Projektarbeit - Fortgeschrittene Programmierkonzepte

Versionsverwaltung mit Git

Hauke Heidemeyer Felix Huhn

10.05.2019
Betreuer: S. Dollase

Kurzfassung

Software wird zu meist von mehreren Programmierern parallel entwickelt und bearbeitet. Da Dateien häufig zeitgleich an unterschiedlicher Stelle modifiziert werden, ist es nicht nur sinnvoll die einzelnen Änderungen nachverfolgen zu können, sondern auch die Möglichkeit zu besitzen diese ggf. rückabzuwickeln. Das beschreibt exakt das Anforderungsprofil von Versionsverwaltungssoftware.
Diese Arbeit soll ein grundlegendes Verständnis für Versionsverwaltung im Allgemeinen vermitteln und befasst sich zu diesem Zweck genauer mit der Software Git. Um einen guten Umgang mit der Software zu ermöglichen, werden zunächst essenzielle Begrifflichkeiten erläutert. Für ein tieferes Verständnis werden auch die Prozesse hinter einzelnen Befehlen näher betrachtet.

1 Einleitung

Eine Herausforderung der Teamarbeit in der Softwareentwicklung, ist es viele Dateien, die z. B. den Quellcode eines Programms enthalten, mit mehreren Entwicklern zeitgleich zu bearbeiten. Es ergeben sich schnell Versionskonflikte, die zu Compilerfehlern führen und viel wertvolle Zeit beanspruchen. Paralleles Arbeiten ist so kaum möglich.
Die Problemstellung ist aber nicht nur für die Entwicklung von Programmen relevant, sondern lässt sich auch ohne weiteres auf andere Bereiche übertragen. Schließlich sind all jene von der Problematik betroffen, die Dateien über einen längeren Zeitraum, mit mehreren Personen erzeugen und bearbeiten.

Als Lösungsansatz dieser Problemstellung wurden verschiedene Versionskontrollsysteme (version control systems, kurz: VCS) entwickelt. Git stellt dabei ein Versionskontrollsystem dar, dass viele Vorteile gegenüber anderen Systemen bietet. Ziel dieser Arbeit ist es die grundlegenden strukturellen Konzepte zu erläutern und zusätzlich einen Einstieg im Umgang mit Git zu erleichtern.
Dazu wird Kapitel zwei dieser Projektarbeit zunächst die Versionsverwaltung im Allgemeinen näher beleuchten und einige Begrifflichkeiten klären. Anschließend wird genauer auf die Funktionsweise der Software Git eingegangen und im Speziellen der Ablauf eines

Commits und die Ermittlung fehlerhafter Commits betrachtet. Des Weiteren werden Entwicklungszweige und die Zusammenführung solcher behandelt. Schließlich soll aufgezeigt werden wie Versionskonflikte gelöst werden können.

2 Definitionen

Versionsverwaltungssoftware beschreibt ein System, das jegliche Änderungen an Dateien dokumentiert. Dadurch ist es auch im Nachhinein möglich zu jedem Zeitpunkt einer Entwicklung zurückzukehren. Es lassen sich aber nicht nur einzelne Dateien, sondern auch ein komplettes Projekt zurücksetzen. Zudem kann nachvollzogen werden wer, wann, welche Änderungen vorgenommen hat. Das erleichtert z. B. die Fehlersuche in komplexen Software-Projekten.[1]

Allgemein kann man zwischen den drei verschiedenen Arten der Versionsverwaltung - *lokale*, *zentrale* und *verteilte Versionsverwaltung* - differenzieren. Diese unterscheiden sich im Wesentlichen in der Art und Weise der Speicherung von Änderungen an Dateien und dem Umgang mit diesen.
Die lokale Versionsverwaltung findet nur auf dem System des jeweiligen Anwenders statt. Das führt vor allem beim Arbeiten in Teams zu Problemen. Als Lösung wurde die zentrale Verwaltung von Versionen entwickelt.
Dabei hält und verarbeitet ein zentraler Server alle Dateien. Mehrere Clients laden Daten hoch oder herunter und verändern diese. Über Jahre hinweg war diese Art von System weit verbreitet. Die Vorteile liegen klar auf der Hand: es besteht eine Übersicht, welche Person woran arbeitet, Zugriffsrechte können vergeben werden und die vorhanden Daten können einfach administriert und verwaltet werden. Nachteilig ist nur, das ein solches System einen „Single-Point-of-Failure“ darstellt. D. h. es muss eine gute Backup Strategie implementiert werden.
Eben diese Problematik wird durch die verteilte Verwaltung angegangen. Jeder Client hält alle Daten und somit die komplette Versionshistorie. Gespeichert wird diese in dem sogenannten *Repository*, entweder auf dem lokalen PC (*local Repository*) oder auf einem Server (*remote Repository*). Dadurch stellt jeder Client nicht nur ein komplettes Backup des Projekts dar, sondern es kann auch ohne eine Verbindung zum Server gearbeitet werden.

2.1 Commit

Ein *Commit* ist ein lokal ausgeführter Befehl, der alle Dateien, die zum Commit markiert wurden, in das local Repository eingliedert. Es ist üblich eine Commit Nachricht zu verfassen, um Mitentwicklern mitzuteilen, welche Änderungen vorgenommen wurden. Per default enthält diese die Namen, der seit dem letzten Commit geänderten und hinzugefügten Dateien.[2]

2.2 Branch

Zentraler Bestandteil von Git sind die Entwicklungszweige genannt *Branches*. Diese können fortgeführt werden, ohne sich gegenseitig zu beeinflussen. Der Master Branch ist typischerweise der Hauptentwicklungszweig und wird standardmäßig von Git angelegt. Die-

[1] Vgl. Chacon, Scott/Straub, Ben: Pro Git. 2. Auflage. Berkely, CA, USA: Apress, 2014, S. 9.
[2] Vgl. a. a. O., S. 35f.

ser besitzt jedoch keinerlei spezielle Funktionalitäten.[3] Git besizt verschiedene Arten von Branches. Ein *remote Branch* exitsiert in dem remote Repository, während ein *tracking Branch* eine lokale Kopie im local Repository ist. Der tracking Branch wird nur manuell synchronisiert. Zudem gibt es den *local Branch*, welcher die Commits im local Repository enthält. Dieser gleicht somit dem tracking Branch, bis auf die neu lokal hinzugefügten Commits.
Ein Entwicklungszweig kann nützlich sein, wenn ein Autor (oder eine Gruppe von Autoren) ein Feature implementieren will. Verschiedene Branches können parallel entwickelt werden und es muss nicht auf die Fertigstellung eines anderen Features gewartet werden.[4] Beispielhaft zu nennen ist an dieser Stelle das App-Development. So kann in einem Branch das User-Interface (UI) entwickelt werden, während zeitgleich ein anderer Programmierer eine Datenbankanbindung implementiert.

2.3 Merge

Der Schlüssel zum parallelen Arbeiten ist es, verschiedene Dateiversionen wieder zusammenzuführen. Wie in Abschnitt 2.2 bereits erläutert, ist es in der Softwareentwicklung häufig sinnvoll, wenn nicht sogar notwendig, Arbeit in mehrere Zweige zu gliedern und diese verschiedenen Personen zuzuordnen. Das Zusammenführen solcher unabhängigen Branches wird als *Merge* bezeichnet.[5]

2.4 Dateizustände in Git

Dateien in Git können immer einem definierten Zustand zugeordnet werden. Unterschieden wird zwischen *Tracked*, *Ignored* und *Untracked*. Tracked bezeichnet jede Datei, die sich im Repository befindet und von Git verwaltet wird.
Ignored hingegen sind all jene Dateien, die nicht verarbeitet werden. Diese liegen zwar im Arbeitsverzeichnis des Benutzers, unterliegen aber nicht der Versionskontrolle. Dateien, die ignoriert werden sollen, müssen explizit in einer Datei mit dem Namen `.gitignore` aufgeführt werden. Dies ist nützlich bei der Arbeit mit Entwicklungsumgebungen, die zusätzliche Dateien generieren, die für die Funktionalität des Projekts keine besondere Bedeutung besitzen (z. B. Compilerlogs).
Untracked ist der Zustand aller anderen Dateien, die Git erkennt, jedoch auch nicht weiter verarbeitet. Neu angelegte Dateien z. B. werden immer zuerst als Untracked interpretiert.[6]

Der Tracked-Zustand ist wiederum in die Unterstufen *Unmodified*, *Modified* und *Staged* gegliedert.
Ist eine Datei Unmodified, so wurde der Inhalt dieser seit dem letzten Commit nicht geändert. Sobald eine Datei modifiziert wurde, ändert sich ihr Status in Modified. Wird eine Datei an den Befehl `git add` übergeben, wechselt diese in den Zustand Staged. Alle Dateien im Zustand Staged werden beim nächsten Commit erfasst und dem Repository hinzugefügt.[7]

[3] Vgl. Chacon/Straub: Pro Git, S. 63.
[4] Vgl. Ram, Karthik: Git can facilitate greater reproducibility and increased transparency in science. Source Code for Biology and Medicine, 8 Feb 2013, Nr. 1, S. 2.
[5] Vgl. Bitbucket: Git Merge.
[6] Vgl. Loeliger, Jon/McCullough, Matthew: Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. O'Reilly Media, Inc., 2012, S. 46ff.
[7] Vgl. Blischak, John D./Davenport, Emily R./Wilson, Greg: A Quick Introduction to Version Control with Git and GitHub. PLOS Computational Biology, 12 01 2016, Nr. 1, S. 3ff.

3 Funktionsweise von Git

Git speichert und verarbeitet Informationen anders als andere VCSs. Der Hauptunterschied zwischen den Systemen ist die Art der Datenverarbeitung. Herkömmliche Systeme speichern Informationen in einer Art Liste, die alle Dateien und deren Änderungen enthält (auch delta-based version control genannt).[8] Da Git die vollständige Versionshistorie auf jedem Client-Computer speichert, können zu jeder Zeit, auch ohne Internetverbindung, Dateien bearbeitet werden. Das ist nicht selbstverständlich, da viele Versionskontrollsysteme eine ständige Verbindung zu einem zentralen Server voraussetzen.

Um die Datenintegrität zu wahren, verwendet Git ein Prüfsummen-System - es werden SHA-1 Hashes[9] eingesetzt. Dadurch ist es nahezu unmöglich, dass die Git-Software Dateiänderungen verpasst.

Git speichert intern alles in Form von Objekten. Jedes Objekt in Git besteht aus einem *Header* und dem eigentlichen Inhalt. Der Header wird dem Inhalt eines Objekts vorangestellt. Dieser kennzeichnet den Objekttyp innerhalb von Git. Wird eine Datei in Git gespeichert, so erzeugt Git die Prüfsumme und erzeugt anschließend ein *Blob* Objekt. Ein Blob ist ein spezielles Objekt, dass den Inhalt einer Datei speichert und zur Identifikation nach der Prüfsumme benannt wird. Der eigentliche Name der Datei wird nicht im Blob Objekt gespeichert. Damit Git einen Blob auch als solchen identifizieren kann, wird dies im Header mit „blob“ vermerkt. Die Ordnerstruktur und die Dateinamen werden in sogenannten *Tree* Objekten gespeichert. Diese speichern die Prüfsummen von verschiedenen Blobs und Trees und die zugehörigen Dateinamen. Das spiegelt eine vereinfachte Struktur des Unix Dateisystems wider. Git kann daher auch als eigenes Mini-Dateisystems mit zusätzlichen Funktionalitäten verstanden werden.[10]

Erfolgt ein Commit(`git commit -m "MESSAGE"`), so erstellt Git ein Blob Objekt von jeder Staged Datei und speichert dessen Namen und die Ordnerstruktur in einem Tree Objekt ab. Anschließend wird ein Commit-Objekt erstellt, dass einen Zeiger auf das oberste Tree Objekt enthält und zudem Metadaten speichert. Diese beinhalten üblicherweise die Dateigröße, den Autor, den Committer und eine kurze Beschreibung des Commits. Zusätzlich wird bei einem weiteren Commit mindestens ein Zeiger auf den vorherigen Commit gespeichert, so dass die Historie der Änderungen lückenlos verfolgbar bleibt.[11] Das erlaubt ein Zurückkehren zu diesem Commit, auch zu einem späteren Zeitpunkt. Somit kann jederzeit auf eine zurückliegende Version zurückgegriffen werden.[12]
Git ist also nicht dateispezifisch, da Änderungen unabhängig vom Dateityp erkannt werden. Prinzipiell kann jeglicher Inhalt gespeichert und verwaltet werden. Zudem werden Daten nicht gelöscht oder überschrieben - alle Änderungen werden in Git verwaltet und lassen sich bei Konflikten wiederherstellen, da diese in Form eines neuen Commits erfasst werden. Um trotzdem Speicherplatzeffizient zu sein, werden unveränderte Dateien lediglich referenziert.

[8] Vgl. Chacon/Straub: Pro Git, S. 13f.
[9] SHA-1 Hashes sind 40-stellige Zeichenketten aus hexadezimal Werten (0-9,A-F). Diese werden anhand des kompletten Inhalts einer Datei berechnet.
[10] Vgl. a. a. O.
[11] Vgl. a. a. O., S. 62f.
[12] Vgl. a. a. O.

3.1 Bisect

Wird ein Projekt über einen längeren Zeitraum bearbeitet, kann es vorkommen, dass sich ein Fehler über mehrere Commits hinweg im Quelltext befindet. Den ersten Commit in dem der Fehler aufgetreten ist zu finden erleichert das debuggen, doch es kann sehr anspruchsvoll und zeitintensiv sein diesen manuell herauszusuchen.
Lösungsansatz hier, kann ein *Bisect* sein, der auf dem Algorithmus der binären Suche basiert und damit auch bei vielen Commits eine kurze Laufzeit bietet. Mit dem Befehl `git bisect start` wird der Bisect gestartet. Mit dem Kommando `git biscect bad COMMITID` wird Git mitgeteilt, dass der genannte Commit fehlerhaft ist; mit `git bisect good COMMITID`, dass der ausgewählte Commit fehlerfrei ist. Anschaulich betrachtet wird ein linker Zeiger auf den letzten fehlerfreien Commit gelegt und ein rechter Zeiger auf den aktuellen fehlerbehafteten. Git wählt nun den Commit in der Mitte aus. Mit `git bisect good` oder `git bisect bad` teilt man Git nun mit, ob auch dieser mittlere Commit fehlerfrei oder fehlerbehaftet ist. Abhängig davon ob dieser Commit fehlerfrei ist oder nicht, wird der Suchraum halbiert und die Suche rekursiv auf den Commits davor oder danach fortgeführt, bis ein einzelner Commit als Fehlerquelle identifiziert werden kann. Dessen Änderungen und die eindeutige Commit Prüfsumme werden zurückgegeben. Davon ausgehend kann dann der Fehler behoben werden.

3.2 Branches

Im Gegensatz zu anderen VCSs sind die Branches und das Zusammenfügen dieser in Git sehr ressourcenschonend und schnell, so dass es sich anbietet verschiedene Branches für verschiedene Aufgaben zu nutzen.
Das es in der Praxis Anwendungen für mehrere parallel geführte Branches gibt, zeigt das Konzept des *progressive-stability branching*. Grundidee ist es einen stabilen Masterbranch zu führen, der den für den Release freigegeben Code enthält. Ein zweiter Branch ist der Entwicklungsbranch. Dieser beinhaltet Neuerungen, die nicht zwangsläufig stabil oder getestet sind. Sobald der Quelltext ausreichend getestet wurde und als stabil angesehen wird, kann der Code in den Masterbranch eingefügt werden.
Neben den bereits genannten Branches existieren noch weitere themenspezifische Zweige, in welchen spezielle Features oder Inhalte entwickelt werden können. Diese werden fortlaufend erstellt und in den Entwicklungszweig übernommen. Dieses Prinzip der mehrschichtigen Stabilitätssicherung lässt sich beliebig auf die verschiedensten Projekte anpassen.[13]

Der Grund für den einfachen Umgang mit Branches wird ersichtlich, wenn dessen interner Aufbau in Git betrachtet wird. Wird ein neuer Entwicklungszweig angelegt, so erzeugt Git einen Pointer, der auf das aktuelle Commit-Objekt zeigt. Um eine eindeutige Spezifizierung des aktuell ausgewählten Branches zu besitzen, existiert ein *Head*-Objekt - ein Zeiger, der auf das jeweilige Branch-Objekt weist. Wird ein Commit ausgeführt, so wird der Zeiger des Branch-Objekts auf den neuen Commit gesetzt. Dadurch entstehen Entwicklungszweige, die von einem ursprünglichen Commit abstammen. Ein Branch ist somit im Wesentlichen nur ein Pointer, der auf einen Commit zeigt.[14]

[13] Vgl. Chacon/Straub: Pro Git, 79-82.
[14] Vgl. a. a. O., 63.

3.3 Entwicklungszweige zusammenführen

Merges können in verschiedene Arten unterteilt werden. Soweit kein Parameter angegeben wird, der ein explizites Verfahren vorschreibt, entscheidet Git automatisch - je nach Ausgangssituation. Ausgelöst wird ein Merge durch den Befehl `git merge ZIELBRANCH`. Im folgenden Abschnitt werden die Unterschiede zwischen den verschiedenen Mergeverfahren näher erläutert.[15]

Zeigt ein Branch auf einen Commit, der Bestandteil eines anderen Branches ist, so ist der Erste gewissermaßen eine alte Version des Zweiten. In dieser Situation wird ein *Fast-Forward-Merge* angewendet. Dazu wird der zurückliegende Branch auf den gleichen Commit gerichtet, auf den der vorausliegende Branch zeigt. Bei dieser Art von Merge treten keine Konflikte auf, da die Dateiversionen aufeinander aufbauen.[16]
Im Gegensatz dazu findet ein *Basic-Merge* statt, wenn zwei verschiedene Entwicklungszweige einen gemeinsamen Ausgangspunkt (*Anker*) besitzen, also von einem einzelnen eindeutig bestimmbaren Commit abstammen. Unter diesen Umständen, erstellt Git einen neuen *Merge Commit*, der auf die beiden letzten Commits der jeweiligen Branches weist - ein sog. *3 Wege Merge*.[17]

Gerade in größeren Projekten kann es vorkommen, dass zwei verschiedene Anker existieren, also die Entstehung von zwei Branches auf zwei Commits zurückzuführen ist, wobei beide Anker sich unterscheiden. Es erscheint auf den ersten Blick sinnvoll an dieser Stelle, den in der Versionsgeschichte am weitesten zurückliegenden Ankerpunkt für einen Merge zu verwenden (so implementiert in der VCS Software Mericurial). Das kann allerdings in manchen Szenarien zu Fehlern führen, so dass bei einem Merge ein falscher Anker als Grundlage genommen wird und somit der Merge zu einem falschen Ergebnis kommt.
Daher ist das algorithmische Standardverfahren für einen Merge mit zwei Ankern der *recursive Merge*. Sobald es mehrere Möglichkeiten bei Auswahl des Ausgangspunkts gibt, wird ein virtueller Anker für den weiteren Verlauf erstellt. Dazu werden die beiden Ausgangspunkte ebenfalls mittels Merge zusammengeführt. Anhand dieses neu erstellten Ankers, wird dann der restliche Merge ausgeführt. Der Vorgang dabei gleicht einem 3 Wege Merge.[18]

Grundlegend lässt sich zwischen *impliziten* und *expliziten* Merges unterscheiden. Implizite Merges erzeugen kein neues Commit-Objekt, sondern wenden lediglich eine Reihe von Commits an. Im Gegensatz dazu, erzeugen letztere ein neues Commit-Objekt, das auf den Ursprungszweig weist.[19]

Git bietet des Weiteren die Möglichkeit mehrere Branches auf einmal zusammenzuführen. Der sogenannte *Octopus Merge* wird mit dem Befehl
`git merge BRANCH1 BRANCH2 BRANCH3 ...` ausgeführt. Konflikte müssen allerdings manuell aufgelöst werden. Hilfestellung leistet das Kommando `git merge -ours BRANCH1 BRANCH2 BRANCH3 ...`, da es die Lösung des ausgewählten Branches (der Branch auf den der Head zeigt) präferiert.[20]

[15] Vgl. Chacon/Straub: Pro Git, 71ff..
[16] Vgl. a. a. O., 72.
[17] Vgl. a. a. O., S. 69ff.
[18] Vgl. Santos: Plastic SCM blog: Merge recursive strategy 2011.
[19] Vgl. Bitbucket: Git Merge Strategy Options and Examples.
[20] Vgl. Laster, Brent: Professional Git. 1. Auflage. Birmingham, UK, UK: Wrox Press Ltd., 2016, S. 222.

Es besteht auch die Möglichkeit `git merge -s subtree BRANCH1 BRANCH2 BRANCH3 ...` zu verwenden. Dann wird bei Verfügbarkeit mehrerer Ausgangspunkte, analog dem rekursiven Merge, ein virtueller Anker gebildet.[21]

Tritt während des Mergevorgangs ein *Konflikt* auf, wird die Befehlsausführung umgehend unterbrochen. Auftreten kann ein solcher Fehler, wenn zwei Branches auf die gleichen Teile einer Datei verweisen. In Klartextdateien wäre das z. B. der zeitgleiche Zugriff auf eine Zeile im Text.[22]
Git unterbricht dann den Merge, zeigt die betroffenen Dateien an und fordert den Benutzer dazu auf den Konflikt manuell zu beheben. Dateien in diesem Status werden von Git als *Unmerged* bezeichnet. In dem lokalen Arbeitsverzeichnis, erstellt Git vier Dateien mit den Infixen *Local*, *Backup*, *Remote* und *Base*.
Die Local-Datei ist die des aktuell ausgewählten Branches; die Remote-Datei bezieht sich auf den Entwicklungszweig, mit dem gemerged werden soll. Git erstellt außerdem automatisch eine Backupdatei, sodass jederzeit alle Aktionen rückgängig gemacht werden können. Die Base-Datei ist der gemeinsame Anker der zwei Branches. Die Änderungen aus den jeweiligen Branches werden in die Originaldatei übernommen. Diese basiert zu Beginn auf der Base Datei. Git erkennt sobald alle Konflikte gelöst sind; `git status` bestätigt dies; und mit dem nächsten Commit wird der Merge vervollständigt.

Eine weitere Methode Entwicklungszweige zusammenzuführen ist es einen *Rebase* per `git rebase ZIELBRANCH (UMZUGSBRANCH)` auszuführen. Anstatt einen neuen Commit zu erstellen, der auf die Commits der beiden Ursprungsbranches zeigt, wird bei einem Rebase eine lineare Commit-Historie beibehalten. Dafür wird der Umzugsbranch, der Branch welcher in den Zielbranch eingegliedert werden soll, mit dem gemeinsamen Anker verglichen. Jeder Commit der sich von dem Ankerpunkt unterscheidet wird in einer separaten Datei gespeichert. Anschließend wird der Pointer des Umzugsbranches auf den letzten Commit des Zielbranches gesetzt. Danach wird jeder Unterschied von den Commits, die in den Dateien gespeichert wurden, als separater Commit ausgeführt. Der Umzugsbranch ist dann dem Zielbranch um x Commits vorraus, wobei x die Anzahl der Dateien der Unterschiede bezeichnet. Daher kann nach einem Rebase der Fast-Forward-Merge ausgeführt werden, so dass der Zielbranch den gleichen letzten Commit wie der Umzugsbranch hat.
Ein Nachteil dieser Technik ist, dass mögliche Konflikte für jeden Commit gesondert behandelt werden müssen. Zudem wird durch diesen Vorgang die Commit-Historie umgeschrieben. Demzufolge müssen Entwickler, die an dem Nebenzweig arbeiten, nach einem Rebase weitere lokale Änderungen erneut mergen. Daher sollte bei größeren, veröffentlichten Branches auf einen Rebase verzichtet werden. Besonders wenn die Arbeit aus den verschiedenen Branches aufeinander aufbaut.[23]

3.4 Konflikte und deren Lösung

Die Stärken der Software Git liegen vor allem im Lösen von Versionskonflikten. Unterschiede zwischen verschiedenen Dateiversionen werden einfach erkannt und es gibt eine Vielzahl von Tools, um aufgedeckte Konflikte zu beheben. Eines dieser Werkzeuge ist das Mergetool.
Aufgerufen mit `git mergetool --tool=TOOLNAME`, wird der Benutzer, bei entsprechender

[21] Vgl. Bitbucket: Git Merge Strategy Options and Examples.
[22] Vgl. Engbert: Merge-Konflikte beheben.
[23] Vgl. Riedel, Sven: Git kurz & gut. O'Reilly Media, 2014, Kapitel: Zweigpunkte mit Rebase verschieben.

Konfiguration, durch die Konfliktlösung geleitet und es werden Unterschiede aufgelistet. Das ist deutlich angenehmer und effizienter als die Konflikte mit einem reinen Texteditor zu beheben.
Ein simples Tool um Differenzen zu erkennen und zu lösen, ist *Meld*. Die grafische Oberfläche erlaubt die Übernahme der jeweiligen Änderungen mit ein paar Klicks zu realisieren. Weitere Tools und Algorithmen, die auf programmspezifischen Regeln basieren, können somit versuchen einen Teil der von Git nicht lösbaren Konflikte automatisiert zu lösen. Ein möglicher Ansatz ist es alle Änderungen aus der Local-Datei zu akzeptieren und diese anschließend zu testen. Wird ein Fehler ausgegeben, kann der Merge rückgängig gemacht werden und es muss eine manuelle Konfliktlösung erfolgen.[24]

Aufgrund der tief verankerten Hash-Funktionalitäten die Git verwendet, erkennt die Software schon kleinste Unterschiede zwischen zwei Versionen. Das ist besonders bei Leerzeichenkonflikten, die jedoch keinen Einfluss auf die Funktionalität eines Programms haben, mitunter sehr umständlich. Git bietet jedoch auch die Möglichkeit, mit den Flags `-Xignore-all-space` oder `-Xignore-space-change`, sogenannten Whitespace zu ignorieren. Die erste Option ignoriert Leerzeichen und Zeilenumbrüche komplett, wohingegen die Zweite lediglich ein oder mehrere hintereinander auftretende Zeichen als gleich ansieht.[25]

Ist die Branchstruktur eines Projekts nicht ausreichend durchdacht, stellt es sicherlich eine größere Herausforderung dar nur ausgewählte Textstellen zu übernehmen. Gerade wenn eine gewisse Dringlichkeit besteht eine Funktion oder Passage in den Masterbranch zu übernehmen. Oft sind dann große Konflikte schwerer zu überblicken und zu lösen. Mithilfe der Funktion `git cherry-pick COMMIT-HASH` lassen sich die Commits jedoch schrittweise durcharbeiten.[26] *Cherry-Pick* sollte nur verwendet werden, wenn ein Merge außer Frage steht. Denn der wesentliche Nachteil ist, dass bei diesem Verfahren doppelte Commit-Objekte entstehen und die ursprüngliche Commit Historie verloren geht.[27]

3.5 Push und Pull

Um effizient mit anderen Personen an einem Projekt zu arbeiten, empfiehlt es sich ein remote Repository einzurichten. Bei erstmaliger Anmeldung und Synchronisation mit dem Server via `git clone SERVERIP/PROJEKTNAME.git` klont Git das Repository und speichert es lokal im tracking Branch ab.[28]
Der Trackingbranch wird lokal mit origin/BRANCHNAME bezeichnet, um diesen von den local Branches unterscheiden zu können. Eine Synchronistation der tracking Branches erfolgt nicht automatisch, sondern durch Ausführung des Befehls `git fetch BRANCH`, welcher die tracking Branches auf den Stand der jeweiligen remote Branches bringt. Dies hat allerdings keinen Einfluss auf den local Branch. Um den local Branch mit dem tracking Branch zu mergen wird der Pull-Befehl `git pull BRANCH` verwendet. Der Pull-Befehl ist eine Kombination der Befehle Fetch und Merge. Somit werden zuerst die tracking Branches synchronisiert und anschließend wird der tracking Branch in den local Branch gemerged

[24] Vgl. Paolucci: Git: automatic merges with server side hooks (for the win!).
[25] Vgl. Chacon/Straub: Pro Git, S. 275.
[26] Vgl. Stachmann, Bjørn/Preissel, René: Git: Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows. dpunkt.verlag, 2017, Kapitel 9.5.
[27] Vgl. Gamage: Intro to Cherry Picking with Git.
[28] Vgl. Chacon/Straub: Pro Git, S. 26.

(vgl. hierzu Kapitel 2.3).[29]
Um anschließend den local Branch, der nun auf dem gleichen Stand wie der tracking Branch ist, dem remote Branch hinzuzufügen, wird der Befehl `git push BRANCH` verwendet.[30]

4 Fazit

Git, eine Versionsverwaltungssoftware, die nicht auf Dateitypen beschränkt arbeitet, ist ein dezentrales System, dass effizient funktioniert, sowohl hinsichtlich Arbeits- und Zeitaufwand für den Anwender, aber auch ressourcenbezogen. Versionsverwaltungssoftware unterstützt den Anwender beim Nachverfolgen von Änderungen und ermöglicht es zu einem gewünschten (vergangenen) Stand zurückzukehren. Durch ein Remote-Repository, das auf einem Server geführt wird, wird gerade die Arbeit in Teams deutlich vereinfacht. Dank dezentraler Ausrichtung des Systems, ist aber auch ein Arbeiten ohne Internetverbindung möglich.

Mit den Befehlen Push bzw. Pull, werden von Git erfasste Dateien hoch bzw. heruntergeladen. Nur Dateien, die zuerst mit dem Befehl `git add` für einen Commit vorgemerkt wurden und anschließend mit `git commit` commitet wurden, werden dabei berücksichtigt. Jeder Commit enthält auch eine Nachricht, in der eingetragen wird, welche Änderungen an den Dateien vorgenommen wurden.

Zentraler Bestandteil in Git sind außerdem die Branches. Sie stellen verschiedene Entwicklungszweige dar, die beispielsweise dafür genutzt werden können verschiedene Programmfunktionen zeitgleich entwickeln zu können. Um parallel entwickelte Funktionen zu einem Programm zusammenzuführen, lassen sich Branches mithilfe der Mergefunktion wieder vereinen. Es gibt verschiedene Mergearten, die von Git automatisch situativ ausgeführt werden. Durch diese Verfahren kann die Software einige Versionskonflikte automatisiert beheben.
Daher ist es nicht verwunderlich, dass eine der größten Stärken der Software Git die Lösung von Versionskonflikten ist. Differenzen in verschiedenen Dateiversionen werden in der Praxis aufgrund der tief verankerten Hash-Funktionen schnell und zuverlässig erkannt.

Für den Fall, dass sich fehlerhafter Codezeilen schon seit einem längeren Zeitraum, und damit über mehrere Commits hinweg, im Repository befinden, wurde der Bisect entwickelt. Damit lassen sich Fehler über mehrere Versionen hinweg verfolgen und beheben.

[29] Vgl. Chacon/Straub: Pro Git, S. 91.
[30] Vgl. a. a. O., S. 88f.

Literatur

Bitbucket: Git Merge. ⟨URL: https://de.atlassian.com/git/tutorials/using-branches/git-merge⟩ – Zugriff am 25.04.2019

Bitbucket: Git Merge Strategy Options and Examples. ⟨URL: https://www.atlassian.com/git/tutorials/using-branches/merge-strategy⟩ – Zugriff am 25.04.2019

Blischak, John D./Davenport, Emily R./Wilson, Greg: A Quick Introduction to Version Control with Git and GitHub. PLOS Computational Biology, 12 01 2016, Nr. 1, 1–18

Chacon, Scott/Straub, Ben: Pro Git. 2. Auflage. Berkely, CA, USA: Apress, 2014

Engbert, Ralf: Merge-Konflikte beheben. Aug 2013 ⟨URL: https://www.ralfebert.de/git/mergekonflikte-beheben/⟩ – Zugriff am 25.04.2019

Gamage, Passan: Intro to Cherry Picking with Git. Sep 2017 ⟨URL: https://www.previousnext.com.au/blog/intro-cherry-picking-git⟩ – Zugriff am 25.04.2019

Laster, Brent: Professional Git. 1. Auflage. Birmingham, UK, UK: Wrox Press Ltd., 2016

Loeliger, Jon/McCullough, Matthew: Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. O'Reilly Media, Inc., 2012

Paolucci, Nicola: Git: automatic merges with server side hooks (for the win!). May 2013 ⟨URL: https://www.atlassian.com/blog/git/git-automatic-merges-with-server-side-hooks-for-the-win⟩ – Zugriff am 25.04.2019

Ram, Karthik: Git can facilitate greater reproducibility and increased transparency in science. Source Code for Biology and Medicine, 8 Feb 2013, Nr. 1, 7

Riedel, Sven: Git kurz & gut. O'Reilly Media, 2014

Santos, Pablo: Merge recursive strategy. Sep 2011 ⟨URL: http://blog.plasticscm.com/2011/09/merge-recursive-strategy.html⟩ – Zugriff am 25.04.2019

Stachmann, Bjørn/Preißel, René: Git: Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows. dpunkt.verlag, 2017