



Fallstudie Entwicklungswerkzeuge

Ausarbeitung

über das Thema

GIT Versionsverwaltungssystem

Autor: Vedad Hamamdžić
vhamamdžić@stud.hs-heilbronn.de

Prüfer: Paul Lajer

Abgabedatum: 13.01.2015

II Inhaltsverzeichnis

II	Inhaltsverzeichnis	I
II	Abbildungsverzeichnis	II
III	Tabellenverzeichnis	III
IV	Listing-Verzeichnis	IV
V	Abkürzungsverzeichnis	V
1	GIT	1
1.1	Was ist ein Versionskontrollsystem	1
1.1.1	Lokale Versionskontrollsysteme	1
1.1.2	Zentralisierte Versionskontrollsysteme	2
1.1.3	Verteilte Versionskontrollsysteme	2
1.2	GIT Historie	3
2	GIT Grundlagen	4
2.1	Begriffe die man kennen sollte	4
2.2	Repository	4
2.3	Clone	4
2.4	Commit	4
2.5	Branch	4
2.6	Merge	5
2.7	Snapshot	5
2.8	Querschnitt von GIT	5
3	Installation von GIT unter Linux	6
3.1	Installation unter Windows	7
3.2	Konfiguration von GIT	7
3.3	Hilfestellungen durch das System	9
4	Mit Git Arbeiten	9
4.1	Ein Git Repository anlegen	10
4.2	Das .git Repository	10
4.3	Ein Git Repository clonen	11
4.4	Änderungen nachverfolgen	12
4.5	Dateien ignorieren	14
4.6	Commithistorie anzeigen	15
4.7	Filtern der Commit historie	17
5	Branching mit Git	18
5.1	Was ist ein Branch?	18

II Abbildungsverzeichnis

Abb. 1	Lokale Architektur	1
Abb. 2	Zentralisierte Architektur	2
Abb. 3	Verteilte Architektur	2
Abb. 4	Liste von Änderungen	5
Abb. 5	Wie GIT speichert	5
Abb. 6	Querschnitt von GIT	6
Abb. 7	File Status Lifecycle	12
Abb. 8	gitk Grafische Oberfläche	17
Abb. 9	Speichern als Modell	18
Abb. 10	Commit Diagramm	18

III Tabellenverzeichnis

Tab. 1	Befehle zum filtern der Commithistorie	17
--------	--	----

IV Listing-Verzeichnis

Lst. 1 Git Repository anlegen	10
Lst. 2 Git Repository Dateien hinzufügen	10
Lst. 3 Git Repository Klonen	11
Lst. 4 Git Statusbefehl nach git clone befehl	13
Lst. 5 Git Statusbefehl nach dem erzeugen einer Datei	13
Lst. 6 Git Statusbefehl nach dem erzeugen einer Datei	14
Lst. 7 Git Statusbefehl nach dem verändern einer Datei	14
Lst. 8 Git Einstellungen der.gitignore Datei	15
Lst. 9 Erstellen der.gitignore Datei	15
Lst. 10Git log Unterschiede der letzten 2 Commits	16

V Abkürzungsverzeichnis

1 GIT

1.1 Was ist ein Versionskontrollsystem

GIT ist ein Versionsverwaltungssystem, soviel wissen wir. Doch was ist das und was macht es im Detail? Ein Versionsverwaltungssystem ist ein System, welches Änderungen an einer Datei oder eine Reihe von Dateien protokolliert, so dass bestimmte Versionen später wieder aufrufbar sind.¹ Um Problemen entgegenzuwirken, die eine amateurhafte Methoden der Versionsverwaltung mit sich bringen, wie z.B das ständige kopieren neuer Versionen in ein Verzeichnis, hierfür wurden diese Systeme entwickelt. Dabei unterscheidet man 3 Arten von Systemen. Der wesentlichste Unterschied, besteht darin wie und wo die Daten gehalten werden.

1.1.1 Lokale Versionskontrollsysteme

Von Lokalen Versionskontrollsystemen spricht man, wenn die Daten auf dem Lokalen System vorliegen (siehe Abbildung 1). Dabei werden die Dateien in einer Version, Database (Repository) gehalten. Nach jedem Checkout wird automatisch eine neue Version im Repository erstellt. Somit entgeht man der Gefahr, durch das oben erwähnte Kopieren in andere Verzeichnisse, eine der Versionen zu überschreiben, da man vergessen hat die Datei umzubenennen. Natürlich ist diese Variante der Versionskontrolle, für große Projekte die im Team bearbeitet werden, eher destruktiv. Ein Beispiel für Lokale Systeme ist RCS(Revision Control System). Für Teamwork, eignen sich eher die anderen beiden Architekturen.

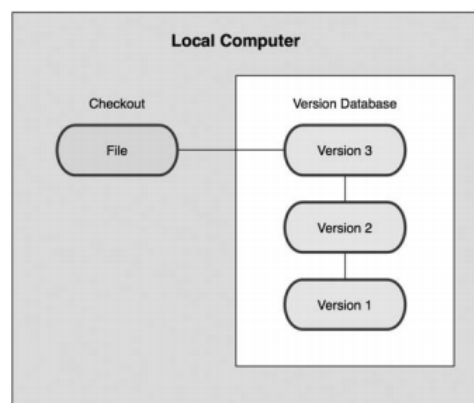


Abbildung 1: Lokale Architektur [Cha09]²

¹[Cha09] Seite 1 Zeile 1

1.1.2 Zentralisierte Versionskontrollsysteme

Bei zentralisierten Versionskontrollsystemen, wird die Versionierung nicht lokal vorgenommen. Die Entwickler haben einen zentralen Punkt (Abbildung 2), einen Server und dort befindet sich der Quellcode des Projektes in einem Repository, zu deutsch Lager. Der Unterschied zu einfachen lokalen Systemen ist nun offensichtlich, man braucht zumindest ein Netzwerk, um solche Systeme zu nutzen. Ein sehr beliebtes zentralisiertes System ist Subversion. Ein weiterer Vorteil gegenüber der lokalen Versionierung, besteht darin das gemeinsames Arbeiten an einem Projekt möglich ist und bei Verwendung eines Servers der Online erreichbar ist, kann das Arbeiten auch ohne Ortsbindung ablaufen. Doch dieser Vorteil der Ortsungebundenheit, bietet einen enormen „Single Point of Failure“, denn wenn der Server ausfällt, ist man nicht in der Lage seiner Arbeit nachzugehen.

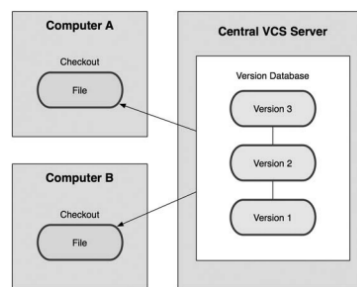


Abbildung 2: Zentralisierte Architektur³

1.1.3 Verteilte Versionskontrollsysteme

GIT gehört zu den verteilten Systemen, der Unterschied zu den Varianten davor ist das sie beides können. Einer Art hybride Lösung, man ist in der Lage Lokal zu versionieren aber auch im Netzwerk Versionen, anderen zur Verfügung zu stellen (Abbildung 3). Jeder kann als Server fungieren und somit wird der „Single Point of Failure“ eliminiert den zentralisierte Systeme haben. In der Praxis ist aber eher üblich, dass man einen Server nutzt vor allem bei Teamarbeiten. Wenn dieser jedoch ausfällt ist man in der Lage, weiter seine Arbeit zu verrichten.

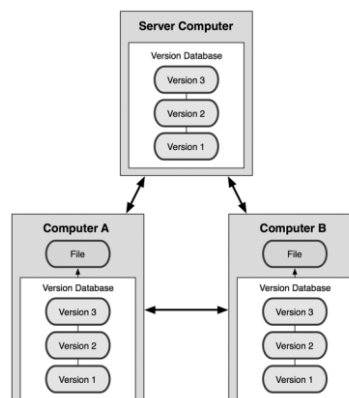


Abbildung 3: Verteilte Architektur⁴

1.2 GIT Historie

Im Jahre 2005 ist es zu Unstimmigkeiten gekommen, zwischen der Entwicklercommunity von Linux und dem Anbieter des proprietären BitKeeper-Systems, dass vorher kostenfrei genutzt wurde. Die Linux-Kernel-Entwickler mussten sich etwas einfallen lassen. Deswegen begann Linus Torvalds im April 2005 mit der Entwicklung von GIT und präsentierte auch sehr schnell die erste Version. Git baute auf den Erfahrungen mit BitKeeper auf, doch die Hauptziele des neuen Systems waren⁵:

- Geschwindigkeit
- Einfaches Design
- Gute Unterstützung von nicht-linearer Entwicklung
- Vollständig verteilt

Durch die kontinuierliche Weiterentwicklung des Systems und die Benutzerfreundlichkeit wurde Git zu einem sehr beliebten Tool. Einen großen Einfluss auf den Erfolg von Git hat auch die Social Coding Plattform GitHub, auf der man viele open source Projekte findet wie z.B :

- Der Linux Kernel⁶
- Ruby on Rails⁷
- Die Javascript Bibliothek JQuery ⁸
- Das CMS Joomla⁹

Das sind natürlich nicht alle Open Source Projekte, die GIT in Verbindung mit GitHub nutzen, aber einige bekannte die sich für Git entschieden haben. Der Dienst, den GitHub bereitstellt ist kostenfrei, doch nur unter der Bedingung das die Projekte öffentlich zugänglich sind. Des Weiteren gibt es Optional wählbare Services, die gegen Bezahlung verfügbar sind, aber es gibt auch eine Enterprise Version, die für Firmen interessant sein kann.

⁵[Cha09] Seite 5

⁶[Tor]

⁷[Rub]

⁸[Jqu]

⁹[joo]

2 GIT Grundlagen

Um grundlegende Funktionen von GIT zu nutzen, ist es unumgänglich gewisse Begriffe zu kennen. Elementar hingegen, ist der Umgang mit der Konsole des jeweiligen Systems. Es existieren einige plugins für Entwicklungsumgebungen, wie z.B. Eclipse die mit einem GUI ausgestattet sind. Jedoch sind diese Plugins meistens nicht soweit entwickelt, um den kompletten Funktionsumfang des Systems bedienbar zu machen, weswegen meine Erläuterungen zum Git System, sich auf Linux als Betriebssysteme beziehen und nur mit der Konsole zu bedienen sind.

2.1 Begriffe die man kennen sollte

Bevor man mit Versionierungssystemen arbeitet, sollte man einige Begriffe kennenlernen.

2.2 Repository

Der Begriff Repository (Englisch für Lager), kommt aus dem Lateinischen Repositorium. Eine Repository ist eine spezielle Datenbank, zur systematischen Ablage von Modellen und deren Bestandteilen das Herz der Datenhaltung eines Versionskontrollsystems. Grundlegende Funktion ist, die Speicherung und das Abrufen von gespeichertem Inhalt samt aller Bestandteile, wie z.B. Bilder. ¹⁰

2.3 Clone

Ein Clone im Git Kontext, ist äquivalent zu dem Begriff in der Biologie. Daher ist eine exakte Kopie von etwas existierendem, in diesem Fall der Repository. Im Fall Git lässt sich ein Clone, auch durch verschiedene Protokolle umsetzen z.B. git:// ein eigens Protokoll oder auch das https:// Protokoll. Weitere Erläuterungen dazu folgen später.

2.4 Commit

Zu deutsch "übergeben", wenn man also eine Änderung im Arbeitsverzeichnis vornimmt, wird diese getrackt (verfolgt). Um diese zu bestätigen, bzw. an Git zu übergeben, ist ein commit erforderlich.

2.5 Branch

Zu Deutsch Zweig ist eine Gabelung des Quellcodes. Aus verschiedenen Gründen, kann es erforderlich sein eine Version des Quellcodes vom Original abzuzweigen, um ggf. eine neue Funktion zu implementieren. Dies läuft dann parallel zur Entwicklung des Originalcodes. Der initiale Commit, wird auch als Master bezeichnet

¹⁰[Ley]

2.6 Merge

Ein Merge, bzw. das Merging ist das zusammenführen zweier Branches.

2.7 Snapshot

Ist die Art und Weise wie GIT Daten betrachtet. Viele anderen Systeme speichern Information als eine fortlaufende Liste von Änderungen an Dateien.

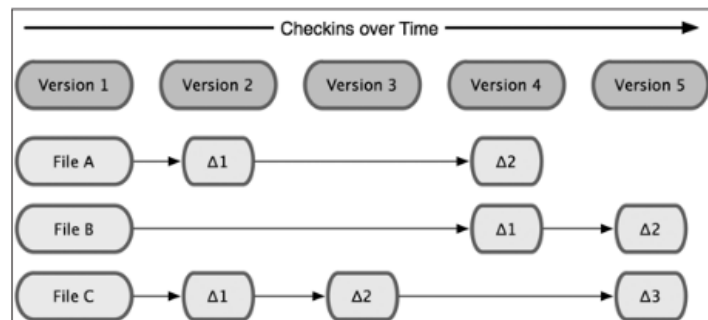


Abbildung 4: Liste von Änderungen¹¹

GIT hingegen Speichert alle Dateien bei jedem Commit. Unveränderte Dateien werden nicht Kopiert, sondern es wird eine Verknüpfung zu der vorherigen Version der Datei angelegt. Abb. 5 zeigt die Art und Weise wie GIT Daten speichert.

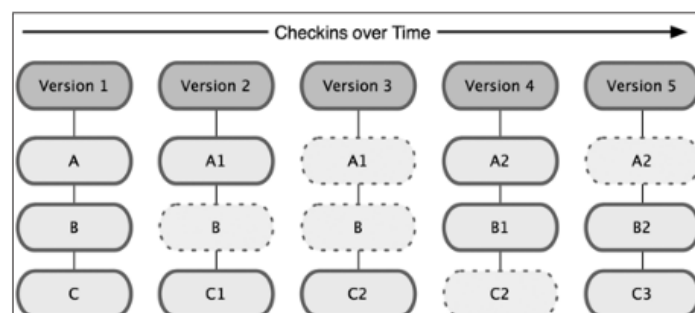
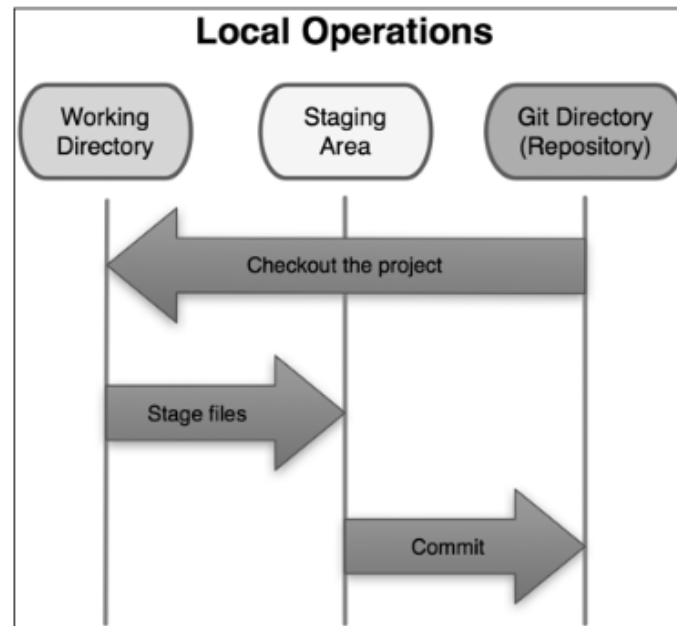


Abbildung 5: Wie GIT speichert¹²

Somit ist ein Snapshot die im letzten Commit gespeicherte Version des Projektes.

2.8 Querschnitt von GIT

GIT ist wie in der Abbildung 6 aufgebaut Das Repository enthält die Metadaten und die lokale Datenbank für das Projekt. Die staging area ist der Bereich, indem sich Dateien befinden, welche Vorgemerkt sind, um bei dem nächsten commit samt Änderungen in der GIT Directory gespeichert werden (snapshot).

Abbildung 6: Querschnitt von GIT¹³

3 Installation von GIT unter Linux

Unter Linux ist die Installation von Git abhängig davon, welche Distribution genutzt wird, d.h. welches Paketmanagement-Programm. Doch es wird empfohlen wenn möglich, Git vom Quellcode aus zu installieren, da man immer die neuste Version erhält.¹⁴ Um Git zu installieren braucht man einige Bibliotheken, die von Git verwendet werden: curl, zlib, openssl, expat und libiconv.

Installation unter Fedora Paketmanagement (YUM)

```
1 $ yum install curl-devel expat-devel gettext-devel \openssl-devel zlib-devel
```

Installation unter Debian/ Ubuntu

```
1 $ sudo apt-get install curl-devel expat-devel gettext-devel \openssl-devel zlib-devel
```

Download GIT

```
1 http://git-scm.com/download
```

Nachdem man diese Pakete installiert hat, ist es noch erforderlich Git selbst, zu downloaden. Auf dieser Seite sind Git Downloads für verschiedenste Betriebssysteme vorhanden.

¹⁴[Cha09]

Um den Quellcode zu laden, wird man auf eine GitHub Repository weitergeleitet. Rechts auf dieser GitHub Page, ist immer ein Link zum Clonen, sowie ein Download link, der auf dem System eine Zipdatei abspeichert. Doch um einen Überblick der verschiedenen Versionen zu bekommen empfehle ich den “Older releases“link.

Anschließend wird Git kompiliert und installiert:

```
1 $ tar -zxf git-2.1.3.tar.gz
2 $ cd git-2.1.3
3 $ make prefix=/usr/local all
4 $ sudo make prefix=/usr/local install
```

Installation über den Paketmanager Git über den Paketmanager zu installieren, ist für Linux Anfänger durchaus praktischer. Je nach Distribution unterscheidet sich allerdings die Eingabe in das Terminal.

Terminal Installation für Fedora/ Ubuntu/ openSUSE

Fedora:

```
1 $ yum install git
```

Ubuntu:

```
1 $ apt-get install git
```

openSUSE :

```
1 $ zypper install git
```

3.1 Installation unter Windows

Auf der Homepage git-scm.com, findet man im Downloadbereich auch eine Windowsversion, diese wird einfach Installiert. Nach Abschluss der Installation, startet man eine separate Git Konsole. Wichtig hierbei ist, dass diese Konsole explizit nur auf Linux Befehle hört.

3.2 Konfiguration von GIT

Nachdem Git nun erfolgreich auf de System installiert ist, bedarf es noch einigen Konfigurationen. Über das tool `git config` ist es möglich, durch Eingabe in das Terminal Konfigurationen vorzunehmen, welche die Arbeitsweise und die Optik von Git beeinflussen. Die Konfiguratsinsdateien, sind an drei verschiedenen Orten im System gespeichert.

- Die Datei `gitconfig` im `etc` Verzeichniss enthält Werte, die für jeden Anwender des Systems und all ihre Projekte gelten. Durch Eingabe von `git config` mit der Option `--system` wird diese Datei verwendet.
- Die Werte in der Datei `~/.gitconfig`, gelten explizit für das Systemkonto, das gerade genutzt wird. Durch die Eingabe von `git config` mit der Option `--global`, wird diese Datei verwendet.
- Um einem Projekt, geltende Werte zuzuweisen, gibt es noch die Datei `git/config` im Verzeichniss des Projektes selbst.

Wichtig ist jedoch, dass die Dateiwerte aus den jeweils vorhergehenden Dateien überschrieben wurden. Als Beispiel: `git/config` überschreibt die Werte in `/etc/gitconfig`

Konfiguration auf Windows Systemen Auch auf Windows sucht Git nach der `.gitconfig` Datei im `$Home` Verzeichnis. In den meisten Fällen, ist das der Pfad `C:\Dokumente und Einstellungen\%USER%`. Auch die `/etc/gitconfig` Datei wird gesucht in diesem Fall, ist diese Datei in dem Verzeichnis, in das Git bei Windows installiert wurde.

Identität Konfigurieren Nachdem die Installation erfolgreich abgeschlossen wurde, ist es von enormer Wichtigkeit die Konfiguration zur Identität vorzunehmen. Git nutzt diese Parameter bei jedem Commit, um die Einstellungen vorzunehmen ist das Terminal öffnen und folgende Werte zu setzen.

```
1 $ git config --global user.name "Max Musterman"
2 $ git config --global user.email "mustermann@muster.de"
```

Wichtig ist dabei, dass diese Einstellungen in diesem Fall nur einmal vorgenommen werden, da die Option `--global` verwendet wird. Will man jedoch für ein explizites Projekt andere Identitätsdaten verwenden, muss man im Verzeichnis des Projekts, die selben Befehle aufrufen. Doch ohne die Option `--global`.

Editor Konfiguration Da man beim Committen eine Nachricht mit geben soll, ist es möglich durch die Konsoleneingabe, einen expliziten Texteditor zu bestimmen.

```
1 $ git config --global core.editor <gewuenschter Editor>
```

Verändert man diese Einstellungsmöglichkeit jedoch nicht, wird die Default Einstellung VIM gewählt.

Konfiguration überprüfen Will man nun die vorgenommenen Einstellungen überprüfen, genügt die Eingabe dieses Befehls.

```
1 $ git config --list
```

Manche der aufgelisteten Variablen, kommen vermutlich öfter vor. Dies liegt jedoch an den verschiedenen Dateien, z.B. `/etc/gitconfig`. In diesem Fall wird die zuletzt aufgelistete Variable genutzt.

3.3 Hilfestellungen durch das System

Für weitere spezielle Einstellungen, gibt es unter Git die Help Option. Die gilt für fast alle Befehle und um diese auszurufen gibt es verschiedene Möglichkeiten.

```
1 $ git help <verb>
2 $ git <verb> --help
3 $ man git-<verb>
```

Braucht man nun andere Optionen zum Befehl `git config`, Tippt man folgendes in die Konsole:

```
1 $ git help config
```

Es erscheint nun ein Manual für den jeweiligen Befehl. Dieses beinhaltet eine kurze Beschreibung zu der Funktion, sowie die Optionen die einem zur Verfügung stehen.

4 Mit Git Arbeiten

Der Grundstein ist gelegt, um das Arbeiten mit Git zu beginnen. Da unser System verteilt ist, also Lokal in jedem Fall vorliegt, ist die erste Aufgabe nach erfolgreichem installieren eine Versionierungsdatenbank zu erzeugen, ein sogenanntes Repository. Im Allgemeinen, setzt das Arbeiten mit Git, auch das Arbeiten mit der Konsole voraus. Es besteht auch die Möglichkeit, Plugins zu nutzen die einige Funktionen unterstützen. Jedoch unterstützen diese Plugins meistens nicht alle Funktionen die Git bietet. Somit sollten die grundlegenden Konsolenbefehle bekannt sein.

- `ls` - listet alle Verzeichnisse und Dateien auf, Bei Windows gilt der Befehl `"dir"`. Mit dem Parameter `-a` werden auch unsichtbare Dateien angezeigt.
- `cd` - Wechselt das Verzeichnis.
- `cd ..` - geht ein Verzeichnis zurück.
- `mkdir` - Erzeugt ein neues Verzeichnis.

Die Git Befehle, welche die Konsole entgegennimmt, beginnen alle mit dem Signalwort GIT.

4.1 Ein Git Repository anlegen

Um nun ein Repository anzulegen, navigiert man sich mit dem `cd`(Change Directory) Befehl in den Ordner, in welchen das Repository angelegt werden soll. Es ist aber auch Möglich ein neues Verzeichnis anzulegen mit `mkdir`.

```
1 name@comp:~$ mkdir GITtest(Legt neuen Ordner an)
2 name@comp:~$ cd GITtest(Springt in das Verzeichnis GITtest)
3 name@comp:~/GITtest$ git init
4 (Erzeugt ein leeres Git Repository im Ordner GITtest)
5 name@comp:~/GITtest$ ls -a
6 . .. .git (Erzeugtes Repository)
```

Listing 1: Git Repository anlegen

Nun ist ein Repository angelegt. Will man nun in Zukunft ein weiteres Repository nutzen, erzeugt oder wechselt man in das Verzeichnis seiner Wahl und tippt `git init`. Damit Git nun Veränderungen an Dateien tracken kann, muss man selbstverständlich diese auch in diesem Verzeichnis speichern. Doch nur das Speichern genügt Git nicht, man muss die Dateien dem Repository hinzufügen.

```
1 $ git add README.txt
2 $ git commit -m 'initial project version '
```

Listing 2: Git Repository Dateien hinzufügen

Einzeln durch das Kommando `git add DATEINAME` oder wenn es alle Dateien im Verzeichnis sind, durch den Befehl `git add -A`. Hat man seine Dateien nun hinzugefügt, folgt ein commit mit dem Kommando `git commit -m "TEXT"`. Hat man nun alles Richtig gemacht, erscheint bei der Statusabfrage von GIT mit dem Befehl `git status` nothing to commit (working directory clean).

4.2 Das .git Repository

Wenn man durch das Kommando `git init` ein Repository erzeugt, erstellt das System einen Unterordner im Verzeichnis. Den `.git` Ordner. Das Verzeichnis hat folgende Struktur:

```
1 $ cd .git
2 $ ls
3 HEAD
```

```
4  branches/  
5  config  
6  description  
7  hooks/  
8  index  
9  info/  
10 objects/  
11 refs/
```

Das sind die Standardinhalte des `.git` Ordners. Die Datei `config` enthält die projektspezifischen Konfigurationsoptionen und im Ordner `info` befindet sich eine Datei, die globale Dateiausschlussmuster enthält, sodass nicht jedes mal eine `.gitignore`-Datei neu erstellt werden muss. Das `hooks`-Verzeichnis enthält die client- oder serverseitigen Hook-Skripte. Der Ordner `branches` wird von neueren Git-Versionen nicht mehr verwendet und die Datei `description` wird nur vom Programm `GitWeb` genutzt. Das Herz des Systems bilden die Dateien `HEAD` und `index`, sowie die Verzeichnisse `objects` und `refs`. Dies sind die Kernkomponenten eines Git-Repositorys. Jegliche Versionen der Dateien werden in der Objektdatenbank gehalten, welche im Unterverzeichnis `.git/objects` liegt. Die darin enthaltenen Daten sind mit `zlib` komprimiert. Die `Head` Datei, ist eine symbolische Referenz auf den jeweiligen Branch, auf dem man sich gerade befindet. Das `refs`-Verzeichnis enthält Referenzen auf Commit-Objekte (Branches) in dieser Datenbank und in der Datei `index`, verwaltet Git die Informationen der Staging-Area. Die Datei `description` wird nur vom Programm `GitWeb` benötigt und der Ordner `Branches` wird bei neueren Git Versionen gar nicht erst benötigt.

4.3 Ein Git Repository klonen

Wenn im Git Kontext von Klonen die Rede ist, geht es darum ein bereits existierendes Repository, für eigene Zwecke zu Kopieren. Dies ist meist bei bereits vorhandenen Projekten der Fall. Mit `clone [url]` wird jede einzelne Version, jeder einzelnen Datei in der Historie des Repositorys heruntergeladen. Das hat zur Folge, dass selbst wenn der Server, von dem man ursprünglich den Klone hat, beschädigt wird, ist man in der Lage ihn wiederherzustellen, da der Klone alle Versionen hat. Als Beispiel ein Klone dieser Seminararbeit.

```
1 $ git clone https://github.com/veddo/FSEW.git
```

Listing 3: Git Repository Klonen

Beim diesem Vorgang legt Git ein Verzeichnis namens `.git` an und installiert die `.git` Datei darin, lädt alle Dateien des Repositorys runter und checkt die Arbeitskopie der letzten Version aus.

4.4 Änderungen nachverfolgen

Man hat jetzt ein voll funktionsfähiges Git Repository und eine ausgecheckte Arbeitskopie. Wie bereits vorher erwähnt, hat Git immer die selbe Abfolge von Kommandos, wenn man eine oder auch mehrere Dateien dem Repository hinzufügen möchte. Zur Wiederholung:

- `git status` - Überprüft den Status der Dateien im Repository.
- `git add` - Lassen sich neue Dateien oder geänderte Dateien hinzufügen.
- `git commit -m "TEXT"` - Erzeugt einen snapshot aller zum Repository hinzugefügten Dateien.

File Status Lifecycle Was da genau vorgeht, beschreibt am besten der File Status Lifecycle. Jede Datei im Arbeitsverzeichnis, kann sich in einem von zwei Zuständen befinden. Änderungen werden verfolgt (eng. tracked) oder sie werden nicht mitverfolgt (engl. untracked). Genauer gesagt: Dateien welche getrackt werden sind nur die, die sich im letzten Commit befinden. Die anderen Daten die nicht, mit dem `add` Kommando einem Commit hinzugefügt wurden, haben den Status Untracked.

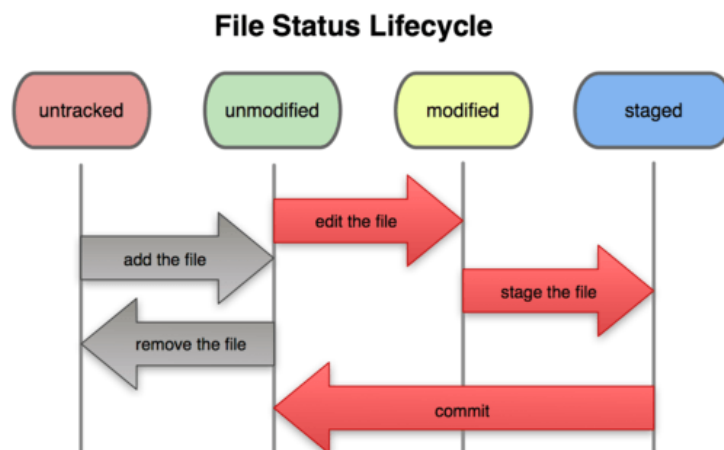


Abbildung 7: File Status Lifecycle¹⁵

Dateien die sich nun im Commit befinden, haben wiederum zwei mögliche Stadien in denen sie sich befinden können, veränderte (engl. modified) oder unverändert (engl. unmodified). Die veränderten Dateien sind somit für den nächsten Commit vorgemerkt (engl. staged).

Alle anderen Dateien, mit dem Status `unmodified` werden hingegen nicht Versioniert. Sobald man nun Dateien wieder bearbeitet, beginnt dieser Vorgang erneut.

Zustand der Dateien Prüfen Der Befehl `git status` wurde bereits weiter oben schon erwähnt und kurz erklärt, damit man eine grobe Vorstellung hat. Nun schauen wir ihn detaillierter an. Im Groben überprüft das Kommando, in welchen Status des File Lifecycle man sich befindet. Hat man nun ein Repository gerade geklont, bekommt man folgendes ausgegeben:

```
1 $ git status
2 On branch master
3 nothing to commit, working directory clean
```

Listing 4: Git Statusbefehl nach `git clone` befehl

Nach Abbildung 4 ist der momentane Status des Arbeitsverzeichnisses `unmodified`, d.h. es wurden weder Dateien hinzugefügt, noch wurde an einer anderen Datei etwas verändert. Wenn doch, würde `git` die veränderten und hinzugefügten Dateien auflisten.

Fügen wir nun eine neue Datei hinzu:

```
1 $ touch TEST.txt
2 $ git status
3 On branch master
4 Untracked files:
5   (use "git add <file>..." to include in what will be committed)
6
7     TEST.txt
8
9 nothing added to commit but untracked files present (use "git add" to
   track)
```

Listing 5: Git Statusbefehl nach dem erzeugen einer Datei

Wie man an der Meldung von Git erkennen kann, gibt es Dateien, die sich im `untracked` Status befinden. Mit der Anmerkung `use git add to track`, weist Git nun daraufhin, dass die neue Datei in den Status `unmodified` gebracht werden muss. Nach dem hinzufügen der Datei, durch `git add[Dateiname]` und einer erneuten Statusabfrage, meldet Git `Changes to be committed` Im File Lifecycle ist man also an erster Stelle.

```
1 $ git status
2 On branch master
3 Changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6       new file:   TEST.TXT
```

Listing 6: Git Statusbefehl nach dem erzeugen einer Datei

Der Status “Changes to be committed” sagt aus, dass sich Git die neue Datei vorgemerkt (gestaged) hat und erstellt beim nächsten Commit, einen snapshot dieser Datei. Wenn man sich nun den File Status Lifecycle noch einmal anschaut, befindet man sich zwischen modified und staged. Nun bedarf es wieder dem Git add Befehl, um die Datei für den Commit vorzubereiten. Nach dem Commit hat man nun wieder eine saubere (clean) working directory.

```
1 $ git status
2 On branch master
3 Changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6
7       modified:   TEST.txt
```

Listing 7: Git Statusbefehl nach dem verändern einer Datei

Verändert man nun diese Datei, indem man einen kleinen Text hinzufügt. Sollte man wieder den Status abfragen(Listing 7) Dem File Status Lifecycle zufolge, ist man im modified Status.

4.5 Dateien ignorieren

In den meisten Fällen kommt es vor, dass man einige Dateien gar nicht versionieren muss/soll, z.B. automatisch generierte Dateien wie Logfiles. In diesem Fall bietet Git eine Möglichkeit, dies zu Konfigurieren. Wie bei anderen Einstellungen ist es nötig, diese in einer Konfigurationsdatei zu hinterlegen, für die es klare Regeln gibt. Eine dieser festen Regeln ist der Name dieser Datei, sie muss .gitignore heißen.

Weitere Regeln im Bezug auf die .gitignore Datei sind:

- Leere Zeilen oder Zeilen, die mit # beginnen, werden ignoriert.
- Standard glob Muster funktionieren.

- Man kann ein Muster mit einem Schrägstrich (/) abschließen, um ein Verzeichnis zu deklarieren.
- Man kann ein Muster negieren, indem man ein Ausrufezeichen (!) voranstellt.

Ein Beispiel für die .gitignore Datei:

```
1 # ein Kommentar – dieser wird ignoriert
2 # ignoriert alle Dateien, die mit .a enden
3 *.a
4 # nicht aber lib.a Dateien (obwohl obige Zeile *.a ignoriert)
5 !lib.a
6 # ignoriert eine TODO Datei nur im Wurzelverzeichnis, nicht aber
7 /TODO
8 # ignoriert alle Dateien im build/ Verzeichnis
9 build/
10 # ignoriert doc/notes.txt, aber nicht doc/server/arch.txt
11 doc/*.txt
12 # ignoriert alle .txt Dateien unterhalb des doc/ Verzeichnis
13 doc/**/*.txt
```

Listing 8: Git Einstellungen der .gitignore Datei

Diese Datei lässt sich auch mit Hilfe der Konsole erzeugen und bearbeiten. Beim Bearbeiten helfen Editoren wie VIM oder NANO.

```
1 # Erzeugt die Datei
2 $ touch .gitignore
3 $vi .gitignore
```

Listing 9: Erstellen der .gitignore Datei

Nach aufrufen des Editors, kann man nun die Dateien ausschließen, welche nicht versioniert werden sollen.

4.6 Commithistorie anzeigen

Manchmal ist es nötig einige Commits genauer einzusehen. Diese Funktion erfüllt der Befehl `git log`. Das Kommando listet die Historie der Commits eines Projekts, in umgekehrter chronologischer Reihenfolge auf. Des Weiteren hat dieser Befehl sehr viele Optionen die man wählen kann. Eine sehr nützliche Option ist `git log -p`, sie zeigt auf welche Änderungen gemacht wurden. Das Ganze lässt sich auch eingrenzen, indem man einen weiteren Parameter hinzufügt. Als Beispiel `git log -p -2` zeigt nur die Änderungen der letzten beiden Commits an.

```
1 commit efb60ed05541f3bfae026c989f8b2f0cf6a2f42e
2 Author: Vedad Hamamdzie <vhamamdzie@stud.hs-heilbronn.de>
3 Date: Sun Nov 16 14:58:44 2014 +0100
4
5     World!
6
7 diff --git a/Gittest.txt b/Gittest.txt
8 index 336f590..5aae200 100644
9 --- a/Gittest.txt
10 +++ b/Gittest.txt
11 @@ -1,1 @@
12 -Hallo World
13 +Hallo World!
14
15 commit 04a24f7d6f2a998ec0f30ae289172e7cf07689d9
16 Author: Vedad Hamamdzie <vhamamdzie@stud.hs-heilbronn.de>
17 Date: Sun Nov 16 14:58:14 2014 +0100
18
19     World
20
21 diff --git a/Gittest.txt b/Gittest.txt
22 index e69de29..336f590 100644
23 --- a/Gittest.txt
24 +++ b/Gittest.txt
25 @@ -0,0,1 @@
26 +Hallo World
```

Listing 10: Git log Unterschiede der letzten 2 Commits

Da die Reihenfolge umgekehrt chronologisch aufgelistet wird, ist der letzte Commit oben zu sehen. Die beiden Commitnachrichten World ! und World, helfen beim Unterscheiden des gesuchten Commits. Man erkennt in Zeile 12 und 13, das zu Hallo World noch ein Ausrufezeichen dazu gekommen ist.

4.7 Filtern der Commit historie

Eine Commithistorie kann vor allem bei Teamarbeit sehr groß werden. Hierzu gibt es Filter Optionen die eine Suche ein wenig erleichtern können.

Option	Beschreibung
-(n)	Begrenzt die Ausgabe auf die letzten n commits
-since, -after	Zeigt nur Commits, die nach dem angegebenen Datum angelegt wurden.
-until, -before	Zeigt nur Commits, die vor dem angegebenen Datum angelegt wurden.
-author	Zeigt nur Commits, die von dem angegebenen Autor vorgenommen wurden.
-committer	Zeigt nur Commits, die von dem angegebenen Committer angelegt wurden.

Tabelle 1: Befehle zum filtern der Commithistorie

Es besteht auch noch die Möglichkeit, sich die Commithistorie grafisch anzeigen zu lassen, und zwar mit dem Tcl/Tk Programm. Dies wird in der Regel mit Git ausgeliefert und hört auf den Befehl gitk. Es ist im wesentlichen eine grafische Version von Git log und akzeptiert fast alle Filteroptionen, die Git log auch akzeptiert.

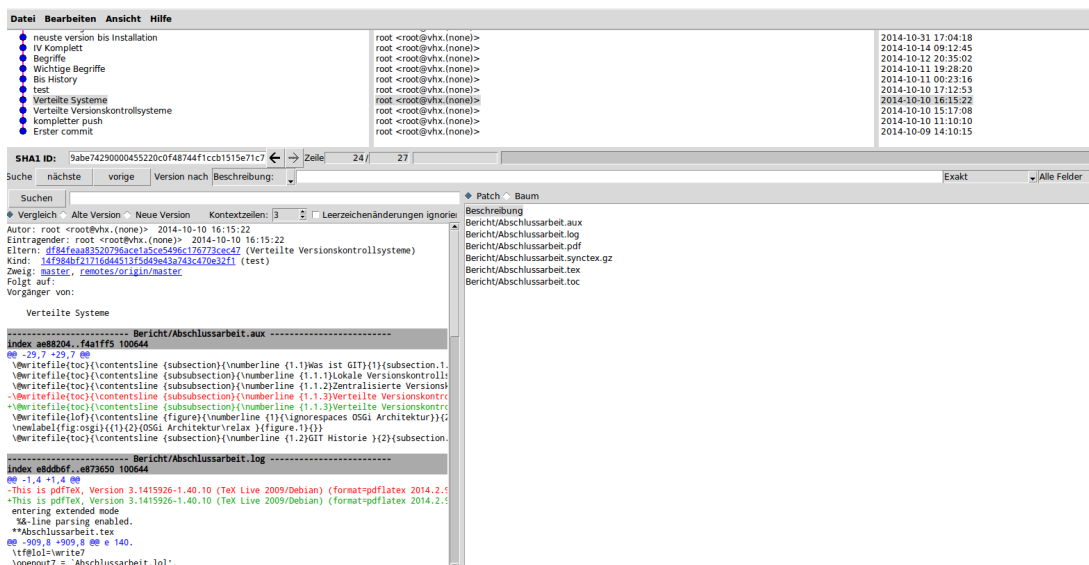


Abbildung 8: gitk Grafische Oberfläche

Falls gitk nicht mit installiert wurde, ist es selbstverständlich nötig es zu installieren, unter Ubuntu apt-get install gitk . Beim aufrufen sieht man die Commit Historie. Diese wird in der oberen Hälfte des Fensters dargestellt. Daneben ein Graph, der die Branches und Merges zeigt. Nach Auswahl eines Commits, zeigt die Vergleichsanzeige in der unteren Hälfte des Fensters die jeweiligen Änderungen in diesem Commit.

5 Branching mit Git

5.1 Was ist ein Branch?

Um das Branching wirklich zu verstehen, muss man im Detail verstehen, wie Git die Daten speichert. Wenn man Commitet, speichert Git ein sogenanntes Commit-Objekt. Dieses enthält einen Zeiger, zu dem Schnappschuss mit den Objekten der Staging-Area, dem Autor, den Commit-Metadaten und einem Zeiger zu den direkten Eltern des Commits. Beim Commiten bekommt jedes Projektverzeichnis eine Prüfsumme und wird als sogenanntes tree-Objekt im Git Repository gespeichert. Somit zeigt das Commit-Objekt auf das tree-Objekt, diese wiederum zeigen auf sogenannte Blobs, welche den Inhalt der Dateien enthalten.

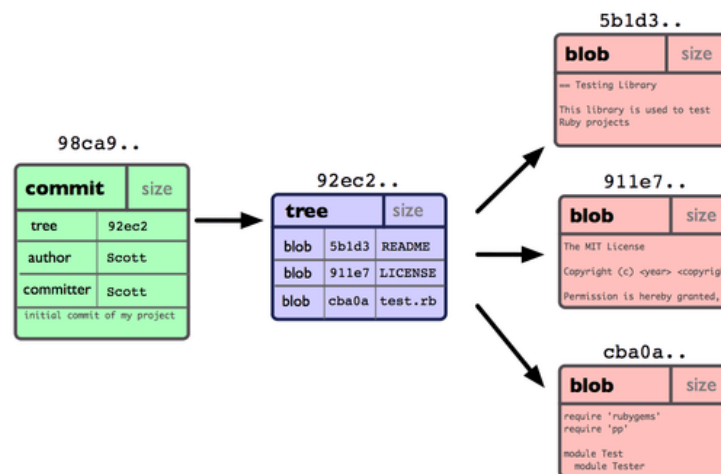


Abbildung 9: Speichern als Modell¹⁶

Das Tree-Objekt fungiert als ein Inhaltsverzeichnis im Verzeichnis. Jede Datei ist dort aufgelistet und spezifiziert. Des Weiteren gibt es noch einen Zeiger, der auf die Wurzel des Projektbaumes und die Metadaten des Commits verweist (Abb.6). Jede weitere Commit wird einen Zeiger enthalten, der auf den vorhergehenden verweist. Nach zwei weiteren Commits könnte die Historie wie folgt aussehen (Abb. 7).

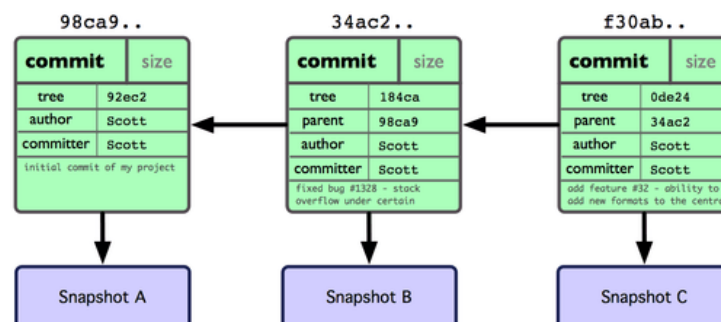


Abbildung 10: Commit Diagramm¹⁷