## Git in Kurz

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 git 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 Archivs. Die Baumstruktur entsteht aus Abzweigung zwischen Commits. Diese Zweige werden verwendet, um die Entwicklung unabhängig von einander gelichzeitig 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. 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.

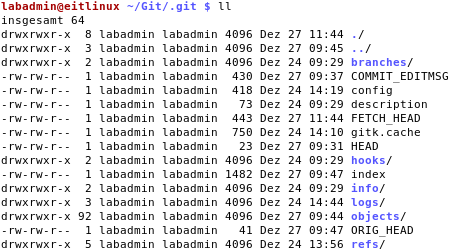
## Git und seine Komandos

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.

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.*

## 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.



Commits befinden sich verschlüsselt in dem Ordner objects. *Refs* enthält Verweise auf die verschiedenen Branches und Tags. Skripte, die bei git Kommandos ausgeführt werden, befinden sich unter hooks. Die *Config* Datei enthält Information und Einstellungen für das Projektarchiv. Erstellt man ein *shared repository* welches als reines Archiv zur Synchronisationgenutzt wird, fügt man dem Kommando das Schlüsselwort ***–bare*** hinzu. Dies führt dazu, dass es keine Arbeitskopie erhält, sodass man auf dem zentralen Projektarchiv keine Dateien editieren oder Commits hinzufügen kann. Um dies zu tun, muss 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.

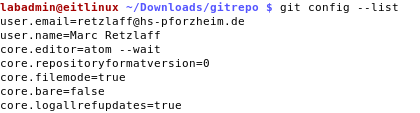


Falls keine archivübergreifende Konfiguration existiert, muss git zuerst mitgeteilt werden, welcher Entwickler die Änderungen lokal durchführt, sodass jeder Commit einem Autor zugewiesen werden kann. Hierfür muss die Konfiguration geändert werden.

## Git config

Wird eine neues Projektarchiv erstellt oder auch nach einer lokalen Installation von git sollte man Einstellungen vornehmen, um git nach seinen eigenen Wünschen zu konfigurieren. Zuerst sollte man allerdings seine persönlichen Daten anlegen, 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*** die zwei wichtigsten Informationen gesetzt. Weitergehend kann der Standard Editor geändert werden: 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 *origin* hinterlegt. So kann 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.



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 Buildprozess 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 exzitiert 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 Teile 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. ABBILDUNG??? 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 Bugfix Branch 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 steht. 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 (*tags)* 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.

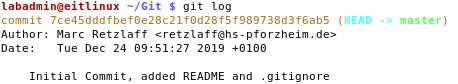


Zur Überprüfung kann man die Historie des Projektarchivs einsehen. Hier sollte der Commit nun zu sehen sein. Dafür verwendet man das Kommando *log.*

## 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*** werden nur Commits mit passender Commit Nachricht angezeigt. Um auch in den Änderungen der jeweiligen Commits zu suchen, wird ***-S<search>*** 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. [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.*

## 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 interaktive *rebase* Modus, bei dem man jeden einzelnen Commit betroffen vom *rebase* 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 schon 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 des Hash oder die Anzahl der Commits.

Um die vorhanden Commits nun zu verschmelzen öffen wir den interaktiven Modus mit:

**git rebase -–interactive –root**

Daraufhin öffnet sich der git hinterlegte Editor, ABBILDUNG, hier wird nun wie oben beschrieben die voranstehenden Schlüsselworte 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 verschmolzen 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 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 Projektarchivs von 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.*

## Git stash

Der Stash stell eine Art Zwischenspeicher da. Da die Arbeitskopie bei einem Zweigwechsel überschreiben wird, sollten alle Änderung committet werden, da sie sonst überschreiben 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 Stash 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 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 ausgecheckten Pfads. 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 Stash 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 Stash nach der Anwendung gelöscht, bei *apply* bleibt er erhalten. Mit ***git stash show*** werden die Änderungen des ersten Stash Eintrags in Kurzform angezeigt. Mit dem Anhang ***-p*** werden die Änderungen in Langform, das heißt zeilenweise wiedergegeben. Um einen Stash 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.

## Git branch

*Branches* werden angewendet, um isolierte Entwicklungszweige zu erhalten. So ist es von Vorteil eine Sanbox des aktuellen Arbeitsstand zu verwenden, um Entwicklungen jeglicher Art vorzunehmen, ohne dabei die Gefahr zu haben instabilen Code im Hauptzweig zu erhalten. Auch kann gerade bei Projekten mit mehreren Entwicklern unter Verwendung von Zweigen sichergestellt werden, dass man sich nicht gegenseitig in die Quere kommt und immer ein gesicherter Hauptstand da ist, auf welchen man sich beziehen kann. Aufgrund dieser Isolation lassen sich Projekte mit mehreren Entwicklern ohne größere Abstimmung und Nachverfolgung untereinander erledigen. Intern sind Zweigefür git nur weiterer Zeiger, welcher 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 (shared repository)übernommen werden muss der Zweigähnlich wie ein Commit synchronisiert werden, beziehungsweise der lokale Zweigeinem Zweigdes Referenzarchiv*s* folgen soll, ein sogenannter *tracking branch.* Dafür gibt es wie fast für jede Funktion von git mehre Möglichkeiten, am einfachsten kann ***git push origin <branch>*** verwendet werden. Um sicherzustellen, dass der Zweigauch unter dem gleichen Namen im Referenzarchiv erstellt wird oder explizit unter einem anderen Namen, kann anstatt *<branch>* auch *<branch>:<remotebranchname>* benutzt werden. Ist man in dem Zweig*,* welcher hinzugefügt wird kann auch *HEAD* stattdessenverwendet werden. 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 werden alle Änderungen in der Arbeitskopie behalten. Intern wechselt *HEAD* auf den neusten Commits des 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. Die kann man, 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 ein 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 das bisherige Programm auf *master* zu haben. Diesmal wird zu Erstellung des Zweiges das Kommando *checkout* verwendet und ein Commit hinzugefügt:

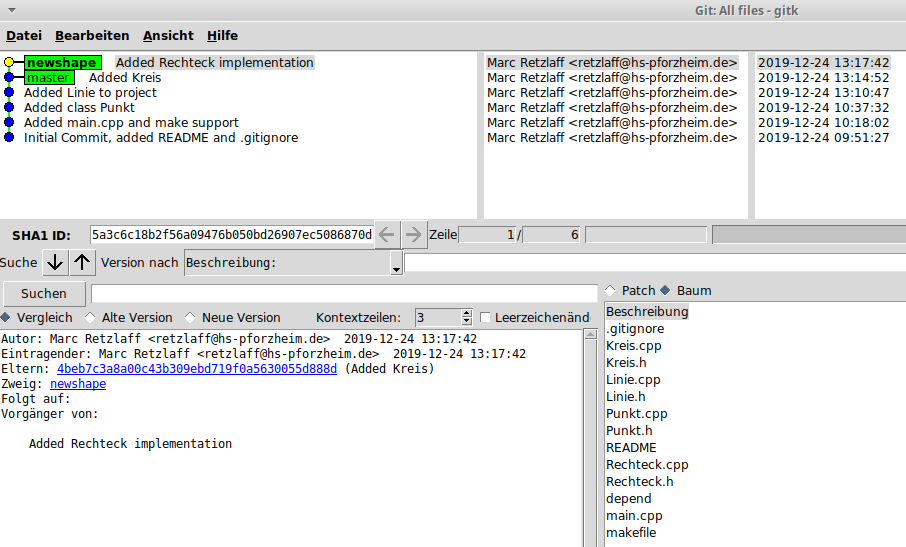


Um einen Überblick über unser Projektarchiv zu bekommen, kann man neben der Kommandozeile auch die git integrierte grafische Oberfläche verwenden.

## Gitk

Um eine schnelle Übersicht 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.*

## Git merge

Git verwendet einen sogenannten Drei-Wege-Merge. Merge wird dazu benutzt, um Zweige zusammenzuführen, beziehungsweise Änderungen von anderen 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 Zusammenführungs Programm (*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. Dies muss händisch 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 ein Merge vollzogen wurde ein *merge-link* erstellt. Dadurch wird sichergestellt, dass bei einem späteren Merge keine schon gelösten Konflikte noch einmal durchgeführt werden müssen, da der Vorfahre durch den *link* mitgezogen wird. Um nun solch einen Zusammenführung durchzuführen wird zuerst der Zielzweig als aktuelles Projektarchiv 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 ein *fast-forward merge* statt. 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 Merge Commit manuell erstellt werden. Die Nachricht für den *merge-commit* wird mit ***-m*** übergeben. Auch kann man eine Strategie für den *merge* wählen. Die zwei wichtigsten sind **-*ours*** und ***-theirs***. Damit kann bei Konflikten generell eine Version bevorzugt werden. Eine weitere nutzvolle Option bietet ***–squash***. Hierdurch werden alle Änderungen, welche in den merge 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 kamen 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 Methodik 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 mit dem Kommando ***git rebase maindev dev*** oder, falls man im 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, zu sehen in ABBILDUNG.



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 umgekehrt ***git rebase dev*** auslöst. Folgend werden nun alle Commits von *dev* auf *maindev* angewendet. Da *dev* durch den ersten *rebase* aus dem neusten Commit herausgelöst wurde, ist die Integration folgend nur noch ein *fast-forward-merge*. Der Zeiger von *maindev* wird also um die Zeiger von *dev* inkrementiert. Alternativ dazu könnte man auch ***git fetch dev:maindev*** verwenden um ein *fast-forward* 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. [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 der rebase wurde unterbrochen. Nun müssen die Konflikte gelöst werden bevor der rebase 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 als gelöst gilt. Gefolgt von ***git commit*** wird nun der *merge-commit* vollzogen. Falls man nicht auf diese Konfliktlösung zurückgreifen will kann der *merge* Versuch 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 mergetool gestartet werden. In diesem Fall ist das Programm "meld" hinterlegt.

## Git mergetool

Falls es zu einem *Merge* Konflikt kommt, sollte dieser bereinigt werden, um fortfahren zu können. Git unterbricht den automatischen *merge commit* 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 den Konflikt lösen, indem dieser die hergestellte Datei 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, local* Version erstellt, sowie eine backup Version der Konflikt behafteten Datei, inklusive den Konflikt Marker Zeilen. Für eine schnelle und übersichtliche Lösung des Konflikts wird meistens ein *mergetool* 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 nun erkennen, dass kein Konflikt mehr besteht. Über *git commit* wird nun der gestoppte *merge* zum Beispiel 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 es nun wie folgt aus:



Nachdem beide Konflikte gelöst sind kann der *rebase* fortgeführt werden. Bei manchen *mergetools*, 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.

## Git remote

Die Bezeichnung *remote* steht im Allgemeinen für den Ursprung des Projektarchivs*.* Grundsätzlich fallen aber auch andere Projektarchivedarunter, zu welchen das eigene eine Verbindung aufnehmen kann. 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. Um das lokale Archiv abzugleichen wird er Befehl *fetch* oder *pull* verwendet.

## Git fetch

Mit ***git fetch*** wird *origin* mit dem lokalen Stand abgeglichen, allerdings werden nun Änderungen 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 *origin* upzudaten*.* 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 *fetch* nutzen. Pull ist somit eine Kombination aus fetch und merge.[4?]

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 den 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***

wird 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, woher er die Änderungen bezieht. Dies funktioniert, da wir zuvor durch *checkout* try lokal besitzen.

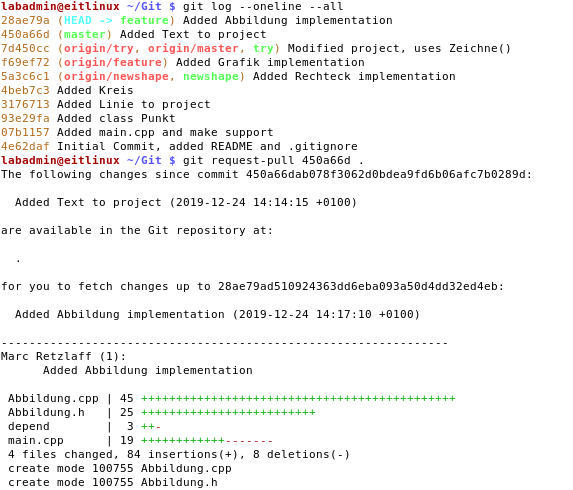


Nachdem weitere Änderungen hinzugefügt wurden auf *master*, und auch auf *feature*, soll diese Änderungen wiederum zusammengeführt werden. Diesmal dient es dazu eine weitere git Funktion zu präsentieren: *request-pull*.

## 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 Kommando werden Differenzen ermittelt und ausgegeben, mit dessen der Ersteller sich nun an eine andere Partei wenden kann. Durch diese Information 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 und Code Integration, Kommentare einfließen lassen, bietet das request-pull Kommando lediglich eine Zusammenfassung von Änderungen. Die ***<url>*** bezieht sich auf das Ziel Archiv. 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 synchron zum lokalen Archiv sein. Daher sollte zuerst 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 (tags) verwendet.

## 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 einen Tag. 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 'Version 1'*** angehangen werden. Um sich die Liste aller vorhanden *tags* 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 den 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.



Im Zuge von weitern Änderungen wird nun eine weitere Art Änderungen zu implementieren verwendet. Es werden nun Änderungen der Arbeitskopie hinzugefügt, anhand einer Datei welche Änderungen beinhaltet. Dies geschieht gemäß ABBILDUNG.



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

## 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 diese 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 repository 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.

## 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 wieder zu erhalten muss ein Commit übergeben werden. In solch einem Fall hilft der *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. [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.



Git besitzt neben den behandelten Kommandos noch weitere, welche des Öfteren von Nutzen sind. Deshalb werden nun die wichtigsten davon erklärt. Als erstes wird der Befehl *show* behandelt.

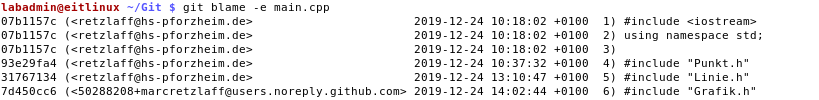
## Git show

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]

Ein weiterer Befehl, um Information über die Geschichte einer Datei zu erhalten bietet *blame.*

## 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 auf das Projektarchivausgeweitet. [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.



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:



## 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 enorm ansteigen, da in jedem Commit dieselbe Datei wieder und wieder gespeichert wird. Auch steigt die Zeit für die Synchronisation an. 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. Zusätzlich 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 referenzierten Dateien im lokalen Container vorhanden sind. Um übergreifend damit arbeiten zu können, müssen alle git Instanzen, welche mit diesem Repository arbeiten LFS beherrschen können. Auch ist es möglich bestehende Repositorys nachträglich mit LFS zu versehen. Hierbei wird, falls gewünscht, auch rückwirkend LFS eingeführt, wobei die Historie umgeschrieben wird. In solch einem Fall sollte jeder beteiligte das Repository neu klonen. LFS lässt sich großzügig konfigurieren über die vorhandene git Konfigurations-Datei. Zusätzlich bietet LFS eigene git Kommandos rund um LFS. 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]