C:\Users\haller\Desktop\Logo_HFU.tif

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: | Ertugrul Özkara |
|  | 245269 |
|  | Steigstraße 54  78554 Aldingen |
|  | ertugrul.oezkara@hs-furtwangen.de |

# Abstract

Ziel dieser Bachelorarbeit war die Implementierung der verteilten Versionsverwaltungssoftware Git in die Java Programmier-Vorlesung von Prof. Illik. Dazu wurden genauere Betrachtungen des Themas Versionsverwaltung unternommen und ein Einblick über die wesentlichen Aufgaben gewonnen. Darauf folgend wurde die Entstehungsgeschichte der Versionskontrolle betrachtet und ein Überblick über die bekanntesten und gängigsten Systeme ermöglicht, für ein besseres Verständnis bei der Hinführung zur Entstehung von Git. Anschließend wurde das Thema Git 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. The term of version control itself were studied and a good insight of what the main tasks are, were cleared. The history of version control also was 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. Also a little comparison was made among the systems. Finally it was shown how to implement Git into the lecture step by step for the common usage of Students and the Prof. during a term.

# Inhaltsverzeichnis

[Abstract III](#_Toc488923923)

[Inhaltsverzeichnis V](#_Toc488923924)

[Abbildungsverzeichnis IX](#_Toc488923925)

[Tabellenverzeichnis XI](#_Toc488923926)

[Abkürzungsverzeichnis XIII](#_Toc488923927)

[1 Einleitung 15](#_Toc488923928)

[2 Grundlagen 16](#_Toc488923929)

[2.1 Definition 17](#_Toc488923930)

[2.2 Was ist Versionsverwaltung 18](#_Toc488923931)

[2.3 Vorteile einer Versionsverwaltung 19](#_Toc488923932)

[2.4 Geschichte der Versionsverwaltung 20](#_Toc488923933)

[2.4.1 SCCS 21](#_Toc488923934)

[2.4.2 CVS 21](#_Toc488923935)

[2.4.3 SVN 22](#_Toc488923936)

[2.5 Bekannte Versionsverwaltungssysteme 23](#_Toc488923937)

[2.5.1 CVS 23](#_Toc488923938)

[2.5.2 SVN 24](#_Toc488923939)

[2.6 Funktionsweise 27](#_Toc488923940)

[2.6.1 Problem(e) 27](#_Toc488923941)

[2.6.2 Lösungswege 29](#_Toc488923942)

[2.7 Arten der Versionsverwaltung 31](#_Toc488923943)

[2.7.1 Lokale Versionsverwaltung 31](#_Toc488923944)

[2.7.2 Zentrale Versionsverwaltung 32](#_Toc488923945)

[2.7.3 Nachteile der zentralen Versionsverwaltung 32](#_Toc488923946)

[2.7.4 Zentral versus verteilt 33](#_Toc488923947)

[2.7.5 Verteilte Versionsverwaltung 34](#_Toc488923948)

[2.7.6 Vorteile der verteilten Versionsverwaltung 35](#_Toc488923949)

[2.8 Verschiedene Konzepte der Versionsverwaltung 36](#_Toc488923950)

[2.8.1 Lock Modify Write 36](#_Toc488923951)

[2.8.2 Copy Modify Merge 36](#_Toc488923952)

[2.9 Zusammenfassung 37](#_Toc488923953)

[3 Git 39](#_Toc488923954)

[3.1 Die Entstehung von Git 39](#_Toc488923955)

[3.2 Die Grundlagen und Eigenschaften von Git 41](#_Toc488923956)

[3.2.1 Die Staging Area oder auch der Index 44](#_Toc488923957)

[3.3 Funktionsweise von Git 44](#_Toc488923958)

[3.4 Vorteile von Git 44](#_Toc488923959)

[3.5 Nachteile von Git 46](#_Toc488923960)

[3.6 Wichtige Begriffe innerhalb von Versionsverwaltungssystemen 46](#_Toc488923961)

[3.7 Branching und Merging 48](#_Toc488923962)

[3.7.1 Die Hauptentwicklungszweige 49](#_Toc488923963)

[3.7.2 Die Unterstützungszweige 50](#_Toc488923964)

[3.7.3 Der Feature-Branch 50](#_Toc488923965)

[3.7.4 Der Release-Branch 52](#_Toc488923966)

[3.7.5 Der Hotfix-Branch 54](#_Toc488923967)

[3.7.6 Branches verwalten 56](#_Toc488923968)

[3.8 Der HEAD 58](#_Toc488923969)

[3.9 Protokollierung 58](#_Toc488923970)

[3.10 Wiederherstellung 58](#_Toc488923971)

[3.11 Archivierung 58](#_Toc488923972)

[3.12 Koordinierung 58](#_Toc488923973)

[3.13 Gleichzeitiges Entwickeln 58](#_Toc488923974)

[3.14 Konflikte 58](#_Toc488923975)

[3.15 Verteilte Abläufe 58](#_Toc488923976)

[3.16 Hosting über GitHub 58](#_Toc488923977)

[3.17 Befehle 58](#_Toc488923978)

[3.18 Kommandozeile zur Bedienung von Git 58](#_Toc488923979)

[3.19 Zusammenfassung 58](#_Toc488923980)

[4 Versionsverwaltungssysteme 59](#_Toc488923981)

[4.1 Ziele der Versionsverwaltung 59](#_Toc488923982)

[4.2 Verschiedene Versionsverwaltungssysteme im Überblick 59](#_Toc488923983)

[4.2.1 Darcs 59](#_Toc488923984)

[4.2.2 Git 59](#_Toc488923985)

[4.2.3 GNU arch 59](#_Toc488923986)

[4.2.4 Apache Subversion 59](#_Toc488923987)

[4.2.5 Fossil 59](#_Toc488923988)

[4.2.6 Mercurial 59](#_Toc488923989)

[4.2.7 Bazaar 59](#_Toc488923990)

[4.2.8 Gegenüberstellung verschiedener Systeme 59](#_Toc488923991)

[4.2.9 Git oder SVN? 59](#_Toc488923992)

[4.3 Zusammenfassung 59](#_Toc488923993)

[5 Umsetzung/Integration des Tools in die Vorlesung 59](#_Toc488923994)

[5.1 Git installieren 59](#_Toc488923995)

[5.2 GitHub 59](#_Toc488923996)

[5.3 Repository aufbauen 59](#_Toc488923997)

[5.3.1 Vorhandene Daten übernehmen 59](#_Toc488923998)

[5.3.2 Repository klonen 59](#_Toc488923999)

[5.4 Veränderung an Repository melden 59](#_Toc488924000)

[5.5 Commit-Verlauf einsehen 59](#_Toc488924001)

[5.6 Rückgängig machen 60](#_Toc488924002)

[5.7 Wartung und Wiederherstellung 60](#_Toc488924003)

[5.8 Git als PlugIn in Eclipse (eGit) 60](#_Toc488924004)

[5.9 Zusammenfassung 60](#_Toc488924005)

[6 Resumee 60](#_Toc488924006)

[Literaturverzeichnis 61](#_Toc488924007)

[Stichwortverzeichnis 63](#_Toc488924008)

[Eidesstattliche Erklärung 65](#_Toc488924009)

[A. [Anhang] 67](#_Toc488924010)

[7 Literaturverzeichnis 67](#_Toc488924011)

# Abbildungsverzeichnis

[Abbildung 1: Arbeit ohne Versionskontrolle 28](#_Toc484444959)

[Abbildung 2: Sperren der Dokumente 29](#_Toc484444960)

# Tabellenverzeichnis

[Tabelle 1 - Grundlegende Begriffe 46](#_Toc487813217)

# 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 |

# 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]](#footnote-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 ein-mal 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]](#footnote-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 jedoch stellt eine immer größer werdende Herausforderung dar. Abhilfe schaffen hier Versionsverwaltungssysteme, welche sich deshalb auch am schnellsten in dieser Branche am schnellsten verbreitet haben.[[3]](#footnote-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 2 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 verschiedene und gängige Versionsverwaltungssysteme unter die Lupe genommen und entsprechende wichtige Begriffe erklärt, um sich ein Bild darüber machen zu können, welche Systeme am geeignetsten für die Arbeit in Programmier-Vorlesung sind. Anschließend wird auf die Versionsverwaltungssoftware Git eingegangen Den Abschluss bildet die Integration von Git in die Vorlesung und die zugehörige Handhabung.

# Grundlagen

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

## 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 2016, S. 1)

**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 Repositorien 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 1 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 2013, S. 5)

## 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 verlorengeglaubte 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]](#footnote-4)

Dadurch soll das gemeinsame Arbeiten an einer Vielzahl von Dokumenten ermöglicht werden. Dabei gilt die höchste Priorität der Vermeidung von Da-tenverlust bei gleichzeitiger Entwicklung durch mehrere Personen.[[5]](#footnote-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 steht man vor einem Problem.[[6]](#footnote-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]](#footnote-7)

## 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]](#footnote-8)

Ein weiterer Vorteil, den Versionskontrolle mit sich bringt, ist das gleichzeitige Entwickeln. Mehrere Entwickler können gleichzeitig und kontrolliert an dem selben Quellcode arbeiten ohne das die Arbeit des jeweils anderen dadurch überschrieben wird.[[9]](#footnote-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 hat.[[10]](#footnote-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]](#footnote-11)

## Geschichte der Versionsverwaltung

Zu versuchen die Geschichte der Versionsverwaltung in vollem Maße darzustellen, würde den Rahmen dieser Arbeit übersteigen. Anhand verschiedener wegweisender Systeme ist es jedoch möglich zu der Entstehung der Entwicklung von Git hin zuführen und diese nachzuvollziehen.[[12]](#footnote-12)

### 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.[[13]](#footnote-13)

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 eingecheckt und die vorherige Sperrung wieder aufgehoben.[[14]](#footnote-14)

### 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.[[15]](#footnote-15)

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.[[16]](#footnote-16) Hauptsächlich fand die Bedienung hier über eine Kommandozeile statt.[[17]](#footnote-17), was alle Versionsverwaltungen im Grunde gemeinsam haben.

Durch CVS wurde bei dem 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 Veränderungen hingewiesen und es musste sich für eine Veränderung entschieden werden.[[18]](#footnote-18)

### SVN

Subversion ist ein zentrales Versionsverwaltungssystem und basiert auf einer Client- Server-Architektur.[[19]](#footnote-19)

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.[[20]](#footnote-20)

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.

## 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.

### 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.[[21]](#footnote-21)

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.[[22]](#footnote-22)

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.[[23]](#footnote-23)

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.[[24]](#footnote-24)

### SVN

Apache Subversion gilt als der Quasi-Nachfolger von CVS.[[25]](#footnote-25) 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 können. Das Arbeitsverzeichnis kann mit einem Checkout direkt vom Server erstellt werden.[[26]](#footnote-26)

Der Vorteil hierbei ist, dass bei großen Repositorien 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 Clienten mit Serververbindung zur Verfügung.[[27]](#footnote-27)

Nachteile dieser zentralen Architektur können jedoch sein das ein Commit 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.[[28]](#footnote-28)

Ein wichtiger Unterschied zwischen SVN und CVS ist der, dass bei SVN auch ganze Verzeichnisse versioniert werden können.[[29]](#footnote-29) Dabei werden immer nur die Unterschiede zwischen den Dateien übertragen.[[30]](#footnote-30) 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 Ständen, da nun zu jedem Zeitpunkt eine genaue Revision zugeordnet werden kann.[[31]](#footnote-31) SVN erlaubt das Verschieben und Umbenennen von Dateien.[[32]](#footnote-32) 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.[[33]](#footnote-33)

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.[[34]](#footnote-34)

Bei Subversion unterscheidet man zwischen drei verschiedenen Entwicklungszweigen. 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.[[35]](#footnote-35)

Eine weitere Besonderheit bei Subversion ist das es abgesehen vom Trunk, zwischen diesen Zweigen technisch gesehen keinen Unterschied gibt. Jeder 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 werden auch dementsprechend verwendet.[[36]](#footnote-36)

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 das 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 mitzubekommen, welche in der Zwischenzeit evtl. getätigt wurden. Die gemachten 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 SVN mitgeteilt werden. Als letzter Schritt ist ein Commit notwendig. Diesem wird eine kurze Erklärung beigefügt, was sich nun alles in der Revision geändert hat. Diese ist später im log-Protokoll wiederzufinden. Wie schon weiter oben erklärt, akutalisiert Subversion das Repository und erhöht die Revision um eins.[[37]](#footnote-37) 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.[[38]](#footnote-38)

## 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.

Zunächst werden auf Probleme eingegangen, welche bei der Bearbeitung einer einzelnen Datei oder bei der Arbeit in einem Großprojekt auftauchen würden ohne die Verwendung einer Versionskontrolle.

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

### 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 wir davon ausgehen, 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]](#footnote-39)

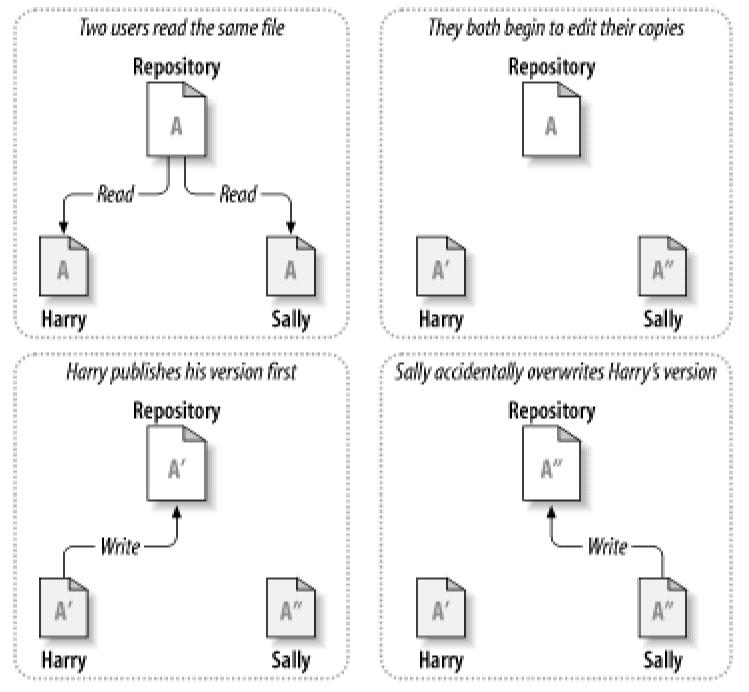


Abbildung : Arbeit ohne Versionskontrolle (Quelle: Taentzer 2014 - Versionsverwaltung von Softwareartefakten, S. 45)

### 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.   
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.[[40]](#footnote-40)

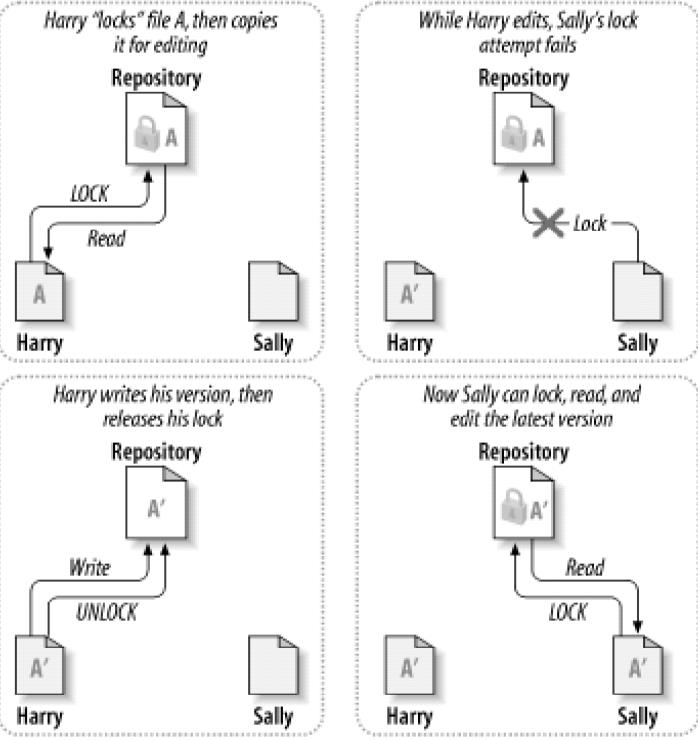


Abbildung Sperren der Dokumente (Quelle: Taentzer 2014 - Versionsverwaltung von Softwareartefakten, S. 46)

Der hier angegebene Schritt ist zwar ein guter Lösungsvorschlag, dennoch ist er nicht optimal. Es kann durchaus vorkommen das 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.  
Darüber hinaus würde eine Sperrung des Dokuments einen unnötigen Zeitverlust für denjenigen darstellen der zwar die selbe 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, könnte 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.[[41]](#footnote-41)

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.[[42]](#footnote-42)

Zu beachten ist, dass es bei diesem Lösungsansatz zu Konflikten kommen könnte. Wenn beide Nutzer etwa die selbe 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 Dokumenten zusammengeführt werden können, ohne dass sich dabei Probleme aufzeigen.[[43]](#footnote-43)

## 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.

### 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 bestimmte Fetaure 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.[[44]](#footnote-44)

### 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.  
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 öfters erwähnt, die Bearbeitung der exakt selben Zeile.[[45]](#footnote-45)

### 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 Wahrscheinlichkeit 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.[[46]](#footnote-46)

In heutigen Projekten werden entwickelte Features erst commitet, 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.[[47]](#footnote-47)

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.[[48]](#footnote-48)

### 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.[[49]](#footnote-49)

Um dieses Problem umgehen zu können ist es ratsam einen weiteren Branch anzulegen. Also einen weiteren Entwicklungszweig mit demselben Inhalt und diese beiden nach den 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.[[50]](#footnote-50)

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.[[51]](#footnote-51)

### 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.[[52]](#footnote-52)

Bei dieser Art der Versionsverwaltung herrscht eine hierarchische Ordnung zwischen den Repositorien. Der Austausch zwischen den Teilteams erfolgt normalerweise über ein oder mehrere zentrale Repositorien, von dem jedes Team ein eigenes erhält. Dies wäre für ein zentrales Versionsverwaltungssystem unüblich.[[53]](#footnote-53)

### 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.[[54]](#footnote-54)

Die verteilte Versionsverwaltung behebt einige der Probleme, welche bei der zentralen Versionskontrolle vorkommen. Da das zentrale Repositorium 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 Repositorien gesammelt und Stück für Stück zusammengesetzt.[[55]](#footnote-55)

## Verschiedene Konzepte der Versionsverwaltung

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

### 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, das 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.[[56]](#footnote-56)

### 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. Eine Auswirkung davon kann jedoch sein, das 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.[[57]](#footnote-57)

## Zusammenfassung

Ab diesem Zeitpunkt sollte man ein grundlegendes Verständnis darüber haben, was Versionsverwaltung bzw. Versionskontrolle ist. Darüber hinaus sollte klar sein wie diese funktioniert und welche Vorteile sie mit sich bringt. Des Weiteren wurde ein grober Überblick über die Entstehungsgeschichte der Versionsverwaltungssysteme gegeben und welche Systeme in den letzten Jahren die gängigsten und auch die Vorreiter in diesem Gebiet waren und teilweise heute noch sind.

Auch sollte klar sein welche Arten der Versionsverwaltung es gibt, was DVCS und CVCS ist und worin der Unterschied zwischen diesen beiden liegt. Bei den Begriffen Repository, Merge, Commit und Branch sollte Klarheit herrschen und man sollte wissen was jeder einzelne dieser Begriffe bedeutet sowie wofür und in welcher Situation gebraucht wird.

# 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, das es zu der 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 sind ebenfalls vorhanden, sowie die Bedienung von Git.

## Die Entstehung von Git

Während der Entwicklung des Linux-Kernels wurden Änderungen am Quellcode in Form von Patches übergeben und weitergereicht. Im Jahre 2002 entschied man sich dafür BitKeeper zu verwenden. BitKeeper ist ein Source-Control-Management (SCM) System und erleichtert die Arbeit bei der Verwaltung von Quellcode.[[58]](#footnote-58) 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.[[59]](#footnote-59)

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.[[60]](#footnote-60)

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.[[61]](#footnote-61)

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.[[62]](#footnote-62)

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)

## 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.[[63]](#footnote-63) 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 somit 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.[[64]](#footnote-64) Diese Referenzierung erfolgt bei Git nicht über einen Namen, sondern über eine Checksumme, welche 40 Zeichen lang ist. Berechnet wird diese entweder aus dem Inhalt oder der Verzeichnisstruktur und verhindert Übertragungsfehler oder hilft dabei beschädigte Dateien ausfindig zu machen.[[65]](#footnote-65)

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 langsam ist. Bei Git allerdings wird eine Datei einfach als ganze ausgecheckt, ohne dass dafür noch weitere Operationen nötig sind.[[66]](#footnote-66)

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.[[67]](#footnote-67) 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.[[68]](#footnote-68)

Der wichtigste 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 dem lokalen und dem Repository auf dem Server keinen Unterschied, jedoch beinhaltet das Master-Repository den Hauptentwicklungszweig.[[69]](#footnote-69)

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.[[70]](#footnote-70)

Bei den meisten ausgeführten Operationen werden zumeist Dateien benötigt, welche lokal auf dem Rechner vorhanden sind. Somit entfällt die meiste Zeit die Kommunikation mit einem anderen Rechner über ein Netzwerk und 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.[[71]](#footnote-71)

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 wurden.  
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 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.[[72]](#footnote-72)

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.[[73]](#footnote-73)

### 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, welchen man an einer Datei vorgenommen hat, zwingend als Commit einchecken muss.  
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.[[74]](#footnote-74)

## Funktionsweise von Git

## 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.[[75]](#footnote-75)

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.[[76]](#footnote-76)

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 gemachten Änderungen wieder übergeben. Bei der zentralen Versionskontrolle wird das durch verschiedene Lese –und schreibrechte erschwert.[[77]](#footnote-77)

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.[[78]](#footnote-78)

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.[[79]](#footnote-79)

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.[[80]](#footnote-80)

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 Anwender unter die Arme greifen. Git weist auch eine stark wachsende Plattform für Tools, Hosting-Plattformen und anderen hilfreichen Plug-ins für die Entwicklung auf.[[81]](#footnote-81)

## Nachteile von Git

## 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.

**Tabelle 1 - Grundlegende Begriffe[[82]](#footnote-82)**

|  |  |
| --- | --- |
| 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 wie z.B. Datum, Uhrzeit, Autor und eine Nachricht (Commit Message), welche die gemachten Veränderungen beschreibt. |
| HEAD: | Referenz auf den letzten durchgeführten Commit und den entsprechenden Entwicklungszweig. Bildet den Kopf oder die Spitze eines Entwicklungsstrangs. |
| SHA-1: | Durch den *Secure Hash Algorithm* 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: | Ein Clone ist eine Kopie eines Repositorys, welcher alle Daten und auch Commits enthält, die darin abgelegt wurden. |
| 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 *merge* 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 *master* 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. |

## 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 von dem 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.[[83]](#footnote-83)

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 einer der größten Herausforderungen für sie dar. 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. Diesen Vorgang wird als Merge-Commit bezeichnet.[[84]](#footnote-84)

Branches sind wie parallele Ablauflinien 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.[[85]](#footnote-85)

Der Hauptentwicklungszweig bei Git wird Master genannt und entspricht in anderen Versionskontrollsystem dem *trunk.* Ein Beispiel hierfür ist Subversion. Der aktuellste Commit eines Branches wird mit der Bezeichnung *HEAD* versehen.[[86]](#footnote-86) 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.[[87]](#footnote-87)  
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 Cache für die durchgeführten Änderungen.  
Eine Bearbeitung dieser Tracking-Branches ist nicht möglich, da diese von den lokalen Entwicklungszweigen abgezweigt sind.[[88]](#footnote-88)

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 dieses letztendlich wieder zusammenzuführen.[[89]](#footnote-89)

Ob es nun darum geht etwas ausprobieren zu wollen, sich um einen Bugfix zu kümmern oder an einem neuen Fetaure herumzuexperimentieren, für all diese Dinge werden zugehörige Branches erstellt, zusammengeführt oder wieder gelöscht. All das ist gängige Praxis unter Entwicklern auch in realen Projekten.[[90]](#footnote-90)

### 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 Entwicklungsmethode ab, welche man anwenden möchte.  
Der Master-Zweig im *origin* ist der Ort, an dem der Quellcode abgelegt wird, der schlussendlich auch veröffentlicht wird. Der Develop-Zweig hingegen enthält den zuletzt abgelieferten und für ein Release bereit stehenden Quellcode zur Verfügung.  
Wenn all der Fortschritt im Develop-Zweig an dem Punkt angelangt ist, das er veröffentlicht werden kann, sollte dieser Fortschritt mit dem Master-Branch zusammengelegt werden.[[91]](#footnote-91)

### 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 unterschieden:

1. Feature-Branch (Funktionszweig/Anforderungen)
2. Release-Branch (Veröffentlichung)
3. Hotfix-Branch (schnelle Fehlerbehebung)

[[92]](#footnote-92)

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.[[93]](#footnote-93)

### Der Feature-Branch

Fetaure-Branches 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, das es so lange bestehen bleibt, so lange der zu entwickelnde und benötigte Teil in Arbeit ist. Danach wird der Zweig in den 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-Branches nur in den Repositories der Entwickler und nicht auf dem *origin*.[[94]](#footnote-94)

Die Planung eines neuen Projektes oder Produktes sollte so gestaltet sein, das Anforderungen paketweise bearbeitet und übergeben werden. Je länger die Bearbeitung an einem Feature läuft, 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 von Features aufeinander haben und ob diese Änderungen ein weiteres Zusammenarbeiten von Anforderungen ermöglichen. Das Auslassen solcher Tests hat zur Folge, dass ein Fehlerhaftes Projekt in den Master-Branch gemerged wird und Fehler erst in diesem Branch entdeckt werden.[[95]](#footnote-95)

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

$ git pull --ff-only

Der letzte Befehl sorgt dafür, das nur ein Fast-Forward-Merge erlaubt ist. Dies bedeutet, dass ein Merge abgebrochen wird, wenn lokale Änderungen vorliegen. Sofern eine Fehlermeldung auftaucht, dann wurde höchstwahrscheinlich vorher auf dem Master-Branch gearbeitet. Diese Änderungen müssen zunächst auf einen Feature-Branch verschoben werden, damit es weiter gehen kann.[[96]](#footnote-96)

Um einen Fetaure 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-Branches>

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

Danach kann man den Feature-Branch löschen.

$ git branch -d <Name des Feature-Branches>

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

$ push origin develop[[97]](#footnote-97)

Weil Feature-Branches 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 auf dem zentralen Repository verknüpft.[[98]](#footnote-98)

### Der Release-Branch

Wie der Name schon sagt dienen Release-Branches 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 Develop-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 released werden.[[99]](#footnote-99)

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 großern oder einen kleineren Release handelt.[[100]](#footnote-100)

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 sinvoll auch gleich die Versionnummer in den Namen des Branches einzutragen. Beispielsweise könnte der Befehl dann so aussehen:

$ 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 comitted. 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 releasen 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

Der Release ist jetzt durch und 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-1.2[[101]](#footnote-101)

### 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.[[102]](#footnote-102)

Tauchen in einem neuen Release schwerwiegender Fehler auf, müssen diese so schnell wie möglich behoben werden. Da auf dem Develop-Zweig inzwischen an einem neuen Release gearbeitet wird, ist es daher nicht ratsam, die fehlerhafte Version in den Develop-Zweig zu mergen um die Fehler dort zu beheben. Stattdessen wird aus dem Master heraus ein neuer Zweig erstellt, welcher Hotfix-Branch genannt wird. Zu beachten sind die Commits mit den entsprechenden *tags* und den Versionsnummern.

$ 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

Anschließend wird der Hotfix-Branch zurück in den Master gemerged aber auch in den Develop-Zweig. Dadurch wird sichergestellt, dass die verbesserten Fehler, nicht erneut auftauchen.[[103]](#footnote-103)

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.[[104]](#footnote-104)

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

$ git branch -d hotfix-<Versionsnummer>

### 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.[[105]](#footnote-105)

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.

$ 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 In-dex 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.[[106]](#footnote-106)

Git sorgt dafür, dass Informationen nicht verloren gehen. Es ist wahrscheinlicher dass ein Checkout fehlschlägt, als dass Dateien überschrieben werden. Git würde einen Checkout fehlschlagen lassen, wenn:[[107]](#footnote-107)

* 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 überschreiben werden würde, welche nicht von Git verwaltet wird.

### Fast Forward Merges

## Der HEAD

## Protokollierung

## Wiederherstellung

## Archivierung

## Koordinierung

## Gleichzeitiges Entwickeln

## Konflikte

## Verteilte Abläufe

## Hosting über GitHub

## Befehle

## Kommandozeile zur Bedienung von Git

## Zusammenfassung

# Versionsverwaltungssysteme

## Ziele der Versionsverwaltung

## Verschiedene Versionsverwaltungssysteme im Überblick

### Darcs

### Git

### GNU arch

### Apache Subversion

### Fossil

### Mercurial

### Bazaar

### Gegenüberstellung verschiedener Systeme

### Git oder SVN?

## Zusammenfassung

# Umsetzung/Integration des Tools in die Vorlesung

## Git installieren

## GitHub

## Repository aufbauen

### Vorhandene Daten übernehmen

### Repository klonen

## Veränderung an Repository melden

## Commit-Verlauf einsehen

## Rückgängig machen

## Wartung und Wiederherstellung

## Git als PlugIn in Eclipse (eGit)

## Zusammenfassung

# Resumee

# Literaturverzeichnis

Fügen Sie hier ihre verwendete Literatur ein. Beachten Sie dabei die Vorgaben zu den Zitierstilen

# Stichwortverzeichnis

Das Stichwortverzeichnis ist optional. Wenn Sie kein Stichwortverzeichnis in Ihrer Arbeit verwenden wollen können Sie dieses Kapitel entfernen!

# 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.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[Ort, Datum Name]

Auch hier müssen die Platzhalter mit den korrekten Daten ersetzt werden.

# [Anhang]

Der Anhang ist optional. Wenn Sie keinen Anhang in Ihrer Arbeit verwenden wollen können Sie dieses Kapitel entfernen!

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.

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 17. Juni 2017.

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.

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

Loeliger, Jon and Matthew McCullough, „Version control with Git. [powerful tools and techniques for collaborative software development ; covers GitHub].“. Online: http://proquest.tech.safaribooksonline.de/book/databases/content-management-systems/9781449345037/firstchapter#X2ludGVybmFsX0h0bWxWaWV3P3htbGlkPTk3ODE0NDkzNDUwMzclMkZpZDI5MzI5NCZxdWVyeT0=.

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/.

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-basierte\_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.

1. Vgl. Merlin Denker und Stefan Srecec, „Versionsverwaltung mit Git. Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog,“ (2015): 1. Geprüft am 17. Juni 2017. Online: http://merlindenker.de/files/Proseminar\_Git.pdf. [↑](#footnote-ref-1)
2. Vgl. Sujeevan Vijayakumaran, *Versionsverwaltung mit Git* (Frechen: mitp, 2016), S. 14 [↑](#footnote-ref-2)
3. Vgl. John Bechara, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement,“ (2009): 5. Geprüft am 17. Juni 2017. Online: https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf. [↑](#footnote-ref-3)
4. Vgl. Sujeevan Vijayakumaran, *Versionsverwaltung mit Git* (Frechen: mitp, 2016), S. 17 [↑](#footnote-ref-4)
5. Vgl. taentzer, „Versionsverwaltung von Softwareartefakten,“, S. 43. Geprüft am 17. Juni 2017. Online: http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf. [↑](#footnote-ref-5)
6. Vgl. Vijayakumaran (wie Anm. 2), S. 17 [↑](#footnote-ref-6)
7. Vgl. Vijayakumaran (wie Anm. 2), S. 17–18 [↑](#footnote-ref-7)
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): 1–7. Geprüft am 17. Juni 2017. Online: http://files.hanser.de/hanser/docs/20051012\_251121378-66\_3-446-40470-8\_Leseprobe1.pdf. [↑](#footnote-ref-8)
9. Vgl. Thomas et al., „*Der pragmatische Programmierer,“* (wie Anm. 8), S. 1 [↑](#footnote-ref-9)
10. Vgl. Thomas et al., „*Der pragmatische Programmierer,“* (wie Anm. 8), S. 1–2 [↑](#footnote-ref-10)
11. Vgl. Thomas et al., „*Der pragmatische Programmierer,“* (wie Anm. 8), S. 2 [↑](#footnote-ref-11)
12. Vgl. Frank Lämmer, „Traditionelle Webentwicklung vs Versionsverwaltung,“. Geprüft am 6. April 2017. Online: http://t3n.de/news/traditionelle-webentwicklung-versionsverwaltung-303580/. [↑](#footnote-ref-12)
13. Vgl. Loeliger, Jon and Matthew McCullough, „Version control with Git. [powerful tools and techniques for collaborative software development ; covers GitHub].“. Online: http://proquest.tech.safaribooksonline.de/book/databases/content-management-systems/9781449345037/firstchapter#X2ludGVybmFsX0h0bWxWaWV3P3htbGlkPTk3ODE0NDkzNDUwMzclMkZpZDI5MzI5NCZxdWVyeT0=., S. 5 [↑](#footnote-ref-13)
14. Vgl. Loeliger und McCullough, „*Version control with Git,“* (wie Anm. 13), S. 5 [↑](#footnote-ref-14)
15. Vgl. Loeliger und McCullough, „*Version control with Git,“* (wie Anm. 13), S. 5 [↑](#footnote-ref-15)
16. Vgl. Loeliger und McCullough, „*Version control with Git,“* (wie Anm. 13), S. 5 [↑](#footnote-ref-16)
17. 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 [↑](#footnote-ref-17)
18. Vgl. Loeliger und McCullough, „*Version control with Git,“* (wie Anm. 13), S. 5 [↑](#footnote-ref-18)
19. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 11 [↑](#footnote-ref-19)
20. Vgl. Loeliger und McCullough, „*Version control with Git,“* (wie Anm. 13), S. 5 [↑](#footnote-ref-20)
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. 27 [↑](#footnote-ref-21)
22. Vgl. Fischer, „*Werkzeuge zum Versions- und Variantenmanagement:,“* (wie Anm. 17), S. 3 [↑](#footnote-ref-22)
23. Vgl. Fischer, „*Werkzeuge zum Versions- und Variantenmanagement:,“* (wie Anm. 17), S. 5 [↑](#footnote-ref-23)
24. Vgl. Lämmer, „*Traditionelle Webentwicklung vs Versionsverwaltung,“* (wie Anm. 12) [↑](#footnote-ref-24)
25. Vgl. Dr.-Ing. Mathias Magdowski, „*Versionskontrolle mit Apache Subversion,“* (wie Anm. 21), S. 28 [↑](#footnote-ref-25)
26. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 11 [↑](#footnote-ref-26)
27. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 11–12 [↑](#footnote-ref-27)
28. Vgl. Volker Mauel, „Vergleich von Git und SVN,“, S. 8. Geprüft am 17. Juni 2017. Online: https://www.volkermauel.de/Downloads/GitVsSvn.pdf. [↑](#footnote-ref-28)
29. Vgl. Alexander a. d. Fünten, „GIT & SVN. Versionsverwaltung,“ (2012): 5. Geprüft am 17. Juni 2017. [↑](#footnote-ref-29)
30. Vgl. Dr.-Ing. Mathias Magdowski, „*Versionskontrolle mit Apache Subversion,“* (wie Anm. 21), S. 28 [↑](#footnote-ref-30)
31. Vgl. Alexander a. d. Fünten, „GIT & SVN. Versionsverwaltung,“ (2012): 6. Geprüft am 17. Juni 2017. [↑](#footnote-ref-31)
32. Vgl. Dr.-Ing. Mathias Magdowski, „*Versionskontrolle mit Apache Subversion,“* (wie Anm. 21), S. 28 [↑](#footnote-ref-32)
33. 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 [↑](#footnote-ref-33)
34. Vgl. Lars Fischer, „Werkzeuge zum Versions- und Variantenmanagement:. CVS/Subversion vs. ClearCase,“: 3. Geprüft am 17. Mai 2017. Online: https://www.wi1.uni-muenster.de/pi/lehre/ws0506/skiseminar/versionsverwaltung.pdf. [↑](#footnote-ref-34)
35. Vgl. Fünten, „*GIT & SVN,“* (wie Anm. 29), S. 6 [↑](#footnote-ref-35)
36. Vgl. Fünten, „*GIT & SVN,“* (wie Anm. 29), S. 6 [↑](#footnote-ref-36)
37. Vgl. Fünten, „*GIT & SVN,“* (wie Anm. 29), S. 6–7 [↑](#footnote-ref-37)
38. Vgl. Volker Mauel, „Vergleich von Git und SVN,“, S. 7. Geprüft am 17. Juni 2017. Online: https://www.volkermauel.de/Downloads/GitVsSvn.pdf. [↑](#footnote-ref-38)
39. Vgl. taentzer, „Versionsverwaltung von Softwareartefakten,“, S. 45. Geprüft am 17. Juni 2017. Online: http://www.uni-marburg.de/fb12/arbeitsgruppen/swt/lehre/files/est1415/ESM141021.pdf. [↑](#footnote-ref-39)
40. Vgl. taentzer, „*Versionsverwaltung von Softwareartefakten,“* (wie Anm. 5), S. 46 [↑](#footnote-ref-40)
41. Vgl. taentzer, „*Versionsverwaltung von Softwareartefakten,“* (wie Anm. 5), S. 47 [↑](#footnote-ref-41)
42. Vgl. taentzer, „*Versionsverwaltung von Softwareartefakten,“* (wie Anm. 5), S. 48 [↑](#footnote-ref-42)
43. Vgl. taentzer, „*Versionsverwaltung von Softwareartefakten,“* (wie Anm. 5), S. 50 [↑](#footnote-ref-43)
44. Vgl. Dederer, Paul, Matrikel-Nr.: 245211, paul.dederer@hs-furtwangen.de, „Effiziente Softwareentwicklung durch Versionskontrolle,“ (2016): 4. Geprüft am 8. Mai 2017. [↑](#footnote-ref-44)
45. Vgl. Philipp Burch, „Versionsverwaltung mit Mercurial (Teil 2) - ActiveVB,“. Geprüft am 11. April 2017. Online: https://activevb.de/tutorials/tut\_versionsverwaltung/tut\_versionsverwaltung\_2.html. [↑](#footnote-ref-45)
46. Vgl. Niels Stargardt, „Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“, S. 1. Geprüft am 17. Juni 2017. Online: https://www.sigs-datacom.de/uploads/tx\_dmjournals/stargardt\_JS\_04\_12\_la6g.pdf. [↑](#footnote-ref-46)
47. Vgl. Stargardt, „*Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“* (wie Anm. 46), S. 1 [↑](#footnote-ref-47)
48. Vgl. Stargardt, „*Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“* (wie Anm. 46), S. 1 [↑](#footnote-ref-48)
49. Vgl. Reinhard Tartler und Martin Steigerwald, „Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin,“. Geprüft am 6. April 2017. Online: http://www.linux-magazin.de/Ausgaben/2007/06/Markt-Modell. [↑](#footnote-ref-49)
50. Vgl. Tartler und Steigerwald, „*Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin,“* (wie Anm. 49) [↑](#footnote-ref-50)
51. Vgl. Tartler und Steigerwald, „*Verteilte Versionsverwaltung mit Bazaar » Linux-Magazin,“* (wie Anm. 49) [↑](#footnote-ref-51)
52. Vgl. John Bechara, „Revisionssichere Archivierung. Verteiltes Dokumentenmanagement,“ (2009): 11. Geprüft am 17. Juni 2017. Online: https://www.unibw.de/inf2/Lehre/FT09/lza/bechara.pdf. [↑](#footnote-ref-52)
53. Vgl. Stargardt, „*Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“* (wie Anm. 46), S. 1 [↑](#footnote-ref-53)
54. Vgl. Stargardt, „*Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“* (wie Anm. 46), S. 35 [↑](#footnote-ref-54)
55. Vgl. Stargardt, „*Verteilte Versionsverwaltungssysteme in der kommerziellen Java-Entwicklung,“* (wie Anm. 46), S. 35 [↑](#footnote-ref-55)
56. Vgl. Bechara, „*Revisionssichere Archivierung,“* (wie Anm. 3), S. 8 [↑](#footnote-ref-56)
57. Vgl. Bechara, „*Revisionssichere Archivierung,“* (wie Anm. 3), S. 9 [↑](#footnote-ref-57)
58. 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). [↑](#footnote-ref-58)
59. Vgl. Mauel, „*Vergleich von Git und SVN,“* (wie Anm. 28), S. 6 [↑](#footnote-ref-59)
60. Vgl. Vijayakumaran (wie Anm. 2), S. 21 [↑](#footnote-ref-60)
61. Vgl. THM-Wiki, „*Git-basierte kollaborative Entwicklung von Webanwendungen (GitHub/GitLab) – THM-Wiki,“* (wie Anm. 58) [↑](#footnote-ref-61)
62. 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. [↑](#footnote-ref-62)
63. Vgl. Scott Chacon und Ben Straub, „Git - Book,“. Geprüft am 19. April 2017. Online: https://git-scm.com/book/de/v1. [↑](#footnote-ref-63)
64. Vgl. Chacon und Straub, „*Git - Book,“* (wie Anm. 63) [↑](#footnote-ref-64)
65. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 2 [↑](#footnote-ref-65)
66. Vgl. Vijayakumaran (wie Anm. 2), S. 51 [↑](#footnote-ref-66)
67. Vgl. Chacon und Straub, „*Git - Book,“* (wie Anm. 63) [↑](#footnote-ref-67)
68. Vgl. Eric A. Meyer, „Git: Verteilte Versionsverwaltung,“, S. 14–17. Geprüft am 6. April 2017. Online: http://www.iks.kit.edu/diverses/gitvortrag/. [↑](#footnote-ref-68)
69. Vgl. Fünten, „*GIT & SVN,“* (wie Anm. 29), S. 9–10 [↑](#footnote-ref-69)
70. Vgl. Fünten, „*GIT & SVN,“* (wie Anm. 29), S. 10 [↑](#footnote-ref-70)
71. Vgl. Chacon und Straub, „*Git - Book,“* (wie Anm. 63) [↑](#footnote-ref-71)
72. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 2–3 [↑](#footnote-ref-72)
73. Vgl. Denker und Srecec, „*Versionsverwaltung mit Git,“* (wie Anm. 1), S. 3 [↑](#footnote-ref-73)
74. Vgl. Valentin Haenel und Julius Plenz, *Git. Verteilte Versionsverwaltung für Code und Dokumente* (Professional reference; München: Open Source Press, 2011), S. 34 [↑](#footnote-ref-74)
75. René Preißel und Bjørn Stachmann, *Git. Dezentrale Versionsverwaltung im Team : Grundlagen und Workflows* (Heidelberg: dpunkt.Verlag, 2016), S. vii [↑](#footnote-ref-75)
76. Preißel und Stachmann (wie Anm. 75), S. vii [↑](#footnote-ref-76)
77. Preißel und Stachmann (wie Anm. 75), S. vii [↑](#footnote-ref-77)
78. Preißel und Stachmann (wie Anm. 75), S. vii [↑](#footnote-ref-78)
79. Preißel und Stachmann (wie Anm. 75), S. viii [↑](#footnote-ref-79)
80. Preißel und Stachmann (wie Anm. 75), S. viii [↑](#footnote-ref-80)
81. Preißel und Stachmann (wie Anm. 75), S. viii [↑](#footnote-ref-81)
82. Tabelle 1 Vgl. Haenel und Plenz (wie Anm. 74)., S. 19-21 [↑](#footnote-ref-82)
83. Vgl. Vijayakumaran (wie Anm. 2), S. 59 [↑](#footnote-ref-83)
84. Vgl. Preißel und Stachmann (wie Anm. 75), S. 5 [↑](#footnote-ref-84)
85. Vgl. Preißel und Stachmann (wie Anm. 75), S. 59 [↑](#footnote-ref-85)
86. Vgl. Sven Riedel, *Git. Kurz & gut* (O'Reillys Taschenbibliothek; Beijing: O'Reilly, 2010), S. 7 [↑](#footnote-ref-86)
87. Vgl. Sujeevan Vijayakumaran, *Versionsverwaltung mit Git* (Frechen: mitp, 2016), S. 14 [↑](#footnote-ref-87)
88. Vgl. Riedel (wie Anm. 86), S. 7 [↑](#footnote-ref-88)
89. Vgl. Haenel und Plenz (wie Anm. 74), S. 61–62 [↑](#footnote-ref-89)
90. Vgl. Haenel und Plenz (wie Anm. 74), S. 61–62 [↑](#footnote-ref-90)
91. Vgl. Vincent Driessen, „A successful Git branching model,“. Geprüft am 8. Juni 2017. Online: http://nvie.com/posts/a-successful-git-branching-model/. [↑](#footnote-ref-91)
92. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-92)
93. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-93)
94. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-94)
95. Vgl. Preißel und Stachmann (wie Anm. 75), S. 162 [↑](#footnote-ref-95)
96. Preißel und Stachmann (wie Anm. 75), S. 164 [↑](#footnote-ref-96)
97. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-97)
98. Vgl. Preißel und Stachmann (wie Anm. 75), S. 165 [↑](#footnote-ref-98)
99. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-99)
100. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-100)
101. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-101)
102. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-102)
103. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-103)
104. Vgl. Driessen, „*A successful Git branching model,“* (wie Anm. 91) [↑](#footnote-ref-104)
105. Vgl. Haenel und Plenz (wie Anm. 74), S. 65 [↑](#footnote-ref-105)
106. Vgl. Haenel und Plenz (wie Anm. 74), S. 67 [↑](#footnote-ref-106)
107. Vgl. Haenel und Plenz (wie Anm. 74), S. 67–68 [↑](#footnote-ref-107)