GFS Ausarbeitung
Thema: Versionsverwaltung mit git
Gauß-Gymnasium Hockenheim
Informatik J1.2 bei Herr Wunderlich
Abgabedatum: 30.03.2014
Lukas Bach

VERSIONSVERWALTUNG MIT GIT

Themen

1 Grundprinzip

Dateiversionierung ist in der Softwareentwicklung meistens ein Problem, vor allem wenn man in Gruppen arbeitet. Nehmen wir mal an, eine kleine Gruppe Entwickler arbeitet an einem Projekt. Zunächst müssen die Dateien zwischen den Personen ständig ausgetauscht werden, da ständig Änderungen vorgenommen werden, aber selbst wenn man einen Cloud-Hoster zur Synchronisierung der Dateien verwendet, trifft man schnell auf die nächsten Probleme. Wenn eine Person an einer Datei arbeitet, an der jemand anderes Änderungen vornehmen muss, so wundert sich diese schnell, warum der eigene Code nichtmehr funktioniert und warum plötzlich Änderungen vorgenommen wurden. Und wenn dann mehrere Personen anfangen, nicht nur in derselben Datei, sondern im selben Codesegment zu arbeiten, wird es sowieso chaotisch.

Aus diesem Grund wird in der Softwareentwicklung meist sogenanntes *Versionsverwaltungssystem*, abgekürzt VCS (*Version Control System*) verwendet [1]. Dieses verlangt von den Benutzern, ihre Änderungen zu gruppieren und kommentieren, sodass andere den Überblick über Änderungen behalten, und legt eine Versionsgeschichte an, die sämtliche Änderungen jeglicher Dateien protokolliert. Auf diese Weise dient es auch als Backup-Programm, da alle Änderungen rückgängig gemacht werden können.

Versionsverwaltungssysteme können nicht nur zur Versionierung von Code-Projekten verwendet werden, sondern können zur Verwaltung von beliebigen Dateien oder Dateigruppen dienen, somit also auch Textautoren, Grafikern oder sonstigen mit Dateien arbeitenden Personen helfen. Die meisten VCS sind allerdings auf die Versionierung von textbasierten Dokumenten, i.d.R. Quellcodes, ausgelegt.

Das VCS erfüllt gleich mehrere Zwecke. Da jede Änderung einer Datei mit einem Kommentar der Person, die die Änderung durchgeführt hat, versehen werden muss, lässt sich auch im Nachhinein der Grund jeder Änderung nachvollziehen. Sollten mehrere Personen an dem Projekt arbeiten, lässt sich auch herausfinden, wer die Änderung gemacht hat, sodass man sich im Zweifelsfall an die richtigen Personen wenden kann.

Gleichzeitig lassen sich Änderungen auch jederzeit rückgängig machen, da alle Versionen sämtlicher Dateien vor und nach jeden Änderungen im archivierten Zustand vorliegen und jederzeit wiederhergestellt werden können.

Ein weiterer wichtiger Grund der Nutzung eines VCS ist die vereinfachte Zusammenarbeit in Teams an demselben Projekt. Da alle Dateien des Projekts entweder an einem zentralen Ort gespeichert sind oder automatisch zwischen den Mitarbeitern synchronisiert werden, verfügt

jeder immer über die aktuelle Version und sämtliche Archive des Projekts. Dadurch kann jeder Dateien bearbeiten und bekommt die Aktualisierungen der anderen automatisch mit. Sogar die Bearbeitung derselben Datei von unterschiedlichen Personen ist dabei durch verschiedene Prinzipien möglich.

Man unterscheidet grundlegend zwischen drei verschiedenen Arten der Versionsverwaltung [2]:

- *Lokale Versionsverwaltung*: Solche Systeme sind nur lokal auf dem Computer des Anwenders vorhanden und verwalten oft nur ein einzige Datei. Sie sind meist in einfachen Programmen integriert und verwalten im Hintergrund automatisch verschiedene Versionen. Vor allem bei solchen finden allerdings keine aktiven Handlungen des Benutzers statt.
- *Zentrale Versionsverwaltung*: Hier wird das gesamte Projekt und seine Versionsgeschichte auf einem Server gespeichert, auf den alle Mitbearbeiter zugreifen können. Wenn mehrere Personen dieselbe Datei bearbeiten wollen, können verschiedene Konzepte angewandt werden, um dies zu ermöglichen. *(vgl. Abschnitt 5)*
- *Verteilte Versionsverwaltung*: Bei dieser verfügt jeder Benutzer über eine lokale Kopie des Projekts und der Versionsgeschichte. Wenn Änderungen vorgenommen werden, werden die geänderten Dateien mit den anderen Projektkopien auf den Rechnern anderer Benutzer synchronisiert und eventuell mit anderen Änderungen in neue Dateien zusammengeführt. Dies bietet den Vorteil, dass der Benutzer keine permanente Netzwerkverbindung zum Projekt-Hoster haben muss, sondern auch offline an den Dateien arbeiten kann. Die Änderungen können dann aktualisiert werden, wenn erneut eine Verbindung zustande kommt.

2 Git als Versionsverwaltungssoftware

> *"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'."*
>
> – Linus Torvalds [3]

Eine der am häufigsten genutzten Versionsverwaltungssysteme ist *Git* (engl. Blödmann, siehe oben), welches von Linus Torvalds zur Codeverwaltung des von ihm gestarteten Projekts Linux entwickelt wurde und als solches Bitkeeper [4] im Jahr 2005 ablöste. Es handelt sich dabei um ein verteiltes VCS, welches als Open Source frei verfügbar ist und meistens in Form einer Konsolenanwendung verwendet wird. Es gibt verschiedene grafische Benutzeroberflächen und teils auch vollständige Feature-Integration in *IDEs* (Entwicklungsumgebungen), im Folgenden wird allerdings primär die Anwendung einer Konsole betrachtet.

3 Allgemeine Funktionsweise

3.1 Repositorys

Bei den vielen Fachbegriffen, die in der Versionsverwaltung anzutreffen sind, ist das häufigste „*Repository*“ (engl. Lager, Depot) [5]. VCS dienen zur Versionierung und Verwaltung von Dateien, und ein Repository, umgangssprachlich auch Repo genannt, bezeichnet die Dateien in dem Projekt. Ein Repository enthält zwei wichtige Elemente: Das Arbeitsverzeichnis sowie den .git Ordner. Im Arbeitsverzeichnis ist eine lokale Kopie des aktuellen Projektes abgespeichert. Der Benutzer kann dort seine Änderungen und Bearbeitungen vornehmen. Die anderen Mitarbeiter bekommen davon zunächst nichts mit, da diese Änderungen nur im lokalen Arbeitsverzeichnis vorgenommen werden. Der .git Ordner dagegen enthält Metadaten zum Projekt sowie die gesamte Versionsgeschichte. Sobald der Benutzer mit seinen Änderungen fertig ist, wird die neue Version der Datei vom Arbeitsverzeichnis in die Dateigeschichte übernommen, die in diesem .git Ordner gesichert wird. Dieser Ordner wird dann mit den anderen Repositorys, die auf den Rechnern von anderen Mitarbeitern gespeichert werden, abgeglichen (vgl. Abschnitt 3.4 und 5).

Ein neues Repository lässt sich in Git einfach über den Befehl `git init` erstellen [6]. Das erzeugt ein neues und vollkommen leeres lokales Repository im aktuellen Verzeichnis. Git wendet durch die Konsolenbefehle dann automatisch Änderungen und Aktualisierungen auf die darin befindlichen Dateien an.

3.2 Bare bzw. zentrale Repositorys

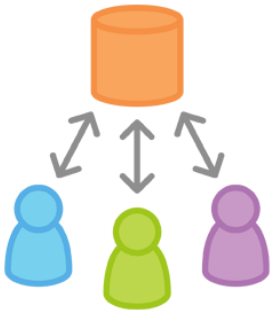

Abb. 1: Zentrales Repository

Während alle mit `git init` erstellen Repositorys sowohl ein lokales Arbeitsverzeichnis als auch die Versionsgeschichte des Projekts beinhalten, lassen sich auch „*Bare Repositorys*“ (engl. Bares/nacktes Repository) [7] erstellen, die nur die Versionsgeschichte sichern. Dazu verwendet man den Befehl `git init –bare`. Solche Bare Repositorys werden als zentrale Projekt-Hosts verwendet, mit dem sich alle Repository-Kopien sämtlicher Mitarbeiter synchronisieren (s. Abb. 1). Jeder, der Änderungen in seinem lokalen Arbeitsverzeichnis vornimmt und diese in seine lokale Kopie des Repositorys übernimmt, kann diese dann auf das zentrale Bare Repository hochladen. Die anderen Mitarbeiter können diese von dort herunterladen.

3.3 Revisionen von Dateien

Die Bezeichnung *Revision* definiert sich im Grund als ein Synonym von „Version“. Sie beschreibt bei VCS die Versionen einzelner Dateien, der Begriff Version selbst beschreibt meist die Version eines ganzen Projektes. Jede Änderung einer Datei lässt sich als Revision dieser bezeichnen, sodass man jeden Zustand einer Datei im Laufe des Projekts individuell ansprechen kann (z.B. Revision 12 in query.h).

3.4 Staging Area und Commits

Bevor eine Revision einer Datei endgültig gespeichert wird, muss sie mehrere Schritte durchlaufen. Sie befindet sich zunächst in der lokalen Arbeitskopie des Benutzers, in der sie von diesem frei bearbeitet werden kann. Während sie bei den meisten anderen VCS danach direkt in das Repository übertragen werden kann, findet sich bei Git zwischen der lokalen Kopie und dem Repository noch eine dritte Stufe: Die *Staging Area* (engl. Bereitstellungsraum) [8]. Geänderte Dateien können über den Konsolenbefehl `git add <datei>` oder `git add <verzeichnis>` zur Staging Area hinzugefügt werden [9]. Dort sind sie noch nur temporär gespeichert und auch nicht in der Versionsgeschichte des Projekts gesichert. Über einen weiteren Befehl, `git commit […]` werden Dateien dann ins Repository „*commitet*“ [10]. Das comittete Dateipacket bzw. die committeten Dateien werden in der Versionsgeschichte unter einem „*Commit*“ (engl. Durchführung, Überstellung) zusammengefasst angespeichert. Mithilfe von zusätzlichen Argumenten lässt sich noch ein Kommentar, genannt „*Commit-Message*“ anhängen, sodass die Änderungen nachvollziehbar in der Versionsgeschichte gespeichert werden können. Die oberen beiden Pfeile in Abb. 2 zeigen das *stagen* der Dateien (hinzufügen zur Staging Area) und das committen in die Versionsgeschichte.

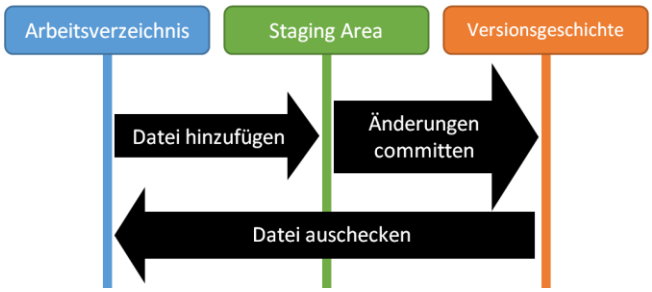

Abb. 2: Arbeitsverzeichnis, Staging Area und Versionsgeschichte

Am sinnvollsten lassen sich Dateien mit `git commit` ohne weitere Parameter committen. Dabei öffnet sich ein Editorfenster, in dem die Commit-Message eingegeben werden kann. Man kann das Kommentar auch direkt an den Befehl anhängen (`git commit –m „<message>“`), dabei kann das Kommentar aber nicht mehrzeilig sein.

Dadurch, dass die Dateien zuerst im „Puffer“ der Staging Area gespeichert und dann in Gruppen comittet werden, kann man viel besser definieren, welche Änderungen in verschiedenen Dateien zusammengehören. So lässt sich beispielsweise, wenn Änderungen in verschiedenen Dateien zur Lösung desselben Problems dienen, diese im Selben Commit in der Versionsgeschichte formulieren.

Von anderen VCS unterscheidet sich Git auch darin, dass mit verschiedenen Revisionen einer Datei nicht nur die Änderungen, sondern die gesamte Datei zu einem Zeitpunkt speichert. Hierbei spricht man auch von einem Snapshot bzw. Schnappschuss. Viele andere VCS speichern bei jeder Revision jeweils nur die Änderungen zu diesem Zeitpunkt, weshalb Git zwar etwas primitiver, dafür aber bei weitem effizienter arbeitet, da zum wiederherstellen einer Datei nicht sämtliche Revisionen der Ursprungsdatei erneut angewandt werden müssen [11].

3.5 Commit-Bezeichnungen

Um einzelne Commits individuell ansprechen zu können, werden sie vom System automatisch benannt. Diese automatische Benennung verwendet nicht eine mit jeder Revision aufsteigende Zahl, da die Versionsgeschichte sich bei git stark verzweigen kann. Stattdessen wird der Inhalt des Repositorys zu einem Zeitpunkt als Benennung verwendet. Dazu wird ein *SHA-1-Checksumme (Secure Hash Algorithm)* [12] aus den Dateien berechnet. Dieser hat eine Länge von 160 bit und wird als eine 40-stellige Hexadezimalzahl angegeben. Aus zwei Versionen mit übereinstimmenden Dateiinhalten wird dieselbe Checksumme berechnet. Sofern zum Beispiel einige Änderungen bzw. Commits einer vergangenen Version rückgängig gemacht werden, sodass die neue Version mit einer älteren übereinstimmt, so sind auch ihre Checksummen identisch. Bei anderen VCS, die aufsteigende Zahlen verwenden, hätten diese Versionen dann trotzdem unterschiedliche Bezeichnungen, obwohl der Inhalt sich gleicht.

Bei allen Befehlen, die eine Commit-Bezeichnung erfordern (in folgenden Code-Definitionen als `<commit>` bezeichnet) lässt sich entweder die komplette Hash-Summe oder nur die ersten sieben hexadezimalen Zahlen (z.B. `68ac43e`) verwenden, da es sehr unwahrscheinlich ist, dass zwei unterschiedliche Hashsummen sich in den ersten vier Zeichen gleichen.

3.6 Tags

In git ist es möglich, wichtige Punkte in der Versionsgeschichte mit einer individuellen Bezeichnung zu markieren, bzw. zu „*taggen*“. Diese Funktion wird in der Regel dann angewandt, wenn man Release-Versionen mit einem Namen versehen will, z.B. mit „v0.1“ oder „v1.3.2.4“. Ein Tag bezeichnet einen Punkt in der Versionsgeschichte und kann somit als Synonym für einen

Hash-Wert *(s. 4.5)* verwendet werden. Auch lässt er sich als Parameter in sämtlichen Befehlen anwenden, die `<commit>` als Argument nutzen.

Einfache Tags lassen sich mit `git tag <versionsname>` erstellen [13]. Dieser neue Tag zeigt dabei auf die aktuelle Version. Alternativ lassen sich auch vergangene Versionen mit `git tag <versionsname> <commit>` taggen. Mit dem Befehl git tag lassen sich dann alle Tags auflisten. Außerdem ist es möglich, mit zusätzlichen Argumenten „*Kommentierte Tags*“ [14] anzulegen. Diese verfügen über eine Signatur, ein Kommentar und zusätzlichen Metadaten.

3.7 Auschecken von Dateien

Wenn Dateien durch Commits zur Versionsgeschichte hinzugefügt werden können, lassen sich alte Commits natürlich auch wiederherstellen. Man spricht vom „*Auschecken*“ der Dateien, im englischen „*Checkout*“. Es lassen sich einzelne Dateien, ganze Commits (also alle in einem Commit modifizierten Dateien) und sogar ganze Entwicklungszweige (vgl. Abschnitt 4.2) auschecken. Das funktioniert über die Befehle `git checkout <commit> <file>` um eine spezifische Datei eines Commits auszuchecken, oder über `git checkout <commit>` um den gesamten Commit auszuchecken [15]. Die Dateien werden dabei ins Arbeitsverzeichnis geladen und dabei auch automatisch gestagt, wo sie weiter bearbeitet und nach Wunsch in die neuste Version committet werden können (s. Abb. 2 unterer Pfeil).

4 Branches

4.1 Allgemeines

Vor allem wenn mehrere Personen an einem Projekt arbeiten, kann es häufig zu Chaos und Unübersichtlichkeit in der Versionsgeschichte kommen. Um den Prozess übersichtlicher zu gestalten, ist es möglich, vom Repository so genannte *Branches* (engl. Zweig) abzuspalten [16]. Neben Branch und Zweig ist auch häufig von einem Entwicklungszweig die Rede. Der Name ist so gewählt, da sich die Versionsgeschichte hier verzweigt bzw. von dem Hauptbranch (genannt „*master*“) abspaltet. Er enthält eine Kopie des Repositorys und der Dateien darin. Nach seiner Erstellung können seine Inhalte unabhängig vom Masterbranch bearbeitet werden. Zweck darin ist, dass man für eine größere Arbeitsaufgabe am Projekt, zum Beispiel den Fix eines Bugs oder das Hinzufügen eines Features, in einem separaten Projekt arbeitet. Dadurch werden die Ursprungsdateien nicht angerührt und bleiben, sofern sie das vorher waren, auch während der Arbeit an einem Branch weiterhin funktionsfähig. Wenn bei der Arbeit an einem Zweig

versehentlich Fehler erzeugt werden, die die Funktionalität des Gesamtprojekts beeinträchtigt, ist dann immer noch eine lauffähige Version im Masterbranch vorhanden. Im Bild (Abb. 3) sind zwei Seitenzweige abgebildet („Little Feature“ und „Big Feature“), die sich vom Masterbranch abzweigen.

Sobald alle dazugehörigen Änderungen vorgenommen, der Branch fertig bearbeitet und das Ziel der Änderungen erfüllt wurden, ohne dass die Änderungen mit den ursprünglichen Dateien im Masterbranch in Konflikt treten, wird der Zweig auf den Masterzweig zurückgeführt und die vorgenommenen Änderungen werden mit den ursprünglich vorhandenen Codeteilen zusammengeführt („*MERGE*“, vgl. Abschnitt 4.3).

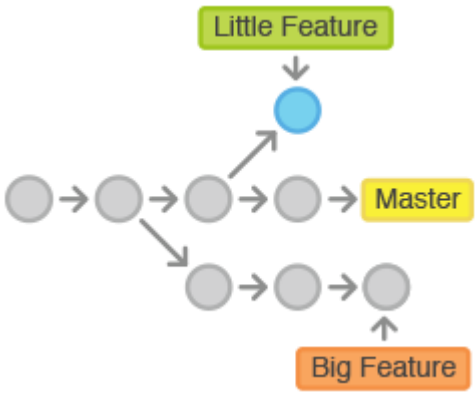

Abb. 3: Branches in der Versionsgeschichte

Neue Branches lassen sich einfach über den Befehl `git branch <branchname>` erstellen [17]. Dabei wird der aktuelle Branch, in dem man sich zurzeit befindet, kopiert und unter dem angegebenen Namen gespeichert.

4.2 Auschecken von Branches

Mit dem `git checkout` Befehl lassen sich nicht nur Dateien und Commits auschecken, sondern auch gesamte Branches. Das bedeutet im Grunde nichts anderes, als zum angegebenen Branch zu wechseln. Wenn das zusätzliche Argument –b verwendet wird (`git checkout -b <branchname>`) lässt sich auch zu einem nicht existierenden Branch wechseln, d.h. der Branch wird erst erstellt mittels einer Kopie des aktuellen Branches, worauf dann zu diesem gewechselt wird.

4.3 Branch-Merging

Wenn ein Branch fertig bearbeitet wurde, also sein Zweck erfüllt wurde, muss er mit dem Masterbranch wieder zusammengeführt werden. Wichtig ist, dass der Branch nicht einfach den Masterzweig überschreiben darf, da es sein kann, dass in diesem in der Zwischenzeit ebenfalls Änderungen vorgenommen worden sind, die erhalten bleiben sollen. Git versucht beim Merging automatisch, die Änderungen so zusammenzuführen, sodass nichts überschrieben wird oder verloren geht. Selbst wenn verschiedene Änderungen in derselben Datei vorgenommen wurden, kann git diese differenzieren und in eine neue Version der Datei bringen.

Rechts (Abb. 4) sieht man, wie eine verzweigte Versionsgeschichte nach einem sogenannten *3-WAY MERGE* [18] wieder zusammenläuft (genannt 3-way, da sowohl die abschließenden Commits der beiden Branches als auch ihr gemeinsamer vorgehender Commit zum Merging verwendet wird). Alternativ gibt es auch den *FAST-FORWARD-MERGE* [19] (engl. Schnellvorlauf), der den aktuellen Branch zum neuen Masterbranch ernennt, ohne tatsächlich alte Änderungen im usprünglichen Masterbranch zu übernehmen.

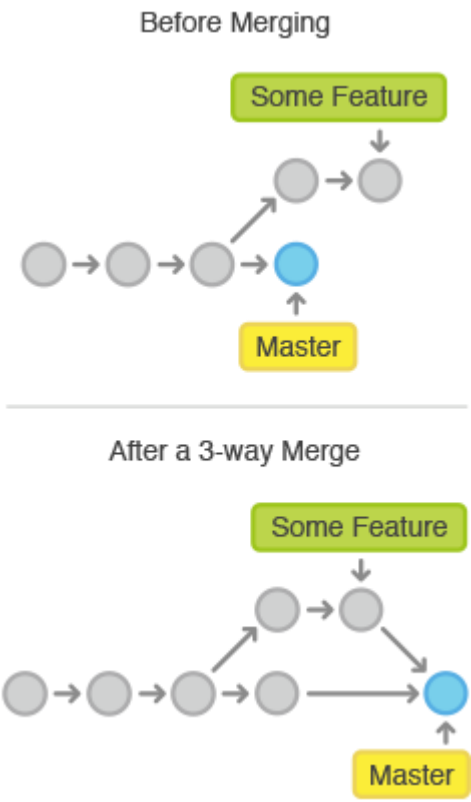

Abb. 4: Versionsgeschichte vor und nach dem Merging

Nur wenn verschiedene Änderungen im selben Codesegment vorgenommen wurden, ist git nicht mehr selbstständig in der Lage, das Problem zu lösen. Es kommt zu einem „*KONFLIKT*“. Git zeigt dann die sich überschneidenden Änderungen auf und bittet den Benutzer, den Code per Hand zusammenzuführen und das Problem selbst zu lösen [20]. Dazu wird meistens das Tool „*MELD*“ [21] verwendet, dass die Unterschiede der Konflikt erzeugenden Dateien aufzeigt und dem Benutzer erlaubt, die zusammengeführte Datei entsprechend zu bearbeiten.

Um das Mergen zu beginnen, reicht der Befehl `git merge <branch>` [22], um den angegebenen mit dem aktuellen Branch zusammenzuführen.

5 Zusammenarbeit mehrerer Repositorys

5.1 Allgemeines

Alle soweit erläuterten Funktionsweisen lassen sich so problemlos anwenden, wenn man als einzige Person an dem Projekt arbeitet. Wenn ein gesamtes Team dabei mitwirkt, gestaltet sich der gesamte Prozess ein bisschen komplizierter. Git ist ein verteiltes VCS und erfordert somit keinen zentralen Rechner, der als Host einer offiziellen Kopie dient, obwohl es möglich und meist auch so angewandt wird. Dennoch bedeutet dass, dass jeder einzelne Rechner, der ein lokales Repository hostet, in der Lage sein muss, diese mit sämtlichen anderen Kopien von jeglichen anderen Mitarbeitern abzugleichen.

5.2 Remote Verbindungen

Damit sich die lokale Arbeitskopie mit den anderen der restlichen Mitarbeiter abgleichen kann, muss jeder über eine Liste der URLs zu den übrigen Repository-Kopien verfügen. Wenn man ein Repository von einem anderen Speicherort herunterlädt, wird die Quelle dessen automatisch als Verbindung abgespeichert. Dies ist mit dem Befehl `git clone <url>` [23] möglich. Dieser Befehl arbeitet sehr ähnlich wie `git init`, er erzeugt ein neues Repository im aktuellen Verzeichnis, kopiert aber automatisch die aktuellen Projektdateien ins Arbeitsverzeichnis, lädt die Versionsgeschichte herunter und speichert den Ursprung als Verbindung. Diese Verbindung erhält den allgemeinen Namen „*ORIGIN*“ [24].

Eine neue Verbindung zu einem weiteren Repository lässt sich über den Befehl `git remote add <name> <url>` [25] erreichen. Dieser Befehl fügt das definierte über Remoteverbindung erreichbare Repository zur lokalen Liste solcher hinzu. Da in der Regel aber zentrale Repositorys eingesetzt werden, ist das meist nicht notwendig, da die Verbindung mit `git clone` automatisch hinzugefügt wird.

Nur weil eine Verbindung zu einem anderen Repository abgespeichert ist, bedeutet das allerdings nicht, dass die Versionsgeschichte automatisch abgeglichen wird. Dies muss über zusätzliche Befehle eingeleitet werden (vgl. folgenden Abschnitt 5.3).

Als URL ist der Verweis zur .git Datei eines Repositorys zu nennen, wobei diese Adresse in der Regel über eine SSH Verbindung läuft, z.B. `ssh://user@host/path/to/repo.git`

5.3 Pulling und Pushing

Mit dem `git remote` Befehl lassen sich die Remote-Verbindungen zwar verwalten, die tatsächliche Abgleichung der Daten muss allerdings erst noch gestartet werden. Dazu gibt es drei Methoden: Fetch, Pull und Push. Fetch lädt alle oder einen spezifischen Branch von einem definierten Remote-Repository herunter und speichert sie als Remote-Branches ab. Das heißt, anstelle sie direkt in die lokalen Branches zu integrieren, mit denen man üblicherweise arbeitet, werden sie als zusätzliche Kopie abgespeichert. Der Benutzer kann dann die dort vorhandenen Commits durchgehen und entscheiden, welche in den lokalen Arbeitszweig übernommen und welche verworfen werden. Um ausgewählte Commits zu übernehmen, kann der Befehl `git merge` benutzt werden, sodass die Änderungen zusammengeführt werden.

Dieser gesamte Prozess gestaltet sich einfacher, wenn man sich zur Nutzung der Methode `git pull <repo>` [26] entscheiden. Er ist eine teilweise automatisierte Version des Fetching. Es wird

der aktuelle Branch aus dem angegebenen Remote-Repository heruntergeladen, allerdings werden sämtliche Änderungen automatisch mit dem merge-Befehl zusammengeführt. Dieses Merging überschreibt nicht einfach die lokalen Dateien mit den aktuellen aus dem Remote-Repository, sondern sorgt auch hier für eine intelligente Zusammenführung. In Abbildung 5 sind im mittleren Bild im zentralen Repository bereits Änderungen vorgenommen worden („Origin/Master“), während der Masterbranch des lokalen Repositorys („Master“) noch einer veralteten Version entspricht. Nach dem Pulling werden die Änderungen wie im unteren Teil auf denselben Stand gebracht.

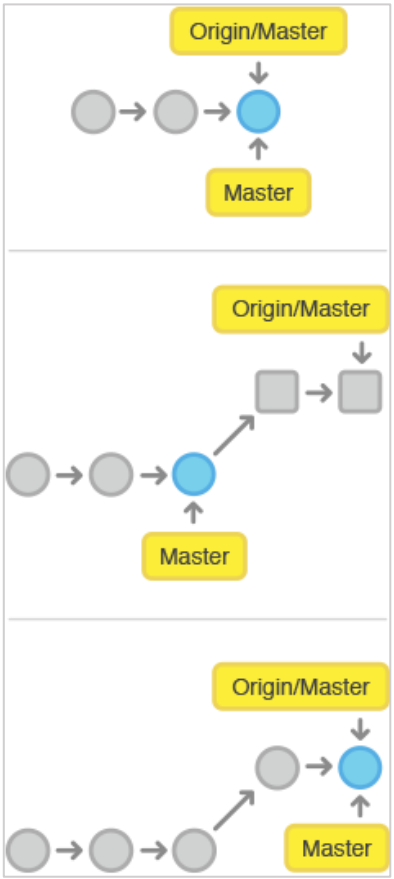

Abb. 5: Versionsgeschichte am Anfang, nach Bearbeitungen im Zentralen Repository und nach Pulling

Ähnlich funktioniert die Methode `git push <repo> <branch>` [27]. Sie lädt keine Änderungen von anderen Repositorys herunter, sondern updatet das angegebene Remote-Repository mit den lokalen Änderungen. Auf diese Weise lassen sich die Arbeitsergebnisse eines Mitarbeiters auf den anderen Repository sichern. Abbildung 6 zeigt hierbei den lokalen Masterbranch, der vor dem Masterbranch des zentralen Repositorys liegt. Durch das Pushen wird das zentrale Repository auf den Stand des lokalen gebracht.

Genauso wie beim mergen von mehreren lokalen Branches können auch hier Konflikte auftreten. Git bittet den Benutzer dann, das Problem manuell zu beheben (vgl. Abschnitt 4.3 unten).

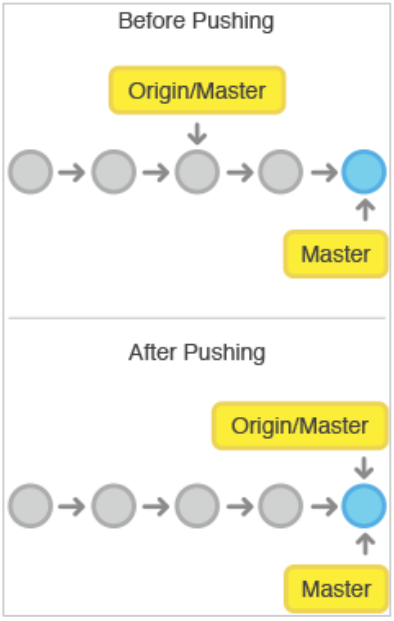

Abb. 6: Versionsgeschichte vor und nach Pushing

6 GitHub als Projekt Hoster

Wenn man ein zentrales Repository verwenden möchte, muss dieses auf einem Server gehostet werden. Entweder installiert man dazu Git auf einem eigenen Server oder mietet einen, der das Hosten von Git-Repositorys anbietet. Der Dienst *GitHub* [28] ist in diesem Bereich bei weitem am populärsten. Er bietet kostenloses Hosten von Open-Source Projekten an und verwendet Git-Repositorys (auch kompatibel zu Subversion-Clients [29]). Gegen Bezahlung lassen sich auch private Repositorys hosten, die nur von den Mitbearbeitern eingesehen werden können [30]. Die meisten Open-Source Programme werden dort gehostet und regelmäßig geupdatet. Des Weiteren bietet GitHub viele zusätzliche Funktionen wie einen Bugtracker, Projektbezogene Wikis, einem Desktop-Client für Entwickler und weitere.

7 Literaturverzeichnis

7.1 Textverweise

[1] *Wikipedia (DE):* Versionsverwaltung. http://de.wikipedia.org/wiki/Versionsverwaltung
Praegnanz.de: Versionsverwaltung. http://praegnanz.de/weblog/versionsmanagement-fuer-anfaenger

[2] *Wikipedia (DE):* Verschiedene Arten der Versionsverwaltung. http://de.wikipedia.org/wiki/Versionsverwaltung#Funktionsweise

[3] *Git Wiki:* Zitat von Linus Torvalds. https://git.wiki.kernel.org/index.php/GitFaq#Why_the_.27Git.27_name.3F

[4] *Wikipedia (DE):* Git als Ersatz von Bitkeeper. http://de.wikipedia.org/wiki/Git#Geschichte

[5] *Wikipedia (EN):* Repository. http://en.wikipedia.org/wiki/Repository_(version_control)

[6] *Git Homepage:* Anwendung von git init. http://git-scm.com/docs/git-init

[7] *Saintsjd.com:* Bare Repository. http://www.saintsjd.com/2011/01/what-is-a-bare-git-repository/

[8] *GitReady.com:* Staging Area. http://de.gitready.com/beginner/2009/01/18/the-staging-area.html

[9] *Atlassian.com:* Referenz zu git add. http://de.gitready.com/beginner/2009/01/18/the-staging-area.html
Git Homepage: Referenz zu git add. http://git-scm.com/docs/git-add

[10] *Producingoss.com:* Definition commit. http://producingoss.com/en/vc.html#vc-vocabulary-commit
Atlassian.com: Referenz zu git commit. https://www.atlassian.com/de/git/tutorial/git-basics#!commit
Git Homepage: Referenz zu git commit. http://git-scm.com/docs/git-commit

[11] *Marcel Eichner:* Geschwindigkeitsvorteile von git. http://de.slideshare.net/ephigenia1/git-praktische-einfhrung-13308756 auf der 5. Folie

[12] *Wikipedia (DE):* SHA-1. http://de.wikipedia.org/wiki/Secure_Hash_Algorithm

[13] *Git Homepage:* Referenz zu git tag. http://git-scm.com/docs/git-tag
Githowto.com: Referenz zu git tag. http://git-scm.com/book/de/Git-Grundlagen-Tags

[14] *Git Homepage:* Kommentierte Tags. http://git-scm.com/book/de/Git-Grundlagen-Tags#Kommentierte-Tags

[15] *Atlassian.com:* Auschecken von Dateien. https://www.atlassian.com/de/git/tutorial/undoing-changes#!checkout
Atlassian.com: Auschecken von Branches. https://www.atlassian.com/de/git/tutorial/git-branches#!checkout
Git Homepage: Referenz zu git checkout. http://git-scm.com/docs/git-checkout

[16] *Johner.org:* Branches. http://www.johner.org/tech-docs/softwareentwicklung/management/versionsmanagement/ (weiter unten)

[17] *Atlassian.com:* Referenz zu git branch. https://www.atlassian.com/de/git/tutorial/git-branches#!branch
Git Homepage: Referenz zu git branch. http://git-scm.com/docs/git-branch

[18] *Git Homepage:* 3-way Merge. http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging#Basic-Merging
Atlassian.com: Gegenüberstellung von 3-way Merge und Fast-Forward Merge. https://www.atlassian.com/git/tutorial/git-branches#!merge

[19] *Git Homepage:* Fast-Forward Merge. http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging#Basic-Branching
(Siehe Atlassian.com Link bei [18])

[20] *Git Homepage:* Konflikt manuell lösen mithilfe von git mergetool. http://git-scm.com/docs/git-mergetool

[21] *Deadlypenguin.com:* Verwendung von Meld.
http://blog.deadlypenguin.com/blog/2011/05/03/using-meld-with-git-diff/
Meld Projektseite: Projektseite. http://meldmerge.org/

[22] *Atlassian.com:* Referenz zu git merge. https://www.atlassian.com/de/git/tutorial/git-branches#!merge

[23] *Atlassian.com:* Referenz zu git clone. https://www.atlassian.com/de/git/tutorial/git-basics#!clone

[24] *Stackoverflow:* Remote Verbindung als origin hinzufügen.
http://stackoverflow.com/questions/7259535/setting-up-a-remote-origin
Git Homepage: Remoteverbindungen (auch Origin). http://git-scm.com/book/ch2-5.html

[25] *Effectivetrainings.de:* Remote Repositorys.
http://www.effectivetrainings.de/blog/2010/01/18/remote-repositories-mit-git-einrichten/
Git Homepage: Remoteverbindungen. http://git-scm.com/book/ch2-5.html

Atlassian.com: Referenz zu git remote. *https://www.atlassian.com/de/git/tutorial/remote-repositories#!remote*

[26] *Atlassian.com:* Referenz zu git pull. *https://www.atlassian.com/de/git/tutorial/remote-repositories#!pull*

Git Homepage: Referenz zu git pull. *http://git-scm.com/docs/git-pull*

[27] *Atlassian.com:* Referenz zu git push. *https://www.atlassian.com/de/git/tutorial/remote-repositories#!push*

Git Homepage: Referenz zu git push. *http://git-scm.com/docs/git-push*

[28] *GitHub:* Projektseite. *https://github.com/*

[29] *GitHub:* Unterstützung von Subversion Clients. *https://help.github.com/articles/support-for-subversion-clients*

[30] *GitHub:* Private Repositorys durch Zahlung. *https://github.com/pricing*

7.2 Abbildungen

8 Anhang: Befehlsliste

8.1 Git-Grundlagen

`git init` – *Erstellt ein leeres Repository im aktuellen Verzeichnis*

`git clone <url> [<path>]` – *Klont ein entferntes Repository ins aktuelle oder definierte Verzeichnis*

`git status` – *Zeigt, welche Dateien hinzugefügt oder committet werden müssen*

`git log [-n <limit>] [--oneline]` – *Zeigt die Commit-History an (Optionale Begrenzung der anzuzeigenden Commits durch –n <limit>, optional auf eine Zeile pro Commit begrenzen mit –oneline)*

`git diff` – *Zeigt ungestagte Änderungen zwischen dem Arbeitsverzeichnis und dem Repository*

`git config [--global] user.name <name>` – *Legt den Benutzernamen fest*

`git config [--global] user.email <email>` – *Legt die E-Mail Adresse des Benutzers fest*

8.2 Dateioperationen

`git add <file/path>` – *Fügt die ausgewählte Datei oder das ausgewählte Verzeichnis zur Staging Area hinzu*

`git commit` – *Öffnet einen Editor zur Eingabe der Commit-Message, mit der alle Dateien aus der Staging Area comittet werden*

`git commit –m „<message>“` – *Commitet gestagte Dateien ohne Öffnung des Texteditor und verwendet <message> als Commit-Message*

`git commit --amend` – *Macht den letzten Commit rückgängig*

`git checkout <commit> <file>` – *Stellt eine Datei aus einem vergangenen Commit im Arbeitsverzeichnis wieder her und fügt sie zur Staging Area hinzu*

`git checkout <commit>` – *Stellt einen kompletten Commit wieder her*

`git clean –f [<path>]` – *Löscht alle modifizierten ungestagten Dateien (aus einem Verzeichnis)*

`git revert <commit>` – *Stellt einen vergangenen Commit wieder her, indem er als neuer Commit zur Versionsgeschichte hinzugefügt wird*

`git reset` – *Setzt Staging Area auf den Stand des aktuellsten Commits, ohne das Arbeitsverzeichnis zu verändern*

8.3 Branches

`git branch` – *Listet alle Branches im Repository auf*

`git branch <branch>` – *Kopiert den aktuellen Branch und speichert ihn als <branch>*

`git branch –d <branch>` – *Löscht <branch>, sofern er keine ungesicherten Änderungen enthält*

`git branch –D <branch>` – *Löscht <branch> unabhängig von ungesicherten Änderungen*

`git branch –m <branch>` – *Benennt den aktuellen Branch um zu <branch>*

`git checkout <branch>` – *Wechselt zum bereits existierenden Branch <branch>*

`git checkout –b <branch>` – *Kopiert den aktuellen Branch, speichert ihn als „<branch>“ und wechselt zu ihm*

`git merge <branch>` – *<branch> wird mit dem aktuellen Branch zusammengeführt*

8.4 Remote-Repositorys

`git remote add <name> <url>` – *Fügt ein Remote-Repository hinzu*

`git fetch <remote> [<branch>]` – *Lädt eine Kopie von einem Remote-Repository herunter, allerdings ohne sie selbstständig zu mergen*

`git pull <remote>` – *Lädt eine Kopie von einem Remote-Repository herunter und mergt sie mit der lokalen Kopie*

`git push <remote> <branch>` – *Lädt den angegebenen Branch auf das Remote-Repository hoch*

Notiz: `<arg>` ist eine notwendige, `[<arg>]` eine optionale Variable.