Skript Versionskontrolle

BBZW, Modul 426

Patrick Bucher

22.02.2024

Inhaltsverzeichnis

Im Modul 426 nimmt die Versionskontrolle eine zentrale Rolle ein. Zwar wird auf das Thema Versionskontrolle nur zu Beginn des Moduls eingegangen. Für die tägliche Arbeit eines Softwareentwicklers ist der geübte Umgang mit der Versionskontrolle jedoch unerlässlich. Dieser Umgang soll auch eingeübt werden, indem in diesem Modul auch Aufgaben per Versionskontrolle eingereicht werden müssen.

In diesem Skript geht es im ersten Teil um das Thema Versionskontrolle im Allgemeinen. Zunächst

soll der Nutzen von Versionskontrollsystemen veranschaulicht werden. Weiter werden verschiedene Arten von Versionskontrollsystemen kurz betrachtet.

Im zweiten und dritten Teil geht es dann um die Versionskontrolle Git, welche sich als quasi-Standard etabliert hat. Zunächst werden die Geschichte, die grundlegenden Konzepte hinter Git sowie dessen Unterschiede zu anderen Versionskontrollen erläutert. Weiter werden einige Hinweise für die Installation und Konfiguration gegeben.

Im dritten Teil geht es um die Benutzung von Git zur Verwaltung von Quellcode. Die wichtigsten Befehle werden grundlegend erläutert – genug, um im Modul 426 bestehen zu können.

1 Versionskontrolle: Motivation

Stellen Sie sich vor, Sie arbeiten an einer Projektarbeit (z.B. ABU-Selbstvertiefungsarbeit oder Berufsmaturaarbeit). Nun löschen Sie aus Versehen eine ganze Seite aus einem Dokument. Wenn Sie dies sofort bemerken, können Sie das wieder rückgängig machen (`[Ctrl-Z]`). Wenn Sie dies jedoch nicht bemerken, und einfach weiterarbeiten und weiterhin speichern, wird die gelöschte Seite verlorengehen.

Selbstverständlich machen Sie als Informatiker systematische Backups. (Falls Sie das nicht machen, lernen Sie so hoffentlich Ihre Lektion!) Aber selbst wenn Sie tägliche Backups machen, können Sie im schlimmsten Fall noch die Arbeit eines ganzen Tages verlieren. Ein Backup reicht also nicht aus.

1.1 Handgestrickte Versionskontrolle

Eine weit verbreitete Lösung ist die sogenannte *handgestrickte Versionskontrolle*, bei der eine wichtige Datei oder ein wichtiges Verzeichnis einfach mehr oder weniger regelmässig (und teilweise mehrfach) kopiert wird.

Sieht ihr Arbeitsverzeichnis zunächst so aus:

```
Projektarbeit.docx
```

Wird es nach einer Weile Arbeit mit der Versionskontrolle “Marke Eigenbau” bald so aussehen:

```
AKTUELLSTE-Projektarbeit.docx
NEU-AKTUELLSTE-Projektarbeit.docx
Projektarbeit.docx
Projektarbeit-backup-Montag.docx
Projektarbeit-alt.docx
Projektarbeit-V3.docx
Projektarbeit-V3-besser.docx
```

```
Projektarbeit-neues-Design.docx
Projektarbeit-nach-Zwischenkorrketur.docx
```

Diese Lösung mag auf den ersten Blick schwachsinnig aussehen, ist aber in der Praxis erschreckend weit verbreitet. Im Zusammenhang mit Backups schützt sie schliesslich auch recht zuverlässig vor Datenverlust.

Die Probleme mit dieser "Versionskontrolle" sind jedoch einfach zu erkennen:

- **Schlechte Übersicht**: Bei so vielen Dateien, die eigentlich das gleiche beinhalten sollten, verliert man schnell den Überblick. Es ist auch nicht ganz einfach zu sagen, welches Artefakt den aktuellen Stand repräsentiert. Im Projektverlauf wird es zusehends schwerer zu erkennen, welche Artefakte noch gebraucht werden, und welche gelöscht werden können.
- **Inkonsequente Versionierung**: Bei welchen Ereignissen ("Zwischenkorrektur", "neues Design") oder zu welchen Zeitpunkten ("Montag", "alt") wird eine Datei zwecks Sicherung kopiert? Dies lässt sich nur schwer erkennen.
- **Schwere Rückverfolgung**: Welche Datei basiert auf welchem Stand? Ist das neue Design vor oder nach der Zwischenkorrektur erstellt worden? Sind die Zwischenkorrekturen schon mit der Version vom Montag gesichert worden? Die Dateien müssen geöffnet und manuell verglichen werden, um Antworten auf diese Fragen zu erhalten.
- **Keine Historisierung**: Welche Änderungen wurden in welcher Reihenfolge gemacht? Und warum wurde eine bestimmte Änderung gemacht? Wurde ein bestimmter Unterabschnitt versehentlich oder willentlich gelöscht?

1.2 Cloud Storage, Dokumentverwaltung

Viele der beschriebenen Probleme können mit einer Cloud-Storage-Lösung (z.B. DropBox, OneDrive) oder mit einer Dokumentverwaltung (z.B. Sharepoint) gelöst werden:

- Die Übersicht verbessert sich dank der automatischen Historisierung. Man sieht nur noch den aktuellen Stand im Arbeitsverzeichnis.
- Man kann zu einem früheren Zustand zurückkehren, ohne umständlich ein Backup einspielen zu müssen.
- Teilweise ist sogar eine Änderungshistorie mit optionalen Kommentaren vorhanden, womit man wichtige Änderungen beschreiben kann.

Solche Systeme sind zwar schon ein grosser Fortschritt gegenüber der manuellen Versionsverwaltung, haben aber auch ihre Probleme:

- Die Versionsübergänge sind immer noch implizit. Oft gibt es für jeden Speicher- und/oder Synchronisierungsvorgang eine neue Version.
- Die Rückverfolgung von Artefakten ist teilweise noch schwer. So kann man beim Kopieren und Verschieben schlecht nachvollziehen, auf welchem Stand ein Artefakt basiert.
- Die Historisierung ist oft nur sehr rudimentär, d.h. man kann sich nur auf einer linearen Zeitachse nach vorne und hinten bewegen. Änderungskommentare sind meist optional.

1.3 Dokumentverwaltung? Codeverwaltung!

Für eine schriftliche Projektarbeit mögen Lösungen wie DropBox oder Sharepoint die wichtigsten Anforderungen an eine Versionierung erfüllen. Für die Verwaltung von Quellcode sind diese Systeme jedoch völlig ungeeignet, denn Dokumente und Quellcode haben wichtige Unterschiede:

- *Dokumente* liegen oft als Binärdateien vor (z.B. `.docx` oder `.pdf`), welche sich nur manuell oder mit formatspezifischen Werkzeugen vergleichen und zusammenführen lassen.
- *Quellcode* hingegen liegt in einfachen Textdateien (*plain text*) vor, die sich weitgehend automatisiert und zeilenweise vergleichen und zusammenführen lassen.
- *Dokumente* werden oft als einzelne Datei bearbeitet. Für eine schriftliche Arbeit nimmt man zunächst Änderungen an einem Hauptdokument vor, und später arbeitet man etwa an einem Foliensatz zur Arbeit für eine Präsenation.
- *Quellcode* ist eine Reihe mehr oder weniger stark zusammenhängender Dateien. Änderungen an einer Datei sind oft nur im Zusammenhang mit Änderungen an anderen Dateien möglich und sinnvoll (Beispiel: Umbenennung einer Klasse, die an vielen Orten verwendet wird). Solche Änderungen über mehrere Dateien hinweg müssen als einzelne Änderung verwaltet werden können.

Für die Verwaltung von Quellcode benötigen wir somit mächtigere und spezialisierte Versionskontrollsysteme.

Zwar lässt sich Quellcode nicht wie einfache Dokumente verwalten. Umgekehrt ist es jedoch möglich, da Dokumente auch als *plain text* verfasst werden können (z.B. mit Markdown oder LaTeX).

1.4 Zentrale Versionskontrolle

Bei zentralen Versionskontrollsystemen werden alle zu verwaltenden Daten auf einem zentralen Server gespeichert. Wer an einem Projekt Änderungen vornehmen möchte, kann sich zu diesem Zweck den aktuellen Stand (einen sog. *Snapshot*) herunterladen, daran seine Änderungen vornehmen, und diese auf den zentralen Server zurückspielen. Wurde dieser Stand in der Zwischenzeit geändert, muss dieser Konflikt teilweise manuell behoben werden.

Zentrale Versionskontrollsysteme haben zahlreiche Vorteile:

- Es gibt einen einzigen (globalen), aktuellen Stand, über den sich alle Beteiligte an einem Projekt einig sind.
- Alle am Projekt Beteiligten sehen, was die anderen für Änderungen vornehmen.
- Auf dem zentralen Server können die Rechte zentral verwaltet und die Daten zentral gesichert werden.

Durch die Zentralisierung ergeben sich aber auch einige Nachteile:

- Der zentrale Server ist ein *Single Point of Failure*: Funktioniert er nicht mehr, ist die Arbeit an einem darauf verwalteten Projekt nur noch eingeschränkt möglich. Bei einem Datenverlust sind nur noch die auf den Clients vorhandenen Snapshots zu retten.

- Ist ein Client offline, kann er nur noch eingeschränkt am Projekt arbeiten. Änderungen können erst definitiv vorgenommen werden, wenn der Client wieder online ist.
- Konflikte, z.B. Änderungen an überschneidenden Codeabschnitten, werden erst beim Hochladen auf den zentralen Server bemerkt.

Trotz dieser Nachteile und aufgrund der obigen Vorteile erfreuten sich zentrale Versionskontrollsysteme lange Zeit einer grossen Beliebtheit. Prominente Beispiele zentraler Versionskontrollsysteme sind *CVS*, *Subversion* und *Perforce*.

1.5 Verteilte Versionskontrolle

Bei einem verteilten oder dezentralen Versionskontrollsystem werden die Daten komplett lokal verwaltet. Jeder Client verfügt über den kompletten Stand. Die Ablage auf einem oder mehreren Servern ist optional, aber der Normalfall bei Projekten, an denen mehrere Personen beteiligt sind.

Durch die dezentrale Struktur ergeben sich einige Vorteile:

- Die Arbeit ist auch offline ohne Einschränkungen möglich. Änderungen können natürlich nur online auf einen Server übertragen, aber offline bereits festgeschrieben werden.
- Der gemeinsame Austausch über einen zentralen Server ist möglich, aber nicht zwingend. Es können auch mehrere Server verwendet werden, die den gleichen Stand oder auch unterschiedliche Stände (selten empfehlenswert) repräsentieren.
- Da alle Dateien auf mehrere Clients und (optional) Server verteilt sind, ist Datenverlust so gut wie ausgeschlossen – bei Projekten mit vielen aktiven Mitarbeitenden selbst ohne systematische Backups. (Linus Torvalds, der die erste Version von Git entwickelte, behauptet, dass er keine Backups vom Linux-Quellcode mache, da der Code ohnehin schon auf der ganzen Welt verteilt sei.)

Die Nachteile gegenüber zentralen Versionskontrollsystemen sind gering bis praktisch vernachlässigbar:

- Aufgrund der Verteilung können Versionskonflikte an mehreren Orten auftreten. (Diese müssen früher oder später zwangsläufig gelöst werden.)
- Es müssen mehr Konzepte gelernt werden als bei einer zentralen Versionsverwaltung.
- Es gibt keine Ausreden mehr, wenn der Server für die Versionskontrolle ausfällt: weiterarbeiten kann man trotzdem. ;-)

Unter den (verteilten) Versionskontrollsystemen ist *Git* mittlerweile die mit Abstand am weitesten verbreitete Lösung. Weitere, weniger prominente verteilte Versionsverwaltungen sind *Mercurial*, *Bazaar* und *Darcs*.

Im Rahmen des Moduls 426 soll ausschliesslich Git zum Einsatz kommen. (Wer andere Vorlieben hat, kann auch mehrere Versionsverwaltungen gleichzeitig verwenden.)

2 Git: Grundlagen

Die Versionskontrolle Git kommt ursprünglich aus dem Linux-Umfeld.

Bei der Entwicklung des Linux-Kernels schickten sich die Entwickler zunächst Patches als Tarballs zu, welche mit Werkzeugen wie `diff(1)` und `tar(1)` erzeugt und mit `patch(1)` auf eine bestehende Codebasis angewendet worden sind.

Mit steigender Entwicklerzahl und wachsender Codebasis wurde dieser Ansatz jedoch immer schwerer praktizierbar. Deshalb stiegen die Linux-Kernel-Entwickler auf die kommerzielle Versionsverwaltung *BitKeeper* um, welche aufgrund einer Abmachung kostenlos verwendet werden durfte. Aufgrund von Meinungsverschiedenheiten zwischen den Linux-Entwicklern und dem Anbieter von BitKeeper lief diese Abmachung jedoch 2005 aus, sodass man für die Weiterentwicklug des Linux-Kernels eine neue Versionskontrolle benötigte.

Da Linus Torvalds – Initiator und oberster Entscheidungsträger des Linux-Kernels – mit den bestehenden Lösungen unzufrieden war, entwickelte er seine eigene Versionskontrolle: Git (engl. *Schwachkopf*). Dabei waren ihm die folgenden Ziele wichtig:

- **Geschwindigkeit**: Git sollte schnell sein, auch bei sehr grossen Projekten.
- **Einfaches Design**: Git sollte sich auf wenige Konzepte beschränken und dadurch einfach verständlich bleiben.
- **Unterstützung für nicht-lineare Entwicklung**: Git sollte in der Lage sein mit tausenden von Entwicklungszweigen (engl. *Branches*) umzugehen und schnell zwischen diesen zu wechseln.
- **Vollständige Verteilung**: Git sollte die dezentrale Entwicklung des Linux-Kernels mit einer dezentralen Architektur abbilden.
- **Effizienz** im Umgang mit grossen Projekten: Git sollte auch für riesige Codebasen schnell funktionieren.

Git wurde Mitte 2005 zum ersten Mal für den Release eines Linux-Kernels verwendet (Release 2.6.12) und Ende 2005 in der Version 1.0.0 freigegeben. Die Version 2.0.0 erschien Mitte 2014. Mittlerweile ist Git zum de-facto Standard unter den Versionskontrollen geworden, d.h. man muss heutzutage gute Gründe finden, wenn man etwas anderes als Git verwenden will.

Weitere Hintergrundinformationen zur Entwicklung von Git liefert Linus Torvalds in einem interessanten Tech Talk.

Fazit: Git hat die obengenannten Ziele nicht nur erreicht, sondern bei weitem übertroffen. Git skaliert nicht nur “nach oben”, indem es für riesige Projekte effizient funktioniert, sondern auch “nach unten”, indem man auch kleinste Projekte mit wenig Zusatzaufwand mit einer Versionskontrolle verwalten kann.

2.1 Unterschiede zu traditionellen Versionskontrollen

Git unterscheidet sich signifikant von vormals populären Versionskontrollen wie CVS oder Subversion. Dies hat einerseits damit zu tun, dass Git *dezentral* und die meisten konventionellen Versionskontrollen *zentral* funktionieren. Die Unterschiede betreffen jedoch auch die interne Arbeitsweise:

- Die meisten Versionskontrollen (fortan VCS, engl. für *Version Control System*) arbeiten *dateiorientiert*, d.h. sie zeichnen die Versionsgeschichte von einzelnen Dateien auf. Git hingegen arbeitet mit kompletten Zwischenständen der Codebasis, d.h. mit sogenannten *Snapshots*.
- Die meisten VCS speichern die *Unterschiede* bzw. *Änderungen* ab, welche an einzelnen Dateien vorgenommen worden sind. Git hingegen speichert komplette *Zwischenstände* ab.

Zusammenfassend kann man sagen, dass traditionelle VCS die Versionsgeschichte als eine *Reihe von Änderungen* verwalten, während bei Git die Versionsgeschichte aus einer *Reihe von Zuständen* besteht.

Traditionelle VCS können mit einem Bankkonto verglichen werden, das bei einem Betrag von 0 startet. Die Bank speichert die einzelnen Transaktionen für dieses Bankkonto ab, und kann aufgrund dieser den Saldo (d.h. den aktuellen Kontostand) berechnen.

Git kann mit einem Dateisystem verglichen werden, das zu bestimmten Zeitpunkten komplett auf einen externen Datenträger gesichert wird.

Traditionelle VCS *speichern Änderungen* und *berechnen Zustände*.

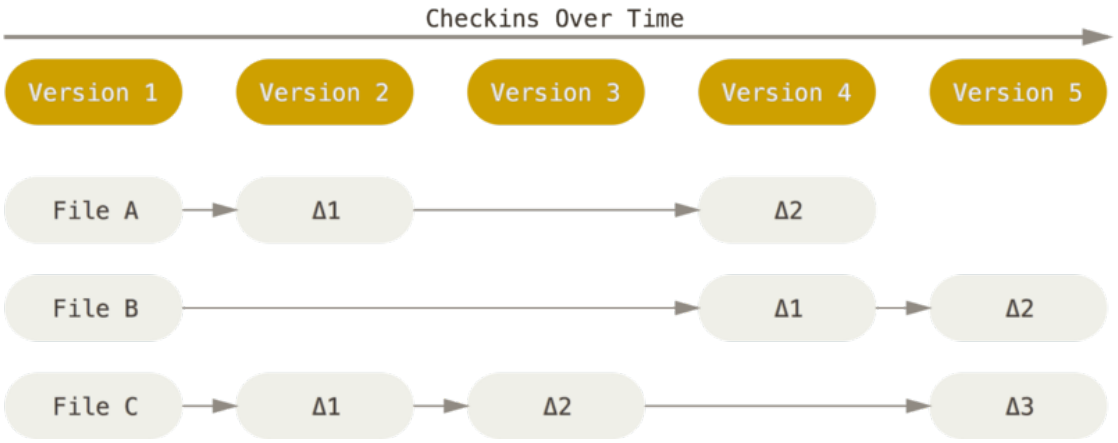

Abbildung 1: Traditionelle VCS mit der Versionsgeschichte als eine Reihe von Änderungen (https://git-scm.com/book/de/v2/Erste-Schritte-Was-ist-Git%3F)

Git *speichert Zustände* und *berechnet Änderungen*.

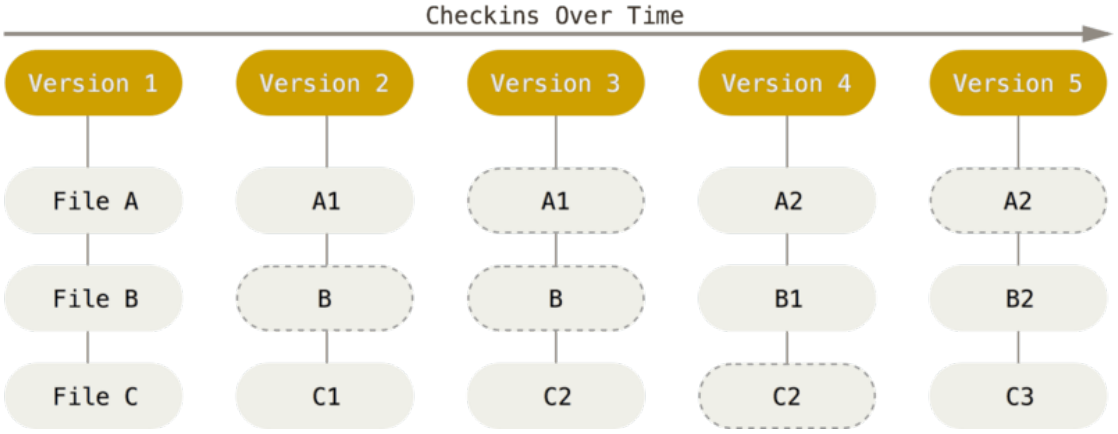

Abbildung 2: Git mit der Versionsgeschichte als eine Reihe von Zuständen (https://git-scm.com/book/de/v2/Erste-Schritte-Was-ist-Git%3F)

2.2 Wichtige Merkmale von Git

Ein Verzeichnis, das von der Versionskontrolle Git verwaltet wird, bezeichnet mal als *Repository*. Ein Repository ist eine Datenbank (im weitesten Sinn), auf die lesend und schreibend zugegriffen werden kann.

Git berechnet für jeden Zustand eine eindeutige Prüfsumme über alle Dateien hinweg, die sich in einem Repository befinden. Diese wird mit dem Hashing-Algorithmus *SHA1* – `sha1sum(1)` – berechnet und besteht aus 40 hexadezimalen Stellen und sieht z.B. so aus:

```
353edf7e8d13b6535b26bcb150c47fbc6e50c0c4
```

Werden Anpassungen an den Dateien im Repository vorgenommen, ändert sich auch die Prüfsumme. Dadurch kann Git Änderungen, aber auch korrupte Daten (etwa durch einen schadhaften Datenträger verursacht) feststellen.

Diese Prüfsumme wird auch verwendet, um Zustände eindeutig zu referenzieren. (Rein theoretisch könnten unterschiedliche Zustände die gleiche Prüfsumme ergeben, die Wahrscheinlichkeit hierfür ist jedoch so gering, dass man es in der Praxis ignorieren kann.)

Git arbeitet *additiv*, d.h. die meisten Änderungen am Repository fügen Daten hinzu. (Wird eine Datei aus dem Repository entfernt, bleibt diese unter einer früheren Version immer noch verfügbar.) Dadurch wird zwar das Repository tendenziell immer grösser, dafür können aber keine Änderungen mehr verlorengehen, die einmal festgeschrieben worden sind.

2.2.1 Bereiche

Git unterscheidet zwischen drei Bereichen, die verschiedene Zustände von Dateien repräsentieren:

1. Im **Arbeitsverzeichnis** befinden sich unveränderte oder veränderte Dateien. Dies ist das Verzeichnis, mit dem ein Benutzer gewöhnlich interagiert.
2. Im **Staging-Bereich** befinden sich Änderungen, die für eine permanente Speicherung vorgemerkt worden sind.
3. Das **`.git`-Verzeichnis** repräsentiert das eigentliche Repository. Es enthält die unwiderruflich festgeschriebenen Änderungen, die via Arbeitsverzeichnis und Staging-Bereich ins Repository gelangt sind. Weiter enthält das `.git`-Verzeichnis auch Metadaten für das Repository. Die verwalteten Dateien werden als spezielle Objektdaten abgelegt.

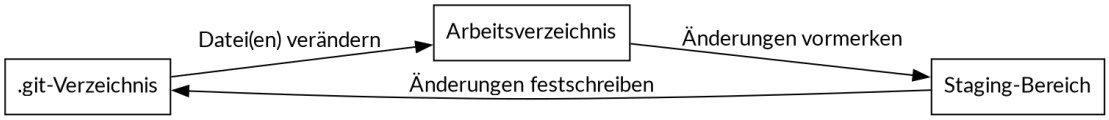

Abbildung 3: Die drei Bereiche von Git mit Zustandsübergängen

2.3 Git-Clients

Das Git-Projekt stellt einen offiziellen Git-Client für die Kommandozeile zur Verfügung: `git(1)`. Dieser unterstützt *alle* Git-Befehle und gilt als Referenzimplementierung.

Oftmals basierend auf der offiziellen Programmbibliothek zu Git, libgit2, gibt es zahlreiche GUI-basierende Clients. Die meisten integrierten Entwicklungsumgebungen (IDEs) verfügen über eine integrierte oder per Erweiterung installierbare Git-Unterstützung. Diese unterstützen jedoch nur eine Untermenge des Funktionsumfangs von Git.

Auch wenn im Modul 426 nur ein Bruchteil der Funktionalität von Git benötigt wird, werden nur Instruktionen für den offiziellen Kommandozeilen-Client von Git gegeben. (Solche Anweisungen sind eindeutig, präzise, plattformunabhängig und benötigen wesentlich weniger Platz als eine Reihe von Screenshots. GUIs ändern sich oft, Befehle bleiben in der Regel gleich.) Wer den `git`-Befehl kennt, wird diesen auch auf einem GUI-Client wiederfinden, was jedoch umgekehrt nicht gilt.

Wer lieber mit einem GUI-Client arbeitet, darf dies selbstverständlich tun. Instruktionen und mögliche Prüfungsfragen beziehen sich aber auf den Git-Client für die Kommandozeile.

2.4 Installation von Git

Die Installation von Git unterscheidet sich je nach verwendeter Plattform. Auf Debian-basierenden Linux-Distributionen wie Ubuntu lässt sich `git` einfach mittels `apt` installieren:

```
# apt install git
```

Bzw.

```
$ sudo apt install git
```

Für Windows empfiehlt sich die Installation des Git-Clients mitsamt Bash-Shell von der offiziellen Webseite.

Auf macOS kann `git` z.B. mit Homebrew sehr einfach installiert werden:

```
$ brew install git
```

Eine umfassende Installationsanleitung findet sich im frei verfügbaren eBook *Pro Git* (Kapitel 1.5: Erste Schritte).

2.5 Konfiguration von Git

Git kann auf drei verschiedenen Ebenen konfiguriert werden: systemweit, pro Benutzer und pro Repository. Die Konfiguration kann entweder interaktiv über Kommandozeilenbefehle erfolgen (`git config`) oder direkt in einer Konfigurationsdatei. Die Befehle und Konfigurationsdateien unterscheiden sich je nach Ebene:

Bereich	Befehl	Konfigurationsdatei
systemweit	`git config --system`	`/etc/gitconfig`
pro Benutzer	`git config --global`	`~/.gitconfig` bzw. `~/.config/git/config`
Repository	`git config`	`.git/config`

Die Tilde (~) steht dabei für das Home-Verzeichnis des Benutzers, also `/home/[benutzer]` unter Linux oder `C:\Users\[benutzer]` unter Windows.

Unter Windows befindet sich die systemweite Konfigurationsdatei im Installationsverzeichnis von `git`. Diese muss jedoch in der Regel nicht angepasst werden.

Einstellungen auf einer tieferen Ebene überschreiben die Einstellungen der höheren Ebene. So kann man bei Bedarf pro Repository spezielle Einstellungen vornehmen, während für die anderen Repositories noch die system- und benutzerspezifischen Einstellungen gelten.

2.5.1 Initiale Konfiguration

Da Git bei jeder Änderung den Namen und die E-Mail-Adresse des Benutzers festschreibt, müssen diese Angaben gleich zu Beginn definiert werden, sinnvollerweise auf Benutzerebene:

```
$ git config --global user.name 'Vorname Nachname'
$ git config --global user.email 'vorname_nachname@sluz.ch'
```

Wer eine Präferenz für einen bestimmten Texteditor auf der Kommandozeile hat (z.B. `ed`, `vi`, `vim`, `emacs`, `nano`), kann diesen auch gleich konfigurieren:

```
$ git config --global core.editor [Texteditor]
```

Zum Beispiel für den Texteditor `nano`:

```
$ git config --global core.editor nano
```

Der Texteditor wird geöffnet, wenn eine mehrzeilige *Commit Message* (Kommentar zum Festschreiben einer Änderung) verfasst werden soll. (Mehr dazu später.)

2.5.2 Konfiguration einsehen

Mit `git config` können nicht nur Einstellungen definiert, sondern auch angezeigt werden. Der folgende Befehl listet die im aktuellen Kontext (d.h. inner- oder ausserhalb eines Repositories) gültigte Konfiguration auf:

```
$ git config --list
user.email=patrick.bucher@sluz.ch
user.name=Patrick Bucher
core.editor=vi
```

Einstellungen können auch einzeln unter Angabe ihres Namens abgefragt werden:

```
$ git config user.email
patrick.bucher@sluz.ch
```

2.6 Hilfe zu Git

Git bietet ein internes Hilfesystem an, das mit dem `help`-Befehl angezeigt werden kann:

```
$ git help
```

Zu jedem Befehlsverb (z.B. `config`) kann eine spezifische Hilfe angezeigt werden, wozu es folgende beiden Möglichkeiten gibt:

```
$ git help [Befehlsverb]
$ git [Befehlsverb] --help
```

Also zum Beispiel:

```
$ git help config
$ git config --help
```

Auf Unix-Systemen lässt sich die Hilfe auch über die Manpages anzeigen:

```
$ man git-[Befehlsverb]
```

Beispielsweise:

```
$ man git-config
```

Eine Liste der Manpages zu Git lässt sich mit `apropos git` anzeigen.

Am Ende dieses Dokuments befinden sich einige Links auf weiterführende Materialien zum Thema Git.

3 Git: Verwendung

Ist Git konfiguriert und installiert, kann es dazu verwendet werden, wozu es entwickelt worden ist: zur Verwaltung von Quellcode. Diesen verwaltet Git bekanntlich in sogenannten Repositories. So eines muss zunächst bereitgestellt werden, bevor damit gearbeitet werden kann.

3.1 Ein Repository bereitstellen

Es gibt zwei Möglichkeiten ein Repository bereitzustellen: Man kann ein neues, leeres Repository erstellen, was bei neuen Projekten sinnvoll ist, oder man kann ein bestehendes Repository von einem zentralen Server klonen.

3.2 Ein Repository erstellen

Ein Repository wird mit dem Befehl `git init` erstellt. In der Regel wird hierzu zunächst ein neues Verzeichnis (hier: `project`) angelegt, in welchem dann der Quellcode zu liegen kommt:

```
$ mkdir project
$ cd project
$ git init
```

Hierdurch wird das Unterverzeichnis `.git` im Verzeichnis `project` erstellt, welches das eigentliche Git-Repository ausmacht. Das `project`-Verzeichnis ist das eigentliche Arbeitsverzeichnis des Git-Repositories.

3.3 Ein Repository klonen

Ein bestehendes Repository kann mit dem Befehl `git clone` von einem sogenannten *Remote* (mehr dazu später) auf den lokalen Computer kopiert werden, z.B.:

```
$ git clone https://code.frickelbude.ch/m426/git-demo.git
```

Eine komplette Kopie des Repositories befindet sich anschliessend im Verzeichnis `git-demo`. Möchte man dieses Verzeichnis anders benennen, kann man `git clone` einen optionalen Namen mitgeben (hier: `my-git-demo`):

```
$ git clone https://code.frickelbude.ch/patrick/git-demo.git my-git-demo
```

In diesem Beispiel wird als Protokoll `https` verwendet. Oftmals kommt auch `ssh` zum Einsatz.

Da die lokale Kopie das gesamte Repository enthält, kann es benutzt werden um notfalls als Restore auf einem Server eingespielt zu werden. (Zur Erinnerung: Git ist eine *dezentrale* Versionskontrolle.)

Das Repository `git-demo` wird im Folgenden für die Demonstration der wichtigsten Git-Befehle verwendet. Es empfiehlt sich, die Schritte auf dem eigenen Computer nachzuvollziehen.

3.4 Zustände von Dateien

Innerhalb des Arbeitsverzeichnisses eines Repositories kann eine Datei zwei Zustände haben: entweder wird sie durch Git verwaltet (*tracked*) oder nicht (*untracked*). Durch Git verwaltete Dateien können weiter drei Zustände einnehmen:

1. *unmodified* (unverändert): die Datei liegt im gleichen Zustand vor, wie sie zuletzt in das Repository festgeschrieben worden ist.
2. *modified* (verändert): die Datei wurde im Arbeitsverzeichnis verändert, die Änderung aber noch nicht für die permanente Speicherung im Repository vorgemerkt.
3. *staged* (vorgemerkt): die Datei wurde im Arbeitsverzeichnis verändert, anschliessend für die permanente Speicherung im Repository vorgemerkt, aber seither nicht mehr verändert.

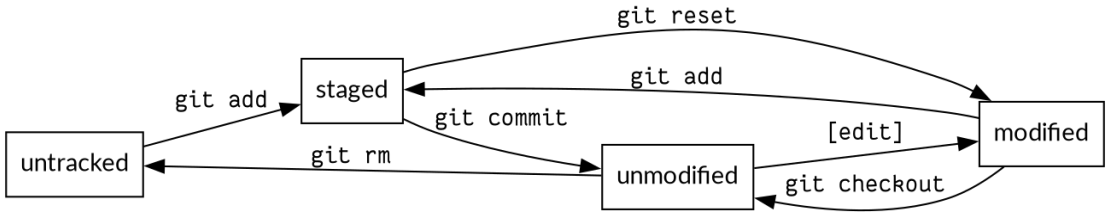

Abbildung 4: Lebenszyklus von Dateizuständen

Einige wichtige Übergänge zwischen diesen Zuständen sind in der Abbildung *Lebenszyklus von Dateizuständen* ersichtlich.

3.4.1 Zustände anzeigen

Der aktuelle Zustand der Dateien im Repository kann mit `git status` angezeigt werden:

```
$ cd git-demo
$ git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   addition.c
        new file:   params.c

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
        modified:   addition.c
        modified:   hello.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        LICENSE
```

Diese Ausgabe ist sehr ausführlich und instruktiv. Für den häufigen Gebrauch ist jedoch eine kompaktere Ausgabe, die man mit dem Parameter `--short` erhält, wesentlich übersichtlicher:

```
$ git status --short
AM addition.c
 M hello.c
A  params.c
?? LICENSE
```

Diese Ausgabe mutet zunächst recht kryptisch an. Dem Dateinamen (rechts) gehen zwei Spalten voraus, welche Auskunft über den aktuellen Zustand geben:

1. links: der Zustand im Staging-Bereich
2. rechts: der Zustand im Arbeitsverzeichnis

In diesem Beispiel nehmen diese Spalten drei verschiedene Werte an, welche die folgende Bedeutungen haben:

1. `?`: *untracked*
2. `A`: *staged* (`A` wie "added")
3. `M`: *modified*

Es gibt noch einige weitere Zustände, die aber hier nicht von Belang sind. Stellt man die Ausgabe in einer übersichtlichen Tabelle dar, wird die Interpretation dadurch etwas erleichtert:

Staging-Bereich	Arbeitsverzeichnis	Datei
`A`	`M`	`addition.c`
	`M`	`hello.c`
`A`		`params.c`
`?`	`?`	`LICENSE`

Was bedeuten nun diese Zustände jeder Datei, bzw. welche Zustandsänderungen sind auf der jeweiligen Datei erfolgt?

- `addition.c`

  - Die Datei wurde verändert (`[edit]`).
  - Die Datei wurde hinzugefügt (`git add`).
  - Die Datei wurde erneut verändert (`[edit]`).

- `hello.c`
  - Die Datei wurde verändert (`[edit]`).

- `params.c`
  - Die Datei wurde verändert (`[edit]`).
  - Die Datei wurde hinzugefügt (`git add`).

- `LICENSE`:
  - Die Datei wurde angelegt und ist *untracked*.

Die Datei `addition.c` hat drei Zustände gleichzeitig: je einen im Repository, im Arbeitsverzeichnis und im Staging-Bereich. Es ist wichtig, diese unterschiedlichen Zustände in den verschiedenen Bereichen auseinanderhalten zu können. Nur so kann man sich sicher sein, welchen Effekt weitere Aktionen im Repository haben werden.

3.5 Dateien hinzufügen

Eine im Arbeitsverzeichnis veränderte Datei kann mittels `git add` zum Staging-Bereich hinzugefügt werden:

```
$ git add hello.c
```

Je nach verwendeter Kommandozeile können auch mehrere Dateien mit einem einzigen Befehl und einem sogenannten *Glob-Pattern* auf einmal hinzugefügt werden, z.B. in der (Git) Bash:

```
$ git add *.c
$ git add utils/*.{h,c}
```

Vorsicht ist geboten beim Hinzufügen *aller* Dateien im Arbeitsverzeichnis:

```
$ git add *
```

Hierdurch können auch Binärdateien (z.B. Kompilate) ins Repository gelangen, die dieses nur unnötig aufblähen.

3.5.1 Dateien ignorieren

Zum Glück bietet Git einen Mechanismus um bestimmte Dateien anhand ihres Namens (bzw. eines Musters desselben) von einem Repository auszuschliessen. Dies ist u.a. für Binärddateien, d.h. für Kompilate wie ausführbare Dateien (`*.exe`), Libraries (`*.dll`, `*.so`) oder Byte-Code (`*.class`, `*.pyc`) sinnvoll. Auch sicherheitsrelevante Informationen wie Tokens oder Passwörter sollen nicht *unverschlüsselt* in einem Repository abgelegt werden.

Um Dateien von einem Repository auszuschliessen, kann eine Datei namens `.gitignore` im Arbeitsverzeichnis (und bei Bedarf auch in verschiedenen Unterverzeichnissen) definiert werden. Die Regeln innerhalb dieser Datei beziehen sich auf Pfade ausgehend von dem Verzeichnis, das die Datei `.gitignore` beinhaltet. Hier ein Beispiel für eine `.gitignore`-Datei mit erläuternden Kommentaren:

```
*.class    # keine .class-Dateien
bin/       # keine bin-Verzeichnisse
/tmp       # kein tmp-Verzeichnis direkt im Arbeitsverzeichnis
*.out      # keine .out-Dateien
!audit.out # Ausnahme für die Datei audit.out
```

Beim Verfassen einer `.gitignore`-Datei lohnt es sich die verfügbaren Regeln genauer zu studieren (`git help gitignore`). Auf GitHub gibt es auch zahlreiche Vorlagen für verschiedene Programmiersprachen und Frameworks.

Für Dateien, die einmal ins Repository gelangt sind, gelten die Regeln in `.gitignore` nicht. Von daher lohnt es sich, ein passendes `.gitignore` schon zu Projektbeginn aufzusetzen.

3.6 Inhaltliche Änderungen anschauen

Gibt `git status` Auskunft darüber, welche Dateien verändert worden sind, kann man sich mit `git diff` die inhaltlichen Änderungen ausgeben lassen. So können verschiedene Zustände miteinander verglichen werden.

Der Befehl `git diff` kann in verschiedenen Varianten verwendet werden:

- `git diff` vergleicht den Zustand des Arbeitsverzeichnisses mit dem Zustand des Staging-Bereichs.
- `git diff --staged` (oder das synonyme `git diff --cached`) vergleicht den Zustand des Staging-Bereichs mit dem festgeschriebenen Zustand des Repositories.
- Mit `git diff [commit-hash]` kann der aktuelle Zustand des Repositories mit einem früheren Zustand verglichen werden.
- Mit `git diff [old-commit-hash] [new-commit-hash]` können zwei frühere Zustände des Repositories miteinander verglichen werden.

Für letztere beiden Befehle muss nicht der ganze Commit-Hash angegeben werden, die 4-8 ersten Zeichen davon reichen in der Regel aus, um die Commits eindeutig zu referenzieren.

3.6.1 Ausgabe von `git diff`

Eine Ausgabe von `git diff` kann folgendermassen aussehen:

```
diff --git a/hello.c b/hello.c
index cb76401..19611e3 100644
--- a/hello.c
+++ b/hello.c
@@ -1,8 +1,6 @@
-#include <stdio.h>
-
 int main(int argc, char *argv[])
 {
-    printf("Hello, World!\n");
+    puts("Hello, World!");

     return 0;
 }
```

Die ersten vier Zeilen beziehen sich auf die interne Arbeitsweise von `git diff` und können für unsere Betrachtung ignoriert werden.

Die folgende Zeile `@@ -1,8 +1,6 @@` gibt den Bereich der Änderungen an:

- Im alten Zustand sind die Zeilen 1 bis 8 betroffen.
- Im neuen Zustand sind die Zeilen 1 bis 6 betroffen.

Es folgt eine Darstellung der inhaltlichen Änderungen. Dabei hat jede Zeile ein Präfix:

- `-` markiert gelöschte Zeilen.
- `+` markiert hinzugefügte Zeilen.
- `[leer]` markiert unveränderte Zeilen.

Wird eine Zeile geändert, erscheint dies als eine gelöschte Zeile (alter Zustand) und hinzugefügte Zeile (neuer Zustand).

3.6.2 Änderungsstatistik anschauen

Möchte man die Änderungen nicht inhaltlich betrachten, sondern nur einen Überblick über das Ausmass der Änderungen erhalten, kann man sich mit `git diff` und dem Parameter `--shortstat` eine Änderungsstatistik ausgeben lassen:

```
$ git diff --shortstat
 1 file changed, 1 insertion(+), 3 deletions(-)
```

Diese Ausgabe hat folgende Bedeutung:

- Es wurde eine Datei verändert.
- Dabei wurde eine Zeile hinzugefügt…
- …und drei Zeilen wurden gelöscht.

3.7 Änderungen festschreiben

Hat man die gewünschten Änderungen vorgenommen und überprüft, können diese mit `git commit` *unwiderruflich* im Repository festgeschrieben (d.h. *committed*) werden.

Jede Änderung soll mit einer erläuternden *Commit Message* kommentiert werden. Hierzu gibt es zwei Möglichkeiten:

Möchte man einen kurzen, einzeiligen Kommentar verfassen, kann man diesen per Parameter `-m` direkt mitgeben:

```
$ git commit -m 'use puts instead of printf'
```

Mehrzeilige Kommentare werden mit dem Texteditor verfasst, der mittels `git config core.editor` konfiguriert worden ist. Dieser erscheint folgendermassen, sobald man `git commit` ausführt:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       modified:   hello.c
#
```

Die mit `#` beginnenden Zeilen sind Kommentare und gelangen nicht in die Commit Message. Im unteren Bereich werden die Dateien aufgelistet, die vom Commit betroffen sind.

Speichert (hierzu wird eine teporäre Datei verwendet) und schliesst man den Texteditor, wird der Commit mit der eingegebenen Message durchgeführt. Beispiel für eine mehrzeilige Commit Message:

```
Use puts() instead of printf(), which has some benefits:
1. no explicit newline needed at the end
2. stdio.h is no longer needed
3. the program runs faster and becomes smaller
```

Form und Inhalt einer Commit Message werden oft für eine Organisation oder einzelne Projekte festgelegt. Häufig soll die Commit Message eine Referenz auf einen bestimmten Vorgang enthalten (Bug, Story, Anforderung, Incident, Ticket, etc.), der i.d.R. als Präfix angegeben wird, z.B.:

```
$ git commit -m 'Bug-#731: close file descriptor after writing finished'
```

3.7.1 Änderungen zu früh, falsche Commit Message?

Da `git commit` Änderungen unwiderruflich im Repository festschreibt, sollte man vorher besser genau prüfen, ob man alle gewollten Änderungen im Staging-Bereich hat, und ob die Commit Message diese Änderungen korrekt und sinnvoll beschreibt.

Geht einmal aus Versehen eine Änderung vergessen, oder fällt einem erst nach dem Commit etwas negativ an der Commit Message auf, kann man diesen Fehler korrigieren, sofern es in der Zwischenzeit keine weiteren Commits gab.

Hierzu kann `git commit` mit dem Parameter `--amend` ("nachbessern") aufgerufen werden. Dabei werden alle Änderungen im Staging-Bereich im Repository festgeschrieben, wozu eine neue Commit Message verfasst werden kann: entweder mit dem Parameter `-m` (einzeilige Message neu verfassen), oder aber mit dem Texteditor, der die alte Commit Message für mögliche Anpassungen anzeigt:

```
$ git commit -m 'removed pointless inkluds' # ups, Tippfehler...
$ git add main.c                            # ups, Datei vergessen...
$ git commit --amend -m 'removed pointless includes'
```

Der letzte Commit wird dabei überschrieben; der nachgebesserte Commit verschmilzt mit diesem.

3.7.2 Staging-Bereich umgehen

Nimmt man grössere Änderungen vor, ist es oft hilfreich wenn man die einzelnen Dateien nach und nach für einen grösseren Commit im Staging-Bereich sammelt. Bei kleineren Änderungen (z.B. Tippfehler korrigieren) kann dieser aber schon einmal guten Gewissens umgangen werden.

Mit dem Parameter `-a` können veränderte Dateien direkt committed werden, sofern diese schon von Git getrackt werden. Hier gilt es zu beachten, dass *alle* veränderten und getrackten Dateien festgeschrieben werden!

```
$ git status --short
 M addition.c
?? newprog.c
$ git commit -a -m 'simplified implementation'
```

Die bereits getrackte Datei `addition.c` wurde in den Commit aufgenommen. Die ungetrackte Datei `newprog.c` blieb jedoch vom Commit ausgenommen:

```
$ git status --short
?? newprog.c
```

3.8 Dateien entfernen

Löscht man eine von Git verwaltete Datei, etwa mit dem `rm`-Befehl oder über einen Dateimanager, bleibt diese zunächst im Repository bestehen. Verwendet man hingegen den Befehl `git rm`, wird die Löschung der Datei sogleich im Staging-Bereich vorgemerkt:

```
$ git rm hello.c
$ rm params.c
$ git status --short
D  hello.c
 D params.c
```

Man erkennt, dass die mit `git rm` gelöschte Datei `hello.c` im Staging-Bereich für die Löschung vorgemerkt ist, während die mit `rm` gelöschte Datei `params.c` bloss im Arbeitsverzeichnis fehlt. Letztere müsste mit einem weiteren Aufruf von `git rm` zum Löschen im Staging-Bereich vorgemerkt werden.

Beim Löschen von getrackten Dateien ist `git rm` dem einfachen `rm` in der Regel vorzuziehen.

3.8.1 Dateien im Staging-Bereich entfernen

Wurde eine Datei bereits dem Staging-Berich hinzugefügt, kann diese nicht mehr ohne Weiteres gelöscht werden:

```
$ git add newprog.c
$ git rm newprog.c
error: the following file has changes staged in the index:
newprog.c
(use --cached to keep the file, or -f to force removal)
```

Die Fehlermeldung verweist auf zwei Möglichkeiten. Einerseits kann die Datei explizit mit dem `--cached`-Parameter aus dem Staging-Bereich entfernt werden:

```
$ git rm --cached newprog.c
```

Andererseits kann die Datei auch aus dem Staging-Bereich und gleichzeitig aus dem Arbeitsverzeichnis gelöscht werden:

```
$ git rm -f newprog.c
```

3.9 Dateien umbenennen (verschieben)

Beim Umbennen bzw. beim Verschieben von Dateien sollte man darauf achten, dass die Änderungsgeschichte dieser Datei verlorengehen kann:

```
$ mv hello.c helloworld.c
$ git status --short
 D hello.c
?? helloworld.c
```

Die Datei unter dem alten Namen `hello.c` wird als im Arbeitsverzeichnis gelöscht angesehen, während die Datei unter dem neuen Namen `helloworld.c` als neu, d.h. untracked angesehen wird.

Verwendet man `git mv` anstelle von `mv`, wird die Umbenennung/Verschiebung als solche gehandhabt, und die Historisierung der Datei bleibt bestehen:

```
$ git mv hello.c helloworld.c
$ git status --short
R  hello.c -> helloworld.c
```

3.10 Dateizustand zurücksetzen

Es kann vorkommen, dass eine Datei versehentlich mit `git add` dem Staging-Bereich hinzugefügt wird:

```
$ git add *
$ git status --short
M  params.c
A  params.exe
```

Durch das Glob-Pattern `*` wurde nicht nur der Code (`params.c`), sondern auch das Kompilat (`params.exe`) dem Staging-Bereich hinzugefügt. Binärdateien sollen aber meistens nicht ins Repository gelangen.

Mit `git reset` kann die Datei wieder aus dem Staging-Bereich entfernt werden. Die Änderung im Arbeitsverzeichnis (sprich: das kompilierte Programm) bleibt jedoch bestehen:

```
$ git reset params.exe
$ git status --short
M  params.c
?? params.exe
```

Wie man erkennen kann, ist `params.c` weiterhin vorgemerkt, während `params.exe` wieder als neue Datei gilt. (Hier wäre es sinnvoll, das Muster `*.exe` zu `.gitignore` hinzuzufügen.)

Verwendet man den Parameter `--hard`, werden auch Änderungen im Arbeitsverzeichnis rückgängig gemacht und gehen so unwiderruflich verloren.

3.11 Änderungen verwerfen

Wurden im Arbeitsverzeichnis Änderungen vorgenommen, die sich als unnötig oder falsch erwiesen haben, können diese mit `git checkout` wieder zurückgenommen werden.

```
$ git status --short
 M helloworld.c
 M params.c
```

Im obigen Beispiel war die Änderung an `params.c` gewollt, diejenige an `helloworld.c` jedoch nicht. Unter der Angabe des Dateinamens können ungewollte Änderungen wieder rückgängig gemacht werden:

```
$ git checkout -- helloworld.c
$ git status --short
 M params.c
```

Die Änderung an `params.c` bleibt hingegen bestehen. Der leere Parameter `--` bezeichnet das Ende der Git-spezifischen Parameter, bzw. der Beginn von Dateiparametern. Da `git checkout` Commit-Hashes und keine Dateien als Argumente erwartet, ist diese Abgrenzung nötig. (Der Dateiname `123abcdef` könnte auch der Beginn eines Commit-Hashes bezeichnen.)

3.12 Zustand wiederherstellen

Soll ein frührer, festgeschriebener Zustand einer Datei wiederhergestellt werden, wird der Commit Hash des jeweiligen Zustands benötigt. (Mehr dazu im nächsten Abschnitt):

```
$ git checkout 7b6c08aa -- params.c
$ git status --short
M  params.c
```

Die Datei `params.c` wurde auf den Stand vom Commit `7b6c08aa` und im Staging-Bereich vorgemerkt. Ohne Angabe einer Datei werden alle Dateien zurückgesetzt:

```
$ git checkout 7b6c08aa
```

Dies ist sinnvoll, wenn man z.B. den Code einer früheren Version kompilieren möchte. Zurück zum aktuellen Stand gerät man folgendermassen:

```
$ git checkout -
```

3.13 Versionsgeschichte einsehen

Damit man sinnvoll zwischen verschiedenen Zuständen hin- und herspringen kann, muss man sich zunächst einmal einen Überblick über die Commit History verschaffen. Hierzu gibt es den Befehl `git log`, welche die Versionsgeschichte des Repositories anzeigt:

```
$ git log
commit aef74c545202548853bc09bf7afa1860f4262323
Author: Patrick Bucher <patrick.bucher@mailbox.org>
Date:   Sat Jul 10 11:34:23 2021 +0200

    new hello world program in C

commit e97e48b1067060cb77f743143c3a17fc082dd6cc
Author: Patrick Bucher <patrick.bucher@mailbox.org>
Date:   Sat Jul 10 11:34:04 2021 +0200

    initial commit: README
```

Die einzelnen Commits werden absteigend von neu nach alt angezeigt. Ist die Ausgabe länger als der Bildschirm hoch, erfolgt diese Ausgabe in einem Pager wie `less(1)`, der eine komfortable Navigation erlaubt.

Standardmässig werden beim Aufruf von `git log` die folgenden Informationen dargestellt:

- `commit`: der 40-stellige SHA-1 Hash des Commits
- `Author`: der Autor mit Name und E-Mail-Adresse
- `Date`: das Datum des Commits
- Kommentar: die Commit-Message (ein- oder mehrzeilig)

3.13.1 Nützliche Parameter zur Steuerung der Ausgabe

Zu `git log` gibt es eine Reihe von Parametern, welche die Ausgabe kontrollieren. Hier sollen nur einige wenige exemplarisch aufgelistet werden:

- `-[n]`: nur die neuesten `n` Commits anzeigen
  - `git log -3`
- `-p`: die Änderungen jedes Commits anzeigen
  - `git log -p`
- `--stat`: Statistiken zu Änderungen anzeigen (Zeilen +/- pro Datei und total)
  - `git log --stat`
- `--pretty`: "schönere" Ausgabe, mit verschiedenen Varianten (`oneline`, `short`, `medium`, `full`, `fuller`, etc.)
  - einzeiliges Format: `git log --pretty=oneline`
  - benutzerdefiniertes Format: `--pretty=format:"%h %cn"`
  - siehe `git help log` (nach `--pretty` suchen)
- `--graph`: ASCII-Graph (v.a. bei Branches interessant)
  - `git log --graph`
- `--since`/`--after` und `--until`/`--before`: Eingrenzung auf Zeitraum
  - `git log --since 2021-07-10T15:00 --until '1 hour ago'`
- `--name-only`: Zeigt Namen betroffener Dateien an

Datumsangaben können absolut oder relativ angegeben werden, und viele verschiedene Formate werden akzeptiert. Ob eine Datumsangabe korrekt interpretiert wird, kann mit `date(1)` getestet werden:

```
$ date -d '2021-06-04 21:30'
Fri Jun  4 09:30:00 PM CEST 2021
$ date -d '3 weeks ago'
Sun Jun 20 05:33:37 PM CEST 2021
$ date -d 'last tuesday'
Tue Jul  6 12:00:00 AM CEST 2021
```

```
$ date -d 'Tante Elfriedes Geburtstag'
date: invalid date ‘Tante Elfriedes Geburtstag’
```

Mit `gitk` gibt es ein grafisches Werkzeug zur Ausgabe der Versionsgeschichte, siehe dazu `git help gitk`.

3.14 Zentrale Repositories

Obwohl Git dezentral funktioniert, braucht man für die Zusammenarbeit mit anderen Entwicklern zentral gelagerte Repositories: *Remote Repositories* oder kurz *Remotes*. Diese werden auf einem zentralen Server oder oftmals auf einem Service wie GitHub oder GitLab gehosted.

Wird ein Repository geklont, verfügt man standardmässig über ein Remote namens `origin` (der “Ursprung” oder die “Quelle”). Die Remotes können mit dem Befehl `git remote` aufgelistet werden.

```
$ git clone https://code.frickelbude.ch/patrick/git-demo.git
$ cd git-demo
$ git remote
origin
```

Mit dem Parameter `-v` werden detailliertere Informationen zu jedem Remote ausgegeben, sprich woher Daten geladen werden (`fetch`), und wohin Daten geschrieben werden (`push`), wozu üblicherweise die gleiche URL verwendet wird:

```
$ git remote -v
origin  git@code.frickelbude.ch:patrick/git-demo.git (fetch)
origin  git@code.frickelbude.ch:patrick/git-demo.git (push)
```

3.14.1 Zusätzliche Remotes definieren

Manchmal ist es nützlich oder notwendig, wenn pro Repository mehrere Remotes definiert werden. Dies ist etwa nützlich, wenn man ein Repository von einem Server oder Dienst auf einen anderen verschieben oder z.B. einen öffentlichen Mirror (nur mit Leserechten) für ein sonst privates Repository einrichten möchte.

Zusätzliche Repositories werden mit `git remote add` definiert. Als Parameter wird der gewünschte Name des Remotes und dessen URL benötigt:

```
$ git remote add github git@github.com:patrickbucher/git-demo.git
```

Die Details eines Remotes können mit `git remote show` angezeigt werden:

```
$ git remote show origin
```

Remotes können mittels `git remote rename` umbenannt werden:

```
$ git remote rename origin the-one-and-only
$ git remote
the-one-and-only
```

3.15 Remotes synchronisieren

Das zentrale Repository wird nicht nur zum initialen Klonen verwendet, denn lokale Änderungen sollen später wieder auf dieses geschrieben werden, und Änderungen anderer Entwickler sollen davon gelesen werden. Das lokale Repository wird mit dem Remote *synchronisiert*.

Hierzu gibt es drei Befehle: `git push` (hochladen), `git fetch` und `git pull` (beide herunterladen).

3.15.1 Auf ein Remote hochladen

Mittels `git push` können die im lokalen Repository festgeschriebenen Änderungen auf das Remote übertaragen werden. Hierzu kann man ein bestimmtes Remote angeben:

```
$ git push github
```

Wird das Remote weggelassen, wird das Standard-Remote (meistens `origin`) verwendet.

Wurde das zentrale Repository in der Zwischenzeit geändert, können Versionskonflikte auftreten. Die Auflösung solcher Konflikte wird an dieser Stelle nicht behandelt.

3.15.2 Von einem Remote herunterladen

Mit den Befehlen `git fetch` und `git pull` können die Änderungen von einem Remote heruntergeladen werden. Der Unterschied zwischen den beiden Befehlen ist, dass `git fetch` nur die Metadaten von einem Remote abholt. Mit `git pull` werden auch die inhaltlichen Änderungen heruntergeladen und auf das lokale Repository angewandt.

```
$ git fetch origin
$ git pull github
```

Wird kein Remote spezifiziert, wird wiederum das Standard-Remote verwendet.

3.16 Versionen markieren/benennen

Da selbst abgekürzte Commit Hashes schwer zu merken sind, sollten wichtige Zwischenstände besser mit einer benutzerdefinierten Markierung (engl. *"tag"*) versehen werden. Oftmals ist eine semantische Versionsnummer (siehe Semver.org) sinnvoll, wie z.B. `v0.0.1` oder `v3.4.1`.

Ein Tag wird mit dem Befehl `git tag` erzeugt und bezieht sich standardmässig auf den aktuellsten Commit:

```
$ git tag v0.0.1
```

Mit dem Parameter `-a` kann ein Tag zusätzlich annotiert werden, z.B. mit einer zusätzlichen *Tag Message* (Parameter `-m`):

```
$ git tag -a v0.0.1 -m 'initial public release'
```

Ein Tag kann auch nachträglich unter der Angabe eines Commit Hashes für einen früheren Zwischenstand vergeben werden:

```
$ git tag -a v0.0.0 e97e48 -m 'tag initial version'
```

3.16.1 Versionen anzeigen

Bestehende Tags können mit `git tag` aufgelistet werden:

```
$ git tag
v0.0.1
v0.0.2
```

Die Reihenfolge sagt dabei nichts über den zeitlichen Verlauf der Tagging-Operationen aus, sondern erfolgt alphabetisch.

Detaillierte Informationen zu einem bestimmten Tag können mit `git show` angezeigt werden:

```
$ git show v0.0.2
tag v0.0.2
Tagger: Patrick Bucher <patrick.bucher@mailbox.org>
Date:   Sat Jul 10 16:43:00 2021 +0200

major bugfixes
```

3.16.2 Versionen teilen

Lädt man lokale Änderungen mit `git push` auf das zentrale Repository hoch, werden die Tags dabei nicht automatisch übertragen. Dies muss mit jeder Push-Operation explizit erfolgen:

```
$ git push origin v0.0.1
```

Alternativ kann der Parameter `--tags` verwendet werden, womit *alle* lokalen Tags auf das Remote hochgeladen werden:

```
$ git push origin --tags
```

Hierbei ist Vorsicht geboten, besonders wenn man "persönliche" Tags definiert hat (z.B. `v0.3.7-a.k.a-the-last-version-that-did-not-suck`).

3.17 Branches

Mit den bisher kennengelernten Git-Befehlen lässt sich Software nur *linear* weiterentwickeln. D.h. es gibt einen Zeitstrahl beginnend beim Initialisierungsdatum des Repositories, auf dem die einzelnen Commits der Reihe nach zu liegen kommen.

Dieses lineare Modell bildet die Realität in der Softwareentwicklung jedoch nur ungenügend ab. Betrachten wir folgendes Beispiel:

3.17.1 Fallbeispiel: Refactoring

Sie arbeiten an einem grösseren *Refactoring*, d.h. Sie bauen bestehenden Code um, ohne dadurch dessen Funktionalität zu verändern, sondern einfach um ihn besser zu strukturieren oder einfacher lesbar zu machen. Sie haben schon Dateien im Staging-Bereich abgelegt, aber noch keinen Commit gemacht, da sich die Arbeit über einen längeren Zeitraum (Stunden, Tage, Wochen) hinzieht.

Nun erhalten Sie plötzlich eine dringende Anfrage, dass Sie eine kleine kosmetische Änderung an ihrem Code vornehmen müssen, beispielsweise um einen Tippfehler in einer Beschriftung zu korrigieren. Die Änderung selber ist leicht gemacht, doch können Sie diese nicht per Commit festhalten, da bereits Änderungen ihres unfertigen Refactorings für den nächsten Commit vorgemerkt sind.

Die kleinere Änderung muss also auf sich warten lassen, bis die grössere Änderung fertig ist. Solche Situationen lassen sich vermeiden, indem man sogenannte *Branches* (d.h. Zweige) verwendet.

3.17.2 Einen neuen Branch erstellen

Ein neuer Befehl lässt sich mit dem Befehl `git checkout -b` erstellen, indem man einen neuen Branchnamen definiert:

```
$ git checkout -b refactoring
```

An Ihrem Arbeitsverzeichnis hat sich dadurch nichts geändert. Es wurde jedoch ein neuer Branch (neben `master`) erstellt, und dieser ist nun aktiv, wie die Ausgabe von `git branch` (Anhand des vorangestellten Asterisk `*`) zeigt:

```
$  git branch
  master
* refactoring
```

Sie können nun an diesem Branch bereits Änderungen im Rahmen Ihres Refactorings vornehmen und auch bereits als Commit festhalten, ohne dadurch den `master`-Branch zu verändern: dieser bleibt stabil.

3.17.3 Zu einem anderen Branch wechseln

Angenommen, Sie müssen nun am `master`-Branch eine kleinere Änderung vornehmen, können Sie folgendermassen zu diesem zurückwechseln:

```
$ git checkout master
Switched to branch 'master'
```

Dies funktioniert aber nur, wenn Sie keine offenen Änderungen (*modified*) in Ihrem aktuellen Branch haben. Änderungen im Staging-Bereich werden in den neuen Branch übernommen!

3.17.4 Änderungen im Branch vornehmen

Sie können nun im `master`-Branch die nötige Änderung vornehmen und Sie per Commit festhalten:

```
$ nano web_views.py
[…]

$ git diff
[…]
-print(f'birthday: {birthday}')
+print(f'day of birth: {birthday}')

$ git add web_views.py
$ git commit -m 'changed label'
```

3.17.5 Änderungen zwischen Branches miteinander vergleichen

Nach dieser kleinen Änderung kehren Sie in den `refactoring`-Branch zurück, um Ihre angefangene Arbeit fortzusetzen:

```
$ git checkout refactoring
Switched to branch 'refactoring'
```

Mit `git diff` lassen sich auch Branches miteinander vergleichen; hier der aktuell aktive (`refactoring`) mit dem Branch `master`:

```
$ git diff master
```

3.17.6 Branches zusammenführen

Sie können die Änderungen vom `master`-Branch in Ihrem Branch nachziehen, indem Sie die Branches zusammenführen (engl. *mergen*), wozu es den Befehl `git merge` gibt:

```
$ git merge master
```

Dadurch werden die Änderungen aus dem angegebenen Branch (`master`) mit dem derzeit aktivem Branch (`refactoring`) zusammengeführt:

```
$ git merge master
Updating 7d09467..2de3fe3
Fast-forward
 web_views.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Dabei werden die dateispezifischen sowie dateiübergreifenden Änderungen summarisch angezeigt.

Wenn Sie mit der Arbeit an Ihrem `refactoring`-Branch fertig sind, können Sie in den `master`-Branch zurückwechseln, und die Änderungen aus dem Refactoring darin nachziehen:

```
$ git checkout master
$ git merge refactoring
```

Dadurch konnten Sie unabhängig voneinander kleinere Änderungen am `master`-Branch durchführen und Ihr grösseres Refactoring weiterführen.

3.18 Abkürzungen definieren

Befehle, die man häufig braucht, und die viel Tipparbeit erfordern, können mithilfe eines *Alias* abgekürzt werden. Dies kann Zeit sparen und die Tastatur schonen, kann aber die Kommunikation mit anderen Entwicklern erschweren, falls diese andere Aliase verwenden.

Aliase werden mit dem bereits bekannten Befehl `git config` erfasst, wobei Konfigurationsoptionen mit dem Präfix `alias.` definiert werden können. Folgendes Alias kürzt die Befehlszeile `git log --pretty=oneline` durch `git lol` ab:

```
$ git config alias.lol 'log --pretty=oneline'
```

Diese Änderungen gelten nur für das jeweilige Repository. Mit `--global` und `--system` können sie benutzer- oder systemweit gesetzt werden. Es gelten die gleichen Regel zum Überschreiben von Konfigurationen wie weiter oben beschrieben.

Die bestehenden Aliase können mit folgender Befehlszeile ausgegeben werden:

```
$ git config -l | grep alias
alias.cm=commit
alias.lol=log --pretty=oneline
```

3.19 Git-Befehle im Überblick

Zu guter letzt sind hier noch einmal alle Befehle aufgelistet, welche in diesem Dokument behandelt worden sind. Genaueres dazu erfährt man über das Git-Hilfesystem oder in der Referenz (siehe letzter Abschnitt).

3.19.1 Hilfe und Konfiguration

- `git version`: Version anzeigen
- `git help`: Hilfe anzeigen
- `git config`: Git konfigurieren

3.19.2 Repository erhalten

- `git init`: Repository erstellen
- `git clone`: Repository kopieren

3.19.3 Zustand und Geschichte

- `git status`: Zustand einsehen
- `git log`: Versionsgeschichte einsehen
- `git diff`: Unterschiede anzeigen
- `git show`: Details zu Commit oder Tag anzeigen

3.19.4 Repository manipulieren

- `git add`: Datei(en) vormerken
- `git commit`: Änderungen festschreiben
- `git rm`: Datei(en) entfernen
- `git mv`: Datei(en) umbenennen
- `git reset`: Datei(en) aus Staging-Bereich entfernen
- `git checkout`: Änderungen rückgängig machen
- `git tag`: Versionen markieren/benennen

3.19.5 Zentrale Repositories

- `git remote`: Verwaltung von Remotes
- `git fetch`: Änderungen herunterladen
- `git pull`: Änderungen herunterladen und anwenden
- `git push`: Änderungen hochladen

3.19.6 Grafische Werkzeuge

- `git gui`: GUI-Client von Git verwenden
- `gitk`: Versionsgeschichte in GUI anzeigen

4 Quellen und Links

- Tech Talk: Linus Torvalds on git
- Pro Git (en)
- Pro Git (de)
- Git Reference (en)
- Videos zu Git (Modul 426)