Praktikum angewandte Systemsoftwaretechnik

Aufgabe 3

Alexander Würstlein

Lehrstuhl Informatik 4

10. November 2016

Versionierung

Typische Aufgaben eines Versionskontrollsystems sind:

- Transportmedium
- Sichern von alten Zuständen
- Zusammenführung von parallelen Entwicklungen

Idealerweise zusätzlich:

- Unabhängige Entwicklung ohne zentrale Infrastruktur

Versionierung

Typische Aufgaben eines Versionskontrollsystems sind:

- Transportmedium
- Sichern von alten Zuständen
- Zusammenführung von parallelen Entwicklungen

Idealerweise zusätzlich:

- Unabhängige Entwicklung ohne zentrale Infrastruktur

Das Versionskontrollsystem des Linux Kernels

Git wurde 2005 von Linus Torvalds zur Unterstützung für die Linux Kernel Entwicklung geschrieben. Es sind viele Erfahrungen im Umgang mit großen Patchmengen und das Vorgängersystem *bitkeeper*. Es unterstützt:

- Dezentrale, parallele Entwicklung
- Koordinierung von Hunderten von Entwicklern
- Visualisierung von Entwicklungszweigen

Wie funktioniert GIT?

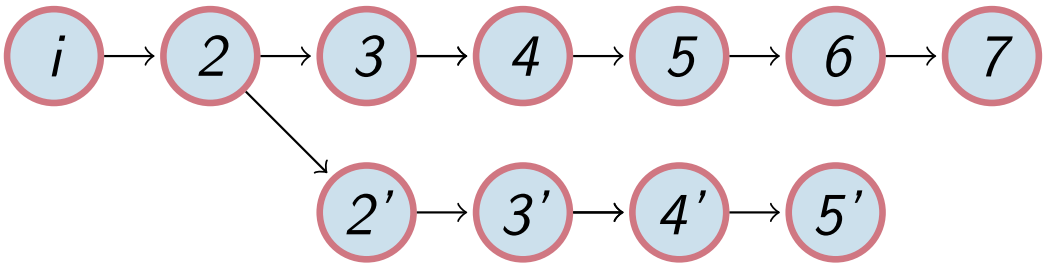
- Es werden immer die vollständigen Daten jedes Versionsstandes gespeichert
- Jede Version ist eindeutig durch einen SHA1-Hash identifizierbar
- Jede Version kennt ihren Vorgänger („parent“)
- Jedes Ende einer Versions-Serie („branch“) bekommt einen Namen (Standard: master)

Wie funktioniert GIT?

- Es werden immer die vollständigen Daten jedes Versionsstandes gespeichert
- Jede Version ist eindeutig durch einen SHA1-Hash identifizierbar
- Jede Version kennt ihren Vorgänger („parent“)
- Jedes Ende einer Versions-Serie („branch“) bekommt einen Namen (Standard: master)

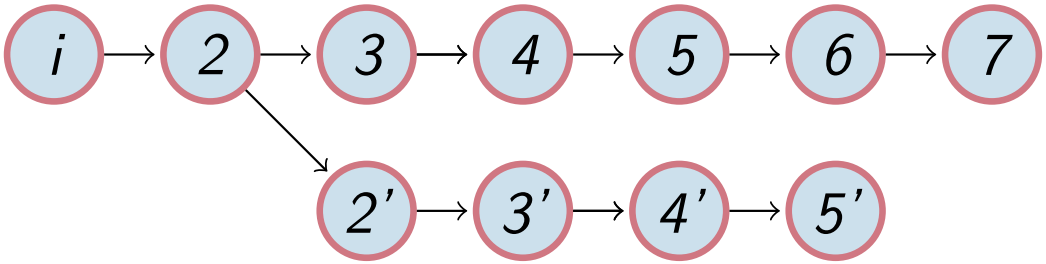
Wie funktioniert GIT?

- Es werden immer die vollständigen Daten jedes Versionsstandes gespeichert
- Jede Version ist eindeutig durch einen SHA1-Hash identifizierbar
- Jede Version kennt ihren Vorgänger („parent“)
- Jedes Ende einer Versions-Serie („branch“) bekommt einen Namen (Standard: master)



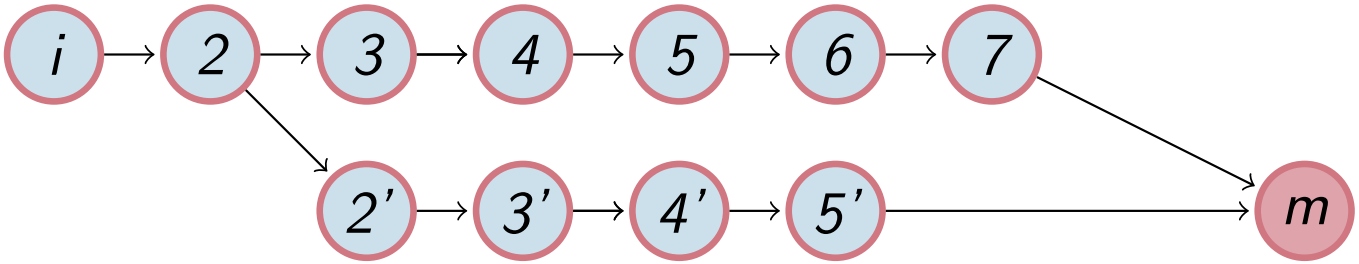
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



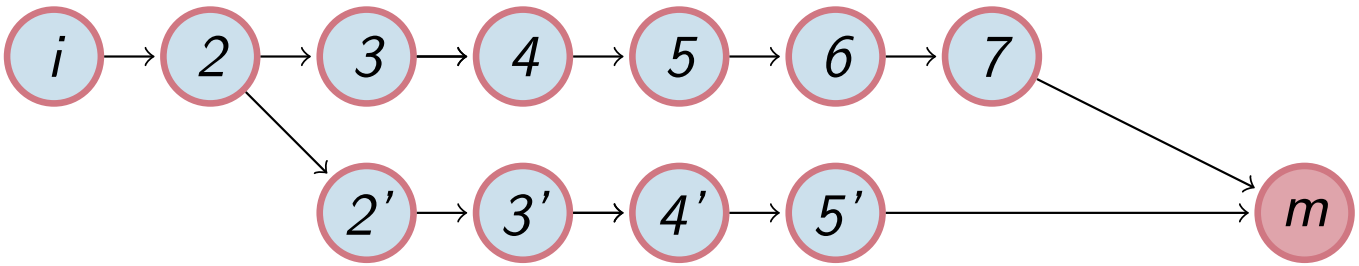
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



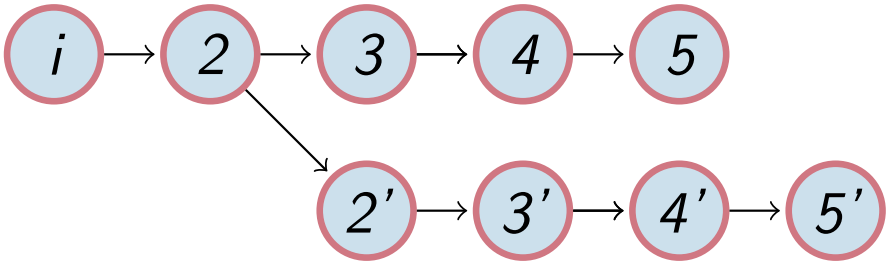
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):

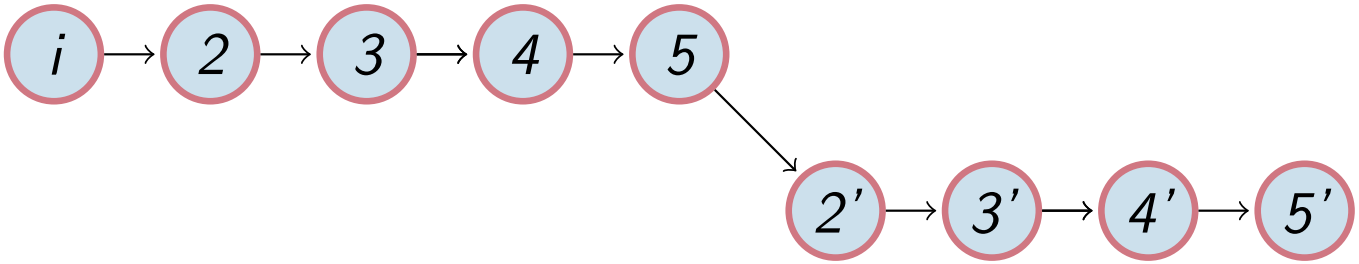


- Git versucht beide Änderungen zusammenzuführen
- Bei nicht eindeutigen Änderungen („Konflikte“) sind manuelle Eingriffe nötig

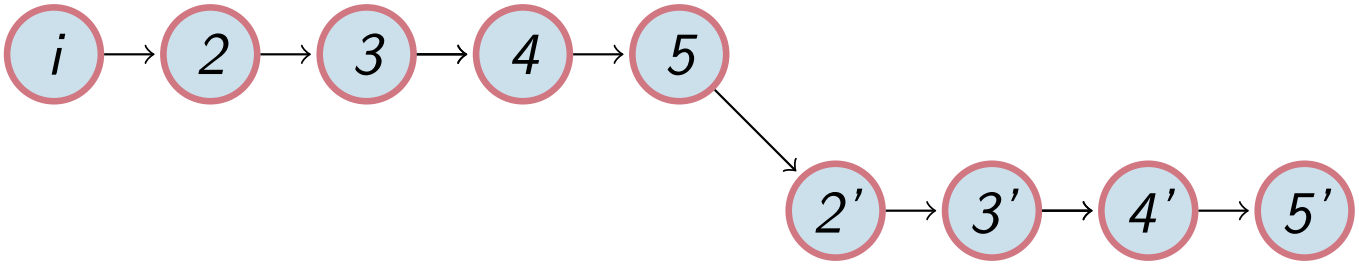
Aufsetzen auf bestehenden Zweigen (rebase)



Aufsetzen auf bestehenden Zweigen (rebase)



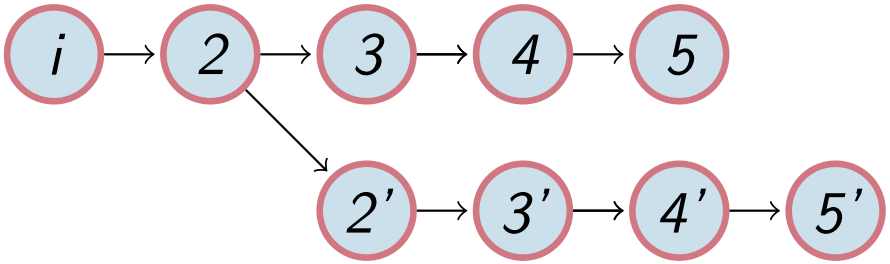
Aufsetzen auf bestehenden Zweigen (rebase)



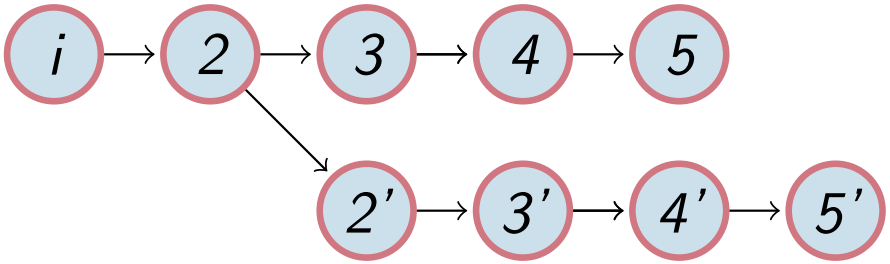
- Patches aus dem "unterem" Zweig werden auf den "oberen" aufgespielt
- Die Historie ist nun linear
- Linearisierte Änderungen lassen sich häufig einfacher bewerten
- Vorsicht!
  - Verzweigungen vom alten Zweig können nun nicht mehr zusammengeführt werden
  - Keine gemeinsamen Vorgänger mehr
  - Nach einem Rebase haben alle Patches neue Hashes

Geschichte neu schreiben (rebase –i)

Geschichte neu schreiben (rebase –i)



Geschichte neu schreiben (rebase –i)



- Alte Versionen können bearbeitet, zusammengefasst, aufgeteilt oder eingefügt werden
- Jeder Patch bekommt einen neuen Hash

Wichtige Git Kommandos zum Austauschen von Code

- Initialisieren einen Repos im aktuellen Verzeichnis:

```
git init
```

- Initiales *Klonen* der Quellen:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.g
```

- Einspielen von eigenen Änderungen in Datei oder aller Änderungen:

```
git commit Datei
```

```
git commit -a
```

- Hinzufügen einer neuen Datei zur Menge der von git versionierten Dateien:

```
git add Datei
```

Wichtige Git Kommandos zum Austauschen von Code (Forts.)

- Markieren einer versionierten Datei als Kandidat für den nächsten commit („staging“):

```
git add Datei
```

- Anzeige der Differenzen zum Vorgänger (bzw. Anzeige des vorbereiteten („staged“) commits):

```
git diff
```

```
git diff [--staged|--cached]
```

- Dateizustände (neu, unbekannt, geändert, staged) anzeigen:

```
git status
```

- Die neuste Änderung untersuchen:

```
git show
```

Wichtige Git Kommandos zum Austauschen von Code (Forts.)

- Einspielen von entfernten Änderungen:

```
git pull
```

- Weitere entfernte Repositories registrieren:

```
git remote add 32-stable git://git.kernel.org/.../longterm/linux-2.6.32.y.git
```

- Registrierte Repositories auflisten:

```
git remote -v
```

- Alle Remotes nachladen (aktueller Branch wird nicht verändert):

`git remote update` oder `git fetch --all`

Wichtige Git Kommandos zum Austauschen von Code (Forts.)

- Lokalen Branch aus dem neuem "Remote" anlegen:

```
git checkout -b work 32-stable/master
```

- Alle registrierten Zweige anzeigen:

```
git branch -a
```

- Unterschiede zwischen lokalem und entferntem Branch untersuchen:

```
git log ..origin/master
```

```
git log -p ..origin/master
```

- Aktuelle Änderungen auf dem entfernten Branch neu aufspielen:

```
git pull --rebase
```

Wichtige Git Kommandos zum Austauschen von Code (Forts.)

- Die letzten **2** Änderungen als Patch formatieren:

`git format-patch HEAD~2` wählt bei einem Merge den **ersten** "Elter"

`git format-patch HEAD^^` wählt bei einem Merge den **zweiten** "Elter"

`git format-patch HEAD^4~~` es ist möglich dies zu kombinieren

- Sendeziel für Patchversand via E-Mail vorgeben:

```
git config sendemail.to=linux-kernel@i4.cs.fau.de
```

- Patchset mit den letzten **3** Änderungen via E-Mail senden:

```
git send-email --compose HEAD~3
```

Sonstige Werkzeuge

- `gitg`, `gitk`, `tig`
- `git gui`
- `git-meld` (globales `git difftool`)
- `https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools`
- `https://gitlab.cs.fau.de`