Sammelsurium

[1] <https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging> 29.10.2018

[2] <https://dev.to/neshaz/how-to-git-stash-your-work-the-correct-way-cna> 29.10.2019

[3] https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning 29.10.2019

[4] <https://git-scm.com/book/de/v1/Git-Branching-Externe-Branches> 04.11.2019

[5] https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag 12.11.2019

[6] [https://githowto.com/git\_internals\_git\_directory 17.11.2019](https://githowto.com/git_internals_git_directory%2017.11.2019)

[11]<https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init> 24.11.2019

[7] <https://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902?pgno=1> 17.11.19

[8] [https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line 18.11.2019](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/resolving-a-merge-conflict-using-the-command-line%2018.11.2019)

[9] [http://gitbu.ch/ch04.html 19.11.2019](http://gitbu.ch/ch04.html%2019.11.2019)

[10] [https://www.atlassian.com/de/git/tutorials/rewriting-history 19.11.2019](https://www.atlassian.com/de/git/tutorials/rewriting-history%2019.11.2019)

[12] [https://www.atlassian.com/de/git/tutorials/using-branches 24.11.2019](https://www.atlassian.com/de/git/tutorials/using-branches%2024.11.2019)

[13] [https://git-scm.com/docs/git-checkout 24.11.19](https://git-scm.com/docs/git-checkout%2024.11.19)

[14] [https://www.git-tower.com/learn/git/faq/detached-head-when-checkout-commit 24.11.19](https://www.git-tower.com/learn/git/faq/detached-head-when-checkout-commit%2024.11.19)

[15] <https://blog.seibert-media.net/blog/2015/01/02/tutorial-git-aufsetzen-teil-2-git-clone/> 24.11.19

[16] [https://www.atlassian.com/de/git/tutorials/syncing 30.11.2019](https://www.atlassian.com/de/git/tutorials/syncing%2030.11.2019)

[17] [https://mijingo.com/blog/creating-and-applying-patch-files-in-git 30.11.2019](https://mijingo.com/blog/creating-and-applying-patch-files-in-git%2030.11.2019)

[18] <https://blog.seibert-media.net/blog/2015/01/06/tutorial-git-aufsetzen-teil-3-git-config/> 1.12.2019  
[19] <https://wiki.siduction.de/index.php?title=Gitk> 1.12.2019

[20] [https://hackernoon.com/how-to-git-pr-from-the-command-line-a5b204a57ab1 1.12.2019](https://hackernoon.com/how-to-git-pr-from-the-command-line-a5b204a57ab1%201.12.2019)

[21] <https://git-scm.com/docs/git-show> 2.12.2019

[22] <https://www.atlassian.com/de/git/tutorials/undoing-changes> 2.12.2019

[23] <https://www.atlassian.com/git/tutorials/saving-changes/git-diff> 2.12.2019

[24] <https://github.blog/2015-06-08-how-to-undo-almost-anything-with-git/> 2.12.2019

[25] [https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History 2.12.2019](https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History%202.12.2019)

[26] [https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame 7.12.19](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame%207.12.19)

[27] 2014\_ProGit p.30 f.

[28] <https://www.ralfebert.de/git/cherry-pick/> 15.12.19

[29] [http://jafrog.com/2012/03/22/git-cherry.html 23.12.2019](http://jafrog.com/2012/03/22/git-cherry.html%2023.12.2019)

[30] <https://www.atlassian.com/git/tutorials/git-lfs> 23.12.2019

Stage (Index)

Ist die lokale Arbeitskopie vorhanden, können nun Änderungen und neue Elemente hinzugefügt werden. Mit dem Befehl ***git status*** werden die derzeitigen Änderungen angezeigt, zu sehen in ABBILDUNG. Unter dem derzeit ausgecheckten Pfad findet man Dateien, welche sich zum letzten Stand des Zweiges verändert haben, sowie neue Dateien, welche noch nicht versionsverwaltet sind. Diese Änderungen können nun dem Index hinzugefügt werden, dazu wird der Befehl ***git add*** benutzt. Hier können explizit Dateien ausgewählt werden, welche für den nächsten Commit vorgemerkt werden. Will man alle Veränderungen übernehmen reicht ein einfaches ***git add . oder git add \**** . Ausgeschlossen durch den \* Operator sind Dateien, welche mit einem Punkt beginnen. Diese müssen explizit hinzugefügt werden. Der Benutzer kann durch den Index gezielt auswählen, welche Teile versionsverwaltet werden sollen. Dadurch ermöglicht man kleine und übersichtliche Commits, welche nur Änderungen beinhalten, welche im Repository oder im speziellen Commit benötigt werden. Man stelle sich eine größere Änderung vor, welche mehrere kleinere Module, unabhängig voneinander, beinhaltet. Anstatt die Änderungen als Gesamtpaket in einen Commit zu verpacken, kann man die Module einzeln dem Index hinzufügen und jeweils ein Commit erstellen. Um differenzierter, nicht nur Dateien, sondern auch einzelne Abschnitte oder Zeilen hinzuzufügen, kann das *add* Kommando noch erweitert werden. Eine Möglichkeit ist ***git add –interactive*** . Hier werden nun mehrere Commandos abgefragt, welche einzeln, nacheinander auf Dateien angewendet werden können. Mit dem Kommando *patch* kann nun eine Datei weiter aufgesplittet werden, um einzelne Zeilen dem Index hinzuzufügen [1]. Um dies für eine einzelne Datei durchzuführen kann ***git add -p <file>*** verwendet werden. ABBILDUNG??? Soll eine Datei aus der Versionsverwaltung entfernt werden, wird sie dem *remove* Kommando übergeben. Der Aufruf lautet hierbei ***git rm <file>.*** Für dauerhafte oder wiederkehrende nicht versionsverwaltete Dateien in der Arbeitskopie ist es hilfreich eine ***.gitignore*** Datei anzulegen beziehungsweise die Datei der .gitignore hinzuzufügen.

Gitignore

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

Stash

Der Stash stell eine Art Zwischenspeicher da. Da die Arbeitskopie bei einem Pfadwechsel überschreiben wird, sollten alle Änderung committet werden, da sie sonst überschreiben werden. Falls man nun den Pfad wechseln will, jedoch Änderungen besitzt, welche erst zu einem späteren Zeitpunkt fertig werden, kann man diese im Stash bis zur Rückkehr zum Pfad ablegen. Auch können darin Ideen oder Ansätze gespeichert werden, um sie nicht zu verlieren. Da ein Stash auf jeglichen Pfad angewendet werden kann, wäre eine weitere mögliche Anwendung darin Änderungen abzulegen, welche das Projekt schnell und temporär verändern, zum Beispiel ein spezielles Debugging Feature. Der einfachste Grund ist die Arbeitskopie zu reinigen, vorheriges aber nicht zwangsweise zu verlieren, falls man darauf zurückkommen sollte. Der Befehl ***git stash*** speichert alle Veränderungen zum Stand des letzten Commits des ausgecheckten Pfads. Mit der Option *–patch* kann wiederum nur ein Teil zwischengespeichert werden, mit *-u* werden nicht versionsverwaltete Dateien mit einbezogen. Um eine Übersicht der Zwischenspeicherungen zu erhalten, listet der Befehl ***git stash list*** alle Gespeicherten auf. Um den Überblick zu behalten kann dem Stash eine Nachricht hinzugefügt werden, dafür wird *save <message>* benötigt. [2] In ABBILDUNG wird die lokale Arbeitskopie gereinigt und alle Änderungen, inklusive der nicht verfolgten Dateien gestashed mit dem Namen "Feature B on master".   
Um den Stash wieder auf die aktuelle Arbeitsmappe anzuwenden wird der Befehl ***git stash pop*** oder ***git stash apply*** verwendet. Im ersten Fall wird der Stash nach der Anwendung gelöscht, bei *apply* bleibt er erhalten. Mit ***git stash show*** werden die Änderungen zum letzten Stash angezeigt. Um einen Stash zu löschen kann der Befehl ***git stash drop <stash-id>*** , wobei die ID zum Beispiel *stash@{2] ist.*

*[3]*

Commit

Commits bilden die zentrale Rolle in der Versionsverwaltung git. Ein Commit entspricht einem bestimmten Stand der Arbeitskopie. Je nach Konfigurationsmanagement und bearbeitenden Pfad haben Commits unterschiedliche Anforderungen. So kann ein Commit in einem Bugfix Branch aus einem einzelnen veränderten Buchstaben bestehen, oder ein Commit entspricht einem neuen Feature auf einem Entwicklungspfad. In Absatz HIERVERLINKEN Gitflow wird deutlich wofür ein Commit auf dem jeweiligen Pfad steht. Ein Commit enthält neben dem Abbild der aktuellen Arbeitskopie, sofern alles gestaged wurde, auch eine Commit Nachricht. Diese sollte eine möglichst kurze, aber dennoch Aussagende Zusammenfassung der Änderungen beinhalten. Die Commit Nachricht ist neben *tags* die schnellste Möglichkeit, Änderungen zu finden. Für schnelle Übersicht und Zusammenarbeit sollte man sich daher ein Schema überlegen, welche Projektweit gilt und angewendet wird. Ein Commit verweist auch immer auf sein *parent.* Anhand diesem kann man feststellen, woher der Commit stammte, beziehungsweise, welche Änderungen im Vergleich zu seinem *parent* einflossen. Durch diese Verknüpfung kann man nun die Commits anordnen und erhält damit die berühmte Baumstruktur. Jeder neue Commit wird an dessen vorhergehenden angehangen und erweitert die Kette. Wird ein neuer Pfad erstellt, zeigen zwei Commits auf den gleichen Vorgänger, es entsteht eine Abzweigung. Im Gegensatz dazu besitzt ein Merge-Commit mehrere Vorgänger. Werden Zweige wieder zusammengefügt hat der folgende Commit zwei *parents*. Durch die Kette an Commits bildet sich eine lückenlose Historie bis zum Ursprung des git Repository. Commits werden intern durch ihre Prüfsumme unterschieden und darüber auch gespeichert. Um auf bestimmte Commits zu verweisen wird dafür oftmals der Anfang ihrer SHA1- Prüfsumme verwendet. [10] Falls man den letzten Commit noch einmal bearbeiten will hilft das Schlüsselwort ***–amend***. Dadurch kann man zum Beispiel die Commit Nachricht überarbeiten, sofern die aktuelle staging area noch leer ist. Hat man vergessen etwas dem Commit hinzuzufügen kann man auch diese hinzufügen oder über ***-m "<message>"*** die Nachricht umändern. Allerdings überschreibt der neue Commit den Alten wodurch alle Referenzen auf diesen verloren gehen. Daher sollte man *–amend* nur nutzen, sofern keine anderen Parteien schon auf diesen aufgebaut haben. Faustregel lautet: Kein *amend* auf veröffentlichte Commits.

[4] für unteren

Push

Der Befehl ***git push*** wird dazu benötigt die lokalen Änderungen im git repository auf den globalen Server zu legen. Somit wird *push* nur benötigt, wenn man einen nicht lokalen Host nutzt, zum Beispiel das Repository auf *github.com* liegt. Der globale Hauptstandort wird meist *origin* genannt.  
Um zum Beispiel Änderungen auf dem Pfad *master* auf dem globalen Repository zu veröffentlichen kann der Befehl ***git push origin master*** benutzt werden. Da es im Allgemeinen nicht gewollt ist, dass jeder direkten Zugriff auf das *repository* besitzt, wird im Normalfall eine Anmeldung verlangt. Dies wird umgangen, indem man die Anmeldung bereits im *remote* Link enthält. Dadurch stehen die Anmeldedaten allerdings einsehbar unter den *remotes.* Daher kann zum Beispiel eine Verifikation über SSH erfolgen. Dafür muss das SSH Schlüsselpaar einmalig angelegt werden, dem Hoster der jeweilige Public Key mitgeteilt werden, sodass man über SSH nun Zugriff hat. Um dies zu nutzen muss dementsprechend der *remote* link auf SSH gewechselt werden.

Pull

Der Befehl ***git pull*** wird benötigt um das lokale repository mit (falls vorhandenen) Änderungen von *origin* upzudaten*.* Erweitert mit –all werde alle Zweige abgeglichen, mit ***git pull origin master*** wird nur *master* abgeglichen und falls vorhanden mit dem lokalen Stand zusammengeführt. Da es hierbei zu Konflikten kommen kann, sollte man nur den *pull* Befehl ausführen, wenn man eine frische Arbeitskopie besitzt, beziehungsweise, alle Änderungen commited hat*.* Ist dies nicht der Fall wird automatisch ein weiterer *merge commit* hinzugefügt.Will man sich nur über Neuerungen informieren ohne Änderungen lokal vorzunehmen, sollte *fetch* nutzen. Pull ist somit eine Kombination aus fetch und merge.

Fetch

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

Tags [5]

Mit sogenannten Tags markiert man einzelne Commits. Damit können zum Beispiel wichtige Versionen hervorgehoben werden. Oftmals erhalten je nach Konfigurationsmanagement Releases und gewisse wichtige Versionen einen Tag. *git* unterscheidet dabei zwichen kommentierten und nicht kommentierten *tags*. Um einen *tag* zu erstellen wird ***git tag <name>*** verwendet. Um zusätzlich eine Nachricht anzufügen kann ***-m 'Version 1'*** angehangen werden. Um sich die Liste aller vorhanden *tags* anzusehen bietet *git* den Befehl ***git tag***. Einen *tag* behandelt git grundsätzlich als Verweis auf einen bestimmten Commit. Damit der *tag* auch global sichtbar wird, muss er wie ein Commit gepushed werden, also zum Beispiel ***git push origin v1.0*** um den *tag v1.0* zu synchronisieren.

Git init [6] -> screenshots von vm .git ordner anfügen

Das Kommando ***git init*** fügt dem aktuellen Verzeichnis ein *.git* Ordner hinzu. Ohne zuerst in den Ordner zu wechseln kann auch der Pfad nach ***git init*** angegeben werden. Falls der übergebene Ordner nicht existiert, wird dieser erstellt. Dieser Ordner enthält alle wichtigen Informationen zum Repository. Der *.git* Ordner reicht alleine, um das Repository zu klonen. Wie in ABBILDUNG zu sehen, besitz *.git* mehrere Unterordner. Commits befinden sich verschlüsselt in dem Ordner objects. *Refs* enthält Verweise auf die verschiedenen Branches und Tags. Scripte die bei git Kommandos ausgeführt werden befinden sich unter hooks. Die *Config* Datei enthält Information und Einstellungen für das Repository. Erstellt man ein *shared repository* welches als reines *push* und *pull* *repository* genutzt wird, fügt man dem Kommando ***–bare*** hinzu. Dies führt dazu, dass es keine Arbeitskopie erhält, sodass man auf dem zentralen Repository keine Dateien editieren oder *commiten* kann. Um dies zu tun, muss das Repository zuerst mit *git clone* geklont werden. [11]

Git merge (3way [7] Bilder einfügen?)

Git verwendet einen sogenannten Drei-Wege-Merge. Merge wird dazu benutzt, um Zweige zusammenzuführen, beziehungsweise Änderungen von anderen zu übernehmen. Um nun verschiedene Commits zusammenzuführen werden beide Stände der Arbeitskopien Datei für Datei verglichen. Im Gegensatz zu einem Zwei-Wege-Merge, beziehungsweise einem einfachen *diff* wird durch die Änderungsverfolgung von *git* auch der frühste gemeinsame Vorgänger beider Dateien einbezogen. Dadurch erkennt das *merge tool* neben der Abweichung beider Stände, unter Einbezug des Vorfahren, auch welcher der beiden Stände die zu übernehmende Änderung besitzt. Konflikte entstehen somit nur, falls beide Stände Abweichung zum Vorfahren besitzen. Des Weiteren wird nachdem ein Merge vollzogen wurde ein *merge-link* erstellt. Dadurch wird sichergestellt, dass bei einem späteren Merge keine schon gelösten Konflikte noch einmal durchgeführt werden müssen, da der Vorfahre durch den *link* mitgezogen wird. Um nun solch einen *merge* durchzuführen wird zuerst der Zielpfad als aktuelles Repository gesetzt. Dann kann der *merge* über ***git merge <source-branch>*** vollzogen werden. Besitzt der Zielpfad keine Abweichenden Commits, sodass der Quellpfad nur Änderungen über den Zielpfad hinaus besitzt, findet ein *fast-forward merge* statt. Da keine Dateien zusammengeführt werden müssen, wird nur der interne Zeiger des Zielzweiges auf den Commit des Quellpfads vorgerückt. Um das zu unterbinden und immer einen dedizierten Commit zu erstellen wird die Option ***-no-ff*** verwendet. Um den *merge-commit* zusätzlich zu überprüfen oder letzte Änderungen manuell einzufügen kann ***-no-commit*** verwendet werden. Die Arbeitskopie enthält danach die Ausgabe der Zusammenführung. Dies sollte man bei wichtigen Implementierungen zu Hauptzweigen verwenden, um die zusammengeführten Dateien manuell zu überprüfen. Danach kann der Merge Commit manuell erstellt werden. Die Nachricht für den *merge-commit* wird mit ***-m*** übergeben. Auch kann man eine Strategie für den *merge* wählen. Die zwei wichtigsten sind **-*ours*** und ***-theirs***. Damit kann bei Konflikten generell eine Version bevorzugt werden. Eine weitere nutzvolle Option bietet ***–squash***. Hierdurch werden alle Änderungen, welche in den merge einfließen zu einem Commit zusammengefasst und als neuer Commit dem Ziel Zweig hinzugefügt. Es wird allerdings keine Verbindung zum Ursprungs Zweig hergestellt. Git sieht den Ursprungs Zweig als nicht zusammengeführt an. Sollte man sich während der Zusammenführung umentscheiden oder es kamen unbekannte Komplikationen auf, kann der Vorgang verworfen werden über ***git reset –merge.***

Konflikte [8]

Konflikte enstehen unter Umständen bei *merge* oder *rebase* Kommandos. Durch den 3-Wege-Merge ensteht ein Konflikt nur, wenn sich beide Versionen an derselben Stelle mit der Version des gemeinsamen Vorfahren unterscheiden. In diesem Fall wird *merge* sowie *rebase* unterbrochen und der Konflikt benötigt eine manuelle Korrektur. Dazu wird oftmals ein *merge-tool* zur Hilfe angewendet, da durch eine *gui* Lösung meist deutlich übersichtlicher und schneller editiert werden kann. In der Konfliktbehafteten Datei werden beide Versionen dargestellt. Zur Identifizierung startet bei dem sich überschneidenden Teil der Zielpfad mit "<<<<<<< HEAD". Die Version des Quellpfads steht gefolgt zwischen "=======" und ">>>>>>> <branch>". Um den Konflikt nun zu lösen müssen die merge-marker, sowie die ungewünschte Version entfernt werden. Danach die korrigierte Datei über ***git add <file>*** wieder hinzufügen, sodass der Konflikt als gelöst gilt. Gefolgt von ***git commit*** wird nun der *merge-commit* vollzogen. Falls man nicht auf diese Konfliktlösung zurückgreifen will kann der *merge* Versuch auch über ***git merge –abort*** abgebrochen werden. Ein Sonderfall stellt eine Datei dar, welche nur in einem der beiden Zweige existiert, man entscheidet sich dann über *git add* oder *git rm* die Datei zu behalten oder zu entfernen.

Git rebase [9]

Neben *merge* steht auch das Kommando *rebase* zur Verfügung, um verschiedene Zweige zu vereinen. Generell wird das *rebase* Kommando dazu genutzt die Historie zu verändern. In der einfachsten Form benötigt man *rebase* um die Historie übersichtlicher zu gestalten. Im Wesentlichen versucht *rebase* den Vorfahren eines Zweiges zu verändern. Nehme man an, man wolle ein Entwicklungszweig, welcher seit längerer Zeit besteht, nun implementieren, aber der Hauptentwicklungszweig ist schon weit vorangeschritten. Damit es nun übersichtlich bleibt, will man den Entwicklungszweig vorschieben, damit der Vorfahre der Abzweigung der neuste Commit des Hauptentwicklungszweiges ist. Dies geschieht mit dem Kommando ***git rebase maindev dev*** oder, falls man im Entwicklungszweig ist ( ***git checkout dev***) reicht ein ***git rebase maindev .*** Dadurch zeigt nun der erste Commit von *dev* auf den letzten Commit von *maindev*, vorausgesetzt es entstand kein Konflikt. Um nun zwei Zweige zu vereinen muss man nur noch einen Schritt weiter gehen, indem man nun umgekehrt mit ***git checkout maindev*** auf den *maindev* Zweig wechselt und umgekehrt ***git rebase dev*** auslöst. Folgend werden nun alle Commits von *dev* auf *maindev* angewendet. Da *dev* durch den ersten *rebase* aus dem neusten Commit herausgelöst wurde, ist die Integration folgend nur noch ein *fast-forward-merge*. Der Zeiger von *maindev* wird also um die Zeiger von *dev* inkrementiert. Alternativ dazu könnte man auch ***git fetch dev:maindev*** verwenden um ein *fast-forward* zu bewirken. Neben diesem Hauptbestandteil von *rebase* bietet das Kommando noch einen deutlich größeren und komplizierteren Umfang. Mit dem Schlüsselwort ***–onto*** wird das Kommando zum Beispiel auf ***git rebase –onto maindev <start> <ziel>*** erweitert, um nur bestimmte *commits* zu übernehmen und deren Historie zu verändern. Generell gilt Vorsicht bei *rebase* da die Historie ein wesentlicher Vorteil der Versionsverwaltung darstellt. Wird die Historie zu arg verschlankt, da ein Großteil der Zweige nach Beendigung per *rebase* Hauptzweigen zugeführt werden, sind die Zweige zwar übersichtlich, allerdings lassen sich einzelne Arbeiten nicht mehr unterscheiden. Auch werden die Zeitpunkte der Commits verändert, sowie die Checksumme, daher sollte man sich vergewissern, dass niemand auf Commits explizit angewiesen ist, welche man per *rebase* verschiebt oder umschreibt. Deshalb sollte man nur in persönlichen Zweigen *rebasen* um Konflikte zu vermeiden, bei dem andere Branches zum Beispiel auf veraltete Commits verweisen, da der Zweig umgeschrieben wurde. Auch hier sollte eine gesunde Mischung aus *rebase* und *merge* verwendet werden um eine schlanke, übersichtliche Historie zu erhalten ohne klare Strukturen und Abgrenzungen zu verlieren. BENUTZT Ein weiteres häufig benutztes Schlüsselwort ist *–i* beziehungsweise *–interactive.* Dadurch öffnet sich der interaktive *rebase* Modus, bei dem man jeden einzelnen Commit betroffen vom *rebase* individuell verändern kann. So kann man die Reihenfolge der Commits verändern, einzelne Commits löschen, Commits verschmelzen und bearbeiten. Dies geschieht in dem man die erste Spalte dementsprechend anpasst. Hierfür werden wiederum spezielle Schlüsselwörter benutzt. Um einen Commit zu löschen, entfernt man allerdings einfach dessen Zeile. Dies wird häufig verwendet, um mehrere Commits schnell zu verwerfen. Die Reihenfolge der Commits entspricht der Reihenfolge, mit der *rebase* sie auf den neuen Vorfahren anwendet. Standardmäßig ist dies genau umgekehrt der Baumreihenfolge, da *rebase* zuerst den zeitlich ersten Commit anwenden muss, um die Reihenfolge wiederherzustellen. Das Schlüsselwort *reword* ermöglicht es die Commit Nachricht neu zu schreiben. Mit *squash* verschmelzt sich der Commit mit letzterem. Dies wird häufig dazu benutzt, um die Baumstruktur übersichtlich zu halten oder Bugfix-Commits für Commits verschwinden zu lassen. Hier kann auch der Anhang ***–root*** von Nutzen sein, falls man einen Zweig überarbeiten will. Dadurch entfällt das vorherige Suche des Hashes oder die Anzahl der Commits.

Git branch [12]

*Branches* werden angewendet, um isolierte Entwicklungszweige zu erhalten. So ist es von Vorteil eine Sanbox des aktuellen Arbeitsstand zu verwenden, um Entwicklungen jeglicher Art vorzunehmen, ohne dabei die Gefahr zu haben instabilen Code im Hauptzweig zu erhalten. Auch kann gerade bei Projekten mit mehreren Entwicklern unter Verwendung von *branches* sichergestellt werden, dass man sich nicht gegenseitig in die Quere kommt und immer ein gesicherter Hauptstand da ist, auf welchen man sich beziehen kann. Aufgrund dieser Isolation lassen sich Projekte mit mehreren Entwicklern ohne größere Abstimmung und Nachverfolgung untereinander erledigen. Intern sind *branches* für git nur weitere zeiger welche auf Commits zeigen. Um einen neuen *branch* zu erstellen wir der Befehl ***git branch <name>*** verwendet. Der neue *branch* zweigt nun von dem aktuelle aktiven *branch* ab. Im Grunde wird dafür allerdings ***git checkout -b <name>*** verwendet, um den neu kreierten *branch* anzulegen und direkt als aktuellen *branch* auszuchecken. Wird ein *branch* nicht mehr benötigt lässt er sich mit ***git branch -b*** löschen. Dies ist nur möglich sofern alle Änderungen in andere *branches* übernommen wurden. Ist dies nicht der Fall und man möchte ihn trotzdem löschen wird ein kapitalisiertes **-B** benötigt. Um alle *branches* des Repository anzusehen wir ***git branch -l*** benötigt. Soll der lokale *branch* auch für das *remote repository* übernommen werden muss der *branch* ähnlich wie ein Commit gepushed werden, beziehungsweise der lokale *branch* einem *branch* des *remote repository* folgen soll, ein sogenannter *tracking branch.* Dafür gibt es wie fast für jede Funktion von git mehre Möglichkeiten, am einfachsten kann ***git push -u origin <branch>*** verwendet werden. Um sicherzustellen, dass der *branch* auch unter dem gleichen Namen im *remote repository* erstellt wird oder explizit unter einem anderen Namen, kann anstatt *<branch>* auch *<branch>:<remotebranchname>* benutzt werden. Ist man in dem *branch,* welcher hinzugefügt wird kann auch *HEAD* stattdessenverwendet werden. Manchmal ist es sinnvoll nachzuvollziehen zu können ob ein bestimmter Commit schon einem Zweig zugeführt wurde. Dafür kann ***git branch –contains <hash>*** verwendet werden. Nun werden alle Zweige aufgelistet, die diesen Commit beinhalten. Dies funktioniert jedoch nur, falls genau dieser Commit zugeführt wurde. Ist der Commit zum Beispiel über einen *chery-pick* oder *patch* zugeführt wurden, erkennt dies git nicht als gleichen Commit. Hierfür hilft ***cherry***.

Git checkout [13]

Mit ***git checkout <branch>*** wird der aktuell zu bearbeitende *branch* gewechselt. Hierbei werden alle Änderungen in der Arbeitkopie behalten. Intern wechselt *HEAD* auf den neusten Commits des *branches.* Allerdings können nicht nur *branches* übergeben werden, auch einzelne Commits und Dateien können mit *checkout* verwendet werden. Bei einer Datei wird damit die letzte Version dieser Datei der Arbeitskopie hinzugefügt beziehungsweise, die vorhande Datei überschrieben. Dies ist nützlich falls man Änderungen an einer einzelnen Datei verwerfen will. Wird ein Commit, besser gesagt dessen Checksumme, übergeben, entsteht dadurch ein sogenannter *detached HEAD*. Das heißt der *HEAD* Zeiger zeigt nun nicht auf den neusten Commits eines *branches.* HEAD ist dafür zuständig, welche Versionen von Dateien dem aktuellen Arbeitsverzeichnis hinzugefügt werden. Ist Head nun nicht an den neusten Commit eines *branches* gekoppelt, fehlt die Zuordnung an einen *branch.* Arbeitet man nun an dem Commit weiter, das heißt, man erstellt einen neuen Commit, dann wird eine neue Abzweigung erstellt. Diese Abzweigung besitzt allerdings keinen Namen, den man referenzieren könnte, wie bei einem *branch.* Daher muss der Abzweigung eine Referenz hinzugefügt werden, um wieder darauf zurückgreifen zu können, sobald man von der Abzweigung wechselt. Die kann man, indem man einen *tag* setzt oder nachträglich einen *branch* kreiert mit ***git branch <name>***. [14]

Git clone [15]

Mit diesem Komando wird ein neues Repository angelegt, jedoch als Vorlage ein bestehendes Repository verwendet. Man klont somit das Repository, welches nun sein eigenes ist. Zusätzlich wird direkt die Verbindung zum ursprünglichen Repository beibehalten und unter *origin* hinterlegt. So kann ohne Umwege Änderungen am ursprünglichen Repository verfolgt und in das eigene übernommen werden. Verwendet wird das Kommando mit ***git clone <remoteAdresse> <lokalesVerzeichnis>***. Als einfaches Beispiel wird in ABBILDUNG ein öffentliches *repository* von GitHub geklont. Im Falle eines privaten *repository* wird *https://<account>:<passwort>@github.com/...* verwendet. Neben https.// kann auch ssh.// oder git:// je nach Anwendung mit ***git clone*** verwendet werden.

Git remote [16]

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

Git patch [17]

Über das *patch* Kommando werden ähnlich zu einem *pull-request* von Hostern schnell größere Änderungen übertragen und integriert. Um einen *patch* zu erstellen wird ursprünglich ***git format-patch <branch> -1 <hash>*** verwendet. Die *-1 <hash>* bedeutet hierbei, dass lediglich der Commit in den *patch* einfließt. Um Zweige zu vergleichen und somit jeden *commit* in einen *patch* zu verpacken, welcher nicht auf dem nicht aktiven Zweig ist, muss lediglich dieser Zweig *format-patch* übergeben werden. Soll eine einzige *.patch* Datei erstellt werden, anstatt eine Datei für jeden Commit wird ***–stdout*** angehangen. Die *patch* Dateierhält je nach *git* Konfiguration einen vordefinierten Header, welcher ursprünglich die Verschickung per Mail direkt möglich machte. Der Empfänger kann, die empfangen Dateien über ***git am <file>*** seinem repository hinzufügen. Diese werden direkt als Commit hinzugefügt. Eine andere Möglichkeit einen Patch zu erstellen schafft ***git diff <von> <bis> > <output>***. Dieser besitzt dadurch lediglich die Unterschiede der Dateien und somit keinen Header. Es gilt zu beachten, dass *diff* nicht jede Datei mit einbezieht, für binäre Dateien muss zum beispiel ***–binary*** als Schlüsselwort übergeben werden. Die resultierende Datei kann danach über ***git apply <output>*** auf die aktuelle Arbeitsmappe angewendet werden.

Git config [18]

Wird eine neues Repository erstellt oder auch nach einer lokalen Installation von git sollte man Einstellungen vornehmen, um git nach seinen eigenen Wünschen auszurichten. Zuerst sollte man allerdings seine persönlichen Daten anlegen, damit bei der Zusammenarbeit mit anderen Nutzern, die Historie jederzeit einem Nutzer zuzuordnen ist. Dafür wird der ***git config*** Befehl benutzt. Mit dem Anhang ***–global*** werden die Einstellungen nicht nur für das aktuelle *repository* benutzt, sondern auch für den angemeldeten Benutzer. Hierfür wird mit ***git config –global user.name <name>*** beziehungsweise ***user.email*** die zwei wichtigsten Informationen gesetzt. Weitergehend kann der Standard Editor geändert werden: core.editor, oder man ändert die Einstellungen in der respektiven Datei, aufzurufen mit ***git config –global –edit***. Des Weiteren können Aliase erstellt werden, um individualisierte Commandos zu erstellen oder bestehende abzukürzen. Dies geschieht grundsätzlich mit ***git config alias.<alias name> <git-command>.***

Git log [25]

Um einen Überblick über den aktuellen Stand des Zweiges kann eine Übersicht der letzten Commits mit ***git log*** aufgerufen werden. Dort sieht man alle Commits des Zweiges mit Information zum Autor, Datum und Hash des Commits. In Kurzform und meist ausreichen kann auch ***git shortlog*** benutzt werden, dadurch wird nur die Commit Nachricht angezeigt. Um zusätzlich den Anfang des jeweiligen Hash zu erhalten kann ***git log –oneline*** benutzt werden. Ein weiteres nützliches Werkzeug bietet der Anhang ***–follow <file>***. Hier werden nun alle Änderungen an der Datei mit dem jeweiligen Commit ausgegeben. Mit ***–grep*** werden nur Commits mit passender Commit Nachricht angezeigt. Um auch in den Änderungen des Commits zu suchen, wird ***-S<search>*** verwendet. Mit ***-<zahl>*** kann die Ausgabe auf eine bestimmte Anzahl an Commits reduziert werden.[27]Zur Orientierung der jeweiligen Commits kann zusätzlich ***–graph*** übergeben werden. Dadurch entsteht an den ersten Stellen der Baum, welcher das Repository wiederspiegelt, um die Commits schneller den Zweigen zuordnen zu können.

Gitk [19]

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

Git request pull [20]

In Verbindungen mit *git* fällt oftmals der Begriff "Pull-Request". Womöglich ist dies auch der Grund warum git in den letzten Jahren ohne Konkurrenz sich immer weiterverbreitet hat. Allerdings hat dies mit dem *git* Kommando ***git request-pull <start> <url>*** nur bedingt Gemeinsamkeiten. Durch das Kommando werden Differenzen ermittelt und ausgegeben, mit dessen der Ersteller sich nun an eine andere Partei wenden kann. Durch diese Information weiß der Integrator, der Besitzer des *repositorys*, welches man erweitern möchte, wo die Änderungen zu finden sind und was diese beinhalten. Dadurch wird der Vorgang erleichtert und kann schneller ablaufen. Im Grunde ist es eine andere Form Änderungen einzureichen und kann als Vorläufer eines *patches* angesehen werden, da hier die Entscheidung der Integration grundsätzlich erst noch gefällt werden muss. Im Gegensatz zum Pull-Request von diversen Hostern, welche über das Kommando automatisch Code Reviews und Code Integration, Kommentare einfließen lassen, bietet das request-pull Kommando lediglich eine Zusammenfassung von Änderungen. Die ***<url>*** bezieht sich auf das Ziel *repository*. Da es sich oftmals um einen Klon handelt und man deshalb das lokale *repository* aktuell zum *remote* ist, wird hierfür ein ***.*** für das lokale verwendet. Unter ***<start>*** kann entweder die Hashsumme eines Commits übergeben werden, oder der Zweig, gegen den man vergleichen will. Dies ist hilfreich falls man ein *request-pull* für einen kompletten zweig erstellen will, da dadurch automatisch der letzte gemeinsame Vorfahre der beiden Zweige ermittelt wird.

Git show [21]

Mit ***git show*** werden Informationen zum letzten Commit des aktiven Zweiges, beziehungsweise der Commit, auf welchen HEAD zeigt, angezeigt. Um Informationen über einen einzelnen Commit zu erhalten kann dessen Hash übergeben werden, genauso können *tags* und trees angezeigt werden.

Git reset [22]

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

Git diff [23]

Mit dem simplen Kommando ***git diff*** werden Dateiunterschiede angezeigt, welche nicht im Index Bereich sind. Hinzugefügte Dateien werden nicht berücksichtigt. Um auch eine Differenz für diese Dateien im Index zu erhalten muss das Schlüsselwort ***–staged*** übergeben werden. Ähnlich wie das Bash-Kommando diff, können auch hier einzelne Dateien, Zweige oder Commit-Hashes mitgegeben werden. Auch ist ***git diff*** für die Erstellung von Patches hilfreich. Dies kann zum Beispiel geschehen über ***git diff –staged > mypatch.patch***.

Git revert[22]

Das Kommando ***git revert HEAD*** wird dazu benötigt einen durchgeführten Commit zu übergehen. Hierfür wird ein neuer Commit durchgeführt welcher den Stand vor dem letzten Commit wiederspiegelt. Dies kann auch erweitert werden indem man den Hash oder respektive ***HEAD~<Anzahl an Commits>*** übergibt. Dadurch ist der ungewollte Commit zwar in der Historie vorhanden, besitzt aber keinen Auswirkungen mehr auf den aktuellen Stand des Zweiges. Diese Methodik wird vor allem für Commits angewandt welche zuvor über den Befehl ***git push*** mit dem *remote repository* synchronisiert wurden. Ist dies geschehen, können keine Commits gelöscht werden, da git den fehlenden Commits bemerkt und daraus schließt, der lokale Stand wäre nicht aktuell. Dies lässt sich zwar durch einen *force push* umgehen, allerdings werden hier Commits entfernt, auf dessen andere Nutzer des Repositorys gegebenfalls Verweise besitzen. Deshalb ist diese Methode nicht weit verbreitet. Um *revert* anwenden zu können darf die Arbeitsmappe keine Änderungen besitzen. Mit dem Schlüsselwort –no-commit wird der automatische Commit zurückgehalten und man kann zusätzliche Änderungen vornehmen.

Git reflog [24]

Während *log* die Historie erstellt indem der derzeitige HEAD Zeiger verfolgt wird, sozusagen von Elternteil zu Elternteil gesprungen wird, dokumentiert *reflog* jegliche Änderungen des HEAD Zeigers. Dadurch enthält der *reflog* Verweise auf Commits und Stände welche möglicherweise sonst nicht mehr sichtbar sind. Deshalb wird ***git reflog*** häufig in Verbindung mit ***git reset HEAD@{x}*** benutzt, wobei ***HEAD@{x}*** aus dem *Reflog* entnommen wird. Auch falls ein Commit durch ein Reset verloren ging ist dieser nicht gelöscht, sondern nur ohne Verweise. Auch in diesem Fall ist die Wahrscheinlichkeit hoch diesen im *reflog* zu finden.

Git blame [26]

Um herauszufinden welcher Autor welche Änderungen an einer Datei durchgeführt hat, wird das Kommando ***git blame <file>*** verwendet. Dadurch wird nun für jede Zeile der Datei der Autor bei der letzten Änderung angegeben. Um die Ausgabe weiter zu verfeinern kann über das Keyword *-L <start>,<end>* die Ausgabe auf bestimmte Zeilen verkleinert werden. Mit *-M* wird der Ersteller von Zeilen, welche sich nicht verändert haben, sondern nur kopiert beziehungsweise verschoben wurden, berücksichtigt. Das heißt, der Autor dieser Zeilen ist nun der Ersteller, und nicht der Autor des Commits, welcher die Zeile verschob oder kopierte. Mit *-C* wir das herausfinden des Erstellers auf das *repository* ausgeweitet.

Git mergetool (Anhang zu "Konflikte")

Falls es zu einem *Merge* Konflikt kommt, sollte dieser bereinigt werden, um fortfahren zu können. Git unterbricht den automatischen *merge commit* falls es zu Konflikten kommt. Mit *git status* werden die Dateien, welche einen Konflikt hervorrufen angezeigt. Um nun den Konflikt zu lösen muss der Benutzer den Konflikt lösen, indem dieser die hergestellte Datei seinen Vorstellungen nach anpasst. Durch solch einen Konflikt werden außerdem mehrere temporäre Dateien erzeugt um jeden Stand wiederspielgen zu können. So werden automatisch eine *remote, base, local* Version erstellt, sowie eine backup Version der Konflikt behafteten Datei, inklusive den Konflikt Marker Zeilen. Für eine schnelle und übersichtliche Lösung des Konflikts wird meistens ein *mergetool* verwendet. Um dieses Programm nicht explizit aufrufen zu müssen, wurde ***git mergetool*** bereitgestellt. Dadurch wird das hinterlegte Programm automatisch aufgerufen. Das hinterlegte Programm kann über ***git config*** verändert werden. Standardmäßig werden dadurch vimdiff oder meld aufgerufen. In diesem Programm kann nun der Konflikt sauber aufgelöst werden. Nach Beendigung des Programms sollte git nun erkennen, dass kein Konflikt mehr besteht. Über *git commit* wird nun der gestoppte *merge* vollzogen.

Git cherry-pick [28]

Das Kommando ***git cherry-pick <hash>*** ähnelt dem *rebase* Kommando. Es wird dazu verwendet einen einzelnen Commit auf einen anderen Zweig anzuwenden im Gegensatz zu einem gesamten Zweig. Dadurch können zum Beispiel Fixes schnell und einfach in andere zweige eingegliedert werden. Nachteil hierbei ist, dass der vorhandene Commit in Form eines *Patch* auf den aktiven Zweig angewandt wird. Dadurch entfällt die Information, woher der Commit stammt. Er wird als ein neuer Commit behandelt. Um trotzdem eine Information über die Herkunft zu erhalten wird ***-x*** verwendet. Dies hängt an die Commit Nachricht eine Zeile ein, welche auf den ursprünglichen Commit verweist. Auch hier können Schlüsselworte wie ***-edit*** oder ***–no-commit*** können angewandt werden.

Git cherry [29]

Um nachvollziehen zu können ob Änderungen einem Zweig zugeführt wurden kann ***git cherry <branch1> <branch2> <limit>*** verwendet werden. Das Kommando überprüft die eingeführten Änderungen jedes Commits, anstatt eine übereinstimmende Prüfsumme wie *branch –contains.* Dadurch werden auch Commits, welche über *cherry-pick, patches* zugeführt wurden, erkannt. Das Kommando überprüft dabei, ob die Commits aus *<branch2>* in *<branch1>* existieren. Das Limit gibt an, wie viele Commits rückwirkend gesucht werden. Dies ist hilfreich, um die Ausgabe einzugrenzen. Es erzeugt eine Liste der Commits aus *<branch2>* welche durch ein Pluszeichen oder Minuszeichen angeführt werden. Ein Minuszeichen steht für ein übereinstimmende Änderungen. Das Pluszeichen für fehlende Änderungen in <*branch1>.*

LFS [30]

Git LFS (large file storage) ist eine Erweiterung für git welche die Handhabung von großen Dateien vereinfacht. Ein großer Vorteil, jedoch in diesem Aspekt Nachteil von dezentralen Versionsverwaltungen ist der Fakt, dass jeder alles besitzt. Dadurch kann bei großen Dateien der Speicherplatzverbrauch enorm ansteigen, da in jedem Commit dieselbe Datei wieder und wieder gespeichert wird. Auch steigt die Zeit für die Synchronisation an. Ist LFS installiert und aktiviert, werden nun bestimmte Dateitypen über dieses System verwaltet. Hier wird vereinfacht ein Container für diese Dateien angelegt, um sie einmalig zu speichern. Zusätzlich werden intern für git weitere Zwischenschritte angewendet, sodass bei einem Verweis auf diese Dateien lediglich eine Referenz, zeigend zu dem Container eingefügt wird. Dadurch wird nun nur noch ein Verweis in Form einer SHA256 Checksumme im Commit gespeichert anstatt der Datei. Wird die Datei verändert wird sie wiederum einmalig im Container gespeichert. Die Container werden mit den nativen git Kommandos *Push&Pull* synchronisiert, sodass alle referenzierten Dateien im lokalen Container vorhanden sind. Um übergreifend damit arbeiten zu können, müssen alle git Instanzen, welche mit diesem Repository arbeiten LFS beherrschen können. Auch ist es möglich bestehende Repositorys nachträglich mit LFS zu versehen. Hierbei wird, falls gewünscht, auch rückwirkend LFS eingeführt, wobei die Historie umgeschrieben wird. In solch einem Fall sollte jeder beteiligte das Repository neu klonen. LFS lässt sich großzügig konfigurieren über die vorhandene git Konfigurations-Datei. Zusätzlich bietet LFS eigene git Kommandos rund um LFS. Hierbei können auch Dateitypen zur LFS Verwaltung hinzugefügt werden über ***git lfs track "\*.<extension>"***. Eine weitere Funktion von LFS ist die Möglichkeit diese Dateien zu sperren. Dadurch können, falls gewünscht, Konflikte mit Dateien verhindert werden, welche sich nicht zusammenführen lassen, wie zum Beispiel Binärdateien.

GIT KOMMANDOLISTE

Git init check

Git clone check

Git status check

Git diff check

Git add check

Git commit check

Git log check

Git branch check

Git checkout check

Git tag check

Git remote check

Git fetch check

Git pull check

Git push check

Git merge check

Git rebase check

Git rm check

Git revert check

Git reset check

Git show check

Git blame check

Git request-pull check

Git gitk check

Git gitgui mit in gitk check

Git patch check

Git config check

Git stash check

Git mergetool check

Git cherry-pick check

Git cherry check