\_feature\_branch
- 4. Index zurücksetzen  
% git reset
- danach Commits neu zusammenbauen
- ~> Auf der Konsole: git add -p
- ~> Graphisch: git cola



## git reflog

- Historie des HEAD-Zeigers
- Historie aller Befehle, die HEAD verändern

## git reflog

```
8af0d010 HEAD@{0}: rebase -i (finish): returning to refs/heads/master  
8af0d010 HEAD@{1}: checkout: moving from master to 8af0d010ae2ab48246d5  
7f97fab HEAD@{2}: commit: Pentax K20D fw version 1.04.0.11 wb presets  
8c37332 HEAD@{3}: rebase -i (finish): returning to refs/heads/master  
8c37332 HEAD@{4}: checkout: moving from master to 8c373324ca196c337dd  
9d66ec9 HEAD@{5}: clone: from git://github.com/darktable-org/darkt...
```

- `git reset --hard HEAD@{2}` stellt alten Zustand wieder her



- Selbstverwaltet auf <https://gitlab.cs.fau.de/>
- Login über Single-Sign-On mit IDM-Kennung
- **Regeln auf der Hauptseite beachten!**
- Abgaberepository pro Gruppe:  
<https://gitlab.cs.fau.de/ezs/WS20/group<groupid>>
- Arbeitskopie als Fork erstellen

ezs > DemoSemester > group42 > [Details](#)



- Abgabe per Merge-Request gegen  
<https://gitlab.cs.fau.de/ezs/WS20/group<groupid>>

# SSH-Schlüssel konfigurieren

- SSH Schlüssel erzeugen:  
~ % ssh-keygen -t rsa -f ~/.ssh/gitlab
- SSH Schlüssel für Authentifizierung hinterlegen:  
~ Kopieren: % xclip -selection clipboard .ssh/gitlab.pub  
~ Einfügen im Gitlab unter: Benutzer → Einstellungen → SSH-Keys
- <https://gitlab.cs.fau.de/help/ssh/README>
- SSH-Config anpassen, damit der neue Schlüssel auch verwendet wird:

\$HOME/.ssh/config

```
Host gitlab.cs.fau.de
  IdentityFile ~/.ssh/gitlab
```



# Dateien ignorieren mit git

- Dateien erscheinen nicht in der Ausgabe von bspw. `git status`
- Änderungen an Dateien werden ignoriert

## .gitignore

```
# Ignore LaTeX temporary files
*.aux
*.log

# except this one
!important.log

# everything under directory
solutions/
```

- Auch global möglich: `$HOME/.gitconfig`



## Per Befehlszeile

```
% git remote set-url origin git@gitlab.cs.fau.de:<user>/<projekt>.git  
% git remote add vorgabe git@gitlab.cs.fau.de:ezs/vezs20-vorgabe.git
```

## .git/config

```
[remote "origin"]  
  fetch = +refs/heads/*:refs/remotes/origin/*  
  url = git@gitlab.cs.fau.de:<username>/<projektname>.git  
  
[remote "vorgabe"]  
  fetch = +refs/heads/*:refs/remotes/origin/*  
  url = git@gitlab.cs.fau.de:ezs/vezs20-vorgabe.git
```



## \$HOME/.gitconfig

```
[user]
    name = Max Mustermann
    email = max.mustermann@fau.de

[core]
    editor = <maxs-lieblingseditor>

[alias]
    co = checkout
    b = branch -a -v
    st = status
    unstage = reset HEAD --
    visual = !gitk
    lg = log --graph \
        --pretty=format:'%C(red)%h%Creset \
        -%C(yellow)%d%Creset %s %C(green)(%cr) \
        %C(bold blue)<%an>%Creset' \
        --abbrev-commit \
        --date=relative
```



- Repository erstellen:

```
% git init
```

- Änderung hinzufügen:

```
% git add <Datei>
```

- oder interaktiv:

```
% git add -i
```

- feingranulares hinzufügen:

```
% git add -p
```

- Änderungen einchecken:

```
% git commit -m "meaningful commit message"
```



## git-Kommandos: Überblick II

- alles was nicht im git ist löschen:  
`% git clean -d <Pfad>`  
nur anzeigen, was gelöscht werden würde:  
`% git clean -n -d <Pfad>`
- aktuelle lokale Änderungen anzeigen:  
`% git diff`
- anzeigen was beim nächsten Commit verändert wird:  
`% git diff --cached`
- oder als Kurzzusammenfassung:  
`% git status`
- geänderte aber noch nicht eingechckte Datei zurücksetzen:  
`% git checkout -- <Datei>`



- das Log anschauen:  
`% git log`  
mit Graph:  
`% git log --graph`
- herausfinden, was im letzten Commit verändert wurde:  
`% git show`
- einen Commit rückgängig machen:  
`% git revert <commit-id>`
- Änderungen sichern, aber noch nicht einchecken:  
`% git add ...`  
`% git stash`



- gesicherte Änderungen wieder hervorholen:  
% git stash apply
- Stashinhalt anzeigen:  
% git stash list
- Stash-Element löschen:  
% git stash drop <id>
- einen Branch anlegen:  
% git branch <Name>
- alle registrierten Branches anzeigen:  
% git branch -a
- zu einem Branch wechseln:  
% git checkout <Name>



- menügeführt das Repository befragen I:  
% `tig`
- grafisch das Repository befragen II:  
% `gitk`
- Aktuelle Änderungen visualisieren:  
% `meld .`



- <http://gitready.com>
- <http://book.git-scm.com/>
- <http://eagain.net/articles/git-for-computer-scientists/>
- <http://sandofsky.com/blog/git-workflow.html>
- <http://365git.tumblr.com/>
  
- Zum Ausprobieren:
  - <https://learngitbranching.js.org/>
  - <https://github.com/git-game/git-game>

