

Eine Einführung in Versionverwaltungssysteme

Wissenswertes für GIT-Anwender

bernd.wunder@leb.eei.uni-erlangen.de

Lehrstuhl für Elektronische Bauelemente
Friedrich-Alexander-Universität Erlangen-Nürnberg

13. Februar 2012



Inhaltsverzeichnis

- 1 Einführung
- 2 Grundlagen
- 3 Tools
- 4 Befehle
- 5 Sonstiges
- 6 Referenzen

Einführung

- Was ist ein Versionsverwaltungssystem?
- Warum Versionsverwaltung?
- Übersicht über VCS
- Warum Git?
- Grundbegriffe

Was ist ein Versionsverwaltungssystem

*Eine Versionsverwaltung ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird. Alle Versionen werden in einem Archiv mit Zeitstempel und Benutzererkennung gesichert und können später wiederhergestellt werden.*¹

Aufgaben

- Protokollierungen der Änderungen
- Wiederherstellung von alten Ständen
- Archivierung der einzelnen Stände eines Projektes
- Kooperative Entwicklung (Entwicklungsteams)
- gleichzeitige Entwicklung mehrerer Entwicklungszweige (branches)

¹Quelle: Wikipedia

Warum Versionsverwaltung?



- 1 viele gleichzeitige laufende Projekte / Features (z.B. Sicherheitsupdates, neue Funktionstests, verschiedene Versionen)
- 2 Um des Chaos Herr zu werden

Warum Versionsverwaltung?



- 1 viele gleichzeitige laufende Projekte / Features (z.B. Sicherheitsupdates, neue Funktionstests, verschiedene Versionen)
- 2 Um des Chaos Herr zu werden
- 3 Überblick über die gesamte Entwicklung behalten
- 4 kolaboratives Arbeiten in einem Team

Übersicht über Versionssverwaltungssysteme

engl. Bezeichnungen: *Version Control System (VCS)*, *Software Configuration Management (SCM)*, *Revision Control System (RCS)*

unvollständige Übersicht einiger VCSe:

Revision Control System (RCS)

Entwicklung:	1980 bis 2004
Einteilung:	zentrales, dateibasiertes VCS
Probleme:	Binärdateien, Verzeichnisse, locks, merge, keine Atomic commits, keine Metadaten, umbenennen
Lizenz:	GPL2
Betriebssysteme:	UNIX, WIN95

Entwickelt für die Versionsverwaltung von Text-Dateien auf einem Computer! RCS ist im Wesentlichen mit dem ersten VCS, dem *Source Code Control System (SCCS)*, vergleichbar.

Concurrent Versions System (CVS)

basierend auf:	RCP (nutzt gleiches Speicherformat)
Entwicklung:	1989 bis 2008
Einteilung:	zentrales VCS
Probleme:	Binärdateien, Verzeichnisse, locks, merge, keine Atomic commits, keine Metadaten, umbenennen
Lizenz:	GPL2
Betriebssysteme:	UNIX, WINDOWS, Mac OS X

CVS wird nicht mehr aktiv weiterentwickelt. Die offizielle Webseite wird nicht mehr weiter betreut. ^a

^aQuelle: Wikipedia

Subversion (SVN)

basierend auf:	CVS (Nachfolger)
Entwicklung:	seit 2000
	Ziele CVS Probleme (siehe oben) zu beseitigen
Einteilung:	zentrales VCS
Probleme:	Binärdateien, Verzeichnisse, locks, merge
Lizenz:	Apache License
Betriebssysteme:	UNIX, WINDOWS, Mac OS X

Subversion versteht sich als Weiterentwicklung von CVS und entstand als Reaktion auf weit verbreitete Kritik an CVS. In der Bedienung der Kommandozeilenversion ist es sehr ähnlich gehalten.^a

^aQuelle: Wikipedia

Git

Die Entwicklung von Git wurde im April 2005 von Linus Torvalds begonnen, um das bis dahin verwendete Versionskontrollsystem BitKeeper zu ersetzen, das durch eine Lizenzänderung vielen Entwicklern den Zugang verwehrte. Die erste Version erschien bereits wenige Tage nach der Ankündigung. Derzeitiger Maintainer von Git ist Junio Hamano.

- 1 Nicht-lineare Entwicklung
- 2 Kein zentraler Server
- 3 Datentransfer zwischen Repositories
- 4 Kryptographische Sicherheit der Projektgeschichte
- 5 Interoperabilität
- 6 Web-Interface



Linus Torvalds ist Initiator von Git und des
Kernels Linux

Vergleich² einiger Versionsverwaltungssysteme

Vergleich | CVS,SVN,GIT,Mercurial

Operation	CVS	SVN	GIT	Mercurial
Atomic Commits:	Nein	Ja	Ja	Ja
Umbenennen/Verschieben von Dateien und Verzeichnissen:	Nein	Ja	Ja	Ja
Intelligente Merges nach Umbenungen/Verschiebungen:	Nein	Nein	Nein	Ja
Verzeichnisse/Dateien mit History im Repository kopieren:	Nein	Ja	Nein	Ja
Remote Kopie in lokales Verzeichnis:	Nein	Nein	Ja	Ja
Änderungen an andere Repository weitergeben:	Nein	Nein	Ja	Ja
Changesets Support	Nein	teilw.	Ja	Ja

²Quelle: Version Control System Comparison

Warum Git?

- Speed** Schnellstes mir bekannte Versionsverwaltungssystem
- Branch** sehr einfach Zweige (Branch) zu erstellen und zusammenzuführen!
- Lokal** alle Informationen sind lokal gespeichert, keine Netzwerkverbindung notwendig!
- Speicher** sehr geringer Speicherverbrauch durch Kompression

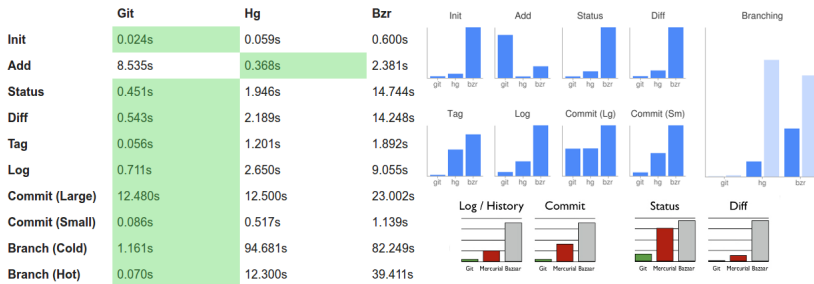


Abbildung: Geschwindigkeitsvergleich von Git, Mercurial und Baszar

Warum Git?

Protokoll http, https, ssh, git, ...

Kooperativ gute Zusammenarbeit mit anderen Versionsverwaltungssystemen. Z.B. mit SVN oder CVS

Stabil sehr aktives Projekt mit vielen hundert Entwicklern

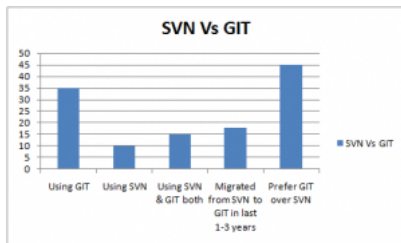


Abbildung: Geschwindigkeitsvergleich von Git, Mercurial und Baszar

Warum Git?

tested Protokiv im Einsatz bei Linux-Kernel (>1000 Entwickler)
me ... i know about it and i *really like it* ;-)

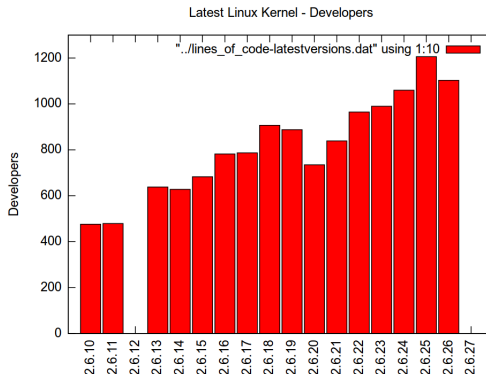


Abbildung: Übersicht³ über die Anzahl der Linux Kernel Entwickler

³Quelle: Michael Schöntzer

Wer nutzt Git?

Viele Open Source Projekte setzen bereits Git ein:
(umfangreiche Liste auf git.wiki.kernel.org)

- Linux Kernel
- Git
- Android (Google's Handy OS)
- CakePHP (PHP Framework)
- Debian - Linux Distribution
- Drupal - CMS System
- Typo3 - CMS System
- Perl
- Ruby on Rails
- VLC
- PostgreSQL
- KDE
- Fluxbox
- X.Org
- GCC (Gnu Compiler Collection)
- JQuery (JavaScript library)
- Qt (Cross-platform graphic toolkit)
- Eclipse
- Gnome
- ...

Grundbegriffe

Repository

Datenbank in dem jeder Dateistand eines Projektes über die Zeit hinweg gespeichert ist.

Working Tree

Arbeitsverzeichnis in dem die Modifikationen durchgeführt werden.

Commit

beinhaltet alle Veränderungen bzw. spiegelt den aktuellen Zustand der in das VCS aufgenommen werden soll wieder. Enthält neben den Änderungen zusätzliche Metadaten (Commit Message, Autor, Datum, Signatur, ...)

HEAD

zeigt auf die neueste Version *Kopf* im aktuellen Zweig (Branch)

Achtung: Unterschiede zwischen GIT und SVN, CVS

Secure Hash Algorithm (SHA-1)

ist eine eindeutige, 160 Bit (40 hexadezimale Zeichen) lange Prüfsumme für beliebige digitale Informationen.

Beispiel:

mit dem GNU/Linux Programm *sha1sum* wird die Prüfsumme für den Text *"Isabella und Lilly Wunder"* berechnet werden:

```
bernd@Power:~$ echo "Isabella und Lilly Wunder" | sha1sum
25989877d4888b5a4f41850069a7c53ac2c8e3ff -
```

Grundbegriffe (GIT)

Branch

bezeichnet einen parallelen Entwicklungszweig. Der Hauptzweig in einem Versionsverwaltungssystem hat meistens einen speziellen Namen. (z.B in SVN->*trunk* und in GIT->*master*)

Objektmodell

Git-Objekte (blob, tree, commit, tag) sind in einer Objektdatenbank gespeichert und über SHA1-Summen identifizierbar. Die History eines Repository lässt sich als Graph von Objekten modellieren.

Grundbegriffe (GIT)

Index

Der *Index* ist ein lokaler Zwischenspeicher. Alle Änderungen werden zuerst in den index geschrieben. Anschließend wird der Index durch einen *commit* in das Repository eingcheckedt.

Clone

ist eine Kopie eines Repositories mit der gesamten History der Entwicklung.

Tag

Ein Tag ist ein symbolischer Name für schwer zu merkende SHA-1 Summen. So können spezielle *Commit* einen Namen, also ein *Tag* erhalten.

Grundlagen und die Git Konzepte

- History
- Lokale, zentrale und verteilte Versionsverwaltungssysteme
- Zentral VS Dezentral
- Der Index
- Objektmodel in Git

Lokale, zentrale und verteilte Versionsverwaltungssysteme

lokal rcs: einfache Ergänzung des Dateisystems.

zentral Subversion (SVN), CVS: Kommunikation nur über zentralen Server.

verteilt Git, Baszar, Merkurial: Kommunikation beliebig.

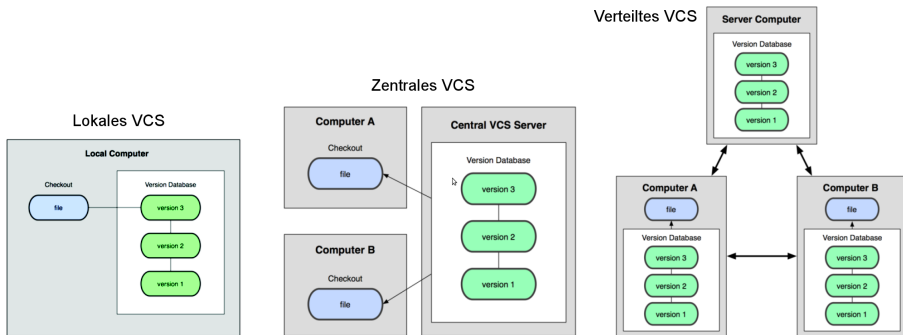


Abbildung: Unterschiedliche Arten von Versionsverwaltungssystemen ⁴

⁴Quelle: Pro Git - Scott Chacon

Die Kathedrale und der Basar

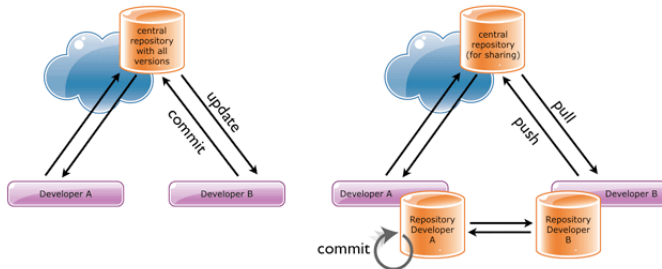
Der legendäre Artikel *The Cathedral and the Bazaar* (1997) von Eric S. Raymond vergleicht zwei verschiedene Entwicklungskonzepte für Software.

- **Basar:** Ein großer Marktplatz voller Entwickler, keine vorwiegende Struktur erkennbar.
- **Kathedrale:** Entwickler folgen der geistigen Führung in einer sehr hierarchischen Struktur.
- Linus Torvalds nutzte als erster die Basar Struktur zur Entwicklung des Linux Kernels
- Ein Beispiel für eine Kathedrale ist die Struktur der Windows Entwicklung
- es gibt auch gemischte Strukturen , z.B. wird der MATLAB Core in einer Kathedrale und die Toolboxen auf dem Basar entwickelt
- ESR war über den extrem schnellen Erfolg des Basars verwundert. Lag der Erfolg an der Struktur oder Linus, ...?
- ESR zeigt anhand von *fetchmail* (früher: *popclient*) die Vorteile von einem Basar auf, siehe Artikel!



Abbildung: Eric S. Raymond (ESR) ist ein US-amerikanischer Autor und Programmierer in der Open-Source-Szene

Zentral VS Dezentral



Zentrales VCS

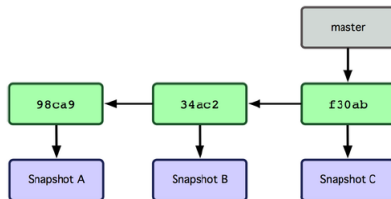
- Performance und Skalierbarkeit hängt von vom Server ab
- Hohe Serverbelastung
- Single Point of Failure
- Backups sehr Wichtig!
- Hoher Administrationsaufwand
- Verbindung zu Server notwendig um VCS nutzen zu können.

Dezentrales VCS

- deutlich geringer Serverbelastung beim *zentralen* Speicher
- jeder Entwickler hat die Komplette Versionsgeschichte local
- Administrationsaufwand je nach verwendeter Architektur deutlich geringer
- **keine** Verbindung zu Server notwendig um VCS nutzen zu können.
- kann z.B. mit der Portablen Version von Git sogar ohne Admin-Rechte installiert und verwendet werden.

History

Nach jeder abgeschlossenen Änderung werden diese in das VCS übertragen. Das VCS verbindet diese Commits miteinander in Form eines Graphen. Es wird ein gerichteter azyklischer Graph aufgebaut (Directed Acyclic Graph, DAG).



Graph

Ein *Graph* besteht aus den beiden Kernelementen *Knoten* und *Kante*. Ein Commit, Tag, Referenz oder andere Objekte werden immer durch einen Knoten im Graphen dargestellt. Die Kante wird durch einen Verweis auf ein oder mehrere Eltern-Objekt(e) dargestellt.

In Git hat jedes Objekt einen eindeutigen SHA1-Prüfsummen. Daraus ergibt sich eine kryptographisch gesicherte Integrität des Repositorys.

History als Graph: Von Branches, Merges und Tags

Branch entsteht wenn mehrere Versionen von einem Elternobjekt abhängen.

Tag ist der Name für eine bestimmte Version.

Merge führt den Parallelen Zweig in den Hauptzweig zurück.

master ist in Git der Name für den Hauptentwicklungszweig.

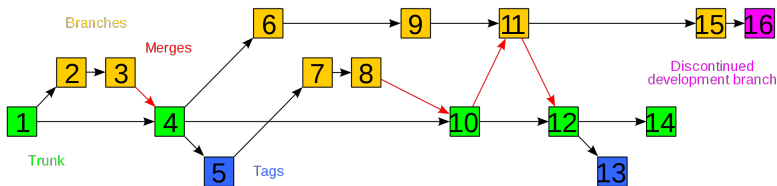


Abbildung: Der Graph bildet das gesamte Repository (Branches, Merges und Tags) ab. In Subversion: master = trunk.

Objekte in Git

Blob Die eigentliche Datei, SHA1-Wert und einige Metadaten.

Tree ist eine Sammlung von Blobs. Ein Tree ist damit äquivalent zu einem Verzeichnisordner

Commit speichert die Commitdaten, User, Datum, Beschreibung und verweise auf Tree, Blob und Tag ab.

Tag verbindet die SHA1-Summe eines anderen Objektes mit einem beliebigen Namen.

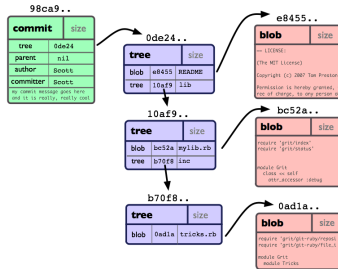


Abbildung: Die verschiedenen Objekte (Commit, Tree und Blob) in Git

GUI-Tools für Git

gitk, git-gui, meld, TortoiseGit

Repository Tool: *gitk*

gitk

In Tcl programmiertes grafisches Frontend zur Anzeige des Repositories. Ist im Git Standard Umfang enthalten und somit **immer** vorhanden. Ermöglicht einen schnellen Überblick über die History, Commits, Diffs, Tags und die Struktur des Repositories.

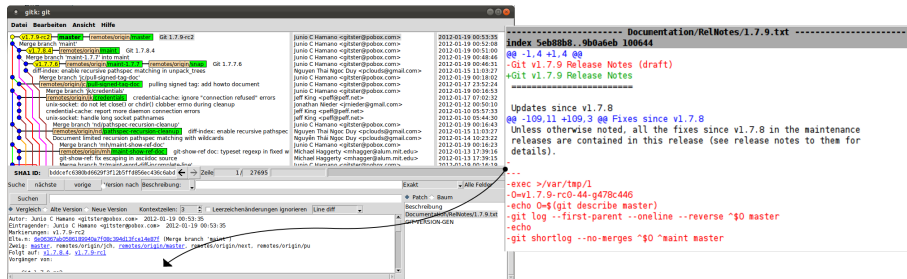


Abbildung: *gitk*: einfach, übersichtlich und immer da!

Commit Tool: *git-gui*

git-gui

Einfaches grafisches Programm um Änderungen bereitzustellen und einen Commits zu erstellen.

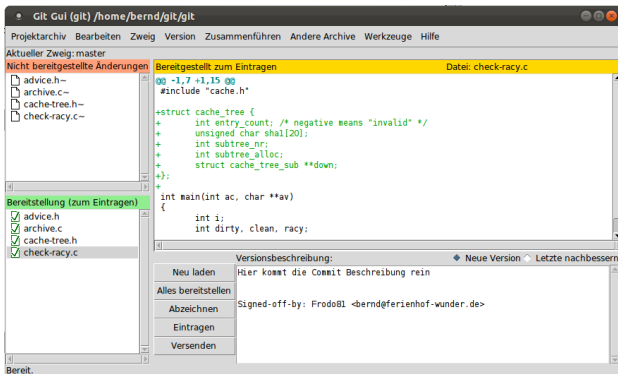


Abbildung: *git-gui*: Frontend für die Erstellung eines Commits. Ist wie *gitk* im Standardumfang enthalten.

Diff-Tool: meld

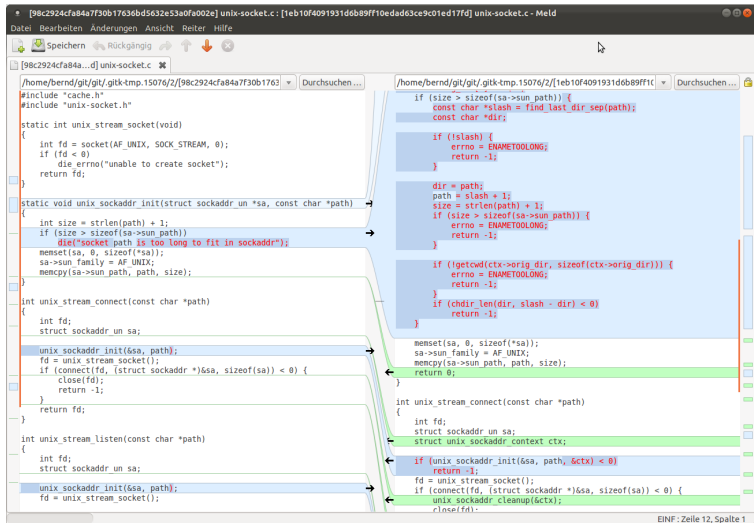


Abbildung: *meld*: einfach, übersichtliches diff-Tool

TortoiseGit

TortoiseGit ist ein kostenloser und freier Windows Client für Git.

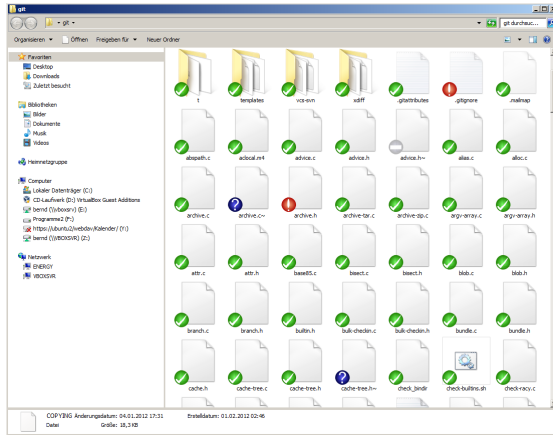


Abbildung: *tortoiseGit*: Git Integration in den Windows Explorer

TortoiseGit 2

GIT-Befehle in das Kontext Menü im Windows Explorer integriert:

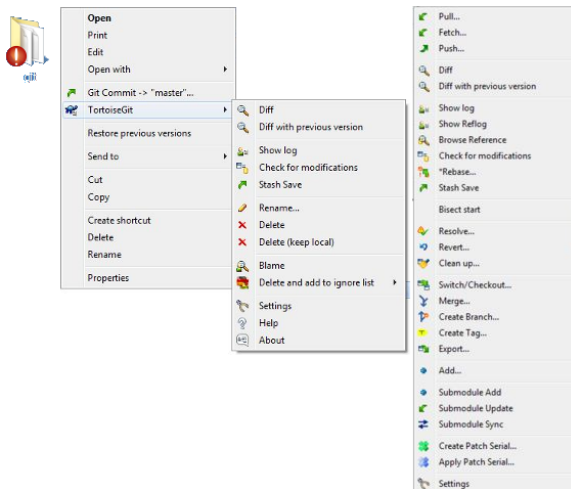


Abbildung: Git Befehle im Kontextmenü des Explorers. Es werden nur die nutzbaren Befehle angezeigt.

Als erstes sollte man seine Benutzerdaten eintragen:

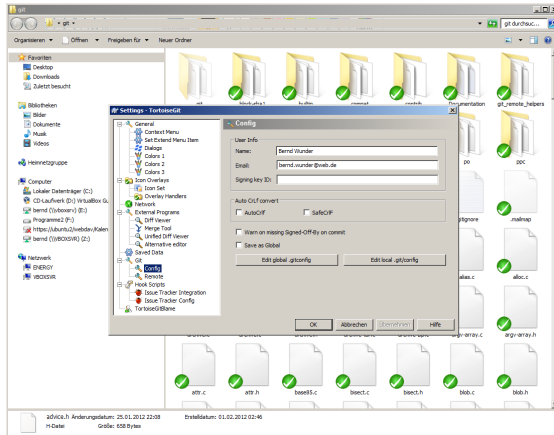


Abbildung: Unter **Settings** kann man sollte man Benutzer und Email einstellen!

TortoiseGit 4

Mit den **Commit...** Button kann man ganz einfach den aktuellen Stand sichern:

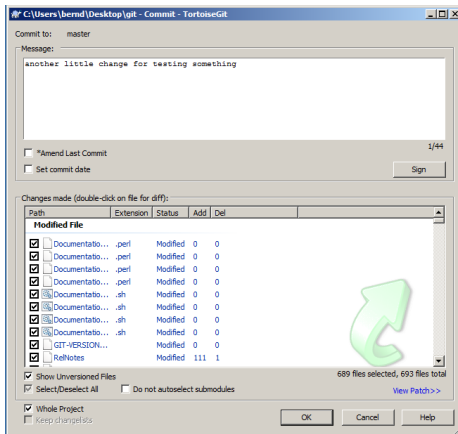
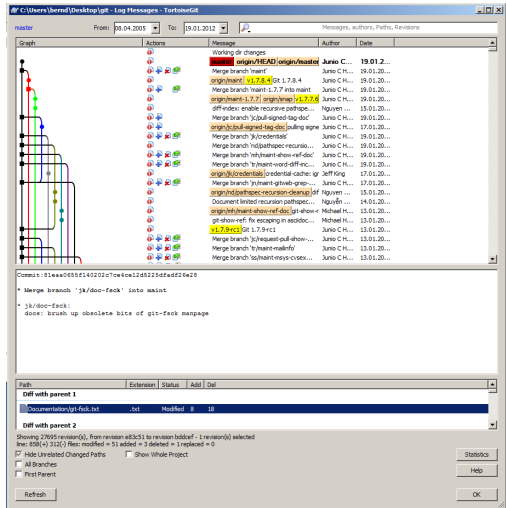


Abbildung: TortoiseGit bietet die Möglichkeit nur die ausgewählten Dateien in einen Commit zusammenzufassen!

TortoiseGit 5

Merkmale von TortoiseGit:

- 1 Intuitiv und schnell zu erlernen
- 2 gute Integration in Windows durch Overlays
- 3 add, commit und push mit einem Dialog
- 4 Unterstützt gängige Speicherorte, u.a. Netzlaufwerke, GitHub, ...



Befehle

Getting Started

Als erstes sollte man Git seine Identität bekannt machen:

```
git config --global user.name "Bernd Wunder"  
git config --global user.email bernd.wunder@leb...
```

neues Repository im aktuellen Verzeichnis anlegen:

```
git init
```

eine Datei dem Repository hinzufügen:

```
git add /Pfad/Zur/Datei
```

alle Dateien im aktuellen Verzeichnis dem aktuellen Repository hinzufügen:

```
git add .
```

einen Commit ausführen:

```
git commit -m "Initial Commit"
```

Befehle 2

Remotes

Unter einem Remote versteht man eine entfernte Quelle. Der **git remote** Befehl dient zum Verwalten der Remotes:

```
bernd@power:~$ git remote  
origin  
power
```

oder ausführlicher mit:

```
bernd@power:~$ git remote -v  
origin    /media/Transcend/Versionsverwaltung_mit_GIT/ (fetch)  
origin    /media/Transcend/Versionsverwaltung_mit_GIT/ (push)
```

einen neuen Alias auf einen entfernten Remote hinzufügen:

```
git remote add newName https://www.weitWeg.de/Repository.git
```

Befehle 3

branch

Für die Erstellung eines neuen Zweiges (branch):

```
git branch lilly
```

Eine ausführliche Übersicht über die vorhandenen Branches:

```
bernd@Power:~$ git branch -v
* lilly 98a74d9 Some further commands
master e2694e7 Some explanation about commands
```

Umbenennen des aktuellen Zweiges nach isabella:

```
git branch -m isabella
```

In einen anderen Zweig wechseln und auschecken (hier in master):

```
git checkout master
```

Sonstiges

Protokolle, .gitignore, Kooperation mit anderen VCS

Protokolle

Zwischen den Repositories können Daten mit einer Reihe unterschiedlicher Protokolle ausgetauscht werden:

- http (Webserver)
- https (Webserver, verschlüsselt)
- ftp (Webserver)
- ssh (Secure Shell, verschlüsselt, admins)
- rsync (GNU/Linux, Synchronisationsprotokoll mit Delta-Kodierung)
- git (git-Protokoll, gepackt)
- email (Patches via Email, Mailing-Liste)

Protokolle - Beispiele

clone-Befehl

Um ein entferntes Repository zu klonen:

```
git clone <remote> <local>
```

git-Quellcode von github.com klonen

über das git-Protokoll von github.com kopieren:

```
git clone git://github.com/gitster/git.git git
```

und über https:

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

oder von einer lokalen Quelle:

```
git clone /home/julia/git git
```

.gitignore

Um bestimmte Dateien oder Muster nicht unter Versionsverwaltung zu stellen kann man die **.gitignore** Datei verwenden. Alle darin enthaltenen Dateien oder Muster werden ignoriert.

In der Regel werden bei einem Buildprozeß Hilfsdateien angelegt die nur für die Tools notwendig sind. Diese Dateien möchte man in der Regeln nicht versionieren. Auch Projektdaten und Konfigurationsdateien von der Entwicklungsumgebung haben in der Versionsverwaltung nichts zu suchen, da diese sich mit unterschiedlichen Benutzern oder Programmen ändern.

Die **.gitignore** Datei kann sich selbst im entsprechenden Projektordner befinden und unter Versionsverwaltung gestellt werden. Daneben kann man zusätzlich eine eigene Datei die nicht im unter Versionsverwaltung steht und sich sinnvollerweise im eigenen Home Verzeichnis befindet mit einbinden:

```
git config --global core.excludesfile ~/.gitignore
```

.gitignore 2

.gitignore

Eine **.gitignore** Datei sieht nun z.B. wie folgt aus:

```
# do not versioning all LaTeX files!  
*.log  
*.out  
# ...  
  
# Verzeichnis komplett ignorieren  
~/temp/  
  
# ! invertiert den Wunsch, so wird die Datei  
# ImportendLogging.log doch von git verwaltet,  
# obwohl obiges Muster *.log zutrifft.  
!ImportendLogging.log
```

Kooperation mit anderen VCS

Git bietet sehr gute Kooperation mit anderen Systemen wie Basar, Mercurial, SVN, CVS

- git-svn -> besser wenn Hauptrepository in GIT
- subgit (Subversion (SVN) + Git) -> besser wenn Hauptrepository in SVN
- git-cvs
- hg-git
- git-hg
- git-bzr
- ...

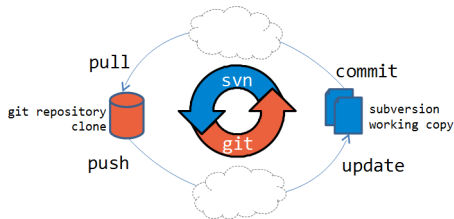


Abbildung: Git arbeitet gut mit anderen VCS zusammen, wie z.B. mit Subversion (SVN)

Submodules

Unter einem Submodul versteht man ein Repository, das sich in einem Unterordner eines anderen Repositories befindet. Dies bietet sich z.B. an um eine eigene Bibliothek auf eine *intelligente* Art einzubinden.

Bindet man eine externe Bibliothek ein und führt Anpassungen durch, können folgende Probleme auftreten:

- Einbinden einer statischen festen Version: Durch die Anpassungen wird ein folgen des Mainstreams sehr aufwendig.
- Bibliothek als gegeben voraussetzen: Anpassungen an der Bibliothek sind damit ausgeschlossen.

Durch die Verwendung von Submodules in Git kann man die beiden beschriebenen Probleme umgehen.

Quellen & Literatur



Bazaar vs Git



Versionsverwaltung



Pro Git - Scott Chacon



Version Control System Comparison



RCS HowTo



Why Switch to Bazaar?



Bazaar



EGit User Guide

Quellen & Literatur 2



Git



Github



The Git Community Book



Git Projekt Page



Git - SVN Crash Course



Warum Git besser als X ist



SVN und GIT im Vergleich



Renaming is the killer app of distributed version control