Fallstudie Entwicklungswerkzeuge

Ausarbeitung

über das Thema

GIT Versionsverwaltungssystem

Autor: Vedad Hamamdzic
email@email.de

Prüfer: Paul Lajer

Abgabedatum: 18.11.2014

I Zusammenfassung

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Abstract

Das ganze auf Englisch.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Listing-Verzeichnis

VI 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.

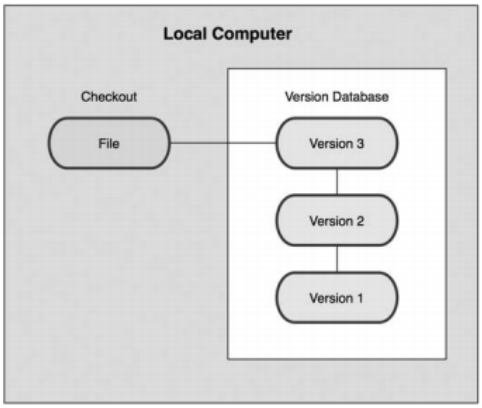

Abbildung 1: Lokale Architektur [Cha09][2]

[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 Reposytory, 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 Versionnierung, 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.

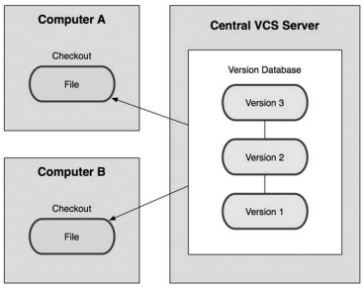

Abbildung 2: Zentralisierte Architektur[3]

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.

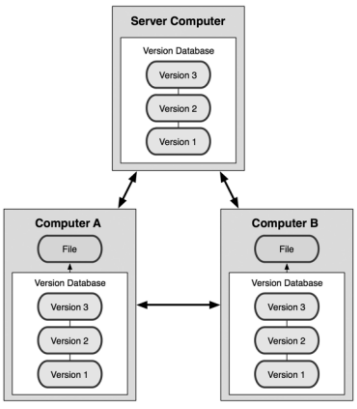

Abbildung 3: Verteilte Architektur[4]

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 waren[5]:

- Geschwindigkeit
- Einfaches Design
- Gute Unterstützung von nicht-linearer Entwicklung (tausende paralleler verschiedener Verzweigungen der Versionen)
- Vollständig verteilt
- Fähig, große Projekte wie den Linux Kernel effektiv zu verwalten

Durch die kontinuierliche Weiterentwicklung des Systems und die Benutzerfreundlichkeit wurde Git zu einen sehr beliebten Tool. Ein 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 Kernel[6]
- Ruby on Rails[7]
- Die Javascript Bibliothek JQuery [8]
- Das CMS Joomla[9]

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 umsetzten 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 SZweigïst 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 eines Branches und des Master zweiges.

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.

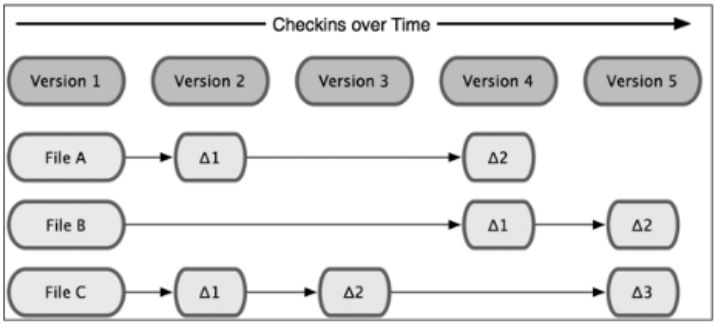

Abbildung 4: Liste von Änderungen[11]

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. Abbx zeit die Art und weise wie GIT Daten speichert.

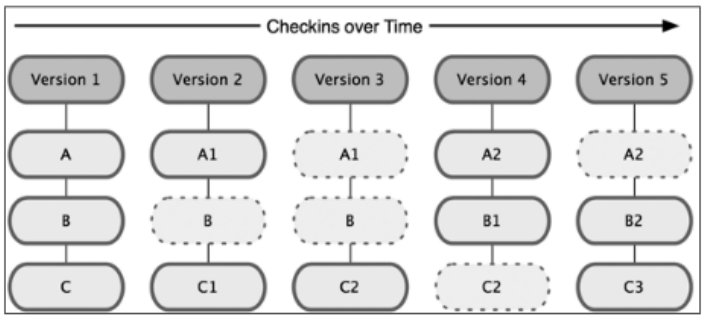

Abbildung 5: Wie GIT speichert[12]

Somit ist ein Snapshot die im Letzten commit gespeicherte Version des Projektes.

2.8 Zustände der Dateien

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).

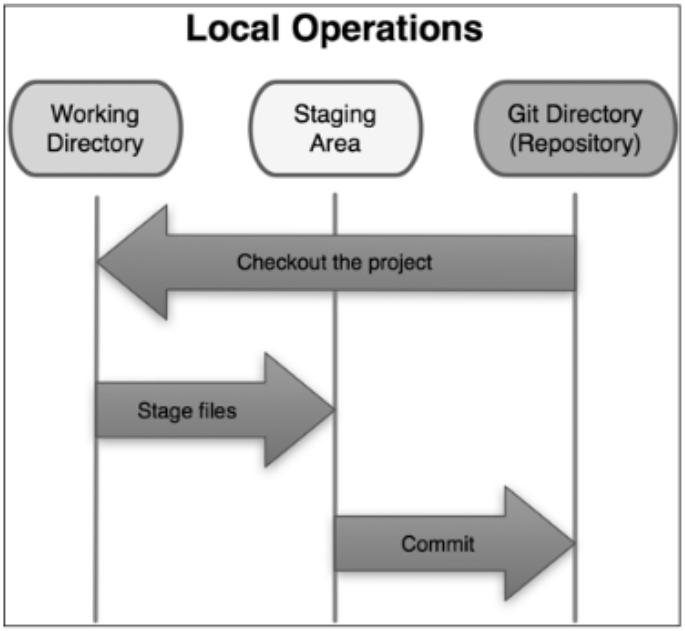

Abbildung 6: Drei Zustände einer Datei[13]

3 Installation von GIT unter Linux

Unter Linux ist die Installation von Git abhängig, 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 Packete installiert hat, ist es noch notwendig 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 Ölder 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 seperate 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 Konfiguratinsdateien, 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/configim 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 das Terminal öffnen und

```
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 −−globalverwendet 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 Commiten eine Nachricht mit geben soll, ist es möglich durch die Konsoleneingabe, einen expliziten Texteditor zu bestimmen.

```
1 $ git config --global core.editor <gew nschter 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 .Iin 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, 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. Diese 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 vorraus. 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 entgegenimmt beginnen alle mit dem Signalwort GIT.

4.0.1 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 ¡Name¿.

```
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 (Erzeugntes 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 Status abfrage von git mit dem Befehl git status nothing to commit (working directory clean).

4.0.2 Ein Git Repository clonen

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 Klon alle Versionen hat. Als Beispiel ein Klon dieser Seminararbeit.

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

Listing 3: Git Repository Klonen

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

4.1 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 Lifecyrcle Was da genau vorgeht, beschreibt am besten der File Status Lifecyrcle. 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.

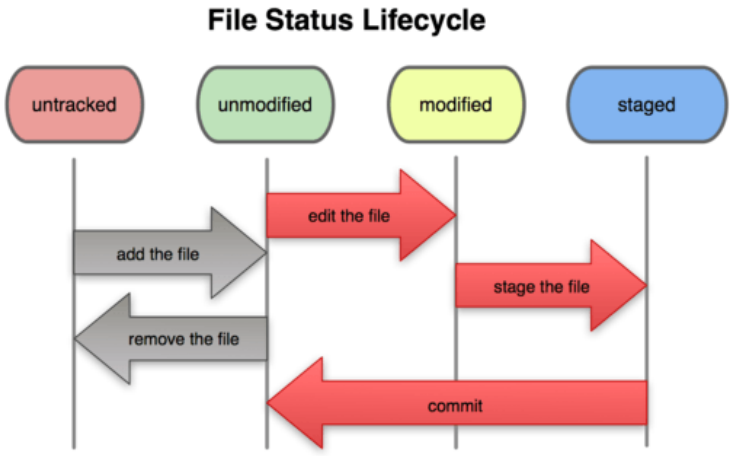

Abbildung 7: Zentralisierte Architektur[15]

Dateien die sich nun im Commit befinden, haben wiederum zwei mögliche Stadien in denen sie sich befinden können, Veränderte (engl. modified) oder unverändert (eng. 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 Life Cyrcle 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 nachdem 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 folgendes: 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 nachdem erzeugen einer Datei

Der Status "Changes to be committedßagt 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 nachdem 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.

4.2 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: Git Erstellen der.gitignore Datei

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

4.3 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 Hamamdzic <vhamamdz@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 Hamamdzic <vhamamdz@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.

4.3.1 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	eigt 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.

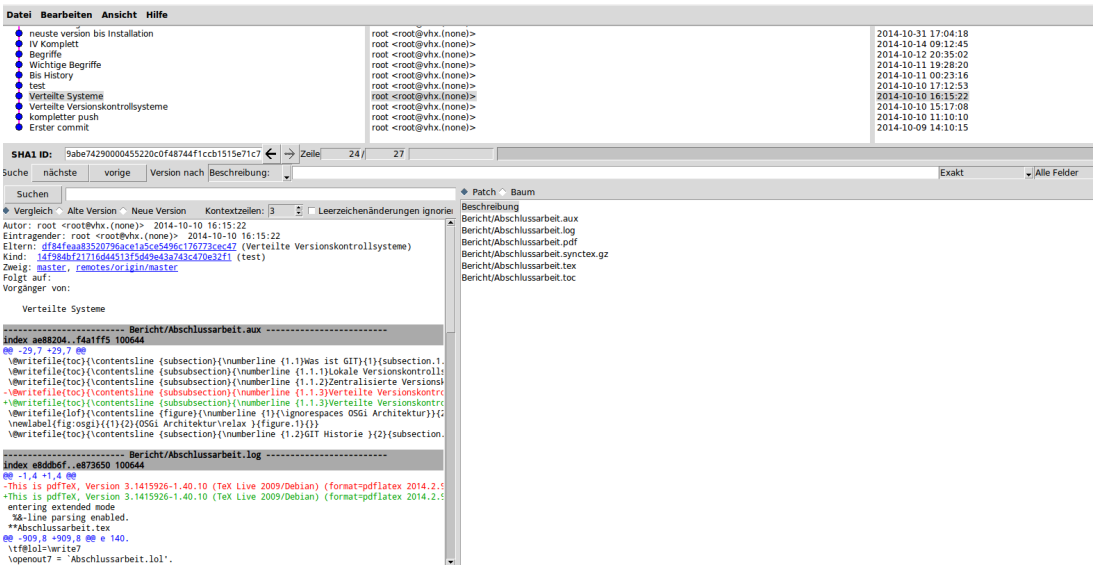

Abbildung 8: gitk Grafische Oberfläche

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

Lorem ipsum dolor sit amet.

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. Sommit zeigt das Commit-Objekt auf das Tree-Objekt diese wiederrum zeigen auf sogenannte Blops, welche den Inhalt der Dateien enthalten.

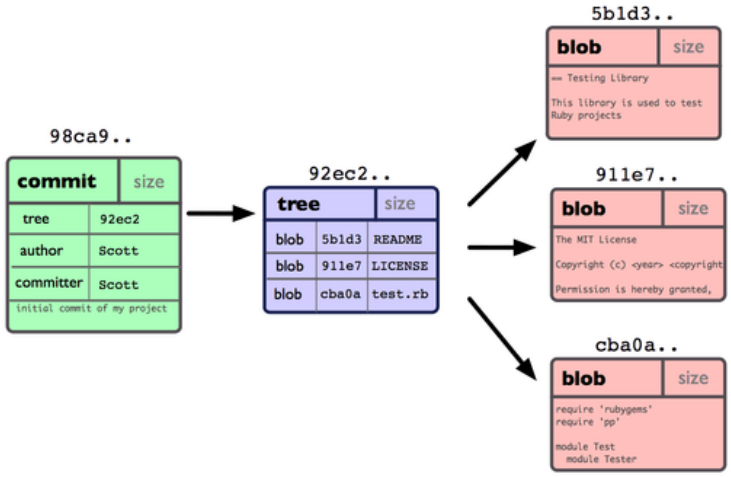

Abbildung 9: Speichern als Modell[16]

Das Tree-Objekt hat die Funktion einen Inhaltsverzeichnisses im Verzeichnis jede Datei ist dort aufgelistet und spezifiziert welcher Dateiname zu welchem Blob gehört. Des Weiteren gibt es noch einen Zeiger, der auf die Wurzel des Projektbaumes und die Metadaten des Commits verweist (Abb.6). Jede weitere Commt wird einen Zeiger enthalten, der auf den Vorhergehenden verweist. Nach zwei weiteren Commits könnte die Historie wie folgt aussehen (Abb. 7).

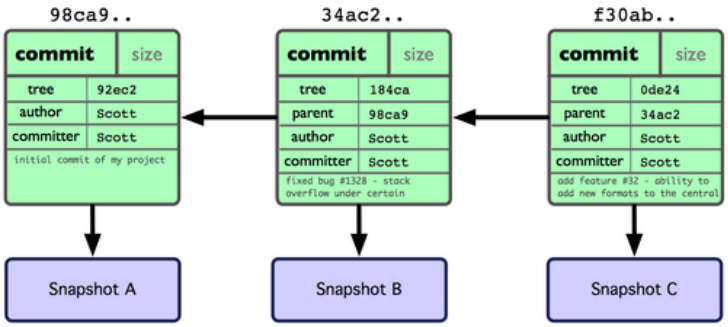

Abbildung 10: Speichern als Modell[17]

Ein Branch in Git ist nichts anderes als ein simpler Zeiger auf einen dieser Commits, der Standardname eines Git-Branches lautet master. Diesen master Branch erhällt man durch den Initialen Commit der sich bei jedem neuen Commit um einz nach vorne verschiebt.

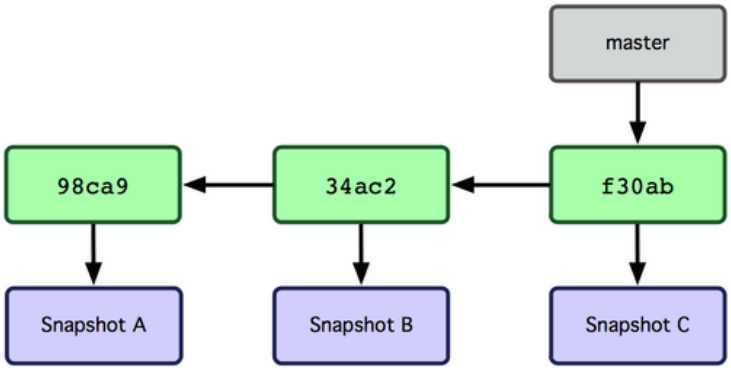

Abbildung 11: Master Branch[18]

Wenn man nun einen neuen Branch generiert wird ein neuer Zeiger erstellt, der auf den Commit zeigt auf dem man gerade arbeitet.

```
1 $ git branch NeuerBranchName
```

Listing 11: Branch Befehl

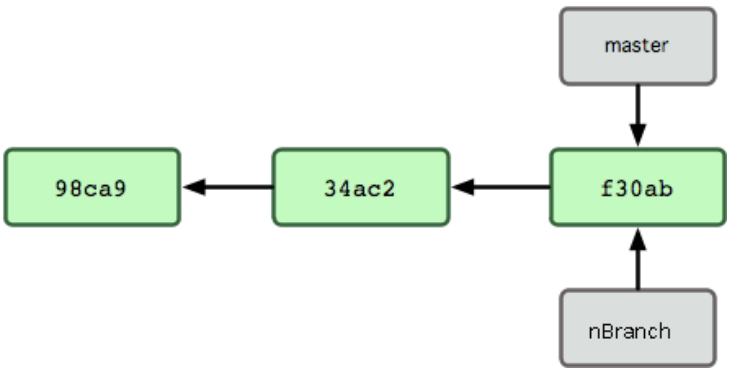

Abbildung 12: Neuer Branch[19]

Nun befinden wir uns immer noch auf dem Masterzweig das bedeutet man muss in den gerade erstellten Branch wechseln. Dies macht man mit dem Checkout Befehl.

```
2 $ git checkout
3 $ git checkout NeuerBranchName
```

Listing 12: Branch Befehl

Um den zu sehen in welchen Branch man sich befindet kann man auf das Kommando git branchzurückgreifen. Der Zweig welcher mit dem Sternchen gekennzeichnet ist, ist der indem man Arbeitet. Dieses Sternchen ist ein spezieller Zeiger der sich HEAD nennt.

```
4   master
5 * neuerBranch
```

Listing 13: Aktuellen Branch prüfen

5.2 Einfaches Branching und Merging

Nachdem Branching ausführlich erläutert wurde, ist Mergigng nun an der Reihe. Merging ist das zusammenführen zweier Branches zu einem Zweig. Das kann man am besten an einem Fallbeispiel erläutern. Man schreibt an einer Software, die schon eine Vorgängerversion besitzt. man erstellt sich einen Branch für die neue Version. Plötzlich wird ihnen ein Fehler gemeldet in der Alten Version. Um nun sicherzustellen das man nicht die bisherigen Änderungen (neue Version) und den Fix des Fehlers bereitstellt, wechselt man in den Zweig der alten version.

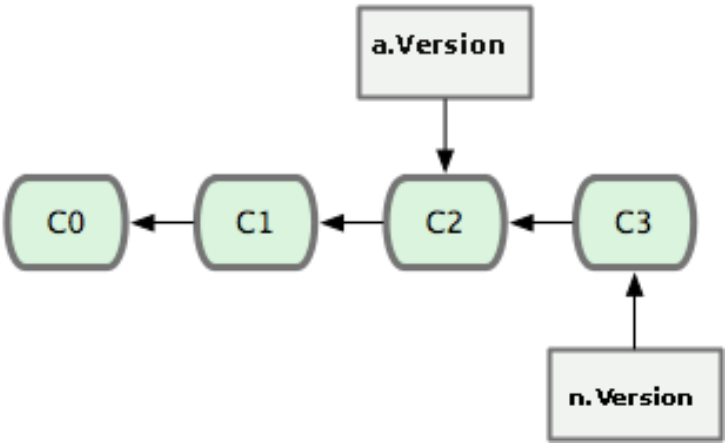

Abbildung 13: Speichern als Modell[20]

Dort erstellt man sich einen weiteren Branch der als zweig aus der alten Version entsteht. Diesen kann man dann Bearbeiten und mit der alten Version mergen und ausliefern ausliefern.

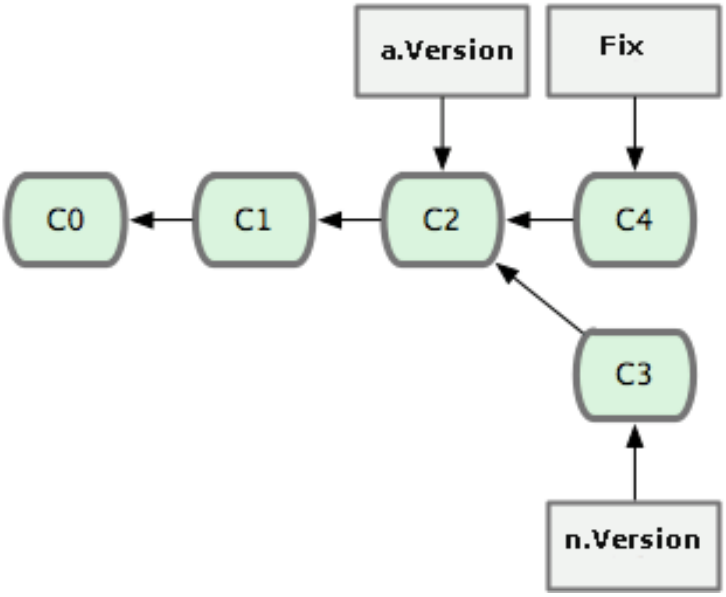

Abbildung 14: Speichern als Modell[21]

Man hat nun nur die Alter Softwareversion bearbeitet ohne den neuen Code mit in die Version zu integrieren. Durch ein einfaches checkout Kommando kann man nun wieder in seinen Zweig zurück gehen und weiter arbeiten. Jedoch sollte man auch beim mergen den File Status Lifecycle nicht außer acht lassen. denn es werden nur 2 branches gemerged die auch Commitet sind.

5.3 Merge-Konflikte

Es kommt gelegentlich auch vor das ein Merge nicht zusammengefügt werden kann, wenn man beispielsweise an den selben stellen in Unterschiedlichen Branches was geändert hat. Diese Meldung über ein 'merge'-Konflikt könnte ungefähr so aussehen:

```
6  Auto-merging test.txt
7  CONFLICT (content): Merge conflict in Gittest.txt
8  Automatic merge failed; fix conflicts and then commit the result.
9  test@test:~/test$ git status
10 # On branch master
11 # Unmerged paths:
12 #   (use "git add/rm <file>..." as appropriate to mark resolution)
13 #
14 # both modified:      test.txt
15 #
16 no changes added to commit (use "git add" and/or "git commit -a")
```

Listing 14: Branch Konfilkt

Die Dateien die einen Mergekonflikt aufweisen werden auf unmerged gesetzt und es werden Konfliktlösungsmarker hinzugefügt. Dies kann dann so aussehen.

```
17 <<<<<<< HEAD
18 Hallo
19 =======
20
21 Hallo Welt
22 >>>>>>> fix
```

Listing 15: Branch Konfilkt

Diesen Konflickt muss man manuel auflösen. Der als Head Markierte Teil ist der Branch auf dem man sich gerade befindet.Überhalb von ====== unterhalb ist der Branch der gemerged werden soll. Es ist nun notwendig sich zu entscheiden was man nimmt. Aber auch ein komplettes ersetzen des ungemergten Teils ist möglich. Nachdem man den Konflikt aufgelöst hat, einfach dem File Status Lifecycle nach weiter arbeiten. Es besteht noch die Möglichkeit ein grafisches Tool zur Bereinigung zu nutzen. Falls keines Installiert ist erscheint eine Liste der Tools die man Problemlos nutzen kann.

```
23 merge tool candidates: meld opendiff kdiff3 tkdiff xxdiff tortoisemerge
       gvimdiff diffuse ecmerge p4merge araxis bc3 emerge vimdiff
```

Listing 16: Merge Tools

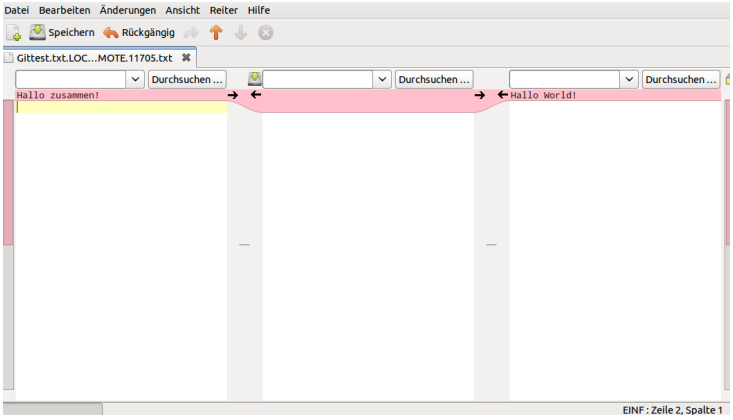

Abbildung 15: Mergetool Meld[22]

Abbildung 12 zeigt das Mergetool Meld, es zeigt den komflikt auf und ermöglicht mit einigen Klicks das lösen des Problems.

6 Git alte Versionen Wiederherstellen

7 Git in Netzwerken

Um das zusammenarbeiten in Grupppen möglich zu machen aber auch dem Singe Poit of Failure entgegenzuwirken gibt es noch die möglichkeit ein Git Repository extern zur Verfügung zu stellen. Es besteht die Möglichkeit git auf dem eigenen Computer Kolaborativ zu nutzen jedoch hat sich das in der Praxis nicht durchgesetzt. Git als Server zu betreiben ist in der Praxis üblich. Ein externes Repository ist im Allgemeinen ein einfaches Repository ohne Arbeitsverzeichnis. Aus dem Einfachen Grund da ein Knotenpunkt ist. Auf dem Server Befinden sich nur einfache inhalte aus der .git Datei und Projektdateien.

7.1 Welche Protokolle untersützt Git

Git ist in der Lage vier verschieden Protokolle zum Datentransfer zu nutzen.

Lokales Protokoll Das lokale Protokoll kommt zum Einsatz wenn man im Team ein Gemeinsames Dateisystem nutzt z.B. ein eingebundenes NFS. Man ist ist der Lage von einem lokalen Datei-basiertem Repository zu clonen, pushen und pullen. Es gibt aber zwei unterschiedliche Befehle die man nutzen kann. Mit Präfix oder ohne.

```
24 $ git clone /opt/git/Myproject.git
```

Listing 17: Ohne Präfix

```
25 $ git clone file:///opt/git/Myproject.git
```

Listing 18: Mit Präfix

Die beiden Arten unterscheiden sich auch von einander. Wenn man nur den Pfad angibt, und sowohl die Quelle, als auch das Ziel sich auf dem selben Dateisystem befinden, versucht Git harte Links zu benutzen. Wenn man die Variante mit dem Präfix nutzt, startet Git den Prozess, den es normalerweise zum Übertragen von Daten über ein Netzwerk verwenden würde, dass ist gewöhnlich die ineffizientere Methode zum Übertragen der Daten. In der Regel benutzt die Variante ohne Präfix da diese auch die Schnellere ist.

Das SSH Protokoll Das vermutlich gebräuchlichste Transport-Protokoll für Git ist SSH. Es ist weit verbreitet und leicht einzurichten des weiteren unterstützt es lesende und schreibende Zugriffe.

```
26 $ git clone ssh://user@server/Myproject.git
```

Listing 19: Mit Präfix

Das Git Protokoll Das Git Protokoll kommt mit Dem Versionsverwaltungsystem. Dieser Service ist vergleichbar mit dem SSH-Protokoll, aber ohne jegliche Authentifizierung. Das Git Protokoll ist das schnellste verfügbare Transfer Protokoll, dar es den selben Daten-Transfer Mechanismus wie das SSH-Protokoll nutzt, aber ohne den Entschlüsselungs- und Authentifizierungs-Overhead.

Das HTTP/S Protokoll Ist der Einfachheit wegen sehr beliebt, man braucht nur die Git Repository in das HTTP Hauptverzeichnis zu legen und einen speziellen post-update hook einrichten, schon ist das Clonen über HTTP eingerichtet.

8 Quellenverzeichnis

[Bre] Brettschneider, Daniel: *Daniel Brettschneiders Blog.* http://www.daniel-brettschneider.de. – Zugriff: 15.02.2013, Archiviert mit WebCite®: http://www.webcitation.org/6ESWiGbhw

[Cha09] Chacon, Scott: *Pro Git.* Berkeley, CA New York : Apress Distributed to the Book trade worldwide by Springer-Verlag, 2009. – ISBN 9781430218340

[joo] joomla: *Joomla CMS.* https://github.com/joomla/joomla-cms

[Jqu] Jquery: *Jquerry, jquery.* https://github.com/jquery/jquery. – Zugriff: 15.02.2013, Archiviert mit WebCite®: https://github.com/jquery/jquery

[Ley] Leymann, Professor Dr. F.: *Repository,Leymann.* http://wirtschaftslexikon.gabler.de/Definition/repository.html. – Zugriff: 20.10.2014, Archiviert mit WebCite®: http://wirtschaftslexikon.gabler.de/Definition/repository.html

[Rub] Ruby: *Ruby on Rails.* https://github.com/rails/rails. – Zugriff: 15.02.2013, Archiviert mit WebCite®: https://github.com/rails/rails

[Tor] Torvalds: *Linux Kernel, Torvalds.* https://github.com/torvalds/linux. – Zugriff: 15.02.2013, Archiviert mit WebCite®: https://github.com/torvalds/linux

Anhang

A GUI

Ein toller Anhang.

Screenshot

Unterkategorie, die nicht im Inhaltsverzeichnis auftaucht.

Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)