

Bachelor-Thesis
in
Wirtschaftsinformatik

Verteilte Versionsverwaltung

Eine Tool-Integration
in
Java-Programmier-Vorlesungen

Referent: Prof. Dipl.-Inform. J. Anton Illik
Korreferent: Prof. Dr.-Ing. Stefan Noll
vorgelegt am: 31.08.2017
vorgelegt von: Ertuğrul Özkara
245269
Steigstraße 54
78554 Aldingen
ertugrul.oezkara@hs-furtwangen.de

Widmung

Ich widme diese Arbeit meinen Eltern.

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Arbeit beigetragen haben.

Ein außergewöhnlicher Dank gilt meinen Betreuern

Herr Prof. Dipl.-Inform. J. Anton Illik und Herrn Prof. Dr.-Ing. Stefan Noll, die mich mit allen notwendigen Informationen versorgt und mich durch die Thesis geführt haben.

Mein besonderer Dank gilt Sadık Şahin, für seine gründlichen Korrekturen und seine Geduld mit mir. Des Weiteren möchte ich Bilal Arslan danken, der mich immer zu Höchstleistungen angespornt hat.

Ein besonderer Dank gilt meinem engen Freund Hamza Çimen, der mir vor allem in der Anfangszeit dieser Arbeit eine große Hilfe war.

Ebenfalls möchte ich Alexander Gerling danken, welcher mich mit seiner Erfahrung stets motiviert hat und immer beruhigende Worte für mich fand.

Aber der größte Dank gilt meinen Eltern, denen ich diese Arbeit widme. Sie haben mich nicht nur während meines gesamten Studiums in jeglicher Art unterstützt und mich immer wieder aufgerappelt, sondern haben mich auch niemals aufgegeben und immer an mich geglaubt.

Ich bedanke mich bei meiner Familie und all meinen Freunden, die mir stets Mut gemacht haben und immer Zeit für mich hatten.

Abschließend danke ich ganz besonders den „Derin Krankcılar“ Ogün Bastan, Halil İbrahim Şahin, Eyyüp Agcakoç, Onur Özbay und Abdullah Kalem, die mich Zeit meines Studiums treu begleitet haben.

Danke für Alles.

Abstract

Die folgende Bachelorarbeit beschäftigt sich mit der Implementierung der verteilten Versionsverwaltungssoftware Git in die Java Programmier-Vorlesung von Prof. Illik. Dazu sind genauere Betrachtungen des Themas Versionsverwaltung unternommen und ein Einblick über die wesentlichen Aufgaben der Versionsverwaltung gewonnen worden. Darauf folgend wurde die Entstehungsgeschichte der Versionskontrolle betrachtet und ein Überblick über die bekanntesten und gängigsten Systeme ermöglicht. Anschließend wurde das Thema Git separat betrachtet und in einer für Informatikstudenten des zweiten Semesters verständlichen Formulierung dargelegt. Ebenso folgte eine kurze Gegenüberstellung verschiedener Systeme in diesem Bereich. Den wichtigsten Teil der Arbeit bildet jedoch das letzte Kapitel, in dem Schrittweise erklärt wird, wie Git so in die Vorlesung integriert wird, dass Studenten und Professor zusammen an einem Semesterübergreifendem Projekt miteinander arbeiten können.

The purpose of this work was to implement the Distributed Version Control System named Git into the java programming lecture of Prof. Illik. For this purpose and to gain insights into the main tasks of version control, aspects of version control were studied in detail. The history of version control was also investigated to get a good overview of the most known and used systems. This helped understanding how the development of version control systems led to Git. Afterwards Git was considered in an own chapter understandable for a student in the second semester. Moreover, a comparison was made among the systems. Finally, a way to implement Git into the lecture for the common usage by students and the professor in a term-overlapping project was described step by step.

Inhaltsverzeichnis

Widmung	III
Danksagung	IV
Abstract	V
Inhaltsverzeichnis	VI
Abbildungsverzeichnis	IX
Tabellenverzeichnis	X
Abkürzungsverzeichnis	XI
1 Einleitung	13
2 Grundlagen	15
2.1 Definition	15
2.2 Was ist Versionsverwaltung	16
2.3 Vorteile einer Versionsverwaltung	18
2.4 Bekannte Versionsverwaltungssysteme	19
2.4.1 CVS	19
2.4.2 SVN	20
2.5 Geschichte der Versionsverwaltung	24
2.5.1 SCCS	24
2.5.2 CVS	25
2.5.3 SVN	26
2.6 Funktionsweise	26
2.6.1 Problem(e)	27
2.6.2 Lösungswege	28
2.7 Arten der Versionsverwaltung	31

2.7.1	Lokale Versionsverwaltung	31
2.7.2	Zentrale Versionsverwaltung	32
2.7.3	Nachteile der zentralen Versionsverwaltung	32
2.7.4	Zentral versus verteilt	33
2.7.5	Verteilte Versionsverwaltung	34
2.7.6	Vorteile der verteilten Versionsverwaltung	35
2.8	Verschiedene Arbeitsweisen der Versionsverwaltung	36
2.8.1	Lock Modify Write	36
2.8.2	Copy Modify Merge	36
2.9	Zusammenfassung	37
3	Git	39
3.1	Die Entstehung von Git	39
3.2	Die Grundlagen und Eigenschaften von Git	41
3.2.1	Die Staging Area oder auch der Index	44
3.3	Vorteile von Git	45
3.4	Wichtige Begriffe innerhalb von Versionsverwaltungssystemen	47
3.5	Branching und Merging	48
3.5.1	Die Hauptentwicklungszeige	50
3.5.2	Die Unterstützungszeige	51
3.5.3	Der Feature-Branch	52
3.5.4	Der Release-Branch	54
3.5.5	Der Hotfix-Branch	57
3.5.6	Branches verwalten	58
3.5.7	Fast Forward Merge	60
3.6	Der HEAD	61
3.7	Merge-Konflikte	61
3.8	Hosting von Git	62

3.9	Zusammenfassung.....	64
4	Versionsverwaltungssysteme im Überblick	67
4.1	Anforderungen an Versionsverwaltungssysteme	67
4.2	Git.....	68
4.3	Apache Subversion	69
4.4	Gegenüberstellung der Systeme	69
4.5	Git oder SVN?	70
4.6	Zusammenfassung.....	71
5	eGit-Plugin in Eclipse	73
5.1	Registrierung bei GitHub	73
5.2	eGit-Integration von Eclipse	75
5.2.1	Lokales und Externes Repository aufbauen	75
5.2.2	Vorhandene Projekte importieren	82
5.3	Commit-Verlauf einsehen.....	84
5.4	Branching mit eGit.....	85
5.5	Rückgängig machen.....	88
5.6	Zusammenfassung.....	89
6	Fazit.....	91
	Eidesstattliche Erklärung	93
	Literaturverzeichnis	94

Abbildungsverzeichnis

Abbildung 1: Arbeit ohne Versionskontrolle	28
Abbildung 2 Sperren der Dokumente	29
Abbildung 3 Registrierung auf GitHub.com	74
Abbildung 4 Erstellung lokales Repository	76
Abbildung 5 Git-Toolbar einblenden.....	77
Abbildung 6 Lokaler Master-Branch in Eclipse	78
Abbildung 7 Neues Repository erstellen	78
Abbildung 8 URL des neuen Repositories	78
Abbildung 9 Remote erstellen	79
Abbildung 10 Quell- und Zielbranch.....	80
Abbildung 11 Pushen auf origin	80
Abbildung 12 In Master Pushen	81
Abbildung 13 Erfolgreicher Push.....	81
Abbildung 14 Git-Projekt importieren	82
Abbildung 15 Quell-Repository	83
Abbildung 16 Allgemeines Projekt	83
Abbildung 17 Nach Commit	84
Abbildung 18 Commit-Verlauf und Versionsunterschiede.....	84
Abbildung 19 Neuen Branch erstellen.....	85
Abbildung 20 Commit auf neuem Branch	85
Abbildung 21 Merge-Details.....	86
Abbildung 22 Erfolgreicher Push nach Branching und Merging	86
Abbildung 23 Fetch	87
Abbildung 24 Rückgängig machen	88
Abbildung 25 Letzten Commit bearbeiten	89

Tabellenverzeichnis

Tabelle 1 Grundlegende Begriffe	47
Tabelle 2 Git vs. SVN	70

Abkürzungsverzeichnis

VCS	Version Control System
SCCS	Source Code Control System
CVS	Concurrent Versions System
SVN	Subversion
RCS	Revision Control System
DVCS	Distributed Version Control System
CCVS	Centralized Control Version System
SCM	Source Control Management
GPU	General Public License
SSH	Secure Shell

1 Einleitung

Versionsverwaltungen oder auch Versionskontrollsysteme (kurz: VCS) verwalten die Änderungen an beliebig vielen Dateien, die im Laufe der Zeit entstehen. Sie ermöglichen es diese in eine ältere Version zurückzuführen ohne dass es dabei eine Rolle spielt, ob es sich um Bilddateien, Quellcode oder jegliche andere Art von Dateien handelt.¹

Versionskontrolle ist vor allem im Bereich der Software-Entwicklung ein immer wichtiger werdendes Thema. Höchstwahrscheinlich erreicht jedes Projekt irgendwann den Punkt, an dem ältere oder gelöschte Stände noch einmal eingesehen werden müssen, es zu verstehen gilt, warum zu einem bestimmten Zeitpunkt eine bestimmte Funktion in ein Programm eingefügt worden ist oder das etwas kaputt ist und man in die Ausgangssituation zurück möchte.

Diese bieten nicht nur einen Ausweg für die beschriebenen Probleme, sondern sie sind ebenso eine Erleichterung im Voranschreiten des Entwicklungsprozesses in Solo- und auch Großprojekten.² Darüber hinaus besteht jederzeit die Gefahr, dass man vor allem in großen Projekten mit mehreren Mitgliedern, die zuvor verrichtete Arbeit eines anderen überschreibt und diese damit verloren geht.

Mit immer größer und anspruchsvoller werdenden Programmen steigt auch die Anzahl der benötigten Entwickler. Diese wiederum haben die Aufgabe Anfälligkeiten auf Fehler so gut wie möglich zu verhindern. Dies wiederum stellt eine immer größer werdende Herausforderung dar. Abhilfe schaffen

¹ Vgl. Denker, Merlin und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog“ (2015): Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf, S. 1.

² Vgl. Vijayakumaran, Sujeevan, „*Versionsverwaltung mit Git*“ (Frechen: mitp, 2016), S. 14.

hier Versionsverwaltungssysteme, welche sich deshalb auch am schnellsten in dieser Branche verbreitet haben.³

Software-Entwicklung findet nicht nur in Unternehmen statt, sondern wird auch in den verschiedensten Einrichtungen gelehrt. Dabei reicht das Spektrum von weiterführenden berufsbezogenen Schulen, bis hin zu Universitäten. Daher ist es nicht unwahrscheinlich, dass die oben aufgeführten Probleme sich auch in Unterrichtsräumen und Hörsälen zu erkennen geben und Lehrende sowie auch Lernende vor Herausforderungen stellen.

Aufgrund dieser Erkenntnis ist es das Ziel dieser Arbeit Versionsverwaltung Studenten, Schülern sowie Lehrkörpern näher zu bringen, diese dafür zu begeistern und die benötigte Versionskontrolle in Programmier-Vorlesungen zu verankern.

Gegenstand von Kapitel zwei ist es dem Leser fundamentales Wissen über Versionsverwaltung zu vermitteln und zu verstehen aus welcher Notwendigkeit heraus diese entstanden ist. Des Weiteren werden verschiedene Konzepte und auch Funktionsweisen angesprochen. Nachfolgend werden gängige Versionsverwaltungssysteme unter die Lupe genommen und entsprechend wichtige Begriffe erklärt, um sich ein Bild darüber zu machen, welche Systeme am geeignetsten für die Arbeit in Programmier-Vorlesungen sind. Anschließend wird auf die Versionsverwaltungssoftware Git eingegangen. Den Abschluss bildet die Integration von Git in die Vorlesung und die zugehörige Handhabung.

³ Vgl. Bechara, John, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement,“ (2009): Geprüft am 17. Juni 2017. Online: <https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf>, S. 5.

2 Grundlagen

Dieses Kapitel geht auf wichtiges und fundamentales Wissen im Bereich von Versionsverwaltung und verteilter Versionsverwaltung ein. Es wird erklärt was Versionsverwaltung ist, warum es notwendig ist, welche Konzepte dahinter stecken und warum heutzutage verteilte Versionsverwaltung bevorzugt wird.

2.1 Definition

Definition 1

„Eine Versionsverwaltung hat die Aufgabe, den Werdegang eines Projektes zu protokollieren. Sie übernimmt die Versionierung und die Archivierung der Daten und ermöglicht den gemeinsamen Zugriff darauf. Dabei kann jede Änderung und jeder Stand verfolgt, rückgängig gemacht oder wiederhergestellt werden.“ (GlossarWiki: <https://glossar.hs-augsburg.de/Versionsverwaltung>, Geprüft am 27.04.2017)

Definition 2

„Eine Versionsverwaltung ist ein System mit dem Änderungen an Dateien auf Befehl der Benutzer aufgezeichnet werden und bei Bedarf jede der vorherigen Dateiversionen wiederhergestellt werden kann. Der zentrale Ort an dem die Änderungen der Dateien aufgezeichnet werden nennt sich Repository. Zusätzlich zur Benutzung eines Repositories von lediglich einem Benutzer, bieten die meisten Versionsverwaltungen die Möglichkeit, dass mehrere Benutzer gleichzeitig an Dateien arbeiten können, ohne dass dabei eine Änderung eines Nutzers verloren geht. Die Kombination dieser Eigenschaften bietet die Grundlage der Softwareentwicklung im Team. Dies bedeutet jedoch keineswegs, dass Versionsverwaltungen lediglich für die Softwareentwicklung interessant sind. Ebenso kann man die Systeme dazu nutzen, um Änderungen an Bildern, Audiodateien oder sogar Binärdateien nachzuverfolgen und alte Versionen wiederherzustellen. Gerade große Projekte in der Bild-

und Tonbearbeitung brauchen viel Platz auf der Festplatte und im Arbeitsspeicher, da sie selbst die Option bieten alle Änderungen nichtdestruktiv [...] durchzuführen und somit eine eigene Versionsverwaltung enthalten. Die Anforderungen an die Rechner könnten jedoch immens gesenkt werden, wenn man lediglich die aktuell nötigen Daten bereithält, anstatt des gesamten Werdegangs einer Datei. Dies kann man erreichen, indem man die programmeigene Versionsverwaltung deaktiviert und stattdessen auf ein System wie Git oder Subversion setzt.“ (Mauel: <https://www.volkermauel.de/Downloads/GitVsSvn.pdf>, 2013, S. 5, Geprüft am 17.06.2017)

2.2 Was ist Versionsverwaltung

„Arbeit, die Sie nicht mit einem Versionskontrollsystem verwalten, ist Arbeit, die Sie möglicherweise noch einmal machen müssen - sei es weil Sie versehentlich eine Datei löschen oder Teile als obsolet betrachten, die Sie später doch wieder benötigen.“ (Haenel und Plenz: 2011, S. 14)

Das obige Zitat unterstreicht das Argument, wie wichtig die Arbeit mit Versionsverwaltung in manchen Bereichen sein kann. Versionsverwaltung kann unter anderem dafür sorgen, dass bei einer sorgfältigen Planung viel Zeit gespart wird. Beispielsweise können verlorengelaubte Dateien wiederhergestellt werden. Dadurch muss bei einem unterlaufenen Fehler, die bereits getane Arbeit nicht wiederholt werden.

Versionsverwaltung bedeutet im Prinzip nichts anderes, als das von Dateien Versionen erstellt werden können, um diese dann je nach Gebrauch zu verwalten. Jede noch so kleine Änderung an einer Datei erstellt eine neue Version dieser. Der Begriff »Version« kann vieles Bedeuten, doch wenn man im Bereich der Versionskontrolle davon spricht, handelt es sich um die Version einer einzelnen Datei oder der Ansammlung mehrerer Dateien.⁴

⁴ Vgl. Vijayakumaran, Sujeevan, "Versionsverwaltung mit Git" (Frechen: mitp, 2016), S. 17.

Dadurch soll das gemeinsame Arbeiten an einer Vielzahl von Dokumenten ermöglicht werden. Dabei gilt die höchste Priorität der Vermeidung von Datenverlust bei gleichzeitiger Entwicklung durch mehrere Personen.⁵

Egal ob eine reine Textdatei bearbeitet wird, ein Bild oder ein Video, es ist üblich, dass der aktuelle Stand immer wieder gesichert wird, damit die bereits verrichtete Arbeit nicht verloren geht oder damit man bei Bedarf auf eine ältere Version zurückgreifen kann. Die Vorgehensweise bei der Benennung dieser "Sicherheitskopien" ist von Mensch zu Mensch unterschiedlich. Manche geben die aktuelle Versionsnummer der Datei bei, andere wiederum erzeugen verschiedene Ordner mit dem entsprechenden Datum.

Diese Vorgehensweisen sind zwar nicht ganz falsch, doch weder sind sie praktisch noch sind sie resistent gegen Fehleranfälligkeit. Wenn zu viele Dateien versioniert werden müssen und diese keine eindeutigen Namen zugewiesen bekommen, dann ist die Wahrscheinlichkeit sehr hoch, dass Versionen von Dateien verloren gehen.⁶

Zur Lösung dieser Probleme wurden Versionsverwaltungssysteme entwickelt, die es erlauben, genau nachzusehen wer etwas wann und weshalb geändert hat, um bei Bedarf die entsprechenden Änderungen rückgängig zu machen. Das System notiert sich nämlich, wer eine Datei geändert hat mit der dazugehörigen Uhrzeit und verweist auf den Abschnitt in der Datei, die geändert worden ist. So entsteht nach und nach mit jeder Änderung eine Änderungshistorie, die dem Arbeiten an mehreren Dateien Erleichterung bringen. Diese Änderungshistorie erlaubt es genau nachzuverfolgen ab welchem Punkt ein Projekt in die falsche Richtung geht oder Fehler aufweist, sodass diese Fehler schnell ausfindig gemacht und auch behoben werden können.⁷

⁵ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017.
Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 43.

⁶ Vgl. Vijayakumaran (wie Anm. 2), S. 17.

⁷ Vgl. Vijayakumaran (wie Anm. 2), S. 17f.

2.3 Vorteile einer Versionsverwaltung

Vor allem bei großen Projekten ist die Nutzung eines Versionsverwaltungssystems fast unumgänglich. Diese bringen nämlich erhebliche Erleichterungen mit an Bord. Die meisten Vorteile die eine Versionskontrolle mit sich bringt wurden zwar weiter oben schon fast alle genannt, dennoch werden hier noch einmal einige wichtige Punkte zusammengefasst.

Wie schon erwähnt, können Änderungen lokalisiert und rückgängig gemacht werden. Der große Vorteil hierbei ist, dass diese Änderungen innerhalb des gesamten Projekts vorgenommen werden können. Durch Fehler wird nicht das endgültige Schicksal eines Projektes beschlossen. Das Wissen, dass man jederzeit zu einem bestimmten Zeitpunkt zurückspringen kann, verleiht Mitarbeitern innerhalb eines Projektes eine gewisse Ruhe. Das Team arbeitet dadurch ungezwungener und schneller, denn die Möglichkeit Fehler schnell wieder beheben zu können, verleiht das Gefühl von Sicherheit. Versionsverwaltungssysteme verhalten sich in diesem Rahmen so ähnlich wie eine Zeitmaschine, die es uns erlaubt an einem bestimmten Datum zu einem bestimmten Punkt des Projektes zurückzuspringen.⁸

Ein weiterer Vorteil, den Versionskontrolle mit sich bringt, ist das gleichzeitige Entwickeln. Mehrere Entwickler können gleichzeitig und kontrolliert an dem selbem Quellcode arbeiten ohne das die Arbeit des jeweils anderen dadurch überschrieben wird.⁹

Des Weiteren hilft die Protokollierung dabei gemachte Änderungen ausfindig zu machen und zu verstehen wer im Team welche Änderung aus welchem Grund und an welcher Stelle des zu bearbeitenden Programms gemacht

⁸ Vgl. Thomas, David, Andrew Hunt, Falk Lehmann und Uwe Petschke, „Der pragmatische Programmierer.“. *Der pragmatische Programmierer, Hunt, Andrew, [2. Aufl.]*. - München [u.a.] : Hanser (2003): Geprüft am 17. Juni 2017. Online: http://files.hanser.de/hanser/docs/20051012_251121378-66_3-446-40470-8_Leseprobe1.pdf, S. 1f.

⁹ Vgl. Thomas et al., „Der pragmatische Programmierer,“ (wie Anm. 8), S. 1.

hat.¹⁰ Dies ist unter anderem auch dann nützlich, falls man ein bestimmtes Fragment des selbst erzeugten Quellcodes nicht mehr nachvollziehen kann. Nicht nur gleichzeitiges Entwickeln ist möglich sondern auch das gleichzeitige Veröffentlichen von Software. Das Team ist dadurch nicht gezwungen kurz vor einem Release seine Arbeiten zu unterbrechen und kann ungehindert auf dem Hauptzweig des Projektes die Arbeit fortsetzen.¹¹

2.4 Bekannte Versionsverwaltungssysteme

Wie schon bei der Geschichte der Versionsverwaltungssysteme, würde eine ausführliche Aufführung aller vorhandenen Systeme den Rahmen dieser Arbeit sprengen, da im Laufe der Zeit viele Open-Source-Produkte, sowie auch viele proprietäre Software durch bekannte Unternehmen, entwickelt worden sind. Folglich werden zwei der bekanntesten Programme aus dem Bereich der Versionskontrolle aufgeführt, welche vor der Existenz von Git die wahrscheinlich gängigsten in diesem Bereich waren und auch heute noch teilweise verwendet werden.

2.4.1 CVS

Obwohl es heutzutage nicht mehr weiterentwickelt wird, gilt das Concurrent Versions System als *das* klassische Versionskontrollsystem. Die Verwaltung läuft entweder lokal über den eigenen Rechner oder über einen Webserver. Das CVS wurde ursprünglich als ein reines Kommandozeilen-Programm entwickelt. Später kam aber auch eine Benutzeroberfläche hinzu.¹²

¹⁰ Vgl. Thomas, David, Andrew Hunt, Falk Lehmann und Uwe Petschke, „Der pragmatische Programmierer.“. *Der pragmatische Programmierer, Hunt, Andrew, [2. Aufl.]*. - München [u.a.] : Hanser (2003): Geprüft am 17. Juni 2017. Online: http://files.hanser.de/hanser/docs/20051012_251121378-66_3-446-40470-8_Leseprobe1.pdf, S. 1f.

¹¹ Vgl. Thomas et al., „Der pragmatische Programmierer,“ (wie Anm. 8), S. 2.

¹² Vgl. Dr.-Ing. Magdowski, Mathias, „Versionskontrolle mit Apache Subversion.“. Geprüft am 8. Mai 2017. Online: http://www.studentbranch.ovgu.de/studentbranch_media/Downloads/IEEE+Versionskontrolle+Workshop/Versionskontrolle_mit_Apache_Subversion.pdf, S. 27.

Ein wichtiger Aspekt bei CVS ist die Verankerung in der Open-Source-Gemeinde. Die Software kann ohne den Anspruch auf Lizenzen frei im Internet erhalten werden. Wie schon weiter oben erwähnt, ist es historisch als der Nachfolger des Revision Control Systems anzusehen.¹³

Das Repository verhält sich hier wie ein Dateiserver mit einem Verzeichnisbaum als Inhalt. Ein Verzeichnis für die allgemeine Bedienung und eines für jedes angelegte Projekt. Jede Datei wird in das entsprechend zugehörige Verzeichnis abgelegt. Zusätzlich enthalten die Verzeichnisse Informationen, welche die Unterschiede zu den Vorgängerversionen aufweisen sowie eine Zeitangabe wann die Datei geändert worden ist. Darüber hinaus ist es möglich für jede Version Kommentare zu hinterlassen.¹⁴

Von der Nutzung dieser Software ist heutzutage allerdings abzuraten, da es inzwischen sehr veraltet ist und neuere Systeme existieren. Diese weisen in vielen Bereichen Verbesserungen auf, sowohl in Anwenderfreundlichkeit als auch in grundlegenden Funktionalitäten.¹⁵

2.4.2 SVN

Apache Subversion gilt als der Quasi-Nachfolger von CVS.¹⁶ Da Subversion ein zentrales Versionsverwaltungssystem ist, wird das Repository auf einem zentralen Server gespeichert. Der wesentliche Unterschied zu verteilten Versionskontrollsystemen wie z.B. zu Git liegt darin, dass die gesamte Historie auf dem Server gespeichert und auch nur dort benötigt wird. Es ist nicht notwendig das Repository zu klonen oder zu kopieren um damit arbeiten zu

¹³ Vgl. Fischer, Lars, „Werkzeuge zum Versions- und Variantenmanagement: CVS/Subversion vs. ClearCase.“. Geprüft am 17. Mai 2017. Online: <https://www.wi1.uni-muenster.de/pi/lehre/ws0506/skiseinar/versionsverwaltung.pdf>, S. 3.

¹⁴ Vgl. Fischer, „Werkzeuge zum Versions- und Variantenmanagement:“, (wie Anm. 13), S. 5.

¹⁵ Vgl. Lämmer, Frank, „Traditionelle Webentwicklung vs Versionsverwaltung.“. Geprüft am 6. April 2017. Online: <http://t3n.de/news/traditionelle-webentwicklung-versionsverwaltung-303580/>.

¹⁶ Vgl. Dr.-Ing. Mathias Magdowski, „Versionskontrolle mit Apache Subversion.“. Geprüft am 8. Mai 2017. Online: http://www.studentbranch.ovgu.de/studentbranch_media/Downloads/IEEE+Versionskontrolle+Workshop/Versionskontrolle_mit_Apache_Subversion.pdf, S. 28.

können. Das Arbeitsverzeichnis kann mit einem Checkout direkt vom Server erstellt werden.¹⁷

Der Vorteil hierbei ist, dass bei großen Repositories ein Checkout schneller ausgeführt werden kann, da immer nur ein einziger Zustand des Projektes von dem Client übernommen werden muss. Auch wird dadurch weniger Speicherplatz auf dem Client beansprucht und sobald das Projekt auf dem Server abgelegt wird, steht es allen Clients mit Serververbindung zur Verfügung.¹⁸

Nachteile dieser zentralen Architektur können jedoch sein das ein Commit (Übergabe einer Version auf einen Branch) fehlschlagen kann, wenn das Netzwerk nicht verfügbar oder der Server offline ist. Änderungen, welche evtl. erst getestet werden sollten, können nicht gemacht werden, bis das Repository wieder erreichbar ist.¹⁹

Ein wichtiger Unterschied zwischen SVN und CVS ist der, dass bei SVN auch ganze Verzeichnisse versioniert werden können.²⁰ Dabei werden immer nur die Unterschiede zwischen den Dateien übertragen.²¹ Auch wenn nur einzelne Dateien geändert wurden, beziehen sich die Revisionen immer auf das gesamte Repository. Die Nummerierung der einzelnen Revisionen erfolgt aufsteigend und beginnt bei 0. Jedes Mal, wenn eine Änderung an das Repository übermittelt wird, zählt die Anzahl der Revision des gesamten Repository um eins hoch. Dies vereinfacht die Wiederherstellung von älteren

¹⁷ Vgl. Denker, Merlin und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog“ (2015): Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf, S. 11.

¹⁸ Vgl. Denker und Srecec, „*Versionsverwaltung mit Git*“, (wie Anm. 1), S. 11f.

¹⁹ Vgl. Mauel, Volker, „Vergleich von Git und SVN.“. Geprüft am 17. Juni 2017. Online: <https://www.volkermauel.de/Downloads/GitVsSvn.pdf>, S. 8.

²⁰ Vgl. Fünter, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online:

https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFm2Re4BosAJpixqOA, S. 5.

²¹ Vgl. Dr.-Ing. Mathias Magdowski, „Versionskontrolle mit Apache Subversion.“. Geprüft am 8. Mai 2017. Online: http://www.studentbranch.ovgu.de/studentbranch_media/Downloads/IEEE+Versionskontrolle+Workshop/Versionskontrolle_mit_Apache_Subversion.pdf, S. 28.

Ständen, da nun zu jedem Zeitpunkt eine genaue Revision zugeordnet werden kann.²² SVN erlaubt das Verschieben und Umbenennen von Dateien.²³ Auch können gelöschte Dateien mühelos wiederhergestellt werden, da die Versionsgeschichte erhalten bleibt. Darüber hinaus sind Logdateien verschiedener Benutzer sowohl für einzelne Dateien oder Verzeichnisse, als auch für Revisionen abrufbar. Weitere Merkmale von Subversion sind die Fähigkeit gleichzeitig entwickeln zu können und das Erkennen und Behandeln von Konflikten.²⁴

Wie schon bei CVS ist Subversion beim Open-Source-Segment angesiedelt und wird hauptsächlich über eine Kommandozeile bedient. Allerdings wurden auch hier zahlreiche verschiedene grafische Oberflächen für die Bedienung von SVN entwickelt. Die Bedienung zum Vorgänger ist sehr ähnlich. Die erste lauffähige Version von SVN wurde Anfang 2004 veröffentlicht.²⁵

Bei Subversion unterscheidet man zwischen drei verschiedenen Entwicklungszeiten. Der erste ist der Hauptentwicklungszweig, welcher auch Trunk genannt wird. Aus diesem Trunk resultieren die Nebenzweige und sind auch bekannt als Branches. Als letztes gibt es die Tags. Diese werden auch als Momentaufnahmen bezeichnet. Sie werden nicht mehr verändert und erweisen sich dann als nützlich, wenn man auf spezielle Versionen zugreifen möchte, ohne die Revisionsnummer zu kennen.²⁶

Eine weitere Besonderheit bei Subversion ist das es abgesehen vom Trunk, zwischen diesen Zweigen technisch gesehen keinen Unterschied gibt. Jeder

²² Vgl. Fünter, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFm2Re4BosAJpixqOA, S.6.

²³ Vgl. Dr.-Ing. Mathias Magdowski, „*Versionskontrolle mit Apache Subversion*“, (wie Anm. 12), S. 28.

²⁴ Vgl. Sieverdingbeck, Ingo and Jasper van den Ven, „Versionsverwaltung mit SVN.“. Geprüft am 8. Mai 2017. Online: <http://rn.informatik.uni-bremen.de/lehre/c++/svn.pdf>, S. 9.

²⁵ Vgl. Fischer, Lars, „Werkzeuge zum Versions- und Variantenmanagement: CVS/Subversion vs. ClearCase.“. Geprüft am 17. Mai 2017. Online: <https://www.wi1.uni-muenster.de/pi/lehre/ws0506/skiseminar/versionsverwaltung.pdf>, S. 3.

²⁶ Vgl. Fünter, „GIT & SVN“, (wie Anm. 20), S. 6.

einzelne dieser Zweige beginnt seine Versionsgeschichte beim Hauptentwicklungszweig oder bei einem anderen bereits existierenden Nebenzweig. Die Einordnung welcher Zweig was darstellt erfolgt letztendlich über die Benennung der Verzeichnisse im Repository und wird auch dementsprechend verwendet.²⁷

Bei der Arbeit mit Subversion wird zunächst einmal eine Arbeitskopie von der Datei mit der man arbeiten möchte erstellt. Diese Arbeitskopie wiederum wird aus dem Repository ausgecheckt. Mit dieser kann man wie gewohnt arbeiten ohne dafür Subversion verwenden zu müssen. Lediglich das Hinzufügen oder das Entfernen von Dateien erfolgen über Subversion, da dieses auch ganze Verzeichnisse versioniert. Für den Fall, dass man Änderungen verwerfen möchte, wird im Hintergrund eine ursprüngliche Version der aktuellen Arbeitskopie gespeichert. Abgeschlossene Änderungen müssen mit dem Repository zusammen aktualisiert werden, um Veränderungen im Repository zu erkennen, welche in der Zwischenzeit evtl. getätigt wurden. Die Veränderungen werden durch Subversion automatisch in die veränderte Arbeitskopie übernommen.

Eine Ausnahme besteht hier allerdings bei Konflikten. Wenn sich Änderungen überschneiden, müssen diese manuell gelöst und der SVN mitgeteilt werden. Als letzter Schritt ist ein Commit notwendig. Diesem wird eine kurze Erklärung beigefügt, welche die Revisionsänderungen beschreibt. Diese ist später im log-Protokoll wiederzufinden. Wie schon weiter oben erklärt, aktualisiert Subversion das Repository und erhöht die Revision um eins.²⁸

²⁷ Vgl. Fünten, „GIT & SVN,“ (wie Anm. 20), S. 6.

²⁸ Vgl. Fünten, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online:

https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFm2Re4BosAJpixqOA, S. 6f.

Das Repository ist durch das lokale Dateisystem über ein Interface zu erreichen. Die Protokollierung der Änderungen erfolgt entweder über eine Datenbank oder direkt über das Dateisystem.²⁹

2.5 Geschichte der Versionsverwaltung

Zu versuchen die Geschichte der Versionsverwaltung in vollem Maße darzustellen, würde den Rahmen dieser Arbeit übersteigen, da im Laufe der Zeit eine große Anzahl an Versionskontrollsystemen entstanden sind. Anhand verschiedener wegweisender Systeme ist es jedoch möglich zu der Entstehung der Entwicklung von Git hinzuführen und diese nachzuvollziehen.³⁰

2.5.1 SCCS

Das Source Code Control System wurde Anfang der 1970er Jahre durch M. J. Rochkind unter Unix entwickelt. Es handelt sich hierbei um das wahrscheinlich erste VCS, welches auf dem Unix-System verfügbar war.³¹

Es wurde ein zentraler Speicher zur Verfügung gestellt, welcher als Repository bezeichnet wurde. Dieses Konzept hat sich durchgesetzt und existiert heute immer noch. Das SCCS bot natürlich zu der damaligen Zeit nur einige Grundfunktionalitäten. Entwickler konnten Dateien ausführen und diese Testen ohne sie beim Auschecken sperren zu müssen. Beim Bearbeiten einer Datei wiederum war dies unumgänglich. Nach getaner Arbeit wurde die Datei wieder eingchecked und die vorherige Sperrung wieder aufgehoben.³²

²⁹ Vgl. Mauel, Volker, „Vergleich von Git und SVN.“. Geprüft am 17. Juni 2017. Online: <https://www.volkermazel.de/Downloads/GitVsSvn.pdf>, S. 7.

³⁰ Vgl. Lämmer, „Traditionelle Webentwicklung vs Versionsverwaltung,“ (wie Anm. 15).

³¹ Vgl. Loeliger, Jon, „Version control with Git.“. Geprüft am 29. August 2017. Online: <http://www.foo.be/cours/dess-20122013/b/OReilly%20Version%20Control%20with%20GIT.pdf>, S. 5.

³² Vgl. Loeliger, „Version control with Git,“ (wie Anm. 31), S. 5.

2.5.2 CVS

Walter Tichy entwickelte Anfang der 80er Jahre das Revision Control System (RCS). Durch das RCS konnte man auf eine effiziente Art und Weise verschiedene Überprüfungen, die man an Dateien vollzogen hatte, speichern.³³

Auf der Grundlage des RCS-Modells wurde einige Jahre später das Concurrent Versions System entwickelt. Dieses bot einige Vorteile gegenüber seinem Vorgänger auf. Das CVS war sehr beliebt und wurde eine lange Zeit als die beliebteste Versionsverwaltung in der Open Source-Gemeinde angesehen.³⁴ Hauptsächlich fand die Bedienung hier über eine Kommandozeile statt.³⁵, was alle Versionsverwaltungen im Grunde gemeinsam haben.

Durch CVS wurde bei Umgang mit Locks (Sperrung) eine neue Ideologie geschaffen. Dies ist wahrscheinlich die wichtigste Erneuerung, welche es mit sich gebracht hat. Erstmals war es möglich private Arbeitskopien von Dateien zu erstellen, an denen man arbeiten wollte. Dies hatte den Vorteil, dass die jeweilige Datei nicht gesperrt werden musste. Somit konnten andere Entwickler ebenfalls zu derselben Zeit an derselben Datei arbeiten, ohne dabei die Arbeit des anderen zu verhindern.

Diese verschiedenen Änderungen durch verschiedene Entwickler konnten später zusammengeführt werden, ohne dass die Arbeit des Anderen dadurch verloren ging.

Der einzige Konflikt bestand in dem Fall das Zwei verschiedene Entwickler dieselbe Zeile bearbeiten wollten. In diesem Fall wurden auf beide Verände-

³³ Vgl. Loeliger, Jon, „Version control with Git.“. Geprüft am 29. August 2017. Online: <http://www.foo.be/cours/dess-20122013/b/OReilly%20Version%20Control%20with%20GIT.pdf>, S. 5.

³⁴ Vgl. Loeliger, „*Version control with Git*,“ (wie Anm. 31), S. 5.

³⁵ Vgl. Fischer, Lars, „Werkzeuge zum Versions- und Variantenmanagement: CVS/Subversion vs. ClearCase.“. Geprüft am 17. Mai 2017. Online: <https://www.wi1.uni-muenster.de/pi/lehre/ws0506/skiseminar/versionsverwaltung.pdf>, S. 3.

rungen hingewiesen und es musste sich für eine Veränderung entschieden werden.³⁶

2.5.3 SVN

Subversion ist ein zentrales Versionsverwaltungssystem und basiert auf einer Client- Server-Architektur.³⁷

Typischerweise entstand Subversion aus den Mängeln und Fehlern, die bei CVS ausgemacht wurden. Der Unterschied zum Vorgänger bestand darin, dass Änderungen direkt gespeichert wurden und der Umgang mit den verschiedenen Entwicklungszweigen sichtbar besser zu handhaben war. Aufgrund dessen wurde Subversion nach seiner Vorstellung im Jahre 2001 schnell zu einer der beliebtesten Versionsverwaltungen in der Softwaregemeinde.³⁸

Die wohl bekannteste und gängigste Versionsverwaltung unserer Zeit ist die Software Git, welches nach Subversion entstanden ist und das eigentliche Thema dieser Arbeit ist. Auf diese Software wird in Kapitel 3 näher eingegangen und ist dementsprechend an dieser Stelle nicht aufgeführt.

2.6 Funktionsweise

Ohne zu verstehen wie Versionskontrolle funktioniert und aus welcher Notwendigkeit heraus die Versionskontrolle entstanden ist, ist es schwer nachzuvollziehen was genau bei einer Versionsverwaltungssoftware passiert und was bestimmte Begriffe innerhalb dieser Systeme Bedeuten und auch bewirken.

³⁶ Vgl. Loeliger, Jon, „Version control with Git.“. Geprüft am 29. August 2017. Online: <http://www.foo.be/cours/dess-20122013/b/OREilly%20Version%20Control%20with%20GIT.pdf>, S. 5.

³⁷ Vgl. Denker, Merlin und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog“ (2015): Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf; S. 11.

³⁸ Vgl. Loeliger, „*Version control with Git*,“ (wie Anm. 31), S. 5.

In diesem Abschnitt wird zunächst auf Probleme bei der Bearbeitung einer einzelnen Datei oder bei der Arbeit in einem Großprojekt bei Verzicht auf Verwendung einer Versionskontrolle eingegangen.

Daraufhin werden für die entsprechenden Problemstellungen Lösungen aufgezeigt, welche die Funktionsweise der heutigen Versionskontrolle darstellt.

2.6.1 Problem(e)

Man stelle sich folgendes Szenario vor. Es befinden sich zwei Nutzer in einem gemeinsamen Projekt, welche auf ein gemeinsames Repository zugreifen und dieselbe Datei bearbeiten möchten. Beide checken das Dokument aus und fangen an es lokal zu bearbeiten, wobei jeweils beide verschiedene Teile desselben Dokuments verändern. Die dabei neu entstandenen Versionen werden nacheinander zurück im Repository abgelegt. Hierbei entsteht folgendes Problem: Wenn man davon ausgeht, dass der zweite Nutzer seine Arbeit nach dem ersten Nutzer im Repository ablegt, dann wird dabei die Arbeit des ersten Nutzers überschrieben und geht somit verloren.³⁹

³⁹ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017. Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 45.

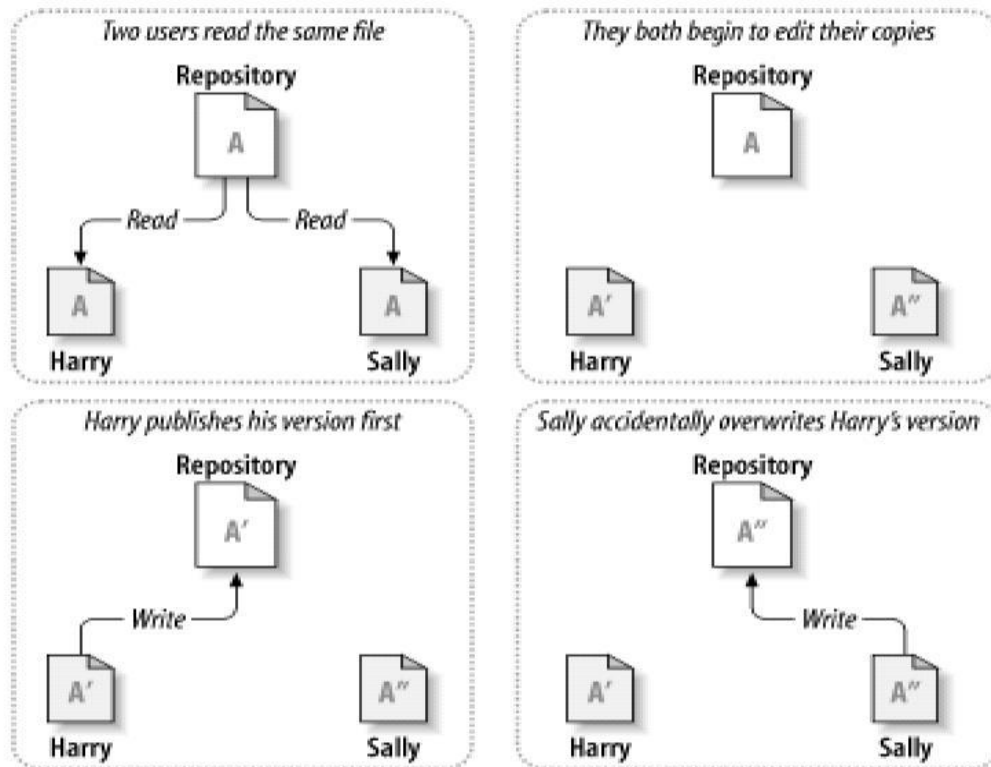


Abbildung 1: Arbeit ohne Versionskontrolle

Quelle: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 43, Geprüft am 17. Juni 2017.

2.6.2 Lösungswege

Eine mögliche Lösung für dieses Problem wäre das Sperren von Dokumenten. Durch das Auschecken aus dem Repository sperrt der erste Nutzer die zu bearbeitende Datei und arbeitet lokal an ihr. Währenddessen möchte die zweite Person auf das Dokument zugreifen. Jedoch ist der Zugriff gesperrt und der zweite Nutzer muss warten, bis der erste Nutzer seine Arbeit an dem Dokument beendet hat.⁴⁰

⁴⁰ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017. Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 46.

Nach getaner Arbeit wird das Dokument zurück ins Repository gelegt und freigegeben. Nun hat die zweite Person Zugriff darauf und kann das Dokument ebenfalls für andere sperren.⁴¹

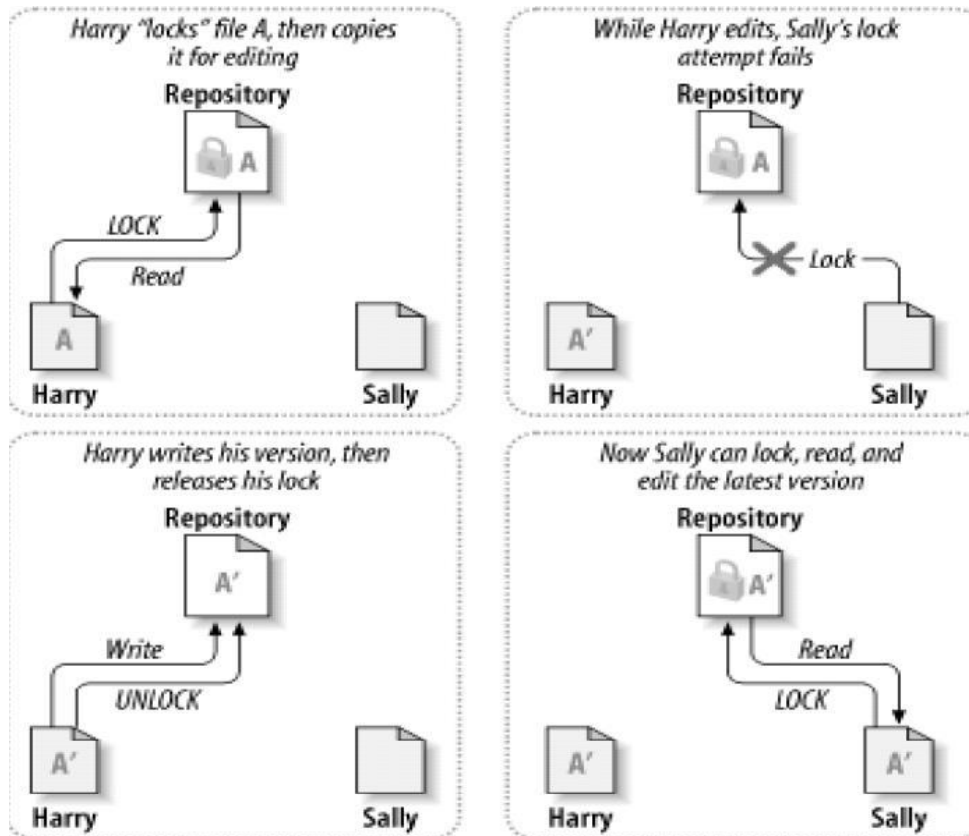


Abbildung 2 Sperren der Dokumente

Quelle: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 43, Geprüft am 17. Juni 2017.

Der hier angegebene Schritt ist zwar ein guter Lösungsvorschlag, dennoch ist er nicht optimal. Es kann durchaus vorkommen, dass gesperrte Dokumente schlicht und einfach vergessen werden, wieder zu entsperren. Die nächste Person, die eben an diesem Dokument arbeiten möchte, müsste sich zunächst mühsam darum kümmern, dass eine Entsperrung erfolgt. Dies würde kostbare Zeit im Laufe des Projektes rauben.

⁴¹ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017. Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 43.

Darüber hinaus würde eine Sperrung des Dokuments einen unnötigen Zeitverlust für denjenigen darstellen der zwar dieselbe Datei bearbeiten möchte, jedoch an einer ganz anderen Stelle. Gleichzeitiges Entwickeln wäre also in diesem Fall nicht möglich.

Ein weiterer wichtiger Aspekt ist die Abhängigkeit von Dokumenten. Beim Arbeiten zweier Personen an zwei verschiedenen Dokumenten, welche in Abhängigkeit zueinander stehen, könnten Probleme aufkreuzen. Änderungen könnten bewirken das Dokument A und Dokument B nicht mehr so zusammenspielen wie sie es müssten. Für das Lösen dieses Problems wäre ein Gespräch notwendig.⁴²

Ein weiterer Lösungsansatz ist das Mischen von Dokumenten. Beide Nutzer können gleichzeitig das Dokument, welches sie bearbeiten möchten aus dem Repository heraus kopieren. Dies ermöglicht ihnen das gleichzeitige Entwickeln am selben Dokument unabhängig voneinander. Einer der Beiden legt als erster seine lokale geänderte Version in das Repository zurück. Der zweite möchte dasselbe tun, muss jedoch zunächst die lokale Version mit der aktuellen Version im Repository vergleichen. Dadurch werden beide Versionen des Dokuments vermischt und die jeweiligen Unterschiede übertragen. Somit kann eine völlig neue Version des Dokuments entstehen, welche die Änderungen beider Nutzer beinhaltet.⁴³

Zu beachten ist, dass es bei diesem Lösungsansatz zu Konflikten kommen könnte. Wenn beide Nutzer etwa dieselbe Zeile bearbeiten entsteht eine Konfliktsituation, die manuell und nach einer Absprache der beiden gelöst werden muss. Hierfür stehen einige Werkzeuge bereit, welche die Unterschiede zwischen den zwei Versionen anzeigen. Jedoch zeigt die Praxis, dass sich Konfliktsituationen eher selten ergeben und Änderungen an Doku-

⁴² Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017.
Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 43.

⁴³ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten,“ (wie Anm. 5), S. 48.

menten zusammengeführt werden können, ohne dass sich dabei Probleme aufzeigen.⁴⁴

2.7 Arten der Versionsverwaltung

Bei der Arbeit mit Versionsverwaltung unterscheidet man zwischen drei verschiedenen Arten. Die erste Art ist die Lokale Versionsverwaltung, welche komplett auf dem lokalen Computer des Anwenders stattfindet.

Bei der zweiten Art handelt es sich um die zentrale Versionsverwaltung. Hierbei ist die Kommunikation mit einem zentralen Server für den Austausch von Dokumenten notwendig.

Die dritte und letzte Art ist die verteilte Versionsverwaltung. Bei dieser Art der Versionsverwaltung wird zwar auch ein Server verwendet, auf dem die gesamten Daten gespeichert sind, jedoch existiert gleichzeitig auch eine Kopie des Repository auf dem lokalen Rechner mit dem gearbeitet werden kann.

Jeder dieser Arten hat seine Vor- und Nachteile. In diesem Teil der Arbeit wird erläutert was diese drei Arten der Versionsverwaltung genau sind und wie diese funktionieren.

2.7.1 Lokale Versionsverwaltung

Diese Art der Versionskontrolle wird als die einfachste angesehen. Meistens ist hierfür keine Software für die Verwendung notwendig. Es handelt sich lediglich um einen selbst angelegten Ordner der lokal verwaltet wird. Dieser beinhaltet verschiedene Versionen von dem Projekt an dem man gerade arbeitet.

So einfach der Umgang mit dieser Art der Versionskontrolle auch ist, weist sie viele Schwachstellen auf. Ein effektives Arbeiten an einem Großprojekt als Team ist bei lokaler Versionsverwaltung nicht möglich. Die Gefahr be-

⁴⁴ Vgl. taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017. Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>, S. 50.

stimmte Feature doppelt zu entwickeln oder sie gar zu vergessen ist hier sehr groß. Falls keine weitere Software bei der Verwendung hinzugezogen wird, kann es dazu kommen, dass Versionsnummern nicht richtig vergeben werden. Als Folgefehler könnten dann falsche Dokumente in den falschen Verzeichnissen landen, was zu einem einzigen Chaos führen kann.⁴⁵

2.7.2 Zentrale Versionsverwaltung

Zu diesem System gehören z.B. die weiter oben beschriebenen CVS und dessen Nachfolger Subversion. Hierbei werden die gesamten Daten auf einem zentralen Server gespeichert. Auf lokaler Ebene werden nur die Arbeitskopien gespeichert. Die neu entstandenen Versionen werden zurück auf das Repository gelegt.⁴⁶

Beim gleichzeitigen Entwickeln zweier Benutzer an demselben Dokument, wird der Merge, also das verschmelzen der beiden Dateien, auf dem Server automatisch erledigt. Voraussetzung hierfür ist, dass sich die beiden Dokumente ausreichend unterscheiden. Ansonsten muss eine manuelle Änderung an den Stellen der Datei erfolgen, die sich zu ähnlich sind. Ein Beispiel hierfür ist wie schon erwähnt, die Bearbeitung der exakt selben Zeile.⁴⁷

2.7.3 Nachteile der zentralen Versionsverwaltung

Im Allgemeinen ist gegen die Nutzung eines zentralen Systems, welches bis dahin immer funktioniert hat, nichts einzuwenden. Jedoch beherbergt diese Art der Versionskontrolle einige Nachteile.

Die Skalierung bei zentralen Versionsverwaltungssystemen, geht nicht so gut von statten wie bei anderen. Hat das Team eine bestimmte Größe erreicht, kommt es zu Komplikationen bei der Zurückstellung der Arbeit. Die Wahr-

⁴⁵ Vgl. Dederer, Paul, Matrikel-Nr.: 245211, paul.dederer@hs-furtwangen.de, „Effiziente Softwareentwicklung durch Versionskontrolle,“ (2016): Geprüft am 8. Mai 2017, S. 4.

⁴⁶ Vgl. Burch, Philipp, „Versionsverwaltung mit Mercurial (Teil 2) - ActiveVB.“. Geprüft am 11. April 2017. Online:
https://activevb.de/tutorials/tut_versionsverwaltung/tut_versionsverwaltung_2.html.

⁴⁷ Vgl. Burch, „*Versionsverwaltung mit Mercurial (Teil 2) - ActiveVB*,“ (wie Anm. 46).

scheinlichkeit dass sich der Zustand des Servers zwischen einem Checkout und einem Checkin verändert ist zu groß. Dadurch muss der aktuelle Stand immer wieder vom Server geladen und vor dem zurückstellen neu getestet werden.⁴⁸

In heutigen Projekten werden entwickelte Features erst committet, wenn sie bereit gestellt und praktisch implementierbar sind. Dadurch verlängern sich die Entwicklungszeiten und die Zwischenstände, welche früher üblich waren, entfallen. Dementsprechend entstehen größere Änderungen, welche schwerer zu überblicken sind. Als Letztes kommt hinzu, dass das mobile Arbeiten, durch die Notwendigkeit einer Netzwerkverbindung, erschwert wird.⁴⁹

Die verteilte Versionsverwaltung zeigt Wege auf, um diese Probleme zu lösen. Dabei kommt es vermehrt zum Einsatz von Entwicklungszweigen, welche natürlich auch schon bei CVS und Subversion existieren, jedoch nicht so einfach zu handhaben sind. Vor allem das Zusammenführen zweier Entwicklungszweige ist mit einem großen Zeitaufwand verbunden.⁵⁰

2.7.4 Zentral versus verteilt

Bei der zentralen Versionsverwaltung landen Revisionen in der Regel im dafür vorgesehenen Repository. Die momentane Arbeit wird zunächst einmal auf der Arbeitskopie gespeichert. Dadurch häufen sich im Laufe der Zeit die Änderungen an, welche später auf dem Server abgelegt werden müssen. Auch werden Komplexe Änderungen an Dokumenten, welche bis zu diesem Zeitpunkt noch nicht die gewünschte Stabilität aufweisen, dadurch erschwert.⁵¹

⁴⁸ Vgl. Stargardt, Niels, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung.“. Geprüft am 17. Juni 2017. Online: https://www.sigs-dacom.de/uploads/tx_dmjournals/stargardt_JS_04_12_la6g.pdf, S. 1.

⁴⁹ Vgl. Stargardt, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“ (wie Anm. 48), S. 1.

⁵⁰ Vgl. Stargardt, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“ (wie Anm. 48), S. 1.

⁵¹ Vgl. Tartler, Reinhard and Martin Steigerwald, „Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin.“. Geprüft am 6. April 2017. Online: <http://www.linux-magazin.de/Ausgaben/2007/06/Markt-Modell>.

Um dieses Problem umgehen zu können ist es ratsam einen weiteren Branch anzulegen. Also einen weiteren Entwicklungszweig mit demselben Inhalt und diese beiden nach der Änderungen wieder zusammenführen. Diese Zusammenführung muss jedoch bei Systemen wie CVS und Subversion manuell erfolgen und ist daher ein zeitaufwändiger Prozess. Um zusätzlichen Datenverlust zu verhindern, muss der Entwickler einzelne Informationen über jede Revision mit dem Hauptzweig zusammenführen.⁵²

Der Vorteil der verteilten Versionsverwaltung an dieser Stelle ist, dass jeder Entwickler seinen eigenen Entwicklungszweig hat. Revisionen landen zunächst auf dem Entwicklungszweig des Entwicklers. Außerdem besitzt er darüber hinaus ein eigenes lokales Repository, welches die gemachten Revisionen aufzeichnet. Getätigte Commits sind dadurch jederzeit konfliktfrei.⁵³

2.7.5 Verteilte Versionsverwaltung

Von den Fähigkeiten und Techniken her unterscheiden sich verteilte Versionsverwaltungen im Prinzip nicht von anderen. Der Unterschied liegt darin, dass jeder Nutzer auf seinem lokalen System nicht nur einen eigenen Arbeitsbereich besitzt, sondern auch eine vollständige und voll funktionsfähige Kopie des Repository. Diese beinhaltet ebenso eine Historie. Jeder Nutzer erhält durch das DVCS (engl. Distributed Version Control System) zusätzlich einen eigenen Entwicklungszweig. Durch die Client-Server-Architektur der zentralen Versionskontrolle (engl. Centralized Version Control System) existiert hier eine Entwicklungslinie, die sich nicht wiederholt.⁵⁴

Bei dieser Art der Versionsverwaltung herrscht eine hierarchische Ordnung zwischen den Repositories. Der Austausch zwischen den Teilteams erfolgt normalerweise über ein oder mehrere zentrale Repositories, von dem jedes

⁵² Vgl. Tartler, Reinhard and Martin Steigerwald, „Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin.“. Geprüft am 6. April 2017. Online: <http://www.linux-magazin.de/Ausgaben/2007/06/Markt-Modell>.

⁵³ Vgl. Tartler und Steigerwald, „Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin,“ (wie Anm. 51).

⁵⁴ Vgl. Bechara, John, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement.“ (2009). Geprüft am 17. Juni 2017. Online: <https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf>, S. 11.

Team ein eigenes erhält. Dies wäre für ein zentrales Versionsverwaltungssystem unüblich.⁵⁵

2.7.6 Vorteile der verteilten Versionsverwaltung

Die Nutzung eines verteilten Systems bringt in erster Linie technische Vorteile mit sich. Das Netzwerk wird entlastet da es die meiste Zeit nicht benötigt wird. Dies erhöht gleichzeitig die Performanz. Auch sinken die Ansprüche gegenüber den technischen Administratoren, welche dadurch entlastet werden. Die Nutzungsmöglichkeit zahlreicher Workflows wird dadurch ebenfalls ermöglicht. Änderungshistorien werden überschaubarer und noch dazu kommt, das Code-Reviews besser durchgeführt werden können. Dies führt zu einer erheblichen Steigerung der Qualität. Ein Grund dafür ist, dass in verteilten Systemen die Nutzung von Branches, welche explizit für Features genutzt werden, gängig ist. Diese werden in den jeweiligen Branch zu Ende entwickelt und letztlich mit dem Hauptentwicklungszweig zusammen geführt. Der Vorteil hierbei besteht darin, dass die entsprechenden Features bearbeitet werden können, ohne einen negativen Effekt auf den Rest des Projekts zu haben. Darüber hinaus erleichtert dies weitere Code-Reviews und Änderungen am Code sind besser nachvollziehbar.⁵⁶

Die verteilte Versionsverwaltung behebt einige der Probleme, welche bei der zentralen Versionskontrolle vorkommen. Da das zentrale Repository bei einem verteilten System nicht so häufig aktualisiert werden muss, wie bei einem zentralen System, erleichtert sich dadurch die Skalierung. Da einzelne Commits auch lokal erfolgen können, werden Arbeiten aus Teilteams zunächst aus deren eigenen Repositories gesammelt und Stück für Stück zusammengesetzt.⁵⁷

⁵⁵ Vgl. Stargardt, Niels, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung.“. Geprüft am 17. Juni 2017. Online: https://www.sigs-datacom.de/uploads/tx_dmjournals/stargardt_JS_04_12_la6g.pdf, S. 1.

⁵⁶ Vgl. Stargardt, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“ (wie Anm. 48), S. 35.

⁵⁷ Vgl. Stargardt, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“ (wie Anm. 48), S. 35.

2.8 Verschiedene Arbeitsweisen der Versionsverwaltung

Im Bereich der Versionsverwaltung existieren zwei verschiedene Arbeitsweisen. Jedes dieser Konzepte bringt seine eigenen Vor- und Nachteile mit sich. Welche Arbeitsweise man bevorzugt, hängt stark von der eigenen Philosophie oder der Unternehmensideologie ab.

2.8.1 Lock Modify Write

Die hier zugrunde liegende Philosophie wird auch als pessimistische Versionskontrolle bezeichnet. Wie man es aus dem Namen schon entnehmen kann, werden bei dieser Arbeitsweise Dateien oder Dokumente während dem Arbeiten gesperrt, um sie für andere unzugänglich zu machen. Erst wenn die Arbeit an einem bestimmten Dokument vollendet ist, wird diese wieder freigegeben. Diese Vorgehensweise bringt jedoch einige Nachteile mit sich. Zunächst einmal entstehen durch dieses Vorgehen administrative Probleme. Wenn es beispielsweise dazu kommt, dass ein Nutzer vergisst die Sperrung einer Datei wieder aufzuheben oder wenn das Client-System abstürzt. In diesem Fall muss die Sperrung unter großem Aufwand wieder entsperrt werden. Meistens ist es dazu notwendig einen zuständigen Administrator hinzuzuziehen.⁵⁸

2.8.2 Copy Modify Merge

Aufgrund einiger Nachteile die beim Lock Modify Write vorhanden sind, wurde die Arbeitsweise des Copy Modify Merge entwickelt. Auch hier kann aus dem Namen die Vorgehensweise entnommen werden. Arbeiten werden in einer eigenen Sandbox verrichtet und später auf den Server hochgeladen. Bei diesem Modell wird versucht den Vorgang des Sperrens zu verhindern. Dadurch fällt es unter die Kategorie der optimistischen Versionskontrolle.

⁵⁸ Vgl. Bechara, John, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement.“ (2009). Geprüft am 17. Juni 2017. Online: <https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf>, S. 8.

Eine Auswirkung davon kann jedoch sein, dass eine weitere Person an derselben Datei arbeitet und diese in der Zwischenzeit auf das Repository hochlädt. In diesem Fall ist es notwendig, die neuere Version mit der vorher hochgeladenen zu verschmelzen bzw. zusammenzuführen. Diesen Vorgang bezeichnet man als Merge.⁵⁹

2.9 Zusammenfassung

In diesem Kapitel wurde die Versionsverwaltung/-kontrolle zusammen mit ihren Vor- und Nachteilen beleuchtet. Des Weiteren wurde ein grober Überblick über die Entstehungsgeschichte der Versionsverwaltungssysteme gegeben und die gängigsten Systeme der letzten Jahre bis heute aufgezeigt.

Die verschiedenen Arten der Versionsverwaltung, darunter die zentrale und die verteilte Versionsverwaltung, und deren Unterschiede, Vor- und Nachteile wurden ebenfalls behandelt. Wichtige Schlüsselbegriffe wie Repository, Merge, Commit und Branch wurden erläutert.

⁵⁹ Vgl. Bechara, John, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement.“ (2009). Geprüft am 17. Juni 2017. Online: <https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf>, S. 9.

3 Git

Im Laufe dieser Arbeit wurden diverse Versionsverwaltungssysteme genannt. Auf die zwei wohl bekanntesten zentral ausgerichteten, CVS und SVN, wurde ausführlicher eingegangen, um sich ein besseres Bild darüber machen zu können, wie Versionskontrolle funktioniert. Eine wesentliche Rolle spielt hierbei zu welcher Art der Versionsverwaltung diese Systeme gehören, denn jede Art der Versionskontrolle bringt seine eigenen Vor- und Nachteile mit sich.

Das Versionsverwaltungssystem, welches das Hauptaugenmerk dieser Arbeit bildet ist die Versionsverwaltung namens Git. Die Besonderheit dieses Systems besteht darin, dass es zur Art der verteilten Versionsverwaltung gehört.

Ziel dieses Kapitels ist es, die Versionsverwaltung Git näher kennenzulernen, die Funktionalitäten und auch den Aufbau zu verstehen. Darunter gehört auch das bessere Verständnis für die verteilte Versionsverwaltung. Auch werden die Vorteile von Git gegenüber anderen Systemen genannt und warum Git im Laufe eines Projektes, anderen Systemen vorgezogen werden sollte.

Ebenso wird in diesem Kapitel auf viele wichtige Begriffe eingegangen, welche bei der Versionskontrolle unabdingbar sind. Entwicklungszweige werden näher erläutert und auch das Zusammenführen von verschiedenen Versionen einer Datei.

Hauptaufgaben der Versionskontrolle sowie die Bedienung von Git werden ebenfalls beschrieben.

3.1 Die Entstehung von Git

Während der Entwicklung des Linux-Kernels wurden Änderungen am Quellcode in Form von Patches übergeben und weitergereicht. Im Jahre 2002 ent-

schied man sich dafür BitKeeper zu verwenden. BitKeeper ist ein Source-Control-Management (SCM) System und erleichtert die Arbeit bei der Verwaltung von Quellcode.⁶⁰ Bis in das Jahr 2005 benutzten die Entwickler von Linux das System BitKeeper um ihre Versionen zu verwalten. Dies jedoch war ab diesem Zeitpunkt aufgrund einer Änderung in der Lizenz nicht mehr ohne weiteres möglich. Daraufhin begann Linux-Gründer Linus Torvalds mit der Entwicklung von Git. Hauptaugenmerk bei der Entwicklung war die Nutzung der verteilten Versionsverwaltung, die Sicherheit gegen Verfälschung und eine hohe Effizienz. Wenige Tage nach dem Projektbeginn konnte Torvalds bereits eine erste Version vorstellen, welche diesen Anforderungen gerecht wurde.⁶¹

Git gehört zu den jüngeren Versionsverwaltungssystemen, obwohl dessen Entwicklungsbeginn schon mehr als zehn Jahre zurück liegt und im Jahre 2005 begann. Die Arbeitsabläufe von Git sollten den Arbeitsabläufen von BitKeeper ähnlich sein. Fast hätte er sich dazu entschieden »Monotone« zu benutzen, jedoch war dieses Programm nicht effizient genug. Schlussendlich entschied er sich dafür, ein komplett eigenes und neues Programm zu schreiben.⁶²

Git weist einige Konzeptmerkmale von BitKeeper und Monotone auf. Jedoch wurde kein Quellcode dieser Systeme bei der Entwicklung verwendet. Git wurde von Null auf entwickelt und enthält nur selbst geschriebenen Programmcode seitens Torvalds und seinem Team.⁶³

⁶⁰ Vgl. THM-Wiki, „Git-basierte kollaborative Entwicklung von Webanwendungen (GitHub/GitLab) – THM-Wiki,“. Geprüft am 6. April 2017. Online: [https://wiki.thm.de/Git-basierte_kollaborative_Entwicklung_von_Webanwendungen_\(GitHub/GitLab\)](https://wiki.thm.de/Git-basierte_kollaborative_Entwicklung_von_Webanwendungen_(GitHub/GitLab)).

⁶¹ Vgl. Maue, Volker, „Vergleich von Git und SVN.“. Geprüft am 17. Juni 2017. Online: <https://www.volkermaue.de/Downloads/GitVsSvn.pdf>, S. 6.

⁶² Vgl. Vijayakumaran (wie Anm. 2), S. 21.

⁶³ Vgl. THM-Wiki, „Git-basierte kollaborative Entwicklung von Webanwendungen (GitHub/GitLab) – THM-Wiki,“ (wie Anm. 60).

Umgangssprachlich bedeutet das Wort "Git" im britischen so viel wie "Blödmann". Seiner Meinung nach ist dieser Name praktikabel und in der Welt der Software noch nicht im Einsatz und dementsprechend eine gute Wahl.⁶⁴

Als weitere Begründung sagte er spaßeshalber:

„I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'.“ (Linus Torvalds)

3.2 Die Grundlagen und Eigenschaften von Git

Git agiert nicht wie die meisten Versionsverwaltungssysteme. Diese erfassen nämlich die Änderungen an einer ursprünglichen Datei als Informationen, die der Reihe nach ablaufen. Diese Änderungen werden als „Diffs“ bezeichnet.⁶⁵ Bei Git wiederum werden die Versionen als Momentaufnahmen gespeichert. Sobald man eine neue Version der bearbeiteten Datei in der entsprechenden Datenbank ablegen möchte, erstellt Git eine solche Momentaufnahme aller vorhandenen Dateien und hält diese fest. Um effizienter arbeiten zu können, wird auf die vorherige Version eine Referenz angelegt. Das bedeutet, unveränderte Dateien werden nicht kopiert, sondern es werden lediglich Verknüpfungen zu diesen vorherigen Versionen erstellt.⁶⁶ Diese Referenzierung erfolgt bei Git nicht über einen Namen, sondern über eine Checksumme, welche 40 Zeichen lang ist und als Hash-Wert bezeichnet wird. Berechnet wird diese entweder aus dem Inhalt oder der Verzeichnisstruktur und verhindert

⁶⁴ Vgl. tinatigertech, „„Git“ your work done and don't mess with your team!“, Geprüft am 11. April 2017. Online: <http://www.tigertech.de/git-your-work-done-and-dont-mess-with-your-team/#more-629>.

⁶⁵ Vgl. Chacon, Scott and Ben Straub, „Git - Book.“. Geprüft am 19. April 2017. Online: <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen>.

⁶⁶ Vgl. Fünten, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFM2Re4BosAJpixqOA, S. 10.

Übertragungsfehler oder hilft dabei beschädigte Dateien ausfindig zu machen.⁶⁷

Wenn bei jeder neuen Versionierung immer nur die Diffs gespeichert würden, würde zwar weniger Speicher verwendet werden, jedoch wäre dies viel langsamer. Vor allem dann wenn ein älterer Stand ausgecheckt wird. Wenn eine Datei mehrere Male verändert wurde, müsste man beim Auschecken eines älteren Standes, bis zu diesem Zustand Diffs anwenden, was sehr zeitaufwändig ist. Bei Git allerdings wird eine Datei einfach als Ganzes ausgecheckt, ohne dass dafür noch weitere Operationen nötig sind.⁶⁸

Hierin liegt ein wichtiger Unterschied zu Git und nahezu allen anderen Versionskontrollsystemen. Die Versionskontrolle an sich wurde bei Git in fast allen Bereichen neu durchdacht, während andere Systeme ihre Eigenschaften auf die ihrer Vorgängerversionen aufbauen. Grob betrachtet könnte man sagen, Git arbeitet nicht wie eine herkömmliche Versionsverwaltung, sondern ähnelt einem kleinen Dateisystem, welches wichtige Werkzeuge besitzt.⁶⁹ Die Sammlung aus Dateien, die sich darin befindet, besteht aus Schlüssel-Daten Paaren. Der Schlüssel wiederum ist der weiter oben erwähnte Hash-Wert der Daten selber. Nicht nur den Dateien selber ist ein Hash-Wert zugeteilt, sondern ebenso den Verzeichnissen und auch den Commits. Der Hash-Wert für das Verzeichnis, enthält alle Informationen des bis dahin erstellten Dateibaums. Dasselbe gilt für den Hash-Wert des Commits, der dadurch alle früheren Commits beinhaltet.⁷⁰

Ein weiterer wichtiger Unterschied zu anderen Versionsverwaltungssystemen liegt darin, dass jeder Entwickler ein eigenes Repository bekommt, welches er nur einmal anlegen muss. Damit erhält er eine Kopie des Haupt-Repository auf seinem Rechner. Auf technischer Ebene gibt es zwischen

⁶⁷ Vgl. Denker, Merlin und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog“ (2015): Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf, S. 2.

⁶⁸ Vgl. Vijayakumaran (wie Anm. 2), S. 51.

⁶⁹ Vgl. Chacon, Scott and Ben Straub, „Git - Book.“. Geprüft am 19. April 2017. Online: <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen>.

⁷⁰ Vgl. Meyer, Eric A., „Git: Verteilte Versionsverwaltung.“. Geprüft am 6. April 2017. Online: <http://www.iks.kit.edu/diverses/gitvortrag/>, S. 14ff.

dem lokalen und dem Repository auf dem Server keinen Unterschied, jedoch beinhaltet das Master-Repository den Hauptentwicklungszweig.⁷¹

Bei Git existieren wie bei anderen Versionsverwaltungen auch Branches und Tags. Der Hauptentwicklungszweig heißt hier jedoch Master. Die Besonderheit von Git liegt darin, dass jeder Entwickler zusätzlich zu seinem eigenen Repository noch einen eigenen Entwicklungszweig erhält und dieser Zweig kann an ein anderes Repository übermittelt werden.⁷²

Bei den meisten ausgeführten Operationen werden zumeist Dateien benötigt, welche lokal auf dem Rechner vorhanden sind. Somit entfällt die Kommunikation mit einem anderen Rechner über ein Netzwerk und die damit verbundenen Wartezeiten, welche von einem Netzwerk normalerweise verursacht werden. Dadurch ermöglicht Git eine viel schnellere Arbeitsweise gegenüber zentralen Versionsverwaltungen, denn die gesamte Historie eines Projekts befindet sich ebenfalls auf dem lokalen Rechner und ermöglicht ein viel schnelleres Arbeiten ohne Verzögerungen.⁷³

Dateien können sich in drei verschiedenen Zuständen befinden. Die erste Form heißt Modified. Dies ist der Zustand wenn eine Änderung noch nicht in die lokale Datenbank aufgenommen wurde. Der zweite Zustand wird Staged genannt und tritt ein wenn eine Änderung für den nächsten Commit vorgemerkt wurde. Der letzte der drei Zustände ist der Committed-Zustand. Dieser ist erreicht, wenn die Änderungen in einer neuen Version festgehalten wird. Aus diesen wiederum leiten sich die drei Hauptbereiche ab, die man in einem Projekt in Git verwendet. Das Git-Verzeichnis, die Working Area und die Staging Area. Da es wichtige Metadaten und die lokale Datenbank enthält, ist das Git-Verzeichnis gleichzustellen mit dem Repository. Die Working Area

⁷¹ Vgl. Fünten, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online:

https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFm2Re4BosAJpixqOA, S. 9f.

⁷² Vgl. Fünten, „GIT & SVN“, (wie Anm. 20), S. 10.

⁷³ Vgl. Chacon, Scott and Ben Straub, „Git - Book.“. Geprüft am 19. April 2017. Online: <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen>.

dient dazu das gegenwärtige Projekt nach Belieben zu bearbeiten und zu modifizieren, weshalb es eine Version dieser beinhaltet. Die Staging Area enthält im Grunde nur eine Datei im Verzeichnis von Git, welche den Inhalt des nächsten Commit bestimmt.⁷⁴

Dementsprechend würde ein Arbeitsablauf bei Git wie folgt aussehen:

Zunächst wird durch einen Checkout eine Version der zu bearbeitenden Datei oder Dateien in die Working Area kopiert. Diese werden bis zum gewünschten Zustand bearbeitet. Als nächstes werden diese Änderungen in die Staging Area kopiert und somit für den nächsten Commit vorgemerkt. Zu guter Letzt werden diese Änderungen durch einen Commit dauerhaft in die Datenbank aufgenommen.⁷⁵

3.2.1 Die Staging Area oder auch der Index

Inhaltlich verteilen sich die Dateien bei Git auf drei Ebenen. Diese sind der Working Tree, der Index und das Repository. Dateien auf dem Working Tree sind gleichzustellen mit den Dateien eines privaten Arbeitsrechners, also ist das der Ort, auf dem man Änderungen lokal durchführt.

Das Repository ist, wie schon oft erwähnt, ein Behälter der alle gespeicherten Änderungen enthält. Aus der Sammlung dieser Änderungen entsteht die Versionsgeschichte oder auch Änderungshistorie.

Im Gegensatz zu vielen anderen Versionskontrollsystemen existiert bei Git eine dritte Ebene: der Index. Bei dieser Ebene handelt es sich um eine Zwischenebene, welche sich zwischen dem Repository und dem Working Tree einordnet. Der Index dient dazu Commits vorzubereiten, so dass man nicht gezwungen ist alle Änderungen, die man an einer Datei vorgenommen hat, zwingend als Commit einzuchecken.

⁷⁴ Vgl. Denker, Merlin und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog“ (2015): Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf, S. 2f.

⁷⁵ Vgl. Denker und Srecec, „*Versionsverwaltung mit Git*“, (wie Anm. 1), S. 3.

Durch die Git-Kommandos *add* und *reset* werden Änderungen zunächst auf den Index geladen bzw. vom Index entfernt. Erst durch einen *commit*-Befehl werden diese Änderungen in das Repository übertragen.⁷⁶

3.3 Vorteile von Git

Git bietet gegenüber anderen Versionsverwaltungssystemen einige Vorteile.

Ein Vorteil ist das Arbeiten mit Entwicklungszweigen. Im Umgang mit Git können viele Entwickler gleichzeitig auf verteilten Repositories arbeiten. Dies führt zu vielen verschiedenen Entwicklungslinien. Git besitzt die Eigenschaft diese verschiedenen Entwicklungslinien zusammenzuführen. Die Technik, die das bewerkstelligt nennt man Merging.⁷⁷

Eine weitere Besonderheit von Git ist die Anpassungsfähigkeit der Arbeitsabläufe. Manch einer würde behaupten Git sei gar keine Versionsverwaltungssoftware, sondern eher ein Baukasten, aus dem man seine eigene Versionsverwaltung zusammenfügen kann. Grund hierfür ist die Flexibilität von Git. Es spielt keine Rolle ob es in einem agilen Team eingesetzt wird oder ob ein einzelner Entwickler es privat nutzen möchte. Selbst in einem internationalen Großprojekt mit mehreren Standortübergreifenden Teams können passende Arbeitsabläufe je nach Gebrauch ausgearbeitet werden.⁷⁸

Git ist ebenfalls eine Open-Source-Software und ermöglicht jedem, der das möchte, etwas zu der Software beizutragen. Wichtig ist hierbei, das Beitragen so reibungslos wie nur möglich zu gestalten. Das wird durch die verteilte Arbeitsweise ermöglicht. Jeder kann bei Git ein bereits bestehendes Projekt auf seinen lokalen Rechner kopieren, es nach Belieben weiterführen und die

⁷⁶ Vgl. Haenel, Valentin and Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 34.

⁷⁷ Vgl. Preißel, René and Bjørn Stachmann. *Git. Dezentrale Versionsverwaltung im Team : Grundlagen und Workflows*. Heidelberg: dpunkt.Verlag, 2016, S. vii.

⁷⁸ Vgl. Preißel und Stachmann (wie Anm. 77), S. vii.

gemachten Änderungen wieder übergeben. Bei der zentralen Versionskontrolle wird das durch verschiedene Lese –und Schreibrechte erschwert.⁷⁹

Hohe Geschwindigkeit ist eine Eigenschaft, die jedes gute Softwareprogramm mit sich bringen muss. Auch hier kann Git überzeugen. Nicht nur das Git schnell ist, es bleibt auch schnell. Selbst wenn man zwischen Projekten wechselt, zwischen denen 200.000 Commits, 40.000 veränderte Dateien und mehrere Jahre der Entwicklung liegen.⁸⁰

Ein schwerwiegender Verlust von Daten wird bei Git durch das Verteilen der Änderungshistorie auf viele verschiedene Repositories verhindert. Die Struktur innerhalb der Repositories sorgt dafür, dass der Inhalt auch Jahre später Verständnisproblemen entgegenwirkt. Ein ständig eingesetzter Hash-Wert sorgt dafür, dass die Ordnung und die Echtheit der Repositories nicht durch Angriffe von außen beschädigt werden.⁸¹

Darüber hinaus sorgt die dezentrale Struktur von Git dafür, dass keine Netzwerkverbindung zu einem zentralen Server notwendig ist um zu entwickeln. Dadurch kann, z.B. mit einem Laptop, jederzeit und an jedem Ort weiter entwickelt werden. Lediglich das Übertragen der Änderungen auf das „zentrale“ Repository benötigt einen Online-Zugang.⁸²

Dank einer großen Open-Source-Anhängerschaft von Git, existieren neben der offiziellen Dokumentation, auch zahlreiche andere Hilfsmittel wie z.B. Anleitungen, Foren und Wikis, die den Anwendern unter die Arme greifen. Git weist auch eine stark wachsende Plattform für Tools, Hosting-Plattformen und anderen hilfreichen Plugins für die Entwicklung auf.⁸³

⁷⁹ Vgl. Preißel, René and Bjørn Stachmann. *Git. Dezentrale Versionsverwaltung im Team : Grundlagen und Workflows*. Heidelberg: dpunkt.Verlag, 2016, S. vii.

⁸⁰ Vgl. Preißel und Stachmann (wie Anm. 77), S. vii.

⁸¹ Vgl. Preißel und Stachmann (wie Anm. 77), S. viii.

⁸² Vgl. Preißel und Stachmann (wie Anm. 77), S. viii.

⁸³ Vgl. Preißel und Stachmann (wie Anm. 77), S. viii.

3.4 Wichtige Begriffe innerhalb von Versionsverwaltungssystemen

Nicht alle Begriffe, die unter diesem Punkt angesprochen werden, werden auch im weiteren Verlauf dieser Arbeit gebraucht. Jedoch schadet es nicht diese zu kennen um sie evtl. für die Unterstützung in anderen Projekten einzusetzen.

Wichtige Begriffe innerhalb von Versionsverwaltungssystemen sind:

Tabelle 1 Grundlegende Begriffe

Repository:	Das Repository ist eine Datenbank, in der bei Git, Änderungen als Commits gespeichert werden.
Working Tree:	Die Ebene auf der man arbeitet um Änderungen an Dateien vorzunehmen.
Commit:	Veränderungen an Dateien oder neue Dateien werden im Repository als Commit gespeichert. Commits enthalten auch Informationen zu den gemachten Veränderungen.
HEAD:	Referenz auf den letzten durchgeführten Commit und den entsprechenden Entwicklungsweig. Bildet den Kopf oder die Spitze eines Entwicklungsstrangs.
SHA-1:	Durch den <i>Secure Hash Algorithm</i> wird eine Prüfsumme erstellt, welche 160 Bit lang ist. Dies entspricht 40 hexadezimalzahlen. Diese Prüfsumme kann beliebigen digitalen Zahlen zugeordnet werden. In Git werden alle Commits den nach ihnen zugeordnetem Hash-Wert benannt. (Commit-ID). Dieser Wert errechnet sich aus dem Inhalt und den Metadaten des Commits und ist dementsprechend inhaltsabhängig.
Staging Area/ Index:	Zwischenebene zwischen dem Working Tree und dem Repository, welcher benutzt wird um einen Commit vorzubereiten. Der Index weist darauf hin, welche Änderungen an welcher Datei als Commit vorbereitet werden.
Clone:	Kopie eines Repositories, welcher alle Daten und auch Commits enthält, die darin abgelegt werden.

Branch:	Ein Branch ist ein Entwicklungszweig in Git. Entwicklungszweige werden verwendet um Releases vorzubereiten, Bugs zu fixen oder neue Features zu entwickeln. Verschiedene Branches können auch zusammengeführt werden. Dieser Vorgang wird als <i>merge</i> beschrieben, also das „Verschmelzen“. Die Branches und das Merging gehören zu den wichtigsten Merkmalen von Git.
Master:	Beim Arbeiten mit Git wird mindestens ein Entwicklungszweig benötigt. Daher wird beim Erstellen eines neuen Repositorys automatisch ein Branch angelegt, welcher den Namen <i>master</i> trägt. Da aus technischer Sicht kein Unterschied zwischen den Branches besteht, kann dieser Zweig beliebig umbenannt oder auch gelöscht werden. Jedoch muss immer mindestens ein Entwicklungszweig bei der Arbeit mit Git vorhanden sein.
Tag:	Kennzeichnungen für besonders wichtige Commits. Mögliche Kennzeichnungen sind Metadaten.

Quelle: Eigene Darstellung in Anlehnung an Haenel und Plenz, S. 19ff.

3.5 Branching und Merging

Eines der wichtigsten Bestandteile von Git sowie anderen Versionsverwaltungssystemen ist das Branching. Im VCS-Umfeld ist ein Branch nichts anderes als ein Entwicklungszweig. Diese ermöglichen eine Abzweigung vom gegenwärtigen Stand der Entwicklung. Im Prinzip bedeutet das, dass der aktuelle Entwicklungsstand kopiert und auf einen neuen Zweig übertragen wird, auf dem dann weiter entwickelt werden kann ohne den Fortschritt eines anderen Zweiges zu verhindern.⁸⁴

Die Ursprüngliche Idee zu Git kam von den Entwicklern des Linux-Kernels, welche über die ganze Welt verteilt arbeiten. Das Zusammenführen von einzelnen Beiträgen stellt eines der größten Herausforderungen für sie dar.

⁸⁴ Vgl. Vijayakumaran (wie Anm. 2), S. 59.

Während das Verzweigen und Zusammenführen bei anderen Versionsverwaltungssystemen einer Ausnahmesituation entspricht und als Arbeit für Fortgeschrittene angesehen wird, ist Git so konzipiert, dass es das Branching und Merging so einfach wie nur möglich gestaltet. Die Versionen von Projekten werden auf den Branches als Punkte dargestellt und entstehen mit jedem einzelnen Commit. Git versioniert immer das gesamte Projekt. Somit zeigt ein Punkt immer die zusammengehörenden Versionen von einem Projekt an. Bei der Änderung einer Version durch zwei verschiedene Entwickler entstehen dadurch zwei verschieden neue Versionen eines Projektes. Diese befinden sich jeweils auf dem eigenen Repository und dem dazugehörigen Branch der Entwickler. Wenn nun die Entwickler die Änderungen des anderen in ihre eigenen Repositories aufnehmen, werden diese durch Git zusammengeführt und beide Entwickler erhalten die Änderungen des jeweils anderen. Dieser Vorgang wird als Merge-Commit bezeichnet.⁸⁵

Branches sind wie parallele Ablaflinien der Entwicklung. Diese Linien kann man in mehrere Bahnen aufteilen, wo die weiteren Entwicklungen ablaufen. Die Aufteilung in diese Bahnen ist jedoch reine Interpretationssache und dient einem besseren Verständnis.⁸⁶

Der Hauptentwicklungszweig bei Git wird Master genannt und entspricht in anderen Versionskontrollsystemen dem Trunk. Ein Beispiel hierfür ist Subversion. Der aktuellste Commit eines Branches wird mit der Bezeichnung HEAD versehen.⁸⁷ Das kleingeschriebene head hingegen existiert mehr als einmal zur selben Zeit und stellt eine Referenz zu einem anderen Commit dar, welcher nicht derzeitig ausgecheckt worden ist.⁸⁸

Darüber hinaus ermöglicht Git mitzuverfolgen, wenn Änderungen an Zweigen vorgenommen wurden, die zu anderen Repositories gehören. Dies wird durch Tracking-Branches ermöglicht. Diese fungieren als eine Art lokaler

⁸⁵ Vgl. Preißel, René and Bjørn Stachmann. *Git. Dezentrale Versionsverwaltung im Team : Grundlagen und Workflows*. Heidelberg: dpunkt.Verlag, 2016, S. 5.

⁸⁶ Vgl. Preißel und Stachmann (wie Anm. 77), S. 59.

⁸⁷ Vgl. Sven Riedel, *Git. Kurz & gut* (O'Reillys Taschenbibliothek; Beijing: O'Reilly, 2010), S. 9.

⁸⁸ Vgl. Vijayakumaran (wie Anm. 2), S. 60.

Cache für die durchgeführten Änderungen. Eine Bearbeitung dieser Tracking-Banches ist nicht möglich, da diese von den lokalen Entwicklungszweigen abgezweigt sind.⁸⁹

Entwicklungszweige und deren Zusammenführungen sind in Git wichtige Werkzeuge, welche alltäglich zum Einsatz kommen. Git bietet eine transparente Übersicht über die einzelnen Branches und gestaltet das Merging, also das Verschmelzen der einzelnen Branches, auf eine einfache, schnelle Art und Weise. Aufgrund dessen kommt es nicht selten vor, dass Entwickler öfter an einem Tag zahlreiche Branches erstellen, um diese letztendlich wieder zusammenzuführen.⁹⁰

Ob es nun darum geht etwas ausprobieren zu wollen, sich um einen Bugfix zu kümmern oder an einem neuen Feature herumzuexperimentieren, für all diese Dinge werden zugehörige Branches erstellt, zusammengeführt oder wieder gelöscht. All das ist gängige Praxis unter Entwicklern auch in realen Projekten.⁹¹

3.5.1 Die Hauptentwicklungszweige

Da Git eine dezentrale Versionsverwaltung ist, existiert technisch gesehen so etwas wie ein zentrales Repository nicht. Unter Git-Usern ist dieses "zentrale" Repository viel mehr als *origin*, also Quelle oder Herkunft bekannt. Wichtig im Zusammenhang mit Branches ist jedoch, dass dieses *origin* in der Regel zwei Hauptentwicklungszweige beinhaltet. Da wäre zunächst der Master-Zweig und als zweites der Develop-Zweig. Dies hängt stark von der Entwicklungsmethode ab, die man anwenden möchte. In der Regel existiert auf dem *origin* nur der Master. Der hier dargelegte Aufbau ist nur einer der Möglichkeiten, wie mit Git gearbeitet werden kann.

Der Master-Zweig im *origin* ist der Ort, an dem der Quellcode abgelegt wird, der schlussendlich auch veröffentlicht wird. Der Develop-Zweig hingegen

⁸⁹ Vgl. Riedel (wie Anm. 87), S. 9.

⁹⁰ Vgl. Haenel, Valentin und Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 61f.

⁹¹ Vgl. Haenel und Plenz (wie Anm. 76), S. 61f.

enthält den zuletzt abgelieferten und für ein Release bereitstehenden Quellcode zur Verfügung.

Wenn all der Fortschritt im Develop-Zweig an dem Punkt angelangt ist, so dass er veröffentlicht werden kann, sollte dieser Fortschritt mit dem Master-Branch zusammengelegt werden.⁹²

3.5.2 Die Unterstützungszweige

Neben diesen Hauptentwicklungszweigen gibt es auch noch eine Vielzahl an Unterstützungszweigen. Diese haben die Aufgabe das gleichzeitige Entwickeln und den Informationsaustausch zwischen den Entwicklern zu fördern. Darüber hinaus sollen sie dabei helfen neue Feature zu entwickeln, Produktionsreleases vorzubereiten und auftretende Fehler schnell zu beheben. Diese Zweige unterscheiden sich von den anderen insofern, dass sie nur eine begrenzte Lebenszeit aufweisen, da sie nach Gebrauch mit einer hohen Wahrscheinlichkeit wieder entfernt werden. Diese Unterstützungszweige werden wie folgt unterschieden⁹³:

- Feature-Branch (Funktionszweig/Anforderungen)
- Release-Branch (Veröffentlichung)
- Hotfix-Branch (schnelle Fehlerbehebung)

Jeder einzelne dieser Zweige hat ihm zugewiesene Aufgaben und alle unterstehen strengen Regeln. Es ist darauf zu achten, welcher Zweig von welchem hervorgeht und welcher Zweig mit welchem zusammengeführt werden darf.⁹⁴

⁹² Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

⁹³ Vgl. Driessen, „A successful Git branching model,“ (wie Anm. 91).

⁹⁴ Vgl. Driessen, „A successful Git branching model,“ (wie Anm. 91).

3.5.3 Der Feature-Branch

Feature-Banches dienen dazu neue Funktionen und Eigenschaften für das Projekt zu einem beliebigen Zeitpunkt zu entwickeln. Die Frist an dem diese Neuerungen in das Projekt integriert werden soll, ist am Anfang noch unbekannt. Ein Feature-Branch zeichnet sich dadurch aus, dass es so lange bestehen bleibt, so lange der zu entwickelnde und benötigte Teil in Arbeit ist. Danach wird der Zweig in den Develop-Zweig überführt oder im Falle eines Fehlschlages entfernt. Der Feature-Branch darf aus dem Develop-Zweig entstehen und darf nur mit ihm wieder zusammengeführt werden. Typischerweise existieren diese Feature-Banches nur in den Repositories der Entwickler und nicht auf dem origin.⁹⁵

Die Planung eines neuen Projektes oder Produktes sollte so gestaltet sein, dass Anforderungen paketweise bearbeitet und übergeben werden. Je länger die Bearbeitung an einem Feature andauert, desto höher ist die Wahrscheinlichkeit, dass bei der Übergabe des Features erhebliche Schwierigkeiten auftreten. Daher sollte die Vervollständigung in eine paar Stunden oder wenigen Tagen erfolgen. Um Fehler auf einem Minimum zu halten, müssen vorher auf dem lokalen Rechner Tests durchgeführt werden. Getestet wird, ob die Änderungen negative Auswirkungen das Zusammenspiel von Features haben und ob diese Änderungen ein weiteres Zusammenarbeiten von Anforderungen ermöglichen. Das Auslassen solcher Tests hätte zur Folge, dass ein fehlerhaftes Projekt in den Master-Branch gemerged wird und Fehler erst in diesem Branch entdeckt werden.⁹⁶

Es ist sinnvoll zunächst den lokalen Master-Branch auf den neuesten Stand zu bringen. Dadurch wird verhindert, dass es zu Konflikten beim Merging kommt.

\$ git checkout master

⁹⁵ Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

⁹⁶ Vgl. Preißel und Stachmann (wie Anm. 77), S. 162.

```
$ git pull --ff-only
```

Der letzte Befehl sorgt dafür, dass nur ein Fast-Forward-Merge erlaubt ist. Dies bedeutet, dass ein Merge abgebrochen wird, wenn lokale Änderungen vorliegen. Falls eine Fehlermeldung auftaucht, wurde höchstwahrscheinlich vorher auf dem Master-Branch gearbeitet. Diese Änderungen müssen zunächst auf einen Feature-Branch verschoben werden, damit weitergearbeitet werden kann.⁹⁷

Um einen Feature Branch aus dem Develop-Zweig zu erzeugen und mit ihm zu arbeiten, verwendet man folgenden Befehl:

```
$ git checkout -b <Name des neuen Branches> develop
```

Wenn die Arbeit im Feature-Branch vollbracht ist, muss er mit dem Develop-Zweig gemerged werden. Dabei muss man als erstes in den Develop-Zweig wechseln, also in den Zweig in den gemerged werden soll. Dies geschieht mit:

```
$ git checkout develop
```

Anschließend erstellt man ein Commit-Objekt und verhindert somit, dass keine Informationen innerhalb der Änderungshistorie verloren gehen und all diejenigen Commits, welche die neuen Änderungen beinhalten zusammen als Gruppe übertragen werden.

```
$ git merge --no-ff <Name des Feature-Banches>
```

Es ist schlichtweg unmöglich durch die Änderungshistorie zu sehen, welche Commits zusammen gehören und gemeinsam ein Feature beigetragen haben. Dazu müsste man sich die einzelnen Protokolldaten manuell ansehen. Falls im Vorfeld zum *mergen* der obige Befehl genutzt worden ist, ist es jedoch wiederum ein Leichtes eine Gruppe von Commits, welche zusammen

⁹⁷ Vgl. Preißel und Stachmann (wie Anm. 77), S. 164.

gehören und ein gemeinsames Feature implementiert haben, gemeinsam zu entfernen.

Danach kann man den Feature-Branch löschen.

```
$ git branch -d <Name des Feature-Branche>
```

Als letztes werden die Änderungen in den Develop-Zweig, welcher sich auf dem origin befindet, übertragen.

```
$ push origin develop98
```

Weil Feature-Branche oft eine kurze Lebensdauer haben, werden sie meist nur lokal angelegt. Jedoch kann es vorkommen, dass die Entwicklung eines neuen Features länger dauert, das Sichern von Zwischenergebnissen sehr wichtig ist oder dass mehrere Entwickler an einem Feature gleichzeitig entwickeln. In diesen Fällen kann der Branch auch im zentralen Repository angelegt werden. Dazu benutzt man folgenden Befehl:

```
$ git push --set-upstream origin <Name des Feature-Branch>
```

Durch diesen Befehl wird zeitgleich der lokale Feature-Branch mit dem zentralen Repository verknüpft.⁹⁹

3.5.4 Der Release-Branch

Wie der Name schon sagt dienen Release-Branche dafür, an den Teilen des Projektes zu arbeiten, welche kurz vor einer Veröffentlichung stehen. Sie erlauben es kleinere Fehler und notwendige Vorbereitungen zu meistern, welche immer wieder kurz vor einem Release entstehen. Durch das Erarbeiten dieser Dinge auf einem separaten Branch wird die Arbeit auf dem Deve-

⁹⁸ Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

⁹⁹ Vgl. Preißel und Stachmann (wie Anm. 77), S. 165.

lop-Zweig erleichtert damit dieser aus dem Feature-Branch neues empfangen und einen neuen Release durchführen kann.

Der Release-Branch geht, wie schon der Feature-Branch, aus dem Develop-Zweig hervor. Er sollte sich erst dann von dem Develop-Zweig entzweigen, wenn all die vorher geforderten Eigenschaften für die nächste Veröffentlichung bereitstehen. In dem Release-Branch werden dann die Feinheiten bearbeitet, welche dem nächsten Release im Wege stehen. Funktionen und Eigenschaften, die nicht auf dem Release-Branch bearbeitet werden, müssen auf eine nächste Abzweigung warten. Nach getaner Arbeit wird der Release-Branch wieder mit dem Develop-Zweig zusammengeführt. Dadurch kann der Develop-Zweig letztendlich in den Master gemerged und dadurch automatisch veröffentlicht werden.¹⁰⁰

Mit dem Erzeugen eines neuen Release-Branches wird diesem auch gleichzeitig eine neue Versionsnummer für das neue Release zugewiesen. Durch das Zuweisen einer Versionsnummer wird auch im Vorfeld bekannt um was für eine Art von Release es sich letztendlich handelt. Nehmen wir an unser Produkt befindet sich in der Version mit der Versionsnummer 1.1.5. Der Develop-Zweig steht für die Veröffentlichung einer neuen Version bereit. Beim Erzeugen eines neuen Release-Branches wird gleich zu Beginn eine neue Versionsnummer vergeben, z.B. 1.2. Dadurch wird klar ob es sich um einen größeren oder einen kleineren Release handelt.¹⁰¹

Zunächst wird ein neuer Branch aus dem Develop-Zweig erstellt und in diesen gewechselt.

```
$ git checkout -b <Name des Branches> develop
```

In diesem Fall ist es sinnvoll auch gleich die Versionsnummer in den Namen des Branches einzutragen. Beispielsweise könnte der Befehl dann so aussehen:

¹⁰⁰ Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

¹⁰¹ Vgl. Driessen, „A successful Git branching model,“ (wie Anm. 91).

```
$ git checkout -b release-1.2 develop
```

Nachdem der Zweig erstellt und in ihn gewechselt wurde, muss ihm die entsprechende Versionsnummer nun noch zugewiesen werden. Dies geschieht mit dem Befehl *bump*.

```
$ ./bump-version.sh 1.2
```

Dadurch werden einige Dateien verändert, um auf die neue Version zu verweisen. Die neue Versionsnummer wird dadurch auch comittet. Dieser Zweig existiert dann solange bis eine offizielle Veröffentlichung der Version stattfindet. Wie schon erwähnt dient dieser Zweig eher dem Bugfixing. Er ist nicht dafür gedacht neue und große Änderungen zu implementieren. Um schlussendlich veröffentlichen zu können sind einige Schritte notwendig. Als erstes muss man den Release-Branch mit dem Master zusammenführen. Dazu muss man erst in den Master wechseln.

```
$ git checkout master
```

Alles was auf dem Master landet wird auch veröffentlicht.

```
$ git merge --no-ff release-1.2
```

Als nächstes muss dieser Commit durch einen *tag* gekennzeichnet werden, damit später aus der Änderungshistorie ersichtlich wird wie dieser Commit zugeordnet werden kann.

```
$ git tag -a 1.2
```

Das Release wurde durchgeführt und ist für die Zukunft gekennzeichnet. Um die in dem Release-Branch gemachten Änderungen nicht zu verlieren, muss dieser wieder mit dem Develop-Zweig zusammengeführt werden, aus dem er hervorgegangen ist.


```
$ git checkout develop
```

```
$ git merge --no-ff release-1.2
```

Jetzt sind wir mit dem Release durch und können den Release-Branch löschen, da er nicht mehr gebraucht wird.

```
$ git branch -d release-1.2102
```

3.5.5 Der Hotfix-Branch

Hotfix-Branches ähneln sich in ihrem Sinn und Zweck den Release-Branches. Hotfix-Branches sind ebenfalls dazu gedacht, bevorstehende Releases zu unterstützen oder vorzubereiten. Der Unterschied zwischen diesen beiden Zweigen ist allerdings, dass der Hotfix-Branch aus einer Notsituation heraus entsteht. Bei einem fehlerhaften Zustand des Produkts, wird ein neuer Branch gebildet, um diese neu entdeckten Fehler so schnell wie möglich zu beheben. Dieser neue Branch entstammt nicht wie die anderen aus dem Develop-Zweig, sondern entsteht aus dem Master heraus. Dadurch ist es möglich auf dem Develop-Zweig ungehindert weiter zu arbeiten während andere Teammitglieder, die neu entdeckten Fehler rasch beheben können.¹⁰³

```
$ git checkout -b hotfix-<Versionsnummer> master
```

Die entsprechende Versionsnummer muss nun dem Branch zugewiesen werden.

```
$ ./bump-version.sh <Versionsnummer>
```

Nachdem die Fehler behoben wurden, werden die Änderungen comitted.

```
$ git commit -m
```

¹⁰² Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

¹⁰³ Vgl. Driessen, „A successful Git branching model,“ (wie Anm. 91).

Anschließend wird der Hotfix-Branch zurück in den Master gemerged aber auch in den Develop-Zweig. Dadurch wird sichergestellt, dass die Fehler im neuen Release ebenfalls behoben sind.¹⁰⁴

Falls während dieses gesamten Vorgangs ein Release-Branch existiert, müssen die Änderungen aus dem Hotfix in den Release-Branch gemerged werden. Da dieser wiederum mit dem Develop-Zweig zusammengeführt wird, werden die Verbesserungen schlussendlich im neuen Release landen. Dies ist jedoch situationsabhängig. Wenn die Verbesserungen aus dem Hotfix zeitnah benötigt werden, können sie auch direkt wieder in den Master gemerged werden.¹⁰⁵

Der Hotfix-Branch wird ab diesem Zeitpunkt nicht mehr benötigt und kann nun gelöscht werden.

```
$ git branch -d hotfix-<Versionsnummer>
```

3.5.6 Branches verwalten

Einen neuen Branch anzulegen ist unter Git kein Problem. Dazu muss lediglich der aktuell ausgecheckte Commit ausfindig gemacht und der entsprechende Hash-Wert in der .git-Datei abgelegt werden.¹⁰⁶

Einige wichtige Befehle zur Bedienung von Branches sind:

```
$ git checkout -b (Name des Branches)
```

Neuen Branch erstellen und in diesen Branch wechseln.

```
$ git checkout master
```

Zum Master-Branch wechseln.

¹⁰⁴ Vgl. Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

¹⁰⁵ Vgl. Driessen, „A successful Git branching model,“ (wie Anm. 91).

¹⁰⁶ Vgl. Haenel, Valentin and Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 65.

\$ git branch -d <branch>

Einen bestehenden oder erstellten Branch löschen. Es können mehrere Branches gleichzeitig ausgewählt werden, jedoch müssen diese komplett im HEAD integriert sein.

\$ git push origin (Name des Branches)

Lädt den Branch in das remote Repository hoch, um es für andere zugänglich zu machen

\$ git branch

Gibt eine Liste mit allen vorhandenen Entwicklungszweigen aus.

\$ git branch -v

Liste mit vorhandenen Branches und den dazugehörigen letzten Commits.

\$ git branch --merged

Zeigt an, welche Branches bereits mit dem aktuell benutzten Zweig zusammengeführt wurden.

\$ git branch --no-merged

Zeigt an, welche Branches mit dem aktuell benutzten Zweig noch nicht zusammengeführt wurden.

\$ git checkout <branch>

Wechseln zwischen Branches.

Was passiert denn nun genau bei einem Checkout? Jeder Branch nimmt Bezug zu einem Commit. Dieser Commit wiederum bezieht sich auf einen Tree. Ein Tree stellt das Abbild einer Verzeichnisstruktur dar. Durch den Befehl `git checkout <branch>` werden die Beziehungen von diesem Commit zu dem Zweig aufgelöst. Der Tree des Commits wird dabei kopiert und auf dem Index und dem Working Tree abgelegt. Er wurde also repliziert. Git weiß in welcher Version sich Dateien auf dem Index und auf dem Working Tree befinden. Daher müssen nur die Dateien ausgecheckt werden, die sich auf dem neuen und dem alten Branch unterscheiden.¹⁰⁷

Git sorgt dafür, dass Informationen nicht verloren gehen. Es ist wahrscheinlicher dass ein Checkout fehlschlägt, als dass Dateien überschrieben werden. Git würde einen Checkout nur fehlschlagen lassen, wenn:¹⁰⁸

- eine Datei den Working Tree überschreiben wollen würde, auf dem sich Änderungen befinden.
- Änderungen an einer Datei sich auf dem Index befinden und ein Wechsel des Zweiges diese Datei verändern würde.
- eine Datei überschrieben werden würde, welche nicht von Git verwaltet wird.

3.5.7 Fast Forward Merge

Eine Situation, der man bei Git oft begegnet, ist das Vorspulen eines Branches, ein so genannter Fast-Forward-Merge.¹⁰⁹ Der Fast-Forward-Merge tritt dann auf, wenn es mehrere Branches gibt, jedoch nur auf einem weiter gearbeitet wurde. Angenommen, es existiert ein Branch-a und ein Branch-b, wobei Branch-b von Branch-a abstammt. Auf Branch-b wurde weiter gearbeitet, auf Branch-a dagegen nicht. Möchte man nun Branch-b mit Branch-a zusammenführen, zählt Git lediglich den Branch-a hoch. Anders ausgedrückt, werden die Commits aus dem Branch-b übernommen, ohne dass diese in-

¹⁰⁷ Vgl. Haenel, Valentin und Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 67.

¹⁰⁸ Vgl. Haenel und Plenz (wie Anm. 76), S. 67f.

¹⁰⁹ Vgl. Haenel und Plenz (wie Anm. 76), S. 84.

tern verändert werden.¹¹⁰ Dadurch ist kein Merge-Commit entstanden, sondern ein Fast-Forward-Merge. Der Vorteil liegt darin, dass die Änderungshistorie übersichtlich bleibt. Ein Nachteil ist jedoch, dass durch die Änderungshistorie nicht mehr zu sehen ist, dass eine Zusammenführung stattgefunden hat.¹¹¹

3.6 Der HEAD

Der HEAD wird meist unbewusst benutzt und stellt einen Bezug zu dem aktuell ausgecheckten Branch dar. Technisch gesehen bedeutet das, dass der HEAD anzeigt wo man sich im Moment befindet und auf was sich die nächsten Operationen und Befehle beziehen. Durch den HEAD wird unter anderem bestimmt, welche Dateien auf dem Working Tree zu finden sind und welche Commits Vorgänger von darauf folgenden Commits werden. Durch den Befehl `git checkout <commit-id>` kann der HEAD auch direkt auf einen bestimmten Commit zeigen, allerdings besteht hier die Gefahr das Commits verloren gehen können. Die meisten Befehle, welche über die Kommandozeile eingegeben werden, nehmen HEAD als erstes Argument an, sofern keine anderen Argumente angegeben wurden. Beispielsweise entspricht der Befehl `git log` dem Befehl `git log HEAD`.¹¹²

3.7 Merge-Konflikte

Das Verschmelzen von mehreren Entwicklungszweigen funktioniert unter Git weitestgehend sehr gut. Bei kleineren Änderungen treten in der Regel keine Probleme auf. Finden jedoch große Veränderungen in den Branches statt, welche man *mergen* möchte, kann es zu den sogenannten Merge-Konflikten kommen. Diese Konflikte treten zumeist dann auf, wenn eine Code-Zeile verändert wurde, die beide Zweige gemeinsam haben. Auch tritt ein Konflikt auf, wenn eine gemeinsame Zeile in einem Zweig gelöscht wird, in dem an-

¹¹⁰ Vgl. Vijayakumaran (wie Anm. 2), S. 69.

¹¹¹ Vgl. Preißel und Stachmann (wie Anm. 77), S. 72.

¹¹² Vgl. Haenel, Valentin and Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 64.

deren Zweig aber noch vorhanden ist. Diese Konflikte zu lösen ist für Git nicht ohne weiteres möglich, da Git nicht weiß, welche Änderung für die Entwickler die Wichtigere ist. Zwar gibt es mehrere Merge-Strategien, welche das Verhalten von Git in solchen Situationen verbessern können, jedoch ist es unumgänglich, diese Probleme manuell zu lösen.¹¹³ Auch ist es möglich einen entsprechenden Algorithmus zu erzeugen, welcher eine Lösung bieten würde, der der Syntax der jeweiligen Programmiersprache entspricht. Ein Algorithmus ist jedoch nicht in der Lage die Bedeutung eines Codes zu erfassen und ist daher in den meisten Fällen nicht zu empfehlen.¹¹⁴

Wenn nach einem Merge Konflikte auftreten, die Git nicht selbstständig beseitigen kann, wird angezeigt in welchen Dateien es zu Problemen gekommen ist. Die entsprechenden Stellen innerhalb dieser Dateien werden mit Markierungen versehen, um auf die Stellen hinzuweisen, welche den Konflikt ausgelöst haben. Diese Markierungen müssen nach der Bearbeitung ebenfalls entfernt werden. Nachdem die notwendigen Änderungen vollzogen worden sind, ist es wichtig die entsprechenden Dateien mit dem Befehl `git add` zu bearbeiten. Nachträglich müssen die Commits mit dem Befehl `git commit` abgeschlossen werden. Ein Weiterarbeiten ohne die Behebung des Konflikts ist unter Git nicht möglich.¹¹⁵

3.8 Hosting von Git

Je nachdem ob man zusammen in einem Team oder alleine arbeitet, bieten sich im Umfeld von Git verschiedene Hosting-Möglichkeiten über Server an. Jede dieser Möglichkeiten hat ihre eigenen Vor- und Nachteile. Einige dieser Möglichkeiten lassen sich selbst hosten, wieder andere werden als Dienste auf fremden Servern zur Verfügung gestellt. Das Hosting für eine einzelne Person gestaltet sich vor allem bei dem Verzicht auf ein Web-Interface einfach. Sofern beispielsweise Git und SSH auf einem Client und einem Server

¹¹³ Vgl. Vijayakumaran (wie Anm. 2), S. 71f.

¹¹⁴ Vgl. Haenel, Valentin und Julius Plenz. *Git. Verteilte Versionsverwaltung für Code und Dokumente*. München: Open Source Press, 2011, S. 84.

¹¹⁵ Vgl. Riedel (wie Anm. 87), S. 113.

installiert ist, kann ein Benutzer über diesen Server mehrere Repositories hochladen.¹¹⁶

Die ganze Sache wird komplizierter, sobald mehrere Personen an einem oder mehreren Remote-Repositories arbeiten möchten. Wenn obendrein für die Entwickler verschiedene Zugriffsrechte geplant sind, erschwert das alles umso mehr. An diesem Punkt setzt GitHub an. Viele Funktionen von GitHub fördern die Zusammenarbeit in großen Projekten, welche mit Git durchgeführt werden, auch für das Projekt-Management. Im Bereich von Softwareentwicklungsprojekten bietet GitHub mit hoher Wahrscheinlichkeit die größte Plattform an. Dies gilt vor allem für Open-Source-Software, für die GitHub kostenlos und ohne Einschränkungen zur Verfügung steht. Projekte mit nicht öffentlichem Code sind dagegen kostenpflichtig. Als registrierter Nutzer ist es kein Problem, ein eigenes Repository zu hosten und sich bei bestehendem Interesse in andere Projekte einzubringen. Je nachdem ob man eine einzelne Privatperson oder ein Unternehmen ist, werden unterschiedliche Kosten fällig. Die meisten Unternehmen lehnen es ab ihre Quellcodes auf öffentlichen Servern abzulegen und entscheiden sich somit für die GitHub Enterprise Variante. Diese ist die teuerste der Varianten in GitHub und ermöglicht es GitHub auf unternehmensinternen Infrastrukturen als eine virtuelle Maschine laufen zu lassen. Im Gegensatz zu vielen Projekten, welche auf den Servern von GitHub aufbewahrt werden, steht der Quellcode von GitHub selbst nicht als Open Source zur Verfügung.¹¹⁷

Eine weitere Hosting-Möglichkeit für Git besteht in der Nutzung von GitLab. Dieses ist in seinen grundlegenden Funktionen GitHub sehr ähnlich. Auch hier lassen sich eigene Repositories verwalten, um das gemeinsame Arbeiten in größeren Teams zu erleichtern. Ein wichtiger Unterschied zu GitHub besteht allerdings darin, dass bei GitLab private Repositories, die nicht zugänglich für andere sein sollen, kostenlos hochgeladen werden können. Da der Quellcode von GitLab selbst zum größten Teil frei zugänglich ist, besteht zusätzlich die Möglichkeit die Verwaltung von GitLab selbst in die Hand zu

¹¹⁶ Vgl. Vijayakumaran (wie Anm. 2), S. 115.

¹¹⁷ Vgl. Vijayakumaran (wie Anm. 2), S. 116.

nehmen. Dadurch können vor allem kleinere Unternehmen oder auch Gruppen die Kontrolle über ihre Quellcodes zum größten Teil selbst bestimmen, ohne dass sie gezwungen sind diese auf fremde Server hochzuladen. Durch die eigene Verwaltung allerdings entsteht ein Nachteil eines unzureichenden Speicherbedarfs. Eine kostenfreie Variante von GitLab ermöglicht für Unternehmen mit kleinen und unterdurchschnittlichen Servern keine zufriedenstellende Arbeitsweise.¹¹⁸

GitHub ist vor allem für Open-Source-Projekte geeignet, welche auf externe Hilfe angewiesen sind. Nicht zuletzt, weil eine große Anzahl an Leuten Erfahrung mit GitHub vorweisen kann und die größte Gemeinschaft im Bereich von Open Source hier anzutreffen ist. Darüber hinaus ist GitHub, was das Angebot an externen Tools angeht, anderen Plattformen weit voraus. GitLab hingegen bietet sich für kleinere Unternehmen und Gruppen an, die ihre Repositories auf ihren eigenen Servern kostengünstig verwalten möchten. GitHub und GitLab sind beide Web-Anwendungen, in deren Natur es liegt im ständigen Wandel zu sein. Bis auf grundlegende Funktionen, können weitere hier aufgeführte Eigenschaften sich schnell ändern und dafür sorgen, dass diese Tools im Laufe der Zeit anders aussehen.¹¹⁹

3.9 Zusammenfassung

Die Versionsverwaltungssoftware Git wurde von Linus Torvalds und seinem Team entwickelt, um die Fertigstellung des Linux-Kernels zu unterstützen. Git ist eine verteilte Versionsverwaltungssoftware, was bedeutet dass Änderungen an einem möglichen Quellcode durch einen Entwickler, nicht zwangsläufig jedes Mal auf einem zentralen Server abgelegt werden müssen. Diese Änderungen können zunächst auf dem lokalen Arbeitsbereich eines jeden Entwicklers aufbewahrt werden. Erst wenn gemachte Änderungen endgültig sind, wird eine Netzwerkverbindung zum Server benötigt, um sie dort abzuliegen. Eines der wichtigsten Merkmale, welches Git ausmacht ist das Branching und Merging. Zwar ermöglichen auch andere Versionskontrollsysteme

¹¹⁸ Vgl. Vijayakumaran (wie Anm. 2), S. 116.

¹¹⁹ Vgl. Vijayakumaran (wie Anm. 2), S. 116f.

das Arbeiten mit Entwicklungszweigen, doch vor allem das Zusammenführen dieser Zweige gestaltet sich im Gegensatz zu Git schwierig. Branches (deutsch: Zweig) stellen Entwicklungszweige dar, auf denen jeder Entwickler lokal auf seinem eigenen Rechner arbeiten kann. Sind diese Arbeiten erledigt, werden sie durch ein Commit (deutsch: Übergabe) auf dem entsprechenden Zweig gespeichert. In größeren Projekten ist es üblich, dass mehrere Teilteams an mehreren Bereichen eines Produktes arbeiten und für jeden dieser Bereiche Entwicklungszweige existieren. Auf diesen Entwicklungszweigen landen zunächst lokal die Änderungen der einzelnen Entwickler. Diese verschiedenen Änderungen auf unterschiedlichen Entwicklungszweigen können miteinander zusammengeführt bzw. verschmolzen werden. Dieser Vorgang wird als Merging bezeichnet. Das bedeutet im Groben, dass zwei Entwicklungszweige mitsamt ihrem Inhalt zu einem gemeinsamen Zweig werden. Dies ermöglicht beispielsweise die Arbeit von zwei verschiedenen Personen am selben Dokument zu einem gemeinsamen Dokument zusammenzufügen. Dabei werden jeweils die Änderungen des Anderen übernommen. Probleme könnten jedoch dann auftauchen, wenn beide Personen Änderungen in derselben Zeile durchgeführt haben. Git weiß nicht welche dieser Änderungen die wichtigere ist und dieses Problem muss per Hand gelöst werden.

Der Hauptentwicklungszweig, auf dem letztendlich alles zusammenkommen muss wird Master genannt. Jeder Commit der auf einem der Entwicklungszweige landet, wird auf dem Master abgelegt. Hierbei ist zu beachten, dass man zwischen dem Master auf dem lokalen Rechner eines jeden Entwicklers und dem Master-Branch auf dem „zentralen“ Repository unterscheiden muss. Da Git eine verteilte Versionsverwaltungssoftware ist, existiert so etwas wie ein „zentrales“ Repository nicht. Daher wird das Repository auf dem Server als *origin* oder als Remote Repository bezeichnet, also als Quelle oder externes Repository. Endgültige Änderungen werden auf diesem abgelegt oder daraus entnommen um das eigene lokale Repository auf den neusten Stand zu bringen.

Die Handhabung der Zweige und auch der Repositories kann stark zwischen Unternehmen oder Projekten variieren. Beispielsweise können mehrere

Repositories auf dem zentralen Server vorhanden sein, um zusätzlich den Austausch von Commits zwischen verschiedenen Entwicklern zu gewährleisten. Des Weiteren können Entwicklungszweige mit unterschiedlichen Aufgabenbereichen existieren, welche strengen Regeln unterstehen. Eine mögliche Vorgehensweise ist die Existenz von Unterstützungszweigen. Diese bestehen unter anderem aus einem Zweig für die weitere Entwicklung, einem Zweig für das beheben von Fehlern kurz vor einem Release und einem Zweig für das schnelle Beheben von Fehlern, welche kurz nach einem Release entdeckt wurden. Keiner dieser genannten Zweige darf zweckentfremdet werden, um darin z.B. Fehler auf dem Entwicklungszweig zu beheben.

Der gemeinsame Ausgangspunkt dieser Unterstützungszweige ist der Develop-Zweig. Dieser enthält den als letztes abgelieferten Quellcode kurz vor einem Release und befindet sich ebenfalls wie der Master-Branch auf dem Server. Wird kurz vor der Veröffentlichung des Quellcodes auf dem Develop-Zweig ein Problem festgestellt, werden dementsprechend die oben genannten Zweige aus dem Develop-Zweig heraus erstellt und nach getaner Arbeit wieder mit ihm zusammengeführt. Werden die endgültigen Commits auf dem Develop-Zweig auf den Master übertragen, sind diese bereit für ein Release. Treten allerdings nach dem Release weitere Probleme oder Fehler auf, wird in diesem Fall aus dem Master-Branch heraus ein Zweig zur Behebung der entsprechenden Commits erstellt. Sind die Fehler behoben, ist es ratsam die Commits direkt wieder auf den Master Branch zu übertragen, sonst würde man der weiteren Arbeit auf dem Develop-Zweig, auf dem inzwischen neue Commits vorhanden sind, im Weg stehen. Möglicherweise könnten dadurch sogar die Commits durcheinander kommen und dadurch würde ein Chaos verursacht werden. Der hier geschilderte Ablauf ist allerdings nur einer der Möglichkeiten, wie man mit Git arbeiten kann. Ein zweiter Branch auf dem externen Repository ist eher die Ausnahme.

4 Versionsverwaltungssysteme im Überblick

Werden an Dateien oder Dokumenten Änderungen vorgenommen, entstehen von diesen Dateien oder Dokumenten neue Versionen. Versionsverwaltungssysteme sind dazu da, um diese zu erfassen. Neu entstandene Versionen werden archiviert und mit einem Zeitstempel sowie einer Benutzer ID versehen, um bei Bedarf diese Versionen abzurufen oder wiederherzustellen. Dadurch kann nachvollzogen werden, welcher Benutzer zu welchem Zeitpunkt Änderungen an einer Datei getätigt hat. Ziel eines Versionsverwaltungssystems ist die Ermöglichung des Zugriffs mehrerer Personen gleichzeitig auf Dateien und das Arbeiten mit diesen Dateien auf verschiedenen Entwicklungszweigen. Deshalb sind Versionsverwaltungssysteme überwiegend in der Software-Entwicklung anzutreffen. Dabei haben sich zwei Systeme für die Versionsverwaltung besonders hervorgetan, diese sind zu einem Git und das von Apache entwickelte Subversion (SVN). Beide Systeme können entweder auf dem eigenen Server oder über einen Hoster benutzt werden. Hier bieten sich die Hosting-Tools GitHub für Git und RiouxSVN für Subversion an. Über den Dienst SourceForge, ist es möglich beide Systeme zu hosten.¹²⁰

4.1 Anforderungen an Versionsverwaltungssysteme

Jedes Versionsverwaltungssystem sollte die folgenden Anforderungen erfüllen¹²¹:

- Protokollierung: Nachvollziehbarkeit welche Änderung durch wen und wann vorgenommen wurde

¹²⁰ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich. Von verteilter und zentralisierter Versionsverwaltung,“. Geprüft am 3. Mai 2017. Online: <https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>.

¹²¹ Vgl. Herkommer, Günter, „Die Trends beim Versionsmanagement.“. Geprüft am 12. April 2017. Online: <http://www.computer-automation.de/steuerungsebene/steuern-regeln/artikel/79313/2/>.

- Wiederherstellung: Frühere Versionen von Dateien sollten immer wiederhergestellt werden können
- Archivierung: Einzelne Stände eines Projekts müssen so abgespeichert werden, dass sie nachvollziehbar sind.
- Koordinierung: Steuerung des Zugriffs auf dieselbe Datei von mehreren Entwicklern
- Gleichzeitiges Entwickeln: Unabhängiges Entwickeln auf verschiedenen Branches

4.2 Git

Im Frühling des Jahres 2005 entschied sich Linus Torvalds und sein Entwicklerteam, an einem neuen Versionsverwaltungssystem zu arbeiten, da die Lizenz der vorher genutzten Software namens BitKeeper nicht mehr zur Verfügung stand. Das neue System sollte genau wie das Alte für eine hohe Sicherheit und Effizienz sorgen. Innerhalb weniger Tage wurde eine erste Version namens Git entwickelt.¹²²

Der Kerngedanke von Git, liegt in der Arbeitsweise der verteilten Versionskontrolle. Es existiert zwar ein zentrales Repository, jedoch wird nur eine Netzwerkverbindung zu diesem gebraucht, um sich eine eigene Arbeitskopie lokal herunterzuladen oder eigene Änderungen darin abzulegen. Durch diese Architektur ist es ebenfalls nicht notwendig nach einem Lock-System zu arbeiten und zusätzliche Lese- und Schreibzugriffe entfallen. Jeder Nutzer arbeitet auf seinen eigenen Branches und wird nicht durch gesperrte Dokumente ausgebremst. Vorteilhaft ist, dass jede Arbeitskopie gleichzeitig als Back-up fungiert, um im Falle eines Datenverlustes gewappnet zu sein. Darüber hinaus vermerkt Git lediglich die Inhalte von Verzeichnissen und löscht leere automatisch.¹²³

¹²² Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich. Von verteilter und zentralisierter Versionsverwaltung.“. Geprüft am 3. Mai 2017. Online: <https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>.

¹²³ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich,“ (wie Anm. 119).

4.3 Apache Subversion

Apache Subversion ist eine freie Software, welche Anfang des Jahres 2000 von CollabNet entwickelt und knapp vier Jahre später veröffentlicht wurde. Dadurch wurde das Concurrent Versions System abgelöst, welches als Vorbild für Subversion fungierte aber nicht mehr in Stand gehalten worden war. Im Jahre 2009 nahm die Apache Software Foundation die Software in ihren Besitz, wodurch auch der heutige Name entstand. Subversion basiert auf der zentralen Versionsverwaltung. Dadurch ist ein zentraler Repository auf einem Server verfügbar. Dieses Repository steht für jeden Nutzer in dem jeweiligen Projekt zur Verfügung. Allerdings ist dafür eine Netzwerkverbindung zum Server erforderlich. Ohne diese Verbindung würde die getane Arbeit nicht auf dem Repository abgelegt werden und der erreichte Fortschritt würde verloren gehen.¹²⁴

Subversion bietet den Download und das Bearbeiten von beliebigen Unterpunkten an, ohne hierbei die Abhängigkeiten der einzelnen Pfade zu beachten. Hierdurch lassen sich auf verschiedene Pfade, unterschiedliche Lese- und Schreibzugriffe für die Nutzer verteilen. Ein weiteres Merkmal von Subversion ist das problemlose Aufzeichnen von leeren oder verschobenen Pfaden ohne den Verlust der zugehörigen Änderungshistorie.¹²⁵

4.4 Gegenüberstellung der Systeme

Git und SVN sind auf die verschiedenen Bedürfnisse der Nutzer zugeschnitten. Welches Versionsverwaltungssystem zum Einsatz kommt, hängt von den benötigten Arbeitsprozessen ab.¹²⁶ Die folgende Tabelle stellt die verschiedenen Gegensätze beider System dar:

¹²⁴ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich. Von verteilter und zentralisierter Versionsverwaltung.“. Geprüft am 3. Mai 2017. Online:

<https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>.

¹²⁵ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich,“ (wie Anm. 119).

¹²⁶ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich,“ (wie Anm. 119).

Tabelle 2 Git vs. SVN

	SVN	Git
Versionsverwaltungsart	zentral	dezentral
Netzwerkverbindung	zu jeder Zeit benötigt	nur bei Synchronisation
Änderungshistorie	nur auf dem Repository komplett	komplett auf Repository und Arbeitskopie
Repository	zentral auf Server	lokale Repository-Kopie
Änderungsaufzeichnung	vermerkt Daten	vermerkt Inhalte
Berechtigungen	für einzelne Pfade	für komplettes Verzeichnis

Quelle: Eigene Darstellung in Anlehnung an

<https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>, Geprüft am 3. Mai 2017.

4.5 Git oder SVN?

Das bessere oder schlechtere System gibt es nicht. Bei der Wahl zwischen Git und SVN spielen die eigenen Bedürfnisse sowie die gewünschte Vorgehensweise eine Rolle. Um sich zwischen diesen beiden Systemen entscheiden zu können, ist es empfehlenswert, die jeweiligen Vorteile zu kennen. Um eine Entscheidungsfindung zu erleichtern, werden die Vorteile an dieser Stelle nochmals zusammengefasst:¹²⁷

Git ist empfehlenswert, wenn:

- die Möglichkeit bestehen soll zu jeder Zeit und an jedem Ort zu arbeiten, ohne auf eine Netzwerkverbindung angewiesen zu sein
- die Daten auf dem Server vor Verlust oder Ausfall abgesichert sein sollen

¹²⁷ Vgl. o.V., „Git vs. SVN – Versionsverwaltung im Vergleich. Von verteilter und zentralisierter Versionsverwaltung.“. Geprüft am 3. Mai 2017. Online: <https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>.

- keine speziellen Lese- und Schreibberechtigungen notwendig sind
- Schnelligkeit ein entscheidendes Kriterium ist

Subversion wiederum ist zu empfehlen, wenn:

- kontrollierte Zugriffe auf bestimmte Bereiche eines Projekts benötigt werden
- ein zentraler Sammelpunkt für ein Projekt vorhanden sein soll
- zahlreiche Binär-Daten zum Einsatz kommen
- Strukturen leerer Verzeichnisse ebenfalls aufgezeichnet werden sollen

4.6 Zusammenfassung

Neue Versionen entstehen durch jede neue Veränderung an einer Datei oder einem Dokument. Versionsverwaltungssysteme sind dazu da, um die Handhabung von diesen veränderten Dateien oder Dokumenten zu gewährleisten. Beispielsweise erhalten neue Versionen durch Versionskontrolle eine Benutzer ID sowie einen zugehörigen Zeitstempel, um bestimmte Änderungen leichter auffinden zu können.

Versionsverwaltungssysteme zielen darauf ab, mehreren Personen gleichzeitig die unabhängige Arbeit auf verschiedenen Entwicklungszweigen zu gewährleisten, weshalb diese vor allem im Bereich der Software-Entwicklung anzutreffen sind.

Zwei der verbreitetsten und bekanntesten Systeme sind Git und Apache Subversion. Git wurde aus einer Not heraus von den Linux-Entwicklern angefertigt, während das Konkurrenzprodukt Subversion, der Apache Software Foundation gehört. Für beide Systeme befinden sich diverse Hosting-Tools wie GitHub oder RiouxSVN im Umlauf. Alternativ können beide Systeme auf eigenen Servern benutzt werden.

Der Unterschied zwischen diesen beiden Systemen liegt in ihren unterschiedlichen Funktionalitäten. Während Subversion auf eine zentrale Architektur aufbaut, die ein gesamtes Projekt nur über eine Netzwerkverbindung

erreichbar macht, besteht bei Git die Möglichkeit Projekte aus zentralen Servern komplett mit ihrer Historie zu kopieren und mit dieser Kopie lokal zu arbeiten. Darüber hinaus wird bei Git grundsätzlich ohne Lese- und Schreibzugriffe gearbeitet. Subversion hingegen setzt auf eine pfadbasierte Zugangsberechtigung. Hinzu kommt, dass durch die dezentrale Architektur von Git, Datenverlust so gut wie ausgeschlossen ist, da jede lokale Kopie gleichzeitig als ein Back-up fungiert. Jedoch verwirft Git inhaltslose Verzeichnisse, während diese bei Subversion weiter aufbewahrt werden.

Die Entscheidung welches System zum Einsatz kommen soll hängt stark von den eigenen Interessen und der gewünschten Arbeitsweise ab.

5 eGit-Plugin in Eclipse

Für eine erfolgreiche Integration der Versionsverwaltungssoftware Git in die Java-Programmier-Vorlesung, wird das Plugin eGit für Eclipse verwendet. Der Einsatz von eGit erleichtert das Zusammenspiel von Git und Eclipse indem es für eine Verbindung zum lokalen und zum remote Repository sorgt, wodurch eine nahtlose Übertragung des Java-Quellcodes zu diesen Repositories ermöglicht wird.

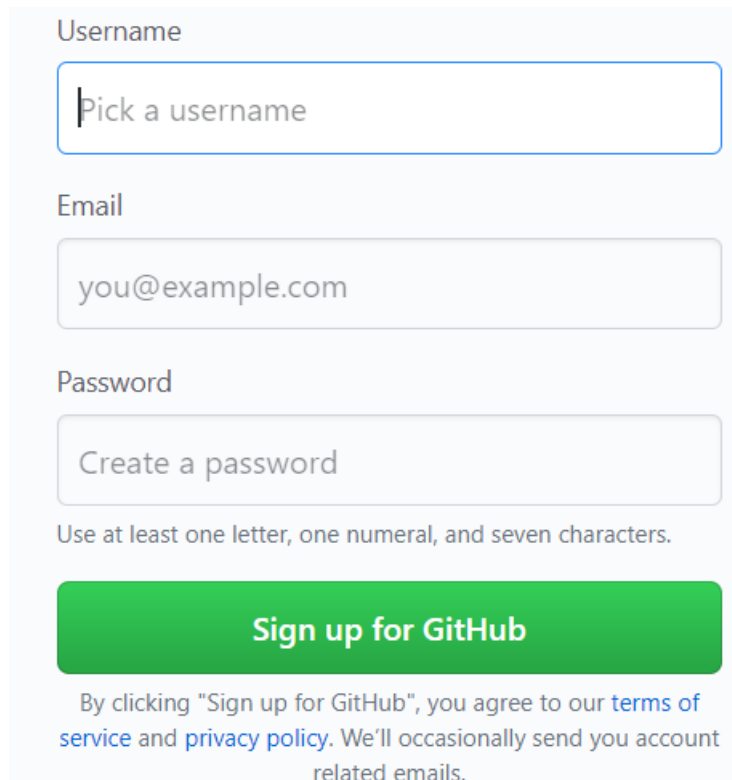
Als erste Voraussetzung muss die Software Eclipse auf dem Rechner vorinstalliert werden. Die zweite Voraussetzung ist die Registrierung bei GitHub, zu der eine simple Anleitung folgt. Die Anleitung zu Eclipse entfällt, da davon ausgegangen wird, dass die Software schon in der Java-Programmier-Vorlesung erfolgreich installiert wurde.

5.1 Registrierung bei GitHub

Rufen Sie zunächst die Seite <https://github.com/> auf. Auf der rechten Seite befindet sich der Registrierungsbereich. Nachdem Sie einen passenden Benutzernamen gefunden haben, der noch nicht vergeben ist, benötigen Sie eine dazugehörige E-Mail-Adresse. Nach dem Auswählen eines Passworts klickt man auf die Schaltfläche „Sign up for GitHub“ und es kann losgehen.

Anschließend wird an die angegebene E-Mail-Adresse eine Nachricht gesendet, in der der Verifizierungsvorgang abgeschlossen wird.

In unserem Fall können wir im zweiten Schritt einfach auf weiter klicken, da wir unsere Repositories öffentlich halten. Private Repositories wären kostenpflichtig.



The image shows the GitHub registration form. It consists of three input fields: 'Username' with a placeholder 'Pick a username', 'Email' with a placeholder 'you@example.com', and 'Password' with a placeholder 'Create a password'. Below the password field is a note: 'Use at least one letter, one numeral, and seven characters.' At the bottom is a green button labeled 'Sign up for GitHub'. Below the button is a disclaimer: 'By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We'll occasionally send you account related emails.'

Abbildung 3 Registrierung auf GitHub.com
Quelle: www.GitHub.com, Geprüft am 21.08.2017.

Im dritten Schritt kann man optional den eigenen Status der Erfahrung wiedergeben. Anschließend ist das Registrieren erfolgreich beendet.

Nun befindet man sich auf der Hauptseite des gerade erstellten Repositorys. In diesem befindet sich zurzeit nur ein einziger Branch und zwar der Master, auf dem die README-Datei vorhanden ist. Der Master wird mit einem *Initial Commit* (deutsch: einleitende Übergabe) eingeleitet, was bedeutet, dass dieser Commit keine Vorgänger besitzt. Ebenso erhält man Informationen über den Namen des Repositorys, welche Commits zur welchen Zeit getätigt wurden, wie viele Branches aktuell auf diesem Repository vorhanden sind und die Möglichkeit zwischen diesen Branches zu wechseln.

Für den Fall, das man ein Repository löschen möchte, begibt man sich auf dem entsprechenden Repository auf den Reiter „Settings“ und scrollt ganz

runter. Nach dem klicken auf das Feld „Delete this repository“ folgt man den Anweisungen und das Repository ist gelöscht.

Damit wären alle nötigen Schritte im Bereich GitHub abgeschlossen. Was jetzt noch fehlt, ist die Installation des Plugins in Eclipse.

5.2 eGit-Integration von Eclipse

Um das eGit Plugin in Eclipse zu installieren, muss man zunächst Eclipse starten. Über den Reiter „Help“ erreicht man „Install new Software“. Hierüber lässt sich die Software downloaden. Dazu muss in dem Feld des neu erschienenen Fensters „Work with:“ folgende URL eingegeben werden: "http://download.eclipse.org/egit/updates". Aus den nun neu erschienenen Checkboxen wird „Git Integration for Eclipse“ gewählt. Nach der Installation und dem erfolgten Neustart von Eclipse, befindet sich unter „Help“ und „Help Contents“ eine Dokumentation über eGit.

5.2.1 Lokales und Externes Repository aufbauen

Um ein lokales sowie externes Repository aufzubauen, sind folgende Schritte notwendig¹²⁸.

Erstellen Sie als Erstes einen Projektordner, in dem Sie arbeiten wollen und in dem sich die benötigten Java-Klassen befinden. Mit einem Rechtsklick auf den Projektordner kommen Sie über „Team“ auf den Befehl „Share Project“.

Wählen Sie den Pfad aus, an dem Ihr lokales Repository angelegt werden soll und geben Sie ihm einen Namen.

¹²⁸ Vgl. Brian Fraser, DrBFraser, „Creating a Repository: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/r5C6yXNaSGo?list=PL-susIzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 21.08.2017.

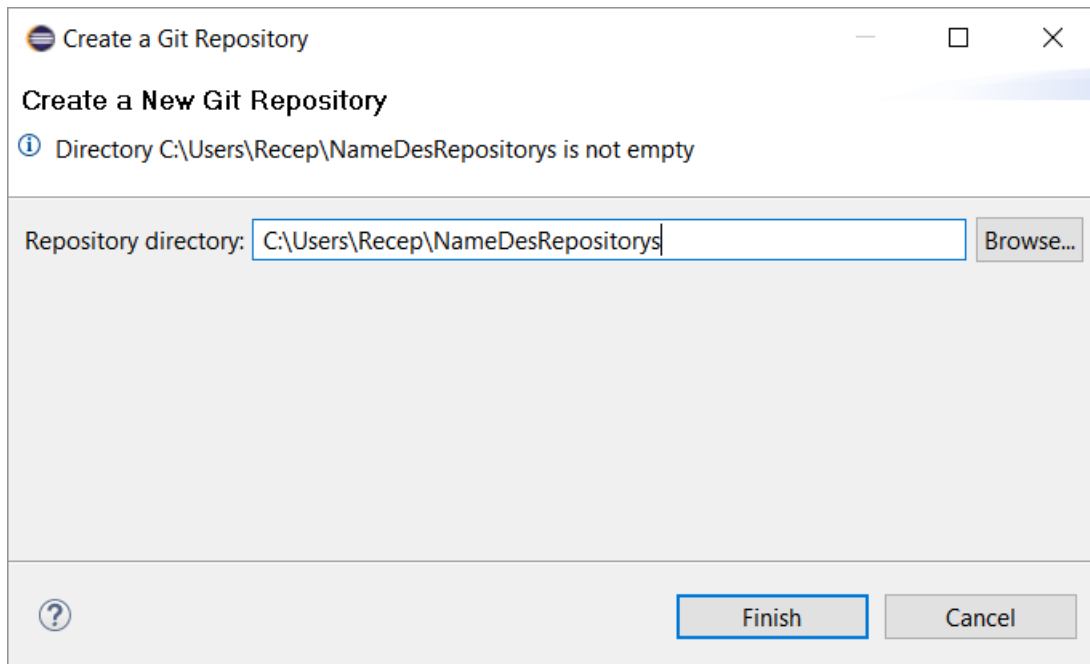


Abbildung 4 Erstellung lokales Repository
Quelle: Eclipse

Danach suchen Sie unter „Window“, „Show View“ und „Other“ den Ordner Git und wählen „Git Repositories“ und „Git Staging“ aus. Dadurch lassen sich nun die vorhandenen Repositories anzeigen.

Über dieselbe Abfolge wird unter dem Ordner „Team“ ebenfalls die Historie angezeigt. Zusätzlich kann man sich eine Git-Toolbar einblenden lassen. Dazu wählt man unter „Window“, „Customize Perspective“ aus und geht in dem neu erschienenen Fenster auf den Reiter „Command Groups Availability“. Abschließend setzt man auf der linken Seite einen Haken bei Git.

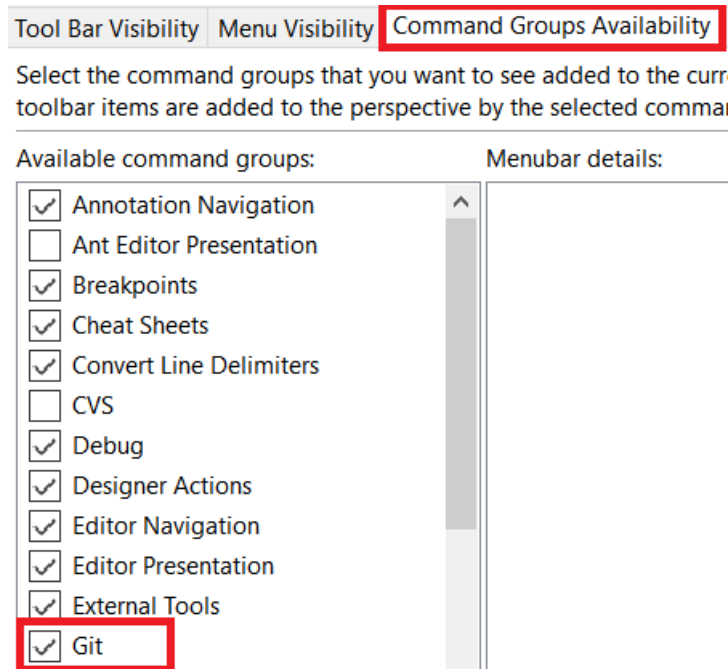


Abbildung 5 Git-Toolbar einblenden
Quelle: Eclipse

Wenn Sie Ihr Projekt *committen* möchten, dann gehen Sie mit einem Rechtsklick auf den Projektordner erneut auf „Team“ und danach ganz oben auf „Commit“. In dem neu erschienenen Fenster können Sie einen Kommentar für Ihren Commit eingeben und Ihre Änderungen übergeben. Wählen Sie zunächst nur „Commit“.

Falls der Commit fehlschlägt und die Fehlermeldung „There are no staged files“ auftaucht, dann klicken Sie mit der rechten Maustaste auf ihre einzelnen Java-Klassen, anschließend auf „Team“ und zum Schluss auf „add to Index“. Ein erneuter Versuch zu committen, sollte nun erfolgreich sein.

Als nächstes sollte der Commit auf dem Master-Branch zusammen mit einem Kommentar zu sehen sein. Für eine bessere Ansicht wird ein Rechtsklick auf das Repository ausgeführt. Anschließend wird „Show In“ und danach „History“ gewählt.

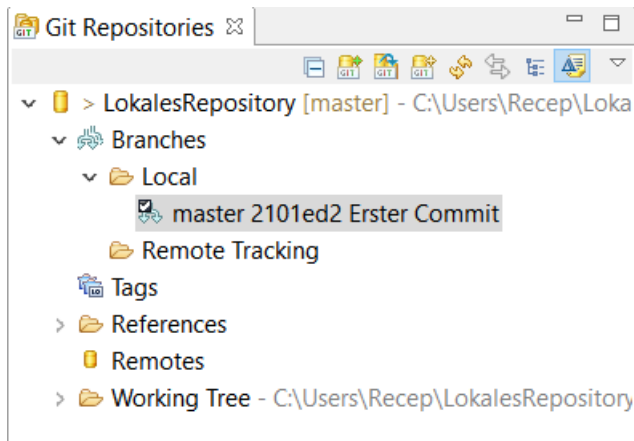


Abbildung 6 Lokaler Master-Branch in Eclipse
Quelle: Eclipse

Im nächsten Schritt wird ein Repository in GitHub erstellt, um eine Verbindung zu diesem über Eclipse zu gewährleisten. Dazu wird mit dem Browser die Startseite von GitHub aufgerufen und oben rechts „New repository“ ausgewählt.

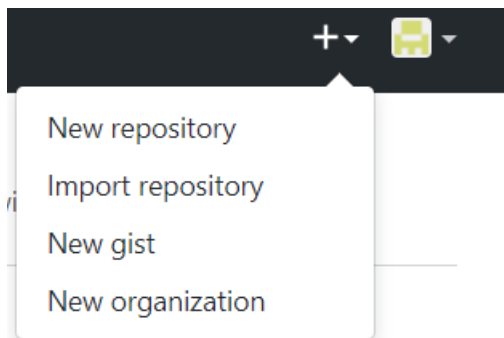


Abbildung 7 Neues Repository erstellen
Quelle: www.github.com, Geprüft am 21.08.2017.

Ist das Repository fertiggestellt, muss die angezeigte URL kopiert werden.



Abbildung 8 URL des neuen Repositorys
Quelle: <https://github.com/Ertugrul-Furtwangen/Repo-A>, Geprüft am 21.08.2017.

Die Benutzung einer URL eines schon existierenden Repositorys könnte Fehler verursachen. Daher wird geraten dieser Anleitung schrittweise zu folgen.

Zurück in Eclipse wird mit der rechten Maustaste „Remotes“ und danach „Create Remote“ ausgewählt.

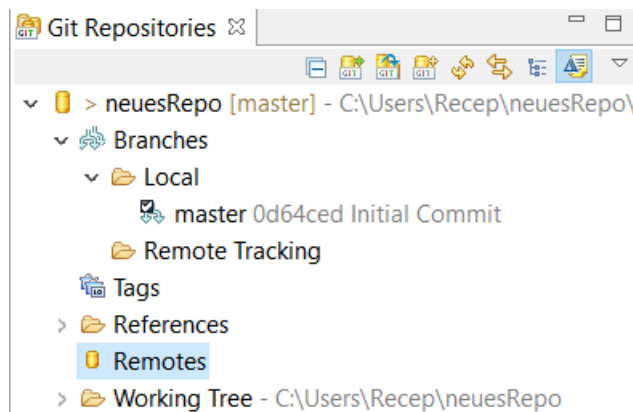


Abbildung 9 Remote erstellen

Quelle: Eclipse

Beim ersten Fenster wird „Ok“ und beim darauffolgenden „Change“ angeklickt, um die eben kopierte URL einfügen zu können. Bevor auf „Finish“ geklickt werden kann, muss sichergestellt werden, dass Benutzername und Passwort übereinstimmen. Daraufhin erscheint das vorherige Fenster. Mit „Advanced“ wird der Quell- und der Zielbranch bestimmt.

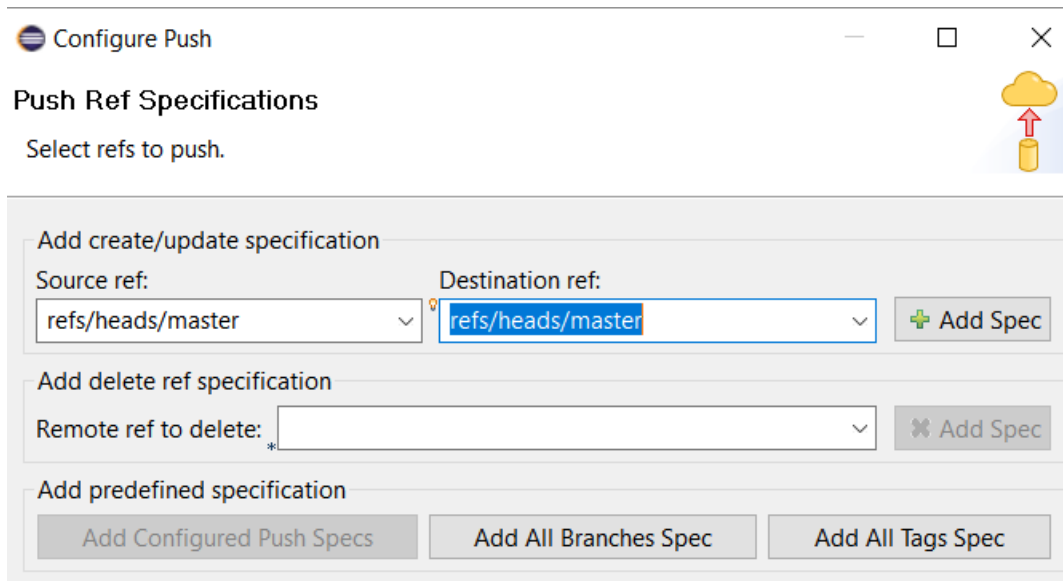


Abbildung 10 Quell- und Zielbranch
Quelle: Eclipse

Abschließend wird „Add Spec“ und weiter unten „Finish“ angewählt.

Unter „Remotes“ und „origin“ wird ein Rechtsklick auf den Pfad mit dem roten Pfeil ausgeführt.

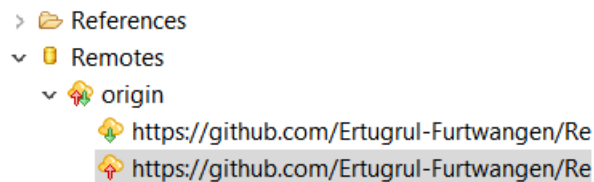


Abbildung 11 Pushen auf origin
Quelle: Eclipse

Mit „Push“ wird das Projekt auf das Repository in GitHub übertragen.

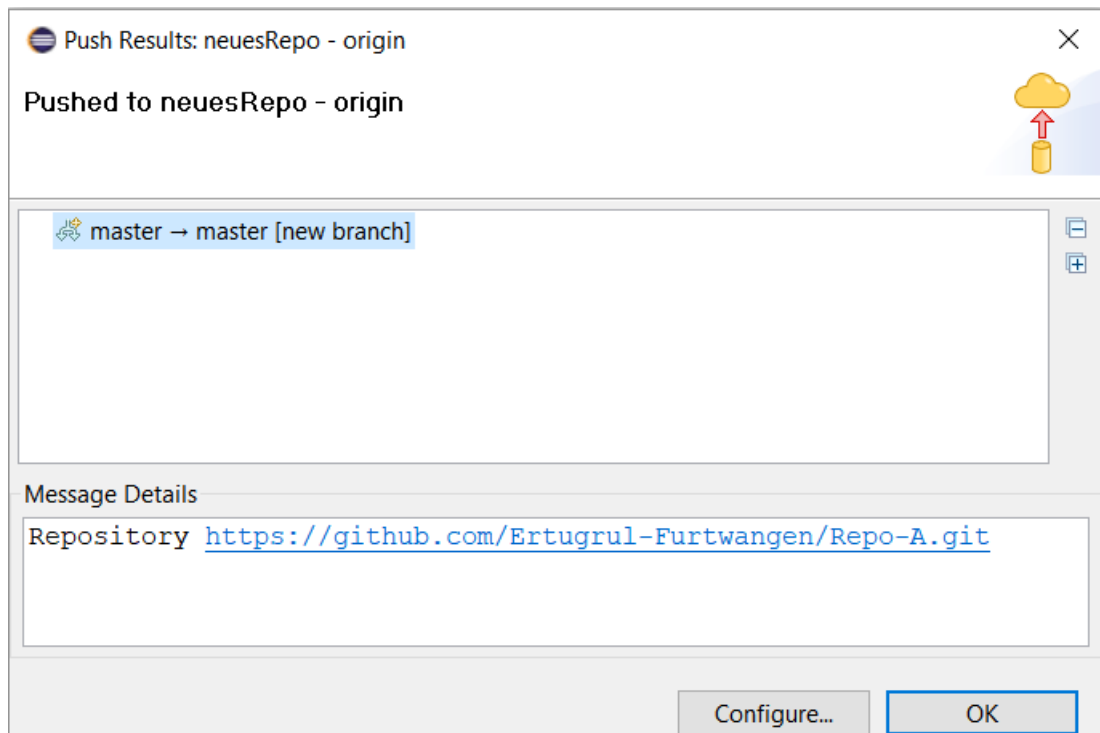


Abbildung 12 In Master Pushen
Quelle: Eclipse

Danach auf „OK“, um den Vorgang abzuschließen. Wenn alles geklappt hat, dann sollte auf dem Browser in GitHub nun das Projekt aus Eclipse mit dessen Inhalten zu sehen sein.

EGit	added no 7	3 days ago
README.md	Initial commit	3 days ago

Abbildung 13 Erfolgreicher Push
Quelle: <https://github.com/Ertugrul-Furtwangen/EGit>, Geprüft am 22.08.2017.

5.2.2 Vorhandene Projekte importieren

Zum Importieren eines bereits vorhandenen Projekts, müssen Sie wie folgt vorgehen¹²⁹.

Es ist möglich vorhandene Projekte in Eclipse zu importieren. Dazu wird in Eclipse auf dem „Package Explorer“ die rechte Maustaste betätigt und anschließend „Import“ gewählt. Anschließend wird der Ordner „Git“ und „Projects from Git“ ausgesucht.

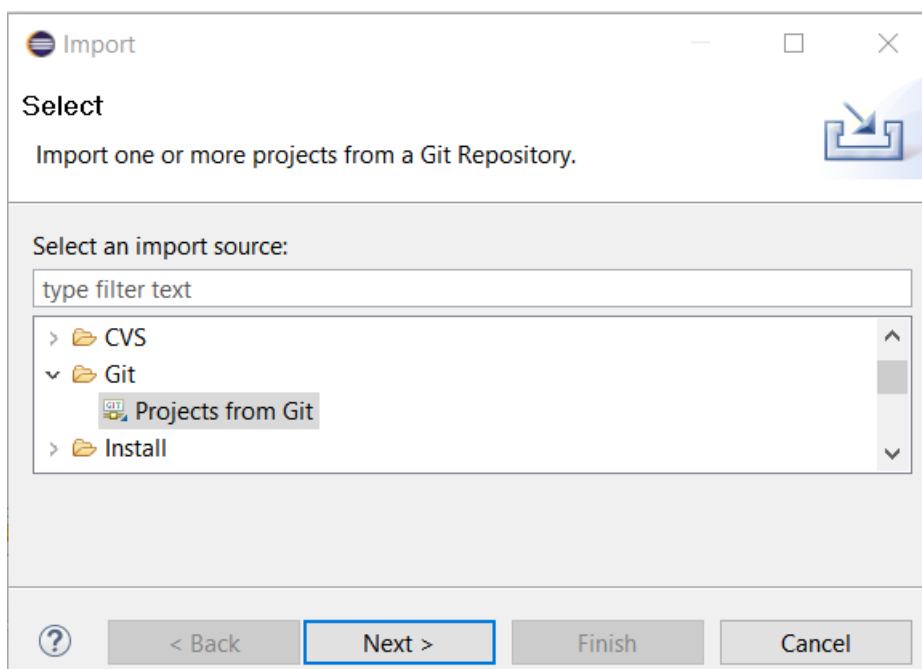


Abbildung 14 Git-Projekt importieren

Quelle: Eclipse

Es wird solange mit „Next“ bestätigt, bis die URL des zu importierenden Projekts, eingegeben werden kann.

¹²⁹ Vgl. Brian Fraser, DrBFRaser, „Checking out an existing project: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/V42r5REJx-M?list=PL-suslzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 22.08.2017.

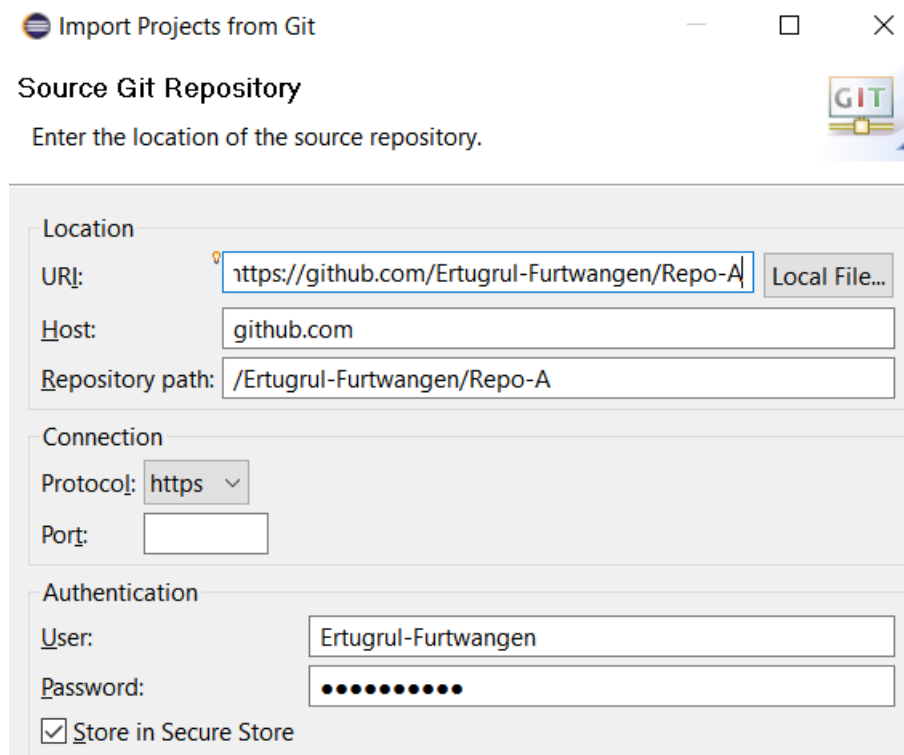


Abbildung 15 Quell-Repository
Quelle: Eclipse

Falls beim Import keine Projekte gefunden werden, wird die Option „Import as general project“ ausgesucht.

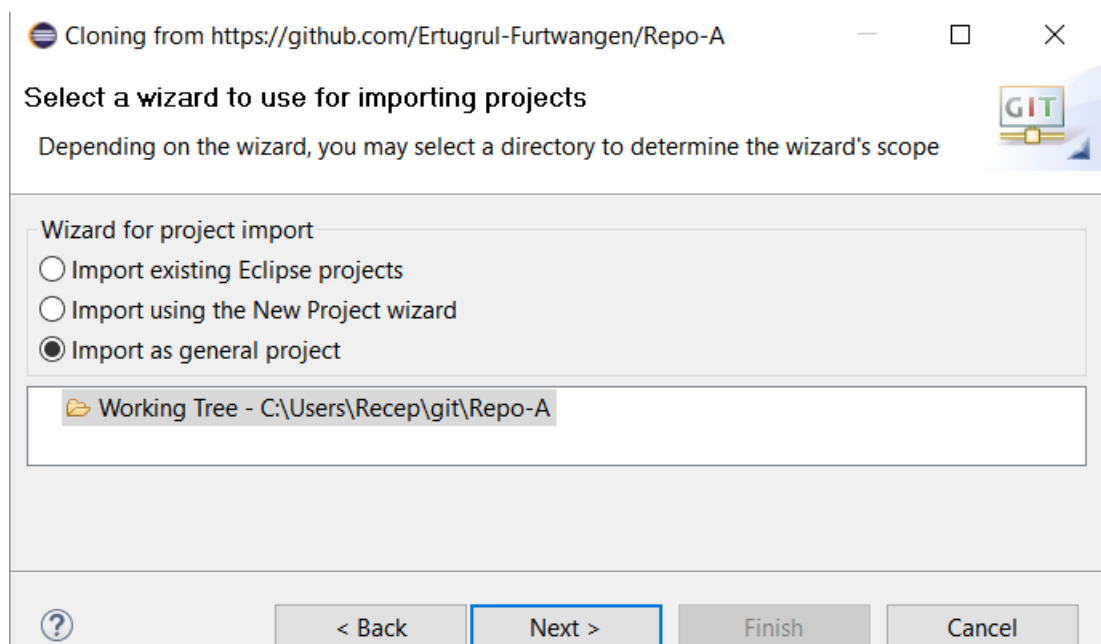


Abbildung 16 Allgemeines Projekt
Quelle: Eclipse

5.3 Commit-Verlauf einsehen

Um Unterschiede zwischen den Commits sehen zu können, müssen in den vorhandenen Java-Klassen zunächst Änderungen vorgenommen werden. Anschließend werden diese Comittet. Für diesen Befehl kann die Toolbar verwendet werden.

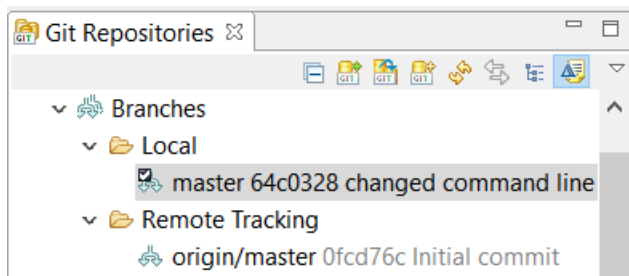


Abbildung 17 Nach Commit
Quelle: Eclipse

Führen Sie einen Rechtsklick auf den Master-Branch aus und wählen Sie „Show In“ und danach „History“.

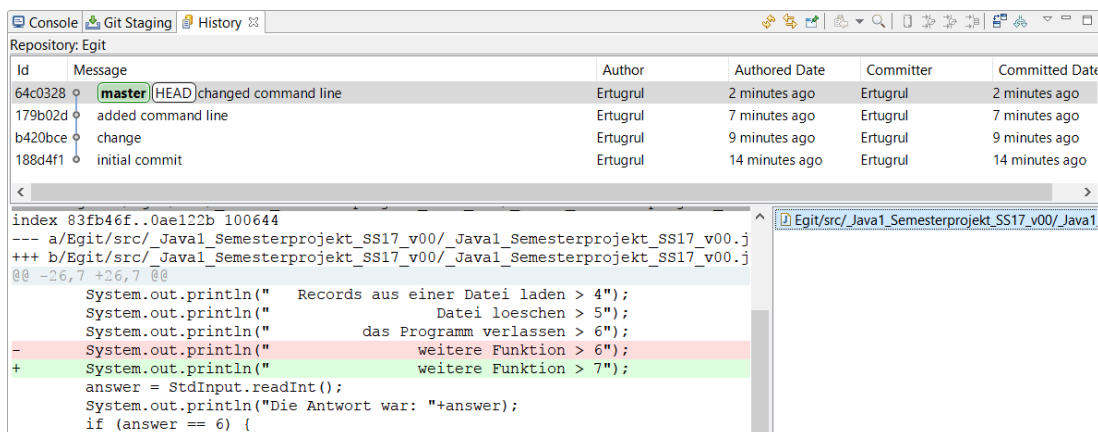


Abbildung 18 Commit-Verlauf und Versionsunterschiede
Quelle: Eclipse

Wählen Sie den Commit und auf der rechten Seite die Datei aus, damit die Änderungen und der Commit-Verlauf unter Eclipse angezeigt werden. Rot markierte Zeilen sind diejenigen, die entfernt wurden, grün markierte Zeilen wurden hinzugefügt.

5.4 Branching mit eGit

Das Branching und Merging unter eGit funktioniert folgendermaßen¹³⁰

Zum Arbeiten mit einem zusätzlichen Branch, wird ein Rechtsklick auf den aktuellen lokalen Branch ausgeführt und der Befehl „Create Branch“ ausgewählt. Anschließend wird beschlossen aus welchem Branch der neue Zweig entstehen soll und ein aussagekräftiger Name für diesen Zweig gewählt. Eclipse wechselt automatisch zu dem neu erstellten Zweig.

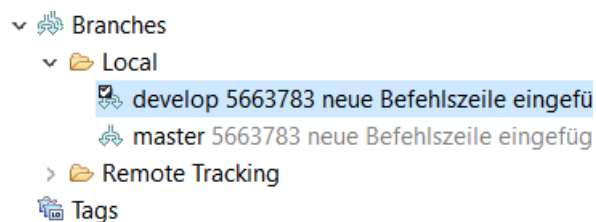


Abbildung 19 Neuen Branch erstellen

Quelle: Eclipse

Da der neue Branch in diesem Fall aus dem master-Branch entstanden ist, hat er sämtliche auf ihm befindliche Daten mitgenommen, einschließlich deselben Hash-Wertes und des Kommentars.

Nach der Arbeit und dem Commit, kann man sehen, dass der Hash-Wert und der Kommentar sich auf dem neuen Branch ändern.

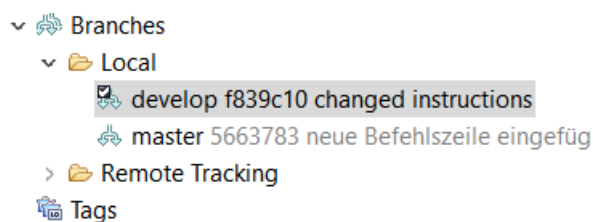


Abbildung 20 Commit auf neuem Branch

Quelle: Eclipse

¹³⁰ Vgl. Brian Fraser, DrBFRaser, „Making Changes: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/rbIGZRWqFVI?list=PL-suslzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 23.08.2017.

Als nächstes werden diese Änderungen zurück in den master-Branch übertragen. Dazu wird in den Branch, in den die Neuerungen übertragen werden sollen, gewechselt. Es folgt ein weiterer Rechtsklick auf den Branch, aus dem die Änderungen übertragen werden sollen und die Wahl des Befehls „Merge“. Dadurch werden beide Zweige miteinander verschmolzen.

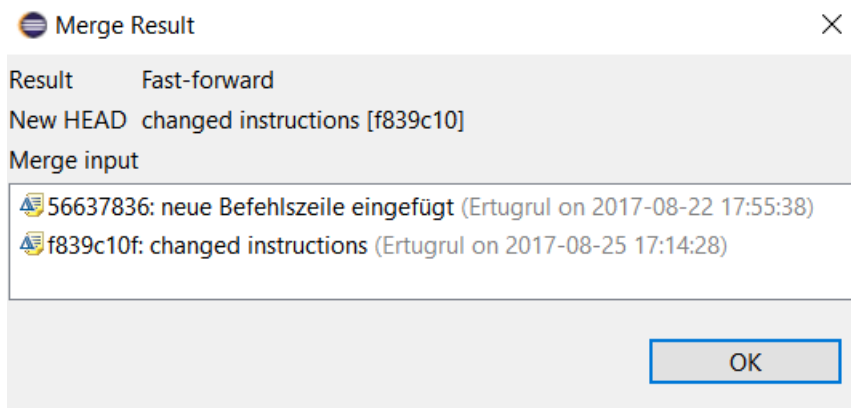


Abbildung 21 Merge-Details
Quelle: Eclipse

Nach dem erfolgreichen Zusammenfügen, sind die Kommentare und der Hash-Wert ein weiteres Mal identisch. Ist die Arbeit auf dem neuen Branch zu Ende, kann dieser gelöscht werden. Andernfalls steht er für weitere Änderungen zur Verfügung.

Um die Änderungen in das Remote-Repository zu übertragen, betätigt man mit der rechten Maustaste den lokalen Master-Branch und wählt „Push Branch“. Nach Abschluss sollten alle Zweige denselben Hash-Wert und dieselben Kommentare aufzeigen.

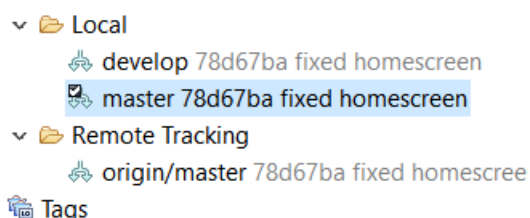


Abbildung 22 Erfolgreicher Push nach Branching und Merging
Quelle: Eclipse

Falls beim Versuch zu pushen ein Konflikt auftritt und die Fehlermeldung „rejected – non-fast-forward“ ausgegeben wird, gehen Sie wie folgt vor:

Wählen Sie unter „Remotes“ und „origin“ mit der rechten Maustaste den Pfad mit dem grünen Pfeil und klicken Sie auf „Configure Fetch“. Durch einen Fetch werden Daten aus dem externen Repository in das lokale übertragen. Die nun angezeigte URL sollte mit der des externen Repositorys übereinstimmen. Klicken Sie auf „Add“ und geben Sie den Namen des Branches an, aus dem die Daten übertragen werden sollen. Üblicherweise ist das der Master-Branch. Bestätigen Sie mit „Finish“ und schließen Sie mit „Save and Fetch“ ab. Spätestens jetzt sollte unter „Remote Tracking“ ein weiterer Branch zu sehen sein.

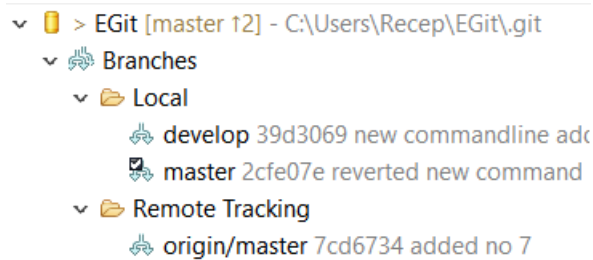


Abbildung 23 Fetch
Quelle: Eclipse

Führen Sie einen Rechtsklick auf den lokalen Master-Branch aus und wählen Sie „Merge“, um den lokalen Master Branch und den „origin“ zusammenzuführen“.

Nachdem der Merge abgeschlossen ist, muss als Nächstes ein Commit auf dem lokalen Master erfolgen und abschließend ein Push auf das externe Repository.

5.5 Rückgängig machen

Um Änderungen rückgängig machen zu können, sollte wie folgt vorgegangen werden.¹³¹

Es kommt durchaus vor, dass es notwendig ist, vorgenommene Änderungen rückgängig zu machen. Um dies zu bewerkstelligen wird ein Rechtsklick auf die Datei ausgeführt, bei der der Rückgang erfolgen soll. Anschließend wird die Option „Replace With“ ausgewählt und zum Schluss der Befehl „Previous Revision“. Die rückgängig gemachte Version muss erneut gestaged und committet werden.

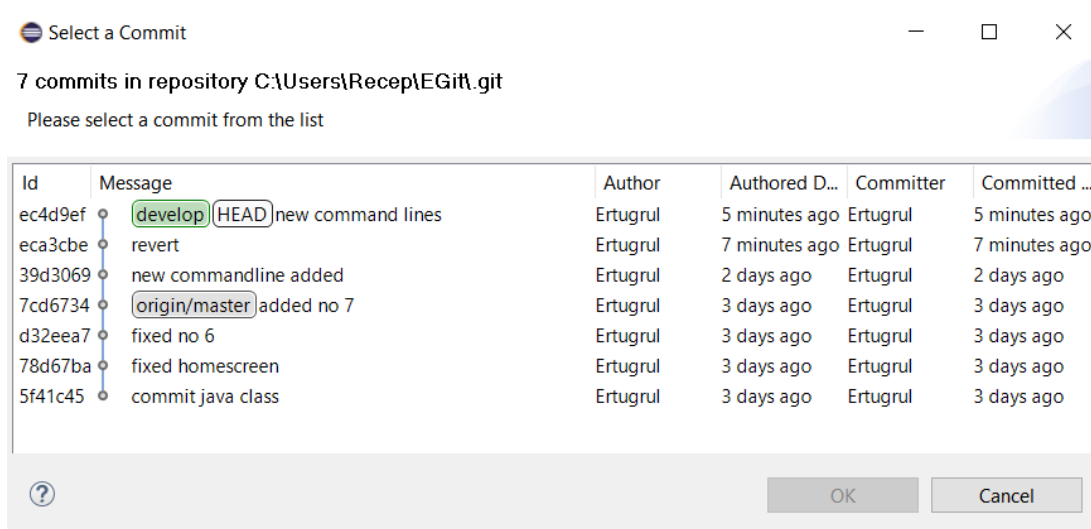


Abbildung 24 Rückgängig machen

Quelle: Eclipse

Auch ist es möglich den letzten getätigten Commit zu bearbeiten. Dafür klickt man auf „Amend“ auf der Konsole.

¹³¹ Vgl. Uwe Bretschneider, „Git tutorial for Eclipse (2) – Reverting file changes (german)“, Aufschaltung 09.05.2014, <https://youtu.be/syckCCoJZAaw>, Geprüft am 25.08.2017.

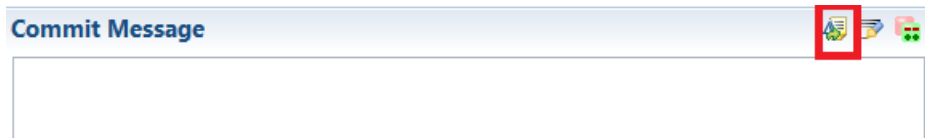


Abbildung 25 Letzten Commit bearbeiten

Quelle: Eclipse

Wenn die Version, die wiederhergestellt werden soll, mehrere Commits zurück liegt, wird anstatt „Previous Revision“ der Befehl „Commit“ ausgeführt. Dadurch erscheint der Commitverlauf des Branches. Hier lässt sich aussuchen, zu welcher Version zurückgegangen werden soll.

5.6 Zusammenfassung

Um mit dem Plugin eGit in Eclipse arbeiten zu können muss zunächst die Software Eclipse installiert werden. Über Eclipse selber wird das Plugin danach heruntergeladen und installiert. Das Plugin bietet grundlegende Funktionen wie Branching und Merging, das Erstellen eines lokalen Repositorys, die Anbindung an ein externes Repository sowie den Austausch von Daten zwischen dem lokalen und dem externen Repository. Voraussetzung hierfür ist die Registrierung und das Anlegen eines Repositorys auf der Webseite Github.com. Des Weiteren lässt sich über das Plugin der Verlauf von getätigten Commits einsehen, deren Hash-Werte mit den zugehörigen Kommentaren und auch eine Ansicht, die Unterschiede zu älteren Versionen anzeigt. Darüber hinaus lassen sich mit ein paar Klicks ältere Versionen von Programmen herstellen, auch wenn diese mehrere Commits zurück liegen. Dadurch lassen sich Fehler, welche man erst im Nachhinein erkennt, rückgängig machen.

6 Fazit

Ziel dieser Arbeit war es, das Versionsverwaltungssystem Git in Java-Programmier-Vorlesungen zu integrieren. Dazu muss zunächst ein Verständnis für die Grundlagen der Versionsverwaltung vorhanden sein. Diese wurden in Kapitel zwei behandelt, um einen Überblick über die Aufgaben und Funktionen der Versionskontrolle zu schaffen.

Des Weiteren wurde der Versionsverwaltungssoftware Git ein eigenes Kapitel gewidmet, um zu verstehen, inwiefern sich Git von seinen Vorgängern und von der Konkurrenz abhebt und worin der Vorteil bei der Arbeit mit Git liegt.

Bei der Arbeit mit Versionsverwaltung hängt viel von der eigenen oder der Unternehmensphilosophie ab. Unterschiedliche Systeme bieten unterschiedliche Arbeitsweisen und Zugriffsberechtigungen an. Aufgrund dessen bietet Kapitel vier eine Gegenüberstellung bekannter Versionskontrollsysteme, die es Unternehmen oder Privatpersonen erleichtern soll, sich für ein System zu entscheiden.

Wichtig bei der Arbeit mit Versionskontrolle im Bereich der Software-Entwicklung ist, dass ein reibungsloser und übersichtlicher Datenaustausch gewährleistet wird. Hierfür bietet die Software Eclipse ein Plugin namens eGit an. Durch dieses Plugin lassen sich alle wichtigen Funktionen von Git über die Benutzeroberfläche von Eclipse gemeinsam nutzen. Eine Schritt für Schritt Anleitung für das Vorgehen von der Registrierung auf GitHub bis zur Installation und Bedienung vom eGit Plugin in Eclipse ist in Kapitel 5 beschrieben.

Durch diese Anleitung und durch die Informationen in den anderen Kapiteln wird den Studenten eine solide Basis im Umgang mit Versionsverwaltungssystemen gegeben. Diese Basis ermöglicht es den Studenten zusammen mit dem Professor an semesterübergreifenden Projekten arbeiten zu können.

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.

[Aldingen, 30.08.2017 Ertugrul Özkara]

Literaturverzeichnis

Bechara, John, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement.“ (2009). Geprüft am 17. Juni 2017. Online: <https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf>.

Burch, Philipp, „Versionsverwaltung mit Mercurial (Teil 2) - ActiveVB.“. Geprüft am 11. April 2017. Online: https://activevb.de/tutorials/tut_versionsverwaltung/tut_versionsverwaltung_2.html.

Chacon, Scott and Ben Straub, „Git - Book.“. Geprüft am 19. April 2017. Online: <https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen>.

Dederer, Paul, Matrikel-Nr.: 245211, paul.dederer@hs-furtwangen.de, „Effiziente Softwareentwicklung durch Versionskontrolle.“ (2016). Geprüft am 8. Mai 2017.

Denker, Merlin and Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog.“ (2015). Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar_Git.pdf.

Dr.-Ing. Mathias Magdowski, „Versionskontrolle mit Apache Subversion.“. Geprüft am 8. Mai 2017. Online: http://www.studentbranch.ovgu.de/studentbranch_media/Downloads/IEEE+Versionskontrolle+Workshop/Versionskontrolle_mit_Apache_Subversion.pdf.

Driessen, Vincent, „A successful Git branching model.“. Geprüft am 8. Juni 2017. Online: <http://nvie.com/posts/a-successful-git-branching-model/>.

Fischer, Lars, „Werkzeuge zum Versions- und Variantenmanagement: CVS/Subversion vs. ClearCase.“. Geprüft am 17. Mai 2017. Online: <https://www.wi1.uni-muenster.de/pi/lehre/ws0506/skiseminar/versionsverwaltung.pdf>.

Fünten, Alexander a. d., „GIT & SVN. Versionsverwaltung.“ (2012). Geprüft am 30. August 2017. Online: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwi7xd_Q8f3VAhXKtRQKHappDsAQFggpMAA&url=http%3A%2F%2Fdme.rwth-aachen.de%2Fen%2Fsystem%2Ffiles%2Ffile_upload%2Fcourse%2F12%2Fproseminar-methoden-und-werkzeuge%2Fausarbeitung-oeffentlich.pdf&usg=AFQjCNHidC4cQa-WXFm2Re4BosAjpixqOA.

GlossarWiki, „Versionsverwaltung – GlossarWiki.“. Geprüft am 27. April 2017. Online: <https://glossar.hs-augsburg.de/Versionsverwaltung>.

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

Herkommer, Günter, „Die Trends beim Versionsmanagement.“. Geprüft am 12. April 2017. Online: <http://www.computer-automation.de/steuerungsebene/steuern-regeln/artikel/79313/2/>.

Lämmer, Frank, „Traditionelle Webentwicklung vs Versionsverwaltung.“. Geprüft am 6. April 2017. Online: <http://t3n.de/news/traditionelle-webentwicklung-versionsverwaltung-303580/>.

Loeliger, Jon, „Version control with Git.“. Geprüft am 29. August 2017. Online: <http://www.foo.be/cours/dess-20122013/b/OReilly%20Version%20Control%20with%20GIT.pdf>.

Mauel, Volker, „Vergleich von Git und SVN.“. Geprüft am 17. Juni 2017. Online: <https://www.volkermauel.de/Downloads/GitVsSvn.pdf>.

Meyer, Eric A., „Git: Verteilte Versionsverwaltung.“. Geprüft am 6. April 2017.
Online: <http://www.iks.kit.edu/diverses/gitvortrag/>.

o.V., „Git vs. SVN – Versionsverwaltung im Vergleich. Von verteilter und zentralisierter Versionsverwaltung.“. Geprüft am 3. Mai 2017. Online: <https://hosting.1und1.de/digitalguide/websites/web-entwicklung/git-vs-svn-versionsverwaltung-im-vergleich/>.

Preißel, René and Bjørn Stachmann. *Git. Dezentrale Versionsverwaltung im Team : Grundlagen und Workflows*. Heidelberg: dpunkt.Verlag, 2016.

Riedel, Sven. *Git. Kurz & gut*. Beijing: O'Reilly, 2010.

Sieverdingbeck, Ingo and Jasper van den Ven, „Versionsverwaltung mit SVN.“. Geprüft am 8. Mai 2017. Online: <http://rn.informatik.uni-bremen.de/lehre/c++/svn.pdf>.

Stargardt, Niels, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung.“. Geprüft am 17. Juni 2017. Online: https://www.sigs-datacom.de/uploads/tx_dmjournals/stargardt_JS_04_12_la6g.pdf.

taentzer, „Versionsverwaltung von Softwareartefakten.“. Geprüft am 17. Juni 2017. Online: <http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf>.

Tartler, Reinhard and Martin Steigerwald, „Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin.“. Geprüft am 6. April 2017. Online: <http://www.linux-magazin.de/Ausgaben/2007/06/Markt-Modell>.

THM-Wiki, „Git-basierte kollaborative Entwicklung von Webanwendungen (GitHub/GitLab) – THM-Wiki.“. Geprüft am 6. April 2017. Online: [https://wiki.thm.de/Git-basier-te_kollaborative_Entwicklung_von_Webanwendungen_\(GitHub/GitLab\)](https://wiki.thm.de/Git-basier-te_kollaborative_Entwicklung_von_Webanwendungen_(GitHub/GitLab)).

Thomas, David, Andrew Hunt, Falk Lehmann und Uwe Petschke, „Der pragmatische Programmierer.“. *Der pragmatische Programmierer*, Hunt, Andrew, [2. Aufl.]. - München [u.a.] : Hanser (2003): 1–7. Geprüft am 17. Juni 2017. Online:

http://files.hanser.de/hanser/docs/20051012_251121378-66_3-446-40470-8_Leseprobe1.pdf.

tinatigertech, „„Git“ your work done and don’t mess with your team!“. Geprüft am 11. April 2017. Online: <http://www.tigertech.de/git-your-work-done-and-dont-mess-with-your-team/#more-629>.

Vijayakumaran, Sujeevan. *Versionsverwaltung mit Git*. Frechen: mitp, 2016.

github.com

Brian Fraser, DrBFRaser, „Creating a Repository: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/r5C6yXNaSGo?list=PL-suslzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 21.08.2017.

Brian Fraser, DrBFRaser, „Checking out an existing project: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/V42r5REJx-M?list=PL-suslzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 22.08.2017.

Brian Fraser, DrBFRaser, „Making Changes: Git & Eclipse“, Aufschaltung 21.05.2013, <https://youtu.be/rbIGZRWqFVI?list=PL-suslzEBiMo0B5RcAikOaqDLKoG9Okub>, Geprüft am 23.08.2017.

Uwe Bretschneider, „Git tutorial for Eclipse (2) – Reverting file changes (german)“, Aufschaltung 09.05.2014, <https://youtu.be/syckCCoJZAaw>, Geprüft am 25.08.2017.