Versionsverwaltung mit Git

Danyel Coban 343803
Karim Abou Zeid 354656

2. Juni 2017

1 Einleitung

Die folgende Ausarbeitung gibt dem Leser eine übersichtliche Einführung in die Versionsverwaltung mit Git, welche sich auf die grundlgenden Konzepte und Funktionalitäten beschränkt. Es wird veranschaulicht, wie Git ein Arbeitsverzeichnis verwaltet und sich im Vergleich zu anderen Versionskontrollsystemen verhält.

Zunächst erläutern wir kurz was genau ein Versionskontrollsystem ist, daraufhin gehen wir auf das lokale Arbeiten mit Git ein und betrachten anschließend wie wir distribuiert mit Git arbeiten können. Anschließend veranschaulichen wir das Konzept und die Umsetzung von Branches.

2 Versionskontrollsysteme

2.1 Was ist ein Versionskontrollsystem?

Ein Versionskontrollsystem protokolliert Änderungen an Dateien. Wir nennen den Ordner, der diese Dateien beinhaltet, das *Arbeitsverzeichnis*. Für alle Dateien im Arbeitsverzeichnis wird ein Verlauf des Zustands über die Zeit hinweg verwaltet. Üblicherweise handelt es sich bei den versionierten Dateien um Quellcode, es ist jedoch prinzipiell möglich beliebige Arten von Dateien zu versionieren.

2.2 Wozu ein Versionskontrollsystem?

Das Versionskontrollsystem erlaubt es uns, unser Arbeitsverzeichnis nach Belieben zu einem früheren Zeitpunkt zurückzusetzen. Dies ist auch für einzelne Dateien oder Änderungen möglich. Wir können also an den versionierten Dateien arbeiten, ohne einen Datenverlust befürchten zu müssen und können diese gegebenenfalls zum Zustand eines früheren Zeitpunkts wiederherstellen.

3 Git Grundlagen

Im Folgenden führen wir knapp, die grundlegensten Funktionalitäten ein, die uns Git bereitstellt, um einen Verzeichnis zu verwalten.

Init Mit dem Befehl `git init` legen wir ein neues Git *Repository* im aktuellen Arbeitsverzeichnis an. Dabei wird das `.git` Verzeichnis innerhalb des Arbeitsverzeichnisses erstellt in dem sich die Git interne Datenbank befindet. Repository ist die Bezeichnung für das Verzeichnis, welches das Arbeitsverzeichnis und die Datenbank enthält.

Add Wir verwenden den Befehl `git add <dateiname>`, um eine Datei für den nächsten *Commit* zu *stagen*, das heisst für den nächsten Commit vorzumerken.

Commit Nachdem wir Modifikationen an Dateien im Arbeitsverzeichnis gemacht haben und an einem Punkt angelangt sind, an dem wir diesen Zustand festhalten möchten, erstellen wir einen Commit. Ein Commit ist ein *Snapshot* des Arbeitsverzeichnisses, verknüpft mit dem Namen und der E-Mail Adresse des Autors, sowie einer *Commitmessage*, welche kurz Aufschluss über die gemachten Änderungen geben soll. Bei einem Snapshot handelt es sich im Grunde um eine Kopie des Arbeitsverzeichnisses, jedoch wird auf Dateien, die sich seit dem letzten Snapshot nicht verändert haben, nur verlinkt. Mit dem Befehl `git commit` wird ein neuer Commit angelegt, der alle vorher gestageten Dateien enthält und wir werden aufgefordert, eine Commitmessage zu verfassen.

Status Den Status der von Git versionierten Dateien im Arbeitsverzeichnis können wir mit dem Befehl `git status` einsehen. Dabei werden uns Dateien angezeigt die sich seit dem letzten Commit verändert haben, gelöscht wurden oder neu hinzugekommen sind. Außerdem sehen wir welche Dateien bereits gestaget sind.

Log Mit dem Befehl `git log` wird eine Liste aller Commits sowie deren Autor und Commitmessage ausgegeben.

Checkout Durch den Befehl `git checkout <commit>` wird das Arbeitsverzeichnis zum Zustand des entsprechenden Commits zurückgesetzt. Dies geht sehr schnell, da die Dateien nur aus der Datenbank kopiert und nicht wie bei anderen Versionskontrollsystemen üblich rekonstruiert werden müssen.

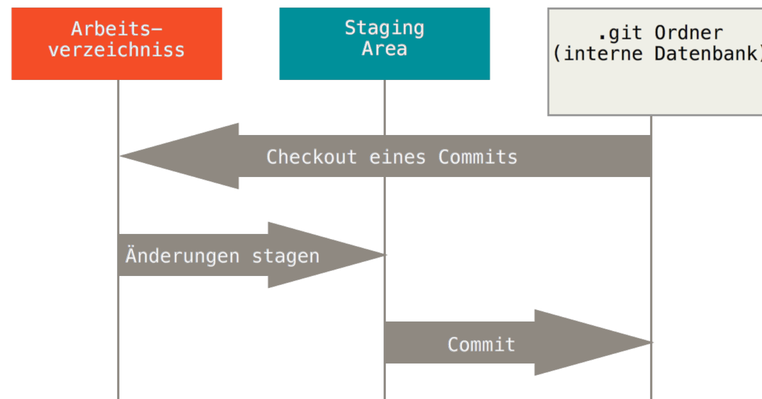

Abbildung 1: Arbeitsfluss

4 Distribuiertes Arbeiten

Git eignet sich hervorragent dazu mit mehreren Autoren an einem Projekt zu arbeiten.

4.1 Git Grundlagen für distribuiertes Arbeiten

Im Folgenden betrachten wir die grundlegenden Befehle um mit einem geteilten Repository zu arbeiten.

Clone Jeder Autor hat immer eine Kopie des gesamten Repository lokal verfügbar. Mit dem Befehl *git clone* wird diese vom Server heruntergeladen.

Push Nachdem Änderungen vorgenommen und committet worden sind, können diese mit dem Befehl `git push` auf den Server gepusht, also hochgeladen, werden.

Pull Mit dem Befehl *git pull* aktualisieren wir die lokale Repository, in dem wir die neuen Commits die sich auf dem Repository des Servers befinden einbinden. Das Einbinden geschieht durch einen sogenannten *Merge*, wir gehen im nächsten Kapitel genauer darauf ein.

4.2 Git Hoster

Ein Git Repository kann prinzipiell auf jedem Server gehostet werden. Es gibt allerdings auch Online-Dienste, die sich darauf spezialisiert haben Git Repositories zu hosten. Der bekannteste ist wohl *GitHub*, aber auch die RWTH Aachen stellt ihren eigenen Git Server zur Verfügung.

Oft bieten diese Git Hoster eine Weboberfläche an, in der das Repository betrachtet werden kann. Zusätzlich gibt es dort meist einen Bereich für Statistiken. Auch ergänzende

Features wie zum Beispiel ein Ticket System, also ein Bereich in dem Aufgaben und Probleme festgehalten werden, oder ein Wiki sind normalerweise vorhanden.

5 Branching

Git unterstützt, wie die meisten anderen Versionskontrollsysteme auch, das sogenannte *Branching* [1]. Das heisst wir zweigen von der Hauptentwicklungslinie ab und arbeiten unabhängig von dieser weiter. Ein Entwicklungszweig heisst *Branch*, auch die Hauptentwicklungslinie ist ein Branch. Jeder Branch ist unabhängig von den anderen Branches, das heisst insbesondere, dass alle Commits die wir in einem Branch machen nur in dieses eingehen.

In vielen anderen Versionskontrollsystemen ist das Branching ein aufwändiges Unterfangen da es zum Beispiel realisiert wird indem das gesamte Arbeitsverzeichnis kopiert wird. Git verwaltet Branches auf eine sehr ressourcenschonende Art und Weise, das Konzept ist tief in Git verwurzelt. Dies hat zur Folge, dass das Wechseln zwischen Branches sowie andere Branch spezifische Operationen schnell ausgeführt werden können. Im Gegensatz zu anderen Versionskontrollsystemen ist bei Git eine Arbeitsweise mit häufigem Branch Wechsel vorgesehen.

5.1 Aufbau eines Branches

Um ohne Verwirrung mit Branches zu arbeiten sollten wir verstehen wie diese Aufgebaut sind und von Git verwaltet werden. Dazu betrachten wir zunächst wie Commits intern aufgebaut sind. Ein Commit beinhaltet Metadaten wie Autor und Commitmessage, einen Zeiger auf den zugehörigen Snapshot sowie einen Zeiger auf den vorherigen Commit. Der initiale Commit ist ein Sonderfall und hat keinen Vorgänger.

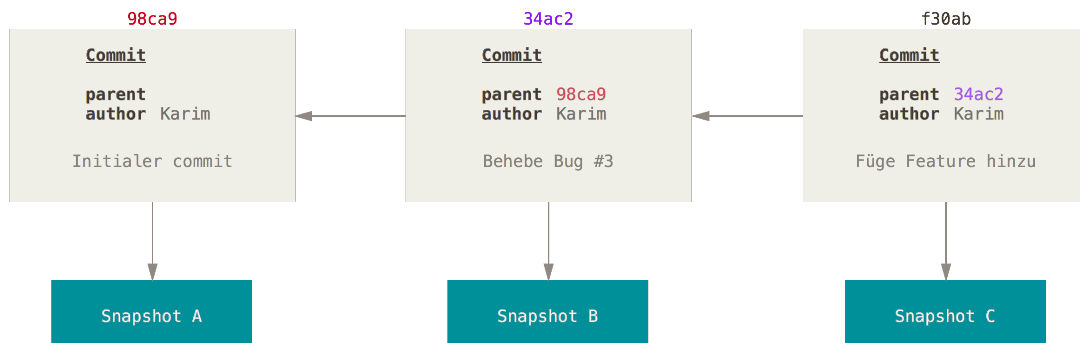

Abbildung 2: Commit Aufbau

Das ist schon alles was wir über Commits wissen müssen um zu verstehen wie Branches verwaltet werden. Ein Branch ist einfach ein Zeiger auf einen bestimmten Commit. Beim initialisieren der Git Repository wird automatisch ein Branch mit dem Namen `master` erstellt.

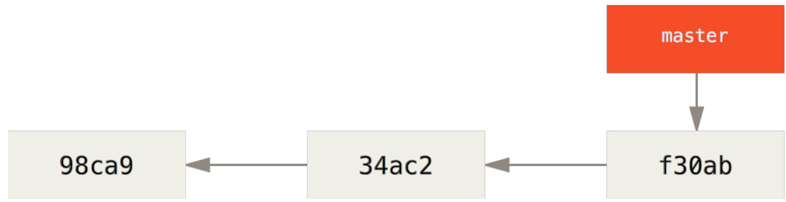

Abbildung 3: Ein Branch

Mit jedem Commit wird der Branch welchen wir gerade verwenden um eins weiter gerückt und zeigt dann auf den neuen Commit. Mit dem befehl `git branch` können wir einen neuen Branch erstellen. Dabei wird einfach ein Zeiger mit dem Namen unseres neuen Branches auf den Commit auf dem wir aktuell arbeiten angelegt.

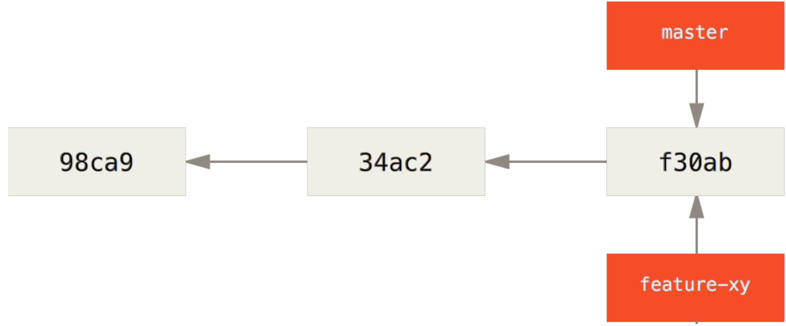

Abbildung 4: Neuer Branch mit dem Namen „“

Git merkt sich durch einen weiteren Zeiger, den `HEAD`, welchen Branch wir aktuell verwenden.

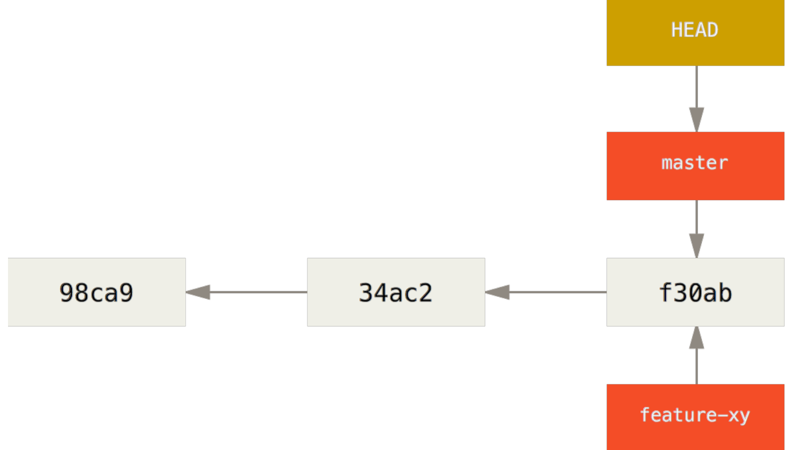

Abbildung 5: Der `HEAD` zeigt auf den `master` Branch

Durch das Erstellen eines Branches wird nur der Zeiger angelegt, wir wechseln nicht

automatisch zu dem neuen Branch. Um zu einem Branch zu wechseln verwenden wir den Befehl `git checkout <branchname>`. Dabei wird der `HEAD` auf den neuen Branch gesetzt und das Arbeitsverzeichnis zum Zustand des Commits auf den der Branch zeigt zurückversetzt, also ein checkout auf diesen Commit ausgeführt.

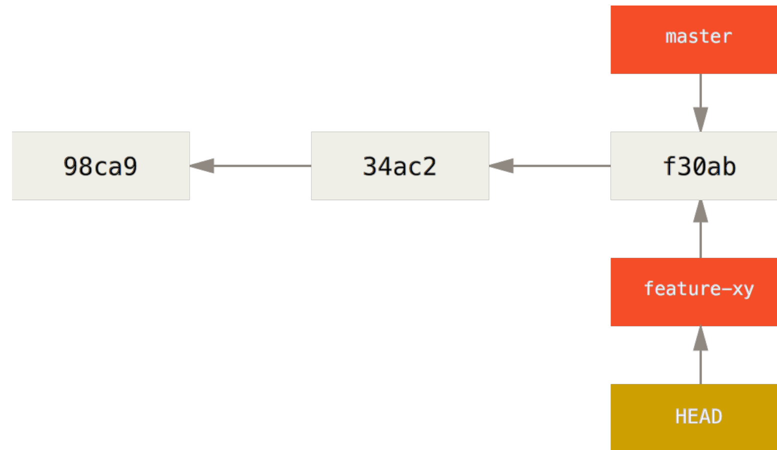

Abbildung 6: Der `HEAD` zeigt auf den Branch `feature-xy`

5.2 Zusammenführen von Branches (Mergen)

Irgendwann kommen wir wahrscheinlich zu dem Zeitpunkt an dem wir zwei Branches wieder zusammenführen möchten. Dies ist nicht unbedingt ohne weiteres möglich, da die Commits der beiden Branches nicht einfach aneinander gehangen werden können. Außerdem kann es sein, dass es einen Konflikt gibt, da in beiden Branches die selbe Zeile geändert wurde. Um Branches zusammenzuführen gibt es zwei Optionen, *mergen* und *rebasen*, auf die wir nun näher eingehen.

5.2.1 Merge

Das Mergen unterscheidet sich je nach Situation.

Fast-Forward Falls Git von dem Branch aus dem wir mergen wollen, durch Verfolgen der Commits den Branch in den wir mergen wollen erreichen kann, führt es einen Merge durch Fast-Forward aus, das heisst es verschiebt einfach den Branch Zeiger auf den neuen Commit.

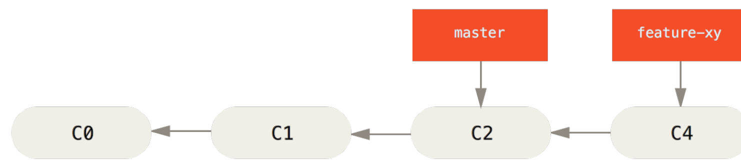

Abbildung 7: Vor dem Merge des Branches feature-xy in den master Branch

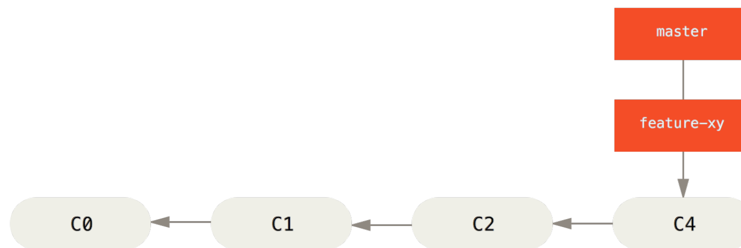

Abbildung 8: Nach dem Merge durch Fast-Forward

3-Way-Merge Sollte kein Merge durch Fast-Forward möglich sein, das heisst der Branch in den wir mergen wollen enthält Commits die in dem Branch aus dem wir mergen wollen nicht enthalten sind, wird es etwas komplizierter. Um solche Branches zusammenzuführen erstellt Git einen neuen Commit welcher die Änderungen in beiden Branches seit ihrem letzten Gemeinsamen Vorgänger in einem neuen Snapshot zusammenführt. Dieser Commit heisst *Merge Commit*. Das Vorgehen nennt sich *3-Way-Merge*, da drei Commits, nämlich die letzten Commits der beiden Branches, sowie deren letzter gemeinsamer Vorgänger für den Merge in betracht gezogen werden. Der merge Commit verweist als Vorgänger auf die letzten Commits der beiden Branches, dies ist ein Sonderfall da Commits normalerweise nur einen Vorgänger haben.

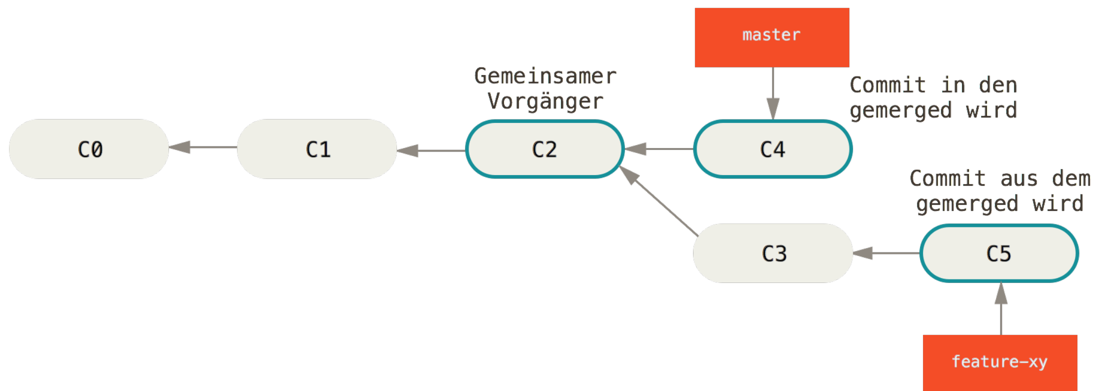

Abbildung 9: Vor dem 3-Wege-Merge

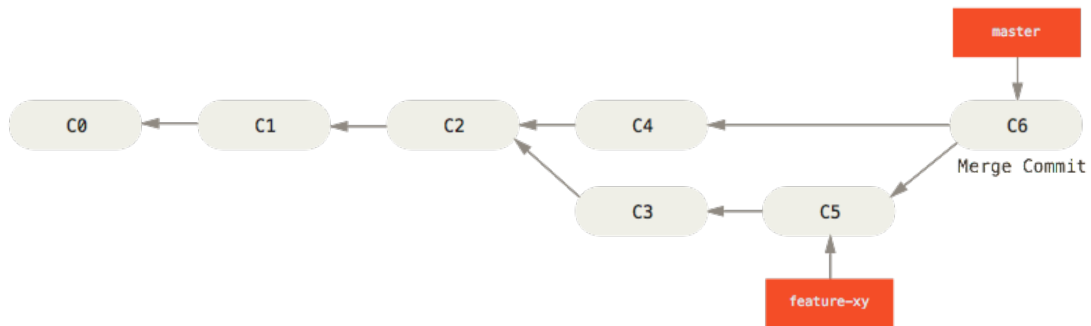

Abbildung 10: Nach dem 3-Wege-Merge

Falls beim Mergen Konflikte auftreten, müssen wir diese selbst lösen in dem wir entscheiden welche der beiden Änderungen wir übernehmen möchten.

5.2.2 Rebase

Ein 3-Way-Merge erzeugt immer einen Merge Commit, durch einen Rebase können wir diesen jedoch umgehen und unseren Verlauf somit etwas aufgeräumter lassen. Dazu wendet Git die Änderungen der Commits aus dem Branch den wir rebasen wollen erneut auf den Branch auf den wir rebasen wollen an. Falls dies konfliktlos möglich ist führt Git danach einfach einen Fast-Forwad Merge aus, ansonsten müssen wir den Konflikt vorher noch manuell lösen.

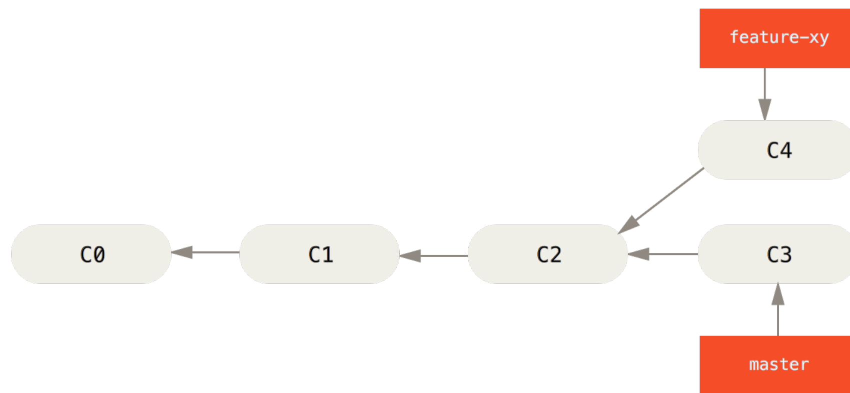

Abbildung 11: Vor dem Rebase

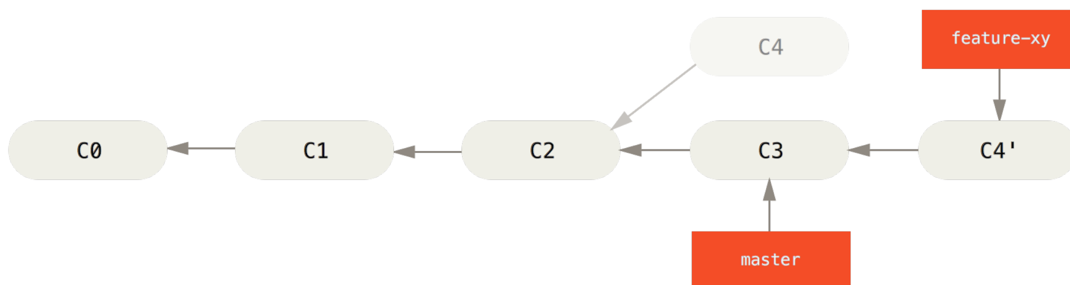

Abbildung 12: `C4` wird erneut auf `C3` angewandt und für das Resultat wird der neue Commit `C4'` erstellt. Danach wird ein Merge durch Fast-Forward durchgeführt. `C4` wird daraufhin verworfen.

Dabei ist zu beachten, dass damit die Commits aus der Branch die wir rebasen neu geschrieben werden. Das bedeuted insbesondere, dass wir keine Branches rebasen sollten falls damit Commits die schoneinmal gepusht wurden neu geschrieben werden. Möglicherweise haben andere Autoren bereits einen Branch an einem dieser Commits erstellt und wir referenzieren nichtmehr zur selben Instanz dieses Commits. Dadurch kann großes Chaos im Repository entstehen.

5.3 Anwendungsbeispiel für Branching

Kommen wir nun zu einem Beispiel wie die Arbeitsweise mit Branches aussehen kann. Wir haben auf dem master Branch unseren Produktionscode. Der Produktionscode sollte immer ausführbar sein und ist optimalerweise bereits getestet. Wenn wir anfangen an einem neuen Feature zu arbeiten, dessen Fertigstellung voraussichtlich etwas länger dauert, legen wir dafür einen neuen Branch an. Dies hat den simplen Grund, dass der neue Code wahrscheinlich nicht bereits beim ersten Commit fertig und getestet ist und somit nicht

auf den master Branch committet werden sollte. Falls wir nämlich in Zukunft merken, dass wir einen kritischen Fehler beheben müssen, wechseln wir einfach zurück in den master Branch. Dort nehmen wir die nötigen Änderungen an unserem Produktionsode vor. Danach wechseln wir wieder in unseren Feature Branch und können weiter arbeiten. Falls wir an einem weiteren Feature arbeiten möchten, erstellen wir für dieses Feature ebenfalls einen Branch. Nun können wir Parallel und vollkommen unabhängig an mehreren Features gleichzeitig arbeiten. Erst wenn ein Feature fertiggestellt und getestet ist mergen wir es in den master Branch.

6 Fazit

Größere Projekte sind ohne ein Versionskontrollsystem kaum denkbar. Git eignet sich dabei im Vergleich zu anderen Versionskontrollsystemen wie zum Beispiel Subversion (SVN) besonders gut, da es auch ohne Netzwerkverbindung den vollen Funktionsumfang bietet und vergleichsweise schnell ist. Microsoft hat ihr Betriebssytem Windows erst vor kurzem zu Git migriert. Das Repository ist ungefähr 300GB groß, was stark für die Skalierbarkeit von Git spricht. Aber auch für kleinere Projekte, wie beispielsweise dieses Paper bringt es einen hohen Mehrwert mit sich. Das ganze bekommen wir für relativ wenig Mehraufwand. Selbst für nicht kollaborative Arbeit empfiehlt es sich sehr ein Projekt mit Git zu verwalten.

Literatur

[1] Scott Chacon. *Pro Git.* Apress, Berkely, CA, USA, 1st edition, 2009.