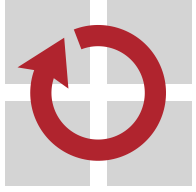
Praktikum angewandte Systemsoftwaretechnik (PASST)

Versionskontrollsysteme / Aufgabe 3

15. November 2018

Tobias Langer, Stefan Reif, Michael Eischer,
Bernhard Heinloth und Florian Schmaus

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Versionskontrollsysteme

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. Dabei sind viele Erfahrungen im Umgang mit großen Patchmengen und das Vorgängersystem *bitkeeper* in die Entwicklung eingeflossen.

Es unterstützt:

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

Wie funktioniert GIT?

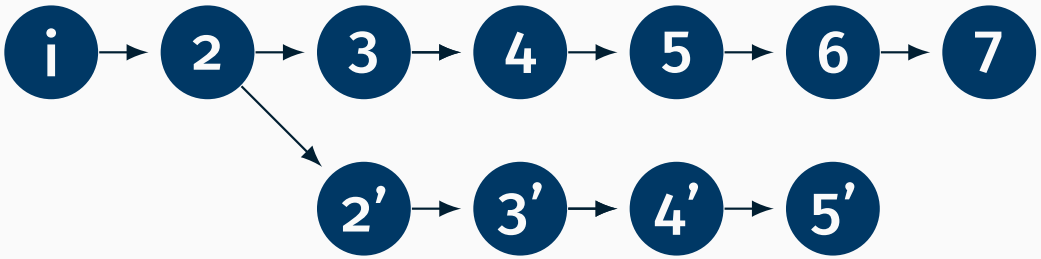
- Es werden immer die vollständigen Daten jedes Versionsstandes gespeichert
- Jede Version ist eindeutig durch einen SHA1-Hash identifizierbar
- Jede Version kennt ihren Vorgänger („parent“)
- Jedes Ende einer Versions-Serie („branch“) bekommt einen Namen (Standard: master)

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)

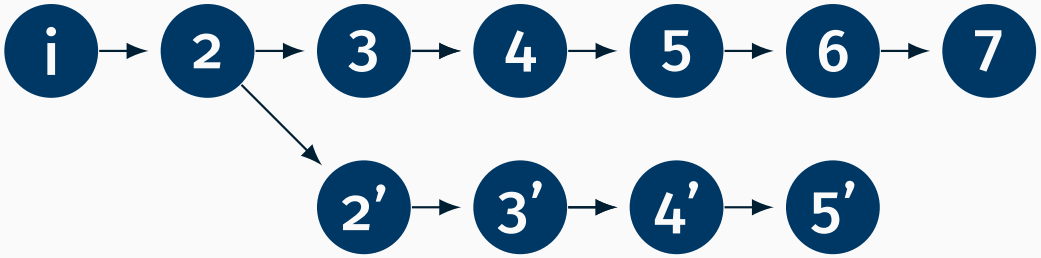
Wie funktioniert GIT?

- 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)



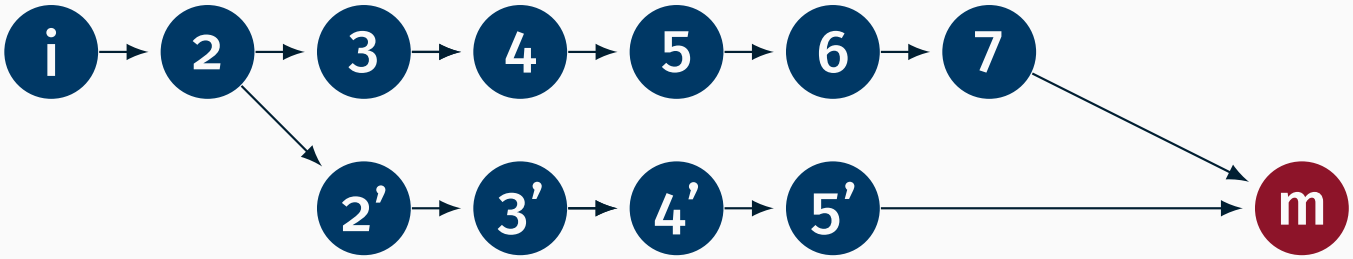
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



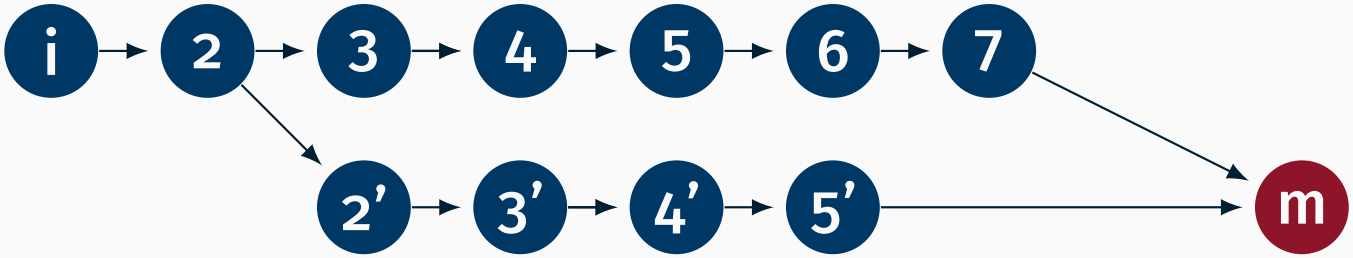
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):



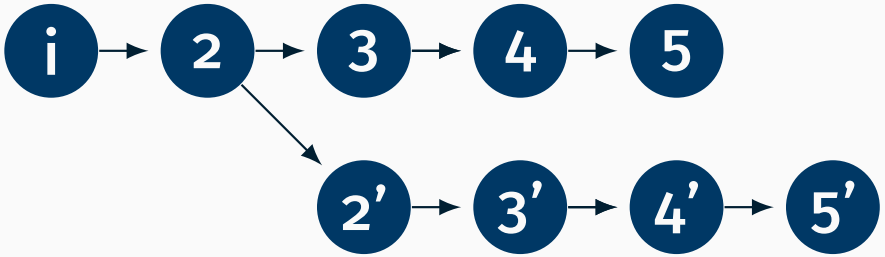
Verzweigungen und Zusammenführungen

Beispiel für parallele Entwicklung (branches und merge):

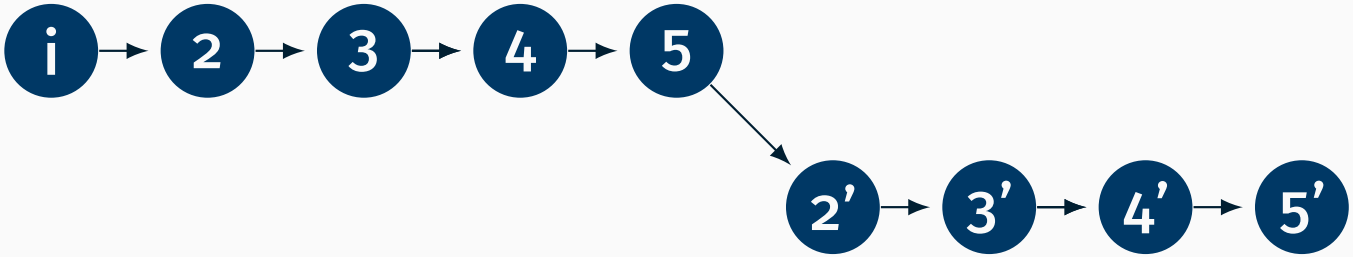


- Git versucht beide Änderungen zusammenzuführen
- Bei nicht eindeutigen Änderungen („Konflikte“) sind manuelle Eingriffe nötig

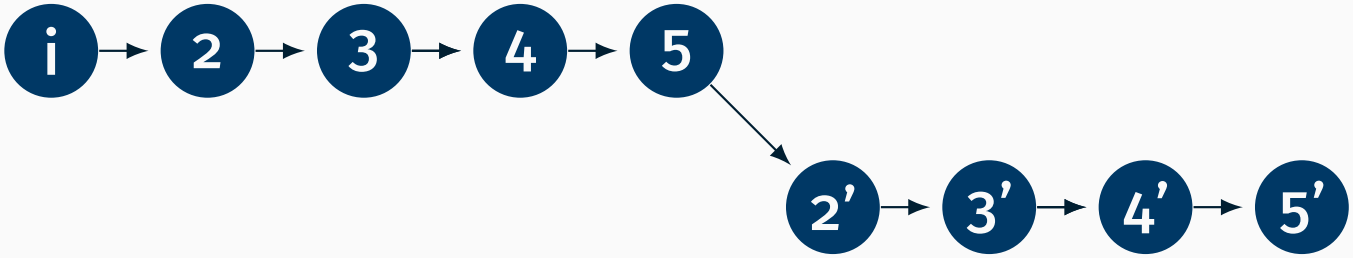
Aufsetzen auf bestehenden Zweigen (rebase)



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

Git interactive rebase (`rebase -i`)

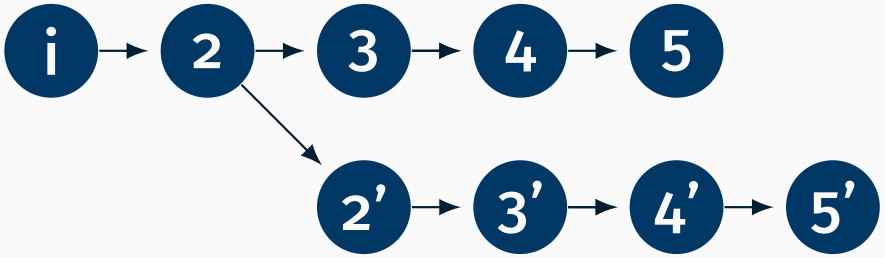
Erlaubt es die Versionsgeschichte „neu“ zu schreiben, also Commits

- umsortieren
- verschmelzen
- aufteilen
- nachträglich ändern
- …

Git interactive rebase (`rebase -i`)

Erlaubt es die Versionsgeschichte „neu“ zu schreiben, also Commits

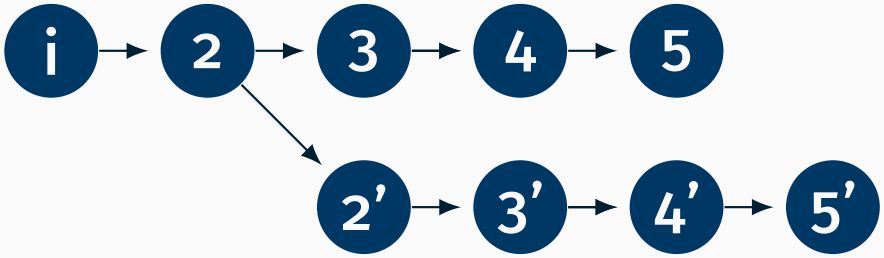
- umsortieren
- verschmelzen
- aufteilen
- nachträglich ändern
- …



Git interactive rebase (`rebase -i`)

Erlaubt es die Versionsgeschichte „neu“ zu schreiben, also Commits

- umsortieren
- verschmelzen
- aufteilen
- nachträglich ändern
- …



In der Regel wirken sich die Operationen auf den Hash aus!

Wichtige Git Kommandos zum Austauschen von Code (1/7)

- Initialisieren einen Repos im aktuellen Verzeichnis

```
01 git init
```

- Initiales Klonen der Quellen

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

- Einspielen von eigenen Änderungen in Datei oder aller Änderungen

```
01 git commit Datei
02 git commit -a
```

Wichtige Git Kommandos zum Austauschen von Code (2/7)

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

```
01 git add Datei
```

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

```
01 git add Datei
```

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

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

Wichtige Git Kommandos zum Austauschen von Code (3/7)

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

```
01 git status
```

- Die neuste Änderung untersuchen

```
01 git show
```

- Einspielen von entfernten Änderungen

```
01 git pull
```

Wichtige Git Kommandos zum Austauschen von Code (4/7)

- Weitere entfernte Repositories registrieren

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

- Registrierte Repositories auflisten

```
01 git remote -v
```

- Alle Remotes nachladen (aktueller Branch wird nicht verändert)

```
01 git remote update
02 # oder
03 git fetch --all
```

Wichtige Git Kommandos zum Austauschen von Code (5/7)

- Lokalen Branch aus dem neuem „Remote“ anlegen

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

- Alle registrierten Zweige anzeigen

```
01 git branch -a
```

- Unterschiede zwischen lokalem und entferntem Branch untersuchen

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

Wichtige Git Kommandos zum Austauschen von Code (6/7)

- Aktuelle Änderungen auf dem entfernten Branch neu aufspielen

```
01 git pull --rebase
```

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

```
01 # bei einem Merge den ersten Vorgänger wählen
02 git format-patch HEAD~2
03
04 # bei einem Merge den zweiten Vorgänger wählen
05 git show HEAD^2
06
07 # es ist moeglich dies zu kombinieren
08 git show HEAD^4~~
```

Wichtige Git Kommandos zum Austauschen von Code (7/7)

- Sendeziel für Patchversand via E-Mail vorgeben

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

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

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

Sonstige Werkzeuge

```
gitg
gitk
tig

git gui

# globales git difftool
git-meld
```

- https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools
- https://gitlab.cs.fau.de

Kernel debuggen

Debuggen des Linux-Kernels (1/2)

- kgdb nicht für alle Fehlertypen der beste Ansatz
  - Was tun, wenn der Kernel sehr früh Oopst?
  - Wie vorgehen, wenn die serielle Schnittstelle debugged werden soll?
  - Wenn alles „optimized out“ ist, wie sehe ich trotzdem was passiert?
- Lösung: `printk()`-Ausgaben auf der Konsole und im Kernel-Log

Debuggen des Linux-Kernels (2/2)

Prototyp:

```
01 int printk(const char *s, ...)
```

Beispiel aus `linux-3.0/init/main.c`:

```
01 printk(KERN_NOTICE "Kernel command line: %s\n", boot_cmd_line);
```

Meldungen *nachlesen*:

```
01 passt [~]> dmesg
02 ...
03 [0.000000] Kernel command line: root=/dev/vda1 console=ttyS0
04 ...
```

Debuggen mit `printk()`

- Alle Ausgaben haben eine Priorität (<n> am Stringanfang)
- Kernel Log-Level muss für Ausgabe mindestens auf $n + 1$ gesetzt sein
- Log-Level wird standardmäßig mit 7 initialisiert
- Anpassung über Kommandozeile (`debug`, `loglevel`) und `klogd(8)`

Mögliche Prioritäten (`linux-3.0/include/linux/printk.h`)

```
01 #define KERN_EMERG   "<0>" /* system is unusable                */
02 #define KERN_ALERT   "<1>" /* action must be taken immediately */
03 #define KERN_CRIT    "<2>" /* critical conditions              */
04 #define KERN_ERR     "<3>" /* error conditions                 */
05 #define KERN_WARNING "<4>" /* warning conditions               */
06 #define KERN_NOTICE  "<5>" /* normal but significant condition */
07 #define KERN_INFO    "<6>" /* informational                    */
08 #define KERN_DEBUG   "<7>" /* debug-level messages             */
```

Code-Navigation mit `cscope`

```
01 make cscope
02 cscope -d
```

- Funktionen finden
- Aufrufer und Aufgerufene
- Integration in Editoren
- ähnliche Funktionalität per Eclipse, etc.

Aufgabe 3

Fehler finden und beheben

- Vorgegebene Kernelquellen mit injizierten Fehlern:
  `/proj/i4passt/kernel/ws18/passt-linux-borked.git/`
- Vorgegebene Kernelconfig:
  `/proj/i4passt/kernel/ws18/passt-linux-borked.config`
- Verschiedene Fehlertypen, *nicht nur* Systemabstürze
  - System muss „normal“ benutzt werden um alle Fehler zu finden
  - kgdb nicht immer das optimale Werkzeug
- Insgesamt zwölf verschiedene Fehler
- Patches an `linux-kernel@i4.cs.fau.de`
  - Achtung, Hannover liest mit!

Bearbeitungszeit bis 27. November

Fragen?