Levels zur Veröffentlichung

[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

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

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.

[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. Dieser Ordner enthält alle wichtigen Informaton zum Repository. Der Ordner alleine reicht 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.

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

Git rebase

Git branch

Git checkout

Git clone

Git remote