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

Stage

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 Pfades verändert haben, sowie neue Dateien, welche noch nicht versionsverwaltet sind. Diese Änderungen können nun gestaged 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 \**** . Der Benutzer kann mit dem Stage Level 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 zu comitten, kann man die Module einzeln Stagen und committen. Um differenzierter, nicht nur Dateien sondern auch einzelne Abschnitte oder Zeilen zu stagen 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 der Stage Area hinzuzufügen [1].   
ABBILDUNG???

In GitKraken können im File View über die GUI einzelne Abschnitte oder Zeilen einer Datei gestaged werden.

ABBILDUNG??

Für dauerhafte oder wiederkehrende nicht versionsverwaltete Dateien in der Arbeitskopie ist es hilfreich eine ***.gitignore*** Datei anzulegen.

Gitignore

Benutzt man Dateien, welche explizit nicht in die Versionsverwaltung inkludiert werden sollen, kann ***.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* sowie das gebaute Programm ausgeblendet. 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. In GitKraken kann eine Datei oder Ordner in der Änderungsansicht durch ein Rechtsklick ignoriert werden, beziehungsweise *.gitignore* hinzugefügt werden, siehe ABBILDUNG.

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 Pfade 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 ***–ammend***. 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.

Pull

Der Befehl ***git pull*** wird benötigt um das lokale repository mit (falls vorhandenen) Änderungen von *origin* upzudaten*.* Erweitert mit –all werde alle Pfade 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 aussführen, wenn man eine frische Arbeitskopie besitzt, beziehungsweise, alle Änderungen commited hat*.* Will man sich nur über Neuerungen informieren ohne Änderungen local vorzunehmen, sollte *fetch* nutzen.

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. Beide Befehle *fetch*  und *pull* haben zum Großteil die gleiche Syntax um einzelne Pfade 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 Informaton 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 verwenden einen sogenannten Drei-Wege-Merge. Merge wird verwendet um Pfade 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 frühstete 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 Zielpfade 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 Zielpfades 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 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.

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 Pfade 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 Pfade 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 Pfades zu verändern. Nehme man an, man wolle ein Entwicklungspfad, welcher seit längerer Zeit besteht, nun implementieren, aber der Hauptentwicklungspfad ist schon weit vorangeschritten. Damit es nun übersichtlich bleibt, will man den Entwicklungspfad vorschieben, damit der Vorfahre der Abzweigung der neuste Commit des Hauptentwicklungspfad ist. Dies geschieht mit dem Kommando ***git rebase maindev dev*** oder, falls man im Entwicklungspfad 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 Pfade zu vereinen muss man nur noch einen Schritt weiter gehen, indem man nun umgekehrt mit ***git checkout maindev*** auf den *maindev* Pfad 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 Pfade nach Beendigung per *rebase* Hauptpfaden zugeführt werden, sind die Pfade 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 Pfaden *rebasen* um Konflikte zu vermeiden, bei dem andere Branches zum Beispiel auf veraltete Commits verweisen, da der Branch umgeschrieben wurde. Auch hier sollte eine gesunde Mischung aus *rebase* und *merge* verwendet werden um eine schlanke, übersichtliche Historie zu erhalten ohne klare Strukturen und Abgrenzungen zu verlieren. Ein weiteres 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. 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 benutzt um die Baumstruktur übersichtlich zu halten oder Bugfix-Commits für Commits verschwinden zu lassen.

Git branch [12]

*Branches* werden angewendet, um isolierte Entwicklungspfade 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.

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 sein eigenes übernommen werden. Verwendet wird das Kommando mit ***git clone <remoteAdresse> <lokalesVerzeichnis>***.

Git remote

Git fetch

Git patch

Git config -> user alias uvm. <https://blog.seibert-media.net/blog/2015/01/06/tutorial-git-aufsetzen-teil-3-git-config/>