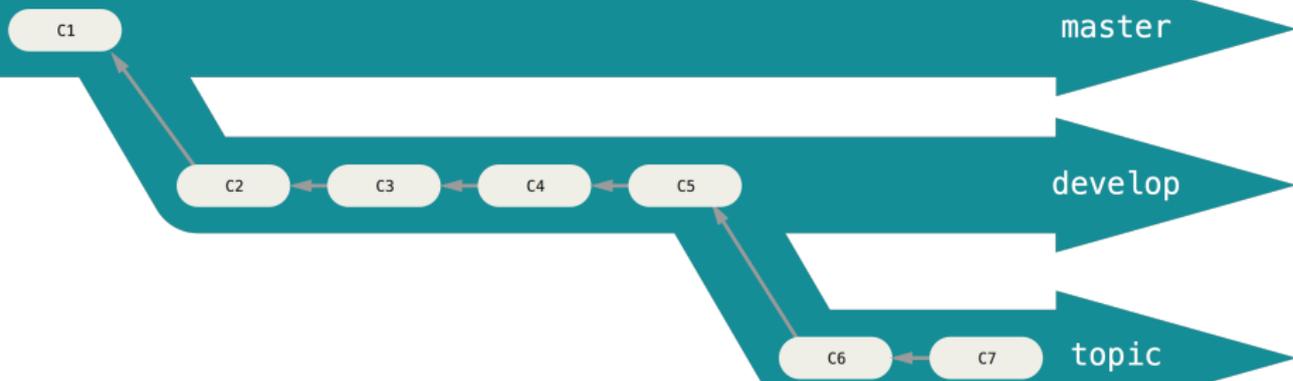


Distributed Source Code Management mit git

Thomas Forbriger

GEOPHYSIKALISCHES INSTITUT



Wozu *Source Code Management* (SCM)?

Erstes Problem (im Studium):

Synchron auf mehreren Rechnern arbeiten

Ich arbeite am Institut, zu Hause und auf meinem Notebook an meinen Programmtexten oder meiner Masterarbeit. Wenn ich auf dem Notebook etwas geändert habe, muss ich es auch nach Hause und auf den Rechner am Institut kopieren, um dort daran weiter arbeiten zu können. Weil ich so schrecklich viele Dateien bearbeite, habe ich die Übersicht verloren und eine Datei nicht kopiert. Jetzt ist mir alles durcheinander geraten. . .

Wozu *Source Code Management* (SCM)?

Zweites Problem (im Studium):

Komplexe Änderungen an einem Programm

Ich habe ein großes Programm geschrieben. Jetzt möchte ich einen Teil des Programms komplett überarbeiten, die Struktur ändern und neue Schleifen einfügen, damit es eine weitere Aufgabe lösen kann. Ich habe Angst, das zwischendrin etwas schief geht und ich die Änderungen dann nicht mehr rückgängig machen kann, weil ich vergessen habe, was alles geändert wurde. Also mache ich mir zur Sicherheit eine Kopie des ursprünglichen Programms. Leider habe ich schon fünf solche Kopien an unterschiedlichen Orten. Ich kann sie nicht mehr auseinander halten. Welches war denn bloß die zuletzt noch funktionstüchtige Kopie?

Wozu *Source Code Management* (SCM)?

Drittes Problem (im Studium):
Sicherheitskopien

Ich arbeite an meiner Masterarbeit. Ich habe schon so viel Zeit und Arbeit in meine Programme und den Text der Masterarbeit gesteckt, dass ich regelmäßig Sicherheitskopien davon machen möchte. Am liebsten auf einen kleinen Wechseldatenträger. Leider liegen in meinen Verzeichnissen die Texte zusammen mit allerhand Datenfiles und Grafikdateien. Es ist so mühsam jedesmal die Quelltexte rauszusuchen und dabei sicherzustellen, dass ich keine vergessen habe. . .

Wozu *Source Code Management* (SCM)?

Viertes Problem (für das SCM eigentlich erfunden wurde):
Arbeiten im Team am selben Quelltext

Ich arbeite mit 17 Kollegen an einem großen Softwareprojekt. Das Programm hat insgesamt 285 Quelldateien in 31 Verzeichnissen. Viele davon habe ich selber noch nie angesehen. Ab und zu muss ich Dateien bearbeiten, die von anderen Kollegen geschrieben wurden. Oder: Kollegen müssen meine Quelltexte an ihre neuen Module anpassen. Wir können uns unmöglich immer alle Änderungen per E-Mail zuschicken und dabei sicherstellen, dass jeder einen identischen Satz der 285 Quelldateien hat und verwendet. Und vor allem: Was passiert wenn mir ein Kollege eine geänderte Datei zuschickt, die ich gerade selber ändere? Wie entsetzlich mühsam ist es jetzt seine Änderungen in die von mir bereits geänderte Datei mit einzuarbeiten. . .

Wozu *Source Code Management (SCM)*?

Fünftes Problem (relevant für *Community-Software*):
Veröffentlichung von Programmcode

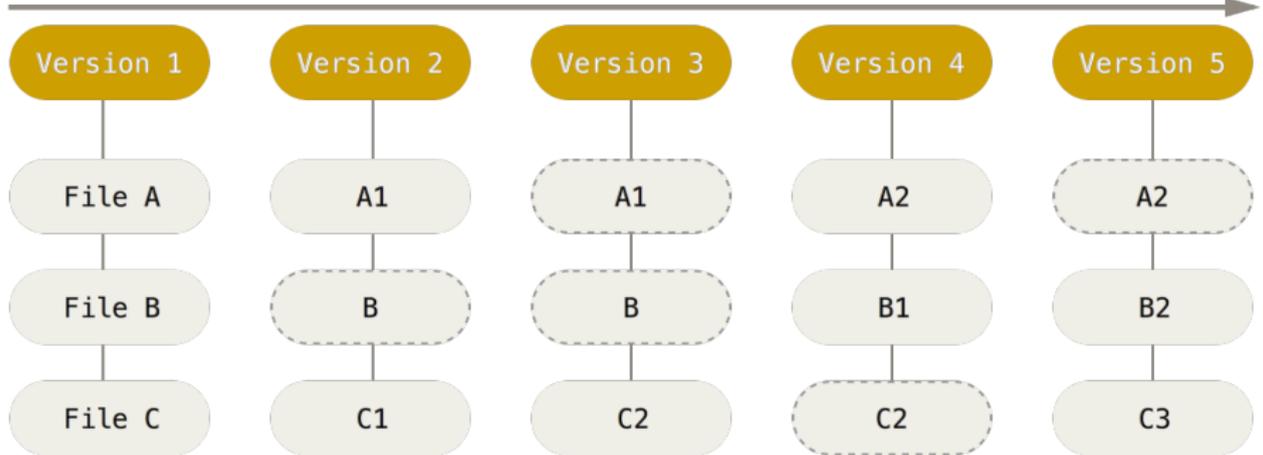
Ich möchte meine Programme *open-source* veröffentlichen. Dabei möchte ich nicht nur den aktuellen Programmcode, sondern auch die ganze Entwicklungsgeschichte für die Nutzer transparent machen. Die Nutzer sollen die Möglichkeit haben, an der weiteren Entwicklung teilzunehmen und eigene Verbesserungen am Programm beizutragen.

- Ein SCM-System verwaltet eine **Datenbank (*Repository*)** in der alle Textdateien (Quelltext) und die Verzeichnisstruktur abgelegt sind.
- Jeder Benutzer der Dateien, bearbeitet diese in einer sogenannten **Arbeitskopie der Dateien**.
- Hat er seine Arbeitskopie geändert, kann er die **Änderungen in die Datenbank** überspielen.
- Die Datenbank **speichert alle Versionen der Dateien** und die Beziehungen zwischen den Versionen.
- **Andere Nutzer erhalten die Änderungen** aus der Datenbank.
- Zusammen mit den Dateiversionen werden **erläuternde Kommentare** gespeichert.
- **Frühere Versionen** können jederzeit wieder hergestellt werden.

So funktionieren SCM-Systeme

Mit *Commits* werden Versionen im SCM-System gespeichert

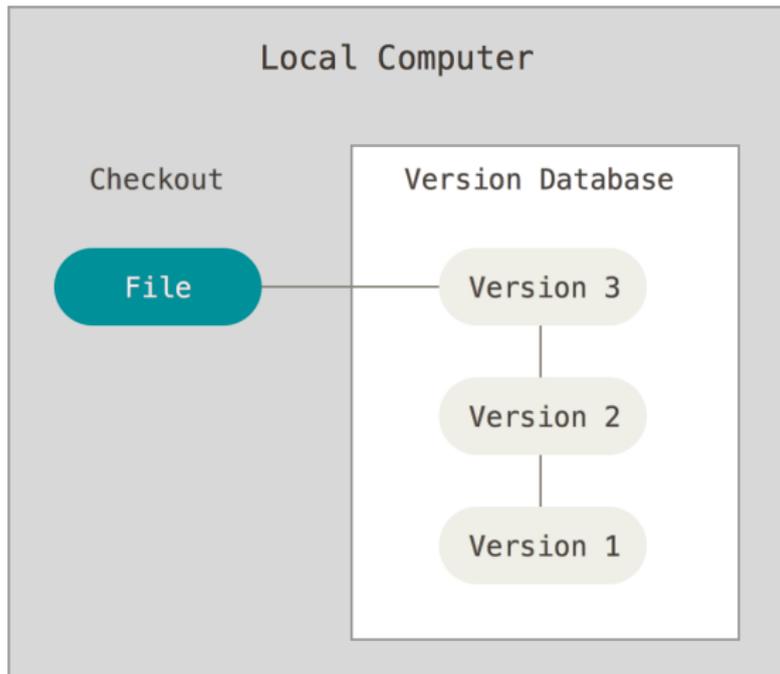
Checkins Over Time



Quelle: <http://git-scm.com/book>

So funktionieren SCM-Systeme

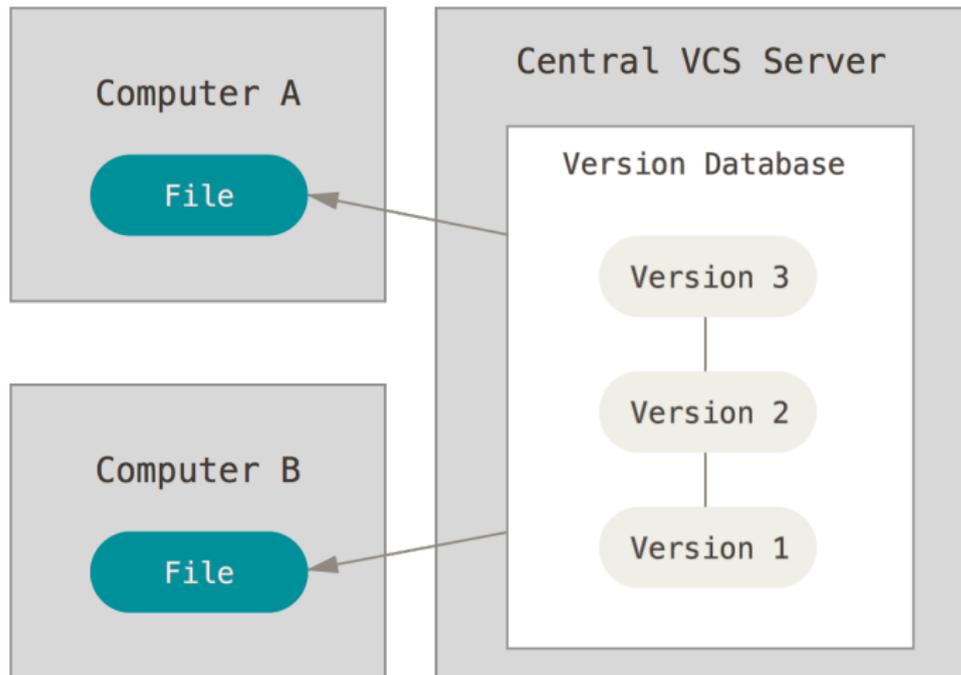
Lokale Datenbank (z.B. [git](#))



Quelle: <http://git-scm.com/book>

So funktionieren SCM-Systeme

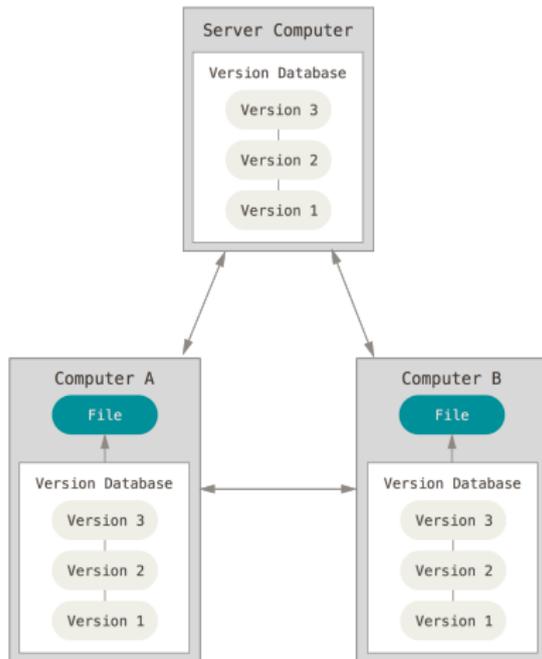
Zentrale Datenbank (z.B. [subversion](#))



Quelle: <http://git-scm.com/book>

So funktionieren SCM-Systeme

Verteilte Datenbanken (z.B. git)



Quelle: <http://git-scm.com/book>

- SCM-Systeme sind auf **alle Textdateien** anwendbar, auch auf \LaTeX -Texte, Dokumentationen oder Steuerdateien für Datenprozessierung.
- SCM-Systeme werden auch zum **Publizieren von Programmcode** verwendet.
- Hinweis: Was ich einspiele landet mit hoher Wahrscheinlichkeit bei vielen anderen Leuten, ob sie das wollen oder nicht und ob ich das will oder nicht.
Daher:
 - **Niemals Passwörter oder ähnlich sensible Informationen** in SCM-Systemen hinterlegen!
 - Nach Möglichkeit **keine großen Binärdateien** in SCM-Systeme einspielen.

So funktionieren SCM-Systeme

Lösungen zu den fünf Problemen

1. Bevor ich nach Hause gehe, spiele ich die Änderungen aus dem Institut auf den Server. Zu Hause mache ich zuerst einen Abgleich.
2. Umfangreiche Änderungen sind kein Problem. Ich kann jederzeit alte Versionen wieder herstellen.
3. Ich muss regelmäßig eine Sicherheitskopie der Datenbank machen (in der nur die kleinen Textdateien sind). Daraus kann ich meine Arbeit jederzeit reproduzieren.
4. Ich muss nur einen Abgleich mit der Datenbank machen, um sicher zu sein, dass ich auf die gleiche Quelltextversion zugreife wie die Kollegen. Der SCM-Client arbeitet dann alle Änderungen in die von mir bereits modifizierten Dateien ein. Wenn meine Änderungen fertig sind, überspiele ich sie in die Datenbank.
5. Ich veröffentliche mein [git-Repository](#) über [gitlab](#) oder [github](#).

So funktionieren SCM-Systeme

Markante Meilensteine aus der Geschichte

- SCCS (1972) Bereits auf den Minicomputern der frühen 70er Jahre (IBM System/370, PDP-11, etc.) wurde das **Source Code Control System** eingeführt.
- RCS (1982) Das **Revision Control System** verwaltet die Versionsgeschichte einzelner Dateien.
- CVS (1990) Das **Concurrent Versions System** ist ein direkter Nachfolger von RCS und kann mehrere Dateien als Teile eines gesamten Projekts behandeln.
- SVN (2000) **Subversion** ist ein Nachfolger von CVS mit ähnlicher Kommandosyntax und deutlich erweiterter Funktionalität. Es kann beispielweise das Verschieben von Dateien in einem Verzeichnisbaum verfolgen.
- git (2005) Weit verbreitetes dezentrales SCM. Ursprünglich entwickelt für den Quellcode des Linux Kernels.

So funktionieren SCM-Systeme

Alternativen

Es gibt zahlreiche Alternativen zu den genannten SCM-Systemen. Dazu lohnt sich ein Blick auf folgende wikipedia-Seiten:

- [List of revision control software](#)
- [Comparison of revision control software](#)
- [Comparison of source code hosting facilities](#)

Dezentrales *Source Code Management* mit `git`

Vorteile des dezentralen SCM mit `git`

- *Commits* an jedem Ort (Zug, Berghütte, Segelschiff, ...)
- Extrem schnell, da kein Serverzugriff erforderlich
- Ich entscheide, welche *commits* für andere sichtbar werden
- Jeder *Clone* ist eine Sicherheitskopie
- Jeder Nutzer darf sofort *Commits* in seinem *Clone* machen

- Extrem flexibler *Branch*-Mechanismus
- Starke graphische Oberfläche (`gitk`, `git gui`)
- Sehr kompakte *Repositories* (effiziente Komprimierung von Textdateien)
- Sehr flexibel; viele *plumbing commands*; nichts ist endgültig
- Ganze *Repositories* können *merged* werden
- Inhärente Integritätsprüfung

Dezentrales *Source Code Management* mit git

Und wo sind die Nachteile?

- Das System erscheint zunächst komplexer
- Unsere [trac/SVN](#)-Server sind dafür nicht nutzbar (Alternative: [gitlab](#) am [SCC](#))

<https://git.scc.kit.edu>

Einführung in git für subversion-Nutzer

Stolpersteine

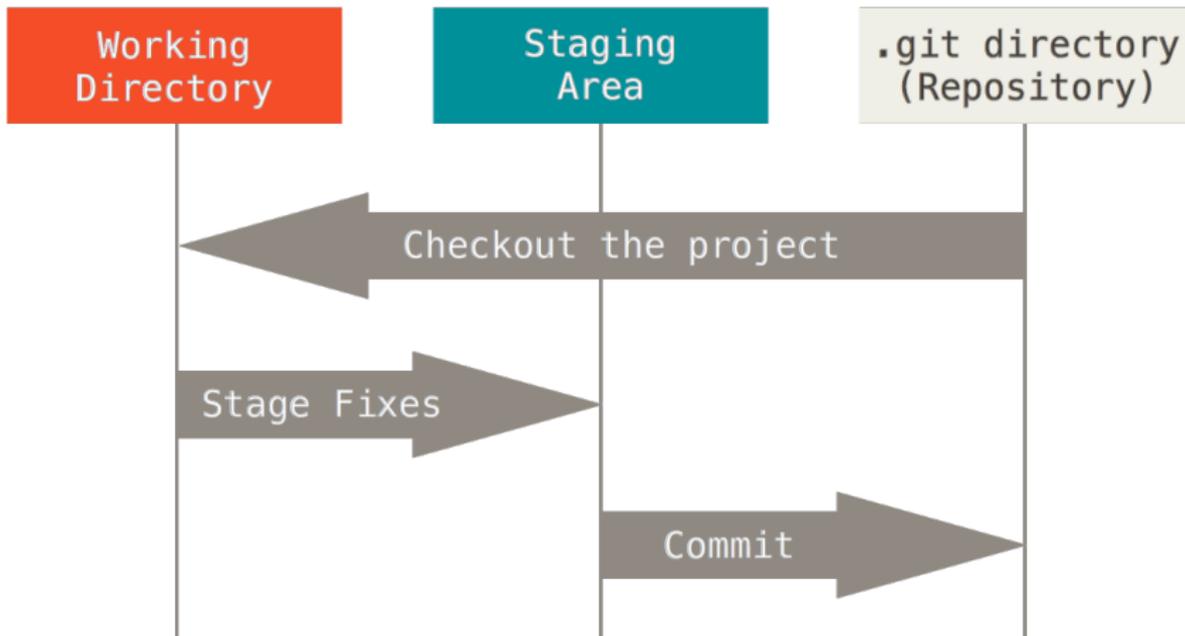
- Gleichnamige Befehle in git und subversion, die aber Unterschiedliches bedeuten:
 - add
 - checkout
 - revert
- update gibt es in git nicht
- *Branches*
 - sind in subversion Unterverzeichnisse
 - sind in git 'Wimpel'

In dieser Hinsicht vermerkt subversion zwei Dimensionen (Verzeichnisstruktur, Commitstruktur).

- git kennt keine *partial checkouts*

Arbeitskopie, Index, Repository

Bereiche in denen Dateien liegen



Quelle: <http://git-scm.com/book>

Identität anlegen, mit der `git` die *commits* kennzeichnet:

```
git config --global user.name "Nyota Uhura"  
git config --global user.email nyota.uhura@starfleet.fop
```

`git config` bietet umfangreiche Möglichkeiten, `git` den persönlichen Bedürfnissen anzupassen.

```
git config --list
```

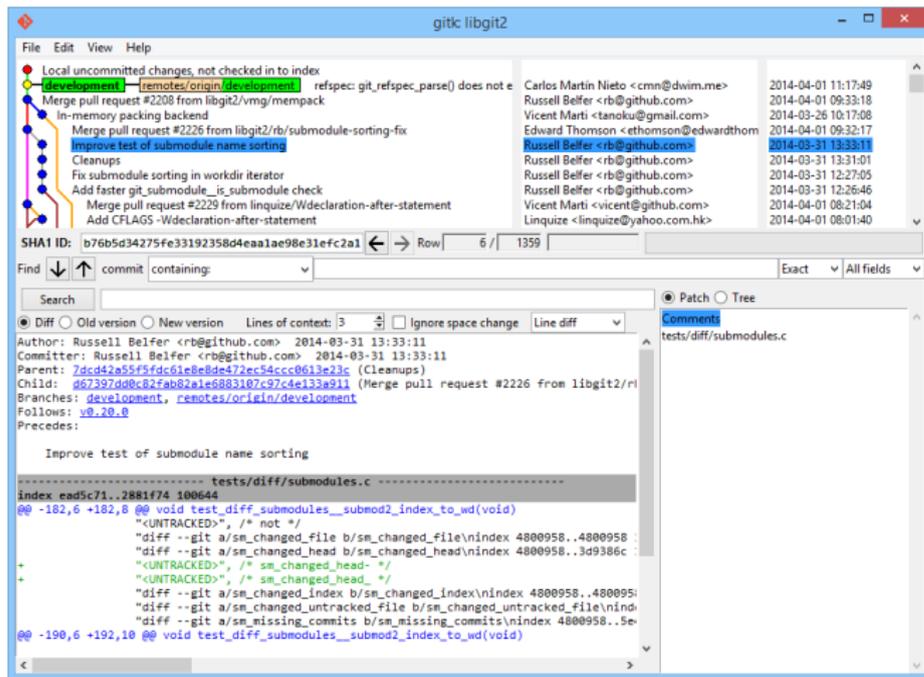
Erste Schritte

git mit fünf Befehlen

- Repository anlegen:
`git init`
- Dateien zum Index (*Staging Area*, *Cache*) hinzufügen:
`git add <Dateiname>`
- *Snapshot* im Index in das Repository aufnehmen:
`git commit`
- Protokoll der *Commits* anzeigen:
`git log`
- Die Frage „was nun?“ stellen:
`git status`

Grafische Navigationshilfen

Das ganze *Repository* auf einen Blick: `gitk --all &`



The screenshot shows the gitk application window titled "gitk libgit2". On the left, a commit history graph shows a sequence of commits, with the current commit highlighted in blue. The commit message for the current commit is "Improve test of submodule name sorting".

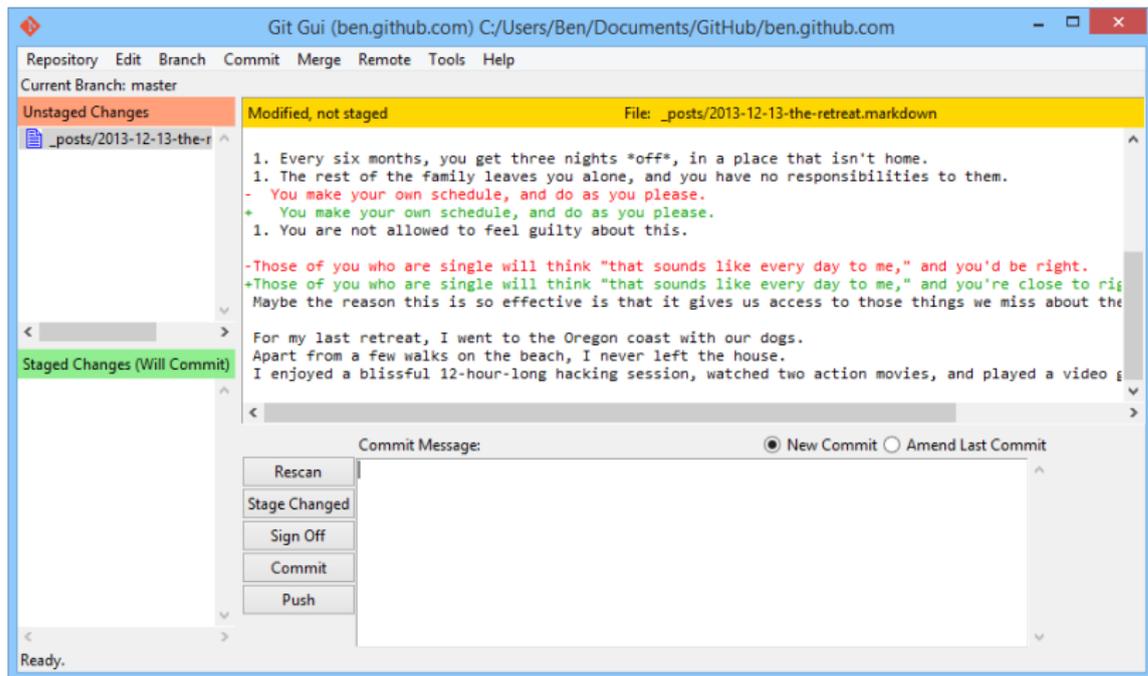
The main area displays the commit details for SHA1 ID: b76b5d34275fe33192358d4eaa1ae98e31efc2a1. The commit was authored by Russell Belfer on 2014-03-31 13:33:11. The commit message is "Improve test of submodule name sorting".

The bottom section shows a diff view of the file "tests/diff/submodules.c". The diff highlights changes in the "index" function, showing that the "sm_changed_index" variable is now used to track changes in the index.

Quelle: <http://git-scm.com/book>

Grafische Navigationshilfen

Commit und blame interaktiv: git gui



The screenshot shows the Git GUI application window. The title bar reads "Git Gui (ben.github.com) C:/Users/Ben/Documents/GitHub/ben.github.com". The menu bar includes "Repository", "Edit Branch", "Commit", "Merge", "Remote", "Tools", and "Help". The current branch is "master".

The interface is divided into several sections:

- Unstaged Changes:** A yellow bar at the top indicates the current file being edited: "_posts/2013-12-13-the-retreat.markdown".
- File Content:** The main area displays the text of the file with changes highlighted in red (deletions) and green (additions). The text includes:
 - 1. Every six months, you get three nights *off*, in a place that isn't home.
 - 1. The rest of the family leaves you alone, and you have no responsibilities to them.
 - You make your own schedule, and do as you please.
 - + You make your own schedule, and do as you please.
 - 1. You are not allowed to feel guilty about this.
 - Those of you who are single will think "that sounds like every day to me," and you'd be right.
 - +Those of you who are single will think "that sounds like every day to me," and you're close to right.
 - Maybe the reason this is so effective is that it gives us access to those things we miss about the
 - For my last retreat, I went to the Oregon coast with our dogs.
 - Apart from a few walks on the beach, I never left the house.
 - I enjoyed a blissful 12-hour-long hacking session, watched two action movies, and played a video g
- Staged Changes (Will Commit):** A green bar at the bottom indicates the content that will be committed.
- Commit Message:** A text area for entering a commit message. The "New Commit" radio button is selected.
- Buttons:** "Rescan", "Stage Changed", "Sign Off", "Commit", and "Push" are visible on the left side.

The status bar at the bottom left shows "Ready."

Quelle: <http://git-scm.com/book>

Grafische Navigationshilfen

Weitere Hilfsmittel

- `git difftool` *GUI* zeigt Dateiunterschiede z. B. mit `meld`
- `git mergetool` *GUI* zum Auflösen von *merge*-Konflikten (z. B. mit `meld`)
- `gitg` *GUI: Repository* sichten und *Commits* interaktiv erstellen
- `qgit` *GUI: Repository* sichten und *Commits* und *Patches* interaktiv erstellen
- `giggle` *GUI*: Interaktive Oberfläche mit Dateieditor; Versionsgeschichte auf Dateien bezogen
- `gitview` *GUI* in `python` auf der Basis von `GTK`
- `tig` *GUI* für die Textkonsole
- ...

Nächste Schritte

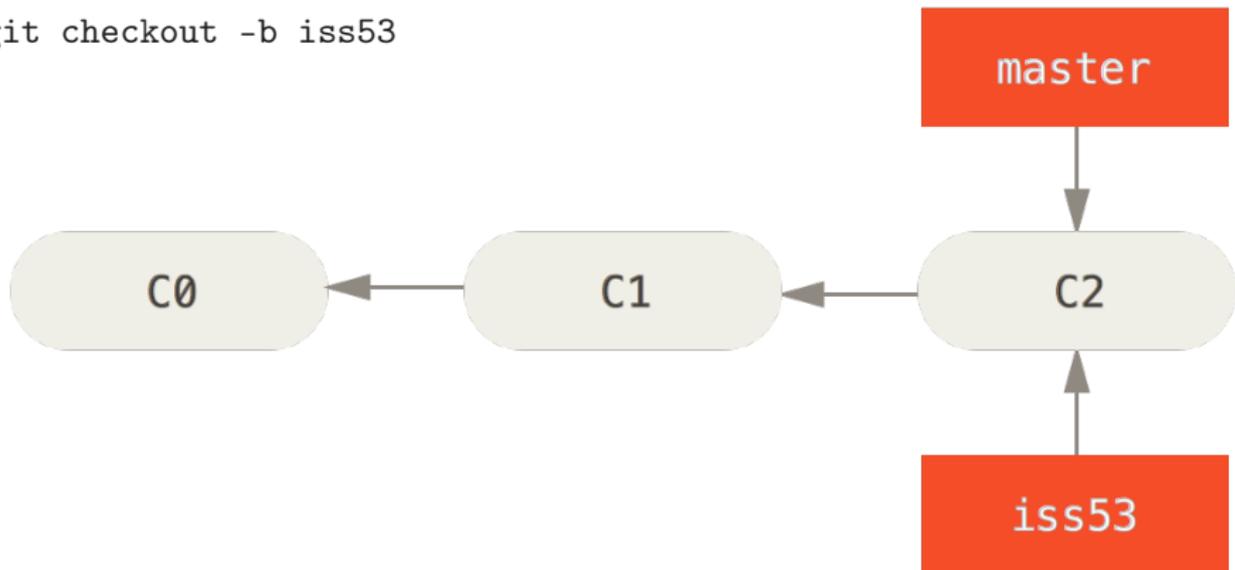
Commits und *Branches*

- Auf einen *Branch* wechseln:
`git checkout <branch>`
- Einen *Branch* anlegen:
`git checkout -b <new branch> <start point>`
- Unterschiede anzeigen:
`git diff <commit> <commit>`
- Zwei Entwicklungslinien zusammenführen:
`git merge <commit>`

Commits und Branches

Einen *Branch* Anlegen, um *Issue #53* zu bearbeiten

```
git checkout -b iss53
```

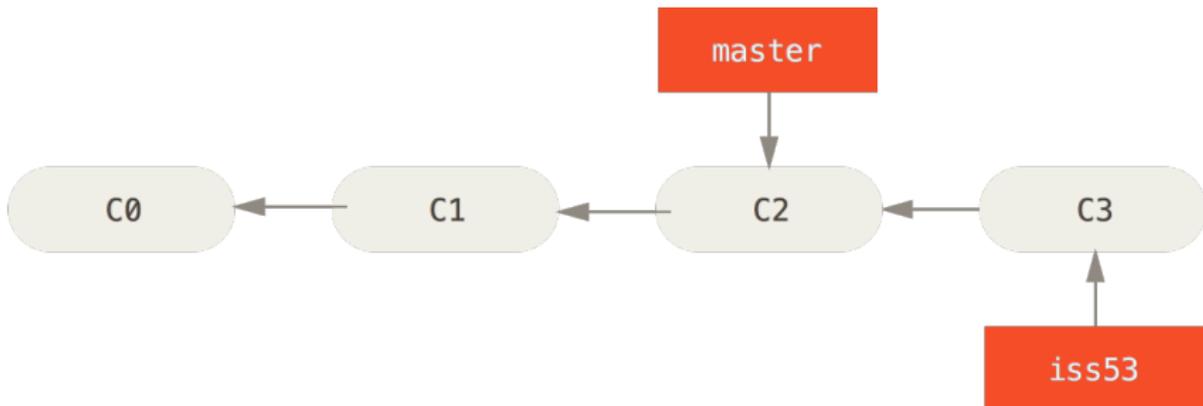


Quelle: <http://git-scm.com/book>

Commits und Branches

Issue #53 bearbeiten

```
edit  
git commit
```

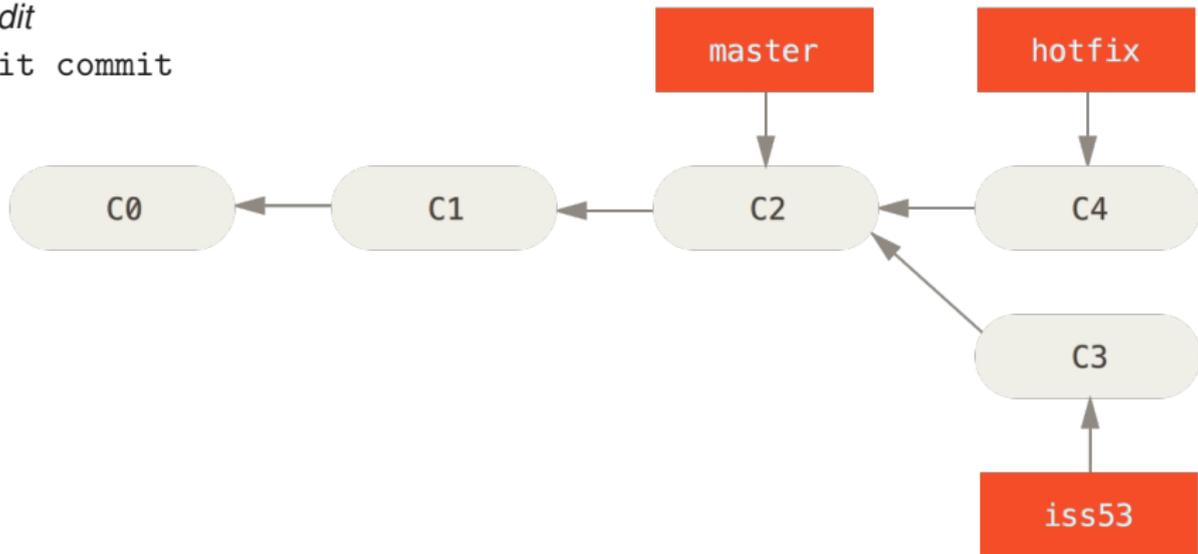


Quelle: <http://git-scm.com/book>

Commits und Branches

Rasch einen Fehler beheben (*hotfix*)

```
git checkout -b hotfix master  
edit  
git commit
```

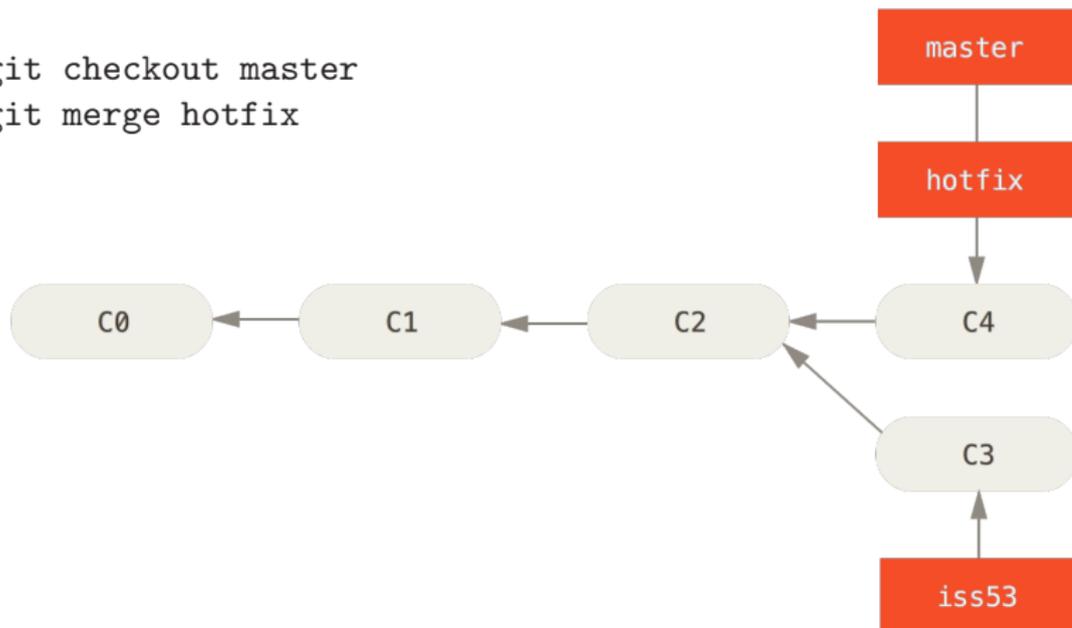


Quelle: <http://git-scm.com/book>

Commits und Branches

Fehlerkorrektur in *master* aufnehmen

```
git checkout master  
git merge hotfix
```

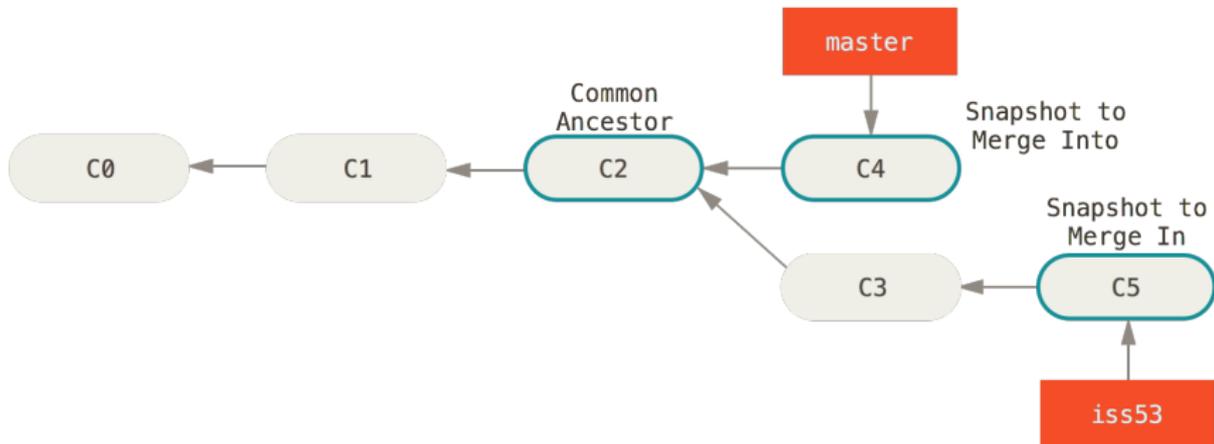


Quelle: <http://git-scm.com/book>

Commits und Branches

Issue #53 weiter bearbeiten

```
git branch -d hotfix  
git checkout iss53  
edit  
git commit
```

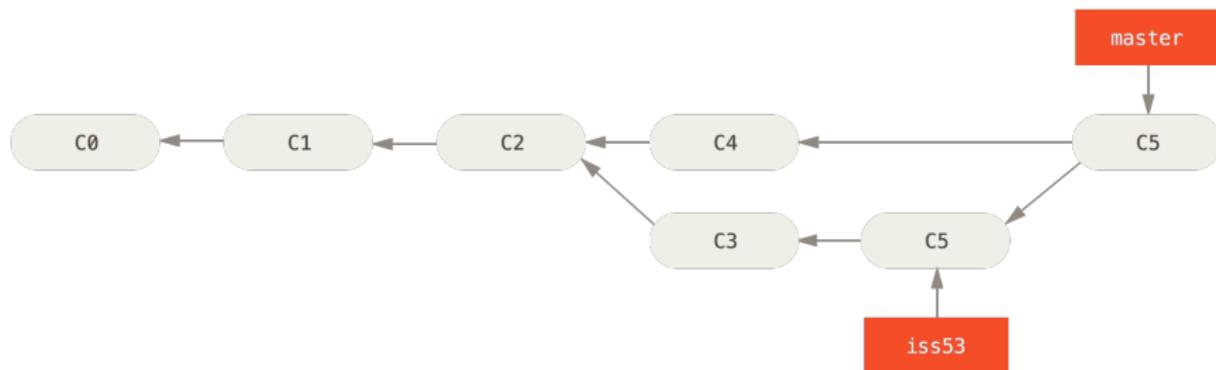


Quelle: <http://git-scm.com/book>

Commits und Branches

Änderungen aus der Bearbeitung von *Issue #53* in *master* aufnehmen (*mergen*)

```
git checkout master
git merge iss53
```



Quelle: <http://git-scm.com/book>

Auf publizierte Repositories zugreifen

- Einen *Clone* anlegen:
`git clone <URL>`
- Ein publiziertes *remote Repository* eintragen:
`git remote add <name> <URL>`
- Neue Inhalte abrufen:
`git fetch`
- Neue Inhalte in *Tracking Branch* übernehmen:
`git pull`

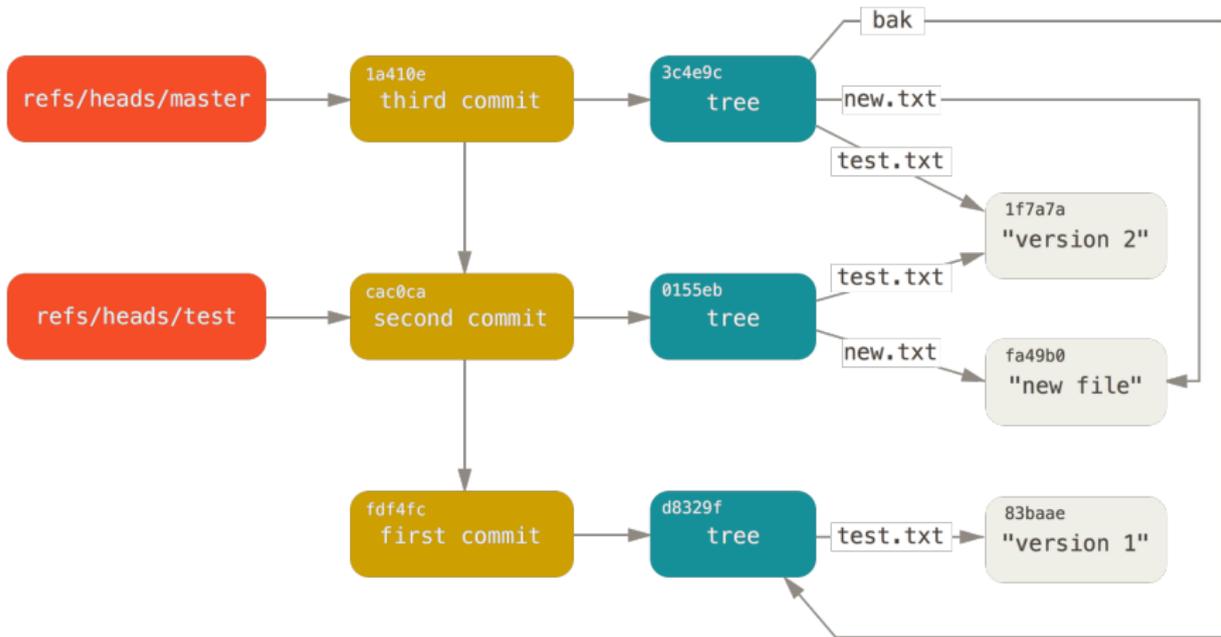
Mit der *Pickaxe* im *Repository* stöbern:

- `git log -i -G 'umerical.*ecipe' --pickaxe-all`
- `git grep -n --color -i -G 'umerical.*ecipe' $(git log -i -G 'umerical.*ecipe' --pickaxe-all --format='%H%n')`

Falls eine Schreibberechtigung für das *Remote Repository* verfügbar ist:

- Eigene Inhalte in das *Remote Repository* übertragen:
`git push`

SHA1 Hash



Quelle: <http://git-scm.com/book>

- `http://git-scm.com/book`
- `https://git.wiki.kernel.org`
- `man git`
- `man gittutorial`
- `man gittutorial-2`
- `man gitrevisions`
- `git help`
- `git help <command>`
- `/usr/share/doc/packages/git/everyday.html`
- `/usr/share/doc/packages/git/user-manual.html`

- git-Clone des DENISE-Teils des trac/SVN-Projektes FWI_elastic:

```
git clone /data14tf2/DENISE.git  
git clone /data14/tforb/distrib/DENISE.git
```
- Daniel Köhns Version von DENISE:

```
git clone  
https://github.com/daniel-koehn/DENISE-Black-Edition.git
```
- Das Seitosh-Repository:

```
firefox https://git.scc.kit.edu/Seitosh/Seitosh  
git clone https://git.scc.kit.edu/Seitosh/Seitosh.git  
git clone git@git.scc.kit.edu:Seitosh/Seitosh
```
- Der Quellcode von git:

```
git clone https://github.com/git/git
```
- Das ObsPy-Paket:

```
git clone https://github.com/obspy/obspy.git
```



Quelle: <http://git-scm.com/downloads/logos>

Die Abbildungen stammen aus

Scott Chacon und Ben Straub, 2014. Pro Git. 2nd ed. Apress.

Veröffentlicht auf <https://github.com/progit/progit2>

unter der Lizenz

Creative Commons Attribution Non Commercial Share Alike 3.0

(<http://creativecommons.org/licenses/by-nc-sa/3.0/>)