

Hochschule Pforzheim

Fakultät für Technik

Studiengang Technische Informatik (B.Sc.)

**Entwicklung eines Tutorials für die
Versionsverwaltung mit git**

Projektarbeit

Erstprüfer Herrn Prof. Richard Alznauer

vorgelegt von Lennart Kaussen
Matrikelnummer 317057

vorgelegt von Marc Retzlaff
Matrikelnummer 317430

Abgabetermin 29.03.2020

Erklärung

Wir versichern, die beiliegende Projektarbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet zu haben.

Pforzheim, den 29.03.2020



(Marc Retzlaff)



(Lennart Kaussen)

Kurzfassung

Diese Projektarbeit befasst sich mit der Erklärung der Versionskontrolle Git. Neben einer Erläuterung der wichtigsten Funktionen anhand eines durchlaufenden Beispiels, werden auch verschiedene Arbeitsabläufe diskutiert. Damit umfasst die Arbeit die lokale Installation von Git, sowie Richtlinien zum Einsatz in größeren Softwareprojekten. Um eine Wissensgrundlage für alle Anwendungen basierend auf Git zu erstellen, beziehen sich die Erklärungen auf die anwendungsübergreifende Kommandozeile. Für ein schnelles Nachschlagen bietet die Kurzreferenz [8] alltägliche Problemstellungen und Lösungsansätze.

Abstract

This project report presents an explanation of the version control Git. In addition to an explanation of the most important functions using an example project, various proven workflows are introduced. Thus, the work includes the local installation of Git, as well as guidelines for the use of Git in large scaling software projects. In order to create a foundation for all applications based on Git, the explanations refer to the cross-application usable command line. For later look up, chapter 8 provides solutions for everyday problems with Git.

Inhalt

1	Einleitung	1
2	Problemstellung	2
3	Versionskontrolle Git	4
3.1	Historie	4
3.2	Grundlegendes Prinzip	5
3.3	Installationsanleitung	7
3.3.1	Installation unter Linux	7
3.3.2	Installation unter Windows.....	8
4	Durchführung eines beispielhaften Projekts.....	9
4.1	Aufbau des Projekts.....	9
4.2	Erstellung eines Projektarchives.....	9
4.2.1	Git init.....	9
4.2.2	Git config	11
4.2.3	Git clone	12
4.3	Änderungen verwalten.....	12
4.3.1	Die Datei .gitignore	13
4.3.2	Funktionsweise des Indexes (git status, git add)	13
4.3.3	Git Commit.....	15
4.3.4	Git log	16
4.4	Umschreiben der Historie	18
4.4.1	Interaktiver Rebase	18
4.5	Arbeiten mit mehreren Zweigen.....	21
4.5.1	Git stash (Zwischenspeicher)	21
4.5.2	Git worktree.....	22
4.5.3	Git branch	23
4.5.4	Git checkout.....	24
4.5.5	Gitk	26
4.6	Zusammenführung von Zweigen.....	27
4.6.1	Git merge	27
4.6.2	Partielle Zusammenführung von Zweigen	29
4.6.3	Git rebase.....	30
4.6.4	Behebung von Konflikten.....	32
4.6.5	Git mergetool.....	33
4.7	Synchronisierung von Archiven	36
4.7.1	Git remote	36

4.7.2	Git push	37
4.7.3	Git blame	39
4.7.4	Git fetch	39
4.7.5	Git pull	40
4.7.6	Git request-pull	41
4.8	Veröffentlichungen kennzeichnen	43
4.8.1	Git tag	43
4.8.2	Git show	44
4.9	Erstellung einer Änderungsdatei	44
4.9.1	Git diff	44
4.9.2	Git patch	45
4.10	Revidieren von Commits	46
4.10.1	Git revert	46
4.10.2	Git reset	47
4.10.3	Git reflog	48
4.11	Selektiv Änderungen durchführen	49
4.11.1	Git cherry-pick	49
4.11.2	Git cherry	50
5	Verbreitete Arbeitsabläufe	51
5.1	Branching Workflow (GitFlow)	51
5.2	Historischer Workflow (Linux Kernel)	53
5.3	Forking Workflow	55
5.4	Individuelle Projektanpassungen	57
6	Interne Dateiverwaltung von Git	60
6.1	Packfiles	61
6.2	Git LFS	61
7	Zusammenfassung und Ausblick	63
8	Kurzreferenz	64
9	Literaturverzeichnis	68
	Abbildungsverzeichnis	71
	Glossar	73

1 Einleitung

Die Anfangszeiten der Programmierung waren von Genies geprägt, welche eigenständig Software entwickelten. Diese kannten jede Zeile ihrer Programme und glänzten durch ihr großes Talent. Mit der Zeit wurde Software aber immer komplexer und konnte nicht mehr von einer einzelnen Person gestemmt werden. Daraufhin wurde Software im Team entwickelt. Die Arbeit im Team verursacht aber neue Probleme, die durch eine effiziente Versionskontrolle behoben werden können.

Das Thema der Projektarbeit umfasst die Versionskontrolle mit Git. Das Hauptziel der Arbeit ist die Erklärung von Git anhand eines Beispiels. Hierbei ist der Fokus auf die Kommandozeile gerichtet, um Betriebssystem und Programmübergreifend zu sein. Da Git sehr umfangreich ist, wird sich innerhalb der Projektarbeit nur auf die wichtigsten Kommandos und Funktionen konzentriert. Mit diesen Grundlagen können übergestellte Themen behandelt werden, welche unter anderem die verschiedenen Arbeitsabläufe beleuchtet. Diese wurden über die Zeit durch unterschiedliche Projekte und Organisationen erarbeitet und im Laufe der Zeit verfeinert. Auch wird ein Blick unter die Haube gewagt und die Dateispeicherung von Git beschrieben. Damit soll die Arbeit den Einstieg in die Versionskontrolle Git erleichtern, setzt allerdings bereits vorhandenes Wissen zum Thema Versionsverwaltung und Konfigurationsmanagement voraus, da der Fokus hierbei hauptsächlich auf der Arbeit mit der Versionskontrolle Git liegt. Somit wird gezeigt, wie die Anforderungen an eine heutige Versionskontrolle in Git bewerkstelligt werden.

2 Problemstellung

Die wohl einfachste Art der Versionskontrolle ist die Speicherung der zu bearbeiteten Daten zu verschiedenen Zeitpunkten (siehe Abbildung 1). Auch können weitere Modifikationen an diesen durchgeführt werden und unterschiedliche Version koexistieren. Allerdings unterstützt diese rudimentäre Methodik nicht die Anforderungen und Lösungen, welche eine effiziente Software Entwicklung mit sich bringt. An dieser Stelle wurden komplexe Programme entwickelt, um Softwareprojekte bestmöglich zu unterstützen und die Dateispeicherung und Verwaltung zu vereinfachen.

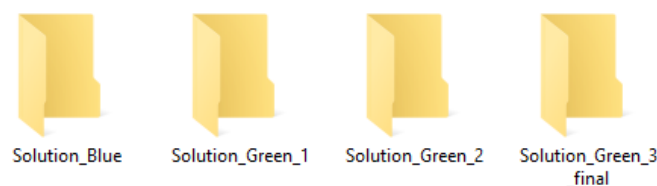


Abbildung 1: Versionsverwaltung anhand einfacher Datenablage

Um die Lösungsansätze, welche die Versionsverwaltung Git bietet, auch nutzen zu können, bedarf es Wissen in zwei größeren Themengebieten. Einerseits müssen die Kommandos und Befehle bekannt sein, um mit diesen die gewollten Änderungen durchzuführen. Da Git für ein Ziel oftmals mehrere Wege bietet, ist eine Kenntnis dieser unterschiedlichen Wege und dessen spezifischen Vor- und Nachteile ein elementarer Grundstein. Hierfür bietet sich eine Erklärung anhand passender Beispiele an, da der Befehl im Kontext einer des Öfteren vorkommenden Anwendung, einfacher einzuordnen ist.

Das zweite Gebiet handelt von der Arbeit rund um Git. Der Überbegriff lautet hierbei Konfigurationsmanagement. Ein Projekt kann sehr umständlich zu bearbeiten sein, falls keine projektspezifischen Anpassungen an die Handhabung von Git durchgeführt wurden. Es ist wichtig, sich der individuellen Forderungen an die Versionskontrolle zu vergewissern und im Voraus einen Weg zur Verwirklichung dieser zu erarbeiten. Nur dadurch kann das volle Potenzial der Versionsverwaltung ausgeschöpft werden, da später eine grundlegende Änderung an der Arbeitsweise oder dem Aufbau nur erschwert durchführbar ist und zu unübersichtlichen Strukturen und verlassenen Daten führen kann. Das erste Problem, welches eine Versionsverwaltung abarbeitet, betrifft die Strukturierung des Projektes und dessen Ablage. Je nach Projekt sollen Dateien verwaltet werden, welche eine klassische Versionskontrolle zur Softwareentwicklung nicht optimal handhabt. In solch einer Situation sollte eine Abwägung oder besondere Vorkehrungen stattfinden. Ein weiteres Problem, ist das gleichzeitige Arbeiten an einer Datei, sowie das parallele Arbeiten an verschiedenen Lösungen. An dieser Stelle erhalten

strukturierte und isolierte Aufteilungen enorme Wichtigkeit, um die produktive Zeit der Entwickler zu fördern. Oftmals kann Git verschiedene Kundenwünsche nicht erfüllen, da die Versionskontrolle ursprünglich für einen bestimmten Einsatz erschaffen wurde und nicht alle Funktionen der konkurrierenden Lösungen somit besitzt, da diese in dem grundlegenden Projekt nicht von Nöten waren. Aus diesem Grund fehlt zum Beispiel eine Zugriffskontrolle innerhalb eines einzelnen Projekts. Die Limitierungen von Git sind daher zu bedenken, und sollten vor der Integration der Versionskontrolle Git in ein Projekt bedacht werden. Weitere Probleme liegen bei Aktionen, die in der Vergangenheit passiert sind. Diese sind zum einem die Wiederherstellung von früheren Versionen, sowie die Dokumentierung von Änderungen und die Klärung von Verantwortlichkeiten. Diese Bedingungen sind eng geknüpft mit der Handhabung des Projekts während der Laufzeit, da Git mehrere Auslegungen besitzt. Als prominentes Beispiel kann hier ein ständiger Kompromiss zwischen einer detaillierten Historie oder einer schnellen und übersichtlichen Historie angeführt werden. Das bedeutet, Git kann je nach Bedarf des Anwenders unterschiedliche Forderungen priorisieren, jedoch werden dadurch andere Eigenschaften von Git eingeschränkt, beziehungsweise weitere Forderungen können nicht wie gewünscht erfüllt werden. Nicht immer können alle Anforderungen im gleichen Maße koexistieren. Aus diesem Grund ist eine Planung vor Start eines Projekts unverzichtbar. Gepaart mit Wissen über die Arbeitsweisen, welche Git besitzt und viele frühere Projekte erfolgreich unterstützt haben. Diese können als Grundlage für eine eigene Arbeitsweise dienen und falls benötigt auf das eigene Projekt und dessen Umgebung spezifisch angepasst werden.

3 Versionskontrolle Git

3.1 Historie

Der Ursprung von Git ist stark verbunden mit der Arbeit an dem Betriebssystem Linux. Änderungen an Linux wurden anfangs über Patches verteilt, später aber mit der Versionskontrolle BitKeeper verwaltet. Nach Lizenz Änderungen von BitKeeper, welche zur Folge hatten, dass die Software nicht mehr kostenlos war, wurde die Zusammenarbeit mit BitKeeper beendet. Daraufhin begann im April 2005 die Entwicklung einer eigenen Software. Bei dieser war Linus Torvalds, der Schöpfer von Linux, maßgeblich beteiligt. Im Juli 2005 wurde die Rolle des Hauptverwalters an Junio Hamio übergeben. Git erhält bis heute immer noch regelmäßige Änderungen (1). Aufgrund dieser Entstehungsgeschichte wurde Git in vielen Aspekten geprägt, welche die Verwendung von Software zur Versionierung bei Großprojekten erleichtert. Auch deshalb legt Git einen großen Wert auf Teamarbeit und die effiziente Nutzung von Entwicklerzeit. Hierbei wurde auf Schnelligkeit der Synchronisierungen und Zusammenführungen geachtet, sowie einer dezentralen Ablage, damit der Entwickler nicht abhängig von gesperrten Dateien durch die gleichzeitige Bearbeitung anderer ist. Mit diesen Vorteilen konnte Git schon früh die steigenden Bedürfnisse von Anwendern abfangen und dadurch an Beliebtheit unter Entwicklern gewinnen. Die Hauptbranche ist dabei der Open Source Bereich, in der eine Linux ähnliche Philosophie betrieben wird.

3.2 Grundlegendes Prinzip

Da in Kapitel 4, den Erklärungen der einzelnen Kommandos, viele andere Kommandos von Git referenziert werden oder zum Verständnis benötigt werden, ist es hilfreich zuvor einen kurzen Einblick in die grundsätzliche Struktur Gits zu erhalten. Ein Projekt, welches durch Git versionskontrolliert sein soll, besteht grundsätzlich aus einem Verzeichnis. Dieses beinhaltet alle Daten, in Form von Dateien, des Projekts, sowie die von Git benötigten Daten zur Konfiguration und Verwaltung, dargestellt als ein Projektverzeichnis in Abbildung 2.

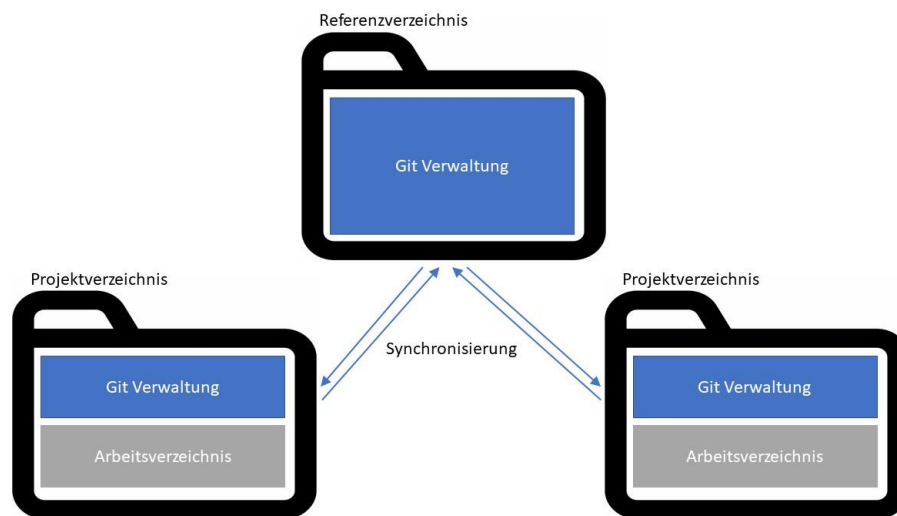


Abbildung 2: Synchronisierung von Projektarchiven

Untereinander müssen diese Verzeichnisse synchronisiert werden, damit mehrere Personen daran arbeiten können. Dies folgt aus dem dezentralen Konzept, welches Git verfolgt. Somit besitzt jede Person, welche an dem Projekt arbeitet, eine komplette Kopie des gesamten Projektverzeichnis. Im Allgemeinen wird ein einziges Referenzarchiv verwendet um die Synchronisierung zu Vereinfachen. Diese Rolle stellt das mittlere Archiv in Abbildung 2 dar. Die Synchronisierung wird aufgrund des dezentralen Ansatzes der Versionsverwaltung Git benötigt. Da Git nach dem Prinzip "Copy-Modify-Merge" arbeitet, eine Datei somit zur gleichen Zeit bearbeitet werden kann, gibt es unterschiedliche Stände des Projekts. Um diese wiederum zu vereinen, werden diese Zusammgeführt und als aktueller Stand auf dem Referenzarchiv für alle am Projekt arbeitende Personen veröffentlicht. Das Archiv wird über die Zeit als Baum dargestellt, wobei der Baum aus einzelnen sogenannten Commits besteht. Ein Commit ist ein Schnappschuss des gesamten Archivs, und wird wiederum in den Git Verwaltungsdaten abgelegt. Die Baumstruktur entsteht aus Abzweigung zwischen Commits. Diese Zweige werden verwendet, um die Entwicklung unabhängig voneinander und oder gleichzeitig voran zu treiben. Zum Beispiel arbeitet ein Team an einem Zweig, während einer besonderen Funktion in einem weiteren Zweig entwickelt wird. Die Gründe für Abzwei-

gungen sind vielseitig. Um einen Schnappschuss zu erstellen werden zuvor Änderungen gegenüber dem letzten Stand markiert, welche diesem neuen Schnappschuss beigefügt werden sollen. Markierte Änderungen landen im Index. Danach werden die Dateien im Index durch ein Commit archiviert und der Historie angehängen. Dieser Stand ist nun die lokale Referenz, gegenüber sich das Arbeitsverzeichnis abgleicht. Der angesprochene Vorgang ist in Abbildung 3 zu sehen. Besitzt der Commit einen wichtigen Stand kann dieser zusätzlich ein Etikett besitzen, mit welchem der Commit schnell und einfach referenziert werden kann.

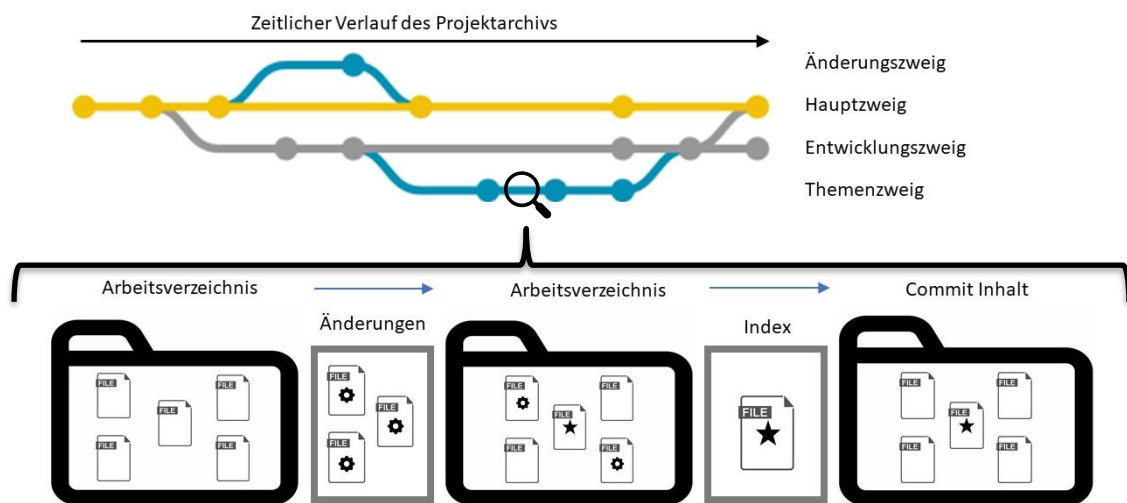


Abbildung 3: Grundlegender Commit Vorgang

3.3 Installationsanleitung

Um mit Git zu arbeiten, benötigt man entweder die Kommandozeilen Version von Git, dessen Installation nachfolgend erklärt wird, oder man installiert eine der vielen grafischen Oberfläche zur vereinfachten Verwaltung, welche Git mitliefern. Um die Dokumentation möglichst langwährend und übergreifend zu gestalten, wird Git im Folgenden anhand der Kommandozeile erklärt. Dies folgt aus sehr unterschiedlich aussehenden und handhabenden Oberflächen und Anwendungen, welche somit nicht untereinander unbedingt übertragbar sind. Durch die Erlernung der elementaren Grundlagen anhand der Kommandozeile, kann ein Verständnis aufgebaut werden, welche die Zurechtfindung in Git Anwendungsprogrammen erleichtert.

3.3.1 Installation unter Linux

Da Git zum Großteil in den Paketlisten der verschiedenen Linux Distributionen vorkommt, kann Git über das Kommando *apt-get* heruntergeladen und installiert werden. Dazu sollte zuerst die Referenzen, beziehungsweise die Paketliste von *apt-get* erneuert werden mit dem Kommando: *sudo apt-get update*.

```
labadmin@eitlinux ~ $ sudo apt-get update
[sudo] Passwort für labadmin:
OK:1 http://archive.canonical.com/ubuntu bionic InRelease
OK:2 http://security.ubuntu.com/ubuntu bionic-security InRelease
OK:3 http://archive.ubuntu.com/ubuntu bionic InRelease
OK:4 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
Paketlisten werden gelesen... Fertig
labadmin@eitlinux ~ $ sudo apt-get install git
```

Abbildung 4: Git Installation unter Linux mit *apt-get*

Ist dies gemäß dem Kommando in Abbildung 4 geschehen, kann folgend Git nun über: *sudo apt-get install git* installiert werden. Nachdem die benötigten Daten heruntergeladen sind und die Installation vollendet ist, kann die installierte Version von Git mit *git --version* überprüft werden.

3.3.2 Installation unter Windows

Unter dem Betriebssystem Windows muss zuerst eine Executable heruntergeladen werden, welche bei Ausführung die notwendigen Dateien und Einstellungen zur Verwendung von Git installiert. Hier können durch den Benutzer schon gewünschte Funktionen aktiviert oder entfernt werden. Hier sollten als Steuerzeichen (CRLF) die Standard Option verwendet werden, sodass Dateien und Commits die gleiche Formatierung erhalten bei Bearbeitung auf unterschiedlichen Betriebssystemen. Des Weiteren sollte die PATH-Umgebungsvariable hinzugefügt werden, damit Git auch aus der Kommandozeile ansprechbar ist und man nicht auf die Git eigene Kommandozeile Git Bash angewiesen ist. Daher empfiehlt sich bei dieser Abfrage die zweite Option, siehe Abbildung 5.

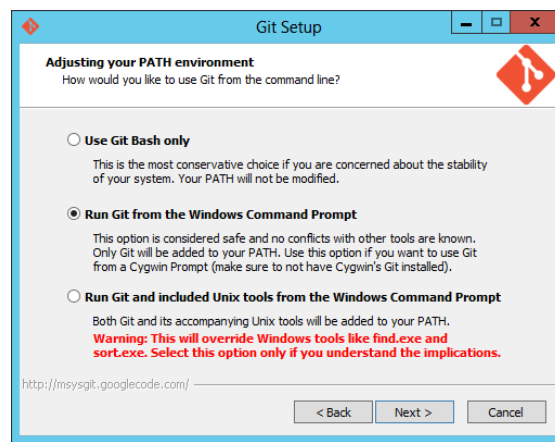


Abbildung 5: Einstellung zur PATH Variable unter Windows

Diese Option sollte standardmäßig ausgewählt sein. Ist die Installation beendet, müsste Git in der Kommandozeile ansprechbar sein. Um sich zu vergewissern kann in der Kommandozeile ***git --version*** ausgeführt werden. Dies sollte die Version der zuvor durchgeführten Installation wiedergeben.

4 Durchführung eines beispielhaften Projekts

4.1 Aufbau des Projekts

Um die wesentlichen Befehle von Git zu behandeln und zu erklären, werden die jeweiligen Kommandos anhand des nachfolgenden Projekts aufgezeigt. Dabei wird ein Kommando in dessen sinnvollen Umfang erklärt, sobald dies verwendet wird. Falls die Funktion eines Kommandos oder dessen Handhabung ungewiss ist, kann über *git* `<Kommando> --help` die Git interne Dokumentation zu eben diesem Kommando aufgerufen werden.

4.2 Erstellung eines Projektarchives

Um nun mit einem Projekt starten zu können, benötigt man ein Projektarchiv (Repository), welches die projektbezogenen Dateien beherbergt. Das Projektarchiv kann entweder lokal erstellt werden oder es wird von einem bereits bestehenden Archiv kopiert. Dies ist der Fall, wenn an einem bestehenden Projekt mitgearbeitet wird. Will man jedoch ein neues Projekt starten, welches die Versionskontrolle Git nutzt, wie in diesem Fallbeispiel, geschieht dies über den Befehl *init*.

4.2.1 Git init

Das Kommando *git init* fügt dem aktuellen Verzeichnis der Dateiverwaltung ein Unterverzeichnis mit dem Namen: ".git" hinzu. Dadurch wird dieses Verzeichnis von Git und anderen Anwendungsprogrammen als ein Git Projektarchiv angesehen. Ohne zuerst mit der Konsole in das spätere Projektverzeichnis zu wechseln, kann auch der Pfad nach *git init* angegeben werden, um das Projekt an einem Ort zu initialisieren. Falls das übergebene Verzeichnis nicht existiert, wird dieses dahingehend erstellt. In dem erstellten Unterverzeichnis speichert Git alle relevanten Daten des Projekts. Dadurch ist es möglich alleinig über diesen Ordner ein Projekt zu kopieren und in seiner Gänze wiederherzustellen. Schaut man sich dieses Verzeichnis an, wie in Abbildung 6 zu sehen, besitzt der Ordner ".git" wiederum mehrere Unterverzeichnisse und Dateien. Es folgt eine kurze Erklärung der wichtigsten Elemente.

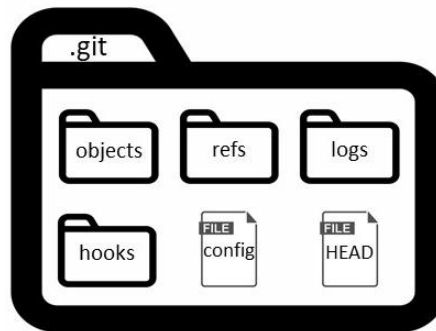


Abbildung 6: Darstellung des .git Ordners (35)

Die Dateien, welche über die Versionsverwaltung verwaltet werden, befinden sich komprimiert und verschlüsselt in dem Ordner *objects*. Im Kapitel Interne Dateiverwaltung von Git, wird hierauf zurückgekommen und erklärt, wie Git die Dateien effizient speichert. Der Ordner *refs* enthält Verweise auf die verschiedenen Zweige (*branches*) und Etiketten (*tags*), welche das Projekt besitzt. Ein weiteres Verzeichnis *logs* enthält wie der Name verspricht, Information zu der Commit Historie, sowie der Historie der einzelnen Zweige und des Arbeitsverzeichnisses. Shell-Skripte, die bei Git Kommandos ausgeführt werden sollen, befinden sich in dem Ordner *hooks*. Hier kann zum Beispiel eine Commit Nachricht vordefiniert werden. Die *config* Datei enthält Information und Einstellungen für das Projektarchiv. Hier wird vermerkt, welcher Autor an diesem Projektarchiv arbeitet, welcher Texteditor standardmäßig geöffnet wird, sowie Aliase und Links zu weiteren Projektverzeichnisse.

Erstellt man ein Referenzarchiv (*shared repository*), welches als reines Archiv zur Synchronisation genutzt wird, fügt man dem Kommando das Schlüsselwort **--bare** hinzu. Dies führt dazu, dass das Archiv keine Arbeitskopie erhält, sodass man auf dem zentralen Projektarchiv keine Dateien editieren oder Commits erstellen kann. Um nun an diesem Projekt ohne Arbeitsverzeichnis zu arbeiten muss dieses zuerst mit *git clone* geklont werden. Dieser Klon besitzt ein Arbeitsverzeichnis und dadurch können Änderungen an dem Projekt werden (2).

Um das Projektarchiv für das verwendete Beispielprojekt zu erstellen, führt man gemäß Abbildung 7 im zuvor erstellen Verzeichnis, hier "Git", den Befehl **git init** aus.

```
labadmin@eitlinux ~/Git $ git init
Leeres Git-Repository in /home/labadmin/Git/.git/ initialisiert
labadmin@eitlinux ~/Git $
```

Abbildung 7: Initialisierung eines Projektarchivs

Falls keine archivübergreifende Konfiguration existiert, das heißt keine Benutzerweite Git Informationen hinterlegt sind, muss man diese Git zuerst mitteilen. Diese Informationen werden dazu benötigt, damit Git weiß, welcher Entwickler die Änderungen lokal durchführt, sodass jeder erstellte Commit einem Entwickler zugewiesen werden kann.

Ohne diese Informationen verweigert Git die Erstellung eines Commits. Um die benötigten Informationen Git mitzuteilen wird ein weiterer Befehl mit dem Namen *config* ausgeführt.

4.2.2 Git config

Wird eine neues Projektarchiv erstellt oder auch nach einer lokalen Installation von Git, sollte man grundsätzliche Einstellungen vornehmen, um Git, beziehungsweise das vorliegende Projekt, nach seinen eigenen Wünschen zu konfigurieren. Als Erstes sollte man allerdings seine persönlichen Daten hinterlegen, damit bei der Zusammenarbeit mit anderen Nutzern, die Historie jederzeit einem Nutzer zuzuordnen ist. Dafür wird der *git config* Befehl benötigt. Mit dem Anhang *--global* werden die Einstellungen nicht nur für das aktuelle Archiv verwendet, sondern auch für alle anderen Archive des am Betriebssystem angemeldeten Benutzers. Hierfür werden mit *git config --global user.name "<Name>"*, beziehungsweise *user.email "<E-Mail>"*, die zwei wichtigsten Informationen gesetzt. Weitergehend kann der Standard Texteditor geändert werden mit: *core.editor*. Man kann die Einstellungen auch in der respektiven Datei, aufzurufen mit *git config --global --edit*, durchführen. Des Weiteren können Aliase erstellt werden, um individualisierte Kommandos zu erstellen oder bestehende Befehle abzukürzen. Dies geschieht grundsätzlich mit *git config alias.<Alias Name> <Git Kommando>*. Sollte das Kommando Leerzeichen beinhalten, werden auf Unix Systemen einzelne Anführungszeichen verwendet und auf Windows Systemen doppelte Anführungszeichen. Dies wird benötigt, um das Kommando zu gruppieren. Ein nicht Git Befehl benötigt ein Ausrufezeichen vor diesem. Um die bisherige Konfiguration einzusehen, wird *git config --list* verwendet (3).

Da beide Einstellungen bereits aufgrund vorheriger Projekte benutzerweit vorliegen, sieht die Konfiguration nach der Initialisierung des Projekts wie folgt aus:

```
labadmin@eitlinux ~/Downloads/gitrepo $ git config --list
user.email=          @hs-pforzheim.de
user.name=
core.editor=atom --wait
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

Abbildung 8: Einsicht in die bestehende Git Konfiguration

Ist ein Projektarchiv allerdings bereits vorhanden und liegt auf einem Hoster oder Server, muss zuerst dieses bestehende Projekt geklont werden, um danach lokal daran arbeiten zu können. Hierfür steht das Kommando *clone* zur Verfügung. Dies ist die zweite Methode ein Projektarchiv lokal anzulegen.

4.2.3 Git clone

Mit diesem Kommando wird ein neues Projektarchiv angelegt, jedoch als Vorlage ein bestehendes Archiv verwendet. Man klonet somit das bestehende Projektarchiv. Zusätzlich wird dem lokal erstellten Archiv direkt eine Verbindung zum ursprünglichen Projektarchiv beigelegt und unter dem Alias *origin* hinterlegt. So können ohne Umwege Änderungen am ursprünglichen Archiv verfolgt und in das eigene Archiv übernommen werden. Verwendet wird das Kommando mit ***git clone <remote Adresse> <lokales Verzeichnis>***. Als einfaches Beispiel wird in Abbildung 9 ein öffentliches Projektarchiv des Hosters GitHub.com geklont.

```
labadmin@eitlinux ~/Downloads $ git clone https://github.com/[REDACTED]/actiontest
Klone nach 'actiontest' ...
remote: Enumerating objects: 51, done.
remote: Counting objects: 100% (51/51), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 51 (delta 7), reused 24 (delta 0), pack-reused 0
Entpacke Objekte: 100% (51/51), Fertig.
labadmin@eitlinux ~/Downloads $
```

Abbildung 9: Klonen eines öffentlichen Projektarchivs von Github.com

Im Falle eines sogenannten privaten Archivs des Hosters *Github.com* kann ***https://<Benutzer>:<Passwort>@github.com/...*** verwendet werden. Die Übergabe von Anmeldedaten eines zugriffberechtigten Benutzers wird benötigt, da das Projekt durch eine Zugriffskontrolle geschützt ist. Übergibt man die benötigten Zugangsdaten nicht direkt, wird der Benutzername und das Passwort nachträglich verlangt, um auf das Projektarchiv zuzugreifen. Neben "https://" kann die Kommunikation zwischen den Archiven auch über "ssh://" oder "git://" ablaufen. Dafür müssen jedoch entsprechende Konfiguration, wie zum Beispiel der Austausch der SSH Schlüssel, zuvor erfolgen, um eine autorisierte Verbindung herzustellen.

4.3 Änderungen verwalten

Mit einem frisch angelegten Projektarchiv und elementaren Einstellung kann mit der Arbeit begonnen werden. Als erster Schritt des Beispielprojekts wird eine README Datei erstellt, um das bevorstehende Projekt kurz zu beschreiben. Zusätzlich dazu wird eine Datei mit dem Namen ".gitignore" erstellt wie in Abbildung 10 zu sehen. Diese vereinfacht die spätere Handhabung mit Dateien des Projekts.

```
labadmin@eitlinux ~/Git $ echo '*o \n geo \n *~' > .gitignore
```

Abbildung 10: Erzeugung der .gitignore Datei

4.3.1 Die Datei .gitignore

Benutzt man Dateien, welche explizit nicht in die Versionsverwaltung inkludiert werden sollen, kann eine Datei namens **".gitignore"** verwendet werden. Dateien und Verzeichnisse, welche in *.gitignore* zeilenweise vorkommen, werden automatisch von Git ausgeblendet und somit nicht dem Index vorgeschlagen bzw. dem nächsten Commit hinzugefügt. In Abbildung 11 zu sehen, werden im Beispielprojekt *object*-Dateien, temporäre Dateien, welche mit einer Tilde enden, sowie das gebaute Programm namens "geo" ausgeblendet.

```
labadmin@eitlinux ~/Git $ cat .gitignore
*.o
geo
*~
```

Abbildung 11: Darstellung der Datei .gitignore

Dieses Prinzip wird unter anderem häufig für Dateien benötigt, welche im Erstellungsprozess verwendet oder erstellt werden, da diese Dateien jeder Entwickler lokal erstellen kann. Dadurch wird die Versionsverwaltung nicht unnötig vergrößert und erhält zudem mehr Übersichtlichkeit. Des Weiteren können Dateien, welche zum Beispiel durch die personalisierte Entwicklungsumgebung des Entwicklers hinzukommen, ausgeschlossen werden.

Die zuvor erstellten Dateien "README" und ".gitignore" sollen nun der Versionsverwaltung hinzugefügt werden. Hierfür spielt das Arbeitsverzeichnis in Verbindungen mit dem Index (*Staging Area*) eine Rolle. Standardmäßig mit der Initialisierung eines Projektarchivs existiert bereits ein Zweig namens *master*.

4.3.2 Funktionsweise des Indexes (git status, git add)

Ist die lokale Arbeitskopie vorhanden, können nun Änderungen und neue Dateien hinzugefügt werden. Mit dem Befehl **git status** werden die derzeitigen Änderungen des Arbeitsverzeichnisses gegenüber dem letzten Stand des Zweiges angezeigt, zu sehen in Abbildung 12. Hier werden nun die zwei zuvor erstellten Dateien angezeigt:

```
labadmin@eitlinux ~/Git $ git status
Auf Branch master

Noch keine Commits

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

    .gitignore
    README

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
(benutzen Sie "git add" zum Versionieren)
```

Abbildung 12: Ausgabe des Befehls *git status*

Ist das Verzeichnis vor der Initialisierung von Git nicht leer, würden nun alle Dateien, welche das Verzeichnis besitzt, aufgelistet werden. In unserem Fall werden die beiden zuvor erstellten Dateien unter "Unversionierte Dateien" aufgeführt. Das bedeutet, diese Dateien sind noch nicht in der Versionsverwaltung vorhanden, beziehungsweise Git besitzt keinen älteren Stand dieser Dateien. Die beiden Dateien oder Änderungen an Dateien, falls diese bestehen, können nun dem Index hinzugefügt werden. Dazu wird der Befehl **git add <Datei>** benutzt. Hiermit können explizit Dateien ausgewählt werden, welche für den nächsten Commit vorgemerkt werden. Will man alle Veränderungen übernehmen reicht ein einfaches **git add .** oder **git add ***. Der Sternoperator bedient sich dabei der Funktion des Betriebssystems. Ausgeschlossen durch diesen sind Dateien, welche mit einem Punkt beginnen. Diese müssen explizit hinzugefügt werden. Der Benutzer kann durch den Index gezielt auswählen, welche Änderungen versionsverwaltet werden sollen. Dadurch ermöglicht man kleine und übersichtliche Commits, welche nur Änderungen beinhalten, welche im Projektarchiv oder im speziellen Commit benötigt werden. Man stelle sich eine größere Änderung vor, welche mehrere kleinere Module, unabhängig voneinander, beinhaltet. Anstatt die Änderungen als Gesamtpaket in einen Commit zu verpacken, kann man die Module einzeln dem Index hinzufügen und jeweils ein Commit erstellen. Zu sehen in Abbildung 13, werden nun die Dateien README und .gitignore dem Index hinzugefügt. Im Anschluss sieht man anhand des Status Kommandos, dass beide Dateien nun für den nächsten Commit vorgemerkt sind:

```
labadmin@eitlinux ~/Git $ git add .
labadmin@eitlinux ~/Git $ git status
Auf Branch master

Noch keine Commits

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

neue Datei:      .gitignore
neue Datei:      README
```

Abbildung 13: Dateien dem Index hinzufügen

Das Kommando bietet einen größeren Umfang. Um differenzierter, nicht nur Dateien, sondern auch einzelne Abschnitte oder Zeilen einer Datei hinzuzufügen, kann das **add** Kommando noch erweitert werden. Eine Möglichkeit ist **git add --interactive**. Hier werden nun mehrere Kommandos abgefragt, welche einzeln, nacheinander auf Dateien angewendet werden können. Mit dem Kommando **--patch** kann eine Datei weiter aufgesplittet werden, um nur einzelne Zeilen der ausgewählten Datei dem Index hinzuzufügen (4 p. 188ff.). Um dies nicht für alle veränderten Dateien durchzuführen, sondern für eine einzelne Datei, wird **git add -p <Datei>** verwendet. Soll eine zuvor versionsverwaltete Datei aus der Versionsverwaltung entfernt werden, wird sie dem **remove** Kommando übergeben. Der Aufruf lautet hierbei **git rm <Datei>**. Für dauerhafte oder wiederkeh-

rende nicht versionsverwaltete Dateien in der Arbeitskopie, ist es hilfreich eine **.gitignore** Datei wie zuvor in Kapitel 4.3.1 beschrieben anzulegen, beziehungsweise die Datei der **.gitignore** hinzuzufügen. Ist eine Datei dem Index fälschlicherweise hinzugefügt worden, kann diese Datei aus dem Index über **git reset <Datei>** wieder entfernt werden, ohne dabei Änderungen an der Datei zu verlieren. Soll die gesamte Datei zurückgesetzt werden, kann dem Befehl das Schlüsselwort **--hard** angehängen werden.

Als nächster Schritt soll der erste Commit getätigt werden um als initialer Schnappschuss des Archivs zu stehen. Diesem Commit sollen die beiden Dateien beigelegt sein.

4.3.3 Git Commit

Commits bilden die zentrale Rolle in der Versionsverwaltung Git. Ein Commit entspricht einem bestimmten Stand der Arbeitskopie. Je nach Konfigurationsmanagement und zu bearbeitenden Zweig haben Commits unterschiedliche Anforderungen. So kann ein Commit in einem Korrektur Zweig aus einem einzelnen veränderten Buchstaben bestehen, oder ein Commit entspricht einem neuen Feature auf einem Entwicklungszweig. In Kapitel 5 wird deutlich wofür ein Commit auf einem sogenannten Themenzweig stehen kann. Ein Commit enthält neben dem Abbild der aktuellen Arbeitskopie, sofern alle Veränderungen der Arbeitskopie dem Index hinzugefügt wurden, auch eine Commit Nachricht. Diese sollte eine möglichst kurze, aber dennoch aussagefähige Zusammenfassung der Änderungen beinhalten. Die Commit Nachricht ist neben Etiketten die schnellste Möglichkeit, Änderungen zu finden. Für eine schnelle Übersicht und vereinfachte Zusammenarbeit mit anderen Personen an dem Projekt, sollte man sich daher ein Schema für die Commit Nachricht überlegen, welche Projektweit gilt und angewendet wird. Ein Commit verweist auch immer auf seinen Vorfahren. Anhand diesem kann festgestellt werden, woher der Commit abstammt, beziehungsweise, welche Änderungen im Vergleich zu seinem Vorfahren einfließen. Durch diese Verknüpfung werden die Commits aneinanderreihen und man erhält damit die grafische Darstellung einer Baumstruktur:



Abbildung 14: Darstellung von Commits über den zeitlichen Verlauf

Jeder neue Commit wird an dessen vorhergehenden angehängt und erweitert die Aneinanderreihung. Wird ein neuer Zweig erstellt und beide Zweige besitzen einen Commit, zeigen beide auf den gleichen Vorgänger, es entsteht eine Abzweigung. Im Gegensatz

dazu, besitzt ein Commit einer Zusammenführung mehrere Vorgänger. Durch diese Aneinanderreihung an Commits bildet Git eine lückenlose Historie bis zum Ursprungsstands des Projektarchivs. Commits werden intern durch ihre Prüfsumme unterschieden und anhand dieser auch gespeichert. Um auf bestimmte Commits zu verweisen wird dafür oftmals der Anfang ihrer SHA1-Prüfsumme verwendet. Ein Commit entsteht über den Befehl **git commit -m '<Nachricht>'** und speichert somit das gesamte Arbeitsverzeichnis, ausgenommen Änderungen, welche nicht dem Index hinzugefügt wurden. Falls man den vorherigen Commit noch einmal überarbeiten will hilft das Schlüsselwort **--amend**. Dadurch kann zum Beispiel die Commit Nachricht überarbeitet werden, sofern der aktuelle Index der Arbeitskopie noch unverändert ist. Hat man zusätzlich vergessen eine Änderung dem Commit hinzuzufügen, kann man dies dem Korrektur Commit hinzufügen und über **-m "<Commit Nachricht>"** die Nachricht korrigieren. Allerdings überschreibt der neue Commit den zuvor erstellten Commit, wodurch alle Referenzen auf diesen verloren gehen. Daher sollte man **--amend** nur verwenden, sofern keine andere Partei auf diesen aufgebaut haben kann. Die Faustregel lautet: Kein **--amend** auf veröffentlichte Commits. Veröffentlicht heißt hierbei über das lokale Projektarchiv hinaus (5).

Dem Projekt wird nun der initiale Commit hinzugefügt. Wie in Abbildung 15 zu sehen, meldet Git zurück, auf welchem Zweig der Commit erstellt wurde, sowie den Anfang der Checksumme und die Commit Nachricht. Zudem sieht man, dass die zwei Dateien neu hinzugefügt wurden und insgesamt durch diese Dateien 4 Zeilen hinzukamen. Die Zeileninformation kann nur Dateien entnommen werden, welche diese Informationen Git bereitstellen. Binär Dateien sind somit davon ausgenommen.

```
labadmin@geitlinux ~/Git $ git commit -m 'Initial Commit, added README and .gitignore'
[master (Basis-Commit) 7ce45dd] Initial Commit, added README and .gitignore
2 files changed, 4 insertions(+)
create mode 100644 .gitignore
create mode 100644 README
```

Abbildung 15: Erstellung des ersten Commits

Zur Überprüfung des gerade erstellten Commits, kann nachfolgend die Historie des Projektarchivs eingesehen werden. Hier sollte der Commit aufgeführt werden. Um die Historie aufzurufen, verwendet man das Kommando *log*.

4.3.4 Git log

Um einen Überblick über den aktuellen Stand des Zweiges zu erhalten, kann eine Übersicht der Commits mit **git log** aufgerufen werden. In der Ausgabe sieht man alle Commits des aktuell zu bearbeitendem Zweig mit Information zum Autor, Datum und Prüfsumme des jeweiligen Commits. Mit dem Schlüsselwort **--all** werden die Commits aller Zweige des Projektarchivs in zeitlicher Reihenfolge gelistet. In Kurzform kann auch **git shortlog** benutzt werden. Dadurch wird nur die Commit Nachricht, gruppiert nach dem

Autor, angezeigt. Um zusätzlich in der Ausgabe den Anfang der jeweiligen Prüfsumme zu erhalten kann **git log --oneline** verwendet werden. Ein weiteres nützliches Werkzeug bietet der Anhang **--follow <Datei>**. Hier werden alle Änderungen an der übergebenen Datei mit dem jeweiligen Commit ausgegeben. Mit **--grep '<Suche>'** werden nur Commits mit passender Commit Nachricht angezeigt. Um auch innerhalb der Änderungen der jeweiligen Commits zu suchen, wird **-S<Suche>** verwendet. Mit **-<Zahl>** kann die Ausgabe auf eine bestimmte Anzahl an Commits reduziert werden. Zur Orientierung der jeweiligen Commits kann zusätzlich **--graph** übergeben werden. Dadurch entsteht an den ersten Stellen ein Baum, welcher das Repository widerspiegelt. Dies hilft dabei die Commits schneller den Zweigen zuzuordnen und die Reihenfolge der Commits einordnen zu können. Um sehen zu können, ob Etiketten oder Zweige dem jeweiligen Commit zugeordnet sind, wird **--decorate** verwendet (4 p. 26ff.).

Führen wir nun, den in Abbildung 16 gezeigten Befehl **git log** aus, wird der bisher einzige Commit angezeigt. Neben der Prüfsumme wird außerdem "Head -> master" angezeigt. Daraus kann entnommen werden, dass der Commit die aktuelle Grundlage der Arbeitskopie ist und dem Zweig *master* zugeordnet ist. HEAD zeigt immer auf den Commit, welcher die Grundlage der aktuellen Arbeitskopie ist. Dies folgt aus Funktionalität, dass Git die Arbeitskopie ersetzt, sobald sich der HEAD Zeiger ändert.

```
labadmin@eitlinux ~/Git $ git log
commit 7ce45dddfbef0e28c21f0d28f5f989738d3f6ab5 (HEAD -> master)
Author: @hs-pforzheim.de>
Date: Tue Dec 24 09:51:27 2019 +0100

    Initial Commit, added README and .gitignore
```

Abbildung 16: Ausgabe des Befehls **git log**

Spulen wir einige Commits und Änderungen vor, sieht das Projektarchiv nun wie in Abbildung 17 aus. Hierbei wird **--oneline** verwendet, um einen schnellen Überblick zu erhalten. Metadaten wie Autor und Datum werden nicht angezeigt.

```
labadmin@eitlinux ~/Git $ git log --oneline
4cefc69 (HEAD -> master) Added depend file for makefile to work
752b82c Added makefile
7edef89 Added main.cpp
7ce45dd Initial Commit, added README and .gitignore
```

Abbildung 17: Ausgabe von **git log** in Zeilenform

Es fällt dem Entwickler auf, dass die 3 nachfolgenden Commits zusammenhängen und somit in einem einzelnen Commit mehr Sinn ergeben. In diesem Fall ist durch die Aufteilung auf mehrere Commits eine Erstellung und nachfolgende Ausführung des Programms erst ab dem letzten Commit möglich, womit der Erstellungsprozess mit *make* lauffähig ist. Grundsätzlich sollte man sich vornehmen, sofern möglich, bei jedem Commit einen ausführbaren Stand zu besitzen. Daher würde es auch sinnvoll sein, die

drei gefolgten Commits zu einem Commit zu verschmelzen. Auch dies ist möglich in Git mit einer Unterfunktion des Kommandos *rebase*.

4.4 Umschreiben der Historie

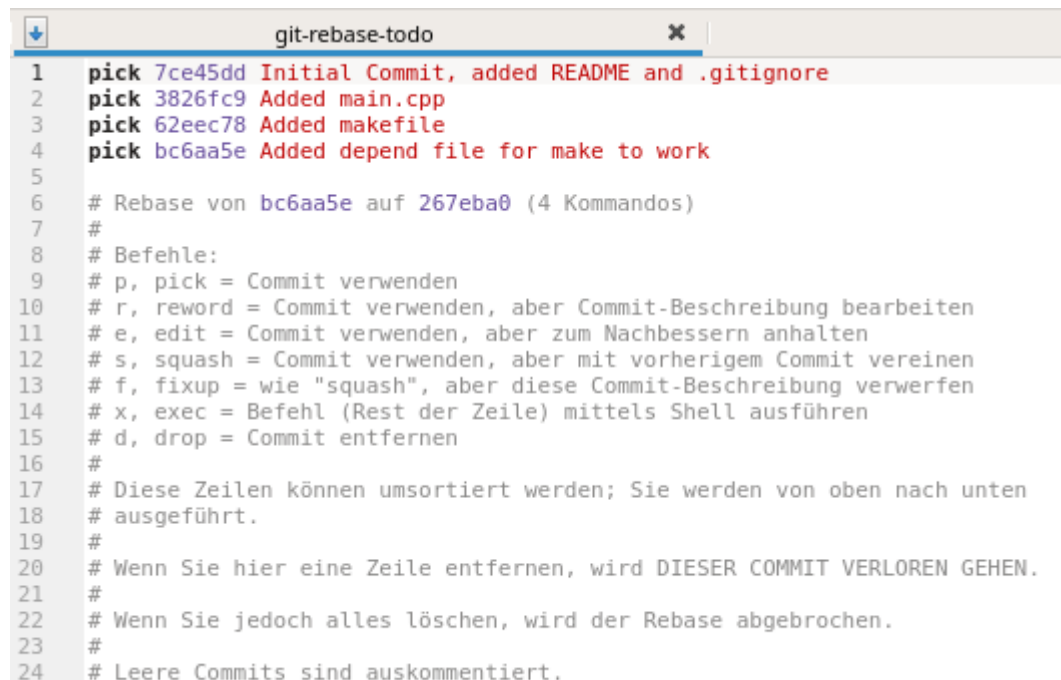
Trotz der Intention einer Versionsverwaltung, eine exakte Historie des Projekts zu bieten ist es in einigen Situationen sinnvoll, diese umzuschreiben. Hierdurch können unnötige Ballaste entfernt und übersichtlichere Strukturen geschaffen werden. Allerdings bietet eine Umschreibung auch die Gefahr, benötigte Daten zu entfernen oder aus ihrem Zusammenhang zu reißen. Zudem können dadurch unter Umständen Synchronisierungsfehler entstehen, da auf nicht mehr existierende Objekte verwiesen wird.

4.4.1 Interaktiver Rebase

Um mehrere Commits miteinander zu verschmelzen (squash) wird das Kommando *git rebase* mit dem Schlüsselwort *--i* beziehungsweise *-interactive* verwendet. Dadurch öffnet sich der interaktive Modus, bei dem der Nutzer jeden einzelnen Commit, welcher durch den Befehl betroffen ist, individuell verändern kann. So kann der Entwickler die Reihenfolge der Commits verändern, einzelne Commits löschen, Commits verschmelzen oder noch einmal bearbeiten. Dies geschieht in dem man die erste Spalte des automatisch geöffneten Textdokuments dementsprechend anpasst. Hierfür werden wiederum spezielle Schlüsselwörter benötigt. Um einen Commit zu löschen, wird dessen Zeile entfernt. Dies wird häufig verwendet, um mehrere Commits auf einmal zu verwerfen. Die Reihenfolge in der die Commits dargestellt werden, entspricht der Reihenfolge, mit der *rebase* die Commits auf den Vorfahren, das heißt, den ersten nicht betroffenen Commit, anwendet. Standardmäßig ist dies genau umgekehrt der Reihenfolge, in der die Commits in der Historie dargestellt werden. Dies folgt daraus, dass das Kommando *rebase* zuerst den zeitlich ersten Commit anwenden muss, um die ursprüngliche Reihenfolge wiederherzustellen. Das Schlüsselwort *reword* ermöglicht es die Commit Nachricht des entsprechenden Commits zu verändern. Mit *squash* verschmelzt sich der Commit mit dem vorhergehenden Commit. Dies wird häufig dazu benutzt, um die Baumstruktur übersichtlich zu halten, somit größere Funktionen oder Module zu gruppieren oder für Korrektur Commits, wobei diese Commits mit dem ursprünglichen Fehler behafteten Commit verschmolzen werden (6 p. Kap. 4). In dem Fall, dass ganze Entwicklungszweige zu einem Commit verschmelzen, kann der Anhang *--root* von Nutzen sein. Dadurch entfällt die Suche der Prüfsumme oder die Anzahl der Commits, welche von dem interaktiven Rebase betroffen sein sollen. Anstatt "HEAD~3", oder die Prüfsumme des Commits vor dem Rebase, in diesem Fall "7ce45dd", siehe Abbildung 17, kann dementsprechend *--root* übergeben werden.

Um die vorhandenen Commits im Beispielprojekt zu verschmelzen wird der interaktive Modus geöffnet mit dem Kommando: ***git rebase --interactive --root***.

Daraufhin öffnet sich der Git hinterlegte Editor, zu sehen in Abbildung 18. Hier wird nun wie bereits beschrieben die voranstehenden Schlüsselwörter je nach Anforderung abgeändert. In Beispielfall sollen die unteren 3 Commits zu einem einzigen Commit zusammengeführt werden. Dafür erhalten die Zeilen 3 und 4 das Schlüsselwort *squash* anstatt *pick*. Dadurch werden beide Commits dem Commit "Added main.cpp" von Zeile 2 hinzugefügt.



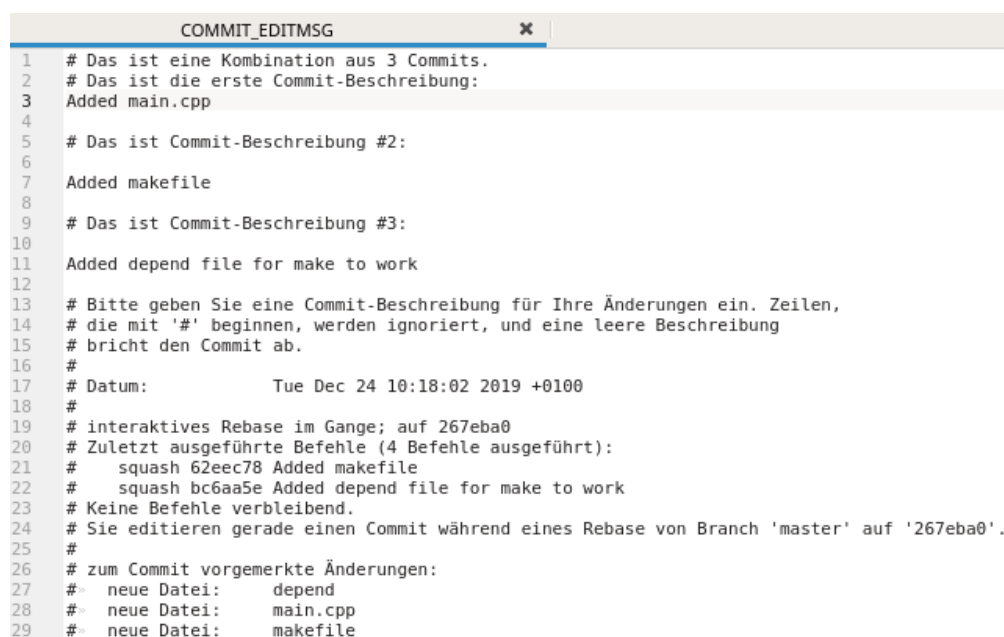
```

git-rebase-todo
1  pick 7ce45dd Initial Commit, added README and .gitignore
2  pick 3826fc9 Added main.cpp
3  pick 62eec78 Added makefile
4  pick bc6aa5e Added depend file for make to work
5
6  # Rebase von bc6aa5e auf 267eba0 (4 Kommandos)
7  #
8  # Befehle:
9  # p, pick = Commit verwenden
10 # r, reword = Commit verwenden, aber Commit-Beschreibung bearbeiten
11 # e, edit = Commit verwenden, aber zum Nachbessern anhalten
12 # s, squash = Commit verwenden, aber mit vorherigem Commit vereinen
13 # f, fixup = wie "squash", aber diese Commit-Beschreibung verwerfen
14 # x, exec = Befehl (Rest der Zeile) mittels Shell ausführen
15 # d, drop = Commit entfernen
16 #
17 # Diese Zeilen können umsortiert werden; Sie werden von oben nach unten
18 # ausgeführt.
19 #
20 # Wenn Sie hier eine Zeile entfernen, wird DIESER COMMIT VERLOREN GEHEN.
21 #
22 # Wenn Sie jedoch alles löschen, wird der Rebase abgebrochen.
23 #
24 # Leere Commits sind auskommentiert.

```

Abbildung 18: Auswahl der Commits beim interaktiven Rebase

Als nächster Schritt wird das Editierte gespeichert und der Editor geschlossen. Dadurch öffnet sich automatisch der Editor ein zweites Mal:



```

COMMIT_EDITMSG
1  # Das ist eine Kombination aus 3 Commits.
2  # Das ist die erste Commit-Beschreibung:
3  Added main.cpp
4
5  # Das ist Commit-Beschreibung #2:
6
7  Added makefile
8
9  # Das ist Commit-Beschreibung #3:
10
11 Added depend file for make to work
12
13 # Bitte geben Sie eine Commit-Beschreibung für Ihre Änderungen ein. Zeilen,
14 # die mit '#' beginnen, werden ignoriert, und eine leere Beschreibung
15 # bricht den Commit ab.
16 #
17 # Datum:          Tue Dec 24 10:18:02 2019 +0100
18 #
19 # interaktives Rebase im Gange; auf 267eba0
20 # Zuletzt ausgeführte Befehle (4 Befehle ausgeführt):
21 #   squash 62eec78 Added makefile
22 #   squash bc6aa5e Added depend file for make to work
23 # Keine Befehle verbleibend.
24 # Sie editieren gerade einen Commit während eines Rebase von Branch 'master' auf '267eba0'.
25 #
26 # zum Commit vorgemerkte Änderungen:
27 #> neue Datei:    depend
28 #> neue Datei:    main.cpp
29 #> neue Datei:    makefile

```

Abbildung 19: Bearbeitung der Commit Nachrichten beim Rebase

Hier wird die Commit Nachricht für den verschmolzenen Commit geändert. Zeilen mit einem vorangestellten "#" werden nicht beachtet, sie dienen als Orientierung. In diesem Beispiel werden die Nachrichten der verschmolzenen Commits verworfen und die ursprüngliche Nachricht von "Added main.cpp" erweitert. Ist die Datei wiederum gespeichert und wird geschlossen, führt Git den *Rebase* der Konfiguration entsprechend aus. Die Kommandozeile sieht wie folgt aus:

```
labadmin@eitlinux ~/Git $ git rebase --interactive --root
Icon theme "adwaita" not found.
Hinweis: Warte auf das Schließen der Datei durch Ihren Editor... Icon theme "adwaita" not found.
[losgelöster HEAD 07b1157] Added main.cpp and make support
Date: Tue Dec 24 10:18:02 2019 +0100
3 files changed, 39 insertions(+)
create mode 100755 depend
create mode 100755 main.cpp
create mode 100755 makefile
Erfolgreich Rebase ausgeführt und refs/heads/master aktualisiert.
```

Abbildung 20: Kommandozeilen Ausgabe nach erfolgreichem Rebase

Und dadurch wurde die Historie des Archivs von ehemals Abbildung 17 umgeschrieben zu folgender Ausgabe:

```
labadmin@eitlinux ~/Git $ git log --oneline
07b1157 (HEAD -> master) Added main.cpp and make support
4e62daf Initial Commit, added README and .gitignore
```

Abbildung 21: Historie nach erfolgreicher Verschmelzung

Anstelle von drei Commits, wobei lediglich letzterer ein ausführbares Programm erstellen konnte, werden die Änderungen nun in einem Commit festgehalten. Zudem besitzt das Archiv nach dem Rebase keine Commits mehr, welche nur Funktionsteile implementieren. Das Programm und die dazugehörigen Erstellungsdateien sind funktionell eine Gruppierung und werden daher auch zusammen der Versionsverwaltung hinzugefügt.

Es wird wieder weiter an dem Beispielprojekt entwickelt. Der Entwickler arbeitet an einer Erweiterung des Programms, will allerdings einen neuen Ansatz zwischenzeitlich verfolgen. Die bisherig durchgeführten Änderungen sind allerdings nicht auf einem Stand, welche einen sinnvollen Commit ergeben. Daher werden die Änderungen Zwischengespeichert, sodass der Entwickler eine frische Arbeitskopie besitzt. Dies ist von Nöten, damit der Entwickler einen neuen Ansatz verfolgen kann, welcher isoliert von Änderungen am Hauptzweig ist. Diese leere Arbeitskopie ist wichtig, da ein Zweigwechsel mit dieser Bedingung verknüpft ist. Später kann an den zwischengespeicherten Änderungen wieder weitergearbeitet werden. Die Funktionalität von temporärer Zwischenspeicherung bietet in Git die Funktion *stash*.

4.5 Arbeiten mit mehreren Zweigen

Anhand der steigenden Anforderungen an Programme, sind diese oftmals nur noch durch ein Team zu erledigen. Um die Zusammenarbeit zu vereinfachen und nicht durch die Arbeit anderer Personen blockiert zu werden, bietet Git die Erstellung mehrerer Zweige. Durch diese Aufteilung haben Entwickler eine Abstraktion und Isolation und können einfacher produktiv am Projekt arbeiten.

4.5.1 Git stash (Zwischenspeicher)

Der Stash stellt eine Art Zwischenspeicher da. Da die Arbeitskopie bei einem Zweigwechsel überschrieben wird, sollten alle Änderung in einen Commit überführt werden. Falls man den Zweig wechseln will, jedoch Änderungen besitzt, welche erst zu einem späteren Zeitpunkt der selbst festgelegten Commit Richtlinie entsprechen, kann man die Arbeitskopie Änderungen im Stash bis zur Rückkehr zum Zweig ablegen. Auch können darin Ideen oder Ansätze gespeichert werden, um sie nicht zu verlieren. Da ein erstellter Zwischenstand nicht zu einem Zweig gebunden ist, somit auf jeden Zweig angewendet werden kann, wäre eine weitere mögliche Anwendung darin Änderungen abzulegen, welche das Projekt schnell und temporär verändern. Zum Beispiel eine spezielles Debugging Funktion. Der einfachste Grund ist allerdings die vorhandene Arbeitskopie zu reinigen, vorheriges aber nicht zwangsweise verlieren zu wollen, falls man noch einmal darauf zurückkommen sollte. Der Befehl **git stash** speichert alle Veränderungen gegenüber dem Stand des letzten Commits des ausgewählten Zweiges. Mit der Option **-patch** kann wiederum nur ein Teil der Änderungen zwischengespeichert werden. Mit **-u** werden nicht versionsverwaltete Dateien in der Arbeitskopie mit einbezogen. Um eine Übersicht über die Zwischenspeicherungen zu erhalten, listet der Befehl **git stash list** alle bereits gespeicherten Zwischenstände auf. Um eine Einordnung der einzelnen Zwischenspeicherungen möglich zu machen, kann dem jeweiligen Zwischenstand eine Nachricht hinzugefügt werden. Dafür wird neben dem Befehl **save "<Nachricht>"** mit übergeben (7). Um die Zwischenspeicherung wieder auf die aktuelle Arbeitsmappe anzuwenden wird der Befehl **git stash pop** oder **git stash apply** verwendet. Im ersten Fall wird der Zwischenstand nach der Anwendung gelöscht, im Gegensatz zu der Nutzung von **apply**, welche den Zwischenstand beibehält. Mit **git stash show** werden die Änderungen des ersten Eintrags im Stash in Kurzform angezeigt. Mit dem Anhang **-p** werden die Änderungen in Langform, das heißt zeilenweise Änderungen, falls möglich, wiedergegeben. Um einen Zwischenstand zu löschen kann der Befehl **git stash drop <stash-id>** verwendet werden, wobei die ID zum Beispiel **"stash@{2}"** ist (4 p. 191ff.).

In dem Beispielprojekt speichert der Entwickler nun die bisherigen Änderungen an seiner Arbeitskopie ab unter dem Namen: "Work in Progress Linie".

```

labadmin@eitlinux ~/Git $ git stash save "Work in Progress Linie"
Speicherte Arbeitsverzeichnis und Index-Status On master: Work in Progress Linie
labadmin@eitlinux ~/Git $ git stash list
stash@{0}: On master: Work in Progress Linie

```

Abbildung 22: Erstellung einer Zwischenspeicherung

Durch die zurückgesetzte Arbeitsmappe kann der Entwickler einen Zweigwechsel durchführen. Wie bei fast jedem Problem, bietet Git mehrere Möglichkeiten zur Problemlösung. Git besitzt einen weiteren Weg mehrere Ansätze zu verfolgen, ohne Änderungen zu verlieren, welche nicht einem Commit beiliegen. Bei dem nachfolgenden Weg wird ein zusätzliches Arbeitsverzeichnis erstellt, in dem der Entwickler arbeiten kann. Dadurch entfällt die Notwendigkeit die aktuelle Arbeitsmappe zurückzusetzen.

4.5.2 Git worktree

Durch das Kommando *git worktree add <Pfad> <Zweig>* wird ein weitere Arbeitsverzeichnis angelegt. Dieses sollte zur besseren Übersicht außerhalb des Projektverzeichnisses erstellt werden. Durch die Erstellung eines weiteren Arbeitsverzeichnisses entfällt das Wechseln zwischen verschiedenen Zweigen im Hauptarbeitsverzeichnis und die damit einhergehenden Bedingungen. Dies ist zum Beispiel von Vorteil, falls bestehende Änderungen im aktuellen Arbeitsverzeichnis in verschiedenen Programmen verwendet werden oder eingerichtet sind. Auch kann das derzeitige Arbeitsverzeichnis durch Testdurchläufe temporär noch benötigt werden und kann dementsprechend noch nicht ausgetauscht werden. In Abbildung 23 zu sehen, kann im Hauptverzeichnis, sowie dem neu erstellten Arbeitsverzeichnis, wie gewohnt gearbeitet werden, ohne gegenseitige Abhängigkeiten zu besitzen, wie das Kommando *git status* deutlich macht. Um in dem neu erstellten Arbeitsverzeichnis zu arbeiten muss nach seiner Erstellung lediglich in dieses Verzeichnis navigiert werden.

```

labadmin@eitlinux ~/Git $ git worktree add ~/worktree feature
Bereite /home/labadmin/worktree vor (Identifikation worktree)
HEAD ist jetzt bei 28ae79a Added Abbildung implementation
labadmin@eitlinux ~/Git $ git status
Auf Branch master
Ihr Branch und 'origin/master' sind divergiert,
und haben jeweils 1 und 1 unterschiedliche Commits.
(benutzen Sie "git pull", um Ihren Branch mit dem Remote-Branch zusammenzuführen)

nichts zu committen, Arbeitsverzeichnis unverändert
labadmin@eitlinux ~/Git $ cd ~/worktree
labadmin@eitlinux ~/worktree $ git status
Auf Branch feature
Ihr Branch ist auf demselben Stand wie 'origin/feature'.

nichts zu committen, Arbeitsverzeichnis unverändert

```

Abbildung 23: Arbeit mit mehreren Arbeitsverzeichnissen

Werden Commits im zusätzlichen erstellten Arbeitsverzeichnis durchgeführt, sind diese ebenfalls im Hauptarchiv wiederzufinden, beziehungsweise werden wie gewohnt mitprotokolliert. Um Inkonsistenz zu verhindern, kann über das Kommando kein Arbeitsverzeichnis erstellt werden, falls der übergebene Zweig bereits in einem anderen Ar-

beitsverzeichnis existiert. Die aktuell erstellten Arbeitsverzeichnisse werden über den Befehl **git worktree list** eingesehen und können nachdem die gewünschten Änderungen erfolgt sind über **git worktree remove <Pfad>** wieder gelöscht werden. Das Hauptarbeitsverzeichnis ist von diesem Befehl ausgeschlossen. Auch können keine Verzeichnisse gelöscht werden, welche undokumentierte Änderungen beinhalten, sofern das Schlüsselwort **-f** nicht mit übergeben wird (8).

Diese Möglichkeit wird allerdings bei dem derzeitigen Fall nicht verwendet und der Zwischenspeicher wird dazu verwendet, die Arbeitskopie zurückzusetzen. Der Entwickler kann anhand der leeren Arbeitskopie einen Wechsel des Zweiges anstoßen.

4.5.3 Git branch

Zweige werden angewendet, um eine isolierte Entwicklungsumgebungen zu erhalten. So ist es von Vorteil eine Spielwiese des aktuellen Arbeitsstands zu verwenden, um Entwicklungen jeglicher Art vorzunehmen, ohne dabei in Gefahr zu geraten, instabilen Code im Hauptzweig zu erhalten. Auch kann gerade bei Projekten mit mehreren Entwicklern, unter Verwendung von Zweigen, sichergestellt werden, dass keine Komplikationen durch gegenseitige Abhängigkeiten entstehen. Zudem wird eine Synchronisierung zwischen den Änderungen mehrerer Personen auf ein Minimum reduziert, da ein Zweig bis zu seiner Zusammenführung nicht mit zeitgleichen Änderungen durch andere Personen interagieren muss. Aufgrund dieser Isolation lassen sich Projekte mit mehreren Entwicklern ohne größere Abstimmungen und Nachverfolgungen untereinander durchführen. Intern werden Zweige von Git nur als weitere Zeiger, welche auf erstellte Commits zeigen behandelt. Um einen neuen Zweig zu erstellen wird der Befehl **git branch <Name>** verwendet. Der neue Zweig führt von dem aktuellen aktiven Zweig ab. Git erkennt dies, da beide nachfolgenden Commits den gleichen Vorfahren besitzen. Wird ein Zweig nicht mehr benötigt lässt dieser sich mit **git branch -b** löschen. Dies ist nur möglich sofern alle Änderungen in andere Zweige übernommen wurden. Ist dies nicht der Fall und man möchte den Zweig trotzdem löschen, wird ein kapitalisiertes **-B** benötigt. Um alle Zweige des Projektarchivs anzusehen wird **git branch -l** verwendet. Soll der lokal erstellte Zweig auch für das Referenzarchiv übernommen werden, muss der Zweig ähnlich wie ein Commit synchronisiert werden. Für Git intern heißt das, dass der lokale Zweig einem Zweig des Referenzarchivs folgen soll. Es entsteht ein sogenannter *tracking branch*. Um dies zu gewährleisten gibt es mehrere Möglichkeiten. Am einfachsten kann **git push origin <Zweig>** verwendet werden. Der Alias *origin* steht hierbei für das Referenzarchiv und kann unter Umständen einen anderen Namen besitzen. Um sicherzustellen, dass der lokale Zweig auch unter dem gleichen Namen im Referenzarchiv erstellt wird oder explizit unter einem anderen Namen, kann anstatt **<Zweig>** auch **<Zweig>:<Zweigname der Referenz>** verwendet werden. Ist man in dem Zweig, welcher hinzugefügt wird, kann auch **HEAD** stattdessen übergeben werden,

da HEAD zu diesem Zeitpunkt dem Zweig gleicht. Um einen Zweig wiederrum aus dem Referenzarchiv zu löschen, wird **git push --delete <Zweig>** verwendet. Von Zeit zu Zeit ist es sinnvoll nachzuvollziehen zu können ob ein bestimmter Commit schon einem Zweig zugeführt wurde. Dafür kann **git branch --contains <Prüfsumme Commit>** verwendet werden. Damit werden alle Zweige aufgelistet, die diesen Commit beinhalten. Dies funktioniert jedoch nur, falls der gesuchte Commit über eine Methode hinzugefügt wurde, welche die Metadaten des Commits beibehält. Ist der Commit zum Beispiel über einen *cherry-pick* oder *patch* zugeführt wurden, erkennt dies Git nicht als gleichen Commit an (9). Hierfür hilft das Kommando **cherry**, beschrieben in Kapitel 4.11.2.

In dem Beispielprojekt wird nun ein neuer Zweig mit dem Namen "feature" erstellt, um den Ansatz zu implementieren. In Abbildung 24 zu sehen signalisiert der Stern jedoch, dass der aktive Zweig immer noch "master" ist.

```
labadmin@eitlinux ~/Git $ git branch feature
labadmin@eitlinux ~/Git $ git branch
  feature
* master
```

Abbildung 24: Erstellung eines weiteren Zweiges

Um nun auf den neu erstellten Zweig *feature* zu wechseln, verwendet man den Befehl *checkout*.

4.5.4 Git checkout

Mit **git checkout <Zweig>** wird der aktuell zu bearbeitendem Zweig gewechselt. Hierbei müssen alle Änderungen an der Arbeitskopie zuvor einem Commit hinzugefügt, zwischengespeichert oder revidiert werden. Ist dies nicht der Fall schlägt das Kommando fehl. Intern wechselt HEAD, zuständig für das Laden der Arbeitskopie, auf den neusten Commit des gewünschten Zweiges. Wird dem Befehl *-b* übergeben, wird der Zweig neu erstellt. Allerdings können nicht nur Zweige übergeben werden, auch einzelne Commits und Dateien können mit *checkout* verwendet und dadurch geladen werden. Bei einer Datei wird damit die letzte Version dieser Datei der Arbeitskopie hinzugefügt, beziehungsweise die vorhandene Datei mit der übergebenen Datei überschrieben. Dies ist nützlich falls man Änderungen an einer einzelnen Datei verwerfen will (10). Ein weiterer Einsatz ist die gezielte Implementierung von einzelnen Dateien und Komponenten, ohne komplette Zweige zusammen zu führen, beschrieben in 4.6.1. Wird ein Commit, beziehungsweise dessen Prüfsumme, übergeben, entsteht dadurch ein sogenannter "*detached HEAD*". Das heißt, der HEAD Zeiger zeigt nun nicht auf den neusten Commit eines Zweiges. Ist HEAD nun nicht an den neusten Commit eines Zweiges gekoppelt, fehlt Git die Zuordnung an einen Zweig. Arbeitet man nun an dem Commit weiter, das heißt, man erstellt einen neuen Commit, dann wird eine neue Abzweigung erstellt, da dem Commit bereits ein Commit folgt. Diese Abzweigung besitzt jedoch

keinen Namen, den man referenzieren könnte, wie bei einem Zweig. Daher muss der Abzweigung eine Referenz hinzugefügt werden, um wieder darauf zurückgreifen zu können, sobald man von der Abzweigung wechselt (11). Dies kann geschehen, indem man ein Etikett setzt oder nachträglich einen Zweig kreiert mit **git branch <Name>**.

Der Entwickler wechselt durch den Befehl **git checkout feature**, gemäß Abbildung 25, auf den Zweig *feature*, um den Ansatz verfolgen zu können.

```
labadmin@eitlinux ~/Git $ git checkout feature
Zu Branch 'feature' gewechselt
```

Abbildung 25: Wechsel der aktiven Arbeitsmappe

Nachdem die Implementierung des Ansatzes durch einen Commit dem Zweig *feature* hinzugefügt ist, wird wieder auf *master* gewechselt, um die angefangenen Änderungen fortzuführen. Dazu wird, in Abbildung 26 zu sehen, der zuvor abgespeicherte Zwischenspeicher "Work in Progress Linie" wieder auf die nun leere Arbeitskopie in *master* angewandt. "stash@{0}" verweist auf diese Zwischenspeicherung, da lediglich dieser gespeichert wurde. Da *pop* statt *apply* verwendet wurde, wird der Zwischenspeicher nach der Anwendung gelöscht und befindet sich nicht mehr im Stash, wie in 4.5.1 behandelt. Die Änderungen befinden sich damit in der Arbeitskopie und können nach Fertigstellung durch einen Commit dem Zweig angehangen werden.

```
labadmin@eitlinux ~/Git $ git checkout master
Bereits auf 'master'
labadmin@eitlinux ~/Git $ git stash pop stash@{0}
Auf Branch master
zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git reset HEAD <Datei>..." zum Entfernen aus der Staging-Area)

    neue Datei:      Linie.cpp
    neue Datei:      Linie.h

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

    geändert:      depend
    geändert:      main.cpp

stash@{0} (3739c154159e9ebfc05cc8fefc57770ce62b9c81) gelöscht
```

Abbildung 26: Anwendung des Zwischenspeichers

Neben dem bereits vorhandenen Zweig *feature* erstellt der Entwickler einen weiteren Zweig namens *newshape*. Dieser Zweig soll eine neue Funktion hinzufügen, ohne Einfluss auf den bisherigen Stand des Hauptzweigs *master* oder dem experimentellen Ansatz in *feature* zu haben. Diesmal wird zu Erstellung des Zweiges direkt das Kommando *checkout* verwendet. Dadurch wird neben der Erstellung des Zweiges, dieser auch direkt als aktiver Zweig ausgewählt. Nach einiger Implementierungsarbeit kann dem Zweig auch ein Commit hinzugefügt werden. In der Kommandozeile sieht dies wie folgt aus:

```

labadmin@eitlinux ~/Git $ git checkout -b newshape
Zu neuem Branch 'newshape' gewechselt
labadmin@eitlinux ~/Git $ git add .
labadmin@eitlinux ~/Git $ git commit -m 'Added Rechteck implementation'
[newshape 5a3c6c1] Added Rechteck implementation
4 files changed, 148 insertions(+), 6 deletions(-)
create mode 100755 Rechteck.cpp
create mode 100755 Rechteck.h

```

Abbildung 27: Erstellung und Arbeit am Zweig "newshape"

Um einen Überblick über das Projektarchiv und dessen Zweige zu erhalten, kann man neben der Kommandozeile auch die Git integrierte grafische Oberfläche verwenden.

4.5.5 Gitk

Um eine schnelle Übersicht über das Projektarchiv zu erhalten kann *Gitk* verwendet werden. Dies startet eine grafische Übersicht, welche die Historie des aktuellen Zweiges anzeigt. Mit dem Schlüsselwort **--all** wird die Übersicht auf das gesamte Archiv erweitert. Zu sehen ist hier auch der symbolische Baum, welcher durch die Commits gebildet wird. *Gitk* bildet nicht nur eine Übersicht, es lässt sich darüber auch mit den Commits arbeiten und Aktionen ausführen. *Gitk* wird außerdem auch durch *git gui* aufgerufen, die native grafische Oberfläche zur Erstellung von Commits (12).

Mit dem Befehl **gitk** wird der derzeitige Pfad und dessen Vorgänger beleuchtet. Dies sieht wie folgt aus:

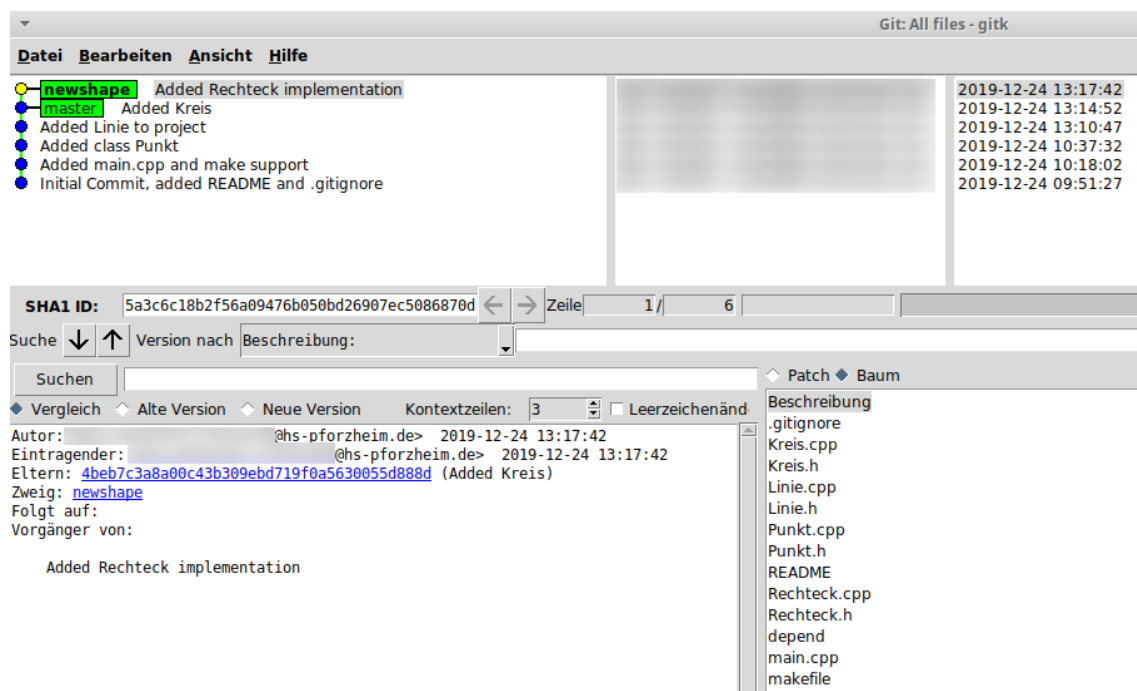


Abbildung 28: Darstellung des Projekts in gitk

Zu erkennen sind die grün markierten Zweige. Man kann anhand dieser Darstellung ablesen, dass der neuste Commit des Zweiges *newshape* auf dem neusten Commit von *master* basiert. Das heißt, um die Änderungen von *newshape* zu übernehmen, muss lediglich der Zeiger von *master* inkrementiert werden, da der *master* Commit bereits ein Vorfahre ist und der *newshape* Commit lediglich auf diesem aufbaut. Dieses Prinzip der Zusammenführung nennt sich *fast-forward*. Da dabei keine Dateien zusammengeführt werden, besitzen diese Zusammenführungen kein Konfliktpotenzial. Durchgeführt wird die Zusammenführung der beiden Zweige über den Befehl *merge*.

4.6 Zusammenführung von Zweigen

Durch die Erstellung von Zweigen und die damit einhergehende Isolation von anderen Änderungen, bietet Git unterschiedliche Möglichkeiten die bestehenden Zweige miteinander zu synchronisieren. Ohne diese Funktion wäre eine effiziente Teamentwicklung nicht möglich.

4.6.1 Git merge

Git verwendet einen sogenannten Drei-Wege-Merge. Das Kommando *merge* wird dazu verwendet Zweige zusammenzuführen, beziehungsweise Änderungen von anderen Zweigen zu übernehmen. Um verschiedene Commits zusammenzuführen, werden beide Stände der jeweiligen Arbeitskopie Datei für Datei verglichen. Im Gegensatz zu einem Zwei-Wege-Merge, beziehungsweise einem einfachen zeilenweisen Abgleich, wird durch die Änderungsverfolgung von Git auch der früheste gemeinsame Vorgänger beider Dateien einbezogen. Dadurch erkennt der Algorithmus zur Zusammenführung neben einer Abweichung der zusammengehörigen Dateien, auch welche Datei, die zu übernehmende Änderung besitzt. Dies wird erst durch den Einbezug des gemeinsamen Vorfahren beider Dateien ermöglicht.

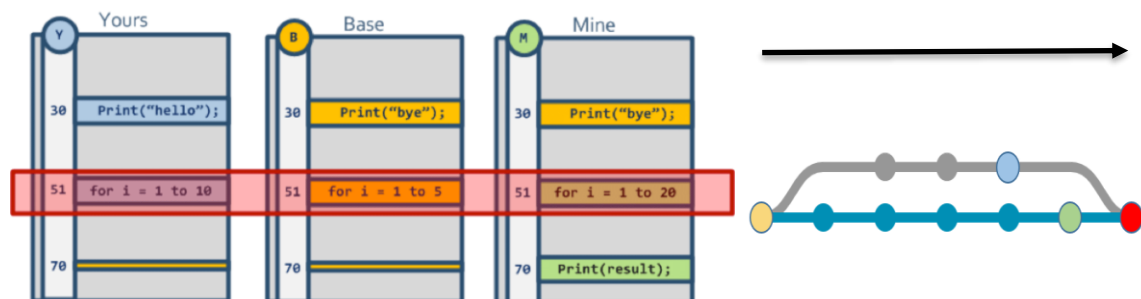


Abbildung 29: Problemstellung einer Drei-Wege-Zusammenführung (13)

In Abbildung 29 sind einige beispielhafte Zeilen zu sehen, welche zusammengeführt werden sollen. Zur Veranschaulichung welches Schaubild, welchen Stand der Datei im Projektarchiv darstellt, hilft die farbliche Markierung der Commits. "Yours" ist hierbei

die Datei aus dem Zweig, welcher mit "Mine" zusammengeführt werden soll. "Base" stellt den gemeinsamen Vorfahren da. In Zeile 30, der ersten Zeile der Darstellung, enthält die Datei "Yours" die Abweichung zu "Base" und "Mine". Dies ist für Git die zu übernehmende Änderungen. Dabei ist der generelle Ansatz von Git, mit dem die Änderungen zusammengeführt werden sollen, die Abweichung zu "Base" zu übernehmen. Führt dieser Ansatz zu nicht gewollten Änderungen, muss die Zusammenführung händisch erfolgen. Der Vorteil der Git Methodik ist, dass Konflikte nur entstehen, falls beide Stände Abweichung zum Vorfahren besitzen. Dies ist zu sehen im rot markierten Bereich in Zeile 50. Dieser Konflikt muss per Hand gelöst werden. Sogenannte *merge tools* helfen dabei, schnell diese zu Konflikt führenden Änderungen blockweise zu übernehmen oder zu ersetzen. Im letzten Beispiel ist eine Änderung nur in einer der beiden Dateien vorhanden, diese wird in die Zusammenführung übernommen, da dies eine Änderung zum Vorfahren ist. Des Weiteren wird nachdem eine Zusammenführung vollzogen wurde, ein sogenannter *merge-link* erstellt. Dadurch wird sichergestellt, dass bei einer späteren Zusammenführung, keine bereits gelösten Konflikte erneut durchgeführt werden müssen, da der Vorfahre durch den *link* erneuert wird. Sollte in Abbildung 29 eine weitere Zusammenführung zu späterer Zeit stattfinden, würde der gemeinsame Vorfahre abhängig des Zielzweigs entweder die Version "Mine" oder "Yours" sein.

Um die Theorie nun in die Praxis umzusetzen, wird zuerst der Zielzweig als aktuelle Arbeitskopie gesetzt. Danach kann die Zusammenführung über *git merge <Quellzweig>* vollzogen werden. Besitzt der Zielzweig keine abweichenden Commits, sodass der Quellzweig nur Änderungen über den Zielzweig hinaus besitzt, findet eine *fast-forward* Zusammenführung statt. Da keine Dateien zusammengeführt werden müssen, wird nur der interne Zeiger des Zielzweiges auf den Commit des Quellzweigs vorgerückt. Um dies zu unterbinden und immer einen dedizierten Commit zu erstellen wird die Option *--no-ff* verwendet. Um den *merge-commit* zusätzlich zu überprüfen oder letzte Änderungen manuell einzufügen kann *--no-commit* verwendet werden. Die Arbeitskopie enthält danach die Ausgabe der Zusammenführung. Diese Methode sollte man bei wichtigen Implementierungen zu Hauptzweigen verwenden, um die zusammengeführten Dateien manuell zu überprüfen oder die Lauffähigkeit und Korrektheit der Arbeitskopie zu sichern. Danach kann der Commit der Zusammenführung manuell vollendet werden mit *git merge --continue*. Die Nachricht für den *merge-commit* wird mit *-m* wie gehabt, übergeben. Neben der von Git bevorzugten Methodik kann der Nutzer anhand eines Schlüsselworts eine andere Strategie für die Zusammenführung wählen. Die zwei Wichtigsten sind *--ours* und *--theirs*. Damit kann bei Konflikten generell eine Version bevorzugt werden. Zu beachten gilt dabei, dass das Zusammenspiel der Änderungen nicht durch eine manuelle Konfliktlösung überprüft wird. Die Lauffähigkeit sollte im Nachhinein somit überprüft werden. Eine weitere nützliche Option bietet *--squash*. Hierdurch werden alle Änderungen, welche in die Zusammenführung einfließen zu einem

Commit zusammengefasst und als neuer, isolierter Commit dem Zielzweig hinzugefügt. Es wird allerdings keine Verbindung zum Ursprungs Zweig hergestellt. Git sieht den Ursprunzweig demnach als nicht zusammengeführt an. Sollte man sich während der Zusammenführung umentscheiden oder es kommen unbekannte Komplikationen auf, kann der Vorgang verworfen werden über **git reset --merge** (13).

4.6.2 Partielle Zusammenführung von Zweigen

In manchen Fällen ist eine Zusammenführung des kompletten Zweigs nicht erwünscht und es sollen nur einzelne Dateien überführt werden. In diesem Fall wird ein *merge* gestartet, ohne dass Änderungen übernommen werden. Konkret wird in den Zielzweig navigiert, um von dort die Zusammenführung zu starten. Durch die Übergabe von verschiedenen Parametern, werden alle einkommenden Änderungen ignoriert und der Commit in jedem Fall unterbrochen. Dieser Schritt wird benötigt, um die partielle Zusammenführung auch in Git zu kennzeichnen, um eine höhere Nachvollziehbarkeit zu erhalten. Der Befehl lautet: **git merge --no-ff --no-commit -s ours <Quellzweig>**. Die Übergabe von *-s ours* führt hierbei dazu, dass die zu integrierenden Änderung verworfen werden. Nicht zu verwechseln mit dem Parameter *--ours*, bei welchem nur im Konfliktfall die einkommenden Änderungen verworfen werden. Ist die Nachvollziehbarkeit nicht von Wichtigkeit kann dieser Schritt übersprungen werden. Um die gewünschten Dateien schließlich zu überführen, werden diese über den Befehl *checkout* geladen. Dies geschieht mit **git checkout <Quellzweig> <Datei[pfad]>**. Damit wird die vorhandene Datei durch die gewünschte Datei ersetzt. Dieser Schritt wird wiederholt bis die gewünschten Änderungen vollständig vorhanden sind. Danach kann der angefangene *merge* beendet werden (14).

Im Beispielprojekt soll der komplette Zweig überführt werden. Führen wir dort die Änderungen zusammen, signalisiert Git, dass es sich dabei um eine *fast-forward* Zusammenführung hat, wie durch gitk in Zeile 5 der Abbildung 28 bereits dargestellt:

```
labadmin@eitlinux ~/Git $ git checkout master
Zu Branch 'master' gewechselt
labadmin@eitlinux ~/Git $ git merge newshape
Aktualisiere 4beb7c3..5a3c6c1
Fast-forward
 Rechteck.cpp | 93 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 Rechteck.h   | 41 ++++++++++++++++++++++++++++++++++++++
 depend      |  5 +++-
 main.cpp     | 15 ++++++-----
 4 files changed, 148 insertions(+), 6 deletions(-)
 create mode 100755 Rechteck.cpp
 create mode 100755 Rechteck.h
```

Abbildung 30: Fast-forward Integration des Zweigs *newshape* in *master*

Nach der Zusammenführung entsteht die in Abbildung 31 dargestellte Historie.

```
labadmin@eitlinux ~/Git $ git log --all --oneline --graph
* 5a3c6c1 (newshape, master) Added Rechteck implementation
* 4beb7c3 Added Kreis
* 3176713 Added Linie to project
| * 80a0116 (HEAD -> feature) Added Grafik implementation
|/
* 93e29fa Added class Punkt
* 07b1157 Added main.cpp and make support
* 4e62daf Initial Commit, added README and .gitignore
```

Abbildung 31: Projekt Historie nach erfolgreicher Integration von *newshape*

Die beiden Zweige *newshape* und *master* in grüner Schrift, beinhalten den neusten Commit. Nach der erfolgten Zusammenführung will der Entwickler die Änderungen des *feature* Zweigs dem Hauptzweig *master* zuführen. Dazu wählt er die zweite Hauptmethodik Änderungen zusammenzuführen. Hierzu wird das Git Kommando *rebase* verwendet.

4.6.3 Git rebase

Neben *merge* steht auch das Kommando *rebase* zur Verfügung, um verschiedene Zweige zu vereinen. Generell wird das *rebase* Kommando dazu genutzt die Historie zu verändern. In der einfachsten Form benötigt man *rebase*, um die Historie übersichtlicher und strukturierter zu gestalten. Im Wesentlichen versucht der Befehl *rebase* in der nicht interaktiven Form (die interaktive Form wurde bereits in Kapitel 4.4.1 behandelt), den Vorfahren eines Zweiges zu verändern. Nehme man an, man wolle ein Entwicklungszweig, welcher seit längerer Zeit besteht, in den vorangeschrittenen Hauptzweig eingliedern. Damit das Archiv übersichtlich bleibt, will man den Entwicklungszweig verschieben, damit der Vorfahre der Abzweigung der neuste Commit des Hauptentwicklungszweiges ist. Dies geschieht anhand dem in Abbildung 32 dargestellten Beispiel mit dem Kommando ***git rebase maindev dev*** oder, falls man im grauen Entwicklungszweig ist (***git checkout dev***), verkürzt sich der Befehl zu: ***git rebase maindev***. Dadurch zeigt nun der erste Commit von *dev* auf den letzten Commit von *maindev*, vorausgesetzt es entstand bei dem Aufsetzen der Commits auf den neusten Stand von *maindev* kein Konflikt. Dies entspricht der rechten Darstellung.



Abbildung 32: Auswirkung des Befehls *rebase*

Um die zwei Zweige schließlich zu vereinen, fehlt noch ein weiterer Schritt, welcher umgekehrt ***git rebase dev*** auslöst. Folgend werden nun alle Commits von *dev* auf *maindev* angewendet. Da *dev* durch den ersten *rebase* aus dem neusten Commit herausgelöst

wurde, ist die Integration folgend nur noch ein *fast-forward-merge*. Der Zeiger von *maindev* wird somit auf den Zeiger von *dev* inkrementiert. Alternativ dazu könnte man auch eine einfache Zusammenführung über das Kommando *merge* erwirken oder *git fetch dev:maindev* verwenden, um dementsprechend ein *fast-forward merge* zu erwirken. Neben diesem Hauptbestandteil des Kommandos *rebase*, bietet das Kommando noch einen deutlich größeren und komplizierteren Umfang. Mit dem Schlüsselwort *--onto* wird das Kommando zum Beispiel, falls der Zweig *dev* ausgewählt wurde, auf *git rebase --onto maindev <Start> <Ziel>* erweitert, um nur bestimmte Commits zu übernehmen und deren Historie dadurch zu verändern. Generell gilt Vorsicht bei *rebase* da die Historie ein wesentlicher Vorteil der Versionsverwaltung darstellt. Wird die Historie zu stark verschlankt, da ein Großteil der Zweige nach Beendigung über das Kommando *Rebase* einem Hauptzweig zugeführt werden, sind die Zweige zwar übersichtlich, allerdings lassen sich einzelne Funktionalitäten nicht mehr unterscheiden. Auch werden die Zeitpunkte der Commits verändert, sowie deren Metadaten. Daher sollte man sich vergewissern, dass niemand auf die ursprünglichen Commits explizit angewiesen ist, welche man über das Kommando verschiebt oder umschreibt. Deshalb wird empfohlen, nur in persönlichen Zweigen den Befehl zu verwenden, um Konflikte zu vermeiden. Diese können entstehen, sofern Verweise auf veraltete Commits bestehen, da diese nach einem *Rebase* nicht mehr bestehen. Bei der Anwendung des Befehls sollte nicht vergessen werden, dass eine nachvollziehbare Historie ein großer Bestandteil einer Versionsverwaltung ist und somit der Einsatz des Kommandos durchdacht gewählt sein sollte, damit das Projektarchiv klare Strukturen besitzt und übersichtliche Abgrenzungen einzelner Themen nicht verliert. Ein weiteres Problem bieten bereits gelöste Konflikte, die durch die erneute Anwendung der Änderungen auf die neusten Commits des gewünschten Zweigs wiederaufkommen können (6 p. Kap. 4).

Wendet der Entwickler das Kommando *rebase* auf den Zweig *feature* an, um diesen mit dem Hauptzweig zu vereinen, geschieht folgendes:

```
labadmin@eitlinux ~/Git $ git checkout feature
Bereits auf 'feature'
labadmin@eitlinux ~/Git $ git rebase master
Zunächst wird der Branch zurückgespult, um Ihre Änderungen
darauf neu anzuwenden ...
Wende an: Added Grafik implementation
Verwende Informationen aus der Staging-Area, um ein Basisverzeichnis nachzustellen ...
M      depend
M      main.cpp
.git/rebase-apply/patch:97: trailing whitespace.
main.o: main.cpp Grafik.h Punkt.h
warning: 1 Zeile fügt Whitespace-Fehler hinzu.
Falle zurück zum Patchen der Basis und zum 3-Wege-Merge ...
automatischer Merge von main.cpp
KONFLIKT (Inhalt): Merge-Konflikt in main.cpp
automatischer Merge von depend
KONFLIKT (Inhalt): Merge-Konflikt in depend
error: Merge der Änderungen fehlgeschlagen.
Anwendung des Patches fehlgeschlagen bei 0001 Added Grafik implementation
Benutzen Sie 'git am --show-current-patch', um den
fehlgeschlagenen Patch zu sehen.

Lösen Sie alle Konflikte manuell auf, markieren Sie diese mit
"git add/rm <konfliktbehaftete_Dateien>" und führen Sie dann
"git rebase --continue" aus.
Sie können auch stattdessen diesen Commit auslassen, indem
Sie "git rebase --skip" ausführen.
Um abzubauen und zurück zum Zustand vor "git rebase" zu gelangen,
führen Sie "git rebase --abort" aus.
```

Abbildung 33: Konfliktbehaftete Durchführung des Kommandos *rebase*

Die automatische Zusammenführung, basierend auf dem Drei-Wege-Merge, konnte zwei Dateien bei der Verschiebung des Commits auf *feature*, nicht automatisch zusammenführen und es entstanden dadurch zwei Konflikte. Daraufhin wurde das Kommando unterbrochen. Deshalb müssen beide Konflikte gelöst werden, bevor der das Kommando fortgeführt werden kann.

4.6.4 Behebung von Konflikten

Konflikte entstehen unter Umständen bei Zusammenführungen, zum Beispiel dem *merge* oder *rebase* Kommandos. Durch den 3-Wege-Merge entsteht ein Konflikt nur, wenn sich beide Versionen der gleichen Datei mit der Version des gemeinsamen Vorfahren unterscheiden. Tritt dieser Fall ein, wird die Zusammenführung unterbrochen und der Konflikt benötigt eine manuelle Korrektur. Mit *git status* werden die Dateien angezeigt, welche einen Konflikt hervorrufen. Dazu wird oftmals ein *merge tool* zur Hilfe angewendet, da durch eine grafische Lösung meist deutlich übersichtlicher und schneller editiert werden kann. In der Konfliktbehafteten Datei werden beide Versionen der Datei dargestellt. Zur Identifizierung von konfliktbehafteten Absätzen oder Zeilen, starten diese mit "`<<<<<<<< HEAD`". Die Version des Quellzweigs steht gefolgt zwischen "`=====`" und "`>>>>>>>> <Zweig>`". Um den Konflikt zu lösen müssen die *merge-marker*, sowie die ungewünschte Version entfernt werden. Danach wird die korrigierte Datei über *git add <Datei>* wieder hinzufügt, sodass der Konflikt für Git als gelöst gilt. Gefolgt von *git commit* wird nun das angefangene Kommando vollzogen. Falls man nicht auf diese Art der Konfliktlösung zurückgreifen will, kann der Zusammenführungsversuch auch über *git merge --abort* abgebrochen werden. Ein Sonderfall stellt eine Datei dar, welche nur in einem der beiden Zweige existiert. Man entscheidet sich dann über *git add* oder *git rm* die Datei zu behalten oder zu entfernen (15).

Um die entstanden Konflikte zu lösen kann das standardmäßig in Git hinterlegte *merge tool* gestartet werden. In diesem Fall ist das Programm "meld" als Standard hinterlegt.

4.6.5 Git mergetool

Um einen entstanden Commit einfach und schnell zu lösen kann eine grafische Oberfläche verwendet werden. Dies geschieht, indem das Programm die konfliktbehafteten Dateien nach den Wünschen des Nutzers anpasst. Durch einen Konflikt werden mehrere temporäre Dateien erzeugt, um jeden Stand widerspiegeln zu können. Diese helfen bei einer Gegenüberstellung, die gewünschte Version auszuwählen. Durch einen Konflikt werden 4 Versionen einer Datei angelegt. Neben einer *remote*, *base* und *local* Version, wird zusätzlich eine *backup* Version je Datei erstellt. Diese *backup* Version besitzt ebenso die *merge-marker*, welche den Konflikt eingrenzen. Um das eingesetzte Lösungsprogramm nicht explizit aufrufen zu müssen, wird von Git der Alias ***git mergetool*** bereitgestellt. Dadurch wird das in Git hinterlegte Lösungsprogramm aufgerufen. Das hinterlegte Programm kann über ***git config merge.tool <Programm Pfad>*** verändert werden. Standardmäßig werden dadurch die Programme *vimdiff* oder *meld* aufgerufen. In diesem Programm kann nun der Konflikt sauber aufgelöst werden. Nach Beendigung des Programms sollte Git erkennen, dass kein Konflikt mehr besteht. Über *git commit* wird die gestoppte Zusammenführung fortgeführt und die zuvor ausgewählten Änderungen angewandt.

Führt man auf das Kommando ***git mergetool*** aus, in Abbildung 34 dargestellt, erkennt Git, dass kein Programm hinterlegt wurde. Allerdings besitzt Git mehrere Verweise auf Standardprogramme, welche aufgerufen werden, sofern diese installiert sind. In diesem Fall wird das Programm *meld* als erste verfügbare Installation festgestellt und dementsprechend aufgerufen.

```
labadmin@eitlinux ~/Git $ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuze diffmerge ecmmerge p4merge araxis bc codecompare emerge vimdiff
Merging:
depend
main.cpp

Normal merge conflict for 'depend':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

Abbildung 34: Ausführung des Git hinterlegten merge tool

Dargestellt in Abbildung 35 sieht man die Konflikte der Dateien "main.cpp". Dabei steht links die Version des Zweigs *master*, sowie rechts die Version aus dem Zweig *feature*. Rot zeigt dabei Konflikte, während in blau Abweichungen dargestellt werden.

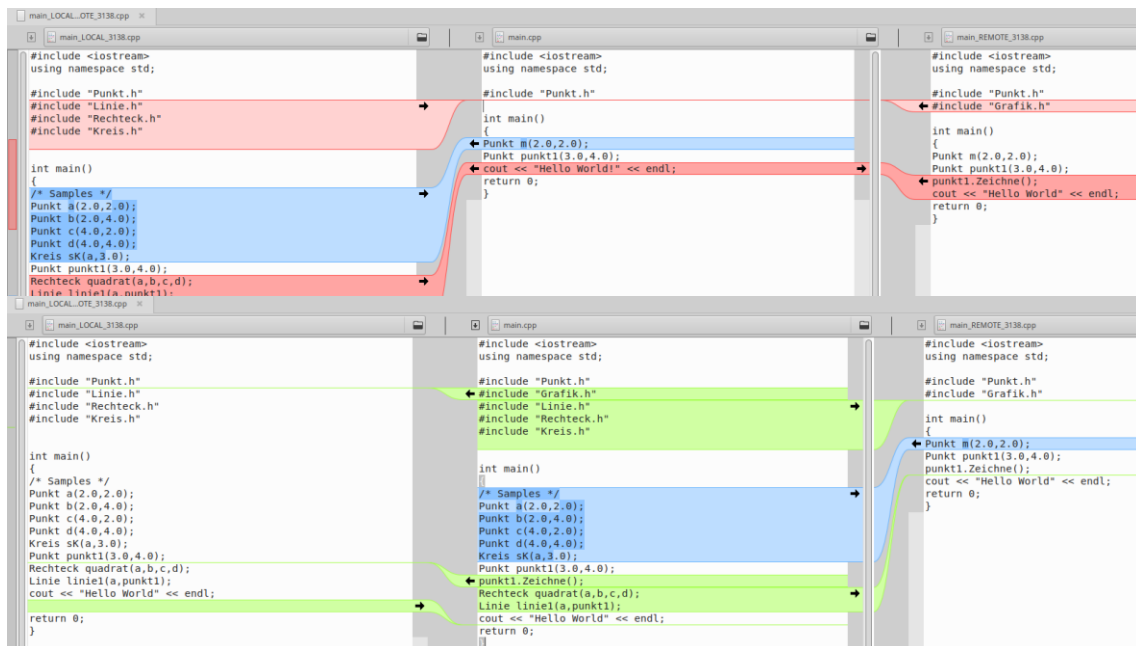


Abbildung 35: Gelöster Konflikt durch das Programm meld

In Grün werden behobene Konflikte dargestellt. Mit Hilfe der Pfeile können die unterschiedlichen Blöcke ausgewählt und der Ausgangs Datei, befindlich in der Mitte hinzugefügt werden. In *meld* kann ein Block gelöscht werden über die Taste "shift" oder zusätzlich hinzugefügt werden, anstatt vorhandenes zu überschreiben. Das Hinzufügen erfolgt mit der Taste "strg". In dem expliziten Fall ist dies bei den "#include" Zeilen von Nutzen. Der untere Teil der Abbildung zeigt die behobene Datei an.

Nachdem beide Konflikte gelöst sind, kann das rebase Kommando fortgeführt werden. Bei manchen *merge tools*, auch *meld*, bleiben die ursprünglichen Konfliktdateien mit dem Anhang ".orig" bestehen, um im Notfall auf diese zurückgreifen zu können. Diese Dateien beinhalten beide Versionen, getrennt durch die zuvor eingeführten Konfliktmarker, beschrieben in Kapitel 4.6.4. Die Projektarchiv Historie nach der Durchführung des Konflikts wird in Abbildung 36 dargestellt. Der Commit aus *feature* wurde als erstes positioniert, beziehungsweise nach oben verschoben, da er nun auf dem neuesten Commit von *master* basiert.

```

labadmin@eitlinux ~/Git $ git rebase --continue
Wende an: Added Grafik implementation
labadmin@eitlinux ~/Git $ git status
Auf Branch feature
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

    depend.orig
    main.cpp.orig

nichts zum Commit vorgemerkt, aber es gibt unversionierte Dateien
(benutzen Sie "git add" zum Versionieren)
labadmin@eitlinux ~/Git $ git log --graph --oneline --all
* f69ef72 (HEAD -> feature) Added Grafik implementation
* 5a3c6c1 (newshape, master) Added Rechteck implementation
* 4beb7c3 Added Kreis
* 3176713 Added Linie to project
* 93e29fa Added class Punkt
* 07b1157 Added main.cpp and make support
* 4e62daf Initial Commit, added README and .gitignore

```

Abbildung 36: Weiterführung des Befehls *rebase*

Im Umkehrschluss können die *feature* Commits und deren Änderungen in die anderen Zweige integrieren werden, indem man eine einfache fast-forward Zusammenführung veranlasst. Diesmal verwendet der Entwickler ein weiteres Mal den Befehl *rebase* dafür. Dazu wechselt er auf den Zweig, in welchem er die Änderungen implementieren will. Danach wird der Befehl ausgeführt und als Übergabe erhält dieser den Zweig, welcher die Änderungen beinhaltet. Dieser Vorgang wird in Abbildung 37 dargestellt.

```

labadmin@eitlinux ~/Git $ git checkout master
Zu Branch 'master' gewechselt
labadmin@eitlinux ~/Git $ git rebase feature
Zunächst wird der Branch zurückgespult, um Ihre Änderungen
darauf neu anzuwenden ...
master zu feature vorgespult.

```

Abbildung 37: Zusammenführung zweier Zweige über das Kommando *rebase*

4.7 Synchronisierung von Archiven

Nun wird das Zusammenspiel mit anderen Entwicklern beleuchtet. Bisher wurde das Projektarchiv nur lokal verwendet. Die Synchronisierung der Projektarchive ermöglicht die gemeinsame Entwicklung eines Projektes und ist elementarer Bestandteil moderner Software Entwicklung. Neben den bisher verwendeten Befehlen rechts zu sehen in Abbildung 38, kommen nun die Kommandos links hinzu. Mit diesen Funktionen wird grundlegend global interagiert und eine Zusammenarbeit ermöglicht.

Als Beispiel für ein globales Referenzarchiv wird dieses auf dem Hoster Github.com erstellt. Um nun eine Verbindung des lokalen Archivs mit dem Referenzarchiv herzustellen wird der Befehl *remote* verwendet.

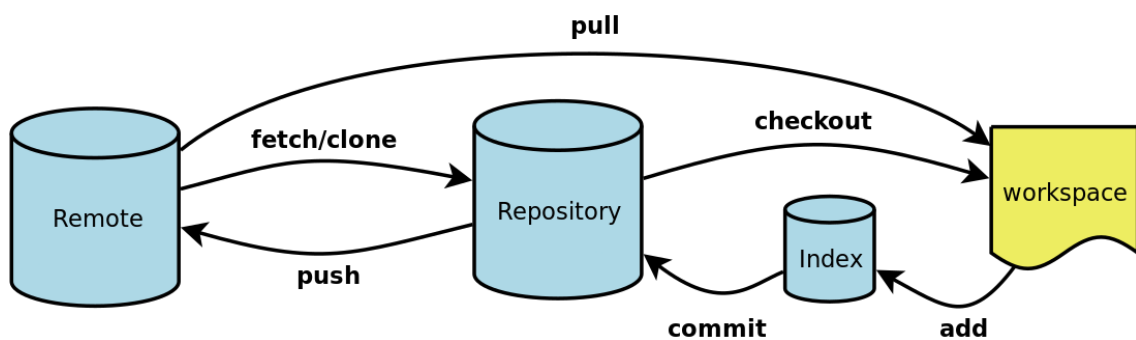


Abbildung 38: Synchronisierung des lokalen Projektarchivs (35)

4.7.1 Git remote

Die Bezeichnung *remote* steht im Allgemeinen für den Ursprung des Projektarchivs. Grundsätzlich fallen aber auch andere Projektarchive darunter, zu welchen das eigene Archiv eine Verbindung aufnehmen soll. Erstellt man ein lokales Archiv ist kein *remote* Eintrag vorhanden, da für Git nur das lokale Archiv existiert. Um die aktuellen *remote* Aliase einzusehen kann ***git remote -v*** verwendet werden. Erstellt man ein Projekt auf einem Hoster, welches man später lokal klonet, wird automatisch ein Eintrag erstellt, welcher auf das im Hoster hinterlegte Archiv zeigt. Der Alias Name ist standardmäßig *origin*. Über ***git remote add <alias> <url>*** wird ein neuer *remote* Eintrag erstellt. Um den Alias nur für einkommende Synchronisation Kommandos *fetch* oder *pull* zu verwenden, kann man die jeweiligen URL zum Referenzarchiv auch über ***git remote set-url <alias> <url>*** für *fetch* und mit dem Anhang ***[...] set-url --push [...] für das Kommando push*** erstellen. Dies wird zum Beispiel verwendet, um Änderungen von einem Hauptarchiv zu erhalten aber eigene Entwicklungen einem eigenen Referenzarchiv hinzuzufügen. Dasselbe wird zum Beispiel bei sogenannten Forks verwendet. Bei größeren Projekten können Untermodule ein eigenes Projektarchiv besitzen (16).

Im voranschreitenden Beispielprojekt wird ein neues, leeres Archiv auf Github.com erstellt. Unter dem simplen Namen "Git" kann auf das erstellte Projektarchiv zugegriffen werden. Nachfolgend wird es als Referenzarchiv dem bereits bestehenden lokalen Archiv hinzugefügt, siehe Abbildung 39.

```
labadmin@eitlinux ~/Git $ git remote add origin https://github.com/_____/Git.git
labadmin@eitlinux ~/Git $ git remote -v
origin  https://github.com/_____/Git.git (fetch)
origin  https://github.com/_____/Git.git (push)
```

Abbildung 39: Hinzufügen eines Referenzarchivs

Um den lokalen Fortschritt mit dem Referenzarchiv zu synchronisieren, wird das Kommando *push* verwendet.

4.7.2 Git push

Der Befehl ***git push*** wird dazu benötigt die lokalen Änderungen im Projektarchiv auf den globalen Server zu legen, beziehungsweise auf das Referenzarchiv. Somit wird *push* nur benötigt, wenn man einen nicht lokalen Host nutzt, zum Beispiel das Archiv auf *github.com* liegt. Das globale Referenzarchiv wird meist *origin* genannt. Um zum Beispiel Änderungen auf dem Zweig *master* auf dem globalen Archiv zu veröffentlichen, kann der Befehl ***git push origin master*** benutzt werden. Mit dem Zusatz ***-f*** wird die Historie und die Änderungen des lokalen Archivs übernommen, auch wenn das Referenzarchiv neuere Commits enthält. Diese werden dementsprechend verworfen. Daher sollte der Anhang ***-f*** nur unter gewissen Umständen benutzt werden. Mit dem Zusatz ***--all*** werden alle Referenzen des lokalen Archives synchronisiert, so muss nicht jeder Zweig einzeln dem Kommando übergeben werden. Da es im Allgemeinen nicht gewollt ist, dass jeder direkten Zugriff auf ein Archiv eines Hosters besitzt, wird im Normalfall eine Anmeldung verlangt. Dies wird umgangen, indem man die Anmeldung bereits im *remote* Link enthält. Dadurch stehen die Anmeldedaten allerdings einsehbar unter den *remotes*. Daher kann zum Beispiel eine Verifikation über das Verbindungsprotokoll SSH erfolgen. Dafür muss das SSH Schlüsselpaar einmalig angelegt werden und dem Hoster der Öffentliche Schlüssel mitgeteilt werden. Um diesen Verbindungstyp auch dementsprechend zu nutzen, muss der *remote* link zusätzlich auf SSH gewechselt werden. Ist keine automatische Anmeldung festgelegt, fragt Git nach den Anmeldedaten bei jedem Zugriff auf das Referenzarchiv (4 p. 61ff.).

Durch die hinzugefügte Verbindung zu Github.com, wird das bisher leere Referenzarchiv mit dem lokalen Archiv überschrieben. Da es sich um ein privates Projekt handelt, erfordert eine Interaktion mit dem Archiv eine Anmeldung, welche nach Eingabe des Befehls erfolgt. Zuletzt sieht man, dass alle Zweige des Referenzarchivs gleichauf mit den lokalen Zweigen sind:

```
labadmin@eitlinux ~/Git $ git push -f origin --all
Username for 'https://github.com': 
Password for 'https://@github.com': 
Zähle Objekte: 41, Fertig.
Delta compression using up to 4 threads.
Komprimiere Objekte: 100% (38/38), Fertig.
Schreibe Objekte: 100% (41/41), 6.58 KiB | 1.31 MiB/s, Fertig.
Total 41 (delta 11), reused 0 (delta 0)
remote: Resolving deltas: 100% (11/11), done.
To https://github.com/~/Git.git
 * [new branch]      feature -> feature
 * [new branch]      master -> master
 * [new branch]      newshape -> newshape
```

Abbildung 40: Lokalen Stand als Referenz setzen

Da das Archiv nun auch für andere Entwickler zur Verfügung steht, finden nun auch unabhängig der lokalen Kopie Änderungen statt und das Projektarchiv füllt sich durch die Arbeit mehrerer Beteiligten. Es kann unter Umständen vorkommen, dass eine Zeilenänderung auffällt, welche sich als zweifelhaft herausstellt. Ohne die einzelnen Änderungsübersichten aller veranlasster Commits zu durchsuchen, um die entsprechende Zeilenänderung einem Autor zuordnen zu können, hilft das Kommando *blame* dabei, den jeweiligen Autor dieser Zeilen zu finden, um eine Korrektur zu veranlassen.

4.7.3 Git blame

Um herauszufinden, welcher Autor welche Änderungen an einer Datei durchgeführt hat, wird das Kommando **git blame** *<Datei>* verwendet. Dadurch wird für jede Zeile der Datei der Autor dieser angegeben. Um die Ausgabe weiter zu verfeinern kann über das Keyword **-L** *<start>*, *<end>* die Ausgabe auf bestimmte Zeilen verkleinert werden. Mit **-M** wird der Ersteller von Zeilen, welche sich nicht verändert haben, sondern nur kopiert beziehungsweise im Archiv verschoben wurden, berücksichtigt. Das heißt, der Autor dieser Zeilen ist nun der Ersteller, und nicht der Autor des Commits, welcher die Zeile verschob oder kopierte. Mit **-C** wird das Herausfinden des Erstellers einer einzelnen Zeile auf das Projektarchiv ausgeweitet, beziehungsweise Dateiübergreifend festgestellt (17). In Abbildung 41 sieht man eine beispielhafte Ausgabe des Kommandos. Diese ist in fünf Spalten einzuteilen. Zuerst steht der Anfang der Prüfsumme des Commits, welcher diese Änderung mit sich brachte. Diese ist gefolgt von dem Autor, das Datum und der Zeile der Datei. Die letzte Spalte zeigt den Inhalt der Datei.

```
labadmin@eitlinux ~/Git $ git blame -e main.cpp
07b1157c (< @hs-pforzheim.de> 2019-12-24 10:18:02 +0100 1) #include <iostream>
07b1157c (< @hs-pforzheim.de> 2019-12-24 10:18:02 +0100 2) using namespace std;
07b1157c (< @hs-pforzheim.de> 2019-12-24 10:18:02 +0100 3)
93e29fa4 (< @hs-pforzheim.de> 2019-12-24 10:37:32 +0100 4) #include "Punkt.h"
31767134 (< @hs-pforzheim.de> 2019-12-24 13:10:47 +0100 5) #include "Linie.h"
7d450cc6 (< @users.noreply.github.com> 2019-12-24 14:02:44 +0100 6) #include "Grafik.h"
```

Abbildung 41: Ansicht nach Ausführung des Befehls *blame*

Änderungen können bereits aus dem lokalen Archiv in das Referenzarchiv hochgeladen werden, damit andere Beteiligte mit diesen Änderungen arbeiten können. Mit den Befehlen *pull* und *fetch* können diese Commits im Referenzverzeichnis auf das lokale Archiv wiederum angewandt werden.

4.7.4 Git fetch

Mit **git fetch** wird das Referenzarchiv unter dem alias *origin* mit dem lokalen Stand abgeglichen. Allerdings werden abweichende Änderungen dem Nutzer lediglich angezeigt und nicht automatisch versucht diese auch zu integrieren. Aufgrund dessen wird oftmals der Befehl *fetch* mit anschließendem *merge* bevorzugt, da man zwischen den Kommandos sich nochmals versichern kann, dass nur gewollten Änderungen aus dem Referenzarchiv einfließen. Beide Befehle *fetch* und *pull* haben zum Großteil die gleiche Syntax um einzelne Zweige oder spezielle *remote* Server als Referenz zu verwenden (4 p. 61ff.).

4.7.5 Git pull

Der Befehl **git pull** wird benötigt um das lokale Projektarchiv mit (falls vorhandenen) Änderungen des Referenzarchiv zu erweitern. Erweitert mit **--all** werde alle Zweige abgeglichen. Mit **git pull origin master** wird nur der Zweig *master* abgeglichen und falls Änderungen vorhanden sind, mit dem lokalen Stand zusammengeführt. Da es hierbei zu Konflikten kommen kann, sollte man nur den *pull* Befehl ausführen, wenn eine frische Arbeitskopie existiert, beziehungsweise, alle Änderungen zuvor einem Commit hinzugefügt wurden. Ist dies nicht der Fall wird automatisch ein weiterer *merge commit* hinzugefügt. Will man sich nur über Neuerungen informieren, ohne Änderungen lokal vorzunehmen, sollte man das Kommando *fetch* nutzen. Pull ist somit eine Kombination aus *fetch* und *merge* (4 p. 61ff.). Ein weiteres Schlüsselwort, welcher dem Befehl angehängen werden kann, ist **--rebase**. Dies führt bei lokalen Abweichungen gegenüber dem Referenzarchiv dazu, dass die lokalen Abweichungen verschoben werden. Im Grunde werden die lokalen Abweichungen bis zur letzten Übereinstimmung entfernt, die Commits der Referenz angewandt und erst danach die ursprünglich lokalen Commits dem Zweig wieder angehängen. Dadurch besteht keine Divergenz zwischen den beiden Zweigen mehr, außer dass der lokale Zweig zusätzliche Commits beinhaltet. Somit ist der lokale Zweig dem Referenzzweig voraus und die Änderungen können nun ohne Probleme synchronisiert werden. Dies ist der Zusammenführung vorzuziehen, da mit dieser Variante keine zusätzlichen Commits benötigt werden, sobald sich der Referenzzweig ändert. Allerdings ändern sich die Metadaten der lokalen Commits, was nicht weiter schlimm sein sollte, da diese nur lokal vorhanden sind (18).

Um die Funktion einzuführen wurde ein Commit dem Referenzarchiv hinzugefügt, daher kann dieser Commit über das Kommando *fetch* nun auch lokal gesichtet werden. In Abbildung 42 sieht man einen neuen Zweig "origin/try" welcher diesen globalen Commit beinhaltet. Das Wort "origin" steht hierbei für Zeiger des Referenzarchivs.

```
labadmin@eitlinux ~/Git $ git fetch origin
labadmin@eitlinux ~/Git $ git log --oneline --all --graph
* 7d450cc (origin/try) Modified project, uses Zeichne()
* f69ef72 (HEAD -> master, origin/master, origin/feature, feature) Added Grafik implementation
* 5a3c6c1 (origin/newshape, newshape) Added Rechteck implementation
* 4beb7c3 Added Kreis
* 3176713 Added Linie to project
* 93e29fa Added class Punkt
* 07b1157 Added main.cpp and make support
* 4e62daf Initial Commit, added README and .gitignore
```

Abbildung 42: Synchronisierung mit dem Referenzarchiv (*fetch*)

Um den globalen Zweig nun lokal zu besitzen wird der Befehl **git checkout try** verwendet. Die Änderungen des Zweigs werden als sinnvolle Erweiterung empfunden und sollen daher auch in den *master* Zweig implementiert werden. Dazu kann neben den bevor behandelten Möglichkeiten auch der Befehl *fetch* verwendet.

Mit dem Kommando: ***git fetch . try:master*** werden alle Commits von *try*, welche nicht auf *master* sind, *master* hinzugefügt. Der Punkt steht hierbei für das lokale Archiv, beziehungsweise woher er die Änderungen bezieht. Dies funktioniert, da der Entwickler den Zweig *try* durch das Kommando *checkout* auch lokal besitzt:

```
labadmin@eitlinux ~/Git $ git fetch . try:master
Von .
f69ef72..7d450cc try      -> master
```

Abbildung 43: Integrierung von Änderungen über den Befehl *fetch*

Nachdem weitere Änderungen hinzugefügt wurden auf *master*, und auch auf *feature*, sollen diese Änderungen wiederum zusammengeführt werden. Diesmal dient die Zusammenführung dazu eine weitere Git Funktion zu präsentieren: *request-pull*.

4.7.6 Git request-pull

In Verbindungen mit Git fällt oftmals der Begriff "Pull-Request". Womöglich ist dies auch der Grund warum Git in den letzten Jahren ohne ernstzunehmende Konkurrenz sich immer weiter ausgebreitet hat. Allerdings hat diese Funktionalität mehrerer Hosts nur wenig mit dem Git internen Kommando ***git request-pull <Start> <URL>*** nur bedingt Gemeinsamkeiten. Durch das Git Kommando werden Differenzen zweier Stände ermittelt und ausgegeben. Mit dieser Ausgabe kann der Ersteller sich an Beteiligte des Projekts wenden. Anhand dieser Informationen weiß der Integrator, der Besitzer des Projektarchivs, welches man erweitern möchte, wo die Änderungen zu finden sind und was diese Änderungen beinhalten. Dadurch wird der Vorgang erleichtert und die Integration beschleunigt. Im Grunde ist es eine andere Form Änderungen einzureichen und kann als Vorläufer eines *patches* angesehen werden, da hier die Entscheidung der Integration grundsätzlich erst noch gefällt werden muss. Im Gegensatz zum Pull-Request von diversen Hosts, welche über das Kommando automatisch Code Reviews, Code Integration und Kommentare einfließen lassen, bietet das *request-pull* Kommando lediglich eine Zusammenfassung von Änderungen. Die ***<URL>*** bezieht sich auf das Archiv, in welchem die Änderungen für den Integrator zu finden sind. Unter ***<Start>*** kann entweder der Anfang der Prüfsumme eines Commits übergeben werden, oder ein Zweig, gegen den man das aktuelle Arbeitsverzeichnis abgleichen will. Dies ist hilfreich, falls man ein *request-pull* für einen kompletten Entwicklungszweig erstellen will, da dadurch automatisch der letzte gemeinsame Vorfahre der beiden Zweige ermittelt wird (19).

Um ein *pull-request* für die Änderungen auf dem *feature* Zweig zu erstellen, schaut man zuerst in die Historie, um die Prüfsumme des gewünschten Commits zu erhalten. In diesem Fall sieht die Historie des Projektarchivs und die Ausgabe des *request-pull* wie folgt aus:

```

labadmin@eitlinux ~/Git $ git log --oneline --all
28ae79a (HEAD -> feature) Added Abbildung implementation
450a66d (master) Added Text to project
7d450cc (origin/try, origin/master, try) Modified project, uses Zeichne()
f69ef72 (origin/feature) Added Grafik implementation
5a3c6c1 (origin/newshape, newshape) Added Rechteck implementation
4beb7c3 Added Kreis
3176713 Added Linie to project
93e29fa Added class Punkt
07b1157 Added main.cpp and make support
4e62daf Initial Commit, added README and .gitignore
labadmin@eitlinux ~/Git $ git request-pull 450a66d .
The following changes since commit 450a66dab078f3062d0bdea9fd6b06afc7b0289d:

    Added Text to project (2019-12-24 14:14:15 +0100)

are available in the Git repository at:
    .
for you to fetch changes up to 28ae79ad510924363dd6eba093a50d4dd32ed4eb:

    Added Abbildung implementation (2019-12-24 14:17:10 +0100)

-----
(1):
    Added Abbildung implementation
Abbildung.cpp | 45 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Abbildung.h   | 25 +++++++++++++++++++++++++++++++++++++
depend        | 3 +-
main.cpp      | 19 ++++++-----
4 files changed, 84 insertions(+), 8 deletions(-)
create mode 100755 Abbildung.cpp
create mode 100755 Abbildung.h

```

Abbildung 44: Ausgabe des Kommandos *request-pull*

Durch die Übergabe des Punktes besitzt die Ausgabe keinen expliziten Link, wo die Änderungen zu finden sind, daher muss man für die Weitergabe an Weitere an dieser Stelle einen öffentlichen Link zu dem persönlichen Referenzarchiv hinzufügen, damit die Empfänger diese auch abrufen können. Auch sollte das Referenzarchiv synchronisiert zum lokalen Archiv sein. Zuerst wird der Zweig *feature* des Referenzarchivs über den Befehl *git push origin feature* aktualisiert. Ist dies geschehen kann die Zusammenführung beider Zweige, da diese sich im gleichen Projekt befinden, gemäß Abbildung 45 integriert werden.

```

labadmin@eitlinux ~/Git $ git pull origin feature:master
Von https://github.com/_____/Git
450a66d..28ae79a feature -> master
Bereits aktuell.
labadmin@eitlinux ~/Git $ git log --oneline -3
28ae79a (HEAD -> feature, origin/feature, master) Added Abbildung implementation
450a66d Added Text to project
7d450cc (origin/try, origin/master, try) Modified project, uses Zeichne()

```

Abbildung 45: Integrierung von Änderungen über den Befehl *pull*

Hierbei werden abweichende Commits von *feature* auf den Zweig aufgespielt. Eine weitere Möglichkeit bietet ein einfacher *fast-forward merge*, sofern *master* der aktive Zweig ist:

```

labadmin@eitlinux ~/Git $ git merge origin/feature
Aktualisiere 450a66d..28ae79a
Fast-forward
Abbildung.cpp | 45 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Abbildung.h   | 25 +++++++++++++++++++++++++++++++++++++
depend        | 3 +-
main.cpp      | 19 ++++++-----
4 files changed, 84 insertions(+), 8 deletions(-)
create mode 100755 Abbildung.cpp
create mode 100755 Abbildung.h

```

Abbildung 46: Fast-forward Zusammenführung anstelle des Befehls *pull*

Dadurch ist das Projekt nach Ansicht des Entwicklers an einem Stand angekommen, welcher veröffentlicht werden kann. Damit alle Beteiligten des Projekts diese Information erhalten und den spezifischen Stand später einfacher referenzieren können, bietet Git eine Möglichkeit, spezielle Commits über das Kommando *tag* mit Etiketten zu versehen.

4.8 Veröffentlichungen kennzeichnen

Für eine schnelle Wartung und Wiederherstellung produktiv angewendeter Versionen, muss eine Referenz auf diese Stände leicht und schnell zu finden sein.

4.8.1 Git tag

Mit Etiketten markiert man einzelne Commits. Damit können zum Beispiel wichtige Versionen hervorgehoben werden. Oftmals erhalten je nach Konfigurationsmanagement Releases und gewisse wichtige Versionen ein Etikett. Neben der reinen Kennzeichnung eines Commits beschleunigt es auch den Rücksprung auf eben diesen Commit, da der Name des Etiketts ausreicht, um die Arbeitsmappe damit zu laden. Git unterscheidet dabei zwischen kommentierten und nicht kommentierten Etiketten. Um ein Etikett zu erstellen wird *git tag <Name>* verwendet. Um zusätzlich eine Nachricht anzufügen kann *-a -m '<Nachricht>'* angehängt werden. Um sich die Liste aller vorhandenen Etiketten anzusehen, wird der Befehl ohne weitere Übergaben ausgeführt. Ein Etikett behandelt Git grundsätzlich als Verweis auf einen bestimmten Commit. Damit das Etikett auch im Referenzarchiv sichtbar wird, muss das Etikett wie ein Commit mit dem Archiv synchronisiert werden. Als Beispiel kann ein Befehl zur Synchronisierung wie folgt aussehen: *git push origin v1.0*. Dies synchronisiert das Etikett mit dem Namen "v1.0" mit dem Referenzarchiv und ist damit für alle sichtbar, sobald diese ihre lokalen Referenzen erneuern. Um weitere Information über ein Etikett zu erhalten, kann ein Etikett dem Befehl *show* übergeben werden (20).

In diesem Fall ist der 3. Meilenstein erreicht und das Programm zum ersten Mal funktionell einsetzbar. Daher wird ein Etikett mit dieser Information im Anhang erstellt, zu sehen in Abbildung 47. Diese Informationen können danach mit *show* ausgelesen werden.

```
labadmin@eitlinux ~/Git $ git tag -a v0.4 -m 'version 0.4 Meilenstein 3 erreicht'
labadmin@eitlinux ~/Git $ git tag
v0.4
labadmin@eitlinux ~/Git $ git show v0.4
tag v0.4
Tagger: <[redacted]>@hs-pforzheim.de>
Date:   Tue Dec 24 14:32:52 2019 +0100

version 0.4 Meilenstein 3 erreicht
```

Abbildung 47: Erstellen eines Etiketts

4.8.2 Git show

Mit **git show** werden Informationen zum letzten Commit des aktiven Zweiges, beziehungsweise der Commit, auf welchen HEAD zeigt, angezeigt. Um Informationen über einen einzelnen Commit zu erhalten kann dessen Prüfsumme übergeben werden. Genauso können Etiketten angezeigt werden. Übergibt man dem Kommando die zuvor erstellte Etiketle, zu sehen in Abbildung 48, wird neben der Information zur Etiketle auch der Commit angezeigt, auf welchen die Etiketle verweist (21).

```
labadmin@eitlinux ~/Git $ git show v0.4
tag v0.4
Tagger: <[redacted]>@hs-pforzheim.de>
Date:   Tue Dec 24 14:32:52 2019 +0100

version 0.4 Meilenstein 3 erreicht

commit 28ae79ad510924363dd6eba093a50d4dd32ed4eb (tag: v0.4, origin/feature, feature)
Author: <[redacted]>@hs-pforzheim.de>
Date:   Tue Dec 24 14:17:10 2019 +0100

    Added Abbildung implementation
```

Abbildung 48: Ausgabe eines Etiketts über das Kommando *show*

4.9 Erstellung einer Änderungsdatei

Im Zuge von vorhandenen Änderungen wird nun eine weitere Art Änderungen zu implementieren verwendet. Es werden nun Änderungen der Arbeitskopie hinzugefügt, anhand einer Datei, welche die gewünschten Änderungen beinhaltet. Diese Datei wird gemäß Abbildung 49 erstellt.

```
labadmin@eitlinux ~/Git $ git diff v0.4 HEAD > patch.diff
```

Abbildung 49: Erstellung einer Änderungsdatei mit dem Befehl *diff*

Hierbei wird die Ausgabe des Befehls *diff* zwischen dem Commit auf welchen das Etikett, sowie der aktuellen Arbeitsmappe, in eine Datei umgeleitet. Diese Datei kann wiederum von Git gelesen werden und entsprechend auf die Arbeitsmappe angewandt werden.

4.9.1 Git diff

Mit dem simplen Kommando **git diff** werden die Dateiänderungen in der Arbeitskopie angezeigt, von Dateien, welche nicht im Index Bereich sind. Neu hinzugefügte Dateien werden darunter nicht berücksichtigt. Um auch eine Differenz für Dateien im Index zu erhalten muss das Schlüsselwort **--staged** übergeben werden. Ähnlich wie das Bash-Kommando *diff*, können auch hier einzelne Dateien, Zweige oder Commit-Hashes mitgegeben werden. Auch ist **git diff** für die Erstellung von Patches hilfreich. Dies kann zum Beispiel geschehen über **git diff --staged > mypatch.patch**. Es gilt zu beachten, dass *diff* nicht jede Datei mit einbezieht, für binäre Dateien muss zum Beispiel **--binary** als Schlüsselwort übergeben werden (22).

Um die erstellte Datei wieder auf eine Arbeitskopie anzuwenden, wird das Kommando *apply* verwendet. Wie in Abbildung 50 zu sehen, besitzt die Arbeitsmappe zuvor keine Änderungen. Nachdem die Änderungsdatei angewendet wurde, befinden sich allerdings zwei Änderungen in der Arbeitskopie:

```
labadmin@eitlinux ~/Git $ git status
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'.

nichts zu committen, Arbeitsverzeichnis unverändert
labadmin@eitlinux ~/Git $ git apply /home/labadmin/Downloads/patch.diff
labadmin@eitlinux ~/Git $ git status
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'.

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

    geändert:      depend
    geändert:      main.cpp

keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git commit -a")
```

Abbildung 50: Anwendung einer Änderungsdatei auf die Arbeitsmappe

Die ursprüngliche Art eine Änderungsdatei zu erstellen, bietet der Befehl *format-patch*.

4.9.2 Git patch

Über das *patch* Kommando werden ähnlich zu einem *pull-request* von Host-Anbieter schnell größere Änderungen übertragen und integriert. Um einen *patch* zu erstellen wird ursprünglich *git format-patch -1 <Prüfsumme>* verwendet. Die *-1 <Prüfsumme>* bedeutet hierbei, dass lediglich der übergebene Commit in den *patch* einfließt. Um Zweige zu vergleichen und somit jeden abweichenden Commit in einen *patch* zu verpacken, welcher nicht auf dem aktiven Zweig ist, muss lediglich dieser Zweig *format-patch* übergeben werden. Soll eine einzige ".patch" Datei erstellt werden, anstatt eine Datei für jeden Commit wird *--stdout* dem Befehl angehängen. Die *patch* Datei erhält je nach Git Konfiguration einen vordefinierten Header, welcher ursprünglich die Verschickung per Mail direkt möglich machte:

```
From: eeb2ec2c60e25721566e866c7dbbe4d7dfdfc2a5 Mon Sep 17 00:00:00 2001
From: @hs-pforzheim.de>
Date: Tue, 24 Dec 2019 14:34:57 +0100
Subject: [PATCH] Added function foo()

---
depend | 2 +-
main.cpp | 31 ++++++-----
2 files changed, 20 insertions(+), 13 deletions(-)
```

Abbildung 51: Kopfzeile einer Änderungsdatei über den Befehl *patch*

Der Empfänger hat nun die Möglichkeit die empfangen Dateien über *git am <Änderungsdatei>* seinem Projektarchiv hinzufügen. Diese werden direkt als Commit hinzugefügt. Die resultierende Datei kann jedoch auch über *git apply <Änderungsdatei>* auf die aktuelle Arbeitsmappe angewendet werden (23).

Wird anstatt dem *diff* Befehl in Abbildung 49, *format-patch* verwendet, erstellt Git automatisch eine nummerierte Datei:

```
labadmin@eitlinux ~/Git $ git checkout enhancement
Zu Branch 'enhancement' gewechselt
labadmin@eitlinux ~/Git $ git format-patch master
0001-Added-function-foo.patch
```

Abbildung 52: Erstellung einer Änderungsdatei mit *format-patch*

Diese Datei kann damit auf *master* angewandt werden, siehe Abbildung 53, um die Änderungen von *feature* zu integrieren. Da die Datei über *format-patch* erstellt wurde, wird der Commit direkt dem Zweig angefügt, anstatt die Änderungen in die Arbeitskopie zu übernehmen. Daher sollte vor der Anwendung eines *patches* die hinzukommenden Änderungen auf Korrektheit überprüft werden. Mit dem Schlüsselwort *--check* wird lediglich geschaut, ob eine Anwendung ohne Probleme möglich ist.

```
labadmin@eitlinux ~/Git $ git checkout master
Zu Branch 'master' gewechselt
Ihr Branch ist auf demselben Stand wie 'origin/master'.
labadmin@eitlinux ~/Git $ git am /home/labadmin/Downloads/0001-Added-function-foo.patch
Wende an: Added function foo()
labadmin@eitlinux ~/Git $ git status
Auf Branch master
Ihr Branch ist 1 Commit vor 'origin/master'.
(benutzen Sie "git push", um lokale Commits zu publizieren)

nichts zu committen, Arbeitsverzeichnis unverändert
```

Abbildung 53: Automatische Anwendung einer Änderungsdatei

4.10 Revidieren von Commits

Zum Abschluss des Beispielprojekts werden weitere Änderungen durchgeführt, allerdings enthält der letzte Commit einen großen Fehler. Um einen vollführten Commit zu revidieren kann der Befehl *revert* verwendet werden. Bei Korrektur zuvor erfolgter Commits ist erhöhte Achtsamkeit von Nöten, da dadurch möglicherweise globale Referenzen verloren gehen, sollten die Commits synchronisiert sein, und Entwickler somit an nicht mehr existierenden Referenzen weiterarbeiten. Diese Problematik ist nur sehr schwer zu beheben.

4.10.1 Git revert

Das Kommando *git revert HEAD* wird dazu benötigt einen durchgeführten Commit zu übergehen. Hierfür wird ein neuer Commit durchgeführt, welcher den Stand vor dem fehlerhaften Commit widerspiegelt. Dies kann auch erweitert werden indem die Prüfsumme oder respektive *HEAD~<Anzahl an Commits>* übergeben wird. Dadurch ist der ungewollte Commit zwar in der Historie vorhanden, besitzt aber keinen Auswirkungen mehr auf den aktuellen Stand des Zweiges. Diese Methodik wird vor allem für Commits

angewandt, welche zuvor über den Befehl **git push** mit dem Referenzarchiv synchronisiert wurden. Ist dies geschehen, können keine Commits gelöscht werden, da Git die fehlenden Commits bemerkt und daraus schließt, der lokale Stand wäre nicht aktuell. Dies lässt sich zwar durch einen sogenannten *force push* umgehen, allerdings werden hier Commits entfernt, auf dessen andere Nutzer des Projektarchivs gegebenenfalls Verweise besitzen. Deshalb ist diese Methode nicht weit verbreitet und es wird generell davon abgeraten. Um den Befehl *revert* anwenden zu können, darf die Arbeitsmappe keine Änderungen besitzen. Mit dem Schlüsselwort **--no-commit** wird der automatische Commit zurückgehalten und man kann zusätzliche Änderungen an der Arbeitsmappe vornehmen (24).

In diesem Beispiel soll der letzte Commit revidiert werden, da er die Ausgabe fehlschlagen lässt. Da auch der Zweig *master* des Referenzarchivs diesen besitzt, ist es sicherer diesen zu revidieren, falls jemand diesen bereits referenziert. Die Historie sieht zuvor gemäß Abbildung 54 aus:

```
labadmin@eitlinux ~/Git $ git log --oneline
70b23a5 (HEAD -> master, origin/master) Modified main(), new debug outputs
98770d6 Added function foo()
```

Abbildung 54: Historie vor Revidierung eines Commits

Wird nun der Befehl **git revert HEAD** ausgeführt, zu sehen in Abbildung 55, öffnet sich der Editor, um die Commit Nachricht anzupassen. Ist der Editor geschlossen wird der revidierende Commit vollzogen. Die Historie zeigt den neuen Commit.

```
labadmin@eitlinux ~/Git $ git revert HEAD
Hinweis: Warte auf das Schließen der Datei durch Ihren Editor... Icon theme "adwaita" not found.
[master 98e4574] Revert "Modified main(), new debug outputs"
 1 file changed, 17 insertions(+)
labadmin@eitlinux ~/Git $ git log --oneline
98e4574 (HEAD -> master) Revert "Modified main(), new debug outputs"
70b23a5 (origin/master) Modified main(), new debug outputs
98770d6 Added function foo()
```

Abbildung 55: Historie nach Revidierung eines Commits

Ist der fehlerhafte Commit nur lokal vorhanden bieten sich einfachere Möglichkeiten diesen zu löschen, anstatt den fehlerhaften Commit über einen weiteren Commit zu revidieren. Hierbei wird der Befehl *reset* benötigt, um den Commit zu verwerfen.

4.10.2 Git reset

Falls man mehrere Änderungen lokal durchgeführt hat, welche sich als fehlerhaft herausstellten, kann ein Zurücksetzen des lokalen Arbeitsverzeichnisses Abhilfe schaffen. Dies geschieht mit **git reset --hard <Prüfsumme>**. Dadurch werden alle durchgeführten Änderungen bis zu dem übergebenen Commit verworfen. Ohne das Schlüsselwort **--hard**, finden sich die Änderungen, welche die verworfenen Commits beinhalten in der

Arbeitskopie wieder. Falls kein Commit mit übergeben wird, wird die Arbeitskopie auf den letzten Commit des aktiven Zweiges zurückgesetzt. Über **git reset <Datei>** kann außerdem eine Datei aus dem Index entfernt werden. Die Änderungen der Datei bleiben aber weiterhin erhalten, solange **--hard** nicht mit übergeben wird (24).

Fall der Commit nicht synchronisiert wurde, ist es einfacher den Commit zu entfernen über das **reset** Kommando, anstatt ein **revert** durchzuführen. In Abbildung 56 zu sehen, wird zuerst ein **reset** ohne das Schlüsselwort **--hard** verwendet. Dadurch befindet sich die modifizierte Datei, welche den ungewünschte Commit beinhaltet, im Arbeitsverzeichnis. Ein zweites Mal wird der Befehl mit diesem Schlüsselwort verwendet. Die Arbeitskopie ist nach erfolgtem Befehl auf dem Stand des übergebenen Commits und die Änderungen sind verworfen:

```
labadmin@eitlinux ~/Git $ git reset 98e4574
Nicht zum Commit vorgemerkte Änderungen nach Zurücksetzung:
M      main.cpp
labadmin@eitlinux ~/Git $ git reset 98e4574 --hard
HEAD ist jetzt bei 98e4574 Revert "Modified main(), new debug outputs"
```

Abbildung 56: Zurücksetzung der Historie durch das Kommando **reset**

Da der Entwickler festgestellt hat, dass es nur ein kleiner Fehler war, will er den verworfenen Commit wiederverwenden, jedoch mit der korrigierten Änderung, welchen den Fehler behebt. Jedoch hat er den Commit über **git reset --hard** verworfen. Dieser taucht somit nicht mehr in der Historie über den Befehl **log** auf. Um den Commit in solch einem Fall wieder zu erhalten, hilft der Befehl **reflog**.

4.10.3 Git reflog

Während **git log** die Historie erstellt indem die Commits verfolgt werden, sozusagen von Elternteil zu Elternteil gesprungen wird, dokumentiert **reflog** jegliche Änderungen des HEAD Zeigers. Dadurch enthält der **reflog** Verweise auf Commits und Stände welche möglicherweise sonst nicht mehr sichtbar sind, da kein bestehender Commit auf diese verweist. Deshalb wird **git reflog** häufig in Verbindung mit **git reset HEAD@{x}** verwendet, wobei **HEAD@{x}** aus dem **reflog** entnommen wird. Auch falls ein Commit durch ein **reset** Kommando verloren ging, ist dieser nicht gelöscht, sondern nur ohne derzeitigen Verweis. Auch in diesem Fall ist die Wahrscheinlichkeit hoch, den verworfenen Commit im **reflog** zu finden. Allerdings werden Commits ohne Referenzierung nach einer festgelegten Zeit verworfen. Somit ist die Rückverfolgung von **reflog** zeitlich begrenzt (25).

Öffnet man den *reflog*, siehe Abbildung 57, besitzt dieser den verworfen Commit: "New debug output support". Mit `git reset HEAD@{2}` wird somit die Historie auf diesen Stand zurückgesetzt und die nachfolgenden *reset* Befehle sind nur noch im *reflog* vorhanden. Auch hierbei werden durch `--hard` alle Änderungen entfernt, beziehungsweise die geänderten Dateien nicht in der Arbeitskopie abgelegt.

```
labadmin@eitlinux ~/Git $ git reflog
98e4574 (HEAD -> master, origin/master) HEAD@{0}: reset: moving to 98e4574
98e4574 (HEAD -> master, origin/master) HEAD@{1}: reset: moving to 98e4574
d8d39b3 HEAD@{2}: commit: New debug output support
```

Abbildung 57: Ansicht des Reflogs

4.11 Selektiv Änderungen durchführen

Im Verlauf des Beispielprojekts wurden mehrere Möglichkeiten einer Integration von Änderungen gezeigt. Neben den Kommandos zur Zusammenführung (*merge* und *rebase*), wurden Änderungen auch von anderen Zweigen synchronisiert (*fetch* und *pull*). Zuletzt wurde behandelt, wie Änderungen in Form von *patch* Dateien übertragen werden können. Eine weitere Form einer Übernahme von Änderungen bietet *cherry-pick*.

4.11.1 Git cherry-pick

Das Kommando `git cherry-pick <Prüfsumme>` ähnelt dem *rebase* Kommando. Es wird dazu verwendet einen einzelnen Commit auf einen anderen Zweig anzuwenden, im Gegensatz zu einem gesamten Zweig. Dadurch können zum Beispiel Korrektur Commits schnell und einfach in andere Zweige eingegliedert werden. Nachteil hierbei ist, dass der vorhandene Commit in Form eines *patch* auf den aktiven Zweig angewandt wird. Dadurch entfällt die Information, woher der Commit stammt. Er wird als ein neuer Commit behandelt. Um trotzdem eine Information über die Herkunft zu erhalten wird `-x` verwendet. Dies hängt an die Commit Nachricht eine weitere Zeile an, welche auf den ursprünglichen Commit verweist. Auch hier können Schlüsselworte wie `--edit` oder `--no-commit` angewandt werden (26). Bildlich sieht eine Integration über das Kommando *cherry-pick* wie folgt aus:

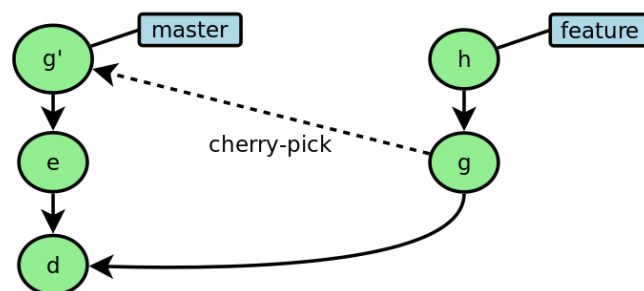


Abbildung 58: Cherry-pick eines Commits (35)

In Abbildung 58 wurde der Commit "g" dem Zweig *master* hinzugefügt. Dieser ist dort allerdings nicht unter dem Verweis "g" gespeichert. Es wird ein neuer Commit "g'" erstellt. Da der Verlust der Herkunft problematisch sein kann, wurde dafür ein weiterer Befehl namens *cherry* eingeführt, welcher hilft, solche Integrationen nachzuvollziehen.

4.11.2 Git cherry

Um nachvollziehen zu können ob Änderungen einem Zweig zugeführt wurden, kann **git cherry -v <Zweig1> <Zweig2> <Limit>** verwendet werden. Das Kommando überprüft die eingeführten Änderungen jedes Commits, anstatt eine übereinstimmende Prüfsumme, wie der Befehl *branch --contains*. Dadurch werden auch Commits, welche über *cherry-pick* oder *patches* zugeführt wurden, erkannt. Das Kommando überprüft dabei, ob die Commits aus <Zweig2> in <Zweig1> existieren. Das Limit gibt an, wie viele Commits rückwirkend durchsucht werden sollen. Dies ist hilfreich, um die Ausgabe einzugrenzen. Mit -v wird neben der Prüfsumme auch die jeweilige Commit Nachricht angezeigt. Der Befehl erzeugt eine Liste der Commits aus <branch2> welche durch ein Pluszeichen oder Minuszeichen angeführt werden. Ein Minuszeichen steht für ein übereinstimmende Änderungen. Das Pluszeichen für fehlende Änderungen in <Zweig1> (27).

Als Beispiel betrachtet man die Historie der Zweige *enhancement* und *master* aus dem Beispielpjekt. In Abbildung 59 zu sehen, besitzen beide Zweige den Commit: "Added function foo()". Der Commit wurde dem Zweig *master* über einen *patch* hinzugefügt. Daher besteht für Git keine Verbindung zwischen dem Commit in *master* und *enhancement*.

```
labadmin@eitlinux ~/Git $ git log --oneline --all
98e4574 (HEAD -> master, origin/master) Revert "Modified main(), new debug outputs"
70b23a5 Modified main(), new debug outputs
98770d6 Added function foo()
eeb2ec2 (enhancement) Added function foo()
```

Abbildung 59: Der Commit "Added function foo()" existiert in beiden Zweigen

Die Ausgabe des Befehls *cherry* erkennt allerdings die übereinstimmenden Änderungen und markiert den Patch-Commit aus *master* als integriert an, indem vor diesem ein Minuszeichen steht:

```
labadmin@eitlinux ~/Git $ git cherry -v enhancement master
- 98770d62647e838223eba24b11901d272d681ce9 Added function foo()
+ 70b23a54858be5bec97008cf6606aaale443e31e Modified main(), new debug outputs
+ 98e4574d413361e2c275fced8693b3522de6a41 Revert "Modified main(), new debug outputs"
```

Abbildung 60: Ausgabe des Kommandos *cherry*

5 Verbreitete Arbeitsabläufe

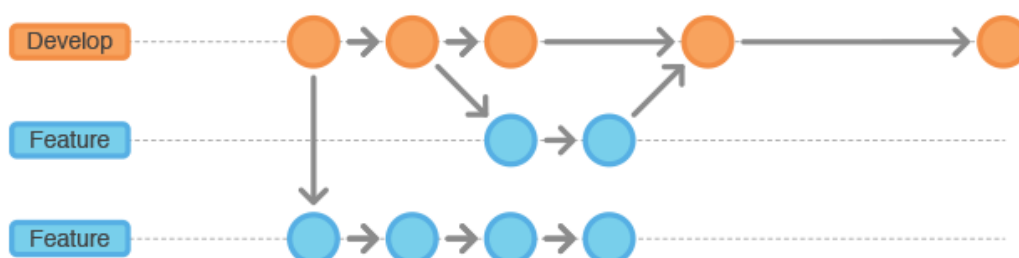
Im Folgenden werden mehrere Vorgehensweisen behandelt, welche die Eingliederung von Entwicklungszweigen und den Ablauf von Zusammenführungen vorschreiben. Zuvor wurden diverse Befehle beleuchtet, welche das Arbeiten mit mehreren Entwicklern möglich macht. Über die vergangenen Jahre haben viele Git nutzende Projektgruppen die für ihre Zwecke sinnvolle Arbeitsweise entwickelt und verfeinert. Grundsätzlich gibt es drei größere Arbeitsweisen, welche sich als effizient herausgearbeitet haben. Diese werden falls benötigt nach Anforderungen des jeweiligen Projekts angepasst und abgewandelt.

5.1 Branching Workflow (GitFlow)

Im Allgemeinen wird GitFlow als die Richtlinie für ein gut durchdachtes und schlank strukturiertes Projektarchiv angesehen. Hierbei wird der generelle Fokus auf die Separierung der Tätigkeiten in feste Zweige gelegt. So besitzt ein Projekt grundsätzlich fünf Arten von Zweigen. Diese Arten können wiederum unterteilt werden in Zweige welche einmalig sind und Arten von Zweigen, welche beliebig oft im Archiv vorkommen dürfen. Eine Art der einmaligen Zweige ist der Hauptzweig, welcher standardmäßig *master* heißt. Dieser wird nur für fertige Veröffentlichungen verwendet. Dadurch ist der aktuelle Stand sehr einfach zu finden und kann schnell und ohne Überlegungen gesichtet werden. Ein weiterer einmaliger Zweig ist der Entwicklungszweig oder auch *development branch* genannt. Dieser führt alle Änderungen und Neuerungen des Projekts zusammen. Unter Umständen kann bei einem größeren Projekt diese Funktion auf mehrere Zweige unterteilt werden. Trotz alledem sollte am Ende ein Hauptentwicklungszweig bestehen, welcher alle Änderungen, sofern diese den Ansprüchen genügen, integriert. Dieser Zusammenschluss von einsatzbereiten Änderungen wird in einen weiteren einmaligen Zweig überführt. Dieser Zweig wird *release* genannt und beinhaltet Änderungen welche für die nächste Hauptversion, also für den nächsten Commit auf dem *master* Zweig, verwendet werden sollen. Somit sollten fertige Funktionen von dem Hauptentwicklungszweig auf den Zweig für Veröffentlichung überführt werden. Die Definition des *release* Zweiges lässt auch noch weitere Änderungen und Vorbereitungen zu einer Veröffentlichung auf diesem Zweig zu. Man sollte allerdings beachten, dass alle Änderungen, welche über den Release Zweig eingeführt werden, spätestens nach der Erstellung einer Veröffentlichung, auch in den Entwicklungszweig integriert werden. Der Entwicklungszweig sollte immer auf dem Stand der neusten Veröffentlichung aufbauen und den aktuellen Stand der Entwicklung darstellen. Somit lässt sich der bisherige Ablauf des Projekts wie in Abbildung 61 darstellen.

Abbildung 61: Die Zweige *master*, *release* und *develop* in Gitflow (35)

Führt man die Entwicklung fort, wird der einzelne Entwicklungszweig schnell überfüllt und unübersichtlich, sobald mehrere Personen an dem Projekt arbeiten. Auch gestaltet sich die gegenseitige Synchronisierung untereinander als Konfliktanfällig und Zeitintensiv. Daher werden nach dem Schema GitFlow weitere Zweige erstellt, welche auf dem Entwicklungszweig basieren. Diese Art von Zweigen wird *feature branch* genannt. Da es von dieser Art mehrere Zweige geben soll, wird dafür oftmals eine projektspezifische Namenskonvention angewendet. Im Allgemeinen wird einfach mit der Art des Zweiges gestartet und die zu implementierende Funktion angehängt, zum Beispiel: "*feature-gui*". Dadurch können einzelne Personen an ihrem eigenen Zweig arbeiten oder eine Gruppe von Personen an einer einzelnen Funktion. Folglich wird der Hauptentwicklungszweig übersichtlicher und die Arbeit an dem Projekt gestaltet sich für die einzelnen Entwickler deutlich einfacher, da dieser separiert von Änderungen anderer Personen und Themen, arbeiten. Bei der Integration eines *feature* Zweigs in den Hauptentwicklungszweig wird im allgemeinen Fall ein ***merge --no-ff*** verwendet, sodass immer ein expliziter Commit erstellt wird. Auch wird dies bei allen anderen Zusammenführungen verwendet. Hiermit wird deutlich, welche Commits zusammen integriert wurden, beziehungsweise welche Commits eine geschlossene Funktion implementieren. Auch ist die Handhabung im Fall, dass die hinzugefügte Funktion nicht mehr erwünscht ist, einfacher. Hierbei kann der einzeln zusammengefasste Commit revidiert werden, anstatt jeder Commit, welche bei der Integration des ungewünschten *feature* Zweigs hinzukam. Grafisch lässt sich der Ablauf zwischen Haupt und Nebenentwicklungszweigen wie folgt darstellen:

Abbildung 62: Vorgehensweise bei *feature* Zweigen in GitFlow

Als letzte Art von Zweigen fehlt noch der Korrektur Zweig, welcher allgemein unter *bugfix branch* bekannt ist. Dieser Zweig dient dazu kleine Änderungen direkt an einer Veröffentlichung durchzuführen, ohne Einfluss von der bereits vorangeschrittenen Entwicklung zu besitzen. Daher zweigt dieser Zweig direkt von *master* ab und baut nur auf die gewünschte Veröffentlichung auf. Dies wird hauptsächlich im Aspekt der Wartung einer Version verwendet. Auch hier ist es empfehlenswert die eingeführten Änderungen ebenso in den Hauptentwicklungszweig zu integrieren, falls die Korrektur auch in späteren Ständen angewandt werden kann. Sollte auch eine Veröffentlichung in Arbeit sein, empfiehlt es sich auch hier die Korrektur einzufügen. Falls sich keine neuere Veröffentlichung auf dem Hauptzweig befindet, kann die korrigierte Version als weitere Veröffentlichung diesem zugefügt werden. Liegt die Veröffentlichung allerdings weiter zurück ist dies nicht empfehlenswert, da sonst der Hauptzweig nicht mehr linear mit den Versionen des Projekts fortschreitet. Für solch einen Fall, bei dem mehrere Versionen des Produkts gleichzeitig unterstützt werden sollen, bietet GitFlow ursprünglich keine Unterstützung. Hier gibt es allerdings Abwandlungen, welche für jede Variante, welche zeitgleich unterstützt werden soll, einen eigenen *master* Zweig besitzt. Dadurch können solche Korrekturen diesem Zweig immer angehängt werden und stellen dadurch die neuste Version dieser Produktvariante dar. Falls das Projekt die hier dargestellten Arbeitsweisen unterstützt, kann die Implementation erleichtert werden indem die hierfür erstellte Erweiterung verwendet wird. Dadurch werden automatisch Abläufe wie das Erstellen von Zweigen und die Durchführung einer Zusammenführung vereinfacht. Diese Erweiterung sollte nativ unterstützt sein und wird über *git flow* aufgerufen. Die Erweiterung hilft dabei die möglichen Arten von Zweigen vereinfacht zu erstellen, da viele Information dem Schema entnommen werden, zum Beispiel welcher Zweig als Vorlage für einen neuen Nebenentwicklungszweig dient.

5.2 Historischer Workflow (Linux Kernel)

Da Git ursprünglich für die Entwicklung des Linux Kernels erstellt wurde, besitzt es dahingehende einen für diesen Zweck optimierten Umfang. Die Kernelentwicklung setzt bis heute auf einen E-Mail-Verteiler, um die daran arbeitenden Personen über Änderungen zu informieren und später diese dem Projekt hinzuzufügen. Aufgrund dessen besitzt Git bis heute Kommandos, welche diese Arbeitsweise unterstützen, wie zum Beispiel *format-patch* und *request-pull*. Das Prinzip hinter der Arbeitsweise wird grundlegend auch im Forking Workflow verwendet, indem Änderungen zuerst vorgeschlagen werden und falls akzeptiert, nur gewisse Personen diese der Hauptentwicklung zuführen dürfen. Diese Verwalter fungieren neben der Integrationsarbeit auch als Kontrolleure um einen sicheren und sauberen Quellcode zu gewährleisten. Im Linux Kernel Projekt wird dieses Prinzip auf die Spitze getrieben, indem einzig Linus Torvalds zwischenzeitlich dem Hauptpfad Änderungen hinzufügen konnte. Da das Projekt eine be-

achtliche Größe besitzt, werden die E-Mail-Verteiler sowie die Verwalter auf mehrere Ebenen und Untergruppen aufgespaltet. Diese Arbeitsweise ist in Abbildung 63 beispielhaft dargestellt.

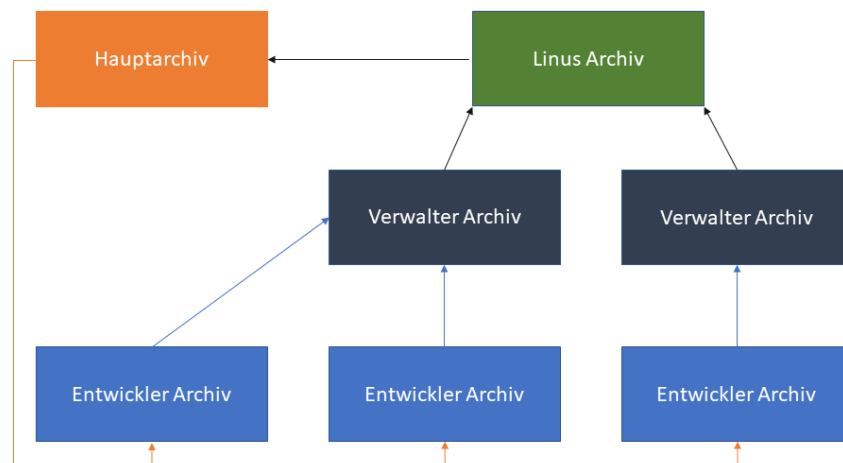


Abbildung 63: Hierarchie in der Linux Kernel Entwicklung

Dabei empfiehlt es sich, diese Separierung nach Themen oder Modulen durchzuführen. Dies kann bei kleineren Projekten eingespart werden. Neben dem automatischen Code-Review bei der Integration durch den Verwalter, wird durch diese Form des Austausches auch eine Diskussion über die vorgeschlagenen Änderungen möglich. Dieses Prinzip des Kommandos *request-pull* wurde durch verschiedene Host-Anbieter aufgegriffen und verfeinert. Daraus entstand das Schlagwort für Git: Pull-Request. Diese Funktion bietet neben einer reinen Zusammenfassung von Änderungen durch weitere Schichten innerhalb des Anbieters weitaus mehr Funktion und generiert dadurch einen automatischen Workflow. Wie in Kapitel 4.9.2 beschreiben, wird durch den Befehl *format-patch* neben der Änderungsübersicht auch eine E-Mail-Kopfzeile angefügt. Dies macht es möglich die durch *request-pull* vorgeschlagenen und im Optimalfall bestätigten Änderungen unkompliziert innerhalb der Kommandozeile in einen Patch zu verwandeln und diesem dem Integrator zur Verfügung zu stellen. Zu dem Zeitpunkt der Entstehung von Git, war das Informationsmedium E-Mail die aktuelle Form des Austausches. Heutzutage werden dieselben Ansätze verwendet, allerdings findet der Austausch nun über globale oder unternehmensinterne Host-Services statt, wie in 5.3 beschrieben. Sollte man allerdings einen solchen Dienst nicht in Anspruch nehmen, kann über diese ältere aber bewerte Arbeitsweise, Änderungen besprochen und die daraus ableitenden Änderungsdateien entwickelt und angewendet werden.

5.3 Forking Workflow

Als Abwandlung des bewerten Linux Workflows haben verschiedene Anbieter ihre eigenen Implementationen dessen angestrebt und verwirklicht. In der Open Source Community hat die nachfolgende Arbeitsweise große Unterstützung gefunden, da sie nicht auf direkte Zusammenarbeit angewiesen ist und Mitwirkende unter Umständen keinen Einfluss auf das Referenzarchiv besitzen. Grundlegen sollte hier noch einmal unterschieden werden. Im Falle einer Open Source Software besitzen ein Großteil der Mitwirkenden keine Berechtigungen am Projekt. Da diese Projekte allerdings im vollen Umfang zur Verfügung stehen, können diese eine lokale Kopie des Projekts erstellen. Diese Kopie wird als *Fork* bezeichnet. In dieser lokalen Kopie kann entsprechend isoliert zum eigentlichen Projekt gearbeitet werden. Im Normalfall wird für Änderungen, welche zurückgeführt werden, ein separater Zweig erstellt, welcher die zu bearbeitende Funktionalität widerspiegelt. Dadurch können zwischenzeitliche Änderungen am ursprünglichen Archiv auch lokal synchronisiert werden, ohne dass diese im fortschreitenden Verlauf kollidieren. Sind die Änderungen für eine Integration in das öffentliche Referenzarchiv bereit, werden diese den Verwaltern über einen Pull-Request vorgeschlagen. Zuvor sollte man den eigens erstellten Zweig auf den aktuellen Stand des Hauptarchivs aktualisieren. Aus Gründen der Übersichtlichkeit und da bei einem Pull-Request aus einem Fork heraus, die Historie des Autors vernachlässigt werden kann, sollte die Aktualisierung durch ein *Rebase* geschehen. Die Verwalter können über die Service-Schicht des jeweiligen Hosting-Anbieters mit diesen vorgeschlagenen Änderungen interagieren. Diese Funktion gleicht somit dem E-Mail-Verteiler. Innerhalb der Umgebung Pull-Request können nun, unkomplizierter als per Mail, Diskussionen geführt und Verbesserungsvorschläge vorgenommen werden. Falls die Änderungen überarbeitet werden sollen, werden neue Commits auf dem Quellzweig des Pull-Request, diesem automatisch angehängt. In dieser Situation zeichnet sich ein weiterer Vorteil gegenüber dem E-Mail-Verkehr ab, da keine weiteren Verteiler und Patches erstellt werden müssen. Sind die Hürden, welche je nach Projekt differieren, überwunden, können die Änderungen über mehrere Optionen integriert werden. Bei einem *Fork* spielt dies allerdings keine größere Rolle, da eine genaue Nachverfolgung aufgrund der Gegebenheiten nicht möglich ist (28). Nicht nur bei sogenannten *Forks* können und werden Pull-Requests eingesetzt. Auch innerhalb eines Projektteams kann diese Funktionalität dazu verwendet werden einzelne Zweige zusammenzuführen, obwohl die Entwickler Zugriff auf die Git eigenen Funktionen besitzen. Hierbei wird ein Pull Request dazu verwendet einen automatischen Workflow einzubinden. Damit kann eine Zusammenführung an mehrere Bedingungen gekoppelt werden. So kann diese erst erfolgen, sofern eine bestimmte Anzahl an Kollegen die Änderungen überprüft haben oder ein Build-Prozess, welcher über die Erstellung eines Pull-Requests gestartet wird, erfolgreich das Projekt gebaut und getestet hat. Dies sind nur zwei Beispiele welche über die Funktio-

nalität von einem Pull-Request, bei jeder Zusammenführung durchgeführt werden müssen, falls zuvor entsprechend konfiguriert. Da diese Zusammenführungen innerhalb des Archivs stattfinden und nicht wie bei einem Fork von außerhalb stammen, spielt die Durchführung der Zusammenführung eine Rolle. Hierbei werden folgend die angebotenen Optionen des Hosters Github.com beleuchtet (29). Einerseits können die Änderungen über *merge --no-ff* hinzugefügt werden. Dadurch wird immer, auch bei einem theoretischen *fast-forward merge*, ein dedizierter Commit erstellt, welche alle einkommenden Änderungen in sich vereint. Dies ist zum Beispiel auf *Github.com* das standardmäßige Verfahren. Hierbei werden alle Information erhalten und die Historie nachweislich gestaltet. In Abbildung 64 wird dies durch den untersten Pfeil dargestellt. Die Verbindung zwischen dem ursprünglichen Zweig bleibt immer erhalten.

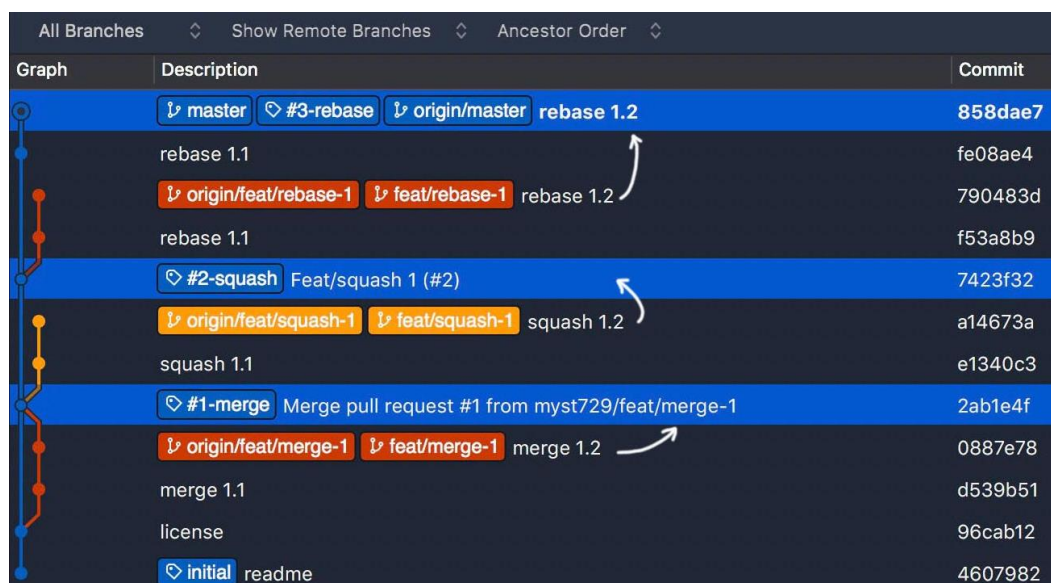


Abbildung 64: Unterschiedlicher Zusammenführungsarten bei einem Pull-Request (35)

Neben dieser Option, wird beim zweiten Pfeil *merge --squash* verwendet. Wie auch schon in Kapitel 4.6.1 beschrieben, werden die Änderungen nicht über einen Merge-Commit integriert, sondern über einen eigenständigen Commit, welcher alle abweichenden Commits des Quellzweigs vereint. Dabei wird keine Verbindung mit dem Quellzweig hergestellt. Dieser ist nach der Zusammenführung ein totes Ende. Gleichermaßen passiert dies bei der dritten Option, symbolisiert durch den obersten Pfeil. Hier werden die Commits dem Zielpfad in ihrer Ausführlichkeit angehängen. Auch hierbei ist der Quellzweig nun ein toter Zweig unter Umständen. Im Gegensatz zur zweiten Option, gehen die Einzelheiten hier nicht verloren, falls der Quellzweig entfernt wird. Allerdings sind diese durch den *Rebase* doppelt vorhanden, jedoch mit unterschiedlichen Metadaten. Dadurch unterscheidet sich die dritte Option der Pull-Request Zusammenführung von dem Kommando *git rebase*, da hierbei die Metadaten nicht unbedingt geändert werden. Beide Optionen können in verschiedenen Szenarien dem Anspruch

genügen, allerdings bietet *merge --no-ff* im Normalfall die beste Kombination. Nicht ohne Grund ist dies der Standard im Gitflow Modell. Im Vergleich mit den anderen Optionen wird hierbei auch im Nachhinein deutlich, welche Entwicklungen zu welchem Zeitpunkt stattfanden, eine Information, welche ohne die toten Zweige bei beiden anderen Optionen, verloren gehen. Mit *--squash* gehen alle Einzelheiten der ursprünglichen Entwicklung verloren, dahingegen ist dies jedoch die sauberste Methode. Die einzelnen Commits sind bei *--rebase* erhalten, allerdings besitzt der Zielzweig deutlich mehr Commits und die Information, wann jeder Commit entstand, ist ohne Quellzweig verfälscht, welcher allerdings keine Verbindung damit besitzt. Einige Entwickler verwenden vor einer *merge --no-ff* Zusammenführung einen interaktiven Rebase, um die entsprechenden Commits zu gruppieren und ihrer Funktionalität zuzuordnen. Dadurch wird die Anzahl der Commits nach der erfolgreichen Entwicklung einer Funktion reduziert und die Historie bleibt übersichtlicher. Allerdings sind die zu vorigen Aufteilungen in einzelne Commits verloren, sowie ein Teil der Metainformation, welche diese beinhalten. Die Annahme ist hierbei, dass diese Informationen nicht mehr von Nöten sind, da die Funktionalität abgeschlossen ist.

5.4 Individuelle Projektanpassungen

Um die für sein Projekt bestmögliche Arbeitsweise zu verwenden, müssen mehrere Faktoren beachtet werden. Wie viele Entwickler arbeiten an diesem Projekt? Ist es möglich das Projekt in mehrere Themen unterzuordnen, sodass parallel an diesen gearbeitet werden kann? Sind mehrere Versionen des Projekts produktiv einzusetzen? Macht es Sinn einen Host-Service zu nutzen? Und die wohl wichtigste Frage vorweg: Wie groß ist oder wird das Projekt? Durch diese und weitere Vorüberlegungen kann ein an den Bedingungen angepasster Workflow von Beginn an festgelegt werden, und sollte im Verlauf nur noch falls benötigt, leicht abgewandelt werden. Als grundlegende Devise lässt sich festhalten, je mehr Personen beteiligt sind, desto sinnvoller ist es auf mehreren Zweigen, beziehungsweise von der allgemeinen Referenz separiert zu arbeiten. Dies kann dennoch bei kleineren Projekten, ein Zweig pro Person heißen oder nur einen Entwicklungszweig für das gesamte Projekt. Hier entscheidet die Art des Projekts, ob diese isolierten Zweige benötigt werden. Diese Antwort gibt der zweiten Frage zur Möglichkeit der Trennung von Entwicklungsabläufen Priorität. Hier kann durch durchdachte Trennung von Funktionalitäten wichtige Zeit bei der Integration und Synchronisierung der jeweiligen Funktionalitäten gespart werden. Ist man sich diesem bewusst, sollte man eine gewisse Vorstellung besitzen, ob dem Projekt ein Entwicklungszweig reicht oder man vorweg schon auf mehrere Zweige differenziert. Diese Differenzierung kann auch im Laufe des Projekts einfach übernommen werden, somit ist eine Festlegung auf eine bestimmte Anzahl nicht sinnvoll. Allerdings sollte man sich der Anzahl der Hauptentwicklungszweigen sicher sein. Diese Frage stellt sich bei besonders großen

Projekten und wird dann wichtig, sobald gesamte Untermodule gebildet werden können. Eine weitere Frage bezieht sich auf die Anzahl an Zweigen für Veröffentlichungen. Standardmäßig besitzt ein Projekt ein *master* Zweig. Werden allerdings mehrere Versionen zeitgleich unterstützt, sollte eine Verteilung dieser auf je einen Zweig angestrebt werden. Dadurch vereinfacht sich die Wartung und Übersichtlichkeit enorm, da der zeitliche Verlauf eines *master* Zweiges auch dem zeitlichen Verlauf der jeweiligen Variante entspricht. Des Weiteren sollte eine Aufteilung von Entwicklungszweigen auf die einzelnen Versionen angestrebt werden. Hier kann noch einmal unterschieden werden: Soll lediglich die Wartung gewährleistet sein, ist eine solche Auftrennung nicht notwendig und kann über temporäre Änderungszweige zur jeweiligen Version durchgeführt werden. Sollen jedoch mehrere Versionen, welche gleichzeitig existieren und sich durch einen unterschiedlichen Funktionsumfang auszeichnen, unterstützt werden, ist die Vervielfachung von *master*, *release* und Hauptentwicklungszweig sinnvoll. Hier stellt sich eine weitere Frage, ist es sinnvoll einen oder mehrere Release Zweige zu besitzen? Hier gehen die allgemeinen Meinungen auseinander. Je nach Anforderungen an diesen Zweig, kann dieser in nahezu jedem Projekt sinnvoll eingeführt werden. Grundsätzlich sollte dieser verwendet werden, sobald mehrere Funktionalitäten dem Hauptentwicklungszweig zugeführt werden, diese aber nicht immer Inhalt der nächsten Veröffentlichung sein sollen. Sind die gewünschten Funktionen nicht linear beisammen, können diese auch über das Kommando *cherry-pick* nachträglich in den *release* Zweig überführt werden. Hierbei ist Vorsicht geboten, da diese Integration nicht grafisch sichtbar ist. Somit kann ab einem gewünschten Umfang auf einen *release* Zweig gewechselt werden, damit die zeitgleiche Weiterentwicklung entkoppelt ist. Ist man der Ansicht, diese Art von Zweig dient nicht der Entwicklung, sondern nur für Verbesserungen und Vorbereitungen auf die bevorstehende Veröffentlichung, kann es auch sinnvoll sein, diese Commits auf einem getrennten Zweig, dem *release* Zweig, zu handhaben. An diesem Zeitpunkt steht der strukturelle Aufbau des Projekts fest. Dementsprechend stellt sich die Frage welche Bedingungen und Vorgänge nun bei einer Zusammenführung durchgeführt werden sollen. Hierbei ist die erste Frage, ob ein Hosting-Service verwendet wird. Ist dies gegeben, gibt es nahezu keine Gründe, nicht jegliche größeren Zusammenführungen über einen Pull-Request abzuhandeln. Hierdurch können auch weitere immer wichtiger werdende Funktionalitäten einfach umgesetzt und ermöglicht, wie das automatische Anstoßen einer Build-Umgebung. Zudem bietet die mögliche Festlegung auf Bedingungen, welche erfüllt sein müssen, bevor die Zusammenführung stattfinden kann, die Möglichkeit zur Vermeidung von fehlerhaften Commits. Nicht immer bietet ein Pull-Request die optimale Methode. Sollen lokale Änderungen von Entwicklern in ihren Themenzweig integriert werden, kann ein einfacher *merge* schneller und unkomplizierter sein, statt unnötige Ressourcen zu belasten, wie ein weiterer Entwickler, welcher die Zusammenführung erst freigeben muss. Zusammenfassend lässt sich somit sagen: die Grundstruktur von Gitflow bietet den optimalen Aufbau des Projektar-

chivs und wird je nach Anforderung erweitert oder verkleinert. Zudem lässt sich die Arbeitsweise durch die Funktionalitäten von Pull-Requests erweitern, falls die Nutzung eines Dienstleisters verwendet wird. Andernfalls können die Zusammenführungen gleichermaßen über die Git eigenen Befehle durchgeführt werden. Hierbei müssen die Vorteile, welche ein Pull-Request vereint anderweitig implementiert werden. Dazu kann ein E-Mail-Verteiler verwendet werden. Andere Funktionen, wie ein notwendiges Code Review vor einer Zusammenführung, können allerdings nur über nicht Git interne Richtlinien implementiert werden. Da gerade im derzeitigen Zeitalter eine Automatisierung von Funktionalitäten angestrebt wird, ist die Integration über Hoster sehr beliebt und oftmals deutlich bevorzugt.

6 Interne Dateiverwaltung von Git

Durch die dezentrale Projektverwaltung von Git benötigt ein Projekt nicht nur zentral eine größere Menge an Speicherplatz. Dadurch ist es wichtig eine kompakte Speicherplatzverwaltung zu besitzen. Git benutzt hierfür eine ähnliche Verwaltung wie die Dateiverwaltung von Linux. Es werden somit die diversen Dateien und Information über Zeiger verwaltet. Zuerst werden einzelne Dateien über ihre SHA-1 Checksumme identifiziert und referenziert. Somit ist das Projektarchiv ein "Content-Addressed Storage" (CAS), denn auf sie kann direkt zugegriffen werden und die Objekte sind über ihren Inhalt adressiert. Bevor dies jedoch geschieht erfolgt eine geringe Komprimierung mit *Zlib*, sowie eine Voranstellung eines Kopfteils. Dadurch entsteht aus einer einzelnen Datei ein sogenannter *blob*. Die *blobs*, das heißt die einzelnen Dateien, werden wiederum durch Verzeichnisse verwaltet. Diese werden *tree* genannt und besitzen lediglich die Checksummen der beinhalteten *blobs*. In einem solchen Verzeichnis werden nun ein oder mehrere Objekte notiert. Darunter können auch weitere Objekte des Typs *tree* sein. Wird ein Commit erstellt, kann dadurch wiederum ein Objekt erstellt werden, welches einen *tree* seinerseits referenziert. Dieses Objekt beinhaltet dann wiederum alle Referenzen der Dateien, welche die damit festgehaltene Arbeitsmappe besitzt. Ein Etikett nutzt das gleiche Vorgehen, jedoch zeigt es auf einen Commit. Folglich entsteht durch die Git interne Dateiverwaltung folgende Strukturen:

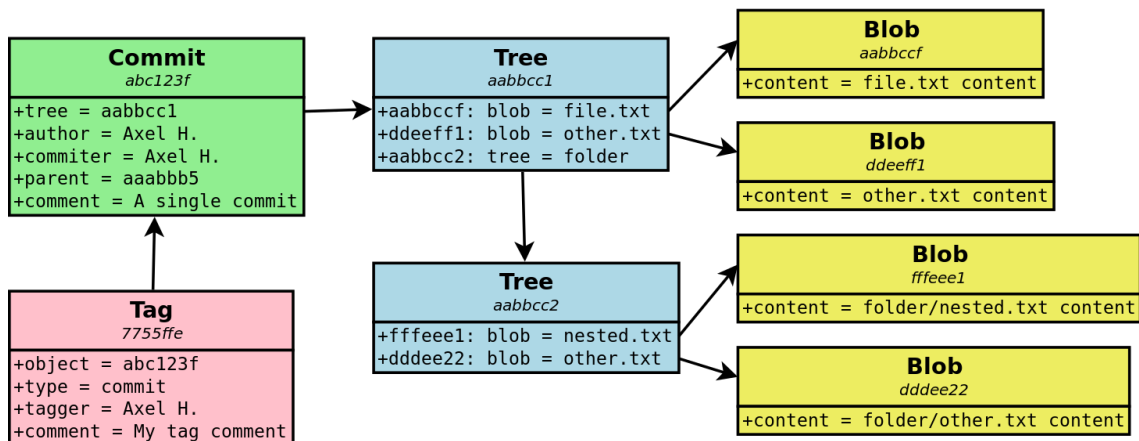


Abbildung 65: Darstellung der Git internen Datenverwaltung (35)

Durch diese Struktur wird neben den Dateien in komprimierter Form nur wenig Speicheraufwand betrieben, um die Verwaltung darzustellen. Der Vorteil der Verwaltung ist die Arbeit mit Referenzen. Hierbei kann ein *blob* in mehreren *trees* vorkommen und somit auch in mehreren Commits. Genauer, es wird lediglich ein *blob* einer Datei erstellt, falls sich diese ändert. Dadurch wird eine redundante Speicherung auf Dateiebene vermieden und Speicherplatz sowie Zeit bei der Synchronisierung gespart. Trotz dieser Maßnahme benötigt Git damit mehr Platz als eine Speicherung von Deltas, wie es andere Versionsverwaltung verwenden. Dies wird deutlich, wenn eine kleine Änderung an einer größeren Datei vorgenommen wird. Diese Datei wird trotz der marginalen Änderung in seiner vollen Größe abgespeichert. Deshalb wird nach einer gewissen Zeit die Speicherung älterer Daten verändert (4 p. 357ff.).

6.1 Packfiles

Durch die Verwaltung über Zeiger werden keine redundanten Dateien gespeichert, jedoch kann eine Änderung an einer Datei sehr gering sein und somit wird ein Großteil der Datei erneut gespeichert. Um diesen Speicherplatz freizugeben werden nach einer bestimmten Zeit die Objekte neu abgespeichert. Diesmal werden diese in Archive verpackt und auf Deltas aufgesplittet. Dadurch werden lokal keine redundanten Daten oder Teile von Dateien gespeichert, und somit der Speicherplatz minimiert. Dies benötigt allerdings Dateiformate, welche eine solche Änderungsnachverfolgung zulassen. Das heißt, binäre Dateien sind davon ausgeschlossen. Das Verpacken wird entweder periodisch durchgeführt, kann aber auch manuell angestoßen werden mit dem Kommando **git gc**. Dieses Format bietet einen möglichst kleinen Fußabdruck der Verwaltung, benötigt aber mehr Zeit die vergangene Version wiederherzustellen. Darum wird diese Art der Abspeicherung nicht direkt gewählt, sondern die Abspeicherung von Dateien in einzelnen Objekten für eine gewisse Zeit bevorzugt (4 p. 369f.).

6.2 Git LFS

Git LFS (large file storage) ist eine Erweiterung für Git, welche die Handhabung von großen Dateien vereinfacht. Ein großer Vorteil, jedoch in diesem Aspekt Nachteil von dezentralen Versionsverwaltungen ist der Fakt, dass jeder alle Daten des Projekts besitzt. Dadurch kann bei großen Dateien der Speicherplatzverbrauch über das gesamte Projekt deutlich ansteigen, da bei jedem Klon des Projekts, dieselbe Datei wieder und wieder verteilt abgespeichert wird. Das Git dieselbe Datei nicht in jedem Commit speichert wurde bereits zuvor in Kapitel 6 behandelt. Die Datei wird nur abgespeichert, falls sich die Checksumme der Datei ändert. Trotzdem kann die Zeit für die Synchronisation deutlich ansteigen, sofern größere Dateien in jeder erstellten Version vorhanden sind. LFS versucht nun dieses System zu umgehen, um diese Dateien bei einem einfachen

Kommando *fetch* oder *clone* nicht übertragen zu müssen. Ist LFS installiert und aktiviert, werden bestimmte Dateitypen über dieses System zwischenverwaltet. Hierbei wird vereinfacht ein Behälter für diese Dateitypen angelegt, um sie einmalig zu speichern, zum Beispiel auf dem Referenzserver oder falls gewünscht auch auf einem zusätzlichen Server. Hierfür werden intern für Git weitere Zwischenschritte angewendet, sodass bei einem Verweis auf diese Dateien lediglich eine Referenz, zeigend zu der jeweiligen Datei im Behälter eingefügt wird. Dadurch wird nur noch ein Verweis in Form einer SHA256 Checksumme im Commit gespeichert, anstatt der gesamten Datei. Wird die Datei verändert wird sie wiederum einmalig im Behälter hinterlegt. Die Behälter werden mit den nativen Git Kommandos *push* und *pull* synchronisiert, sodass alle kürzlich referenzierten Dateien auch im lokalen Behälter vorhanden sind und direkt verwendet werden können. Im Allgemeinen werden somit Dateien, welche über LFS gehandhabt werden, nur lokal gespeichert, falls diese innerhalb einer konfigurierbaren Zeit benötigt wurden. Dateien, welche theoretisch vorhanden sind, aber nicht referenziert wurden, sind damit nicht lokal vorhanden. Um übergreifend damit arbeiten zu können, müssen alle Git Instanzen, welche mit an diesem Projekt arbeiten LFS beherrschen können. Auch ist es möglich bestehende Projektarchive nachträglich mit LFS zu versehen. Hierbei wird, falls gewünscht, auch rückwirkend LFS eingeführt. Hierbei wird allerdings die Historie umgeschrieben. In solch einem Fall sollte jeder beteiligte das Archiv neu klonen. LFS lässt sich großzügig konfigurieren über die vorhandene Git Konfigurations-Datei. Zusätzlich bietet LFS eigene Git Kommandos rund um LFS an. Hierbei können auch Dateitypen zur LFS Verwaltung hinzugefügt werden über ***git lfs track "*.<extension>"***. Eine weitere Funktion von LFS ist die Möglichkeit diese Dateien zu sperren. Dadurch können, falls gewünscht, Konflikte mit Dateien verhindert werden, welche sich nicht zusammenführen lassen, wie zum Beispiel Binärdateien (30).

7 Zusammenfassung und Ausblick

Git bietet in seinem großen Umfang viele Möglichkeiten, um das Arbeiten im Team zu vereinfachen und bietet Lösungen für viele der anfänglich genannten Probleme. Die Probleme zur Zeitstrukturierung wurden anhand des dezentralen Konzepts gelöst. Ein Entwickler befindet sich nicht mehr in Stillstand, sofern eine für ihn benötigte Datei in Verwendung eines anderen ist. Paralleles Arbeiten ist auch durch die Verwendung von Zweigen begünstigt und Änderungen können durch einen Drei-Wege-Merge und dessen Unterarten, in gewünschter Form zusammengeführt werden. Durch vorherige Aufteilungen und eingeführten Richtlinien können die notwendigen Zusammenführungen konfliktarm durchgeführt werden und sind damit auf einen minimalen Zeitaufwand reduziert. Durch die Baumstruktur von Git und die Einfachheit von einzelnen Commits ist die Wiederherstellung älterer Stände des Projekts schnell möglich. Auch hier können durchdachte Etiketten und Zweige zur Versionsablage, die Wiederaufnahme eines älteren Standes vereinfachen und somit eine Arbeit mit diesem, in Sekundenschnelle starten. Durch die detaillierte Änderungsübersicht, sowie den vielzähligen Befehlen zur Arbeit mit dieser, ist die Verantwortlichkeit leicht zu klären. Einzig Anforderungen, welche bei der Erstellung von Git nicht relevant waren, sind ein Problem. So kann ein Zugriffsrecht lediglich eingeführt werden, indem man das Projekt auf mehrere Projektarchive aufteilt, wobei der Zugriff auf diese Projektarchive, das Zugriffsrecht durchsetzen. Auch hier hilft allerdings der jahrelange Aufstieg von Git, da mehrere Unternehmen diese fehlenden Funktionen mit der Zeit adressierten. Diese Unternehmen wie Github, Atlassian oder GitLab bieten in ihren produktiv einsetzbaren Versionen von Git, solche Funktionen an neben weiteren Funktionen, welche die aktuellen Trends wie eine automatisch Testdurchführung unterstützen. Des Weiteren wurde deutlich, dass es nicht reicht sich auf die Versionskontrolle zu verlassen, um automatisch mit den Qualitäten dieser arbeiten zu können. Es bedarf Wissen über die Versionskontrolle, sowie Wissen wie diese zu handhaben ist. Somit ist Git kein Wundermittel der Softwareentwicklung, welches die Durchführung eines Projektes automatisiert, sondern kann unter gewissen Vorraussetzung Vorteile in Sachen Produktivität und Dokumentation bieten.

Diese Arbeit umfasste nur einen Teil von Git und liefert ein Grundwissen für das Arbeiten mit Git. Sämtliche neue Änderungen, sowie Tools mit einer Grafischen Benutzeroberfläche können durch dieses Grundwissen gut erlernt werden.

8 Kurzreferenz

Eine Kurzübersicht über wiederkehrende Probleme und Aufgabenstellungen und ihre möglichen Lösungen (31).

Erstellen eines Projektarchivs (4.2)

git init

Verwandelt Ordner zu einem Projektarchiv

git clone <URL>

Kopiert ein bestehendes Projektarchiv, um lokal daran zu arbeiten

Konfigurieren der Git Einstellungen (4.2.2)

git config user.name "<Name>"

Setzt den Namen des Autors, welcher den erstellten Commits angeheftet wird. Weitere häufig verwendete Optionen: user.mail, core.editor.

Handhabung von Änderungen (4.3)

git status

Einsicht in den Status der lokalen Arbeitskopie. Hier werden hinzugefügte, modifizierte oder gelöschte Dateien angezeigt und ob diese Änderung der Datei dem Index hinzugefügt ist.

git add .

Fügt alle Änderungen der aktuellen Arbeitskopie dem Index hinzu.

git commit -m "<Nachricht>"

Erstellt lokal einen Commit mit den, dem Index hinzugefügten, Änderungen.

Synchronisierung von Änderungen (0)

git fetch <Referenz Alias>

Aktualisiert Referenzen; Änderungen in der Referenz werden lokal sichtbar.

git pull <Referenz Alias>

Aktualisiert Referenzen und führt dessen Änderungen ein. Kombination aus *git fetch* und *git merge*.

git push <Referenz Alias> [Zweig]

Aktualisiert das Referenzarchiv mit den lokal hinzugefügten Änderungen des aktiven Zweiges. Besitzt die Referenz den Zweig nicht wird dieser angehängt.

Erfolgte Änderungen einsehen (4.3.4)

git log --oneline --all

Listet die Historie des gesamten Archivs in Kurzform auf.

git log --follow <Datei> [-M]

Listet Commits in den Änderungen an der übergebenen Datei wahrgenommen wurden. Mit *-M* werden Umbenennungen berücksichtigt.

git show <Commit>

Zeigt Information sowie Änderungen des übergebenen Commits auf.

Änderungen revidieren (4.10)

git reset <Commit> [--hard]

Setzt die Arbeitskopie auf Stand des übergebenen Commits wieder, die auf den Commit aufbauenden Änderungen sind in der Arbeitskopie. Mit *--hard* werden die Änderungen verworfen. Das Kommando sollte nur lokal vorhandenen Commits verwerfen.

git revert <Commit>

Erstellt einen neuen Commit, welcher die eingeführten Änderungen des übergebenen Commits revidiert. Benutzt, um Commits im Referenzarchiv zu revidieren.

git checkout <Prüfsumme des Commits zum gewünschten Dateistands> -- <Dateipfad>

Setzt eine Datei auf einen beliebigen Stand zurück.

Zweige erstellen und Zusammenführen (4.5.3 f.)

git checkout <Zweig> [-b]

Wechselt den aktiven Zweig, mit *-b* wird dieser neu erstellt.

git branch -D <Zweig>

Löscht den übergebenen Zweig, ohne zu überprüfen ob die durchgeführten Änderungen bereits in andere Zweige übernommen wurden.

git push <Referenz Alias> --delete <Zweig>

Löscht den übergebenen Zweig in dem Referenzarchiv.

```
git merge <Zweig>
```

Fügt Änderungen des übergebenen Zweiges dem aktiven Zweig hinzu und erstellt in Folge dessen einen Commit.

Änderungen finden (4.3.4)

In Commit Nachrichten:

```
git log --all -i --grep '<Suche>'
```

Durchsucht die Commit Nachrichten nach der übergebenen Nachricht. Durch -i achtet die Suche nicht auf Groß- und Kleinschreibung.

In der Änderungsübersicht:

```
git log --all -i -p -S '<Suche>'
```

Durchsucht die Änderungen aller Commits auf das Hinzufügen oder Entfernen der gewünschten Suche. Mit -p wird zudem die Änderungsübersicht den zutreffenden Commits angehängt. Mit -G anstatt -S kann auch ein regulärer Ausdruck übergeben werden.

Referenz und lokales Archiv besitzt Neuerungen (4.7.5)

```
git pull --rebase <Referenz>
```

Führt bei der Übernahme der Referenzänderungen eine Neuplatzierung vor, sodass die lokalen Neuerungen auf den Referenzänderungen aufbauen. Damit ist nun nur das lokale Archiv voraus und Änderungen können synchronisiert werden. Dies verändert die Historie.

```
git pull <Referenz>
```

Erstellt einen neuen Commit welcher Änderungen der Referenz und dem lokalen Archiv vereint. Das lokale Archiv ist danach voraus und kann synchronisiert werden. Der Nachteil ist, dass mit der Zeit eine erhebliche Anzahl solcher Commits entstehen, welche keine Neuerungen einführen.

Referenzarchiv hinzufügen (4.7.1)

```
git remote add <Name> <URL>
```

Fügt dem lokalen Projektarchiv eine Verlinkung unter dem Alias <Name> hinzu.

Commit auf dem falschen Zweig (4.11.1)

git checkout <Richtiger Zweig>

git cherry-pick <Falscher Zweig>

git checkout <Falscher Zweig>

git reset HEAD~ --hard

In den richtigen Zweig wechseln um den fehlerhaften Commit über das Kommando *cherry-pick* zu integrieren und danach den falschen Commit zu verwerfen. Ist der Commit nicht der neuste auf dem fehlerhaften Zweig, muss der Befehl *cherry-pick*, sowie *reset* dementsprechend abgeändert werden.

9 Literaturverzeichnis

1. **Neumann, Alexander.** heise.de. [Online] 08. 04 2015. [Zitat vom: 23. 03 2020.] <https://www.heise.de/developer/meldung/Vor-10-Jahren-Linus-Torvalds-baut-Git-2596654.html>.
2. **Atlassian.** *atlassian.com*. [Online] [Zitat vom: 24. 11 2019.] <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init>.
3. **Groß, Torsten.** seibert-media.net. [Online] 06. 01 2015. [Zitat vom: 1. 12 2019.] <https://blog.seibert-media.net/blog/2015/01/06/tutorial-git-aufsetzen-teil-3-git-config/>.
4. **Scott Chacon, Ben Straub.** *Pro Git*. s.l. : Apress, 2014.
5. **Atlassian.** *atlassian.com*. [Online] [Zitat vom: 19. 11 2019.] <https://www.atlassian.com/de/git/tutorials/rewriting-history>.
6. **Valentin Haernel, Julius Plenz.** *Git - Verteilte Versionsverwaltung für Code und Dokumente*. s.l. : Open Source Press, 2011.
7. **Zoric, Nesha.** dev.to. [Online] 15. 03 2018. [Zitat vom: 29. 10 2019.] <https://dev.to/neshaz/how-to-git-stash-your-work-the-correct-way-cna>.
8. **Vanderwal, Brian.** atomicobject.com. [Online] 26. 06 2016. [Zitat vom: 28. 01 2020.] <https://spin.atomicobject.com/2016/06/26/parallelize-development-git-worktrees/>.
9. **Atlassian.** *atlassian.com*. [Online] [Zitat vom: 24. 11 2019.] <https://www.atlassian.com/de/git/tutorials/using-branches>.
10. **git-scm.com.** [Online] [Zitat vom: 24. 11 2019.] <https://git-scm.com/docs/git-checkout>.
11. **Fournova Software GmbH.** git-tower.com. [Online] [Zitat vom: 24. 11 2019.] <https://www.git-tower.com/learn/git/faq/detached-head-when-checkout-commit>.
12. **Siduction.** siduction.de. [Online] [Zitat vom: 1. 12 2019.] <https://wiki.siduction.de/index.php?title=Gitk>.
13. **Santos, Pablo.** drdobbs.com. [Online] 24. 12 2013. [Zitat vom: 17. 11 2019.] <https://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902?pgno=1>.
14. **jejese.** stackoverflow.com. [Online] 30. 04 2018. [Zitat vom: 24. 02 2020.] <https://stackoverflow.com/questions/449541/how-to-selectively-merge-or-pick-changes-from-another-branch-in-git>.

- 15. Github.** github.com. [Online] [Zitat vom: 18. 11 2019.]
<https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line>.
- 16. Atlassian.** atlassian.com. [Online] [Zitat vom: 30. 11 2019.]
<https://www.atlassian.com/de/git/tutorials/syncing>.
- 17. Atlassian.** atlassian.com. [Online] [Zitat vom: 07. 12 2019.]
<https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame>.
- 18. Atlassian.** atlassian.com. [Online] [Zitat vom: 02. 01 2020.]
<https://www.atlassian.com/de/git/tutorials/comparing-workflows>.
- 19. Yehezkel, Alon.** hackernoon.com. [Online] 18. 06 2018. [Zitat vom: 1. 12 2019.]
<https://hackernoon.com/how-to-git-pr-from-the-command-line-a5b204a57ab1>.
- 20. Atlassian.** atlassian.com. [Online] [Zitat vom: 12. 11 2019.]
<https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag>.
- 21. git-scm.com.** [Online] [Zitat vom: 2. 12 2019.] <https://git-scm.com/docs/git-show>.
- 22. Atlassian.** atlassian.com. [Online] [Zitat vom: 02. 12 2019.]
<https://www.atlassian.com/git/tutorials/saving-changes/git-diff>.
- 23. Irelan, Ryan.** mijingo.com. [Online] [Zitat vom: 30. 11 2019.]
<https://mijingo.com/blog/creating-and-applying-patch-files-in-git>.
- 24. Atlassian.** atlassian.com. [Online] [Zitat vom: 02. 12 2019.]
<https://www.atlassian.com/de/git/tutorials/undoing-changes>.
- 25. Wehner, Joshua.** github.blog. [Online] 08. 01 2015. [Zitat vom: 02. 12 2019.]
<https://github.blog/2015-06-08-how-to-undo-almost-anything-with-git/>.
- 26. Ebert, Ralf.** ralfebert.de. [Online] 17. 08 2013. [Zitat vom: 15. 12 2019.]
<https://www.ralfebert.de/git/cherry-pick/>.
- 27. Bednova, Irina.** jafrog.com. [Online] 22. 03 2012. [Zitat vom: 23. 12 2019.]
<http://jafrog.com/2012/03/22/git-cherry.html>.
- 28. Bruckner, Felix.** oio.de. [Online] 22. 09 2014. [Zitat vom: 03. 02 2020.]
<https://blog.oio.de/2014/09/22/git-workflows-teil-1-warum-wir-workflows-brauchen/>.
- 29. Github.** github.com. [Online] [Zitat vom: 03. 02 2020.]
<https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-request-merges>.
- 30. Atlassian.** atlassian.com. [Online] [Zitat vom: 23. 12 2019.]
<https://www.atlassian.com/git/tutorials/git-lfs>.

- 31. Github.** github.com. [Online] [Zitat vom: 19. 01 2020.]
<https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf>.
- 32. Haustant, Axel.** illustrated-git.readthedocs.io. [Online] [Zitat vom: 20. 02 2020.]
<https://illustrated-git.readthedocs.io/en/latest/>.
- 33. Seibert Media GmbH.** seibert-media.net. [Online] [Zitat vom: 19. 01 2020.]
<https://m.infos.seibert-media.net/Productivity/Git-Workflows+-+Der+Gitflow-Workflow.html>.
- 34. Deng, Leo.** myst729.github.io. [Online] 10. 07 2019. [Zitat vom: 03. 02 2020.]
<https://myst729.github.io/posts/2019/on-merging-pull-requests/>.
- 35. Shvets, Alexander.** githowto.com. [Online] [Zitat vom: 17. 11 2019.]
https://githowto.com/git_internals_git_directory.

Abbildungsverzeichnis

Abbildung 1: Versionsverwaltung anhand einfacher Datenablage.....	2
Abbildung 2: Synchronisierung von Projektarchiven.....	5
Abbildung 3: Grundlegender Commit Vorgang	6
Abbildung 4: Git Installation unter Linux mit <i>apt-get</i>	7
Abbildung 5: Einstellung zur PATH Variable unter Windows	8
Abbildung 6: Darstellung des .git Ordners (31)	10
Abbildung 7: Initialisierung eines Projektarchivs	10
Abbildung 8: Einsicht in die bestehende Git Konfiguration.....	11
Abbildung 9: Klonen eines öffentlichen Projektarchivs von Github.com.....	12
Abbildung 10: Erzeugung der .gitignore Datei.....	12
Abbildung 11: Darstellung der Datei .gitignore	13
Abbildung 12: Ausgabe des Befehls <i>git status</i>	13
Abbildung 13: Dateien dem Index hinzufügen.....	14
Abbildung 14: Darstellung von Commits über den zeitlichen Verlauf	15
Abbildung 15: Erstellung des ersten Commits	16
Abbildung 16: Ausgabe des Befehls <i>git log</i>	17
Abbildung 17: Ausgabe von <i>git log</i> in Zeilenform.....	17
Abbildung 18: Auswahl der Commits beim interaktiven Rebase.....	19
Abbildung 19: Bearbeitung der Commit Nachrichten beim Rebase	19
Abbildung 20: Kommandozeilen Ausgabe nach erfolgreichem Rebase	20
Abbildung 21: Historie nach erfolgreicher Verschmelzung	20
Abbildung 22: Erstellung einer Zwischenspeicherung	22
Abbildung 23: Arbeit mit mehreren Arbeitsverzeichnissen	22
Abbildung 24: Erstellung eines weiteren Zweiges	24
Abbildung 25: Wechsel der aktiven Arbeitsmappe	25
Abbildung 26: Anwendung des Zwischenspeichers	25
Abbildung 27: Erstellung und Arbeit am Zweig "newshape"	26
Abbildung 28: Darstellung des Projekts in gitk.....	26
Abbildung 29: Problemstellung einer Drei-Wege-Zusammenführung (12).....	27
Abbildung 30: Fast-forward Integration des Zweigs <i>newshape</i> in <i>master</i>	29
Abbildung 31: Projekt Historie nach erfolgreicher Integration von <i>newshape</i>	30
Abbildung 32: Auswirkung des Befehls <i>rebase</i>	30
Abbildung 33: Konfliktbehaftete Durchführung des Kommandos <i>rebase</i>	31
Abbildung 34: Ausführung des Git hinterlegten merge tool	33
Abbildung 35: Gelöster Konflikt durch das Programm meld.....	34
Abbildung 36: Weiterführung des Befehls <i>rebase</i>	35
Abbildung 37: Zusammenführung zweier Zweige über das Kommando <i>rebase</i>	35
Abbildung 38: Synchronisierung des lokalen Projektarchivs (35)	36

Abbildung 39: Hinzufügen eines Referenzarchivs	37
Abbildung 40: Lokalen Stand als Referenz setzen	38
Abbildung 41: Ansicht nach Ausführung des Befehls <i>blame</i>	39
Abbildung 42: Synchronisierung mit dem Referenzarchiv (<i>fetch</i>)	40
Abbildung 43: Integrierung von Änderungen über den Befehl <i>fetch</i>	41
Abbildung 44: Ausgabe des Kommandos <i>request-pull</i>	42
Abbildung 45: Integrierung von Änderungen über den Befehl <i>pull</i>	42
Abbildung 46: Fast-forward Zusammenführung anstelle des Befehls <i>pull</i>	42
Abbildung 47: Erstellen eines Etiketts.....	43
Abbildung 48: Ausgabe eines Etiketts über das Kommando <i>show</i>	44
Abbildung 49: Erstellung einer Änderungsdatei mit dem Befehl <i>diff</i>	44
Abbildung 50: Anwendung einer Änderungsdatei auf die Arbeitsmappe	45
Abbildung 51: Kopfzeile einer Änderungsdatei über den Befehl <i>patch</i>	45
Abbildung 52: Erstellung einer Änderungsdatei mit <i>format-patch</i>	46
Abbildung 53: Automatische Anwendung einer Änderungsdatei	46
Abbildung 54: Historie vor Revidierung eines Commits	47
Abbildung 55: Historie nach Revidierung eines Commits	47
Abbildung 56: Zurücksetzung der Historie durch das Kommando <i>reset</i>	48
Abbildung 57: Ansicht des Reflogs	49
Abbildung 58: Cherry-pick eines Commits (35)	49
Abbildung 59: Der Commit "Added function foo()" existiert in beiden Zweigen	50
Abbildung 60: Ausgabe des Kommandos <i>cherry</i>	50
Abbildung 61: Die Zweige <i>master</i> , <i>release</i> und <i>develop</i> in Gitflow (34)	52
Abbildung 62: Vorgehensweise bei <i>feature</i> Zweigen in GitFlow	52
Abbildung 63: Hierarchie in der Linux Kernel Entwicklung	54
Abbildung 64: Unterschiedlicher Zusammenführungsarten bei einem Pull-Request (33)	56
Abbildung 65: Darstellung der Git internen Datenverwaltung (35)	60

Glossar

Arbeitsverzeichnis: Das lokale Verzeichnis, indem der Entwickler einen Stand modifiziert und verwaltet.

Branch: Zweig. Dieser stellt eine isolierte Entwicklungsumgebung dar und hilft dabei parallel und strukturiert zu arbeiten. Zweige besitzen verschiedene Funktionen innerhalb eines Projekts.

Commit: Schnappschuss, welcher den aktuellen Stand der Arbeitsmappe festhält. Einem Commit werden alle Inhalte des Indexes hinzugefügt. Die Aneinanderreihung von Commits erstellt die bekannte Baumstruktur.

Content-Addressed Storage (CAS): Art der Speicherverwaltung bei der die Information über ihren Inhalt referenziert werden, anstatt über ihren physikalischen Standort.

Copy-Modify-Merge: Prinzip bei dem eine Datei zur selben Zeit durch mehrere Personen bearbeitet werden kann. Nach vollzogenen Änderungen werden alle Neuerungen zusammengeführt. Gegensätzlich zu dem Prinzip: "Lock-Modify-Unlock".

Fork: Kopie eines Projektes, welches man nur lesen kann. Wird bei Open Source Projekten verwendet, wobei die Änderungen über Pull-Requests dem originalen Archiv zugeführt werden.

Gitflow: Anerkannter Arbeitsablauf mit dem die Aufteilung und der Ablauf eines Projekts mit mehreren Zweigen geregelt wird.

Large File Storage (LFS): Eine Erweiterung für Git welche die Handhabung und Synchronisationszeiten mit größeren Dateien vereinfachen soll.

Pull-Request: Von Git Host Anbietern verwendete Funktion, welche Zusammenführungen über einen strukturierten Ablauf abhandelt. Innerhalb dieser Funktion werden einkommende Änderungen vorgeschlagen und über diese diskutiert.

Repository: Projektarchiv, hier werden alle Daten des aktuellen Standes sowie der gesamten Projekthistorie gespeichert.

Shared repository: Referenzarchiv, dies ist das Projektarchiv mit welchem alle am Projekt arbeitenden Personen synchronisieren. Dadurch besitzt das Projekt einen gesicherten Hauptstand. Dieses Projektarchiv besitzt keine Arbeitsmappe.

Staging Area: Im Index werden alle Dateien verwaltet, welche abweichend zum letzten festgehaltenen Stand der Arbeitsmappe sind. Dateien im Index können hinzugefügt, modifiziert oder entfernt werden. Alle im Index befindliche Inhalte werden dem nächsten Commit beigelegt.

Squash: Verschmelzung. Dabei werden mehrere Commits vereinigt. Dies wird dazu verwendet die Historie zu vereinfachen und zusammenhängende Commits zu schaffen.

Tag: Etikett, welches einen spezifischen Commit referenziert. Etiketten werden dazu verwendet Veröffentlichung und wichtige Stände zu markieren für einen einfacheren Bezug zu späteren Zeitpunkten.