Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Versionsverwaltung mit Git

vorgelegt von

Rana Mostafa

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Seminar Softwareentwicklung in der Wissenschaft

Studiengang:	Software System Entwicklung
Matrikelnummer:	7106452

Betreuer:	Hermann Lenhart

Hamburg, 2021-08-28

Inhaltsverzeichnis

1 Einleitung

Das folgende Kapitel bietet eine kompakte Einführung in Grundbegriffe Versionsverwaltung, drei Arten der Versionsverwaltung Systeme und warum Versionsverwaltung wichtig ist?

1.1 Was ist eine Versionsverwaltung? Welche Arten?

Versionsverwaltung ist ein System, welches Änderungen an einer oder einer Reihe von Dateien (z. B. Quellcode eines Programmes oder um Bilddateien in der Bildbearbeitung) über die Zeit hinweg protokolliert, sodass wir später auf eine bestimmte Version zurückgreifen können. Das heißt, Versionskontrollsysteme erlauben den Zugriff auf unterschiedliche Revisionen einer Datei.[1]

Es bestehen drei Arten der Versionsverwaltung. Die lokale Versionsverwaltung, das ist die einfachste Form. Etwas weiterverbreitet und weiterentwickelt ist schon die zentralisierte Versionsverwaltung, und die aktuell beste Stufe ist die verteilte Versionsverwaltung.

- Lokale Versionsverwaltung: Die erste Generation von Versionsverwaltung wird in der Regel nur auf einem einzelnen System eingesetzt und oft nur eine einzige Datei versioniert. Sie findet auch heute noch Verwendung in Büroanwendungen. Beispiel solche System ist SCCS.[1]
- Zentrale Versionsverwaltung: Ein großes Problem war der Zusammenarbeit mit anderen Entwicklern auf anderen Systemen und die Lösung dafür ist eine zentralisierte Versionsverwaltung. Diese Systeme ist als Client-Server-System aufgebaut, d.h. es gibt einen zentralen Server, auf dem die Dateien in sogenannten Repositorys festgehalten und verwaltet werden und es gibt auch Clients, die auf den Server zugreifen. Allerdings es besteht die Gefahr die Daten zu verlieren, wenn der Server ausfällt oder wenn die auf dem zentralen Server eingesetzte Festplatte beschädigt wird. Beispiel solche Systeme ist Subversion.[1]
- Verteilte Versionsverwaltung: Die verteilte Versionsverwaltung hingegen verwendet kein zentraler

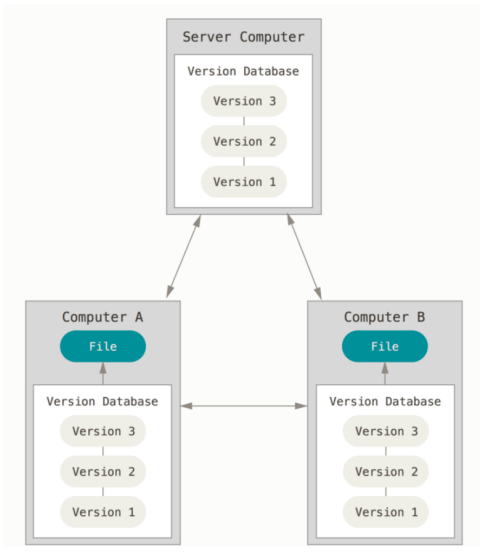

Figure 1.1: verteilte Versionsverwaltung[1]

Server. Dadurch ist es auch möglich zu arbeiten, wenn der Server ausfällt oder keine Verbindung möglich ist. Denn jeder Benutzer verfügt über eine lokale Kopie der vollständigen Historie des Projekts oder Repository.
Wie auf dem Figure 1.1 zu sehen ist, alle PCs sind im Prinzip genau gleich, die können Daten austauschen ohne, dass das Server PC benötigt wird. Der Server kann Austausch ermöglichen aber ist nicht nötig und wenn er ausfällt, wird nichts passieren, das Projekt kann trotzdem weitergehen. Beispiel solche Systeme ist Git.[1]

1.2 Warum Versionsverwaltung?

Versionsverwaltung ist besonders wichtig, denn ein solches System erlaubt es, einzelne Dateien oder auch ein ganzes Projekt in einen früheren Zustand zurückzuversetzen, nachzuvollziehen. Um herauszufinden, wer zuletzt welche Änderungen vorgenommen hat, die möglicherweise Probleme verursachen oder auch wer eine Änderung ursprünglich vorgenommen hat usw. Darüber hinaus durch die Erleichterung der Teamkommunikation, unterstützt diese Systeme die Zusammenarbeit am Projekt. Beispielsweise in der Softwareentwicklung, während sich ein Projekt weiterentwickelt, können Teams Tests durchführen oder Fehler beheben, um die Entwicklung zu optimieren.[3]

2 Versionsverwaltung mit Git

In diesem Kapitel werden wir die Geschichte von Git betrachten, wie Git Daten interpretiert sowie Informationen speichert und verwaltet und einige wichtige Fachbegriffe und grundlegende Befehle zur Arbeit mit Git vorstellen.

2.1 Git

Git ist eins der wohl bekanntesten verteilten Versionsverwaltung Systeme. Es steht als Open Source Software unter der GNU General Public License für Webdesigner, DevOps, Entwickler und Co. zur Verfügung.[1]
Insbesondere ermöglicht es Git, dass viele Leute gleichzeitig an einem größeren Softwareprojekt arbeiten können.[3] Git kann grundsätzlich für die Versionsverwaltung beliebiger Arten von Dateien verwendet werden, stellt somit ein wichtiges Werkzeug im Softwareentwicklungsprozess dar.[1]
Der Hauptunterschied zwischen Git und anderen Versionsverwaltungssystemen besteht in der Art und Weise wie Git die Daten betrachtet. Also wie Git Daten interpretiert sowie Informationen speichert und verwaltet.

Die meisten Versionsverwaltungssysteme erfassen Informationen, wie in Figure 5.1 dargestellt. Also als eine Reihe von Änderungen, die an einer ursprünglichen Datei vorgenommen wurden. Also Delta Format. Ein Delta ist zum Beispiel, wenn wir von einer Version von drei Daten starten und über den Zeitraum einfach nur die Unterschiede speichern und am Ende die gesamte Datei aufbauen.[1]

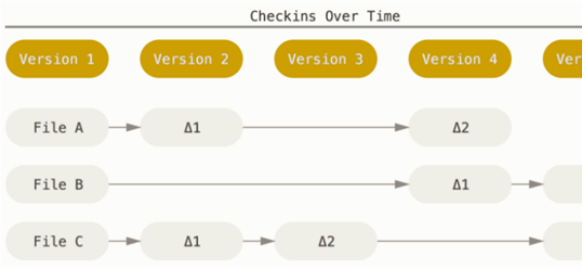

Figure 2.1: Speichern von Daten als Änderungen an einer Basisversion jeder Datei[1]

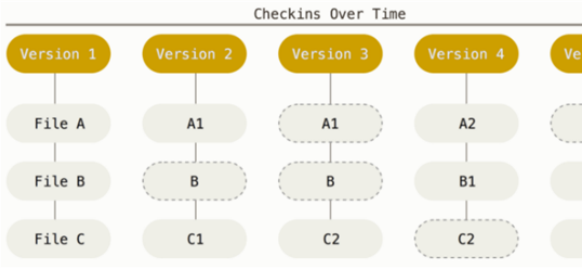

Figure 2.2: Speichern von Daten als Momentaufnahmen des Projekts im Laufe der Zeit[1]

Git guckt sich das etwas anders an, Git speichert lediglich die geänderte Datei und verweist auf die alten Daten. Das heißt, wenn wir eine zweite Version von unserem Projekt erstellen, haben wir nicht extra Speicherpunkte für Date, sondern nur einen Verweis auf die alte, unveränderte Version. Somit können wir über die Zeit ein kleines Dateiformat und Dateisystem aufbauen, was zu jedem Zeitpunkt die vollen Änderungen hat, ohne dass wir erst extra Daten zusammenbauen müssen, sehe Figure 5.2.
Darüber hinaus haben wir die Wahl, mit Git über eine Kommandozeile zu arbeiten, integriert in eclipse, intellj oder netbeans zu arbeiten oder über grafische Schnittstelle wie Beispielsweise sourcetree zu verwenden.[1]
In diesem Paper werden wir die Arbeitsweise von Git über die Kommandozeile demonstrieren.

2.2 Wichtige Befehle zur Arbeit mit Git

Einige wichtige Fachbegriffe und Befehle werden im Folgenden häufig vorkommen. Deswegen möchte ich nun einer kurzen Erläuterung dazu geben:

- Commit: ist ein Versionsstand in Git, also eine Aufzeichnung der Änderungen beispielsweise hinzufügen einer Datei oder Ändern einer Datei. Dadurch wird also eine neue Version des Projekts im Repository erzeugt.[3]

  Commit enthält sowohl die Veränderungen als auch Metadaten wie eine ID, den Autor der Veränderungen, Datum und Uhrzeit und eine Nachricht (Commit Message), die die Veränderungen beschreibt.[2]

- Clone: Mithilfe des Befehles können wir ein bestehendes Entfernte-Repository in ein lokales Verzeichnis kopieren. Also beschreibt ein Download eines gesamten Repositorys.[3]

- Pull: Der Pull-Befehl heißt Daten in ein lokales Repository mit Daten aus dem entfernten Repository abgleichen und Änderungen übernehmen. Pull lädt Änderungen von einem Repository in das lokale Repository, das heißt neue Versionen von einem anderen Repository übernehmen.[3]

- Push: Der Push-Befehl ist das Gegenteil vom Pull, neue Versionen auf ein anderes Repository übertragen. Also die im lokalen Repository vorliegenden Änderungen in ein entfernte zu übertragen.[3]

3 Git: Repository

3.1 Was ist eine Repository in Git?

Der zentrale Bestandteil beim Arbeiten mit Git ist das Repository. Ein Repository oder auch Git-Projekt umfasst die gesamte Sammlung von Dateien und Ordnern, die einem Projekt zugeordnet sind. Darüber hinaus gibt es bei Git, im Gegensatz zu den meisten anderen Versionskontrollsystemen," nicht nur die aktuelle Version des Repositorys, sondern gleich eine Kopie des ganzen Repositorys auf dem lokalen Rechner". Mit anderen Worten, Repository ist eine Datenbank, in der Git die verschiedenen Zustände jeder Datei eines Projekts über die Zeit hinweg ablegt. Insbesondere wird jede Änderung als Commit verpackt und abgespeichert.[3] Darüber hinaus können Git-Benutzer Daten mit anderem Benutzer auszutauschen. Dafür muss ein entferntes Repository erstellt werden, die von einem Git-Server verwaltet wird, beispielsweise github. In diesem Fall wird eine Verbindung zwischen unserem lokalen Repository und dem Online-Repository hergestellt.[1]

3.2 Lokales und remotes Arbeiten mit Git

Üblicherweise ist der Arbeitsablauf mit lokalem Git so, dass erstens Änderungen an den Dateien mittel Editor vorgenommen werden, anschließend zum Index hinzugefügt und am Ende per Commit in das Repository übertragen.
In den meisten Fällen werden wir mit Git wohl an remote Repository und mit anderem Benutzer arbeiten. Dazu muss zunächst eine Kopie des gewünschten Repositorys in einem neuen Ordner im Arbeitsverzeichnis erstellt werden. Zuerst wird eine Kopie des remotes Repository im Lokale Verzeichnis mit Hilfe der clone-Befehl erstellt werden, dann mit diesem Repository lokal gearbeitet werden. Damit die lokale Arbeit auch für andere Benutzer desselben Projekts sichtbar wird, wird dann die lokale Kopie gepusht.[1]

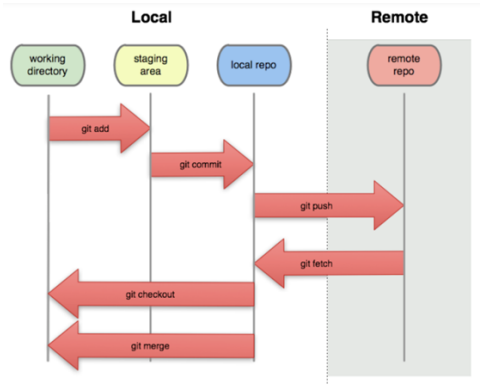

Figure 3.1: die Architektur von Git[5]

4 Git: Zustände und Breiche

Man unterscheidet in Git zwei Arten von Dateien: Die im Git gesichert: ”tracked” und nicht im Git gesichert: ”untracked”. Getrackte Dateien können drei Zustände haben:

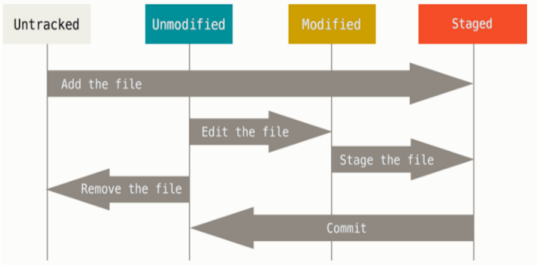

Figure 4.1: die drei Zustände in Git[1]

- Modified-Zustand bedeutet also, dass wir eine Datei haben, die verändert wurde und lokal auf deinem Rechner ist, aber noch nicht in Repository ist.
- Staged-Zustand bedeutet, dass diese geänderte Datei für die aktuelle Version markiert wurden, aber noch nicht übertragen sind.
- Committed-Zustand bedeutet dann, dass die markierte Datei endlich übertragen wurde und der letzte Zustand auch wirklich gespeichert ist.[1]

Das führt zu den drei Hauptbereichen eines Git-Projekts:

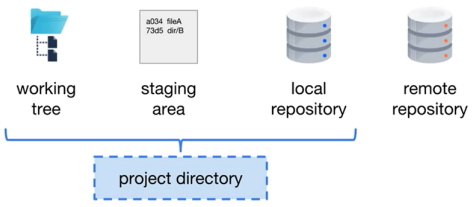

Figure 4.2: Breiche in Git[6]

- Working Directory (das Arbeitsverzeichnis) entspricht der Speicherort auf dem Computer, der die Verzeichnisse und Dateien eines einzelnen Commits enthält. Also Hier können die Dateien des Projekts angezeigt und bearbeitet werden.
- Staging Area (der Stagingbereich) enthält eine Liste von Dateien, die in das nächste Commit aufgenommen werden sollen.
- Das Repository (lokales und remotes Repository) umfasst alles, was im committed-Bereich liegt.

Darüber hinaus sind das Arbeitsverzeichnis, Staging-Bereich und lokales Repository in der Regel alle in einem einzigen Verzeichnis auf lokalen Computer enthalten und werden als Projektverzeichnis bezeichnet.[1]

5 Branching and Merging in Git

Ein weiteres wichtiges Konzept in Git ist Branch. Ein Branch oder auch Zweig ist eine Verzweigung zu einer neuen Version. Also ist eine Reihe von Commits, die mit dem letzten Commit im Zweig beginnen und bis zum ersten Commit des Projekts zurückverfolgen. Das Branching in Git ermöglicht uns, weiterzuarbeiten, ohne die Haupt-Entwicklungslinie zu verändern, also parallel andere Arbeiten auszuführen.[1]
Sobald ein neues Repository angelegt wird, erstellt Git einen Branch, den sogenannten Master. Dabei bezeichnet der HEAD Zeiger, eine Referenz auf den neuesten Commit im aktuellen Branch.
Darüber hinaus können wir in Git beliebig viele Branchs anlegen, um beispielsweise Releases vorzubereiten wie bei Release Branch im Figure 5.4, oder auch Features Branch um neue Features zu entwickeln[2].
Außerdem können wir die Branchs löschen oder ihn in das offizielle Projekt zusammenführen und das Zusammenführen von Branche in Git sprech ist Mergen.[1]

5.1 Git Merge

Man unterscheidet im Git zwischen zwei Fälle:

- Fast-Forward-Merge: Wenn der einzufügende Branch ein direkter Nachfolger des anderen Branches ist, führt Git einen sogenannten Fast-Forward-Merge aus. Wie auf dem Figure 5.1 zu sehen ist, Commit c2 wurde an den Master-Zweig vorgenommen, und dann wurde ein erstellt. Commit C4 wurde für den hotfix-Branch erstellt. Wenn hotfix-Branch bereit in den Master-Zweig zusammengeführt zu werden, kann eine Fast-Forward-Merge ausgeführt werden und nach dem Mergen enthalten beide Zweige genau die gleichen Commits, wie auf dem Figure 5.2 zu sehen ist.[1]

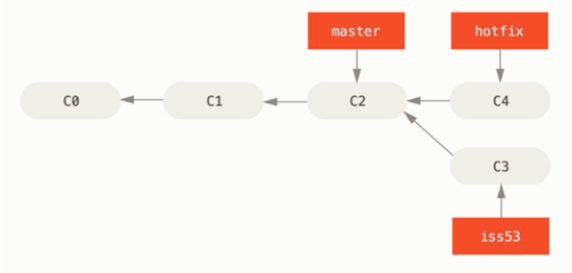

Figure 5.1: Hotfix-branch basierend auf Master[1]

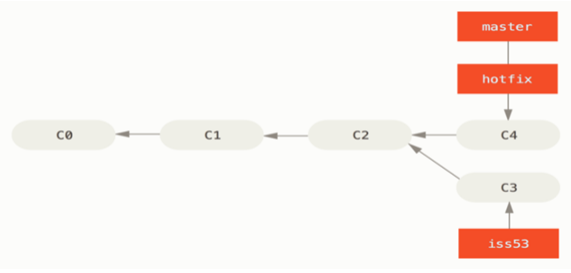

Figure 5.2: Das Fast-Forward-Merge in Git[1]

• Merge-Commit: Wenn wir uns nun Figure 5.3 anschauen, ist jetzt in diesem Fall eine Fast-Forward-Merge nicht möglich. Denn der Entwicklung bei c3 geht weiter und der Master Branch ist auf c4, weiß Issue Branch nicht von c4. Git erstellt dann beim Mergen nun einen neuen Commit c6, welcher nun auf die beiden Branches c4, c5 als seine Vorgänger verweist. Darüber hinaus kann es bei einem Merge-Commit mergen zu sogenannten Konflikten kommen, wenn z. B. dieselbe Datei in beiden Branches an derselben Stelle modifiziert wurde.[1]

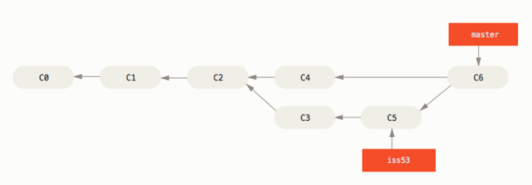

Figure 5.3: Merge-Commit in Git[1]

5.2 Git-flow-Workflow

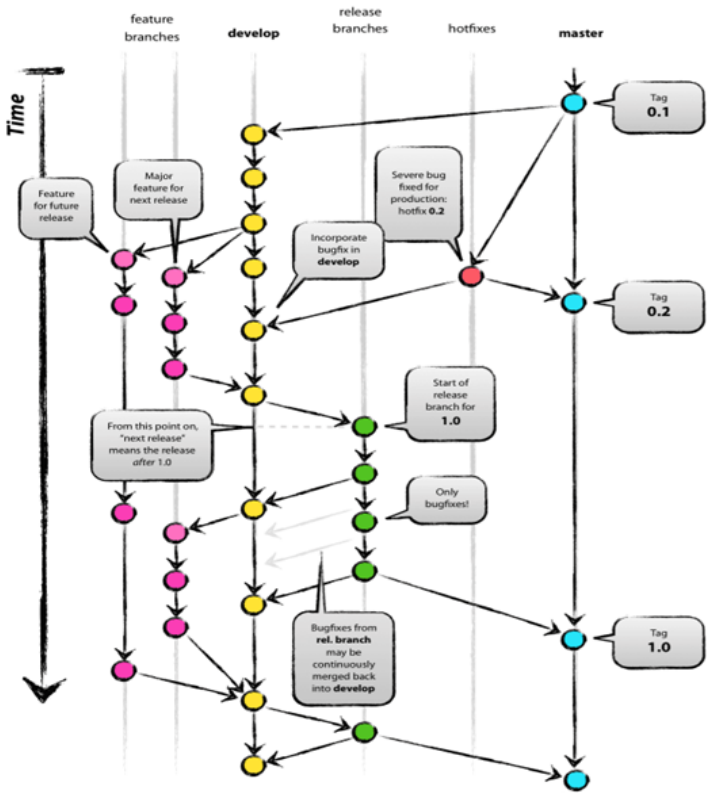

Figure 5.4: bekanntes Modell der Git-Workflow[4]

Ein Git-Workflow ist eine Empfehlung zur Verwendung von Git. Sie sind eine doku-

mentierte Verzweigungs- und Integrationsstrategie, an die sich Teams halten können, um jede Phase der Softwareentwicklung einfach zu verwalten.[4]
Figure 5.4 zeigt ein sehr bekanntes Modell der Git-Workflow Die Hauptzweige in diesem Workflow sind:
• Master: wird für leufenden stabielen reales reseviert
• Develop: wird parallel zum Master-Branch existiert, wenn der Quellcode im Develop Branch einen stabilen Punkt erreicht und zur Veröffentlichung bereit ist, sollten alle Änderungen irgendwie in den Master Branch zusammengeführt werden.
• Features: werden verwendet, um neue Features für die kommende oder zukünftige Release zu entwickeln.
• Hotfix: sind den Release-Branches sehr ähnlich, da sie auch dazu gedacht sind, sich auf eine neue Release vorzubereiten. Sie entstehen sofort auf einen unerwünschten Zustand einer Live-Produktionsversion zu reagieren.
• Release: unterstützen die Erstellung eines neuen Produktionsrelease. Darüber hinaus ermöglichen sie kleinere Fehlerkorrekturen und die Aufbereitung von Metadaten für ein Release.[4]

6 Git mit Kommandozeile(Getting Started)

In diesem Abschnitt möchte ich die grundlegenden Befehle zur Arbeit mit Git über die Kommandozeile mithilfe des folgenden Beispiels vorstellen. Dazu werden wir bereits die meisten der Befehle, die man für die alltägliche Arbeit mit Git benötigt.

Um ein Projekt mit Git zu starten, sollten ein paar Grundeinstellungen vorgenommen werden. Jeder Benutzer muss sich in Git mit einem Benutzernamen und E-Mail-Adresse konfigurieren. So werden user.name und user.email von Git bei Commits verwendet.[2]

Zur erst fangen wir an, eine Repos Verzeichnis mit dem Befehl **mkdir** für das lokale Repository zu erstellen. Dann wechseln wir Verzeichnisse in das Repo-Verzeichnis mit befahl **cd**. Diese Funktionen kann man mit der folgenden Syntax aufrufen:
$ mkdir repos
$ cd repos

Um das Projekt mit Git zu verwalten, werden wir das Befehl **init** verwenden. Der Befehl erzeugt im aktuellen Verzeichnis ein neues Unterverzeichnis mit dem Namen .git, welches ein Git Repository Grundgerüst enthält.[2]
$ git init
Initialized empty git repository in C: /User....

Git hat jetzt im Unterverzeichnis. git das lokale Repository angelegt.
Nachdem das Projekt initialisiert wurde, wollen wir eine Textdatei mithilfe des Befehls **touch** erstellen. Nach der Erstellung ist nun diese Textdatei zwar erstellt, aber noch nicht versioniert also befindet sich im Bereich working diroktory. Mit dem Befehl **add** fügen wir die Datei in staging area und dann mit Commit fügen wir die in eine neue Version des Projektes ein.[2]
$ touch Newfile
$ git add Newfile
$ git commit -m "new file added"
[master (root-commit) 8b2e852] new file added
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 Newfile

Wie schon im Abschnitt 5 erwähnt wurde, Branching ist ein sehr nützliches Feature von Git. Um ein Projekt parallel in verschiedene Richtungen zu verwalten.

Mit dem Befehl **branch** legen wir ein neues Branch an und mit dem Befehl **checkout** wechseln wir zum neuen Branch. Außerdem können wir mit demselben Befehl **branch** alle Branchs, die wir angelegt haben, zeigen lassen.

```
$ git branch featureX
$ git branch
 featureX
* master
$ git checkout featureX
Switched to breanch ‘featureX‘
```

Nach der Erstellung von featureX führen wir nun eine kleine Änderung auf diese Textdatei durch. Mit dem Befehl **diff** können wir den Unterschied zwischen Versionen eines Projekts ermitteln. In unserm Fall sehen wir, dass eine neue Zeile in Textdatei hinzugefügt wurde. Zunächst fügen wir die Änderung in staging area mit **add** und **commit**.[2]

```
$ git diff
diff –git a/Newfile b/Newfile
@@ -0,0 +1 @@
+Hallo
$ git add Newfile
$ git commit -m "Hallo added"
[featureX 79782c3] Hallo added
1 files changed, 1 insertions(+)
```

Möchten wir nun die beide Branches wieder zusammenführen, müssen wir zuerst in den Branch wechseln, welcher fortgeführt werden soll, dann den Befehl **merg** ausführen.[2]

```
$ git checkout master
Switched to breanch ‘master‘
$ git merge featureX
updating 07c7e43..79782c3
Fast-forward
 Newfile | 1 +
 1 files changed, 1 insertions(+)
 create mode 100644 Newfile
```

Das Löschen von Branch funktioniert mit dem Befehl **branch -d**.

```
$ git branch -d featureX
Deleted breanch featureX (was 79782c3)
```

7 Fazit

Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien, erleichtert die Arbeit an kleinen sowie an großen Projekten und erlaubt eine unkomplizierte Koordination selbst bei einer Vielzahl von Entwicklern. Dabei können Benutzer mit lokalen Kopien des Repositorys arbeiten. Ein Repository in Git enthält den Projektverlauf als Commits und ein Commit ist eine Snapshot des gesamten Projekts zu einem bestimmten Zeitpunkt. Alle Commits eines Projekts gehören zu einem Branch. Darüber hinaus können Änderungen dabei von einem Branch auch wieder in einen anderen einfließen. Standardmäßig gehören Commit zum Master-Branch und das Merging kombiniert die Arbeit mehrere Branches.

Bibliography

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

[2] HAENEL, Valentin; PLENZ, Julius: Git - Verteilte Versionskontrolle für Code und Dokumente. 2. Aufl. Open Source Press, 2015

[3] GITHUB GUIDES: Git Handbook, `https://guides.github.com/introduction/git-handbook/`. Online; Abgerufen am 10.08.2021

[4] DRIESSEN, Vincent. : A successful Git branching model. nvie.com, 2010 `https://nvie.com/posts/a-successful-git-branching-model/`, Online; Abgerufen am 21.08.2021

[5] EDUREKA!: Mastering Git and GitHub `https://www.edureka.co/blog/git-tutorial/`. Online; Abgerufen am 25.08.2021

[6] COURSERA, ATLASSIAN UNIVERSITY: Version Control with Git `https://www.coursera.org/lecture/version-control-with-git/git-locations-SMJIy`. Online; Abgerufen am 25.08.2021