Versionsverwaltung mit Git

Fortgeschrittene Programmierkonzepte in Java, Haskell und Prolog

Merlin Denker (348043), Stefan Srecec (293318)

Betreuer: Thomas Ströder

Abbildung 1: Git Logo[4]

Dieser Ausarbeitung liegt das im Literaturverzeichnis unter [1] vermerkte Buch **Pro Git** von **Scott Chacon** und **Ben Straub** in der 2. Auflage zugrunde.

1 Einführung

1.1 Versionsverwaltung

Bei Git handelt es sich um ein Open Source Versionskontrollsystem. Versionskontrollsysteme (kurz: VCS) speichern die Änderungen an einer oder mehreren Dateien, die im Laufe der Zeit gemacht werden, und ermöglichen es dem Benutzer, eine Datei in eine ältere Version zurückzuversetzen. Dies funktioniert prinzipiell mit allen Arten von Dateien, egal ob es sich dabei um den Quellcode eines Programmes oder um Bilddateien in der Bildbearbeitung handelt.

Die erste Generation von Versionskontrollen waren lokale VCS, wie zum Beispiel SCCS, das bereits im Jahre 1972 entwickelt wurde [5, S. 368f]. Diese Programme speichern alle Änderungen, die man an einer Datei vornimmt, als Delta[1] in einer eigenen Datei auf der lokalen Festplatte. Da es allerdings immer häufiger vorkam, dass man nicht alleine an einem Projekt arbeitete, sondern mit anderen Personen auf anderen Systemen zusammenarbeiten musste, wurde das Konzept der zentralen VCS entwickelt.

Bei zentralen VCS werden die Dateien in sogenannten *Repositories* auf einem Server verwaltet. Die Clients können sich Datei-Versionen vom Server herunterladen und neue Versionen hochladen. Diese Arbeitsweise bietet einige Vorteile, die wir in Abschnitt 4 der Ausarbeitung genauer beleuchten werden. Ein solcher Server stellt jedoch auch einen Schwachpunkt dar, da er bei einem Ausfall das gesamte System lahmlegt.

Um dieses Problem zu beheben, wurden verteilte VCS wie Git entwickelt. In Git kopiert der Benutzer nicht nur eine Version der Dateien auf seinen Rechner, sondern er kopiert stattdessen das gesamte Repository. Fällt ein Server aus, kann das Projekt von jedem beliebigen Client aus wiederhergestellt werden.

[1] Delta = Differenz (zwischen zwei Dateien)

1.2 Geschichte von Git

Git entstand 2005 während der Arbeit am Linux-Kernel. Die Linux Community fing an ihr eigenes System zu entwickeln, da die Lizenz des bis dato benutzten verteilten VCS BitKeeper auslief und eine kostenlose Nutzung nicht mehr möglich war. Die treibende Kraft hinter dem Projekt war Linux Erfinder Linus Torvalds, der mit Git eine schnelle und leicht bedienbare Versionsverwaltung schaffen wollte, die selbst große Projekte wie den Linux-Kernel effektiv managen konnte. Großen Wert wurde dabei auf das Konzept der Branches gelegt, wodurch eine nichtlineare Entwicklung ermöglicht wurde. Dieses Konzept wird in Abschnitt 3.10 der Ausarbeitung noch genauer erläutert.

2 Grundlagen

Im folgenden Abschnitt werden wir betrachten, wie Git Daten interpretiert sowie Informationen speichert und verwaltet.

Die meisten VCS erfassen Informationen, wie in Abbildung 2 dargestellt, als eine Reihe von Änderungen, die an einer ursprünglichen Datei vorgenommen wurden. Git hingegen speichert die Versionen als eine Folge von sogenannten *Snapshots*. Ein solcher Snapshot entspricht einer Momentaufnahme aller Dateien und wird von Git erstellt, sobald man seinen aktuellen Zustand als eine neue Version in seiner Datenbank festhalten will. Um dabei die Effizienz zu erhöhen, wird auf das Kopieren nicht modifizierter Dateien verzichtet und stattdessen eine Referenz auf die vorherige Version angelegt.

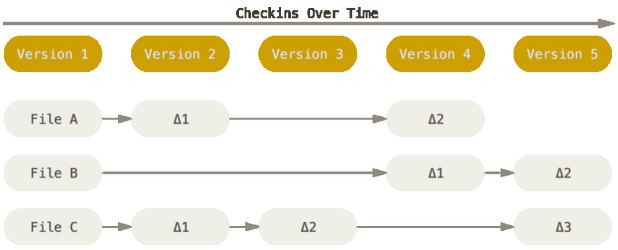

Abbildung 2: Speichern als Delta [1, S. 32]

Ein weiterer Vorteil von Git besteht darin, dass durch das Kopieren des gesamten Repositories auf den eigenen Rechner beinahe alle Operationen lokal durchführbar sind. Möchte man sich zum Beispiel die Projekthistory ausgeben lassen, ist es nicht notwendig einen zentralen Server zu kontaktieren, da man bereits alle Informationen in der lokalen Datenbank hat. Dadurch ergibt sich sowohl eine schnellere Ausführung der einzelnen Befehle als auch eine Unabhängigkeit von der Verfügbarkeit externer Server.

Die Referenzierung von Dateien realisiert Git intern nicht über einen Namen, sondern über eine 40 Zeichen lange Checksumme. Diese wird aus dem Inhalt oder der Verzeichnisstruktur berechnet und dient dazu, Übertragungsfehler oder Dateibeschädigungen zu identifizieren.

Es gibt genau drei Zustände, in denen sich eine Datei befinden kann:

- *modified*, wenn sie verändert wurde, aber die Änderungen noch nicht in die lokale Datenbank übernommen wurden
- *staged*, wenn sie sie bereits für den nächsten Commit vorgemerkt wurde
- *committed*, wenn die Änderungen durch einen Commit-Befehl in einer neuen Version festgehalten wurden

Daraus leiten sich die in Abbildung 3 dargestellten Hauptbereiche eines Git-Projektes ab. Es existieren das *Git Verzeichnis*, die *Working Area* und die *Staging Area*. Das Git Verzeichnis ist gleichzusetzen mit dem Repository und enthält sowohl Metadaten als auch die lokale Datenbank. In der Working Area befindet sich eine spezifische Version des Projektes, welche dort beliebig modifiziert werden kann. Bei der Staging Area handelt es sich im Prinzip nur um eine Datei im Git Verzeichnis, durch die festgelegt wird, welche Änderungen in den nächsten Commit aufgenommen werden. Sie entkoppelt somit die Working Area von der Datenbank im Verzeichnis und verschafft einem dadurch mehr Flexibilität, da man selbst wählen kann welche Änderungen man im nächsten Commit übernehmen will.

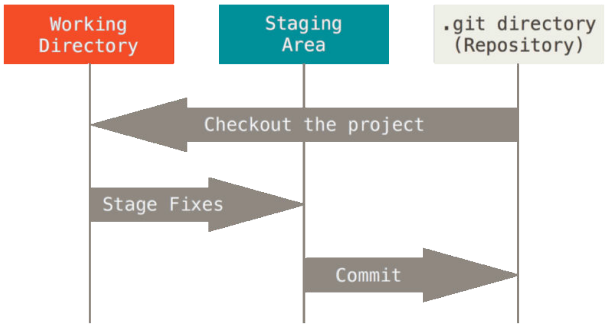

Abbildung 3: Bereiche eines Git-Repositories und Operationen dazwischen [1, S. 35]

Somit lässt sich ein typischer Arbeitsablauf in Git wie folgt darstellen:

1. Durch einen Checkout (vgl. 3.9) kopiert man eine Version der Dateien in die Working Area.
2. Diese Dateien werden beliebig bearbeitet.
3. Die gewünschten Änderungen werden für den nächsten Commit vorgemerkt, indem sie in die Staging Area kopiert werden (vgl. 3.3).
4. Durch einen Commit (vgl. 3.4) werden die Änderungen dauerhaft in die Datenbank übernommen.

3 Einblick in die Funktionen von Git

In diesem Abschnitt möchten wir die grundlegenden Befehle zur Arbeit mit Git vorstellen. Hierzu werden wir diese mithilfe des folgenden Beispiels demonstrieren.

Die Programmierer Kevin und Falk wollen zusammen ein Computerspiel entwickeln und dazu ein Git-Repository nutzen. Wir gehen in unserem Beispiel davon aus, dass Kevin bisher alleine an dem Projekt gearbeitet hat und Falk noch keine Projektdateien besitzt. Außerdem setzen wir voraus, dass beide Programmierer Git auf ihren Rechnern installiert haben. Sie wollen ihr Repository analog zu Abbildung 4 realisieren:

- *Beide haben auf ihren Rechnern eine lokale Kopie des Repositories.*
- *Zum Datenaustausch richten sie außerdem ein Repository auf dem Server von Kevin ein, der über das Internet erreichbar ist.*

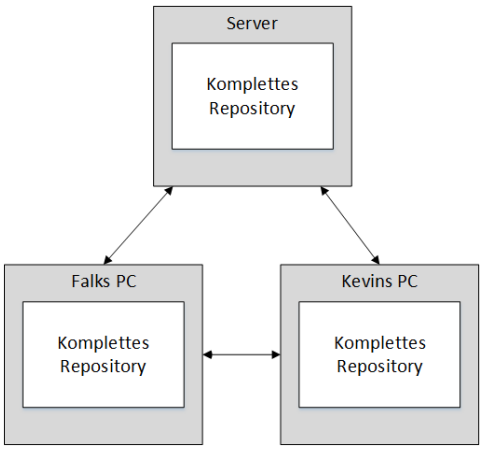

Abbildung 4: Verteiltes VCS mit zentralem Server

Um die im Beispiel vorgestellte Architektur einzurichten, benötigt man bereits die meisten der Befehle, die man für die alltägliche Arbeit mit Git benötigt. In den folgenden Unterabschnitten wird daher dieser Prozess von uns beispielhaft dargestellt, um die Befehle einzeln zu demonstrieren.

Wir werden in diesem Abschnitt die Arbeitsweise von Git über die Kommandozeile demonstrieren, da nur hiermit alle Befehle realisiert werden können. Die Bedienung von Git mit vereinfachten GUI-Tools ist nicht Teil dieser Ausarbeitung. Alle hier vorgestellten Befehle haben eine Vielzahl von optionalen Parametern, die wir im Rahmen dieser Ausarbeitung nicht komplett vorstellen können. Die komplette Syntax eines Befehls kann über das im Abschnitt 3.1 vorgestellte `help` Kommando eingesehen werden.

3.1 help

Wie jedes gute Programm beinhaltet auch Git eine Hilfsfunktion, über die man sich die Syntax und die Funktionsweise aller Kommandos ausgeben lassen kann. Diese Hilfsfunktion kann man über das Kommando `help` mit der folgenden Syntax aufrufen:

```
# git help <verb>
```

Der Parameter `<verb>` kann ein anderes Git-Kommando sein, zu dem man Hilfe benötigt.

Kevin möchte auf seinem Server ein neues Git-Repository erstellen. Da er das noch nie gemacht hat, ruft er die Hilfsfunktion für die init-Funktion mit dem folgenden Kommando auf:

```
# git help init
```

3.2 init

Der init-Befehl wird verwendet, um ein neues Repository anzulegen und darin ein neues oder bestehendes Projekt zu verwalten. Der Befehl erzeugt im aktuellen Verzeichnis ein neues Unterverzeichnis mit dem Namen **.git**, welches alle benötigten Daten für das leere Git-Repository enthält. Die Syntax ist simpel, da der init-Befehl in den meisten Fällen keine Argumente benötigt.

```
# git init
```

Kevin wechselt mit seiner Shell in das Verzeichnis auf dem Server, in dem er das Repository erstellen möchte. Dort führt er das folgende Kommando aus:

```
# git init --bare
```

*Git erzeugt dann ein neues Unterverzeichnis **.git**. Es beinhaltet das Grundgerüst eines leeren Repositories, welches wegen des Parameters `--bare` keine Working Area hat. Danach muss er noch ein Repository auf seinem lokalen Rechner erstellen. Dazu wechselt er in das lokale Verzeichnis, in dem sein Projekt liegt, und führt dort den Befehl `git init` aus. Git erzeugt wieder einen Ordner **.git** und betrachtet dabei das aktuelle Verzeichnis als Working Area.*

3.3 add

Ein frisch erzeugtes Repository ist vorerst leer. Um Dateien von der Working Area in die Staging Area zu kopieren, benutzen wir den folgenden Befehl:

```
# git add <filepattern>
```

Der Befehl wird sowohl dafür verwendet, Dateien in die Versionsverwaltung von Git aufzunehmen, als auch, um sie für den nächsten Commit zu markieren. Der Parameter `<filepattern>` gibt mithilfe eines File-Patterns an, welche Dateien in die Staging Area kopiert werden sollen. Hierbei können auch Namen von Unterverzeichnissen verwendet werden.

Beispiele:

- ***.c** erfasst alle c-Dateien im aktuellen Verzeichnis
- **index.html** erfasst nur die Datei index.html im aktuellen Verzeichnis
- **.** erfasst das gesamte Verzeichnis mit allen Dateien und Unterverzeichnissen
- **img/*** erfasst alle Dateien im Unterverzeichnis img

Der Befehl `add` kann mehrmals ausgeführt werden, falls man Dateien vergessen hat oder sie nochmal überarbeitet. Als neue Version im Repository werden sie erst mit einem `commit` übernommen. Die jeweils letzte per `add` hinzugefügte Version einer Datei wird beim `commit` übertragen.

Kevin möchte sein gesamtes bestehendes Projekt in das Repository einfügen. Dazu führt er den folgenden Befehl in seiner Working Area aus

```
# git add .
```

Git kopiert nun alle Dateien, die sich zum aktuellen Zeitpunkt im aktuellen Verzeichnis befinden, in die Staging Area des lokalen Repositories.

3.4 commit

Um die Dateien in der Staging Area endgültig als neue Version in das lokale Repository zu übertragen, verwenden wir den folgenden Befehl:

```
# git commit
```

Git legt dann eine neue Version im Repository an und verschiebt alle Dateien aus der Staging Area in diese Version. Die Staging Area ist somit wieder leer. Wurde eine Datei nach dem Stagen[2] erneut modifiziert, werden diese Änderungen nicht automatisch in den Commit übernommen, da Git nur den Zustand berücksichtigt, den die Datei zum Zeitpunkt des letzten `add` Befehls hatte. Um die neuen Änderungen ebenfalls in die aktuelle Version zu übernehmen, muss die Datei erneut gestaged werden.

[2] Stagen = Kopieren einer Datei in die Staging Area

Der Befehl `commit` akzeptiert eine Vielzahl von optionalen Parametern, von denen der wohl wichtigste `-m <msg>` sein dürfte. Hiermit übergibt man Git eine Nachricht `<msg>`, die als Kommentar an den Commit angehängt wird. Diese kann später im Log des Repositories eingesehen werden.

Um eine neue Version in seinem lokalen Repository zu erzeugen, führt Kevin den folgenden Befehl aus:

```
# git commit -m 'initial commit'
```

Git erzeugt dann die erste Version des Projekts im lokalen Repository auf Kevins Rechner mit dem Hinweis 'initial commit'. Die Staging Area auf seinem Rechner ist nun wieder leer.

3.5 remote

Bevor man Änderungen an ein anderes Repository übertragen oder von ihm erhalten kann, muss dieses als Remote-Repository zum lokalen Repository hinzugefügt werden. Dies bewirken wir mit dem Kommando `remote` und der Option `add`.

```
# git remote add <name> <url>
```

Der Parameter `<name>` stellt hier einen nur zur lokalen Verwendung einfach lesbaren Namen dar. Über `<url>` wird festgelegt, wo das Remote-Repository liegt. In der URL sind gegebenenfalls notwendige Zugangsdaten enthalten.

Bevor Kevin die lokal vorliegenden Änderungen auf den Server übertragen kann, muss er Git zunächst mitteilen, wo und wie dieser Server zu erreichen ist. Dazu benutzt er folgenden Befehl:

```
# git remote add mein_server kevin@kevinsserver.de:~/pfad/zum/repo
```

Git merkt sich nun, dass es das Repository mein_server unter kevinsserver.de erreichen kann und sich dort mit dem Username kevin authentifizieren soll. Das Repository befindet sich auf dem Server unter dem Verzeichnis ~/pfad/zum/repo, welches das Verzeichnis ist, in dem zuvor `init` *ausgeführt wurde.*

3.6 push

Der Befehl `push` überträgt die im lokalen Repository vorliegenden Änderungen in ein Remote-Repository. Er wird zum Beispiel wie folgt aufgerufen:

```
# git push <remote_repository> <branch>
```

Über die Parameter `<remote_repository>` und `<branch>` wird Git angegeben, zu welchem Remote-Repository die Daten übertragen und in welchen Branch (vgl. 3.10) sie eingefügt werden sollen.

Ein `push` ist nur möglich, wenn im Remote-Repository in der Zwischenzeit kein anderer Entwickler Änderungen hinzugefügt hat, die man selbst noch nicht in das eigene Repository übertragen hat. Außerdem dürfen in der lokalen Working-Area keine Änderungen vorliegen, welche noch nicht committed wurden.

Damit Falk die Änderungen von Kevin herunterladen kann, müssen diese zunächst auf den Server übertragen werden. Dazu benutzt Kevin den folgenden Befehl:

```
# git push mein_server master
```

Git überträgt nun alle vorliegenden Änderungen auf Kevins Server und fügt sie in den Entwicklungszweig master ein. Dieser stellt den Hauptzweig jedes Repositories dar.

3.7 clone

Mithilfe des Befehls `clone` kann man ein bestehendes Remote-Repository in ein lokales Verzeichnis kopieren. Der Befehl führt zudem automatisch einen Checkout (vgl. 3.9) aus.

```
# git clone <repository>
```

Der Parameter `<repository>` ist hier analog zu dem Befehl `remote add` (vgl. 3.5) eine URL, über die Git das Remote-Repository erreichen kann.

Zu dem so lokal erzeugten Repository wird automatisch ein Remote-Repository hinzugefügt (vgl. 3.5), welches den Namen origin erhält. Wenn man einen anderen Name verwenden möchte, kann man dies mittels des Parameters `-o <name>` bewirken.

Nachdem Kevin das Repository auf dem Server aufgesetzt hat, kann Falk dieses nun auf seinen lokalen Rechner kopieren. Dazu führt er in dem Verzeichnis, welches später seine Working Area werden soll, den folgenden Befehl aus:

```
# git clone -o kevins_server falk@kevinsserver.de:/home/kevin/pfad/zum/repo
```

Git meldet sich mit dem Username falk am Server kevinsserver.de an und kopiert das Repository unter /home/kevin/pfad/zum/repo auf Falks Rechner in das Verzeichnis, in dem er den Befehl ausgeführt hat. Wir nehmen hierfür an, dass der User falk die notwendigen Rechte hat, um auf den Pfad zuzugreifen. Zu dem lokalen Repository von Falk wird ein Remote-Repository hinzugefügt, welches durch den Namen kevins_server angesprochen werden kann. Anschließend führt Git einen Checkout in Falks Working Area durch.

3.8 pull

Um Änderungen aus einem anderen Repository in das eigene zu übernehmen, kann man diese auch aktiv herunterladen, statt zu warten, bis jemand die Daten aus dem anderen Repository per `push` Kommando überträgt. Dazu dient der Befehl `pull`. Eine einfache Art, diesen zu verwenden, wäre beispielsweise:

```
# git pull <repository>
```

Der Parameter `<repository>` gibt an, von welchem Remote-Repository die Änderungen geholt werden sollen. Ein `pull` ist nur dann möglich, wenn in der lokalen Working-Area keine Änderungen vorliegen, die noch nicht committed wurden.

Falk hat sich mittlerweile in das Projekt eingearbeitet. Dies hat einige Zeit gedauert und Kevin hat inzwischen Änderungen an dem Projekt vorgenommen, die er auf den Server übertragen hat. Damit Falk diese Änderungen erhält, muss er einen `pull` *ausführen:*

```
# git pull kevins_server
```

Git lädt nun alle Änderungen auf dem Server herunter und fügt sie in Falks Repository ein.

3.9 checkout

Ein Checkout beschreibt den Vorgang, einen bestimmten Zustand des Projektes aus dem Repository in die lokale Working Area zu kopieren. Meist wird damit die aktuellste Version, welche die Änderungen anderer Entwickler enthält, in die eigene Working Area geholt. Der Befehl kann aber auch dazu genutzt werden, alte Versionen wiederherzustellen. Die grundlegende Syntax lautet folgendermaßen:

```
# git checkout <branch>
```

Git kopiert dann die aktuelle Version des Entwicklungszweiges `<branch>` (vgl. 3.10) in die lokale Working Area.

Damit Falk die eben heruntergeladenen Änderungen auch einsehen kann, muss er diese nun noch in seine Working-Area kopieren. Dazu benutzt er den folgenden Befehl:

```
# git checkout master
```

Git kopiert nun die neueste Version des Projekts aus dem Haupt-Entwicklungszweig in die lokale Working-Area von Falk.

3.10 branch

Branching ist ein sehr nützliches Feature von Git und ermöglicht es, ein Projekt parallel in verschiedene Richtungen zu entwickeln. Branches kann man sich am besten als Verzweigungen vorstellen, die sich von einem Entwicklungsstrang lösen und beliebig weiter verzweigt oder wieder in den Ursprungszweig zurückgeführt werden können. Git realisiert einen Branch als einen simplen Zeiger auf einen Commit. Mit dem initialen Commit eines Projektes wird auch automatisch ein Branch mit dem Namen *Master* erzeugt. Jeder weitere Commit besitzt einen Zeiger auf seinen direkten Vorgänger. So entsteht während der Projektentwicklung eine Kette von Commits, wobei der Master-Branch stets auf das letzte Element verweist.

Einen neuen Branch erstellt man mit folgendem Befehl:

```
# git branch <newBranch>
```

Zwischen verschieden Branches wechseln kann man mit dem Kommando:

```
# git checkout <branchName>
```

Möchte man zwei Branches wieder zusammenführen, muss man in den Branch wechseln, welcher fortgeführt werden soll, und den folgenden Befehl ausführen:

```
# git merge <branchName>
```

Der Branch mit dem Namen `<branchName>` wird dann in den aktuellen Branch eingefügt. Man unterscheidet dabei zwischen zwei Arten des Zusammenführens. Wenn der einzufügende Branch ein direkter Nachfolger des anderen Branches ist, führt Git einen sogenannten Fast Forward merge aus. In diesem einfachen Fall wird der Zeiger einfach auf die Stelle des anderen Branches vor bewegt. Ist der eine Branch hingegen kein direkter Vorgänger des anderen muss ein 3-Wege merge ausgeführt werden. Git erstellt nun auf Basis der Snapshots der beiden Branches und eines automatisch gewählten gemeinsamen Vorgängers einen neuen Commit, welcher nun auf die beiden Branches als seine Vorgänger verweist.

Kevin möchte ein neues Menüdesign für das Spiel entwerfen. Für die Entwicklung des Menüs erstellt er einen neuen Branch namens design (C2):

```
# git branch design
```

Währenddessen arbeitet Falk im Master-Branch weiter an dem Hauptspiel und tätigt weitere Commits (C3). Beiden fällt ein Bug auf und Kevin beschließt, diesen erst zu beheben, bevor er mit der Entwicklung im Design weitermacht. Dazu wechselt er zurück in den Master-Branch, erstellt einen neuen Branch namens bugfix (C4) und wählt diesen aus :

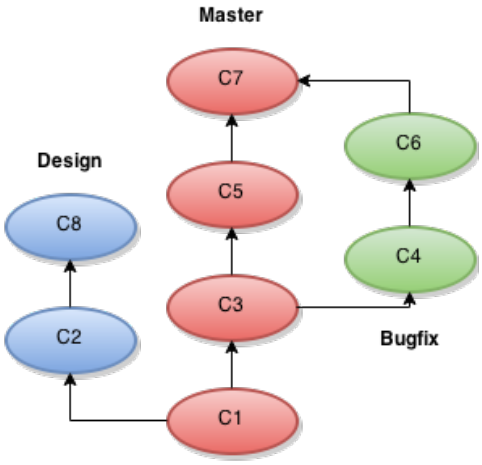

Abbildung 5: Beispiel Branches

```
# git checkout master
# git branch bugfix
# git checkout bugfix
```

Erst nachdem die Fehlerbehebung fertig ist und getestet wurde, wird sie wieder in den Master-Branch eingefügt. Durch den 3-Weg-Merge wird aus C5, C6 und dem gemeinsamen Vorgänger C3 ein Commit C7 erstellt.

```
# git checkout master
# git merge bugfix
```

Da Falk jedoch das neu entwickelte Menü-Design nicht wirklich gefällt, löscht er den entsprechenden Branch mithilfe des Parameters -d im `branch` *Befehl einfach wieder:*

```
# git branch -d design
```

3.10.1 Konfliktbehandlung

Bei einem `merge` kann es zu sogenannten Konflikten kommen, wenn dieselbe Datei in beiden Branches an derselben Stelle modifiziert wurde. Eine automatische Zusammenführung der beiden Zweige ist in diesem Fall nicht möglich und es erscheint eine entsprechende Meldung.

Mithilfe des Befehls `status` kann man sich anzeigen lassen, welche Stellen in welchen Dateien genau von dem Konflikt betroffen sind. Der Entwickler muss nun alle von Git markierten Konfliktstellen durchgehen und diese manuell lösen. Dazu kann er sich entweder für eine der beiden Varianten entscheiden oder eine komplett neue Lösung einsetzen. Wurden alle Konflikte gelöst, müssen die betroffenen Dateien durch ein `add` als bereinigt markiert werden und die Branches können nun wie beschrieben zusammengeführt werden.

Nehmen wir an im vorangehenden Beispiel zum Thema Branching hätten die Branches Master und Bugfix in der Datei index.html die selbe Stelle auf unterschiedliche Art und Weise verändert. Ein automatischer merge ist nun nicht möglich, da Git nicht weiß welche der beiden Implementierungen es in den neuen Commit übernehmen soll. Man wird stattdessen folgende Meldung erhalten:

```
# git merge bugfix
# CONFLICT (content): Merge conflict in index.html
# Automatic merge failed; fix conflicts and then commit the result.
```

Ein Aufrufen des Befehls `status` *würde anzeigen, dass es in der Datei index.html zu einem Konflikt gekommen ist:*

```
# git status
# Unmerged paths:
# both modified: index.html
```

Git markiert dabei die in Konflikt stehenden Stellen einer Datei mit Markern. In diesen ist der obere Teil aus dem aktuellen Branch und der untere aus dem zu mergenden Branch:

```
# «««< HEAD
# Änderung aus Master-Zweig
# =======
# Änderung aus bugfix-Zweig
# »»»> bugfix
```

Um den Konflikt zu lösen, muss man nun die beiden Versionen in der Datei sinnvoll zusammenfügen, die Datei erneut stagen und den merge erneut durchführen.

---

4 Vergleich mit anderen Versionskontrollsystemen

Im Folgenden möchten wir uns andere Konzepte von Versionsverwaltung näher ansehen und die Vor- und Nachteile dieser Systeme gegenüber Git betrachten.

4.1 Lokale Versionsverwaltung mit SCCS

Source Code Control System (SCCS) war eines der ersten Versionsverwaltungs-Systeme und wurde 1972 veröffentlicht. Der originale Code wird nicht länger gewartet, es existieren jedoch diverse Forks [3] des Projekts. Der Quellcode der Variante von Solaris wird unter CDDL bis heute weiterentwickelt.

SCCS verwaltet keine Projekte, in denen viele Dateien zu einem Repository zusammengefasst werden, sondern erzeugt für jede verwaltete Datei eine Art Miniatur-Repository in Form einer rcs-Datei. Diese Datei enthält den ursprünglichen Code der Datei und alle von da an vorgenommenen Änderungen als Delta [5, S. 364] mit Meta-Informationen wie Zeitpunkt, Autor und Kommentare. Mithilfe dieser Informationen ist es möglich, frühere Versionen einer Datei wiederherzustellen. Wenn jedoch die rcs-Datei verloren geht, ist die gesamte Historie ebenfalls verloren. Das System sieht zudem keine Kopien auf anderen Clients oder Servern vor. Das Erstellen einer bestimmten Version einer Datei kann zudem erheblich länger dauern als dies bei Git der Fall ist, da SCCS zunächst sämtliche Deltas einzeln anwenden muss.

Solaris SCCS unterstützt heutzutage Branching und im Gegensatz zu Git auch File Locking [2]. Beim File Locking wird eine Datei im Repository als gesperrt markiert, sodass keine Änderungen mehr daran vorgenommen werden können, bis diese wieder entsperrt wird. Dieser Mechanismus wird auch bei der zentralen Versionsverwaltung genutzt und beugt Konflikten vor. Dieses Verfahren ist in Git aufgrund der verteilten Architektur nicht möglich.

4.2 Zentrale Versionsverwaltung mit SVN

Subversion (SVN) ist ein Versionsverwaltungs-System, welches auf einer *Client-Server-Architektur* basiert. Bei diesen sogenannten zentralen Versionsverwaltungs-Systemen wird das Repository auf einem zentralen Server gespeichert. Der entscheidende Unterschied zu Git liegt darin, dass nur der Server die gesamte Historie speichert. Die Clients brauchen das Repository nicht wie bei Git zu klonen oder kopieren, sondern erstellen ihr Arbeitsverzeichnis direkt per Checkout vom Server.

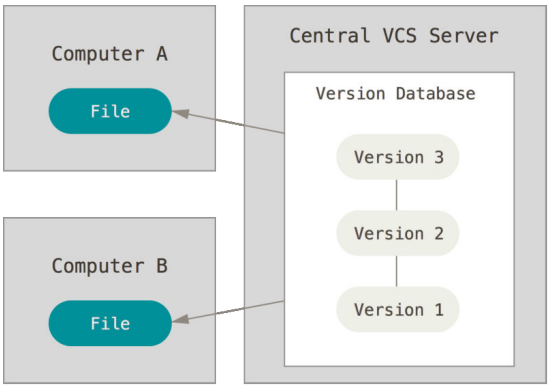

Abbildung 6: Zentrale Versionsverwaltung [1, S. 29]

Dies hat den Vorteil, dass der erste Checkout bei großen Repositories deutlich schneller durchgeführt werden kann, da nur ein einzelner Zustand des Projekts auf den Client kopiert werden muss. Auf diese Weise wird auch weniger Speicherplatz auf den Clients belegt. Wenn der Client seine Änderungen mit einem Commit wieder in das Repository überträgt, stehen diese sofort allen anderen Clients mit Verbindung zum Server

[3] Fork = Programm, welches auf dem Quelltext eines anderen Programmes beruht und auf dieser Basis unabhängig davon weiterentwickelt wurde

zur Verfügung. Um zu verhindern, dass zwei Personen an derselben Datei Änderungen vornehmen, die nicht automatisch vom System zusammengeführt werden können, unterstützt auch SVN File Locking [3, S. 296].

Aus der zentralen Architektur von SVN resultieren aber auch einige Nachteile gegenüber verteilten Systemen wie Git. Da auf jedem Client immer nur eine einzige Version vorliegt, kann im Falle eines Verlusts des Repositories auf dem Server die Historie nicht wiederhergestellt werden. Ohne eine bestehende Netzwerkverbindung kann SVN zudem nicht sinnvoll genutzt werden, da man keine alten Versionen einsehen kann und es auch nicht möglich ist, die lokale Version zum Sichern oder Verbreiten auf den Server zu kopieren. Eben-sowenig kann man Änderungen anderer Nutzer aus dem Repository übernehmen.

Für SVN existiert eine umfangreiche Sammlung von Bibliotheken, um das System mit verschiedenen Programmiersprachen ansprechen zu können. Eine solche Bibliothek oder API existiert für Git derzeit nicht, befindet sich aber in Form der libgit2 in der Entwicklung.

5 Zusammenfassung

In den vorherigen Abschnitten haben wir einige Besonderheiten von Git kennengelernt und sind auf die Unterschiede zu anderen Systemen eingegangen. Die Art und Weise wie Git Daten speichert ermöglicht eine sehr schnelle Bearbeitung der meisten Befehle und die Gefahr von Datenverlusten wird dabei minimiert. Wir haben gelernt, wie man ein neues Repository erstellt oder ein bestehendes auf seinem eigenen Rechner einbindet. Änderungen aus der Working Area müssen mit Hilfe eines `add` Befehls in der Staging Area vorgemerkt werden und können dann mit einem Commit dauerhaft gespeichert werden. Durch das Branching kann man von einer linearen Entwicklung abweichen und für bestimmte Aufgaben eigene Branches anlegen. Git erleichtert die Arbeit an kleinen sowie an großen Projekten und erlaubt eine unkomplizierte Koordination selbst bei einer Vielzahl von Entwicklern.

Literatur

[1] CHACON, Scott; STRAUB, Ben: **Pro Git - Everything you need to know about Git**. *2. Aufl. Apress*, **2014**

[2] CHRISTIAS, Panagiotis: **UNIX man pages : sccs()**. *Stand 05.05.2015. http://www.lehman.cuny.edu/cgi-bin/man-cgi?sccs*, **1994**

[3] COLLINS-SUSSMAN, Ben; FITZPATRICK, Brian W.; PILATO, C. Michael: **Version Control with Subversion**. *2. Aufl. O'Reilly Media*, **2008**

[4] LONG, Jason: **Full color Git logo for light backgrounds**. *https://git-scm.herokuapp.com/images/logos/downloads/Git-Logo-2Color.png Stand 07.05.2015*, **o.J.**

[5] ROCHKIND, Marc J.: **The Source Code Control System**. *IEEE Transactions on Software Engineering.* **1 (4)**: 364-370, **December 1975**