

Git-Workshop, Teil I

Freitagsrunde TechTalks, TU Berlin

Julius Plenz

25. November 2011



Veröffentlicht unter der CreativeCommons-Lizenz (By, Nc, Sa)

<http://wiki.freitagsrunde.org/TechTalks>

Bevor wir beginnen ...

- ▶ Wer verwendet Linux? – Windows? – Mac?
- ▶ Wer arbeitet gelegentlich auf der Shell?
- ▶ Wer hat momentan noch *kein* Git installiert?
- ▶ Wer kennt oder hat schon mal eines der folgenden Systeme benutzt?
 - ▶ CVS/RCS
 - ▶ SVN
 - ▶ Mercurial, Darcs, Perforce, Bazaar

Wer kennt Git?

Wer hat schonmal ...

- ▶ `git` eingegeben
- ▶ Ein Git-Repository selbst erstellt?
- ▶ ... oder geklont?
- ▶ Einen Commit gemacht?
- ▶ Per Git mit anderen Leuten zusammengearbeitet?

Ablaufplan

- ▶ Teil I:
 - ▶ Grundlegende Arbeitsschritte in Git
 - ▶ Das Objektmodell – eine theoretische Grundlage
 - ▶ Parallele Entwicklung: Branches und Merges
 - ▶ Entwicklung koordinieren: Ein Branching-Modell
- ▶ Teil II:
 - ▶ Die Geschichte umschreiben: Rebase
 - ▶ Verteiltes Git: Commits hoch- und runterladen
 - ▶ Verschiedene Workflows
 - ▶ Kollaboration live: Wir entwickeln gemeinsam ein Projekt
 - ▶ Oder: Wunschkonzert – was euch interessiert
- ▶ Teil I findet komplett *lokal* auf dem eigenen Rechner statt – erst in Teil II nutzen wir die verteilten Eigenschaften von Git aus

Motivation: Warum Versionskontrolle?

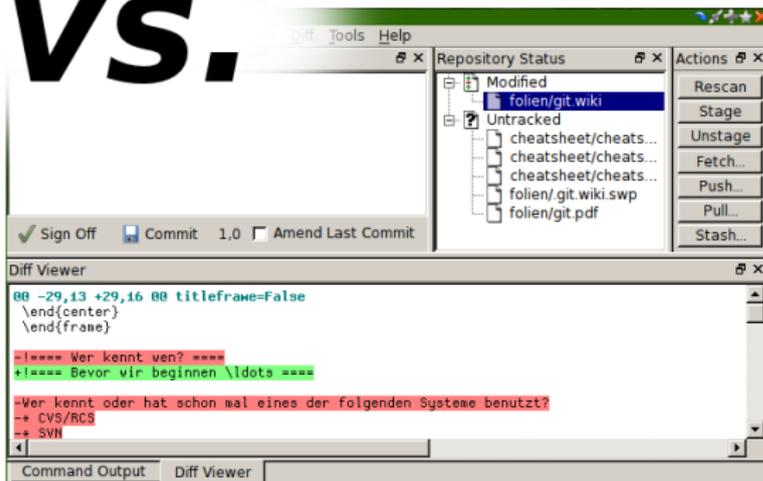
- ▶ **Sicherheit:** Versionskontrolle schützt vor Verlusten
- ▶ **Dokumentation:** Wer hat wann was gemacht?
- ▶ **Fokussierung:** Entwicklung logisch gliedern
- ▶ **Kollaboration:** Mit anderen Leuten an den gleichen Dateien arbeiten
- ▶ **Partizipation:** Jeder kann mitmachen (GitHub etc.)

Interface

```
00RVT
commit 556a014f913c5f638e4adebeccd1587b0f3c46ed
Author: Julius Plenz <julius@plenz.com>
Date: Tue Nov 22 19:19:07 2011 +0100

Index-Diagramme hinzugefügt
zsh> git status
# On branch master
# Your branch is ahead of 'origin/master' by 6 commits.
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   folien/git.wiki
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       cheatsheet/
#       folien/.git.wiki.swp
#       folien/git.pdf
no changes added to commit (use "git add" to track new files)
zsh> git add folien/git.wiki
```

VS.



Wer bin ich? – Name und E-Mail einstellen

- ▶ Für alle Projekte (wird in `~/.gitconfig` gespeichert)
 - ▶ `git config --global user.name "Max Mustermann"`
 - ▶ `git config --global user.email max@mustermann.de`
- ▶ ... oder alternativ nur für das aktuelle Projekt:
 - ▶ `git config user.email maintainer@cool-project.org`
- ▶ Außerdem, für die, die wollen: Farbe!
 - ▶ `git config --global color.ui auto`

Ein Projekt importieren oder erstellen

- ▶ Ein neues Projekt erstellt man wie folgt:
 - ▶ `mkdir projekt`
 - ▶ `cd projekt`
 - ▶ `git init`
- ▶ Um ein bestehendes Projekt zu importieren, »klont« man es mit seiner gesamten Versionsgeschichte:
 - ▶ `git clone git://git.plenz.com/git-tips`

Begriffsbildung

- ▶ **Index/Staging Area:** Bereich zwischen dem Arbeitsverzeichnis und dem Repository, in die Änderungen für den nächsten **Commit** gesammelt werden
- ▶ **Commit:** Eine Änderung an einer oder mehrerer Dateien, versehen mit Metadaten wie Autor, Datum und Beschreibung
- ▶ **Repository:** Eine Datenbank für Commits, dort wird die Versionsgeschichte aufgezeichnet
- ▶ **Referenz:** Jeder **Commit** wird durch eine eindeutige SHA1-Summe identifiziert. Eine Referenz »zeigt« auf einen bestimmten Commit
- ▶ **Branch:** Ein »Zweig«, eine Abzweigung im Entwicklungszyklus, z. B. um ein neues Feature einzuführen.

Ein typischer Arbeitsablauf

- ▶ Eine *datei* verändern, und die Änderungen in das Repository »einchecken«:

1. `vim datei`
2. `git status`
3. `git add datei`
4. `git commit -m 'datei angepasst'`
5. `git show`

Index / Staging Area

- ▶ Im *Index* bzw. der *Staging-Area* werden Veränderungen für den nächsten Commit vorgemerkt
- ▶ So kann der Inhalt von einem Commit schrittweise aus einzelnen Veränderungen zusammengestellt werden

Ausgangsstellung

- ▶ Alle auf dem gleichen Stand

Working-Tree

```
#!/usr/bin/python  
print "Hello World!"
```

Index

```
#!/usr/bin/python  
print "Hello World!"
```

Repository

```
#!/usr/bin/python  
print "Hello World!"
```

Veränderungen machen

- ▶ Veränderungen werden im Working-Tree gemacht

Working-Tree

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```

Index

```
#!/usr/bin/python
print "Hello World!"
```

Repository

```
#!/usr/bin/python
print "Hello World!"
```

Dem Index hinzufügen – git add

- ▶ Die Veränderungen im Working-Tree → Index

Working-Tree

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```

Index

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```

Repository

```
#!/usr/bin/python
print "Hello World!"
```



git add

Einen Commit erzeugen – git commit

- ▶ Alle Veränderungen im Index → Commit

Working-Tree

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```

Index

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```

Repository

```
#!/usr/bin/python
+# Autor: Valentin
+
print "Hello World!"
```



git commit

Resultat

- ▶ Alle wieder auf dem gleichen Stand

Working-Tree

```
#!/usr/bin/python  
# Autor: Valentin  
print "Hello World!"
```

Index

```
#!/usr/bin/python  
# Autor: Valentin  
print "Hello World!"
```

Repository

```
#!/usr/bin/python  
# Autor: Valentin  
print "Hello World!"
```

Referenzen und ignorierte Dateien

Relative Referenzen:

- ▶ HEAD: Der letzte Commit (wird per `git show` angezeigt)
- ▶ HEAD[^]: Der vorletzte Commit
- ▶ HEAD~*N*: Der *N*-letzte Commit

Informationen über das Repository erhalten

- ▶ Den jüngsten Commit im vollen Umfang anschauen:
 - ▶ `git show`
- ▶ Die gesamte Versionsgeschichte, die zum aktuellen Zustand führt, anzeigen:
 - ▶ `git log`
- ▶ Was hat sich verändert?
 - ▶ `git diff`
- ▶ Das Repository visualisieren:
 - ▶ `gitk`
- ▶ ... oder textbasiert:
 - ▶ `tig`

Änderungen rückgängig machen

Einen neuen Commit erstellen, der eine alte Änderung rückgängig macht:

- ▶ `git revert commit`

Den Index zurücksetzen:

- ▶ `git reset HEAD`

Den Zustand von vor zwei Commits wiederherstellen:

- ▶ `git checkout HEAD~2`

Die Version von *datei* anschauen, wie sie vor zwei Commits war:

- ▶ `git show HEAD~2:datei`

Die letzten zwei Commits **unwiederbringlich** löschen:

- ▶ `git reset --hard HEAD~2`

Branches: Abzweigungen

Wir arbeiten schon die ganze Zeit im master-Branch!

Was genau sind Branches? – Nichts anderes als Referenzen auf den jeweils obersten Commit einer Versionsgeschichte.

Branches ...

- ▶ erstellen: `git branch name`
- ▶ auschecken: `git checkout name`
- ▶ erstellen und direkt auschecken: `git checkout -b name`
- ▶ auflisten: `git branch -v`
- ▶ löschen: `git branch -d name`

Idealisierter Workflow: Ein Branch pro neuem Feature oder Bugfix.

Beispiel: Zwei Branches

Zwei Branches erstellen, und auf jedem einen Commit machen.
Dann das Resultat in gitk anschauen.

- ▶ `git branch eins`
- ▶ `git checkout eins`
- ▶ Commit machen
- ▶ `git checkout master`
- ▶ `git checkout -b zwei`
- ▶ Commit machen
- ▶ `gitk --all`

Beispielprojekt: Was wollen wir speichern

Angenommen, wir wollen folgendes Verzeichnis speichern:

```
/
├── hello.py
├── README
├── test/
│   └── test.sh
```

Objektmodell

- ▶ *Blob*: Enthält den Inhalt einer Datei
- ▶ *Tree*: Eine Sammlung von Tree- und Blob-Objekten
- ▶ *Commit*: Besteht aus einer Referenz auf einen Tree mit zusätzlichen Informationen
 - ▶ *Author* und *Committer*
 - ▶ *Parents*
 - ▶ *Commit-Message*

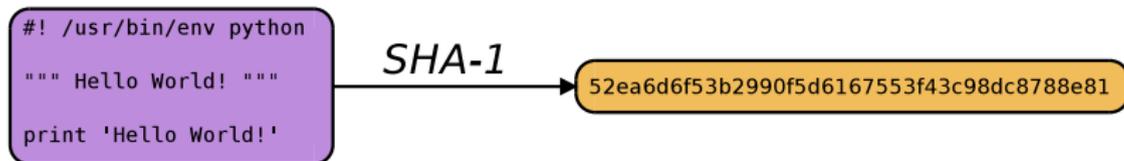
blob	67
52ea6d6...	
<pre>#!/usr/bin/env python """ Hello World! """ print 'Hello World!'</pre>	

tree	101
a26b00a...	
blob	6cf9be8, README
blob	52ea6d6, hello.py
tree	c37fd6f, test

commit	245
e2c67eb...	
tree	a26b00a...
parent	8e2f5f9...
committer	Valentin
author	Valentin
Kommentar fehlte	

SHA-1 IDs

- ▶ Objekte werden mit *SHA-1 IDs* identifiziert
- ▶ Dies ist der *Objekt-Name*
- ▶ Wird aus dem Inhalt berechnet
- ▶ *SHA-1* ist eine sogenannte Hash-Funktion; sie liefert für eine Bit-Sequenz mit der maximalen Länge von $2^{64} - 1$ Bit (≈ 2 Exbibyte) in eine Hexadezimal-Zahl der Länge 40 (d. h. 160 Bits)
- ▶ Die resultierende Zahl ist eine von 2^{160} ($\approx 1.5 \cdot 10^{49}$) möglichen Zahlen und ziemlich einzigartig

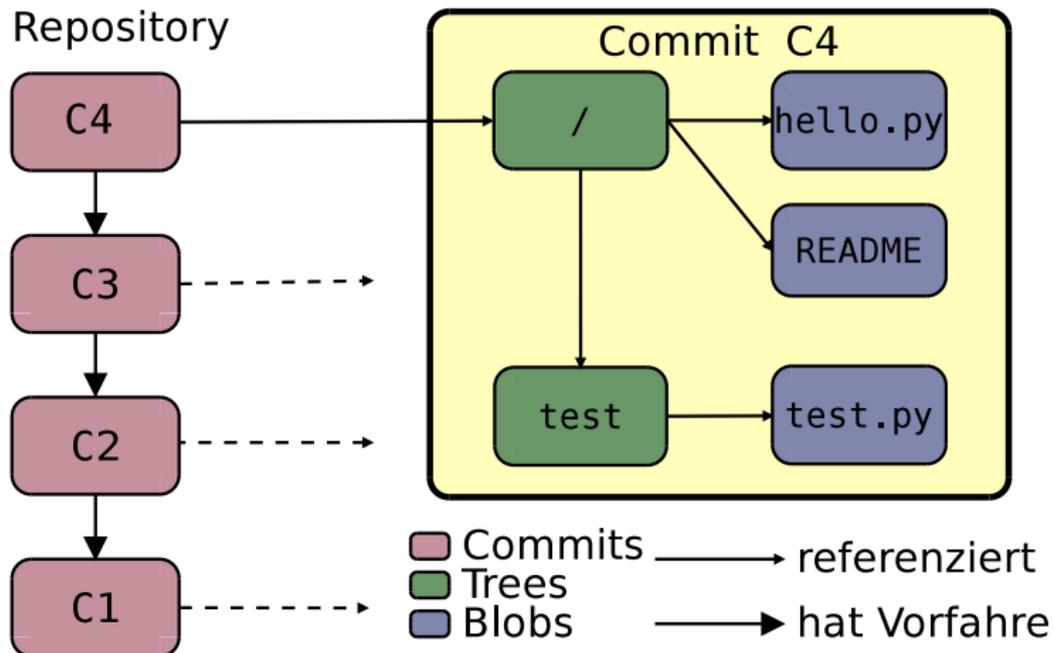


Objektverwaltung

- ▶ Alle Objekte werden von Git in der *Objektdatenbank* (genannt Repository) gespeichert
- ▶ Die Objekte sind durch ihre SHA-1 ID eindeutig adressierbar
- ▶ Für jede Datei erzeugt Git ein Blob-Objekt
- ▶ Für jedes Verzeichnis erzeugt Git ein Tree-Objekt
- ▶ Ein Tree-Objekt enthält die Referenzen (SHA1 IDs) auf die in dem Verzeichnis enthaltenen Dateien

Zusammenfassung

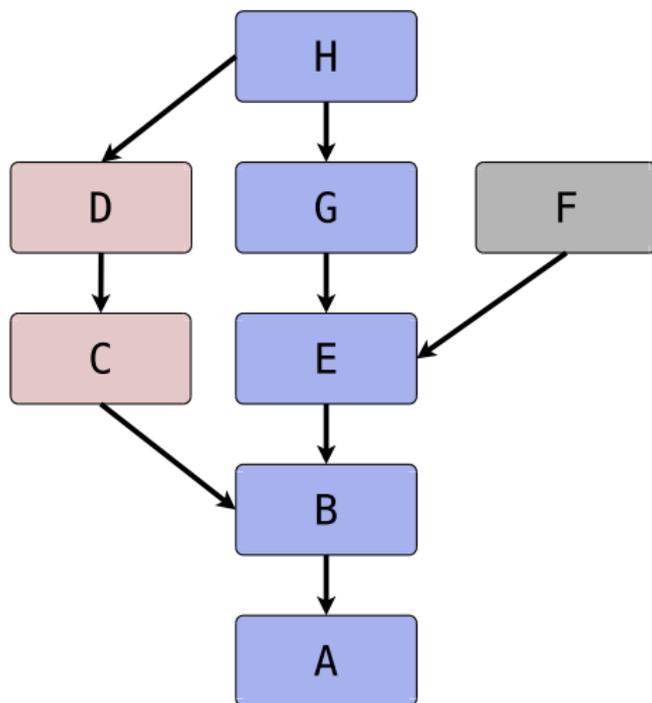
Ein Git-Repository enthält Commits; diese wiederum referenzieren Trees und Blobs, sowie ihren direkten Vorgänger



Commit Graph

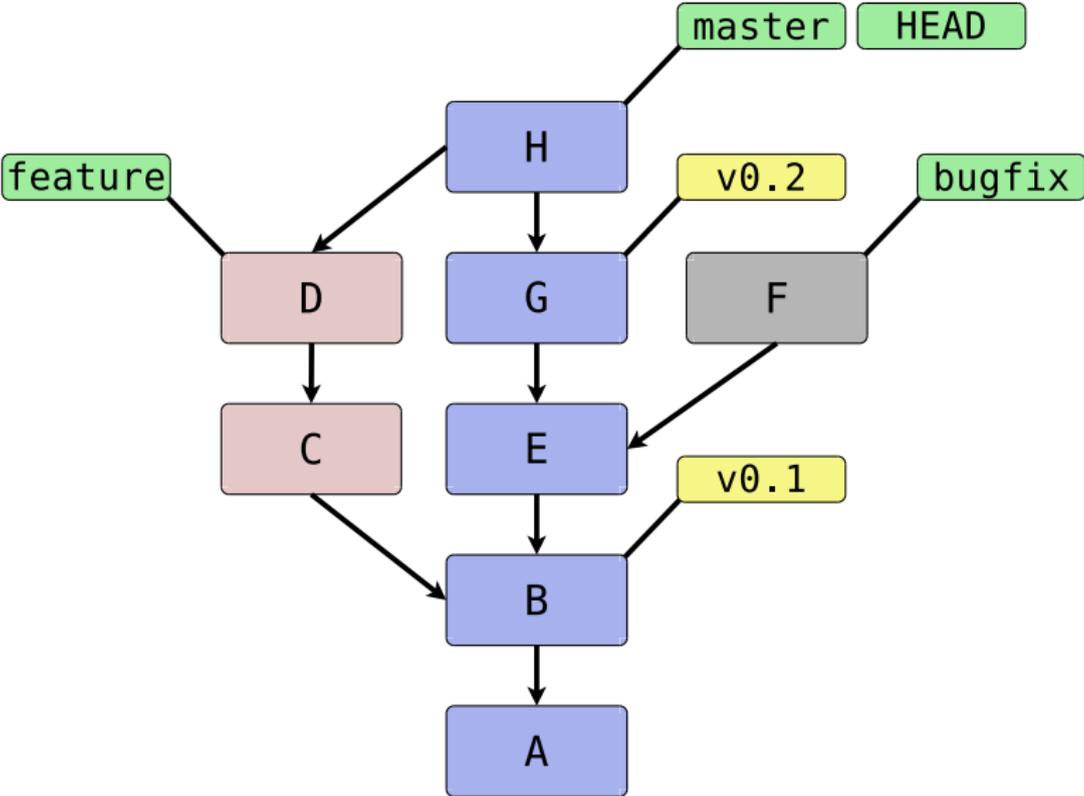
Ein Repository ist ein *Gerichteter Azyklischer Graph*

Engl.: Directed Acyclic Graph (DAG)



Branches und Tags

Branches und Tags sind Zeiger auf Knoten in dem Graphen.



Graph-Struktur

- ▶ Die gerichtete Graph-Struktur entsteht, da in jedem Commit Referenzen auf direkte Vorfahren gespeichert sind
- ▶ Integrität kryptographisch gesichert
- ▶ Git-Kommandos manipulieren die Graph-Struktur

Merging: Branches Zusammenfügen

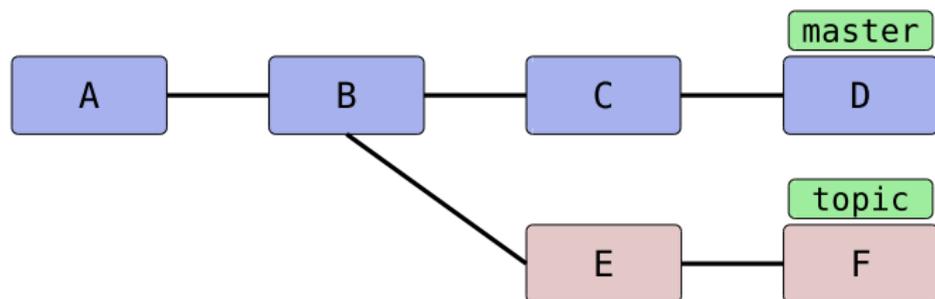
Simple Merge:

- ▶ `git merge neues-feature`

Fast-Forward Merge:

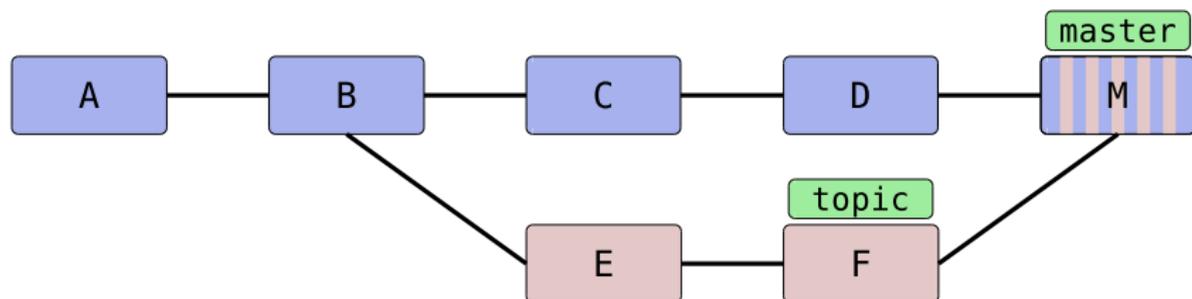
- ▶ Wird *topic* in *master* gemerget und *topic* basiert auf *master*, dann wird kein Merge-Commit erstellt, sondern nur der Zeiger »weitergerückt« bzw. »vorgespult«.

Vor dem Merge



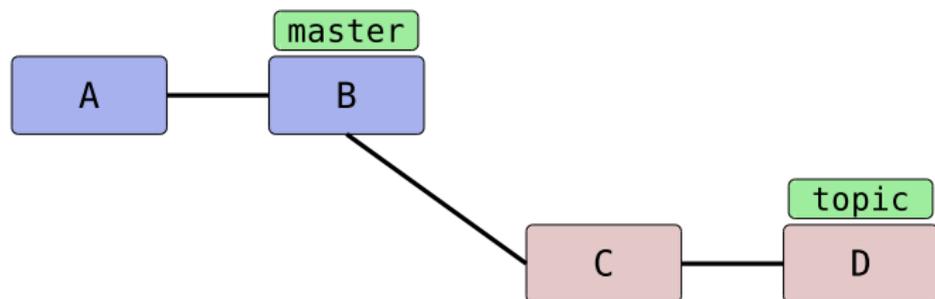
- ▶ topic ist fertig und soll in master integriert werden

Nach dem Merge



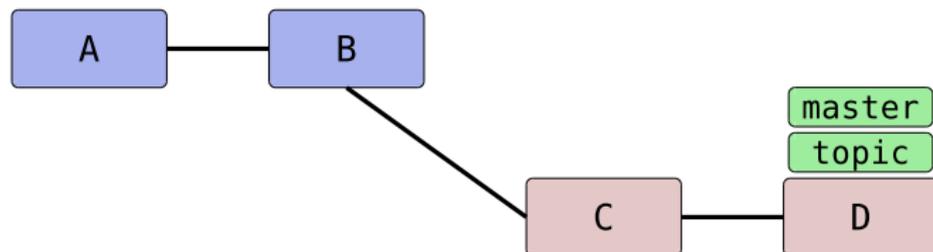
- ▶ Im master ausführen: `git merge topic`

Vor dem Fast-Forward



- ▶ In master hat sich nichts getan, topic ist fertig

Nach dem Fast-Forward



- ▶ master wird »weitergerückt«, bzw. »vorgespult«

Hilfe, Konflikte!

Bei einem merge kann es zu Konflikten kommen. Wie geht man damit um?

- ▶ `vim konfliktdateien`
- ▶ `git add konfliktdateien`
- ▶ `git commit -m "Merge-Konflikt behoben"`

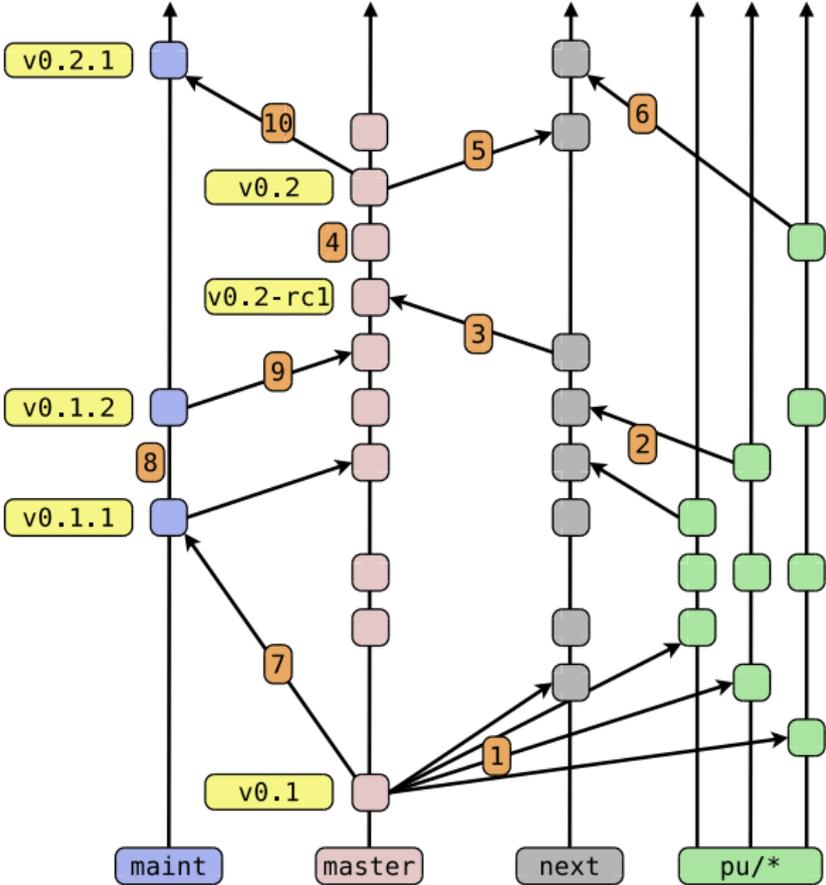
Das Unterfangen abbrechen:

- ▶ `git reset --hard HEAD`

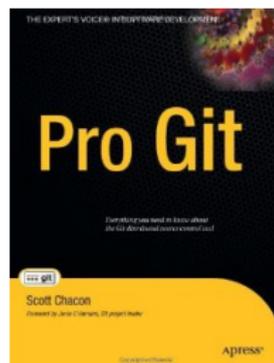
Branch-Modell

- ▶ Dieses Modell wird in vielen OS-Projekten verwendet
- ▶ Es gibt vier Haupt-Branches:
 - ▶ `maint`: Maintenance (Security und Bugfixes)
 - ▶ `master`: Vorbereitung des neuen Releases
 - ▶ `next`: Neue Features werden auf Stabilität getestet (Beta)
 - ▶ `pu`: *proposed updates*, unfertige, experimentelle Features
- ▶ `next` und `pu` werden möglicherweise verändert (neu aufgebaut, etc.)
- ▶ Bugfixes werden von `maint` in `master` übernommen
- ▶ Topic-Branches werden bei Vollendung in `next` übernommen
- ▶ Wird `next` stabil, wandern die Änderungen in den `master`

Branch-Modell – Visualisiert



Ressourcen: Bücher



- ▶ **Pro Git** von Scott Chacon, APress, 2009
- ▶ Kostenlos online verfügbar unter <http://progit.org/book/>



- ▶ **Git. Verteilte Versionskontrolle für Code und Dokumente**, Open Source Press, 2011
- ▶ <http://gitbu.ch/>

Ressourcen: Webseiten

- ▶ Webseite von Git: <http://git-scm.com/>
- ▶ Schnelle Übersicht: <http://gitref.org/>
- ▶ Tipps und Tricks: <http://gitready.com/>
- ▶ Hilfe bei Fragen: <http://stackoverflow.com/>
- ▶ Kostenlose Git-Repos: <http://github.com/>

Vorbereitung Für Teil II

1. Experimentiere mit den gelernten Kommandos
2. Verursache und löse einen Merge-Konflikt
3. Schau Dir den Google TechTalk von Linus Torvalds zum Thema Git an (von 2007)
4. Informiere dich, wie ein OSS-Projekt deiner Wahl Git verwendet
 - ▶ Und: Erstelle Dir einen Account bei GitHub o. Ä. – nächstes Mal werden wir das brauchen!
 - ▶ Bei Fragen und Wünschen: E-Mail schreiben!

Danke!

Vielen Dank für eure Teilnahme!

Fragen und Feedback gerne per Mail:

julius@plenz.com

Git-Workshop, Teil II

Freitagsrunde TechTalks, TU Berlin

Julius Plenz

2. Dezember 2011



Veröffentlicht unter der CreativeCommons-Lizenz (By, Nc, Sa)

<http://wiki.freitagsrunde.org/TechTalks>

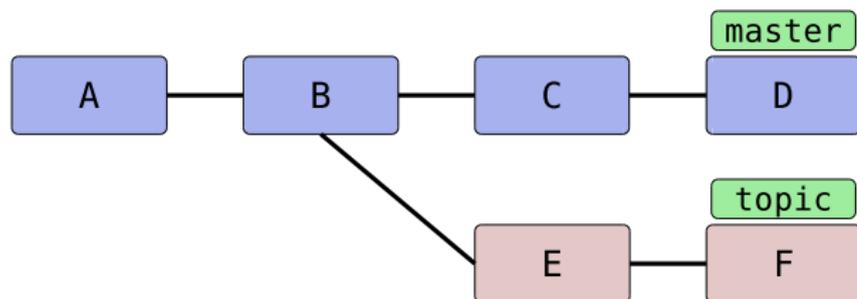
Ablaufplan

- ▶ Teil I:
 - ▶ Grundlegende Arbeitsschritte in Git
 - ▶ Das Objektmodell – eine theoretische Grundlage
 - ▶ Parallele Entwicklung: Branches und Merges
 - ▶ Entwicklung koordinieren: Ein Branching-Modell
- ▶ Teil II:
 - ▶ Die Geschichte umschreiben: Rebase
 - ▶ Einschub: Wunschkonzert – was euch interessiert
 - ▶ Verteiltes Git: Commits hoch- und runterladen
 - ▶ Verschiedene Workflows
 - ▶ Kollaboration live: Wir entwickeln gemeinsam ein Projekt
- ▶ Teil I findet komplett *lokal* auf dem eigenen Rechner statt – erst in Teil II nutzen wir die verteilten Eigenschaften von Git aus

Einen Commit ändern

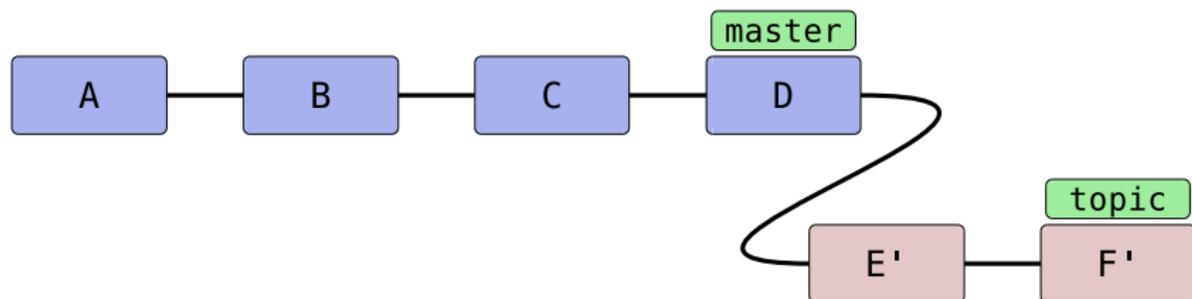
- ▶ Commit ändern = Neuen Commit erstellen, alten wegschmeißen
- ▶ Den letzten Commit (HEAD) ändern:
 1. `$EDITOR datei`
 2. `git add datei`
 3. `git commit --amend`
- ▶ Tiefer liegende Commits (HEAD~1 etc.) können so nicht geändert werden!

Vor dem Rebase



- ▶ `topic` soll auf der neusten Version von `master` basieren

Nach dem Rebase



► `git rebase master topic`

Rebase: Auf eine neue Basis bauen

- ▶ **Rebase:** Einen Branch auf eine »neue Basis« stellen.

master als neue Basis für *topic*

```
git checkout topic  
git rebase master
```

Alternativ

```
git rebase master topic
```

Rebasing: eine Warnung

- ▶ **Wichtig:** Man darf *niemals* Commits aus einem bereits veröffentlichten Branch – auf dem also womöglich Andere ihre Arbeit basieren – durch `git rebase` verändern!
- ▶ **Daher: Nur Unveröffentlichtes gegen Veröffentlichtes rebase:**
 - ▶ `git rebase origin/master`
 - ▶ `git rebase v1.1.23`

Rebase Interaktiv

- ▶ Das ist *Advanced Git Magic* – und will geübt sein!
- ▶ Rebase-Prozess anhalten, Commits »mittendrin« ändern, weiterlaufen lassen

Interaktives Rebase

```
git rebase -i master topic
```

- ▶ Anwendungsfälle *nur* lokal und für die *eigenen* Commits
 - ▶ Patch-Serie neu strukturieren
 - ▶ Typos aus den eigenen Commits entfernen
 - ▶ Offensichtliche Fehler glattbügeln

Rebase Interaktiv: Beispiele

Zwei Commits zusammenfassen

```
git rebase -i HEAD~n
```

→ pick des zweiten Commits durch `fixup` ersetzen

- ▶ Einen Commit verschieben: Die Zeilen vertauschen
- ▶ Einen Commit editieren: mit `edit` markieren
- ▶ Einen Commit aufteilen: Siehe Cheatsheet

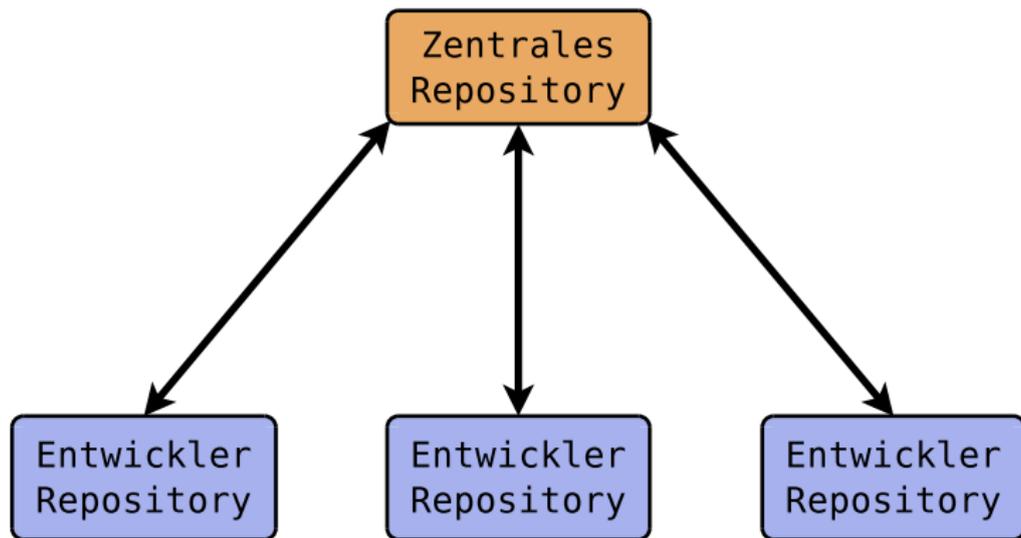
Whitespace und EOL

- ▶ Was ist kaputter Whitespace?
- ▶ `git diff --check` (z. B. → Hook)
- ▶ *Zeilenende*: Windows (CRLF) vs. UNIX (LF)
- ▶ `core.eol` bestimmt, was zu tun ist: `lf`, `crLf` oder `native`
- ▶ Git-Attribut `text` für Dateien, die automatisch konvertiert werden sollen
 - ▶ `echo '*.c text' > .gitattributes`
- ▶ `core.safecrLf`: Konvertierung verbieten, wenn ein Mix aus CRLF und LF vorhanden ist
- ▶ Mehr Infos: `gitattributes(5)`

Hinaus in die weite Welt!

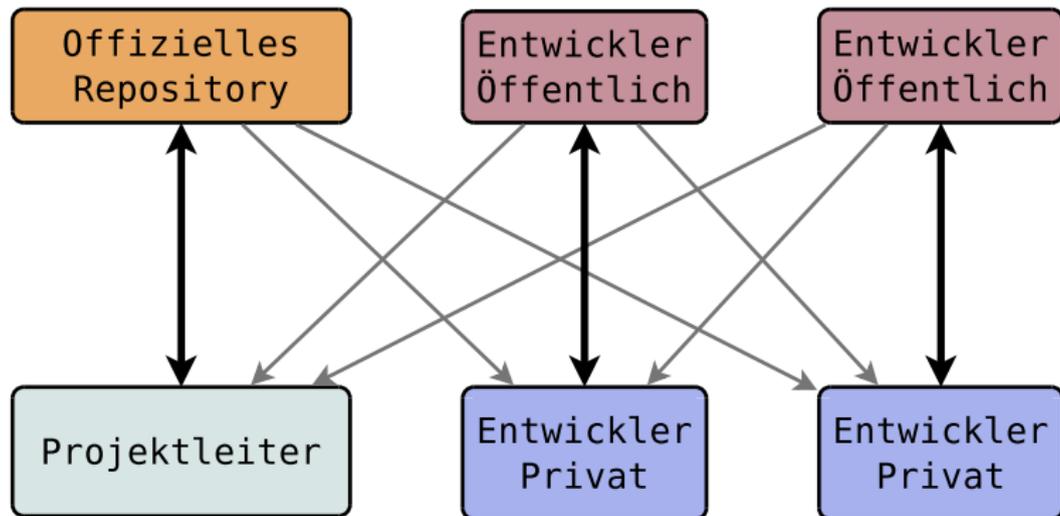
- ▶ Wir wollen unsere Arbeit mit der anderer Entwickler austauschen!
- ▶ Durch die verteilte Architektur von `git` braucht es keinen *zentralen* Server zu geben.
- ▶ Das Entwicklerteam muss sich auf einen *Workflow* einigen:
 - ▶ Shared Repository
 - ▶ Maintainer/Blessed Repository
 - ▶ Patch-Queue per E-Mail
 - ▶ ... oder auch alles durcheinandergemixt.

Zentralisiert



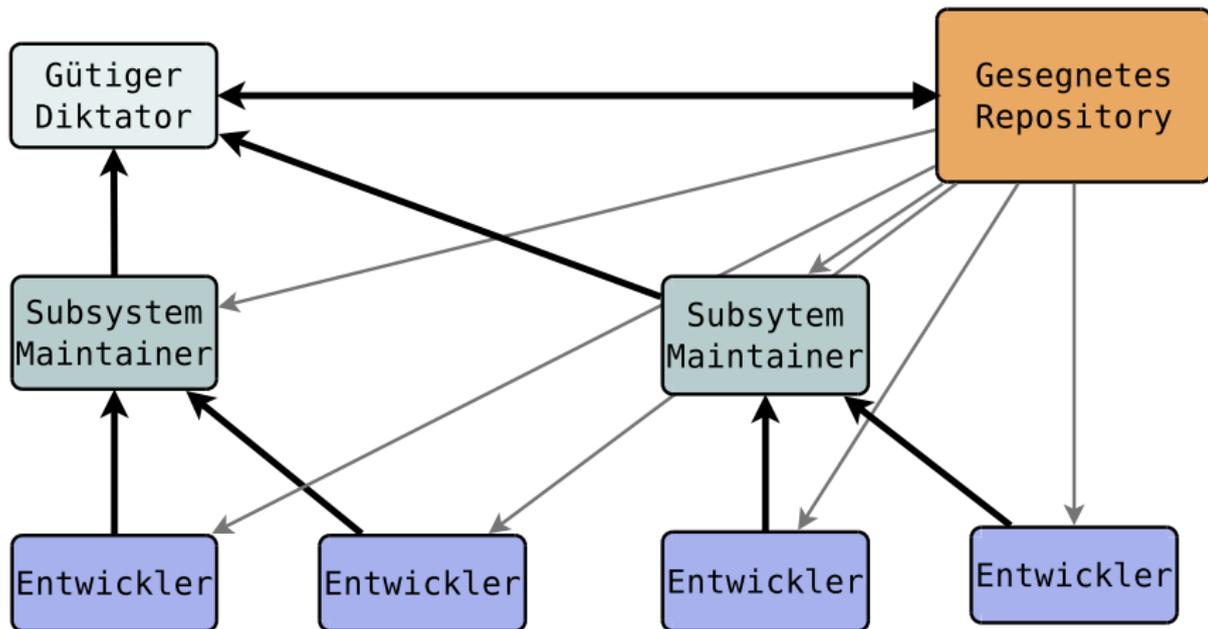
- ▶ Ein einziges zentrales Repository
- ▶ Alle Entwickler haben Schreibzugriff

Öffentliche Entwickler-Repositories



- ▶ Ein öffentliches Repository pro Entwickler
- ▶ Der Projektleiter integriert Verbesserungen

Patch-Queue per Email



- ▶ Stark vom Kernel und Git selbst verwendet

Remote Repositories / Remote Branches

Remote Repositories verwalten:

- ▶ `git remote -v`
- ▶ `git remote add name url`
- ▶ `git remote rm name`
- ▶ `git remote update`
 - ▶ Fragt bei allen Remote Repositories an, ob es neue Commits gibt. (Eigene Commits werden durch dieses Kommando **nicht** veröffentlicht!)

Details der Repositories ändern (z. B. Vertipper):

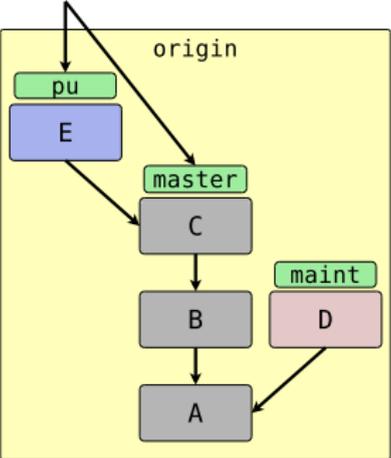
- ▶ `vim .git/config`

Remote Branches auflisten:

- ▶ `git branch -r`

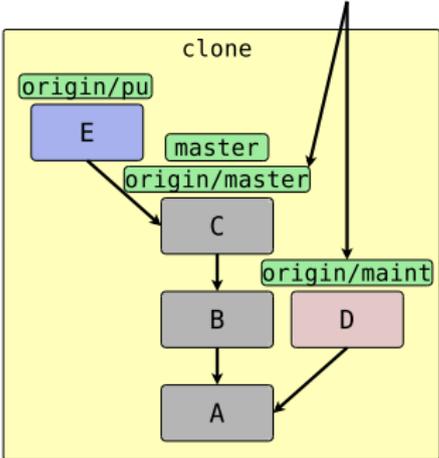
Remote Branches vs. Remote Tracking Branches

Remote-Branches



git clone

Remote-Tracking-Branches



Fremden Code holen, eigenen versenden

Aus einem anderen Repository neuen Code »ziehen«:

- ▶ `git pull remote branch`
 - ▶ `git pull blessed master`

Was hinter den Kulissen passiert:

1. `git fetch remote branch`
2. `git merge remote/branch`

Eigene Commits »pushen« oder per E-Mail senden:

- ▶ `git push remote branch`
- ▶ `git format-patch seit-wann`

Konventionen

- ▶ Wiederholter Einsatz von `git pull` erzeugt viele unnötige Merges
- ▶ Konvention:
 - ▶ Nicht im `master` entwickeln
 - ▶ `git remote update, master immer Fast-Forwarden`
 - ▶ Eigene Branches per merge in `master` integrieren

FF-Merge erzwingen

```
git merge --ff-only origin/master  
git config --global alias.fm 'merge --ff-only'
```

GitHub – „Social Coding“

- ▶ GitHub stellt Git-Repositories zur Verfügung
 - ▶ Kostenlos und viel genutzt
 - ▶ Web-basiertes Interface
 - ▶ Aktionen „Fork“, „Follow“ und „Watch“
- ▶ Account erstellen:
 - ▶ → <http://www.github.com>
 - ▶ Authentifizierung per SSH-Schlüssel (ggf. erstellen)
- ▶ Ein eigenes Repository hochladen:
 - ▶ Repository auf GitHub erstellen
 - ▶ `git remote add github`
`ssh://git@github.com:user/projekt.git`
 - ▶ `git push github master`

Let's develop!

- ▶ Wir entwickeln in Plaintext – Markdown macht's schön!
- ▶ Eine Hand voll Teams (5?)
- ▶ Jedes Team wählt einen Koordinator
- ▶ Koordinatoren arbeiten dem Maintainer zu
- ▶ Entwickler: Tauscht euch aus! (Remotes, Patches, ...)

Danke!

Vielen Dank für eure Teilnahme!

Fragen und Feedback gerne per Mail:

julius@plenz.com