

Git

Paul Bienkowski (2bienkow) \LaTeX

KunterBuntesSeminar

2013-05-29

Gliederung

- 1 Einleitung
 - Wozu Git?
 - Warum gerade git?
- 2 Git Internals
 - Objekte
 - Remotes
- 3 Basics
 - Working copy
 - Branching
 - Merging

Wozu Git?

Lasst uns zusammen ein Programm schreiben!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

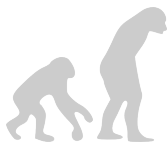
- 1 Wir arbeiten im Etherpad!



Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

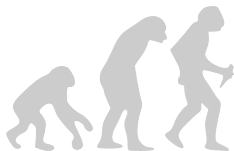
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!



Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

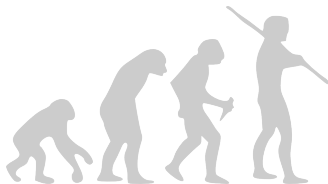
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!



Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

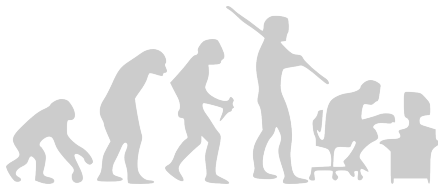
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!
- 4 Du schickst mir dann ein Diff per Mail!



Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!
- 4 Du schickst mir dann ein Diff per Mail!
- 5 Wir nehmen Versionskontrolle!

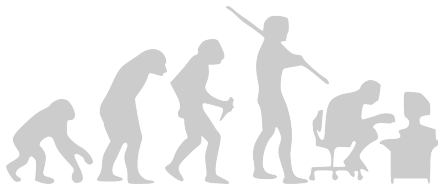


Wozu Git?

Lasst uns zusammen ein Programm schreiben! (*Evolution eines Hackers*)

- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!
- 4 Du schickst mir dann ein Diff per Mail!
- 5 Wir nehmen Versionskontrolle!

→ gilt auch für Hausaufgaben in \LaTeX



Was ist Versionierung?

- *Snapshots* einzelner Dateiversionen speichern
- History aufbewahren und wiederherstellbar machen
- verschiedene Versionen zusammenführen
- (*optional*) Synchronisation mit entfernten Kopien (Kollaboration)

→ mehr als nur ein Backup

Warum gerade git?

Vorteile:

- verteilt (serverunabhängig, jeder *clone* ist eigenständig/vollständig)
- schnell (lokal, Implementation in C)
- optimal für Quelltext
- FOSS

Nachteile:

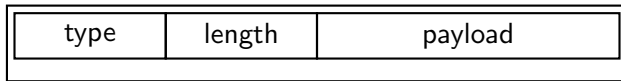
- wenig geeignet für Binärdateien
- gesamte History kann groß werden

Objekte

- Git als map-type storage
- Hashes bilden auf Daten ab
- Speicherung in einzelnen Dateien (“der Kernel macht das”)
- jedes Objekt in git ist eine solche Datei, jeweils ein “Snapshot” einer Version
 - *blob* für Dateien
 - *tree* für Verzeichnisse
 - Referenzen auf einzelne *blobs* + Metadaten
 - *commit* für das ganze Repository
 - Wurzelverzeichnis als *tree* + Metadaten
- *tree* und *commit* enthalten Informationen in Klartext, *blob* den Dateiinhalt

Objekte

- Git als map-type storage
- Hashes bilden auf Daten ab
- Speicherung in einzelnen Dateien (“der Kernel macht das”)
- jedes Objekt in git ist eine solche Datei, jeweils ein “Snapshot” einer Version
 - *blob* für Dateien
 - *tree* für Verzeichnisse
 - Referenzen auf einzelne *blobs* + Metadaten
 - *commit* für das ganze Repository
 - Wurzelverzeichnis als *tree* + Metadaten
- *tree* und *commit* enthalten Informationen in Klartext, *blob* den Dateiinhalt



zlib compress

Clones

- in einem Repository entstehen nur neue Objekte (Dateien)
- bestehende werden nicht verändert
- jedes Objekt hat einen eindeutigen Namen (SHA kollidiert *praktisch* nicht)
- → Synchronisation ohne zentrale Verwaltung möglich

Remotes

GRAPH!

Remotes

- Referenz auf entferntes Repository (clone)
- Erstellen mit

```
git remote add <name> <url>
```

- "Standard"-Remote *origin* wird beim *clonen* erstellt
- Operationen
 - fetch* Objekte herunterladen
 - pull* *fetch* + *merge*
 - push* Objekte hochladen + Referenzen aktualisieren (branches)

Pull / Push

Push

```
git push [options] <remote> <from-ref>:<to-ref>
```

```
git push -u origin master
```

```
git push -u origin :useless-branch
```

```
git push
```

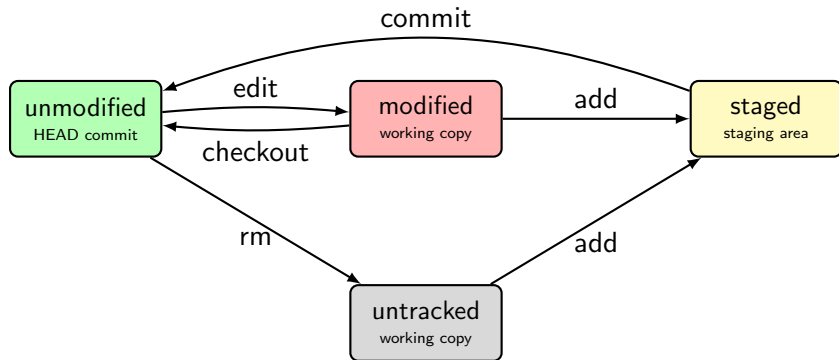
Pull

```
git pull <remote> <branch>
```

```
git pull origin master
```

```
git pull
```

Das Leben einer Datei



Working copy

- In `.git/` liegen alle Versionen aller Dateien als *blob* vor
- Wie soll man damit arbeiten?
- → aktuelle Version (*HEAD*) liegt im Hauptverzeichnis
- verständliche Dateinamen (statt Hashes)
- wechseln der Version per CHECKOUT

```
git checkout 4b5c8e2f95a4407c4d0c596565b367eaca07af57
```

→ nicht möglich, wenn unversionierte Änderungen vorliegen

Branching

- Commits liegen ungeordnet vor
- → Welche ist die “aktuelle” Version?
- Ein **Branch** kann auf einen Commit zeigen (“Pointer”)
- Branches haben **Namen** (z.B. *master* oder *my-feature*)
- Man kann zwischen Branches wechseln wie zwischen Commits (CHECKOUT)
- Nach dem committen zeigt der aktuelle Branch auf den neuen Commit

Branching

```
git status
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout -b foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "D"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout -b hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "E"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git merge hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git branch -d hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "F"
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "F"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git merge foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git branch -d foobar
```

– tikz picture here, finaltrue to show –

Merging

Was passiert?

- gemeinsamen Vorgänger finden
- Änderungen ermitteln
- beide Änderungssätze auf gemeinsamen Vorgänger anwenden
- neuen Commit erstellen (automatische *message*)

Konflikte

- bei Änderung gleicher Zeile kann git nicht entscheiden, welche Änderung übernommen werden soll
- Auto-merging <filename>
CONFLICT (content): Merge conflict in <filename>
Automatic merge failed; fix conflicts and then commit the result.

Merging - Konflikte

```
vor dem Konflikt  
<<<<<< HEAD  
meine Änderungen  
=====  
deine Änderungen  
>>>>>> fremder-branch  
nach dem Konflikt
```

Übersicht

Begriffe

commit	Versions-Snapshot
working directory	Arbeitskopie der aktuell gewählten Version
clone	Kopie eines Repositories
remote	Referenz im lokalen Repository auf (entfernten) clone
branch	Zeiger auf einen Zweig der History, wird aktualisiert

Kommandos

clone	ein Repository von einer URL kopieren (init + remote add + pull)
status	aktuellen Status des working directories anzeigen
add	Dateien/Verzeichnisse stagen
commit	Version in Kontrolle aufnehmen
pull	commits auf remote übertragen und remote-branch updaten
push	commits von remote empfangen und in aktuellen branch mergen
log	Vorgänger-Versionen auflisten
checkout	bestimmte Version für einzelne Dateien oder das WD auswählen
rm	Dateien löschen
branch	Branches verwalten (auswählen mit checkout)
diff	Versionen vergleichen
merge	zwei Änderungen zusammenführen