

# Versionsverwaltung mit Git

Praxiseinstieg



Mit praktischer Referenzkarte

## **Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)**

Liebe Leserinnen und Leser,

dieses E-Book, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Mit dem Kauf räumen wir Ihnen das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Jede Verwertung außerhalb dieser Grenzen ist ohne unsere Zustimmung unzulässig und strafbar. Das gilt besonders für Vervielfältigungen, Übersetzungen sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Je nachdem wo Sie Ihr E-Book gekauft haben, kann dieser Shop das E-Book vor Missbrauch durch ein digitales Rechtemanagement schützen. Häufig erfolgt dies in Form eines nicht sichtbaren digitalen Wasserzeichens, das dann individuell pro Nutzer signiert ist. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Beim Kauf des E-Books in unserem Verlagsshop ist Ihr E-Book DRM-frei.

Viele Grüße und viel Spaß beim Lesen,

*Ihr mitp-Verlagsteam*



Neuerscheinungen, Praxistipps, Gratiskapitel,  
Einblicke in den Verlagsalltag –  
gibt es alles bei uns auf Instagram und Facebook



[instagram.com/mitp\\_verlag](https://instagram.com/mitp_verlag)



[facebook.com/mitp.verlag](https://facebook.com/mitp.verlag)



Sujeevan Vijayakumaran

# **Versionsverwaltung mit Git**

**Praxiseinstieg**



**Bibliografische Information der Deutschen Nationalbibliothek**

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-7475-0305-8

3. Auflage 2021

[www.mitp.de](http://www.mitp.de)

E-Mail: [mitp-verlag@sigloch.de](mailto:mitp-verlag@sigloch.de)

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

© 2021 mitp Verlags GmbH & Co. KG, Frechen

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Lektorat: Sabine Schulz

Sprachkorrektorat: Petra Heubach-Erdmann

Coverbild: Verne Ho @verneho / unsplash.com,

Coverbild gestaltet von Bernhard Hanakam

Satz: III-satz, Husby, [www.drei-satz.de](http://www.drei-satz.de)

## Repositories anlegen

### **git init [<pfad>]**

Erzeugt ein neues leeres Repository im aktuellen Arbeitsverzeichnis. Bei optionaler Angabe des Zielpfades wird ein neues Verzeichnis mit dem Repository darin angelegt.

### **git clone <repository> [<pfad>]**

Klont das als weiteren Parameter angegebene Repository in ein neues Verzeichnis im aktuellen Verzeichnis. Zusätzlich wird es als Remote-Repository `origin` konfiguriert. Bei der Angabe eines weiteren Verzeichnisses wird das Repository unter einem anderen Verzeichnisnamen geklont.

## Erweiterung und Bearbeitung der Historie

### **Arbeiten im Staging-Bereich**

#### **git add <pfad>**

Mit diesem Befehl können Änderungen von Dateien zum Staging-Bereich hinzugefügt werden, die anschließend in einem Commit gespeichert werden können.

#### **git reset**

Setzt den aktuellen HEAD auf einen bestimmten Stand zurück. Anwendbar sowohl bei Änderungen, die noch nicht in einem Commit sind, als auch bei Änderungen in diversen Commits.

#### **git restore**

Setzt Dateien im Arbeitsverzeichnis des Repositorys zurück. Mit `--source` kann optional die Quelle angegeben werden. Mit `--staged` kann eine Änderung aus dem Staging-Bereich wieder herausgenommen werden.

## Arbeiten mit Commits und Branches

### **git commit**

Erzeugt einen Commit aus den Änderungen aus dem Staging-Bereich. Ohne zusätzlichen Parameter öffnet sich der konfigurierte Editor. Alternativ kann die Commit-Message mit `-m` direkt beim Aufruf gesetzt werden.

### **git branch [<name>]**

Listet alle lokalen Branches auf und erzeugt einen neuen Branch mit dem angegebenen Namen. Dabei wird nicht auf den neuen Branch gewechselt.

### **git checkout [<branch>]**

Wechselt auf den angegebenen Branch. Mit `-b` kann optional ein neuer Branch erstellt werden, auf den direkt gewechselt wird.

### **git switch [<branch>]**

Wechselt ebenfalls auf den angegebenen Branch und ist ab Git Version 2.23 verfügbar. Mit `-c` kann optional ein neuer Branch erstellt werden, auf den direkt gewechselt wird.

### **git diff**

Zeigt die Unterschiede zwischen Commits, Commits und Arbeitsverzeichnisse, oder Commits und Staging-Bereich an.

### **git merge <branch>**

Führt mindestens zwei Branches zusammen.

### **git rebase**

Übernimmt Commits von anderen Branches und fügt diese auf dem aktuellen Branch an.

## Verteilte Repositorys

### **git fetch**

Lädt alle Objekte des konfigurierten Remote-Repositorys herunter und macht sie lokal zugänglich. Häufig werden mit dem Parameter alle Daten von allen konfigurierten Remote-Repositorys heruntergeladen.

### **git pull**

Lädt die Änderungen vom konfigurierten Branch vom Remote-Repository **und** führt ein Merge oder Rebase durch.

### **git push**

Schiebt die lokalen Commits des Branches auf den Server.

### **git remote**

Ermöglicht die Konfiguration von Remote-Repositorys.

## Status im Repository

### **git status**

Zeigt den Status im Repository an und gibt sehr hilfreiche Informationen aus, welche folgenden Befehle nun ausgeführt werden können.

### **git log**

Zeigt die Historie des Repositorys an.

## Multi-Repositories verwalten

### git subtree

Ermöglicht die Einbindung von anderen Git-Repositories in das aktuelle Repository, in dem das Subtree in das Repository gemerget wird.

### git submodule

Ermöglicht ebenfalls die Einbindung von anderen Git-Repositories in das aktuelle Repository. Die Submodule-Repositories werden über ein Commit referenziert und eingebunden.

## Sonstiges

### git blame <pfad>

Zeigt für die angegebene Datei Zeile für Zeile an, wer die Zeile zuletzt modifiziert hat, inklusive Commit-ID und Zeitpunkt.

### git stash

Schiebt Änderungen aus dem aktuellen Arbeitsverzeichnis in einen Zwischen-Speicher und kann diese Änderungen auch erneut wieder herausholen.

### git config

Ermöglicht es, die Konfiguration anzupassen.

### git reflog

Zeigt an, welche Tätigkeiten lokal im Repository ausgeführt wurden. Ist sehr nützlich, wenn Commits aus Versehen gelöscht wurden.

### git clean

Räumt das lokale Repository auf, etwa um ignorierte oder nicht versionierte temporäre Dateien gesammelt zu entfernen.

# Inhaltsverzeichnis

<b>Einleitung</b> .....	11
Aufbau des Buches .....	12
Konvention .....	13
Hinweise und Tipps .....	14
Feedback .....	14
Danksagung .....	14
<b>1 Einführung</b> .....	15
1.1 Lokale Versionsverwaltung .....	16
1.2 Zentrale Versionsverwaltung .....	17
1.3 Verteilte Versionsverwaltung .....	18
1.4 Geschichtliches .....	20
<b>2 Die Grundlagen</b> .....	23
2.1 Installation .....	23
2.2 Das erste Repository .....	26
2.3 Git-Konfiguration .....	28
2.4 Der erste Commit .....	29
2.4.1 Versionierte Dateien mit »git mv« verschieben .....	46
2.5 Änderungen rückgängig machen mit Reset und Revert .....	47
2.5.1 Revert .....	47
2.5.2 Reset .....	48
2.6 Git mit GUI .....	51
2.6.1 Commits mit Git GUI .....	53
2.7 Wie Git arbeitet .....	54
2.8 Git-Hilfe .....	60
2.9 Zusammenfassung .....	60
<b>3 Arbeiten mit Branches</b> .....	63
3.1 Allgemeines zum Branching .....	63
3.2 Branches anlegen .....	65
3.3 Branches mergen .....	72
3.4 Merge-Konflikte .....	76
3.5 Mergetools .....	80

3.6	Merge-Strategien . . . . .	83
3.6.1	resolve . . . . .	83
3.6.2	recursive . . . . .	83
3.6.3	octopus. . . . .	84
3.6.4	ours . . . . .	84
3.6.5	subtree. . . . .	84
3.7	Rebasing. . . . .	84
3.8	Stash und Clean. . . . .	90
3.8.1	Das Arbeitsverzeichnis säubern . . . . .	94
3.8.2	Dateien ignorieren . . . . .	96
3.9	Zusammenfassung . . . . .	98
4	<b>Verteilte Repositorys</b> . . . . .	99
4.1	Projekt mit einem Remote-Repository . . . . .	100
4.2	Branch-Management. . . . .	109
4.3	Tracking-Branches. . . . .	111
4.4	Projekt mit drei Remote-Repositorys . . . . .	114
4.5	Der Workflow mit drei Repositorys . . . . .	117
4.6	Zusammenfassung . . . . .	121
5	<b>Git-Hosting.</b> . . . . .	123
5.1	GitHub . . . . .	126
5.1.1	Repository anlegen . . . . .	126
5.1.2	SSH-Keys anlegen und hinzufügen . . . . .	129
5.1.3	SSH-Agent konfigurieren . . . . .	131
5.1.4	Lokales Git-Repository konfigurieren . . . . .	133
5.1.5	Repository klonen. . . . .	135
5.1.6	Der GitHub-Workflow . . . . .	135
5.1.7	GitHub-Repositorys um externe Tools erweitern. . . . .	152
5.2	GitLab . . . . .	152
5.2.1	Installation . . . . .	153
5.2.2	Konfiguration . . . . .	153
5.3	Weitere Git-Hosting-Lösungen. . . . .	158
5.4	CI/CD: Continuous Integration und Continuous Delivery . . . . .	158
5.4.1	Der Workflow . . . . .	159
5.4.2	GitHub Actions. . . . .	161
5.4.3	GitLab CI/CD . . . . .	164
5.5	Zusammenfassung . . . . .	168

<b>6</b>	<b>Workflows</b>	169
6.1	Interaktives Rebasing	170
6.1.1	Branches pseudo-sichern	171
6.1.2	Den letzten Commit verändern	171
6.1.3	Mehrere Commits verändern	174
6.1.4	Reihenfolge der Commits anpassen	176
6.1.5	Commits ergänzen	176
6.1.6	Commits squashen	178
6.1.7	Commits autosquashen	180
6.1.8	Commits droppen	181
6.1.9	Commit aufteilen	181
6.2	Workflow mit einem Branch und Repository für eine Person	183
6.3	Workflow mit mehreren Personen, einem Repository und einem Branch	184
6.4	Git Flow	186
6.4.1	Feature-Branches	187
6.4.2	Release-Branches	190
6.4.3	Release taggen	191
6.4.4	Hotfix-Branches	192
6.4.5	Zusammenfassung zu Git Flow	193
6.5	Git Flow mit mehr als einem develop-Branch	194
6.6	Git Flow mit mehreren Repositorys	195
6.7	GitHub-Flow	197
6.8	GitLab-Flow	197
6.9	Weitere Aspekte in Workflows	199
6.10	Zusammenfassung	201
<b>7</b>	<b>Hooks</b>	203
7.1	Client-seitige Hooks	203
7.1.1	Commit-Hooks	204
7.1.2	E-Mail-Hooks	207
7.1.3	Weitere Hooks	207
7.2	Server-seitige Hooks	208
7.2.1	pre-receive-Hook	208
7.2.2	update-Hook	208
7.2.3	post-receive-Hook	209
7.2.4	Beispiel-Hooks	209
7.3	Git-Attribute	211

<b>8</b>	<b>Umstieg von Subversion</b>	215
8.1	Zentrale vs. verteilte Repositorys	215
8.2	Checkout vs. Clone	216
8.3	svn commit vs. git commit & git push	216
8.4	svn add vs. git add	216
8.5	Binärdateien im Repository	217
8.6	SVN- in Git-Repository konvertieren	217
8.6.1	git-svn	218
8.6.2	Nach der Umwandlung	221
8.6.3	Committen mit git-svn	221
8.7	Zusammenfassung	223
<b>9</b>	<b>Tipps und Tricks</b>	225
9.1	Große Dateien mit Git LFS verwalten	225
9.2	Partielles Klonen	227
9.3	Aliasse setzen und nutzen	228
9.4	Mehr aus dem Log holen	229
9.4.1	Begrenzte Ausgaben	229
9.4.2	Schönere Logs	231
9.5	Ausgeföhrte Aktionen im Repository mit git reflog	232
9.6	Garbage Collection mit git gc	235
9.7	Finde den Schuldigen mit git blame	236
9.8	Wortweises diff mit word-diff	236
9.9	Verschobene Zeilen farblich hervorheben mit git diff --color-moved	237
9.10	Datei-Inhalte suchen mit git grep	238
9.11	Änderungen häppchenweise stagern und committen	239
9.12	Auf Fehlersuche mit git bisect	241
9.13	Arbeiten mit Patches	243
9.13.1	Patches erstellen	243
9.13.2	Patches anwenden	245
9.14	Repositorys in Repositorys mit git submodules	247
9.15	Subtree als Alternative für Submodule	250
9.16	Komplette Historie neu schreiben mit git filter-repo	252
9.17	Tippfehler in Git-Befehlen automatisch korrigieren	253
9.18	Git Worktree	254
9.19	Liquid Prompt für Git	256
9.19.1	Installation	256
9.19.2	Im Einsatz mit Git	257
9.20	Zusammenfassung	258

<b>10</b>	<b>Grafische Clients</b>	261
10.1	Git GUI	261
10.2	Gitk	263
10.3	SourceTree	267
10.4	GitHub Desktop	269
10.5	Gitg	271
10.6	Tig	272
10.7	TortoiseGit	274
10.8	GitKraken	276
10.9	Weiteres	277
<b>11</b>	<b>Nachvollziehbare Git-Historien</b>	279
11.1	Gut dosierte Commits	279
11.2	Gute Commit-Messages	281
<b>12</b>	<b>DevOps</b>	289
12.1	DevOps im Überblick	289
12.2	Das Problem	290
12.3	DevOps-Pipeline	294
12.4	DevSecOps	296
12.5	Zusammenfassung	298
<b>13</b>	<b>Frequently Asked Questions</b>	299
<b>A</b>	<b>Befehlsreferenz</b>	305
A.1	Repository und Arbeitsverzeichnis anlegen	305
A.2	Erweiterung und Bearbeitung der Historie	306
A.2.1	Arbeiten im Staging-Bereich	306
A.2.2	Arbeiten mit Commits und Branches	307
A.3	Status-Ausgaben und Fehler-Suche	310
A.4	Verteilte Repositorys	311
A.5	Hilfsbefehle	313
A.6	Sonstige	314
	<b>Stichwortverzeichnis</b>	317



# Einleitung



»Das ist Git. Es bietet einen Überblick über die kollaborative Arbeit in Projekten durch die Nutzung eines wunderschönen Graphen-Theorie-Modells.«

Sie: »Cool. Aber wir nutzt man es?«

Er: »Keine Ahnung. Merke dir einfach all diese Befehle und tippe sie ein. Wenn du auf Fehler stößt, dann sichere deine Arbeit woanders, lösche das Projekt und lade eine frische Kopie herunter.«

»If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of >It's really pretty simple, just think of branches as...< and eventually you'll learn the commands that will fix everything.«

»Und wenn das auch nicht hilft, dann enthält git.txt die Telefonnummer von einem Freund, der sich mit Git auskennt. Warte einfach ein paar Minuten ab à la >Es ist wirklich gar nicht so schwer, stell dir nur die Branches vor als ...<, und schließlich lernst du die Befehle, die jedes Problem fixen.«<sup>1</sup>

Versionskontrolle ist ein wichtiges Thema für Software-Entwickler. Jeder, der ohne jegliche Versionskontrollprogramme arbeitet, ist vermutlich schon einmal an den Punkt gestoßen, wo man sich ältere Stände ansehen wollte. Dabei fragt man sich gegebenenfalls, warum und wann man eine Funktion eingeführt hat, oder man

<sup>1</sup> »xkcd: Git«, Copyright Randall Munroe (<https://xkcd.com/1597/>) ist lizenziert unter der Creative Commons Lizenz CC BY-NC 2.5 (<https://creativecommons.org/licenses/by-nc/2.5/>)

möchte auf einen älteren Stand zurückspringen, wenn man etwas kaputt gemacht hat. Genau an dieser Stelle kommen Versionsverwaltungsprogramme ins Spiel. Git ist eines dieser Programme, die nicht nur die bereits genannten Probleme lösen. Es ist Kernbestandteil des Entwicklungsprozesses, um sowohl kollaborativ im Team als auch alleine an einem Projekt zu arbeiten. Dabei ist es gleichgültig, ob man programmiert, Systeme administriert oder gar Bücher schreibt ist.

Randall Munroe beleuchtet in seinem Webcomic xkcd viele verschiedene Themen. Das hier abgedruckte xkcd-Comic zum Thema Git wurde während meiner Arbeit an der ersten Auflage dieses Buches veröffentlicht. Viele meiner Freunde und Bekannten aus dem Open-Source-Umfeld posteten das Comic in den verschiedenen sozialen Netzwerken und machten eins deutlich: Viele Leute nutzen zwar Git, wissen aber nur grob, was dort passiert. Wenn etwas nicht wie geplant funktioniert oder man zu einem fehlerhaften Zustand im Arbeitsprojekt kommt, dann weiß man erst mal nicht weiter und fragt seinen persönlichen Git-Experten, wie den einen Kollegen, der glücklicherweise ein Git-Buch geschrieben hat.

Das Ziel dieses Buches ist nicht nur, dass Sie die gängigen Befehle erlernen, die Sie beim Arbeiten mit Git brauchen. Ich lege auch großen Wert auf die Einbindung und Anpassung des Entwicklungsprozesses. Darüber hinaus sollten Sie Git als Ganzes verstehen und nicht nur die Grundlagen, damit Sie mit einem Programm arbeiten, das Sie verstehen und bei dem bei Konflikten keine Hürden vorhanden sind.

## Aufbau des Buches

Dieses Buch besteht aus insgesamt dreizehn Kapiteln, davon gehören die ersten vier Kapitel zu den Grundlagen und die übrigen acht zu den fortgeschrittenen Themen.

Das erste Kapitel führt in das Thema der Versionsverwaltung mit Git ein, um den Einsatzzweck und die Vorteile von Git zu verdeutlichen. Das zweite Kapitel behandelt die grundlegenden Git-Kommandos. Dies beinhaltet die Basis-Befehle, die für das Arbeiten mit Git notwendig sind. Im anschließenden dritten Kapitel geht es um die Nutzung von Branches, eines der elementaren Features von Git. So lernen Sie, mit Branches parallele Entwicklungslinien zu erstellen, zwischen diesen verschiedenen Branches hin und her zu wechseln und sie wieder zusammenzuführen. Der Grundlagenteil endet mit dem vierten Kapitel, bei dem es um den Einsatz von verteilten Repositorys geht, die es ermöglichen, mit Repositorys zu arbeiten, die auf entfernten Servern, wie etwa GitHub oder GitLab, liegen.

Bei den fortgeschrittenen Themen liegt der Fokus besonders auf dem Einsatz von Git in Software-Entwicklungsteams. Wichtig ist dabei, über eine gute Möglichkeit zu verfügen, Git-Repositorys hosten zu können, damit man kollaborativ in einem Team an Projekten arbeiten kann. Während die wohl gängigste, bekannteste und einfachste Hosting-Möglichkeit GitHub ist, gibt es auch einige Open-Source-Alter-

nativen, wie zum Beispiel GitLab, die sich ebenfalls sehr gut für den Einsatz in Firmen oder anderen Projektgruppen eignen. Das ist das Thema im fünften Kapitel, in dem auch der Workflow bei GitHub und GitLab thematisiert wird. Im anschließenden sechsten Kapitel geht es um die verschiedenen existierenden Workflows. Um die Features von Git sinnvoll einzusetzen, sollten Sie einen Workflow nutzen, der sowohl praktikabel ist als auch nicht zu viel Overhead im Projekt führt. Die Art und Weise, mit Git zu arbeiten, unterscheidet sich vor allem bei der Anzahl der Personen, Branches und Repositorys. Im sechsten Kapitel geht es im Anschluss darum, Git-Hooks zu verwenden, um mehr aus dem Projekt herauszuholen oder simple Fehler automatisiert zu überprüfen und somit zu vermeiden. So lernen Sie, was Hooks sind, wie sie programmiert werden und damit zu automatisieren. Generell ist dieses Kapitel für den Git-Nutzer kein alltägliches Thema. Hooks werden im Alltag eher unregelmäßig programmiert.

Die weiteren drei Kapitel befassen sich mit dem Umstieg von Subversion nach Git, wobei sowohl die Übernahme des Quellcodes inklusive der Historie als auch die Anpassung des Workflows thematisiert wird. Das neunte Kapitel ist eine Sammlung vieler verschiedener nützlicher Tipps, die zwar nicht zwangsläufig täglich gebraucht werden, aber trotzdem sehr nützlich sein können. Im zehnten Kapitel folgt dann noch ein Kapitel mit einem Überblick über die grafischen Git-Programme unter den verschiedenen Betriebssystemen Windows, macOS und Linux. In der zweiten Auflage sind die vergleichsweise kurzen Kapitel 11 und 13 neu dazugekommen. Hier werden zum einen nützliche Hilfestellungen gegeben, um eine möglichst nachvollziehbare Git-Historie zu erzeugen, und zum anderen werden häufige Probleme von Anfängern und Erfahrenen beleuchtet und die dazugehörigen Lösungen aufgezeigt. Neu in der dritten Auflage ist das 12. Kapitel. Hier wird das Thema DevOps kurz und kompakt zusammengefasst, wofür Git das grundlegende Werkzeug ist.

Um den Einsatz von Git und die einzelnen Funktionen sinnvoll nachvollziehen zu können, werden alle Git-Kommandos anhand eines realen Beispiels erläutert. Über die Kapitel des Buches hinweg entsteht eine kleine statische Webseite, an der die Funktionen verdeutlicht werden. Denn was bringt es, die Kommandos von Git ohne den Bezug zu realen Projekten und dessen Einsatzzwecke zu kennen? Eine kleine Webseite hat insbesondere den Vorteil, dass Sie nicht nur Unterschiede im Quellcode nachvollziehen, sondern auch sehr einfach die optischen Unterschiede auf einer Webseite erkennen können.

## Konvention

In diesem Buch finden Sie zahlreiche Terminal-Ausgaben abgedruckt. Diese sind größtenteils vollständig, einige mussten aus Platz- und Relevanz-Gründen jedoch gekürzt werden. Eingaben in der Kommandozeile fangen immer mit dem »\$« an.

Dahinter folgt dann der eigentliche Befehl. Das Dollarzeichen ist der Prompt, der in der Shell dargestellt wird, und muss daher nicht eingetippt werden. Zeilen, die kein solches Zeichen besitzen, sind Ausgaben der Befehle. Das sieht dann etwa so aus:

```
$ git log  
commit 9534d7866972d07c97ad284ba38fe84893376e20  
[...]
```

Zeilen, die nicht relevant sind oder verkürzt wurden, sind als »[...]« dargestellt.

## Hinweise und Tipps

Die einzelnen Kapitel bauen zwar aufeinander auf, doch ist es nicht immer möglich, alle Themen an Ort und Stelle ausführlich zu behandeln. Zudem werden wohl eher wenige Leser das Buch von vorne bis hinten durcharbeiten. Das Buch beinhaltet daher einige Hinweise und Tipps. Teilweise sind es Hinweise auf nähere Details in anderen Teilen des Buches, teilweise Tipps und Warnungen für die Nutzung von Git. Dies sind häufig nützliche Inhalte, die sich auf das gerade behandelte Thema beziehen, hin und wieder aber auch Querverweise zu näheren Erläuterungen in anderen Kapiteln.

## Feedback

Als Autor habe ich sehr wohl den Anspruch, dass Sie als Leser das, was in diesem Buch behandelt wird, sowohl richtig verstehen als auch anwenden können. Ich bin daher offen für Feedback und Verbesserungsvorschläge – entweder per E-Mail an [mail@svij.org](mailto:mail@svij.org) oder Kurzes gerne auch via Twitter an @svijee (<https://twitter.com/svijee>). Ich bin sehr an Ihrem Feedback interessiert!

## Danksagung

Ich freue mich, dass ich erneut die Möglichkeit vom Verlag erhalten habe, dieses Buch in der nun dritten aktualisierten Auflage veröffentlichen zu dürfen. Mein Dank gilt daher erneut dem Verlag mitp und insbesondere meiner Lektorin Sabine, mit der ich nun mittlerweile fünf Jahre an diesem Buch zusammenarbeitete.

Weiterhin gilt mein Dank auch dieses Mal meiner Familie und allen, die mir immer wieder neuen kleinen und großen Input und Feedback liefern.

# Einführung

Versionsverwaltung – Was ist denn nun eigentlich genau ein Versionsverwaltungsprogramm? Wodurch zeichnet es sich aus und warum wird es gebraucht? Das sind einige der häufigen ersten Fragen, die zu Beginn aufkommen. Die prinzipielle Bedeutung leitet sich schon aus dem Wort selbst ab: Es handelt sich um die Verwaltung von Versionen. Konkret bedeutet es, dass Sie von Dateien Versionen erzeugen können, die dann sinnvoll verwaltet werden.

Das Wort »Version« klingt zunächst erst einmal nach einer größeren Änderung, doch auch eine kleine Änderung erzeugt eine neue Version einer Datei. Je nach Kontext gibt es ein unterschiedliches Verständnis für den Begriff »Version«. Wenn bei Git von Versionen gesprochen wird, ist damit so gut wie immer die Version einer einzelnen Datei oder einer Sammlung von Dateien gemeint. Im Sinne der Software-Entwicklung werden neue Versionen von Programmen veröffentlicht, also zum Beispiel die Git-Version 2.29.

Aber wofür brauchen Sie nun ein Versionsverwaltungsprogramm wie Git? Viele kennen vermutlich folgendes Problem: Sie gehen einer Tätigkeit nach – sei es das Schreiben an einem Text, das Bearbeiten eines Bildes oder eines Videos – und der aktuelle Stand soll immer mal wieder zwischengespeichert werden. Hauptgrund ist, dass dauernd eine Sicherung der Datei vorhanden sein soll, und ein weiterer Grund ist, dass Sie wieder auf einen älteren Stand zurückspringen können, falls Sie doch einige Schritte rückgängig machen wollen. Die Vorgehensweise zum manuellen Erzeugen solcher Versionen ist unterschiedlich – die einen fügen Zahlen mit Versionsnummern am Ende des Dateinamens an, die anderen erzeugen wiederum Ordner mit dem aktuellen Datum, in denen die Dateien liegen. So passt es häufiger, dass neben *Bachelorarbeit\_v1.odt* und *Bachelorarbeit\_v2.odt* noch ein *Bachelorarbeit\_v3\_final.odt* und *Bachelorarbeit\_v3\_final\_new.odt* liegt. Beide genannten Möglichkeiten funktionieren zwar prinzipiell, sind allerdings weder praktikabel noch wirklich sicher und vor allem fehleranfällig. Das ist besonders dann der Fall, wenn Sie den Dateien keine eindeutigen Namen gegeben haben. Dies trifft insbesondere dann zu, wenn zu viele Versionen einer einzigen Datei rumliegen oder mehrere Dateien gleichzeitig versioniert werden müssen.

Genau bei diesem Problem kommen Versionsverwaltungsprogramme zum Einsatz. Mit diesen werden neben den reinen Veränderungen noch weitere Informationen zu einer Version gespeichert. Darunter fallen in der Regel der Autorennname, die Uhrzeit der Änderung und eine Änderungsnotiz. Diese werden bei jeder neuen

Version gespeichert. Durch die gesammelten Daten können Sie so schnell und einfach eine Änderungshistorie ansehen und verwalten. Falls zwischendurch Fehler in den versionierten Dateien eingeflossen sind, können Sie leicht untersuchen, wann und durch welche Person die Fehler eingeführt wurden, und diese wieder rückgängig machen. Versionsverwaltungsprogramme lassen sich demnach nicht nur von einzelnen Personen nutzen, sondern ermöglichen das Arbeiten im Team mit mehr als einer Person.

Mit Versionsverwaltungsprogrammen lassen sich alle möglichen Dateitypen verwalten. Sie sollten allerdings beachten, dass eine Versionierung nicht für jeden Dateityp praktikabel ist. Besonders hilfreich sind solche Anwendungen vor allem für Arbeiten mit reinen Text-Dateien. Darunter fallen insbesondere Quellcode von Programmen, Konfigurationsdateien oder auch Texte und somit auch Bücher. Der Vorteil bei reinen Textdateien ist, dass Sie die Unterschiede bei Änderungen für jede Zeile nachvollziehen können – das ist bei binären Dateiformaten nicht möglich. Auch für Grafiker kann der Einsatz eines Versionsverwaltungsprogramms sinnvoll sein, denn mit zusätzlichen Tools können auch die Veränderungen zwischen zwei Versionen von Bildern dargestellt werden.

Insgesamt gibt es drei verschiedene Konzepte zur Versionsverwaltung: die lokale, die zentrale und die verteilte Versionsverwaltung.

## 1.1 Lokale Versionsverwaltung

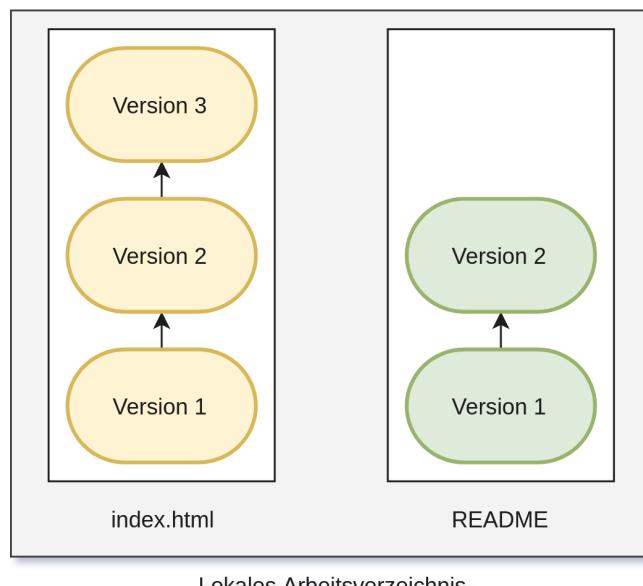


Abb. 1.1: Lokale Versionsverwaltung arbeitet Datei-basiert und lediglich lokal.

Die lokale Versionsverwaltung findet sich eher seltener in produktiven Umgebungen, da sie lediglich lokal arbeitet und häufig nur einzelne Dateien versioniert. Die zuvor erwähnte manuelle Erzeugung von Versionen von Dateien wäre zum Beispiel eine lokale Versionsverwaltung mit einer einzelnen Datei. Sie ist zwar einfach zu nutzen, doch ist es fehleranfällig und wenig flexibel. Echte Versionsverwaltungssoftware, die nur lokal arbeitet, gibt es allerdings auch, darunter »SCSS« und »RCS«. Der größte Nachteil lokaler Versionsverwaltung ist, dass im Normalfall nur eine Person mit den Dateien arbeiten kann, da diese nur lokal auf dem einen Gerät verfügbar sind. Weiterhin besteht keine Datensicherheit, da die Dateien nicht automatisch auf einem anderen Gerät gesichert werden. Der Anwender ist somit allein verantwortlich für ein Backup der Dateien inklusive der Versionshistorie.

## 1.2 Zentrale Versionsverwaltung

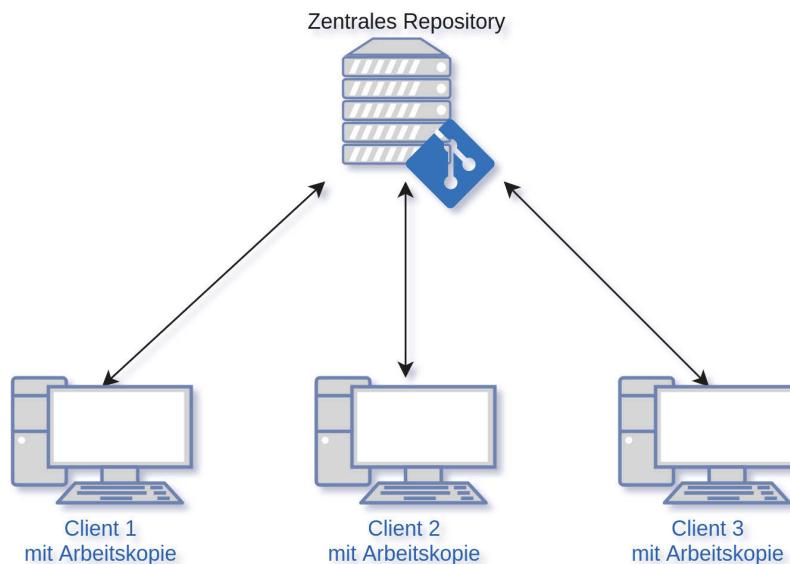


Abb. 1.2: Zentrale Versionsverwaltung arbeitet mit Arbeitskopien auf Clients.

Zentrale Versionsverwaltungen befinden sich heute vergleichsweise noch häufig im Einsatz. Bekannte und verbreitete Vertreter dieser Art sind Subversion und CVS. Das Hauptmerkmal zentraler Versionsverwaltungen ist, dass das Repository lediglich auf einem zentralen Server liegt. Das Wort »Repository« ist Englisch und steht für »Lager«, »Depot« oder auch »Quelle«. Ein Repository ist somit ein Lager, in dem die versionierten Dateien liegen. Autorisierte Nutzer verfügen über eine lokale Arbeitskopie einer Version, auf der sie ihre Arbeiten erledigen.

Die Logik und die Daten der Versionsverwaltung liegen größtenteils auf dem zentralen Server. Beim Wechsel von Revisionen oder beim Vergleichen von Änderungen wird stets mit dem Server kommuniziert. Wenn der Server also offline ist, kann der Nutzer zwar mit der Arbeitskopie ein wenig weiterarbeiten. Allerdings ist die Einsicht älterer Versionen oder das Ansehen anderer Entwicklungslinien nicht möglich, da es sich lediglich um eine Arbeitskopie einer Version und keine Kopie des vollständigen Repositorys handelt.

## 1.3 Verteilte Versionsverwaltung

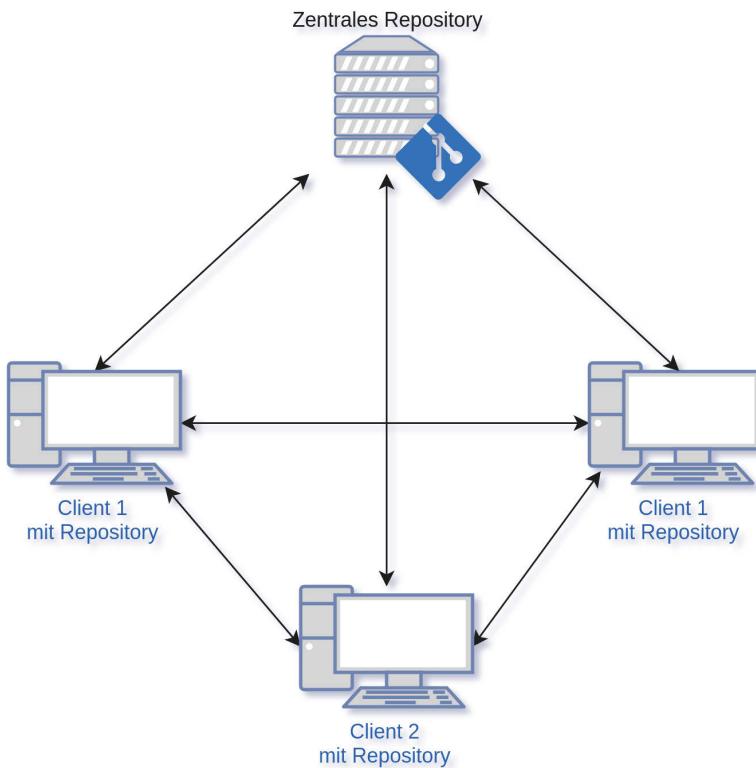


Abb. 1.3: Verteilte Versionsverwaltung arbeitet mit Repositorys auf Clients und Servern.

Git gehört zu den verteilten arbeitenden Versionsverwaltungsprogrammen. Neben Git gibt es auch andere verteilte Versionskontrollprogramme, wie Bazaar oder Mercurial. Im Gegensatz zur zentralen Versionsverwaltung besitzt jeder Nutzer des Repositorys nicht nur eine Arbeitskopie, sondern das komplette Repository. Wenn Sie also zwischen verschiedenen Revisionen wechseln oder sich die Historie einzelner Dateien anschauen möchten, dann geschieht das Ganze auf dem lokalen

Rechner. Zuvor muss nur das Repository »geklont« werden. Alle Funktionen stehen dann auch offline zur Verfügung. Ein wesentlicher Vorteil davon ist, dass nicht nur unnötiger Datenverkehr vermieden wird, sondern auch die Geschwindigkeit deutlich höher ist, was durch die fehlende Netzwerklatenz bedingt ist.

Zusätzlich besitzen verteilte Versionsverwaltungssysteme eine höhere Datenausfallsicherheit, da die Kopien der Daten des Repositorys in der Regel auf verschiedenen Rechnern liegen. Bei einem Ausfall des Git-Servers ist es daher möglich, weiterzuarbeiten. Nichtsdestotrotz sollten Sie von wichtigen Daten natürlich immer Backups anfertigen, ganz egal ob es sich um lokale, zentrale oder verteilte Versionsverwaltung handelt.

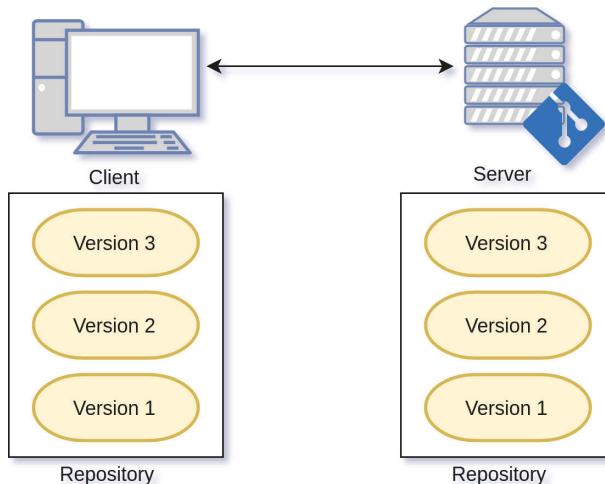


Abb. 1.4: Die Versionshistorie liegt sowohl lokal auf dem Client als auch auf dem Server.

Um den Unterschied zwischen zentralen und verteilten Versionsverwaltungsprogrammen klarer zu machen, kann folgendes Beispiel helfen. Stellen Sie sich vor, dass das Repository ein dicker Aktenordner ist. Darin enthalten sind alle aktuellen Dateien, ältere Versionen der Dateien sowie die Änderungshistorie mitsamt den Kommentaren zu den Änderungen. Sie müssen mit diesen Dateien arbeiten. Wenn es sich um ein zentrales System handelt, dann befindet sich der Aktenordner an einer zentral zugänglichen Stelle, die hier nun Archiv genannt wird. Für Sie heißt es, dass Sie zum Archiv und zu dem Ordner gehen müssen. Dort wird dann eine Arbeitskopie der benötigten Dateien erzeugt und anschließend laufen Sie wieder zurück zum Arbeitsplatz. Wenn Sie die Änderungshistorie von einer oder mehreren Dateien ansehen möchten, müssen Sie immer wieder zum Archiv laufen und den Aktenordner durchblättern, um sich diese anzusehen. Da es sowohl Zeit als auch Energie kostet, immer zum zentralen Aktenordner zu laufen, bietet

es sich an, eine Kopie des ganzen Ordners zu erstellen und mit an Ihren Arbeitsplatz zu nehmen.

Genau das ist dann eine verteilte Versionsverwaltung, da nun zwei vollständige Kopien des Aktenordners existieren – einmal an zentraler Stelle im Archiv und einmal am eigenen Arbeitsplatz. Der Vorteil ist, dass nach der ersten Kopie nur noch die Veränderungen hin- und hergetragen werden müssen. Alles andere kann bequem vom Arbeitsplatz aus gemacht werden, ohne ständig aufzustehen und herumlaufen zu müssen. Konkret bedeutet das, dass Sie an Ihrem Arbeitsplatz sitzen und Ihre Aufgaben erledigen. Sobald die Arbeit abgeschlossen ist, tragen Sie nur die neuen Dateien zum Archiv, wo Sie eine Kopie anfertigen und diese im zentralen Aktenordner abheften. Großer Vorteil ist, dass Sie auch weiterhin arbeiten können, wenn der Weg zum Aktenordner unzugänglich ist, etwa genau dann, wenn Sie unterwegs sind.

### Zusammenfassung

- Die lokale Versionsverwaltung funktioniert lediglich auf einem einzelnen Rechner.
- Bei der zentralen Versionsverwaltung liegt das »Gehirn« auf einem zentralen Server, von dem sich alle Mitarbeiter eine Arbeitskopie ziehen können.
- Bei der verteilten Versionsverwaltung liegt das vollständige Repository sowohl auf mindestens einem Server sowie auf allen Clients, wo mit Klonen gearbeitet wird.

## 1.4 Geschichtliches

Seinen Ursprung hatte Git bei der Entwicklung des Linux-Kernels. Letzterer wurde lange Zeit mit BitKeeper verwaltet, das damals ein proprietäres Programm war. Nachdem die Hersteller von BitKeeper die Lizenz geändert hatten, konnten die Linux-Kernel-Entwickler um Linus Torvalds BitKeeper nicht mehr kostenfrei verwenden, weswegen Linus Torvalds mit der Entwicklung von Git begann. Erst im Mai 2016 wurde BitKeeper unter einer Open-Source-Lizenz veröffentlicht.

Die Entwicklung von Git begann im Jahr 2005 und es gehört somit zu den jüngeren Versionsverwaltungssystemen und das, obwohl es mittlerweile mehr als 15 Jahre alt ist. Linus Torvalds fand es wichtig, dass das zukünftig eingesetzte Programm zur Entwicklung des Linux-Kernels drei spezielle Eigenschaften besitzt. Das sind zum Ersten Arbeitsabläufe, die an BitKeeper angelehnt sind, zum Zweiten die Sicherheit gegen böswillige und unbeabsichtigte Verfälschung des Repositorys sowie zum Dritten eine hohe Effizienz. Das Projekt »Monotone« wäre nahezu perfekt für diese Aufgabe gewesen. Das einzige Problem war nur, dass es

nicht sonderlich effizient arbeitete. Letztendlich entschied sich Linus Torvalds für die Entwicklung eines komplett neuen Programms, was er dann Git nannte.

Interessant ist auch die Namensgebung von Git. Das Wort »Git« ist das englische Wort für »Blödmann«. Linus Torvalds selbst sagte spaßeshalber: »I'm an egoistical bastard, and I name all my projects after myself. First ›Linux‹, now ›Git‹.« (Deutsch: »Ich bin ein egoistisches Arschloch und ich benenne alle meine Projekte nach mir selbst. Erst ›Linux‹ und jetzt eben ›Git‹.«). Natürlich gab es auch echte Gründe, das Projekt »Git« zu taufen. Zum einen enthält das Wort lediglich drei Buchstaben, was das regelmäßige Tippen auf der Tastatur erleichtert, zum anderen gab es kein bestehendes UNIX-Kommando, mit dem es kollidieren würde.



# Die Grundlagen

In diesem Kapitel lernen Sie die grundlegenden Funktionen und Kommandos von Git kennen. So gut wie alle in diesem Kapitel behandelten Befehle dürften beim täglichen Arbeiten mit Git zum Einsatz kommen. Damit Sie den Sinn und Zweck einzelner Befehle und Funktionen von Git sowohl nutzen als auch nachvollziehen können, arbeiten Sie in diesem Kapitel hauptsächlich an einem Beispielprojekt, das die Nutzung und Arbeitsweise von und mit Git verdeutlicht.

Das Beispiel-Projekt ist eine kleine Website, die nach und nach aufgebaut wird. HTML-Kenntnisse sind prinzipiell nicht notwendig, können aber natürlich auch nicht schaden. Damit es nicht ganz so trocken und langweilig ist, sollten Sie die Beispiele auf dem eigenen Rechner auf jeden Fall nachmachen. An der ein oder anderen Stelle bietet es sich auch an, etwas herumzuexperimentieren, denn nur durch Praxis wird Ihnen das Arbeiten mit Git klar und Sie haben hinterher in echten Projekten keine großen Probleme.

Als Beispiel wird eine kleine persönliche Website mit dem HTML5-Framework »Bootstrap« erstellt. Auf die genaue Funktionsweise des Frameworks gehe ich nicht näher ein, da es sich hier ja um Git und nicht um HTML und CSS dreht.

Git ist traditionell ein Kommandozeilenprogramm, weshalb der Fokus auf der Arbeit mit Git in der Kommandozeile liegt. Einschübe mit grafischen Git-Programmen gibt es dennoch. Grafischen Git-Programmen ist mit Kapitel 10, »Grafische Clients« ein vollständiges Kapitel gewidmet.

## 2.1 Installation

Bevor Sie losgelegen, müssen Sie Git installieren. Git gibt es nicht nur für die gängigen Betriebssysteme Windows, macOS und Linux, sondern unter anderem auch für FreeBSD, Solaris und sogar Haiku. Die gängigen Linux-Distributionen stellen Git unter dem Paketnamen »git« in der Paketverwaltung zur Verfügung. Nutzer von Windows und mac OS können sich Git von der Projektwebsite <https://git-scm.com/downloads> herunterladen.

Während der Arbeit an diesem Buch ist die Git-Version 2.29 die neueste Version. Große Unterschiede zu den vorherigen Versionen seit 2.0 existieren hingegen nicht, es sind vielmehr zahlreiche Kleinigkeiten, die über die Zeit eingeflossen sind. Bei Bedarf werden neue Funktionen aus vergleichsweise neuen Versionen

hervorgehoben. Gleichermaßen gilt für möglicherweise ältere Versionen von Git, die sich noch in den Paketverwaltungen älterer Linux-Distributionen finden. Obwohl die Version 2.0 von Git schon über sieben Jahre alt ist, werden an den ein oder anderen Stellen im Buch noch Unterschiede zu den mittlerweile sehr alten Versionen hervorgehoben. Als Neuankömmling sehen Sie so, was sich getan hat, und wenn Sie nach etlichen Jahren Pausen dann doch wieder Git anfassen, dann geht Ihnen auch nichts verloren.

Seit der Version 2.5, die im Mai 2015 erschien, hat Git unter Windows keinen Preview-Status mehr, sondern ist als vollwertige stabile Version verfügbar.

Die Installation unter Windows ist größtenteils selbsterklärend. Ein paar Kleinigkeiten gibt es aber doch zu beachten. Ein Punkt bei der Installation ist die Abfrage des genutzten Editors. In früheren Versionen wurde automatisch der Konsolen-Texteditor »vim« installiert und konfiguriert. Dieser ist vor allem für gängige Windows-Nutzer ohne Erfahrung in der Nutzung von »vim« eine schwierige Wahl. Eine Konfiguration eines anderen Editors war zuvor erst nachträglich möglich.

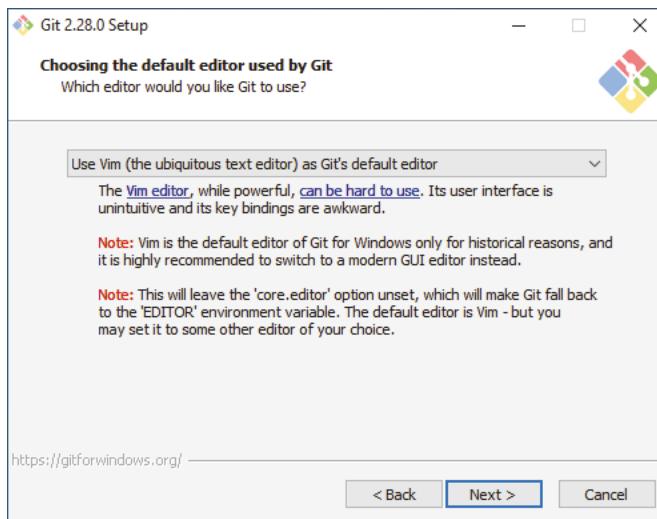


Abb. 2.1: Die Installation erlaubt die Auswahl verschiedener Editoren.

Mittlerweile können verschiedene Editoren ausgewählt werden. Eine Auswahl davon ist vorgegeben, aber auch andere Editoren lassen sich bereits bei der Installation konfigurieren.

Ein weiterer Punkt sind die Unterschiede bei der Nutzung der Shell. Die Shell ist das Fenster, in dem die Kommandozeilenbefehle eingetippt werden. Git lässt sich sowohl in der Windows-Cmd nutzen als auch in der »Git Bash« verwenden.