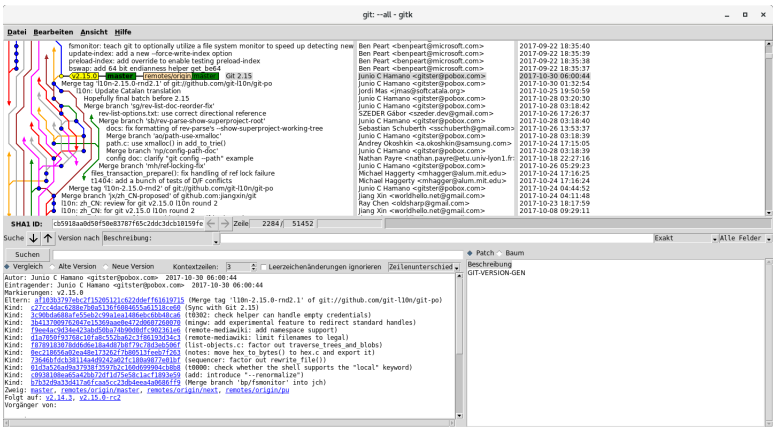

Seminararbeit im Studiengang Bachelor Informatik

Versionsverwaltung mit Git

Sascha Girrulat

MatNr.: 6702562

sascha@girrulat.de

Betreuer: Dr. Daniela Keller

LG Programmiersysteme • Informatik
FernUniversität Hagen • Universitätsstraße 11 (IZ) • 58097 Hagen

Kurzfassung

Diese Seminararbeit stellt Versionskontrollsysteme und deren Funktion im Allgemeinen und Git im Speziellen vor. Neben einem geschichtlichen Überblick wird, nach einem kurzen Exkurs in den Bereich Kollaboration, auf grundlegende Eigenschaften von Versionskontrollsystemen eingegangen. Ebenso werden wesentliche Unterschiede aufgezeigt. Dabei wird insbesondere auf Git eingegangen. Um einen praktischen Einblick in die Arbeit mit Git zu vermitteln, werden die gebräuchlichsten Befehle und Vorgehensweisen in Beispielen vorgestellt. Nachdem in groben Zügen auf das Objektmodell eingegangen wurde, werden die eingesetzten Befehle nochmal in einer Kommandoübersicht durch weitere Befehle ergänzt. Anschließend werden auch einige Einschränkungen von Git beschrieben. Um die Leistungsfähigkeit in der Praxis zu unterstreichen, wird zum Abschluss das Projekt vorgestellt, für das Git ursprünglich entwickelt wurde - Der Linux Kernel.

Insgesamt gibt diese Arbeit einen ersten Einblick in den Einsatz von Git. Sie verdeutlicht, warum Versionskontrollsysteme essenziell für den Einsatz in erfolgreichen Softwareprojekten sind.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CSSC Compatibly Stupid Source Control.

CVS Concurrent Versions System.

DVCS Distributed Version Control System.

Git The stupid content tracker.

MySc Frühe Version von CSSC.

RCS Revision Control System.

SCCS Source Code Control System.

SHA Secure Hash Algorithm.

SSH Secure Shell.

SVN Subversion.

VCS Version Control System.

1 Einleitung

Das heutige Umfeld der Informationstechnologie ist geprägt von einer Vielzahl unterschiedlicher Arten von Software. Allein das Alltagsleben stützt sich täglich auf verschiedenste Systeme, auf denen eine Softwarekomponente ihre Arbeit verrichtet. Allen voran ist sicherlich das Internet mit zahlreichen Online Plattformen zu nennen. Eine Eigenschaft, die fast alle großen Softwareprojekte gemeinsam haben ist, dass sie in Teams unterschiedlicher Größe entwickelt werden. Je nach Zusammenstellung dieser Teams kommt es hier nicht selten zu geografischen und zeitlichen Herausforderungen, um solche Systeme gemeinsam effektiv zu entwickeln. Weitere Beispiele für verteilte Softwareprojekte findet man in der Welt der Open Source Software. So ist Git[1], auf dem der Fokus dieser Arbeit liegt, ebenfalls ein Beispiel aus der Open Source Community.

Nach Jez Humble und David Farley stellen sich in Projekten mit verteilten Teams einige Herausforderungen, auf die in Kapitel 2 Bezug genommen wird [JD10, S. 26, 33]:

- Ein neues Teammitglied bekommt seinen Arbeitsplatz und benötigt schnellen Zugriff auf meist verschiedene Versionen der entwickelten Software, möchte diese erzeugen und in einem definierten Umfeld ausführen. Auch gibt es eventuell verschiedene Umgebungen[2] und der Entwickler muss neben der eingesetzten Version feststellen, welche Unterschiede zwischen den Umgebungen bestehen [JD10, S. 26].
- Was macht eine spezielle Version einer Software aus? Wie kann eine konkrete Ausprägung einer Software, die sich im produktiven Einsatz befindet, reproduziert werden? Reproduzierbarkeit kann z.B. nötig sein, um einen akuten Fehlerfall nachzustellen [JD10, s. 33].
- Was wurde durch wen und wann geändert? Und aus welchen Gründen? Auch das kann für den Fall einer Fehlersuche von Vorteil sein. Darüber hinaus kann mit Beantwortung dieser Frage die Projekthistorie aus Sicht der Entwicklung reproduziert werden [JD10, S. 33].

Diese Fragen sind natürlich nicht auf große Projekte beschränkt, sondern gewinnen hier vermehrt an Wichtigkeit. Vergleichbare Problemstellungen finden sich aber genauso in Entwicklungsprojekten, an denen nur wenige Personen beteiligt sind.

In Kapitel 2 wird vorgestellt, wie Versionskontrollsysteme bei der Lösung dieser Problemstellungen helfen können und was ihre eigentlichen Aufgaben sind. Hier wird, neben einer grundlegenden Definition und einem geschichtlichen Überblick auf Gemeinsamkeiten und Vorgehensweisen verschiedener Versionskontrollsysteme eingegangen. In Kapitel 3 wird

[1] https://git-scm.com/

[2] In der Softwareentwicklung wird häufig von produktiven Umgebungen gesprochen. Damit ist eine Zusammenstellung aus verschiedenen Systemen in definierter Konfiguration gemeint, auf denen die entwickelte Software produktiv eingesetzt wird. Um Fehlentwicklungen zu vermeiden, wird die Software häufig in sogenannten Entwicklungsumgebungen, die der Produktionsumgebung nachempfunden sind, getestet [JD10, S. 49, 250].

das Versionskontrollsystem Git vorgestellt. Hierbei werden zunächst die Grundlagen beschrieben, anschließend wird eine praktische Anwendung vorgestellt. Aufbauend darauf wird auf grundlegende Workflows und die Arbeitsweise von Git eingegangen. Um einen weiteren Praxisbezug herzustellen sowie weitere Vor- und Nachteile von Git herauszustellen, werden in Kapitel 4 zuerst Einschränkungen von Git und abschließend der Einsatz von Git im Linux Kernel Projekt betrachtet.

2 Versionsverwaltung

2.1 Definition

Versionsverwaltungssysteme sind auch bekannt als Versionskontrollsysteme (Version Control System), Quellcode Verwaltung (Source Control) oder Revisionskontrollsysteme (Revision Control System). Mit diesen Begriffen sind Systeme gemeint, die es Entwicklern, Teams oder Organisationen erlauben, eine vollständige Historie mit allen Änderungen am Quellcode ihrer gemeinsam entwickelten Software zu verwalten. Ausschlaggebend ist hierbei, dass für alle Nutzer transparent wird, wer wann welche Änderungen durchgeführt hat und vor allem warum diese Änderungen durchgeführt wurden. Eine weitere wichtige Eigenschaft ist, dass verschiedenen Teams das gleichzeitige Arbeiten an verschiedenen Teilen der Software ermöglicht wird, ohne sich gegenseitig zu behindern[3]. [JD10, S. 381]

In den nachfolgenden Abschnitten wird nicht im Detail auf alle existierenden Versionskontrollsysteme eingegangen. Vielmehr werden wenige Systeme vorgestellt, um einen Überblick über die grundlegende Entwicklung von Versionsverwaltungssystemen zu vermitteln.

2.2 Geschichtliche Entwicklung

Das erste Versionskontrollsystem namens SCCS enstand 1972 und wurde von Marc J. Rockkind bei Bell Labs geschrieben[4] [JD10, S. 382]. Ab diesem Zeitpunkt enstand eine Vielzahl verschiedener Versionskontrollsysteme. Als Alternative zu dem kommerziellen SCCS folgte Anfang 1980 das von Walter F. Tichy an der Purdue University entwickelte erste Open Source Versionskontrollsystem Revision Control System (RCS) [WF91, Fou13]. Ross Ridge veröffentlichte 1993 mit einer Beta Version von MySc einen freien Ersatz für SCCS. In späteren Versionen wurde MySc in Compatibly Stupid Source Control (CSSC) umbenannt[Fou01, Fou93]. Alle drei Systeme finden in der Praxis nur noch wenig Anwendung. Daher wird diesbezüglich nicht auf weitere Details eingegangen.

2.2.1 CVS

Das 1986 durch Dick Grune veröffentlichte Concurrent Versions System (CVS) war das erste freie Versionskontrollsystem mit einem zentralen Repository. Dies wurde erreicht, indem RCS mit Hilfe eines Wrappers um eine Client-/Serverkomponente erweitert wurde. Dadurch war es erstmals möglich, dass mehrere Entwickler gleichzeitig an einem Repository und konkurrierend an denselben Dateien arbeiten konnten. Neben den innovativen Ansätzen gab es hier aber noch einige technische Einschränkungen, die ein kollaboratives Arbeiten erschwerten. Beispielsweise war die Nutzung des verbrauchten Speicherplatzes nicht optimal. Das Erzeugen von Abzweigungen (Abschnitt 2.4.3) wurde durch einfaches

[3] Dies hängt natürlich nicht nur von dem Versionskontrollsystem ab, sondern auch vom Design der entwickelten Software. Diese wird zumeist modular aufgebaut, so dass die Möglichkeit einer parallelen Entwicklung unterstützt wird.

[4] http://www.belllabs.com/

Kopieren erreicht. Dies war nicht nur zeitaufwändig, sondern verbrauchte auch entsprechend zusätzlichen Speicherplatz. Ein späteres Zusammenführen (Abschnitt 2.4.4) dieser Zweige führte daher zu Dateikonflikten und verursachte hierduch erheblichen Mehraufwand. Auch gab es keine Funktionalität, um Binärdateien zu verwalten, so dass der Speicherplatz eher ineffizient genutzt wurde. Das Erstellen von Tags wurde mit steigendem Inhalt des Repositorys ebenfalls immer zeitaufwändiger, da alle enthaltenen Dateien bearbeitet werden mussten. Die aus heutiger Sicht größte Einschränkung war aber sicher die Tatsache, dass Commits (Abschnitt 2.4.1) in das Repository nicht atomar waren. Wurde die Übertragung der Dateien in das zentrale Repository unterbrochen, so wurde dieses in einem inkonsistenten und nicht mehr nutzbaren Zustand hinterlassen und musste administrativ repariert werden. [JD10, S. 382-383]

2.2.2 SVN

Das Ziel des quasi als Nachfolger zu CVS entwickelten Versionsverwaltungssystems Subversion (SVN) war es, die technischen Einschränkungen von CVS zu beheben. Es wurde darauf geachtet, dass es als Ersatz fungieren kann und dass sich ein Umstieg sowohl aus administrativer Sicht, als auch aus Sicht eines Nutzers möglichst einfach gestaltet. Das Benutzerinterface funktioniert daher ähnlich zu CVS, so dass Entwickler sich nach einem Umstieg leichter zurecht finden. Als zentrale Neuerung sind im Gegensatz zu RCS und SCCS nicht mehr die Dateien zentraler Bestandteil der Versionierung, sondern die sogenannte SVN Revision. Jede Revision enthält einen eindeutigen Stand aller Dateien im Repository zu einem bestimmten Zeitpunkt und ist global gültig und eindeutig. Das ermöglicht direkte Vergleiche verschiedener Revisionen um festzustellen, welche Veränderungen zwischen zwei Revisionen durchgeführt wurden. Alle Änderungen, wie z.B. das Kopieren, Hinzufügen oder Entfernen von Dateien werden atomar durchgeführt. Im Gegensatz zu CVS geht die Historie einer Datei nicht verloren, wenn sie kopiert wird. Das Erstellen von Tags oder Branches wurde ebenfalls verbessert. Hierzu wurde eine Konvention eingeführt, welche drei verschiedene Verzeichnisse innerhalb eines Repositorys vorgibt:

- **trunk**: Zentraler Zweig, der die führende Version enhält. Vornehmlich werden Branches und Tags erzeugt, indem von dieser Version eine Kopie erstellt wird.
- **tags**: Verzeichnis, in dem Unterverzeichnisse als Kopie einer konkreten Revision angelegt werden. In der Regel werden Tags genutzt, um einen bestimmten Stand eines Zweiges beispielsweise mit einer Versionsnummer zu markieren.
- **branches**: Ein Verzeichnis, in dem Unterverzeichnisse als Arbeitskopien von anderen Zweigen angelegt werden. Auf diesen Abzweigungen kann unabhängig von der Quelle (z.B. *trunk*) gearbeitet werden.

Tags und Branches werden erzeugt, indem ein Verzeichnis angelegt wird und der Inhalt von z.B. *trunk* kopiert wird. So lange keine Änderungen in den erzeugten Verzeichnissen durchgeführt werden, sind diese lediglich Zeiger auf eine Menge von Objekten, die mit der Kopie der Quelle übereinstimmen. Die Trennung zwischen Tags und Branches ist nur

eine Konvention. Das führt auch zu einem der Probleme beim Arbeiten mit Subversion. Tags sind nicht eindeutig und veränderbar. So macht SVN technisch keinen Unterschied zwischen `trunk`, `tags`, `branches` oder einem beliebigen anderen Verzeichnis innerhalb des Repositories. Alle Verzeichnisse können in der gleichen Art und Weise bearbeitet werden. [JD10, S. 383-385]

Ein großer Vorteil von SVN gegenüber vorherigen Versionskontrollsystemen ist, dass mit SVN auch gearbeitet werden kann, wenn das Netzwerk nicht zur Verfügung steht. Alle Änderungen an Dateien werden zuerst auf einer lokalen Kopie durchgeführt und mit einem separaten Kommando an das zentrale Repository gesendet. Ferner seien sogenannte Externals erwähnt, die es als weitere Funktionalität ermöglichen, erstmals Inhalte von anderen Repositories einzubinden. Dies kann nützlich sein, um Abhängigkeiten zwischen Repositorys abzubilden oder Binärdateien auszulagern. [JD10, S. 384]

SVN stellt zwar eine erhebliche Verbesserung gegenüber CVS dar, hat aber aufgrund der zusätzlichen Client-/Server Komponente einige neue Probleme. So sind zwar die Aktionen auf Seiten des zentralen Repositorys atomar, aber nicht auf Seiten des Clients, so dass bei unvorhergesehenen Fehlern wieder Inkonsistenzen entstehen können. Des Weiteren verwaltet SVN mit Hilfe eines Unterordners *.svn* in jedem enthaltenen Verzeichnis dessen Verwaltungsinformationen. Das ermöglicht eine unabhängige Verwaltung aller Verzeichnisse untereinander, führt aber ggf. dazu, dass die lokale Kopie eine Summe aus unterschiedlichen Revisionen verschiedener Unterordner ist und keinen eindeutigen Versionsstand repräsentiert. Neben den genannten sind einige weitere Vor- und Nachteile von SVN in [JD10, S. 382-385] beschrieben.

2.2.3 Git

Git ist ein verteiltes bzw. dezentrales Versionskontrollsystem. Es wurde von Linus Torvalds und Junio Hamano entwickelt und ist für gängige Plattformen wie Linux, BSD, Windows u.a. verfügbar. Im Englischen wird Git als *git - the stupid content tracker* bezeichnet. Der Name Git kommt nach einem Zitat von Linus Torvalds wie folgt zustande [Org06]:

> *„I'm an egoistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'“*

Alternativ stellt Linus Torvalds, etwas scherzhaft, noch weitere Varianten als Übersetzung des Akronyms *Git* zur Verfügung [Org06]:

- *„Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronounciation of ¨get¨ may or may not be relevant.“*
- *„Stupid. Contemptible and despicable. Simple. Take your pick from the dictionary of slang.“*

- *„¨Global information tracker¨: you're in a good mood, and it actually works for you. Angels sing and light suddenly fills the room.“*
- *„¨Goddamn idiotic truckload of sh*t¨: when it breaks“*

Linus Torvalds nutzte für die Entwicklung des Linux Kernel[5] ursprünglich das kommerzielle System BitKeeper als Versionsverwaltung. Nach Unstimmigkeiten mit dem Hersteller von BitKeeper entschloss Linus Tovalds sich 2005 dazu, ein neues Versionskontrollsystem zu schreiben. Dieses sollte weit mehr als eine Alternative zu den bisherigen sein [VJ11, S. 13]. So lautete eine Aussage von Linus Torvalds zu CVS [JD10, S. 385]:

„There is no way to do CVS right“

Neben dem Beheben der Probleme, mit denen die populären Systeme CVS (Abschnitt 2.2.1) und SVN (Abschnitt 2.2.2) zu kämpfen hatten, sollte bei der Entwicklung von Git das zentrale Augenmerk auf Geschwindigkeit und Integrität liegen [VJ11, S. 13].

Die Erfolgsgeschichte von Git ist aber nicht nur in den technischen Neuerungen begründet. Schon wenige Wochen nach dem Start mit der Arbeit an Git konnten die ersten Versionen bereits Quellcode verwalten. Die grundlegenden Konzepte der anfänglichen Entwicklung sind bis heute gleich geblieben. Die spätere erfolgreiche Migration und Verwaltung des Linux Kernel Repositorys mit Git führte zu einem hohen Ansehen und einer starken Verbreitung von Git. Die Anforderungen, welche die Verwaltung des Linux Kernels mit über 1000 Entwicklern an ein Versionskontrollsystem stellt, sind immens. Allein zwischen zwei Versionen finden sich mehrere hunderttausend Änderungen in über 1000 Dateien und etliche Merges zwischen verschiedenen Branches (Abschnitt 4.2). Git ist heutzutage insbesondere aus großen Projekten kaum mehr wegzudenken. [VJ11, S. 13]

2.3 Kollaboration

Die Möglichkeit, Änderungen transparent und nachvollziehbar in einem Repository zu speichern und bei Bedarf wieder zurückzunehmen, Entwicklungsstände, die zu unterschiedlichen Zeitpunkten erzeugt wurden, zu vergleichen, zusammenzuführen und Teile davon wieder herzustellen, ermöglicht es, die Zusammenarbeit in Teams oder Organisationen erheblich zu verbessern. Beispielsweise werden alle Beteiligten damit konfrontiert, dass mehrere Personen an gleichen Inhalten zur gleichen Zeit arbeiten müssen. Das kann den Umgang mit Konflikten, die durch eine solche Zusammenarbeit entstehen können, verbessern - unabhängig davon, ob diese Konflikte technischer oder organisatorischer Art sind. Der Einsatz eines Versionsverwaltungssystems sollte als eine Vorgehensweise angesehen werden, die einen positiven Einfluss auf die Zusammenarbeit hat. Da nicht alle Systeme die gleichen Funktionen unterstützen, sollte daher bei der Auswahl des einzusetzenden Systems darauf geachtet werden, welche Eigenschaften der Zusammenarbeit gefördert werden

[5] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

sollen. Hierzu zählen nach Jennifer Davis und Katherine Daniels [JK16, S. 178] beispielsweise:

- Das Erstellen und Abzweigen von Repositories,
- das Beitragen zu Repositories bzw. das Verwalten von Beiträgen,
- das Festlegen von Prozessen oder
- das Verwalten von Berechtigungen innerhalb von Repositories.

Der Einsatz eines Versionskontrollsystems hat auch einen positiven Einfluss auf Risiken, die beispielsweise bei Änderungen an einer produktiven Softwareplattform (z.B. eines Internetportals) entstehen. Hierdurch wird es möglich, im Fehlerfall eine frühere Version der Software wieder einzusetzen. [JK16, S. 178]

Ein Versionskontrollsystem ist nicht nur alleinig zum Versionieren von Quellcode geeignet. Eine Aussage von Jez Humble und David Farley in [JD10, S. 33] lautet hierzu:

„Keep Absolutely Everything in Version Control“

Jede Datei, die benötigt wird, um Software reproduzierbar zu erzeugen und zu verwalten, sollte sich unter Versionskontrolle befinden. Das können beliebige Dateien sein, wie z.B.

- Quellcode,
- Tests,
- Skripte zum Kompilieren oder zur Datenbankverwaltung,
- Bibliotheken oder
- Konfigurationen.

So können neue Mitarbeiter schnell in Teams integriert werden und ein effizienter Start wird gefördert. Es ist ebenso wichtig, dass alle benötigten Informationen sich unter Versionskontrolle befinden und allen Beteiligten zur Verfügung stehen. Hierzu gehören neben Dokumentationen beispielsweise Projekt- und Releasepläne der Manager oder Dokumente über durchgeführte Analysen. Darüber hinaus sollten auch externe Abhängigkeiten mit verwaltet werden. So könnten hier ebenso

- DNS Zonendateien,
- Regeln für eine Firewall oder
- die Konfiguration für eine Entwicklungsumgebung

mit versioniert werden.

Im Kern kann also alles, was nötig ist, um Umgebungen (insbesondere produktive[6]) von Grund auf neu zu erzeugen, mit verwaltet werden. Erst wenn all diese Informationen und Dateien, die sich über die Zeit verändern, in einem Versionskontrollsystem verwaltet werden, ist es möglich, das Projekt oder die Software von bzw. zu einem beliebigen Zeitpunkt in der Historie wieder herzustellen. [JD10, S. 33]

2.4 Allgemeine Grundlagen

Aus den vorhergehenden Abschnitten lässt sich zusammenfassend feststellen, dass es aus Sicht der Benutzer für den gemeinsamen Zugriff auf Dateien bestimmte Anforderungen gibt. Nach [JCL17, S. 37] bestehen Forderungen nach **Aktualität**, **Konsistenz**, **Isolation**, **Integration**, **Gruppenwahrnehmung** und **Nachvollziehbarkeit**. So sollen Entwickler in der Lage sein, immer an einer gültigen Fassung einer Datei zu arbeiten, als sei diese Datei exklusiv für sie reserviert, sowie die eigenen Änderungen daran wieder so zu integrieren, dass eine Nachvollziehbarkeit mit einer Änderungshistorie, die wenigstens Namen und Datum enthält, gewährleistet ist. Wie sich mit Git eine Umsetzung dieser Forderungen in der Praxis darstellt, wird in den folgenden Unterkapiteln gezeigt. Vorab werden noch einige, in diesem Zusammenhang relevante, Grundbegriffe erläutert.

2.4.1 Commit

Werden Veränderungen am Repository bzw. an den darin enthaltenen Dateien durchgeführt, werden diese als Commits gespeichert. Hierzu werden die einzelnen Veränderungen an Dateien oder Metadaten wie z.B. Löschen, Hinzufügen oder Verändern von Dateiberechtigungen durch den Autor, mit verschiedenen Kommandos zusammengefasst. Diese werden mit Namen, Zeitpunkt und Beschreibung dem Repository als Commit hinzugefügt. [VJ11, S. 20]

Unter Git referenziert ein Commit immer einen eindeutigen Zustand aller verwalteten Dateien. Diese Referenz enhält eine inhaltsabhängige, eindeutige, 160-Bit lange Prüfsumme, die auch als *Commit-ID* bezeichnet wird (Abschnitt 4.2). Die inhaltsabhängige Prüfsumme gewährleistet auch, dass Inhalte nachträglich nicht unbemerkt verändert werden können. Die Prüfsumme wird mit dem *Secure Hash Algorithm* SHA-1 erzeugt. [VJ11, S. 20-21]

2.4.2 Tag

Die meisten Versionskontrollsysteme benutzen Markierungen (engl. Tag), um bestimmte Stände des Repositories mit Namen zu versehen. Üblicherweise werden zur Bezeichnung von Markierungen Versionsstände genutzt, wie z.B. 2.6.12-rc2 (Abschnitt 4.2), v1.0 oder ähnliches. [SB14, S. 48]

[6] Es gibt sicher eine Vielzahl von Szenarien, die es als wirtschaftlich arbeitendes Unternehmen erfordern, zu so etwas in der Lage zu sein - ausgenommen vielleicht Anbieter einschlägiger Suchmaschinen, die beispielsweise nicht vom Abbrennen eines einzelnen Datenzentrums abhängig sind.

Unter Git sind Tags symbolische Namen für Referenzen zu einzelnen Commits. Es gibt hier zwei verschiedene Arten von Commits. Zum Einen leichtgewichtige Tags (*lightweight tags*), die lediglich die Versionsnummer enthalten sowie ausführlichere Tags (*annotated*), die darüber hinaus noch Metadaten wie Autor, Beschreibung oder eine GPG-Signatur[7] beinhalten. [VJ11, S. 21]

2.4.3 Branch

Ein Repository enthält in der Regel einen zentralen Zweig (engl. Branch), auf dem die Entwicklungsarbeit stattfindet. Ein Abzweig an einer bestimmten Stelle zu einem bestimmten Zeitpunkt erzeugt einen neuen Branch. Dieser Abzweig kann von dem zentralen oder einem anderen Branch erfolgen. Wie diese Branches erzeugt werden, ist abbhängig vom eingesetzten Versionskontrollsystem. So wird unter SVN oder CVS der Branch mittels eines Kopiervorgangs, unter Git jedoch über eine Referenz erzeugt. Häufig werden Branches verwendet, um Features zu entwickeln, Releases zu erzeugen oder Bugfixes an bestehenden Versionen durchzuführen [VJ11, S. 21]. Um effizient mit Branches umzugehen, gibt es verschiedene Strategien. So wird von Jez Humble und David Farley in [JD10, S. 408-412] auf Releasebranches, Featurebranches, Teambranches oder eine Entwicklung mit nur einem Hauptbranch[8] eingegangen.

2.4.4 Merge

Wenn mehrere Entwickler an einem Repository arbeiten (Abschnitt 2.3), entstehen häufig parallele Abzweige von einer gemeinsamen Basis. An diesen wird gleichzeitig gearbeitet, sodass diese Zweige auseinander laufen. Das spätere Wiederzusammenführen bezeichnet man als Merge. Wie auch andere Versionskontrollsysteme bietet Git hier mit `merge`, `mergetool`, `rebase` und `cherry-pick` eine umfangreiche Unterstützung [RB17, S. vii]. Beim Zusammenführen von Zweigen ensteht ein sogenannter Merge-Commit (Abbildung 2.1). Dieser hat seinen Ursprung in den beiden vorhergehenden Knoten der Zweige *topic* und *master*.

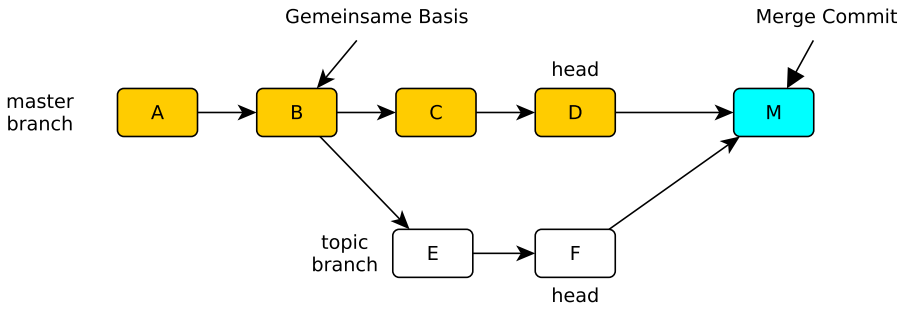

Abbildung 2.1: Zusammenführen zweier Branches. Angelehnt an [VJ11, 83]

[7] GnuPG ist ein Werkzeug, mit dem Daten sowohl ver-/entschlüsselt werden können, als auch zum Erzeugen und Überprüfen von Signaturen. Viele Programme nutzen GnuPG für eigene Zwecke. So kann auch Git GnuPG benutzen, um Tags zu signieren und die Identität des Authors sicherzustellen. [WFgCG17]

[8] Umgangssprachlich werden verschiedene Begriffe für diesen Branch benutzt. So sind Begriffe wie z.B. Trunk, Masterbranch oder Mainlinebranch nicht unüblich. Gemeint ist hiermit aber immer der zentrale Branch, der standardmäßig von dem genutzen Versionskontrollsystem vorgesehen ist. Für SVN z.B. ist das das Verzeichnis *trunk* und unter Git einfach die Referenz auf den Namen *master*.

Komplexere Szenarien erlauben es, mit Git auch Merges aus mehr als zwei Branches durchzuführen, sogenannte *three-way-merges* oder *octopus-merges* [VJ11, S. 87]. Git ermöglicht auch, Zweige aus verschiedenen Repositories zusammenzuführen. So gibt es nach [RB17, 3] noch die Möglichkeit, Repositories nach spezielleren Funktionen zu unterteilen. So kann ein dediziertes *Blessed Repository* ausschließlich für das Erstellen von Versionen der erzeugten Software genutzt werden. Die Änderungen können aus einem *Fork Repository* importiert werden. Das *Fork Repository* dient zur Entkopplung des Haupt-Repositories. So können größere Umbauten, Merges oder Features unabhängig entwickelt werden [RB17, S. 123]. Eine vergleichbare Vorgehensweise findet man im Repository des Linux Kernels [Lin15] (Abschnitt 4.2).

2.4.5 Repository

Das Repository ist ein Datenspeicher, in dem gemeinsame Dateien eines Projekts durch das Versionskontrollsystem verwaltet werden. Der Benutzer kann hier Dateien beispielsweise hinzufügen, entfernen oder verändern und in einem Commit zusammenfassen. Das Versionskontrollsystem erzeugt für alle in Commits zusammengefassten Änderungen eine Historie. [JCL17, S. 38].

2.4.6 HEAD

Bei *HEAD* handelt es sich um eine Referenz, die auf den letzten Commit im aktuellen Repository bzw. Branch verweist. Hier muss zwischen verschiedenen Versionskontrollsystemen unterschieden werden. So ist unter Git die *HEAD*[9] Version eine symbolische Referenz auf die letzte Version des lokal genutzten Branches. Diese ist also abhängig von der gerade gewählten Version im Working Tree (Abschnitt 2.4.9) [VJ11, S. 20]. Für SVN hingegen bezeichnet *HEAD* die letzte veröffentlichte Version im Repository.

2.4.7 Checkout

Bei vielen Versionskontrollsystemen gibt es einen gleichlautenden Befehl. Um Missverständnisse zu vermeiden, muss man hier zwischen den anderen erwähnten Versionskontrollsystemen und Git unterscheiden. Der Begriff *checkout* beschreibt im Rahmen von Versionskonrollsystemen wie z.B. CVS oder SVN (Abschnitt 2) das Erstellen einer lokalen Kopie eines entfernten Repositorys [VJ11, S 137].

Im Kontext von Git (Abschnitt 3) beschreibt *checkout* zum Einen das Wechseln zu einem konkreten Commit. Das kann unter Git ein Branch, Tag oder ein beliebiges Commit-Objekt (Abschnitt 2.4.1) sein. Zum Anderen können damit auch Dateien aus vorherigen Commits wiederhergestellt werden [VJ11, S 76].

[9] Git benutzt eine spezielle Syntax für solche Referenzen. So können Vorgänger (Abschnitt 3.1.5.2) beispielsweise mit `HEAD^` referenziert werden. Das Zeichen `^` dient dabei als Zähler. So bezeichntet `HEAD^^` den zweiten Vorgänger und ist äquivalent zu `HEAD~2`. [VJ11, S. 65]

2.4.8 Clone

Was z.B. unter SVN oder CVS der *checkout* eines entfernten Repositorys ist, ist unter Git das *Clonen* eines entfernten Repositorys. Es bezeichnet das Herunterladen eines Git-Repositories von einer Quelle mit der vollständigen Historie und allen enhaltenen Commits. Es erstellt sozusagen einen Klon des ursprünglichen Repositorys. [VJ11, S. 21]

2.4.9 Working Tree

Es gibt unterschiedliche Begriffe für den *Working Tree*. So werden synonyme Begriffe wie Arbeitskopie (engl. *Working Copy*), Arbeitsbereich, *sandbox* oder auch *checkout* verwendet. Es bezeichnet die gerade aktuelle Version des lokalen Arbeitsverzeichnisses, an der ggf. Änderungen vorgenommen werden können. [VJ11, S. 20]

2.4.10 Index

Der Index ist ein Alleinstellungsmerkmal von Git und bezeichnet eine Ebene zwischen der Arbeitskopie und dem Repository. Hier können Änderungen für einen zukünftigen Commit vorgemerkt werden [VJ11, S. 20]. Der Index und dessen Rolle wird im Folgenden nochmal gesondert betrachtet (Abschnitt 3.1.6). Dieser Bereich wird häufig auch *Stage* oder *Staging Area* genannt [SB14, S. 11].

2.5 Arten von Versionsverwaltungssystemen

2.5.1 Lokal

Frühe Versionskontrollsysteme wie RCS oder CSSC arbeiten ausschließlich auf dem lokalen Dateisystem und sind nicht dafür entwickelt, ein kollaboratives Arbeiten zu unterstützen. Der hauptsächliche Anwendungsfall ist, einem Entwickler eine lokale Versionsverwaltung zu ermöglichen. Die Inhalte können weiteren Personen nicht mit Mitteln des Versionskontrollsystems zur Verfügung gestellt werden.

2.5.2 Zentral

Versionskontrollsysteme wie SVN oder CVS stellen ein zenrales Repository mit Hilfe einer Serverkomponente zur Verfügung. Entsprechend dazu gibt es für verschiedene Betriebssysteme Clientkomponenten. So gibt es für Linux oder andere Unix-Derivate frei verfügbare Server- und Clientkomponenten, die zum Teil grafisch sowie in einem Terminal zu bedienen sind. Die Kommunikation der Clientkomponente erfolgt im Regelfall direkt mit dem Server. Das bedeutet, der Benutzer holt mit dem Client die Dokumente aus dem zentralen Repository, führt seine Änderungen durch und fasst diese in einem Commit zusammen. Hierfür stellt das Versionskontrollsystem entsprechende Unterstützung bereit, beispielsweise mit den Befehlen `checkout` (Abschnitt 2.4.7) zum Holen der Dokumente und `commit` (Abschnitt 2.4.1), zum Zusammenfassen und gleichzeitigen Übertragen an die Serverkomponente. Die Bearbeitung der Dokumente im lokalen Repository erfolgt überwiegend mit unabhängigen Werkzeugen. [JCL17, S. 38-40]

2.5.3 Verteilt

Versionskontrollsysteme, die verteilt arbeiten, werden im Englischen Distributed Version Control System (DVCS) genannt und zeichnen sich dadurch aus, dass der Benutzer lokal ein vollwertiges Repository zur Verfügung hat, ohne dass eine Serverkomponente ein zentrales Repository zur Verfügung stellen muss. Das ist aber nicht zu verwechseln mit lokalen Versionskontrollsystemen wie RCS oder SCCS. Der Unterschied ist, dass verteilte Systeme das gleichzeitige Arbeiten von mehreren Benutzern stärker unterstützen als zentrale Versionskontrollsysteme. Dabei ist es unerheblich, ob Benutzer gleichzeitig an mehreren Branches, an verschiedenen Repositories oder an den gleichen Dateien arbeiten. [JD10, S. 393-394]

Ein Auszug aus den von Jez Humble und David Farley in [JD10, S. 393-394] beschriebenen Eigenschaften eines DVCS lautet wie folgt:

- Es ist keine Serverkomponente nötig, um einen Commit in ein lokales Repository durchzuführen.
- Repositorys verschiedener Benutzer und Quellen können miteinander verbunden werden und deren Inhalte verglichen, integriert oder verändert werden.
- Um Veränderungen durchzuführen und in einem Commit zusammenzufassen, ist es unerheblich, ob eine Netzwerkverbindung zu einer möglichen Quelle besteht.
- Änderungen können an eingeschränkte Nutzergruppen veröffentlicht werden, ohne dass jemand gezwungen ist, diese zu übernehmen.
- Dadurch, dass jeder Benutzer eine eigene vollwertige Kopie des Repositorys hat, sind DVCS robuster gegenüber Datenverlusten. Darüber hinaus können Benutzer dadurch lokal Branches anlegen und Experimente durchführen, ohne diese in ein zentrales Repository zu übertragen. Die geringere Beanspruchung von zentralen Ressourcen, führt zu einer besseren Skalierbarkeit.
- Commits, Branches und Tags können lokal sortiert und zusammengefasst werden, bevor die Übertragung in ein gemeinsam genutztes Repository durchgeführt wird.[10]

Im Gegensatz zu zentralen Versionskontrollsystemen erfolgt das Erstellen eines Commits lokal und die Übertragung in ein weiteres Repository mit einem zusätzlichen Befehl. Dabei können die Daten beliebig oft in weitere Repositorys übertragen werden.

2.5.4 Streaming

Ein populäres Beispiel für ein Versionskontrollsystem, das auf Datenströmen (*Streams*) basiert, ist das kommerziell vermarktete ClearCase von IBM. Das zentrale Merkmal hierbei ist, dass *Branches* durch sogenannte Streams ersetzt werden. Das ermöglicht eine Baumstruktur von abhängigen Zweigen, die durch eine Vererbungsstruktur verbunden sind. Es

[10] Das wird im Kontext von Git als *rebase* bezeichnet.

können Änderungen innerhalb eines Arbeitsbereichs durchgeführt werden, ohne andere Benutzer zu beeinflussen. Nach Abschluss dieser Arbeiten können diese veröffentlicht bzw. *promoted* werden. Diese werden dann allen abhängigen Streams gleichzeitig zur Verfügung gestellt. Das kann hilfreich sein, um z.B. grundlegende Softwarebibliotheken auszutauschen oder wichtige Fehlerbehebungen durchzuführen. In bisher erwähnten Versionskontrollsystemen müssen solche Änderungen einzeln auf jeden Branch übertragen werden.

3 Git

In diesem Kapitel werden die wesentlichsten Grundlagen und Befehle beschrieben, um mit Git Dateien zu verwalten. Zunächst wird mit einfachen Befehlen ein Git-Repository erstellt und später am Beispiel dieses Repositories auf weitere Eigenschaften und Befehle eingegangen.

3.1 Grundlagen

Hier wird beispielhaft ein Repository *git-example* mit einem Skript (`git-stats`) erstellt, welches ein paar einfache Statistiken über die Autoren eines Git-Repositories erzeugt.

Alle Beispiele werden auf der Linux Kommandozeile durchgeführt. Git ist für alle gängigen Linux Derivate verfügbar. Die Autoren von [SB14, S. 12-14] gehen auf weitere Details zur Installation von Git auf anderen Betriebssystemen ein.

Die in den Beispielen eingesetzte Version von Git ist 2.15.0.

```
$ git --version
git version 2.15.0
```
(3.1)

Der folgende Abschnitt basiert zum Teil auf den Ausführungen der Autoren aus [VJ11, S.22-57].

3.1.1 Konfiguration

Zu Beginn werden einige grundlegende Konfigurationen vorgenommen. So werden, damit die Nachvollziehbarkeit gewährleistet ist, zunächst Name und Mailadresse des Benutzers konfiguriert. Auch ist die farbliche Darstellung der Ausgaben durchaus hilfreich. Hier kann Git so konfiguriert werden, dass die Farbausgabe automatisch unterdrückt wird, sollte die Ausgabe in eine Datei umgeleitet werden.

```
$ git config --global user.name "Markus Moeglich"
$ git config --global user.email markus@moeglich.de
$ git config --global color.ui auto
```
(3.2)
```
$ git config --global --list
user.name=Markus Moeglich
user.email=markus@moeglich.de
color.ui=auto
```

Die *global* eingestellten Optionen können ebenfalls direkt in der Datei `~/.gitconfig` eingesehen und bearbeitet werden. Zusätzlich können für jedes Repository spezifische Konfigurationen vorgenommen werden. Hierzu muss der Parameter `global` entfernt werden. Die zugehörige Konfigurationsdatei findet sich in jedem Repository unter dem Pfad `.git/config`.

3.1.2 Erstellen eines Repositories

Damit Dateien nun mit Git versioniert werden können, muss ein (lokales) Repository erstellt werden:

```
$ git init git-example.git
Leeres Git-Repository in /home/seminar/git-example.git/.git/ initialisiert
```
(3.3)

Existiert das Verzeichnis `git-example.git` noch nicht, wird es durch den Befehl angelegt. Zusätzlich wird innerhalb von `git-example.git` noch ein weiteres Verzeichnis `.git` erzeugt, in dem neben der Konfiguration alle Daten abgelegt werden, die Git zur weiteren Verwaltung des Repositorys benötigt.

Um den Status des aktuell erzeugten Repositorys auszugeben, kann der Befehl `$ git status` verwendet werden:

```
$ git status
Auf Branch master

Noch keine Commits

nichts zu committen (erstellen/kopieren Sie Dateien und benutzen
Sie "git add" zum Versionieren)
```
(3.4)

Die von Git erzeugte Ausgabe macht darauf aufmerksam, dass noch keine Commits erzeugt wurden und dass man nun neue Dateien erstellen und hinzufügen kann.

3.1.3 Die ersten Commits

Hier werden nun die ersten Dateien zu dem Repository hinzugefügt. Dazu werden mit folgendem Befehl, zwei Dateien aus dem Internet heruntergeladen. Zuvor wird mit dem Befehl `$ mkdir helpers` ein Unterordner in dem erzeugten Repository angelegt.

```
$ wget https://raw.githubusercontent.com/sagiru/Seminar-1908-Git/master/
    LICENSE
$ wget https://raw.githubusercontent.com/sagiru/Seminar-1908-Git/master/
    helpers\
/git-stats -O helpers/git-stats
```
(3.5)

Ab jetzt können die Dateien mit Git verwaltet werden. Ein erstes Hinzufügen einer Datei mit dem Befehl `$ git add LICENSE` führt zu folgender Ausgabe von `$ git status`:

```
$ git status
Auf Branch master

Noch keine Commits

zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-
      Area)

  neue Datei:   LICENSE

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit
      vorzumerken)

  helpers/
```
(3.6)

Die Ausgabe macht darauf aufmerksam, dass zum Einen eine neue Datei zum Commit vorgemerkt ist und dass sich zum Anderen noch eine Datei im Verzeichnis befindet, die noch nicht mit Git verwaltet wird. Hierbei ist anzumerken, dass Git auch die vorgemerkte Datei noch nicht versioniert. Dazu muss zuerst ein Commit erstellt werden. Darauf wie Git hier unterscheidet, wird später (Abschnitt 3.1.6) eingegangen.

Der erste Commit in dem angelegten Repository wird mit dem Befehl `$ git commit` erzeugt. Eine Bemerkung zu dem Commit kann, je nach voreingestelltem Systemeditor jetzt eingegeben oder optional mit dem Parameter `-m` direkt auf der Kommandozeile (Listing 3.7) übergeben werden. In der Bemerkung wird die erste Zeile als Betreff und getrennt durch eine Leerzeile, alle weiteren Zeilen als ausführliche Beschreibung verwendet. Dieses Format ist in den meisten Versionskontrollsystemen üblich.

```
$ git commit
[master 86c3661] Add gpl license file
 1 file changed, 1674 insertions(+)
```
(3.7)

Darüber hinaus bietet es sich an, beim Erstellen solcher Bemerkungen gewisse Regeln einzuhalten. Beispielsweise beschreiben Jez Humble und David Farley in [JD10, S. 37] eine Situation, in der es sinnvoll ist, nicht nur zu wissen, was der Autor geändert hat, sondern auch warum und in welchem Kontext. Wenn nicht klar ist, was der Autor sich bei Änderungen gedacht hat oder Zusammenhänge nicht aus dem Commit hervorgehen, kann ein gefundener und behobener Fehler vor dem Veröffentlichen einer Softwareversion durchaus zu Folgefehlern führen. Solche Situationen enden nicht selten darin, dass viele Stunden Arbeit investiert werden müssen, um diese zu bereinigen. [JD10, S. 37]

Nachdem der erste Commit erzeugt wurde, kann dieser nun mit `$ git show` (Abschnitt 3.1.7.6) betrachtet werden. In diesem Fall wurde beispielhaft eine etwas ausführlichere Bemerkung für den Commit gewählt:

```
$ git show
commit 86c36617ece5884928cdb20eacbbf0511a47d96a
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 17:39:52 2017 +0100

    Add gpl license file

    The software and all the files included in this repository should be
    licensed by the GNU General Public Licsense 3.0. Thats why we put the
    original license file from [1] into this repository.

    [1] https://www.gnu.org/licenses/gpl-3.0.txt

diff --git a/LICENSE b/LICENSE
new file mode 100644
index 0000000..9cecc1d
--- /dev/null
+++ b/LICENSE
@@ -0,0 +1,674 @@
+                    GNU GENERAL PUBLIC LICENSE
+                       Version 3, 29 June 2007
...
```
(3.8)

Mit `$ git show` können alle wichtigen Informationen eines Commits eingesehen werden. Neben Zeitpunkt, Autor und Beschreibung beinhaltet dieser auch die *Commit-ID* (Abschnitt 2.4.1). Die weiteren Informationen werden in einem Format namens *Unified-Diff*[11] dargestellt. [VJ11, 25]

Mit `$ git add` wird nun eine zweite Datei hinzugefügt:

```
$ git add helpers/git-stats
```
(3.9)

Anschließend wird ein weiterer Commit erzeugt:

```
$ git commit -m "Add helper script to generate author statistics."
[master ce31bd5] Add helper script to generate author statistics.
 1 file changed, 23 insertions(+)
  create mode 100644 helpers/git-stats
```
(3.10)

Mit den beiden zuvor ausgeführten Commits, wurde eine, wenn auch kurze, Historie erstellt. Um diese zu untersuchen, dienen die Befehle `$ git whatchanged`[12] oder `$ git log`. Hierauf wird in Abschnitt 3.1.7.6 weiter eingegangen. Für diesen Fall aber ist `$ git whatchanged` ausreichend:

[11] Details zu dem *Unified-Diff* Format stellen die Autoren von *GNU diffutils* unter [DPR17, S. 12-13] zur Verfügung.

[12] Der Befehl `$ git whatchanged` stammt aus frühen Zeiten von Git und ist eigentlich der heutige Befehl `$ git log` mit einigen Parametern, um die Ausgabe entsprechend zu formatieren.

```
$ git whatchanged
commit ce31bd52900e7c45c07ef83291b79a471e71ba77
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 17:50:35 2017 +0100

    Add helper script to generate author statistics.

:000000 100644 0000000... 2db9553... A helpers/git-stats

commit 86c36617ece5884928cdb20eacbbf0511a47d96a
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 17:39:52 2017 +0100

    Add gpl license file

    The software and all the files included in this repository should be
    licensed by the GNU General Public Licsense 3.0. Thats why we put the
    original license file from [1] into this repository.

    [1] https://www.gnu.org/licenses/gpl-3.0.txt

:000000 100644 0000000... 9cecc1d... A LICENSE
```
(3.11)

Die Ausgabe wird im sogenannten *raw* Format dargestellt. Es werden alle Commits mit Informationen über den Autor, Commit-ID, Zeit und den veränderten Dateien in zeitlicher Reihenfolge dargestellt. Die konkreten Veränderungen am Inhalt der Dateien werden hier nicht dargestellt.

Erwähnenswert ist noch, dass Git im Gegensatz zu anderen Versionskontrollsystemen die Dateiberechtigungen speichert. Die Ausgabe des letzten Commits (Listing 3.10) enthält folgende Zeile:

```
create mode 100644 helpers/git-stats
```
(3.12)

Diese Ausgabe beschreibt, dass es sich hierbei um eine normale Datei handelt (100), die mit einer unter Unix üblichen Berechtigung (664) angelegt ist. Damit diese Datei aber entsprechend ausführbar ist, muss noch ein weiterer Commit, mit einer Änderung an den Dateiberechtigungen, erzeugt werden:

```
$ chmod a+x helpers/git-stats
$ git commit -a -m "Add executable bit to helper script."
[master 878b171] Add executable bit to helper script.
 1 file changed, 0 insertions(+), 0 deletions(-)
  mode change 100644 => 100755 helpers/git-stats
```
(3.13)

Mit dem zuvor ausgeführten Befehl wurde ein Commit erzeugt, ohne die Änderungen vorher mit `$ git add` hinzuzufügen. Der Parameter `-a` bewirkt hier, dass alle vorhandenen Änderungen zu einem Commit zusammengefasst werden und nicht einzeln hinzugefügt

werden müssen. Zu beachten ist allerdings, dass Dateien, die Git noch nicht kennt, davon nicht betroffen sind. Diese müssen nach wie vor zuerst mit `$ git add` hinzugefügt werden.

Abschließend erhält der letzte Commit noch eine Markierung bzw. einen Tag.

```
$ git tag -m "Tagging first Release 0.0.1" tags/0.0.1
```
(3.14)

Damit wurde ein Tag mit dem Namen `tags/0.0.1` angelegt, der den letzten Commit bzw. *HEAD* referenziert:

```
$ git tag
tags/0.0.1
```
(3.15)

Auf den weiteren Umgang mit Tags und die unterschiedlichen Arten von Tags, wird in den Abschnitten 3.1.7.4 und 3.1.5.3 eingegangen.

3.1.4 Verteiltes Git

In diesem Abschnitt wird das Arbeiten mit entfernten Repositorys gezeigt (Abbildung 3.1). Die wesentliche Funktionsweise von Git ist auf kollaborative Arbeit an Repositorys ausgelegt (Abschnitt 4.2).

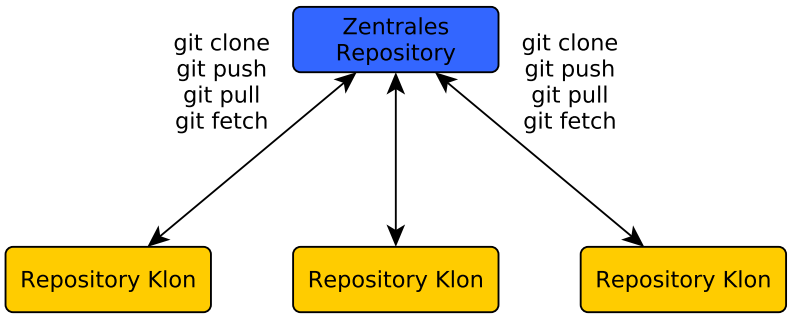

Abbildung 3.1: Zentraler Workflow. Angelehnt an [VJ11, S. 138]

.

3.1.4.1 Erstellen eines Klons

Um eine lokale Kopie eines entfernten Repositories zu erstellen bzw. ein Repository zu klonen, benötigt man den Befehl `$ git clone`. Zum Transport über das Netzwerk können verschiedene Protokolle benutzt werden. Unterstützt werden *ssh, git, http[s], ftp[s]* und *file*. Eine entsprechende Repository-URL kann neben dem Protokoll noch den Pfad des Repositories, einen Benutzernamen oder einen Port enthalten:

```
[Benutzer@]<Protokoll>://<Adresse>[:Port]}/Pfad/zum/Repository
```
(3.16)

Besonderheiten gibt es für das Klonen lokaler Repositories und solcher, die über über Secure Shell (SSH) geklont werden. Hier gibt es jeweils eine Kurzform. Für das Klonen über SSH muss das Protokoll nicht mit angegeben werden und bei lokal erreichbaren

Repositories kann vollständig auf die Angabe einer Adresse verzichtet werden. Hier reicht einfach die Angabe eines Pfads `/pfad/zum/repo`.

Das Beispiel aus Abschnitt 3.1.3 ist als Repository verfügbar und kann mit folgendem Befehl geklont werden.

```
$ git clone https://github.com/sagiru/git-example.git
Klone nach 'git-example' ...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 10 (delta 0), reused 10 (delta 0), pack-reused 0
Entpacke Objekte: 100% (10/10), Fertig.
```
(3.17)

Git speichert die Herkunft des entfernten Repositorys ebenfalls in der Datei `.git/config`:

```
[remote "origin"]
  url = git@github.com:sagiru/git-example.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```
(3.18)

Die gleichen Informationen werden mit Hilfe des Git Befehls `$ git remote` ausgegeben:

```
$ git remote -v
origin git@github.com:sagiru/git-example.git (fetch)
origin git@github.com:sagiru/git-example.git (push)
```
(3.19)

In Listing 3.19 wird zum Einen der Name, in diesem Fall *origin*, ausgegeben und zum Anderen, aufgrund der Verwendung des Parameters `-v`, auch der Pfad zum entfernten Repository. Mit `$ git remote` können weitere entfernte Repositories unter anderem Namen konfiguriert werden[13]. Ebenso wird standardmäßig der Branch *master* konfiguriert und als sogenannter *remote-tracking-branch* eingerichtet. Diese Branches verfolgen die Änderungen gleichnamiger Branches aus den entfernten Repositorys und werden bei einer Synchronisation automatisch durch Git aktualisiert. [VJ11, S. 141-143]

Das weitere Bearbeiten des Repositoryinhalts kann wie in Abschnitt 3.1.3 beschrieben durchgeführt werden.

3.1.4.2 Änderungen übertragen

Um Daten in ein entferntes Repository zu übertragen, wird der Befehl `$ git push` benutzt. Als Standardziel wird das Repository verwendet, von dem auch ursprünglich die Kopie erstellt wurde[14].

[13] Hierzu siehe `$ git help remote`
[14] In der Regel ist das *origin*.

```
$ git push
Zähle Objekte: 3, Fertig.
Delta compression using up to 4 threads.
Komprimiere Objekte: 100% (2/2), Fertig.
Schreibe Objekte: 100% (3/3), 355 bytes | 355.00 KiB/s, Fertig.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:sagiru/git-example.git
   ce31bd5..878b171 master -> master
```
(3.20)

In Listing 3.19 wurde weder der Name des entfernten Repositories, noch der des verwendeten Branches angegeben. In diesem Fall entscheidet Git nach der vorhandenen Konfiguration selbst und ergänzt zu `$ git push origin master`. Wenn der Branch in dem entfernten Repository einen anderen Namen hat oder ein neuer Branch unter einem anderen Namen angelegt werden soll, kann dieser Vorgang wie in Listing 3.21 durchgeführt werden:

```
$ git push origin master:other_branch
Total 0 (delta 0), reused 0 (delta 0)
To github.com:sagiru/git-example.git
 * [new branch]    master -> other_branch
```
(3.21)

Mit dem `$ git push` können aber auch entfernte Referenzen, wie Branches oder Tags gelöscht werden. Dazu muss neben dem Namen des entfernten Repositories eine leere Referenz mit dem Zielbranch angeben werden: [VJ11, S. 153-155]

```
$ git push origin :<branch-/tagname>
```
(3.22)

3.1.4.3 Änderungen herunterladen

Um Änderungen aus entfernten Repositories herunterzuladen, existieren zwei Befehle. Zum Einen `$ git pull` und zum Anderen `$ git fetch`. Der hauptsächliche Unterschied zwischen den beiden Befehlen ist, dass `$ git fetch` lediglich die Änderungen anderer Entwickler aus dem entfernten Repository herunterlät (Listing 3.23).

```
$ git fetch origin
Von github.com:sagiru/git-example
 * [neues Tag]     tags/0.0.1 -> tags/0.0.1
```
(3.23)

Bei `$ git pull` versucht Git nach Herunterladen der Änderungen, diese aus einem entsprechend konfigurierten *remote-tracking-branch* zu integrieren (Listing 3.23). Der Befehl ist also eine Kombination aus `$ git fetch` und `$ git merge`[15]. [VJ11, 144-152]

[15] Wenn stattdessen `$ git pull --rebase` verwendet wird, kommt hier eine Kombination von `$ git fetch` und `$ git rebase` zum Einsatz. Siehe hierzu auch [VJ11, 144-152] oder [SB14, 85-88].

```
$ git pull origin
Aktualisiere ce31bd5..878b171
Fast-forward
 helpers/git-stats | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
   mode change 100644 => 100755 helpers/git-stats
   Aktueller Branch master ist auf dem neuesten Stand.
```
(3.24)

3.1.5 Objektmodell

Wie in vorherigen Abschnitten bereits erwähnt, ist das Repository ein Datenspeicher vergleichbar mit einer Datenbank. Dahinter steht ein Datenmodell, welches sicherstellt, dass die einzelnen Teile eines Commits und deren Zusammenhänge über die Zeit gewährleistet sind. Dazu verwendet Git ein auf Objekten basierendes Modell. Dieser Abschnitt geht auf die verschiedenen Objekttypen und deren Zusammenhänge ein. Er basiert größtenteils auf den Ausführungen der Autoren aus [VJ11, S. 49-59].

Für jedes Objekt wird innerhalb des Repositories unter `.git/objects` eine Datei abgelegt. Für alle Objekte wird eine eindeutige SHA-1 Prüfsumme erzeugt, die gleichzeitig als Dateiname und Referenz für das abgespeicherte Objekt dient. Git unterscheidet zwischen den Objekttypen Tree, Blob, Commit und Tag.

3.1.5.1 Trees und Blobs

Zum Speichern von Verzeichnissen verwendet Git ein sogenanntes *Tree-Object*. Eine Ausgabe der Dateistruktur aus dem verwendeten Beispiel (Abschnitt 3.1.3) erhält man mit dem Linux Befehl `$ tree`:

```
$ tree
.
|-- helpers
|   |-- git-stats
|-- LICENSE

1 directory, 2 files
```
(3.25)

Um nun den Dateibaum mit der Objektstruktur aus dem Git-Repository zu vergleichen, kann der Befehl `$ git ls-tree HEAD` genutzt werden:

```
$ git ls-tree HEAD
100644 blob 9cecc1d4669ee8af2ca727a5d8cde10cd8b2d7cc LICENSE
040000 tree a72b974ebc96599b9ab0fc82fe8b3457da148744 helpers
```
(3.26)

Git unterscheidet hier zwischen den angelegten Verzeichnissen und den eigentlichen Dateien, wie `helpers/git-stats`. Verzeichnisse werden als *tree* dargestellt und enthalten wie auch im Dateisystem Referenzen auf weitere Objekte entweder vom Typ *tree* oder *blob*. Inhalte von Dateien werden in *blob*-Objekten gespeichert. Allerdings ist der Dateiname

nicht Bestandteil des *blob*, sondern wird in dem referenzierenden *tree* gespeichert. Dies verdeutlicht beispielsweise das *tree* Objekt zu dem Ordner `helpers`:

```
$ git ls-tree a72b974ebc96599b9ab0fc82fe8b3457da148744
100755 blob 2db9553130cc5709ea51f21c38d6a55b48876cb0 git-stats
```
(3.27)

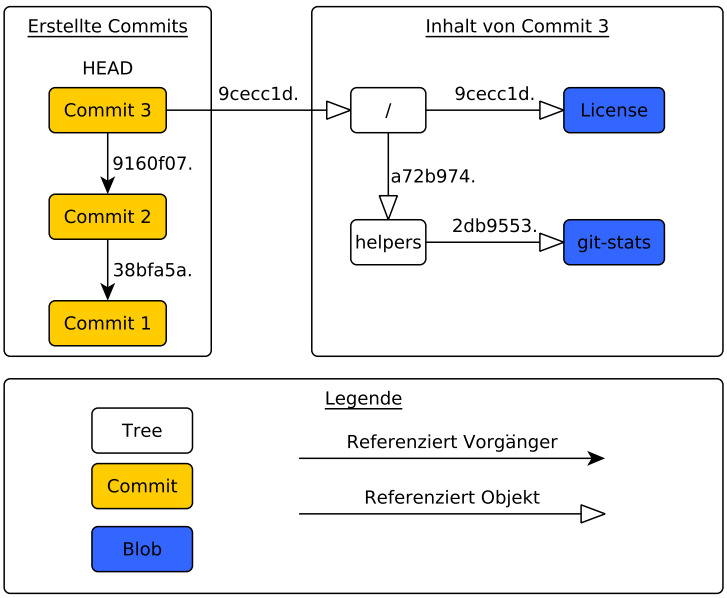

Abbildung 3.2: Objektmodell zum Beispiel aus Abschnitt 3.1.3. Angelehnt an [VJ11, S. 53]

.

3.1.5.2 Commit

Ein Commit-Objekt enthält neben den Daten aus Listing 3.8 eine Referenz zu **einem** *tree* und ggf. einem vorhergehenden Commit (engl. *parent*). Im Fall eines Merge-Commits (Abbildung 2.1) werden entsprechend so viele Vorgänger referenziert, wie Branches zusammengeführt wurden. Hier wird jeweils auf HEAD des jeweiligen Branches referenziert. Der in einem Commit referenzierte *tree* ist immer ein Verweis auf einen sogenannten *top-level-tree* (`/`). Dieser referenziert die Wurzel des entstandenen Objektbaums und enhält keinen Dateinamen (Abbildung 3.2).

Dadurch, dass jeder Commit seine vorhergehenden Commits referenziert, entsteht ein gerichteter azyklischer Graph, bei dem ein Knoten durch einen Commit und Kanten durch Referenzen repräsentiert werden. Ein weiterer Vorteil dieser Vorgehensweise ist, dass in dem Commit nicht immer alle enthaltenen Objekte neu gespeichert werden, sondern nur diejenigen, die verändert wurden. Alle anderen referenzieren das ursprünglich angelegte *blob* Objekt. Dieser Effekt wird als Deduplizierung bezeichnet und führt dazu, dass ein Repository, in dem eine Datei viele Male existiert, bzw. referenziert wird, kaum größer ist als ein Repository mit nur dieser einen Datei. [VJ11, 56-57]

```
$ git show --pretty=raw
commit 878b1715fe18bb84733837d20b8f0ae3f4f1be1e
tree 51c2dda0c9920b1d0600d0d5fae74aabd39fcd46
parent ce31bd52900e7c45c07ef83291b79a471e71ba77
author Markus Moeglich <markus@moeglich.de> 1511888829 +0100
committer Sascha Girrulat <sascha@girrulat.de> 1511889029 +0100

    Add executable bit to helper script.

    diff --git a/helpers/git-stats b/helpers/git-stats
    old mode 100644
    new mode 100755
```
(3.28)

Zusätzlich unterscheidet Git noch den Committer vom Autor. Diese können sich unterscheiden, wenn Commits z.B. einen Reviewprozess durchlaufen müssen und nicht von dem Autor selbst in das aktuelle Repository integiert werden (Listing 3.28).

3.1.5.3 Tag

Ein Tag markiert einzelne Commits. Hierzu werden Versionsnummern als Namen verwendet (Abschnitt 2.4.2). Neben der *SHA-1-ID* des zu referenzierenden Commits wird ein Tagname, der Typ und der Autor (*tagger*) mit Name und Mailadresse gespeichert. Je nach Art des Tags kann optional noch eine Beschreibung angegeben werden. Bei Tags ist nicht nur die Prüfsumme, sondern auch der Name eindeutig. Wird versucht, ein Tag mit einem bereits existierenden Namen zu erzeugen, reagiert Git mit einer Fehlermeldung und lehnt das Erstellen entsprechend ab.

Git unterscheidet zwischen drei Arten von Tags:

1. **Lightweight Tags:** Dieser Typ ist die einfachste Variante. Er enthält neben den benötigten Daten wie Autor, Commitreferenz und Prüfsumme lediglich den Namen. Erzeugt wird er mit dem Befehl:

   ```
   $ git tag <name>
   ```
   (3.29)

2. **Annotated Tags:** Annotated bedeutet nichts anderes als kommentiert. Zusätzlich zu den Daten aus den *Lightweight Tags* wird hier wie bei einem Commit ein Texteditor geöffnet. Alternativ kann auf der Kommandozeile ebenfalls mit dem Parameter `-m` eine Beschreibung übergeben werden.

   ```
   $ git tag <name> -m "Das ist ein annotated Tag."
   ```
   (3.30)

3. **Signierte Tags:** Voraussetzung für diese Art von Tags ist ein vorhandener privater *GPG-Key*. Diese Tags werden mit einer Signatur versehen, die von anderen entsprechend überprüft werden kann, um die Identität des Autors eindeutig zu verifizieren[16]. Erstellt werden signierte Tags mit dem zusätzlichen Parameter `-s`:

[16] https://gnupg.org

```
$ git tag -s <name> -m "Das ist ein signierter Tag"
```
(3.31)

Zuvor muss Git der zu verwendende Schlüssel noch bekannt gemacht werden. Das kann mit folgendem Befehl erreicht werden:

```
$ git config --global user.signingkey <Key-Id>
```
(3.32)

3.1.5.4 Branch

Eine einfache Antwort auf die Frage, warum ein Branch kein Objekt ist, liefert die Ausgabe der Referenzdatei des Branches *master*:

```
$ cat .git/refs/heads/master
614195f7f35391a18486f1da885776bc9cbb7f0b
```
(3.33)

Die ausgegebene Prüfsumme ist in diesem Fall lediglich die *SHA-1-ID* des letzten Commits aus Abschnitt 3.1.3. Branches sind einfache Textdateien, die eine *Commit-ID* zu einem referenzierten Commit enthalten. Da diese Referenz veränderbar ist und keine weiteren Informationen enthält, wird hier kein weiterer Objekttyp benötigt. Da der Name des Branches direkt von der erstellten Referenzdatei abhängig ist, sind auch die Namen eindeutig.

Neben der fortgeschrittenen Manipulation von Git Objekten, gehen die Autoren Scott Chacon und Ben Straub in [SB14, S. 408-418] ausführlicher auf die Objektverwaltung ein.

3.1.6 Bäume

Git organisiert das Repository in drei verschiedenen Bereichen. Scott Chacon, einer der Autoren von [SB14], spricht in einem seiner Vorträge [Sco11] von drei Bäumen *HEAD*, *Index* und dem *Work Tree*.

Der *Work Tree* (Abschnitt 2.4.9) wurde bereits als Arbeitsbereich angesprochen. Mit *HEAD* ist neben dem Verweis auf den neuesten Commit auch im weitesten Sinne das Repository gemeint. Das Repository dient als Datenspeicher für alle Commits, deren Inhalte und Historie. Der Index (Abschnitt 2.4.10) in Git stellt im Vergleich zu vorherigen Versionskontrollsystemen eine Neuerung dar. Der Index ist ein Bereich zwischen dem Repository und dem Arbeitsbereich. Er dient dazu, den nächsten *HEAD* bzw. Commit vorzubereiten. Die Befehle `$ git add`, `$ git reset` und `$ git commit` arbeiten auf diesen drei Bereichen. [VJ11, 34-35]

Abbildung 3.3: *Work Tree*, *HEAD* und Index. Angelehnt an [VJ11, 34]

.

3.1.7 Ergänzende Befehle

Die bisher verwendeten Befehle werden in diesem Abschnit um weitere ergänzt, die für die alltägliche Arbeit mit Git sinnvoll sind[17].

Alle Git Befehle bieten eine Vielzahl an Optionen. So verstehen viele der Befehle, die Änderungen an dem Repository vornehmen, den Parameter `dry-run`, der es ermöglicht, Vorgänge erst zu simulieren. Außerdem bieten die meisten Git Befehle mit `-v` eine umfangreichere Ausgabe (engl. *verbose*) an. Eine Übersicht über die unterstützten Optionen kann mit den unter Linux üblichen Methoden angezeigt werden. So z.B. mit `$ git help <Befehl>` oder mit dem klassischen Aufruf der Manpage `$ man git <Befehl>`.

Viele der Befehle erwarten ein bestimmtes Objekt als Argument. So finden sich häufig Argumente, die als *tree-ish* oder *commit-ish* bezeichnet werden. Damit sind aber lediglich entsprechende Objekte gemeint, die z.B. einen Tree referenzieren. So benötigt `$ git ls-tree` aus Abschnitt 3.1.5 ein *tree-ish* und `$ git show` kann auf allen Objekten aus Abschnitt 3.1.5 arbeiten. [VJ11, 52]

3.1.7.1 Dateien hinzufügen

Mit `$ git add` können grundsätzlich Dateien hinzugefügt werden. Der Befehl selbst bietet, wie fast alle Git Befehle, eine Vielzahl an Optionen. Eine sei an dieser Stelle besonders erwähnt, `$ git add -p`. Das `-p` ermöglicht, die durchgeführten Änderungen an den Dateien selektiv auf den Index zu legen und somit für einen Commit vorzumerken. Das kann hilfreich sein, um Änderungen an verschiedenen Teilen in einzelne Commits aufzuteilen oder bestimmte Änderungen wegzulassen. [VJ11, S. 36-37]

3.1.7.2 Dateien verschieben

Will man Dateien verschieben, kann man den Befehl `$ git mv` benutzen. Dieser Befehl arbeitet ähnlich wie der gleichnamige Linux-Befehl. Um den Vorgang zu erzwingen, wenn beispielsweise die Zieldatei schon existiert, kann der Parameter `-f` für *force* angegeben werden. Die Dateien können aber auch genauso mit unter Linux üblichen Mitteln wie `cp, rm, mv` etc. verschoben werden. Anschließend muss die Änderung lediglich mit dem Befehl `$ git add` hinzugefügt werden.[VJ11, S. 43-44]

```
$ git mv LICENSE GPLv3_LICENSE
$ git status
Auf Branch topic
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-
      Area)

    umbenannt:    LICENSE -> GPLv3_LICENSE
```
(3.34)

[17] Weitere siehe hierzu [VJ11], [SB14] oder [RB17].

3.1.7.3 Dateien entfernen

Der Befehl `$ git rm` steht für *remove*, bewirkt aber nicht das Gegenteil von `add`. Dieser Befehl entfernt, vergleichbar mit dem Linux-Befehl, Dateien oder Verzeichnisse. Um nicht leere Verzeichnisse zu löschen, muss noch ein `-r` für rekursiv angegeben werden. Wenn Dateien bereits verändert wurden, kann zusätzlich noch mit einem `-f` der Vorgang erzwungen werden.

```
$ git rm LICENSE
rm 'LICENSE'
$ git status
Auf Branch topic
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-
      Area)

    gelöscht:     LICENSE
```
(3.35)

Das Entfernen von Dateien aus dem Repository kann aber genauso mit unter Linux üblichen Mitteln vorgenommen werden. Die Änderungen müssen dann mit `$ git add` dem Index hinzugefügt werden. Die Dateien werden aber nicht vollständig entfernt. Der neu erstellte Commit verweist lediglich nicht mehr auf das entsprechende Objekt (Abschnitt 3.1.5.2). [VJ11, S. 43-44]

3.1.7.4 Tags verwalten

Die eindeutige Identifikation von Versionen bzw. konkreten Zuständen eines Repositories mit SHA-1-Prüfsummen ist aus technischer Sicht problemlos. Als Mensch hingegen kann man sich solche Prüfsummen weder gut merken, noch geht aus den Prüfsummen eine Historie hervor. Git ermöglicht es mit dem Befehl `$ git tag` Objekte, zumeist Commits, namentlich zu markieren. Als Namen dieser Markierungen wird man in der Praxis eine Vielzahl an Varianten finden. Eine durchaus übliche ist aber die Kombination aus drei bis vier fortlaufenden Zahlen. Um den Tag zusätzlich zu identifizieren, kann hier auch noch ein Präfix, z.B. `release/` oder `v` verwendet werden. Wie Tags angelegt und angezeigt werden, ist in Listing 3.14 und Listing 3.15 gezeigt. Ebenso wurde in Abschnitt 3.1.5.3 bereits auf die verschiedenen Typen von Tags eingegangen. Ergänzend hierzu sei noch erwähnt, dass Tags mit dem zusätzlichen Parameter `-d` gelöscht und mit `-f` überschrieben werden können. Das Herunterladen von Tags erfolgt über `$ git fetch` (Abschnitt 3.1.4) und das Veröffentlichen über `$ git push`:

```
$ git push origin tags/0.0.1
```
(3.36)

Um alle lokalen Tags zu veröffentlichen, kann der `$ git push` Befehl auch mit dem Parameter `--tags` kombiniert werden. [VJ11, 70-71,162-163]

```
$ git push --tags
```
(3.37)

3.1.7.5 Branches verwalten

Ähnlich wie bei Tags markiert auch ein Branch eine SHA-1-Prüfsumme, in diesem Fall die eines Commits. Im Gegensatz zu Tags sind diese lediglich dynamische Referenzen. Da hier nur eine Textdatei mit der Referenz erstellt werden muss, ist der Vorgang entsprechend schnell. Das Erstellen eines Branches benötigt als Argument mindestens einen eindeutigen Namen (Abschnitt 3.1.5.4). Wenn keine Referenz auf einen Commit übergeben wird, wählt Git einfach den aktuellen *HEAD*. Das folgende Beispiel erstellt einen Branch namens *topic* von der Referenz *master*.

```
$ git branch topic master
```
(3.38)

Um auf einen anderen Branch zu wechseln, kann der Befehl `$ git checkout <name>` benutzt werden.

In bestimmten Situationen verweigert Git den Checkout, beispielsweise wenn eine Datei überschrieben würde oder ein Wechsel des Branches zusätzliche Änderungen an einer lokal geänderten Datei zur Folge hätte. Der Parameter `-f` erzwingt auch hier den gewünschten Vorgang. Mit `$ git checkout` können auch Branches angelegt und direkt darauf gewechselt werden:

```
$ git checkout -b topic master
$ git branch
  master
  * topic
```
(3.39)

Der Befehl `$ git branch` unterstützt noch weitere Parameter. So kann mit den Parametern `-d/-D`[18] ein Branch gelöscht werden oder mit `-m` ein Branch umbenannt werden. Lokale Branches können mit `-l`[19] angezeigt werden, entfernte aus dem entfernten Repository mit `-r` und alle zusammen mit `-a`. [VJ11, 65-67]

Weitere Details zur Arbeit mit Branches sind beispielsweise in [JD10, 388-389, 408-415] und [SB14, 56-88] enthalten.

3.1.7.6 Historie untersuchen

In Abschnitt 3.1.3 wurde bereits auf `$ git whatchanged` eingegangen. Wie bereits erwähnt, verbirgt sich dahinter der Befehl `$ git log`. Mit diesem Befehl kann die Versionshistorie des Repositorys untersucht werden. Dies wird hier am Beispiel aus dem genannten Abschnitt gezeigt:

[18] Der Parameter `-D` wird benötigt, wenn auf dem Branch Objekte sind, die noch nicht dem aktuellen Branch hinzugefügt wurden [VJ11, 67].
[19] Das ist das Standardverhalten, wenn kein Parameter übergeben wird.

```
$ git log
commit 878b1715fe18bb84733837d20b8f0ae3f4f1be1e
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 18:07:09 2017 +0100

    Add executable bit to helper script.

commit ce31bd52900e7c45c07ef83291b79a471e71ba77
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 17:50:35 2017 +0100

    Add helper script to generate author statistics.

commit 86c36617ece5884928cdb20eacbbf0511a47d96a
Author: Markus Moeglich <markus@moeglich.de>
Date:  Tue Nov 28 17:39:52 2017 +0100

    Add gpl license file

    The software and all the files included in this repository should be
    licensed by the GNU General Public Licsense 3.0. Thats why we put the
    original license file from [1] into this repository.

    [1] https://www.gnu.org/licenses/gpl-3.0.txt
```
(3.40)

Der Unterschied zu Listing 3.11 ist die etwas reduzierte Ausgabe. Um die Ausgabe noch weiter einzuschränken, kann der Parameter `--oneline` genutzt werden.

```
$ git log --oneline
878b171 Add executable bit to helper script.
ce31bd5 Add helper script to generate author statistics.
86c3661 Add gpl license file
```
(3.41)

Auch kann die Ausgabe auf einen oder mehrere Commits eingeschränkt werden. Hierzu muss zusäzlich die entsprechende *SHA-1-ID* oder die Referenz übergeben werden. Soll ein bestimmter Bereich untersucht werden, kann der Anfang mit `..` vom Ende getrennt werden:

```
$ git log 86c3661..master
```
(3.42)

Die Trennung mit zwei Punkten wird auch von vielen anderen Befehlen verstanden, so wie beispielsweise von `$ git diff` oder `$ git show` (Abschnitt 3.1.3). Die Auswahl des angezeigten Bereichs erfolgt immer exklusive des ersten und inklusive des zweiten (nach `..`) angegebenen Commits. [VJ11, 45-48]

3.1.7.7 Aktuelle Änderungen anzeigen

Veränderte Dateien können entweder im Arbeitsbereich (Abbildung 3.3) oder bereits auf dem Index vorliegen (Abschnitt 3.1.7.1). Für die Anzeige, wird `$ git diff` verwendet.

Wird der letzte Absatz aus der Lizenzdatei (Abschnitt 3.1.3) entfernt, zeigt `$ git status` folgende Situation:

```
$ git status
Auf Branch topic
Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit
      vorzumerken)
    (benutzen Sie "git checkout -- <Datei>...", um die Änderungen im
        Arbeitsverzeichnis zu verwerfen)

     geändert:     LICENSE
```
(3.43)

Der Befehl `$ git diff` führt nun zu folgender Ausgabe im *Unified-Diff* Format:

```
$ git diff
diff --git a/LICENSE b/LICENSE
index 9cecc1d..dc263b1 100644
--- a/LICENSE
+++ b/LICENSE
@@ -665,10 +665,3 @@ might be different; for a GUI interface, you would use
     an "about box".
 if any, to sign a "copyright disclaimer" for the program, if necessary.
  For more information on this, and how to apply and follow the GNU GPL, see
   <http://www.gnu.org/licenses/>.
   -
   -  The GNU General Public License does not permit incorporating your
       program
   -into proprietary programs. If your program is a subroutine library, you
   -may consider it more useful to permit linking proprietary applications
       with
   -the library. If this is what you want to do, use the GNU Lesser General
   -Public License instead of this License. But first, please read
   -<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```
(3.44)

Ein anschließendes `$ git add LICENSE`, führt zum Verschieben auf den Index und `$ git diff` zu einer leeren Ausgabe:

```
$ git status
Auf Branch topic
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-
      Area)

    geändert:     LICENSE
```
(3.45)

Um die Änderungen auf dem Index zu betrachten, muss der Befehl `$ git diff` um den Parameter `--staged` ergänzt werden[20]. [SB14, 26-29]

[20] Die Parameter `--staged` und `--cached` können synonym verwendet werden und stehen in diesem Fall für das Betrachten der *Staging Area* (Abschnitt 2.4.10).

4 Weiteres zu Git

4.1 Einschränkungen

Auch wenn Git im Vergleich zu anderen Versionskontrollsystemen (Abschnitt 2.2) eine innovative Neuerung mit einem großen Funktionsumfang ist, gibt es auch hier Einschränkungen. Im Folgenden sind drei erläutert.

4.1.1 Umgang mit Binärdateien

Wie in Abschnitt 3.1.5 erläutert, ist der Umgang mit Dateien und Speicherplatz innerhalb von Git effizient organisiert. Eine Ausnahme bilden hier Dateien in Binärformaten (z.B. Grafiken, Videos, Audiodateien). Wie bereits beschrieben, werden Dateien innerhalb von Git als Objekte gespeichert und referenziert. Zusätzlich komprimiert Git diese Dateien, so dass trotz vieler Kopien und Änderungen der Speicherplatzverbrauch sehr gering ist. Binäre Formate lassen sich allerdings häufig nicht mehr weiter komprimieren oder zusammenfassen, so wird bei Änderungen solcher Dateien jede Version als eigenes Objekt gespeichert und vorgehalten. Angenommen, eine mit Git verwaltete Software wird mehrfach am Tag in einem Binärformat kompiliert und ebenfalls in dem gleichen Repository versioniert, dann bekommt jeder Entwickler alle Binärdateien, die über die gesamte Zeit erzeugt wurden, in seiner lokalen Kopie abgelegt. Es gibt hier aber externe Tools[21] und organisatorische Workflows[22], die sich mit diesem Problem beschäftigen. [RB17, S. 300]

4.1.2 Alles oder nichts

Bei der Betrachtung von anderen Versionskontrollsystemen (Abschnitt 2.2.2) fällt auf, dass viele die Möglichkeit bieten, nur bestimmte Teilbereiche herunterzuladen[23]. Auch können hier häufig verschiedene Berechtigungen auf unterschiedliche Verzeichnisse definiert werden. Beides ist mit Git nicht möglich. So muss man zum Einen immer einen Klon des gesamten Repositorys erstellen und zum Anderen können Lese- und Schreibrechte nur für das gesamte Repository vergeben werden. Das macht es in der Praxis häufig nötig, weitere Tools[24] einzusetzen oder andere Vorgehensweisen[25] zu verwenden, um dieses Problem zu umgehen. [RB17, 300-302]

4.1.3 Grafische Werkzeuge

Git selbst bietet keine grafische Unterstützung, um Informationen über den Inhalt zu visualisieren. So gibt es zwar neben den beschriebenen Befehlen (Abschnitt 3.1.7.6) mit `$ git blame`, `$ git annotate` und `git log -graph` noch weitere Möglichkeiten, mit denen sich umfangreichere Informationen anzeigen lassen, aber eben nur textuell Es gibt eine

[21] https://git-lfs.github.com/
[22] Lange Historien auslagern [RB17, S. 235-244]
[23] Der Clone Befehl (Abschnitt 3.1.4.1) bietet die Möglichkeit, mit depth nur eine bestimmte Anzahl von zusammenhängenden Commits herunterzuladen. Dies löst aber das hier beschriebene Problem nicht. [RB17, S. 244]
[24] z.B. https://www.gerritcodereview.com
[25] Forking-Workflow [RB17, S. 163-173].

Vielzahl an grafischen Tools, hier seien nur einige wie `gitk`, `gitg` oder `gitkraken` genannt. [RB17, S. 302]

4.2 Zahlen und Fakten - Der Linux Kernel

Das Git-Repository des Linux Kernels wurde am 16. April 2005 initial von Linus Torvalds mit folgendem Commit [Lin15] eröffnet:

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:  Sat Apr 16 15:20:36 2005 -0700

    Linux-2.6.12-rc2

    Initial git repository build. I'm not bothering with the full history,
    even though we have it. We can create a separate "historical" git
    archive of that later if we want to, and in the meantime it's about
    3.2GB when imported into git - space that would just make the early
    git days unnecessarily complicated, when we don't have a lot of good
    infrastructure for it.

    Let it rip!
```
(4.1)

Der Linux Kernel ist eines der populärsten Git-Repositories und unterstreicht die technischen Möglichkeiten von Git mit einigen, durchaus beindruckenden Statistiken.

Seit der Version (2.6.12-rc2) vom 16 April 2005[26] ist der Inhalt des Repositories auf ungefähr 61 Tausend Dateien mit annähernd 18 Millionen Zeilen Quellcode angewachsen. In dem Zeitraum zwischen 2005 und 2017 haben mehr als 17 Tausend verschiedene Autoren in über 693 Tausend Commits circa 36 Millionen Zeilen Quellcode hinzugefügt und 18 Millionen entfernt. Das sind 40 Commits pro Autor und durchschnittlich 157 Commits am Tag. Das Repository enthält 530 versionierte Tags mit durchschnittlich 1.309 Commits pro Version. Insgesamt wurden etwa 51 Tausend Merges durchgeführt, von denen Linus Torvalds etwas mehr als 22 Tausend selbst durchgeführt hat. Abbildung 4.1 gibt einen Überblick der 20 Autoren, die diese Statistiken anführen.

Statistiken wurden entweder mit `$ gitstats` [Hei13] oder Standard Git Befehlen ermittelt. So lassen sich z.B. unter Linux die durchgeführten Merges von Linus Torvalds mit folgendem Befehl ermitteln:

```
$ git log --all --merges --author="Linus Torvalds"| grep -c '^commit'
```
(4.2)

Solche Statistiken sind nicht unbedingt als Qualitätsmaßstab geeignet, um Rückschlüsse auf die Effizienz einzelner Personen zu ziehen. Wie in Kapitel 2.3 beschrieben, sollten diese Informationen aber trotzdem allen Beteiligten zur Verfügung gestellt werden. So wird nach Jez Humble und David Farley in [JD10, S. 138] eine Messung der geschriebenen Zeilen Quellcode am Tag wahrscheinlich eher dazu führen, dass Entwickler kürzere Zeilen

[26] Stand 13.11.2017

Author	Commits (%)	+ lines	- lines	First commit	Last commit	Age	Active days	# by commits
Linus Torvalds	23224 (3.35%)	26422	38999	2005-11-13	2017-10-31	4369 days, 3:07:49	3554	1
David S. Miller	9008 (1.30%)	167772	148984	2005-11-16	2017-10-29	4365 days, 5:37:42	2663	2
Mark Brown	6853 (0.99%)	263267	118031	2006-02-02	2017-10-25	4283 days, 13:06:33	1582	3
Takashi Iwai	6072 (0.88%)	196516	208220	2005-11-07	2017-10-17	4362 days, 2:00:11	1851	4
H Hartley Sweeten	5938 (0.86%)	127865	200037	2009-04-01	2017-08-08	3051 days, 22:27:43	437	5
Al Viro	5716 (0.82%)	106109	144084	2005-09-22	2017-09-29	4390 days, 12:27:18	1251	6
Ingo Molnar	5376 (0.77%)	123581	93184	2005-11-14	2017-10-27	4365 days, 8:55:51	1593	7
Mauro Carvalho Chehab	5183 (0.75%)	499573	466579	2005-11-14	2017-09-12	4320 days, 10:50:41	1278	8
Arnd Bergmann	5040 (0.73%)	56104	103135	2005-11-15	2017-10-20	4357 days, 0:31:00	973	9
Greg Kroah-Hartman	4507 (0.65%)	891939	1224072	2002-04-09	2017-10-19	5671 days, 12:16:44	1258	10
Johannes Berg	4075 (0.59%)	219785	152466	2005-12-11	2017-10-24	4335 days, 17:30:23	1347	11
Tejun Heo	3974 (0.57%)	150514	120807	2005-11-11	2017-10-09	4350 days, 2:37:04	971	12
Russell King	3941 (0.57%)	119236	97351	2005-11-12	2017-09-09	4318 days, 18:45:05	1298	13
Thomas Gleixner	3821 (0.55%)	89313	83060	2005-11-13	2017-10-21	4359 days, 20:33:28	903	14
Hans Verkuil	3465 (0.50%)	226685	221494	2005-11-14	2017-09-21	4329 days, 21:27:01	937	15
Christoph Hellwig	3380 (0.49%)	157135	244759	2005-11-14	2017-10-24	4362 days, 2:25:17	898	16
Chris Wilson	3223 (0.46%)	106755	72794	2009-01-30	2017-10-16	3181 days, 0:17:10	935	17
Geert Uytterhoeven	3086 (0.44%)	86731	61477	2006-04-11	2017-10-18	4208 days, 5:59:38	899	18
Dan Carpenter	3082 (0.44%)	11175	10469	2006-11-20	2017-10-25	3991 days, 19:29:29	1163	19
Rafael J. Wysocki	3028 (0.44%)	73457	46779	2006-01-06	2017-10-24	4309 days, 6:08:21	1088	20

Abbildung 4.1: Top 20 Linux Kernel Autoren (Stand 13.11.2017)

Quellcode als effizientere schreiben. Die gemessene Anzahl an Commits führt hingegen evtl. dazu, dass Änderungen pro Commit kleiner werden und dadurch besser nachvollziehbar. So schreiben Jeniffer Davis und Katherine Daniels in [JK16, S. 179] zu der Anzahl geschriebener Zeilen Quellcode:

> *„Lines of code is not an accurate measure of value. There are different types of developers, some that refactor hundreds of confusing lines into tens of lines of simple-to-read abstractions that can be build upon by others in the team.“*

Eine genauere Betrachtung von Abbildung 4.1 bekräftigt diese Aussage. Linus Torvalds führt zwar die Liste der meisten Commits an, hat aber weder die meisten Zeilen hinzugefügt noch entfernt. Greg Kroah-Hartman hingegen liegt nach der Anzahl der Commits auf Platz 10 und hat mit stattlichen 1.224.072 hinzugefügten und 891.939 entfernten Zeilen, wie auch die meisten anderen aus der Liste, weit mehr Zeilen zum Kernel beigetragen als Linus Torvalds selbst. Die Betrachtung der reinen Zahlen reicht also nicht aus, um eine repräsentative Einschätzung einzelner Personen zu erhalten.

Es können verschiedenste Statistiken über ein Projekt bzw. Repository erhoben werden, die geeignet sind, ein vollständiges Bild über die Aktivitäten innerhalb des Projektes zu bilden. Beispielsweise könnten das nach Jez Humble und David Farley in [JD10, S. 138] folgende sein:

- Testabdeckung
- Anzahl der Fehler
- Anzahl der Commits
- Anzahl der fehlgeschlagenen/erfolgreichen Versuche, die Software zu erstellen
- Metriken über den Quellcode (Zyklen, Duplikate o.a.)

5 Fazit

Im Rahmen dieser Arbeit wurden grundlegende Funktionen und Arbeitsweisen von Versionskontrollsystemen im Allgemeinen und Git im Besonderen behandelt. Die Möglichkeit, alle Dateien dauerhaft in jeder Version zuverlässig und reproduzierbar zu erhalten, ist ein erheblicher Zugewinn verglichen mit der Zeit, in der Quellcode ein einsames Dasein auf der Festplatte eines Computers fristete. Ein wichtiger Punkt ist jedoch nicht nur das leichtere Verwalten von Dateien, sondern auch der Gedanke, dass überholte Ideen oder Entwicklungen ohne das Risiko von Verlusten entfernt oder überarbeitet werden können. Diese Sicherheit schafft Freiheit, neue Dinge auszuprobieren oder zu verbessern. Das unterstreichen auch die Autoren Jez Humble und David Farley in [JD10, S. 35] mit dem folgenden Satz:

„*Version Control: The Freedom to Delete*“.

Ungeachtet der beschriebenen Einschränkungen ist Git im Vergleich zu vorhergehenden Versionskontrollsystemen ein leistungsfähiges Werkzeug, welches nicht nur technische Verbesserungen mitbringt, sondern die Zusammenarbeit in Teams und Projekten auch äußerst effizient unterstützt. Die Bedienung von Git mag zu Anfang nicht besonders übersichtlich sein, insbesondere bei fortgeschrittenen Themen im Rahmen umfangreicher Projekte, ermöglicht die Architektur von Git aber eine Skalierbarkeit, die mit anderen Versionskontrollsystemen nur schwer zu erreichen ist.

Diese in LaTex gesetzte Seminararbeit stellt mithilfe des verwendeten Beispiels den Einsatz von Git in einem kleinen Rahmen vor und wurde ebenfalls mit Git versioniert[27]. Dies zeigt, dass Projekte mit kleinem Umfang, die nicht in Teams geteilt und bearbeitet werden, gleichermaßen von der Arbeit mit Git profitieren.

Für eine Anwendung über den Inhalt dieser Seminararbeit hinaus, ist es erforderlich, Themen wie Bisektion, Submodule, Workflows und die Zusammenarbeit von Git mit anderen Versionskontrollsystemen zu betrachten. Die intensivere Auseinandersetzung mit dem Objektmodell führt ebenfalls zu einem besseren Verständnis der Funktionsweise von Git.

In der Praxis hat der Umgang mit Git gezeigt, dass sich Softwarequalität und Kollaboration in Teams durch den Einsatz von Plattformen wie Gerrit[28] oder GitHub[29] nochmals verbessern lassen. Die Verwendung von Versionskontrollsystemen ist eine Voraussetzung sowohl für den Einsatz agiler Vorgehensmodelle in der Softwareentwicklung, als auch für eine erfolgreiche Auseinandersetzung mit Themen wie *Continuous Integration* oder *Continuous Delivery*.

[27] https://github.com/sagiru/Seminar-1908-Git/
[28] https://www.gerritcodereview.com/
[29] https://github.com

Glossar

Branch Durch den Benutzer erstellter Abzweig an einer bestimmten Stelle bzw. Stand eines Repositories. **Abschnitt 2.4.3**.

Commit Stellt einen eindeutigen Stand des Repositories dar. Enthält i.d.R. Informationen über den Zeitpunkt der Erstellung, Name des Autors und eine Bemerkung über die durchgeführten Änderungen. **Abschnitt 2.4.1**.

HEAD Eine Referenz auf einen letzten/neuesten Commit eines Branches/Repositories. **Abschnitt 2.4.6**.

Open Source Mit Open Source(OS) Software wird Software bezeichnet, deren Quelltext frei und öffentlich verfügbar ist. Ausschlaggebend ist, dass er der Allgemeinheit uneingeschränkt zur Verfügung steht, bzw. auf Anfrage zur Verfügung gestellt wird. Dieser darf also gelesen, genutzt und verändert werden. Die Veränderungen können, dürfen und sollten der Allgemeinheit ebenso wieder zur Verfügung gestellt werden. Mehrheitlich kann Open Source Software kostenlos genutzt werden. Dieses Vorgehen wird von der Open Source Gemeinschaft (Open Source Community) durch allgemein anerkannte Regeln und Lizenzmodelle bekräftigt[Ini97, Ini09].

Repository Datenspeicher, in dem Dateien durch das Versionskontrollsystem verwaltet werden. **Abschnitt 2.4.5**.

Repository-URL Zieladresse des Repositories. Enhält Informationen über die Zugriffsmethode, Zugriffsparameter und den Ort des Repositories [VJ11, S. 141].

Secure Hash Algorithm Wird durch SHA abgekürzt und ist ein für kryptografische Zwecke entwickelter Hash-Algorithmus, der eine Prüfsumme aus digitalen Daten z.B. Dateien bildet. Es handelt sich um eine nicht umkehrbare mathematische Funktion, die eine Bit-Folge auf eine Prüfsumme von 160 Bit Länge abbildet [VJ11, S. 50].

Secure Shell Wird durch SSH abgekürzt und ermöglicht eine Verbindung zu einem entfernten Rechner mittels des gleichnamigen Protokolls.

Tag Durch den Benutzer erstellte Markierung für einen bestimmten Stand innerhalb eines Repositories. Wird i.d.R. für Releases oder Versionen verwendet und wird i. d. R. nicht weiter verändert. **Abschnitt 2.4.2**.

Wrapper Als Wrapper wird eine Software bezeichnet, die eine andere Software vollständig oder teilweise umschließt und ggf. um weitere Funktionalitäten ergänzt oder verändert.

Literaturverzeichnis

[DPR17] MacKenzie David, Eggert Paul, and Stallman Richard. In *Comparing and Merging Files*. © Free Software Foundation, 1992-1994, 1998, 2001-2002, 2004, 2006, 2009-2017. Version from 6 May 2017.

[Fou93] Free Software Foundation. Gnu mysc. https://www.gnu.org/software/cssc/mysc.txt. Letzter Zugriff 01.11.2017, 1993.

[Fou01] Free Software Foundation. Gnu cssc. https://www.gnu.org/software/cssc/. Letzter Zugriff 01.11.2017, 2001.

[Fou13] Free Software Foundation. Gnu rcs. https://www.gnu.org/software/rcs/. Letzter Zugriff 01.11.2017, 2013.

[Hei13] Hokkanen Heikki. Gitstats - git history statistics generator. http://gitstats.sourceforge.net/. Letzter Zugriff 13.11.2017 12:12, 2013.

[Ini97] Open Source Initiative. Open source initiative - the open source definition. https://opensource.org/osd-annotated. Version 1.9, 1997.

[Ini09] Open Source Initiative. Open source initiative - licenses by name. https://opensource.org/licenses/alphabetical. Letzter Zugriff 01.11.2017, 2009.

[JCL17] Haake Jörg, Icking Christian, and Ma Lihong. In *1678 Verteilte Systeme - Begleittext*, West Lafayette, Indiana 47907, 1991, 2017. © FernUniversität in Hagen - Fakultät für Mathematik und Informatik.

[JD10] Humble Jez and Farley David. *Continuous Delivery - Reliable Software Releases Through Build, Test and Deployment Automation*. Addison Wesley, Boston, 2010.

[JK16] Davis Jennifer and Daniels Katherine. *Effective DevOps - Building a Culture of collaboration, Affinity and Tooling at Scale*. O´Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, 2016.

[Lin15] Torvalds Linus. Linux kernel git repository. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Letzter Zugriff 13.11.2017 12:09, 2015.

[Org06] Linux Kernel Organization. Git faq. https://git.wiki.kernel.org/index.php/GitFaq#Why_the_.27Git.27_name.3F. Letzter Zugriff 03.11.2017, 2006.

[RB17] Preißel Reńe and Stachmann Bjørn. *Git - Dezentrale Versionsverwaltung im Team Grundlagen und Workflows*. dpunkt.verlag GmbH, Heidelberg, 2017.

[SB14] Chacon Scott and Straub Ben. *Pro Git - Everything You Need to Know about Git*. Apress, New York, 2014. 2. Auflage.

[Sco11] Chacon Scott. A tale of three trees. https://speakerdeck.com/schacon/a-tale-of-three-trees. Letzter Zugriff 28.11.2017 19:36, 2011.

[VJ11] Haenel Valentin and Plenz Julius. *Git - Verteilte Versionsverwaltung für Code und Dokumente*. Open Source Press, München, 2011.

[WF91] Tichy Walter F. Rcs—a system for version control. In *ECSCW 2009: Proceedings of the 11th European Conference on Computer Supported Cooperative Work*, West Lafayette, Indiana 47907, 1991, 1991. Purdue University.

[WFgCG17] Koch Werner, Free Software Foundation, and g10 Code GmbH. Using the gnu privacy guard. https://gnupg.org/documentation/manuals/gnupg.pdf. Version 2.2.4, 2017.