Seminararbeit

im Studiengang

Allgemeine Informatik

Git

Einführung in die Befehle des verteilten Versionsverwaltungssystems Git

Referent : Fabian Berner M.Sc

Koreferent : -

Vorgelegt am : 26.04.2013

Vorgelegt von : Matthias Beyer
Matrikelnummer: CENSORED
CENSORED, CENSORED CENSORED
CENSORED

Abstract

Die Seminararbeit „Git - Einführung in die Befehle des verteilten Versionsverwaltungssystems Git“ von Matthias Beyer gibt eine grundlegende Übersicht über die Konzepte und Befehle des freien und verteilen Versionsverwaltungssystems „Git“. Die Arbeit bedient sich dabei verschiedener wissenschaftlicher Quellen aus der ACM Digital Library, sowie der offiziellen Projektseite im Internet und der Linux-Manpage des Programms. Ziel der Arbeit ist es dabei nicht, ein Tutorial für die Benutzung des Systems zu sein. Die Arbeit gibt eine Übersicht über die Möglichkeiten von Git, wobei auch die fortgeschritteneren Konzepte kurz angesprochen, allerdings nicht ausgeführt werden.

This term paper „Git - Einführung in die Befehle des verteilten Versionsverwaltungssystems Git“, written by Matthias Beyer gives a basic overview over the concepts and commands of the free, distributed version control system „Git“. This paper uses several scientific sources from the ACM Digital Library as well as the official project website and the linux-manpage of the program. The aim of this paper is not to provide a tutorial on how to use the system. It rather gives an overview what is possible with Git, while advanced concepts are addressed, but not implemented.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

VCS Version Control System

SVN Subversion

1 Einleitung

Der Siegeszug der Computer und die Verfügbarkeit von Rechnern hat nicht nur Vorteile gebracht. Jede dieser Rechenmaschinen muss programmiert werden. Mit der steigenden Komplexität der Rechner wurde auch die Software, die auf diesen verwendet wird immer komplexer. Wenn früher Großrechner mit wenigen tausend Zeilen Programmcode programmiert waren, so laufen heute selbst auf kleinsten Geräten wie MP3-Playern Betriebssysteme, deren Entwicklung mehrere Millionen Quellcodezeilen mit sich zieht.

Aber nicht nur die Verwaltung der großen Menge an Quellcode ist zum Problem geworden, sondern auch die Tatsache, das Rund um die Uhr auf der ganzen Welt Menschen Code zu einem Programm beitragen können wollen. Hierfür ist als bekanntestes Beispiel der Linux-Kernel zu nennen, die Open-Source-Alternative zu Microsoft Word, Open Office oder Libre Office, das Android-Projekt, Facebooks HipHop-Compiler, Ruby on Rails, die Desktopumgebung KDE und viele weitere Bibliotheken, Programme und Plattformen werden von hunderten, manche von tausenden Entwicklern tagtäglich bearbeitet, verbessert und verändert. Es werden Funktionalitäten hinzugefügt, Fehler verbessert, Quellcode optimiert, umgeschrieben oder gelöscht. All das muss verwaltet werden und um dies zu vereinfachen, gibt es Versionskontrollsysteme (VCS, engl. für Version Control System) wie CVS, Subversion (SVN), Perforce, BitKeeper, Mercurial und selbstverständlich Git.

> *„Version Control systems have become an indispensable tool for organizations that maintain digital resources. Often used for maintaining source code, Version Control is also used for text documents, engineering diagrams, content management, and wikis.“ Quelle: [Fra11]*

Git wird von sehr vielen großen Unternehmen aber auch und vor allem von Open Source Software Projekten verwendet.

> *„ Apple, Google, Microsoft, IBM, Facebook, Linux, Perl, Eclipse, Android, and Ruby on Rails all use Git. “* Quelle: [LJW13]

2 Grundlagen

2.1 Umgang mit der Kommandozeile

Um die Arbeitsweise des hier vorgestellten Versionskontrollsystemes optimal darstellen zu können, wird in dieser Arbeit die Kommandozeile zur Kontrolle des Programmes verwendet. Es gibt für Git mehrere grafische Lösungen, welche allerdings zum Erlernen der Softwareverwaltung mit Git nicht optimal sind.

In folgender Arbeit werden Kommandozeilenaufrufe wie folgt dargestellt:

Listing 2.1: Kommandozeilenaufruf Beispiel

```
1 programmaufruf # Kommentar
```

Die verwendete Kommandozeile ist Bash.

2.2 Theorie zu verteilten Versionskontrollsystemen

Git ist ein *verteiltes* Versionskontrollsystem. Dies bedeutet, dass es keinen zentralen Server gibt, auf welchem die Änderungen verwaltet werden. Der Vorteil gegenüber zentralisierten Systemen ist Unabhängigkeit. Daten sind immer lokal vorhanden und müssen nicht aufwändig über einen Server bezogen werden. Außerdem sind Backups nicht nötig, da jeder Entwickler im Team die Änderungen von den jeweils anderen Entwicklern mit beziehen und somit automatisch auch Backups anlegt.

2.3 Wichtige Begriffe

2.3.1 Repository

Das Repository ist das Projektverzeichnis eines Projektes mit allen Daten, die zu einem Projekt gehören. Dazu gehören die Quellcodedateien, Lizenzdateien, Build-Dateien, Readme-Datei, Dokumentation und Installationshinweise.

2.3.2 Commit

Ein Commit ist eine Menge von Änderungen, welche dokumentiert wird. Dazu gehört eine Commit-Nachricht, also eine Beschreibung, was sich geändert hat, sowie eine Commit-Hashsumme. Letztere ist eine eindeutige Identifikation für ein Commit.

2.4 Theorie zu Branching und Merging

„ Wenn mehrere Entwickler mit Git an derselben Software arbeiten, entstehen Verzweigungen im Commit-Graphen. “ Quelle: [RP12, S. 45]

Branching ist ein wichtiges Konzept von Git. Branches entstehen durch verteiltes Arbeiten, durch Änderungen, die nicht einfach in den normalen Workflow einfließen können und durch Versionierung.

Ein Branch ist eine Aneinanderreihung von Commits und basiert immer auf einem Commit. Es werden weitere Commits auf einem bestimmten Commit aufgebaut. Dadurch entsteht ein neuer Branch. Branches können zusammengefügt werden. Dieser Vorgang wird „merge“ oder „merging“ genannt. Bei diesem Vorgang kann es passieren, dass in einem Branch A eine Änderung getätigt wurde, welche in einem Branch B anders durchgeführt wurde. Wenn diese beiden Branches nun gemerged werden sollen, gibt es einen Konflikt.

2.5 Warum ist Git besser als andere Versionskontrollsysteme?

Git ist schnell. Geschwindigkeitsvergleiche der wichtigsten Operationen zwischen Git und SVN zeigen dies deutlich. Außerdem ist Git ressourcenschonend und sicherer als andere VCS (vgl. [Git13]).

In folgender Tabelle sind die Zeiten in Sekunden angegeben. SVN wurde im Best-Case Szenario getestet, also mit einem Server ohne Load und 80MB/s Bandbreite zur Client-Maschine. In der Praxis sind solche Werte meist nicht realisierbar, wobei Git von solchen Dingen nicht beeinflusst wird (nach [Git13]).

Tabelle 1: Geschwindigkeitsvergleich zwischen Git und SVN

Operation	Beschreibung	Git	SVN	Multiplikator
Commit Files	Add, commit and push 113 modified files (2164+,2259-)	0.64	2.60	4x
Commit Images	Add, commit and push 1000 1k images	1.53	24.70	16x
Diff Current	Diff 187 changed files (1664+,4859-) against last commit	0.25	1.09	4x
Diff Recent	Diff against 4 commits back (269 changed/3609+,6898-)	0.25	3.99	16x
Diff Tags	Diff two tags against eachother	1.17	83.57	71x
Log 50	Log of the last 50 commits (19k of output)	0.01	0.38	31x
Log All	Log of all commits (26056 commits, 9.4M of output)	0.52	169.0	325x
Log File	Log of the history of a single file (483 revs.)	0.60	82.84	138x
Update	Pull of Commit A scenario (113 files changed, 2164+, 2259-)	0.90	2.82	3x
Blame	Line annotation of a single file	1.91	3.04	1x

Quelle: [Git13]

3 Einführung in die Befehle des verteilten Versionsverwaltungsystems Git

3.1 Ein Git-Repository einrichten

3.1.1 Git konfigurieren

Um Git zu benutzen ist es ratsam, jedoch nicht notwendig, es zu konfigurieren. Dazu reicht die Angabe von Namen und Email-Adresse, da diese später in jeder Commit-Nachtricht stehen.

Listing 3.1: Git konfigurieren

```
1 git config --global user.name "Max␣Mustermann"
  git config --global user.email "max@mustermail.tld"
```

Die Kommandozeilenaufrufe bekommen die Option, global zu agieren, um diese Konfiguration für jedes Projekt zu übernehmen. Projekte können allerdings auch einzeln konfiguriert werden.

3.1.2 Git init

Um ein Projekt mit Git anzulegen, wird ein ganz normaler Ordner erstellt, in welchem das Projekt abgespeichert werden soll. Dann wird in diesen gewechselt und

Listing 3.2: Git initialisieren

```
git init
```

ausgeführt. Nach einer kurzen Ausgabe erscheint wieder eine Eingabeaufforderung. Ein neues Git Projektverzeichnis wurde angelegt. Auf den ersten Blick hat sich nichts an dem Ordner verändert. Git speichert alle Änderungen in einem versteckten Verzeichnis innerhalb des „Project root“, dem Verzeichnis, in welchem das komplette Projekt liegt.

3.2 Änderungen hinzufügen und committen

Git „weiß“, wenn das Projektverzeichnis erstellt wurde noch nichts von Dateien, die es beobachten soll. Außerdem wird eine Datei wenn sie beobachtet wird noch nicht versioniert. Dazu gibt es die Befehle „add“ und „commit“, welches die grundlegendsten Befehle von Git sind, ohne diese lässt sich Git nicht verwenden.

3.2.1 git add

Mittels add-Befehl muss Git angewießen werden, eine Datei zu beobachten, (zu „tracken“) beziehungsweiße diese auf den „Index“ zu schreiben. Dieser Vorgang wird auch „staging“ genannt. Er wird außerdem verwendet um mehrere Dateien zu einem „Commit“ hinzuzufügen.

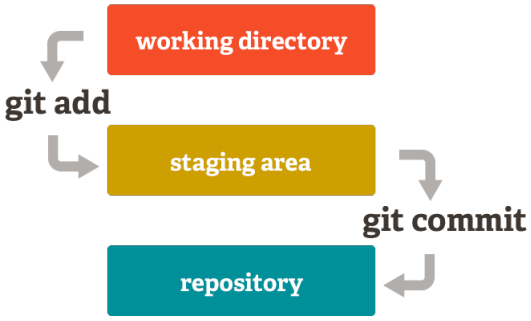
Listing 3.3: Eine Datei zum Index hinzufügen

```
1 git add ./pfad/zur/datei.txt
```

3.2.2 git commit

Mit dem commit-Befehl wird ein „Commit“ geschrieben. Ein Commit ist eine Menge an Änderungen. Diese sollten idealerweiße zusammen gehören und zusammen auch einen Sinn machen. Mittels dieses Befehles wird eine Menge von Änderungen aus der Staging-Area ins Repository übernommen.

Abbildung 1: Vom Working directory zum Repository



Quelle: [Git13]

3.3 Branching

Das Erstellen von Entwicklungszweigen ist ein zentraler Bestandteil von Git. Entwicklungszweige (Branches) sind Reihen von Commits, die aufeinander Folgen und auf einem bestimmten Commit basieren. Es können beliebig viele Branches angelegt werden. Ein Branch hat einen Namen, dieser zeigt immer auf das neuste Commit des Branches. Der aktuelle Branch bekommt außerdem den Zeiger „HEAD“. Commits vor dem aktuellen HEAD können per HEAD~1, HEAD~2 usw. referenziert werden (letztes Commit und vorletztes Commit respektiv vor dem aktuellen HEAD).

Mittels des checkout-Befehls kann der Branch gewechselt werden.

Listing 3.4: Einen Branch auschecken

```
1 git branch anderer-branch # 'anderer-branch' anlegen
  git checkout anderer-branch # Branch wechseln
```

(Der checkout-Befehl kann aber auch direkt zu einem Commit springen oder auch zu einem Tag.)

3.3.1 Warum Branching wichtig ist

Git ist beim Erstellen eines Branches von Haus aus sehr schnell. Dies hat zur Folge, das Branches sehr intensiv eingesetzt werden können. Das verteilte Arbeiten mit Git basiert genau auf diesen Vorraussetzungen. Jeder Entwickler kann einen Branch für seine Arbeit anlegen, kann aber auch leicht zu einem anderen Branch wechseln, falls kurzfristig Änderungen oder Korrekturen nötig sind, ohne dabei seine aktuelle Arbeit rückgängig machen zu müssen. Somit können Fehlerkorrekturen auf einem Release A einer Software basierend angelegt werden, obwohl schon intensiv an Release B gearbeitet wird.

3.3.2 Branches effizient verwenden

Am effizientesten sind Branches, wenn für jedes Feature, welches in eine Software eingebaut werden soll, ein Branch angelegt wird. Dabei kann jeder Feature-Branch weitere Unterbranches haben. Eventuell können Fehlerkorrekturen, welche in einem Featurebranch gemacht wurden, in andere Featurebranches übernommen werden.

3.3.3 Branches mergen

Die Commits eines Branches können in einen anderen Branch übernommen werden. Dieser Vorgang wird „merge“ genannt.

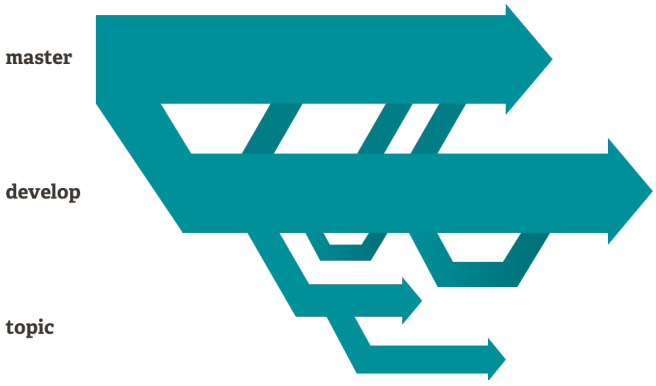
Listing 3.5: Einen Branch mergen

```
git merge anderer-branch
```

> *„Git ist sehr gut darin, Änderungen an Programmquelltexten zusammenzuführen, wenn verschiedene Entwickler verschiedene Stellen darin bearbeitet haben. Dies funktioniert oftmals selbst dann, wenn betroffene Dateien verschoben oder umbenannt wurden.“[RP12, S. 55]*

Leider kann Git nicht immer alle Konflikte auflösen. Wenn zwei Entwickler die gleiche Stelle in einem Quellcode unterschiedlich verändert haben, weiß Git nicht, welche Änderung es übernehmen soll. Git kann angewießen werden, immer die eigenen Änderungen zu behalten, sowie immer die Änderungen des anderen zu übernehmen. Es existieren außerdem noch weitere Algorithmen um

Abbildung 2: Branching und Merging



Quelle: [Git13]

Änderungen zu mergen. Diese sind der Dokumentation von Git zu entnehmen.

3.3.4 Tags hinzufügen

Git unterstützt das Taggen von Commits. Tags können als Branches gesehen werden, die sich nie bewegen. Ein Tag stellt eine Markierung eines Commits dar. Zum Beispiel kann ein Commit als Release markiert werden.

Listing 3.6: Einen Tag anlegen

```
1 git tag -a v1.0
```

Tags sollten nicht verschoben oder geändern werden, zumindest nicht nachdem ein Tag in einem anderen Repository auftauchen konnte, da dies sonst zu Fehlern bei anderen Entwicklern führen könnte.

3.4 Änderungen rückgängig machen

Ein wichtiges Konzept der Versionskontrolle ist es, Änderungen ohne großen Aufwand rückgängig machen zu können. Dazu existieren in Git zwei Kommandos.

Mittels des revert-Befehls lässt sich ein Commit rückwärts anwenden, also alle Zeilen die hinzugekommen sind werden entfernt und alle Zeilen die entfernt wurden werden wieder hinzugefügt. Dabei kann es zu Konflikten kommen, wenn die früher hinzugefügten Zeilen inzwischen wieder verändert wurden.

Außerdem existiert der reset-Befehl, welcher die Versionshistorie bis zu einem angegebenen Punkt löschen kann.

Listing 3.7: Änderungen rückgängig machen

```
1 git revert <commit-hash>
  git reset HEAD~5 --hard
```

Beim reset ist zwischen hartem und weichem Zurücksetzen zu unterscheiden. Hartes zurücksetzen löscht die Commits aus der Historie, sowie die Änderungen aus dem Arbeitsverzeichnis. Standardmäßig wird „–soft“ verwendet. Diese Option löscht die Commits, belässt die Änderungen aber im Arbeitsverzeichnis. Nun gibt es die Möglichkeit neue Commits anzulegen.

3.5 Mit entfernten Repositories arbeiten

Um mit einem entfernten Repository zu arbeiten, braucht ein Nutzer die nötigen Schreibrechte auf dem Server und muss zu Git einen „Remote“ hinzufügen.

Nun kann mittels dem push-Befehl das Repository auf den Server gepushed werden. Wird mit mehreren Personen im Team auf dem Server gearbeitet, ist es notwendig, erst die neusten Änderungen vom Server holen und dann zu pushen, da es sonst zu Konflikten kommt. Die Änderungen müssen immer erst geholt werden, da ein Server niemals in der Lage sein wird Änderungskonflikte zu lösen. Mittels dem fetch-Befehl werden Änderungen geholt, müssen dann aber noch von Hand gemerged werden. Das kann auch in einem Schritt gemacht werden, indem der pull-Befehl verwendet wird.

Listing 3.8: Einen Remote hinzufügen

```
git remote add mein-remote https://meinedomain.tld:
    meinrepo.git
git push mein-remote mein-branchname
```

Wie die genaue URL für das Git Repository ist, hängt vom verwendeten Server ab.

3.6 Weitere Funktionalitäten

Git verfügt außer den bis hier genannten Befehlen noch über weitere. Nachfolgend werben ein paar wenige davon vorgestellt.

3.6.1 Der rebase-Befehl

Mittels des rebase-Befehls lässt sich die Versionshistory umschreiben. Es gibt die Möglichkeit, Commits zusammenzufassen oder aufzusplitten. Auch ist es möglich einen Branch auf einen anderen Branch „umzupflanzen“. Ist die Arbeit an einem Feature für eine Software auf Version 1 basierend gestartet, aber inzwischen ist Version 2 der Software veröffentlicht worden, gibt es die Möglichkeit den Branch, mit welchem das Feature geschrieben wird, einfach auf die neue Version zu verschieben.

Dies sollte allerdings nur geschehen, wenn die Änderungen noch nicht auf einen Remote gepushed sind, da es sonst zu Fehlern kommen kann, wenn andere Entwickler den Branch für weitere Arbeiten verwendet haben.

3.6.2 Änderungen ansehen

Unter Git gibt es zwei Möglichkeiten, die Historie anzusehen. Der log-Befehl listet Änderungen auf. Er versteht eine große Anzahl an Optionen.

Listing 3.9: Den Log anzeigen

```
git log --oneline --graph --all v1.0.0..v1.2.0
```

Dieser Kommandozeilenaufruf zum Beispiel zeigt alle Commits (einzeilig) als Branch-Graph zwischen dem Tag 'v1.0.0' und dem Tag 'v1.2.0'.

Außerdem existiert der show-Befehl, mittels welchem ein bestimmtes Commit angesehen werden kann.

Listing 3.10: Ein Commit anzeigen

```
1 git show <commit-hash>
```

3.6.3 Der bisect-Befehl

Mittels des bisect-Befehls lassen sich Fehler, welche sich durch Commits eingeschlichen haben, leicht auffinden. Dazu muss das „bisecting“ gestartet und ein Commit als funktionierende Version und eines als nicht funktionierende Version der Software markiert werden. Mittels des bisect-Befehls wird nun in der Historie gesucht und Commits als funkiotnierend oder nicht funktionierend markiert, bis das Commit, welches einen Fehler eingeschleußt hat, gefunden ist.

Um den bisect-Befehl anwenden zu können bedarf es einer sehr sauberen Arbeitsweiße mit Git. Oft muss jedes Commit eine lauffähige Software darstellen, damit diese auch getestet werden kann.

3.6.4 git cherry-pick

Mittels des cherry-pick-Befehls lassen sich Commits kopieren. Er bietet die Möglichkeit ein Commit von einem Branch auf einen anderen Branch zu übertragen, ohne den Branch mergen zu müssen. Dies kann hilfreich sein um Fehlerkorrekturen zu übernehmen. Bei einem cherry-pick wird ein komplett neues Commit, mit gleichen Änderungen sowie gleichen Metainformationen (Commit-Message) erstellt.

3.6.5 Dateien ignorieren

Oft werden nicht alle Dateien zu einem Git Repository hinzugefügt. Bei einem in C geschriebenen Projekt sind das zum Beispiel die Object-Files, bei einem latex-Projekt unter anderem die aux-Dateien. Diese können ignoriert werden, indem sie in einer Datei aufgelistet werden. Diese Datei heisst „gitignore“ und kann wie folgt aussehen:

Listing 3.11: Die gitignore-Datei

```
1 cat .gitignore
  *.pdf
3 *.aux
  *.out
5 *.bbl
  *.blg
7 *.toc
```

```
*.log
*.idx
*.lof
*.lot
```

Eine gitignore-Datei kann mehrmals existieren, zum Beispiel in jedem Unterverzeichnis des Projektes.

Außerdem gibt es eine Globale gitinore-Datei anlegen, welche dann im Home-Verzeichnis eines Nutzers liegt und „.gitignore_global“ heisst (unter Linux).

4 Ausblick

4.1 Weitere Befehle

Git ist mittels folgender Befehle an andere VCS anbindbar:

- git-svn für Anbindung an SVN
- git-hg für Anbindung an Mercurial

Hier eine Auflistung weiterer Befehle von Git:

- git clean - Um unbeobachtete (untracked) Dateien vom Repo zu entfernen
- git submodule - Submodule initialisieren oder updaten
- git blame - Welche Änderung wurde wann von welchem Autor geschrieben

Quelle: [LT13]

4.2 Internet-Tutorials

Für Git existieren Tutorials im Internet und Bücher. Für Anfänger und Fortgeschrittene sind folgende empfehlenswert:

- `git-scm.com/book` - Pro Git Buch, veröffentlicht unter Creative Commons
- `de.gitready.com` - Deutsches Tutorial für Anfänger und Fortgeschrittene
- git - Erschienen im dpunkt.verlag und zur Erstellung dieser Arbeit verwendet

4.3 Grafische Oberflächen für Git

Für Git existieren inzwischen ein paar grafische Oberflächen.

- git-gui - Die Standardoberfläche für Git
- tig - ncurses Oberfläche für die Kommandozeile
- SourceTree - Grafischer Client für Mac und Windows

Quelle: [Git13]

5 Fazit

> *„ It [anm.: Git] is extremely flexible and guarantees data integrity while being powerful, fast and efficient. “ Quelle: [Sty11]*

Git ist ein Tool um Entwicklung, im besonderen verteilte Entwicklung im Sinne von Projekten mit mehreren Entwicklern, zu unterstützen und zu beschleunigen. Ist die grundlegende Arbeitsweise von Git durch den Anwender verstanden, fällt auch die Anwendung leicht und es unterstützt den Workflow in erheblichem Maße.

Viele Entwickler sind noch der Auffassung, das Versionskontrolle nur vom Projektmaintainer erfolgen sollte. Doch Versionskontrolle muss sehr viel feiner geschehen, als nur einzelne Versionen einer Software zu markieren und Fehlerkorrekturen zu übernehmen. Versionskontrolle sollte durch jeden Entwickler erfolgen und Git ist das Tool, mit dem jeder Entwickler sehr einfach Software versionieren und Änderungen kontrollieren kann.

Literaturverzeichnis

[Fra11] Fraser, Neil: Version control workshop. In: *Proceedings of the 11th ACM symposium on Document engineering*. New York, NY, USA : ACM, 2011 (DocEng '11). – ISBN 978–1–4503–0863–2, 267–268

[Git13] Git Projekt: *Git*. Website. `http://git-scm.com`. Version: Mai 2013. – Projektseite von git

[LJW13] Lawrance, Joseph ; Jung, Seikyung ; Wiseman, Charles: Git on the cloud in the classroom. In: *Proceeding of the 44th ACM technical symposium on Computer science education*. New York, NY, USA : ACM, 2013 (SIGCSE '13). – ISBN 978–1–4503–1868–6, 639–644

[LT13] Linus Torvalds, Junio C H.: *git - the stupid content tracker*. (Linux Manpage), Mai 2013. – Git version 1.8.2.3, Manpage am 21.05.2013 aufgerufen

[RP12] Rene Preissel, Bjorn S.: *Git*. 1. dpunkt.verlag, 2012. – ISBN 9783898648004

[Sty11] Styn, Henry V.: Git. In: *Linux J.* 2011 (2011), August, Nr. 208. `http://dl.acm.org/citation.cfm?id=2020786.2020790`. – ISSN 1075–3583

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

CENSORED, den 26.04.2013 Matthias Beyer