

Git

Paul Bienkowski (2bienkow) \LaTeX

KunterBuntesSeminar

2013-05-29

Gliederung

1 Einleitung

- Wozu Git?
- Warum gerade git?

2 Inside Git

- Objekte
- Working copy
- Vom Leben einer Datei

3 Lokale Operationen

- Status
- Staging & Committing
- Checkout
- Log
- Branching
- Merging

4 Entfernte Operationen

- Remotes

5 Anhang

- Tipps und Tricks
- Übersicht

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*)

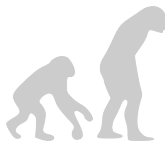
- 1 Wir arbeiten im Etherpad!



Wozu Git?

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

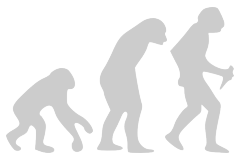
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!



Wozu Git?

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

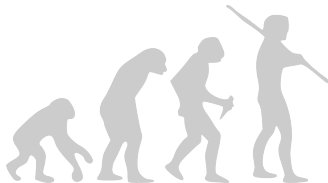
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!



Wozu Git?

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

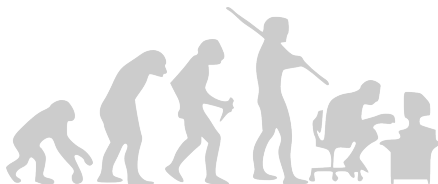
- 1 Wir arbeiten im Etherpad!
- 2 Wir arbeiten auf rzssh1!
- 3 Du schickst mir dann die neue Version per Mail!
- 4 Du schickst mir dann ein Diff per Mail!



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!

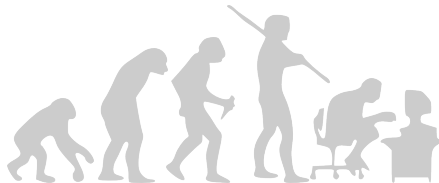


Wozu Git?

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:

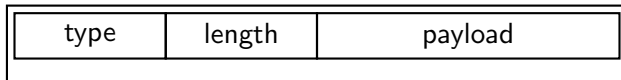
- wenig geeignet für Binärdateien
- gesamte History kann groß werden

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

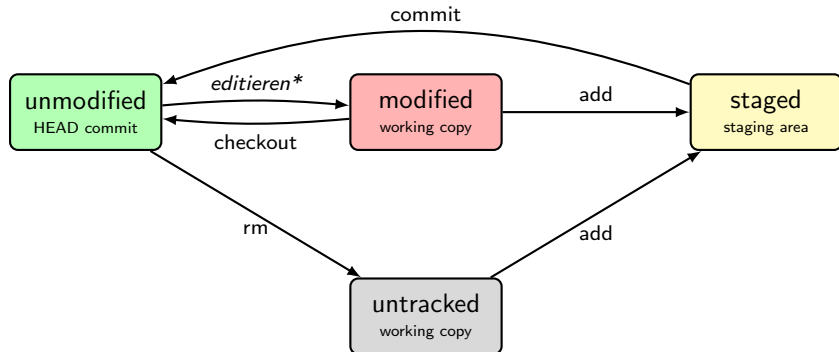
Working copy

- In `.git/` liegen alle Objekte, mit Hashes als Dateinamen
- Wie soll man damit arbeiten?
- → aktuelle Version (*HEAD*) liegt im Hauptverzeichnis
- verständliche Dateinamen (statt Hashes)

Der Index (Staging Area)

- “Laderampe”
- erst wählen, welche Änderungen übernommen werden sollen (*add*)
- dann diese Änderungen speichern (*commit*)

Vom Leben einer Datei



Status / Diff

Aktuellen Status des Working Directories:

```
git status
```

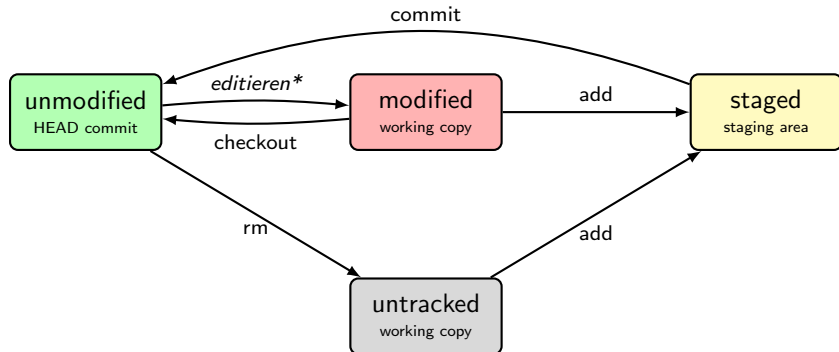
Änderungen zwischen Versionen:

```
git diff
```

```
git diff myfile.txt
```

```
git diff some-ref other-ref
```

Vom Leben einer Datei



Staging & Committing

Dateien zum Index (Stage) hinzufügen:

```
git add <files/directories>
```

```
git add -i
```

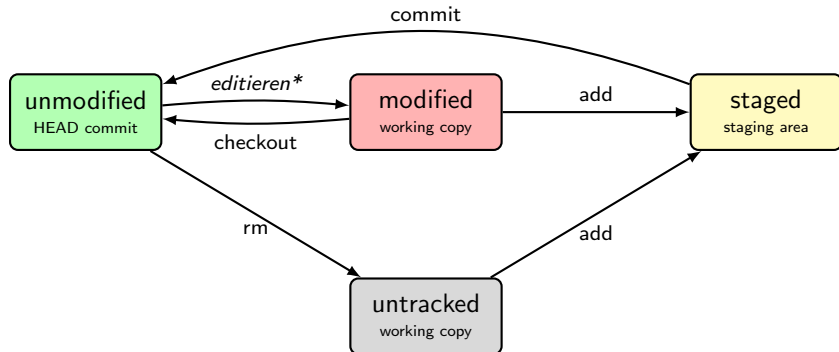
Commit-Objekt erzeugen:

```
git commit
```

```
git commit -m "Message"
```

```
git commit -a -m "Message"
```

Vom Leben einer Datei



Checkout

- Erinnerung: aktuelle Version liegt im Working Directory
- diese können wir mit `per CHECKOUT` wechseln

```
git checkout 4b5c8e2f95a4407c4d0c596565b367eaca07af57
```

```
git checkout other-branch
```

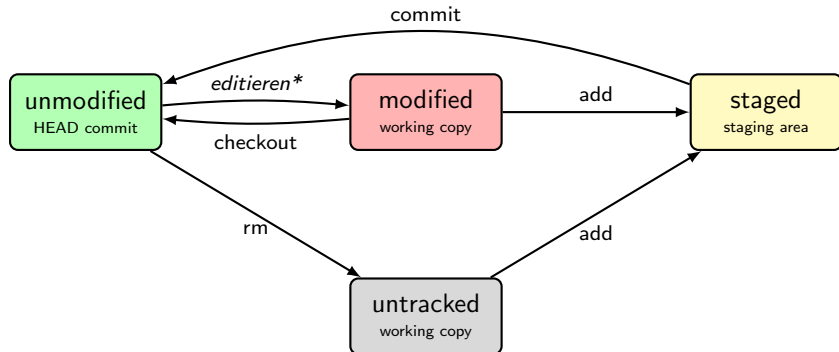
→ nicht möglich, wenn unversionierte Änderungen vorliegen

- einzelne Dateien können auch auf HEAD (oder speziellen commit) zurückgesetzt werden

```
git checkout edited-file.txt
```

```
git checkout aa5bcd5a some-file.txt
```

Vom Leben einer Datei



Log

- zeigt History an
- nur von HEAD rückwärts
- enthält Commit-Nachricht, Datum, Autor, ...
- Befehl:

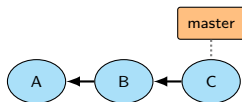
```
git log
```


Branching

- Commit-Objekte liegen ungeordnet vor
- → welche ist die neueste 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

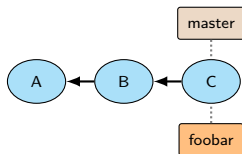
Branching

```
git status
```



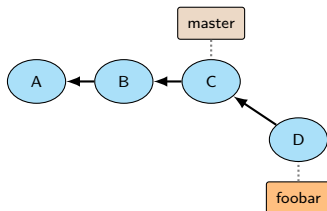
Branching

```
git checkout -b foobar
```



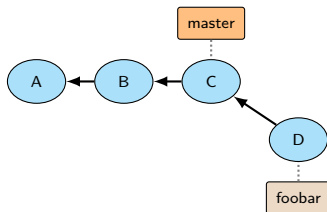
Branching

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



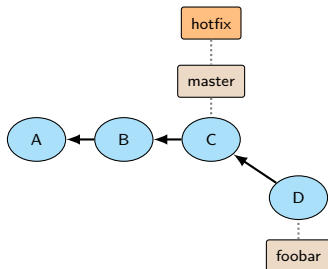
Branching

```
git checkout master
```



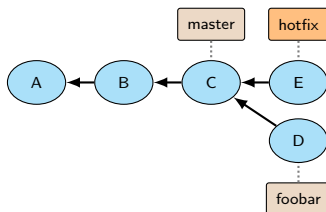
Branching

```
git checkout -b hotfix
```



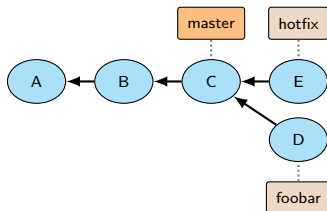
Branching

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



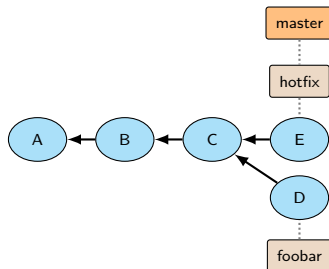
Branching

```
git checkout master
```



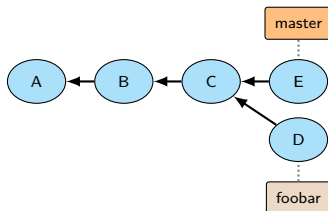
Branching

```
git merge hotfix
```



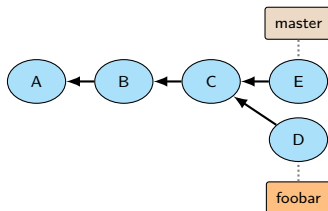
Branching

```
git branch -d hotfix
```



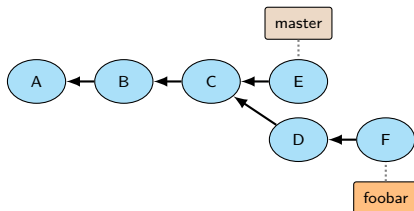
Branching

```
git checkout foobar
```



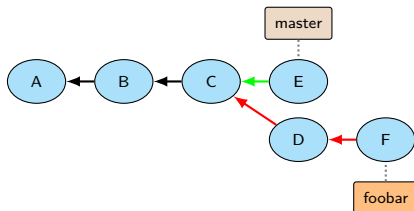
Branching

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



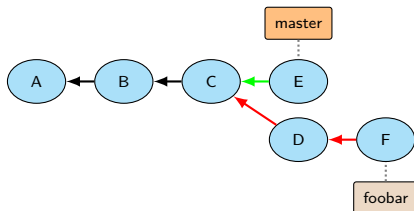
Branching

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



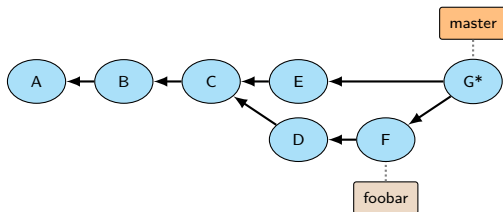
Branching

```
git checkout master
```



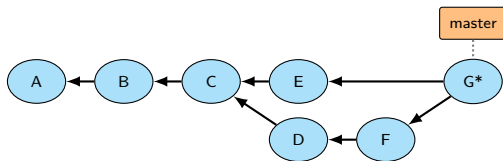
Branching

```
git merge foobar
```



Branching

```
git branch -d foobar
```



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.`

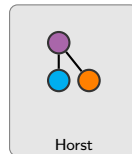
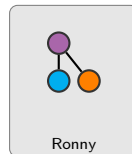
Merging - Konflikte

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

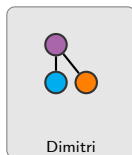
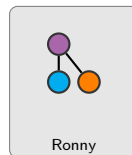
Clones

- 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

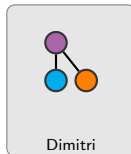
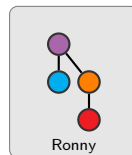
Remotes



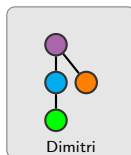
Remotes



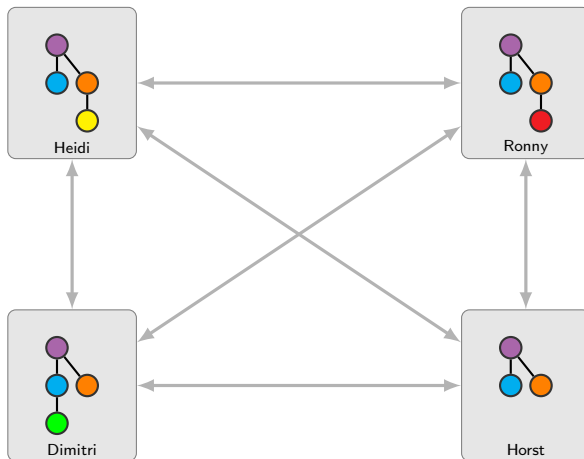
Remotes



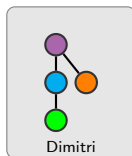
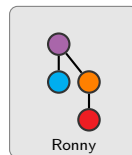
Remotes



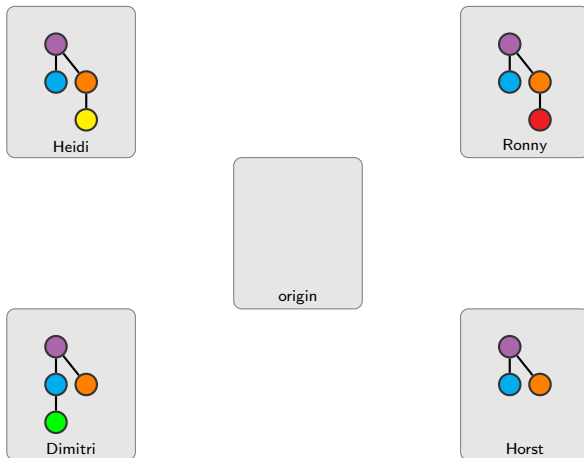
Remotes



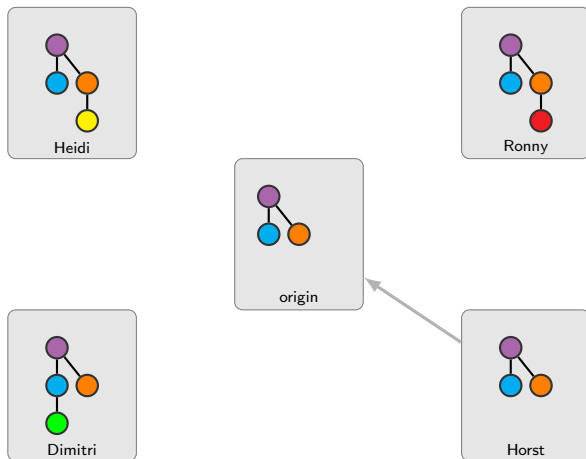
Remotes



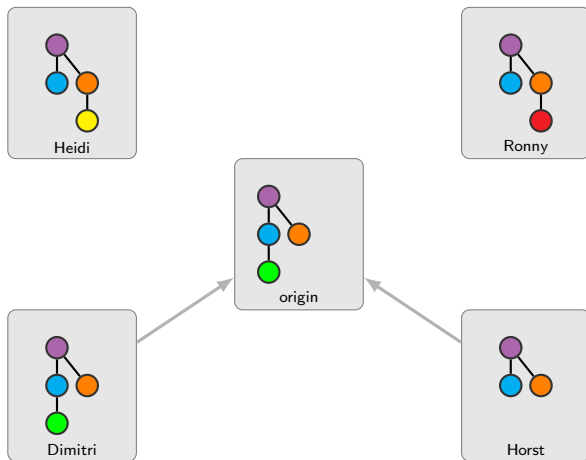
Remotes



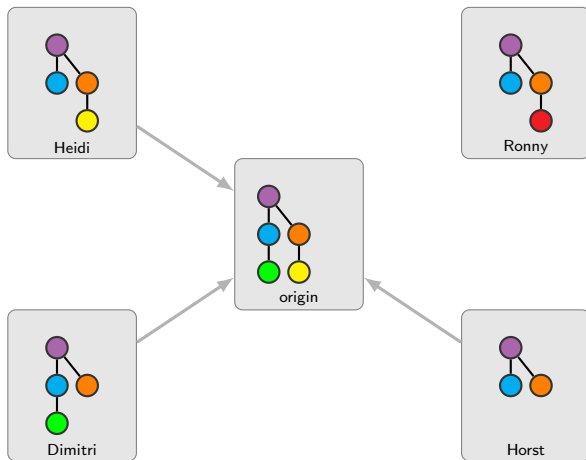
Remotes



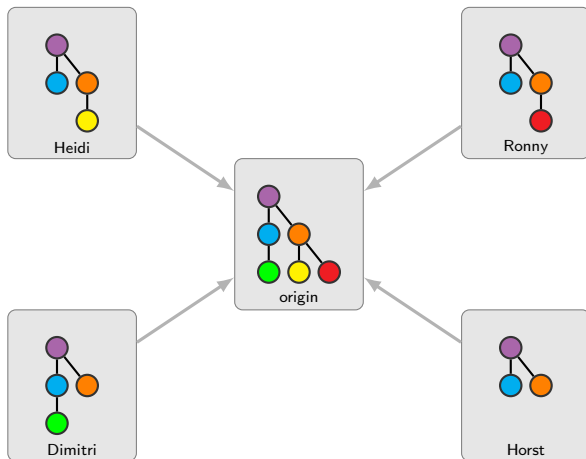
Remotes



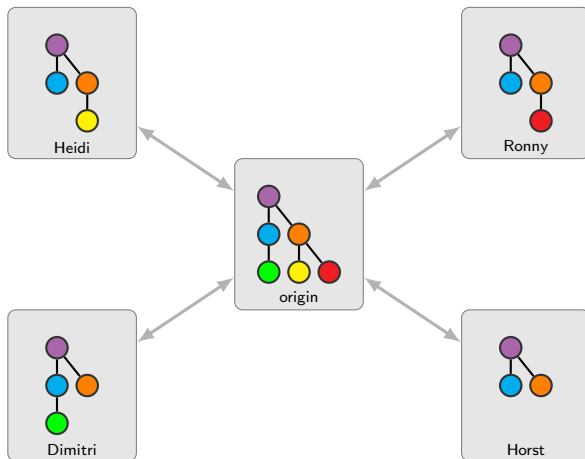
Remotes



Remotes



Remotes



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
```

Und wann mach ich das jetzt?

Und wann mach ich das jetzt?

- vor dem Arbeiten: **pull** (am aktuellen Stand arbeiten)
- Änderungen durchführen
- nach jeder Einheit (z.B. ein Bugfix, ein kleines Feature): **commit**
- möglichst häufig: **push** (minimiert Merge-Konflikte)

Und wann mach ich das jetzt?

- vor dem Arbeiten: **pull** (am aktuellen Stand arbeiten)
- Änderungen durchführen
- nach jeder Einheit (z.B. ein Bugfix, ein kleines Feature): **commit**
- möglichst häufig: **push** (minimiert Merge-Konflikte)

Commit often and early!

Tipps und Tricks

- Log mit Änderungen (diff) zeigen:

```
git log -p
```

- Bunter Log mit Baum-Struktur:

```
git log --graph --all --format=format:'%C(yellow)%h%C(reset) -  
%C(cyan)%ci%C(reset) %C(green)(%cr)%C(reset) %C(bold yellow)%d%C(reset)%n  
%C(white)%s%C(reset) %C(bold white)- %cn%C(reset)%n'  
--abbrev-commit --date=relative
```

- ~/.gitconfig anpassen!

~/ .gitconfig

[user]

name = Firstname Lastname

email = user@host.tld

[color]

diff = auto

status = auto

branch = auto

interactive = auto

ui = true

[push]

default = simple

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