## 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 Verzeichnis. Dies beinhaltet alle Daten (in Form von Dateien) des Projekts, sowie die von Git benötigten Daten zur Konfiguration und Verwaltung, dargestellt als ein Projektverzeichnis in ABBILDUNG. Untereinander müssen diese Verzeichnisse synchronisiert werden, damit mehrere Personen daran arbeiten können. Dies folgt aus dem dezentralen Konzept, welches Git verfolgt. Somit besitzt jede Person, welche an dem Projekt arbeitet, eine komplette Kopie des gesamten Projektverzeichnis. Im Allgemeinen wird ein Referenzarchiv verwendet um die Synchronisierung zu Vereinfachen. Diese Rolle wird in ABBILDUNG dargestellt. 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". Dieser Vorgang ist in ABBILDUNG zu sehen. Besitzt der Commit einen wichtigen Stand kann dieser zusätzlich ein Etikett besitzen, mit welchem der Commit schnell und einfach referenziert werden kann. 

## 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.

# Die ersten Schritte in Git

Um die wesentlichen Befehle von Git zu behandeln und erklären, werden die jeweiligen Kommandos anhand des nachfolgenden 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, wie in KAPITEL erklärt, 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. Dies folgt aus den unterschiedlichen Oberflächen und Anwendungen, welche die grafischen Oberflächen darbieten. Durch die Erlernung der elementaren Grundlagen kann ein Verständnis aufgebaut werden, welche die Zurechtfindung in Git Anwendungsprogrammen erleichtert. Falls die Funktion des Kommandos oder dessen Handhabung ungewiss ist, kann über ***git <Kommando> --help*** die Git interne Dokumentation zu eben diesem Kommando aufgerufen werden.

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

## Erstellen eines Projektarchives

### Git Init

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

Die Dateien, welche über die Versionsverwaltung verwaltet werden, befinden sich komprimiert und verschlüsselt in dem Ordner *objects*. In KAPITEL wird hierauf zurückgekommen und erklärt wie Git die Dateien effizient speichert. Der Ordenr *refs* enthält Verweise auf die verschiedenen Zweige (*branches*) und Etiketten (tags), welche das Projekt besitzt. Ein weiters Verzeichnis *logs* enthält wie der Name verspricht Information zu der Commit Historie, sowie der Historie der einzelnen Zweige und des Arbeitsverzeichnis. Shell-Skripte, die bei Git Kommandos ausgeführt werden sollen, befinden sich in dem Ordner hooks. Hier kann zum Beispiel eine Commit Nachricht vordefiniert werden. Die *Config* Datei enthält Information und Einstellungen für das Projektarchiv. Hier wird vermerkt, welcher Autor an diesem Projektarchiv arbeitet, welcher Texteditor standardmäßig geöffnet wird, sowie Aliase und Links zu weiteren Projektverzeichnisse. Erstellt man ein Referenzarchiv (*shared repository),* welches als reines Archiv zur 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 nun an diesem Projekt zu arbeiten, muss dieses zuerst mit *git clone* geklont werden. Dieser Klon besitzt nun ein Arbeitsverzeichnis. Dadurch können nun Dateien editiert werden. [11]

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

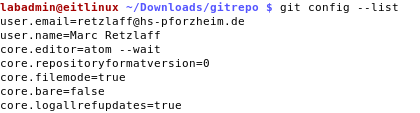


Falls keine archivübergreifende Konfiguration existiert, das heißt keine Benutzerweite Git Informationen hinterlegt sind, muss man diese Git zuerst mitteilen. Diese Informationen werden dazu benötigt, damit Git weiß, welcher Entwickler die Änderungen lokal durchführt, sodass jeder erstellte Commit einem Entwickler zugewiesen werden kann. Ohne diese Informationen verweigert Git einen Commit zu erstellen. Um die benötigten Informationen Git mitzuteilen wird ein weiterer Befehl mit dem Namen *config* ausgeführt.

### 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 am Betriebssystem angemeldeten Benutzers. Hierfür werden mit ***git config --global user.name "<Name>"***,beziehungsweise ***user.email "<E-Mail>"*** die zwei wichtigsten Informationen gesetzt. Weitergehend kann der Standard Texteditor geändert werden mit: *core.editor*, oder man ändert die Einstellungen in der respektiven Datei, aufzurufen mit ***git config --global --edit***. Des Weiteren können Aliase erstellt werden, um individualisierte Kommandos zu erstellen oder bestehende abzukürzen. Dies geschieht grundsätzlich mit ***git config alias.<Alias Name> <Git Kommando>.*** Um die bisherige Konfiguration einzusehen, wird ***git config --list*** verwendet.[18]

Da beide Einstellungen bereits aufgrund vorherigen Projekten 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 dieses bestehende Projekt geklont werden, um dann lokal daran arbeiten zu können. Hierfür steht das Kommando *clone* zur Verfügung. Dies ist die zweite Methode ein Projektarchiv anzulegen.

### Git clone

Mit diesem Kommando wird ein neues Projektarchiv angelegt, jedoch als Vorlage ein bestehendes Archiv verwendet. Man klont somit das Projektarchiv. Zusätzlich wird dem lokal erstellen Archiv direkt eine Verbindung zum ursprünglichen Projektarchiv beigefügt und unter dem Alias *origin* hinterlegt. So können ohne Umwege Änderungen am ursprünglichen Archiv verfolgt und in das Eigene Archiv übernommen werden. Verwendet wird das Kommando mit ***git clone <remote Adresse> <lokales Verzeichnis>***. Als einfaches Beispiel wird in ABBILDUNG ein öffentliches Projektarchivdes Hosters GitHub.com geklont. Im Falle eines sogenannten privaten Archivskann *https://<Benutzer>:<Passwort>@github.com/...* verwendet werden. Bedeutet, das Projekt ist durch eine Zugriffskontrolle geschützt. Übergibt man die benötigten Zugangsdaten nicht direkt, wird der Benutzername und das Passwort nachträglich verlangt, um auf das Projektarchiv zuzugreifen. Neben "https://" kann die Kommunikation zwischen den Archiven auch über "ssh://" oder "git://" ablaufen. Dafür müssen jedoch entsprechende Konfiguration, wie zum Beispiel der Austausch der SSH Schlüssel, zuvor erfolgen, um eine autorisierte Verbindung herzustellen.

## Änderungen dokumentieren

Mit einem frisch angelegten Projektarchiv und elementaren Einstellung kann nun mit der Arbeit begonnen werden. 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. Diese vereinfacht die spätere Handhabung mit Dateien des Projekts.



### Gitignore

Benutzt man Dateien, welche explizit nicht in die Versionsverwaltung inkludiert werden sollen, kann eine Datei namens: ***.gitignore*** verwendet werden. Dateien und Verzeichnisse, welche in *.gitignore* zeilenweise vorkommen, werden automatisch von Gitausgeblendet 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 Dateien jeder Entwickler lokal erstellen kann. Dadurch wird die Versionsverwaltung nicht unnötig vergrößert und erhält zudem mehr Übersichtlichkeit. Des Weiteren können Dateien, welche zum Beispiel durch die personalisierte Entwicklungsumgebung des Entwicklers hinzukommen, ausgeschlossen werden.

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

### Index (Stageing Area)

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



Ist das Verzeichnis vor der Initialisierung von Git nicht leer, würden nun alle Dateien, welche das Verzeichnis besitzt, aufgelistet werden. In unserem Fall werden die beiden zuvor erstellten Dateien unter "Unversionierte Dateien" aufgeführt. Dass bedeutet, diese Dateien sind noch nicht in der Versionsverwaltung vorhanden, beziehungsweise Git besitzt keinen älteren Stand dieser Dateien. Die beiden Dateien oder Änderungen an Dateien, falls diese bestehen, können nun dem Index hinzugefügt werden. Dazu wird der Befehl ***git add*** 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 Projektarchiv oder im speziellen Commit benötigt werden. Man stelle sich eine größere Änderung vor, welche mehrere kleinere Module, unabhängig voneinander, beinhaltet. Anstatt die Änderungen als Gesamtpaket in einen Commit zu verpacken, kann man die Module einzeln dem Index hinzufügen und jeweils ein Commit erstellen. Zu sehen in ABBILDUNG, werden nun README und .gitignore 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 eine Datei weiter aufgesplittet werden, um nur einzelne Zeilen der ausgewählten Datei dem Index hinzuzufügen [1, S.188ff.]. Um dies nicht für alle veränderten Dateien durchzuführen, sondern für eine einzelne Datei, wird ***git add -p <Datei>*** verwendet. Soll eine zuvor versionsverwaltete Datei aus der Versionsverwaltung entfernt werden, wird sie dem *remove* Kommando übergeben. Der Aufruf lautet hierbei ***git rm <datei>.*** Für dauerhafte oder wiederkehrende nicht versionsverwaltete Dateien in der Arbeitskopie, ist es hilfreich eine ***.gitignore*** Datei wie zuvor in Kapitel XX beschrieben anzulegen, beziehungsweise die Datei der .gitignore hinzuzufügen. Ist eine Datei dem Index fälschlicherweise hinzugefügt worden, kann diese Datei aus dem Index über ***git reset <Datei>*** wieder entfernt werden, ohne dabei Änderungen an der Datei zu verlieren. Soll die gesamte Datei zurückgesetzt werden, kann dem Befehl das Schlüsselwort ***--hard*** angehangen werden.

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

### 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 der Arbeitskopie 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 eine schnelle Übersicht und vereinfachte Zusammenarbeit mit anderen Personen an dem Projekt, sollte man sich daher ein Schema für die Commit Nachricht überlegen, welche Projektweit gilt und angewendet wird. Ein Commit verweist auch immer auf seinen Vorfahren*.* Anhand diesem kann man feststellen, woher der Commit abstammt, beziehungsweise, welche Änderungen im Vergleich zu seinem Vorfahreneinflossen. Durch diese Verknüpfung kann man nun die Commits aneinandereihen und erhält damit die grafische Darstellung einer Baumstruktur:

Jeder neue Commit wird an dessen vorhergehenden angehangen und erweitert die Aneinandereihung. Wird ein neuer Zweig erstellt und beide zweige besitzen einen Commit, zeigen beide auf den gleichen Vorgänger, es entsteht eine Abzweigung. Im Gegensatz dazu besitzt ein Commit einer Zusammenführung mehrere Vorgänger*.* Durch diese Aneinandereihung an Commits bildet Git eine lückenlose Historie bis zum Ursprungstands des Projektarchivs. 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 der Arbeitskopie noch unverändert ist. Hat man zusätzlich vergessen eine Änderung dem Commit hinzuzufügen, kann man dies nun dem Korrektur Commit hinzufügen und über ***-m "<Commit Nachricht>"*** die Nachricht korrigieren. Allerdings überschreibt der neue Commit den zuvor erstellenten Commit wodurch alle Referenzen auf diesen verloren gehen. Daher sollte man *--amend* nur nutzen, sofern keine andere Partei auf diesen aufgebaut haben kann. Die Faustregel lautet: Kein --*amend* auf veröffentlichte Commits. Veröffentlicht heißt hierbei über das lokale Projektarchiv hinaus.[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 die zwei Dateien neu hinzugefügt wurden und insgesamt durch diese Dateien 4 Zeilen hinzukamen. Die Zeileninformation kann nur Dateien entnommen werden, welche diese Informationen Git bereitstellen. Binär Dateien sind somit davon ausgenommen.

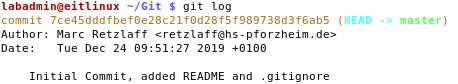


Zur Überprüfung kann wird nachfolgend die Historie des Projektarchivs eingesehen. Hier sollte der Commit zu sehen sein. Dafür verwendet man das Kommando *log.*

### Git log

Um einen Überblick über den aktuellen Stand des Zweiges zu erhalten, kann eine Übersicht der letzten Commits mit ***git log*** aufgerufen werden. In der Ausgabe sieht man alle Commits des Zweiges mit Information zum Autor, Datum und Hash des jeweiligen Commits. Mit dem Schlüsselwort ***--all*** werden die Commits aller Zweige des Projektarchivs in zeitlicher Reihenfolge angezeigt. In Kurzform kann auch ***git shortlog*** benutzt werden. Dadurch wird nur die Commit Nachricht, gruppiert nach dem Autor, angezeigt. Um zusätzlich in der Ausgabe den Anfang des jeweiligen Hash zu erhalten kann ***git log --oneline*** benutzt werden. Ein weiteres nützliches Werkzeug bietet der Anhang ***--follow <Datei>***. 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 innerhalb der Änderungen der jeweiligen Commits zu suchen, wird ***-S<Suche>*** verwendet. Mit ***-<Zahl>*** kann die Ausgabe auf eine bestimmte Anzahl an Commits reduziert werden.Zur Orientierung der jeweiligen Commits kann zusätzlich ***--graph*** übergeben werden. Dadurch entsteht an den ersten Stellen ein Baum, welcher das Repository wiederspiegelt. Dies hilft dabei die Commits schneller den Zweigen zuzuordnen und die Reihenfolge der Commits einordnen zu können. Um sehen zu können, ob Etiketten oder Zweige dem jeweiligen Commit zugeordnet sind, wird ***--decorate*** verwendet. [1, S.26ff.]

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



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



Es fällt nun dem Entwickler auf, dass die 3 nachfolgenden Commits nur zusammen Sinn ergeben. In diesem Fall ist durch die Aufteilung auf mehrere Commits eine Ausführung des Programms erst ab dem letzten Commit möglich. Grundsätzlich sollte man sich vornehmen, sofern möglich, bei jedem Commit einen ausführbaren Stand zu besitzen. Daher würde es auch sinnvoll sein, die drei gefolgten Commits zu einem Commit zu verschmelzen. Auch dies ist möglich in Git mit einer 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 der Nutzer jeden einzelnen Commit, welcher durch den Befehl betroffen ist individuell verändern zu verändern. So kann der Entwickler die Reihenfolge der Commits verändern, einzelne Commits löschen, Commits verschmelzen oder bearbeiten. Dies geschieht in dem man die erste Spalte des automatisch geöffneten Textdokuments dementsprechend anpasst. Hierfür werden wiederum spezielle Schlüsselwörter benötigt. Um einen Commit zu löschen, entfernt wird dessen Zeile entfernt. Dies wird häufig verwendet, um mehrere Commits auf einmal zu verwerfen. Die Reihenfolge in der die Commits dargestellt werden, entspricht der Reihenfolge, mit der *rebase* die Commits auf den Vorfahren, das heißt, den ersten nicht betroffenen Commit, anwendet. Standardmäßig ist dies genau umgekehrt der Reihenfolge, in der die Commits in der Historie dargestellt werden. Da das kommando *rebase* zuerst den zeitlich ersten Commit anwenden muss, um die ursprüngliche Reihenfolge wiederherzustellen. Das Schlüsselwort *reword* ermöglicht es die Commit Nachricht des entsprechenden Commits zu verändern. Mit *squash* verschmelzt sich der Commit mit dem vorhergehenden Commit. Dies wird häufig dazu benutzt, um die Baumstruktur übersichtlich zu halten, somit größere Funktionen oder Module zu gruppieren oder für Korrektur Commits, wobei diese Commits mit dem ursprünglichen Commit verschmolzen werden. [9] In dem Fall, dass ganze Entwicklungszweige zu einem Commit verschmelzen, kann der Anhang ***--root*** von Nutzen sein. Dadurch entfällt die Suche der Prüfsumme oder die Anzahl der Commits, welche von dem interaktiven Rebase betroffen sein sollen. Anstatt "HEAD~3", oder die Prüfsumme des Commits vor dem Rebase, in diesem Fall "7ce45dd", kann dementsprechend *--root* übergeben werden.

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

**git rebase -–interactive --root**

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



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 von "Added main.cpp" erweitert. Ist die Datei wiederum gespeichert und wird geschlossen führt Git den *Rebase* nun der Konfiguration entsprechend aus. Die Kommandozeile sieht nun wie folgt aus:



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

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

Es wird wieder weiter an dem Beispielprojekt entwickelt. Der Entwickler arbeitet an einer Erweiterung des Programms, will allerdings einen neuen Ansatz zwischenzeitlich verfolgen. Die bisherig durchgeführten Änderungen sind allerdings nicht auf einem Stand, welche einen sinnvollen Commit ergeben. Daher werden die Änderungen Zwischengespeichert, sodass der Entwickler eine frische Arbeitskopie besitzt um damit den neuen Ansatz, isoliert von den bisherigen Änderungen, bearbeiten kann. Diese leere Arbeitskopie ist wichtig, da ein Zweigwechsel mit dieser Bedingung verknüpft ist. Der Entwickler kann allerdings entscheiden, auf der frischen Arbeitskopie weiterzuarbeiten, oder zuerst einen neuen Zweig für den weiteren Ansatz zu eröffnen und auf diesem Zweig den Ansatz zu verfolgen. Später kann an den zwischengespeicherten Änderungen wieder weitergearbeitet werden. Die Funktionalität von temporärer Zwischenspeicherung bietet in 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 der selbst festgelegten Commit Richtlinie entsprechen, kann man die Arbeitskopie Änderungen im Stash bis zur Rückkehr zum Zweig ablegen. Auch können darin Ideen oder Ansätze gespeichert werden, um sie nicht zu verlieren. Da ein erstellter Zwischenstand nicht Zweiggebunden ist, somit auf jeden Zweig angewendet werden kann, wäre eine weitere mögliche Anwendung darin Änderungen abzulegen, welche das Projekt schnell und temporär verändern. Zum Beispiel eine spezielles Debugging Funktion. Der einfachste Grund ist allerdings die vorhandene Arbeitskopie zu reinigen, vorheriges aber nicht zwangsweise verlieren zu wollen, falls man noch einmal darauf zurückkommen sollte. Der Befehl ***git stash*** speichert alle Veränderungen gegenüber dem Stand des letzten Commits des ausgewählten Zweiges. Mit der Option ***-patch*** kann wiederum nur ein Teil der Änderungen zwischengespeichert werden. Mit *-u* werden nicht versionsverwaltete Dateien in der Arbeitskopie mit einbezogen. Um eine Übersicht über die Zwischenspeicherungen zu erhalten, listet der Befehl ***git stash list*** alle bereits gespeicherten Zwischenstände auf. Um eine Einordnung der einzelnen Zwischenspeicherungen möglich zu machen, kann dem jeweiligen Zwischenstand eine Nachricht hinzugefügt werden. Dafür wird ***save <Nachricht>***verwendet. [2] Um die Zwischenspeicherung wieder auf die aktuelle Arbeitsmappe anzuwenden wird der Befehl ***git stash pop*** oder ***git stash apply*** verwendet. Im ersten Fall wird der Zwischenstand nach der Anwendung gelöscht, im Gegensatz zu der Nutzung von *apply*, welche den Zwischenstand beibehält. Mit ***git stash show*** werden die Änderungen des ersten Eintrags im Stash in Kurzform angezeigt. Mit dem Anhang ***-p*** werden die Änderungen in Langform, das heißt zeilenweise Änderungen, falls möglich, wiedergegeben. Um einen Zwischenstand zu löschen kann der Befehl ***git stash drop <stash-id>*** verwendet werden, wobei die ID zum Beispiel "*stash@{2}" ist.* [1, S.191ff.]

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



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

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

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

### Git branch

Zweigewerden angewendet, um eine isolierte Entwicklungsumgebungen zu erhalten. So ist es von Vorteil eine Spielwiese des aktuellen Arbeitsstands zu verwenden, um Entwicklungen jeglicher Art vorzunehmen, ohne dabei in Gefahr zu geraten, instabilen Code im Hauptzweig zu erhalten. Auch kann gerade bei Projekten mit mehreren Entwicklern, unter Verwendung von Zweigen sichergestellt werden, dass keine Komplikationen durch gegenseitige Abhängigkeiten entstehen. Zudem wird eine Synchronisierung zwischen den Änderungen mehrerer Personen auf ein Minimum reduziert, da ein Zweig bis zu seiner Zusammenführung nicht mit zeitgleichen Änderungen durch andere Personen interagieren muss. Aufgrund dieser Isolation lassen sich Projekte mit mehreren Entwicklern ohne größere Abstimmungen und Nachverfolgungen untereinander durchführen. Intern werden Zweigevon Git nur als weitere Zeiger, welche auf erstellte Commits zeigen behandelt. Um einen neuen Zweigzu erstellen wird der Befehl ***git branch <Name>*** verwendet. Der neue Zweigführt nun von dem aktuellen aktiven Zweigab. Git erkennt dies, da beide nachfolgenden Commits den gleichen Vorfahren besitzen. Wird ein Zweignicht mehr benötigt lässt dieser 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 den Zweig trotzdem löschen, wird ein kapitalisiertes **-B** benötigt. Um alle Zweigedes Projektarchivs anzusehen wird ***git branch -l*** verwendet. 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.* Um dies zu gewährleisten gibt es mehre Möglichkeiten. Am einfachsten kann ***git push origin <Zweig>*** verwendet werden. Der Alias *origin* steht hierbei für das Referenzarchiv und kann unter Umständen einen anderen Namen besitzen. Um sicherzustellen, dass der lokale Zweig auch unter dem gleichen Namen im Referenzarchiv erstellt wird oder explizit unter einem anderen Namen, kann anstatt *<Zweig>* auch *<Zweig>:<Zweigname der Referenz>* benutzt werden. Ist man in dem Zweig*,* welcher hinzugefügt wird, kann auch *HEAD* stattdessenübergeben werden, da HEAD zu diesem Zeitpunkt dem Zweig gleicht. Um einen Zweig wiederrum aus dem Referenzarchiv zu löschen, wird ***git push --delete <Zweig>*** verwendet. Von Zeit zu Zeit ist es sinnvoll nachzuvollziehen zu können ob ein bestimmter Commit schon einem Zweig zugeführt wurde. Dafür kann ***git branch --contains <Prüfsumme Commit>*** verwendet werden. Damit werden alle Zweige aufgelistet, die diesen Commit beinhalten. Dies funktioniert jedoch nur, falls der gesuchte Commit über eine Methode hinzugefügt wurde, welche die Metadaten des Commits beibehält. Ist der Commit zum Beispiel über einen *chery-pick* oder *patch* zugeführt wurden, erkennt dies Git nicht als gleichen Commit an. Hierfür hilft das Kommando ***cherry***, beschrieben in KAPITEL. [12]

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



Um nun auf den neu erstellten 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, zwischengespeichert oder revidiert werden. Ist dies nicht der Fall schlägt das Kommando fehl. Intern wechselt *HEAD*, zuständig für das Laden der Arbeitskopie, auf den neusten Commits des gewünschten Zweiges*.* Allerdings können nicht nur Zweige übergeben werden, auch einzelne Commits und Dateien können mit *checkout* verwendet und dadurch geladen werden. Bei einer Datei wird damit die letzte Version dieser Datei der Arbeitskopie hinzugefügt, beziehungsweise die vorhandene Datei mit der übergebenen Datei überschrieben. Dies ist nützlich falls man Änderungen an einer einzelnen Datei verwerfen will. [13] Wird ein Commit, beziehungsweise dessen Prüfsumme, übergeben, entsteht dadurch ein sogenannter "*detached HEAD"*. Das heißt, der *HEAD* Zeiger zeigt nun nicht auf den neusten Commit eines Zweiges*.* Ist HEAD nun nicht an den neusten Commit eines Zweigesgekoppelt, fehlt für Git die Zuordnung an einen Zweig*.* Arbeitet man nun an dem Commit weiter, das heißt, man erstellt einen neuen Commit, dann wird eine neue Abzweigung erstellt, da dem Commit bereits ein Commit folgt. Diese Abzweigung besitzt jedoch keinen Namen, den man referenzieren könnte, wie bei einem Zweig*.* Daher muss der Abzweigung eine Referenz hinzugefügt werden, um wieder darauf zurückgreifen zu können, sobald man von der Abzweigung wechselt. Dies kann geschehen, indem man ein Etikettsetzt oder nachträglich einen Zweig kreiert mit ***git branch <name>***. [14]

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



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



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

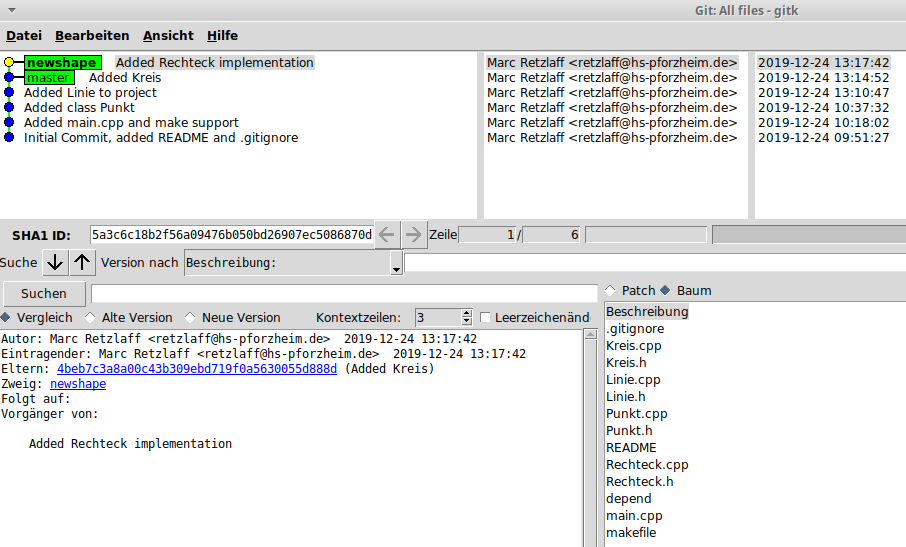


Um einen Überblick über unser Projektarchiv und dessen zweige zu erhalten, kann man neben der Kommandozeile auch die Git integrierte grafische Oberfläche verwenden.

### 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 auf das gesamte Archiv erweitert. Zu sehen ist hier auch der symbolische Baum, welcher durch die Commits gebildet wird. *Gitk* bildet nicht nur eine Übersicht, es lässt sich darüber auch mit den Commits arbeiten und Aktionen ausführen. *Gitk* wird außerdem auch durch aufgerufen *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 anhand dieser Darstellung ablesen, dass der neuste Commit des Zweiges *newshape* auf dem neusten Commit von *master* basiert. Das heißt, um die Änderungen von *newshape* zu übernehmen, muss lediglich der Zeiger von *master* inkrementiert werden, da der *master* Commit bereits ein Vorfahre ist und der *newshape* Commit lediglich auf diesem aufbaut. Dieses Prinzip der Zusammenführung nennt sich *fast-forward.* Da dabei keine Dateien zusammengeführt werden, besitzen diese Zusammenführungen kein Konfliktpotenzial. Durchgeführt wird die Zusammenführung der beiden Zweige über den Befehl *merge.*

## 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 jeweiligen Arbeitskopie Datei für Datei verglichen. Im Gegensatz zu einem Zwei-Wege-Merge, beziehungsweise einem einfachen zeilenweisen Abgleich, wird durch die Änderungsverfolgung von Git auch der frühste gemeinsame Vorgänger beider Dateien einbezogen. Dadurch erkennt das Programm zur Zusammenführung (*merge tool)* neben einer Abweichung der zusammengehörigen Dateien, auch welche Datei, der die zu übernehmende Änderung besitzt. Dies wird durch den Einbezug des gemeinsamen Vorfahren beider Dateien ermöglicht. In ABBIDLUNG sind einige beispielhafte Zeilen zu sehen, welche nun zusammengeführt werden sollen. Zur Veranschaulichung welches Schaubild, welchen Stand der Datei im Projektarchiv darstellt, hilft die farbliche Markierung der Commits. "Yours" ist hierbei die Datei aus dem Zweig, welcher mit "Mine" zusammengeführt werden soll. "Base" stellt den gemeinsamen Vorfahren da. In Zeile 30, der ersten Zeile der Darstellung, enthält die Datei "Yours" die Abweichung zu "Base" und "Mine". Dies ist für Git die zu übernehmende Änderungen. Dabei ist der generelle Git Ansatz, mit welchem Änderungen zusammengeführt werden sollen, die Änderung zu "Base" zu übernehmen. Führt dieser Ansatz zu nicht gewollten Änderungen, muss die Zusammenführung händisch erfolgen. Der Vorteil der Git Methodik ist, dass Konflikte nur entstehen, falls beide Stände Abweichung zum Vorfahren besitzen. Dies ist zu sehen im rot markierten Bereich in Zeile 50. Dieser Konflikt muss per Hand gelöst werden. Sogenannte "merge tools" helfen dabei, schnell diese zu Konflikt führenden Änderungen Blockweise zu übernehmen oder ersetzen. Im letzten Beispiel ist eine Änderung nur in einer der beiden Dateien vorhanden, diese wird in die Zusammenführung übernommen, da dies eine Änderung zum Vorfahren ist.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

[7]

Des Weiteren wird nachdem eine Zusammenführung vollzogen wurde, ein sogenannter *merge-link* erstellt. Dadurch wird sichergestellt, dass bei einer späteren Zusammenführung, keine bereits gelösten Konflikte erneut durchgeführt werden müssen, da der Vorfahre durch den *link* erneuert wird. Sollte in ABBILDUNG eine weitere Zusammenführung zu späterer Zeit stattfinden, würde der gemeinsame Vorfahre abhängig des Zielzweigs "Mine" oder "Yours" sein. Um die Theorie nun in die Praxis umzusetzen, wird zuerst der Zielzweig als aktuelle Arbeitskopie gesetzt. Dann kann die Zusammenführung über ***git merge <Quellzweig>*** vollzogen werden. Besitzt der Zielzweig keine abweichenden Commits, sodass der Quellzweig nur Änderungen über den Zielzweig hinaus besitzt, findet eine *fast-forward* 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. Diese Methode sollte man bei wichtigen Implementierungen zu Hauptzweigen verwenden, um die zusammengeführten Dateien manuell zu überprüfen oder die Laufähigkeit der Arbeitskopie zu sichern. Danach kann der Commit der Zusammenführung manuell vollendet werden. Die Nachricht für den *merge-commit* wird mit ***-m*** wie gehabt, übergeben. Neben der von Git bevorzugten Methodik kann der Nutzer anhand eines Schlüsselworts eine andere Strategie für die Zusammenführung wählen. Die zwei Wichtigsten sind **-*ours*** und ***-theirs***. Damit kann bei Konflikten generell eine Version bevorzugt werden. Zu beachten gilt dabei, dass das Zusammenspiel der Änderungen nicht durch eine manuelle Konfliktlösung überprüft wird. Die Laufähigkeit sollte überprüft werden. Eine weitere nutzvolle Option bietet ***--squash***. Hierdurch werden alle Änderungen, welche in die Zusammenführung einfließen zu einem Commit zusammengefasst und als neuer, isolierter Commit dem Zielzweig hinzugefügt. Es wird allerdings keine Verbindung zum Ursprungs Zweig hergestellt. Git sieht den Ursprungszweig demnach als nicht zusammengeführt an. Sollte man sich während der Zusammenführung umentscheiden oder es kommen unbekannte Komplikationen auf, kann der Vorgang verworfen werden über ***git reset --merge***. [7]

Führen wir die Änderungen des Beispielprojekts zusammen, signalisiert Git, dass es sich dabei um eine *fast-forward* Zusammenführung hat, wie durch gitk bereits dargestellt:



Die Historie des Projekts sieht nach der Zusammenführung wie folgt aus:



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

### Git Rebase

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

BESCHRIFTEN



Um die zwei Zweige zu vereinen, fehlt noch ein weiterer Schritt, welcher umgekehrt ***git rebase dev*** auf dem *maindev* Zweig auslöst. Folgend werden nun alle Commits von *dev* auf *maindev* angewendet. Da *dev* durch den ersten Rebase aus dem neusten Commit herausgelöst wurde, ist die Integration folgend nur noch ein *fast-forward-merge*. Der Zeiger von *maindev* wird somit auf den Zeiger von *dev* inkrementiert. Alternativ dazu könnte man auch einen einfache Zusammenführung über das Kommando ***merge*** erwirken oder ***git fetch dev:maindev*** verwenden um dementsprechend ein *fast-forward merge* zu erwirken. 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 stark verschlankt, da ein Großteil der Zweige nach Beendigung über das Kommando Rebaseeinem Hauptzweig zugeführt werden, sind die Zweige zwar übersichtlich, allerdings lassen sich einzelne Funktionalitäten nicht mehr unterscheiden. Auch werden die Zeitpunkte der Commits verändert, sowie deren Metadaten. Daher sollte man sich vergewissern, dass niemand auf die ursprünglichen Commits explizit angewiesen ist, welche man über das Kommando verschiebt oder umschreibt. Deshalb wird empfohlen, nur in persönlichen Zweigen den Befehl zu verwenden, um Konflikte zu vermeiden. Diese können entstehen, sofern Verweise auf veraltete Commits bestehen, da diese nach einem Rebase nicht mehr bestehen. Bei der Anwendung von Rebase sollte nicht vergessen werden, dass eine nachvollziehbare Historie ein großer Bestandteil einer Versionsverwaltung ist und somit der Einsatz des Kommandos durchdacht gewählt sein sollte, damit das Projektarchiv klare Strukturen besitzt und übersichtliche Abgrenzungen einzelner Themen nicht verliert. Ein weiteres Problem bieten bereits gelöste Konflikte, die durch die erneute Anwendung der Änderungen auf die neusten Commits des gewünschten Zweigs wiederaufkommen können. [9] Zudem bietet das Rebase Kommando auch eine interaktive Ansicht, welche zuvor behandelt wurde KAPITEL.

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



Die automatische Zusammenführung durch Git konnte zwei Dateien nicht automatisch zusammenführen und es entstanden zwei Konflikte. Daraufhin wurde das Kommando Rebase unterbrochen. Deshalb müssen beide Konflikte gelöst werden, bevor der Befehl durch den Entwickler 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 der gleichen Datei mit der Version des gemeinsamen Vorfahren unterscheiden. In diesem Fall wird *merge* sowie *rebase* unterbrochen und der Konflikt benötigt eine manuelle Korrektur. Mit *git status* werden die Dateien angezeigt, welche einen Konflikt hervorrufen. Dazu wird oftmals ein *merge tool* zur Hilfe angewendet, da durch eine grafische Lösung meist deutlich übersichtlicher und schneller editiert werden kann. In der Konfliktbehafteten Datei werden beide Versionen 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 zu lösen müssen die *merge-marker*, sowie die ungewünschte Version entfernt werden. Danach wird die korrigierte Datei über ***git add <Datei>*** wieder hinzufügt, sodass der Konflikt für Git als gelöst gilt. Gefolgt von ***git commit*** wird nun der *merge commit* vollzogen und das ursprüngliche Kommando vollendet. Falls man nicht auf diese Art der Konfliktlösung zurückgreifen will, kann der Zusammenführungsversuch auch über ***git merge --abort*** abgebrochen werden. Ein Sonderfall stellt eine Datei dar, welche nur in einem der beiden Zweige existiert. Man entscheidet sich dann über *git add* oder *git rm* die Datei zu behalten oder zu entfernen.

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

### Git mergetool

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

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



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 in blau Abweichungen dargestellt werden.



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



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



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



Nun wird das Zusammenspiel mit anderen Entwicklern beleuchtet. Bisher wurde das Projektarchiv nur lokal vorhanden. Als Beispiel für ein globales Referenzarchiv wird dieses auf dem Hoster Github.com erstellt. Um nun eine Verbindung des lokalen Archivs mit dem Referenzarchiv herzustellen wird der Befehl *remote* verwendet. Die Synchronisierung der Projektarchive ermöglicht die gemeinsame Entwicklung eines Projekt und ist elementarer Bestandteil moderner Software Entwicklung.

## 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 *remote* Eintrag vorhanden, da für Git nur das lokale Archiv existiert. Um die aktuellen *remote* Aliase einzusehen kann ***git remote -v*** verwendet werden. Erstellt man ein Projekt auf einem Hoster, welches man später lokal klont, wird automatisch ein Eintrag erstellt, welcher nun auf das im Hoster hinterlegte Archiv zeigt. Der Alias Name ist standardmäßig *origin.* Über ***git remote add <alias> <url>*** wird ein neuer *remote* Eintrag erstellt. Um den Alias nur für ein Synchronisation Kommando *fetch* oder *pull* zuverwenden, kann man die jeweiligen URL zum Referenzarchiv auch über ***git remote set-url <alias> <url>*** für *fetch* und mit dem Anhang ***[…] set-url --push […]*** für das Kommando *push* erstellt werden. Dies wird zum Beispiel verwendet, um Änderungen von einem Hauptarchiv zu erhalten aber nicht jeder Entwickler die erstellten Änderungen diesem Archiv hinzufügt. Bei größeren Projekten können Untermodule ein eigenes Projektarchiv besitzen.[16]

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



Um den lokalen Fortschritt mit dem Referenzarchiv zu synchronisieren, wird 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. Das globale Referenzarchiv wird meist *origin* genannt. Um zum Beispiel Änderungen auf dem Zweig *master* auf dem globalen Archiv zu veröffentlichen, kann der Befehl ***git push origin master*** benutzt werden. Mit dem Zusatz ***-f*** wird die Historie und die Änderungen des lokalen Archivs übernommen, auch wenn das Referenzarchiv neuere Commits enthält. Diese werden dementsprechend verworfen. Daher sollte 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 dem Kommando übergeben werden. Da es im Allgemeinen nicht gewollt ist, dass jeder direkten Zugriff auf ein Archiv eines Hostersbesitzt, wird im Normalfall eine Anmeldung verlangt. Dies wird umgangen, indem man die Anmeldung bereits im *remote* Link enthält. Dadurch stehen die Anmeldedaten allerdings einsehbar unter den *remotes.* Daher kann zum Beispiel eine Verifikation über das Verbindungsprotokoll SSH erfolgen. Dafür muss das SSH Schlüsselpaar einmalig angelegt werden und dem Hoster der Öffentliche Schlüssel mitgeteilt werden. Um diesen Verbindungstyp auch dementsprechend zu nutzen, muss der *remote* link zusätzlich auf SSH gewechselt werden. Ist keine automatische Anmeldung festgelegt, fragt Git nach diesen bei jedem Zugriff auf das Referenzarchiv. [1, S.61ff.]

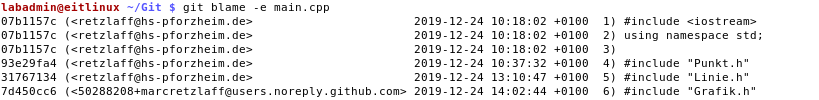
Durch die hinzugefügte Verbindung zu Github.com, wird das bisher leere Referenzarchiv mit dem lokalen Archiv überschrieben. Da es sich um ein privates Projekt handelt, erfordert eine Interaktion mit dem Archiv eine Anmeldung, welche nach Eingabe des Befehls erfolgt. Zuletzt sieht man, dass alle Zweige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



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

### Git blame

Um herauszufinden, welcher Autor welche Änderungen an einer Datei durchgeführt hat, wird das Kommando ***git blame <Datei>*** verwendet. Dadurch wird nun für jede Zeile der Datei der Autor dieser angegeben. Um die Ausgabe weiter zu verfeinern kann über das Keyword *-L <start>, <end>* die Ausgabe auf bestimmte Zeilen verkleinert werden. Mit *-M* wird der Ersteller von Zeilen, welche sich nicht verändert haben, sondern nur kopiert beziehungsweise im Archiv verschoben wurden, berücksichtigt. Das heißt, der Autor dieser Zeilen ist nun der Ersteller, und nicht der Autor des Commits, welcher die Zeile verschob oder kopierte. Mit *-C* wird das Herausfinden des Erstellers einer einzelnen Zeile auf das Projektarchivausgeweitet, beziehungsweise Dateiübergreifend. [26] In ABBIDLUNG sieht man eine beispielhafte Ausgabe des Kommandos. Diese ist in fünf Spalten einzuteilen. Zuerst steht der Anfang der Prüfsumme des Commits, welcher diese Änderung mit sich brachte. Diese ist gefolgt von dem Autor, das Datum und der Zeile der Datei. Die letzte Spalte zeigt den Inhalt der Datei.



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

### Git fetch

Mit ***git fetch*** wird das Referenzarchiv unter dem alias *origin* mit dem lokalen Stand abgeglichen. Allerdings werden abweichende Änderungen dem Nutzer lediglich angezeigt und nicht automatisch versucht diese auch zu integrieren. Aufgrund dessen wird oftmals der Befehl *fetch* mit anschließendem *merge* bevorzugt, da man zwischen den Kommandos sich nochmals versichern kann, dass nur gewollten Änderungen aus dem Referenzarchiv einfließen. Beide Befehle *fetch* und *pull* haben zum Großteil die gleiche Syntax um einzelne Zweige oder spezielle *remote* Server als Referenz zu verwenden. [1, S61ff.]

### Git pull

Der Befehl ***git pull*** wird benötigt um das lokale Projektarchiv mit (falls vorhandenen) Änderungen des Referenzarchivzu erweitern*.* Erweitert mit ***--all*** werde alle Zweige abgeglichen. Mit ***git pull origin master*** wird nur der Zweig *master* abgeglichen und falls Änderungen vorhanden sind mit dem lokalen Stand zusammengeführt. Da es hierbei zu Konflikten kommen kann, sollte man nur den *pull* Befehl ausführen, wenn 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 das Kommando *fetch* nutzen. Pull ist somit eine Kombination aus fetch und merge.[1, S.61ff.] Ein weiteres Schlüsselwort, welcher dem Befehl angehangen werden kann, ist ***--rebase.*** Dies führt bei lokalen Abweichungen gegenüber dem Referenzarchiv dazu, dass die lokalen Abweichungen verschoben werden. Im Grunde werden die lokalen Abweichungen bis zur letzten Übereinstimmung entfernt, die Commits der Referenz angewandt und erst danach die ursprünglich lokalen Commits dem Zweig wieder angehangen. Dadurch besteht keine Divergenz zwischen den beiden Zweigen mehr, außer dass der lokale Zweig zusätzliche Commits beinhaltet. Somit ist der lokale Zweig dem Referenzzweig voraus und die Änderungen können nun synchronisiert werden. Dies ist der Zusammenführung vorzuziehen, da mit dieser Variante keine zusätzlichen Commits benötigt werden, sobald sich der Referenzzweig ändert. [31]

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



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

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

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

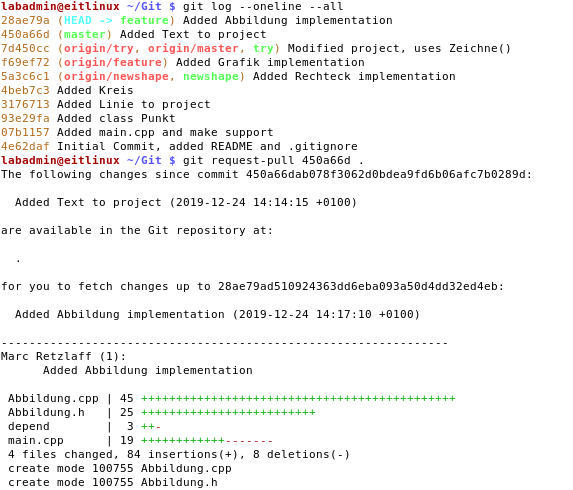


Nachdem weitere Änderungen hinzugefügt wurden auf *master*, und auch auf *feature*, sollen diese Änderungen wiederum zusammengeführt werden. Diesmal dient es die Zusammenführung dazu eine weitere Git Funktion zu präsentieren: *request-pull*.

### 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 weiter ausgebreitet hat. Allerdings hat diese Funktionalität mehrerer Hoster nichts mit dem Git internenKommando ***git request-pull <start> <url>*** nur bedingt Gemeinsamkeiten. Durch das Git Kommando werden Differenzen zweier Stände ermittelt und ausgegeben. Mit dieser Ausgabe kann der Ersteller sich nun an Beteiligte des Projekts wenden. Anhand dieser Informationen weiß der Integrator, der Besitzer des Projektarchivs, welches man erweitern möchte, wo die Änderungen zu finden sind und was diese beinhalten. Dadurch wird der Vorgang erleichtert und die Integration 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 der Anfang der Prüfsumme eines Commits übergeben werden, oder ein Zweig, gegen den man das aktuelle Arbeitsverzeichnis vergleichen will. Dies ist hilfreich, falls man ein *request-pull* für einen kompletten Entwicklungszweig erstellen will, da dadurch automatisch der letzte gemeinsame Vorfahre der beiden Zweige ermittelt wird. [20]

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



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



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



Dadurch ist das Projekt nach Ansicht des Entwicklers an einem Release Stand ankommen. Damit alle Beteiligten des Projekts dies Mitbekommen und den Stand einfach referenzieren können bietet Git eine Möglichkeiten, spezielle Commits über das Kommando *tag* mit Etiketten zu versehen.

## 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. Neben der reinen Kennzeichnung eines Commits beschleunigt es auch den Rücksprung auf eben diesen Commit, da der Name des Etiketts ausreicht um die Arbeitsmappe damit zu laden. 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, wird der Befehl ohne weitere Übergaben ausgeführt. Ein Etikettbehandelt Git grundsätzlich als Verweis auf einen bestimmten Commit. Damit das Etikettauch im Referenzarchiv sichtbar wird, muss es wie ein Commit mit dem Archiv synchronisiert werden. Als Beispiel kann ein solcher Befehl wie folgt aussehen: ***git push origin v1.0*** Dies synchronisiert das Etikett mit dem Namen "v1.0" mit der Referenz und ist nun für alle sichtbar. Um weitere Information über ein Etikett zu erhalten, kann ein Etikett dem Befehl *show* übergeben werden. [5]

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



### Git show

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



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



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

## Eine Änderungsdatei erstellen

### Git diff

Mit dem simplen Kommando ***git diff*** werden die Dateiänderungen in der Arbeitskopie angezeigt, von Dateien, welche nicht im Index Bereich sind. Neu hinzugefügte Dateien darunter 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***. Es gilt zu beachten, dass *diff* nicht jede Datei mit einbezieht, für binäre Dateien muss zum Beispiel ***--binary*** als Schlüsselwort übergeben werden. [23]

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



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

### Git patch

Über das *patch* Kommando werden ähnlich zu einem *pull-request* von Host-Anbieter schnell größere Änderungen übertragen und integriert. Um einen *patch* zu erstellen wird ursprünglich ***git format-patch <Zweig> -1 <Prüfsumme>*** verwendet. Die *-1 <Prüfsumme>* bedeutet hierbei, dass lediglich der übergebene Commit in den *patch* einfließt. Um Zweige zu vergleichen und somit jeden abweichenden Commitin 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*** dem Befehl angehangen. Die *patch* Dateierhält je nach GitKonfiguration einen vordefinierten Header, welcher ursprünglich die Verschickung per Mail direkt möglich machte, siehe ABBILDUNG.



Der Empfänger kann die empfangen Dateien über ***git am <Datei>*** seinem Projektarchiv hinzufügen. Diese werden direkt als Commit hinzugefügt. Die resultierende Datei kann jedoch auch über ***git apply <output>*** auf die aktuelle Arbeitsmappe angewendet werden. [17]

Wird anstatt dem *diff* Befehl in ABBIDLUNG DIFF > patch.diff *format-patch* verwendet, erstellt Git automatisch eine nummerierte Datei:.



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



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

## 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 fehlerhaften 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 die fehlenden Commits bemerkt und daraus schließt, der lokale Stand wäre nicht aktuell. Dies lässt sich zwar durch einen sogenannten *force push* umgehen, allerdings werden hier Commits entfernt, auf dessen andere Nutzer des Projektarchivs gegeben falls Verweise besitzen. Deshalb ist diese Methode nicht weit verbreitet und es wird generell davon abgeraten. Um den Befehl *revert* anwenden zu können, darf die Arbeitsmappe keine Änderungen besitzen. Mit dem Schlüsselwort ***--no-commit*** wird der automatische Commit zurückgehalten und man kann zusätzliche Änderungen an der Arbeitsmappe vornehmen.

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 den fehlerhaften Commit über einen weiteren Commit zu revidieren. 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 <Prüfsumme>.*** Dadurch werden alle durchgeführten Änderungen bis zu dem übergebenen Commit unwiderruflich verworfen. Ohne das Schlüsselwort ***--hard***, finden sich die Änderungen, welche die verworfenen Commits beinhalten in der Arbeitskopie wieder. Falls kein Commit mit übergeben wird, wird die Arbeitskopie zurückgesetzt. Über ***git reset <Datei>*** kann außerdem eine Datei aus dem Index entfernt werden. Die Änderungen der Datei bleiben aber weiterhin erhalten, solange ***--hard*** nicht mit übergeben wird. [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 die modifizierte Datei, welche den ungewünschte Commit beinhaltet im Arbeitsverzeichnis. Ein zweites Mal wird der Befehl mit diesem Schlüsselwort verwendet. Die Arbeitskopie ist nach erfolgtem Befehl auf dem Stand des übergebenen Commits.



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 *git log* die Historie erstellt indem die Commits verfolgt werden, sozusagen von Elternteil zu Elternteil gesprungen wird, dokumentiert *reflog* jegliche Änderungen des HEAD Zeigers. Dadurch enthält der *reflog* Verweise auf Commits und Stände welche möglicherweise sonst nicht mehr sichtbar sind, da kein bestehender Commit auf diese verweist. Deshalb wird ***git reflog*** häufig in Verbindung mit ***git reset HEAD@{x}*** benutzt, wobei ***HEAD@{x}*** aus dem *reflog* entnommen wird. Auch falls ein Commit durch ein *reset* Kommando verloren ging ist dieser nicht gelöscht, sondern nur ohne derzeitige Verweise. Auch in diesem Fall ist die Wahrscheinlichkeit hoch, dies verworfenen Commits im *reflog* zu finden. Allerdings werden diese Commits nach einer festgelegten Zeit verworfen. Somit ist die Rückverfolgung von *reflog* zeitlich begrenzt. [24]

Öffnet man 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 *reset* Befehle sind nur noch im *reflog* vorhanden. Auch hierbei werden durch ***--hard*** alle Änderungen entfernt, beziehungsweise die geänderten Dateien nicht in der Arbeitskopie abgelegt.



## Einzelne Commits überführen

Im Verlauf des Beispielprojekts wurden mehrere Möglichkeiten einer Integration von Änderungen gezeigt. Neben den Kommandos zur Zusammenführung (merge und rebase), wurden Änderungen auch von anderen Zweigen synchronisiert (fetch und pull). 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 <Prüfsumme>*** ähnelt dem *rebase* Kommando. Es wird dazu verwendet einen einzelnen Commit auf einen anderen Zweig anzuwenden im Gegensatz zu einem gesamten Zweig. Dadurch können zum Beispiel Korrektur Commits schnell und einfach in andere Zweige eingegliedert werden. Nachteil hierbei ist, dass der vorhandene Commit in Form eines *patch* auf den aktiven Zweig angewandt wird. Dadurch entfällt die Information, woher der Commit stammt. Er wird als ein neuer Commit behandelt. Um trotzdem eine Information über die Herkunft zu erhalten wird ***-x*** verwendet. Dies hängt an die Commit Nachricht eine Zeile an, welche auf den ursprünglichen Commit verweist. Auch hier können Schlüsselworte wie ***--edit*** oder ***--no-commit*** angewandt werden. [28] Da der Verlust der Herkunft problematisch sein kann, wurde dafür ein weiterer Befehl namens *cherry* eingeführt, welcher hilft solche Integrationen nachzuvollziehen

### Git cherry

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



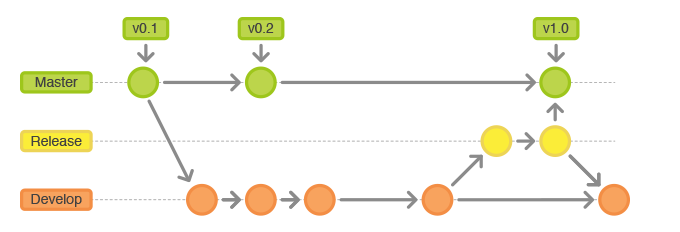
# Workflows mit Git

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 überprüft haben oder ein Build-Prozess, welcher über die Erstellung eines Pull-Requests gestartet wird, erfolgreich das Projekt gebaut und getestet hat. Dies sind nur zwei Beispiele welche über die 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 bei dem Hoster Github.com angebotenen 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]