## Git in Kurz

Visualisieren!!

Da in den Erklärungen der einzelnen Kommandos viele andere Kommandos referenziert werden oder zum Verständnis benötigt werden, ist es hilfreich zuvor einen kurzen Einblick in die grundsätzliche Struktur Gits zu erhalten. Ein Projekt besteht grundsätzlich aus einem Archiv. Dies beinhaltet alle Daten des Projekts, sowie die von Git benötigten Daten zur Konfiguration und Verwaltung. 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 geleichzeitig voran zu treiben. Zum Beispiel arbeitet ein Team an einem Zweig, während einer besonderen Funktion in einem weiteren Zweig entwickelt wird. Die Gründe für Abzweigungen sind vielseitig. Um einen Schnappschuss zu erstellen werden zuvor Änderungen markiert, welche diesem Schnappschuss beigefügt werden sollen. Dies nennt man den Index. Danach werden die Dateien im Index durch ein Commit archiviert und der Historie "angehangen". Commits und Zweige werden zwischen den verschiedenen Projektarchiven der einzelnen Entwickler synchronisiert über die Befehle namens *push* und *pull.* Da nicht jeder mit jedem synchronisieren soll, besteht im Normalfall eine Referenzarchiv, welches als zentrale Referenz des aktuellen Standes gilt.

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



## Git Installation 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 eigenen Git Bash angewiesen ist. Daher empfiehlt sich die zweite Option, siehe ABBILDUNG. Diese Option sollte standardmäßig ausgewählt sein. Ist die Installation beendet, sollte Git nun 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.

## Git und seine Komandos

Um die wesentlichen Befehle von git zu behandeln und erklären, werden die jeweiligen Kommandos anhand des folgenden Projekts aufgezogen. Dabei wird ein Kommando in dessen sinnvollen Umfang erklärt, sobald dies verwendet wird. Um mit git zu arbeiten benötigt man entweder die Kommandozeilen Version von git oder man installiert eine der vielen grafischen Oberflächen. Um die Dokumentation möglichst langwährend und übergreifend zu gestalten, wird git im Folgenden anhand der Kommandozeile erklärt. Falls die Funktion des Kommandos oder dessen Handhabung ungewiss ist kann über ***git <Kommando> --help*** die Dokumentation zu eben diesem Kommando aufgerufen werden.

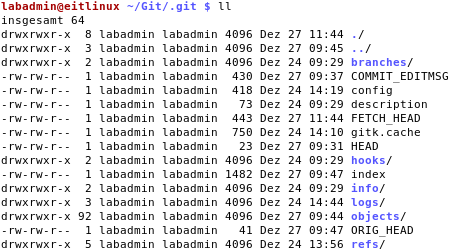
# Die ersten Schritte in Git

Um nun mit git arbeiten zu können, benötigt man ein Projektarchiv (Repository). Dies kann lokal erstellt werden oder es wird an einem bestehenden Projekt gearbeitet. Im ersten Fall geschieht dies über den Befehl *init.*

## Erstellen eines Projektarchives

### Git Init

Das Kommando ***git init*** fügt dem aktuellen Verzeichnis ein *.git* Ordner hinzu. Ohne zuerst in den späteren Projekt Ordner zu wechseln kann auch der Pfad nach ***git init*** angegeben werden. Falls der übergebene Ordner nicht existiert, wird dieser erstellt. Dieser Ordner enthält alle wichtigen Informationen zum Projektarchiv. Der "*.git"* Ordner reicht alleine, um ein Archiv zu klonen. Wie in ABBILDUNG zu sehen, besitzt der Ordner "*.git"* mehrere Unterordner.



Anders anordnen und in der reihenfolge beschschreiben

Commits befinden sich verschlüsselt in dem Ordner objects. *Refs* enthält Verweise auf die verschiedenen Zweige (*branches*) und Etiketten (tags). Skripte, die bei git Kommandos ausgeführt werden sollen, befinden sich unter hooks. Die *Config* Datei enthält Information und Einstellungen für das Projektarchiv. Erstellt man ein Referenzarchiv (*shared repository)* welches als reines Archiv zur Synchronisationgenutzt 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 hinzufügen kann. Um dies zu tun, muss somit das Archiv zuerst mit *git clone* geklont werden. Ist der grundlegende *.git* Ordner erstellt, sieht git den übergeordneten Ordner nun als ein Projektarchiv an. Es kann nun damit gearbeitet werden. [11]

Um ein Projektarchiv für das Beispielprojekt zu erstellen, führt man gemäß ABBILDUNG im zuvor erstellen Ordner, hier "Git", den Befehl ***git init*** aus.

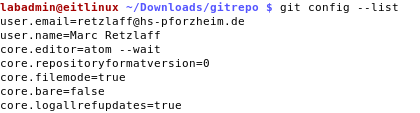


Falls keine archivübergreifende Konfiguration existiert, muss git zuerst mitgeteilt werden, welcher Entwickler die Änderungen lokal durchführt, sodass jeder Commit einem Autor zugewiesen werden kann. Hierfür muss die Konfiguration geändert werden.

### Git config

Wird eine neues Projektarchiv erstellt oder auch nach einer lokalen Installation von git sollte man grundsätzliche Einstellungen vornehmen, um git 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 benutzt. Mit dem Anhang ***–global*** werden die Einstellungen nicht nur für das aktuelle Archivbenutzt, sondern auch für alle anderen Archive des angemeldeten Benutzers. Hierfür wird mit ***git config –global user.name "<name>"*** beziehungsweise ***user.email "<mail>"*** die zwei wichtigsten Informationen gesetzt. Weitergehend kann der Standard Editor geändert werden mit: core.editor, oder man ändert die Einstellungen in der respektiven Datei, aufzurufen mit ***git config –global –edit***. Des Weiteren können Aliase erstellt werden, um individualisierte Commandos zu erstellen oder bestehende abzukürzen. Dies geschieht grundsätzlich mit ***git config alias.<alias name> <git-command>.*** Um die bisherige Konfiguration einzusehen, wird ***git config –list*** verwendet.[18]

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



Ist ein Projektarchiv allerdings bereits vorhanden und liegt auf einem Hoster oder Server, muss zuerst das bestehende Projekt geklont werden, um dann lokal daran arbeiten zu können. Hierfür steht das Kommando *clone* zur Verfügung.

### Git clone

Mit diesem Kommando wird ein neues Repository angelegt, jedoch als Vorlage ein bestehendes Repository verwendet. Man klont somit das Repository. Zusätzlich wird direkt die Verbindung zum ursprünglichen Repository beibehalten und unter dem Alias *origin* hinterlegt. So können ohne Umwege Änderungen am ursprünglichen Repository verfolgt und in das Eigene übernommen werden. Verwendet wird das Kommando mit ***git clone <remoteAdresse> <lokalesVerzeichnis>***. Als einfaches Beispiel wird in ABBILDUNG ein öffentliches Projektarchivvon GitHub geklont. Im Falle eines privaten Archivskann *https://<account>:<passwort>@github.com/...* Verwendet werden. Andernfalls wird zuvor der Benutzername und das Passwort zu einem Account verlangt, welcher Zugriff auf dieses Archiv besitzt. Neben "https://" kann auch "ssh://" oder "git://" je nach Anwendung mit ***git clone*** verwendet werden. Dafür müssen jedoch entsprechende Konfiguration, wie zum Beispiel der Austausch der SSH Schlüssel, zuvor erfolgen.



## Änderungen dokumentieren

Als erster Schritt des Beispielprojekts wird nun eine README Datei erstellt, um das bevorstehende Projekt kurz zu beschreiben. Zusätzlich dazu wird eine Dateien mit dem Namen ".gitignore" erstellt wie in ABBILDUNG zu sehen.



### Gitignore

Benutzt man Dateien, welche explizit nicht in die Versionsverwaltung inkludiert werden sollen, kann eine Datei namens: ***.gitignore*** verwendet werden. Dateien und Ordner, welche in *.gitignore* zeilenweise vorkommen, werden automatisch in ***git status*** ausgeblendet und somit nicht dem Index vorgeschlagen bzw. dem nächsten Commit hinzugefügt. In ABBILDUNG zu sehen, werden *object-*Dateien*,* temporäre Dateien, welche mit einer Tilde enden,sowie das gebaute Programm namens "geo" ausgeblendet. Dieses Prinzip wird unter anderem häufig für Dateien benötigt, welche im Erstellungsprozess verwendet oder erstellt werden, da diese jeder Entwickler lokal erstellen kann, somit die Versionsverwaltung nicht unnötig vergrößert wird und mehr Übersichtlichkeit erhält.

Die zuvor erstellten Dateien sollen nun der Versionsverwaltung hinzugefügt werde. Hierfür spielt die Arbeitskopie in Verbindungen mit dem Index (Staging Area) eine Rolle. Standardmäßig mit der Initialisierung eines Projektarchivs existiert ein Zweig namens *master*.

### Index (Stageing Area)

Ist die lokale Arbeitskopie vorhanden, können nun Änderungen und neue Elemente hinzugefügt werden. Mit dem Befehl ***git status*** werden die derzeitigen Änderungen angezeigt, zu sehen in ABBILDUNG. Hier werden nun die zwei neu erstellten Dateien angezeigt.



Beide Dateien werden unter nicht versionierte Dateien aufgeführt. Dass bedeutet, diese Dateien sind noch nicht in der Versionsverwaltung vorhanden. Diese Dateien oder Änderungen von Dateien, falls diese bestehen, können nun dem Index hinzugefügt werden. Dazu wird der Befehl ***git add*** benutzt. Hier 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 \**** . Ausgeschlossen durch den \* Operator 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 Repository 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, werden nun beide Dateien dem Index hinzugefügt. Im Anschluss sieht man anhand des Status Kommandos, dass beide Dateien nun für den nächsten Commit vorgemerkt sind.



Das Kommando bietet einen größeren Umfang. Um differenzierter, nicht nur Dateien, sondern auch einzelne Abschnitte oder Zeilen hinzuzufügen, kann das *add* Kommando noch erweitert werden. Eine Möglichkeit ist ***git add –interactive*** . Hier werden nun mehrere Commandos abgefragt, welche einzeln, nacheinander auf Dateien angewendet werden können. Mit dem Kommando *patch* kann nun eine Datei weiter aufgesplittet werden, um einzelne Zeilen dem Index hinzuzufügen [1]. Um dies für eine einzelne Datei durchzuführen kann ***git add -p <file>*** verwendet werden. Soll eine Datei aus der Versionsverwaltung entfernt werden, wird sie dem *remove* Kommando übergeben. Der Aufruf lautet hierbei ***git rm <file>.*** Für dauerhafte oder wiederkehrende nicht versionsverwaltete Dateien in der Arbeitskopie ist es hilfreich eine ***.gitignore*** Datei, siehe Kapitel XX, anzulegen, beziehungsweise die Datei der .gitignore hinzuzufügen.

Als nächster Schritt soll nun ein Commit getätigt werden um als erster Schnappschuss des Initialen Archivs zu stehen.

### Git commit

Commits bilden die zentrale Rolle in der Versionsverwaltung git. Ein Commit entspricht einem bestimmten Stand der Arbeitskopie. Je nach Konfigurationsmanagement und 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 Entwicklungspfad. In Absatz HIERVERLINKEN Gitflow wird deutlich wofür ein Commit auf dem jeweiligen Pfad stehen kann. Ein Commit enthält neben dem Abbild der aktuellen Arbeitskopie, sofern alle Veränderungen dem Index hinzugefügt wurden, auch eine Commit Nachricht. Diese sollte eine möglichst kurze, aber dennoch Aussagende Zusammenfassung der Änderungen beinhalten. Die Commit Nachricht ist neben Etiketten die schnellste Möglichkeit, Änderungen zu finden. Für schnelle Übersicht und Zusammenarbeit sollte man sich daher ein Schema überlegen, welche Projektweit gilt und angewendet wird. Ein Commit verweist auch immer auf seinen Vorfahren*.* Anhand diesem kann man feststellen, woher der Commit stammte, beziehungsweise, welche Änderungen im Vergleich zu seinem Vorfahreneinflossen. Durch diese Verknüpfung kann man nun die Commits anordnen und erhält damit die grafische Darstellung einer Baumstruktur, siehe ABBILDUNG. 

Jeder neue Commit wird an dessen vorhergehenden angehangen und erweitert die Kette. Wird ein neuer Zweig erstellt, zeigen zwei Commits auf den gleichen Vorgänger, es entsteht eine Abzweigung. Im Gegensatz dazu besitzt ein Commit einer Zusammenführung mehrere Vorgänger. Werden Zweige wieder zusammengefügt hat der folgende Commit zwei Vorfahren*.* Durch die Kette an Commits bildet sich eine lückenlose Historie bis zum Ursprung des git Repository. Commits werden intern durch ihre Prüfsumme unterschieden und darüber auch gespeichert. Um auf bestimmte Commits zu verweisen wird dafür oftmals der Anfang ihrer SHA1- Prüfsumme verwendet. Falls man den letzten Commit noch einmal bearbeiten will hilft das Schlüsselwort ***–amend***. Dadurch kann man zum Beispiel die Commit Nachricht überarbeiten, sofern der aktuelle Index noch leer ist. Hat man vergessen etwas dem Commit hinzuzufügen kann man dies dem Korrektur Commit hinzufügen oder über ***-m "<message>"*** die Nachricht umändern. Allerdings überschreibt der neue Commit den Alten wodurch alle Referenzen auf diesen verloren gehen. Daher sollte man *–amend* nur nutzen, sofern keine andere Partei auf diesen aufgebaut haben. Die Faustregel lautet: Kein *amend* auf veröffentlichte Commits. Veröffentlicht heißt hierbei über das lokale Projektarchiv hinaus.[10]

Dem Projekt wird nun der initiale Commit hinzugefügt. Wie in ABBILDUNG 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 zwei Dateien neu hinzugefügt wurden und insgesamt 4 Zeilen hinzukamen.

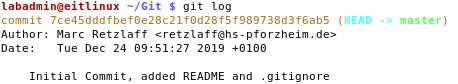


Zur Überprüfung kann man die Historie des Projektarchivs einsehen. Hier sollte der Commit nun zu sehen sein. Dafür verwendet man das Kommando *log.*

### Git log

Um einen Überblick über den aktuellen Stand des Zweiges zu erhalten, kann eine Übersicht der letzten Commits mit ***git log*** aufgerufen werden. Dort sieht man alle Commits des Zweiges mit Information zum Autor, Datum und Hash des Commits. Mit dem Schlüsselwort ***–all*** werden die Commits aller zweige in zeitlicher Reihenfolge angezeigt. In Kurzform kann auch ***git shortlog*** benutzt werden, dadurch wird nur die Commit Nachricht gruppiert nach Autor angezeigt. Um zusätzlich den Anfang des jeweiligen Hash zu erhalten kann ***git log --oneline*** benutzt werden. Ein weiteres nützliches Werkzeug bietet der Anhang ***--follow <file>***. Hier werden nun alle Änderungen an der Datei mit dem jeweiligen Commit ausgegeben. Mit ***–grep '<Suche>'*** werden nur Commits mit passender Commit Nachricht angezeigt. Um auch in den 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 der Baum, welcher das Repository wiederspiegelt, um die Commits schneller den Zweigen zuordnen zu können. Um sehen zu können ob Etiketten oder Zweige dem jeweiligen Commit zugeordnet sind, wird ***–decorate*** verwendet. [27]

Führen wir nun in ABBILDUNG zu sehen ***git log*** aus, wird der bisher einzige Commit angezeigt.



Spulen wir einige Commits und Änderungen vor, sieht das Projektarchiv nun wie folgt aus:



Es fällt nun dem Entwickler auf, dass die 3 nachfolgenden Commits nur zusammen Sinn ergeben. Daher würde es auch sinnvoll sein, diese zu einem Commit zu verschmelzen. Auch dies ist möglich mit einer Funktion des Kommandos *rebase.*

## Die Historie umschreiben

### Verschmelzen (Squash) – Git Rebase

Um mehrere Commits miteinander zu verschmelzen (squash) wird das Kommando ***git rebase*** mit dem Schlüsselwort ***--i***beziehungsweise ***-interactive***verwendet*.* Dadurch öffnet sich der interaktiveModus, bei dem man jeden einzelnen Commit betroffen vom Befehl individuell verändern kann. So kann man die Reihenfolge der Commits verändern, einzelne Commits löschen, Commits verschmelzen und bearbeiten. Dies geschieht in dem man die erste Spalte dementsprechend anpasst. Hierfür werden wiederum spezielle Schlüsselwörter benutzt. Um einen Commit zu löschen, entfernt man allerdings einfach dessen Zeile. Dies wird häufig verwendet, um mehrere Commits schnell zu verwerfen. Die Reihenfolge der Commits entspricht der Reihenfolge, mit der *rebase* sie auf den neuen Vorfahren anwendet. Standardmäßig ist dies genau umgekehrt der Baumreihenfolge, da *rebase* zuerst den zeitlich ersten Commit anwenden muss, um die Reihenfolge wiederherzustellen. Das Schlüsselwort *reword* ermöglicht es die Commit Nachricht neu zu schreiben. Mit *squash* verschmelzt sich der Commit mit Letzterem. Dies wird häufig dazu benutzt, um die Baumstruktur übersichtlich zu halten oder Korrektur Commits für bereits erstellte Commits verschwinden zu lassen. [9] Hier kann auch der Anhang ***–root*** von Nutzen sein, falls man einen Zweig überarbeiten will. Dadurch entfällt die vorherige Suche der Checksumme oder die Anzahl der Commits.

Um die vorhanden Commits nun zu verschmelzen wird der interaktive Modus geöffnet:

**git rebase -–interactive –root**

Daraufhin öffnet sich der git hinterlegte Editor,zu sehen in ABBILDUNG. Hier wird nun wie oben beschrieben die voranstehenden Schlüsselwörter je nach Anforderung geändert. In diesem Fall sollen die unteren 3 Commits zu einem zusammengeführt werden. Dafür erhalten die Zeilen 3 und 4 das Schlüsselwort *squash* anstatt *pick.* Dadurch werden beide dem Commit von Zeile 2 hinzugefügt.



Als nächster Schritt wird das Editierte gespeichert und der Editor geschlossen. Dadurch öffnet sich nun automatisch der Editor ein zweites Mal, zu sehen in ABBILDUNG.

Hier wird nun 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 erweitert. Ist die Datei wiederum gespeichert und wird geschlossen führt git den *rebase* nun dementsprechend aus.  
Die Kommandozeile sieht nun wie folgt aus:



Und dadurch wurde die Historie des Archivs von ehemals ABBILDUNG GIT LOG umgeschrieben zu:

Es wird nun weiter am Projekt entwickelt. Der Entwickler arbeitet an einer Erweiterung, will nun allerdings einen anderen Ansatz zwischenzeitlich verfolgen. Die bisherigen Änderungen der Erweiterungen sind allerdings nicht auf einem sinnvoll Stand, welche einen Commit möglich machen. Daher werden die Änderungen nun zwischengespeichert, sodass der Entwickler zwischenzeitlich einen anderen Ansatz verfolgen kann. Hierfür bietet git die Funktion *stash.*

## Arbeit mit Zweigen

### Git stash

Der Stash stellt eine Art Zwischenspeicher da. Da die Arbeitskopie bei einem Zweigwechsel überschreiben wird, sollten alle Änderung in einen Commit überführt werden. Falls man nun den Zweig wechseln will, jedoch Änderungen besitzt, welche erst zu einem späteren Zeitpunkt fertig werden, kann man diese im Stash bis zur Rückkehr zum Zweig ablegen. Auch können darin Ideen oder Ansätze gespeichert werden, um sie nicht zu verlieren. Da ein Zwischenstand auf jeden Zweig angewendet werden kann, wäre eine weitere mögliche Anwendung darin Änderungen abzulegen, welche das Projekt schnell und temporär verändern, zum Beispiel ein spezielles Debugging Feature. Der einfachste Grund ist die vorhandene Arbeitskopie zu reinigen, vorheriges aber nicht zwangsweise zu verlieren, falls man darauf zurückkommen sollte. Der Befehl ***git stash*** speichert alle Veränderungen zum Stand des letzten Commits des ausgewählten Zweiges. Mit der Option ***-patch*** kann wiederum nur ein Teil zwischengespeichert werden. Mit *-u* werden nicht versionsverwaltete Dateien mit einbezogen. Um eine Übersicht der Zwischenspeicherungen zu erhalten, listet der Befehl ***git stash list*** alle gespeicherten Zwischenstände auf. Um den Überblick zu behalten kann dem jeweiligen Zwischenstand eine Nachricht hinzugefügt werden. Dafür wird ***save <message>***benötigt. [2] Um den Stash 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, bei der Nutzung von *apply* bleibt dieser erhalten. Mit ***git stash show*** werden die Änderungen des ersten Eintrags in Kurzform angezeigt. Mit dem Anhang ***-p*** werden die Änderungen in Langform, das heißt zeilenweise wiedergegeben. Um einen Zwischenstand zu löschen kann der Befehl ***git stash drop <stash-id>*** verwendet werden, wobei die ID zum Beispiel *stash@{2] ist. [3]*

In dem Beispielprojekt speichert der Entwickler nun die bisherigen Änderungen:



Durch die leere Arbeitsmappe kann nun an einem neuen Zweig gearbeitet werden. Falls man allerdings sich diesen Aufwand nicht bereiten will, besitzt Git eine weitere Methode an einem weiteren Zweig separat zu arbeiten. Hierfür wird ein zusätzliches Arbeitsverzeichnis erstellt.

### Git worktree

[https://spin.atomicobject.com/2016/06/26/parallelize-development-git-worktrees/#](https://spin.atomicobject.com/2016/06/26/parallelize-development-git-worktrees/)

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 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 danach lediglich in dieses Verzeichnis navigiert werden.



Werden Änderungen am zusätzlichen erstellten Arbeitsverzeichnis durchgeführt, sind diese ebenfalls im Archiv 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 Arbeitsverzeichnis 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 wurde.

Noch immer befinden sich die Änderungen im Zwischenspeicher und der Entwickler kann nun in seinem leeren Arbeitsverzeichnis den neuen Ansatz verfolgen. Hierfür will dieser einen isolierten Entwicklungsstrang besitzten.

### Git branch

Zweigewerden angewendet, um isolierte Entwicklungszweige 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 man keine Komplikation durch Abhängigkeiten erhält. Auch besitzt man dadurch vorzugsweise einen gesicherter Hauptstand, auf diesen man sich im Normalfall beziehen kann. Aufgrund dieser Isolation lassen sich Projekte mit mehreren Entwicklern ohne größere Abstimmung und Nachverfolgungen untereinander durchführen. Intern sind Zweigefür git nur weitere Zeiger, welche auf Commits zeigen. Um einen neuen Zweigzu erstellen wird der Befehl ***git branch <Name>*** verwendet. Der neue Zweigführt nun von dem aktuellen aktiven Zweigab. Wird ein Zweignicht mehr benötigt lässt er sich mit ***git branch -b*** löschen. Dies ist nur möglich sofern alle Änderungen in andere Zweigeübernommen wurden. Ist dies nicht der Fall und man möchte ihn trotzdem löschen wird ein kapitalisiertes **-B** benötigt. Um alle Zweigedes Projektarchivs anzusehen wird ***git branch -l*** benötigt. Soll der lokale Zweig auch für das Referenzarchiv übernommen werden muss der Zweigähnlich wie ein Commit synchronisiert werden. Für git intern heißt das, dass der lokale Zweigeinem Zweigdes Referenzarchiv*s* folgen soll, es entsteht ein sogenannter *tracking branch.* Dafür gibt es wie fast für jede Funktion von git mehre Möglichkeiten, am einfachsten kann ***git push origin <Zweig>*** verwendet werden. Um sicherzustellen, dass der Zweigauch unter dem gleichen Namen im Referenzarchiv erstellt wird oder explizit unter einem anderen Namen, kann anstatt *<Zweig>* auch *<Zweig>:<Zweigname der Referenz>* benutzt werden. Ist man in dem Zweig*,* welcher hinzugefügt wird, kann auch *HEAD* stattdessenverwendet werden. Um solch einen Zweig wiederrum aus dem Referenzarchiv zu löschen, wird ***git push --delete <Zweig>*** verwendet. Manchmal ist es sinnvoll nachzuvollziehen zu können ob ein bestimmter Commit schon einem Zweig zugeführt wurde. Dafür kann ***git branch –contains <hash>*** verwendet werden. Nun werden alle Zweige aufgelistet, die diesen Commit beinhalten. Dies funktioniert jedoch nur, falls genau dieser Commit zugeführt wurde. Ist der Commit zum Beispiel über einen *chery-pick* oder *patch* zugeführt wurden, erkennt dies git nicht als gleichen Commit. Hierfür hilft das Kommando ***cherry***. [12]

In dem Beispielprojekt wird nun ein neuer Zweig mit dem Namen "feature" erstellt. In ABBILDUNG zu sehen signalisiert der Stern jedoch, dass der aktive Zweig immer noch "master" ist.



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

### Git checkout

Mit ***git checkout <branch>*** wird der aktuell zu bearbeitendem Zweiggewechselt. Hierbei müssen alle Änderungen in der Arbeitskopie zuvor einem Commit hinzugefügt werden, zwischengespeichert oder revidiert werden. Ist dies nicht der Fall schlägt das Kommando fehl. Intern wechselt *HEAD* auf den neusten Commits des gewünschten Zweiges*.* Allerdings können nicht nur Zweige übergeben werden, auch einzelne Commits und Dateien können mit *checkout* verwendet werden. Bei einer Datei wird damit die letzte Version dieser Datei der Arbeitskopie hinzugefügt beziehungsweise, die vorhandene Datei überschrieben. Dies ist nützlich falls man Änderungen an einer einzelnen Datei verwerfen will. [13] Wird ein Commit, besser gesagt dessen Checksumme, übergeben, entsteht dadurch ein sogenannter "*detached HEAD"*. Das heißt der *HEAD* Zeiger zeigt nun nicht auf den neusten Commit eines Zweiges*.* HEAD ist dafür zuständig, welche Versionen von Dateien dem aktuellen Arbeitsverzeichnis hinzugefügt werden. Ist HEAD nun nicht an den neusten Commit eines Zweigesgekoppelt, fehlt 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. Diese Abzweigung besitzt allerdings 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. Dies kann geschehen, indem man ein Etikettsetzt oder nachträglich einen Zweig kreiert mit ***git branch <name>***. [14]

Der Entwickler wechselt somit mit dem Befehl ***git checkout feature***, gemäß ABBILDUNG, auf den Zweig *feature* um eine neue Implementierung hinzuzufügen.



Nachdem die Implementierung durch einen Commit hinzugefügt ist, wird wieder auf *master* gewechselt, um die angefangenen Änderungen wieder fortzuführen. Dazu wird, in ABBIDLUNG zu sehen, der zuvor abgespeicherte Zwischenspeicher wieder auf die nun leere Arbeitsmappe in *master* angewandt. Da *pop* statt *apply* verwendet wurde, wird der Zwischenspeicher nach der Anwendung gelöscht.



Es wird nun ein weiterer Zweig erstellt, um eine neue Funktion hinzuzufügen, ohne Einfluss auf den bisherigen Stand in dem Zweig *master* zu haben. Diesmal wird zu Erstellung des Zweiges das Kommando *checkout* verwendet und ein Commit hinzugefügt:

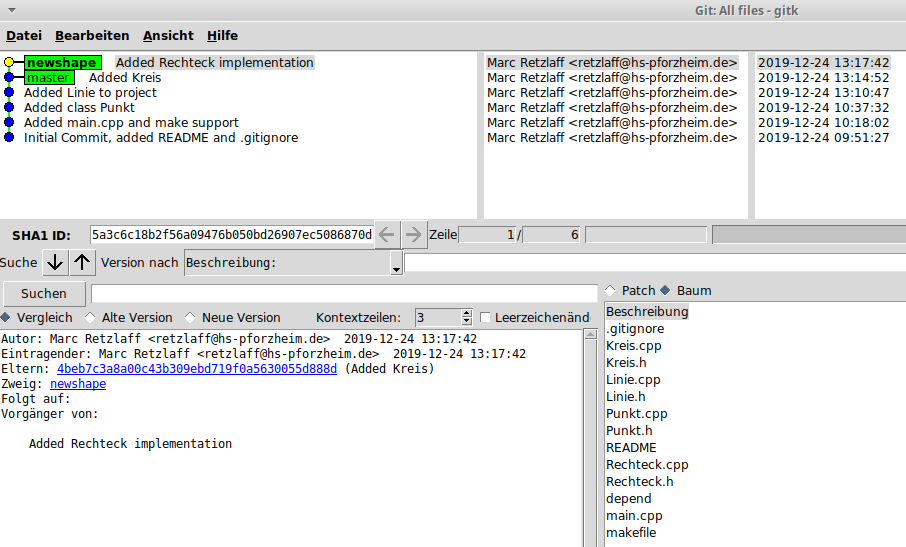


Um einen Überblick über unser Projektarchiv zu bekommen, kann man neben der Kommandozeile auch die git integrierte grafische Oberfläche verwenden.

### 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 über 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 aufgerufen durch *git gui*, die native grafische Oberfläche zur Erstellung von Commits. [19]

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



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

## Änderungen zusammenführen

### Git merge

Git verwendet einen sogenannten Drei-Wege-Merge. Merge wird dazu benutzt, um Zweige zusammenzuführen, beziehungsweise Änderungen von anderen Zweigen zu übernehmen. Um nun verschiedene Commits zusammenzuführen, werden beide Stände der Arbeitskopien Datei für Datei verglichen. Im Gegensatz zu einem Zwei-Wege-Merge, beziehungsweise einem einfachen zeilenweisen Abgleich, wird durch die Änderungsverfolgung von *git* auch der frühste gemeinsame Vorgänger beider Dateien einbezogen. Dadurch erkennt das Programm zur Zusammenführung (*merge tool)* neben der Abweichung beider Stände, unter Einbezug des Vorfahren, auch welcher der beiden Stände die zu übernehmende Änderung besitzt. In ABBIDLUNG sind einige Beispielhafte Zeilen zu sehen, welche nun zusammengeführt werden sollen. Zur Veranschaulichung welche Datei, welchen Stand der Datei im Projektarchiv darstellt, hilft ABBILDUNG. In Zeile 30 enthält die Datei "Yours" die Abweichung zu "Base" und "Mine". Dies ist die zu übernehmende Änderungen. Dabei ist dies der generelle Ansatz, nachdem Änderungen zusammengeführt werden sollen. Führt dieser Ansatz zu nicht gewollten Änderungen, muss die Zusammenführung händisch erfolgen. Der Vorteil dabei ist, dass Konflikte nur entstehen, falls beide Stände Abweichung zum Vorfahren besitzen. Zu sehen im rot markierten Bereich in Zeile 50. Dieser Konflikt muss per Hand gelöst werden. Sogenannte "merge tools" helfen dabei, schnell Änderungen Blockweise zu übernehmen oder ersetzen. Im letzten Beispiel ist eine Änderung nur in einer der beiden Dateien vorhanden, diese wird in die Zusammenführung übernommen.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

https://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

Des Weiteren wird nachdem eine Zusammenführung vollzogen wurde ein sogenannter *merge-link* erstellt. Dadurch wird sichergestellt, dass bei einer späteren Zusammenführung keine schon gelösten Konflikte noch einmal durchgeführt werden müssen, da der Vorfahre durch den *link* erneuert wird. Sollte in ABBILDUNG eine weitere Zusammenführung stattfinden, würde der gemeinsame Vorfahre nun "Mine" sein. Um nun solch eine Zusammenführung durchzuführen wird zuerst der Zielzweig als aktuelle Arbeitskopie gesetzt. Dann kann die Zusammenführung über ***git merge <source-branch>*** vollzogen werden. Besitzt der Zielzweig keine abweichenden Commits, sodass der Quellzweig nur Änderungen über den Zielzweig hinaus besitzt, findet eine *fast-forward* Zusammenführungstatt. Da keine Dateien zusammengeführt werden müssen, wird nur der interne Zeiger des Zielzweiges auf den Commit des Quellzweigs vorgerückt. Um das 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. Dies sollte man bei wichtigen Implementierungen zu Hauptzweigen verwenden, um die zusammengeführten Dateien manuell zu überprüfen. Danach kann der Commit der Zusammenführung manuell erstellt werden. Die Nachricht für den *merge-commit* wird mit ***-m*** übergeben. Auch kann man eine 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 wurde. Eine weitere nutzvolle Option bietet ***–squash***. Hierdurch werden alle Änderungen, welche in die Zusammenführung einfließen zu einem Commit zusammengefasst und als neuer Commit dem Ziel Zweig hinzugefügt. Es wird allerdings keine Verbindung zum Ursprungs Zweig hergestellt. Git sieht den Ursprungs Zweig 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***. [7]

Führen wir die Änderungen an unserem Beispielprojekt nun zusammen, signalisiert uns git, dass es sich dabei um eine *fast-forward* Zusammenführung gehandelt hat:



Die Historie des Projekts sieht nun wie folg aus:



Man sieht die beiden Zweige *newshape* und *master,* welche beide den neusten Commit beinhalten. Nun will der Entwickler die Änderungen des *feature* Zweigs dem Hauptzweig *master* zuführen. Dazu wählt er die zweite Hauptmethodik Änderungen zusammenzuführen. Hierzu verwendet er *rebase.*

### 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 zu gestalten. Im Wesentlichen versucht *rebase* den Vorfahren eines Zweiges zu verändern. Nehme man an, man wolle ein Entwicklungszweig, welcher seit längerer Zeit besteht, nun implementieren, aber der Hauptentwicklungszweig ist schon weit vorangeschritten. Damit es nun übersichtlich bleibt, will man den Entwicklungszweig vorschieben, damit der Vorfahre der Abzweigung der neuste Commit des Hauptentwicklungszweiges ist. Dies geschieht anhand dem Beispiel in ABBILDUNG mit dem Kommando ***git rebase maindev dev*** oder, falls man im grauen Entwicklungszweig ist (***git checkout dev***) reicht ein ***git rebase maindev .*** Dadurch zeigt nun der erste Commit von *dev* auf den letzten Commit von *maindev*, vorausgesetzt es entstand kein Konflikt.



Um nun zwei Zweige zu vereinen muss man nur noch einen Schritt weiter gehen, indem man nun umgekehrt mit ***git checkout maindev*** auf den *maindev* Zweig wechselt und ***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 also um die Zeiger von *dev* inkrementiert. Alternativ dazu könnte man auch einen einfache Zusammenführung über das Kommando ***merge*** erwirken oder ***git fetch dev:maindev*** verwenden um dementsprechend ein *fast-forward merge* zu bewirken. Neben diesem Hauptbestandteil von *rebase* bietet das Kommando noch einen deutlich größeren und komplizierteren Umfang. Mit dem Schlüsselwort ***–onto*** wird das Kommando zum Beispiel auf ***git rebase –onto maindev <start> <ziel>*** erweitert, um nur bestimmte *commits* zu übernehmen und deren Historie zu verändern. Generell gilt Vorsicht bei *rebase* da die Historie ein wesentlicher Vorteil der Versionsverwaltung darstellt. Wird die Historie zu arg verschlankt, da ein Großteil der Zweige nach Beendigung per *rebase* Hauptzweigen zugeführt werden, sind die Zweige zwar übersichtlich, allerdings lassen sich einzelne Arbeiten nicht mehr unterscheiden. Auch werden die Zeitpunkte der Commits verändert, sowie die Checksumme, daher sollte man sich vergewissern, dass niemand auf Commits explizit angewiesen ist, welche man per *rebase* verschiebt oder umschreibt. Deshalb sollte man nur in persönlichen Zweigen *rebasen* um Konflikte zu vermeiden, bei dem andere Zweige zum Beispiel auf veraltete Commits verweisen, da der Zweig umgeschrieben wurde. Auch hier sollte eine gesunde Mischung aus *rebase* und *merge* verwendet werden um eine schlanke, übersichtliche Historie zu erhalten ohne klare Strukturen und Abgrenzungen zu verlieren. Ein weiteres Problem bieten bereits gelöste Konflikte, die durch die erneute Anwendung der Änderungen auf die neusten Commits des gewünschten Zweigs wiederaufkommen können. [9] Zudem bietet das rebase Kommando auch eine interaktive Ansicht, welche zuvor behandelt wurde KAPITEL.

Wenden wir nun das rebase Kommando auf den Zweig *feature* an, geschieht folgendes:



Es wurden zwei Konflikte erkannt und das Kommando rebase wurde unterbrochen. Nun müssen die Konflikte gelöst werden bevor der Befehl fortgeführt werden kann.

### Konflikte

Konflikte entstehen unter Umständen bei *merge* oder *rebase* Kommandos. Durch den 3-Wege-Merge entsteht ein Konflikt nur, wenn sich beide Versionen an derselben Stelle mit der Version des gemeinsamen Vorfahren unterscheiden. In diesem Fall wird *merge* sowie *rebase* unterbrochen und der Konflikt benötigt eine manuelle Korrektur. 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 dargestellt. Zur Identifizierung startet bei dem sich überschneidenden Teil der Zielpfad mit "<<<<<<< HEAD". Die Version des Quellpfads steht gefolgt zwischen "=======" und ">>>>>>> <branch>". Um den Konflikt nun zu lösen müssen die merge-marker, sowie die ungewünschte Version entfernt werden. Danach die korrigierte Datei über ***git add <file>*** wieder hinzufügen, sodass der Konflikt für git als gelöst gilt. Gefolgt von ***git commit*** wird nun der *merge commit* vollzogen. Falls man nicht auf diese 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.

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

### Git mergetool

Falls es zu einem Konflikt kommt, sollte dieser bereinigt werden, um fortfahren zu können. Git unterbricht die automatische Zusammenführung,falls es zu Konflikten kommt. Mit *git status* werden die Dateien, welche einen Konflikt hervorrufen angezeigt. Um nun den Konflikt zu lösen muss der Benutzer die konfliktbehafteten Dateien anpassen, indem er die von git erstellten Versionen eben dieser Dateien seinen Vorstellungen nach anpasst. Durch solch einen Konflikt werden außerdem mehrere temporäre Dateien erzeugt, um jeden Stand widerspiegeln zu können. So werden automatisch eine *remote, base* und *local* Version erstellt, sowie eine backup Version je Datei. Diese Version besitzt ebenso die Zeilen, welche den Konflikt abgrenzen. Für eine schnelle und übersichtliche Lösung des Konflikts wird meistens ein *merge tool* verwendet. Um dieses Programm nicht explizit aufrufen zu müssen, wurde ***git mergetool*** bereitgestellt. Dadurch wird das hinterlegte Programm automatisch aufgerufen. Das hinterlegte Programm kann über ***git config*** verändert werden. Standardmäßig werden dadurch 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 nun die gestoppte Zusammenführung fortgeführt.

Führt man nun das Kommando ***git mergetool*** wie in ABBILDUNG aus, wird automatisch meld mit den bestehenden Konflikten gestartet.



In ABBILDUNG sieht man die Konflikte der Dateien "main.cpp". Dabei steht links die Version aus dem Zweig *master,* sowie rechts die Version aus dem Zweig *feature.* Rot zeigt dabei Konflikte, während blau Änderungen anzeigt.



Mit Hilfe der Pfeile können nun die unterschiedlichen Blöcke ausgewählt und der Ausgangs Datei in der Mitte hinzugefügt werden. In *meld* kann der Block komplett gelöscht werden über die Taste "shift" oder zusätzlich hinzugefügt werden, anstatt vorhandenes zu überschreiben, mit der Taste "strg". Dies ist zum Beispiel bei den "#include" Zeilen der Fall. Gelöst sie die Datei nun wie folgt aus:



Nachdem beide Konflikte gelöst sind kann der *rebase* fortgeführt werden. Bei manchen *merge tools*, auch *meld,* bleiben die ursprünglichen Konfliktdateien mit dem Anhang ".orig" bestehen. Diese Dateien beinhalten beide Versionen, getrennt durch sogenannte Konfliktmarker, beschrieben in KAPITEL. Die Projektarchiv Historie sieht nun wie in ABBILDUNG aus. Der Commit aus feature wurde nun ganz nach oben verschoben, da er nun auf dem neusten Commit von *master* basiert.



Im Umkehrschluss kann man nun die Änderung in die anderen Zweige integrieren indem man eine einfache fast-forward Zusammenführung veranlasst. Diesmal verwendet der Entwickler ein weiteres Mal *rebase* dafür. Dazu wechselt man auf den Branch, in den man die Änderungen implementieren will. Danach wird wie in ABBILDUNG ein rebase ausgeführt mit dem Zweig, welcher die Änderungen beinhaltet.



Nun wird das Zusammenspiel mit anderen Entwicklern beleuchtet. Bisher ist das Projektarchiv nur lokal vorhanden. Als Beispiel für ein Referenzarchiv wird in diesem Beispiel Github als Hoster verwendet. Um nun eine Verbindung des lokalen Archivs mit dem Referenzarchiv herzustellen wird der Befehl *remote* verwendet.

## Arbeiten im Team

### Git remote

Die Bezeichnung *remote* steht im Allgemeinen für den Ursprung des Projektarchivs*.* Grundsätzlich fallen aber auch andere Projektarchivedarunter, zu welchen das eigene Archiv eine Verbindung aufnehmen soll. Erstellt man ein lokales Archiv ist kein Eintrag vorhanden, da für git nur das Lokale existiert. Um die aktuellen remote Aliase einzusehen kann ***git remote -v*** verwendet werden. Erstellt man ein neues Projekt auf einem Hoster, welches man dann lokal klont, wird automatisch ein Eintrag erstellt, welcher nun auf den Hoster zeigt. Der Alias Name ist *origin.* Über ***git remote add <alias> <url>*** wird ein neuer Eintrag erstellt. Um den Alias jeweils nur für *fetch* oder *pull* zuverwenden, kann man die jeweiligen URL auch über ***git remote set-url <alias> <url>*** für *fetch* und mit Anhang ***[…] set-url –push […]*** respektive *push* erstellt werden. Dies wird zum Beispiel verwendet, um neue Änderungen von einem Hauptzweig zu erhalten aber jeder Entwickler hingegen zu einem eigenen Zweig wiederrum hochlädt. [16]

In dem Beispielprojekt wird nun ein neues leeres Archiv auf Github erstellt. Ist dies geschehen wird es als Referenzarchiv dem lokalen Archiv hinzugefügt, siehe ABBILDUNG.



Um die lokalen Änderungen nun zu synchronisieren wir das Kommando *push* verwendet.

### Git push

Der Befehl ***git push*** wird dazu benötigt die lokalen Änderungen im Projektarchiv auf den globalen Server zu legen, beziehungsweise das Referenzarchiv. Somit wird *push* nur benötigt, wenn man einen nicht lokalen Host nutzt, zum Beispiel das Archiv auf *github.com* liegt. Der globale Hauptstandort wird meist *origin* genannt. Um zum Beispiel Änderungen auf dem Zweig *master* auf dem globalen Repository 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 dieses Kommando nur unter gewissen Umständen benutzt werden. Mit dem Zusatz ***–all*** werden alle Referenzen des lokalen Archives synchronisiert, so muss nicht jeder Zweig einzeln abgefragt werden. Da es im Allgemeinen nicht gewollt ist, dass jeder direkten Zugriff auf das Archivbesitzt, 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 SSH erfolgen. Dafür muss das SSH Schlüsselpaar einmalig angelegt werden, dem Hoster der jeweilige Public Key mitgeteilt werden, sodass man über SSH nun Zugriff hat. Um dies zu nutzen muss dementsprechend der *remote* link auf SSH gewechselt werden. Ist keine Anmeldung automatische Anmeldung festgelegt, fragt git bei jedem Zugriff auf das Referenzarchiv. [4?]

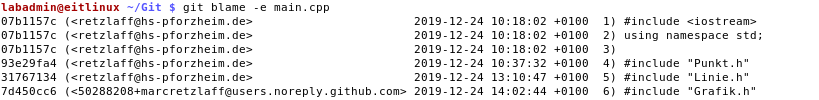
Da eine Verbindung zu Github nun besteht, wird das bisher leere Referenzarchiv mit dem lokalen Archiv überschrieben. Zuletzt sieht man, dass alle Zweige des Referenzarchivs gleichauf mit den lokalen Zweigen sind.



Da das Archiv nun auch für andere Entwickler zur Verfügung steht, finden nun auch unabhängig der lokalen Kopie Änderungen statt. Oftmals werden Änderungen von anderen Entwicklern am Projekt nicht sonderlich beachtet oder gesichtet. Es kann aber unter Umständen vorkommen, dass eine Zeilenänderung auffällt, welche sich als zweifelhaft herausstellt. Ohne die einzelnen Änderungsübersichten aller veranlasster Commits zu durchsuchen, hilft das Kommando *blame* dabei den Autor dieser Zeile zu finden.

### Git blame

Um herauszufinden welcher Autor welche Änderungen an einer Datei durchgeführt hat, wird das Kommando ***git blame <file>*** verwendet. Dadurch wird nun für jede Zeile der Datei der Autor bei der letzten Änderung 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 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 Projektarchivausgeweitet, beziehungsweise Dateiübergreifend. [26] In ABBIDLUNG sieht man die Ausgabe. Diese ist in fünf Spalten einzuteilen. Zuerst steht der Ausschnitt der Checksumme 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.



Um das lokale Archiv abzugleichen wird er Befehl *fetch* oder *pull* verwendet.

### Git fetch

Mit ***git fetch*** wird das Referenzarchiv unter dem alias *origin* mit dem lokalen Stand abgeglichen, allerdings werden Änderungen nur angezeigt und nicht automatisch versucht diese auch zu integrieren. Aufgrund dessen wird oftmals *fetch* mit anschließendem *merge* bevorzugt, da man dazwischen sich nochmals versichern kann, dass die gewollten Änderungen einfließen. Beide Befehle *fetch* und *pull* haben zum Großteil die gleiche Syntax um einzelne Zweige oder spezielle *remote* Server als Referenz zu verwenden. [4?]

### Git pull

Der Befehl ***git pull*** wird benötigt um das lokale Projektarchiv mit (falls vorhandenen) Änderungen von dem Referenzarchivupzudaten*.* Erweitert mit ***–all*** werde alle Zweige abgeglichen, mit ***git pull origin master*** wird nur *master* abgeglichen und falls vorhanden mit dem lokalen Stand zusammengeführt. Da es hierbei zu Konflikten kommen kann, sollte man nur den *pull* Befehl ausführen, wenn man eine frische Arbeitskopie besitzt, beziehungsweise, alle Änderungen zuvor einem Commit hinzugefügt hat*.* 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 *fetch* nutzen. Pull ist somit eine Kombination aus fetch und merge.[4?] Ein weiteres Schlüsselwort ist ***–rebase.*** Dies führt bei lokalen Abweichungen zum 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 ursprünglichen lokalen Commits wieder angehangen. Dadurch besteht keine Divergenz zwischen den beiden Zweigen mehr, außer dass der lokale Zweig zusätzliche Commits am Ende beinhaltet. Somit ist der lokale Zweig dem Referenzzweig voraus und die Änderungen können nun synchronisiert werden. Dies ist der Zusammenführung vorzuziehen, da mit dieser Variante keine zusätzlichen Commits benötigt werden, sobald sich der Referenzzweig ändert. [31]

In der Zwischenzeit wurde ein Commit dem Referenzarchiv hinzugefügt. Dies wurde durch den Befehl ***git fetch origin*** festgestellt. In ABBILDUNG sieht man einen neuen Zweig "origin/try" welcher einen Commit beinhaltet. "origin" steht dabei für Zeiger des Referenzarchivs.



Um den Zweig nun lokal zu besitzen wird der Befehl ***git checkout try*** verwendet. Die Änderungen sollen nun auch in *master* implementiert werden. Dazu wird nun *fetch* verwendet. Mit dem Kommando:

***git fetch . try:master***

werden alle Commits von *try*, welche nicht auf *master* sind, *master* hinzugefügt. In kurz: Der Commit von try ist nun in *master.* Der Punkt steht hierbei für das lokale Archiv, beziehungsweise woher er die Änderungen bezieht. Dies funktioniert, da wir zuvor durch *checkout* den Zweigtry auch lokal besitzen.

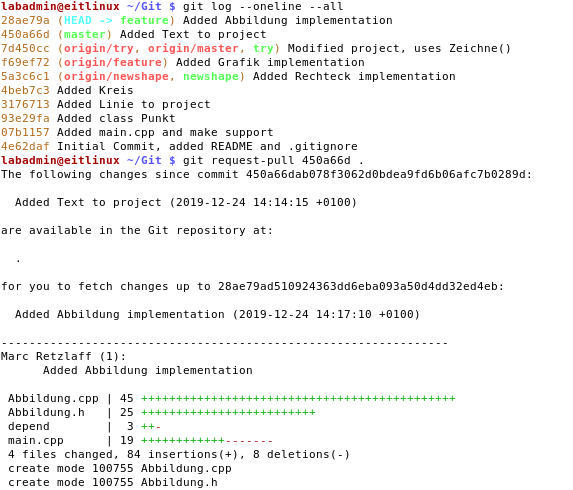


Nachdem weitere Änderungen hinzugefügt wurden auf *master*, und auch auf *feature*, sollen diese Änderungen wiederum zusammengeführt werden. Diesmal dient es dazu eine weitere git Funktion zu präsentieren: *request-pull*.

### 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 weiterverbreitet hat. Allerdings hat diese Funktion mit dem *git* Kommando ***git request-pull <start> <url>*** nur bedingt Gemeinsamkeiten. Durch das git Kommando werden Differenzen ermittelt und ausgegeben. Mit dieser Ausgabe kann der Ersteller sich nun an eine andere Partei wenden. Anhand dieser Informationen weiß der Integrator, der Besitzer des Projektarchivs, welches man erweitern möchte, wo die Änderungen zu finden sind und was diese beinhalten. Dadurch wird der Vorgang erleichtert und kann schneller ablaufen. 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 Hostern, 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, auf welchem die Änderungen für den Integrator zu finden sind. Unter ***<start>*** kann entweder die Hashsumme eines Commits übergeben werden, oder der Zweig, gegen den man vergleichen will. Dies ist hilfreich falls man ein *request-pull* für einen kompletten Zweig erstellen will, da dadurch automatisch der letzte gemeinsame Vorfahre der beiden Zweige ermittelt wird. [20]

Um nun ein *pull-request* zu erstellen, schaut man zuerst in die Historie, um den Hash des gewünschten Commits zu erhalten. In diesem Fall sieht es somit wie folgt aus:



Durch die Übergabe des Punktes besitzt nun 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 einem persönlichen Referenzarchiv übergeben, damit der Integrator diese auch abrufen kann. Auch sollte das Referenzarchiv synchronisiert zum lokalen Archiv sein. Daher sollte zuerst der Zweig feature abgeglichen werden. Da wir uns im gleichen Projekt befinden, können die Änderungen von *feature* gemäß ABBILDUNG integriert werden.



Eine andere Möglichkeit wäre, sofern *master* der aktive Zweig ist:



Dadurch ist das Projekt an einem Release Stand. Um dies auch in git zu notieren, werden Etiketten verwendet.

## Veröffentlichung markieren

### Git tags

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. Gitunterscheidet dabei zwischen kommentierten und nicht kommentierten Etiketten. Um ein Etikettzu erstellen wird ***git tag <name>*** verwendet. Um zusätzlich eine Nachricht anzufügen kann ***-a*** ***-m '<Nachricht>'*** angehangen werden. Um sich die Liste aller vorhandenen Etiketten anzusehen bietet *git* den Befehl ***git tag***. Ein Etikettbehandelt git grundsätzlich als Verweis auf einen bestimmten Commit. Damit das Etikettauch im Referenzarchiv sichtbar wird, muss es wie ein Commit synchronisiert werden, also zum Beispiel ***git push origin v1.0*** um das Etikett *v1.0* zu synchronisieren. Um weitere Information über ein Etikett zu erhalten wir das Etikett dem Befehl *show* übergeben. [5]

In diesem Fall ist der 3 Meilenstein erreicht und das Programm erstmal funktionell nutzbar. Daher wird ein Etikett mit Anhang erstellt, zu sehen in ABBILDUNG.



### 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 Hash übergeben werden, genauso können Etiketten angezeigt werden. [21] Übergibt man dem Kommando die zuvor erstellte Etikette, zu sehen in ABBILDUNG, wird neben den Information zur Etikette auch der Commit angezeigt, auf welchen die Etikette verweist.



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 Änderungen beinhaltet. Diese Datei wird gemäß ABBILDUNG erstellt.



Hier wird nun die Ausgabe des Befehls *diff* in eine Datei umgeleitet, welche die Änderungen des aktuellen Standes zum Etikett "v4.0" besitzt.

## Eine Änderungsdatei erstellen

### Git diff

Mit dem simplen Kommando ***git diff*** werden Dateiunterschiede angezeigt, welche nicht im Index Bereich sind. Hinzugefügte Dateien werden 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***. [23]

Um die erstellte Datei wieder auf eine Arbeitskopie anzuwenden, wird das Kommando *apply* verwendet. Wie in ABBILDUNG zu sehen, besitzt es zuvor keine Änderungen. Nachdem die Änderungsdatei angewendet wurde, befinden sich 2 Änderungen in der Arbeitskopie.



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

### Git patch

Über das *patch* Kommando werden ähnlich zu einem *pull-request* von Hostern schnell größere Änderungen übertragen und integriert. Um einen *patch* zu erstellen wird ursprünglich ***git format-patch <branch> -1 <hash>*** verwendet. Die *-1 <hash>* bedeutet hierbei, dass lediglich der Commit in den *patch* einfließt. Um Zweige zu vergleichen und somit jeden *Commit* in einen *patch* zu verpacken, welcher nicht auf dem nicht 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*** angehangen. Die *patch* Dateierhält je nach *git* Konfiguration einen vordefinierten Header, welcher ursprünglich die Verschickung per Mail direkt möglich machte, siehe ABBILDUNG.



Der Empfänger kann, die empfangen Dateien über ***git am <file>*** seinem Projektarchiv hinzufügen. Diese werden direkt als Commit hinzugefügt. Eine andere Möglichkeit einen Patch zu erstellen schafft ***git diff <von> <bis> > <output>***. Dieser besitzt dadurch lediglich die Unterschiede der Dateien und somit keinen Header. Es gilt zu beachten, dass *diff* nicht jede Datei mit einbezieht, für binäre Dateien muss zum Beispiel ***–binary*** als Schlüsselwort übergeben werden. Die resultierende Datei kann danach über ***git apply <output>*** auf die aktuelle Arbeitsmappe angewendet werden. [17]

Wird anstatt ABBIDLUNG DIFF > patch.diff nun *format-patch* verwendet, erstellt git automatisch eine nummerierte Datei, zu sehen in ABBILDUNG.



Auch diese Datei wird wieder angewandt gemäß ABBILDUNG. Da die Datei allerdings über *format-patch* erstellt wurde, wird der Commit direkt dem Zweig angefügt, anstatt die Änderungen in die Arbeitskopie zu übernehmen.



Zum Abschluss des Beispielprojekts werden weitere Änderungen durchgeführt, allerdings enthält der letzte Commit einen großen Fehler. Um schnell einen vollführten Commit zu revidieren kann der Befehl *revert* verwendet werden.

## Änderungen rückgängig machen

### 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 letzten Commit wiederspiegelt. Dies kann auch erweitert werden indem man den Hash oder respektive ***HEAD~<Anzahl an Commits>*** übergibt. 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 Referenzarchivsynchronisiert wurden. Ist dies geschehen, können keine Commits gelöscht werden, da git den 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 Repositorys gegeben falls Verweise besitzen. Deshalb ist diese Methode nicht weit verbreitet. Um *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 vornehmen.

In diesem Beispiel soll der letzte Commit revidiert werden, da er die komplette 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 aus:



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



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

### Git reset

Falls man mehrere Änderungen lokal durchgeführt hat, welche sich als fehlerhaft herausstellten, kann ein Zurücksetzen der lokalen Arbeitsmappe Abhilfe schaffen. Dies geschieht mit ***git reset –hard <commit>.*** Dadurch werden alle durchgeführten Änderungen unwiderruflich verworfen. Ohne das Schlüsselwort ***–hard***, finden sich die Änderungen in der Arbeitskopie wieder. Falls ein Commit mit übergeben wird, wird der Zweig auf den Stand des Commits zurückgesetzt. Über ***git reset <file>*** kann außerdem die Datei aus dem Index entfernt werden. Die Änderungen der Datei bleiben aber weiterhin erhalten. [22]

Fall der Commit nicht synchronisiert wurde, ist es einfacher den Commit zu entfernen über das *reset* Kommando. In ABBILDUNG zu sehen wird zuerst ein *reset* ohne das Schlüsselwort ***–hard*** verwendet. Dadurch befindet sich eine modifizierte Datei im Arbeitsverzeichnis. Ein zweites Mal wird der befehl mit diesem Schlüsselwort verwendet. Die Arbeitskopie ist auf dem Stand des zurückgesetzten Commits danach.



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

### Git reflog

Während *log* die Historie erstellt indem der derzeitige HEAD Zeiger verfolgt wird, 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. Deshalb wird ***git reflog*** häufig in Verbindung mit ***git reset HEAD@{x}*** benutzt, wobei ***HEAD@{x}*** aus dem *reflog* entnommen wird. Auch falls ein Commit durch ein *reset* verloren ging ist dieser nicht gelöscht, sondern nur ohne Verweise. Auch in diesem Fall ist die Wahrscheinlichkeit hoch diesen im *reflog* zu finden. Commits ohne einen Verweis werden nach einer festgelegten Zeit verworfen. [24]

Öffnet man nun den *reflog,* sieheABBILDUNG*,* 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 Kommandos sind nur noch im *reflog* vorhanden. Auch hierbei werden durch ***–hard*** alle Änderungen entfernt, beziehungsweise ohne, die geänderten Dateien in der Arbeitskopie abgelegt.



## Einzelne Commits überführen

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

### Git cherry-pick

Das Kommando ***git cherry-pick <hash>*** 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 Zeile ein, welche auf den ursprünglichen Commit verweist. Auch hier können Schlüsselworte wie ***-edit*** oder ***–no-commit*** können angewandt werden. [28] Da der Verlust der Herkunft problematisch sein kann, wurde dafür ein weiterer Befehl eingeführt: *cherry.*

### Git cherry

Um nachvollziehen zu können ob Änderungen einem Zweig zugeführt wurden kann ***git cherry -v <branch1> <branch2> <limit>*** verwendet werden. Das Kommando überprüft die eingeführten Änderungen jedes Commits, anstatt eine übereinstimmende Prüfsumme wie *branch –contains.* Dadurch werden auch Commits, welche über *cherry-pick* oder *patches* zugeführt wurden, erkannt. Das Kommando überprüft dabei, ob die Commits aus *<branch2>* in *<branch1>* existieren. Das Limit gibt an, wie viele Commits rückwirkend durchsucht werden. Dies ist hilfreich, um die Ausgabe einzugrenzen. Mit ***-v*** wird neben dem HASH auch die 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 <*branch1>.* [29]

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



Die Ausgabe des Befehls *cherry* erkennt allerdings die übereinstimmenden Änderungen und markiert den Patch-Commit aus *master* als integriert an:



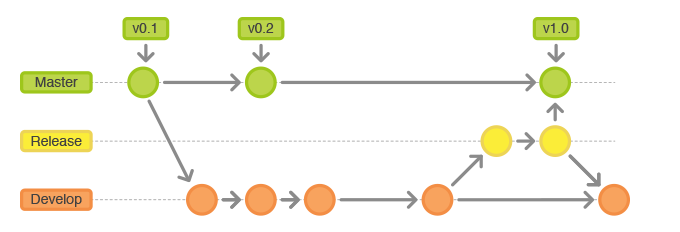
# Workflows mit Git

In Folgenden werden mehrere Vorgehensweisen beleuchtet wie man Entwicklungszweige eingliedern kann. 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.

## GitFlow

Im Allgemeinen wird GitFlow als die Richtlinie für ein gut durchdachtes und schlank strukturiertes Projektarchives angesehen. Hierbei wird der generelle Fokus auf die Separierung der Tätigkeiten in feste Zweige gelegt. So besitzt ein Projekt grundsätzlich 5 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*. Dieser führt alle Änderungen und Neuerungen des Projekts zusammen. Unter Umständen kann bei einem größeren Projekt diese Funktion auf mehrere Zweige unterteilt werde. 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 Funktion von dem Hauptentwicklungszweig auf den Zweig für Veröffentlichung überführt werden. Die Definition des *release* Zweiges lässt auch noch weitere Änderungen und Vorbereitungen 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 darstellen.

<https://m.infos.seibert-media.net/Productivity/Git-Workflows+-+Der+Gitflow-Workflow.html>



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 projektangepasste Namenskonvention angewendet. Im Allgemeinen wird einfach mit der Art des Zweiges gestartet, 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 Entwicklern deutlich einfacher, da diese separiert von Änderungen anderer 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:

Als letzte Art von Zweigen fehlt noch der Korrektur Zweig, welcher allgemein unter *bugfix* 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 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 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 Version, welche unterstützt werden soll, einen eigenen *master* besitzen. Dadurch können solche Korrekturen diesem Zweig immer angehangen werden und stellen dadurch die neuste Version dieser Produktversion 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 Zusammenführen vereinfacht. Diese Erweiterung sollte nativ unterstützt sein und wird über ***git flow*** aufgerufen. Diese Erweiterung hilf 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.

## The Linux Way

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

Dabei empfiehlt es sich, diese Separierung nach Themen oder Modulen durchzuführen. Dies kann natürlich 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 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 Medium 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 KAPITEL beschrieben. Sollte man allerdings einen solchen Dienst nicht in Anspruch nehmen, kann über diese ältere aber bewerte Arbeitsweise, Änderungen besprochen und die daraus abgleitenden Änderungsdateien angewendet werden.

## Pull Request Workflow

<https://blog.oio.de/2014/09/22/git-workflows-teil-1-warum-wir-workflows-brauchen/>

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. Dies wird als *Fork* bezeichnet. In dieser lokalen Kopie kann nun isoliert gearbeitet werden. Im Normalfall wird für Änderungen, welche zurückgeführt werden, ein separater Zweig erstellt, welcher die zu bearbeitende Funktionalität wiederspiegelt. 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, die Historie des Autors vernachlässig 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 angehangen. 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.

<https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-request-merges>

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 durchgesehen haben oder ein Build-Prozess, welcher über die Erstellung eines Requests gestartet wird, erfolgreich das Projekt gebaut und getestet hat. Dies sind nur zwei Beispiele welche über die Funktionalität von einem Pull-Request, bei jeder Zusammenführung durchgeführt werden müssen, falls konfiguriert. Da diese Zusammenführungen nun innerhalb des Archivs stattfinden und nicht wie bei einem Fork von außerhalb stammen, spielt die Durchführung der Zusammenführung eine Rolle. Hierbei werden nun die Github.com Optionen beleuchtet. Einerseits können die Änderungen über *merge --no-ff* hinzugefügt werden. Dadurch wird immer, auch bei einem theoretischen *fast-foward 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 wird dies durch den untersten Pfeil dargestellt. Die Verbindung zwischen dem ursprünglichen Zweig bleibt immer erhalten.



<https://myst729.github.io/posts/2019/on-merging-pull-requests/>

Neben dieser Option, wird beim zweiten Pfeil *merge --squash* verwendet. Wie auch schon in KAPITEL (git merge) 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 angehangen. Auch hierbei ist der Quellzweig nun ein toter Zweig unter Umständen. Im Gegensatz zur zweiten Option, werden 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 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.

## Die projektangepasste Kombi machts

Um nun für sein Projekt die bestmögliche Arbeitsweise herauszufinden, 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 den Bedingungen angepasster Workflow bei Beginn festgelegt werden, und sollte im Verlauf nur noch falls benötigt, leicht abgewandelt werden. Als Grundmotto lässt sich sagen, je mehr Personen beteiligt sind, desto sinnvoller ist es auf mehreren Zweigen, beziehungsweise von der Referenz separiert zu arbeiten. Dies kann dennoch bei kleineren Projekten, ein Zweig pro Person heißen oder nur einen Entwicklungszweig. Hier entscheidet die Art des Projekts, ob dies Sinn ergibt. Diese Antwort gibt der zweiten Frage 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 allerding 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 Zweiges auch dem Verlauf der Versionen 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. Hierbei kann nun 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 Verbesserungen und Vorbereitungen, kann es auch sinnvoll sein, diese auf einem getrennten Zweig zu bearbeiten. An diesem Zeitpunkt steht der strukturelle Aufbau des Projekts fest, damit 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 keine Gründe, nicht jegliche größeren Zusammenführungen über einen Pull-Request abzuhandeln. Hierdurch können auch weitere immer wichtiger werdende Funktionalitäten einfach umgesetzt und ermöglicht, wie eine automatische Anstoßung einer Build-Umgebung. Zudem bietet die mögliche Festlegung auf Bedingungen, welche erfüllt sein müssen, bevor die Zusammenführung stattfinden kann. 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 freigeben muss. Zusammenfassend lässt sich somit sagen: die Grundstruktur von Gitflow bietet den optimalen Aufbau des Projektarchivs 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.

# Git cheatsheet

<https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf>

#### Erstellen eines Projektarchives

git init - Verwandelt Ordner zu einem Projektarchiv

git clone <url> - Kopiert ein bestehendes Projektarchiv, um lokal daran zu arbeiten

#### Konfiguren der Git Einstellungen

Git config user.name "<Name>" – Setzt Namen, welcher den Commits angeheftet wird. Weitere Optionen: user.mail, core.editor.

#### Handhabung von Änderungen

Git status – Einsicht in den Status der lokalen Arbeitskopie.

Git add . – Fügt alle Änderungen der Arbeitskopie dem Index hinzu.

Git commit -m "<Nachricht>" – Erstellt lokal einen Commit mit Änderungen, welche dem Index hinzugefügt wurden.

#### Synchronisierung von Änderungen

Git fetch <remote>– Aktualisiert Referenzen, Änderungen in der Referenz werden lokal sichtbar.

Git pull <remote> - Aktualisiert Referenzen und führt dessen Änderungen ein. Kombination aus git fetch und git merge.

Git push <remote> - Aktualisiert das Referenzarchiv mit den lokal hinzugefügten Änderungen.

#### Erfolgte Änderungen einsehen

Git log –oneline –all -Listet die Historie des gesamten Archivs in Kurzform auf.

Git log –follow <Datei> - Listet Änderungen an einer Datei auf.

Git show <commit> - Zeigt Information sowie Änderungen eines Commits auf.

#### Änderungen revidieren

Git reset <commit> - Setzt Arbeitkopie auf Stand des übergebenen Commits wieder, die nachfolgenden Änderungen sind in der Abreitskopie. Mit --hard werden die Änderungen verworfen.

Git revert <commit> - Erstellt einen neuen Commit welche eingeführten Änderungen des übergebenen Commits revidiert. Benutzt, um Commits im Referenzarchiv zu revidieren.

#### Zweige erstellen und zusammenführen

Git checkout <Zweig> - Wechselt den aktiven Zweig, mit -b wird dieser neu erstellt.

Git branch -D <Zweig> - Löscht einen Zweig ohne zu Überprüfen ob Änderungen in andere zweige übernommen wurden.

Git merge <Zweig> - Fügt Änderungen des übergebenen Zweiges dem aktiven zweig hinzu und erstellt in Folge dessen ein Commit.

#### Änderungen suchen

In Commit Nachrichten:  
Git log -- all -i --grep '<Suche>' - Durchsucht die Commit Nachrichten nach der übergegebenen 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 angehangen. Mit -G anstatt -S kann auch ein regulärer Ausdruck übergeben werden.

#### Referenzarchiv und lokales Projektarchiv besitzen Neuerungen

Git pull --rebase – 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 – 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.

#### Referenz hinzufügen

Git remote add <Name> <url> - Fügt eine Referenz unter <Name> hinzu.

# Gits interne Dateiverwaltung

## Wie speichert Git die Daten <https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>

Durch die dezentrale Projektverwaltung von Git benötigt ein Projekt nicht nur zentral eine größere Menge an Speicherplatz. Dadurch ist 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 sie sind über ihren Inhalt adressiert. Bevor dies jedoch geschieht erfolgt eine geringe Komprimierung mit Zlib, sowie eine Voranstellung eines Headers. 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 Tree Objekte sein. Wird nun ein Commit erstellt kann dadurch wiederum ein Objekt erstellt werden, welches einen Tree referenziert. Dieser beinhaltet dann wiederum alle Referenzen der Dateien, welche das damit festgehaltene Projektarchiv besitzt. Ein Etikett nutzt das gleiche Vorgehen, jedoch zeigt es auf einen Commit. Folglich entsteht dadurch eine solche Struktur: ABBILDUNG

Durch diese Struktur wird neben den Dateien in komprimierter Form nur wenig Speicheraufwand betrieben, um die Verwaltung darzustellen. Der Vorteil der Verwaltung ist nun die Arbeit mit Referenzen. Hierbei kann ein Blob in mehreren Trees vorkommen und somit auch in mehreren Commits. Genauer, es wird ein Blob einer Datei nur 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.

## Pack

<https://git-scm.com/book/en/v2/Git-Internals-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 bevorzugt.

## 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 alles 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 GITINTERNALS behandelt. Hier wird die Datei nur abgespeichert, falls sich die Checksumme 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 ***fetch*** oder ***clone*** nicht übertragen zu müssen. Ist LFS installiert und aktiviert, werden nun bestimmte Dateitypen über dieses System verwaltet. Hier wird vereinfacht ein Container für diese Dateien angelegt, um sie einmalig zu speichern, zum Beispiel auf dem Referenzserver oder falls gewünscht auch einem zusätzlichen Server. Hierfür werden intern für Git weitere Zwischenschritte angewendet, sodass bei einem Verweis auf diese Dateien lediglich eine Referenz, zeigend zu dem Container eingefügt wird. Dadurch wird nun nur noch ein Verweis in Form einer SHA256 Checksumme im Commit gespeichert anstatt der Datei. Wird die Datei verändert wird sie wiederum einmalig im Container gespeichert. Die Container werden mit den nativen git Kommandos *Push* und *Pull* synchronisiert, sodass alle kürzlich referenzierten Dateien auch im lokalen Container 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 somit nicht lokal vorhanden. Um übergreifend damit arbeiten zu können, müssen alle git Instanzen, welche mit diesem Repository 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]