



4.

Auflage



René Preißel · Bjørn Stachmann

# git

Dezentrale Versionsverwaltung im Team  
Grundlagen und Workflows

dpunkt.verlag

**René Preißel · Bjørn Stachmann**

# **Git**

**Dezentrale Versionsverwaltung im Team  
Grundlagen und Workflows**

4., aktualisierte und erweiterte Auflage



**dpunkt.verlag**

René Preißel  
Björn Stachmann  
E-Mail: git@eToSquare.de

Lektorat: René Schönenfeldt  
Lektoratsassistenz: Stefanie Weidner  
Projektkoordination: Miriam Metsch  
Copy-Editing: Annette Schwarz, Ditzingen  
Satz: Da-Tex, Leipzig  
Herstellung: Susanne Bröckelmann  
Umschlaggestaltung: Helmut Kraus, www.exclam.de  
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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:  
Print 978-3-86490-452-3  
PDF 978-3-96088-127-8  
ePub 978-3-96088-128-5  
mobi 978-3-96088-129-2

4., aktualisierte und erweiterte Auflage 2017  
Copyright © 2017 dpunkt.verlag GmbH  
Wieblinger Weg 17  
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Dieses Buch wurde selbstverständlich mit Git erstellt:

Version (Commit-Hash):

commit 550bf4fb886ab74fa2424ce1e52bb841a88d3e4f

Merge: d45ef03 a9870d9

Author: bstachmann <bstachmann@yahoo.de>

Merge pull request #24 from kapitel26/auflage-4/bst-lektorat

Anzahl Commits:

2070

Änderungsstatistik:

789 files changed, 340611 insertions(+), 228 deletions(-)

Status:

On branch auflage-4/auflage-4.0

Your branch is up-to-date with 'origin/auflage-4/auflage-4.0'.

nothing to commit, working directory clean

---

# Vorwort

## Warum Git?

Git hat eine rasante Erfolgsgeschichte hinter sich. Im April 2005 begann Linus Torvalds, Git zu implementieren, weil er keinen Gefallen an den damals verfügbaren Open-Source-Versionsverwaltungen fand. Heute, im Herbst 2016, liefert Google Millionen von Suchtreffern, wenn man nach »git version control« sucht. Für neue Open-Source-Projekte ist es zum Standard geworden, und viele große Open-Source-Projekte sind zu Git migriert.

**Arbeiten mit Branches:** Wenn viele Entwickler gemeinsam an einer Software arbeiten, entstehen parallele Entwicklungsstränge, die immer wieder auseinanderlaufen und zusammengeführt werden müssen. Genau dafür ist Git entwickelt worden. Es bietet daher umfassende Unterstützung zum *Branchen*, *Mergen*, *Rebasen* und *Cherry-Picken*.

**Flexibilität in den Workflows:** Manche sagen, dass Git im Grunde gar keine Versionsverwaltung sei, sondern ein Baukasten, aus dem sich jeder seine eigene Versionsverwaltung zusammensetzen kann. Git ist außergewöhnlich flexibel. Ein einzelner Entwickler kann es für sich allein nutzen, agile Teams finden leichtgewichtige Arbeitsweisen damit, aber auch große internationale Projekte mit zahlreichen Entwicklern an mehreren Standorten können passende Workflows entwickeln.

**Contribution:** Die meisten Open-Source-Projekte existieren durch freiwillige Beiträge von Entwicklern. Es ist wichtig, das Beitreten so einfach wie nur möglich zu machen. Bei zentralen Versionsverwaltungen wird dies oft erschwert, weil man nicht jedem schreibenden Zugriff auf das Repository geben möchte. Jeder kann ein Git-Repository klonen, damit vollwertig arbeiten und dann später die Änderungen weitergeben (»Mit Forks entwickeln« (Seite 163)).

**Nachvollziehbare Herkunft von Sourcecode:** Die Entwickler von Git haben es als *Content Tracker* bezeichnet. Damit meinen sie ein Werkzeug, das die Herkunft von Inhalten, insbesondere Source-

code, aufzeigen kann. Git kann dies selbst dann, wenn Code zusammengeführt wurde (*Merge*) oder Dateien verschoben und umbenannt wurden. Sogar kopierte Codeabschnitte können erkannt und zugeordnet werden.

**Performance:** Auch bei Projekten mit vielen Dateien und langen Historien bleibt Git schnell. In weniger als einer halben Minute wechselt es zum Beispiel von der aktuellen Version auf eine sechs Jahre ältere Version der Linux-Kernel-Sourcen – auf einem kleinen MacBook Air. Das kann sich sehen lassen, wenn man bedenkt, dass über 200.000 Commits und 40.000 veränderte Dateien dazwischenliegen.

**Robust gegen Fehler und Angriffe:** Da die Historie auf viele dezentrale Repositorys verteilt wird, ist ein schwerwiegender Datenverlust unwahrscheinlich. Eine genial simple Datenstruktur im Repository sorgt dafür, dass die Daten auch in ferner Zukunft interpretierbar bleiben. Der durchgängige Einsatz kryptografischer Prüfsummen erschwert es Angreifern, Repositorys unbemerkt zu korrumppieren.

**Offline- und Multisite-Entwicklung:** Die dezentrale Architektur macht es leicht, offline zu entwickeln, etwa unterwegs mit dem Laptop. Bei der Entwicklung an mehreren Standorten ist weder ein zentraler Server noch eine dauerhafte Netzwerkverbindung erforderlich.

**Administrierbarkeit:** Git ist einfach zu betreiben und zu administrieren. Alle Daten und Konfigurationen werden in einfachen Dateien gespeichert. Für Backups oder Umzüge genügen die Standardtools des Betriebssystems. Es muss kein Git-spezifischer Dienst eingerichtet werden. Alle Operationen werden durch Kommandozeilenbefehle bereitgestellt. Repositorys werden meist über SSH oder HTTP zugänglich gemacht, sodass man die Authentifizierung und Autorisierung des Betriebssystems bzw. Webservers nutzen kann. Zahlreiche mächtige Befehle erlauben es, das Repository zu manipulieren. Die dezentrale Natur von Git macht es leicht, Änderungen zuerst an einem Klon zu erproben, bevor man sie öffentlich macht.

**Starke Open-Source-Community:** Neben der detaillierten offiziellen Dokumentation unterstützen zahlreiche Anleitungen, Foren, Wikis etc. den Anwender. Es existiert ein Ökosystem aus Tools, Hosting-Plattformen, Publikationen, Dienstleistern und Plug-ins für Entwicklungsumgebungen, und es wächst stark.

**Erweiterbarkeit:** Git bietet neben komfortablen Befehlen für den Anwender auch elementare Befehle, die einen direkteren Zugang zum Repository erlauben. Dies macht Git sehr flexibel und ermöglicht individuelle Anwendungen, die über das hinausgehen, was Git von Haus aus bietet.

## Neues in der vierten Auflage

Vor einiger Zeit haben wir eine Befragung unter Git-Anwendern durchgeführt. Eine Erkenntnis dabei war, dass heute fast alle Projekte Plattformen wie GitHub, Bitbucket oder GitLab nutzen, um ihre Git-Repositorys zu managen. Wir haben die Workflows »Ein Projekt aufsetzen« und »Mit Feature-Banches entwickeln« dementsprechend angepasst und zeigen, wie man auf solchen Plattformen Projekte einrichtet und wie man dort mithilfe sogenannter *Pull-Requests* zusammenarbeiten kann.

Open-Source-Projekte auf GitHub arbeiten oft ohne Feature-Banches. Sie nutzen stattdessen *Pull-Requests* und sogenannte *Forks*. Das sind Klone des originalen Repository, mit denen ein Entwickler unabhängig arbeiten kann, die aber eine Referenz auf das ursprüngliche Projekt behalten, damit Änderungen aus den *Forks* später leicht übernommen werden können. Einen solchen Workflow haben wir jetzt beschrieben.

- »Ein Projekt aufsetzen« (Seite 123)
- »Mit Feature-Banches entwickeln« (Seite 143)
- »Mit Forks entwickeln« (Seite 163)

Die Einführung »Erste Schritte mit der Kommandozeile« ab Seite 9 wurde um Hinweise ergänzt, die Windows-Usern den Einstieg in das Arbeiten mit der Git-Bash erleichtern.

Immer mehr Entwickler nutzen das Rebasing-Feature von Git. Deshalb haben wir den Abschnitt »Empfehlungen zum Rebasing« ab Seite 85 neu geschrieben.

Neu hinzugekommen ist auch ein Workflow über die *LFS*-Erweiterung, die es ermöglicht, große Binärdateien mit Git zu verwalten:

- »Ein Projekt mit großen binären Dateien versionieren« (Seite 213)

Ebenfalls neu hinzugekommen ist ein Abschnitt über das Worktree-Feature:

- »Worktrees – mehrere Workspaces mit einem Repository« ab Seite 289

Bücher neigen dazu, von Auflage zu Auflage dicker zu werden. Wir möchten aber, dass dieses Buch ein kompakter Leitfaden für das Arbeiten mit Git bleibt. Deshalb beschlossen wir, dass es ein wenig abspecken soll, und entfernten ein paar Abschnitte über selten genutzte Features von Git. Man findet sie jetzt in unserem Blog:

- »Mit Bisection Fehler suchen«  
<http://kapitel26.github.io/git/2016/10/02/git-bisect>
- »Andere Versionsverwaltungen parallel nutzen«  
<http://kapitel26.github.io/git/2016/10/02/git-parallel-vcs>
- »Patches per Mail versenden«  
<http://kapitel26.github.io/git/2016/10/02/git-bundles>
- »Bundles – Pull im Offline-Modus«  
<http://kapitel26.github.io/git/2016/10/02/git-bundles>
- »Git Notes – Notizen an Commits«  
<http://kapitel26.github.io/git/2016/10/02/git-commit-notes>



## Ein Buch für professionelle Entwickler

Wenn Sie Entwickler sind, im Team Software herstellen und wissen wollen, wie man Git effektiv einsetzt, dann halten Sie jetzt das richtige Buch in der Hand. Dieses Buch ist kein theorielastiger Wälzer und auch kein umfassendes Nachschlagewerk. Es beschreibt nicht alle Befehle von Git (es sind mehr als 100). Es beschreibt erst recht nicht alle Optionen (einige Befehle bieten über 50 an). Stattdessen beschreibt dieses Buch, wie man Git in typischen Projektsituationen einsetzen kann, z. B. wie man ein Git-Projekt aufsetzt oder wie man mit Git ein Release durchführt.

## Die Zutaten

**Gleich ausprobieren!**  
→ Seite 9

**Erste Schritte:** Auf weniger als einem Dutzend Seiten zeigt ein Beispiel alle wichtigen Git-Befehle.

**Was sind Commits?**  
→ Seite 31

**Einführung:** Auf weniger als hundert Seiten erfahren Sie, was man benötigt, um mit Git im Team arbeiten zu können. Zahlreiche Beispiele zeigen, wie man die wichtigsten Git-Befehle anwendet. Dar-

über hinaus werden wesentliche Grundbegriffe, wie zum Beispiel Commit, Repository, Branch, Merge oder Rebase, erklärt, die Ihnen helfen zu verstehen, wie Git funktioniert, damit Sie die Befehle gezielter einsetzen können. Hier finden Sie auch einen Abschnitt mit Tipps und Tricks, die man nicht jeden Tag braucht, die aber manchmal nützlich sein können.

**Workflows:** Workflows beschreiben Szenarien, wie man Git im Projekt einsetzen kann, zum Beispiel wenn man ein Release durchführen möchte (»Periodisch Releases durchführen« (Seite 185)). Für jeden Workflow wird beschrieben,

- welches Problem er löst,
- welche Voraussetzungen dazu gegeben sein müssen und
- wer wann was zu tun hat, damit das gewünschte Ergebnis erreicht wird.

»Warum nicht anders?«-Abschnitte: Jeder Workflow beschreibt genau einen konkreten Lösungsweg. In Git gibt es häufig sehr unterschiedliche Wege, um dasselbe Ziel zu erreichen. Im letzten Teil eines jeden Workflow-Kapitels wird erklärt, warum wir genau diese eine Lösung gewählt haben. Dort werden auch Varianten und Alternativen erwähnt, die für Sie interessant sind, wenn in Ihrem Projekt andere Voraussetzungen gegeben sind, oder wenn Sie mehr über die Hintergründe wissen wollen.

**Schritt-für-Schritt-Anleitungen:** Häufig benötigte Befehlsfolgen, wie zum Beispiel »Branch erstellen« (Seite 62), haben wir in Schritt-für-Schritt-Anleitungen beschrieben.

#### **Tipps und Tricks**

→ Seite 109

#### **Workflow-Verzeichnis**

→ Seite 307

#### **Schritt-für-Schritt-Anleitungen**

→ Seite 305

## Warum Workflows?

Git ist extrem flexibel. Das ist gut, weil es für die unterschiedlichsten Projekte taugt. Vom einzelnen Sysadmin, der »mal eben« ein paar Shell-Skripte versioniert, bis hin zum Linux-Kernel-Projekt, an dem Hunderte von Entwicklern arbeiten, ist jeder damit gut bedient. Diese Flexibilität hat jedoch ihren Preis. Wer mit Git zu arbeiten beginnt, muss viele Entscheidungen treffen. Zum Beispiel:

- In Git hat man dezentrale Repositorys. Aber möchte man wirklich nur dezentral arbeiten? Oder richtet man doch lieber ein zentrales Repository ein?
- Git unterstützt zwei Richtungen für den Datentransfer: Push und Pull. Benutzt man beide? Falls ja: Wofür verwendet man das eine? Wofür das andere?

- Branching und Merging ist eine Stärke von Git. Aber wie viele Branches öffnet man? Einen für jedes Feature? Einen für jedes Release? Oder überhaupt nur einen?

Um den Einstieg zu erleichtern, haben wir 12 Workflows beschrieben:

- Die Workflows sind Arbeitsabläufe für den Projektalltag.
- Die Workflows geben konkrete Handlungsanweisungen.
- Die Workflows zeigen die benötigten Befehle und Optionen.
- Die Workflows eignen sich gut für eng zusammenarbeitende Teams, wie man sie in modernen Softwareprojekten häufig antrifft.
- Die Workflows sind *nicht* die einzige richtige Lösung für das jeweilige Problem. Aber sie sind ein guter Startpunkt, von dem man ausgehen kann, um optimale Workflows für das eigene Projekt zu entwickeln.

Wir konzentrieren uns auf die agile Entwicklung im Team für kommerzielle Projekte, weil wir glauben, dass sehr viele professionelle Entwickler (die Autoren inklusive) in solchen Umgebungen arbeiten. Nicht berücksichtigt haben wir die speziellen Anforderungen, die sich für Großprojekte ergeben, weil sie die Workflows deutlich aufgebläht hätten und weil wir glauben, dass sie für die meisten Entwickler nicht so interessant sind. Ebenfalls unberücksichtigt bleibt die Open-Source-Entwicklung, obwohl es auch dafür sehr interessante Workflows mit Git gibt.

## Tipps zum Querlesen

Als Autoren wünschen wir uns natürlich, dass Sie unser Buch von Seite 1 bis Seite 302 am Stück verschlingen, ohne es zwischendrin aus der Hand zu legen. Aber, mal ehrlich: Haben Sie genug Zeit, um heute noch mehr als ein paar Seiten zu lesen? Wir vermuten, dass in Ihrem Projekt gerade die Hölle los ist und dass das Arbeiten mit Git nur eines von hundert Themen ist, mit denen Sie sich gerade beschäftigen. Deshalb haben wir uns Mühe gegeben, das Buch so zu gestalten, dass man es gut querlesen kann. Hier sind ein paar Tipps dazu:

### Muss ich die Einführungskapitel lesen, um die Workflows zu verstehen?

Falls Sie noch keine Vorkenntnisse in Git haben, sollten Sie das tun. Grundlegende Befehle und Prinzipien sollten Sie kennen, um die Workflows korrekt einsetzen zu können.

## Ich habe schon mit Git gearbeitet. Welche Kapitel kann ich überspringen?

Auf der letzten Seite in jedem Einführungskapitel 1 bis 14 gibt es eine Zusammenfassung der Inhalte in Stichworten. Dort können Sie sehr schnell sehen, ob es in dem Kapitel für Sie noch Dinge zu entdecken gibt oder ob Sie es überspringen können. Die folgenden Kapitel können Sie relativ gut überspringen, weil sie nur für einige Workflows relevant sind:

- Kapitel 6, Das Repository
- Kapitel 9, Mit Rebasing die Historie glätten
- Kapitel 12, Versionen markieren
- Kapitel 29, Abhängigkeiten zwischen Repositorys
- Kapitel 13, Tipps und Tricks

Zusammenfassung am Ende der Einführungskapitel

Überspringen Sie diese Kapitel, wenn Sie es eilig haben.

## Wo finde ich was?

**Workflows:** Ein Verzeichnis aller Workflows mit Kurzbeschreibungen und Überblicksabbildung finden Sie im Anhang.

**Workflow-Verzeichnis**  
→ Seite 307

**Schritt-für-Schritt-Anleitungen:** Wir haben alle Anleitungen im Anhang aufgelistet.

**Anleitungsverzeichnis**  
→ Seite 305

**Befehle und Optionen:** Wenn Sie beispielsweise wissen wollen, wie man die Option `find-copies-harder` verwendet und zu welchem Befehl sie gehört, dann schauen Sie in den Index. Dort sind fast alle Verwendungen von Befehlen und Optionen aufgeführt. Oft haben wir die Nummer jener Seite fett hervorgehoben, wo Sie am meisten Informationen zu dem Befehl oder der Option finden.

**Index** → Seite 312

**Fachbegriffe:** Fachbegriffe, wie zum Beispiel »First-Parent-History« oder »Remote-Tracking-Branch«, finden Sie natürlich auch im Index.

**Index** → Seite 312

## Beispiele und Notation

In den »Erste Schritte«-Kapiteln verwenden wir einige Git-Fachbegriffe, die wir erst in späteren Kapiteln ausführlicher behandeln. Diese sind kursiv gesetzt, z. B. *Repository*. Im Index sind jene Seitenzahlen fett hervorgehoben, wo der Begriff dann definiert oder ausführlicher erläutert wird.

**Index** → Seite 312

**Grafische Werkzeuge  
für Git → Seite 292**

Viele Beispiele in diesem Buch beschreiben wir mit Kommandozeilenaufufen. Das soll nicht heißen, dass es dafür keine grafischen Benutzeroberflächen gibt. Im Gegenteil: Git bringt zwei einfache grafische Anwendungen bereits mit: `gitk` und `git-gui`. Darüber hinaus gibt es zahlreiche Git-Frontends (z. B. Atlassian SourceTree<sup>1</sup>, TortoiseGit<sup>2</sup>, SmartGit<sup>3</sup>, GitX<sup>4</sup>, Git Extensions<sup>5</sup>, tig<sup>6</sup>, qgit<sup>7</sup>), einige Entwicklungsumgebungen, die Git von Haus aus unterstützen (IntelliJ<sup>8</sup>, Xcode 4<sup>9</sup>), und viele Plug-ins für Entwicklungsumgebungen (z. B. EGit für Eclipse<sup>10</sup>, NBGit für NetBeans<sup>11</sup>, Git Extensions für Visual Studio<sup>12</sup>). Wir haben uns trotzdem für die Kommandozeilenbeispiele entschieden, weil

- Git-Komandozeilenbefehle auf allen Plattformen fast gleich funktionieren,
- die Beispiele auch mit künftigen Versionen funktionieren werden,
- man damit Workflows sehr kompakt darstellen kann und weil
- wir glauben, dass das Arbeiten mit der Komandozeile für viele Anwendungsfälle unschlagbar effizient ist.

In den Beispielen arbeiten wir mit der Bash-Shell, die auf Linux- und Mac-OS-Systemen standardmäßig vorhanden ist. Auf Windows-Systemen kann man die »Git-Bash«-Shell (sie ist in der »msysgit«-Installation enthalten) oder »cygwin« verwenden. Die Komandozeilenaufufe stellen wir wie folgt dar:

```
> git commit
```

An den Stellen, wo es inhaltlich interessant ist, zeigen wir auch die Antwort, die Git liefert hat, in etwas kleinerer Schrift dahinter an:

---

<sup>1</sup> <http://www.sourcetreeapp.com/>

<sup>2</sup> <http://code.google.com/p/tortoisegit/>

<sup>3</sup> <http://www.syntevo.com/smartygit/>

<sup>4</sup> <http://gitx.frim.nl/>

<sup>5</sup> <http://code.google.com/p/gitextensions/>

<sup>6</sup> <http://jonas.nitro.dk/tig/>

<sup>7</sup> <http://sourceforge.net/projects/qgit/>

<sup>8</sup> <http://www.jetbrains.com/idea/>

<sup>9</sup> <http://developer.apple.com/technologies/tools/>

<sup>10</sup> <http://eclipse.org/egit/>

<sup>11</sup> <http://nbgit.org/>

<sup>12</sup> <http://code.google.com/p/gitextensions/>

```
> git --version  
git version 2.3.7
```

Eine kurze Einführung in das Arbeiten mit Git-User-Interfaces zeigen wir am Beispiel von SourceTree in Kapitel »Erste Schritte mit SourceTree« ab Seite 23. Wo wir auf Menüpunkte oder Buttons Bezug nehmen, setzen wir diese in Kapitälchen, z. B. ANSICHT | AKTUALISIEREN.

## Danksagungen

### Danksagungen zur ersten Auflage

Rückblickend sind wir erstaunt, wie viele Leute auf die eine oder andere Weise zum Entstehen dieses Buchs beigetragen haben. Wir möchten uns ganz herzlich bei all jenen bedanken, ohne die dieses Buch nicht das geworden wäre, was es jetzt ist.

An erster Stelle danken wir Anke, Jan, Elke und Annika, die sich inzwischen kaum noch daran erinnern, wie wir ohne einen Laptop unter den Fingern aussehen.

Dann danken wir dem freundlichen Team vom dpunkt.verlag, insbesondere Vanessa Wittmer, Nadine Thiele und Ursula Zimpfer. Besonderer Dank gebührt aber René Schönfeldt, der das Projekt angestoßen und vom ersten Tag bis zur letzten Korrektur begleitet hat. Außerdem bekennen wir uns bei Maurice Kowalski, Jochen Schlosser, Oliver Zeigermann, Ralf Degner, Michael Schulze-Ruhfus und einem halben Dutzend alterer Gutachter für die wertvollen inhaltlichen Beiträge, die sehr geholfen haben, das Buch besser zu machen. Für den allerersten Anstoß danken wir Matthias Veit, der eines Tages zu Björn kam und meinte, Subversion wäre nun doch schon etwas in die Jahre gekommen und man solle sich doch mal nach etwas Schönerem umsehen, zum Beispiel gäbe es da so ein Tool, das die Entwickler des Linux-Kernels neuerdings nutzen würden ...

### Danksagungen zur zweiten Auflage

Ein besonders herzlicher Dank geht an unseren Leser Herrn Ulrich Windl, der das Buch aufmerksamer gelesen hat als die meisten und uns eine lange Liste von Korrekturvorschlägen, Fragen und Verbesserungsvorschlägen zugesandt hat. Sie haben wesentlich dazu beigetragen, dass diese Auflage besser und präziser ist als die erste.

Ebenfalls danken wir Henrik Heine, Malte Finsterwalder und Tjabo Vierbücher für gute Hinweise und Korrekturvorschläge.

## Danksagungen zur dritten Auflage

Dieses Mal gab es nicht so viel Feedback. Ein paar kleinere Fehlermeldungen haben wir dennoch erhalten, und so konnten wir das Buch wieder ein wenig verbessern. Vielen Dank dafür.

## Danksagungen zur vierten Auflage

Nützliche Hinweise haben wir erhalten von: Thomas Braun und Herbert Feichtinger. Vielen Dank dafür!

## »Standing on the Shoulders of Giants«

Ein besonderer Dank geht an Linus Torvalds, Junio C. Hamano und die vielen Committer im Git-Projekt dafür, dass sie der Entwickler-Community dieses fantastische Tool geschenkt haben.

# Inhaltsverzeichnis

## Erste Schritte

<b>1</b>	<b>Grundlegende Konzepte .....</b>	<b>1</b>
1.1	Dezentrale Versionsverwaltung – alles anders? .....	1
1.2	Das Repository – die Grundlage dezentralen Arbeitens .....	3
1.3	Branching und Merging – ganz einfach! .....	5
1.4	Zusammenfassung .....	7
<b>2</b>	<b>Erste Schritte mit der Kommandozeile .....</b>	<b>9</b>
2.1	Git einrichten .....	9
2.2	Ein paar Hinweise für Windows-User .....	9
2.3	Git einrichten .....	11
2.4	Das erste Projekt mit Git .....	12
2.5	Zusammenarbeit mit Git .....	15
2.6	Zusammenfassung .....	21
<b>3</b>	<b>Erste Schritte mit SourceTree .....</b>	<b>23</b>
3.1	SourceTree konfigurieren .....	23
3.2	Das erste Projekt mit Git .....	23
3.3	Zusammenarbeit mit Git .....	26
3.4	Zusammenfassung .....	30

## Arbeiten mit Git

<b>4</b>	<b>Was sind Commits? .....</b>	<b>31</b>
4.1	Zugriffsberechtigungen und Zeitstempel .....	32
4.2	Die Befehle add und commit .....	32
4.3	Exkurs: Mehr über Commit-Hashes .....	33
4.4	Eine Historie von Commits .....	34
4.5	Eine etwas andere Sichtweise auf Commits .....	34
4.6	Viele unterschiedliche Historien desselben Projekts .....	36
4.7	Zusammenfassung .....	38

<b>5</b>	<b>Commits zusammenstellen .....</b>	<b>39</b>
5.1	Der status-Befehl .....	39
5.2	Der Stage-Bereich speichert Momentaufnahmen .....	43
5.3	Was tun mit Änderungen, die nicht übernommen werden sollen? .....	45
5.4	Mit .gitignore Dateien unversioniert lassen .....	46
5.5	Stashing: Änderungen zwischenspeichern .....	47
5.6	Zusammenfassung .....	48
<b>6</b>	<b>Das Repository .....</b>	<b>49</b>
6.1	Ein einfaches und effizientes Speichersystem .....	49
6.2	Verzeichnisse speichern: Blob und Tree .....	50
6.3	Gleiche Daten werden nur einmal gespeichert .....	51
6.4	Kompression ähnlicher Inhalte .....	51
6.5	Ist es schlimm, wenn verschiedene Daten zufällig denselben Hashwert bekommen? .....	52
6.6	Commits .....	52
6.7	Wiederverwendung von Objekten in der Commit-Historie ...	53
6.8	Umbenennen, verschieben und kopieren .....	54
6.9	Zusammenfassung .....	56
<b>7</b>	<b>Branches verzweigen .....</b>	<b>59</b>
7.1	Parallele Entwicklung .....	59
7.2	Bugfixes in älteren Versionen .....	60
7.3	Branches .....	60
7.4	Aktiver Branch .....	61
7.5	Branch-Zeiger umsetzen .....	64
7.6	Branch löschen .....	64
7.7	Und was ist, wenn man die Commit-Objekte wirklich loswerden will? .....	66
7.8	Zusammenfassung .....	66
<b>8</b>	<b>Branches zusammenführen .....</b>	<b>67</b>
8.1	Was passiert bei einem Merge? .....	68
8.2	Konflikte .....	69
8.3	Fast-Forward-Merges .....	74
8.4	First-Parent-History .....	75
8.5	Knifflige Merge-Konflikte .....	76
8.6	Zusammenfassung .....	78
<b>9</b>	<b>Mit Rebasing die Historie glätten .....</b>	<b>81</b>
9.1	Das Prinzip: Kopieren von Commits .....	81
9.2	Und wenn es zu Konflikten kommt? .....	83

---

9.3	Was passiert mit den ursprünglichen Commits nach dem Rebasing? .....	84
9.4	Empfehlungen zum Rebasing .....	85
9.5	Cherry-Picking .....	88
9.6	Zusammenfassung .....	88
<b>10</b>	<b>Repositorys erstellen, klonen und verwalten .....</b>	<b>89</b>
10.1	Ein Repository erstellen .....	89
10.2	Das Repository-Layout .....	89
10.3	Bare-Repositorys .....	90
10.4	Vorhandene Dateien übernehmen.....	90
10.5	Ein Repository klonen .....	91
10.6	Wie sagt man Git, wo das Remote-Repository liegt?.....	91
10.7	Kurznamen für Repositorys: Remotes .....	92
10.8	Zusammenfassung .....	93
<b>11</b>	<b>Austausch zwischen Repositorys .....</b>	<b>95</b>
11.1	Fetch, Pull und Push .....	95
11.2	Remote-Tracking-Banches .....	96
11.3	Einen Remote-Branch bearbeiten .....	97
11.4	Ein paar Begriffe, die man kennen sollte .....	98
11.5	Fetch: Branches aus einem anderen Repository holen .....	99
11.6	Fetch: Aufrufvarianten .....	99
11.7	Erweiterte Möglichkeiten .....	104
11.8	Zusammenfassung .....	104
<b>12</b>	<b>Versionen markieren .....</b>	<b>105</b>
12.1	Arbeiten mit Tags erstellen .....	105
12.2	Welche Tags gibt es? .....	106
12.3	Die Hashes zu den Tags ausgeben .....	106
12.4	Die Log-Ausgaben um Tags anreichern .....	107
12.5	In welcher Version ist es »drin«?.....	107
12.6	Wie verschiebt man ein Tag? .....	107
12.7	Und wenn ich ein »Floating Tag« brauche? .....	108
12.8	Zusammenfassung .....	108
<b>13</b>	<b>Tipps und Tricks .....</b>	<b>109</b>
13.1	Keine Panik – es gibt ein Reflog! .....	109
13.2	Lokale Änderungen temporär ignorieren.....	110
13.3	Änderungen an Textdateien untersuchen .....	111
13.4	alias – Abkürzungen für Git-Befehle .....	112
13.5	Branches als temporäre Zeiger auf Commits nutzen .....	113
13.6	Commits auf einen anderen Branch verschieben .....	114
13.7	Mehr Kontrolle bei Fetch, Push und Pull .....	115

**Workflows**

14	Workflow-Einführung .....	117
14.1	Warum Workflows? .....	117
14.2	Welche Workflows sind wann sinnvoll? .....	118
14.3	Aufbau der Workflows .....	119

**Workflows: Entwickeln mit Git**

15	Ein Projekt aufsetzen.....	123
16	Gemeinsam auf einem Branch entwickeln .....	135
17	Mit Feature-Banches entwickeln .....	143
18	Mit Forks entwickeln .....	163

**Workflows: Release-Prozess**

19	Kontinuierlich Releases durchführen .....	175
20	Periodisch Releases durchführen .....	185
21	Mit mehreren aktiven Releases arbeiten .....	199

**Workflows: Repositorys pflegen**

22	Ein Projekt mit großen binären Dateien versionieren .....	213
23	Große Projekte aufteilen .....	221
24	Kleine Projekte zusammenführen .....	229
25	Lange Historien auslagern .....	235
26	<a href="http://kapitel26.github.io">http://kapitel26.github.io</a> .....	245
27	Ein Projekt nach Git migrieren .....	247

## Mehr über Git

<b>28</b>	<b>Integration mit Jenkins .....</b>	<b>263</b>
28.1	Vorbereitungen .....	263
28.2	Ein einfaches Git-Projekt einrichten .....	264
28.3	Hook als Build-Auslöser .....	265
28.4	Ein Tag für jeden erfolgreichen Build .....	267
28.5	Pull-Requests bauen .....	269
28.6	Automatischer Merge von Branches .....	272
<b>29</b>	<b>Abhängigkeiten zwischen Repositorys .....</b>	<b>275</b>
29.1	Abhängigkeiten mit Submodulen .....	275
29.2	Abhängigkeiten mit Subtrees .....	281
29.3	Zusammenfassung .....	286
<b>30</b>	<b>Was gibt es sonst noch? .....</b>	<b>289</b>
30.1	Worktrees – mehrere Workspaces mit einem Repository .....	289
30.2	Interaktives Rebasing – Historie verschönern .....	290
30.3	Umgang mit Patches .....	291
30.4	Archive erstellen .....	291
30.5	Grafische Werkzeuge für Git .....	292
30.6	Repository im Webbrowser anschauen .....	293
30.7	Zusammenarbeit mit Subversion .....	294
30.8	Hooks – Git erweitern .....	294
30.9	Mit Bisection Fehler suchen .....	294
<b>31</b>	<b>Die Grenzen von Git .....</b>	<b>297</b>
31.1	Hohe Komplexität .....	297
31.2	Komplizierter Umgang mit Submodulen .....	299
31.3	Ressourcenverbrauch bei großen binären Dateien .....	300
31.4	Repositorys können nur vollständig verwendet werden .....	300
31.5	Autorisierung nur auf dem ganzen Repository .....	301
31.6	Mäßige grafische Werkzeuge für die Historienauswertung ...	302

## Anhang

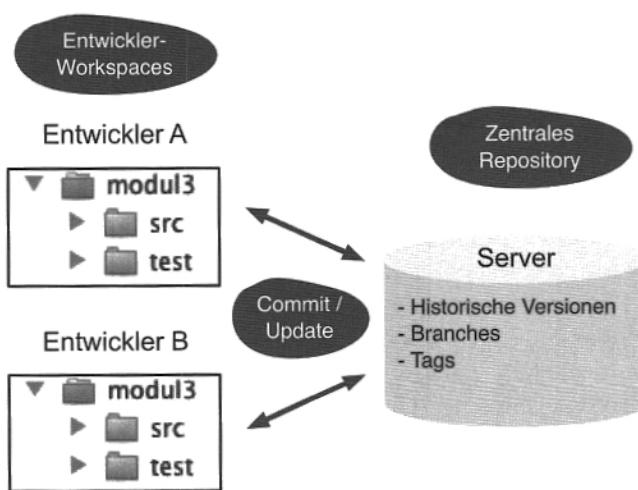
Schritt-für-Schritt-Anleitungen .....	305
Workflow-Verzeichnis .....	307
Index .....	313

# 1 Grundlegende Konzepte

Dieses Kapitel macht Sie mit den Ideen einer dezentralen Versionsverwaltung vertraut und zeigt die Unterschiede zu einer zentralen Versionsverwaltung auf. Anschließend erfahren Sie, wie dezentrale *Repositories* funktionieren und warum Branching und Merging keine fortgeschrittenen Themen in Git sind.

## 1.1 Dezentrale Versionsverwaltung – alles anders?

Bevor wir uns den Konzepten der dezentralen Versionsverwaltung widmen, werfen wir kurz einen Blick auf die klassische Architektur zentraler Versionsverwaltungen.

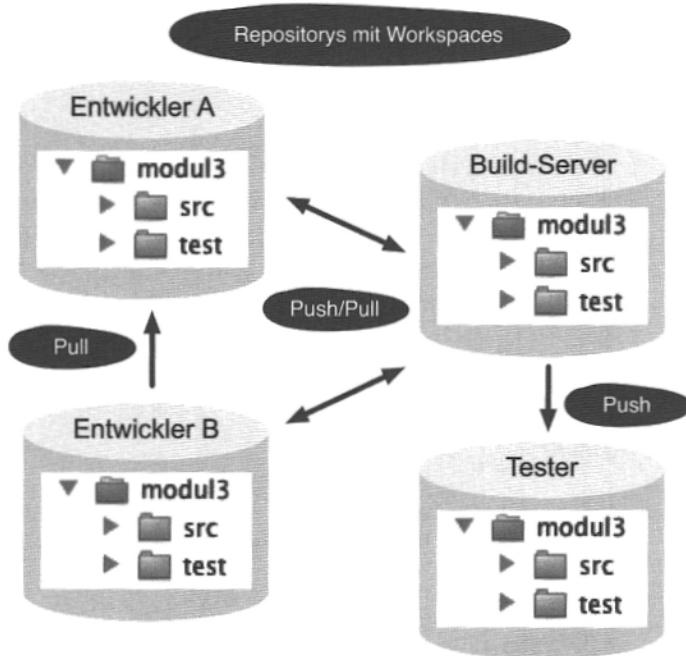


**Abb. 1–1**  
Zentrale  
Versionsverwaltung

Abbildung 1–1 zeigt die typische Aufteilung einer zentralen Versionsverwaltung, z. B. von CVS oder Subversion. Jeder Entwickler hat auf seinem Rechner ein Arbeitsverzeichnis (*Workspace*) mit allen Projektdateien. Diese bearbeitet er und schickt die Änderungen regelmäßig per *Commit* an den zentralen Server. Per *Update* holt er die Änderungen

der anderen Entwickler ab. Der zentrale Server speichert die aktuellen und historischen Versionen der Dateien (*Repository*). Parallelle Entwicklungsstränge (*Branches*) und benannte Versionen (*Tags*) werden auch zentral verwaltet.

**Abb. 1–2**  
Dezentrale  
Versionsverwaltung



#### **Das Repository**

→ Seite 49

Bei einer dezentralen Versionsverwaltung (Abbildung 1–2) gibt es keine Trennung zwischen Entwickler- und Serverumgebung. Jeder Entwickler hat sowohl einen *Workspace* mit den in Arbeit befindlichen Dateien als auch ein eigenes lokales *Repository* (genannt *Klon*) mit allen Versionen, *Branches* und *Tags*. Änderungen werden auch hier durch ein *Commit* festgeschrieben, jedoch zunächst nur im lokalen *Repository*. Andere Entwickler sehen die neuen Versionen nicht sofort. *Push-* und *Pull-*Befehle übertragen Änderungen dann von einem *Repository* zum anderen. Technisch gesehen sind in der dezentralen Architektur alle *Repositorys* gleichwertig. Theoretisch bräuchte es keinen Server: Man könnte alle Änderungen direkt von Entwicklerrechner zu Entwicklerrechner übertragen. In der Praxis spielen *Repositorys* auf Servern auch in Git eine wichtige Rolle, zum Beispiel in Form von folgenden spezifischen *Repositorys*:

#### **Was sind Commits?**

→ Seite 31

#### **Austausch zwischen Repositorys** → Seite 95

**Blessed Repository:** Aus diesem *Repository* werden die »offiziellen« Releases erstellt.

**Ein Projekt aufsetzen**  
→ Seite 123

**Shared Repository:** Dieses Repository dient dem Austausch zwischen den Entwicklern im Team. In kleinen Projekten kann hierzu auch das *Blessed Repository* genutzt werden. Bei einer Multisite-Entwicklung kann es auch mehrere geben.

**Workflow Repository:** Ein solches *Repository* wird nur mit Änderungen befüllt, die einen bestimmten Status im Workflow erreicht haben, z. B. nach erfolgreichem Review.

**Fork Repository:** Dieses Repository dient der Entkopplung von der Entwicklungshauptlinie (zum Beispiel für große Umbauten, die nicht in den normalen Release-Zyklus passen) oder für experimentelle Entwicklungen, die vielleicht nie in den Hauptstrang einfließen sollen.

Folgende Vorteile ergeben sich aus dem dezentralen Vorgehen:

**Hohe Performance:** Fast alle Operationen werden ohne Netzwerkzugriff lokal durchgeführt.

**Effiziente Arbeitsweisen:** Entwickler können lokale *Branches* benutzen, um schnell zwischen verschiedenen Aufgaben zu wechseln.

**Offline-Fähigkeit:** Entwickler können ohne Serververbindung *Commits* durchführen, *Branches* anlegen, Versionen taggen etc. und diese erst später übertragen.

**Flexibilität der Entwicklungsprozesse:** In Teams und Unternehmen können spezielle *Repositories* angelegt werden, um mit anderen Abteilungen, z. B. den Testern, zu kommunizieren. Änderungen werden einfach durch ein Push in dieses *Repository* freigegeben.

**Backup:** Jeder Entwickler hat eine Kopie des *Repositories* mit einer vollständigen Historie. Somit ist die Wahrscheinlichkeit minimal, durch einen Serverausfall Daten zu verlieren.

**Wartbarkeit:** Knifflige Umstrukturierungen kann man zunächst auf einer Kopie des *Repositories* erproben, bevor man sie in das Original-*Repository* überträgt.

## 1.2 Das Repository – die Grundlage dezentralen Arbeitens

Das *Repository* ist im Kern ein effizienter Datenspeicher. Im Wesentlichen enthält es:

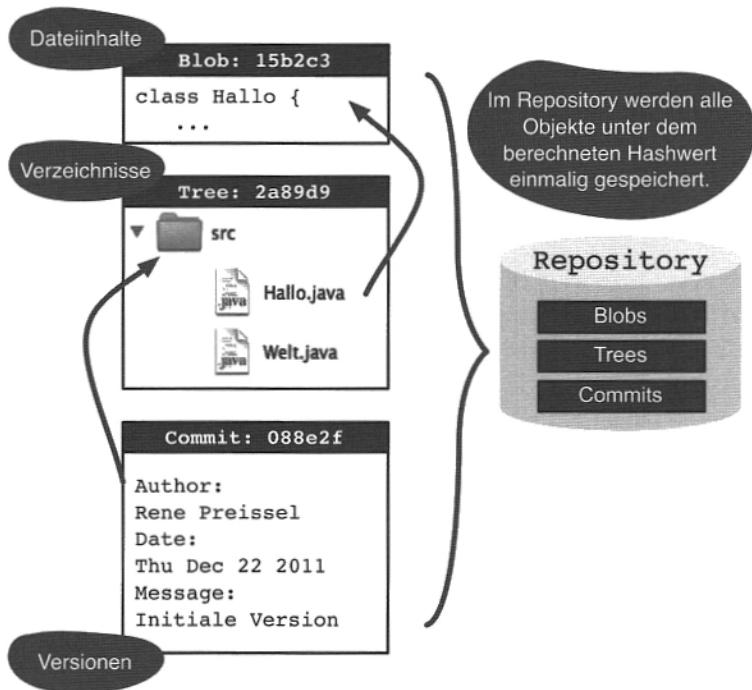
**Das Repository**  
→ Seite 49

**Inhalte von Dateien (Blobs):** Dies sind Texte oder binäre Daten. Die Daten werden unabhängig von Dateinamen gespeichert.

**Verzeichnisse (Trees):** Verzeichnisse verknüpfen Dateinamen mit Inhalten. Verzeichnisse können wiederum andere Verzeichnisse beinhalten.

**Versionen (Commits):** Versionen definieren einen wiederherstellbaren Zustand eines Verzeichnisses. Beim Anlegen einer neuen Version werden der Autor, die Uhrzeit, ein Kommentar und die Vorgängerversion gespeichert.

**Abb. 1–3**  
Ablage von Objekten im Repository



Für alle Daten wird ein hexadezimaler *Hashwert* berechnet, z. B. `1632acb65b01c6b621d6e1105205773931bb1a41`. Diese *Hashwerte* dienen als Referenz zwischen den Objekten und als Schlüssel, um die Daten später wiederzufinden (Abbildung 1–3).

Die *Hashwerte* von *Commits* sind die »Versionsnummern« von Git. Haben Sie so einen *Hashwert* erhalten, können Sie überprüfen, ob diese Version im *Repository* enthalten ist, und können das zugehörige Verzeichnis im *Workspace* wiederherstellen. Falls die Version nicht vorhanden ist, können Sie das *Commit* mit allen referenzierten Objekten aus einem anderen *Repository* importieren (*Pull*).

Folgende Vorteile ergeben sich aus der Verwendung von Hashwerten und der Repository-Struktur:

**Hohe Performance:** Der Zugriff auf Daten über den *Hashwert* geht sehr schnell.

**Redundanzfreie Speicherung:** Identische Dateiinhalte müssen nur einmal abgelegt werden.

**Dezentrale Versionsnummern:** Da sich die *Hashwerte* aus den Inhalten der Dateien, dem Autor und dem Zeitpunkt berechnen, können Versionen auch »offline« erzeugt werden, ohne dass es später zu Konflikten kommt.

**Effizienter Abgleich zwischen Repositorys:** Werden *Commits* von einem *Repository* in ein anderes *Repository* übertragen, müssen nur die noch nicht vorhandenen Objekte kopiert werden. Das Erkennen, ob ein Objekt bereits vorhanden ist, ist dank der *Hashwerte* sehr performant.

**Integrität der Daten:** Der *Hashwert* wird aus dem Inhalt der Daten berechnet. Man kann Git jederzeit prüfen lassen, ob Daten und *Hashwerte* zueinander passen. Unabsichtliche Veränderungen oder böswillige Manipulationen der Daten werden so erkannt.

**Automatische Erkennung von Umbenennungen:** Werden Dateien umbenannt, wird das automatisch erkannt, da sich der *Hashwert* des Inhalts nicht ändert. Es sind somit keine speziellen Befehle zum Umbenennen und Verschieben notwendig.

## 1.3 Branching und Merging – ganz einfach!

Das Verzweigen (Branching) und das Zusammenführen (Merging) sind bei den meisten Versionsverwaltungen Ausnahmesituationen und gehören zu den fortgeschrittenen Themen. Ursprünglich wurde Git für die Entwickler des Linux-Kernels geschaffen, die dezentral über die ganze Welt verteilt arbeiten. Das Zusammenführen der vielen Einzelergebnisse ist dabei eine der größten Herausforderungen. Deshalb ist Git so konzipiert, dass es das Branching und Merging so einfach und sicher wie nur möglich macht.

In Abbildung 1–4 ist dargestellt, wie durch paralleles Arbeiten *Branches* entstehen. Jeder Punkt repräsentiert eine Version (*Commit*) des Projekts. In Git kann immer nur das gesamte Projekt versioniert werden, und somit repräsentiert so ein Punkt die zusammengehörigen Versionen mehrerer Dateien.

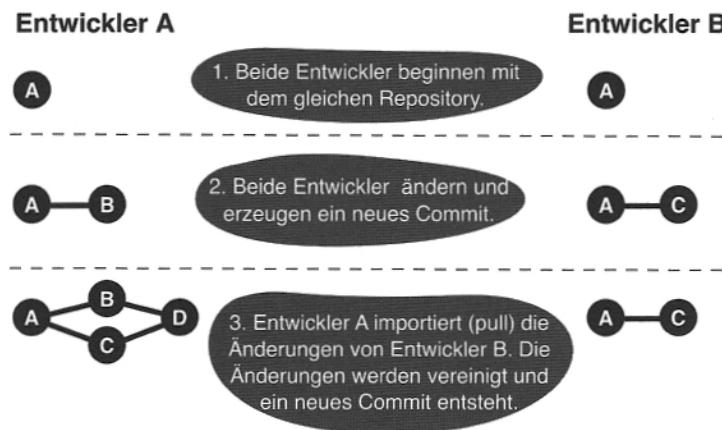
Beide Entwickler beginnen mit derselben Version. Nachdem beide Entwickler Änderungen durchgeführt haben, wird jeweils ein neues *Commit* angelegt. Da beide Entwickler ihr eigenes *Repository* haben, existieren jetzt zwei verschiedene Versionen des Projekts – zwei *Branches* sind entstanden. Wenn ein Entwickler die Änderungen des anderen

**Branches verzweigen**

→ Seite 59

in sein *Repository* importiert, kann er Git die Versionen zusammenführen lassen (*Merge*). Ist dies erfolgreich, so entsteht daraus ein *Merge-Commit*, das beide Änderungen enthält. Wenn der andere Entwickler dieses *Commit* abholt, sind beide wieder auf einem gemeinsamen Stand.

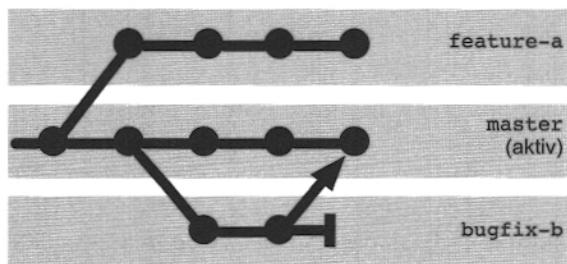
**Abb. 1-4**  
Branches entstehen  
durch paralleles  
Arbeiten.



**Mit Feature-Branches  
entwickeln** → Seite 143

Im vorigen Beispiel ist eine Verzweigung ungeplant entstanden, einfach weil zwei Entwickler parallel an derselben Software gearbeitet haben. Natürlich kann man in Git eine Verzweigung auch gezielt beginnen und einen *Branch* explizit anlegen (Abbildung 1-5). Dies wird häufig genutzt, um die parallele Entwicklung von Features zu koordinieren (*Feature-Branches*).

**Abb. 1-5**  
Explizite Branches für  
unterschiedliche  
Aufgaben



Beim Austausch zwischen *Repositories* (*Pull* und *Push*) kann explizit entschieden werden, welche *Branches* übertragen werden. Neben dem einfachen Verzweigen und Zusammenführen erlaubt Git auch noch folgende Aktionen mit *Branches*:

**Umpflanzen von Branches:** Die *Commits* eines *Branch* können auf einen anderen *Branch* verschoben werden.

*Mit Rebasing die Historie glätten*  
→ Seite 81

**Übertragen einzelner Änderungen:** Einzelne *Commits* können von einem *Branch* auf einen anderen *Branch* kopiert werden, z. B. Bugfixes (was *Cherry-Picking* genannt wird).

*Interaktives Rebasing*  
→ Seite 290

**Historie aufräumen:** Die Historie eines *Branch* kann umgestaltet werden, d. h., es können *Commits* zusammengefasst, umsortiert und gelöscht werden. Dadurch können die Historien besser als Dokumentation der Entwicklung genutzt werden (was man *interaktives Rebasing* nennt).

## 1.4 Zusammenfassung

Nach dem Lesen der letzten Abschnitte sind Sie mit den grundlegenden Konzepten von Git vertraut. Selbst wenn Sie jetzt das Buch aus der Hand legen sollten (was wir nicht hoffen!), können Sie an einer Grundsatzdiskussion über dezentrale Versionsverwaltungen, die Notwendigkeit und Sinnhaftigkeit von Hashwerten sowie über das permanente Branching und Merging in Git teilnehmen.

Vielleicht stellen Sie sich aber auch gerade folgende Fragen:

- Wie soll ich mit diesen allgemeinen Konzepten mein Projekt verwalten?
- Wie koordiniere ich die vielen *Repositorys*?
- Wie viele *Branches* benötige ich?
- Wie integriere ich meinen Build-Server?

*Erste Schritte mit der Kommandozeile*  
→ Seite 9

Um eine Antwort auf die erste Frage zu bekommen, lesen Sie schnell das nächste Kapitel. Dort erfahren Sie konkret, mit welchen Befehlen Sie ein *Repository* anlegen, Dateien versionieren und *Commits* zwischen *Repositorys* austauschen können.

*Workflow-Einführung*  
→ Seite 117

Als Antwort auf die anderen Fragen finden Sie nach den Grundlagenkapiteln detaillierte Workflows.

Falls Sie ein vielbeschäftigter Manager sind und noch nach Gründern suchen, warum Sie Git einsetzen müssen oder auch nicht, dann schauen Sie sich am besten als Nächstes das Kapitel »Die Grenzen von Git« ab Seite 297 an.



## 2 Erste Schritte mit der Kommandozeile

Sie können Git sofort ausprobieren, wenn Sie möchten. Dieses Kapitel beschreibt, wie man das erste Projekt einrichtet. Es zeigt Kommandos zum Versionieren von Änderungen, zum Ansehen der Historie und zum Austausch von Versionen mit anderen Entwicklern.

Falls Sie lieber mit einem grafischen User-Interface starten wollen, finden Sie im nächsten Kapitel eine Anleitung dazu.

*Erste Schritte mit  
SourceTree → Seite 23*

### 2.1 Git einrichten

Zunächst müssen Sie Git installieren. Sie finden alles Nötige hierzu auf der Git-Website:

*<http://git-scm.com/download>*

### 2.2 Ein paar Hinweise für Windows-User

Die Beispiele in diesem Buch wurden mit der Bash-Shell unter Mac OS und Linux entwickelt und getestet. Als die erste Auflage dieses Buchs erschien, gab es bereits eine Version für Windows. Die Integration war aber noch holprig. Die gute Nachricht für Sie: Inzwischen hat Git auch in der Welt von Windows große Verbreitung gefunden, und aktuelle Versionen bieten eine hervorragende Integration, sodass fast alle Beispiele ohne Anpassung auch unter Windows funktionieren. Für die Kommandozeile werden zwei Arten der Integration unterstützt:

- **Eingabeaufforderung (cmd.exe):** Der Git-Befehl git kann von der normalen Windows-Kommandozeile aus aufgerufen werden.
- **Git-Bash:** Git bringt eine Windows-Version der, auf Unix-artigen Systemen weit verbreiteten, Bash-Shell mit. Hier gibt es neben git auch ein paar weitere auf Linux viel genutzte Befehle, wie z. B. grep, find, sort, wc, tail und sed.

## Installation von Git unter Windows

Der Windows-Installer, der von der oben genannten URL geladen werden kann, bietet etliche Optionen. In den meisten Fällen können sie es einfach bei der voreingestellten Auswahl belassen. Folgendes ist empfehlenswert:

- **ADJUSTING YOUR PATH ENVIRONMENT:** Eine gute Wahl ist USE GIT FROM THE WINDOWS COMMAND PROMPT, denn Sie können dann Git nicht nur in der Git-Bash, sondern auch in der Windows-Eingabeaufforderung nutzen, ohne dass Befehle der Windows-Kommandozeile verändert werden.
- **CONFIGURING THE LINE ENDING CONVERSIONS:** Windows nutzt andere Zeichen zur Markierung von Zeilenenden als Linux. Git kann Zeilenenden automatisch konvertieren. Das ist nützlich, wenn Entwickler mit unterschiedlichen Betriebssystemen am selben Projekt arbeiten. Für den Einstieg ist CHECKOUT AS-IS, COMMIT AS-IS am einfachsten.
- **CONFIGURING THE TERMINAL EMULATOR TO USE WITH GIT BASH:** Empfehlenswert ist USE MINTTY, weil das Eingabefenster für die Git-Bash dann etwas mehr Komfort bietet.

## Arbeiten mit der Windows-Eingabeaufforderung

Sie können den git-Befehl in der Eingabeaufforderung (cmd.exe) nutzen. Alles andere, wie z.B. Navigation, Verzeichnisanlage, Dateioperationen etc., machen Sie wie gewohnt. In der Ausgabe zeigt Git in Pfadnamen immer »/« als Trenner. Als Parameter dürfen Pfadnamen wahlweise mit »/« oder »\« angegeben werden. Letzteres ist empfehlenswert, weil dann die Autovervollständigung für Dateipfade mit der Tab-Taste funktioniert.

Die Beispiele aus diesem Kapitel und auch aus den meisten Einstiegskapiteln können direkt in der Windows-Eingabeaufforderung nachvollzogen werden. In späteren Kapiteln werden vereinzelt Features der Bash-Shell und Linux-Befehle genutzt, die es in der Windows-Eingabeaufforderung nicht gibt. Deshalb empfehlen wir, gleich mit der Git-Bash zu beginnen.

## Arbeiten mit der Git-Bash unter Windows

In der Git-Bash ist eine Tab-Vervollständigung nicht nur für Dateipfade sondern auch für git-Befehle und -Optionen eingerichtet. Drückt man einmal Tab, dann wird versucht, das begonnene Kommando zu vervollständigen. Für Einsteiger noch wichtiger: Drückt man zweimal Tab, werden mögliche Vervollständigungen angezeigt:

```
> git com<TAB>
> git commit

> git c<TAB><TAB>
checkout      cherry      cherry-pick  citool
clean        clone       commit       config

> git commit --a<TAB><TAB>
--all      --amend    --author=
```

*Tipp:*  
Autovervollständigung

Achtung! In der Git-Bash müssen Sie »/« als Pfadtrener nutzen! Die »:«-Notation für Laufwerke ist nicht zulässig. Man ersetzt z. B. G:\test durch /g/test!

Von den Befehlen in der Bash braucht man für den Anfang nicht viele. cd zur Navigation zwischen Verzeichnissen und mkdir zum Anlegen von Verzeichnissen funktionieren ganz ähnlich wie unter Windows. Statt dir nutzt man ls oder ll, um ein Inhaltsverzeichnis zu sehen.

## 2.3 Git einrichten

Git ist in hohem Maße konfigurierbar. Für den Anfang genügt es aber, wenn Sie Ihren Benutzernamen und Ihre E-Mail-Adresse mit dem config-Befehl eintragen:

```
> git config --global user.name hmustermann
> git config --global user.email "hans@mustermann.de"
```

Nicht notwendig, aber empfehlenswert, ist es, Ihren Lieblingstexteditor zu registrieren. Dieser wird immer dann aufgerufen, wenn Git eine Texteingabe benötigt z. B. für einen Commit-Kommentar:

```
> git config --global core.editor vim          # VI improved
> git config --global core.editor "atom --wait" # Atom editor
> git config --global core.editor notepad     # Windows notepad
```

## 2.4 Das erste Projekt mit Git

Am besten ist es, wenn Sie ein eigenes kleines Projekt verwenden, um Git zu erproben. Unser Beispiel namens `erste-schritte` kommt mit zwei Textdateien aus:

**Abb. 2-1**  
Unser Beispielprojekt



*Tipp: Sicherungskopie nicht vergessen!*

Erstellen Sie eine Sicherungskopie, bevor Sie das Beispiel mit Ihrem Lieblingsprojekt durchspielen! Es ist gar nicht so leicht, in Git etwas endgültig zu löschen oder »kaputtzumachen«, und Git warnt meist deutlich, wenn Sie dabei sind, etwas »Gefährliches« zu tun. Trotzdem: Vorsicht bleibt die Mutter der Porzellankiste.

### Projektverzeichnis

Die Beispiele nutzen ein Top-Level-Verzeichnis `/projekte` zur Ablage der Projekte. Dadurch bleiben die Pfadnamen auch dort kurz, wo absolute Pfade angegeben sind. Wahrscheinlich werden Sie Ihre Projekte an anderer Stelle einrichten wollen, z. B. unter

`/home/hmustermann/projekte`

oder

`C:\Users\hmustermann\projekte`

**Achtung!** Denken Sie also daran, `/projekte` in den Beispielen durch Ihr Verzeichnis zu ersetzen! Windows-User müssen in der Git-Bash »umslashen«, z. B. zu

`/c/Users/hmustermann/projekte`

### Repository anlegen

Als Erstes wird das *Repository* angelegt, in dem die Historie des Projekts gespeichert werden soll. Dies erledigt der `init`-Befehl im Projektverzeichnis. Ein Projektverzeichnis mit einem *Repository* nennt man einen *Workspace*.

```
> cd /projekte/erste-schritte
> git init
Initialized empty Git repository in /projekte/erste-schritte/.git/
```

Git hat im Verzeichnis `/projekte/erste-schritte` ein *Repository* angelegt, aber noch keine Dateien hinzugefügt. Achtung! Das *Repository* liegt in einem verborgenen Verzeichnis namens `.git` und wird im Explorer (bzw. Finder) unter Umständen nicht angezeigt.



**Abb. 2-2**  
Das  
Repository-Verzeichnis

## Das erste Commit

Als Nächstes können Sie die Dateien `foo.txt` und `bar.txt` ins *Repository* bringen. Eine Projektversion nennt man bei Git ein *Commit*, und sie wird in zwei Schritten angelegt. Als Erstes bestimmt man mit dem `add`-Befehl, welche Dateien in das nächste *Commit* aufgenommen werden sollen. Danach überträgt der `commit`-Befehl die Änderungen ins *Repository* und vergibt einen sogenannten *Commit-Hash* (hier `2f43cd0`), der das neue *Commit* identifiziert.

```
> git add foo.txt bar.txt
> git commit --message "Beispielprojekt importiert."
master (root-commit) 2f43cd0] Beispielprojekt importiert.
 2 files changed, 2 insertions(+), 0 deletions(-)
  create mode 100644 bar.txt
  create mode 100644 foo.txt
```

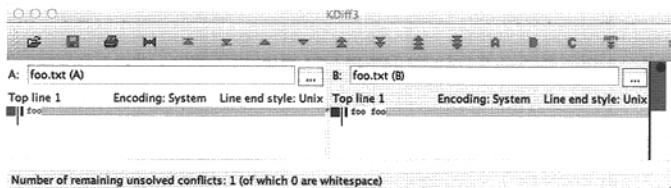
## Status abfragen

Jetzt ändern Sie `foo.txt`, löschen `bar.txt` und fügen eine neue Datei `bar.html` hinzu. Der `status`-Befehl zeigt alle Änderungen seit dem letzten *Commit* an. Die neue Datei `bar.html` wird übrigens als *untracked* angezeigt, weil sie noch nicht mit dem `add`-Befehl angemeldet wurde.

```
> git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
#        working directory)
#
#       deleted:    bar.txt
#       modified:   foo.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       bar.html
no changes added to commit (use "git add" and/or "git commit -a")
```

Wenn Sie mehr Details wissen wollen, zeigt Ihnen der `diff`-Befehl jede geänderte Zeile an. Die Ausgabe im `diff`-Format empfinden viele Menschen als schlecht lesbar, sie kann dafür aber gut maschinell verarbeitet werden. Es gibt glücklicherweise eine ganze Reihe von Tools und Entwicklungsumgebungen, die Änderungen übersichtlicher darstellen können (Abbildung 2–3).

**Abb. 2–3**  
*Diff-Darstellung in grafischem Tool (kdiff3)*



```
> git diff foo.txt
diff --git a/foo.txt b/foo.txt
index 1910281..090387f 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1 +1 @@
-foo
\ No newline at end of file
+foo foo
\ No newline at end of file
```

## Ein Commit nach Änderungen

Änderungen fließen nicht automatisch ins nächste *Commit* ein. Egal ob eine Datei bearbeitet, hinzugefügt oder gelöscht<sup>1</sup> wurde, mit dem `add`-Befehl bestimmt man, dass die Änderung übernommen werden soll.

<sup>1</sup> Es klingt paradox, `git add` für eine gelöschte Datei aufzurufen. Gemeint ist damit, dass der `add`-Befehl die Löschung für das nächste *Commit* vormerkt.

```
> git add foo.txt bar.html bar.txt
```

Ein weiterer Aufruf des `status`-Befehls zeigt, was in den nächsten *Commit* aufgenommen wird:

```
> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   bar.html
#       deleted:    bar.txt
#       modified:   foo.txt
#
```

Mit dem `commit`-Befehl werden genau diese Änderungen übernommen:

```
> git commit --message "Einiges geändert."
[master 7ac0f38] Einiges geändert.
 3 files changed, 2 insertions(+), 2 deletions(-)
 create mode 100644 bar.html
 delete mode 100644 bar.txt
```

## Historie betrachten

Der `log`-Befehl zeigt die Historie des Projekts. Die *Commits* sind chronologisch absteigend sortiert.

```
> git log
commit 7ac0f38f575a60940ec93c98de11966d784e9e4f
Author: Rene Preisel <rp@eToSquare.de>
Date:   Thu Dec 2 09:52:25 2010 +0100

  Einiges geändert.

commit 2f43cd047baadc1b52a8367b7cad2cb63bca05b7
Author: Rene Preisel <rp@eToSquare.de>
Date:   Thu Dec 2 09:44:24 2010 +0100

  Beispielprojekt importiert.
```

## 2.5 Zusammenarbeit mit Git

Sie haben jetzt einen *Workspace* mit Projektdateien und ein *Repository* mit der Historie des Projekts. Bei einer klassischen zentralen Versionsverwaltung (etwa CVS<sup>2</sup> oder Subversion<sup>3</sup>) hat jeder Entwickler einen

<sup>2</sup> <http://www.nongnu.org/cvs/>

<sup>3</sup> <http://subversion.apache.org/>

eigenen *Workspace*, aber alle Entwickler teilen sich ein gemeinsames *Repository*. In Git hat jeder Entwickler einen eigenen *Workspace* mit einem eigenen Repository, also eine vollwertige Versionsverwaltung, die nicht auf einen zentralen Server angewiesen ist. Entwickler, die gemeinsam an einem Projekt arbeiten, können *Commits* zwischen ihren *Repositories* austauschen. Um dies auszuprobieren, legen Sie einen zusätzlichen *Workspace* an, in dem Aktivitäten eines zweiten Entwicklers simuliert werden.

## Repository klonen

Der zusätzliche Entwickler braucht eine eigene Kopie (genannt *Klon*) des *Repositorys*. Sie beinhaltet alle Informationen, die das Original auch besitzt, d. h., die gesamte Projekthistorie wird mitkopiert. Dafür gibt es den `clone`-Befehl:

```
> git clone /projekte/erste-schritte  
                  /projekte/erste-schritte-klon  
Cloning into erste-schritte-klon...  
done.
```

Die Projektstruktur sieht nun so wie in Abbildung 3–6 auf Seite 27 aus.

## Änderungen aus einem anderen Repository holen

Ändern Sie die Datei `erste-schritte/foo.txt`.

```
> cd /projekte/erste-schritte  
> git add foo.txt  
> git commit --message "Eine Änderung im Original."
```

Das neue *Commit* ist jetzt im ursprünglichen *Repository* `erste-schritte` enthalten, es fehlt aber noch im *Klon* `erste-schritte-klon`. Zum besseren Verständnis zeigen wir hier noch das Log für `erste-schritte`:

```
> git log --oneline  
a662055 Eine Änderung im Original.  
7ac0f38 Einiges geändert.  
2f43cd0 Beispielprojekt importiert.
```

Ändern Sie im nächsten Schritt die Datei `erste-schritte-klon/bar.html` im *Klon-Repository*:

```
> cd /projekte/erste-schritte-klon  
> git add bar.html  
> git commit --message "Eine Änderung im Klon."
```



**Abb. 2-4**  
Das Beispielprojekt und sein Klon

```

> git log --oneline
1fcc06a Eine Änderung im Klon.
7ac0f38 Einiges geändert.
2f43cd0 Beispielprojekt importiert.
  
```

Sie haben jetzt in jedem der beiden *Repositorys* zwei gemeinsame *Commits* und jeweils ein neues *Commit*. Als Nächstes soll das neue *Commit* aus dem Original in den Klon übertragen werden. Dafür gibt es den `pull`-Befehl. Beim Klonen ist der Pfad zum *Original-Repository* im Klon hinterlegt worden. Der `pull`-Befehl weiß also, wo er neue *Commits* abholen soll.

```

> cd /projekte/erste-schritte-klon
> git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /projekte/erste-schritte
  7ac0f38..a662055 master      -> origin/master
Merge made by recursive.
 foo.txt |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
  
```

Der `pull`-Befehl hat die neuen Änderungen aus dem Original abgeholt, mit den lokalen Änderungen im Klon verglichen und beide Änderungen im *Workspace* zusammengeführt und ein neues *Commit* daraus erstellt. Man nennt dies einen *Merge*.

**Branches zusammenführen**  
→ Seite 67

Achtung! Gelegentlich kommt es beim *Merge* zu Konflikten. Dann kann Git die Versionen nicht automatisch zusammenführen. Dann müssen Sie die Dateien zunächst manuell bereinigen und die Änderungen danach mit einem *Commit* bestätigen.

Ein erneuter `log`-Befehl zeigt das Ergebnis der Zusammenführung nach dem `pull` an. Diesmal nutzen wir eine grafische Variante des Logs.

```
> git log --graph
*   9e7d7b9 Merge branch 'master' of /projekte/erste-schritte
|\ 
| * a662055 Eine Änderung im Original.
* | 1fcc06a Eine Änderung im Klon.
|/
* 7ac0f38 Einiges geändert.
* 2f43cd0 Beispielprojekt importiert.
```

Die Historie ist nun nicht mehr linear. Im Graphen sehen Sie sehr schön die parallele Entwicklung (mittlere *Commits*) und das anschließende *Merge-Commit*, mit dem die *Branches* wieder zusammengeführt wurden (oben).

## Änderungen aus beliebigen Repositorys abholen

Der `pull`-Befehl ohne Parameter funktioniert nur in geklonten *Repositorys*, da diese eine Verknüpfung zum originalen *Repository* haben. Beim `pull`-Befehl kann man den Pfad zu einem beliebigen *Repository* angeben. Als weiterer Parameter kann der *Branch* (Entwicklungszweig) angegeben werden, von dem Änderungen geholt werden. In unserem Beispiel gibt es nur den *Branch* `master`, der als Default von Git automatisch angelegt wird.

```
> cd /projekte/erste-schritte
> git pull /projekte/erste-schritte-klon master
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /projekte/erste-schritte-klon
 * branch           master      -> FETCH_HEAD
Updating a662055..9e7d7b9
Fast-forward
 bar.html |  2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

## Ein Repository für den Austausch erstellen



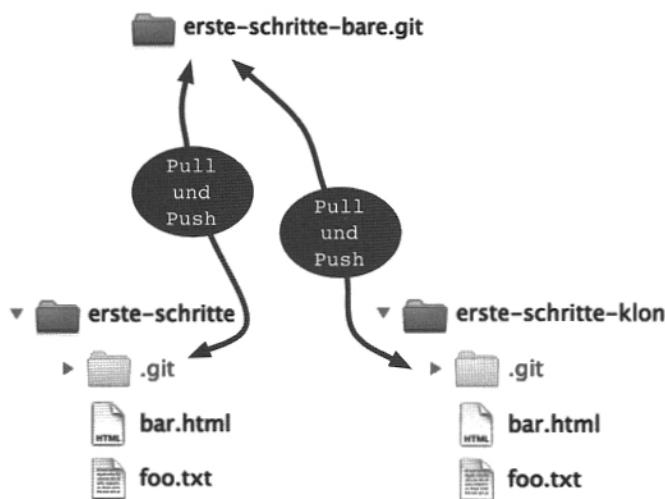
**Abb. 2–5**  
»Bare-Repository«:  
ein Repository ohne  
Workspace

Neben dem `pull`-Befehl, der *Commits* von einem anderen *Repository* holt, gibt es auch einen `push`-Befehl, der *Commits* in ein anderes *Repository* überträgt. Der `push`-Befehl sollte allerdings nur auf *Repositories* angewendet werden, auf denen gerade kein Entwickler arbeitet. Am besten erzeugt man sich dazu ein *Repository* ohne einen *Workspace*, der es umgibt. Ein solches *Repository* wird als *Bare-Repository* bezeichnet. Es wird durch die Option `--bare` des `clone`-Befehls erzeugt. Man kann es als zentrale Anlaufstelle verwenden. Entwickler übertragen ihre *Commits* (mit dem `push`-Befehl) dorthin und holen sich mit dem `pull`-Befehl die *Commits* der anderen Entwickler dort ab. Man verwendet die Endung `.git`, um ein *Bare-Repository* zu kennzeichnen. Das Ergebnis sehen Sie in Abbildung 2–5.

```
> git clone --bare /projekte/erste-schritte
                  /projekte/erste-schritte-bare.git
Cloning into bare repository erste-schritte-bare.git...
done.
```

## Änderungen mit Push hochladen

**Abb. 2-6**  
Austausch über ein gemeinsames Repository



Zur Demonstration des push-Befehls ändern Sie noch mal die Datei `erste-schritte/foo.txt` und erstellen ein neues *Commit*:

```
> cd /projekte/erste-schritte
> git add foo.txt
> git commit --message "Weitere Änderung im Original."
```

Dieses *Commit* übertragen Sie dann mit dem push-Befehl in das zentrale *Repository* (Abbildung 2-6). Dieser Befehl erwartet dieselben Parameter wie der pull-Befehl: den Pfad zum *Repository* und den zu benutzenden *Branch*.

```
> git push /projekte/erste-schritte-bare.git master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 293 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /projekte/erste-schritte-bare.git/
  9e7d7b9..7e7e589  master -> master
```

## Pull: Änderungen abholen

Um die Änderungen auch in das *Klon-Repository* zu holen, nutzen wir wieder den `pull`-Befehl mit dem Pfad zum zentralen *Repository*.

```
> cd /projekte/erste-schritte-klon
> git pull /projekte/erste-schritte-bare.git master

remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ../erste-schritte-bare
 * branch           master      -> FETCH_HEAD
Updating 9e7d7b9..7e7e589
Fast-forward
 foo.txt |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

**Achtung!** Hat ein anderer Entwickler vor uns ein `push` ausgeführt, verweigert der `push`-Befehl die Übertragung. Die neuen Änderungen müssen dann zuerst mit `pull` abgeholt und lokal zusammengeführt werden.

**Push verweigert!**  
**Was tun?** → Seite 103

## 2.6 Zusammenfassung

**Workspace und Repository:** Ein *Workspace* ist ein Verzeichnis, das ein *Repository* in einem Unterverzeichnis `.git` enthält. Mit dem `init`-Befehl legt man ein *Repository* im aktuellen Verzeichnis an.

**Commit:** Ein Commit definiert einen Versionsstand für alle Dateien des *Repositorys* und beschreibt, wann, wo und von wem dieser Stand erstellt wurde. Mit dem `add`-Befehl bestimmt man, welche Dateien ins nächste *Commit* aufgenommen werden. Der `commit`-Befehl erstellt ein neues *Commit*.

**Informationen abrufen:** Der `status`-Befehl zeigt, welche Dateien lokal verändert wurden und welche Änderungen ins nächste *Commit* aufgenommen werden. Der `log`-Befehl zeigt die Historie der *Commits*. Mit dem `diff`-Befehl kann man sich die Änderungen bis auf die einzelne Zeile heruntergebrochen anzeigen lassen.

**Klonen:** Der `clone`-Befehl erstellt eine Kopie eines *Repositorys*, die Klon genannt wird. In der Regel hat jeder Entwickler einen vollwertigen Klon des Projekt-*Repositorys* mit der ganzen Projekthistorie in seinem *Workspace*. Mit diesem Klon kann er autark ohne Verbindung zu einem Server arbeiten.

**Push und Pull:** Mit den Befehlen `push` und `pull` werden *Commits* zwischen lokalen und entfernten *Repositorys* ausgetauscht.



## 3 Erste Schritte mit SourceTree

Es gibt zahlreiche User-Interfaces für Git: eigenständige Anwendungen, Plug-ins für Entwicklungsumgebungen und Dienste im Web. Eines davon stellen wir hier exemplarisch vor. Wir haben dazu *Atlassian SourceTree* gewählt, weil es eine hohe Verbreitung hat und weil wir selbst gern damit arbeiten. Sie finden es hier:

**Grafische Werkzeuge  
für Git → Seite 292**

<https://www.atlassian.com/software/sourcetree/overview>

**Anmerkung:** *SourceTree* ist kostenlos, aber nicht Open Source. Es ist für Windows und Mac OS erhältlich. Es gibt leider keine Linux-Version. Die Screenshots haben wir unter Mac OS aufgenommen.

### 3.1 SourceTree konfigurieren

Auch *SourceTree* ist in hohem Maße konfigurierbar. Für den Anfang genügt es aber, wenn Sie Ihren Benutzernamen und Ihre E-Mail-Adresse im Menü SOURCETREE / EINSTELLUNGEN / ALLGEMEIN eintragen.

Falls Sie Git auch für die Kommandozeile installiert haben, empfehlen wir, folgende Einstellungen durchzuführen, um widersprüchliche Konfigurationen zu vermeiden:

- Unter SOURCETREE / EINSTELLUNGEN / GIT klicken Sie auf den Button BENUTZE SYSTEM GIT.
- Unter SOURCETREE / EINSTELLUNGEN / ALLGEMEIN aktivieren Sie die Option SOURCETREE ERLAUBEN, ÄNDERUNGEN AN DER GLOBALEN GIT- UND MERCURIAL-KONFIGURATION DURCHZUFÜHREN.

### 3.2 Das erste Projekt mit Git

Am besten ist es, wenn Sie ein eigenes Projekt verwenden, um Git zu erproben. Beginnen Sie mit einem einfachen kleinen Projekt. Unser Beispiel zeigt ein winziges Projekt namens `erste-schritte` mit zwei Textdateien.

**Abb. 3-1**  
Unser Beispielprojekt



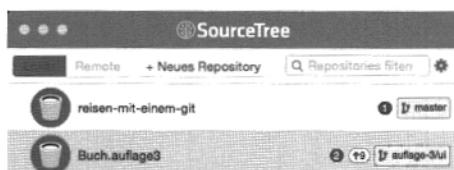
*Tipp: Sicherungskopie nicht vergessen!*

Erstellen Sie eine Sicherungskopie, bevor Sie das Beispiel mit Ihrem Lieblingsprojekt durchspielen! Es ist gar nicht so leicht, in Git etwas endgültig zu löschen oder »kaputtzumachen«, und Git warnt meist deutlich, wenn Sie dabei sind, etwas »Gefährliches« zu tun. Trotzdem: Vorsicht bleibt die Mutter der Porzellankiste.

## Repository anlegen

Als Erstes wird das *Repository* angelegt, in dem die Historie des Projekts gespeichert werden soll. Das Repository legen Sie mit + NEUES REPOSITORY gefolgt von LOKALES REPOSITORY ERSTELLEN an. Im Feld ZIELPFAD geben Sie an, wo Ihre Projektdateien liegen. Ein Projektverzeichnis mit einem *Repository* nennt man einen *Workspace*.

**Abb. 3-2**  
Repository-Übersicht von SourceTree



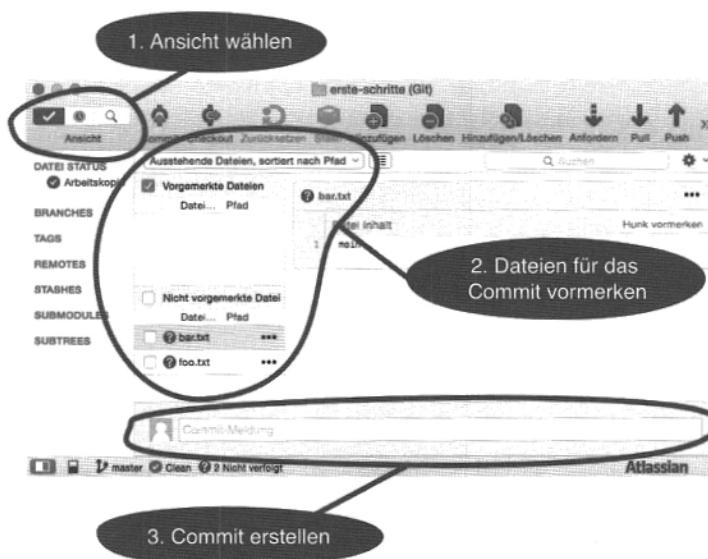
Von der Repository-Übersicht aus (Abbildung 3-2) können Sie per Doppelklick ein Fenster für das jeweilige Repository öffnen.

## Das erste Commit

Als Nächstes können Sie die Dateien *foo.txt* und *bar.txt* ins *Repository* bringen. Eine Projektversion nennt man in Git ein *Commit* und wird mit SourceTree in drei Schritten angelegt (Abbildung 3-2):

1. Falls Sie nicht schon dort sind: Wechseln Sie in die Ansicht DATEI-STATUS.
2. Im Abschnitt NICHT VORGEMERKTE DATEIEN setzen Sie ein Häkchen neben jene Dateien, die Sie mit dem nächsten Commit versionieren möchten. Die Dateien wechseln dann nach oben in den Abschnitt VORGEMERKTE DATEIEN.

3. Im Feld COMMIT-MELDUNG geben Sie einen Text ein und bestätigen ihn mit dem COMMIT-Button.



**Abb. 3–3**  
Commits erstellen

## Dateistatus ansehen

Jetzt ändern Sie `foo.txt`, löschen `bar.txt` und fügen eine neue Datei `bar.html` hinzu. Die Ansicht DATEI STATUS (Abbildung 3–3) aktualisiert sich von alleine, sobald Sie Ihre Änderungen speichern. Für jede neue, geänderte oder gelöschte Datei erscheint ein Eintrag unter NICHT VORGEMERKTE DATEIEN. Selektieren Sie einen oder mehrere dieser Einträge (ohne Häkchen zu setzen), dann zeigt der rechte Bereich die Änderungen im Detail an.

## Ein Commit nach Änderungen

Um Änderungen zu übernehmen, setzen Sie Häkchen an den Einträgen unter NICHT VORGEMERKTE DATEIEN, tragen einen Commit-Kommentar ein und bestätigen ihn mit dem COMMIT-Button.

Achtung! Änderungen müssen in Git immer vorgemerkt werden (*Staging*). Wenn Sie eine Datei ändern, nachdem Sie das Vormerk-Häkchen gesetzt haben, müssen Sie das Häkchen erneut setzen, damit auch die neuen Änderungen übernommen werden.

Wenn Sie das Häkchen im Kopf von NICHT VORGEMERKTE DATEIEN setzen, werden alle angezeigten Änderungen auf einmal vor-  
gemerkt.

**Commits zusammenstellen**  
→ Seite 39  
**Tipp:** Alle Änderungen übernehmen

## Historie betrachten

Die Ansicht Log (Abbildung 3–4) zeigt die Commits. Wenn Sie in der Liste oben einen Eintrag selektieren, zeigt der untere Bereich die Änderungen dieses Commits (bezogen auf dessen ersten Vorgänger) an. Sie können auch zwei Commits auswählen, dann werden die Unterschiede zwischen diesen angezeigt.

**Abb. 3–4**  
Commit-Log-Ansicht



## 3.3 Zusammenarbeit mit Git

Sie haben jetzt einen *Workspace* mit Projektdateien und ein *Repository* mit der Historie des Projekts. Bei einer klassischen zentralen Versionsverwaltung (etwa CVS<sup>1</sup> oder Subversion<sup>2</sup>) hat jeder Entwickler einen eigenen *Workspace*, aber alle Entwickler teilen sich ein gemeinsames *Repository*. In Git hat jeder Entwickler einen eigenen *Workspace mit einem eigenen Repository*, also eine vollwertige Versionsverwaltung, die nicht auf einen zentralen Server angewiesen ist. Entwickler, die gemeinsam an einem Projekt arbeiten, können *Commits* zwischen ihren *Repositories* austauschen.

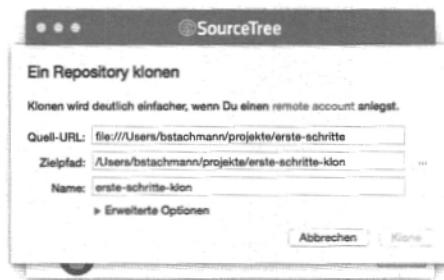
### Repository klonen

Jeder Entwickler, der neu zu einem Projekt hinzukommt, braucht eine eigene Kopie (genannt *Klon*) des *Repositorys*. Sie beinhaltet alle Informationen, die das Original auch besitzt, d. h., die gesamte Projekthistorie wird mitkopiert.

<sup>1</sup> <http://www.nongnu.org/cvs/>

<sup>2</sup> <http://subversion.apache.org/>

In der Repository-Übersicht (Abbildung 3–2) klicken Sie + NEUES REPOSITORY, gefolgt von VON URL KLONEN. Im Feld QUELL-URL geben Sie an, von wo geklont wird; meistens ist das eine https-URL von dem Server, auf dem das *Repository* liegt. Da unser Beispiel-*Repository* lokal erstellt wurde, geben wir eine file-URL ein. Der Zielpfad kann frei gewählt werden (Abbildung 3–5). Für das Beispiel legen Sie den Klon am besten neben dem Original ab.



**Abb. 3–5**  
Repository klonen

Die Projektstruktur sieht nun so wie in Abbildung 3–6 aus.



**Abb. 3–6**  
Das Beispielprojekt und sein Klon

## Änderungen aus einem anderen Repository holen

Jetzt sollten zwei *Repository*-Fenster in SourceTree geöffnet sein: ERSTE-SCHRITTE und ERSTE-SCHRITTE-KLON. Über das Menü FENSTER können Sie hin- und herwechseln.

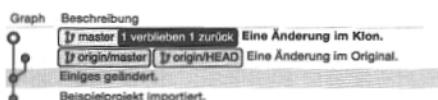
**Tipp:** Dateien bearbeiten

Wenn Sie in der Ansicht DATEI STATUS den Filter ALLE DATEIEN (statt AUSSTEHEND) wählen, werden auch unveränderte Dateien angezeigt. Über das Kontextmenü können Sie die Dateien dann öffnen.

Beginnen Sie im Fenster ERSTE-SCHRITTE. Ändern Sie die Datei `foo.txt` im Original-Repository und führen Sie dort ein Commit durch.

Wechseln Sie jetzt in das Fenster ERSTE-SCHRITTE-KLON, um die Änderungen abzuholen. Klicken Sie auf den Button ANFORDERN.

**Abb. 3-7**  
Daten wurden geholt.



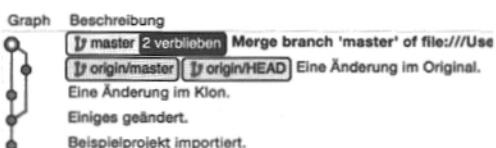
SourceTree zeigt an, dass frische Commits zum Integrieren bereitstehen.

**Abb. 3-8**  
Anzeige neuer Commits



Mit dem PULL-Button können Sie die Änderungen integrieren. Im Dialog, der dann erscheint, klicken Sie auf OK.

**Abb. 3-9**  
Die Änderungen wurden integriert.



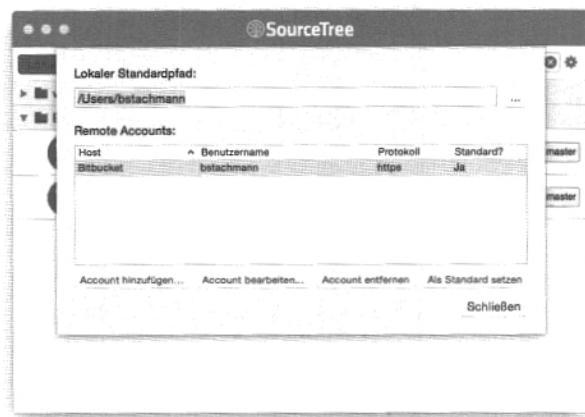
Die Aktion PULL hat die neuen Änderungen aus dem Original abgeholt, mit den lokalen Änderungen im Klon verglichen und beide Änderungen im *Workspace* zusammengeführt und ein neues *Commit* daraus erstellt. Man nennt dies einen *Merge*. Die Historie ist nun nicht mehr linear. Im Graphen sehen Sie sehr schön die parallele Entwicklung (mittlere *Commits*) und das anschließende *Merge-Commit*, mit dem die *Branches* wieder zusammengeführt wurden (oben).

**Branches zusammenführen**  
→ Seite 67

Achtung! Gelegentlich kommt es beim *Merge* zu Konflikten. Dann kann Git die Versionen nicht automatisch zusammenführen. In so einem Fall müssen Sie die Dateien zunächst manuell bereinigen und die Änderungen danach mit einem *Commit* bestätigen.

## Repository für den Austausch über einen Server erstellen

Zum Austausch zwischen den Entwicklern nutzt man in der Regel ein Repository auf einem Server. Um das auszuprobieren, können Sie einen Account bei Bitbucket<sup>3</sup> oder GitHub<sup>4</sup> anlegen. In der Repository-Übersicht (Abbildung 3–2) sehen Sie rechts oben einen Button mit einem Zahnrad. Dort können Sie unter EINSTELLUNGEN Ihren Account eintragen.



**Abb. 3–10**  
Einstellung in der  
Repository-Übersicht

Sobald Sie den Account eingetragen haben, können Sie im Kontextmenü von ERSTE-SCHRITTE mit PUBLISH TO REMOTE ... das Beispiel-Repository veröffentlichen. Nennen Sie das Repository ERSTE-SCHRITTE-SHARED.

## Änderungen mit push hochladen

Zur Demonstration der Aktion PUSH ändern Sie noch mal in ERSTE-SCHRITTE die Datei foo.txt und erstellen ein neues Commit.

Dieses Commit übertragen Sie dann mit dem PUSH-Button in das zentrale Repository.

## Repository verknüpfen

Öffnen Sie nun wieder das Fenster für ERSTE-SCHRITTE-KLON und wählen Sie im Menü REPOSITORY den Punkt REMOTE HINZUFÜGEN ... Es erscheint ein Dialog, in dem Sie URL, Benutzername etc. eingeben können. In diesem Fall ist das aber nicht nötig. Rechts neben dem

<sup>3</sup> <http://www.bitbucket.com>

<sup>4</sup> <http://www.github.com>

Feld URL/PFAD ist ein Button mit einer Weltkugel. Dieser zeigt eine Auswahl bereits bekannter Repositorys an. Das eben erzeugte ERSTE-SCHRITTE-SHARED ist natürlich dabei. Wenn Sie es auswählen und ORIGIN nennen, können Sie danach mit dem PULL-Button Commits aus dem gemeinsamen Repository abholen.

### Pull: Änderungen abholen

Jetzt können Sie mit dem PULL-Button die Änderungen vom Server abholen.

*Push verweigert!*  
Was tun? → Seite 103

Achtung! Hat ein anderer Entwickler vor Ihnen die Aktion *Push* ausgeführt, verweigert Git die Übertragung. Die neuen Änderungen müssen dann zuerst mit der Aktion PULL abgeholt und lokal zusammengeführt werden.

## 3.4 Zusammenfassung

**Workspace und Repository:** Ein *Workspace* ist ein Verzeichnis, das ein *Repository* in einem Unterverzeichnis `.git` enthält. Mit + NEUES REPOSITORY, gefolgt von LOKALES REPOSITORY ERSTELLEN im Repository-Übersichtsfenster, erstellen Sie ein Repository mit Workspace.

**Commit:** Ein Commit definiert einen Versionsstand für alle Dateien des Repositorys und beschreibt, wann, wo und von wem dieser Stand erstellt wurde. Durch das Setzen von Häkchen in der Ansicht DATEI-STATUS bestimmt man, welche Dateien aufgenommen werden. Durch Eingabe eines Kommentars und Bestätigung mit dem COMMIT-Button erstellt man ein Commit.

**Informationen abrufen:** Die Ansicht DATEI STATUS zeigt, welche Dateien verändert wurden. Die Ansicht LOG zeigt die Historie der Commits. Selektiert man dort ein Commit, werden die Details der Änderung angezeigt.

**Klonen:** Mit + NEUES REPOSITORY, gefolgt von VON URL KLONEN im Repository-Übersichtsfenster, erstellt man einen Klon eines anderen Repositorys, dessen URL man kennt. In der Regel hat jeder Entwickler einen vollwertigen Klon des Projekt-Repositorys mit der ganzen Projekthistorie in seinem *Workspace*. Mit diesem Klon kann er autark ohne Verbindung zu einem Server arbeiten.

**Push und Pull:** Mit den Aktionen PUSH und PULL werden Commits zwischen lokalen und entfernten Repositorys ausgetauscht.

## 4 Was sind Commits?

Der wichtigste Begriff in Git ist das *Commit*. Git verwaltet Versionen von Software, und jede Version wird als Commit in einem *Repository* abgelegt. Ein Commit umfasst dabei immer das ganze Projekt. Mit einem Commit wird für jede von Git verwaltete Datei im Projekt eine Kopie im Repository gespeichert.

Abbildung 4–1 zeigt eine Zusammenfassung<sup>1</sup> wichtiger Informationen zu einem Commit.

```
commit 9acc5d5efec1d2d62f7e98bcc3880cda762cb831
Author: Bjørn Stachmann <bstachmann@yahoo.de>
Date:   Sat Dec 18 18:20:45 2010 +0100
```

**Abb. 4–1**  
Informationen zu  
einem Commit

Abschnitt darüber, was Commits sind.

```
buch/commits/commits.tex | 28 ++++++-----+
1 files changed, 25 insertions(+), 3 deletions(-)
```

Als Erstes sieht man den sogenannten Commit-Hash 9acc5d5e...cb831. Dann folgen Informationen zu Autor und Commit-Zeitpunkt und ein Kommentar. Abschließend zeigt eine Zusammenfassung, welche Dateien sich gegenüber der vorigen Version verändert haben. Was die Zusammenfassung nicht zeigt: Auch dieses Commit enthält nicht nur die geänderte Datei commits.tex, sondern alle Dateien des Projekts. Für jedes Commit berechnet Git einen 40 Zeichen langen eindeutigen Code, den sogenannten *Commit-Hash*. Kennt man diesen Commit-Hash, kann man die Dateien des Projekts aus dem Repository so wiederherstellen, wie sie zum Zeitpunkt des Commits festgehalten wurden. Das Wiederherstellen einer Version bezeichnet man in Git als *Checkout*.

---

<sup>1</sup>Die Zusammenfassung wurde mit git log -stat -1 erstellt.

## 4.1 Zugriffsberechtigungen und Zeitstempel

Git speichert die Zugriffsberechtigungen (POSIX File Permissions: Read, Write, Execute) für jede Datei, nicht aber den Änderungszeitstempel. Beim Checkout wird der Änderungszeitstempel auf die aktuelle Uhrzeit gesetzt.

*Warum werden Änderungszeitstempel nicht gespeichert?*

**Anmerkung:** Der Grund dafür ist, dass viele Build-Tools den Änderungszeitstempel als Auslöser für das erneute Bauen von Dateien nutzen: Ist die letzte Änderung jünger als das letzte Build-Ergebnis, muss neu gebaut werden. Da Git beim *Checkout* den Änderungszeitstempel immer auf die aktuelle Zeit setzt, lösen auch Wechsel auf ältere Versionen den Build-Vorgang korrekt aus.

## 4.2 Die Befehle add und commit

Mit den beiden Befehlen `add` und `commit` erstellt man Commits.

*Mit `.gitignore` Dateien unversioniert lassen → Seite 46*

Schritt für Schritt

### Alle Änderungen in einem Commit übernehmen

*Erstelle ein Commit mit allen aktuellen Änderungen im Workspace. Dies umfasst neu hinzugekommene Dateien und Löschungen. Ausgenommen sind nur jene Dateien, die in `.gitignore` eingetragen sind (siehe auch »Mit `.gitignore` Dateien unversioniert lassen« ab Seite 46).*

#### 1. Änderungen anmelden

Mit dem `add`-Befehl werden Änderungen für das nächste Commit angemeldet. Die Option `--all` erfasst alle Änderungen im ganzen Workspace, egal ob geändert, hinzugefügt oder gelöscht wurde.

```
> git add --all
```

Es kann auch die Kurzform `git add -A` verwendet werden.

#### 2. Commit erstellen

Das neue Commit wird erstellt.

```
> git commit
```

## 4.3 Exkurs: Mehr über Commit-Hashes

Auf den ersten Blick wirken die 40 Zeichen langen Commit-Hashes etwas sperrig. Andere Versionsverwaltungen verwenden einfach fortlaufende Nummern (Subversion<sup>2</sup>) oder Versionsnamen wie »1.17« (CVS<sup>3</sup>). Es gibt jedoch gute Gründe, weshalb sich die Entwickler von Git für Hashes entschieden haben:

- Commit-Hashes können lokal erzeugt werden. Eine Kommunikation mit anderen Rechnern oder zentralen Servern ist dabei nicht erforderlich. Ein neues Commit kann man also jederzeit und überall erstellen. Die Commit-Hashes werden aus dem Inhalt der Dateien und den Metadaten (Autor, Commit-Zeitpunkt) errechnet. Die Wahrscheinlichkeit, dass zwei verschiedene Änderungen zufällig denselben Commit-Hash bekommen, ist extrem gering. Immerhin stehen  $2^{160}$  verschiedene Werte zur Verfügung.
- Noch wichtiger ist jedoch Folgendes: Commit-Hashes sind mehr als nur Namen für festgehaltene Softwarestände. Sie sind gleichzeitig auch deren Prüfsumme. Mit dem `fsck`-Befehl von Git kann man die Integrität des Repositorys prüfen lassen. Passen die Inhalte nicht zu den Commit-Hashes, wird ein Fehler gemeldet. Das sieht dann so aus:

```
> git fsck  
error: sha1 mismatch 2b6c746e5e20a64032bac627f2729f72a9cba4ee  
error: 2b6c746e5e20a64032bac627f2729f72a9cba4ee:  
object corrupt or missing
```

Man kann Commit-Hashes auch verkürzt angeben. Meist genügen wenige Zeichen, um ein Commit zu identifizieren.

Verkürzung von Commit-Hashes

```
> git diff 9acc5d5efec1d2d62f7e98bcc3880cda762cb831  
> git diff 9acc5
```

Gibt man zu wenige Zeichen an, liefert Git eine Fehlermeldung:

```
> git diff 9acc  
error: short SHA1 9acc is ambiguous.  
fatal: bad revision '9acc'
```

Außerdem ist es möglich, sprechende Namen (wie zum Beispiel `release-1.2.3`) für Commits zu vergeben. Diese Namen nennt man *Tags*.

Versionen markieren  
→ Seite 105

```
> git checkout release-1.2.3
```

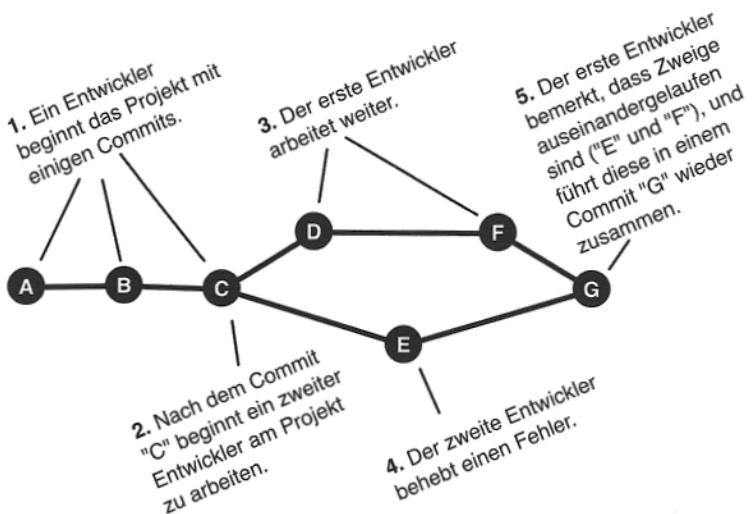
<sup>2</sup> <http://subversion.apache.org/>

<sup>3</sup> <http://www.nongnu.org/cvs/>

## 4.4 Eine Historie von Commits

Das Repository enthält nicht nur die einzelnen Commits, es speichert auch die Beziehungen zwischen den Commits. Jedes Mal, wenn man die Software verändert und dies mit einem Commit bestätigt, merkt Git sich, welches die Vorgängerversion für dieses Commit war. Auf diese Weise entsteht ein Graph (Abbildung 4-2) aus Commits, der die Entwicklung des Projekts abbildet.

**Abb. 4-2**  
Commit-Historie



### Branches verzweigen

→ Seite 59

Interessant wird es, wenn mehrere Entwickler gleichzeitig an der Software arbeiten. Dann entstehen häufig Verzweigungen im Commit-Graphen wie bei »C« und Zusammenführungen wie bei »G«.

## 4.5 Eine etwas andere Sichtweise auf Commits

Man kann ein Commit als einen eingefrorenen Versionsstand ansehen. Man kann es aber auch als eine Menge von Änderungen auffassen, die das Commit im Verhältnis zum Vorgänger eingeführt hat. Man spricht dann auch von einem *Diff* oder einem *Changeset*. Das Repository ist also auch eine Geschichte von Änderungen.

Schritt für Schritt

## Unterschiede zwischen Commits

Der `diff`-Befehl zeigt die Unterschiede zwischen zwei beliebigen Commits.

### a. Zwei Commits

Man kann sich die Unterschiede zwischen zwei Commits auflisten lassen. Statt Commit-Hashes darf man auch symbolische Namen (Branches, Tags, HEAD etc.) angeben.

```
> git diff 77d231f HEAD
```

### b. Unterschiede zum Vorgänger

Mit der Dach-Ausrufezeichen-Notation zeigt der `diff`-Befehl die Unterschiede eines Commits zu seinem unmittelbaren Vorgänger (das *Changeset*).

```
> git diff 77d231f^!
```

### c. Auf Datei(en) beschränken

Man kann die Anzeige der Differenzen auf Dateien oder Verzeichnisse beschränken.

```
> git diff 77d231f 05bcfd1-- buch/bisection/
```

### d. Änderungsstatistik

Oder man lässt sich mit der Option `--stat` nur die Anzahl von Änderungen pro Datei anzeigen.

```
> git diff --stat 77d231f 05bcfd1
```

### e. Fließtexte

Wenn Sie Fließtexte vergleichen, kann die Option `--word-diff` (»Änderungen an Textdateien untersuchen« ab Seite 111) hilfreich sein.

## 4.6 Viele unterschiedliche Historien desselben Projekts

Anfangs ist die dezentrale Architektur von Git gewöhnungsbedürftig. In zentralen Versionsverwaltungen (wie zum Beispiel CVS oder Subversion) gibt es einen zentralen Server, der die Geschichte des Projekts enthält. Bei Git hingegen ist es so, dass jeder Entwickler einen eigenen Klon des Repositorys besitzt (manchmal sogar mehrere). Wenn ein Entwickler Commits erstellt, geschieht dies lokal. Sein Repository hat dann eine andere Geschichte als die Repositorys der anderen Entwickler, die dasselbe Projekt geklont haben. Jedes Repository kann seine eigene Geschichte erzählen. Mit den Befehlen `fetch`, `pull` und `push` können Commits zwischen den Repositorys ausgetauscht werden. Mit dem `merge`-Befehl können die verschiedenen Historien jederzeit wieder zusammengeführt werden.

In vielen Projekten gibt es ein Repository (meist auf dem Projektserver), das die *offizielle* Historie des Projekts darstellt. Ein solches Repository nennt man *Main-Repository* (gelegentlich auch *Blessed Repository*). Dies ist jedoch nur eine Konvention. Aus technischer Sicht sind alle Klone gleichwertig. Wenn beispielsweise das Main-Repository beschädigt wird, kann man einen anderen Klon zum Main-Repository erklären.

### Schritt für Schritt Commit-Historie zeigen

*Der `log`-Befehl zeigt die Historie der Commits.*

#### a. Einfache Log-Ausgabe

> `git log`

```
commit 2753f19072d332dc550f5ec0612a4486ffe3ab4a
Author: Björn Stachmann <bstachmann@yahoo.de>
Date:   Sat Dec 25 11:30:32 2010 +0100
```

TODO für Abbildung verschoben.

```
commit e0ffbdbd9f183e405b280a6c3a970bd860d3de81
Author: Björn Stachmann <bstachmann@yahoo.de>
```

...

### b. Ein paar nützliche Optionen

```
> git log -n 3      # Nur die letzten 3 Commits  
> git log --oneline # Nur eine Zeile je Commit  
> git log --stat    # Nur Statistik zeigen
```

#### Achtung!

Der `log`-Befehl zeigt nicht alle Commits im Repository an. Ohne Parameter aufgerufen, zeigt er die Vorfahren der *HEAD-Revision* an. Die *HEAD-Revision* ist jenes Commit, das im Workspace gerade aktiv ist. Sie wird durch den `commit`-Befehl aktualisiert. Auch mit den Befehlen `checkout` und `reset` (»Branches verzweigen« ab Seite 59) kann die *HEAD-Revision* gewechselt werden.

Der `log`-Befehl kennt zahlreiche Optionen, mit denen man bestimmt, welche Commits in welchem Format angezeigt werden. Im Folgenden sind einige davon beschrieben, die wir selbst häufig nutzen.

### Ausgabe begrenzen: `-n`

Oft ist es nützlich, die Ausgabe zu begrenzen. Das folgende Kommando zeigt beispielsweise nur die letzten drei Commits:

```
> git log -n 3
```

### Ausgabeformat wählen: `--format`, `--oneline`

Das Ausgabeformat für Logs kann mit `--format` beeinflusst werden. So liefert beispielsweise `--format=fuller` viele Details. Für den schnellen Überblick ist das einzeilige Format `--oneline` hilfreich.

```
> git log --oneline  
2753f19 TODO für Abbildung verschoben.  
e0ffbdb Notizen.  
4200ba2 Abschnitt über verschiedene Historien desselben Projekts.  
...
```

### Änderungsstatistik: `--stat`, `--shortstat`

Ebenfalls nützlich sind die Statistiken: `--stat` zeigt, welche Dateien geändert wurden, `--dirstat` zeigt, in welchen Verzeichnissen etwas geändert wurde, und `--shortstat` zeigt eine kurze Zusammenfassung darüber, wie viele Dateien geändert, hinzugefügt und gelöscht wurden.

```
> git log --shortstat --oneline
2753f19 TODO für Abbildung verschoben.
 1 files changed, 2 insertions(+), 2 deletions(-)
e0ffbdb Notizen.
 1 files changed, 27 insertions(+), 4 deletions(-)
4200ba2 Abschnitt über verschiedene Historien desselben Projekts.
 1 files changed, 15 insertions(+), 6 deletions(-)
...

```

### Option: log --graph

Möchte man die Beziehungen zwischen den Commits sehen, zeigt --graph die Historie als »Grafik«.

```
> git log --graph --oneline
*   6d7f278 Merge branch 'master' into redaktion
|\ 
| *  419b389 merge: Formatierungen eingebaut.
| |\ 
| | * 8f5b053 Quickstart: Formatierungen eingebaut.
| | * 5f22c8d Neue Macros zur Formatierung.
| |
* | ab36269 TODOs
* | c2cae84 Intro zu den ersten Schritten.
|/
*   63788eb merge: Abschnitt 'Beispiele und Notation' ergänzt.
```

## 4.7 Zusammenfassung

**Repository:** Im Verzeichnis .git eines Projekts liegt das Git-Repository. Es enthält die Historie des Projekts in Form von Commits. Weil Git dezentral organisiert ist, gibt es für ein Projekt oft viele Repositorys mit unterschiedlichen Historien. Git ist so gebaut, dass es diese bei Bedarf gut wieder zusammenführen kann.

**Commit (auch Version, Revision oder Changeset genannt):**

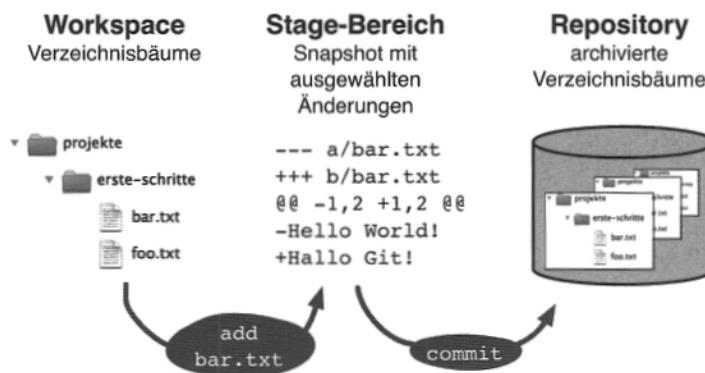
Der commit-Befehl erstellt ein Commit. Es speichert einen definierten Stand des Projekts. Es umfasst den Zustand aller Dateien des Projekts. Jedes Commit enthält Metainformationen über den Autor und den Commit-Zeitpunkt. Insbesondere speichert Git die Vorgänger/Nachfolger-Beziehung. Die Beziehungen bilden den Versionsgraphen des Projekts. Der log-Befehl zeigt Commits aus dem Repository.

**Commit-Hash:** Commit-Hashes identifizieren Commits. Sie dienen gleichzeitig als Prüfsumme für die Integrität des gespeicherten Softwarestands. Commit-Hashes sind 40 Zeichen lang. Für Git-Kommandos dürfen Commit-Hashes verkürzt angegeben werden, z. B. als 5ff8aa9, sofern es nur ein Commit mit diesem Start-Hash gibt.

## 5 Commits zusammenstellen

Ein neues *Commit* muss nicht unbedingt alle Änderungen aus dem Workspace übernehmen. Tatsächlich gibt Git dem Benutzer hier die volle Kontrolle. Im Extremfall kann bis auf die einzelne Codezeile genau bestimmt werden, welche Änderungen in das nächste *Commit* einfließen.

*Commits* entstehen in zwei Schritten. Als Erstes werden die Änderungen mit dem `add`-Befehl in einem Zwischenspeicher gesammelt, den man *Stage-Bereich* oder *Index* nennt. Erst danach überträgt der `commit`-Befehl die Änderungen aus dem *Stage-Bereich* in das Repository.



### 5.1 Der status-Befehl

Der `status`-Befehl zeigt, welche Änderungen aktuell im Workspace vorliegen und welche davon bereits im *Stage-Bereich* für das nächste *Commit* angemeldet sind.

```
> git status
# On branch staging
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:  bar.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in ...)
#
# modified:  foo.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# neu.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Die Ausgabe zeigt verschiedene Rubriken:

**Changes to be committed:** Zeigt jene Dateien, für die beim nächsten Commit Änderungen in das Repository übernommen werden sollen.

**Changed but not updated:** Zeigt Dateien, die Änderungen haben, aber noch nicht für das nächste Commit angemeldet sind.

**Untracked files:** Listet alle neuen Dateien auf.

Hilfreich ist, dass Git hier gleich anzeigt, mit welchen Befehlen man den Status wieder ändern kann. Hier sehen Sie beispielsweise, dass man mit `git reset HEAD blah.txt` die Änderungen an der Datei »blah.txt« aus dem *Stage-Bereich* wieder entfernen kann.

Für CVS<sup>1</sup> - und Subversion<sup>2</sup> -Anwender ist die Verwendung des Begriffs *Update* hier leider etwas verwirrend. In diesen Systemen bezeichnet man mit »Update« die Übernahme von Änderungen aus dem Repository in den Workspace. In Git hingegen bezeichnet man damit die Übernahme von Änderungen aus dem Workspace in den Stage-Bereich. Das ist genau die entgegengesetzte Richtung. Babylon lässt grüßen.

Wenn viele Änderungen vorliegen, kann es nützlich sein, mit `--short` ein kompakteres Ausgabeformat zu wählen. Die Bedeutung der einzelnen Buchstabencodes zeigt `git help status`:

---

<sup>1</sup> <http://www.nongnu.org/cvs/>

<sup>2</sup> <http://subversion.apache.org/>

```
> git status --short  
M blah.txt  
M foo.txt  
M bar.txt  
?? neue-datei.txt
```

Schritt für Schritt

## Selektives Commit – Änderungen auswählen

Ein neues Commit wird erstellt. Es sollen aber nicht alle Änderungen übernommen werden. Dabei kann man ganze Dateien wählen oder gezielt mit `--interactive` einzelne Codeabschnitte herauspicken.

### 1. Änderungen ansehen

```
> git status
```

In den Rubriken »Changed but not updated« und »Untracked files« zeigt der status-Befehl an, welche Dateien noch nicht für das nächste Commit angemeldet sind.

### 2. Änderungen sammeln

Mit dem add-Befehl werden die Änderungen dem *Stage-Bereich* hinzugefügt. Man kann die Dateipfade einzeln angeben. Gibt man ein Verzeichnis an, werden neue und geänderte Dateien darunter (auch aus den Unterverzeichnissen) hinzugefügt. Der add-Befehl kann beliebig oft aufgerufen werden. Man kann dabei auch Dateipfade mit den Jokerzeichen »\*« und »?«, sogenannte *Globs*, verwenden.

```
> git add foo.txt bar.txt # ausgewählte Dateien  
> git add verzeichnis/    # ein Verzeichnis und  
                           # alles darunter  
> git add .              # aktuelles Verzeichnis  
                           # und alles darunter
```

Wenn Sie eine noch feinere Kontrolle wollen, können Sie den interaktiven Modus mit der Option `--interactive` verwenden. Dann können Sie sogar einzelne Codefragmente, im Extremfall einzelne Codezeilen, für das Commit anmelden.

### 3. Commit erstellen

Zum Abschluss werden die Änderungen mit einem Commit übernommen:

```
> git commit
```

Der *Stage-Bereich* ist danach geleert. Der Workspace wird durch das Commit nicht berührt. Änderungen, die nicht mit `add` hinzugefügt wurden, bleiben im Workspace erhalten.

*Tipp: rm-Befehl zum  
Löschen*

Der `rm`-Befehl löscht Dateien oder ganze Verzeichnisse und meldet die Löschungen gleich im *Stage-Bereich* an.

Achtung! Selektive Commits können sehr nützlich sein, um Änderungen voneinander zu trennen, die inhaltlich nicht zusammenhängen. Beispiel: Eine neue Klasse wurde erstellt. Nebenher wurden ein paar Fehler in anderen Klassen korrigiert. Die Trennung in mehrere Commits macht die Historie klarer und erleichtert es, einzelne Fehlerkorrekturen gezielt früher auszuliefern (Cherry-Picking).

Aber man sollte bedenken, dass durch selektive Commits Softwarestände im Repository erzeugt werden, die es lokal so nie gegeben hat. Sie sind also nie getestet worden und können im schlimmsten Fall nicht einmal kompiliert werden. Wir empfehlen, selektive Commits wenn möglich zu vermeiden. Oft genügt es, eine Notiz zu machen, dass man einen Fehler beheben möchte, anstatt ihn sofort zu beheben. Dann stellt man erst das Feature fertig, bevor man die Korrektur vornimmt. Falls der Fehler schnell behoben werden soll, kann man den Zwischenstand der Feature-Entwicklung sichern (Stashing), dann die Korrektur durchführen und danach die Feature-Entwicklung fortsetzen.

#### Schritt für Schritt

### Add und Commit in einem Schritt

Man kann sich den `add`-Befehl oft sparen. Wenn man nur Dateien geändert hat, ohne neue hinzuzufügen, dann kann man die Option `--all` des `commit`-Befehls nutzen.

```
> git commit --all # alle geänderten Dateien
```

Stattdessen kann man auch einzelne Dateien direkt angeben. Die Änderungen werden direkt übernommen und müssen nicht vorher in den Staging-Bereich übertragen werden.

```
> git commit foo.txt bar.txt # ausgewählte Dateien
```

## 5.2 Der Stage-Bereich speichert Momentaufnahmen

Eines sollten Sie über den *Stage-Bereich* unbedingt wissen: Er ist mehr als nur eine Liste mit den Namen der Dateien für das nächste Commit. Er speichert nicht nur, *wo* etwas geändert wurde, sondern auch, *was* geändert wurde. Git erzeugt dazu eine Momentaufnahme (Snapshot) der betroffenen Dateien mit genau den ausgewählten Änderungen. Abbildung 5–2 veranschaulicht dies. In Zeile 1 stimmen *Workspace*, *Stage-Bereich* und *Repository* noch überein. Dann bearbeitet der Entwick-

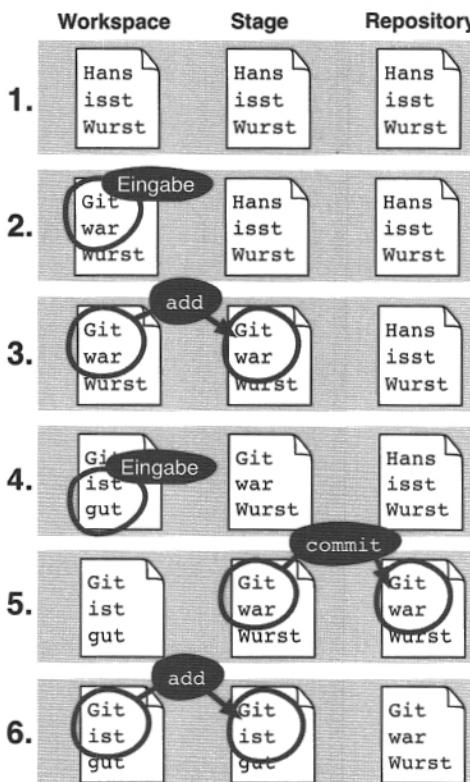


Abb. 5–2  
Änderungen vom *Workspace* über den *Stage-Bereich* ins *Repository* bringen

ler die Datei im *Workspace* (Zeile 2). Mit dem `add`-Befehl überträgt er die Änderung in den *Stage-Bereich* (Zeile drei), aber noch nicht in das *Repository*. In Zeile 4 ändert der Entwickler die Datei erneut. Jetzt enthalten alle drei Bereiche unterschiedliche Stände. Ein `commit`-Befehl überträgt die erste Änderung in das *Repository* (Zeile 5). Die zweite Änderung steht weiterhin im *Workspace*. Mit dem `add`-Befehl wird sie in den *Stage-Bereich* übertragen.

#### Schritt für Schritt

### Was steht im Stage-Bereich? Was nicht?

*Mit dem add-Befehl wurden Änderungen für das nächste Commit ange meldet. Danach wurden weitere Änderungen im Workspace durchge führt. Der diff-Befehl zeigt, was Sache ist.*

#### a. Was steht im Stage-Bereich?

Änderungen, die bereits mit dem `add`-Befehl zum *Stage-Bereich* hinzugefügt wurden, werden mit der Option `--staged` angezeigt. Der folgende Befehl zeigt die Unterschiede zwischen dem *Stage-Bereich* und dem aktuellen *HEAD-Commit* im *Repository*:

```
> git diff --staged    # stage vs. repository
```

#### b. Was ist noch nicht angemeldet?

Ohne Angabe von Optionen zeigt der `diff`-Befehl die lokalen Änderungen im *Workspace*, die noch nicht angemeldet sind, also den Unterschied zwischen *Stage-Bereich* und dem *Workspace*.

```
> git diff      # stage vs. workspace
```

## 5.3 Was tun mit Änderungen, die nicht übernommen werden sollen?

Es kommt vor, dass man bestimmte Änderungen nicht in Commits übernehmen möchte, wie z. B.:

- ─ experimentelle Änderungen für Debug-Zwecke
- ─ versehentlich hinzugefügte Änderungen
- ─ Änderungen, die noch nicht fertig sind
- ─ Änderungen in generierten Dateien

Git bietet verschiedene Möglichkeiten, damit umzugehen:

- ─ Zurücksetzen von experimentellen oder versehentlich durchgeführten Änderungen mit dem `reset`-Befehl
- ─ Ignorieren von Dateien, die man gar nicht übernehmen möchte, mittels `.gitignore`
- ─ Zwischenspeichern von Änderungen, die man erst später übernehmen möchte, mit dem `stash`-Befehl

### Schritt für Schritt

## Zurücknehmen von Änderungen aus dem Stage-Bereich

Der `reset`-Befehl kann den *Stage-Bereich* zurücksetzen. Der erste Parameter `HEAD` gibt an, dass auf die aktuelle `HEAD`-Version zurückgesetzt wird. Der zweite Parameter gibt an, welche Dateien oder Verzeichnisse zurückgesetzt werden sollen.

```
> git reset HEAD .
```

oder

```
> git reset HEAD foo.txt src/test/
```

Achtung! Der *Stage-Bereich* wird beim `reset` überschrieben. In der Regel ist das kein Problem, weil dieselben Änderungen meist noch im Workspace stehen (Abbildung 5–2). Wenn die gleichen Dateien nach dem `add` noch weiter bearbeitet wurden, können Informationen verloren gehen.

## 5.4 Mit `.gitignore` Dateien unversioniert lassen

Generierte Dateien, temporäre Dateien oder von Editoren erstellte Sicherungskopien möchte man in der Regel nicht unter Versionsverwaltung stellen. Durch Einträge in eine Datei `.gitignore`, in der Regel im Hauptverzeichnis des Projekts, kann man sie für Git »unsichtbar« machen. Man kann darin Dateipfade, aber auch Verzeichnisse angeben. Die Jokerzeichen »\*« und »&« sind zulässig (*Glob-Syntax*). Bei Pfadangaben sollte man Folgendes wissen: Eine einfache Pfadangabe, wie `generated/`, sorgt dafür, dass Verzeichnisse dieses Namens überall ignoriert werden, z. B. auch `src/demo/generated`. Stellt man ein `/` voran, z. B. `/generated/`, dann wird nur der exakte Pfad – ausgehend vom dem Pfad, wo `.gitignore` liegt – ignoriert.

```
#  
# Einfacher Dateipfad  
#  
irgendwie/unerwuenscht.txt  
#  
# Verzeichnisse werden mit einem "/" abgeschlossen  
#  
generated/  
#  
# Dateitypen als glob-Ausdrücke  
#  
*.bak  
#  
# "!" markiert Ausnahmen. "demo.bak"  
# wird versioniert, obwohl "*.bak"  
# ausgenommen wurde.  
#  
!demo.bak
```

*Tipp: Mehrere .gitignore-Dateien*

Sie können übrigens eine `.gitignore`-Datei in einem Unterverzeichnis des Projekts anlegen. Sie gilt dann für alle Dateien und Pfade unterhalb dieses Verzeichnisses. Dies kann zum Beispiel dann nützlich sein, wenn Ihr Projekt verschiedene Programmiersprachen umfasst, für die jeweils eine andere Konfiguration erforderlich ist.

*Lokale Änderungen temporär ignorieren*

→ Seite 110

**Anmerkung:** Einträge in `.gitignore` wirken nur auf Dateien, die noch nicht von Git verwaltet werden. Wenn eine Datei bereits versioniert wird, wird der `status`-Befehl Änderungen daran anzeigen, und diese können auch mit dem `add`-Befehl für das nächste Commit angemeldet werden. Möchte man bereits versionierte Dateien ignorieren, kann man dies mit der Option `--assume-unchanged` des `update-index`-Befehls tun.

## 5.5 Stashing: Änderungen zwischenspeichern

Wenn man bei der Arbeit unterbrochen wird, z. B. weil schnell ein Bugfix gemacht werden muss, hat man oft angefangene Änderungen vor sich, die man so noch nicht in ein Commit fassen möchte. In so einem Fall hilft der `stash`-Befehl, mit dem Änderungen lokal zwischengespeichert und später wieder hervorgeholt werden können.

Schritt für Schritt

### Änderungen zwischenspeichern

*Der `stash`-Befehl übernimmt alle Änderungen im Workspace und im Stage-Bereich in einen Zwischenspeicher.*

> `git stash --include-untracked`

Schritt für Schritt

### Änderungen aus dem Zwischenspeicher zurückholen

*Änderungen aus dem Zwischenspeicher können jederzeit mit dem Befehl `stash pop` in den Workspace zurückgeholt werden.*

#### a. Letzte gespeicherte Änderungen zurückholen

> `git stash pop`

#### b.1. Was steht im Zwischenspeicher?

Erst einmal nachsehen, welche Änderungen gespeichert wurden:

```
> git stash list
stash@{0}: WIP on master: 297432e Mindmap aktualisiert.
stash@{1}: WIP on master: 213e335 Einführung einer
Workflow-Beschreibung
```

#### b.2. Ältere gespeicherte Änderung zurückholen

> `git stash pop stash@{1}`

## 5.6 Zusammenfassung

**Stage-Bereich:** Im Stage-Bereich (auch Index genannt) wird das nächste Commit vorbereitet. Er enthält eine Momentaufnahme von Dateiinhalten.

**add erzeugt Momentaufnahmen:** Mit add wird eine Momentaufnahme von geänderten Dateien im Stage-Bereich erzeugt. Ändert man dieselben Dateien noch einmal, gehen die neuen Änderungen nicht automatisch ins nächste Commit ein.

**Selektives Commit:** Beim add-Befehl kann man angeben, welche Dateien bei der Momentaufnahme berücksichtigt werden. Alle anderen Dateien bleiben unverändert.

**Auswählen von Codeabschnitten:** Mit --interactive kann man sogar einzelne Abschnitte von veränderten Zeilen (*Hunks*) auswählen. Nur diese Änderungen werden dann als Momentaufnahme in den Stage-Bereich übertragen.

**Status:** Der status-Befehl zeigt an, welche Dateien ins nächste Commit eingehen und welche Dateien lokal verändert wurden, aber noch nicht im Stage-Bereich angemeldet sind.

**Stage-Bereich zurücksetzen:** Mit git reset HEAD . werden alle Dateien auf den Stand des aktuellen Commits zurückgesetzt.

**.gitignore:** In dieser Datei konfiguriert man, welche Dateien und Verzeichnisse nicht von Git verwaltet werden.

**Stashing:** Mit dem stash-Befehl kann man die aktuellen Änderungen in Workspace und Stage-Bereich zwischenspeichern. Später kann man sie mit git stash pop wieder zurückholen.

# 6 Das Repository

Man kann Git durchaus benutzen, ohne zu wissen, wie das *Repository* funktioniert. Wir glauben aber, dass man ein besseres Verständnis für die Workflows gewinnt, wenn man weiß, wie Git seine Daten speichert und organisiert. Wenn Sie ein absoluter Theorieverächter sind, können Sie dieses Kapitel auch überfliegen und nur die Schritt-für-Schritt-Anleitungen durchsehen.

Git ist in zwei Ebenen aufgebaut. Auf der oberen Ebene finden Sie Befehle wie `log`, `reset` oder `commit`, die komfortabel zu bedienen sind und zahlreiche Optionen und Aufrufvarianten bieten. Die Git-Entwickler nennen diese Befehle »Porcelaine« (gemeint sind die sichtbaren Teile eines Badezimmers, wie etwa Waschbecken und WCs).

Die Ebene darunter wird »Plumbing« (Installation, Rohre) genannt. Hier gibt es eine Reihe von einfachen Befehlen mit wenigen Optionen, auf denen die »Porcelaine«-Befehle aufbauen. »Plumbing«-Befehle werden nur selten direkt verwendet. Dieses Kapitel gibt einen kleinen Einblick in die »Plumbing«-Ebene des Systems.

## 6.1 Ein einfaches und effizientes Speichersystem

Das Herz von Git ist die sogenannte *Object Database*. Man kann darin beliebige Text- oder Binärdaten ablegen, zum Beispiel den Inhalt einer Datei. Der `hash-object`-Befehl mit der Option `-w` (sie steht für »write«) schreibt einen Datensatz in die *Object Database*.

```
> git hash-object -w hallo.txt  
28cf67640e502fe8e879a863bd1bbcd4366689e8
```

Als Ergebnis der Speicherung wird ein 40 Zeichen langer Code geliefert. Dieser ist der Schlüssel für das gespeicherte Objekt. Merkt man sich

diesen, kann man später wieder auf das Objekt zugreifen. Der Zugriff erfolgt z. B. mit dem cat-file-Befehl und der Option -p (für »print«).

```
> git cat-file -p 28cf67640e  
Hallo Welt!
```

Die Implementierung der Object Database ist sehr effizient. Selbst bei großen Projekten mit sehr langer Commit-Historie (wie zum Beispiel dem Linux-Kernel mit mehr als 200.000 Commits und fast zwei Millionen Objekten) erfolgt der Zugriff auf Objekte aus dem Repository fast augenblicklich. Git ist extrem gut geeignet für Projekte mit sehr vielen kleinen Quelltextdateien. Die Grenzen zeigen sich erst, wenn das Gesamtdatenvolumen der Daten sehr groß wird. Wer beispielsweise große binäre Dateien verwalten möchte, der ist mit einem Git-Repository nicht so gut bedient.

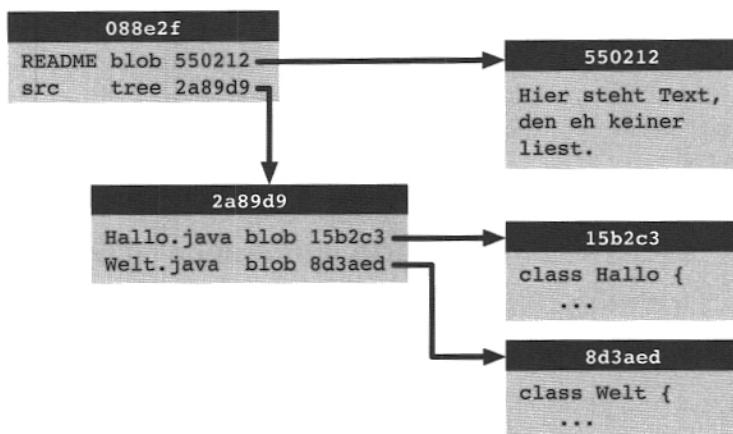
## 6.2 Verzeichnisse speichern: Blob und Tree

Zum Speichern von Dateien und Verzeichnissen verwendet Git eine einfache Baumstruktur mit zwei Knotentypen. Die Inhalte von Dateien werden unverändert, Byte für Byte, als Blob-Objekte in der *Object Database* abgelegt. Verzeichnisse werden durch sogenannte Tree-Objekte dargestellt, die wie folgt aussehen:

```
> git cat-file -p 2790ef78  
100644 blob 507d3a30ae9ed53bcf953744c5f5c9391a263356 README  
040000 tree 91c7822ab43800b0e3c13049519587df4fd74591 src
```

Das Tree-Objekt listet die enthaltenen Dateien und Unterverzeichnisse auf. Abbildung 6–2 zeigt dies. Jeder Eintrag enthält Informationen über die Zugriffsrechte (z. B. 100644), den Typ (blob oder tree), einen Hashcode für den Dateinhalt sowie den Namen der Datei bzw. des Verzeichnisses.

**Abb. 6–1**  
*Ein kleines Projekt*  
beispiel-workspace/  
  README  
  /src  
    Hallo.java  
    Welt.java



**Abb. 6–2**  
Darstellung von  
Verzeichnissen im  
Repository

## 6.3 Gleiche Daten werden nur einmal gespeichert

Um Speicher zu sparen, werden gleiche Daten nur einmal gespeichert. Im folgenden Beispiel erhalten die Dateiinhalte von `foo.txt` und `kopie-von-foo.txt` denselben Hashcode, weil sie gleich sind:

```
> git hash-object -w foo.txt
a42a0aba404c211e8fdf33d4edde67bb474368a7

> git hash-object -w kopie-von-foo.txt
a42a0aba404c211e8fdf33d4edde67bb474368a7
```

Git spart durch dieses Vorgehen nicht nur Speicher, sondern gewinnt gleichzeitig Performance. Bei vielen Operationen ist dieses Verfahren rasend schnell, weil es in den Algorithmen oft nur Hashcodes vergleicht, ohne die eigentlichen Daten anzusehen.

## 6.4 Kompression ähnlicher Inhalte

Git kann aber mehr, als nur identische Dateiinhalte zusammenzufassen. Beim Programmieren entstehen laufend neue Dateiinhalte, die sich nur in wenigen Zeilen von ihren Vorgängerversionen unterscheiden. Git kann diese Dateiinhalte mit einem Deltaverfahren, das nach einer Ursprungsversion nur die Änderungen ablegt, in sogenannten *Pack Files* speichern.

Hierzu ruft man den `gc`-Befehl auf, wenn man Speicher sparen möchte. Git entfernt dann nicht benötigte Commits, die von keinem Branch-Head mehr erreichbar sind, und speichert die verbleibenden

Commits in *Pack Files*. Bei Projekten, die hauptsächlich Quelltexte enthalten, wird eine erstaunlich hohe Kompression erreicht. Oft ist der entpackte Workspace-Inhalt mit der aktuellen Version des Projekts größer als das Git-Repository mit der über Jahre gesammelten Historie des Projekts.

## 6.5 Ist es schlimm, wenn verschiedene Daten zufällig denselben Hashwert bekommen?

Das wäre in der Tat schlecht, weil Git Inhalte über Hashwerte identifiziert. Git könnte also falsche Daten liefern, wenn die Inhalte verschiedener Dateien zufällig denselben Hashwert hätten. Man nennt dies eine Hashkollision.

Die gute Nachricht ist aber, dass eine Hashkollision ein extrem unwahrscheinliches Ereignis ist. Das liegt daran, dass es immerhin  $2^{160}$  mögliche Hashwerte gibt. Das Linux-Kernel-Projekt hat nach fünf Jahren intensiver Entwicklung »nur« ca.  $2^{21}$  Objekte im Repository.<sup>1</sup>

## 6.6 Commits

Auch Commits werden in der *Object Database* abgelegt. Das Format ist einfach:

```
> git cat-file -p 64b98df0
tree 319c67d41a0b3f7464550b41db4bb1584939ad2a
parent 6c7f1ba0828a5b595026e08d2476808105a6b815
author Bjørn Stachmann <bs@test123.de> 1295906997 +0100
committer Bjørn Stachmann <bs@test123.de> 1295906997 +0100
```

Abschnitt über Trees & Blobs.

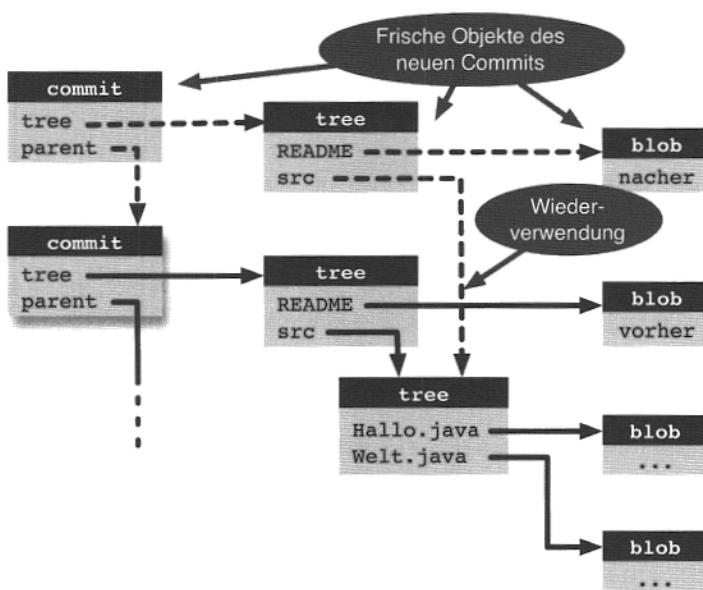
Neben den Metainformationen – wie Autor, Committer, Zeitpunkt und Kommentar – enthält das *Commit-Objekt* einige Hashwerte für weitere Objekte aus der *Object Database*. Das Tree-Objekt beschreibt den Inhalt des Commits. Es verweist auf das Hauptverzeichnis des Projekts und wird wie oben beschrieben durch *Trees* und *Blobs* dargestellt. Das *Parent-Objekt* repräsentiert das Vorgänger-Commit.

---

<sup>1</sup>In der Theorie hat der SHA1-Hash Schwächen. Es wurde ein Algorithmus beschrieben, in dem man mit »nur«  $2^{51}$  Operationen eine SHA1-Kollision finden könnte. Ein Forschungsprojekt der Technischen Universität Graz hat von 2007 bis 2009 versucht, eine(!) solche *Hashkollision* tatsächlich zu finden. Es wurde aber ohne Sucherfolg eingestellt. Zusammenfassend kann gesagt werden, dass die Sicherheit für die Zwecke einer Versionsverwaltung heute ganz okay ist.

## 6.7 Wiederverwendung von Objekten in der Commit-Historie

Bis auf das allererste Commit hat jedes Commit mindestens ein *Vorgänger-Commit (Parent-Objekt)*. Oft verändert ein Commit nur wenige Dateien eines Projekts, und der Großteil der Dateien und Verzeichnisse bleibt unverändert. Wann immer es möglich ist, verwendet Git die Objekte aus vorigen Commits wieder.



**Abb. 6-3**  
Wiederverwendung  
eines Tree-Objekts

Abbildung 6-3 zeigt ein Beispiel. Ein Commit (unten, durchgezogene Pfeile) enthält eine Datei README und ein Verzeichnis src mit weiteren Dateien. Auf diesem baut ein weiteres Commit (oben, gestrichelte Pfeile) auf, in dem nur die Datei README geändert wurde. Man sieht, dass für README ein neues Blob-Objekt entstanden ist. Für das src-Verzeichnis hingegen konnten das Tree-Objekt und die dazugehörigen Blob-Objekte unverändert weitergenutzt werden.

## 6.8 Umbenennen, verschieben und kopieren

Viele Versionsverwaltungssysteme ermöglichen es, die Geschichte von Dateien auch über Umbenennungen und Verschiebungen hinweg zu verfolgen. Meist wird dies dadurch erreicht, dass man einen besonderen Befehl zum Verschieben oder Umbenennen verwendet. In Subversion<sup>2</sup> zum Beispiel verschiebt man Dateien mit `svn move`. Dort ist es schlecht, wenn man Dateien beispielsweise per »Drag and Drop« verschiebt. Subversion erfährt dann nichts von der Verschiebung und protokolliert stattdessen eine Löschung einer Datei und die Erstellung einer neuen Datei.

Git verfolgt einen anderen Ansatz: Es speichert keine Informationen darüber, welche Dateien verschoben wurden. Stattdessen besitzt es einen »Rename Detection«-Algorithmus: Fehlt eine Datei in einem Commit, die beim Vorgänger noch da war, prüft Git, ob an anderer Stelle eine Datei mit gleichem oder sehr ähnlichem Inhalt aufgetaucht ist. Wenn das so ist, geht Git davon aus, dass die Datei verschoben wurde. Abbildung 6–4 zeigt dies: Die verschobene Datei `foo.txt` fehlt im zweiten Commit. Git sucht dann unter den neu hinzugekommenen Dateien nach ähnlichem Inhalt und findet diesen in `src/foo-verschoben.txt`. Dies wird dann als Umbenennung interpretiert.

**Abb. 6–4**  
Eine Datei wird verschoben.

	beispiel-workspace/	beispiel-workspace/
(Commit 1)	foo.txt	(foo.txt fehlt)
	/src	/src
	bar.txt	bar.txt
		foo-verschoben.txt
(Commit 2)		

### Schritt für Schritt

## Umbenennungen und Verschiebungen verfolgen

Git soll anzeigen, welche Dateien umbenannt oder verschoben wurden.

### 1. Sich einen Überblick verschaffen

Mit der Option `-M` (für »Move«) beim `log`-Befehl aktiviert man die *Rename Detection*. Für die Ausgabe nutzt man die Option `--summary`, die Informationen über Dateiänderungen darstellt. Die Ausgabe dabei ist jedoch lang. Wenn man es kompakter mag, filtert man die gesuchten `rename`-Zeilen mit dem `grep`-Befehl heraus. Die Prozentzahl gibt dabei an, wie ähnlich Quell- und Zielfile sein müssen.

<sup>2</sup> <http://subversion.apache.org/>

```
> git log --summary -M90% | grep -e "^\ rename"  
rename foo.txt => foo-umbenannt.txt (90%)  
rename src/{vorher => nachher}/bar.txt (100%)
```

## 2. Historie einer verschobenen Datei verfolgen

Mit der Option `--follow` wird das Log für eine einzelne Datei auch über Verschiebungen und Umbenennungen hinweg verfolgt. Die Ausgabe entspricht einer normalen Log-Ausgabe. Ohne diese Option würde das Log bei dem Commit enden, in dem die Datei umbenannt wurde.

```
> git log --follow foo-umbenannt.txt
```

Schritt für Schritt

## Kopien aufspüren

Git kann auch kopierte Daten aufspüren. Dafür verwendet man die Option `-C`:

```
> git log --summary -C90% | grep -e "^\ copy"
```

Bei Bedarf kann man die Option `--find-copies-harder` hinzunehmen. Git rechnet dann länger. Dafür untersucht es dann alle Dateien eines Commits und nicht nur jene, die in diesem Commit verändert worden sind.

Man kann Git auch so konfigurieren, dass *Rename Detection* standardmäßig aktiviert ist. Dann muss man die Optionen `-M` und `--follow` nicht für jeden `log`-Aufruf einzeln angeben.

*Tipp: Rename  
Detection als Default*

```
> git config diff.renames true
```

Schritt für Schritt

## Herkunft von Codeabschnitten bestimmen

*Sie wollen herausfinden, wer wann welche Codezeilen zuletzt bearbeitet hat.*

### 1. Herkunftsinformationen zeilenweise ausgeben

Git kann die Herkunft von Codezeilen selbst dann bestimmen, wenn größere Codeabschnitte aus anderen Dateien kopiert oder verschoben wurden. Der `blame`-Befehl zeigt für jede Zeile an, wann und durch wen diese Zeile zuletzt geändert wurde.

```
> git blame -M -C -C zusammenkopiert.txt
f5fdbad0 foo.txt  (Rene 2010-11-14 18:30:42 +0100 1) Eins,
a5b80903 bar.txt  (Björn 2011-01-31 21:32:49 +0100 2) Zwei oder
f5fdbad0 foo.txt  (Rene 2010-11-14 18:30:42 +0100 3) Drei!
```

Mit der Option `-M` (für Move) werden Kopien und Verschiebungen innerhalb einer Datei aufgedeckt. Mit der Option `-C` werden auch Kopien aus anderen Dateien im selben Commit entdeckt. Man kann `-C` auch mehrfach angeben; dann durchsucht Git auch Dateien aus anderen Commits. Bei großen Repositorys kann das dann aber auch mal etwas länger dauern.

## 6.9 Zusammenfassung

**Object Database:** Dateien, Verzeichnisse und alle Metainformationen für Commits werden in dieser Datenbank abgelegt.

**SHA1-Hashes:** Über den SHA1-Hash kann man die Objekte aus der Objektdatenbank abrufen. Der SHA1-Hash ist eine kryptografische Prüfsumme über den Inhalt der Datei.

**Identische Daten werden nur einmal gespeichert:** Objekte mit gleichem Inhalt haben denselben SHA1-Hash und werden nur einmal abgelegt.

**Ähnliche Daten werden komprimiert:** Hierzu gibt es ein Deltaverfahren, das nur die Änderungen speichert.

**Blob:** Inhalte von Dateien werden in sogenannten *Blobs* abgelegt.

**Tree:** Verzeichnisse werden in sogenannten *Tree*-Objekten abgelegt. Sie enthalten eine Liste von Dateinamen mit dem SHA1-Hash des dazugehörigen Inhalts (wiederum ein Blob oder ein Tree).

**Commit-Graph:** Die Commit-Objekte bilden zusammen mit den Tree- und Blob-Objekten den Commit-Graphen.

**Rename Detection:** Die Verschiebungen und Umbenennungen müssen dazu nicht vor dem Commit angemeldet werden. Git erkennt dies nachträglich durch Ähnlichkeiten in den Inhalten der Dateien. Beispiel: `git log --follow`

**Wer war's:** Mit dem `blame`-Befehl kann Git die Herkunft von Codezeilen sogar dann bestimmen, wenn diese verschoben oder kopiert wurden.



## 7 Branches verzweigen

Es gibt wichtige Gründe, weshalb Versionshistorien nicht immer linear, *Commit auf Commit*, verlaufen:

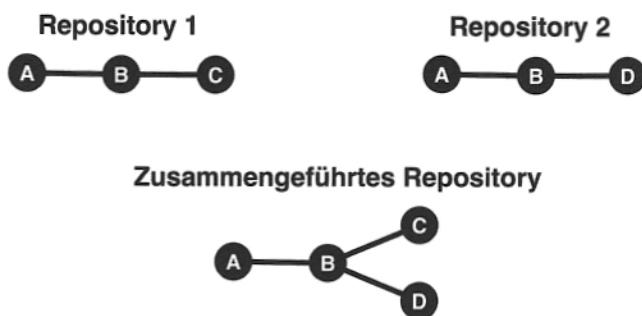
- Zwei oder mehr Entwickler arbeiten unabhängig voneinander am selben Projekt.
- Bugfixes für ältere Versionen müssen erstellt und ausgeliefert werden.
- Mehrere Features sollen parallel entwickelt und erst integriert werden, wenn sie fertig sind.
- Die Software soll für ein Release stabilisiert werden, während parallel schon an der nachfolgenden Version gearbeitet wird.

In beiden Fällen entstehen Verzweigungen im Graphen der *Commit-Historie*.

### 7.1 Parallele Entwicklung

Wenn mehrere Entwickler mit Git an derselben Software arbeiten, entstehen Verzweigungen im Commit-Graphen. Die obere Hälfte von Abbildung 7-1 zeigt, wie zwei Entwickler verschiedene Nachfolgerversionen (Commits C und D) für ein Commit B in ihren lokalen Repositories erstellen. Unten sieht man das Repository nach einer Zusammenführung (wie man das macht, erfahren Sie in Kapitel 11). Es ist eine Verzweigung entstanden. Diese Art von Verzweigung lässt sich kaum vermeiden, wenn parallel entwickelt wird.

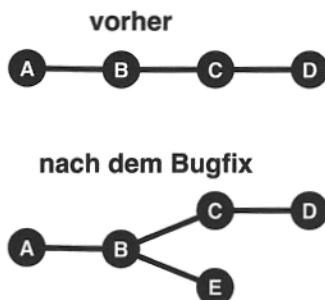
Abb. 7-1  
Parallele Entwicklung



## 7.2 Bugfixes in älteren Versionen

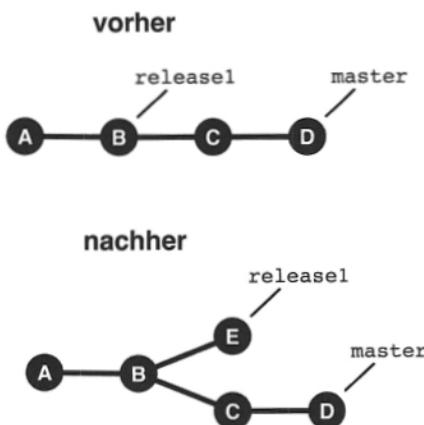
Außerdem entstehen Verzweigungen, wenn man Fehler in älteren Softwareversionen beseitigt. Abbildung 7-2 zeigt ein Beispiel: Während die Entwickler schon intensiv an der Version für das kommende Release arbeiten (Commits C und D), wird ein Fehler in der ausgelieferten Version (Commit B) entdeckt. Da die neuen Features noch nicht ausgeliefert werden sollen, wird ein Bugfix E auf Basis von B erstellt.

Abb. 7-2  
Bugfix in alter Version



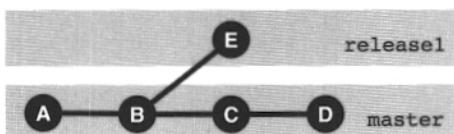
## 7.3 Branches

Um in verzweigten Versionsgraphen den Überblick zu behalten, gibt es Branches. Ein *Branch* zeigt auf einen Knoten im Versionsgraphen. Im Beispiel von Abbildung 7-3 sieht man zum einen den Branch `release1` für die ausgelieferte Version und zum anderen einen Branch `master` für die aktuelle Entwicklung. Bei jedem neuen Commit »wandert« der aktive Branch mit. Im unteren Bild rechts sieht man, wie der Branch `release1` mit dem Bugfix »weitergewandert« ist und dadurch eine Verzweigung im Versionsbaum erzeugt hat.



**Abb. 7-3**  
Branches im  
Versionsgraphen

Man kann sich *Branches* als parallele Linien der Entwicklung vorstellen. Man kann dies durch Swimlanes (Schwimmbahnen) im Commit-Graphen (Abbildung 7-4) visualisieren.



**Abb. 7-4**  
Branches als parallele  
Entwicklungslinien

**Anmerkung:** Git kennt keine feste Zuordnung von Commits zu *Branches*; die Aufteilung in Bahnen ist eine Interpretation, also zu einem gewissen Grad willkürlich.

## 7.4 Aktiver Branch

In einem Git-Repository gibt es immer genau einen aktiven Branch. Der `branch`-Befehl (ohne Optionen) zeigt eine Liste mit allen Branches. Der aktive Branch wird mit einem \* hervorgehoben.

```
> git branch
  ein-zweig
* master
  noch-ein-zweig
```

Bei einem *Commit* wird immer der aktive *Branch* weitergeführt. Dieser zeigt dann auf das neu entstandene Commit. Mit dem `checkout`-Befehl kann man den aktiven *Branch* wechseln.

Dabei werden Dateien im Workspace ausgetauscht, sodass sie dem *Commit* entsprechen, auf das der neue *Branch* zeigt.

```
> git checkout ein-zweig
```

**Anmerkung:** Falls es lokale Änderungen im Workspace (oder im Stage-Bereich) gibt, wird Git diese erhalten. Wenn das nicht möglich ist, erscheint die Meldung »Your local changes to the following files would be overwritten by checkout«, und der Wechsel wird nicht ausgeführt. Sie können in diesem Fall ihre Änderungen mit dem `stash`-Befehl sichern.

#### Schritt für Schritt

### Branch erstellen

*Ein neuer Branch wird angelegt.*

#### 1a. Vom aktuellen Commit abzweigen

> `git branch ein-zweig`

#### 1b. Von beliebigem Commit abzweigen

Man kann auch von beliebigen anderen Stellen abzweigen. Dafür muss man lediglich angeben, bei welchem Commit der neue Branch starten soll.

> `git branch noch-ein-zweig 38b7da45e`

#### 1c. Von vorhandenem Zweig abzweigen

> `git branch noch-ein-zweig alter-zweig`

### 2. Auf den neuen Branch wechseln

Der `branch`-Befehl erzeugt lediglich einen neuen Branch, wechselt aber nicht dorthin. Letzteres geschieht durch den `checkout`-Befehl.

> `git checkout ein-zweig`

### Abkürzung: Branch erzeugen und gleich wechseln

> `git checkout -b ein-zweig`

Schritt für Schritt

## Checkout verweigert. Was nun?

Normalerweise kann man mit dem checkout-Befehl einfach zwischen den Branches hin- und herspringen. Falls der Workspace jedoch noch lokale Änderungen enthält, muss man entscheiden, wie man mit diesen umgehen möchte.

### 1. Checkout

Hier wird ein Checkout verweigert:

```
> git checkout ein-zweig  
error: Your local changes to the following files would be  
overwritten by checkout: foo.txt  
Please, commit your changes or stash them before you can  
switch branches.  
Aborting
```

Es liegen Änderungen im Workspace oder Stage-Bereich vor, die noch nicht durch ein Commit bestätigt wurden. Sie müssen entscheiden, was mit den Änderungen geschehen soll.

#### 2a. Commit und dann wechseln

```
> git commit --all  
> git checkout ein-zweig
```

#### 2b. Verwerfen und dann wechseln

Sie können den Wechsel auch mit der Option --force erzwingen. Achtung! Dabei werden lokale Änderungen unwiderruflich überschrieben!

```
> git checkout --force ein-zweig
```

#### 2c. Zwischenspeichern und dann wechseln

Sie können die Änderungen mit dem stash-Befehl (siehe auch »Stashing: Änderungen zwischenspeichern« ab Seite 47) zwischenspeichern, um sie später mit dem stash pop-Befehl zurückzuholen.

```
> git stash  
> git checkout ein-zweig
```

## 7.5 Branch-Zeiger umsetzen

Der Branch-Zeiger für den aktiven Branch wird mit jedem Commit weitergeführt. Daher ist es nur selten erforderlich, den Branch-Zeiger direkt zu setzen. Manchmal kommt es vor, dass man sich verrannt hat und zu einem früheren Stand zurückkehren möchte. Dann kann man den Branch-Zeiger mit dem reset-Befehl zurücksetzen.

```
> git reset --hard 39ea21a
```

Hier wurde der Zeiger für den aktiven Branch auf das Commit 39ea21a gesetzt. Die Option `--hard` sorgt dafür, dass Workspace und Stage-Bereich ebenfalls auf den Stand von 39ea21a gesetzt werden.

**Achtung!** `reset -hard` überschreibt Workspace und Stage-Bereich. Es können Änderungen verloren gehen! Sie sollten die offenen Änderungen mit `git stash` vorher zwischenspeichern (siehe »Stashing: Änderungen zwischenspeichern« ab Seite 47).

*Tipp: stash vor dem  
reset*

## 7.6 Branch löschen

Schritt für Schritt

### Branch löschen

*Ein Branch kann mit `branch -d` gelöscht werden.*

#### a. Abgeschlossenen Branch löschen

Den aktiven Branch darf man nicht löschen. Deshalb wechselt man auf den `master`-Branch.

```
> git checkout master  
> git branch -d b-branch
```

#### b. Offenen Branch löschen

Git warnt und verweigert die Löschung, falls der betroffene Branch noch nicht in den aktiven Branch, z. B. den `master`, oder seinen Upstream-Branch (Abschnitt 11.3) überführt wurde. Möchte man den Branch trotzdem löschen, kann man die Option `-D` angeben.

```
error: The branch '-branch' is not fully merged.  
If you are sure you want to delete it,  
run 'git branch -D b-branch'.
```

```
> git branch -D b-branch  
Deleted branch b-branch (was 742dcf6).
```

Schritt für Schritt

## Versehentlich gelöschten Branch wiederherstellen

*Git löscht zunächst nur den Zeiger auf das Commit, der zur Verwaltung des Branch verwendet wird. Die Commit-Objekte bleiben im Repository erhalten. Man kann den Branch wiederherstellen, wenn man sich den Commit-Hash aus der Meldung nach dem Löschen (siehe Beispiel oben) gemerkt hat.*

### a. Branch herstellen (Commit-Hash bekannt)

```
> git branch a-branch 742dcf6
```

#### b.1. Commit-Hash ermitteln

Wenn man den Commit-Hash nicht mehr weiß, kann man diesen meist im lokalen *Reflog* finden. Dort wird die Historie der Commits der Branch-Referenzen gespeichert.

```
> git reflog  
c765ale HEAD@{0}: checkout: moving from b-branch to master  
88117f6 HEAD@{1}: merge b-branch: Fast-forward  
9332b08 HEAD@{2}: checkout: moving from a-branch to b-branch  
441cdef HEAD@{3}: commit: Wichtige Dinge ergänzt
```

Mehr darüber lesen Sie in »Keine Panik – es gibt ein Reflog!« ab Seite 109.

#### b.2. Branch wiederherstellen (Commit-Hash aus Reflog)

```
> git branch b-branch HEAD@{1}
```

## 7.7 Und was ist, wenn man die Commit-Objekte wirklich loswerden will?

**Ein Repository klonen**

→ Seite 91

**Repositorys erstellen,**

**klonen und verwalten**

→ Seite 89

Der gc-Befehl (Garbage Collect) räumt im Repository auf und entfernt Commit-Objekte, die von den aktuellen Branches aus nicht erreichbar sind. Wenn man wirklich sichergehen möchte, empfiehlt es sich, das Repository zu klonen und das Original-Repository zu löschen. Dann hat man definitiv einen sauberen Stand.

## 7.8 Zusammenfassung

**Verzweigungen im Commit-Graphen:** Bei paralleler Entwicklung und beim Bugfixen auf alten Versionen verzweigt sich der Commit-Graph.

**Branches:** Ein Branch gibt einem Zweig im Commit-Graphen einen Namen. Der Branch hat einen Zeiger auf das jüngste Commit in diesem Zweig.

**Aktiver Branch:** Im Normalfall ist immer ein Branch aktiv. Macht man dort neue Commits, wird der Zeiger weitergesetzt.

**Branch erstellen:** Mit dem branch-Befehl erstellt man einen neuen Branch.

**Checkout:** Mit dem checkout-Befehl wechselt man auf einen anderen Branch.

**Reflog:** Git führt ein Log über jede Änderung an den Branch-Zeigern, insbesondere auch über jedes Commit. Dies ist hilfreich, wenn man versehentlich gelöschte Branches wiederherstellen möchte.

**Garbage:** Commits, die nicht Vorgänger eines Branch sind, gelten als Garbage. Sie können mit dem gc-Befehl aufgeräumt werden.

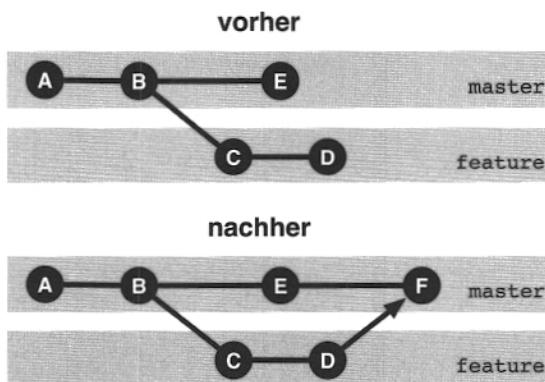
## 8 Branches zusammenführen

Die wichtigste Operation auf Branches ist das Zusammenführen mit dem `merge`-Befehl. Auch wenn die zugrunde liegenden Algorithmen komplex sind, ist der Aufruf einfach. Man gibt den Namen des *Branch* an, dessen Änderungen integriert werden sollen. Es entsteht dann ein neues Commit mit den zusammengeführten Inhalten.

Abbildung 8–1 zeigt ein Beispiel: Während einige Entwickler auf einem *Branch* namens `feature` emsig weiterentwickeln, hat ein anderer Entwickler einen Fehler auf dem *Branch* `master` behoben (Commit E) und eine korrigierte Version der Software ausgeliefert. Kurz darauf wird das Feature fertiggestellt und soll ebenfalls ausgeliefert werden. Die nächste Version auf dem `master`-Branch soll sowohl den Bugfix als auch das neue Feature enthalten. Mit dem `merge`-Befehl führt man die Zweige zusammen. Dabei entsteht ein sogenanntes *Merge-Commit* (hier: F), das zwei Vorgänger hat (D und E).

```
> # auf dem Branch "master"
> git merge feature
```

**Anmerkung:** Verändert wurde nur der `master`, der *Branch* `feature` bleibt unverändert.



**Abb. 8–1**  
*Merge:*  
Zusammenführen von  
Branches

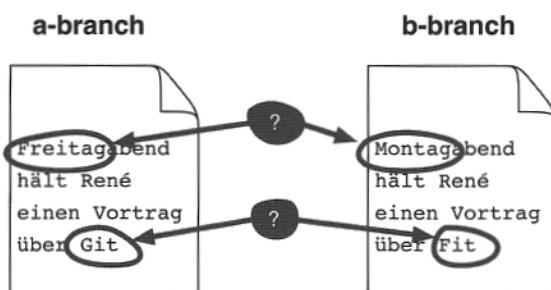
Achtung! Wenn Änderungen im Workspace oder im Stage-Bereich vorliegen, erlaubt Git kein Merge. Änderungen sollten vorher mit einem Commit oder mit dem stash-Befehl gesichert werden.

## 8.1 Was passiert bei einem Merge?

Ein Ziel von Git ist es, die Zusammenarbeit von dezentral arbeitenden Entwicklern so leicht wie nur möglich zu machen. Deshalb soll der merge-Befehl Branches weitgehend automatisch, also ohne Benutzerinteraktion, zusammenführen. Doch wie ist das möglich?

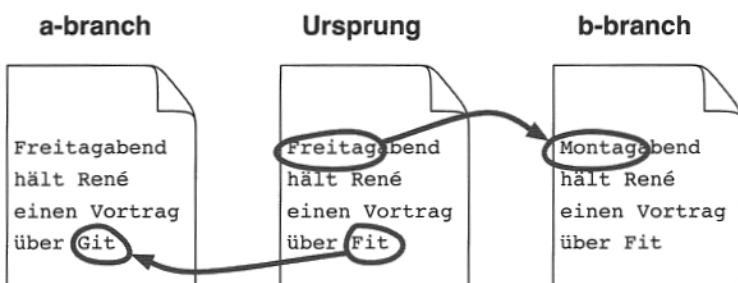
Abbildung 8–2 zeigt zwei verschiedene Versionen eines Dateiinhalts, eine vom a-branch und eine vom b-branch. Es ist recht leicht zu erkennen, welche Zeilen sich unterscheiden. Aber welche Variante ist die richtige? »Freitag« oder »Montag«? »Git« oder »Fit«? Wie soll ein Algorithmus das entscheiden?

**Abb. 8–2**  
Zwei Versionen:  
Welche ist richtig?



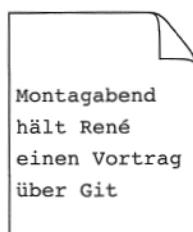
Der Schlüssel dazu liegt oft in der *Commit-Historie*. Der Trick besteht darin, den letzten gemeinsamen Vorgänger zu finden. Etwas vereinfacht ausgedrückt ist das die Stelle, wo sich die Wege der *Branches* getrennt haben. Vergleicht man die Ursprungsversion mit den Varianten in den *Branches*, wird das Bild klarer.

**Abb. 8–3**  
3-Wege-Betrachtung



Im Beispiel in Abbildung 8–3 erkennt man, dass im b-branch in der ersten Zeile der »Freitagabend« durch »Montagabend« ersetzt wurde. Im a-branch wurde die erste Zeile gar nicht verändert. Das ist ein starkes Indiz dafür, dass man den »Montagabend« mitnehmen sollte, wenn man beide *Branches* zusammenführt. Auf dieselbe Weise schließt man, dass in der letzten Zeile »Git« und nicht »Fit« zu übernehmen ist. Abbildung 8–4 zeigt das Ergebnis, das der Merge-Algorithmus<sup>1</sup> liefern würde.

**Merge-Ergebnis**



**Abb. 8–4**  
*Merge-Ergebnis*

## 8.2 Konflikte

Git ist sehr gut darin, Änderungen an Programmquelltexten zusammenzuführen, wenn verschiedene Entwickler verschiedene Stellen darin bearbeitet haben. Dies funktioniert oftmals selbst dann, wenn betroffene Dateien verschoben oder umbenannt wurden. Leider gibt es trotzdem immer wieder Konflikte, die auch Git nicht automatisch auflösen kann.

**Bearbeitungskonflikte** treten auf, wenn zwei Entwickler dieselben Codezeilen in unterschiedlicher Weise geändert haben. Dann kann Git nicht entscheiden, welche von beiden Änderungen die richtige ist.

**Inhaltliche Konflikte** treten dann auf, wenn zwei Entwickler verschiedene Codeteile so geändert haben, dass sie zusammen nicht mehr funktionieren. Dies passiert zum Beispiel dann, wenn ein Entwickler eine Funktion inhaltlich verändert und ein anderer Entwickler parallel dazu etwas programmiert, was noch auf der alten Arbeitsweise dieser Funktion beruht.

<sup>1</sup>Tatsächlich ist es gar nicht so ganz einfach, den oben erwähnten gemeinsamen Vorgänger zu finden. Git implementiert daher drei verschiedene Merge-Algorithmen. Default ist der »recursive«-Algorithmus, auf den die Linux-Kernel-Entwickler schwören. Außerdem sind der klassische 3-Wege- und der »octopus«-Algorithmus implementiert. »octopus« kann viele Branches gleichzeitig zusammenführen.

## Bearbeitungskonflikte

Wenn Git einen Konflikt nicht auflösen kann, erscheint eine Fehlermeldung:

```
> git merge ein-branch
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Folgendes ist dann geschehen:

1. Git hat *kein* Commit erstellt. Normalerweise tut Git dies nach einem Merge automatisch. Im Konfliktfall muss man zuerst die Probleme aus dem Weg räumen und danach das Commit manuell erstellen.
2. In `.git/MERGE_HEAD` steht der Commit-Hash des anderen Branch.
3. Die Dateien im Workspace spiegeln das Merge-Ergebnis wider.
4. Konfliktfrei zusammengeführte Änderungen sind im Stage-Bereich (Seite 39) schon für das nächste Commit angemeldet.
5. *Konfliktmarkierungen* wurden eingefügt.
6. Die Konfliktstellen sind noch nicht für das nächste Commit angemeldet.

Der `status`-Befehl zeigt jetzt in der Sektion `Changes to be committed`: die automatisch zusammengeführten Dateien an und in der Sektion `Unmerged paths`: jene Dateien, die der Benutzer noch manuell nachbearbeiten sollte.

```
> git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Changes to be committed:
#
# modified:   blah.txt
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
# both modified:    foo.txt
#
```

## Konfliktmarkierungen

Eine Konfliktmarkierung zeigt beide Varianten. Zuerst sieht man die Zeilen, wie sie auf dem aktuellen Branch (`HEAD`) vorher aussahen. Dahinter sieht man, wie sie auf dem anderen Branch (`MERGE_HEAD`, hier `ein-branch`) aussahen:

```
Im Frühtau
<<<<< HEAD
zu Tale
=====
zum Schwimmen
>>>> ein-branch
wir geh'n. Fallera!;
```

Aus historischen Gründen wird der gemeinsame Vorgänger dabei standardmäßig nicht angezeigt. Sie können sich das 3-Wege-Format aber konfigurieren:

*Tipp: 3-Wege-Format für Konflikte*

```
> git config merge.conflictstyle diff3
```

Ein Bearbeitungskonflikt wird dann wie folgt dargestellt:

```
Im Frühtau
<<<<< HEAD
zu Tale
||||| merged common ancestors
zu Berge
=====
zum Schwimmen
>>>> ein-branch
wir geh'n. Fallera!;
```

Zwischen der eigenen und der fremden Variante wird gezeigt, wie der Code im letzten gemeinsamen Vorgänger aussah.

## Bearbeitungskonflikte lösen

Am besten löst man Bearbeitungskonflikte mit einem Merge-Tool, wie zum Beispiel `kdiff3`. Man startet das Merge-Tool mit dem `mergetool`-Befehl:

```
> git mergetool
```

Dort löst man die Konflikte, speichert die Änderungen und beendet die Anwendung wieder. Danach stehen die zusammengeführten Änderungen im Stage-Bereich und können mit einem Commit bestätigt werden.

**Tipp: Konflikte in Binärdateien**

Für Binärdateien gibt es keine textuelle Konfliktmarkierung. Hier muss man sich die Ursprungsversionen ansehen. Drei Versionen einer Datei spielen beim Konflikt eine Rolle: die Version auf dem aktuellen Branch (*ours*), die Version auf dem anderen Branch (*theirs*) und der letzte gemeinsame Vorgänger dieser beiden Branches (*ancestor*). Der `show`-Befehl kann die Versionen abrufen:

```
> git show :1:picture.png    >ancestor.png
> git show :2:picture.png    >ours.png
> git show :3:picture.png    >theirs.png
```

**Tipp: Ignore Whitespace**

Merge- und Diff-Tools zeigen in der Regel auch Änderungen am Whitespace an. Wenn ein Entwickler beispielsweise Tabs durch Spaces ersetzt hat, werden alle Zeilen markiert, obwohl sich inhaltlich vielleicht gar nichts getan hat. Die Tools bieten in der Regel eine Option, um Whitespace-Änderungen zu ignorieren. Diese sollte man nutzen.

**Tipp: Auto-Formatter**

Noch besser ist es natürlich, wenn alle Entwickler denselben automatischen Formatierer für Quelltexte nutzen. Damit sind Formatierungen als Fehlerquelle für Konflikte weitgehend ausgeschlossen.

Es geht natürlich auch ohne Tool, wie die folgende Anleitung zeigt:

**Schritt für Schritt****Merge manuell durchführen****1a. Betroffene Dateien editieren**

Für jede Konfliktstelle überlegt man sich, welche Variante man übernehmen möchte. Danach entfernt man den Rest und die Konfliktmarkierungen mit einem Texteditor. Für Binärdateien geht das nicht, da ist nur Schritt 1b möglich.

**1b. --ours oder --theirs übernehmen**

Alternativ kann man mit dem `checkout`-Befehl auch nur die eigenen (oder die fremden) Versionen der Dateien komplett übernehmen:

```
> git checkout --theirs tests/
```

**2. Änderungen anmelden**

```
> git add .
```

**3. Commit**

```
> git commit
```

Es kann vorkommen, dass man ein Merge versehentlich ausführt oder Fehler bei der Konfliktauflösung macht. Dann sollte man nicht direkt weiterarbeiten, sondern das Merge explizit abbrechen, damit keine Reste der Zusammenführung im Workspace verbleiben und damit Git das nächste Commit nicht als *Merge-Commit* kennzeichnet.

Ein Merge kann mit `--abort` abgebrochen werden:

*Tipp: Merge abbrechen*

```
> git merge --abort
```

## Und was ist mit den inhaltlichen Konflikten?

Problematisch sind die inhaltlichen Konflikte, denn die kann Git nicht erkennen und erst recht nicht automatisch lösen. Gefährlich dabei ist, dass der `merge`-Befehl bei einem rein inhaltlichen Konflikt ein gültiges *Merge-Commit* erzeugt.

**Achtung!** Auch wenn alle zusammengeführten Versionen korrekt sind und Git keine Bearbeitungskonflikte gemeldet hat, kann das *Merge-Commit* defekt sein!

Wer sich davor schützen will, inhaltlich inkonsistente Softwarestände auszuliefern, sollte mehr tun:

**Absicherung durch automatisierte Tests:** Wenn diese Tests regelmäßig ausgeführt werden und eine gute Abdeckung haben, entdeckt man inhaltliche Konflikte oft schnell.

*Integration mit Jenkins* → Seite 263

**Assertions, Pre- und Postconditions:** Je mehr Erwartungen explizit geprüft werden, desto früher erkennt man Probleme.

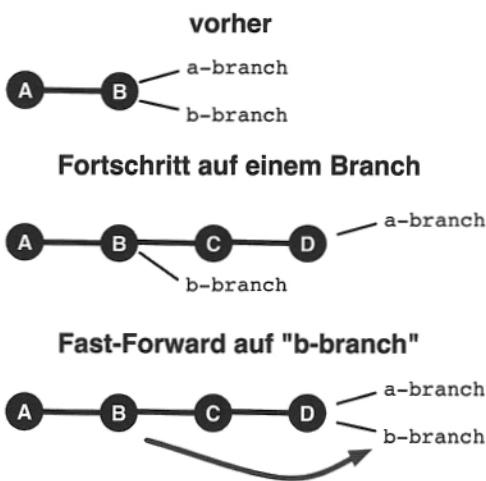
**Klare Schnittstellen, lose Kopplung:** Je sauberer die Architektur in diesem Punkt ist, desto weniger wahrscheinlich sind überraschende Seiteneffekte durch Codeänderungen an unterschiedlichen Stellen.

**Statische Typprüfungen:** Wenn die Programmiersprache dies unterstützt, werden Probleme durch Signaturänderungen bereits zur Compile-Zeit erkannt.

Es ist übrigens zulässig, mehrere zu integrierende *Branches* beim `merge`-Befehl anzugeben. Dies nennt man dann ein *Octopus-Merge*.

### 8.3 Fast-Forward-Merges

Abb. 8-5  
»Fast-Forward-Merge«

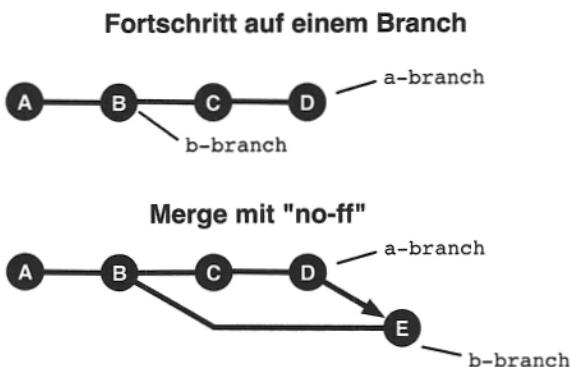


Nicht selten tritt Folgendes ein: Es gibt mehrere *Branches*, aber nur auf einem wurde weitergearbeitet. In Abbildung 8-5 ist auf a-branch weiterentwickelt worden, auf dem b-branch ist nichts passiert. Wenn man auf dem Branch b-branch ein Merge mit a-branch ausführt, hat Git es einfach: Es setzt nur den Zeiger für b-branch hoch. Es entsteht kein *Merge-Commit*. Man nennt dies ein *Fast-Forward-Merge*.

```
> git checkout b-branch
> git merge a-branch
Updating 9d4caed..9332b08
Fast-forward
  foo.txt    |    2 +-
  1 files changed, 1 insertions(+), 1 deletions(-)
```

Der Vorteil von *Fast-Forward-Merges* besteht darin, dass die Versionshistorie einfach und linear bleibt. Der Nachteil ist, dass man der Historie später nicht mehr ansieht, dass eine Zusammenführung stattgefunden hat. Deshalb verwenden wir in einigen Workflows in diesem Buch lieber die Option `--no-ff`, um zu erzwingen, dass beim *Merge* ein neues Commit entsteht (Abbildung 8-6).

```
> git merge --no-ff a-branch
```



**Abb. 8-6**  
Variante ohne  
»Fast-Forward-Merge«

## 8.4 First-Parent-History

Ein *Merge-Commit* hat in der Regel zwei Vorgänger<sup>2</sup>. Im folgenden Beispiel sind es die Commits ed1c70e und f1d55be:

```
> git log --merges
commit 7f3eae07c42df05f894fdd4754e38ab9e66a5051
Merge: ed1c70e f1d55be
Author: ...
```

Das erste angegebene Commit nennt man den *First-Parent*, im obigen Beispiel ist es ed1c70e. Es ist jenes Commit, das HEAD war, als das Merge durchgeführt wurde. Man sieht also, wo das *Merge* stattgefunden hat.

Wenn alle Entwickler auf demselben Branch entwickeln, dann ergibt es sich eher willkürlich, wann und wo Merges durchgeführt werden. In diesem Fall ist es eher uninteressant, welches Commit der *First-Parent* ist.

Anders sieht es aus, wenn mit *Feature-Branches* entwickelt wird. Dann wird ein Feature nach dem anderen auf dem Feature-Branch integriert, und es entsteht auf dem Integrations-Branch (im Beispiel master) eine Folge von *Merge-Commits* (Abbildung 8-7), deren *First-Parent* immer das *Merge-Commit* des vorigen Features ist.

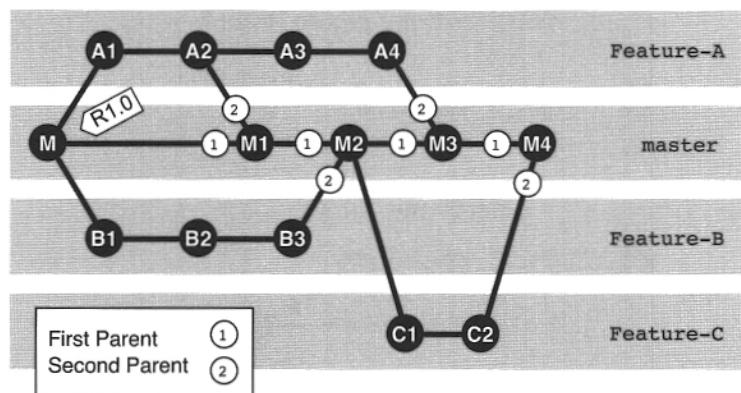
Wenn man vom HEAD aus der Historie über die *First-Parents* rückwärts folgt, erhält man einen Überblick über die Feature-Integrationen. Diese Folge nennt man *First-Parent-History*. Die Option `--first-parent` des log-Befehls zeigt sie an:

**Gemeinsam auf einem Branch entwickeln**  
→ Seite 135

**Mit Feature-Branches entwickeln** → Seite 143

<sup>2</sup> Bei den sogenannten Octopus-Merges kann es auch mehr als zwei Vorgänger geben.

**Abb. 8-7**  
First-Parent-History



```
> git log --first-parent --oneline R1.0..master
```

```
7f3eae0 Merge branch 'Feature-C' Fertig (M4)
ed1c70e Merge branch 'Feature-A' Fertig (M3)
eeb6ec2 Merge branch 'Feature-B' Fertig (M2)
8ce3213 Merge branch 'Feature-A' Teillieferung (M1)
```

Das Schöne an der First-Parent-History ist, dass sie eine verdichtete Darstellung der Historie liefert. Man sieht, welche Features integriert wurden, ohne jedes einzelne Commit der Feature-Branches betrachten zu müssen.

Achtung! Das funktioniert nur, wenn man auf dem Integrations-Branch keine *Fast-Forward-Merges* durchführt. Andernfalls würden einzelne Commits der Feature-Branches direkt in der *First-Parent-History* des master landen.

Achtung! Außerdem sollte man auf dem Integrations-Branch (hier: master) keine internen Merges durchführen, sondern darauf achten, dass die Features alle nacheinander integriert werden, sodass eine lineare Historie von *Feature-Merges* entsteht.

## 8.5 Knifflige Merge-Konflikte

Die meisten Zusammenführungen erledigt Git automatisch oder mit geringer manueller Nachhilfe. Wenn zwei Branches sich stark unterschiedlich entwickelt haben, kann es aber zu kniffligen Konflikten kommen.

Wir sprechen in diesem Abschnitt immer nur von zwei Branches. Falls Sie bei einem Octopus-Merge auf Probleme stoßen, sollten Sie den Merge abbrechen und versuchen, Branch für Branch vorzugehen.

Das Wichtigste ist, erst einmal Informationen zu sammeln, um zu verstehen, was auf den Branches passiert ist. Hierbei hilft die ..-Notation beim log-Befehl. Zum Beispiel zeigt a..b jene Commits aus dem

*Tipp: Octopus-Merges schrittweise auflösen*

*Tipp: Historie lesen!*

Branch b, die im Branch a nicht enthalten sind. Man kann sich zeigen lassen, was »wir« (auf dem aktuellen Branch) getan haben, also welche von »unseren« Commits im anderen Branch noch nicht enthalten sind.

```
> git log MERGE_HEAD..HEAD
```

Umgekehrt kann man zeigen, was »die anderen« gemacht haben:

```
> git log HEAD..MERGE_HEAD
```

Eine grafische Darstellung der Verzweigung kann ebenfalls nützlich sein:

```
> git log --graph --oneline --decorate HEAD MERGE_HEAD
```

Mit der Option --merge des log-Befehls kann man die Anzeige auf *Merge-Commits* einschränken:

```
> git log --merge
```

Auch das Vergleichen der Ausgangsversion mit den Spitzen der Branches kann nützlich sein. Hierzu benötigt man die *Merge-Base*, d. h. den gemeinsamen Vorgänger der Branches im Merge:<sup>3</sup>

```
> git merge-base HEAD MERGE_HEAD
ed3b1832c48b359111d00bddb071c42ba6f38324
> git diff --stat ed3b18 HEAD      % Unsere Änderungen
> git diff --stat ed3b18 MERGE_HEAD % Fremde Änderungen
```

Man kann auch den difftool-Befehl verwenden, wenn man ein grafisches Tool anstelle einer Textausgabe haben möchte.

Jetzt sieht man, welche Entwickler am Konflikt beteiligt sind. Am besten ist es, alle an einen Tisch zu holen. Dann kann jeder dafür sorgen, dass seine Änderungen bei der Zusammenführung korrekt berücksichtigt werden.

Falls »die anderen« nicht erreichbar sind, wird es schwieriger, weil man sich auf dem »anderen« Branch ja nicht auskennt. Technisch betrachtet ist ein Merge eine symmetrische Operation. Im Kopf hat man oft eine asymmetrische Sicht. Man stellt sich die Frage: »Wie bekomme ich die Änderungen der anderen in meinen Code?« Manchmal hilft es, die Frage umzukehren. Nehmen Sie den »anderen« Versionsstand als Ausgangspunkt und überlegen Sie, wie Sie »Ihre« Änderungen dort einbringen. Manchmal hilft der Wechsel des Blickwinkels.

<sup>3</sup> In seltenen Fällen kann der gemeinsame Vorgänger nicht eindeutig bestimmt werden. Dann liefert der merge-base-Befehl mehrere Commits.

Tipp: »die anderen«  
dazuholen

Tipp: Die Denkrichtung  
wechseln hilft  
manchmal.

## Egal, es wird schon irgendwie gehen

Unter Zeitdruck können Merge-Tools Sie dazu verleiten, willkürlich auf die eine oder andere Variante zu klicken, deren Code »irgendwie besser« aussieht. Widerstehen Sie dieser Versuchung. Wenn man nach Analyse von Diffs und Logs und nach der Zuhilfenahme der »anderen« Versionen immer noch unsicher ist, wie die Konflikte aufzulösen sind, dann sollte man den Merge abbrechen. Ein paar mögliche Strategien sind dann:

**Branch restrukturieren:** Die sauberste Lösung besteht wohl darin, einen der Branches durch Refactoring und mithilfe von interaktivem Rebasing aufzuräumen. Das ist allerdings viel Arbeit.

**Merge in kleinen Schritten:** Wenn einer der beiden Branches aus feingranularen Commits besteht, kann man Commit für Commit vorgehen. Der Vorteil besteht darin, dass bei kleinen Commits Konflikte meist leicht aufzulösen sind. Das kann zäh sein, wenn es um viele Commits geht. Es empfiehlt sich auf jeden Fall, einen lokalen Branch hierfür anzulegen.

**Verwerfen und Cherry-Pick:** In manchen Fällen ist es besser, die Änderungen des schlechteren Branch nicht zu übernehmen. Einzelne Verbesserungen kann man dann mit dem `cherry-pick`-Befehl übernehmen.

**Raten und testen:** Wenn sich die betroffene Funktionalität gut testen lässt, kann man natürlich versuchen, bei der Konfliktauflösung zu raten und das Ergebnis so lange zu verbessern, bis die Tests grün sind. Sagen Sie dann aber bitte nicht, wir hätten das empfohlen.

## 8.6 Zusammenfassung

**Merge:** Die Zusammenführung von Zweigen im Commit-Graphen nennt man Merge.

**Merge-Commit:** Das Ergebnis des `merge`-Befehls ist ein sogenanntes Merge-Commit.

**3-Wege-Merge:** Git nutzt den Commit-Graphen, um beim Merge den letzten gemeinsamen Vorfahren zu finden. Dann führt Git die Änderungen, die auf dem einen Branch seit dem Vorfahren erfolgt sind, mit den Änderungen zusammen, die auf dem anderen Branch vorgenommen wurden. Solange die Änderungen an verschiedenen Codestellen passiert sind, führt Git die Versionen automatisch zusammen.

**Konflikt:** Jene Stellen, die Git nicht automatisch zusammenführen kann, etwa weil dieselbe Zeile auf unterschiedliche Weise geändert wurde, nennt man Konflikt.

**Inhaltlicher Konflikt:** Es kommt immer wieder vor, dass Änderungen an verschiedenen Stellen stattfinden, aber trotzdem inhaltlich nicht zusammenpassen. Git kann solche inhaltlichen Konflikte nicht erkennen. Das Projekt muss eigene Vorkehrungen treffen, zum Beispiel automatische Tests, um sich davor zu schützen.

**Fast-Forward-Merge:** Es kommt recht häufig vor, dass einer der Branches beim Merge ein Vorfahr des anderen ist. In diesem Fall setzt Git einfach den Branch-Zeiger weiter. Es ist kein Merge-Commit notwendig.



## 9 Mit Rebasing die Historie glätten

Viele Verzweigungen in einer *Commit-Historie* sind unübersichtlich. Git ermöglicht es, die Historie zu begradigen. Das wichtigste Werkzeug hierfür ist der `rebase`-Befehl, der Folgen von Commits an andere Stellen im *Commit-Graphen* verschieben kann. Dies will man dann tun,

- wenn man Commits versehentlich auf dem falschen Branch ausgeführt hat. Typisch wäre etwa ein Bugfix, den man auf der Entwicklungslinie (`develop`) abgezweigt hat, der aber eigentlich als Hotfix von der Release-Linie (`master`) hätte abgezweigt werden sollen.
- wenn mehrere Entwickler intensiv an der gleichen Software arbeiten und ihre Änderungen häufig integrieren. Ohne *Rebasing* entsteht dann eine Historie aus vielen kleinen Verzweigungen und Zusammenführungen (eine sogenannte Diamantenkette). Mit dem `rebase`-Befehl kann man stattdessen eine glatte lineare Historie herstellen.

**Periodisch Releases durchführen**

→ Seite 185

**Gemeinsam auf einem Branch entwickeln**

→ Seite 135

### 9.1 Das Prinzip: Kopieren von Commits

Das Prinzip beim *Rebasing* ist einfach: Git nimmt eine Folge von Commits, die verschoben werden sollen, und spielt diese auf dem Ziel-Branch in genau derselben Reihenfolge erneut ein. Dabei entsteht für jedes der ursprünglichen Commits eine getreue Kopie mit den gleichen Änderungen (Changeset), dem gleichen Autor, Zeitpunkt und Kommentar.

**Achtung!** Es sieht auf den ersten Blick so aus, als ob Git beim *Rebasing* Commits verschieben würde. Tatsächlich sind die »verschobenen« Commits immer neue Commits und haben auch einen anderen Commit-Hash. Dies ist dann von Bedeutung, wenn von Original-Commits bereits weitere Branches abgezweigt wurden. Mehr dazu folgt ab Seite 84.

*Rebasing erzeugt immer neue Commits!*

**Achtung!** Da die neuen Commits an einer anderen Stelle im *Commit-Graphen* eingespielt werden, kann es natürlich zu Konflikten kommen, weil die Änderungen dort nicht passen. Solche Änderungen müssen dann wie Merge-Konflikte manuell aufgelöst werden.

**Bearbeitungskonflikte**

→ Seite 70

## »Diamantenketten« vermeiden

**Gemeinsam auf einem Branch entwickeln**  
→ Seite 135

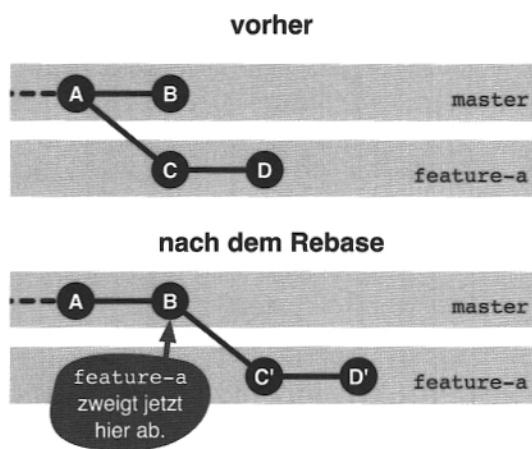
Wenn mehrere Entwickler gemeinsam an einer Software arbeiten und häufig die Änderungen integrieren, entsteht eine *Commit-Historie* aus Verzweigungen und Zusammenführungen, die an eine Kette aus Diamanten erinnert. Mit dem `rebase`-Befehl kann man stattdessen eine – inhaltlich gleichwertige – lineare Geschichte erzeugen.

**Abb. 9-1**  
*Diamantenkette*



Das Beispiel in Abbildung 9–2 zeigt, wie das geht: Ein Branch `feature-a` wurde vom `master` abgezweigt und hat zwei Commits `c` und `d`. Auch auf dem `master` wurde weiterentwickelt: Commit `b`.

**Abb. 9-2**  
*Einfaches Rebasing*



Man könnte nun die Änderungen mit `git merge master` zusammenführen. Nur hätte man dann wieder einen neuen »Diamanten« in der Kette. Stattdessen kann man mit dem `rebase`-Befehl die Historie wieder »glatt ziehen«. Als Parameter gibt man, wie beim `merge`-Befehl, den Branch an, dessen neueste Änderungen man in den aktiven Branch übernehmen möchte:

```
> # Branch "feature-a" ist aktiv
> git rebase master
```

Git tut nun Folgendes:

**Welche Commits?** Es wird ermittelt, welche Commits aus dem aktiven Branch `feature-a` noch nicht im Ziel-Branch `master` enthalten sind. Im Beispiel sind dies `C` und `D`.

**Wohin?** Git bestimmt das Ziel-Commit. Die neuen Commits sollen vom Kopf des Ziel-Branch `master` abzweigen. Im Beispiel ist das Ziel-Commit `B`.

**Kopieren der Commits:** Ausgehend vom Ziel-Commit werden alle oben ermittelten Commits noch einmal neu, aber mit den gleichen Änderungen ausgeführt. Im Beispiel entstehen dabei die Commits `C'` und `D'`.

**Aktiven Branch auf Kopie umsetzen:** Der aktive Branch wird auf das oberste kopierte Commit umgesetzt. Im Beispiel wird der Branch `feature-a` also auf das Commit `D'` umgesetzt.

Oftmals muss man den `rebase`-Befehl gar nicht direkt aufrufen. Man kann stattdessen die Option `--rebase` des `pull`-Befehls zum Abgleich mit entfernten Repositorys nutzen.

**Anmerkung:** Die alten Commits `C` und `D` sind übrigens immer noch im Repository vorhanden. Sie sind nur nicht mehr direkt sichtbar, da der Branch `feature-a` jetzt auf `D'` verweist. Kein Branch verweist mehr auf `C` und `D`. Über die Commit-Hashes sind sie weiterhin erreichbar. Erst nach einer *Garbage Collection* mit dem `gc`-Befehl werden sie aus dem Repository verschwinden.

*Tipp: Pull und Rebase in einem Schritt*

*Original-Commits werden nicht gelöscht!*

## 9.2 Und wenn es zu Konflikten kommt?

Genau wie beim `merge`-Befehl kann es auch beim *Rebasing* zu Konflikten kommen, wenn Änderungen nicht zusammenpassen. Einen wichtigen Unterschied gibt es aber doch: Beim *Merge* entsteht ein einzelnes Commit mit dem Ergebnis der Zusammenführung. Beim *Rebasing* hingegen werden mehrere Commits Schritt für Schritt neu erzeugt. Wenn alles glattgeht, sieht der Inhalt des letzten Commits genauso aus wie jener, den der `merge`-Befehl erzeugt hätte, denn Git verwendet für beide Befehle die gleichen Algorithmen zur Konfliktbehandlung. Wenn der `rebase`-Befehl aber auf einen Konflikt stößt, wird der Vorgang unterbrochen. Die Dateien werden, wie beim *Merge*, mit Konfliktmarkierungen versehen. Man kann die Dateien manuell oder mit einem *Merge-Tool* bereinigen. Danach fügt man sie dem Staging-Bereich hinzu. Dann kann man den `rebase`-Befehl mit der Option `--continue` fortfahren lassen:

```
> git add foo.txt
> git add bar.txt
> git rebase --continue
```

*Tipp: Abbrechen oder überspringen*

Mit der Option `--abort` kann man den *Rebasing*-Vorgang auch ganz abbrechen, und mit `--skip` kann man das konfliktbehaftete Commit überspringen. Es wird dann einfach weggelassen, d. h., seine Änderungen tauchen in dem neuen Branch nicht auf.

```
> git rebase --abort
```

**Achtung!** Anders als beim Merge ist bei einer Unterbrechung des Rebasing erst ein Teil der zu kopierenden Commits angewendet worden.

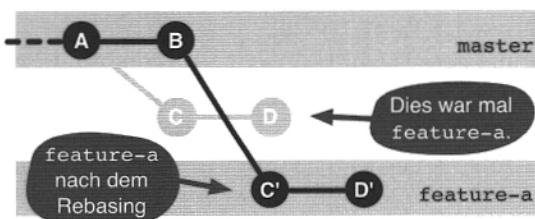
**Anmerkung:** Der Ursprung beim `rebase`-Befehl muss nicht unbedingt ein Branch sein. Es kann auch ein beliebiges Commit angegeben werden. Somit lässt sich sehr genau steuern, welche Commits verschoben werden.

### 9.3 Was passiert mit den ursprünglichen Commits nach dem Rebasing?

Beim *Rebasing* werden Commits kopiert. Die Originale (im Beispiel c und d) bleiben zunächst erhalten. Im Normalfall, d. h., wenn von diesen Commits keine weiteren Branches abgezweigt wurden, wird die nächste *Garbage Collection* (`gc`-Befehl) sie einfach aus dem Repository entfernen.

Abb. 9-3

Alter und neuer Branch nach dem Rebasing



## 9.4 Empfehlungen zum Rebasing

Das *Rebasing* ist eine sehr hilfreiche Technik, um die Historie eines Repositorys verständlicher zu gestalten. Aber es gibt Risiken, die man kennen und verstehen sollte, wenn man erfolgreich damit arbeiten möchte:

- *Rebasing* kopiert Commits.
- *Rebasing* erzeugt Versionen der Software, die es nie im Workspace eines Entwicklers gab.

Letzteres ist klar: Die durch *Rebase* entstandenen Versionen sind synthetisch, sie wurden nie gebaut und getestet. Das Hilfsmittel liegt auf der Hand: Automatisiertes Bauen und Testen.

Etwas kniffliger hingegen ist die Sache mit den kopierten Commits.

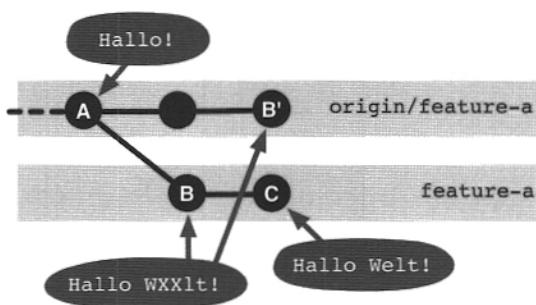
**Integration mit Jenkins**  
→ Seite 263

### Warum können kopierte Commits problematisch sein?

Ein Beispiel dafür zeigt Abbildung 9–4. Es ist Folgendes passiert:

1. Commit A enthält eine Zeile »Hallo!«
2. Entwickler Linus ändert in Commit B die Zeile zu »Hallo WXXlt!«
3. Entwickler Junio führt eine *Rebase* und ein Push durch. Es entsteht ein Commit B' mit der gleichen Änderung: »Hallo WXXlt!«
4. Entwickler Linus bemerkt den Tippfehler und erstellt ein Commit C mit der Zeile »Hallo Welt!«

**Abb. 9–4**  
Probleme mit kopierten Commits



Jetzt führt Linus, nichts ahnend von Junios Rebasing-Machenschaften, ein Pull durch, und es entsteht eine Konfliktmarkierung, die etwa so aussieht:

```
<<<<< HEAD
Hallo Welt!
=====
Hallo WXXlt!
>>>> origin/master
```

Oben ist noch alles in Ordnung. Linus sieht erwartungsgemäß die von ihm korrigierte Zeile, die er selber durchgeführt hat. In der unteren Hälfte sieht es aber so aus, als hätte jemand den bereits behobenen Fehler erneut eingebaut. Linus versucht herauszufinden, wer der Schuldige ist, macht sich auf die Suche und stößt schließlich auf das Commit B, dessen Autor er selber ist. Erst wenn man im Log mit `--pretty=fuller` alle Details anzeigt, erkennt er mehr:

```
commit 7206283c7b00d2402eb73e6cf8a1aecb030e755git 1
Author: LINUS <l@mail.org>
AuthorDate: Fri Oct 28 17:36:11 2016 +0200
Commit: JUNIO <j@mail.org>
CommitDate: Fri Oct 28 19:36:32 2016 +0200
```

Durch das Kopieren (B zu B' in Abb. 9–4) wird das 3-Wege-Verfahren beim Merge ausgetrickst. Dieses basiert auf der Annahme, dass alle Gemeinsamkeiten bereits im letzten gemeinsamen Vorgänger (im Beispiel A) enthalten sind und dass alles, was danach kommt, zusammenzuführende Neuerungen sind. Das Verfahren »sieht«, dass ein Entwickler `WXXlt` geschrieben hat und ein anderer `Welt`, und meldet einen Konflikt. Es »sieht« nicht, dass das `WXXlt` im Commit B in der Historie von C bereits enthalten ist.

### **Empfehlungen, um Ärger mit kopierten Commits zu vermeiden**

Solche Situationen fressen im Projektalltag viel Zeit und führen oft zu Fehlern, weil bei solchen Merges leicht Fehlentscheidungen getroffen werden. Wichtig ist, dass Kopien nach einem Rebase erhalten bleiben, Originale aber »entsorgt« werden.

Der parameterlose Aufruf des `rebase`-Befehls ist in der Regel harmlos, weil er nur lokale Commits betrifft, die noch nicht übertragen wurden (genauer: nicht im Upstream-Branch enthalten sind).

```
> git rebase
```

Achtung! Die betroffenen Commits sollten nicht in andere Repositories übertragen werden sein, es sollten keine weiteren Branches davon abzweigen und es sollten auch keine Tags darauf gesetzt sein. Denn in all diesen Fällen bleiben Referenzen auf die Originale erhalten. Deshalb empfiehlt es sich auch nicht, Feature-Banches von anderen Feature-Banches abzweigen zu lassen.

Die Möglichkeiten für sinnvolles Rebasing hängen vom gewählten Workflow ab.

Arbeiten die Entwickler gemeinsam auf dem master-Branch, ist im Grunde genommen nur rein lokales *Rebasing* sinnvoll.

Beim Arbeiten mit Feature-Banches kann man die Möglichkeiten zum *Rebasing* erweitern, wenn man sich auf die Regel einigt, dass Feature-Banches privat sind und dem Entwickler gehören, der das Feature bearbeitet. Dann kann dieser innerhalb seines Feature-Branch nach Belieben *Rebases* durchführen und kann diese auch im zentralen Repository des Projekts sichern. Empfehlenswert ist es in diesem Falle, den master-Branch (und ggf. weitere Integrationsbranches) vor *Rebasing* zu schützen. Repository-Verwaltungen, wie etwa GitHub, bieten entsprechende Optionen.

Beim Arbeiten mit Forks ist jeder Entwickler König in seinem Fork und kann dort tun und lassen, was er will. Der Maintainer des Main-Repository wird den master-Branch mit *Rebasing*-Schutz versehen und keinen Pull-Request akzeptieren, der Commits, die bereits integriert wurden, verändert.

## Noch ein paar Tipps

Bei Merge-Konflikten aufgrund kopierter Commits kann es mitunter helfen, die Zusammenführung per *Rebase* (statt Merge) zu versuchen, denn Git macht die Zusammenführung dann Commit für Commit und erkennt dabei oft (aber nicht immer), wenn Änderungen doppelt vorkommen, und wendet diese nicht erneut an. Für B in B' in Abb. 9-4 hätte das sehr wahrscheinlich geholfen.

Falls es doch mal passiert ist und auf einem Integrationsbranch, z. B. master, ein *Rebase* durchgeführt wurde, sollte man alle Entwickler benachrichtigen und empfehlen, das Repository neu zu klonen. Ist das Repository groß, kann man auch gezielt den betroffenen Branch zurücksetzen:

```
> git fetch  
> git checkout master  
> git reset --hard origin/master
```

**Gemeinsam auf einem Branch entwickeln**

→ Seite 135

**Mit Feature-Banches entwickeln** → Seite 143

**Mit Forks entwickeln**

→ Seite 163

*Tipp: Schwierige Konflikte per Rebase lösen*

*Tipp: Branch zurücksetzen*

## 9.5 Cherry-Picking

Es gibt eine weitere Möglichkeit, um Commits zu kopieren, und zwar den `cherry-pick`-Befehl. Man gibt dabei an, welches Commit man haben möchte, und Git erstellt auf dem aktuellen Branch ein neues Commit mit den gleichen Änderungen (Changeset) und Metainformationen.

```
> git cherry-pick 23ec70f6b0
```

Folgendes sollte man über das Cherry-Picking wissen:

- `cherry-pick` arbeitet kontextfrei, ohne die Historie zu berücksichtigen. `merge` und `rebase` können Änderungen oft auch nach Umbenennungen und Verschiebungen noch richtig einordnen; `cherry-pick` kann das nicht.
- Cherry-Picking wird gelegentlich genutzt, um kleine Bugfixes auf verschiedene Release-Versionen zu übertragen.
- Ein anderer Anwendungsfall ist das Übertragen nützlicher Änderungen aus einem Feature-Branch, den man verwerfen möchte.
- **Achtung:** Cherry-Picking kann zu den oben genannten Problemen mit kopierten Commits führen.

## 9.6 Zusammenfassung

**Rebasing:** Git kann Commits an andere Stellen im Commit-Graphen kopieren. Dabei bleiben die Änderungen und die Metainformationen (Autor, Zeitpunkt) gleich, aber es gibt einen neuen Commit-Hash. Mit dem `rebase`-Befehl kann man den Commit-Graphen auf vielfältige Weise umbauen.

**Nur vor dem Push:** Im Normalfall sollte man den `rebase`-Befehl nur auf Commits anwenden, die noch nicht in andere Repositorys übertragen wurden. Es könnte sonst später zu überraschenden Merge-Konflikten kommen.

**Historie glätten:** Wenn man Konflikte beim parallelen Entwickeln mit dem `merge`-Befehl löst, entsteht eine Historie mit vielen Verzweigungen und Zusammenführungen. Wenn man statt des Merge immer ein Rebasing durchführt, kann man eine lineare Historie erzeugen.

**Konflikte beim Rebasing:** Git spielt die kopierten Commits Stück für Stück wieder ein. Kommt es zu einem Konflikt, weil die Änderungen nicht zum Workspace passen, wird der Vorgang unterbrochen. Der Entwickler kann, ähnlich wie beim Merge, den Konflikt manuell auflösen und das Rebasing fortsetzen.

**rebase --onto:** Mit dieser Variante ist es möglich, einen Branch an eine völlig andere Stelle im Commit-Graphen zu verschieben.

# 10 Repositorys erstellen, klonen und verwalten

## 10.1 Ein Repository erstellen

Mit dem `init`-Befehl erstellt man ein neues Git-Repository. Git legt dazu ein Verzeichnis als *Workspace* an, in dem später die Dateien des Projekts liegen werden (im Beispiel: `myproject`).

```
> git init myproject
```

## 10.2 Das Repository-Layout

Im *Workspace*-Verzeichnis legt Git dann ein Unterverzeichnis `.git` für das Repository an. Dort werden versionierte Inhalte und die Metadaten dazu abgelegt. Es enthält:

**HEAD** (Datei): Zeigt an, welcher Branch gerade aktiv ist. Im Betrieb kommen meist noch ein paar weitere ähnliche Dateien dazu, z. B. `FETCH_HEAD` oder `ORIG_HEAD`.

**config** (Datei): Enthält die projektspezifische Konfiguration. Diese kann mit `git config local` bearbeitet werden.

**description** (Datei): Kann eine Beschreibung enthalten.

**hooks** (Verzeichnis): Falls man ein wenig Scripting mit Bash beherrscht, kann man die Funktionalität von Git leicht durch sogenannte *Hooks* erweitern. Hooks sind Skripte, die in diesem Verzeichnis abgelegt werden und von Git zu bestimmten Zeitpunkten als Callback aufgerufen werden. Zum Beispiel könnte man in einem Skript `prepare-commit-msg` programmieren, dass jede Commit-Meldung automatisch um eine Task-Nummer ergänzt wird.

**objects** (Verzeichnis): Dies ist die *Object Database* (siehe Kapitel 6). Hier speichert Git Commits und Dateiinhalte.

**refs** (Verzeichnis): Alle Branches, Remote-Tracking-Branches und Tags werden hier abgelegt.

*Tipp: In .git/logs protokolliert Git Änderungen an Branches.*

**logs** (Verzeichnis): Auch *Reflog* genannt. Sofern `git config core.logAllRefUpdates` eingeschaltet ist, werden hier alle Änderungen an Branches protokolliert. Das sollten Sie sich merken. Es kann sehr hilfreich sein, um Branches wiederherzustellen, wenn man mal etwas kaputt gemacht hat.

**info** (Verzeichnis): weitere Informationen über das Repository

Mehr darüber erfahren Sie mit:

```
> git help gitrepository-layout
```

### 10.3 Bare-Repositories

Ein Repository, das nur zum Austausch und nicht zum Entwickeln benutzt wird, benötigt keinen *Workspace*. Mit der Option `--bare` erzeugt man ein Verzeichnis, das nur die oben beschriebene Repository-Struktur enthält, die man sonst im `.git`-Unterverzeichnis vorfindet. Man nennt das dann ein *Bare-Repository*.

```
> git init --bare myserver-repo.git
```

Die Namenskonvention für *Bare-Repository*-Verzeichnisse ist, den Namen auf `.git` enden zu lassen.

### 10.4 Vorhandene Dateien übernehmen

*Ein Projekt aufsetzen*

→ Seite 123

Gibt man keinen Namen für das neue Repository an, wird im aktuellen Verzeichnis ein `.git`-Verzeichnis angelegt. Das ist sinnvoll, wenn man ein bereits vorhandenes Projekt unter Git-Versionsverwaltung stellen möchte.

**Achtung!** Der `init`-Befehl fügt weder Dateien zum Repository hinzu noch erstellt er ein Commit. Vergessen Sie also nicht, die gewünschten Dateien auch zu versionieren.

```
> git init
> git status
> vi .gitignore      # eventuell .gitignore erstellen
> git add --all      # oder nur, was Sie hinzufügen möchten
> git commit -m "initialen projektstand übernehmen"
```

Falls das Projekt bereits eine Historie in einer anderen Versionsverwaltung hat und Sie es zu Git migrieren möchten, finden Sie weiter hinten den Workflow »Ein Projekt nach Git migrieren« (Seite 247).

## 10.5 Ein Repository klonen

Das Klonen von Repositorys spielt in Git eine große Rolle. Es gibt viele Gründe dafür, zu klonen:

- Jeder Entwickler benötigt mindestens einen eigenen Klon, um überhaupt mit Git arbeiten zu können.
- Oft wird ein Klon als zentrales Repository genutzt, um den »offiziellen« Stand des Projekts darzustellen.
- Bei einer Multisite-Entwicklung wird jeder Standort seinen Hauptklon haben, der regelmäßig mit den Hauptklonen an anderen Standorten abgeglichen wird.
- Für unabhängige Entwicklungen, die eine andere Richtung nehmen als das Hauptprojekt (z. B. wenn man radikale Umbauten vorhat), verwendet man oft einen eigenen Klon. Einen solchen Klon nennt man auch »Fork«.
- Wenn man knifflige Arbeiten am Repository durchführt, die das Projekt oder das Repository beschädigen könnten, ist es oft sinnvoll, einen separaten Klon dafür zu nehmen.
- Ein Klon kann auch als Backup verwendet werden.

Die Handhabung des `clone`-Befehls ist einfach. Man gibt den Ort des Original-Repository als Parameter an, und Git erstellt ein Unterverzeichnis mit dem geklonten Repository im aktuellen Arbeitsverzeichnis.

Normalerweise führt Git nach dem Klonen sofort einen Checkout in den Workspace durch. Mit der Option `--bare` kann man ein Repository ohne Workspace erzeugen lassen. Dies ist nützlich für Repositorys auf Serverseite, auf denen kein Entwickler direkt arbeitet.

Der Workflow »Ein Projekt aufsetzen« (Seite 123) beschreibt unter anderem, wie man ein zentrales Repository für ein Projekt aufsetzen kann.

## 10.6 Wie sagt man Git, wo das Remote-Repository liegt?

Wenn das andere Repository lokal vorhanden ist, gibt man einfach den Pfad des Verzeichnisses an, in dem das Repository liegt, z. B. `/Users/stachi/git-buch.git`. Das Klonen eines lokalen Repositorys sieht beispielsweise so aus:

```
> git clone /Users/stachi/git-buch.git
```

Wenn man es mit mehreren Repositorys aus unterschiedlichen Quellen zu tun hat, ist die URL-*Form* jedoch klarer, bei der der Protokolltyp (hier `file`) vorangestellt wird:

```
> git clone file:///Users/stachi/git-buch.git
```

Neben `file` werden weitere Protokolle für den Zugriff auf nicht lokale Repositorys angeboten. Am häufigsten wird das *ssh-Protokoll* verwendet, weil es eine sichere Authentifizierung ermöglicht und die Infrastruktur dafür oft bereits vorhanden ist, wenn man mit Linux- oder Unix-Servern arbeitet.

```
> git clone ssh://stachi@server.de:git-buch.git
```

Darüber hinaus wird der Zugriff auch über die `http`-, `https`-, `ftp`-, `ftps`- und `rsync`-Protokolle oder über ein proprietäres Protokoll namens `git` ermöglicht.

## 10.7 Kurznamen für Repositorys: Remotes

Wenn man öfter auf einen anderen *Klon* zugreifen möchte, kann man ihn mit dem `remote`-Befehl unter einem Kurznamen (genannt: *Remote*) registrieren lassen.

```
> git remote add klon file:///tmp/git-buch-klon.git
```

Man kann dann bei Git-Befehlen den Kurznamen, im Beispiel `klon`, anstelle der *Repository-URL* verwenden.

Wenn ein Repository geklont wird, trägt Git den Pfad für das *Original-Repository* automatisch als `origin` ein. Ruft man den `remote`-Befehl mit der Option `--verbose` auf, so listet Git die Verknüpfungen auf und zeigt, welche Pfade für das Holen (`fetch`) bzw. das Übertragen (`push`) von Commits verwendet werden:

```
> git remote --verbose
origin ssh://stachi@server.de:git-buch.git (fetch)
origin git@github.com:rpreissel/git-workflows.git (push)
klon file:///tmp/git-buch-klon.git (fetch)
klon file:///tmp/git-buch-klon.git (push)
```

Noch mehr Informationen erhält man mit dem `remote show`-Befehl:

```
> git remote show klon
```

Mit dem `remote rm`-Befehl kann man die Kurznamen auch wieder löschen:

```
> git remote rm klon
```

## 10.8 Zusammenfassung

Repository erstellen mit dem `init`-Befehl.

Repository klonen mit dem `clone`-Befehl.

Ein Bare-Repository ist ein Repository ohne Workspace. Es wird mit der Option `--bare` des `init`-Befehls oder des `clone`-Befehls erstellt.

**Repository-URLs:** Der Ort von anderen Repositorys kann im URL-

Format angegeben werden, z. B. `ssh://stachi@server.de: git-buch.git`.

Folgende Protokolle werden unterstützt: `file`, `ssh`, `http`, `https`, `ftp`, `ftps`, `rsync` und `git`.

**Kurznamen:** Über den `remote`-Befehl kann man Kurznamen für Repositorys definieren, damit man nicht immer die langen URLs angeben muss.



# 11 Austausch zwischen Repositorys

Beim Entwickeln mit Git arbeitet man lokal auf einem eigenen Klon des Projekt-Repository und erstellt dort Branches und Commits. Gelegentlich löscht man auch Branches oder strukturiert Dinge per Rebase um. Irgendwann möchte man aber seine Entwicklungen mit anderen Entwicklern teilen oder Änderungen von anderen Entwicklern übernehmen.

## 11.1 Fetch, Pull und Push

Mit den Befehlen `fetch`, `pull` und `push` ist ein Austausch zwischen Git-Repositories möglich. Meist wird dabei ein gemeinsames Projekt-Repository auf einem Server genutzt, der für alle Entwickler erreichbar ist. Grundsätzlich ist aber auch ein dezentraler Austausch zwischen beliebigen Repositories ohne einen zentralen Server möglich, sofern entsprechende Lese- bzw. Schreibberechtigungen gegeben sind (Abbildung 11-1).

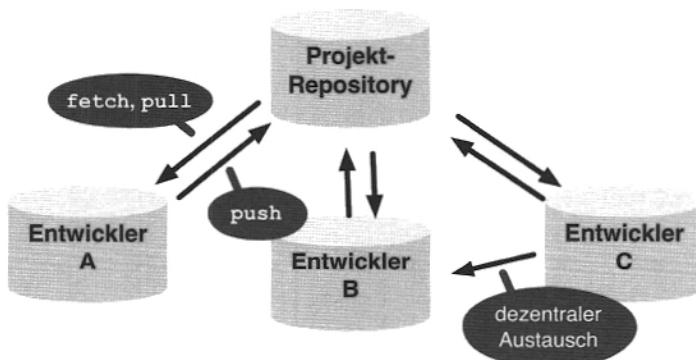


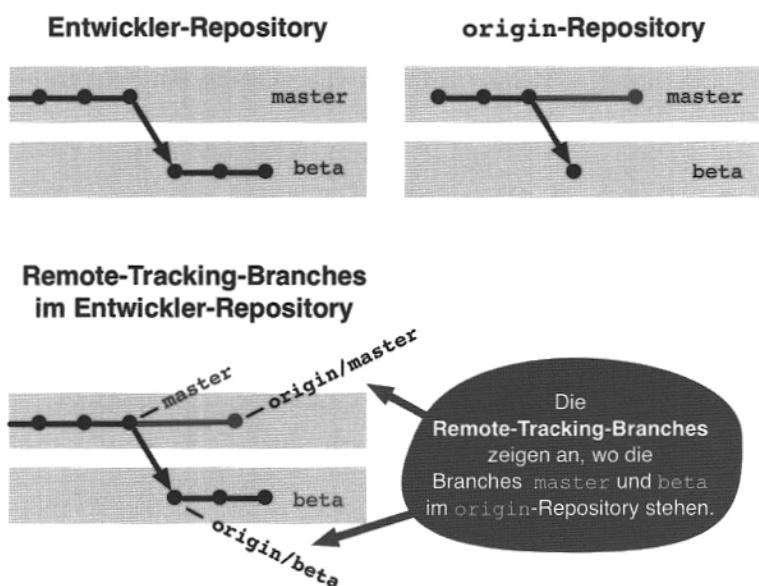
Abb. 11-1  
Austausch zwischen  
Repositories

## 11.2 Remote-Tracking-Banches

Wenn man verstehen möchte, wie `fetch` und `pull` funktionieren, ist es hilfreich, das Konzept der Remote-Tracking-Banches zu kennen. Branches in Git sind ein lokales Konzept. Jeder Klon hat einen eigenen Satz von Branches, der sich von den Branches in anderen Klonen unterscheidet. Sobald die Entwickler neue Commits erstellen, Branches anlegen oder Branches löschen, entwickeln sich die Klonen auseinander.

Oft möchte man seinen lokalen Stand mit dem Stand eines anderen Repositorys vergleichen. Dafür gibt es *Remote-Tracking-Banches*. Diese sind lokale Stellvertreter für Branches aus anderen Repositorys.

**Abb. 11-2**  
Remote-Tracking-  
Branches



Sie werden beim Aufruf der Befehle `fetch` und `pull` aktualisiert. Ein *Remote-Tracking-Branch* zeigt auf das Commit, auf das der Original-Branch im anderen Repository gezeigt hat, als zum letzten Mal aktualisiert wurde (Abbildung 11-2).

Auch *Remote-Tracking-Banches* haben Namen. Diese setzen sich aus dem Namen des Herkunfts-Repository und dem dortigen Branch-Namen zusammen, z. B. `origin/beta` für den Branch `beta` aus dem Repository `origin`. Mit den Optionen `--list` und `--remote` zeigt der `branch`-Befehl sie an:

```
> git branch --list --remote --verbose  
...  
origin/alpha      7d3e697 Einlesen der Dateien  
origin/beta      d479689 Hilfsklasse eingeführt  
origin/gamma     0559c1b Zeitzonen berücksichtigen  
origin/master    2b547c0 Bugfix #1234
```

*Remote-Tracking-Banches* dürfen in Befehlen fast überall verwendet werden, wo Commits referenziert werden:

```
> git merge origin/gamma  
> git log origin/alpha..experiments
```

**Achtung!** Man sollte nicht direkt auf *Remote-Tracking-Banches* entwickeln, denn sie sind nur Stellvertreter für *Remote-Banches*. Versucht man trotzdem ein checkout, gerät man in den *Detached Head State*, in dem neue Commits nicht automatisch einem Branch zugeordnet werden. Das kann, gerade für Einsteiger, verwirrend sein.

```
> git checkout origin/gamma # Meist keine gute Idee!
```

Note: checking out 'origin/alpha'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

...

## 11.3 Einen Remote-Branch bearbeiten

In Git arbeitet man grundsätzlich auf lokalen Branches. Wenn man einen *Remote-Branch* weiterentwickeln möchte, erstellt man mit dem branch-Befehl einen lokalen Branch dafür und verknüpft diesen mit dem entsprechenden *Remote-Tracking-Branch*. Man nennt `origin/alpha` dann den *Upstream-Branch* für `alpha`.

```
> git branch alpha origin/alpha  
> git checkout alpha
```

Es geht noch einfacher. Beim checkout legt Git den lokalen Branch automatisch an, wenn es genau einen *Remote-Tracking-Branch* mit passendem Namen und noch keinen lokalen Branch dieses Namens gibt.

```
> git checkout alpha
```

Die Verknüpfung von lokalen Branches mit ihren *Upstream-Branche*s zeigt der branch-Befehl mit doppelter -v Option:

```
> git branch --list -vv
* alpha      c5e2c7e [origin/alpha: ahead 1] add comment
  master     18dc60b [origin/master: ahead 1] fix bug
```

Auch der remote show-Befehl kann Informationen über *Branches* zeigen:

```
> git remote show origin
* remote origin
  Fetch URL: git@github.com:projekte/beispielprojekt.git
  Push  URL: git@github.com:projekte/beispielprojekt.git
  HEAD branch: master
  Remote branches:
    alpha                  tracked
    beta                  tracked
    master                tracked
  Local branches configured for 'git pull':
    alpha          merges with remote alpha
    beta          merges with remote beta
    master        merges with remote master
  Local refs configured for 'git push':
    alpha          pushes to alpha
    (up to date)
    beta          pushes to beta
    (local out of date)
    master        pushes to master
    (fast-forwardable)
```

## 11.4 Ein paar Begriffe, die man kennen sollte

Es lohnt sich die folgenden Begriffe zu merken. Wir werden sie im Folgenden häufig verwenden, und auch in der Git-Dokumentation kommen sie oft vor.

**Branch:** ein lokaler Branch im eigenen Repository

**Remote-Branch:** ein Branch in einem anderen Repository

**Remote-Tracking-Branch:** ein spezieller lokaler Branch, der anzeigt, wo ein Remote-Branch zum Zeitpunkt des letzten Abgleichs stand, z. B. `origin/alpha`

**Upstream-Branch:** Verknüpfung zwischen einem lokalen und einem Remote-Tracking-Branch. Im Beispiel ist `origin/alpha` der Upstream-Branch für `alpha`.

## 11.5 Fetch: Branches aus einem anderen Repository holen

Der `fetch`-Befehl holt Branches mitsamt benötigten Commits und Tags aus anderen Repositorys ab. Man kann angeben, von welchem Repository man Branches abholen möchte (im Beispiel `origin`) und welche Branches man holen möchte (hier `alpha` und `master`):

```
> git fetch origin alpha master
```

Dabei passiert Folgendes:

**Fehlende Commits holen:** Oft haben die angeforderten Remote-Banches neue Commits in der Historie, die lokal noch nicht vorhanden sind. Diese werden übertragen.

**Remote-Tracking-Banches setzen:** Für jeden angeforderten Branch wird ein *Remote-Tracking-Branch* erzeugt oder aktualisiert (im Beispiel: `origin/alpha` und `origin/master`).

**Tags holen:** Tags werden ebenfalls übertragen, wenn sie in die Geschichte der abgeholt Branches zeigen.

**Versionen markieren**  
→ Seite 105

Lokale Branches und der Workspace werden vom `fetch`-Befehl nicht berührt. Man kann ihn jederzeit »gefährlos« aufrufen, um zu untersuchen, was sich in anderen Repositorys getan hat. Viele Tools, wie z. B. SourceTree, machen das automatisch.

## 11.6 Fetch: Aufrufvarianten

Man darf die Branch-Angaben auch weglassen. Git holt dann alle Branches aus dem anderen Repository:

```
> git fetch origin # alle Branches holen
```

Die Repository-Angabe darf man ebenfalls weglassen. In diesem Fall holt Git alle Branches aus dem Repository `origin` ab:

```
> git fetch # alle Branches von 'origin' holen
```

### Fetch: Ausgabe der Aktualisierungen

Zunächst gibt der `fetch`-Befehl etwas Statistik zur Übertragung an. Danach zeigt er an, welche Remote-Tracking-Banches neu hinzugekommen, aktualisiert oder gelöscht wurden. Man sieht also, was seit dem

*Tipp: Ausgabe von  
fetch beachten!*

letzten Fetch im Projekt passiert ist. Hier sieht man beispielsweise, dass origin/feature-a von 9f36660 zu 41c3bc4 aktualisiert wurde:

```
From git@github.com:projekte/beispielprojekt.git  
9f36660..41c3bc4 alpha -> origin/alpha
```

Jetzt kann man sich ein Diff oder Log zeigen lassen, um zu sehen, was gerade geholt wurde:

```
> git diff 9f36660..41c3bc4  
> git log 9f36660..41c3bc4
```

### Welche Änderungen sind noch nicht integriert?

Möchte man sehen, welche Änderungen irgendwann geholt (nicht nur beim letzten Fetch) und noch nicht integriert wurden, kann man dies wie folgt tun:

```
> git diff alpha...origin/alpha  
> git log alpha...origin/alpha
```

Möchte man die Änderungen übernehmen, kann man das mit dem merge-Befehl tun:

```
> git fetch origin alpha  
> git merge origin/alpha
```

### Pull: Änderungen holen und integrieren

Dass man Änderungen zuerst aus einem anderen Repository abholt und dann lokal integriert, kommt oft vor.

Pull = Fetch + Merge

Deshalb gibt es einen Befehl, der beides macht. Der pull-Befehl führt zunächst ein Fetch aus und integriert für den aktuellen Branch die Änderungen dann gleich per Merge.

```
> git pull origin feature-a
```

*Tipp: Commit vor dem Pull  
Bearbeitungskonflikte  
→ Seite 70*

Beim Pull kann es zu Merge-Konflikten kommen. Deshalb sollte man darauf achten, dass man vor dem Pull einen sauberen Workspace (ohne uncommitted Changes) hat. Falls es dann zu Problemen beim Merge kommt, kann man den Workspace einfach zurücksetzen, ohne dass man Änderungen verliert:

```
> git merge --abort
```

## Pull: Aufrufvarianten

Meist ruft man den pull-Befehl parameterlos auf. Dann wird einfach der aktuelle Branch mit seinem *Upstream-Branch* zusammengeführt.

```
> git pull    # Integriere den Upstream-Branch
              # für den aktuellen Branch
```

Als kombinierter Befehl akzeptiert der pull-Befehl sowohl Optionen vom fetch-Befehl als auch vom merge-Befehl. Gibt man beides an, müssen die Fetch-Optionen hinten stehen. Das ist ein wenig überraschend, da das Fetch ja zuerst ausgeführt wird.

**Branches zusammenführen**  
→ Seite 67

```
> git pull --no-ff --all  # Reihenfolge beachten!
              # --no-ff ist Option von merge
              # --all ist Option von fetch
```

## Push: Änderungen veröffentlichen

**Push** Der push-Befehl überträgt Daten vom lokalen Repository in ein anderes Repository. Der Aufruf sieht ähnlich aus wie beim fetch-Befehl. Man gibt das Ziel-Repository und die zu übertragenden Branches<sup>1</sup> an:

```
> git push origin master alpha
```

Dabei passiert Folgendes:

**Fehlende Commits übertragen:** Commits aus der Historie der angegebenen Branches werden übertragen, soweit sie im Ziel-Repository fehlen.

**Remote-Banches setzen:** Die ausgewählten Branches werden im *Remote-Repository* auf den Stand gesetzt, den sie im lokalen Repository haben.

**Tags übertragen:** Wenn Tags direkt angegeben wurden oder eine der Optionen --tags oder --follow-tags gesetzt wurde.

**Achtung!** Während der fetch-Befehl nur *Remote-Tracking-Branches* setzt, verändert der push-Befehl die »echten« Branches im Ziel-Repository.

---

<sup>1</sup> Auch Tags sind hier erlaubt.

## Push: Aufrufvariante

Wenn man gar nichts angibt, wird der aktuelle Branch auf seinen *Upstream-Branch* übertragen.

```
> git push
```

**Achtung!** Das Verhalten des `push`-Befehls ist konfigurierbar. Die Konfigurationseinstellung dazu ist `push.default`. Das oben beschriebene Verhalten nennt sich `simple` und eignet sich für Workflows, wo man beim *Push* seine Änderungen in dasselbe zentrale Repository hochlädt, aus dem man mit *Pull* Änderungen abholt.

Mit der Option `--all` werden alle Branches übertragen.

```
> git push --all
```

Mit `--set-upstream` wird für alle erfolgreich übertragenen Branches die *Upstream*-Konfiguration eingerichtet. Das ist nützlich, wenn man einen Branch überträgt, den man frisch angelegt hat und den man lokal noch weiter bearbeiten möchte:

```
> git push --set-upstream origin beta
```

Die Option `--delete` löscht Branches in anderen Repositorys:

```
> git push --delete alpha
```

### Fast-Forward-Merges

→ Seite 74

Die Option `--force` erlaubt es, Branches im anderen Repository auch dann zu überschreiben, wenn dadurch Commits abgeschnitten werden:

```
> git push --force # Lass das!
```

## Push: Konflikte

**Achtung!** Ein *Push* ist sicher, wenn das neue Commit ein Nachfahre des vorherigen Commits ist. Man nennt das dann *Fast-Forward*, und es geht nichts verloren, denn der alte Stand ist weiterhin in der Historie des Branch enthalten. Ist das Commit hingegen kein Nachfahre des vorigen Standes, dann verweigert Git den Push für diesen Branch, um zu verhindern, dass Commits aus der Historie eines Branch abgeschnitten werden. Man kann den Schutz mit der Option `--force` umgehen. Das ist jedoch selten gut.<sup>2</sup>

---

<sup>2</sup> Mit der Konfiguration `receive.denyNonFastForwards` kann man das Überschreiben im Remote-Repository verbieten.

Schritt für Schritt

## Push verweigert! Was tun?

*Ein Push wurde verweigert, weil auf dem gleichen Branch im anderen Repository ebenfalls Änderungen hinzugekommen sind. Der Konflikt muss lokal gelöst werden, bevor der Push durchgelassen wird.*

### 1. Konflikt feststellen

Der push-Befehl meldet den Konflikt durch die folgende, etwas umständliche Meldung:

```
> git push
To /tmp/git-buch-klon.git
! [rejected]      alpha -> alpha (fetch first)
error: failed to push some refs to '/Users/stachi/Buch/'
hint: Updates were rejected because the remote contains work that
hint: you do not have locally. This is usually caused by another
hint: repository pushing to the same ref. You may want to first
hint: integrate the remote changes (e.g., 'git pull ...') before
hint: pushing again. See the 'Note about fast-forwards' in
hint: 'git push --help' for details.
```

### 2. Pull durchführen

```
> git pull
```

### 3. Gegebenenfalls Merge-Konflikte bereinigen

```
> git mergetool
> git commit --all
```

### 4. Noch mal: Push

```
> git push
```

## 11.7 Erweiterte Möglichkeiten

**Mehr Kontrolle bei Fetch, Push und Pull**

→ Seite 115

Sie haben bereits gesehen, dass man die Befehle `fetch`, `pull` und `push` mit Branch-Namen als Parameter aufrufen kann. Für die normalen Workflows beim Entwickeln reicht das auch völlig aus. Git erlaubt hier aber noch mehr: Es dürfen auch sogenannte *Refs* und *Refspecs* angegeben werden. Damit bekommt man die volle Kontrolle darüber, was beim Übertragen geholt und gesetzt wird. Das ist hilfreich, wenn man Skripte zur Automatisierung von Git-Workflows programmiert.

## 11.8 Zusammenfassung

**Fetch, Pull und Push:** Mit diesen Befehlen werden Branches, Tags und Commits zwischen Repositorys übertragen.

**Fetch:** aus einem anderen Repository abholen und Remote-Tracking-Branches aktualisieren

**Pull:** wie Fetch, nur mit nachfolgendem Merge auf dem aktuellen Branch

**Push:** Commits in ein anderes Repository übertragen

**Remote-Branch:** ein Branch in einem anderen Repository

**Remote-Tracking-Branch:** Lokaler Stellvertreter für einen Remote-Branch

**Upstream-Branch:** Verknüpfung zwischen einem (lokalen) Branch mit einem Remote-Tracking-Branch

**Man arbeitet immer auf lokalen Branches:** Um einen Remote-Branch weiterzuentwickeln, erstellt man einen lokalen Branch, der diesen als Upstream-Branch hat.

**Schneller Checkout:** `checkout <branch>` erstellt automatisch einen solchen Branch mit passender Upstream-Verknüpfung, wenn der Branch noch nicht existiert und der Name eindeutig ist.

**Bei Push-Konflikten:** `Pull` ausführen. Konflikte bereinigen. Dann erneut ein `Push` versuchen.

**Refspecs:** Genaue Kontrolle darüber, was übertragen wird, geben so genannte *Refs* und *Refspecs* (»Mehr Kontrolle bei Fetch, Push und Pull« ab Seite 115).

# 12 Versionen markieren

Die meisten Projekte vergeben Nummern oder Namen für die ausgelieferten Versionen ihrer Software, z. B. »1.7.3.2« oder »gingerbread«. In Git gibt es hierfür *Tags*. *Tags* sind feste Referenzen auf bestimmte Commits. Anders als Branches werden *Tags* in der Regel nicht verändert. Sie dienen zur Dokumentation und nicht zur Weiterentwicklung.

## 12.1 Arbeiten mit Tags erstellen

Schritt für Schritt

### Ein Commit mit einem Tag markieren

#### 1. Normales Tag erstellen

Im folgenden Beispiel wird für den aktuellen Stand des Branch `master` ein Tag namens `1.2.3.4` mit dem Kommentar »Frisch gebaut.« vergeben:

```
> git tag 1.2.3.4 master -m "Frisch gebaut."
```

#### 2. Push: Einzelnes Tag

Tags werden beim Push nicht automatisch mit übertragen. Gibt man den Tag-Namen jedoch explizit an, wird das Tag übertragen:

```
> git push origin 1.2.3.4
```

Wählt man die Option `--tags` beim `push`-Befehl (Seite 101), werden für jeden übertragenen Branch auch die Tags übertragen.

*Tipp: Push – alle Tags übertragen*

```
> git push --tags
```

*Tipp: Signierte Tags*

Wenn man das Programm GnuPG (Gnu Privacy Guard) nutzt, kann man Tags auch mit einer digitalen Unterschrift versehen. Hierzu gibt man die Option `-s` an. Voraussetzung ist, dass man eine Default-Mail-Adresse in Git eingetragen hat, die gleichzeitig eine eingetragene User-ID in GnuPG ist.

```
> git tag 1.2.3.4 master -s -m "Unterschrieben."
```

Achtung! Erstellt man ein Tag einfach mit einer der Optionen `-m`, `-a`, `-s` oder `-u`, erzeugt Git für das *Tag* ein eigenes Objekt im Repository. Es enthält Informationen über den Benutzer und den Zeitpunkt der Erstellung des *Tags*. Ohne diese Optionen erstellt Git nur ein sogenanntes *Lightweight Tag*, das nur den *Commit-Hash* kennt.

## 12.2 Welche Tags gibt es?

Wenn der `tag`-Befehl ohne Parameter aufgerufen wird, zeigt er, welche *Tags* es gibt. Das können viele sein. Deshalb kann man der Option `-l` ein Muster mitgeben, z. B. `1.2.*`, um die Ausgabe einzuschränken.

```
> git tag -l 1.2.*
1.2.0.0      Anfang.
...
1.2.3.3      Neu gebaut.
1.2.3.4      Frisch gebaut.
```

## 12.3 Die Hashes zu den Tags ausgeben

Der `show-ref`-Befehl mit der Option `--tags` listet *Commit-Hashes* der *Tag-Objekte* auf. Mit der Option `--dereference` werden auch die *Hashes* der *Commit-Objekte* aufgeführt, markiert durch `^{}`:

```
> git show-ref --dereference --tags
...
f63cd7181787c9973788a97648796468cec474aa refs/tags/1.2.3.3
cef89bbd7121aac3cc38fe3a342045c9401bd6b9 refs/tags/1.2.3.3^{}
4a0228bdd0ab5e0180422c82bf706c42671a81af refs/tags/1.2.3.4
cef89bbd7121aac3cc38fe3a342045c9401bd6b9 refs/tags/1.2.3.4^{}
```

## 12.4 Die Log-Ausgaben um Tags anreichern

Mit der Option `--decorate` des `log`-Befehls werden zu jedem Commit die Tags und Branches ausgegeben:

```
> git log --oneline --decorate  
cef89bb (HEAD, tag: 1.2.3.4) Wieder alles umgebaut.  
9d4caed Merge branch 'Aenderungen'.  
dcd1c6c Noch was geändert.  
cc1a68 (tag: 1.2.3.3) Etwas geändert
```

## 12.5 In welcher Version ist es »drin«?

Oft stellt sich die Frage, ob ein bestimmter Bugfix oder ein bestimmtes Feature in der Version, die ein Kunde installiert hat, bereits enthalten ist. Wenn das Commit bekannt ist, ist die Frage leicht zu beantworten. Die Option `--contains` des `tag`-Befehls listet alle Tags auf, die das gegebene Commit bereits enthalten, d. h. bei denen es Vorgänger des Commits mit dem Tag ist.

```
> git tag --contains f63cd71  
1.2.3.3  
1.2.3.4
```

Achtung! Das Ergebnis kann irreführend sein, wenn Commits kopiert wurden. Wenn beispielsweise Versionen durch *Cherry-Picking* zusammengestellt werden, ist es kniffliger, herauszufinden, ob die Änderung enthalten ist. Man könnte das Log für ein bestimmtes Tag nach dem Commit-Kommentar durchsuchen:

```
> git log --oneline --grep "Gesuchter Kommentar." 1.2.3.3
```

Aber auch das funktioniert natürlich nur, wenn Kommentare vergeben wurden, die die Änderungen identifizieren – entweder durch eine aussagekräftige Beschreibung oder eine Ticket-ID aus dem Bugtracking-System. Dies ist ein weiterer guter Grund dafür, das Kopieren von Commits zu vermeiden.

## 12.6 Wie verschiebt man ein Tag?

Am besten verschiebt man ein Tag gar nicht. In Git sind Tags als feste Markierungen für Versionen vorgesehen. Solange man es noch nicht mit Push in andere Repositorys übertragen hat, kann man ein Tag ändern, indem man es mit der Option `--force` neu erstellt. Wenn das Tag

schon verbreitet ist, kann es für große Verwirrung sorgen, wenn eine zweite Variante in Umlauf gebracht wird: Ein Entwickler könnte dann unter demselben Tag-Namen etwas anderes sehen als ein anderer Entwickler.

## 12.7 Und wenn ich ein »Floating Tag« brauche?

Wenn man eine bewegliche Markierung benötigt, z. B. für den Stand, der aktuell in der Produktion installiert ist, nimmt man in Git einfach einen Branch.

## 12.8 Zusammenfassung

**Tags erstellen:** Das geschieht mit dem `tag`-Befehl.

**Push:** Der `push`-Befehl überträgt nur Tags, deren Namen man explizit angibt, z. B. `git push origin 1.2.3.4`, es sei denn, man gibt `--tags` an.

**Pull und Fetch:** Die Befehle `pull` und `fetch` holen alle Tags für die betroffenen Branches automatisch mit ab, es sei denn, man gibt `--no-tags` an.

**Alle Tags anzeigen:** Das kann man mit `git tag -l` erreichen.

**Tags im Log anzeigen** kann man mit `git log --decorate`.

**Gegebenes Commit in Tag:** Ob ein Tag ein gegebenes Commit enthält, zeigt der `tag`-Befehl mit der Option `--contains`.

**»Floating Tags« gibt es nicht:** In Git sind Tags feste Markierungen, die man nicht nachträglich verschieben sollte. Wenn man bewegliche Markierungen benötigt, nimmt man einfach Branches.

# 13 Tipps und Tricks

Wir wollten die einführenden Kapitel nicht überfrachten und haben uns auf die wesentlichen Konzepte und auf typische Anwendungsfälle beschränkt. In diesem Kapitel hingegen finden Sie eine Sammlung von Tipps und Tricks, von denen einige in bestimmten Situationen sehr hilfreich sein können, von denen Sie aber andere vielleicht nie benötigen werden, auch wenn Sie lange mit Git arbeiten. Wahrscheinlich genügt es, wenn Sie dieses Kapitel kurz überfliegen, damit Sie wissen, was es hier gibt. Die Details können Sie dann bei Bedarf nachschlagen.

## 13.1 Keine Panik – es gibt ein Reflog!

»Git ist like a dog. It can smell your fear.«

Als dieser Spruch in meiner Twitter-Timeline auftauchte, musste ich schmunzeln. Git wirkt auf Einsteiger tatsächlich etwas einschüchternd. Aber falls Git wirklich ein Hund ist, dann wohl doch eher ein Hütehund, der seine (Entwickler-)Herde zu schützen versucht.

Man sollte wissen, dass Git Objekte im Repository nicht sofort löscht. Sobald man etwas verändert, erstellt Git neue Objekte im Repository; die alten werden nicht gelöscht. Selbst eine *Garbage Collection*, z. B. durch den gc-Befehl, löscht nur Objekte, die ein bestimmtes Mindestalter haben. Die Default-Einstellung hierfür liegt bei zwei Wochen (Konfigurationsoption: `gc.pruneexpire`).

Außerdem führt Git Buch über alle Änderungen, die man an Branches durchführt. Dieses sogenannte *Reflog* wird im Verzeichnis `.git/logs` abgelegt. Wann immer ein Commit, ein Checkout, ein Rebase, ein Reset o. Ä. den HEAD oder einen Branch verändert, entsteht ein Eintrag im *Reflog*. Der `reflog`-Befehl zeigt den Verlauf des HEAD-Standes:

```
> git reflog
9b2227f HEAD@{0}: commit: TODO erledigen.
1451b31 HEAD@{1}: commit: Tippfehler korrigieren
1ca1983 HEAD@{2}: checkout: moving from master to mybranch
```

**Das Repository**  
→ Seite 49

Noch detailliertere Informationen kann man mit den folgenden Befehlen erhalten:

```
> git reflog --all  
> git log --walk-reflogs --decorate --pretty --all
```

Hat man dort das Commit mit den »verlorenen« Änderungen ausfindig gemacht, dann kann man die Änderungen leicht wieder zurückholen, z. B. mit dem `cherry-pick`-Befehl, dem `rebase`-Befehl oder mit einem einfachen `Merge`.

Achtung! Für lokale Klone ist das *Reflog* normalerweise aktiv. *Bare-Repositorys*, die man auf Servern ablegt, führen per Default kein *Reflog* durch. Man kann dies aber einschalten:

```
> git config --global core.logAllRefUpdates true
```

## 13.2 Lokale Änderungen temporär ignorieren

Manchmal ändert man Dateien, die von Git verwaltet werden, ohne dass man das Ergebnis in die Versionierung übernehmen möchte. Ein Beispiel: Beim Schreiben dieses Buches haben wir während des Arbeitens häufig Kapitel auskommentiert, um das Dokument schneller erzeugen zu können. Diese Änderungen wollten wir natürlich nicht »mitversionieren«. Ein weiteres Beispiel: Um einen Fehler zu finden, baut man zusätzliche Debug-Ausgaben ein, die man aber später nicht mehr benötigt.

Einträge in `.gitignore` helfen hier nicht weiter, denn diese wirken nur auf Dateien, die noch nicht durch Git verwaltet werden (siehe auch »Mit `.gitignore` Dateien unversioniert lassen« ab Seite 46).

Man kann das Problem durch *selektive Commits* umgehen. Das ist aber etwas mühsam, weil man dann bei jedem nachfolgenden Commit erneut wählen muss, welche Änderungen man übernehmen möchte und welche nicht.

**Selektives Commit – Änderungen auswählen** → Seite 41

Schritt für Schritt

## Versionierte Dateien ignorieren

Von Git verwaltete Dateien werden temporär ignoriert, sodass Änderungen an diesen Dateien nicht übernommen werden.

### 1. Versionierte Dateien ignorieren

Die Option `--assume-unchanged` des `update-index`-Befehls setzt eine Marke im *Stage-Bereich*, die dafür sorgt, dass Git künftig nicht mehr prüft, ob sich die Datei geändert hat, sondern einfach so tut, als ob die Datei unverändert wäre.

```
> git update-index --assume-unchanged foo.txt
```

### 2. Arbeiten

Jetzt können Sie weiterarbeiten. Git wird Änderungen an der Datei `foo.txt` weder mit dem `status`-Befehl anzeigen noch mit dem `add`-Befehl übernehmen. Änderungen an allen anderen Dateien werden ganz normal verarbeitet.

### 3. Das Ignorieren wieder aufheben

Mit `--no-assume-unchanged` können Sie die Wirkung von `--assume-unchanged` für einzelne Dateien aufheben. Bei einem `--really-refresh` wird der Status für alle Dateien zurückgesetzt.

```
> git update-index --really-refresh
```

## 13.3 Änderungen an Textdateien untersuchen

Der normale *Diff-Algorithmus* von Git vergleicht zeilenweise. Dort ändert man oft einzelne Zeilen oder fügt welche hinzu. Die benachbarten Zeilen bleiben unverändert. In Texten ist das anders. Bei Änderungen wird oft umbrochen, d. h., Wörter wandern von einer Zeile in die andere. Bei dem zeilenweisen Diff ist nur schlecht zu erkennen, was genau verändert wurde.

```
> git diff  
...  
-Walter geht jeden  
-Tag in die Schile.  
+Walter geht in  
+die Schule.  
...
```

Mit der Option `--word-diff` kann Git die Änderungen auch wortweise anzeigen. Für Fließtexte ergibt das oft eine übersichtlichere Darstellung:

```
> git diff --word-diff  
...  
Walter geht[-jeden -]  
[-Tag-] in  
die [-Schile.-]{+Schule.+}  
...
```

Mit `--word-diff=color` kann man die Unterschiede farbig hervorheben lassen.

## 13.4 alias – Abkürzungen für Git-Befehle

Wenn man Git viel über die Kommandozeile nutzt, kann es nützlich sein, für häufig verwendete Befehle Kurzformen zu definieren:

```
> git config --global alias.ci commit  
> git config --global alias.st status
```

Hier werden die Aliase `ci` und `st` eingerichtet. Sie können sofort verwendet werden. Zum Beispiel:

```
> git st
```

Die *Aliase* werden auch bei der Tab-Completion von Git berücksichtigt (sofern sie installiert ist). Deshalb kann es auch nützlich sein, sprechende Aliase für seltener benutzte Befehle einzurichten. Zum Beispiel:

```
> git config --global alias.ignore-temporarily  
'update-index --assume-unchanged'
```

## 13.5 Branches als temporäre Zeiger auf Commits nutzen

Wenn man Fehler sucht oder knifflige Merge-Konflikte bereinigt, möchte man sich oft wichtige Commits merken, z. B. den Punkt, an dem noch alles gut war, oder die Merge-Base. Einer der Autoren hat hierzu anfangs immer Commit-Hashes auf einem Blatt karierten Papier notiert. Das muss nicht sein! Man kann einfach einen Branch anlegen, sobald man ein interessantes Commit entdeckt:

```
> git branch tmp/der-dumme-fehler 8b167
```

Ab jetzt kann man das Commit später jederzeit über diesen Namen referenzieren. Auch die Tab-Completion »kennt« jetzt diesen Namen. Man kann sich beispielsweise anzeigen lassen, welche Tags das betreffende Commit enthalten:

```
> git tag --contains tmp/der-dumme-fehler  
1.0.2  
1.0.3
```

Das folgende Kommando erzeugt einen Branch `tmp/merge-base` als Zeiger auf die *Merge-Base* der Branches `master` und `feature`:

```
> git branch tmp/merge-base  
    'git merge-base master feature'
```

**Achtung!** Für die oben gezeigte geschachtelte Befehlsausführung müssen Backtick-Anführungszeichen eingegeben werden.

Das `tmp`-Präfix ist eine Namenskonvention: Es gibt den temporären Branches einen Namespace, damit man sie besser von den »normalen« Branches unterscheiden kann. Eine technische Notwendigkeit ist das nicht. Sie können die temporären Branches nennen, wie Sie wollen.

Später kann man die Branches wie folgt abräumen:

```
> git branch -D  
'git branch --no-color --list tmp/*  
| grep -v '*'  
| xargs'
```

## 13.6 Commits auf einen anderen Branch verschieben

Wenn man eine Änderung vornimmt, ist es natürlich am besten, das gleich auf dem richtigen Branch zu tun. Branch-Wechsel sind ja kein großes Ding in Git. Manchmal möchte man sich aber nicht aus dem Flow bringen lassen und führt kleinere Korrekturen auf dem aktuellen Feature-Branch durch. Dadurch vermischen sich Commits, die zum Feature gehören, mit solchen, die nichts damit zu tun haben. Das hat zwei Nachteile: Zum einen erschwert es die Qualitätssicherung durch Reviews und zum anderen können die Korrekturen nur zusammen mit dem Feature ausgeliefert werden. Das ist besonders dann ungünstig, wenn sich die Fertigstellung des Features verzögert. In solchen Fällen kann es sinnvoll sein, einige Commits auf einen anderen Branch zu verschieben.

Schritt für Schritt

### Commits auf einen anderen Branch verschieben

*Einige Commits aus einem Branch A sollen auf einen Branch B verschoben werden.*

#### 1. Commits umsortieren und die Trennstelle mit tmp/SPLIT markieren

Commits müssen am Ende des Branch stehen, um sie verschieben zu können.

```
> git rebase --interactive
```

Im Editor sortieren Sie die Zeilen so, dass oben die Zeilen stehen, die auf A verbleiben sollen, und unten jene, die nach B verschoben werden sollen. Dazwischen erzeugen Sie mit einer exec-Zeile einen temporären Branch tmp/SPLIT, der die Trennstelle markiert:

```
pick 6a2f459 soll auf A bleiben 1
pick 05c2935 soll auf A bleiben 2
exec git branch -f tmp/SPLIT
pick af22ed6 soll nach B verschoben werden 1
pick 4f30adf soll nach B verschoben werden 2
```

Der Branch tmp/SPLIT zeigt jetzt auf das letzte Commit, das auf A verbleiben soll.

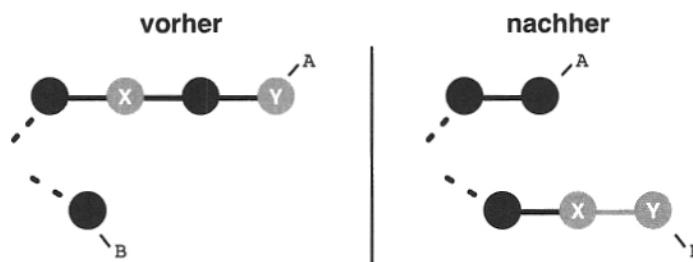
## 2. Verschiebung durchführen

Erst wird ein temporärer Branch tmp/MOVE erzeugt und dann verschoben. Dann wird tmp/MOVE per merge-Befehl in B übernommen. Schließlich werden die überschüssigen Commits auf A abgeschnitten:

```
> git checkout -B tmp/MOVE A
> git rebase tmp/SPLIT --onto B

> git checkout B
> git merge --ff tmp/MOVE

> git branch --force A tmp/SPLIT
```



*Abb. 13-1  
Commits von Branch A nach B verschieben*

Achtung! Das Verschieben von Commits kann für andere Entwickler verwirrend sein (siehe »Empfehlungen zum Rebasing« ab Seite 85). Man sollte entweder nur solche Commits verschieben, die man noch nicht mit anderen Entwicklern geteilt hat, oder man muss die anderen Entwickler darüber informieren und sie bitten, ihrerseits Commits zu verschieben, die auf Basis der verschobenen Commits entwickelt wurden.

## 13.7 Mehr Kontrolle bei Fetch, Push und Pull

Als Parameter für fetch, pull und push darf man mehr als nur einfache Branch-Namen angeben.

### Refs

In Git gibt es einen Sammelbegriff *Ref* für Dinge, die auf ein Commit verweisen, dazu gehören Branches, Remote-Tracking-Branches und Tags. Bei den Befehlen fetch, pull und push dürfen anstelle von Branch-Namen auch andere Refs angegeben werden, z. B. das Tag 1.0.0:

```
> git pull origin 1.0.0
```

**Das Repository-Layout**

→ Seite 89

*Refs* werden in einer Verzeichnisstruktur unterhalb von `.git/refs` abgelegt. Der `show-ref`-Befehl zeigt Pfade und Commit-Hashes aller *Refs* an:

```
> git show-ref
2218ac5898a66f54aa9d06d7dafa780d50ea1b66 refs/heads/feature-a
...
c55f7ffa0750fc79fc0417ce36268eed89ce18b refs/remotes/origin/master
...
848a18578fc56eb033f94f731739e81153b983b4 refs/tags/1.0.0
```

Diese Pfade können als Parameter angegeben werden:

```
> git fetch refs/tags/1.0.0
```

*Tipp: Refs abkürzen*

Die Pfade der *Refs* müssen nicht immer vollständig angegeben werden: Statt `refs/remotes/origin/master` genügt in der Regel auch `remotes/origin/master` oder `origin/master`.

*Tipp: Tab-Completion*

In der Kommandozeile gibt es eine Tab-Completion für *Refs*. Probieren Sie es aus: Geben Sie `git log refs/` ein und drücken Sie dann die Tab-Taste.

## Refspecs

Die Befehle `fetch`, `pull` und `push` übertragen *Refs* von einem Repository zum anderen.

```
> git fetch origin master
```

Hier etwa holt der `fetch`-Befehl die *Ref* `refs/heads/master` aus dem Repository `origin` und setzt die lokale *Ref* `refs/remotes/origin/master`. Git hat anhand einiger Konventionen und Konfigurationen ermittelt, welche beiden *Refs* hier gemeint sein müssen. Man kann, falls nötig, aber auch beide *Refs* explizit angeben. Eine sogenannte *Refspec* gibt in der Form `quelle:ziel` an, welche *Ref* geholt (quelle) und welche *Ref* gesetzt (ziel) werden soll.

Normalerweise wird die Ziel-*Ref* nur überschrieben, wenn das neue Commit Nachfahre des vorherigen Standes ist (Fast-Forward). Diesen Schutz kann man durch ein vorangestelltes `+` ausschalten. Dann wird die Ziel-*Ref* ohne Prüfung überschrieben. Der vorige Befehl könnte auch wie folgt geschrieben werden:

```
> git fetch \
    origin +refs/heads/master:refs/remotes/origin/master
```

*Fast-Forward-Merges*

→ Seite 74

# 14 Workflow-Einführung

## 14.1 Warum Workflows?

In den letzten Kapiteln haben Sie die Grundkonzepte von Git kennengelernt. Es wurden nur die wichtigsten Befehle mit den wichtigsten Parametern benutzt.

Selbst bei dieser Auswahl haben Sie sich vielleicht die Frage gestellt: Wann wird jetzt noch mal Merge oder Rebase eingesetzt?

Versucht man sich dann im Internet schlau zu machen, stößt man auf verschiedenste Anwendungen und auf noch mehr Befehle und Parameter. Diese Flexibilität ist einerseits eine Stärke von Git, andererseits erschwert sie den Einstieg und lässt Git komplex erscheinen.

**Die Grenzen von Git**  
→ Seite 297

Die folgenden *Workflows* beschreiben deswegen die typische Nutzung von Git in Entwicklungsprojekten. Dabei wird der Fokus auf die Erledigung von Aufgaben gelegt und nicht auf noch mehr Parameter. Es wird bewusst immer nur eine Lösung beschrieben. Diese wird aber so detailliert erläutert, dass dieser Workflow ausreichend für Ihre Arbeit ist und Sie nicht erst wieder in Hilfdateien nachschauen müssen.

Selbst nach langjähriger Nutzung von Git gibt es Aufgaben, die man seltener erledigen muss, z. B. ein Repository aufteilen. Dann dienen die Workflows als kompakte Ablaufbeschreibung der Befehle.

In den Workflows werden auch unbekanntere Befehle und Parameter verwendet, wenn die Aufgabe es erfordert. Dadurch lernen Sie im »Vorbeilesen« weitere Möglichkeiten von Git kennen.

Besonders bei der Einführung von Git helfen die konkreten Ablaufbeschreibungen dabei, die typischen Anfangsprobleme zu umgehen bzw. zu mildern.

## 14.2 Welche Workflows sind wann sinnvoll?

Bei der Auswahl der Workflows haben wir uns an einem typischen Projektablauf orientiert.

### Der Projektstart

Wenn Sie frisch mit einem Projekt beginnen und sich für Git als Versionsverwaltung entschieden haben, dann besteht die erste Aufgabe darin, eine Infrastruktur für die Versionierung auszuwählen und aufzusetzen. Der Workflow »Ein Projekt aufsetzen« (Seite 123) beschreibt die Möglichkeiten.

Wird ein Projekt nicht neu gestartet, sondern soll ein bestehendes Projekt nach Git migriert werden, dann beschreibt der Workflow »Ein Projekt nach Git migrieren« (Seite 247), wie der Umstieg erfolgen kann.

### Die Entwicklung

Nachdem die Infrastruktur definiert ist, muss im Team der Umgang mit Branches geklärt werden. Sie können alle gemeinsam auf dem master-Branch arbeiten, wie im Workflow »Gemeinsam auf einem Branch entwickeln« (Seite 135) beschrieben, oder es wird für jede Aufgabe ein separater Feature-Branch angelegt, wie es der Workflow »Mit Feature-Branches entwickeln« (Seite 143) zeigt. Oder Sie machen es wie viele Open-Source-Projekte auf GitHub und nutzen Forks zur Kollaboration, wie es in »Mit Forks entwickeln« (Seite 163) beschrieben wird.

### Ein Projekt ausliefern

#### Versionen markieren

→ Seite 105

#### Branches verzweigen

→ Seite 59

Git macht keine Vorgaben, wenn es um den Release-Prozess geht. Mit Tags und Branches stellt Git jedoch mächtige Werkzeuge bereit, um ein sehr breites Spektrum von Release-Prozessen zu unterstützen.

Der Workflow »Periodisch Releases durchführen« (Seite 185) beschreibt einen Release-Prozess für ein typisches (Web-)Projekt mit regelmäßigen Releases, wobei es nur ein produktives Release gibt. Der Workflow zeigt, wie nach Abschluss der Entwicklung die Tests des neuen Release und die Entwicklung für das nächste Release parallel passieren können und wie schwerwiegende Fehler am letzten Release behoben werden können.

Wenn es gleichzeitig mehrere aktive Releases für ein Softwareprodukt gibt, dann müssen Hotfixes geordnet auf die Releases verteilt werden. Unter Umständen ist auch ein Übertragen von Änderungen von aktuelleren Releases auf ältere Releases (Backport) notwendig. Der

Workflow »Mit mehreren aktiven Releases arbeiten« (Seite 199) beschreibt die notwendigen Schritte.

Betreiben Sie eine Deployment-Pipeline, um Ihr Produkt regelmäßig und weitestgehend automatisch auszuliefern, dann beschreibt der Workflow »Kontinuierlich Releases durchführen« (Seite 175), wie jedes Commit zu einem potenziellen Release-Kandidaten wird.

## Repositories pflegen

Die Standardversion von Git eignet sich zur Versionierung von Quelltexten. Es ist kein Problem, auch kleine Binärdateien mit zu versionieren. Falls Sie allerdings große Binärdateien verwalten wollen, sollten Sie sich die Erweiterung *Git Large Files Storage* (LFS) ansehen. Wie Sie diese nutzen können, beschreiben wir im Workflow »Ein Projekt mit großen binären Dateien versionieren« (Seite 213).

Anforderungen an Projekte verändern sich mit der Zeit und mit neuen Erkenntnissen. Genauso kann es passieren, dass einmal getroffene Entscheidungen, wie Projekte auf Git-Repositories aufgeteilt werden sollen, neu überdacht werden müssen.

Ein anfangs kleines monolithisches Projekt ist gewachsen und muss modularisiert werden. Der Workflow »Große Projekte aufteilen« (Seite 221) beschreibt, wie das zugehörige große Repository in kleinere aufgeteilt werden kann.

Auch das Gegenteil kann passieren: Der Umgang mit einem initial auf viele Repositories aufgeteilten Projekt kann sich als zu komplex herausstellen. Das Zusammenbringen von Repositories wird im Workflow »Kleine Projekte zusammenführen« (Seite 229) beschrieben.

Wenn Projekte schon lange existieren und viele Veränderungen durchlaufen haben, dann kann das zugehörige Repository groß werden. Die Historien aller Dateien liegen bei jedem Entwickler lokal vor. Insbesondere wenn in früheren Versionen große binäre Dateien versioniert wurden, kann das zu unnötigem Ressourcenverbrauch führen. Der Workflow »Lange Historien auslagern« (Seite 235) beschreibt, wie eine Projekthistorie geteilt werden kann und nur noch die jüngeren Versionen bei jedem Entwickler lokal gespeichert werden.

**Ressourcenverbrauch bei großen binären Dateien → Seite 300**

## 14.3 Aufbau der Workflows

Die Beschreibung der Workflows in den folgenden Kapiteln ist immer gleich aufgebaut. Im Folgenden wird der Inhalt jedes Abschnitts kurz beschrieben.

## Der Einstieg

Die Workflows beginnen mit einer kurzen Motivation: Hier erklären wir, warum und in welchem Kontext ein Workflow angewendet werden sollte. Die zentralen Aufgaben, die möglichen Entscheidungen und die Besonderheiten werden beschrieben. Am Ende gibt es eine kurze Zusammenfassung der Punkte, die in diesem Workflow behandelt werden.

Nach diesem Abschnitt werden Sie wissen, ob der Workflow für Ihr Projekt relevant ist.

## Der Überblick

Der Überblick-Abschnitt beschreibt die grundlegenden Abläufe anhand eines Beispiels. Es werden die wichtigsten Begriffe, Konzepte und Git-Befehle beschrieben, die in diesem Workflow benötigt werden.

Nach diesem Abschnitt werden Sie verstehen, wie der Workflow prinzipiell funktioniert und welche Git-Mittel eingesetzt werden, aber noch nicht, welche Befehle und Optionen man genau einsetzt.

## Die Voraussetzungen

Jeder Workflow funktioniert nur unter bestimmten Voraussetzungen.

Nach dem Abschnitt »Voraussetzungen« werden Sie wissen, ob Ihr Projekt schon für den Einsatz des Workflows bereit ist. Sie werden erkennen, welche Voraussetzungen noch erbracht werden müssen oder warum dieser Workflow bei Ihnen nicht angewendet werden kann.

## Der kompakte Workflow

Diese kompakte Übersicht auf einer Seite beschreibt den Workflow in wenigen Sätzen und stellt in einer Abbildung die wichtigsten Konzepte und Ideen dar.

Wenn Sie dieses Buch als Nachschlagewerk nutzen, soll diese Seite Ihnen als Erinnerungshilfe dienen.

## Die Abläufe und ihre Umsetzung

Ein Workflow kann aus einem oder mehreren Abläufen bestehen. Abläufe beschreiben einzelne Teilaufgaben, die Sie für die Erledigung Ihrer Aufgabe benötigen. In diesem Abschnitt wird detailliert für jeden Ablauf erläutert, welche Git-Befehle mit welchen Parametern und in welcher Reihenfolge ausgeführt werden müssen. Es wird bewusst nur eine von uns vorgeschlagene Umsetzung beschrieben.

Nach diesem Abschnitt werden Sie wissen, wie Sie mit Git den jeweiligen Ablauf umsetzen können.

## Warum nicht anders?

Git ist ein großer Werkzeugkasten, und viele Aufgaben kann man mit verschiedenen Werkzeugen lösen. In diesem Abschnitt werden alternative Lösungen diskutiert, und wir argumentieren, warum in der beschriebenen Umsetzung genau das eine Werkzeug eingesetzt wird.

Dabei gibt es sowohl alternative Strategien, die aus unserer Sicht nicht sinnvoll sind oder nicht funktionieren, als auch Lösungen, die ebenso funktionieren würden, aber nur mit einem anderen Kontext. Es werden auch Lösungen diskutiert, die dann sinnvoll sind, wenn es um Spezialfälle der Aufgabe geht.

Dieser Abschnitt bringt Ihnen unsere Gründe für die Auswahl der Git-Mittel näher. Außerdem zeigt er Ihnen alternative Lösungsideen auf.



## 15 Ein Projekt aufsetzen

Nachdem man sich für den Einsatz von Git entschieden hat, besteht der erste Schritt darin, die Dateien und Verzeichnisse als Git-Repository zur Verfügung zu stellen. Dabei ist zu entscheiden, ob das Projekt in einem oder mehreren Repositorys bereitgestellt werden soll. Da Git nur für das gesamte Repository einen Branch oder ein Tag anlegen kann, hängt die Entscheidung sehr stark von den Release-Einheiten Ihres Projekts ab.

Nachdem die Aufteilung des Projekts erfolgt ist, muss ein Repository für jedes Modul angelegt und gefüllt werden. Dabei sind leere Verzeichnisse und nicht zu versionierende Dateien besonders zu behandeln.

Für die Arbeit im Team muss ein Repository pro Modul als das zentrale Repository definiert werden. Alle Entwickler werden dieses zentrale Repository nutzen, um den aktuellen Stand zu holen und ihre Änderungen einzuspielen.

Es muss entschieden werden, ob und welche zentrale Repository-Verwaltung man benutzen will. Oder ob man das Repository ohne zentrale Repository-Verwaltung veröffentlicht.

Dieser Workflow beschreibt:

- wie ein Projektverzeichnis in ein Repository überführt wird,
- wie leere Verzeichnisse versioniert werden,
- wie mit der Zeilenendenproblematik umgegangen wird,
- welche Fähigkeiten zentrale Repository-Verwaltungen bieten,
- welche Möglichkeiten es gibt, ein zentrales Repository direkt zur Verfügung zu stellen, und
- wie Teammitglieder auf das zentrale Repository zugreifen.

**Repositorys können  
nur vollständig  
verwendet werden**

→ Seite 300

**Das Repository**

→ Seite 49

## Überblick

Dieser Workflow besteht aus zwei Teilen. Im ersten Schritt wird ein Repository für ein Projektverzeichnis angelegt. Im zweiten Schritt wird ein zentrales Repository allen Entwicklern zur Verfügung gestellt.

### **Was sind Commits?**

→ Seite 31

In Abbildung 15–1 wird gezeigt, wie das Projekt `projecta` in ein Repository überführt wird. Dabei ist besonders auf das leere Verzeichnis `EmptyDir` zu achten. Denn leere Verzeichnisse werden von Git normalerweise nicht versioniert. Indem man eine beliebige Datei, z. B. `.gitignore`, in dem Verzeichnis anlegt, kann man die Versionierung erzwingen.

Ebenso muss man beim initialen Commit darauf achten, keine unnötigen Dateien zu versionieren, z. B. Build-Ergebnisse oder temporäre Dateien. Im Beispiel kann man das an dem Verzeichnis `TempDir` sehen. In ihm werden Backup-Dateien gespeichert, und es soll nicht versioniert werden. Um dieses Verzeichnis auch bei zukünftigen Commits auszuschließen, wird die Datei `.gitignore` im Wurzelverzeichnis des Projekts angelegt und das zu ignorierende Verzeichnis dort spezifiziert.

Im zweiten Schritt wird das neue Repository anderen Entwicklern zugänglich gemacht. Dabei wird das lokale Repository in eine zentrale Repository-Verwaltung übertragen. Alle Teammitglieder können von dort klonen und ihre Versionen austauschen.

## Voraussetzungen

**Zentrale Repository-Verwaltung:** Eine zentrale Repository-Verwaltung wird benötigt, um den Zugriff auf die gemeinsamen Repositorys zu ermöglichen.

**Rechtevergabe auf Projektebene:** Git kennt nur Rechte für das Lesen und Schreiben des gesamten Repositorys, d. h., es können keine feingranularen Rechte auf einzelnen Verzeichnissen vergeben werden.

**Keine großen Binärdateien:** Git speichert alle Versionen einer Datei im lokalen Repository und muss diese Versionen auch beim Austausch zwischen Repositorys übertragen. Große Binärdateien sind ein Problem, da viele Daten übertragen und abgelegt werden müssen. Deswegen sollte man darauf achten, die Binärdateien nicht im Repository abzulegen.

### **Autorisierung nur auf**

**dem ganzen**

**Repository**

→ Seite 301

### **Ein Projekt mit großen**

**binären Dateien**

**versionieren**

→ Seite 213

## Workflow kompakt

### Ein Projekt aufsetzen

Ein Projektverzeichnis wird in ein neues Repository importiert. Dieses Repository wird als zentrales Repository für die Entwicklung im Team zur Verfügung gestellt.

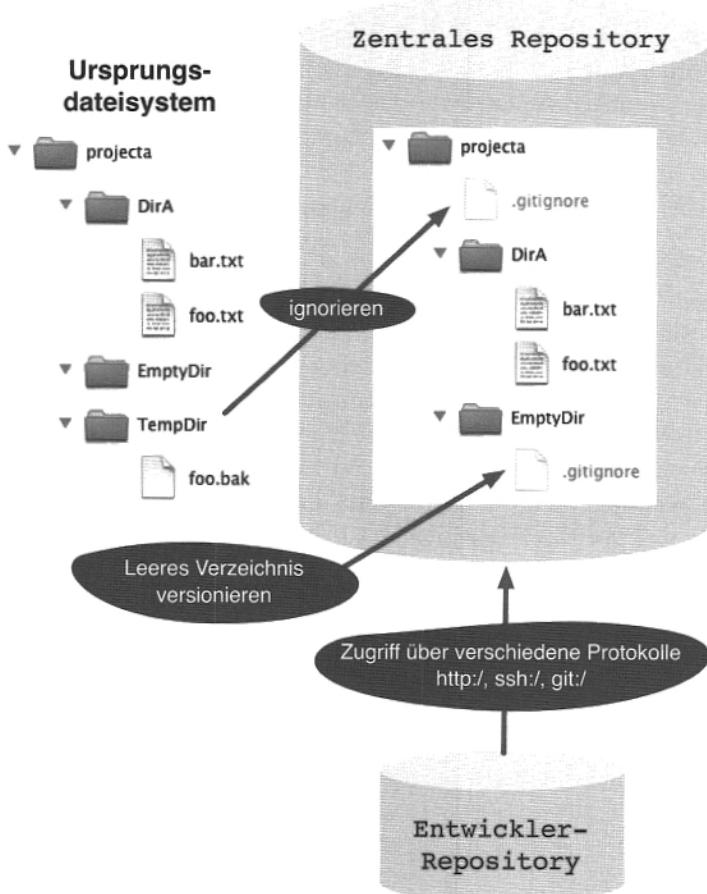


Abb. 15-1  
Workflow im Überblick

## Ablauf und Umsetzung

Die nachfolgenden Abläufe gehen von dem einfachen Beispielprojekt projecta in Abbildung 15–1 aus.

### Neues Repository vom Projektverzeichnis anlegen

Dieser Abschnitt zeigt, wie für ein vorhandenes Projekt ein Bare-Repository erzeugt wird. Ein Bare-Repository ist die Voraussetzung, um das Repository später im Team zu teilen.

Der Ausgangspunkt ist ein Verzeichnis im Dateisystem, das schrittweise in das fertige Bare-Repository umgewandelt wird.

#### Schritt 1: Leere Verzeichnisse vorbereiten

**Verzeichnisse speichern: Blob und Tree** → Seite 50

Git ist im Kern ein Content-Tracker, d. h., Git kann sehr effizient Versionen von Dateien verschiedenen Typs verwalten. Dagegen werden Verzeichnisse nur als Strukturierungseinheiten betrachtet und nur im Zusammenhang mit Dateien versioniert.

Leere Verzeichnisse sind für Git also nicht relevant und können auch nicht mit dem add-Befehl zu einem Commit hinzugefügt werden.

Solange die Entwicklungsumgebung auch auf die leeren Verzeichnisse nicht angewiesen ist, kann man diesen Umstand einfach ignorieren. Einige Entwicklungsumgebungen bzw. -werkzeuge gehen jedoch von der Existenz dieser Verzeichnisse aus, und es kommt zu Problemen, wenn sie fehlen.

Durch Anlegen einer beliebigen Datei kann man jedoch Git dazu bringen, auch ein leeres Verzeichnis zu beachten. Theoretisch kann man jeden beliebigen Dateinamen wählen, solange die Entwicklungsumgebung die Datei nicht beachtet.

Es hat sich durchgesetzt, diese Datei `.gitignore` oder `.gitkeep` zu nennen.<sup>1</sup> Die `.gitignore`-Datei wird normalerweise von Git benutzt, um bestimmte Dateien von der Versionierung auszuschließen. Das erweist sich gleich auch noch als nützlich.

Als Beispiel soll das Verzeichnis `EmptyDir` aus Abbildung 15–1 dienen. Mit dem Unix-Kommando `touch` legt man eine leere Datei an.

```
> cd projecta/EmptyDir
> touch .gitignore
```

---

<sup>1</sup>Dateien, die mit einem Punkt beginnen, werden von Unix-Systemen als versteckte Dateien betrachtet und von vielen Entwicklungsumgebungen ebenso ignoriert.

Häufig werden in solchen leeren Verzeichnissen temporäre Dateien angelegt, z.B. Build-Ergebnisse. Um zu verhindern, dass diese zukünftig aus Versehen in ein Commit einfließen, kann man in der Datei `.gitignore` eine Zeile mit einem `/*` einfügen. Dadurch werden alle Dateien in diesem Verzeichnis ignoriert und tauchen nicht beim status-Befehl als »untracked« auf. Die Zeile mit dem `/*` würde aber auch verhindern, dass die Datei `.gitignore` initial versioniert wird. Deswegen muss man sie wieder explizit vom Ignorieren ausnehmen.

Hier sehen Sie den Inhalt einer typischen `.gitignore`-Datei für leere Verzeichnisse:

```
# Alle Dateien bis auf .gitignore ignorieren  
*  
!.gitignore
```

## Schritt 2: Unnötige Dateien und Verzeichnisse ignorieren

Entwicklungs- und Build-Werkzeuge erzeugen häufig temporäre Dateien, z.B. `class`-Dateien in Java. Diese Dateien sollen nicht mitversioniert werden. Um das zu verhindern, legt man eine Datei `.gitignore` an und schließt alle nicht erwünschten Dateien und Verzeichnisse aus. Die `.gitignore`-Datei kann in jedem Verzeichnis angelegt werden. Die Einträge wirken immer ab dieser Ebene und in allen Unterverzeichnissen.

Dabei wird in jeder Zeile der Datei `.gitignore` ein Muster für einen Dateinamen angegeben. Im Beispiel aus Abbildung 15–1 sollen das Verzeichnis `TempDir` und alle Dateien mit der Endung `.bak` ausgeschlossen werden.

```
#Inhalt von .gitignore  
/TempDir  
*.bak
```

Um einfach nachvollziehen zu können, welche Dateien ignoriert werden, hat es sich bewährt, nur in der Wurzel des Projekts die `.gitignore`-Datei anzulegen. Auch tiefere Unterverzeichnisse können dort ausgeschlossen werden. Eine Ausnahme bilden nur die `.gitignore`-Dateien in leeren Verzeichnissen, wo die Datei für die Versionierung sorgt.

**Mit `.gitignore`  
Dateien unversioniert  
lassen → Seite 46**

*Tipp: Legen Sie nur eine `.gitignore`-Datei im Wurzelverzeichnis an.*

## Schritt 3: Behandlung der Zeilenenden definieren

Vor dem eigentlichen Import der Dateien muss noch entschieden werden, wie mit den Zeilenenden von Textdateien umgegangen werden soll.

Probleme mit Zeilenenden treten immer auf, wenn man gleichzeitig auf verschiedenen Betriebssystemen entwickelt bzw. Textdateien von verschiedenen Betriebssystemen benutzt.

Windows nutzt CRLF (Carriage Return and Line Feed), um Zeilenumbrüche zu codieren. Unix-Systeme und Mac-Rechner nutzen LF (Line Feed) für Zeilenumbrüche. In der Vergangenheit hatten Texteditoren auf den verschiedenen Plattformen Probleme, mit den Zeilenumbrüchen der anderen Plattformen umzugehen. Mittlerweile ist dieses Problem größtenteils gelöst.

Doch noch immer passiert es, dass ein Texteditor ohne oder mit Wissen des Benutzers die Zeilenumbrüche an die jeweilige Plattform anpasst. Das wiederum führt dazu, dass Git eine Zeile als geändert erkennt, obwohl inhaltlich nichts passiert ist. Man kann sich gut vorstellen, wie viele Merge-Konflikte daraus entstehen.

Git bietet als Lösung des Problems an, die Zeilenumbrüche im Repository auf LF zu standardisieren, d. h., wenn die Standardisierung aktiviert ist, wandelt Git bei jedem `commit`-Befehl alle Zeilenenden in LF um und beim Herausholen, wenn gewünscht, in den jeweils plattformabhängigen Standard.

Es gibt insgesamt drei verschiedene Möglichkeiten, mit Zeilenenden umzugehen:

**core.autocrlf false:** Die Zeilenenden werden nicht beachtet. Git speichert die Zeilenenden im Repository, so wie sie in der Datei vorliegen. Auch beim Herausholen bleiben die Zeilenenden unverändert.  
**core.autocrlf true:** Die Zeilenenden werden beim Hinzufügen standardisiert (LF) und beim Herausholen an die jeweilige Plattform angepasst.

**core.autocrlf input:** Die Zeilenenden werden beim Hinzufügen standardisiert (LF), aber beim Herausholen nicht angepasst.

Da man meistens nicht ausschließen kann, dass ein Repository in Zukunft auch auf anderen Plattformen benutzt wird, ist es sinnvoll, von Anfang an mit standardisierten Zeilenumbrüchen zu arbeiten.

Auf Windows-Systemen ist deswegen die Einstellung `true` und auf Unix-Systemen die Einstellung `input` vor dem ersten Import zu setzen:

```
> git config --global core.autocrlf input
```

Sowohl bei `true` als auch bei `input` kann es zu Problemen kommen, wenn Git eine Datei als Textdatei erkennt und diese anpasst, die Datei aber eigentlich binär ist. Mithilfe der Datei `.gitattributes` kann man die automatische Erkennung überschreiben und für jede einzelne Datei bzw. für jede Dateiendung das Format separat definieren.<sup>2</sup> Nachfolgend sehen Sie exemplarisch eine `.gitattributes`-Datei:

---

<sup>2</sup> Mit der `.gitattributes`-Datei kann man auch das Merge-Verhalten ändern. Für Einzelheiten schauen Sie in der Hilfe nach.

```
# Zeilenendenbehandlung für alle Textdateien aktivieren  
* text=auto  
# SVGs als Textdateien und PDFs als Binärdateien behandeln  
*.svg text  
*.pdf binary
```

#### Schritt 4: Repository anlegen und Dateien importieren

Nachdem in den vorigen Schritten der Import der Projektdateien vorbereitet wurde, soll in diesem Schritt das Repository angelegt werden:

```
> cd projecta  
> git init
```

Anschließend werden alle Dateien für das erste Commit mit dem add-Befehl hinzugefügt. Dabei werden alle vorhandenen Dateien, inklusive der .gitignore-Dateien, ins Commit aufgenommen und die ignorierten Dateien weggelassen.

Vor dem add-Befehl ist es sinnvoll, noch einmal mit dem status-Befehl zu überprüfen, welche Dateien als »untracked« erkannt werden. Manchmal vergisst man eine temporäre Datei oder ein Verzeichnis beim Ignorieren und fügt es so ungewollt zum Repository hinzu.

```
> git status  
> git add .
```

Zum Abschluss wird das Commit mit dem commit-Befehl abgeschlossen:

```
> git commit -m "init"
```

**Commits zusammenstellen**  
→ Seite 39

*Tipp: Überprüfen Sie vor dem add-Befehl mit dem status-Befehl die Dateien.*

#### Zentrale Repository-Verwaltung

Bisher existiert ein normales Repository mit Workspace für das neue Projekt. Um auf diesem Repository mit pull und push-Befehlen zu arbeiten, muss das Repository noch als Bare-Repository über eine zentrale Repository-Verwaltung zur Verfügung gestellt werden.

Es gibt verschiedene kommerzielle und freie Repository-Verwaltungen. Die meisten Lösungen gibt es als Hosting-Variante oder zum Selberbetreiben im Unternehmen (On-Premise).

Nachfolgend eine unvollständige Liste von Repository-Verwaltungen:

**GitHub** ist der Urvater der Git-Hosting-Lösungen. Viele Open-Source-Projekte werden dort verwaltet. Für den Einsatz im Unternehmen gibt es kommerzielle Lizenzenten und eine Enterprise-Variante zum Selberbetreiben<sup>3</sup>.

<sup>3</sup> <https://github.com>

**GitLab** ist eine freie Open-Source-Lösung, die meistens On-Premise betrieben wird. Es gibt auch eine Hosting-Variante und verschiedene kommerzielle Zusatz-Features<sup>4</sup>.

**Atlassian Bitbucket** ist die Hosting-Lösung von Atlassian und Bitbucket Server ist die On-Premise-Lösung. Bitbucket (Server) integriert sich sehr gut mit den anderen Werkzeugen von Atlassian, z. B. Jira<sup>5</sup>.

Im Detail unterscheiden sich die Repository-Verwaltungen, doch meistens bieten sie mindestens folgende Features:

- Eine Benutzerverwaltung erlaubt das Anlegen, Löschen und Bearbeiten von Benutzern und dessen Rechten.
- Bare-Repositorys können angelegt, gruppiert und verschoben werden.
- Auf Bare-Repositorys kann mit HTTP(S) und SSH zugegriffen werden.
- Commits, Branches und Inhalte können betrachtet und durchsucht werden.
- Es können Lese- und Schreibrechte auf Repositorys vergeben werden. Dedizierte Schreibrechte auf Branches sind möglich.
- Pull-Requests/Merge-Requests werden unterstützt und ermöglichen den Feature-Branch-Workflow.

**Mit Feature-Branches entwickeln** → Seite 143

### Schritt 1: Neues zentrales Repository anlegen

Typischerweise benötigt man in der zentralen Repository-Verwaltung einen eingerichteten Benutzer, der das Recht hat, neue Repositorys anzulegen. Bei den gehosteten Varianten von GitHub, GitLab und Bitbucket kann man sich einfach einen neuen Benutzer einrichten. Jeder neue Benutzer hat das Recht, neue Repository anzulegen.

In Abbildung 15–2 sieht man exemplarisch, wie das Anlegen eines Repositorys in GitHub erfolgt. Die Auswahl: PUBLIC (öffentlich) oder PRIVATE (private) entscheidet darüber, ob das Klonen des Repositorys für jeden erlaubt ist oder durch Rechte geschützt werden kann.

In unserem Workflow ist es wichtig, die letzte Checkbox INITIALIZE THIS REPOSITORY WITH A README nicht auszuwählen, da ansonsten kein leeres Repository angelegt wird, sondern eines mit einer README-Datei.

---

<sup>4</sup> <https://gitlab.com>

<sup>5</sup> <https://de.atlassian.com/>

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner **Repository name**

kapitel26 /  ✓

Great repository names are short and memorable. Need inspiration? How about `curly-octo-disco`.

Description (optional)

**Public**  
Anyone can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

**Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** | Add a license: **None** ⓘ

**Create repository**

**Abb. 15-2**  
Repository anlegen in GitHub

## Schritt 2: Lokales Repository in das zentrale Repository kopieren

Nach dem Bestätigen des CREATE REPOSITORY-Knopfes legt GitHub ein neues leeres Repository an. Im konkreten Beispiel legt GitHub das `projecta`-Repository unter der URL: <https://github.com/kapitel26/projecta.git> an.

Als Nächstes müssen die Daten des lokalen Repositorys in das neue zentrale Repository kopiert werden. Dazu konfigurieren wir in unserem lokalen Repository die GitHub-URL als Remote:

```
> git remote add origin https://github.com/kapitel26/projecta.git
```

Danach kann man mit dem `push`-Befehl den `master`-Branch inklusive aller Commits übertragen:

```
> git push --set-upstream origin master
```

**--set-upstream:** Der lokale `master`-Branch wird dauerhaft mit dem `origin/master`-Branch verknüpft. Dadurch reicht zukünftig ein einfacher `push`-Befehl aus.

### Schritt 3: Das zentrale Repository klonen

Die anderen Teammitglieder können nun das Repository klonen. Dazu wird einfach die URL zu dem zentralen Repository angegeben.

```
> git clone https://github.com/kapitel26/projecta.git
```

Möchte man anstelle von HTTP(S) SSH benutzen, dann kann man sich die notwendige URL aus GitHub kopieren. Das Klonen würde dann so aussehen:

```
> git clone git@github.com:kapitel26/projecta.git
```

Spätestens wenn man über SSH einen Push durchführen will, muss man in der zentralen Repository-Verwaltung die eigenen öffentlichen SSH-Schlüssel hinterlegen.

## Warum nicht anders?

### Warum nicht auf eine zentrale Repository-Verwaltung verzichten?

Die Alternative zu einer Repository-Verwaltung besteht darin, Bare-Repositories über ein Netzwerklauwerk, einen Webserver oder eine SSH-Infrastruktur zu teilen.

Dabei unterstützt Git verschiedene Protokollvarianten:

**file:** Zugriff über geteiltes Netzlaufwerk

**git:** proprietärer Serverdienst mit Netzwerkkommunikation

**http:** Zugriff über einen Webserver

**ssh:** Zugriff über eine Secure-Shell-Infrastruktur

Bei der Eigenbau-Variante entsteht typischerweise bei jedem neuen Repository der Aufwand des Einrichtens. Die Rechteverwaltung der Benutzer erfolgt über die zugrunde liegende Technologie (Dateisystem, Webserver, SSH). Pull-Requests/Merge-Requests werden nicht über ein Werkzeug unterstützt.

Wenn Sie nur sehr wenige Repositorys haben, diese nur mit wenigen Entwicklern, ohne Rechteeinschränkungen, verwenden und keine Pull-Requests benötigen, können Sie über Eigenbau nachdenken. Die meisten Teams sind besser beraten, eine Repository-Verwaltung einzusetzen.

Falls Sie den Eigenbau angehen, benötigen Sie als Erstes ein Bare-Repository, welches Sie dann auf der jeweiligen Technologie bereitstellen. Ein Bare-Repository besteht nur aus dem Inhalt des .git-Verzeichnisses und hat keinen Workspace.

Die Umwandlung eines normalen Repositorys erfolgt mithilfe des `clone`-Befehls und dem Parameter `--bare`. Bare-Repositorys bekommen typischerweise die Endung `.git`, um sie von normalen Repositorys zu unterscheiden:

```
> git clone --bare projecta projecta.git
```

**--bare:** Der Klon soll keinen Workspace bekommen, sondern nur die Repository-Objekte beinhalten.

**projecta:** Dieser Parameter ist der Name des vorbereiteten Repositorys.

**projecta.git:** Dieser Parameter ist der Name des neu zu erzeugenden Bare-Repositorys.

Das entstandene Bare-Repository kann man nun auf ein Netzwerklauwerk oder einen SSH-Pfad kopieren. Im Falle eines Webservers muss ein CGI-Skript (`git-core`) im jeweiligen Webserver eingerichtet werden.



## 16 Gemeinsam auf einem Branch entwickeln

Bei einer zentralen Versionsverwaltung, wie z. B. Subversion<sup>1</sup>, arbeiten die Teams häufig auf einem gemeinsamen Branch. Dieser Branch wird von jedem Entwickler in seinen lokalen Workspace kopiert. Dort finden dann Änderungen statt, die anschließend in die Versionsverwaltung zurückgeschrieben werden.

Einen ganz ähnlichen Workflow kann man auch mit Git umsetzen. Diese Ähnlichkeit erleichtert den Umstieg und macht die Benutzung unkompliziert und schnell.

Jeder Entwickler legt einen Klon des zentralen Repositorys an und arbeitet auf seiner Kopie des master-Branch.

Sobald die Entwicklungsergebnisse anderen zur Verfügung gestellt werden sollen, wird der lokale master-Branch mit dem zentralen master-Branch vereinigt. Dabei entstehen Merge-Commits, wenn es in der Zwischenzeit Änderungen anderer Entwickler gab. Der Merge erfolgt im lokalen Repository, und der vereinigte Stand wird anschließend in das zentrale Repository übertragen.

Der Vorteil dieses Workflows ist das schnelle Erkennen von Konflikten, da regelmäßig und zeitnah die eigenen Änderungen mit den Änderungen anderer Entwickler vereinigt werden.

Andererseits entstehen bei diesem Workflow viele Merge-Commits, und damit wird die Commit-Historie unübersichtlich. Es ist auch nicht mehr möglich, die First-Parent-History sinnvoll einzusetzen, die im Workflow »Mit Feature-Banches entwickeln« (Seite 143) gezeigt wird.

Der Workflow in diesem Kapitel beschreibt,

- wie die lokale Entwicklung auf dem master-Branch funktioniert,
- wie die eigenen Ergebnisse im zentralen Repository veröffentlicht werden und
- wie die Ergebnisse anderer Entwickler benutzt werden können.

**Branches zusammenführen**  
→ Seite 67

<sup>1</sup><http://subversion.apache.org/>

## Überblick

Abbildung 16–1 zeigt ein typisches Szenario für diesen Workflow. Oben ist das zentrale Repository zu sehen, links unten und rechts unten jeweils ein Entwickler-Repository (Entwickler A und Entwickler B).

**Commits zusammenstellen**  
→ Seite 39

Jeder Entwickler beginnt auf seinem lokalen `master`-Branch mit der Arbeit. Er erzeugt kleinschrittige Commits, um notfalls zu einem alten Stand zurückkehren zu können und um seine Arbeitsschritte zu dokumentieren.

**Austausch zwischen Repositorys** → Seite 95

Nachdem eine Aufgabe abgeschlossen ist oder wenn ein anderer Entwickler den Zwischenstand benötigt, werden mit dem `push`-Befehl die Commits in das zentrale Repository übertragen. Solange sich der zentrale `master`-Branch in der Zwischenzeit nicht verändert hat, wird der `push`-Befehl erfolgreich sein.

**Branches zusammenführen**  
→ Seite 67

Typischerweise wird es aber Commits anderer Entwickler auf dem zentralen `master`-Branch geben. Deswegen muss der `pull`-Befehl benutzt werden, um die Änderungen des zentralen Repositorys mit dem lokalen Repository zu vereinigen. Dabei entsteht ein Merge-Commit im lokalen Repository, das dann mit dem `push`-Befehl in das zentrale Repository übertragen wird. Das ist in Abbildung 16–1 gut an dem letzten Merge-Commit zu sehen. Der Entwickler A hatte bereits sein Commit übertragen, und der Entwickler B musste sein Commit mit dem vorhandenen Commit vereinigen.

Der `pull`-Befehl kann auch jederzeit benutzt werden, um die Änderungen der anderen Entwickler in das eigene Repository zu holen.

## Voraussetzungen

»Schöne« Historie nicht notwendig: Die Commit-Historie wird nur als Sicherheitsnetz gegen Datenverlust und für Vergleiche mit alten Versionsständen benötigt. Im Workflow »Mit Feature-Banches entwickeln« (Seite 143) sind andere Einsatzmöglichkeiten der Commit-Historie beschrieben.

**Integration mit Jenkins** → Seite 263

Continuous Integration für den zentralen `master`-Branch: Bei diesem Workflow entstehen viele Merge-Commits, und damit besteht immer auch die Gefahr, dass es zu Problemen mit den zusammengeführten Versionsständen kommt. Deswegen ist es wichtig, den `master`-Branch des zentralen Repositorys kontinuierlich zu bauen und zu testen, um die Entwickler schnell bei Problemen zu informieren.

Workflow kompakt

## Gemeinsam auf einem Branch entwickeln

Alle Entwickler arbeiten auf dem gleichen Branch in ihren lokalen Repositorys und integrieren die Ergebnisse in den Haupt-Branch des zentralen Repository.

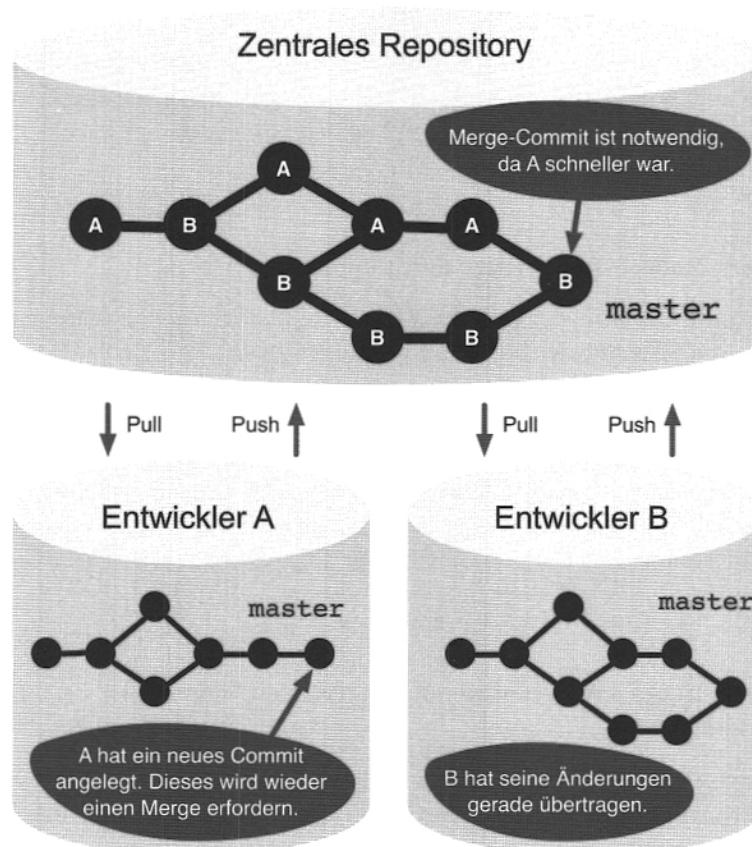


Abb. 16-1  
Workflow im Überblick

## Ablauf und Umsetzung

Für die folgenden Abläufe gehen wir von einem zentralen Repository und einem lokalen Klon für die Entwicklung aus.

### Auf dem master-Branch arbeiten

#### Schritt 1: master-Branch aktualisieren

Bevor man mit einer neuen Aufgabe beginnt, ist es immer sinnvoll, den neuesten Stand aus dem zentralen Repository zu holen. Dadurch werden mögliche Konfliktszenarien an Dateien minimiert.

```
> git pull
```

#### Schritt 2: Lokale Änderungen durchführen

##### Commits zusammenstellen

→ Seite 39

Nachdem der aktuelle Stand des Master-Repositorys geholt worden ist, kann die lokale Entwicklung durchgeführt werden.

Dabei werden in Git typischerweise kleinschrittige Commits angelegt, die genau eine Teilaufgabe, eine Refaktorisierung oder eine Fehlerbehebung umfassen. Dadurch wird es dem Entwickler einfach möglich sein, auf einen alten Stand zurückzugehen oder Dateien mit aktuellen Versionen zu vergleichen. Vor dem nächsten Schritt sollten immer die lokalen Änderungen durch ein Commit abgeschlossen werden:

```
> git commit -m "Methode X überarbeitet"
```

#### Schritt 3: Lokale Änderungen mit den zentralen Änderungen zusammenführen

Wenn die Entwicklungsaufgaben abgeschlossen sind oder wenn ein anderer Entwickler einen Zwischenstand einfordert, müssen die lokalen Änderungen in das zentrale Repository übertragen werden.

Da dies nur funktionieren wird, wenn es in der Zwischenzeit keine Änderungen im zentralen Repository gab, ist es immer sinnvoll, vorab mit dem `pull`-Befehl alle zentralen Änderungen zu holen:

```
> git pull
```

Already up-to-date.

Falls es keine Änderungen im zentralen Repository gab, wird das durch `up-to-date` mitgeteilt.

Wenn es Änderungen gibt, aber der Merge automatisch durchgeführt werden kann, wird Git keinen Konflikt (CONFLICT) melden und am Ende eine Statistik mit den geänderten Dateien ausgeben:

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From projektX
  2cd173f..e10bb4d master      -> origin/master
Merge made by recursive.
 foo | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Wenn es Konflikte zwischen den zentralen und den lokalen Änderungen gibt, wird das in der Ausgabe mit einem CONFLICT gekennzeichnet:

```
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From projektX
  9139636..fa60160 master      -> origin/master
Auto-merging foo
CONFLICT (content): Merge conflict in foo
```

Die Konfliktlösung kann mit den normalen Mitteln erfolgen. Die bereinigten Dateien werden durch den add-Befehl dem bisher unvollständigen Merge-Commit hinzugefügt. Anschließend muss der konfliktbehafte Merge-Versuch noch mit dem commit-Befehl abgeschlossen werden:

```
> git add foo
> git commit
```

Wenn man bei diesem commit-Befehl keine Beschreibung übergibt, wird Git automatisch eine Merge-Beschreibung inklusive der aufgetretenen Konflikte generieren:

```
Merge branch 'master' of projektX

Conflicts:
  foo
```

#### Schritt 4: Lokale Änderungen in das zentrale Repository übertragen

Wenn der vorige Schritt erfolgreich abgeschlossen wurde, gibt es einen konsolidierten Projektstand im lokalen Repository.

Bevor dieser Stand nun in das zentrale Repository übertragen wird, sollten unbedingt lokale Tests stattfinden, um Probleme aufzudecken.

**Bearbeitungskonflikte**

→ Seite 70

**Commits**

**zusammenstellen**

→ Seite 39

*Tipp: Führen Sie lokale Tests durch.*

Wenn man mit der Qualität der Version zufrieden ist, können mit dem push-Befehl die lokalen Änderungen in die Zentrale übertragen werden:

```
> git push
```

Wenn der push-Befehl ohne Fehlermeldung beendet wird, sind die Commits erfolgreich im zentralen Repository angekommen.

#### **Konflikte beim Push**

→ Seite 101

Hat in der Zwischenzeit jedoch ein anderer Entwickler Commits übertragen, wird der push-Befehl eine Fehlermeldung liefern:

```
To projektX.git
! [rejected]           master -> master (non-fast-forward)
error: failed to push some refs to '/Users/rene/temp/projekt.git/'
To prevent you from losing history, non-fast-forward updates were
rejected. Merge the remote changes (e.g. 'git pull') before pushing
again. See the 'Note about fast-forwards' section of
'git push --help' for details.
```

Um die neuen Änderungen zu holen, ist wieder der pull-Befehl notwendig. Dabei wird ein weiterer Merge-Commit angelegt.

```
> git pull
```

Anschließend wiederholt man diesen Schritt 4 und startet mit einem neuen push-Befehl.

## Warum nicht anders?

### Warum kein Rebase anstelle von Merge?

#### **Mit Rebasing die Historie glätten**

→ Seite 81

Der beschriebene Workflow erzeugt viele Merge-Commits, und damit wird die Commit-Historie schwer lesbar.

Eine Alternative wäre es, die lokalen Änderungen durch ein Rebase mit den zentralen Änderungen zu vereinigen (Parameter --rebase beim pull-Befehl):

```
> git pull --rebase
```

Bei einem Rebase werden die lokalen Änderungen Commit für Commit noch mal auf dem zentralen master-Branch ausgeführt. Das heißt, es entstehen neue Commits, die jedoch die gleichen Änderungen beinhalten.

In Abbildung 16–2 sind die unterschiedlichen Commit-Historien dargestellt, die bei Merge und Rebase entstehen würden.

Auf den ersten Blick fällt auf, dass es beim Rebasing keine Verzweigung mehr gibt und somit die Historie linear ist. Diesem Vorteil steht

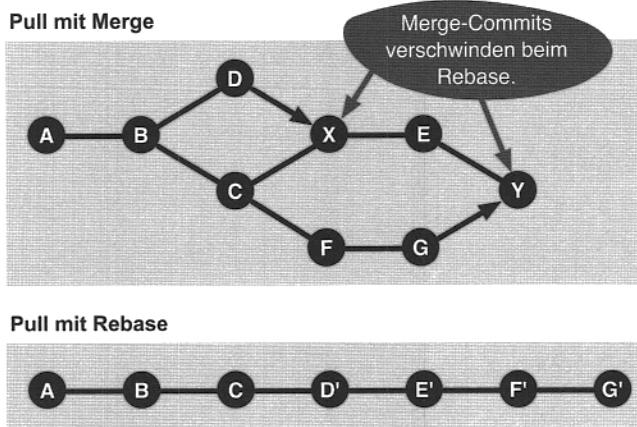


Abb. 16–2  
Rebase anstelle von  
Merge

entgegen, dass es jetzt Commits in der Historie gibt, die so niemals ein Entwickler in seiner lokalen Umgebung erarbeitet hat. Das passiert immer dann, wenn durch ein Rebase mehrere Commits kopiert werden. Dann wird der Entwickler nur das letzte Commit überprüfen, und die mitkopierten Vorgänger werden nicht getestet.

Als Beispiel soll das F'-Commit dienen (Abbildung 16–2 unten). Es ist durch ein Rebase des G-Commits entstanden. Solange es keine Konflikte beim Kopieren der Änderungen des F-Commits gibt, wird dieses nicht beachtet werden. Es ist jedoch denkbar, dass es inhaltliche Fehler gibt, die dazu führen, dass dieser Versionsstand nicht funktioniert.

Solange man nur wissen will, wer welche Änderungen wann eingebaut hat, sind diese »unbekannten« Commits irrelevant. Doch sobald man auf Fehlersuche in der Commit-Historie geht, können diese Commits stören.

Für den Einstieg in Git ist Merge definitiv einfacher zu vermitteln. Ist das Team sicher im Umgang mit Git und ist das Rebasing-Konzept von jedem verstanden worden, führt der Einsatz von Rebase zu einfacheren Historien und sollte in Betracht gezogen werden.

Falls man dauerhaft mit Rebase arbeiten möchte, dann kann man durch die folgende Konfigurationseinstellung Rebase als Standardverhalten des `pull`-Befehls definieren:

```
> git config branch.master.rebase true
```

**branch.master.rebase:** Dieser Parameter bestimmt, für welchen Branch Rebase als Standard aktiviert werden soll. Der mittlere Teil `master` kann durch andere Branch-Namen ersetzt werden. Ohne Angabe des Branch-Namens gilt Rebase für alle Branches.

**Inhaltliche Konflikte**  
→ Seite 69

*Tipp: So definieren Sie Rebase als Standard.*



## 17 Mit Feature-Branches entwickeln

Wenn alle im Team auf einem gemeinsamen Branch entwickeln, entsteht eine sehr unübersichtliche History mit vielen zufälligen Merge-Commits. Dadurch wird es schwierig, Änderungen für ein bestimmtes Feature oder einen Bugfix<sup>1</sup> nachzuvollziehen.

Während der Entwicklung von Features sind kleinschrittige Commits hilfreich, um jederzeit auf einen alten funktionierenden Stand zurückzufallen. Doch wenn man sich einen Überblick über die im Release enthaltenen neuen Features verschaffen will, sind grobgranulare Commits sinnvoller. Bei diesem Workflow werden die kleinschrittigen Commits auf dem *Feature-Branch* und die Release-Commits auf dem *master-Branch* angelegt.

Die Integration von Feature-Branches wird mithilfe von *Pull-Requests* durchgeführt. Dadurch können Code-Reviews und weitere Qualitätsmaßnahmen, wie Builds und Tests, vor dem eigentlichen Merge in den *master-Branch* durchgeführt werden.

Dieser Workflow zeigt, wie Feature-Branches so eingesetzt werden, dass

- die Entwicklung von Features untereinander entkoppelt ist,
- Pull-Requests für Code-Reviews und andere Qualitätsmaßnahmen benutzt werden können,
- die Commits, die ein Feature implementieren, einfach aufzufinden sind,
- die First-Parent-History des *master-Branch* nur grobgranulare Feature-Commits beinhaltet, die als Release-Dokumentation dienen können,
- wichtige Änderungen des *master-Branch* während der Feature-Entwicklung benutzt werden können.

**Gemeinsam auf einem Branch entwickeln**

→ Seite 135

**Commits zusammenstellen**

→ Seite 39

---

<sup>1</sup>In Git werden Features und Bugs unter dem Begriff *Topic* zusammengefasst. Entsprechend wird häufig auch von *Topic-Branches* gesprochen.

## Überblick

Abbildung 17-1 zeigt die Grundstruktur, die beim Arbeiten mit Feature-Branches entsteht. Ausgehend vom master-Branch wird für jedes Feature oder jeden Bugfix (nachfolgend werden Bugfixes nicht mehr explizit aufgeführt) ein neuer Branch angelegt. Dieser Branch wird benutzt, um alle Änderungen und Erweiterungen durchzuführen. Sobald das Feature in den master-Branch integriert werden soll, wird ein Pull-Request angelegt. Pull-Requests können vor dem Merge mit dem master-Branch durch verschiedene Qualitätsmaßnahmen (Code-Reviews, Builds, Tests) überprüft werden.

Der Austausch zwischen Feature-Branches findet ausschließlich über den master-Branch statt. Gibt es Abhängigkeiten zwischen Features, so muss das Feature, welches Lieferungen für andere Features bereitstellen muss, zeitlich früher eingeplant werden. Wird die Entwicklung der anderen Features dadurch blockiert, so muss das Feature in kleinere Features aufgeteilt werden, sodass die wichtigen Lieferungen schneller erfolgen (siehe feature/a1 und feature/a2).

Der Entwickler eines Feature-Branch kann sich notwendige Neuerungen des master-Branch jederzeit durch einen Merge in den Feature-Branch holen.

## Voraussetzungen

**Feature-basiertes Vorgehen:** Die Planung des Projekts bzw. Produkts muss auf Features basieren, d. h., fachliche Anforderungen werden in Feature-Aufgabenpakete überführt. Features haben untereinander wenige Abhängigkeiten.

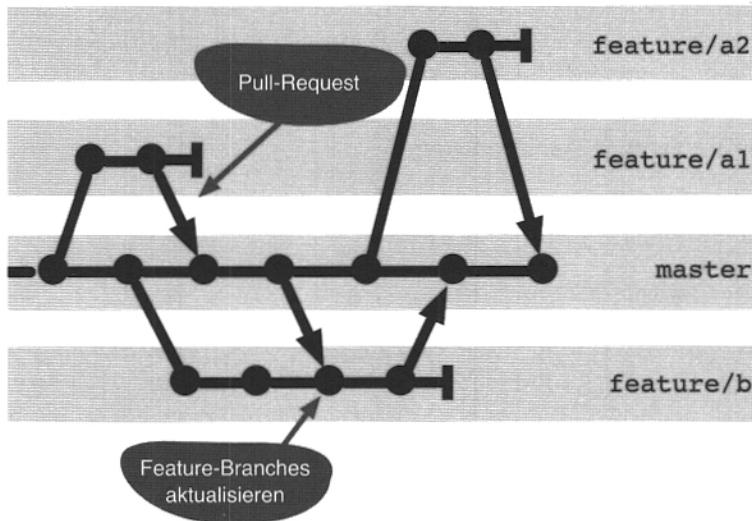
**Kleine Features:** Die Entwicklung eines Features muss in Stunden oder Tagen abgeschlossen werden können. Je länger die Feature-Entwicklung parallel zu der restlichen Entwicklung läuft, umso größer ist das Risiko, dass bei der Integration des Features große Aufwände entstehen.

**Zentrale Repository-Verwaltung mit Pull-Requests:** Im Projekt ist eine zentrale Repository-Verwaltung im Einsatz, die Pull-Requests unterstützt.

Workflow kompakt

## Mit Feature-Branches entwickeln

*Jedes Feature oder jeder Bugfix wird in einem separaten Branch entwickelt. Nach der Fertigstellung wird das Feature oder der Bugfix mithilfe eines Pull-Request in den master-Branch integriert.*



**Abb. 17-1**  
Workflow im Überblick

## Ablauf und Umsetzung

Für die folgenden Abläufe wird von einem zentralen Repository ausgegangen. Die Entwicklung findet wie immer in einem lokalen Klon statt. Das zentrale Repository wird im Klon über das Remote origin ange- sprochen.

### Feature-Branch anlegen

Sobald ein neues Feature bearbeitet werden soll, wird ein neuer Branch erzeugt. Dabei ist darauf zu achten, dass der Branch ausgehend vom master-Branch angelegt wird.

#### Schritt 1: master-Branch aktualisieren

Wenn gerade Zugriff auf das zentrale Repository besteht, ist es sinnvoll, als Erstes den master-Branch auf den neuesten Stand zu bringen. Dabei kann es zu keinen Merge-Konflikten kommen, da bei Feature-basierten Arbeiten im lokalen Repository nicht auf dem master-Branch gearbeitet wird.

**Ein Projekt aufsetzen**  
→ Seite 123  
**Wie sagt man Git, wo das Remote-Repository liegt?**  
→ Seite 91

**Fetch: Branches aus einem anderen Repository holen**  
→ Seite 99

```
> git checkout master
> git pull --ff-only
```

**--ff-only:** Nur ein Fast-Forward-Merge ist erlaubt. Das heißt, wenn lokale Änderungen vorliegen, wird der Merge abgebrochen.

Falls der Merge mit einer Fehlermeldung abbricht, dann wurde vorab aus Versehen direkt auf dem `master`-Branch gearbeitet. Diese Änderungen müssen als Erstes in einen Feature-Branch verschoben werden (»Commits auf einen anderen Branch verschieben« ab Seite 114).

### Schritt 2: Feature-Branch anlegen

**Branches verzweigen**  
→ Seite 59

Anschließend kann der neue Branch angelegt werden, und die Arbeit kann beginnen:

```
> git checkout -b feature/a1
```

*Tipp: Verwenden Sie einheitliche Namen für Branches.*

Es ist sinnvoll, sich im Team auf eine einheitliche Namensgebung von Feature- und Bugfix-Branches festzulegen. Git unterstützt auch hierarchische Namen für Branches, so werden Feature-Branches häufig mit dem Präfix `feature` begonnen.

Werden Features oder Bugfixes mit einem Tracking-Werkzeug verwaltet (z. B. Redmine<sup>2</sup>, Jira<sup>3</sup>), so können deren eindeutige Nummern oder Token für den Branch-Namen verwendet werden.

### Schritt 3: Feature-Branch zentral sichern

In diesem Workflow werden Pull-Requests verwendet. Damit Pull-Requests in der zentralen Repository-Verwaltung angelegt werden können, müssen die Feature-Branches auch zentral verfügbar sein.

**Austausch zwischen Repositorys** → Seite 95

Dazu wird der Branch im zentralen Repository mit dem `push`-Befehl angelegt:

```
> git push --set-upstream origin feature/a1
```

**--set-upstream:** Dieser Parameter verknüpft den lokalen Feature-Branch mit dem neuen Remote-Branch. Das heißt, zukünftig kann bei allen `push`- und `pull`-Befehlen auf ein explizites Remote verzichtet werden.

**origin:** Das ist der Name des Remote (der Alias für das zentrale Repository), auf dem der Feature-Branch gesichert werden soll.

---

<sup>2</sup> <http://www.redmine.org>

<sup>3</sup> <https://www.atlassian.com/software/jira>

Änderungen an dem lokalen Feature-Branch können zukünftig durch einen einfachen push-Befehl zentral gesichert werden:

```
> git push
```

## Alternative: Feature-Branch in zentraler Repository-Verwaltung anlegen

Die meisten zentralen Repository-Verwaltungen erlauben auch das Anlegen von Branches direkt über die Weboberfläche (Abbildung 17–2).

**Branch erstellen**

Repository: gitbook / jenkinsexample

Branchtyp: Feature

Weitere Informationen zu Branchtypen

Branch von: master

Branchname: feature/a1

Diagramm: master —> feature/a1

Buttons: Branch erstellen, Abbrechen

**Abb. 17–2**  
Feature-Branch in  
Weboberfläche anlegen

Anschließend einfach mit dem fetch-Befehl alle Remote-Branches aktualisieren. Dabei wird man den gerade angelegten Feature-Branch als neuen Remote-Branch sehen.

```
> git fetch
```

Mit dem checkout-Befehl erzeugt man den neuen lokalen Feature-Branch.

```
> git checkout feature/a1
```

Wenn Sie mit der Git-Bash arbeiten, können Sie sich die Eingabe des Branch-Namens erleichtern, indem Sie nach den ersten Buchstaben die TAB-Taste drücken. Es wird dann automatisch der Branch-Name vervollständigt. Wenn Sie zweimal TAB drücken, bekommen Sie alle passenden Branches angezeigt.

*Tipp: Tab-Completion bei Branch-Namen*

## Feature in den master-Branch integrieren

### **Warum**

**Feature-Branches  
nicht erst kurz vor dem  
Release integrieren?**

→ Seite 156

Wie wir bereits in den Voraussetzungen definiert haben, ist es wichtig, dass Features nicht zu lange parallel existieren. Ansonsten nimmt die Gefahr von Merge-Konflikten und inhaltlichen Inkompatibilitäten stark zu. Selbst wenn das Feature noch nicht in das nächste Release einfließen soll, ist es sinnvoll, die Integration zeitnah durchzuführen und besser mit einem Feature-Toggle die Funktionalität zu deaktivieren.

In diesem Abschnitt wird beschrieben, wie das Feature mithilfe eines Pull-Request in den master-Branch integriert wird. Pull-Requests (manchmal auch *Merge-Requests* genannt) werden von vielen zentralen Repository-Verwaltungen, z. B. Atlassian Bitbucket Server<sup>4</sup>, GitHub<sup>5</sup> oder GitLab<sup>6</sup>, unterstützt. Nachfolgend werden die Schritte exemplarisch mit Atlassian Bitbucket Server vorgestellt.

Pull-Requests kommen ursprünglich aus der Open-Source-Entwicklung, um kontrolliert Änderungen zu integrieren. Es wird kein einfacher lokaler Merge zwischen Branches durchgeführt, sondern eine Anfrage an einen Integrator bzw. an eine zentrale Repository-Verwaltung versandt, mit der Aufforderung, diesen Merge nach Überprüfung durchzuführen.

Im Kontext von Feature-Branches ermöglichen Pull-Requests die Integration des Feature in den master-Branch und können gleichzeitig die Qualität des Features vorab durch automatische Builds oder Reviews erhöhen.

### Schritt 1: Pull-Request anlegen

Nachdem das Feature entwickelt wurde, werden die Commits des Feature-Branch in die zentrale Repository-Verwaltung übertragen.

> git push

Dann kann man über eine Weboberfläche den Pull-Request anlegen. Dabei gibt man den Quell-Branch (Feature-Branch) und den Ziel-Branch (master-Branch) an (Abbildung 17-3).

### Schritt 2: Pull-Request bearbeiten

### **Pull-Requests bauen**

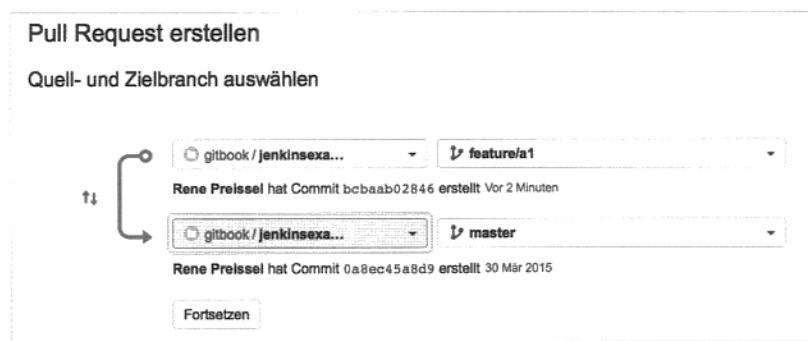
→ Seite 269

Als Ergebnis entsteht ein neuer Pull-Request, der wiederum über die Weboberfläche bearbeitet werden kann. Wenn erforderlich oder gewünscht, kann ein Code-Review auf den Änderungen des Pull-Request

<sup>4</sup> [www.atlassian.com/software/bitbucket/server](http://www.atlassian.com/software/bitbucket/server)

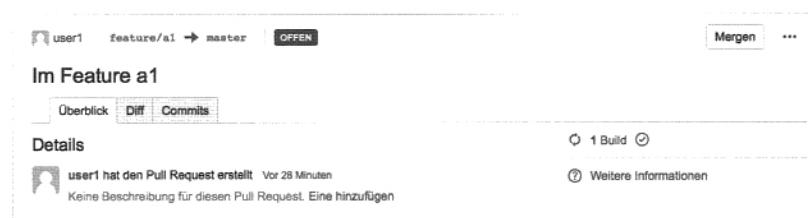
<sup>5</sup> <https://github.com>

<sup>6</sup> <https://gitlab.com/>



**Abb. 17-3**  
Pull-Request anlegen

durchgeführt werden<sup>7</sup>. Die Reviewer können direkt in der Weboberfläche den gesamten Pull-Request oder einzelne Codezeilen kommentieren und ihre Zustimmung oder Ablehnung für den Pull-Request hinterlegen. Wenn ein entsprechendes Build-System angebunden ist, kann der Pull-Request gebaut und getestet werden. Die Ergebnisse des Builds und der Tests werden wiederum am Pull-Request festgehalten (Abbildung 17-4).



**Abb. 17-4**  
Pull-Request bearbeiten

Wenn Nacharbeiten am Feature notwendig sind, weil ein Reviewer oder das Build-System Probleme festgestellt hat, müssen diese durch neue Commits auf dem Feature-Branch erledigt werden.

```
> git commit  
> git push
```

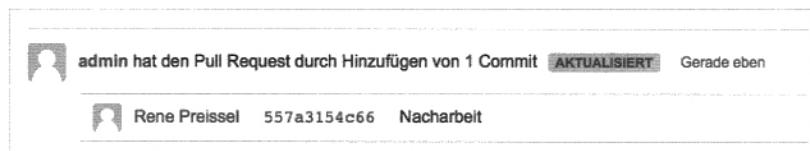
Sobald die neuen Commits durch Push in das zentrale Repository übertragen wurden, wird der Pull-Request aktualisiert, und es ist für jeden ersichtlich, dass Änderungen erfolgt sind (Abbildung 17-5).

Wenn ein Merge des Feature-Branch mit dem master-Branch wegen Konflikten in Dateien nicht möglich ist, wird das ebenso am Pull-Request angezeigt (Abbildung 17-6). Dann muss der Feature-Branch lokal mit dem aktuellen master-Branch vereinigt und das entstandene

<sup>7</sup> Einige Repository-Verwaltungen erlauben es, diesen Code-Review als zwingende Voraussetzung für das Mergen zu definieren.

**Abb. 17–5**

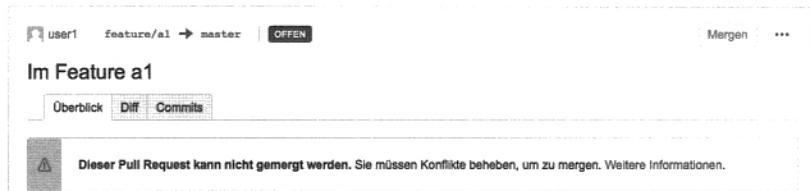
Pull-Request um neue Commits erweitern



Merge-Commit wiederum durch Push in das zentrale Repository übertragen werden (siehe »Änderungen des master-Branch in den Feature-Branch übernehmen« ab Seite 151).

**Abb. 17–6**

Konflikte im Pull-Request



### Schritt 3: Pull-Request abschließen

Sind alle Voraussetzungen erfüllt, kann der Pull-Request abgeschlossen werden, d.h., ein Merge wird durchgeführt. Dazu wird der entsprechende Webdialog gestartet (Abbildung 17–7).

**Abb. 17–7**

Pull-Request mergen



Alternativ kann man auch den Pull-Request ohne Merge beenden: In Bitbucket Server: ABLEHNEN bzw. in GitHub: CLOSE PULL REQUEST. Dann wird der Pull-Request ohne Merge geschlossen und es kann auf dem Feature-Branch weiter gearbeitet werden.

### Schritt 4: Feature-Branch löschen

Typischerweise wird nach dem Merge der Feature-Branch gelöscht. Das Löschen des zentralen Feature-Branch ist direkt über die Merge-Weboberfläche möglich (siehe Abbildung 17–7 unten). Alternativ kann

der remote Feature-Branch auch über die Kommandozeile gelöscht werden:

```
> git push --delete origin feature/a1
```

Das Löschen des lokalen Feature-Branch muss immer manuell passieren:

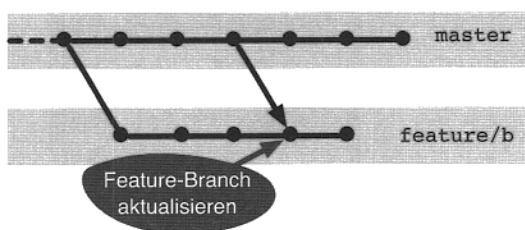
```
> git checkout master
> git branch -d feature/a1
```

**-d:** Löscht den übergebenen Branc.

## Änderungen des master-Branch in den Feature-Branch übernehmen

Im besten Fall findet die Entwicklung eines Features unabhängig von anderen Features statt.

Manchmal gibt es jedoch auf dem master-Branch wichtige Änderungen, die für die Entwicklung des Features notwendig sind, z. B. große Refaktorierungen oder Neuerungen an grundlegenden Services. Ein anderer typischer Fall ist, dass ein Pull-Request Konflikte zwischen dem master und dem Feature-Branch meldet. In solchen Situationen müssen die Änderungen des master-Branch in den Feature-Branch übernommen werden.



**Abb. 17-8**  
Änderungen aus dem  
Master-Branch in den  
Feature-Branch  
übernehmen

Abbildung 17–8 veranschaulicht die Situation. Es soll ein Merge vom master-Branch in den Feature-Branch durchgeführt werden:

### Schritt 1: origin/master-Branch aktualisieren

Als Erstes müssen die Änderungen des origin/master-Branch in das lokale Repository importiert werden. Der fetch-Befehl aktualisiert alle Remote-Branches:

```
> git fetch
```

### Schritt 2: Änderungen in den Feature-Branch übernehmen

**Branches zusammenführen**

→ Seite 67

Im zweiten Schritt müssen die Änderungen durch einen Merge in den Feature-Branch übernommen werden:

> `git merge origin/master`

Falls es zu Konflikten kommt, müssen diese mit den normalen Mitteln gelöst werden.

Der Zwischenstand kann beliebig oft vom master-Branch in den Feature-Branch übernommen werden. Git kann sehr gut mit mehrfachen Merges umgehen. Allerdings wird dadurch die Commit-Historie komplexer und schwerer lesbar.

## Warum nicht anders?

### Warum nicht auf Pull-Requests verzichten?

Kann man den Feature-Branch-Workflow nicht auch ohne Pull-Requests durchführen, d. h., die Integration des Feature-Branch manuell durchführen?

Das funktioniert natürlich auch. Dabei verliert man aber einerseits den Vorteil der automatischen und manuellen Prüfungen vor der Integration. Andererseits ist der manuelle Merge des Feature-Branch in den master-Branch nicht trivial, insbesondere wenn man eine richtige und vollständige First-Parent-History des master-Branch erzeugen will. Zusätzlich erhält man durch Pull-Requests in der Weboberfläche eine gute Übersicht über das, was im Projekt passiert.

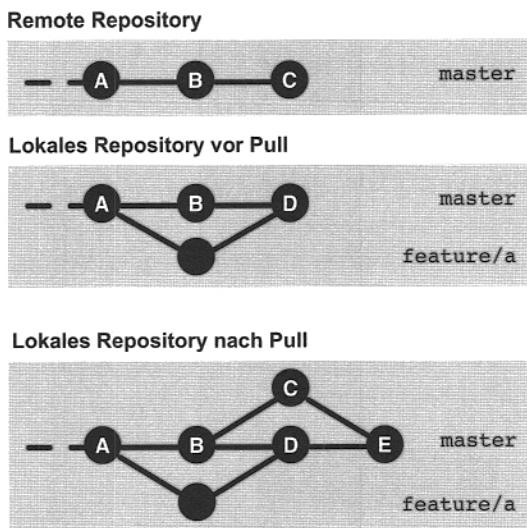
Der Punkt 1 ist leicht zu verstehen. Mit Pull-Request kann man ein Quality-Gate vor der Integration in den master-Branch errichten. Es kommen keine Commits in den master-Branch, die nicht den definierten Qualitätsansprüchen genügen.

Für Punkt 2 muss man sich z. B. das Szenario anschauen, wenn der lokale Merge des Feature-Branch in den master-Branch funktioniert hat, aber der nachfolgende Push abgewiesen wird. In Abbildung 17–9 oben und in der Mitte ist die beschriebene Situation skizziert: Remote wurde das c-Commit und lokal das d-Commit angelegt.

Würde man jetzt, wie typischerweise in solchen Situationen, einen `pull`-Befehl absetzen, dann entstünde ein neues Merge-Commit e (Abbildung 17–9 unten). Damit würde in der First-Parent-History des master-Branch das c-Commit nicht mehr enthalten sein.

In der First-Parent-History sollen aber alle Features enthalten sein, deswegen muss bei einem fehlgeschlagenen `push`-Befehl das lokale

**Branch-Zeiger umsetzen** → Seite 64



**Abb. 17-9**  
Keine verwendbare  
First-Parent-History  
nach dem Pull-Befehl

Feature-Merge-Commit (`D-Commit`) mit dem `reset`-Befehl entfernt und der Merge mit dem aktuellen `master`-Branch wiederholt werden:

```
> git reset --hard HEAD^
```

Mit genügend Sorgfalt ist dieser Ablauf auch ohne Pull-Requests möglich, doch ist es schwer zu überprüfen, ob jeder Entwickler zu jedem Zeitpunkt die richtige Operation durchführt. Das Ergebnis ist dann häufig eine kaputte bzw. inkorrekte First-Parent-History.

#### Git-Erweiterungen

### Git-Flow: High-Level-Operationen

*Git-Flow*<sup>8</sup> ist eine Sammlung von Skripten, um den Umgang mit Branches, insbesondere Feature-Branches, zu vereinfachen. Wenn Sie ohne Pull-Request arbeiten wollen, dann sollten Sie sich die Skripte anschauen.

So kann ein neuer Feature-Branch folgendermaßen erzeugt und gleichzeitig aktiviert werden:

```
> git flow feature start feature/a1
```

Am Ende kann der Feature-Branch in den `master`-Branch übernommen und gleichzeitig gelöscht werden.

```
> git flow feature finish feature/a1
```

<sup>7</sup> <https://github.com/nvie/gitflow>

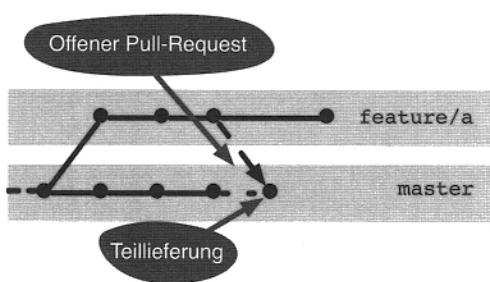
## Warum nach einem Pull-Request nicht auf dem Feature-Branch weiterarbeiten?

Wenn es Abhängigkeiten zwischen Features gibt und die Entwicklung des liefernden Feature lange dauert, teilt man ein Feature in Teillieferungen auf: Aus einem großen Feature werden mehrere kleine Features.

Sobald die erste Teillieferung fertig ist, kann ein Pull-Request erstellt werden, und sobald dieser abgeschlossen ist, können andere Features die Änderungen aus dem master-Branch holen.

Jetzt könnte der Entwickler der nächsten Teillieferung auf die Idee kommen, den bereits vorhandenen Feature-Branch für die weitere Entwicklung zu nutzen (Abbildung 17-10).

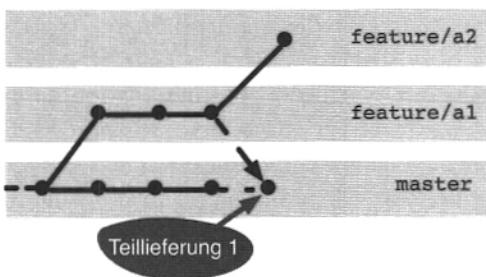
**Abb. 17-10**  
Wiederverwendung  
eines Feature-Branch



Das Problem entsteht dabei, sobald Commits für die neue Teillieferung durch Push in das zentrale Repository übertragen werden und der Pull-Request der ersten Teillieferung noch nicht abgeschlossen ist. Dann wird der noch offene Pull-Request aktualisiert und die noch unvollständigen Änderungen werden ggf. schon integriert. Genauso kommt es zu Problemen, wenn Nacharbeiten an der ersten Teillieferung notwendig werden und diese in Konflikt zu den neuen Änderungen stehen.

Deswegen sollte man einen Feature-Branch nicht für weitere Teillieferungen wiederverwenden, sondern einen neuen Branch anlegen. Sollte die Bearbeitung des ersten Pull-Request länger dauern und benötigt man für die zweite Teillieferung die Ergebnisse der ersten Teillieferung, dann ist es möglich, einen neuen Feature-Branch bei dem letzten Commit des ersten Feature-Branch zu beginnen (Abbildung 17-11).

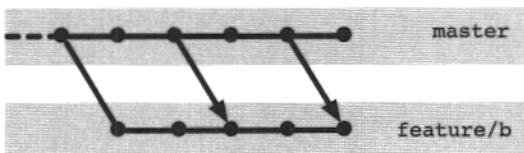
Sind noch Nacharbeiten notwendig, können diese autark auf dem ersten Feature-Branch durchgeführt werden. Die Änderungen sind dann ganz normal über den master-Branch in den zweiten Feature-Branch zu holen.



**Abb. 17-11**  
Zweite Teillieferung auf  
neuen Feature-Branch

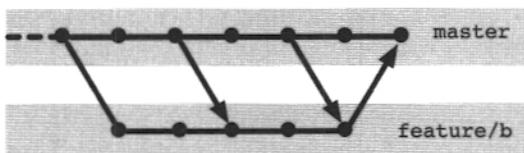
### Warum keinen Rebase des Feature-Branch vor dem Pull-Request durchführen?

Hat man ein größeres Feature entwickelt und mehrfach die Änderungen aus dem master-Branch geholt, entsteht eine Historie wie in Abbildung 17-12.



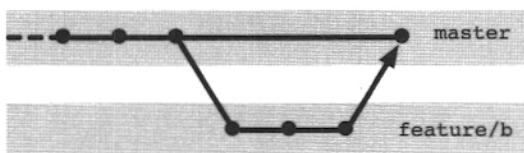
**Abb. 17-12**  
Feature-Branch mit  
mehrfacher Integration  
des master-Branch

Man sieht in dem Beispiel zwei Merge-Commits im Feature-Branch. Nach dem Pull-Request sieht es aus wie in Abbildung 17-13.



**Abb. 17-13**  
Viele Merge-Commits  
nach Integration

Würde man vor dem Erstellen des Pull-Request ein Rebase durchführen, verschwinden die Merge-Commits auf dem Feature-Branch und die Historie würde nach der erfolgreichen Integration wie in Abbildung 17-14 aussehen.



**Abb. 17-14**  
Feature-Branch nach  
Rebase und Integration

Der Vorteil dieses Vorgehens ist eine »schönere« Historie, in der man die Feature-Branches sehr gut erkennen kann. Der Nachteil ist, dass nach dem Rebase der `push`-Befehl mit der Option `--force` ausgeführt werden muss, d. h., die bisher vorhandene Historie des Feature-Branch wird überschrieben. Wenn mehr als ein Entwickler an dem Feature arbeitet, erfordert das eine Abstimmung im Team. Außerdem entstehen, wie bei jedem Rebase, Commits, die von keinem Entwickler überprüft worden sind.

Wenn alle im Team sich sehr gut mit Git auskennen und Rebase verstanden haben, ist diese Variante hilfreich für eine »schöne« Historie. Für den Einstieg mit Feature-Branches oder bei wechselnden Teammitgliedern ist die Gefahr von Fehlern vorhanden.

*Tipp: Force-Push für  
master-Branch  
verbieten*

Falls Sie sich für das Rebasen von Feature-Branches entscheiden, sollten Sie als Vorsichtsmaßnahme unbedingt alle Nicht-Feature-Branches (`master-Branch` etc.) vor dem versehentlichen Überschreiben mittels Force-Push schützen.

## Warum Feature-Branches nicht erst kurz vor dem Release integrieren?

Bei der Arbeit mit Feature-Branches kommt das Release-Management häufig auf die Idee, die Entscheidung, welche Features in das neue Release kommen sollen, erst kurz vor dem Liefertermin zu treffen.

Konzeptionell scheint das mit dem Feature-Branch-Ansatz auch sehr einfach zu gehen. Jedes Feature wird in einem Branch vollständig entwickelt, jedoch noch nicht in den `master-Branch` integriert. Kurz vor dem entscheidenden Tag wird erst beschlossen, welche Features in den `master-Branch` integriert werden sollen.

In einer idealen Welt – mit völlig unabhängigen Features und ohne Programmierfehler – wäre dieses Vorgehen auch anwendbar. In der Realität führt dieses Vorgehen jedoch meistens zu größeren Merge-Konflikten bei der Integration und zu langen Stabilisierungsphasen.

Außerdem wird es komplizierter, abhängige Features zu entwickeln. Normalerweise würde man bei Abhängigkeiten zwischen Features das erste Feature in den master-Branch integrieren und das andere Feature könnte sich die Änderungen holen. Bei der Lösung mit der späten Integration müssen die Feature-Banches direkt die Änderungen austauschen (siehe den folgenden Abschnitt). Damit wäre die unabhängige Integration dieser Features kurz vor dem Release unmöglich.

Auch bewährte Prozesse für qualitative Software, wie Continuous Integration und Refaktorierungen, sind bei der späten Integration kaum umzusetzen.

**Integration mit Jenkins** → Seite 263

## Warum nicht direkt Commits zwischen Feature-Banches austauschen?

Der beschriebene Workflow sieht keinen direkten Austausch von Commits zwischen Feature-Banches vor. Die Integration findet immer über Lieferungen im master-Branch statt.

Wäre es nicht einfacher, direkt zwischen Feature-Banches Merges durchzuführen?

Der entscheidende Vorteil von Feature-Banches ist die klare Zuordnung von Änderungen zu Features und die klare Zuordnung von Verantwortlichkeiten. Jeder Entwickler übernimmt die Verantwortung für seine Änderungen, und diese werden im Pull-Request durch Code-Reviews und andere Qualitätsmaßnahmen überprüft.

Würde man direkt zwischen Feature-Banches Commits austauschen, dann würde derjenige Entwickler, der als Erstes den Pull-Request stellt, alle Änderungen der anderen integrierten Features mit abgeben müssen. Er wäre also verantwortlich, dafür zu sorgen, dass der Code die entsprechende Qualität hat.

Abgesehen davon würde die Historie unübersichtlicher werden, und es wäre nicht mehr so einfach möglich, die Commits während der Feature-Entwicklung von jenen der Integration zu unterscheiden.

Schritt für Schritt

## Offene Feature-Branches anzeigen

*Es werden alle noch nicht mit dem master-Branch zusammengeführten Feature-Branches angezeigt.*

### 1. Offene Feature-Branches anzeigen

Arbeitet man konsequent mit Feature-Branches, hat man häufig mehr als einen aktiven Feature-Branch in seinem Repository. Mit dem branch-Befehl können alle noch nicht integrierten Branches angezeigt werden:

```
> git branch --no-merged master
```

**--no-merged master:** Zeigt alle Branches mit Commits an, die nicht im master-Branch enthalten sind, d. h. alle Feature-Branches, die noch nicht mit dem master-Branch zusammengeführt wurden.

Schritt für Schritt

## Integrierte Features anzeigen

*Es werden die zuletzt mit dem master-Branch zusammengeführten Feature-Branches angezeigt.*

### 1. Integrierte Features anzeigen

Arbeitet man konsequent mit Feature-Branches, erhält man mit der First-Parent-History des master-Branch eine Auflistung der letzten integrierten Pull-Requests:

```
> git log --first-parent --oneline master
```

**--first-parent:** Zeigt nur die Commits an, die über den ersten Parent eines Merge-Commit zu erreichen sind.

**--oneline:** Zeigt nur den abgekürzte Commit-Hash und die erste Zeile der Commit-Message an.

Schritt für Schritt

## Alle Änderungen eines integrierten Features anzeigen

*Alle Änderungen des Features werden als Diff angezeigt.*

Insbesondere bei nachträglichen Code-Reviews ist es wichtig, herauszufinden, welche Änderungen vorgenommen wurden.

In Abbildung 17-15 ist ein Beispiel eines Feature-Branch zu sehen. Nachfolgend werden die Commits entsprechend dieser Abbildung referenziert.

### 1. Commits des Features finden

Für einen Diff benötigt man alle Commits im master-Branch, die zu dem Feature gehören. Normalerweise wird dies nur ein Merge-Commit sein. Im Beispiel wird das Commit H gefunden.

```
> git checkout master
> git log --first-parent --oneline --grep="feature/a"
c52ce0a Pull Request für feature/a
```

--grep: Sucht in der Log-Meldung nach einem bestimmten Text.

### 2. Diff durchführen

Die Änderungen können angezeigt werden, indem ein Diff des gefundenen Commit mit dem First Parent des Commit durchgeführt wird. Damit werden genau die Änderungen sichtbar, die dieses Feature in Bezug auf den master-Branch eingebracht hat. Im Beispiel werden die Änderungen zwischen Commit G und H angezeigt.

```
> git diff c52ce0a^1 c52ce0a
```

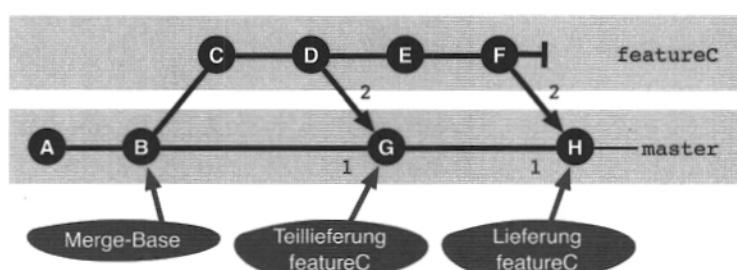


Abb. 17-15  
Beispiel eines  
Feature-Branch

Schritt für Schritt

## Alle Commits eines Feature-Branch finden

*Alle Commits, die während der Feature-Entwicklung angelegt wurden, werden angezeigt.*

Während eines nachträglichen Code-Reviews will man genauer nachvollziehen, wie Änderungen eines Feature-Branch eingebaut wurden. Dazu kann es sinnvoll sein, alle Commits des bereits integrierten Feature-Branch einzeln anzuschauen.

In Abbildung 17-15 ist ein Beispiel für ein Feature-Branch zu sehen. Im Folgenden werden die Commits entsprechend dieser Abbildung referenziert.

### 1. Merge-Commits des Features finden

Als Erstes benötigt man alle Commits auf dem master-Branch, die etwas mit dem Feature zu tun haben. Normalerweise wird dies nur ein Merge-Commit sein. Im Beispiel wird das Commit H gefunden.

```
> git checkout master  
  
> git log --first-parent --oneline --grep="feature/a"  
c52ce0a Pull-Request für feature/a
```

**--grep:** Sucht in der Log-Meldung nach einem bestimmten Text.

### 2. Feature-Branch-Start finden

Um alle Commits anzuzeigen, wird das Commit des master-Branch gesucht, von dem der Feature-Branch abgezweigt ist. Ausgangspunkt ist das gefundene Commit des vorherigen Schritts (Commit H). Dieses muss ein Merge-Commit sein und hat zwei Parents. Mit dem merge-base-Befehl wird das Commit gesucht, das der gemeinsame Ausgangspunkt des ersten (Commit G) und des zweiten Parent (Commit F) ist – die Merge-Base (Commit B).

```
> git merge-base c52ce0a^2 c52ce0a^1  
ca52c7f9bfd010abd739ca99e4201f56be1cfb42
```

### 3. Feature-Commits anzeigen

Nachdem der Ausgangspunkt gefunden wurde, können jetzt mit dem log-Befehl alle Commits des Feature-Branch angezeigt werden. Dazu wird gefragt, welche Commits notwendig sind, um von der Merge-Base (Commit B) auf das letzte Commit des Feature-Branch zu kommen. Das letzte Commit des Feature-Branch ist der zweite Parent (Commit F) des Merge-Commit (Commit H).

```
> git log --oneline ca52c7f..c52ce0a^2
```



## 18 Mit Forks entwickeln

Open-Source-Projekte sind auf freiwillige Mitarbeit angewiesen. Dies führt oft zu einem Zielkonflikt. Einerseits soll die Hürde zum Beitreten so niedrig wie möglich sein, um eine aktive Community aufzubauen. Andererseits sollen Qualitätsziele erreicht werden, und man möchte nicht jedem dahergelaufenen Entwickler Push-Recht für das Main-Repository geben.

Mit *Forks* und Pull-Requests bieten GitHub und andere zentrale Repository-Verwaltungen Hilfsmittel, um das Dilemma aufzulösen. Ein *Fork* ist ein serverseitiger Klon, auf dem unabhängig gearbeitet werden kann, ohne das originale *Main-Repository* zu verändern. Ein Pull-Request ist eine Aufforderung, Änderungen aus einem anderen Branch oder Repository zu übernehmen. Der *Fork* behält aber eine Verknüpfung zum Original, sodass per Pull-Requests Änderungen später leicht zurückgeführt werden können.

Wir zeigen hier einen branchfreien Workflow aus dem Projekt ZeroMQ, der von Pieter Hintjens<sup>1</sup> beschrieben und empfohlen wurde und der nicht nur für Open-Source-Projekte interessant ist. Wir zeigen,

- wie man einen *Fork* zum unabhängigen Entwickeln einrichtet,
- wie man seinen *Fork* aktualisiert, wenn im Main-Repository Änderungen von anderen Entwicklern integriert wurden,
- wie man einen Pull-Request erstellt und
- wie man als *Maintainer* zu diesem Pull-Request ein Review durchführt und die Änderungen dann integriert oder zurückweist.

---

<sup>1</sup> »Git Branches Considered Harmful« <http://hintjens.com/blog:24>

»Collective Code Construction Contract« <https://rfc.zeromq.org/spec:42/C4>

## Überblick

Ausgangspunkt für alle Entwicklungen ist ein zentrales Main-Repository (»Main« in Abbildung 18–1), dessen master-Branch den gültigen Stand des Projekts repräsentiert. Nur wenige *Maintainer* haben dort Push-Berechtigung.

Ein Entwickler erstellt sich einen *Fork* (Schritt 1 in Abbildung 18–1). Auf diesem hat er volle Rechte, insbesondere auch Push-Berechtigung. Er entwickelt auf dem master-Branch, ohne das originale Main-Repository zu berühren. Ist die Arbeit abgeschlossen, stellt er einen Pull-Request (Schritt 2 in Abbildung 18–1). Daraufhin prüft ein *Maintainer* die Änderungen (*Diff*) und entscheidet, ob sie übernommen werden sollen. Dann erstellt die zentrale Repository-Verwaltung ein Merge-Commit auf dem master-Branch im originalen Main-Repository (Schritt 3 in Abbildung 18–1).

## Voraussetzungen

Zentrale Repository-Verwaltung mit Forks und Pull-Requests:

z. B. GitHub, Bitbucket oder GitLab

**Inkrementelle Entwicklung:** Features sollten in Stunden oder Tagen abgeschlossen werden können. Längere Zyklen erhöhen das Risiko von Schwierigkeiten bei der Integration.

**Stabiler master-Branch:** In diesem Workflow ist der master-Branch im Main-Repository Ausgangspunkt und Referenz für jede Weiterentwicklung. Er muss jederzeit compile- und funktionsfähig gehalten werden. Automatisierte Prüfung ist dabei natürlich zu empfehlen.

**Quality-Gate für den master:** Das Projekt benötigt ein gemeinsames Verständnis dafür, was genau mit »stabil« gemeint ist, sodass schnell entschieden werden kann, ob Änderungen in den master übernommen werden.

## Workflow kompakt Mit Forks entwickeln

Entwicklung findet nur auf Forks statt, nie im eigentlichen Repository des Projekts. Forks sind unabhängige Klone des Main-Repository. Im Fork wird auf dem master-Branch entwickelt. Sind die Entwickler fertig, stellen sie einen Pull-Request. Der Maintainer holt dann die Änderungen vom Fork ab und integriert sie in den master-Branch des Main-Repository.

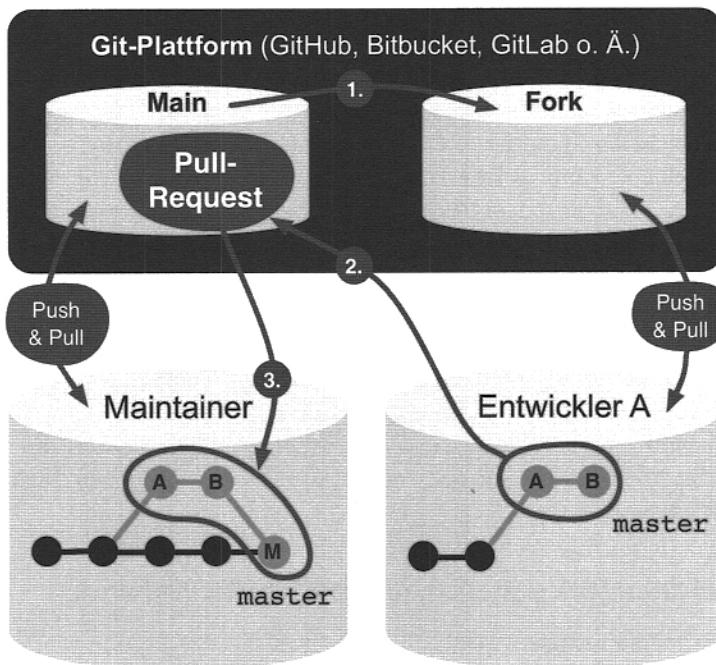


Abb. 18-1  
Workflow im Überblick

## Ablauf und Umsetzung

**Anmerkung:** Dieser Workflow zeigt das Vorgehen beim Arbeiten mit GitHub. Er kann aber mit anderen zentralen Repository-Verwaltungen sehr ähnlich angewandt werden.

### Rollen

Dieser Workflow kennt zwei Rollen:

- **Entwickler** ändern, erweitern und verbessern den Sourcecode in kleinen Inkrementen. Das Ergebnis nennt man *Patch*. Die Entwickler erstellen unabhängig voneinander Patches, versehen sie mit einer Beschreibung und reichen sie bei einem *Maintainer* ein.
- **Maintainer** integrieren Patches in das Main-Repository. Sie entscheiden, wann welche Patches übernommen werden, und sorgen dafür, dass zu jedem Zeitpunkt ein baubarer und getester Stand zur Verfügung steht, von dem aus Entwickler Weiterentwicklungen beginnen können.

**Anmerkung:** Natürlich kann ein *Maintainer* auch Entwickler sein. Es empfiehlt sich dann aber, einen anderen *Maintainer* die Prüfung und Integration der eigenen *Patches* durchführen zu lassen.

Im Folgenden stellen wir in den Abschnittsüberschriften die Rollenbezeichnung voran, um anzuseigen, wer die beschriebenen Aktionen durchführen soll.

### Entwickler: *Fork* erstellen, falls noch nicht vorhanden

#### Schritt 1: *Fork* anlegen

Auf der GitHub-Seite für das Main-Repository klickt man auf FORK. GitHub legt dann in Ihrem Account ein Projekt an, das einen Klon des *Main-Repository* enthält.

Abb. 18-2  
Der *Fork*-Button



## Schritt 2: Lokalen Klon des *Fork* anlegen

Unter CLONE OR DOWNLOAD findet man die Repository-URL zum Klo-  
nen. Anschließend wird das Repository, wie gewohnt, geklont.

```
> git clone git@github.com:entwickler-a/myproject.git
```

**Repositorys erstellen,  
klonen und verwalten**  
→ Seite 89

## Schritt 3: Herkunft konfigurieren

Um den *Fork* nach Weiterentwicklungen im originalen Main-Reposi-  
tory leicht aktualisieren zu können, wird eine Referenz auf das originale  
Main-Repository angelegt.

```
> cd myproject  
> git remote add \  
    upstream git@github.com:projekt/myproject.git
```

**Wie sagt man Git, wo  
das Remote-  
Repository liegt?**  
→ Seite 92

**upstream:** Unter diesem Namen *upstream* soll eine Referenz auf das  
Main-Repository angelegt werden, die man später beim *pull*-Befehl  
angeben kann, um Änderungen von dort nachzuholen.

**git@github.com:projekt/myproject.git:** Hier gibt man die URL des  
Main-Repository an.

Unser lokaler Klon hat jetzt Referenzen auf zwei Repositorys: *origin*  
für unseren *Fork* und *upstream* für das originale Main-Repository.

## Entwickler: *Fork* aktualisieren

Wenn bereits ein *Fork* vorhanden ist, muss nur der *master*-Branch um  
die Neuerungen aus dem originalen Main-Repository aktualisiert wer-  
den.

```
> git pull upstream master
```

**textttupstream:** Normalerweise bezieht sich der *pull*-Befehl auf das  
*origin*-Repository (hier wäre es das vom *Fork*). Deshalb wird hier  
*upstream* explizit angegeben, denn unter diesem Namen wurde eine  
Referenz auf das Main-Repository angelegt.

**master:** Die Neuerungen aus dem *master*-Branch des Main-Repository  
sollen geholt werden.

```
> git push
```

## Entwickler: Auf dem *Fork* entwickeln

Entwickler arbeiten grundsätzlich auf dem master-Branch ihrer *Forks*. Wenn etwas abgeschlossen ist, wird, wie üblich, ein Commit erstellt. Die Ergebnisse dürfen jederzeit mit parameterlosem push-Aufruf gesichert werden, da ja der *Fork* des Entwicklers als origin eingetragen ist und das originale Main-Repository nicht berührt.

```
> git commit  
> git push
```

Wenn man alleine arbeitet, braucht es nicht mehr als das. Falls mehrere Entwickler gemeinsam am selben Feature arbeiten, kann wie in »Gemeinsam auf einem Branch entwickeln« (Seite 135) beschrieben gearbeitet werden.

## Entwickler: Pull-Request erstellen

Wenn die Entwicklung abgeschlossen ist, erstellt ein Klick auf NEW PULL REQUEST auf der GitHub-Seite des *Fork* einen Pull-Request. Im nachfolgenden Dialog können die Voreinstellungen einfach übernommen werden, denn das Repository ist ja als *Fork* des Main-Repository registriert, und es gibt ja keine anderen Branches als master. Git zeigt an, ob ein konfliktfreier Merge möglich ist. Falls nicht, muss der *Fork* aktualisiert werden, um die Merge-Konflikte dort zu lösen, damit der Pull-Request weiter bearbeitet werden kann.

Abb. 18-3

Pull-Request kann übernommen werden



## Maintainer: Review durchführen

Der *Maintainer* erhält eine E-Mail mit einem Link auf eine Übersichtsseite zu diesem *Pull-Request*. Auf dem Tab CONVERSATION findet man alle Kommentare, die zu diesem Request gemacht wurden. Unter FILES CHANGED kann man alle Änderungen einsehen. Diese sind Grundlage für das Review.

Ziel des Reviews ist es festzustellen, ob die Änderungen in den master des Main-Repository übernommen werden können, d.h., dass die Software gebaut werden kann, alle Tests erfolgreich passiert wurden und keine der Regeln des Projekts (*Quality-Gate*) gebrochen wurde.

Folgendes kann beim Review festgestellt werden:

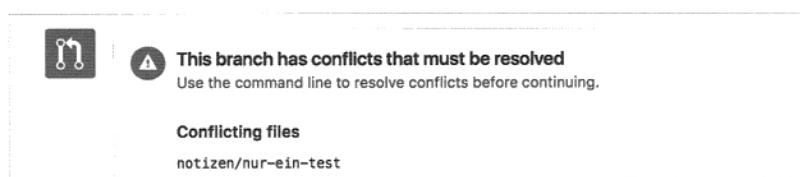
- Merge-Konflikte
- Mängel, die nicht integriert werden dürfen
- Mängel, die später behoben werden können
- Der Pull-Request kann übernommen werden.

Natürlich sollte man versuchen, die Änderungen in kleinen überschaubaren Commits abzuliefern. Kommt es aber doch einmal dazu, dass viele Dateien verändert wurden, dann erkennt man besser, was passiert ist, wenn man die Commits in aufsteigender Reihenfolge einzeln durchgeht. Deshalb findet man unter dem Tab **COMMITS** eine Auflistung aller Commits, die in diesen Request einfließen.

*Tipp: Große Änderungen Commit für Commit reviewen.*

## Review-Ergebnis: Merge-Konflikt

Wenn es zwischen dem `master`-Branch des Main-Repository und dem des Forks Merge-Konflikte gibt, die nicht automatisch aufgelöst werden können, meldet GitHub Folgendes:



**Abb. 18-4**  
Merge-Konflikte!

### Schritt 1: Maintainer: Zur Konfliktauflösung auffordern

Auf der GitHub-Seite für diesen Pull-Request trägt man einen entsprechenden Kommentar ein, z. B. »Bitte behebe die Merge-Konflikte, damit ich deine Änderungen übernehmen kann«. Der Entwickler erhält dann automatisch eine E-Mail.

### Schritt 2: Entwickler: Merge-Konflikte lösen

Neuerungen aus dem originalen Main-Repository holen:

```
> git pull upstream master
```

Git meldet jetzt, in welchen Dateien noch Konflikte aufzulösen sind. Zum Abschluss überträgt man das Ergebnis, und GitHub benachrichtigt den *Maintainer*.

```
> git commit  
> git push
```

**Branches zusammenführen**  
→ Seite 67

## Review-Ergebnis: Mangel, der nicht integriert werden darf

Ein Ziel dieses Workflow ist, auf dem master-Branch jederzeit einen funktionsfähigen Stand des Produkts bereitstellen zu können. Deshalb dürfen Änderungen, die den master-Branch brechen, nicht integriert werden.

### Schritt 1: Maintainer: Mangel kommunizieren

Wenn ein Fehler oder Problem entdeckt wird, kann man direkt in der Ansicht FILES CHANGED oder in der Ansicht COMMIT einen Kommentar eingeben, der den Entwickler auffordert, das Problem zu beheben. Alle Kommentare können unter dem Tab CONVERSATION eingesehen werden. Der Entwickler wird automatisch benachrichtigt.

*Tipp: GitHub-Reviews*

Wenn man viele Anmerkungen machen möchte, kann man beim Eingeben des ersten Kommentars START A REVIEW anklicken, dann werden die Kommentare zunächst gesammelt und nicht sofort verschickt. Dies passiert erst, wenn man FINISH REVIEW klickt.

### Schritt 2: Entwickler: Mangel beheben

Der Entwickler arbeitet weiter auf dem master-Branch des Fork und versucht das Problem zu lösen. Beim nächsten Push wird der *Pull-Request* aktualisiert, und der *Maintainer* kann erneut ein Review durchführen.

```
> git commit  
> git push
```

## Review-Ergebnis: Mangel, der später behoben werden kann

Es kommt nicht selten vor, dass beim Review Dinge entdeckt werden, die keine Fehler sind und auch keine Regeln des *Quality-Gate* brechen, die aber trotzdem behandlungswürdig sind, wie etwa Code, der ein API verwendet, von dem es eine neuere und bessere Version gibt. So etwas verhindert nicht das Schließen des *Pull-Request*, sollte aber dokumentiert werden.

### Schritt 1: Maintainer: Mangel dokumentieren

In der Ansicht FILES CHANGED kann man die Quelltexte direkt bearbeiten und dort beispielsweise TODO-Kommentare eintragen. Alternativ kann man auch den eingebauten Issue Tracker von GitHub nutzen.

## Review-Ergebnis: Der Pull-Request kann übernommen werden

Nach dem Passieren des Quality-Gate können die Änderungen übernommen werden. Dies tut der *Maintainer* auf dem Tab CONVERSATION:



**Abb. 18-5**  
Pull-Request kann übernommen werden

Danach muss noch einmal CONFIRM MERGE bestätigt werden, und der Pull-Request ist erledigt.

## Release erstellen

In einer Continuous Delivery-Umgebung kann die Erstellung von Releases durch den Build-Server automatisiert werden, beispielsweise so:

1. aktuellen master-Branch holen
2. komplizieren und testen
3. Release-Artefakte bauen
4. Integrationstests durchführen
5. falls nicht erfolgreich: hier abbrechen
6. Tag erstellen und per Push hochladen
7. Release-Artefakte zum Artefakt-Repository hochladen
8. Version zum Deployment freigeben

**Versionen markieren**  
→ Seite 105

Kommt es zu einem Fehler, wird dieser in einem *Fork* behoben, per Pull-Request integriert, und auch dieser neue master-Stand durchläuft dasselbe Verfahren.

## Release stabilisieren

Nicht jedes Projekt kann Continuous Delivery in dieser Form betreiben, z. B. sollte ein Projekt, das eine Library herstellt, die von vielen anderen Projekten genutzt wird, nicht kontinuierlich neue Versionen mit neuen Features und geändertem Verhalten ausliefern. In solchen Fällen möchte man *Stable Release Versions* anbieten, die funktional nicht mehr erweitert werden, sondern nur Fehlerbehebungen erhalten.

Zur Stabilisierung erstellt man einen *Fork*, der ebenfalls im *Forking*-Workflow bearbeitet wird. Das Projekt ZeroMQ<sup>2</sup> zum Beispiel, hat einen *Fork*<sup>3</sup>, der ausschließlich zur Pflege der Linie 4.1 genutzt wird. Seine *Maintainer* nehmen alleinig Pull-Requests mit Bugfixes an. Mit jedem Release wird ein Tag mit erhöhtem Patch-Zähler erstellt: 4.1.0, 4.1.1, 4.1.1 etc.

Bugfixes werden meist im Hauptprojekt erstellt und als Backports in den Stabilisierungs-*Fork* übernommen. Bugfixes können auch im Stabilisierungs-*Fork* implementiert werden. Das ist dann sinnvoll, wenn sich die betroffenen Codestellen im Hauptprojekt stark weiterentwickelt haben. Die so entstehenden Bugfixes passen nicht mehr zum Hauptprojekt. Deshalb behandelt man den Stabilisierungs-*Fork* als separates Projekt und verzichtet auf einen direkten Rückfluss zum Hauptprojekt in Form Pull-Requests. Solche Bugfixes werden also zum Hauptprojekt portiert oder dort sogar neu implementiert.

**Anmerkung:** Da jeder aktive Stabilisierungs-*Fork* als separates Projekt betrieben wird, steigt der Aufwand, wenn viele *Stable Release Versions* parallel gepflegt werden und Garantien gegeben werden müssen, dass bestimmte Bugfixes alle Versionen erreichen. Dann sollte man besser den Workflow »Mit mehreren aktiven Releases arbeiten« (Seite 199) in Erwägung ziehen, der mehr Automatisierung bei der Übertragung von Bugfixes in verschiedene Release-Linien ermöglicht. Das Projekt ZeroMQ beispielsweise pflegt aktiv nur noch die Linien 4.1 und 3.1. Ältere Versionen wurden seit Jahren nicht mehr angerührt.

## Warum nicht anders?

### Warum keine Branches (außer master)?

Ziel dieses Workflows ist es, eine aktive Community für Projekte aufzubauen, indem man das Beitragen zum Projekt so leicht und angenehmen wie möglich macht.

**Periodisch Releases durchführen**  
→ Seite 185

Ausgefeilte Branching-Strategien, wie etwa GitFlow, erfordern disziplinierte Zusammenarbeit zwischen den Entwicklern. Jeder muss wissen, wann für welche Art von Änderung von welchem Branch abgezweigt wird, wem welche Branches gehören, wer die Berechtigungen vergibt, wo Merges stattfinden, wer diese durchführt und wer schlussendlich die Branches wieder löscht.

---

<sup>2</sup> <https://github.com/zeromq/libzmq>

<sup>3</sup> <https://github.com/zeromq/zeromq4-1>

Sich darüber abzustimmen, gelingt nur in Projekten, an denen mehrere Entwickler über längere Zeit gemeinsam entwickeln. Für spontan Beitragende ist der Einstieg eher schwer zu finden.

Beim Aufbau einer Community ist schnelles Feedback wichtig. Je früher ein Beitragender erfährt, dass seine Verbesserungen Teil des Produkts geworden sind, desto wahrscheinlicher ist es, dass er wieder beitragen wird. Bei Projekten mit komplexem Staging kann es Wochen dauern, bis eine Änderung das Produkt wirklich erreicht. Projekte mit Forking-Workflow hingegen, wie z. B. ZeroMQ, integrieren Patches oft noch am selben Tag in den master, der den offiziellen Stand des Projekt repräsentiert.

**Anmerkung:** Es spricht übrigens überhaupt nichts dagegen, dass Entwickler lokal Branches für eigene Zwecke nutzen, z. B. um unferige Implementierungen zu »parken«, denn die *Forks* gehören Ihnen. Dort können sie machen, was sie wollen. Die offizielle Kommunikation und Koordination des Projekts sollte sich aber immer auf den master-Branch des Main-Repository beziehen, damit alle Beteiligten einen gemeinsamen Referenzpunkt haben.

## Warum später zu behebende Mängel nicht einfach in die Conversation eintragen?

Findet man beim Review Pull-Request einen Mangel, trägt man dies normalerweise als Kommentar in GitHub ein. Dieser wird Teil der CONVERSATION zum Pull-Request. Später wird der Pull-Request geschlossen und erhält den Status CLOSED, und der Kommentar wird nie wieder gesehen.

Deshalb empfehlen wir nur solche Mängel in der CONVERSATION einzutragen, die bis zum Abschluss des Pull-Request auf jeden Fall geschlossen werden. Dinge, die erst danach gelöst werden, sollten irgendwo gespeichert werden, wo sie im Status »offen« verbleiben, bis sie tatsächlich erledigt sind, z. B. als TODO-Kommentare im Sourcecode oder in einem Issue Tracker.



## 19 Kontinuierlich Releases durchführen

Bei Projekten, die einen Continuous-Delivery-Prozess<sup>1</sup> umsetzen, ist jedes Commit auf dem master-Branch ein potenzieller Release-Kandidat und wird durch eine Deployment-Pipeline verarbeitet.

Eine Deployment-Pipeline besteht aus mehreren Schritten. Typischerweise wird zuerst ein Build durchgeführt und dann werden (binäre) Deployment-Artefakte erzeugt. In weiteren Schritten werden mit diesen Artefakten verschiedene Tests (Quality-Gates) durchlaufen. Überstehen die Deployment-Artefakte alle Tests, werden sie als Release-Kandidaten markiert und im letzten Schritt in Produktion gebracht.

Bei diesem Workflow wird bewusst nur einmal gebaut. Die gleichen Artefakte werden für alle Tests und den produktiven Betrieb benutzt.

Bei einem Continuous-Delivery-Prozess werden keine speziellen Branches für Übergänge zwischen Stabilisierungsphasen benötigt. Lediglich die Entwicklung neuer Features und Bugfixes findet auf separaten Feature-Branches statt. Vor der Integration der Feature-Branches in den master-Branch finden bereits viele Tests statt, sodass die Wahrscheinlichkeit von Problemen in der Deployment-Pipeline gering ist. Sobald ein Feature-Branch in den master-Branch integriert wird, läuft die Pipeline los.

Wird ein Release-Kandidat in Produktion gebracht, wird der zugehörige Commit mit einem Release-Tag markiert.

Dieser Workflow zeigt,

- welche Branches in einem Continuous-Delivery-Prozess notwendig sind,
- wie die Bugfix- und die Feature-Entwicklung durchgeführt werden und
- wie mit Fehlern in der Deployment-Pipeline umgegangen wird.

**Periodisch Releases durchführen**

→ Seite 185

**Mit Feature-Branches entwickeln** → Seite 143

**Versionen markieren**

→ Seite 105

---

<sup>1</sup> Jez Humble, David Farley: »Continuous Delivery. Reliable Software Releases Through Build, Test, and Deployment Automation« – (Addison-Wesley, 2010)

## Überblick

In Abbildung 19–1 sind die wichtigsten Bestandteile dieses Workflows zu sehen.

**Mit Feature-Branches entwickeln** → Seite 143

**Integration mit Jenkins** → Seite 263

Der master-Branch dient als Grundlage, um Release-Kandidaten zu bauen. Die Entwicklung und das Bugfixing finden auf separaten Feature-Branches statt.

Das Build-System erkennt, wenn es ein neues Commit auf dem master-Branch gibt, und startet den ersten Schritt der Deployment-Pipeline. Typischerweise wird in diesem Schritt das Commit gebaut und das Ergebnis in geeigneter Form paketiert. Die entstehenden (binären) Artefakte werden in einem Artefakt-Repository abgelegt. Das kann einfach nur ein Dateisystem oder ein eigenes System sein, z. B. Artifactory<sup>2</sup> oder Nexus<sup>3</sup>.

Dabei ist es wichtig, die Zuordnung der Artefakte zur Commit-ID dauerhaft festzuhalten. Das kann über Metainformationen direkt im Artefakt, über zusätzliche Dateien oder über Metadaten im Artefakt-Repository geschehen. Die nächsten Schritte der Deployment-Pipeline holen sich immer nur über das Artefakt-Repository die benötigten Artefakte.

Der letzte Schritt der Deployment-Pipeline ist das tatsächliche Deployment des Artefakts auf dem Zielsystem. Zusätzlich zur Aktualisierung des produktiven Systems wird im Git-Repository noch ein Release-Tag angelegt. Dazu wird die zugeordnete Commit-ID benötigt.

Falls die Pipeline einen Fehler meldet, ist es möglich, den Merge des Feature-Branch rückgängig zu machen.

## Voraussetzungen

**Nur ein produktives Release:** Es gibt nur ein produktives Release, d. h., es werden nicht mehrere Versionen des Produkts oder Projekts parallel gepflegt.

**Artefakt-Repository vorhanden:** Es existiert ein Artefakt-Repository, in dem die Build-Ergebnisse abgelegt werden können, und zu jedem Artefakt kann eine Commit-ID zugeordnet werden.

**Deployment-Pipeline existiert:** Es existiert eine Deployment-Pipeline mit umfassenden Tests, die die Stabilität und Release-Reife eines Versionsstandes überprüft.

---

<sup>2</sup> <http://www.jfrog.com/open-source/>

<sup>3</sup> <http://www.sonatype.org/nexus/>

Workflow kompakt

## Kontinuierlich Releases durchführen

Für ein Projekt, das Continuous Delivery umsetzt, wird ein Release erstellt. Jedes Commit auf dem master-Branch ist ein potenzieller Release-Kandidat. Erfolgreiche Releases werden durch ein Tag markiert.

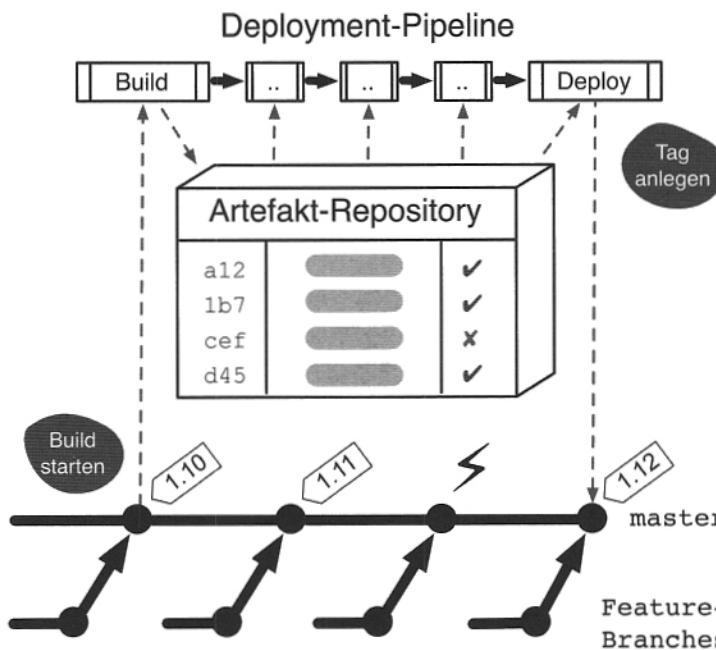


Abb. 19–1  
Workflow im Überblick

## Ablauf und Umsetzung

Dieser Workflow basiert auf der Anbindung eines Build-Systems für das Betreiben der Deployment-Pipeline und auf der Zusammenarbeit mit einem Artefakt-Repository. Beide Werkzeuge werden nicht Bestandteil der folgenden Beschreibung sein. Es gibt in dem Bereich zu viele Varianten, um diese abschließend und aktuell in einem Buch über Git darzustellen. Um sich über die Anbindung von Jenkins als Build-System zu informieren, lesen Sie das Kapitel »Integration mit Jenkins« ab Seite 263.

Nachfolgend wird davon ausgegangen, dass ein Build-System und ein Artefakt-Repository existieren. Das Build-System ist so konfiguriert, dass es bei Änderungen des `master`-Branch die Deployment-Pipeline startet.

### Release-Kandidat erzeugen

**Mit Feature-Branches entwickeln** → Seite 143

Wie bereits in der Übersicht beschrieben, ist der `master`-Branch die Grundlage, um die Release-Kandidaten zu bauen. Neue Features und Bugfixes werden deshalb auf separaten Feature-Branches entwickelt.

#### Schritt 1: Arbeiten mit Feature-Branches

Normalerweise werden Feature-Branches nach Fertigstellung durch einen manuellen Merge mit dem `master`-Branch zusammengeführt. Möchte man die Qualität vor der Übernahme in den `master`-Branch überprüfen, ist es sinnvoll, vor dem Merge schon Qualitätssicherungsmaßnahmen durchzuführen, z.B. durch einen automatischen Build oder durch ein verpflichtendes Code-Review. Für diese Art der Qualitätssicherung eignen sich Pull-Requests sehr gut. Dazu ist eine zentrale Repository-Verwaltung hilfreich, die dieses Feature unterstützt, z.B. Atlassian Bitbucket Server<sup>4</sup>, GitHub<sup>5</sup> oder GitLab<sup>6</sup>. Egal welches Werkzeug benutzt wird, am Ende ist der Feature-Branch durch einen Merge-Commit mit dem `master`-Branch vereinigt.

#### Schritt 2: Aktuelle Commit-ID ermitteln

Nachdem der Feature-Branch in den `master`-Branch integriert wurde, startet die Deployment-Pipeline mit dem Bau des Merge-Commit. Das Ergebnis sind Build-Artefakte, die in einem Artefakt-Repository abge-

---

<sup>4</sup> [www.atlassian.com/software/bitbucket/server](http://www.atlassian.com/software/bitbucket/server)

<sup>5</sup> <https://github.com>

<sup>6</sup> <https://gitlab.com>

legt werden. Dabei ist die Zuordnung zur Commit-ID wichtig, um am Ende der Pipeline das Commit mit einem Tag versehen zu können.

Die Ermittlung von Commit-IDs für einen Branch oder Tag übernimmt der rev-parse-Befehl:

```
> git rev-parse --verify master
```

**--verify:** Stellt sicher, dass nur ein Parameter übergeben wird und dieser in einen gültigen Hashwert umgewandelt werden kann.

Diesen Befehl kann man während des Builds aufrufen und das Ergebnis dem Artefakt zuordnen.

Alternativ stellen auch die meisten Build-Systeme die aktuell verwendete Commit-ID in einer Umgebungsvariable zur Verfügung, z.B. GIT\_COMMIT im Jenkins.

### Schritt 3: Umgang mit Fehlern in der Deployment-Pipeline

Im besten Fall läuft die Deployment-Pipeline ohne Fehler durch, und der Release-Kandidat wird produktiv gesetzt. Wenn es zu Fehlern kommt, ist es in den meisten Fällen einfach, diese zeitnah zu beheben. Bei einem Continuous-Delivery-Prozess sind Änderungen typischerweise kleinteilig, und somit sind die möglichen Fehlerursachen häufig leicht zu identifizieren.

Ist der Fehler gefunden, wird ein neuer Feature-Branch mit der Fehlerkorrektur erzeugt und abgegeben. Dadurch wird die Deployment-Pipeline erneut gestartet und dieses Mal hoffentlich erfolgreich durchlaufen.

Wenn es schwierig ist, die Fehlerursache zu finden, oder wenn die Lösung zeitaufwendig ist, kann es sinnvoll sein, die Änderungen erst mal wieder vom master-Branch zu entfernen. Dadurch können andere Features und Bugfixes während der Fehlerkorrektur produktiv gesetzt werden.

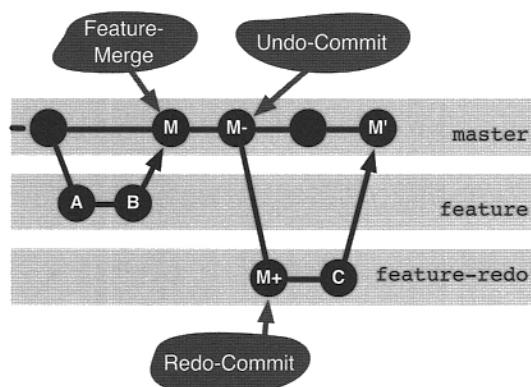
Für das Zurücknehmen von Änderungen gibt es den revert-Befehl. Bei diesem gibt man eine Commit-ID an, und Git wird auf dem aktuellen Branch ein neues Commit mit den inversen Änderungen erzeugen. Für den fehlerhaften Feature-Branch bedeutet das, dass man den Merge-Commit rückgängig machen muss:

```
> git revert --mainline 1 <merge-commit-id>
```

**--mainline:** Für die Berechnung des inversen Deltas muss sich der revert-Befehl immer auf genau einen Parent beziehen. In diesem Fall handelt es sich um einen Merge-Commit, und die Änderung soll in Bezug auf den ersten Parent, also den master-Branch, zurückgenommen werden.

Den revert-Befehl kann man entweder direkt auf dem master-Branch ausführen oder man legt einen neuen Feature-Branch an und führt den Befehl dort aus<sup>7</sup>. Wenn man mit einem neuen Feature-Branch arbeitet, wird dieser nach dem Revert normal über einen Pull-Request abgegeben.

**Abb. 19-2**  
Fehler in  
Feature-Branch  
beheben



**Warum nach einem  
Revert nicht auf dem  
originalen  
Feature-Branch  
weiterarbeiten?**

→ Seite 182

Für die eigentliche Fehlerkorrektur legt man einen neuen Feature-Branch an und erzeugt auf diesen das inverse Commit von der Fehlerkorrektur. Das heißt, das Commit, das beim vorigen revert-Befehl entstanden ist, wird nochmals invertiert, und damit werden die originalen Änderungen auf den neuen Feature-Branch gebracht:

```
> git checkout master
> git pull
> git checkout -b feature-a-korrektur
> git revert <undo-commit-id>
```

Auf den neuen Branch kann man nun die Fehlerkorrektur durchführen und am Ende einen neuen Pull-Request stellen. In Abbildung 19-2 ist der Ablauf des Undo und Redo mit Fehlerbehebung skizziert.

#### Schritt 4: Release-Tag anlegen

**Versionen markieren**  
→ Seite 105

Ist die Deployment-Pipeline ohne Fehler durchlaufen worden und der Release-Kandidat auf das Zielsystem ausgerollt, dann soll das Release-Commit durch ein Tag markiert werden.

Als Erstes muss die zugehörige Commit-ID über das Artefakt-Repository, über einen Parameter der Deployment-Pipeline oder direkt über das Build-Artefakt ermittelt werden.

<sup>7</sup> Einige Repository-Verwaltungen haben für das Rückgängigmachen von Pull-Requests auch schon direkte Unterstützung im UI, z. B. GitHub.

Dann kann man mit dem tag-Befehl ein neues Release-Tag erzeugen. Dabei ist es sinnvoll, den Release-Tags ein Präfix zu geben, z. B. rel/.

Im Kontext eines Continuous-Delivery-Prozesses ist es meistens nicht möglich bzw. sinnvoll, semantische Versionsnummern<sup>8</sup> zu vergeben. Das heißt, es entstehen keine Versionen mit dem typischen dreistufigen Major.Minor.Bugfix-Muster. Häufig wird einfach nur durchnummiert, ggf. wird noch eine Major-Nummer mitgeführt<sup>9</sup>.

```
> git tag -a rel/1.42 -m "Rel 1.42" <release-commit-id>
```

Dieser Befehl wird in der Deployment-Pipeline als Bestandteil des Deploy-Schrittes ausgeführt. Auch dafür existieren ggf. schon Plug-ins in dem jeweiligen Build-Werkzeug.

## Warum nicht anders?

### Warum nicht auf Feature-Banches verzichten?

Der oben beschriebene Workflow geht von dem Einsatz von Feature-Banches aus. Wäre es nicht auch möglich, dass alle gemeinsam auf dem master-Branch arbeiten?

Dabei würde jeder Entwickler auf seinem lokalen master-Branch neue Commits anlegen und am Ende dann durch Pull und Push die Änderungen auf den zentralen master-Branch bringen. Die Arbeit mit der Deployment-Pipeline würde genauso funktionieren wie oben beschrieben.

Allerdings würde dadurch die Möglichkeit wegfallen, die Features und Bugfixes vor dem offiziellen Merge auf dem Build-Server zu bauen und zu testen. Es wäre natürlich immer noch möglich, die Tests lokal durchzuführen. Doch das erfordert viel Disziplin von den Entwicklern, und es können Fehler durch Unterschiede in den Umgebungen übersehen werden.

Im Endeffekt wird dadurch die Wahrscheinlichkeit erhöht, dass es in der Deployment-Pipeline zu Fehlern kommt.

**Gemeinsam auf einem Branch entwickeln**

→ Seite 135

**Pull-Requests bauen**

→ Seite 269

### Warum nicht auf Tags verzichten?

Sind die Release-Tags wirklich notwendig? Würde es nicht ausreichen, den Commit-Hash zu kennen?

<sup>8</sup> <http://semver.org/>

<sup>9</sup> Alternativ wird das Release mit dem aktuellen Zeitstempel und der verkürzten Commit-ID gebildet, z. B. 201505050931-3ed45.

Typischerweise kommen die Release-Nummern bei der Kommunikation mit den Anwendern zum Einsatz – zum Beispiel im Falle eines Fehlers, wenn der Anwender einen Report erstellen soll. In dem Kontext sind die Hashwerte nicht besonders einprägsam und auch sehr lang.

Wenn die Eingabe von Fehlerreports automatisiert werden kann und der Anwender sich nicht um die Eingabe der Versionsnummer kümmern muss, fällt diese Begründung für Release-Tags weg.

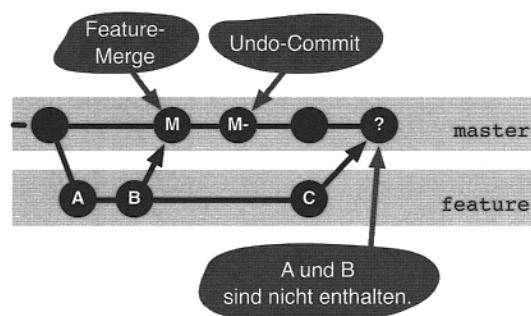
Wenn man Continuous Deployment sehr konsequent einsetzt, dann sind mehrere Dutzend Releases pro Tag keine Seltenheit. In dem Kontext kann die schiere Anzahl von Release-Tags zum Problem werden. In dem Fall kann es sinnvoll sein, die Tags nicht in das Standard-Entwicklungs-Repository zu übertragen, sondern ein spezielles zentrales Repository für die Release-Tags zu haben. Dann würden die Entwickler die Tags per Default nicht sehen.

### Warum nach einem Revert nicht auf dem originalen Feature-Branch weiterarbeiten?

In Abschnitt »Schritt 3: Umgang mit Fehlern in der Deployment-Pipeline« wurde erklärt, wie man Merges von Feature-Branches mit dem revert-Befehl rückgängig machen kann. Für die Fehlerbehebung wurde ein neuer Feature-Branch angelegt, das erzeugte Undo-Commit mit dem revert-Befehl nochmals invertiert und dann auf dem neuen Feature-Branch die Fehlerkorrektur durchgeführt.

Wäre es nicht einfacher, den neuen Feature-Branch an dem Commit vor dem Merge wieder aufzusetzen, d. h., auf das Revert des Reverts zu verzichten (Abbildung 19–3)?

**Abb. 19–3**  
Fehlerhafte Behebung  
von Fehlern in  
Feature-Branches



Leider funktioniert das mit Git nicht. Da der Merge mit dem master-Branch bereits durchgeführt wurde und dieses Merge-Commit auch dauerhaft vorhanden bleibt, würden bei einem erneuten Merge des Feature-Branch nur die neuen Änderungen übernommen werden. Im Endeffekt wäre dann zwar die Fehlerkorrektur enthalten, aber die eigentlichen Feature-Commits wären es nicht.



## 20 Periodisch Releases durchführen

Dieser Workflow beschreibt einen Release-Prozess für ein typisches Webprojekt. Bei unserem Webprojekt gibt es immer nur ein produktives Release und ein zukünftiges Release. Im produktiven Release sollen schwerwiegende Fehler und Sicherheitsrisiken sehr schnell behoben werden (sogenannte Hotfixes). Das neue Release soll vor dem Veröffentlichen eine ausführliche mehrtägige Testphase durchlaufen. Parallel soll die Weiterentwicklung für das nächste Release erfolgen können.

Dieser Workflow zeigt, wie ein Release-Prozess für ein Projekt mit Git so umgesetzt wird, dass

- Hotfixes auf dem produktiven Release unterstützt werden,
- die parallele Arbeit am neuen Release während der Testphase möglich ist,
- garantiert ist, dass alle Fehler, die als Hotfix oder während der Testphase behoben werden, in den Entwicklungsstand zurückfließen,
- die Historie der Releases und Hotfixes einfach abzurufen ist und
- Vergleiche zwischen Releases und Entwicklungsständen einfach möglich sind.

## Überblick

Abbildung 20–1 zeigt die Branches, die für die Entwicklung und den Release-Prozess benötigt werden.

*Mit Feature-Banches entwickeln → Seite 143*

Die Entwicklung findet auf dem `develop`-Branch statt. Dabei ist es unerheblich, ob Feature-Banches benutzt werden oder nicht. Wichtig ist nur, dass der `develop`-Branch den Code enthält, der zum nächsten Release führen soll.

*Gemeinsam auf einem Branch entwickeln → Seite 135*

Während der Vorbereitung des Release wird ein eigener `release`-Branch benutzt, um das zukünftige Release zu stabilisieren. Parallel kann auf dem `develop`-Branch für das nächste Release weiterentwickelt werden.

Sobald die Stabilisierung abgeschlossen ist, wird ein Release-Commit auf dem `master`-Branch angelegt und gleichzeitig ein Release-Tag erzeugt.

Tritt im produktiven Release ein schwerwiegender Fehler auf, wird ein neuer `hotfix`-Branch angelegt. Nach der Fehlerbehebung wird ein Hotfix-Commit auf dem `master`-Branch ausgeführt und ein Release-Tag angelegt.

Der `release`-Branch und die `hotfix`-Branches existieren nur so lange, wie die Stabilisierung bzw. die Fehlerbehebung stattfindet.

Die Änderungen während der Stabilisierung und des Hotfix werden immer durch Merges in den `develop`-Branch zurückgeführt.

## Voraussetzungen

*Mit mehreren aktiven Releases arbeiten → Seite 199*

**Nur ein produktives Release:** Es gibt nur ein produktives Release, d. h., es werden nicht mehrere Versionen des Produkts oder Projekts parallel gepflegt.

**Stabiler Entwicklungsstand:** Der Entwicklungs-Branch ist gut getestet, und die zu erwartenden Fehler in der Testphase sind überschaubar.

**Vollständiges Release:** Es gehen immer alle Neuerungen bzw. Änderungen des Entwicklungs-Banches in das nächste Release ein.

Workflow kompakt

## Periodisch Releases durchführen

Für ein Projekt wird regelmäßig ein neues Release erstellt. Die Vorbereitung des Release findet in einem separaten Branch statt. Auf dem aktuell produktiven Release können Hotfixes durchgeführt werden.

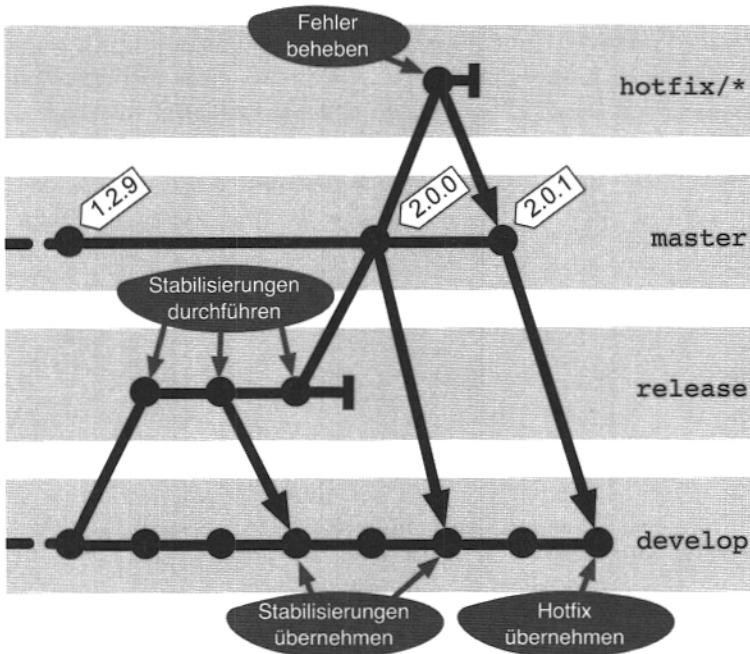


Abb. 20–1  
Workflow im Überblick

## Ablauf und Umsetzung

Die beiden folgenden Abschnitte beschreiben die einmalige Vorbereitung des Repositorys, um den Entwicklungs-Branch von dem Release-Branch zu trennen.

### Vorbereitung: develop-Branch anlegen

Für den Workflow wird der master-Branch als Release-Branch benutzt. Die eigentliche Entwicklung soll auf dem develop-Branch stattfinden. Dieser existiert standardmäßig nicht und muss angelegt werden.

Falls Sie bisher auf dem master-Branch entwickelt haben, dann legen Sie einfach einen neuen develop-Branch an:

```
> git branch develop master
```

Anschließend führen Sie einen Push durch, damit der neue Branch für alle anderen im Team sichtbar ist:

```
> git push --set-upstream origin develop
```

Danach teilen Sie jedem im Team mit, dass ab sofort auf dem develop-Branch entwickelt wird.

### Vorbereitung: master-Branch auf das erste Commit setzen

*Commit-Historie  
zeigen → Seite 36*

*First-Parent-History  
→ Seite 75*

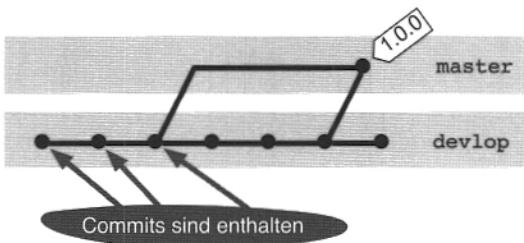
Der master-Branch soll bei diesem Workflow nur Commits enthalten, die ein neues Release oder ein Hotfix-Release darstellen. Die First-Parent-History des master-Branch kann als Release-Historie benutzt werden, z. B. mit dem log-Befehl:

```
> git checkout master
> git log --first-parent --oneline
590lec9 Hotfix-Release-2.0.1
b955c9c Release-2.0.0
5d0173d Release-1.0.0
3a05e26 init
```

**--first-parent:** Bei einem Commit wird nur der erste Vorgänger (Parent) betrachtet.

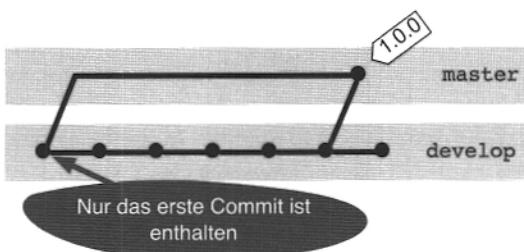
**--oneline:** Die Commit-Logs sollen nur einzeilig ausgegeben werden.

Da der master-Branch bisher für die Entwicklung genutzt wurde, zeigt er auf ein beliebiges Commit. Es ist wichtig, den Anfang des master-Branch korrekt festzulegen. Wenn der master-Branch dort bleiben würde, dann sind nach dem ersten Release in der First-Parent-History die bisherigen Commits des master-Branch enthalten (Abbildung 20–2).



**Abb. 20-2**  
master-Branch beginnt  
beim falschen Commit.

Besser ist es, den master-Branch beim ersten Commit des develop-Branch beginnen zu lassen. Dann wird nur ein unnötiges Commit (siehe »Erstes Commit« in Abbildung 20-3) in der Release-Historie enthalten sein.



**Abb. 20-3**  
master-Branch beim  
ersten Commit  
beginnen lassen

### Schritt 1: Erstes Commit ermitteln

Leider gibt es in Git keinen einfachen Befehl, um das erste Commit eines Branch zu ermitteln. Der beste Weg besteht darin, das Log auszugeben und den letzten Eintrag zu suchen:

```
> git checkout develop
> git log --oneline --first-parent | tail -1
3a05e26 init
```

**--oneline:** Die Commit-Logs werden nur einzeilig ausgegeben.

**--first-parent:** Für die Ausgabe wird nur der erste Parent der Commits beachtet. Dies führt zu weniger Ausgaben und somit zu einem schnelleren Ergebnis.

**| tail -1:** Gibt von der Log-Ausgabe nur die letzte Zeile aus.

### Schritt 2: master-Branch verschieben

Nachdem das erste Commit gefunden wurde, kann der master-Branch verschoben werden<sup>1</sup>. Dazu wird der master-Branch aktiviert und mit dem reset-Befehl auf das Start-Commit verschoben:

```
> git checkout master  
> git reset --hard 3a05e26
```

Anschließend muss der neue Zustand des master-Branch mit dem push-Befehl in das zentrale Repository übertragen werden. Da der Branch-Zeiger direkt manipuliert wurde, wird der normale push-Befehl sehr wahrscheinlich einen Fehler bringen (reject), und man muss die Ausführung mit der Option die Option -f erzwingen:

```
> git push -f
```

**-f:** Erzwingt das Push, auch wenn kein Fast-Forward möglich ist. Es kann sein, dass man diese Möglichkeit in der zentralen Repository-Verwaltung explizit freischalten muss.

Danach teilen Sie dem Team mit, dass einmalig der master-Branch aktualisiert werden muss, damit nicht zufällig beim nächsten Push der alte Zustand wiederhergestellt wird:

```
> git checkout master  
> git pull
```

### Release vorbereiten und erstellen

Der folgende Abschnitt beschreibt, welche Schritte notwendig sind, um ein Projekt mit Git zu releasesn.

Die Entwicklung des Projekts findet auf dem develop-Branch statt. Auf diesem werden auch die notwendigen Unitests und Integrations-tests durchgeführt.

Sobald die Entwicklung abgeschlossen ist und das Release durchgeführt werden soll, werden typischerweise noch weitere und intensivere Tests vorgenommen. Das bedeutet, dass am Code nur noch releasekritische Bugfixes und ggf. Workarounds implementiert werden. Wie lange diese Release-Phase dauert, hängt stark vom Entwicklungsprozess, von der vorhandenen Codequalität und von den Testaufwänden ab. Es kann sich dabei um einige Stunden oder um mehrere Wochen handeln.

---

<sup>1</sup>Falls Sie mit einem leeren Repository gestartet sind und bisher keinen master-Branch haben, dann legen Sie einfach einen neuen Branch bei dem gefundenen Commit an (siehe »Branch erstellen«, Seite 62).

Damit nicht die Entwicklung für das nächste Release während der Release-Phase stockt, wird die Stabilisierung des Release in einem eigenen release-Branch durchgeführt. Dieser existiert nur so lange, bis das neue Release stabilisiert wurde. Beim nächsten Release wird wieder ein neuer release-Branch angelegt.

### Schritt 1: release-Branch anlegen

Der release-Branch wird basierend auf dem aktuellen develop-Branch angelegt. Der checkout-Befehl kann genutzt werden, um den neuen Branch anzulegen und zu aktivieren:

```
> git checkout -b release master
```

Anschließend wird der Branch wieder in das zentrale Repository übertragen:

```
> git push --set-upstream origin release
```

### Schritt 2: Stabilisierung mit dem release-Branch

Im release-Branch werden nur Fehler behoben, die ein Release verhindern. Die Fehlerbehebung erfolgt dabei nach dem Grundsatz der minimalen Änderung. Falls es keine einfache minimale Lösung des Fehlers gibt, wird notfalls ein Workaround implementiert.

Die neuen Commits auf dem release-Branch müssen regelmäßig in den develop-Branch gebracht werden. Dadurch werden einmal behobene Fehler auch in der aktuellen Entwicklung beseitigt.

```
> git checkout develop  
> git merge release
```

Wurden im release-Branch Workarounds eingebaut, so werden diese auch in den develop-Branch übernommen. Im develop-Branch können die Workarounds ausgebaut (z. B. mit dem revert-Befehl, Seite 179) und durch eine bessere Implementierung ersetzt werden (Abbildung 20–4).

### Schritt 3: Release erstellen

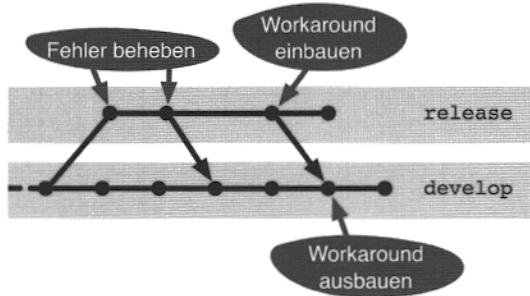
Nachdem der release-Branch erfolgreich getestet wurde, kann das Release erstellt werden.

Dafür muss ein Merge vom release-Branch in den master-Branch durchgeführt werden. Dabei ist es wichtig, dass es keine Commits im

**Branch erstellen**  
→ Seite 62

**Branches zusammenführen**  
→ Seite 67

**Abb. 20–4**  
Umgang mit Bugfixes  
und Workarounds



master-Branch gibt, die noch nicht im release-Branch getestet worden sind.<sup>2</sup> Solche Commits würden zu einem Merge führen, sodass im master-Branch ein Release entsteht, das in dieser Zusammenstellung nicht getestet wurde.

**Commit-Historie zeigen** → Seite 36

Der folgende log-Befehl überprüft, ob es Commits gibt, die im master-Branch vorhanden sind, aber im release-Branch fehlen. Keine Ausgabe bedeutet dabei, dass es keine neuen Commits im master-Branch gibt.

```
> git log release..master --oneline
```

Falls der log-Befehl zu einer Ausgabe führt, so ist ein erneuter Merge vom master- in den release-Branch notwendig, und die Tests für das Release sind nochmals durchzuführen.

Falls die Log-Ausgabe leer bleibt, kann das Merge des release-Branch in den master-Branch durchgeführt werden.

Normalerweise würde Git bei diesem Merge einen Fast-Forward durchführen, da wir ja vorab getestet haben, dass es keine neuen Commits im master-Branch gibt. Um jedoch eine sinnvolle First-Parent-History auf dem master-Branch zu erreichen, erzwingt die Option --no-ff ein neues Commit. Im Kommentar des neuen Commit sollten die wichtigen Informationen zu dem Release hinterlegt werden:

```
> git checkout master
> git merge release --no-ff -m "Release-2.0.0"
```

**--no-ff:** Keinen Fast-Forward-Merge durchführen, d.h., es wird immer ein neues Commit angelegt.

**Fast-Forward-Merges**  
→ Seite 74

Neben dem Commit soll ein neues Tag für das Release angelegt werden. Das Tag dient dazu, schnell auf das Release-Commit zugreifen zu können, z. B. für den diff-Befehl.

<sup>2</sup> Das kann zum Beispiel passieren, wenn ein Hotfix durchgeführt und das Ergebnis nicht mit dem release-Branch vereinigt wurde.

```
> git tag -a release-2.0.0 -m "Release-2.0.0"
```

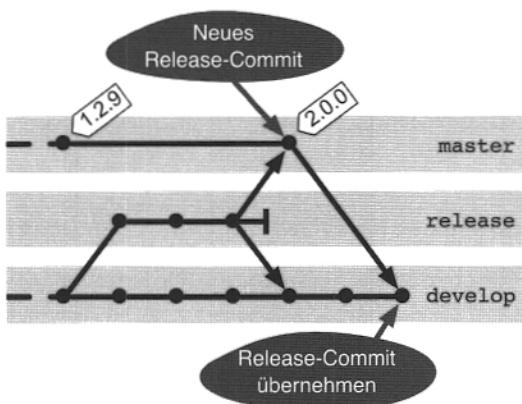
Zum Abschluss wird der `release`-Branch gelöscht. Dieser diente nur zur Stabilisierung und wird beim nächsten Release wieder neu angelegt.

**Branch löschen**  
→ Seite 64

```
> git branch -d release
> git push --delete origin release
```

#### Schritt 4: `develop`-Branch aktualisieren

Nachdem nun das Release durchgeführt wurde, soll garantiert werden, dass alle Änderungen des Release auch im `develop`-Branch enthalten sind.



**Abb. 20-5**  
*Commits beim Releasen*

Auch wenn bereits alle Bugfix-Commits des `release`-Branch in den `develop`-Branch übernommen wurden, existiert immer noch das neue Release-Commit (Abbildung 20-5). Dieses Release-Commit verändert zwar keine Dateien und ist somit unbedeutend für den `develop`-Branch, doch bei Abfragen wie »Welches Commit ist im `master`-Branch, aber nicht im `develop`-Branch?« würde dieses Commit regelmäßig auftauchen. Deswegen wird der `master`-Branch mit dem `develop`-Branch zusammengeführt:

```
> git checkout develop
> git merge master -m "Nach Release-2.0.0"
```

**Branches zusammenführen**  
→ Seite 67

Damit ist das neue Release aus Sicht der Versionsverwaltung vollständig angelegt worden.

#### Hotfix durchführen

Ein *Hotfix* ist eine dringliche Änderung, die schnellstmöglich und unabhängig von anderen Änderungen verteilt werden soll. Ein Hotfix wird

direkt gegen den Stand des aktuellen Release implementiert. Weniger wichtige Fehler werden in Webanwendungen typischerweise mit dem nächsten Release behoben. Doch ein Fehler, der das Arbeiten mit dem System unmöglich macht bzw. zu Sicherheitsrisiken führen kann, muss sofort behoben werden.

### Schritt 1: hotfix-Branch anlegen und Fehler beheben

Die Behebung des Fehlers findet in einem eigenen hotfix-Branch statt. Um auch die parallele Bearbeitung von mehreren Hotfixes zu ermöglichen, bekommt jeder Hotfix einen eigenen Branch.

#### *Branch erstellen*

→ Seite 62

Ausgangspunkt ist dabei der master-Branch. Dieser zeigt auf das letzte produktive Release:

```
> git checkout -b hotfix/a1 master
```

Nun können die notwendigen Änderungen am Projekt durchgeführt werden.

### Schritt 2: Überprüfen, ob parallele Hotfixes stattgefunden haben

#### *Commit-Historie*

zeigen → Seite 36

Ist der Fehler behoben und soll ein neues Release erzeugt werden, muss überprüft werden, ob es in der Zwischenzeit bereits einen anderen Hotfix gab. Dazu nutzt man den log-Befehl, um Commits zu finden, die im master-Branch enthalten sind, aber nicht im hotfix-Branch:

```
> git log hotfix/a1..master --oneline
```

Falls es zu einer Ausgabe von Commits kommt, dann gab es zwischenzeitlich Änderungen am master-Branch. Bevor der Hotfix eingespielt werden kann, muss überprüft werden, ob die übrigen Änderungen mit dem Hotfix zusammen funktionieren. Dazu müssen die Änderungen des master-Branches in den hotfix-Branch mit einem Merge überführt werden.

```
> git merge master
```

Anschließend führt man nochmals die notwendigen Tests durch.

### Schritt 3: Hotfix freigeben

#### *Fast-Forward-Merges*

→ Seite 74

Um den Hotfix offiziell freizugeben, wird der hotfix-Branch mit dem master-Branch vereinigt. Auch hier wird ein Fast-Forward-Merge unterdrückt, um ein neues Commit anzulegen. Im Kommentar des Merge gibt man die notwendigen Release-Informationen an:

```
> git checkout master
> git merge hotfix/a1 --no-ff -m "Hotfix-Release-2.0.1"
```

Neben dem Commit wird wieder ein neues Tag für das Release angelegt:

```
> git tag -a release-2.0.1 -m "Hotfix-Release 2.0.1"
```

Zum Schluss kann der hotfix-Branch gelöscht werden:

```
> git branch -d hotfix/a1
```

**Versionen markieren**

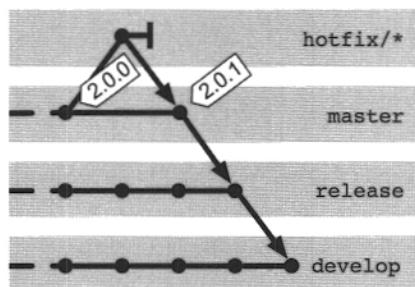
→ Seite 105

**Branch löschen**

→ Seite 64

#### Schritt 4: Hotfix-Änderungen in andere Branches übernehmen

Fehler, die mit einem hotfix-Branch behoben wurden, müssen in die anderen aktiven Branches übertragen werden.



**Abb. 20–6**  
Hotfixes in den  
release- und den  
develop-Branch  
übernehmen

Trat der Hotfix während einer Release-Phase auf, dann muss der Hotfix erst in den release-Branch gebracht werden. Anschließend werden dann die Änderungen des release-Branch in den develop-Branch übertragen (Abbildung 20–6):

```
> git checkout release
> git merge master -m "Hotfix 2.0.1"
> git checkout develop
> git merge release -m "Hotfix 2.0.1"
```

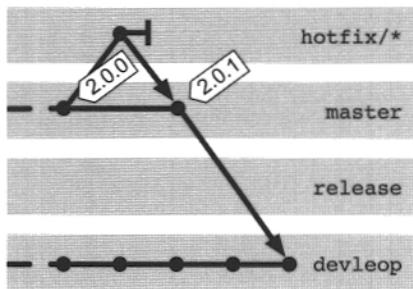
**Branches zusammenführen**

→ Seite 67

Trat der Hotfix nicht während einer Reelase-Phase auf, werden die Änderungen direkt in den develop-Branch übernommen (Abbildung 20–7):

```
> git checkout develop
> git merge master -m "Hotfix 2.0.1"
```

**Abb. 20-7**  
Hotfixes in den  
develop-Branch  
übernehmen



## Git-Flow – ein ähnliches Branch-Modell

Vincent Driessen<sup>3</sup> beschreibt in seinem Blog-Post ein Branch-Modell, das unter den Namen *Git-Flow* bekannt geworden ist.

Das Git-Flow-Modell unterscheidet sich kaum von dem hier beschriebenen Workflow. Die Namen der Branches sind identisch. Das Git-Flow-Modell geht jedoch immer von einem Feature-Branch-Workflow aus, siehe »Mit Feature-Banches entwickeln« (Seite 143). Außerdem gibt es noch eine kleine Abweichung beim Mergen. Im Git-Flow-Modell wird der release-Branch gleichzeitig in den master-Branch und den develop-Branch übertragen, während in dem von uns beschriebenen Branch-Modell der release-Branch erst in den master-Branch und dann in den develop-Branch übertragen wird.

Alles in allem sind sich die beiden Modelle aber sehr ähnlich.

## Warum nicht anders?

### Warum nicht nur mit Tags?

**Versionen markieren**

→ Seite 105

In dem beschriebenen Workflow werden ein master-Branch und zusätzlich Tags für die Kennzeichnung der Releases benutzt. Würde der Einsatz von Tags nicht ausreichend sein?

Zum reinen Kennzeichnen und damit Reproduzieren der Releases würde tatsächlich der Einsatz von Tags reichen.

Wenn es jedoch darum geht, die Historie der Releases und der Hotfix-Releases nachzuvollziehen, dann sind Tags alleine unpraktisch. So kann anhand der Tag-Namen die chronologische Reihenfolge nur erraten werden. Mit einem master-Branch kann man dagegen die First-Parent-History benutzen.

<sup>3</sup> <http://nvie.com/posts/a-successful-git-branching-model>

## Warum nicht auf Tags verzichten?

Tags sind symbolische Namen für Commits. Möchte man den aktuellen Entwicklungsstand mit einem bestimmten Release vergleichen (`diff`-Befehl), dann sind Tags praktischer als die Hashwerte:

```
> git diff release-1.0.0
```

Würde man auf Tags verzichten, müsste man als Erstes im `master`-Branch das richtige Commit suchen und dann den Hashwert angeben:

```
> git diff 5d0173d
```

## Warum keine Fast-Forward-Merges?

In Git sind Branches nur Referenzen auf Commits. Wenn ein Branch aktiviert ist (`checkout`-Befehl), dann wird die Referenz bei jedem neuen Commit automatisch aktualisiert. Es gibt keine historischen Informationen, welches Commit in welchem Branch angelegt wurde. Die einzige »heuristische« Möglichkeit besteht darin, die First-Parent-History eines Branch zu nutzen.

Ein Fast-Forward-Merge führt dazu, dass zwei Branches auf dasselbe Commit zeigen.<sup>4</sup> Wenn jetzt die First-Parent-History benutzt wird, ist nicht mehr nachvollziehbar, in welchem der beiden Branches die Vorgänger-Commits angelegt wurden.

Unterdrückt man einen Fast-Forward-Merge, wird immer ein neues Commit angelegt. Der erste Parent zeigt auf das letzte Commit des aktuellen Branch und der zweite Parent auf das hinzugefügte Commit.

### **Fast-Forward-Merges**

→ Seite 74

### **First-Parent-History**

→ Seite 75

## Warum den Hotfix nicht direkt auf dem `master`-Branch implementieren?

Der Workflow beschreibt, dass für das Beheben eines schweren Fehlers ein separater `hotfix`-Branch angelegt werden soll. Prinzipiell wäre es auch möglich, direkt im `master`-Branch zu arbeiten.

Dadurch würden jedoch unter Umständen in der First-Parent-History des `master`-Branch Commits auftauchen, für die es keine zugeordneten Releases gab. Das passiert immer dann, wenn für einen Hotfix mehr als ein Commit angelegt werden muss.

Auch das parallele Erstellen von Hotfixes kann erschwert werden.

### **First-Parent-History**

→ Seite 75

<sup>4</sup> Dieses Verhalten ist auch in vielen Fällen sinnvoll, z. B. wenn zwei Branches gegenseitig den `merge`-Befehl aufrufen. Auf diese Weise entstehen keine unnötigen leeren Commits.



## 21 Mit mehreren aktiven Releases arbeiten

Bei der Entwicklung von Softwareprodukten ist es typischerweise notwendig, mit mehreren aktiven Releases gleichzeitig umzugehen. Nicht alle Kunden können immer und zu jedem Zeitpunkt auf die aktuellste Version der Software wechseln. Treten schwerwiegende Fehler und Sicherheitsrisiken auf, müssen diese sehr schnell in allen aktiven Releases behoben werden (sogenannte Hotfixes). Es muss sichergestellt werden, dass alle Hotfixes aus älteren Versionen auch in die aktiveren Versionen einfließen. Unter Umständen ist auch ein *Backport* eines Features oder Hotfix von einer neueren Version auf eine ältere Version notwendig.

Vor dem Veröffentlichen eines neuen Release soll dieses ausführlich getestet werden. Weiterentwicklung und Hotfixes sollen parallel entwickelt werden können.

Dieser Workflow zeigt, wie ein Release-Prozess für ein Produkt mit mehreren aktiven Releases so umgesetzt wird, dass

- Hotfixes auf mehreren aktiven Releases unterstützt werden,
- die parallele Arbeit am neuen Release während der Testphase möglich ist,
- garantiert ist, dass alle Fehler, die während der Testphase behoben werden, in alle neueren Releases und den Entwicklungsstand zurückfließen,
- Backports von neueren Releases auf ältere Releases unterstützt werden und
- Vergleiche zwischen verschiedenen Releases und Entwicklungsständen einfach möglich sind.

## Überblick

Abbildung 21–1 zeigt die Branches, die für die Entwicklung und den Release-Prozess benötigt werden.

**Mit Feature-Branches entwickeln** → Seite 143

**Gemeinsam auf einem Branch entwickeln**

→ Seite 135

**Versionen markieren**

→ Seite 105

Die Entwicklung findet auf dem master-Branch statt. Dabei ist es unerheblich, ob Feature-Branches benutzt werden oder nicht. Wichtig ist nur, dass der master-Branch den Code enthält, der zum nächsten Release führen soll.

Für jedes aktive Release gibt es einen eigenen Branch. Ein neuer Release-Branch wird ausgehend vom master-Branch angelegt, sobald die Entwicklung abgeschlossen ist und die Testphase für das neue Release beginnen soll. Parallel zur Testphase kann auf dem master-Branch für das nächste Release weiterentwickelt werden.

Sobald die Testphase abgeschlossen ist, wird auf dem Release-Branch ein neues Release-Tag angelegt. Die Änderungen während der Testphase werden immer durch Merges in den master-Branch zurückgeführt.

Soll ein Hotfix durchgeführt werden, wird dieser auf dem Branch des ältesten betroffenen aktiven Release umgesetzt. Anschließend wird durch Merges der Hotfix in die neueren Releases übernommen. Zum Schluss wird durch ein weiteres Merge der Hotfix in den master-Branch zurückgeführt.

Sind Hotfixes nur in älteren Releases notwendig, aber nicht in den aktuellen Releases, wird trotzdem ein Merge durchgeführt. Dabei werden aber die Änderungen nicht übernommen (Merge-Strategie OURS). Dadurch wird sichergestellt, dass nachfolgende Merges mit relevanten Hotfixes keine unbeabsichtigten Seiteneffekte haben.

Wurde ein Hotfix in einem aktuelleren Release durchgeführt und soll er in ein älteres Release übernommen werden (Backport), so werden die relevanten Commits mit dem cherry-pick-Befehl kopiert. Anschließend wird wieder mit der Merge-Kaskade der Hotfix in die aktuelleren Releases und den master-Branch übernommen.

## Voraussetzungen

**Wenig Entwicklungsarbeit auf älteren Releases:** Auf älteren Releases werden nur Hotfixes und minimale Feature-Erweiterungen durchgeführt.

**Ähnliche Dateistruktur zwischen Releases:** Die zugrunde liegende Aufteilung der Verzeichnisse und Dateien muss zwischen allen aktiven Releases ähnlich sein. Ansonsten sind die Merges der Hotfixes zwischen den aktiven Releases nicht möglich.

Workflow kompakt

## Mit mehreren aktiven Releases arbeiten

Für ein Softwareprodukt werden mehrere Releases parallel betreut. Jedes Release wird auf einem separaten Branch vorbereitet. Hotfixes können zwischen Releases ausgetauscht werden.

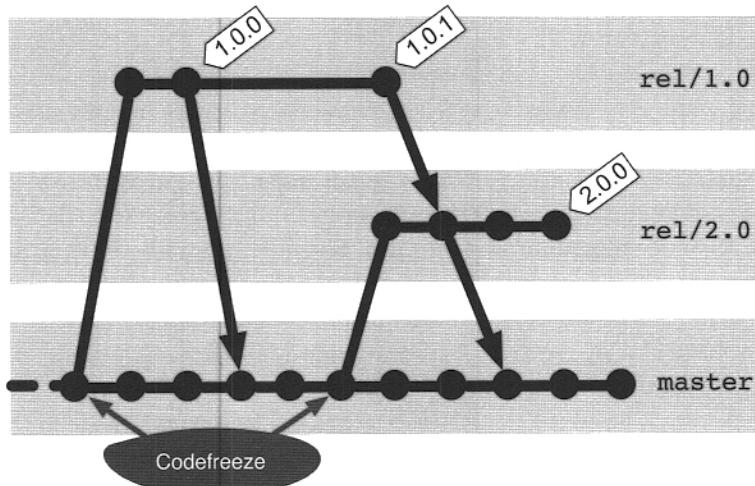


Abb. 21-1  
Workflow im Überblick

## Ablauf und Umsetzung

### Release vorbereiten und erstellen

Der folgende Abschnitt beschreibt, welche Schritte notwendig sind, um ein neues Release zu erstellen.

Die Entwicklung des Projekts findet auf dem master-Branch statt. Auf diesem werden auch die notwendigen Unitests und Integrations-tests durchgeführt.

Sobald die Entwicklung abgeschlossen ist und das Release durchgeführt werden soll, werden typischerweise noch weitere und intensivere Tests vorgenommen. Während der Testphase werden nur noch release-kritische Bugfixes und ggf. Workarounds implementiert. Wie lange diese Phase dauert, hängt stark vom Entwicklungsprozess, von der vorhandenen Codequalität und von den Testaufwänden ab. Es kann sich dabei um einige Stunden oder um mehrere Wochen handeln.

Damit nicht die Entwicklung für das nächste Release während der Testphase stockt, wird die Stabilisierung des Release schon auf dem release-Branch durchgeführt.

Bei der Benennung der Releases gehen wir von dem Muster MAJOR.MINOR.PATCH<sup>1</sup> aus. Immer wenn ein Hotfix durchgeführt wird, wird nur die letzte Nummer erhöht. Die Namen der Release-Banches enthalten nur den MAJOR.MINOR-Bestandteil. Die Release-Tags bekommen das vollständige Muster.

#### Schritt 1: release-Branch anlegen

##### Branch erstellen

→ Seite 62

Der release-Branch wird basierend auf dem aktuellen master-Branch angelegt. Der checkout-Befehl kann genutzt werden, um den neuen Branch anzulegen und zu aktivieren. Wir gehen im Folgenden vom Release 2.0.0 aus, d. h., der Release-Branch bekommt den Namen 2.0. Für die bessere Unterscheidung von Release-Banches und normalen Branches legen wir noch das Präfix release fest:

```
> git checkout -b release/2.0 master
```

#### Schritt 2: Stabilisierung mit dem release-Branch

Im release-Branch werden nur Fehler behoben, die ein Release verhindern. Die Fehlerbehebung erfolgt dabei nach dem Grundsatz der minimalen Änderung. Falls es keine einfache minimale Lösung des Fehlers gibt, wird notfalls ein Workaround implementiert.

---

<sup>1</sup> <http://semver.org>

Die neuen Commits auf dem release-Branch müssen regelmäßig in den master-Branch gebracht werden. Dadurch werden einmal behobene Fehler auch in der aktuellen Entwicklung beseitigt.

**Branches zusammenführen**  
→ Seite 67

```
> git checkout master  
> git merge release/2.0
```

Wurden im release-Branch Workarounds eingebaut, so werden diese auch in den master-Branch übernommen. Im master-Branch können die Workarounds ausgebaut (z. B. mit dem revert-Befehl, Seite 179) und durch eine bessere Implementierung ersetzt werden (Abbildung 20–4).

**Versionen markieren**  
→ Seite 105

### Schritt 3: release-Tag anlegen

Nachdem der release-Branch erfolgreich getestet wurde, kann das Release abgeschlossen werden. Dazu wird ein neues Tag für das Release angelegt. Da es sich um das erste Release für diesen Branch handelt, beginnen wir mit der Patch-Nummer 0. Mit der Verwendung des Präfix release ergibt sich der Tag-Name: release/2.0.0.

```
> git tag -a release/2.0.0 -m "Release-2.0.0"
```

Spätestens nach dem Anlegen des Release-Tags müssen die Commits, die in dem Release-Branch angelegt worden sind, durch einen Merge in den master-Branch übernommen werden (siehe Schritt 2).

## Hotfix in mehreren Releases durchführen

Ein Hotfix ist eine dringliche Änderung, die schnellstmöglich und unabhängig von anderen Änderungen verteilt werden soll. Sind mehrere Releases beim Kunden im Einsatz, kann es notwendig sein, den Hotfix für alle aktiven Releases umzusetzen.

**Hotfix in ein älteres Release übertragen**  
→ Seite 207

### Schritt 1: Aktive Releases ermitteln

Ein Hotfix sollte in dem ältesten betroffenen Release umgesetzt werden und dann in die aktuelleren übernommen werden. Deswegen ist es notwendig, alle aktiven Releases zu ermitteln und dann zu untersuchen, ob das Problem in dem jeweiligen Release existiert.

Das Ermitteln ist dank des Präfix release sehr einfach. Dabei gehen wir davon aus, dass die Release-Banches für inaktive Releases gelöscht worden sind (siehe »Release deaktivieren« ab Seite 206).

```
> git branch --list release/*
release/1.0
release/1.1
release/1.2
release/1.3
release/2.0
```

**--list:** Listet alle Branches auf, die einem bestimmten Muster entsprechen.

Anschließend muss der älteste aktive Branch ermittelt werden, bei dem das Problem auftritt, z. B. wird release/1.2 ermittelt.

### Schritt 2: hotfix-Branch anlegen und Fehler beheben

**Mit Feature-Branches entwickeln** → Seite 143

**Branch erstellen**  
→ Seite 62

Der Hotfix wird auf einem separaten Branch durchgeführt und erst nach Abschluss auf den Release-Branch übernommen. Dadurch ist es möglich, auch parallel mehrere Hotfixes durchzuführen.

Der Ausgangspunkt ist dabei der release-Branch des ältesten betroffenen Release:

```
> git checkout -b hotfix/a1 release/1.2
```

Nun können die notwendigen Änderungen am Projekt durchgeführt werden.

### Schritt 3: Überprüfen, ob parallele Hotfixes stattgefunden haben

**Commit-Historie zeigen** → Seite 36

Ist der Fehler behoben und soll ein neues Release erzeugt werden, muss überprüft werden, ob es in der Zwischenzeit bereits einen anderen Hotfix gab. Dazu nutzt man den log-Befehl, um Commits zu finden, die im release-Branch enthalten sind, aber nicht im hotfix-Branch:

```
> git log hotfix/a1..release/1.2 --oneline
```

Falls es zu einer Ausgabe von Commits kommt, dann gab es zwischenzeitlich Änderungen am release-Branch. Bevor der Hotfix eingespielt werden kann, muss überprüft werden, ob die übrigen Änderungen mit dem Hotfix zusammen funktionieren. Dazu müssen die Änderungen des release-Branhes in den hotfix-Branch mit einem Merge überführt werden.

```
> git merge release
```

Anschließend führt man nochmals die notwendigen Tests durch.

#### Schritt 4: Hotfix in den release-Branch übernehmen

Zum Abschluss wird der hotfix-Branch mit dem release-Branch vereinigt. Ist auf dem Hotfix-Branch mehr als ein Commit angelegt worden, muss beim Merge darauf geachtet werden, dass ein Fast-Forward unterdrückt wird. Dadurch wird immer ein Merge-Commit angelegt, und es ist einfacher, den Hotfix auf ältere Releases mit dem cherry-pick-Befehl zu übertragen (siehe »Hotfix in ein älteres Release übertragen« ab Seite 207).

**Fast-Forward-Merges**  
→ Seite 74

```
> git checkout release/1.2
> git merge hotfix/a1 --no-ff -m "Hotfix a1 abgeschlossen"
```

Neben dem Commit wird wieder ein neues Tag für das Release angelegt. Dabei muss die Patch-Nummer des letzten Release-Tags für diesen Branch um eins erhöht werden:

**Versionen markieren**  
→ Seite 105

```
> git tag --list release/1.2.*
release/1.2.0
release/1.2.1
```

**--list:** Listet alle Tags auf, die einem bestimmten Muster entsprechen.

Damit bekommt das nächste Release-Tag den Namen release/1.2.2:

```
> git tag -a release/1.2.2 -m "Hotfix-Release 1.2.2"
```

Zum Schluss kann der hotfix-Branch gelöscht werden:

**Branch löschen**  
→ Seite 64

```
> git branch -d hotfix/a1
```

#### Schritt 5: Hotfix in die aktuelleren Release-Banches übernehmen

Der Hotfix wurde in das älteste betroffene Release eingebaut und muss auf alle neueren Releases übertragen werden. Dazu wird schrittweise von einem Release zum nächstaktuelleren Release ein Merge durchgeführt (Merge-Kaskade) und nach erfolgreichem Testlauf auch ein Release-Tag angelegt:

```
> git checkout release/1.3
> git merge release/1.2
> git tag -a release/1.3.5 -m "Hotfix-Release 1.3.5"
> git checkout release/2.0
> git merge release/1.3
> git tag -a release/2.0.1 -m "Hotfix-Release 2.0.1"
```

Zum Abschluss werden die Änderungen noch in den master-Branch übernommen:

```
> git checkout master  
> git merge release/2.0
```

**Merge manuell durchführen**  
→ Seite 72

Manchmal stellt man fest, dass ein Hotfix eines älteren Release im aktuellen Release gar nicht mehr notwendig ist. Zum Beispiel wurden die betroffenen Codezeilen bereits anderweitig geändert oder entfernt. Trotzdem muss der Merge durchgeführt werden. Man nutzt jedoch die Merge-Strategie ours, um die Änderungen des Hotfix zu ignorieren. Bei zukünftigen Merges von weiteren Hotfixes wird dieser Hotfix dann nicht mehr übertragen.

```
> git checkout release/2.0  
> git merge -s ours release/1.3 -m "Hotfix a1 ignoriert"
```

**-s ours:** Erzeugt ein Merge-Commit mit dem aktuellen Branch und dem angegebenen Branch als Parents. Es werden jedoch keine Änderungen des angegebenen Branch übernommen. Das heißt, das neue Commit wird sich von dem Vorgänger-Commit des aktuellen Branch nicht unterscheiden. Bei zukünftigen Merges werden jedoch die Änderungen des angegebenen Branch bereits als enthalten angesehen und nicht noch mal übertragen.

Auf das Release-Tag kann bei ignorierten Hotfixes verzichtet werden, da ja keine Änderungen einfließen und es somit auch kein neues Release für diesen Branch geben muss.

## Release deaktivieren

Je mehr aktive Releases existieren, umso mehr Aufwand entsteht bei der Umsetzung von Hotfixes. Diese müssen in jedem aktiven Release nachvollzogen und getestet werden. Deswegen ist es sinnvoll, die Anzahl der aktiven Releases zu minimieren.

Sobald das Ende des Release den Kunden mitgeteilt wurde, kann der zugehörige Branch im Repository gelöscht werden:

```
> git branch -d release/1.0
```

Die Tags bleiben weiterhin bestehen, und wenn es notwendig ist, kann ein neuer Branch für das Release angelegt werden.

## Hotfix in ein älteres Release übertragen

Im Abschnitt »Hotfix in mehreren Releases durchführen« ab Seite 203 sind wir davon ausgegangen, dass Hotfixes in dem ältesten betroffenen Release behoben werden.

Manchmal kann es passieren, dass ein Hotfix in einem aktuelleren Release eingebaut wurde und dass dann doch noch in ein älteres Release übertragen werden muss.

In unserem Beispiel gehen wir davon aus, dass ein Hotfix vom 1.2-Release auf das 1.0-Release übernommen werden soll.

### Schritt 1: Hotfix-Commit identifizieren

Als Erstes muss der relevante Hotfix-Commit identifiziert werden. Da wir darauf geachtet haben, dass Hotfixes auf separaten Branches entwickelt und dann durch einen Merge in den release-Branch übertragen werden, handelt es sich um genau einen Commit.<sup>2</sup>

```
> git log --oneline --grep="Hotfix a1"
d52fe0a Hotfix a1 abgeschlossen
```

**--grep:** Sucht in der Log-Meldung nach einem bestimmten Text.

### Schritt 2: Hotfix-Commit kopieren

Wenn das Commit identifiziert wurde, kann es mit dem cherry-pick-Befehl auf einen anderen Release-Branch kopiert werden. Bei dem identifizierten Hotfix-Commit wird es sich meistens um ein Merge-Commit handeln. Bei einem Merge-Commit muss der cherry-pick-Befehl mit einer Parent-Nummer aufgerufen werden. Erst dadurch ist das Parent-Commit eindeutig, zu dem das Delta berechnet werden soll, das dann auf den aktuellen Branch kopiert wird. Da wir die Änderungen bezogen auf den release-Branch kopieren wollen, wird es der erste Parent sein (Abbildung 21–2).

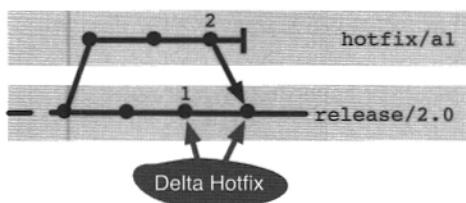


Abb. 21–2  
Parent beim  
Cherry-Pick auswählen

<sup>2</sup> Natürlich können bei dem log-Befehl auch mehrere Commits gefunden werden. Es sollte aber nur einer sein, der den Hotfix in den Release-Branch übertragen hat.

Zuerst muss der Branch aktiviert werden, auf den das Commit übertragen werden soll. Dann wird der cherry-pick-Befehl ausgeführt.

```
> git checkout release/1.0
> git cherry-pick -x --mainline 1 d52fe0a
```

**-x:** Fügt zum Commit-Kommentar noch die Zeile »CHERRY PICKED FROM COMMIT ...« hinzu.

**--mainline:** Bestimmt den Parent, der für die Delta-Berechnung benutzt werden soll.

Nach den notwendigen Tests wird ein neues Release-Tag angelegt:

```
> git tag -a release/1.0.4 -m "Hotfix-Release 1.0.4"
```

### Schritt 3: Hotfix in die aktuelleren Release-Banches übernehmen

Auch bei diesem Ablauf müssen die Änderungen durch einen Merge in alle aktuelleren aktiven Releases übertragen werden. Da das Commit allerdings von einem aktuellen Branch kopiert wurde, existiert der Hotfix bereits in einigen Branches. Trotzdem ist es notwendig, die Merges durchzuführen, damit bei zukünftigen Hotfixes das kopierte Commit nicht unkontrolliert doppelt übertragen wird.

Das heißt, für alle Release-Banches, die älter sind als der originale Hotfix-Branch (`release/1.2`), wird ein normaler Merge durchgeführt. In unserem Beispiel wäre das nur der `release/1.1`-Branch:

```
> git checkout release/1.1
> git merge release/1.0
> git tag -a release/1.1.6 -m "Hotfix-Release 1.1.6"
```

Für den originalen Hotfix-Branch (`release/1.2`) wird ein Merge mit der `ours`-Strategie durchgeführt. Da der Hotfix schon enthalten ist, muss er nicht noch mal übertragen werden.<sup>3</sup> Es ist auch kein Release-Tag anzulegen, da es bereits ein Release-Tag mit dem Inhalt gibt.

**Achtung!** Überprüfen Sie vorher genau, dass wirklich nur das kopierte Hotfix-Commit durch einen Merge übernommen würde. Ansonsten gehen alle sonstigen Änderungen verloren.

```
> git checkout release/1.2
> git merge -s ours release/1.1
```

---

<sup>3</sup>In vielen Fällen würde auch ein normaler Merge funktionieren, da Git die Änderung nicht zweimal anwenden würde. Wenn es aber zu Nacharbeiten beim Cherry-Pick gekommen ist, kann es Probleme geben.

Alle Releases, die aktueller sind als der originale Hotfix-Branch (`release/1.2`), können wieder den normalen Merge benutzen. Es werden dabei keine Änderungen einfließen, da der vorherige Merge mit der OURS-Strategie für keine Neuerungen gesorgt hat. Es ist trotzdem sinnvoll, die Merges durchzuführen, um sicher zu sein, dass alle Hotfixes in allen aktiven Releases enthalten sind. Die Release-Tags dagegen sind nicht nötig.

```
> git checkout release/1.3  
> git merge release/1.2  
> git checkout release/2.0  
> git merge release/1.3
```

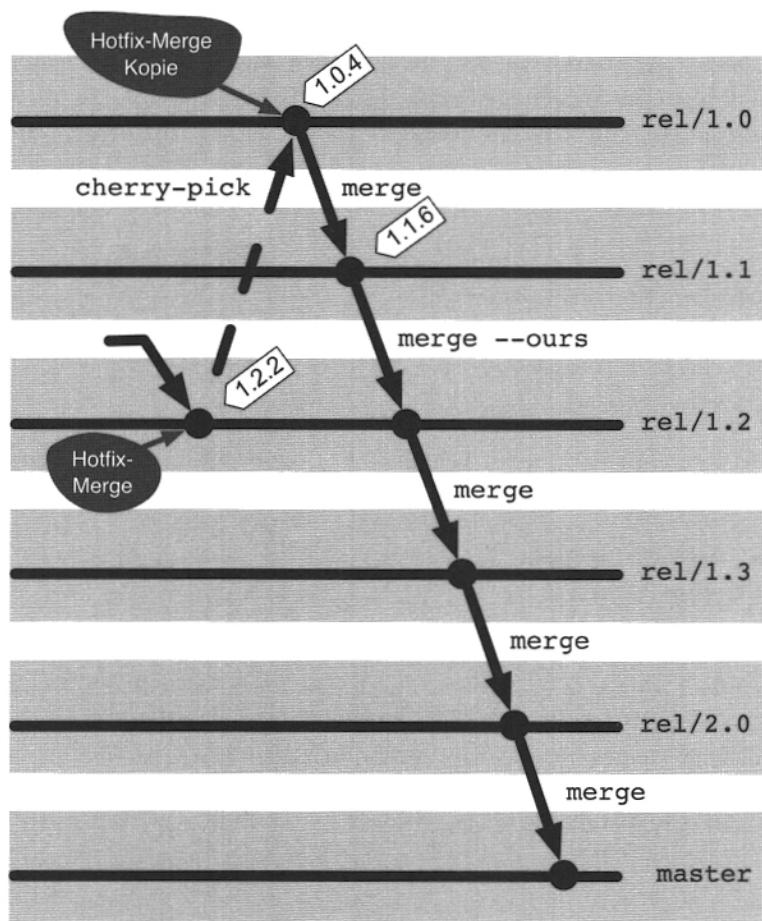
Ganz am Ende muss natürlich das aktuellste Release wieder in den `master`-Branch übernommen werden:

```
> git checkout master  
> git merge release/2.0
```

In Abbildung 21–3 ist der ganze Ablauf noch mal grafisch aufbereitet. Als Erstes wird das `HF`-Commit durch den `cherry-pick`-Befehl kopiert und es entsteht ein `HF'`-Commit. Anschließend wird dieses durch verschiedene Merges bis in den `master`-Branch übertragen.

Im Ergebnis sind alle Commits von älteren Releases durch Merges in jedes aktuellere Release eingeflossen.

**Abb. 21-3**  
Cherry-Pick und Merges  
im Überblick



## Warum nicht anders?

### Warum nicht nur mit Cherry-Pick arbeiten?

Der beschriebene Workflow arbeitet normalerweise mit Merges und nur in Ausnahmefällen mit Cherry-Picks. Durch die vielen Merge-Commits entsteht unter Umständen eine unübersichtliche Historie.

Eine Alternative wäre, alle Hotfixes durch Cherry-Picks zu übertragen. Das heißt, ein Hotfix wird auf einem beliebigen Release durchgeführt und dann in alle aktiven betroffenen Releases kopiert. Dadurch wären die Historien der Release-Branches unabhängig voneinander.

Ein Nachteil der Cherry-Pick-Variante ist, dass es nur schwer nachzuvollziehen ist, ob wirklich alle Bugfixes in alle aktuelleren Releases übertragen worden sind. Es gibt zwar den `cherry`-Befehl, der überprüft, welche Commits per Cherry-Pick kopiert wurden. Doch dieser Befehl

funktioniert nur, wenn es zu keinen Konflikten kam. Durch Konfliktlösungen ändert sich der Inhalt des kopierten Patches, und der cherry-Befehl würde nicht mehr erkennen, dass es sich um ein kopiertes Commit handelt.

Die Konsequenz wäre, dass man außerhalb von Git nachhalten müsste, welche Hotfixes auf welchen Branch kopiert worden sind.



## 22 Ein Projekt mit großen binären Dateien versionieren

Im Workflow »Ein Projekt aufsetzen« (Seite 123) wurde darauf hingewiesen, dass man in einem Git-Repository keine großen (binären) Dateien ablegen soll. Das liegt am Grundkonzept einer dezentralen Versionsverwaltung. Alle Versionen aller Dateien werden in jedem lokalen Repository abgelegt. Bei Quelltextdateien können durch Komprimierung die Datenmengen sehr gering gehalten werden. Bei großen binären Dateien, wie Bildern, Filmen oder virtuellen Maschinen, klappt das nicht so gut, da diese meist schon komprimiert vorliegen. Das Problem ist dabei auch meistens nicht der lokale Speicherplatz, sondern die Netzwerkbandbreite. Bei jedem `clone`- und `fetch`-Befehl müssen alle noch nicht vorhandenen Versionen transportiert werden. Meistens werden aber gar nicht alle Versionen von binären Dateien gleichzeitig benötigt, sondern nur genau die aktuellste oder eine bestimmte Version für ein Release.

Am besten geht man dem Problem ganz aus dem Weg, indem man die binären Artefakte außerhalb von Git versioniert. Falls es allerdings notwendig ist, die binären Dateien zusammen mit den Quelltextdateien zu versionieren, kann man den *Git Large File Storage (LFS)* benutzen. Dieser ermöglicht das externe Speichern von großen Dateien in einen eigenen Storage-Server und das automatische Holen der gerade benötigten Version beim Wechseln des Branch bzw. beim `pull`-Befehl. Dieser Workflow beschreibt,

- wie *LFS* installiert und aktiviert wird,
- wie Dateitypen und Verzeichnisse für das Speichern im *LFS* definiert werden,
- die Besonderheiten beim Arbeiten mit *LFS* und
- wie mit Merge-Konflikten in *LFS*-Dateien umgegangen wird.

## Überblick

### Das Repository

→ Seite 49

Abbildung 22–1 zeigt die verschiedenen Beteiligten bei dem Einsatz des Large File Storage. Es gibt einen lokalen Workspace mit dem lokalen Repository im .git-Verzeichnis und dazu ein verknüpftes zentrales Repository. Zusätzlich wird noch ein Large File Storage Server benötigt.

In der .gitattributes-Datei wird definiert, welche Dateitypen (\*.jpg) oder welche Verzeichnisse (images/) durch das Large File Storage verwaltet werden sollen.

Die optionale .lfsconfig-Datei ermöglicht es, die URL des *LFS*-Servers im Repository abzulegen.

Auf allen Entwicklerrechnern muss der *LFS*-Client installiert und aktiviert werden. Dabei wird ein *LFS*-Filter in Git registriert, der in der .gitattributes-Datei benutzt wird. Bei jedem commit-Befehl werden neue und geänderte Dateien, die von *LFS* verwaltet werden, in eine *Link-Datei* umgewandelt und nur diese wird im Repository gespeichert. Der zugehörige Dateinhalt wird zusätzlich in einem lokalen Cache im .git-Verzeichnis abgelegt. Beim push-Befehl werden die Dateinhalte aus dem lokalen Cache an den *LFS*-Server übertragen.

### Aktiver Branch

→ Seite 61

Wird im Workspace ein neues Commit ausgewählt, z. B. beim Branch-Wechsel oder beim pull-Befehl, wird jede *Link-Datei* durch den Dateinhalt ersetzt. Dazu wird erst im lokalen Cache nach dem Inhalt gesucht. Falls es dort nicht die richtige Version gibt, wird der Inhalt vom *LFS*-Server geladen.

## Voraussetzungen

**Large File Storage Server:** Es wird ein Large File Storage Server benötigt. Die meisten zentralen Repository-Manager sind gleichzeitig auch *LFS*-Server, z. B. Atlassian Bitbucket Server<sup>1</sup>, GitHub<sup>2</sup> oder GitLab<sup>3</sup>. Alternativ gibt es auch Artefakt-Repositorys, die als *LFS*-Server dienen können, z. B. Artifactory<sup>4</sup>.

**LFS Client:** Auf jedem Rechner mit einem Repository, welches *LFS* nutzt, muss der *LFS* Client<sup>5</sup> installiert sein.

---

<sup>1</sup> <https://www.atlassian.com/software/bitbucket/server>

<sup>2</sup> <https://github.com>

<sup>3</sup> <https://gitlab.com/>

<sup>4</sup> <https://www.jfrog.com>

<sup>5</sup> <https://github.com/github/git-lfs>

Workflow kompakt

## Ein Projekt mit großen binären Dateien versionieren

Das Large File Storage (LFS) wird benutzt, um große binäre Dateien separat vom Git-Repository zu speichern und zu versionieren.

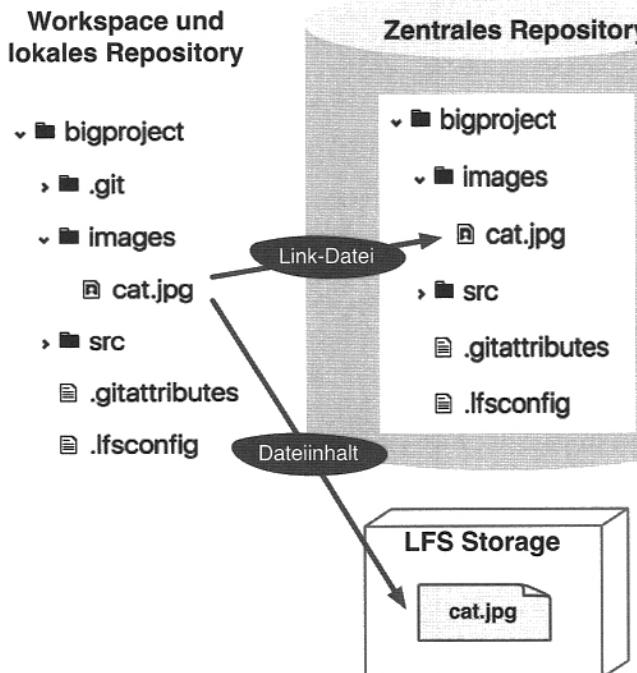


Abb. 22-1  
Workflow im Überblick

## Ablauf und Umsetzung

Die nachfolgenden Abläufe gehen von einem Beispielprojekt `bigproject`, wie in Abbildung 22–1 zu sehen, aus.

```
> cd bigproject
```

Auf das Einrichten und Betreiben eines *LFS*-Servers wird nicht näher eingegangen. Zum Ausprobieren nutzen Sie am besten eine Repository-Verwaltung, die den *LFS*-Server gleich mitbringt (siehe Voraussetzungen).

### **LFS-Client installieren und aktivieren**

Der *LFS*-Client muss auf allen Rechnern, die ein Repository mit *LFS* benutzen wollen, installiert werden. Unter <https://git-lfs.github.com/> finden Sie die Installationspakete für Windows, Linux und MacOS.

*LFS* beruht auf der clientseitigen Umwandlung von großen Dateien in kleine *Link-Dateien*. Dafür wird ein erweiterbarer Filtermechanismus von Git genutzt. Dieser ist dafür gedacht, Dateien vor dem Ablegen im Repository oder nach dem Holen aus dem Repository abzuändern, z. B. um den Code einheitlich zu formatieren, die Zeilenenden an das Betriebssystem anzupassen oder die Inhalte zu verschlüsseln.

Damit der vom *LFS*-Client mitgebrachte Filter benutzt werden kann, muss dieser in Git registriert werden. Dazu sind in der globalen Konfigurationsdatei (`~/.gitconfig`) entsprechende Einträge notwendig. Der `lfs install`-Befehl erledigt die Registrierung:

```
> git lfs install
```

Der `lfs install`-Befehl muss pro Rechner nur einmal ausgeführt werden.

### **LFS für ein Repository einrichten**

#### **Schritt 1: Dateien für die Verwaltung mit *LFS* festlegen**

Wie im vorigen Abschnitt beschrieben, nutzt *LFS* die Filtermechanismen von Git. Der *LFS*-Filter wird in der `.gitattributes`-Datei für Dateien und Verzeichnisse aktiviert.

Die Regeln für die Angabe des Datei- bzw. Verzeichnisnamens in der `.gitattributes`-Datei sind identisch zu den Regeln der `.gitignore`-Datei:

- Dateinamen ohne führenden / gelten auch für alle Unterverzeichnisse (`bigfile.bin`).
- Dateinamen mit führenden / sind absolut und gelten ausgehend vom aktuellen Verzeichnis (`/bigfile.bin`).
- Dateinamen mit / am Ende kennzeichnen Verzeichnisse inklusive aller Unterverzeichnisse (`bigfiles/`).
- Wildcards mit \* sind erlaubt (`*.bin`).

Das Hinzufügen von neuen Einträgen in die `.gitattributes`-Datei übernimmt der `lfs-track`-Befehl. Man gibt ein oder mehrere Dateien und Verzeichnisse an:

```
> git lfs track images/
```

Bei der Angabe von Wildcards müssen die Dateinamen in Anführungszeichen eingeschlossen werden, da ansonsten die aktuell vorhandenen Dateien und nicht der Name mit den Wildcards eingetragen wird:

```
> git lfs track '*.jpg'
```

Der `lfs-track`-Befehl ändert nur die `.gitattributes`-Datei. Die passenden Dateien bleiben unangetastet. Allerdings wird man beim nächsten `status`-Befehl sehen, dass alle betroffenen Dateien als geändert markiert sind. Fügt man diese Dateien zum nächsten Commit hinzu und führt das Commit aus, werden die Dateien in Links umgewandelt. In alten Commits bleiben die Dateien weiterhin vollständig erhalten. Nur für zukünftige Commits greift die Optimierung von *LFS*<sup>6</sup>. Die `.gitattributes`-Datei muss ebenso wie jede normale Datei versioniert werden.

```
> git add images/cat.jpg
> git add .gitattributes
> git commit
```

Wenn man *LFS* für ein vorhandenes Repository mit mehreren Branches aktiviert, muss man das Anlegen der `.gitattributes`-Datei für jeden Branch wiederholen.

Um Dateien von der Verwaltung mit *LFS* wieder auszunehmen, gibt es den `lfs-untrack`-Befehl:

```
> git lfs untrack images/ '*.jpg'
```

**Commits zusammenstellen**  
→ Seite 39

<sup>6</sup> Möchte man auch in alten Commits die großen Dateien per *LFS* auslagern, dann gibt es unter <https://github.com/bozaro/git-lfs-migrate> ein Migrationswerkzeug. Das Ergebnis ist ein neues Repository mit neuen Commits.

Der `lfs-untrack`-Befehl entfernt nur die Einträge für die Dateien aus der `.gitattributes`-Datei. Erst bei der nächsten Änderung einer betroffenen Datei wird der Inhalt wieder im Repository gespeichert. Alte Commits beinhalten weiterhin nur die *Link-Dateien*.

Um zu sehen, welche Dateinamen bzw. -verzeichnisse mit *LFS* verwaltet werden, ruft man den `lfs-track`-Befehl ohne Dateinamen auf:

```
> git lfs track
```

Um alle konkreten Dateien aufzulisten, die durch *LFS* im aktuellen Commit verwaltet werden, nutzt man den `lfs-ls-files`-Befehl:

```
> git lfs ls-files
```

### Schritt 2: *LFS*-Server festlegen

#### *Ein Projekt aufsetzen*

→ Seite 123

Bisher haben wir *LFS* nur lokal eingerichtet und definiert, welche Dateien verwaltet werden. Es fehlt noch die Konfiguration, unter welcher URL der *LFS*-Client den *LFS*-Server findet.

Dabei geht der *LFS*-Client folgende Liste der Reihe nach durch und nimmt die erste verfügbare Option:

- der Inhalt des `lfs.url`-Konfigurationsparameters
- der Inhalt des `remote.<name>.lfsurl`-Konfigurationsparameters
- Wenn der Zugriff auf das zentrale Repository über HTTP(S) stattfindet, wird an die Remote-URL des zentralen Repository der Postfix `/info/lfs` angehängt.

#### *Das Repository-Layout*

→ Seite 89

Die Konfigurationsparameter werden aus den normalen Konfigurationsdateien gelesen. Allerdings gibt es zusätzlich noch die Möglichkeit, eine `.lfsconfig`-Datei in das Repository zu legen. Diese wird immer als Erstes herangezogen, um die Konfigurationsparameter zu lesen. Dadurch ist es möglich, die *LFS*-Server-URL mit dem Repository auszuliefern.

Wenn man eine passende Repository-Verwaltung hat und HTTP(S) als Transferprotokoll nutzt, dann muss man die *LFS*-URL nicht definieren. Das Hinzufügen von `/info/lfs` an die Remote-URL wird funktionieren.

In allen anderen Fällen ist es notwendig, die *LFS*-Server-URL zu definieren. Am einfachsten ist das, indem man den `lfs.url`-Parameter in der `.lfsconfig`-Datei setzt:

```
> git config --file .lfsconfig lfs.url=<lfs-server-url>
```

**--file:** Der `config`-Befehl schreibt die Werte in die übergebene Datei.

Die `.lfsconfig`-Datei muss anschließend noch zum Commit hinzugefügt und versioniert werden:

```
> git add .lfsconfig  
> git commit
```

## Mit einem *LFS*-Repository arbeiten

Das Arbeiten mit einem Repository, das *LFS* nutzt, unterscheidet sich kaum von dem Arbeiten mit einem Repository ohne *LFS*.

### Repository klonen

Das Klonen wird genauso durchgeführt wie immer:

```
> git clone <repo-mit-lfs-url>
```

*Repositories erstellen, klonen und verwalten*  
→ Seite 89

Es wird allerdings nach dem eigentlichem Klonen des Repository noch ein Download für jede *LFS*-Datei gestartet. Je nach Anzahl und Größe der Dateien und der Qualität der Netzwerkanbindung kann das lange dauern. Eine Optimierung ist möglich, indem der `lfs-clone`-Befehl benutzt wird. Dieser startet nicht für jede Datei separat ein Download, sondern er optimiert die Anzahl.

```
> git lfs clone <repo-mit-lfs-url>
```

### Besonderheiten beim Arbeiten mit *LFS*-Dateien

*LFS* arbeitet bei den meisten Kommandos völlig transparent. So werden *LFS*-Dateien beim Commit automatisch in Link-Dateien umgewandelt, beim Branch-Wechsel und beim Pull vom *LFS*-Server geholt und beim Push zum *LFS*-Server übertragen.

Es gibt einen `lfs-status`-Befehl, um zu sehen, welche *LFS*-Dateien noch nicht zum *LFS*-Server übertragen wurden:

```
> git lfs status  
On branch master  
Git \Stichwort{LFS} objects to be pushed to origin/master:  
  
images/cat.jpg (15 KB)
```

Ein weiterer Unterschied ist beim `diff`-Befehl zu sehen. Als Ergebnis werden nicht die Änderungen am Dateiinhalt, sondern in den *Link-Dateien* angezeigt:

*Unterschiede zwischen Commits* → Seite 35

```
> git diff
```

```
diff --git a/images/cat.jpg b/images/cat.jpg
index a21d03a..d4f6fea 100644
--- a/images/cat.jpg
+++ b/images/cat.jpg
@@ -1,3 +1,3 @@
version https://git-lfs.github.com/spec/v1
-oid sha256:ff717526...
-size 23
+oid sha256:8d1a9b35...
+size 15
```

### Merge-Konflikte bei LFS-Dateien

**Branches zusammenführen**  
→ Seite 67

Im vorigen Abschnitt hat man gesehen, dass Git beim Diff nicht den eigentlichen Dateiinhalt, sondern die *Link-Dateien* vergleicht. Dasselbe passiert auch im Falle eines Merge-Konflikts, d. h., im Workspace befindet sich nach einem missglückten Merge die *Link-Datei* mit Konfliktmarkierungen. Da es sich bei LFS-Dateien typischerweise um große Binärdateien handelt, ist ein normaler Merge meistens auch nicht möglich. Das Beste, was man tun kann, ist, eine der beiden Versionen zu wählen. Dafür gibt es den checkout-Befehl mit den Optionen `--ours` (eigene Version) und `--theirs` (Version des anderen Branch).

**Bearbeitungskonflikte**  
→ Seite 70

```
> git checkout --ours images/cat.jpg
```

Als Ergebnis befindet sich die gewählte Dateiversion im Workspace. Danach geht es normal mit der Konfliktauflösung weiter.

Gibt es ein Merge-Werkzeug für die Binärdateien, kann man dieses in Git konfigurieren (siehe config-Befehl) und anschließend den mergetool-Befehl aufrufen. Dieser Befehl speichert den tatsächlichen Dateiinhalt aller Versionen in temporären Dateien und übergibt diese an das konfigurierte Merge-Werkzeug, sodass man in dem Merge-Werkzeug die Dateien zusammenfügen kann.

```
> git mergetool
```

### Alle LFS-Dateien holen und unnötige Dateien löschen

**Und was ist, wenn man die Commit-Objekte wirklich loswerden will?** → Seite 66

Für Backup-Zwecke oder wenn man plant, längere Zeit offline zu sein, ist es möglich, alle LFS-Dateien in den lokalen Cache zu holen. Dafür gibt es den `lfs-fetch`-Befehl mit der Option `--all`:

```
> git lfs fetch --all
```

Um überflüssige Dateien aus dem lokalen Cache zu entfernen, gibt es den `lfs-prune`-Befehl:

```
> git lfs prune
```

## 23 Große Projekte aufteilen

Häufig beginnt ein Softwareprojekt als kleines monolithisches System. Im Laufe der Entwicklung wächst das Projekt und das Team wird größer. Modularisierung wird immer wichtiger. Als Erstes wird typischerweise die interne Struktur des Projekts modularisiert. Irgendwann möchte man auch einzelne Module separat entwickeln und einem eigenen Release-Zyklus unterwerfen.

Da Git-Repositorys immer als Ganzes versioniert werden, muss für Module, die separat veröffentlicht werden sollen, auch ein neues Git-Repository angelegt werden.

Die Herausforderung bei der Modularisierung eines Git-Repositorys besteht darin, so viel wie möglich der alten Dateiversionen in das neue Repository zu übernehmen. Gleichzeitig soll das neue Repository keine Dateien enthalten, die nicht innerhalb des Moduls verwendet werden. Auch Commits, in denen keine Änderungen an den Dateien des Moduls durchgeführt wurden, werden nicht benötigt.

Im Gesamt-Repository wird die Historie des Moduls nicht entfernt, damit auch alte Projektstände reproduziert werden können. In der Konsequenz werden dann die historischen Daten des separierten Moduls in beiden Repositorys liegen.

Meistens wird das separierte Modul weiterhin vom Gesamtprojekt benötigt und soll als externes Modul integriert werden. Für diese Art der externen Einbindung gibt es in Git das Submodulkonzept.

Dieser Workflow zeigt, wie man mit Git ein Modul so extrahiert, dass

- nur die notwendigen Dateien des Moduls in ein neues Repository übertragen werden,
- die Historie der Moduldateien im neuen Repository erhalten bleibt und
- das Modul wieder als externes Submodul eingebunden werden kann.

*Repositorys können nur vollständig verwendet werden*

→ Seite 300

*Abhängigkeiten zwischen Repositorys*

→ Seite 275

## Überblick

Für den folgenden Ablauf nehmen wir eine Projektstruktur wie in Abbildung 23–1 oben an.

Das Beispiel für diesen Workflow orientiert sich an einer Java-Verzeichnisstruktur. In einem Gesamtprojekt gibt es drei Module. Die Dateien der Module sind auf die Unterverzeichnisse `src` und `test` aufgeteilt. Das heißt, ein Modul besteht jeweils aus zwei getrennten Teilen. Das Modul `modul3` soll in ein eigenes Git-Repository separiert werden.

Im ersten Schritt werden aus einem Klon des originalen Repositories mit dem `filter-branch`-Befehl alle unnötigen Dateien und Commits entfernt. Anschließend wird die Verzeichnisstruktur des neuen Modul-Repositoriums angepasst. Als Letztes wird im Gesamtprojekt das Modul `modul3` entfernt und das neue Modul-Repository als Submodul im Verzeichnis `extern` wieder eingebunden. Das Ergebnis sieht so aus wie in Abbildung 23–1 unten.

Im neuen Modul-Repository wird es möglich sein, die historischen Änderungen an den Dateien nachzuvollziehen, d. h., zu verfolgen, wer wann was geändert hat. Es wird jedoch meistens nicht möglich sein, alte Versionen vollständig (compilefähig) zu reproduzieren. Das liegt daran, dass ein Modul häufig aus Dateien anderer Module hervorgegangen ist. Würde man versuchen, eine alte Projektversion aus dem Modul-Repository wiederherzustellen, entstünde ein Flickenteppich von Dateien in verschiedenen Verzeichnissen. Außerdem war in der Vergangenheit das Modul bestimmt von anderen Dateien abhängig, die nun nicht mehr vorhanden sind.

Im Gesamt-Repository sind alte Versionen des Gesamtprojekts inklusive der Moduldateien immer wiederherstellbar.

## Voraussetzungen

**Interne Modularisierung:** Das Projekt wurde bereits intern modularisiert, d. h., es gibt ein Modul, das separat entwickelt und versioniert werden kann.

**Moduldateien liegen in wenigen Verzeichnissen:** Beim Extrahieren der alten Versionen der Moduldateien muss jedes Verzeichnis separat behandelt werden. Wenn die Dateien sehr verstreut sind, wird der Aufwand sehr groß.

Workflow kompakt

## Große Projekte aufteilen

Ein Modul wird aus einem Projekt entfernt und in ein eigenes Repository migriert. Die Commit-Historie bleibt erhalten, unnötige Dateien und Commits werden entfernt. Das separierte Modul wird als externes Submodul wieder integriert.

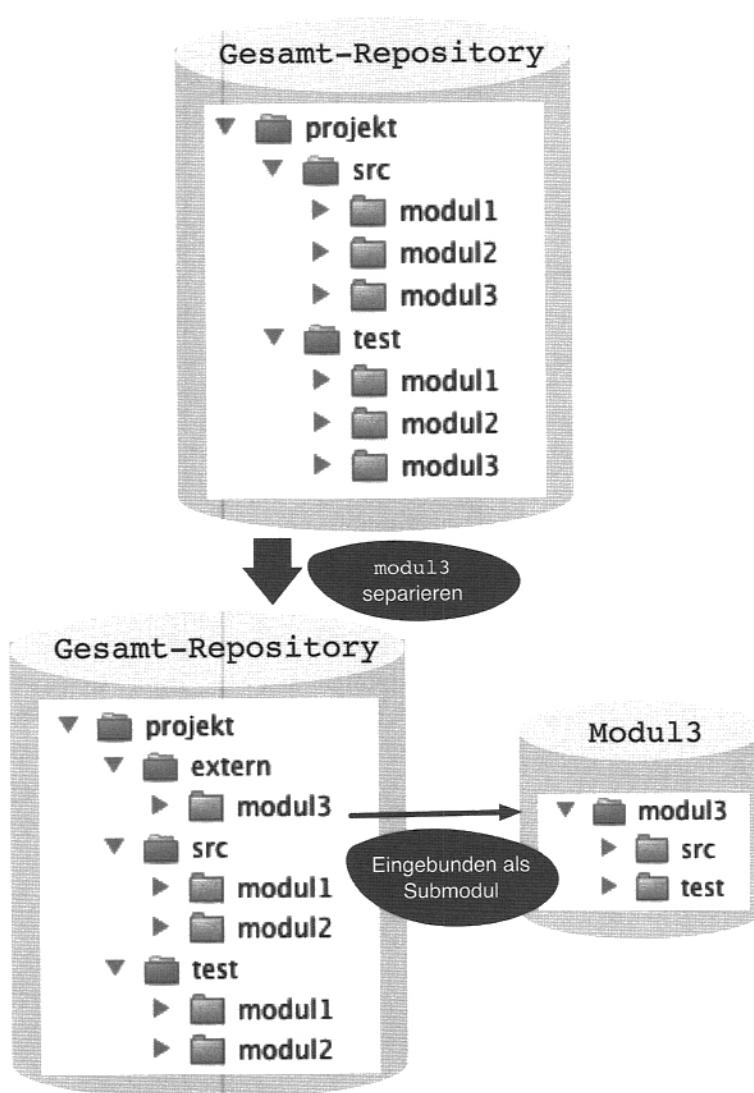


Abb. 23-1  
Workflow im Überblick

## Ablauf und Umsetzung

*Versehentlich  
gelöschten Branch  
wiederherstellen  
→ Seite 65*

Achtung! Einige der folgenden Befehle verändern das Repository sehr grundlegend. Auch wenn es in Git oft möglich ist, Änderungen rückgängig zu machen, sollten Sie unbedingt ein Backup Ihres Repositorys anlegen, bevor Sie mit den weiteren Schritten beginnen.

```
> git clone --no-hardlinks --bare \  
      projekt.git projekt.backup.git
```

**--no-hardlinks:** Diese Option garantiert, dass das geklonte Repository und das originale Repository keine Dateien teilen.

## Modul-Repository separieren

### Schritt 1: Gesamt-Repository klonen

Als Ausgangspunkt für das Modul-Repository wird eine Kopie des Gesamt-Repositorys erzeugt:

```
> git clone --no-hardlinks  
      --bare projekt.git modul3-work.git
```

### Schritt 2: Unnötige Dateien und Commits entfernen

Als Nächstes müssen die unnötigen Dateien und Commits entfernt werden. Das ist der aufwendigste Schritt und entscheidend für die Erhaltung der Historie.

Zum Entfernen von Repository-Inhalten gibt es den `filter-branch`-Befehl. Dieser erzeugt für jedes vorhandene Commit ein neues Commit. Durch unterschiedliche Filter können die neuen Commits verändert werden.

Der folgende `filter-branch`-Befehl entfernt das Verzeichnis `src/modul1` aus den Commits:

```
> cd modul3  
  
> git filter-branch --force  
      --index-filter  
        'git rm -r --cached --ignore-unmatch src/modul1'  
      --tag-name-filter cat  
      --prune-empty -- --all
```

- index-filter 'git rm -r -cached -ignore-unmatch ...':** Mit dieser Option können Dateien aus einem Commit entfernt werden. Der enthaltene `rm`-Befehl wird für jedes Commit ausgeführt. In unserem Beispiel entfernen wir das `src/modul1`-Verzeichnis.  
Wenn Ihr Projekt nicht so modular aufgebaut ist, dann müssen Sie entsprechend mehr Dateien und Verzeichnisse entfernen.<sup>1</sup>
- tag-name-filter cat:** Diese Option erzeugt für Tags an vorhandenen Commits gleichnamige Tags an den neuen Commits.
- prune-empty:** Die Option entfernt alle Commits, die durch die vorherigen Filter keine Dateien mehr enthalten – also »unnötig« sind.
- all:** Die Option wendet den Filter auf alle Branches des Projekts an.

Für das Beispielprojekt müssen wir den Befehl mehrfach auch für die Verzeichnisse `test/modul1`, `src/modul2` und `test/modul2` aufrufen.

Die genaue Beschreibung aller Optionen entnehmen Sie bitte der Git-Hilfe für den `filter-branch`-Befehl.

### Schritt 3: Unnötige Branches und Tags entfernen

In dem entstandenen Modul-Repository sind nicht alle Tags und Branches sinnvoll, z. B. diejenigen Tags und Branches, die keinen Bezug zum extrahierten Modul haben. Die unnötigen Branches und Tags werden entfernt.

```
> git tag -d v1.0.1
> git branch -D v2.0_bf
```

**Versionen markieren**

→ Seite 105

**Branch löschen**

→ Seite 64

### Schritt 4: Verkleinern des Modul-Repositorys

Damit Git auch wirklich alle unnötigen Dateien aus den internen Verwaltungsdaten entfernt, ist ein nochmaliges Klonen notwendig:<sup>2</sup>

```
> git clone --no-hardlinks
--bare modul3-work.git modul3.git
```

Das bisherige Modul-Repository `modul3-work.git` wird nicht mehr benötigt und kann gelöscht werden:

```
> rm -rf modul3-work.git
```

<sup>1</sup> Wenn Ihr Modul in der Projekthistorie aus anderen Modulen hervorgegangen ist, können Sie mit dem `log`-Befehl die Vorgängerdateien suchen. Wenn Sie auch diese Versionen erhalten wollen, dürfen diese Dateien nicht mit dem `filter-branch`-Befehl gelöscht werden.

<sup>2</sup> Der `gc`-Befehl wäre nicht ausreichend, um sofort alle unnötigen Dateien zu löschen. GC ist sehr defensiv implementiert und löscht unnötige Commits, Trees und Blobs erst nach einer konfigurierbaren Wartezeit.

**Schritt 5: Verzeichnisstruktur des Modul-Repositorys anpassen**

Bisher sieht die Verzeichnisstruktur des neuen Modul-Repositorys genauso aus wie die des Gesamtprojekts. Nur die unnötigen Module fehlen. Die Anpassung der Verzeichnisstruktur kann nun durch normale Dateioperationen erfolgen. Dazu ist ein Klon mit Workspace notwendig:

```
> git clone modul3.git modul3
```

Das `src/modul3`-Verzeichnis wird in `src` umbenannt und das `test/modul3`-Verzeichnis in `test`:

```
> cd modul3
> mv src/modul3 modul3
> rmdir src
> mv modul3 src
> mv test/modul3 modul3
> rmdir test
> mv modul3 test
```

*Commits zusammenstellen*

→ Seite 39

Anschließend werden die Änderungen normal mit dem `commit`-Befehl bestätigt und mit dem `push`-Befehl ins Bare-Repository übertragen:

```
> git add --all
> git commit -m "Verzeichnisstruktur angepasst"
> git push
```

*Tipp: Alte Branches ignorieren*

Falls es weitere Branches im Modul-Repository gibt, dann müssen die Dateioperationen auf allen Branches durchgeführt werden.

Es ergibt häufig wenig Sinn, die Branches des Gesamtprojekts zu übernehmen. Das Modul beginnt einen neuen Release-Zyklus, und die alten Branches sind uninteressant.

**Schritt 6: Modulverzeichnisse aus Gesamt-Repository entfernen**

Nachdem das separierte Modul in ein eigenes Repository migriert wurde, wird in den nächsten Schritten das Gesamt-Repository angepasst. Die nun unnötigen Verzeichnisse des separierten Moduls müssen entfernt werden: `src/modul3` und `test/modul3`. Die Anpassung erfolgt ganz normal auf Dateiebene in einem Klon des Gesamt-Repositorys.

Falls es im Projekt weitere Branches gibt, die an die neue Struktur angepasst werden sollen, müssen die Änderungen dort ebenso vorgenommen werden. Der `cherry-pick`-Befehl kann hilfreich sein, um die Änderungen automatisch in mehrere Branches zu übertragen.

## Modul-Repository als externes Repository einbinden

Nach dem vorherigen Ablauf gibt es zwei getrennte Repositorys. Normalerweise wird das Gesamtprojekt aber weiterhin das separierte Modul benötigen – deswegen ist eine Integration notwendig.

Die Integrationsmöglichkeiten hängen sehr stark von der verwendeten Entwicklungsplattform ab. So würde man in Java-Maven-Projekten das separierte Modul einzeln bauen und die entstehenden Artefakte in einem Maven-Repository ablegen. Im Gesamtprojekt würde man die Artefakte als Abhängigkeit definieren und während des Builds aus dem Maven-Repository holen.

Falls man die Integration mit Git durchführen möchte, steht das Konzept der Submodule zur Verfügung. Mit Submodulen können Verzeichnisse in einem Git-Repository mit anderen Git-Repositorys verknüpft werden.

*Komplizierter Umgang mit Submodulen*  
→ Seite 299

### Schritt 1: Externes Modul ins Gesamt-Repository einbinden

In unserem Beispiel wollen wir das Repository von modul3 in das Verzeichnis extern/modul3 des Gesamtprojekts einbinden. Ausgangspunkt ist ein Klon des Gesamt-Repositories. Wir befinden uns im Wurzelverzeichnis des Projekts und fügen mit dem submodule add-Befehl das Modul-Repository hinzu. Der erste Parameter ist der Pfad oder die URL zu dem Modul-Repository, und der zweite Parameter ist das zukünftige Verzeichnis im aktuellen Repository:

```
> git submodule add /global-path-to/modul3.git  
      extern/modul3
```

Der submodule add-Befehl erzeugt einen Klon des externen Repositorys in dem angegebenen Verzeichnis. Zusätzlich wird im aktuellen Repository protokolliert, dass das Verzeichnis ein externes Repository referenziert.

Das Verzeichnis extern/modul3 zeigt jetzt auf den aktuellsten Commit (HEAD) des externen Repositorys. In der Anleitung »Neue Version eines Submoduls verwenden« (Seite 279) erfahren Sie, wie ein anderes Commit ausgewählt werden kann.

Noch ist das Submodul nur im Workspace sichtbar. Erst durch einen commit-Befehl werden die Änderungen in das Repository übernommen:

```
> git add --all  
> git commit -m "Modul3 hinzugefügt"
```

*Submodule einbinden*  
→ Seite 277

*Commits zusammenstellen*  
→ Seite 39

Mit dem bekannten push-Befehl können die Submodulverknüpfungen zum zentralen Repository übertragen werden.

## Warum nicht anders?

### Warum kein neues Repository?

**Ein Projekt aufsetzen**

→ Seite 123

Eine Alternative zu diesem Workflow wäre es, für das Modul einfach ein neues Repository anzulegen. Damit würde es keine Projekthistorie des Moduls im neuen Repository geben. Es wäre weiterhin möglich, im Gesamt-Repository die alten Stände zu finden.

Solange diese Einschränkung Sie nicht stört, ist diese Lösung sehr einfach umzusetzen.

### Warum nicht den --subdirectory-filter verwenden?

In diesem Workflow wird der filter-branch-Befehl mit dem --index-filter benutzt. Dieser erlaubt es, Dateien aus Commits zu entfernen.

Der Subdirectory-Filter --subdirectory-filter des filter-branch-Befehls entfernt dagegen alle Dateien außer dem angegebenen Verzeichnis. Zusätzlich wird das ausgewählte Verzeichnis zur neuen Wurzel des Repositorys.

Solange das zu separierende Modul genau in einem Verzeichnis liegt, ist dieser Befehl einfach anzuwenden. In unserem Beispiel war jedoch das Modul auf zwei Verzeichnisse aufgeteilt, und somit konnte dieser Filter nicht eingesetzt werden.

Auch wenn einzelne Dateien des Moduls ehemals in anderen Verzeichnissen lagen oder das Modulverzeichnis umbenannt wurde, wird der Subdirectory-Filter die Historien nur unvollständig importieren.

## 24 Kleine Projekte zusammenführen

In der Anfangsphase eines Projekts werden häufig Prototypen für kritische Designentscheidungen und Technologien implementiert. Ist die Evaluierung abgeschlossen, möchte man die erfolgreichen Prototypen als Grundlage für die erste Version des Gesamtprojekts kombinieren.

In einem solchen Szenario werden die Prototypen häufig in einem eigenen Repository versioniert. Sobald das Gesamtprojekt gestartet wird, ist ein gemeinsames Repository erwünscht, d. h., die Dateien der verschiedenen Prototypen müssen zusammengeführt werden.

In einem anderen Szenario wurden anfangs Projekte zu stark modularisiert und in unterschiedlichen Repositorys versioniert. Später stellte sich heraus, dass oft gleichzeitige Änderungen vorgenommen wurden und Dateien häufig zwischen Repositorys verschoben werden mussten. Auch in diesem Fall ist ein Gesamt-Repository sinnvoller.

Dieser Workflow zeigt, wie man mit Git mehrere Repositorys so zusammenführt, dass

- ─ die Historien aller Dateien und
- ─ die Tags aller Repositorys erhalten bleiben.

## Überblick

**Fetch: Branches aus einem anderen Repository holen**  
→ Seite 99

Dieser Workflow basiert auf der Fähigkeit von Git, in ein Repository Commits von verschiedenen Remote-Repositorys zu importieren (fetch-Befehl). Git setzt nicht voraus, dass die importierten Commits einen gemeinsamen Ursprung haben.

In Abbildung 24–1 sind oben exemplarisch zwei Repositorys namens backend und ui dargestellt.

Nach dem Importieren der Commits in ein Repository existieren zwei getrennte Commit-Historien. Wechselt man zwischen den Commits des backend-Projekts und des ui-Projekts, dann werden im Workspace immer nur die Dateien des jeweiligen Projekts zu sehen sein.

**Branches zusammenführen**  
→ Seite 67

Der entscheidende Schritt zu einem gemeinsamen Projektstand ist, durch einen merge-Befehl die unabhängigen Commit-Historien zusammenzuführen.

Als Vorbereitung für den Merge ist es sinnvoll, bei jedem der Projekte ein neues Wurzelverzeichnis (backend bzw. ui) anzulegen und alle vorhandenen Dateien in dieses Verzeichnis zu verschieben. Nach dem Merge sind dann im Wurzelverzeichnis des Gesamtprojekts das backend-Verzeichnis und das ui-Verzeichnis parallel vorhanden (Abbildung 24–1 unten).

Dieses Vorgehen verhindert, dass es während des merge-Befehls zu Konflikten kommt.

## Voraussetzungen

**Versionen markieren**  
→ Seite 105

**Unterschiedliche Tag-Namen:** Die Projekte müssen eindeutige Tag-Namen benutzen, d. h., es darf keine gleichnamigen Tags in verschiedenen Repositorys geben.<sup>1</sup>

---

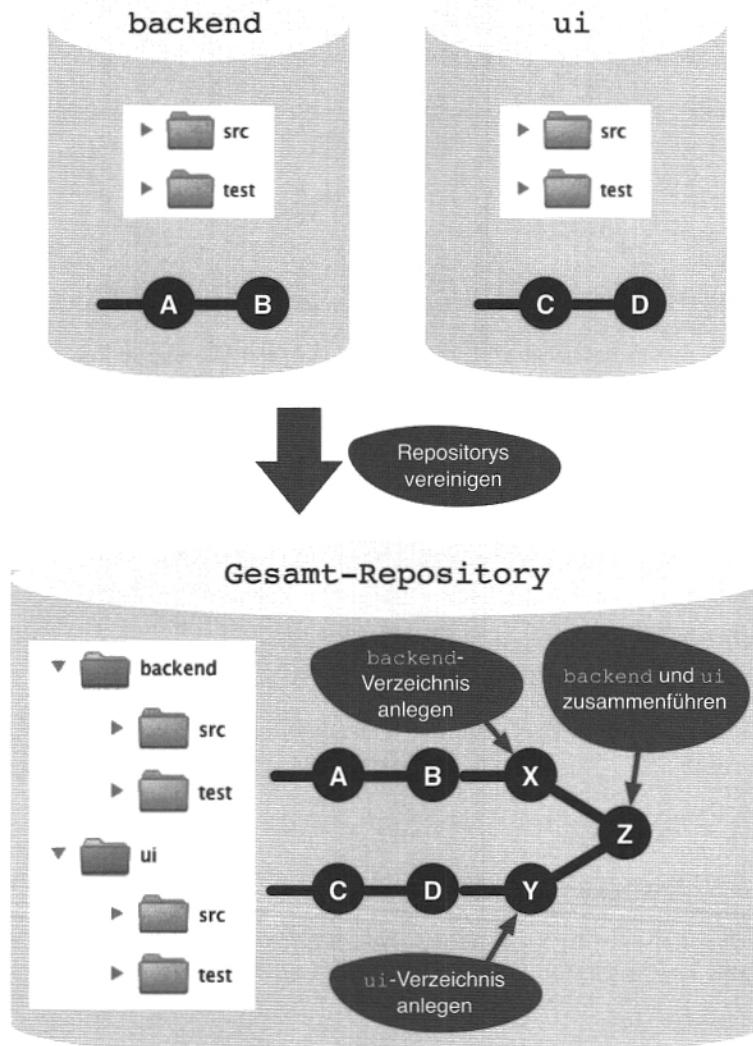
<sup>1</sup>Falls es gleichnamige Tag-Namen gibt, muss man diese Tags löschen und neue Tags mit eindeutigen Namen anlegen.

Workflow kompakt

## Kleine Projekte zusammenführen

Mehrere Projekte mit eigenem Repository werden in einem gemeinsamen Repository vereinigt. Die Commit-Historien der Projekte bleiben erhalten.

Abb. 24-1  
Workflow im Überblick



## Ablauf und Umsetzung

### Repositories vereinigen

Im folgenden Ablauf gehen wir exemplarisch von zwei Repositorys (ui und backend) mit jeweils einem master-Branch aus. Als Ergebnis soll ein Gesamt-Repository entstehen, das ebenso nur einen master-Branch enthält.

#### Schritt 1: Gesamt-Repository anlegen

Als Erstes wird das neue Gesamt-Repository als Klon des backend-Repositorys angelegt und in den neuen Workspace gewechselt:

```
> git clone backend gesamt
> cd gesamt
```

#### Schritt 2: Dateien in Projektverzeichnis verschieben (erstes Repository)

Damit das Zusammenführen mit einem weiteren Projekt zu keinen Dateikonflikten führt, wird ein neues Verzeichnis angelegt:

```
> mkdir backend
```

**Commits zusammenstellen**  
→ Seite 39

Anschließend werden alle Dateien in das neue Verzeichnis verschoben. Dazu nutzt man den mv-Befehl. Dieser verschiebt Dateien und Verzeichnisse auf Betriebssystemebene und führt gleichzeitig die notwendigen add-Befehle und rm-Befehle aus, um die Änderungen ins nächste Commit aufzunehmen:<sup>2</sup>

```
> git mv src test backend
```

Als Letztes werden die Änderungen mit dem commit-Befehl noch abgeschlossen:

```
> git commit -m "backend-Verzeichnis angelegt"
```

#### Schritt 3: Zweites Repository importieren

**Wie sagt man Git, wo das Remote-Repository liegt? → Seite 92**

Um das ui-Repository zu importieren, legen wir ein neues Remote im Gesamt-Repository an:

```
> cd gesamt
> git remote add ui ../ui/
```

---

<sup>2</sup> Der mv-Befehl ist nur eine Erleichterung für die Kommandozeilennutzung. Git erkennt das Verschieben von Dateien auch selbstständig, wenn direkt auf Dateiebene gearbeitet wird. Dann muss nur anschließend das Commit mit dem add-Befehl und dem rm-Befehl vorbereitet werden.

Mit dem `fetch`-Befehl werden alle Git-Objekte (Branches, Tags, Commits) des ui-Repositorys in das Gesamt-Repository importiert:

```
> git fetch ui
```

*Fetch: Branches aus einem anderen Repository holen*  
→ Seite 99

Achtung! Wenn es im ui-Repository Tag-Namen gibt, die im Gesamt-Repository bereits vorhanden sind, werden diese einfach ignoriert.

#### Schritt 4: Dateien in Projektverzeichnis verschieben

Als Nächstes soll im importierten UI-Projekt auch ein neues Projektverzeichnis ui angelegt werden. Da der Branch im UI-Projekt auch `master` heißt, dieser Name aber schon vom Branch des Backend-Projekts benutzt wird, muss man einen anderen Namen für den lokalen Branch wählen (`uimaster`):

```
> git checkout -b uimaster ui/master
```

*Branch erstellen*  
→ Seite 62

**-b:** Es wird ein neuer Branch angelegt und aktiviert.

**uimaster:** der Name des lokalen Branch

**ui/master:** die Referenz auf den Branch `master` im Remote des ui-Repositorys

Das Anlegen des Projektverzeichnisses und das Verschieben der Dateien ist identisch mit Schritt 2:

```
> mkdir ui  
> git mv src test ui  
> git commit -m "ui-Verzeichnis angelegt"
```

#### Schritt 5: Projekte zusammenführen

Nachdem beide Projekte in das Gesamt-Repository importiert wurden und jeweils in einem eigenen Projektverzeichnis liegen, wird in diesem Schritt der Merge durchgeführt.

Der Merge soll im `master`-Branch stattfinden, deswegen wird dieser nun aktiviert:

```
> git checkout master
```

*Branches zusammenführen*  
→ Seite 67

Mit dem `merge`-Befehl wird der `uimaster`-Branch mit dem `master`-Branch zusammengeführt. Da beide Projekte unterschiedliche Projektverzeichnisse haben, kann es zu keinen Merge-Konflikten kommen.

```
> git merge uimaster
```

Das Ergebnis der Merge-Operation kann man sich mit dem »grafischen« log-Befehl veranschaulichen. Es ist sehr gut zu erkennen, dass die Commits der beiden originalen Projekte unabhängig voneinander entstanden sind.

```
> git log --graph --oneline
*   e40fc2b Merge branch 'uimaster'
|\ 
| * ace51c9 ui-Verzeichnis angelegt
| * 40feb24 foo und bar hinzugefuegt
* f8bd134 backend-Verzeichnis angelegt
* fa1482a bar hinzugefuegt
* bddfa53 foo hinzugefuegt
```

**Branch löschen**  
→ Seite 64

Der uimaster-Branch kann nun gelöscht werden, da er nur temporär für den Merge benötigt wurde:

```
> git branch -d uimaster
```

Damit existiert ein Gesamt-Repository, in dem die Historien und die Tags beider Projekt-Repositories enthalten sind.

## Warum nicht anders?

### Warum nicht ohne neue Projektverzeichnisse?

Kann man nicht auf Schritt 2 und 4 verzichten? Warum müssen die Dateien der Projekte in eigene Verzeichnisse verschoben werden?

Würde man die neuen Verzeichnisse nicht anlegen, dann versucht der merge-Befehl, die Wurzelverzeichnisse der beiden Projekte und die enthaltenen Dateien zu vereinigen. Gleichnamige Dateien würden vereinigt werden, ggf. müssten Merge-Konflikte gelöst werden.

Wenn man zwei bisher autarke Projekte vereinigt, dann ist es in den seltensten Fällen sinnvoll, gleichnamige Dateien zusammenzubringen. In den meisten Fällen wird man eine Datei umbenennen oder verschieben wollen. Das wiederum ist einfacher direkt auf Dateiebene durchzuführen als mitten in einer Merge-Operation.

Der beschriebene Ablauf mit neuen Modulverzeichnissen ermöglicht es, nach dem Merge die Dateien auf Dateiebene neu zu organisieren und anschließend wieder zu versionieren.

## 25 Lange Historien auslagern

Git-Repositorys haben – trotz effizienter Speicherverwaltung<sup>1</sup> – die Tendenz, im Laufe der Zeit immer größer zu werden. Dieser Effekt ist meistens vernachlässigbar, wenn im Repository nur Sourcecode versioniert wird. Die Größe eines solchen Repositorys ist für aktuelle Festplatten und Netzwerkbandbreiten nicht relevant.

Werden jedoch auch große binäre Dateien (Bibliotheken, Release-Artefakte, Testdatenbanken, Bilder) versioniert, kann die Repository-Größe unangenehm auffallen.

Was bei einer zentralen Versionsverwaltung nur auf dem Serversystem zu mehr Ressourcenverbrauch führen wird, trifft bei einer dezentralen Versionsverwaltung alle Entwickler. Beim Klonen wird immer das gesamte Repository mit allen historischen Dateien kopiert.

Dieser Workflow beschreibt, wie man die Historie eines Git-Repositorys so auslagern kann, dass

- das neue Projekt-Repository weniger Ressourcen verbraucht und es trotzdem möglich ist, Recherchen (`log`-Befehl, `blame`-Befehl, `annotate`-Befehl) mit den alten Commits durchzuführen.

*Ressourcenverbrauch bei großen binären Dateien → Seite 300  
Ein Projekt mit großen binären Dateien versionieren → Seite 213*

---

<sup>1</sup>Dateien werden nur einmal gespeichert, unabhängig davon, ob sie in verschiedenen Commits vorkommen. Zusätzlich werden alle Dateien gepackt.

## Überblick

Dieser Workflow hat drei Säulen:

**replace-Befehl:** Der replace-Befehl kann temporär die Vorgängerbeziehung eines Commits verändern.

**filter-branch-Befehl:** Der filter-branch-Befehl kann alle Commits eines Repositorys kopieren und dabei verändern. Die manipulierte Vorgängerbeziehung wird dauerhaft entfernt.

**alternates-Datei:** Mit der alternates-Datei können Commits aus anderen Repositorys eingebunden werden.

In Abbildung 25–1 ist oben der Ausgangsstand unseres Projekt-Repositorys skizziert. Es gibt drei Commits: A, B und c. Die Historie vor dem c-Commit soll entfernt werden.<sup>2</sup>

Als Erstes wird mithilfe des replace-Befehls der Vorgänger des c-Commits entfernt. Anschließend wird mit dem filter-branch-Befehl ein neues Projekt-Repository erzeugt, das nur noch das veränderte c-Commit beinhaltet. Damit ist die Auslagerung auch schon abgeschlossen. Die Entwicklung findet nun nur noch auf dem neuen Projekt-Repository statt. Das bisherige Projekt-Repository dient nur noch als Archiv.

Um Recherchen in der ganzen Historie durchzuführen, wird mit der alternates-Datei das Archiv-Repository in das neue Projekt-Repository eingebunden. Mit dem replace-Befehl bekommt das c'-Commit den historisch korrekten Vorgänger zugewiesen (Abbildung 25–1 unten).

## Voraussetzungen

**Koordinierte Unterbrechung:** Alle Teammitglieder müssen einer gleichzeitigen Unterbrechung der Arbeit mit dem Repository zustimmen und anschließend auf dem neuen Klon weiterarbeiten.

**Historie wird nur selten benötigt:** Wenn die historischen Informationen sehr häufig und von vielen Entwicklern benötigt werden, dann ist es sinnvoller, den größeren Ressourcenverbrauch zu akzeptieren.

**Commit-Hashwerte sind egal:** Die Hashwerte von Git können benutzt werden, um unerlaubte Änderungen an alten Versionsständen zu entdecken. Dieser Workflow schreibt jedoch die Historie neu und erzeugt dabei neue Commit-Hashwerte.

---

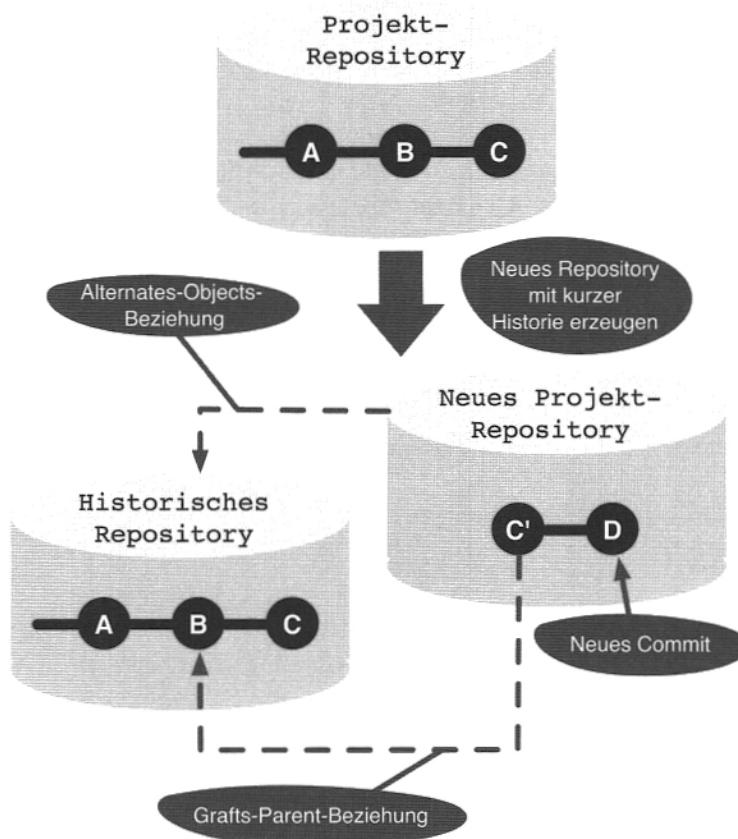
<sup>2</sup>Natürlich ist es auch möglich, mehr von der Historie im Entwicklungsrepository zu belassen.

Workflow kompakt

## Lange Historien auslagern

Ein Repository, das eine sehr lange Commit-Historie mit vielen und großen Dateien enthält, wird verkleinert. Die älteren Commits werden in ein separates Repository ausgelagert. Recherchen in der Historie sind weiterhin möglich.

Abb. 25-1  
Workflow im Überblick



## Ablauf und Umsetzung

### Historie auslagern

Dieser Abschnitt beschreibt im Detail, wie die Historie eines Repositorys ausgelagert werden kann. Genauer gesagt wird ein neues Repository erzeugt, das nur noch die gekürzte Historie beinhaltet.

**Achtung!** Die Klone des alten Repositorys werden nicht mit dem neuen Repository zusammenarbeiten können. Deswegen müssen alle Änderungen der Entwickler vor den nächsten Schritten in das zentrale Repository integriert werden. Alle Entwickler sind zu informieren, dass keine weiteren Änderungen in den Klonen vorgenommen werden können.

Der Ausgangspunkt ist in Abbildung 25–1 oben zu sehen. Das Beispiel geht von einem Bare-Repository mit drei Commits im master-Branch aus. Es soll ein neues Projekt-Repository mit dem c-Commit angelegt werden.

**Commit-Historie zeigen** → Seite 36

Für die nachfolgenden Schritte benötigen wir die vollständigen Hashwerte der Commits c und b. Diese kann man mit dem folgenden log-Befehl ermitteln:

```
> cd projekt.git  
> git log --pretty=oneline  
166a7e047a85b318720dc6e857a5321f9a3df7b4 C  
dcbddd5cd590de3d30e1ecca1882c9187e7eab95 B  
577b8e2cf613c43ed969453477fadc189482c1fb A
```

**--pretty=oneline:** Gibt die Log-Ausgabe in einer Zeile aus. Im Gegensatz zu --oneline wird allerdings der vollständige Hashwert ausgegeben.

### Schritt 1: History-Tag anlegen

Dieser Schritt dient als Vorbereitung, um das zukünftige Archiv-Repository einbinden zu können. Zum Einbinden des Archivs wird man zukünftig den Hashwert des letzten historischen Vorgänger-Commits kennen müssen. In unserem Fall ist das das b-Commit. Eine elegante Lösung, um diese Information dauerhaft abzulegen, besteht darin, ein Tag (`history-cut/master`) am neuen »ersten« Commit (c-Commit) anzulegen und den Hashwert im Tag-Kommentar abzulegen. Dieses Tag wird auch im zukünftigen Projekt-Repository vorhanden sein. Es ergibt keinen Sinn, ein Tag am b-Commit anzulegen, da dieses im neuen Projekt-Repository nicht enthalten sein wird.

Der tag-Befehl bekommt den Hashwert des c-Commits übergeben, und der Hashwert des b-Commits wird in der Tag-Beschreibung hinterlegt.

**Versionen markieren**  
→ Seite 105

```
> git tag -a  
history-cut/master  
166a7e047a85b318720dc6e857a5321f9a3df7b4  
-m "Vorgänger: dcddd5cd590de3d30e1ecca1882c9187e7eab95"
```

**history-cut/master:** Der Name des neuen Tags. In Git ist es möglich, Tag- und Branchnamen hierarchisch anzulegen. Dazu wird das /-Zeichen benutzt.

Hat man mehrere Branches, dann muss man den obigen Schritt für alle Branches durchführen. Das heißt, für jeden Branch muss man entscheiden, wo die Historie aufhören soll, und ein Tag history-cut/<branch-name> mit der Vorgängerinformation anlegen.

### Schritt 2: Klon anlegen

Die nachfolgenden Schritte verändern den Repository-Inhalt unwiederbringlich. Da wir das bisherige Repository als Archiv weiternutzen wollen, müssen wir einen Klon anlegen. Auch dieser wird wieder ein Bare-Repository werden, da er für den push-Befehl benutzt werden wird.

**Ein Projekt aufsetzen**  
→ Seite 123

```
> cd ..  
> git clone --bare projekt.git temp-projekt.git
```

### Schritt 3: Historie mit dem replace-Befehl verändern

Nun kann man in dem geklonten Repository die Historie entsprechend den eigenen Vorstellungen zurechtstutzen.

Der replace-Befehl ermöglicht es unter anderem, die Vorgängerbeziehungen von Commits zu manipulieren. Dabei legt Replace eine Kopie des zu verändernden Commits an, bei der die Vorgängerbeziehungen angepasst sind. Das neue Commit wird zusätzlich durch eine spezielle Referenz (.git/refs/replace/<sha-originaler-commit>) als Ersatz für das originale Commit gekennzeichnet. Alle anderen Git-Befehle verwenden nur noch das neue Commit und sehen somit die Historie nicht mehr.

In unserem Beispiel soll das c-Commit keinen Vorgänger mehr haben. Der folgende replace-Befehl erzeugt eine Kopie des Commits ohne Vorgängerbeziehungen:

```
> cd temp-projekt.git
> git replace --graft 166a7e0
```

**--graft:** Der Parameter erlaubt die Vorgängerbeziehung zu verändern.

Der folgende Parameter ist der Hashwert des Commits, das verändert werden soll. Anschließend kommen die Hashwerte der neuen Vorgänger-Commits oder nichts, wenn es keine Vorgänger geben soll.

Wenn Sie mehrere Branches bearbeiten, dann müssen Sie pro Branch den `replace`-Befehl ausführen.

**Commit-Historie zeigen** → Seite 36

Zur Kontrolle, ob die Manipulation geklappt hat, kann man den `log`-Befehl benutzen. In unserem Beispiel sollte nur noch das c-Commit auftauchen:

```
> git log --pretty=oneline
166a7e047a85b318720dc6e857a5321f9a3df7b4 C
```

#### Schritt 4: Repository permanent ändern

Nachdem das Repository mittels des `replace`-Befehls angepasst wurde, kann jetzt mit dem `filter-branch`-Befehl eine permanente neue Commit-Historie erstellt werden. Der `filter-branch`-Befehl nimmt alle Commits des angegebenen Branch und erzeugt neue Commits gemäß dem angegebenen Filter. In diesem speziellen Fall benötigt man keinen verändernden Filter, da es nur darum geht, die Commit-Historie entsprechend des `replace`-Befehls zu ändern.

Nur der Parameter `--tag-name-filter` wird benutzt, um die vorhandenen Tags an die neuen Commits zu binden.

```
> git filter-branch --tag-name-filter cat -- --all
Rewrite 166a7e047a85b318720dc6e857a5321f9a3df7b4 (2/2)
Ref 'refs/heads/master' was rewritten
Ref 'refs/tags/history-cut/master' was rewritten
WARNING: Ref 'refs/tags/release-1' is unchanged
Ref 'refs/tags/release-2' was rewritten
history-cut/master -> history-cut/master
(166a7e047a85b318720dc6e857a5321f9a3df7b4
-> 259ee224ac1f2d73898ec2ed25ad4dccc3c40f70)
release-1 -> release-1 (577b8e2cf613c43ed969453477fadcl89482c1fb
-> 577b8e2cf613c43ed969453477fadcl89482c1fb)
release-2 -> release-2 (166a7e047a85b318720dc6e857a5321f9a3df7b4
-> 259ee224ac1f2d73898ec2ed25ad4dccc3c40f70)
```

**--tag-name-filter cat:** Alle Tags werden neu angelegt und zeigen auf die neuen Commits.

**--all:** Alle Branches des Repositorys werden gefiltert.

In der Ausgabe des filter-branch-Befehls ist ersichtlich, dass das Commit c mit dem Hashwert 166a7 kopiert wurde und den neuen Hashwert 259ee zugewiesen bekommen hat.

In der Ausgabe ist auch eine »WARNING« zu sehen. Es gibt ein Tag release-1, für das es in der neuen Historie kein Commit mehr gibt. In unserem Beispiel zeigt das release-1-Tag auf das A-Commit. Dieses ist aber nach den Änderungen nicht mehr Bestandteil der Historie.

Diese Tags müssen manuell gelöscht werden, da die Tags ansonsten verhindern, dass Git die zugehörigen alten Commits endgültig löschen kann.

```
> git tag -d release-1
```

### Schritt 5: Repository verkleinern

Zu diesem Zeitpunkt ist das Repository vollständig auf die neue Historie umgebaut. Der filter-branch-Befehl löscht jedoch die alten Commits nicht, sondern referenziert diese noch unter anderen Namen. Deswegen ist das neue Repository noch nicht kleiner als das originale.

Durch ein nochmaliges Klonen kann man jedoch ein neues Repository erzeugen, das nur noch die neue Historie beinhaltet. Anschließend kann man das temporäre Repository entfernen.

```
> git clone --bare temp-projekt.git neu-projekt.git  
> rm -rf temp-projekt.git
```

Das neue Repository kann noch etwas komprimiert werden, indem der gc-Befehl verwendet wird. Der gc-Befehl erledigt verschiedene Aufräumarbeiten im Repository. Unter anderen werden neue Dateien komprimiert und alle nicht mehr referenzierten Objekte unwiederbringlich gelöscht.

```
> cd neu-projekt.git  
> git gc --prune=now
```

**--prune:** Alle nicht mehr benötigten Dateiversionen werden entfernt.

Alle Entwickler können nun darüber informiert werden, dass ein neues Repository zur Verfügung steht und geklont werden kann.

### Archiv-Repository einbinden

Wenn auf historische Informationen zugegriffen werden soll, muss das aktuelle Repository mit dem Archiv-Repository verknüpft werden. Diese Verknüpfung findet nur lokal im Entwickler-Repository statt und kann so individuell von jedem Entwickler aktiviert werden.

**Versionen markieren**

→ Seite 105

Für den folgenden Ablauf gehen wir davon aus, dass ein Entwickler bereits einen eigenen Klon (neu-projekt-Verzeichnis) des neuen Repositorys hat. In diesem Repository gibt es bereits ein neues `B`-Commit (Abbildung 25–1 unten).

### Schritt 1: Archiv-Repository klonen

Um auf die historischen Informationen zuzugreifen, benötigt man einen Klon des Archiv-Repositorys.<sup>3</sup> Da in dem Archiv-Repository keine Entwicklung stattfinden wird, reicht ein Bare-Klon aus.

```
> git clone --bare project.git archiv-projekt.git
```

### Schritt 2: Archiv-Repository einbinden

Im Entwickler-Repository müssen die Commits des Archiv-Repositorys verfügbar gemacht werden.

Damit ein Repository auf die Commits eines anderen Repositorys zugreifen kann, können sogenannte »Alternates«-Pfade spezifiziert werden. Dafür ist die `.git/objects/info/alternates`-Datei zuständig. In dieser Datei wird pro Zeile der absolute Pfad zu einem objects-Verzeichnis eines anderen Repositorys angegeben.

Achtung! Es muss unbedingt der Pfad zu dem objects-Verzeichnis angegeben werden. Der Pfad auf das Wurzelverzeichnis des Projekts reicht nicht aus.

Mit dem `echo`-Befehl wird eine neue Zeile an die `alternates`-Datei angefügt:

```
> cd neu-projekt
> echo /gitrepos/archiv-projekt.git/objects \
>> .git/objects/info/alternates
```

### Schritt 3: Historien verbinden

Als Letztes muss mit dem bereits bekannten `replace`-Befehl das `C`-Commit mit dem `B`-Commit des Archiv-Repositorys verknüpft werden.

Dabei ist das vorbereitete History-Tag sehr hilfreich. Es enthält alle notwendigen Informationen (siehe Schritt 1 auf Seite 238).

```
> git show history-cut/master --pretty=oneline
```

---

<sup>3</sup>Es wäre auch möglich, eine für alle zugängliche Kopie des Archiv-Repositorys auf ein Netzwerklaufwerk zu legen.

```
tag history-cut/master
Vorgänger: dcbddd5cd590de3d30e1ecca1882c9187e7eab95
259ee224ac1f2d73898ec2ed25ad4dccd3c40f70 C
diff --git a/foo.txt b/foo.txt
..
```

In der Ausgabe sind zwei Commit-Hashwerte zu sehen. Der erste, `dcbdd`, entspricht dem historisch korrekten Vorgänger, dem `B`-Commit. Der zweite Hashwert, `259ee`, entspricht dem neuen `C'`-Commit im aktuellen Repository.

Beim `replace`-Befehl müssen die Hashwerte in der umgekehrten Reihenfolge angegeben werden. Zuerst kommt das `C'`-Commit, dann ein Leerzeichen und dann der neue Vorgänger, das `B`-Commit.

```
> git replace --graft 259ee22 dcbddd5
```

Um den Erfolg zu testen, nutzt man am besten den `log`-Befehl. In der Ausgabe müssen jetzt auch das `A`- und das `B`-Commit erscheinen:

```
> git log --pretty=oneline
da8ba94d6bd9ec293f22a558756a91927f8b3525 D
259ee224ac1f2d73898ec2ed25ad4dccd3c40f70 C
dcbddd5cd590de3d30e1ecca1882c9187e7eab95 B
577b8e2cf613c43ed969453477fadcc189482c1fb A
```

Im aktuellen Entwickler-Repository sind jetzt alle historischen Informationen verfügbar.

Um die Historie wieder zu entfernen, reicht es aus, das Replace-Commit mit dem `replace`-Befehl zu entfernen:

```
> git replace -d 259ee22
```

**-d:** Löscht das zugehörige Replace-Commit.

## Warum nicht anders?

### Warum kein Fetch des Archiv-Repositorys?

Der beschriebene Workflow nutzt die `objects/info/alternates`-Datei, um Commits in ein Repository einzubinden. Eine Alternative ist es, den normalen `fetch`-Befehl zu nutzen, um Commits zu importieren.

Der beschriebene Workflow geht jedoch davon aus, dass der Zugriff auf die Historie nur selten und temporär erforderlich ist. In diesem Fall ist die Lösung mit der `alternates`-Datei sinnvoller, da das eigene Repository nicht durch weitere Commits vergrößert wird.

*Fetch: Branches aus  
einem anderen  
Repository holen*  
→ Seite 99

## Warum kein Shallow-Klon?

Der `clone`-Befehl unterstützt einen Parameter `--depth`, der beim Klonen nur eine bestimmte Anzahl von Vorgänger-Commits eines Branch oder aller Branches überträgt. Das Ergebnis ist ein lokaler Klon, in dem man normal arbeiten kann, bei dem allerdings ein Großteil der Historie fehlt<sup>4</sup>.

Der Nachteil dieser Lösung ist, dass jeder Entwickler diesen Parameter beim Klonen verwenden muss. Außerdem kommen einige Git-Werkzeuge nicht mit Shallow-Klonen zurecht.

Shallow-Klone sind dann sinnvoll, wenn man mal temporär einen kleinen Klon erzeugen will.

---

<sup>4</sup> Vor der Git-Version 1.9 gab es mehr Einschränkungen bei der Arbeit mit Shallow-Klonen.

## 26 <http://kapitel26.github.io>

In diesem Buch gibt es kein sechsundzwanzigstes Kapitel. Aber es gibt ein kapitel26 im Internet: unser Blog zu Git und anderen technischen Themen, die uns interessieren.

Besuchen Sie uns doch mal!



<http://kapitel26.github.io/>

Feedback und Errata



<http://kapitel26.github.io/git-buch/errata.html>



## 27 Ein Projekt nach Git migrieren

Für eine erfolgreiche Migration von einer anderen Versionsverwaltung nach Git müssen Sie mehr tun, als einige Softwarestände in ein Git-Repository zu übertragen. Dieser Workflow zeigt, wie Sie die Migration eines Projekts organisieren und was Sie dabei bedenken sollten:

- Wissensaufbau und Know-how-Transfer
- strategische Entscheidungen, die Sie treffen sollten
- Übertragung der Inhalte in ein Git-Repository
- Ablauf der eigentlichen Migration
- Nachziehen von Änderungen, die seit der Erstellung des Git-Repositorys in der alten Versionsverwaltung entstanden sind

### Überblick

Der Migrationsprozess gliedert sich in mehrere Phasen. Falls mehrere Projekte nacheinander migriert werden sollen, können später einige dieser Phasen übersprungen werden.

1. Git lernen, Erfahrungen sammeln
2. Entscheidungen treffen
3. Branches finden
4. Repository vorbereiten
5. Branches übernehmen
6. Repository in Betrieb nehmen
7. aufräumen

## Voraussetzungen

Das Projekt wird aus einer anderen Versionsverwaltung übernommen. Wir haben darüber folgende Annahmen gemacht:

**Zugriffsrechte:** Sie sollten freien schreibenden Zugriff auf alle Dateien und Verzeichnisse im Workspace haben. Insbesondere dürfen die Dateien nicht auf »Read only« stehen. Gegebenenfalls muss die Konfiguration der alten Versionsverwaltung angepasst werden.

**Ignorieren von Verzeichnissen:** Das Git-Repository wird im Workspace der anderen Versionsverwaltung angelegt. Durch ein *Ignore* des .git-Verzeichnisses in der alten Versionsverwaltung muss man es vor versehentlicher Löschung schützen können.

**Achtung!** Anders als die meisten der vorigen Workflows ist dieser sehr stark von äußeren Faktoren abhängig: Welche Versionsverwaltung wurde verwendet? Wie sind die Projekte organisiert? Wie wurden Branches genutzt? Sie werden diesen Workflow vermutlich nicht exakt so umsetzen können, wie er hier beschrieben ist. Planen Sie also etwas Zeit ein, um den Workflow an Ihre Gegebenheiten anzupassen.

Workflow kompakt

## Ein Projekt nach Git migrieren

Ein Projekt aus einer anderen Versionsverwaltung wird nach Git migriert. Alle Softwarestände, die weiterentwickelt werden sollen, werden in das Git-Repository übernommen. Danach kann mit dem neuen Repository weitergearbeitet werden. Bei Bedarf können »nachträgelnende« Änderungen aus der alten Versionsverwaltung nachgezogen werden.

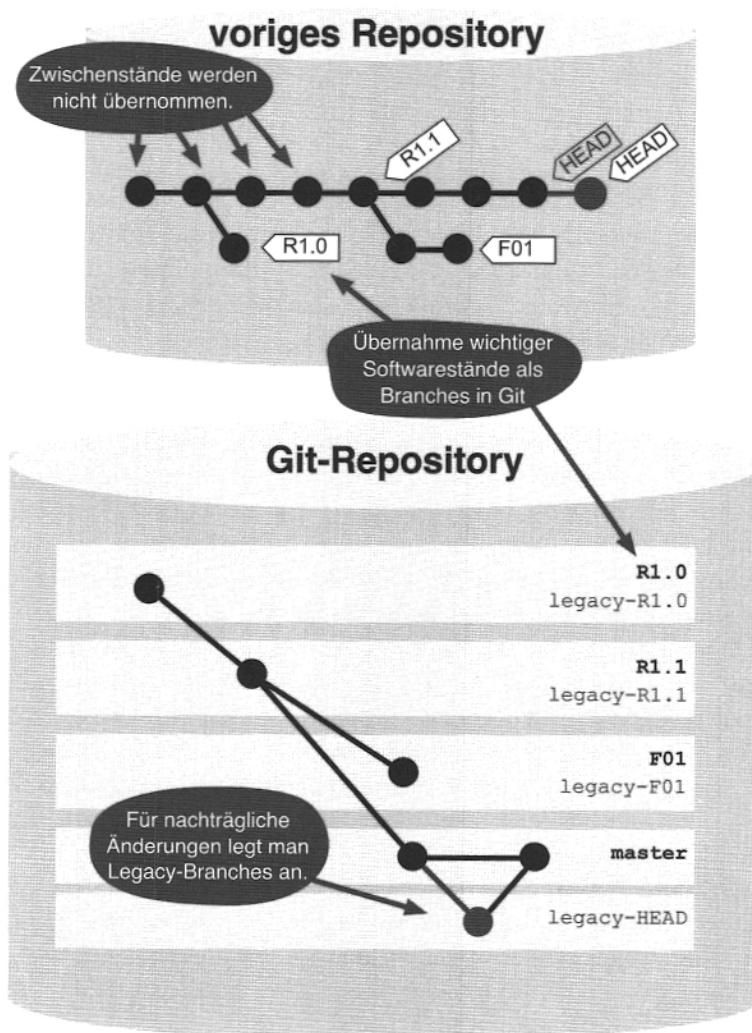


Abb. 27-1  
Workflow im Überblick

## Ablauf und Umsetzung

### Git lernen, Erfahrungen sammeln

Git ist nicht schwer zu erlernen, denn die Grundkonzepte sind logisch und gut durchdacht (davon haben wir Sie mit diesem Buch hoffentlich überzeugen können). Für Entwickler, die viel mit zentralen Versionsverwaltungen gearbeitet haben, sind einige Aspekte jedoch gewöhnungsbedürftig, wie etwa das Arbeiten mit dezentralen Repositorys oder der Umgang mit Branches. Deshalb empfehlen wir, dass sich ein oder zwei Entwickler darauf vorbereiten, das Team als Coach bei der Migration zu unterstützen.

### Schritt 1: Git erproben

#### Ein Projekt aufsetzen

→ Seite 123

Starten Sie ein kleines unkritisches Beispielprojekt. Am besten wählen Sie etwas, was noch nicht in der alten Versionsverwaltung steht. Vielleicht wollen Sie ja ohnedies gerade eine neue Utility-Klasse entwickeln, eine neue Java-Bibliothek erproben oder ein paar Shell-Skripte zur Serververwaltung schreiben.

Beginnen Sie mit der Git-Kommandozeile, auch wenn Sie später wahrscheinlich ein Git-Frontend oder ein Plug-in für Ihre Entwicklungsumgebung nutzen werden. So lernen Sie Git zunächst »ungefiltert« kennen. Die Frontends vereinfachen zwar vieles, verschleiern dabei aber oft auch, was wirklich in Git passiert. Wenn es bei der Migration zu Problemen kommt, sollte wenigstens einer im Team wissen, was hinter den Kulissen vorgeht. Entwickeln Sie zunächst einfach »drauflos«: Die elementaren Befehle `add`, `commit`, `push`, `pull` und `log` genügen.

Danach sollten Sie sich den Branches widmen, denn das ist der Bereich, wo es anfänglich zu den meisten Schwierigkeiten kommt. Probieren Sie die beiden Workflows »Gemeinsam auf einem Branch entwickeln« (Seite 135) und »Mit Feature-Banches entwickeln« (Seite 143) aus (das geht natürlich am besten, wenn mindestens zwei Entwickler dabei sind). Erzeugen Sie absichtlich ein paar Konflikte, um die Konfliktauflösung einzüben.

Dann geht es an das Frontend. Hier gibt es inzwischen eine gute Auswahl. Testen Sie jene Varianten, die zu Ihrer Entwicklungsumgebung und Zielplattform passen. Achten Sie vor allem darauf, wie gut die Merge-Konfliktauflösung unterstützt wird. In diesem Punkt unterscheiden sich die Umgebungen deutlich.

Als Coach sollten Sie mindestens die folgenden Dinge beherrschen:

- Sie haben so viel Erfahrung gesammelt, dass Sie Ihren Kollegen elementare Git-Befehle beibringen können: `add`, `commit`, `status`, `diff`, `log`, `push` und `pull`.
- Die folgenden Befehle sollten Sie sich besonders genau angesehen haben, denn sie funktionieren anders als ihre Gegenstücke in klassischen Versionsverwaltungen: `branch`, `checkout`, `reset`, `merge` und `rebase`.
- Sie haben mit mindestens einer der beiden Branching-Strategien »Gemeinsam auf einem Branch entwickeln« (Seite 135) und »Mit Feature-Banches entwickeln« (Seite 143) gearbeitet.
- Sie können Git und ggf. ein Frontend dazu auf einem Entwickler-rechner einrichten.
- Sie können Merge-Konflikte auflösen (ggf. auch über das Front-end).

## Schritt 2: Optional – parallel zur anderen Versionsverwaltung arbeiten

Sie können Git auch sehr gut lernen, indem Sie es für eine Weile parallel zur alten Versionsverwaltung nutzen, d.h., Sie entwickeln lokal mit Git, das Projekt bleibt aber in der zentralen Versionsverwaltung. Eine Anleitung dazu finden Sie in unserem Blog<sup>1</sup>.

## Entscheidungen treffen

Ein paar wichtige Entscheidungen sollten frühzeitig getroffen werden.

### Schritt 1: Alle Projekte auf einmal migrieren?

Dieser Workflow beschreibt, wie man *ein* Projekt migriert. Haben Sie mehrere Projekte, können Sie diesen Workflow natürlich mehrfach nacheinander durchführen. Sie können die Migration aber auch für alle Projekte parallel durchführen. Dies hat sowohl Vorteile als auch Nachteile im Vergleich zur Einzelprojektmigration.

#### Vorteile

- Sie können Git früher großflächig einsetzen.
- Sie profitieren früher von den Vorteilen von Git.
- Sie müssen nach der Migration nur noch ein Versionsverwaltungs-system unterstützen.

<sup>1</sup> »Andere Versionsverwaltungen parallel nutzen«

<http://kapitel26.github.io/git/2016/10/02/git-parallel-vcs>

## Nachteile

- Der Know-how-Transfer wird schwieriger. In den ersten zwei bis drei Wochen nach der Migration werden viele Fragen gestellt. Wenn es viele Entwickler gibt, aber nur einen einzigen Coach, der sich mit Git auskennt, kann das den Erfolg der Migration gefährden.
- Probleme bei der Migration können leicht zu *großen Problemen* werden, wenn viele Projekte auf einmal davon betroffen sind.
- Falls Sie in der Anfangsphase eine Entscheidung treffen, die sich später als ungünstig erweist – etwa das falsche Branching-Modell oder eine ungünstige Namenskonvention wählen –, muss dies nachher in vielen Projekten korrigiert werden.

**Unsere Empfehlung:** Wenn Sie nicht sicher sind, empfehlen wir, mit einem einzelnen Projekt zu beginnen, um danach zu entscheiden, wie Sie weiter vorgehen werden.

## Schritt 2: Welche Projekte sollen migriert werden?

Bei dieser Entscheidung können wir Ihnen nicht helfen.

## Schritt 3: Soll die bestehende Struktur übernommen werden?

Wie sind Ihre Projekte derzeit organisiert?

- Liegen alle im selben Repository? Oder haben sie separate Repositories?
- Haben die Projekte einen gemeinsamen Release-Zyklus? Sind sie sogar Teil desselben Produkts?
- Gibt es häufig projektübergreifende Änderungen?
- Sind die Projekte eher eng oder eher lose gekoppelt?

Wie die ideale Git-Repository-Struktur für Ihre Projekte aussieht, können wir leider nicht sagen. Einige grobe Faustregeln können wir aber geben:

- Wenn Sie derzeit alle Projekte in einem Repository halten, spricht dies eher dafür, das auch in Git so zu tun.
- Wenn Projekte einen gemeinsamen Release-Zyklus haben, dann spricht dies auch eher für ein gemeinsames Repository in Git, weil dann der Workflow »Periodisch Releases durchführen« (Seite 185) projektübergreifend angewendet werden kann.
- Auch häufige projektübergreifende Änderungen sind ein Indikator für ein gemeinsames Repository.

Sind die Projekte lose gekoppelt, so spricht dies eher für separate Repositories.

Git ist für Binärdateien, die groß sind und/oder häufig geändert werden, nicht gut geeignet. Sie können solche Dateien zwar mit Git verwalten, sollten sie aber besser in einem separaten Repository halten, damit die Performance in den normalen (Sourcecode-) Projekten nicht darunter leidet.

Ressourcenverbrauch  
bei großen binären  
Dateien → Seite 300

**Unsere Empfehlung:** Im Zweifel beginnen Sie einfach mit einer Struktur, die der Struktur in Ihrer bisherigen Versionsverwaltung ähnelt. Sie können dann später, wenn die Projekte migriert sind, mit den Workflows »Kleine Projekte zusammenführen« (Seite 229), »Große Projekte aufteilen« (Seite 221) und »Lange Historien auslagern« (Seite 235) eine bessere Struktur formen.

#### **Schritt 4: Können Sie sich eine Unterbrechung der Entwicklung durch die Migration erlauben?**

Falls ja, vereinfacht dies den Migrationsvorgang ein wenig. Sie stellen das neue Repository bereit und das alte ab. Die Entwickler stellen auf Git um und führen ein Release aus Git durch.

Falls Sie jedoch ein 24/7-System betreiben, wollen Sie vielleicht auch während der Migration so lange noch Hotfixes aus der alten Versionsverwaltung aufspielen können, bis die Entwickler in der Lage sind, Release-Versionen aus dem Git-Repository zu liefern. In diesem Fall müssen Sie sich damit beschäftigen, wie man Änderungen aus dem alten Repository nachziehen kann.

#### **Schritt 5: Welche Branching-Strategie wollen Sie einsetzen?**

»Gemeinsam auf einem Branch entwickeln« (Seite 135) oder »Mit Feature-Banches entwickeln« (Seite 143) – Sie sollten dies rechtzeitig entscheiden, um den Entwicklern den neuen Workflow vor der Migration vermitteln zu können, damit nachher gleich produktiv weitergearbeitet werden kann.

**Unsere Empfehlung:** Wenn Sie nicht sicher sind, beginnen Sie mit »Gemeinsam auf einem Branch entwickeln« (Seite 135), weil das der Arbeitsweise in klassischen Versionsverwaltungen ähnlicher ist.

#### **Schritt 6: Welches Frontend wollen Sie verwenden?**

Schließlich müssen Sie die Entwickler rechtzeitig vor der Migration mit der richtigen Software versorgen.

## Branches finden

Als Nächstes müssen Sie herausfinden, welche Softwarestände im alten Repository in Git als Branches weiterentwickelt werden sollen:

- Die Hauptlinie der Entwicklung, in anderen Versionsverwaltungen *Trunk* oder *Main Line* genannt, soll sicher übernommen werden.
- Jede Version, für die noch Bugfixes oder Erweiterungen geliefert werden müssen, sollte als Branch in Git übernommen werden. Wenn Sie ein Produkt entwickeln, das beim Kunden installiert wird, könnten das viele Versionen sein. Entwickeln Sie hingegen eine Webanwendung, kann es sein, dass Sie mit nur zwei Branches auskommen: einem für die produktive Version, auf dem Hotfixes gemacht werden, und einem für die Feature-Entwicklung zum nächsten Release.
- Wenn Sie in Ihrer bisherigen Versionsverwaltung mit Feature-Branches arbeiten, sollten Sie diese entweder in Git übernehmen oder kurz vor der Migration fertigstellen und schließen. Letzteres macht die Migration natürlich leichter.

*Tipp: »Floating Tags« zu Branches*

In vielen Versionsverwaltungen gibt es sogenannte Floating Tags, d. h. Tags wie RELEASE3, die nach Hotfixes verschoben werden können. Solche Tags sind oft Kandidaten für Release-Branches in Git.

*Tipp: Zeitpunkt clever wählen*

Je weniger Branches Sie übernehmen müssen, desto weniger Arbeit macht die Migration. Überlegen Sie also gut, was Sie wirklich benötigen und auch, wann Sie die Migration durchführen. Vielleicht gibt es einen Zeitpunkt, zu dem nur wenige Branches aus der alten Versionsverwaltung übernommen werden müssen.

*Tipp: Graph der Beziehungen zeichnen*

Anschließend zeichnen Sie einen Graphen mit den »Verwandtschaftsbeziehungen« zwischen den gefundenen Branches. Im einfachsten Fall ist dies eine Sequenz mit der ältesten Release-Version unten und der neuesten oben.

## Repository vorbereiten

Als Nächstes wird das Git-Repository erstellt. Damit Sie es in aller Ruhe einrichten und testen können, sollten Sie damit ein paar Tage (oder Wochen) vor der eigentlichen Migration beginnen. Das bedeutet aber, dass in der Zwischenzeit neue Änderungen im alten Repository entstehen. Es wird also später notwendig werden, diese Änderungen nachzuziehen.

Um das zu erreichen, arbeitet man auf der Git-Seite mit zwei Branches, von denen der eine die Entwicklung in Git darstellt und der andere die Entwicklung im alten Repository. Letzteren nennen wir im Folgenden *Legacy-Branch*. Auf dem *Legacy-Branch* wird nicht entwickelt,

dort werden nur Softwarestände aus dem alten Repository übernommen. Übertragen werden diese Änderungen dann später durch einen Merge auf dem Entwicklungs-Branch.

Wenn das Konzept des Legacy-Branch Sie irgendwie an *Remote-Tracking-Banches* erinnert, dann haben Sie gerade ein Muster wiedererkannt. In beiden Fällen geht es darum, im lokalen Repository Vorgänge aus einem anderen Repository widerzuspiegeln.

In diesem Beispiel verwenden wir folgende Namenskonvention: Für Branches oder Tags der alten Versionsverwaltung nehmen wir Großbuchstaben, z. B. `RELEASE3`. In Git verwenden wir Kleinbuchstaben und nennen den Entwicklungs-Branch dazu dann `release3` und den Legacy-Branch dazu `legacy-release3`.

### Schritt 1: Projekt aus der alten Versionsverwaltung holen

Ein Workspace mit den Dateien des Projekts wird aus der alten Versionsverwaltung geholt. In anderen Versionsverwaltungen nennt man dies oft einen Checkout.<sup>2</sup>

### Schritt 2: Git-Repository anlegen

In diesem Workspace aus der alten Versionsverwaltung wird jetzt ein Git-Repository angelegt. Es entsteht ein Workspace, der mit beiden Versionsverwaltungen verknüpft ist. Wir nennen dies einen *Dual Workspace*.

```
> cd old-vcs-workspace  
> git init
```

### Schritt 3: Lokales Backup erstellen

Wenn man mit zwei verschiedenen Versionsverwaltungen gleichzeitig arbeitet, kann es durchaus passieren, dass man mal ein `force` oder `clean` an der falschen Stelle angibt. Deshalb ist ein Backup keine schlechte Idee:<sup>3</sup>

```
> git clone --no-hardlinks --bare .  
/backups/myproject.git  
> git remote add backup /backups/myproject.git
```

---

<sup>2</sup>Bei dem Begriff *Checkout* herrscht leider Begriffsverwirrung. In Git bedeutet er etwas ganz anderes, nämlich den Wechsel von einem Branch zum anderen (Seite 61).

<sup>3</sup>Mindestens einem der Autoren ist das mindestens einmal passiert.

Später sollte man gelegentlich sichern:

```
> git push --all backup
```

Zum Wiederherstellen klont man das Repository in ein temporäres Verzeichnis, wechselt dann auf den gewünschten Git-Branch und verschiebt das .git-Verzeichnis in den Workspace der alten Versionsverwaltung.

#### Schritt 4: Metadateien ignorieren lassen

Zunächst muss dafür gesorgt werden, dass sich die beiden Versionsverwaltungen nicht gegenseitig den Workspace zerschießen.

Die Metadateien der alten Versionsverwaltung sollen nicht ins Git-Repository übernommen werden. Erstellen Sie dazu eine .gitignore-Datei. Dort tragen Sie die Pfade oder Dateimuster ein, die ignoriert werden sollen. Der status-Befehl darf danach keine Metadateien der alten Versionsverwaltung mehr anzeigen.

```
> git commit .gitignore -m "ignore legacy metafiles"
```

Umgekehrt muss auch die alte Versionsverwaltung so konfiguriert werden, dass das .git-Verzeichnis und die .gitignore-Datei erhalten bleiben. In CVS kann man dies beispielsweise durch das Erstellen einer .cvsignore-Datei im User-Verzeichnis tun.

### Branches übernehmen

Die zu übernehmenden Tags und Branches der alten Versionsverwaltung werden Schritt für Schritt abgearbeitet. Man beginnt mit dem ältesten Branch oder Tag, das übernommen werden soll.

#### Schritt 1: Gegebenenfalls auf Vorgänger-Branch wechseln

Beim ersten Branch können Sie diesen Schritt überspringen, weil es noch keinen Vorgänger-Branch gibt.

Im Überblicksgraphen (vgl. Seite 254) können Sie sehen, welches der Vorgänger ist. Wenn Sie RELEASE3 migrieren wollen, dann wechseln Sie jetzt auf den Vorgänger, also auf den Legacy-Branch für RELEASE2:

```
> git checkout legacy-release2
```

#### Schritt 2: Legacy-Branch anlegen

Man legt einen Legacy-Branch an, der den Stand des Tags/Branch, z. B. RELEASE3, aus dem alten Repository widerspiegeln soll:

```
> git branch legacy-release3
```

**Mit .gitignore  
Dateien unversioniert  
lassen** → Seite 46

### Schritt 3: Stand aus der alten Versionsverwaltung übernehmen

Jetzt wechseln wir in der alten Versionsverwaltung auf den Softwarestand, den wir übernehmen wollen, z. B. RELEASE3:

```
> git status
```

Der status-Befehl zeigt jetzt, welche Änderungen es von RELEASE2 auf RELEASE3 gegeben hat. Sie sollten kurz prüfen, ob es plausibel aussieht. Falls ja, werden die Änderungen in den neuen Legacy-Branch übernommen:

```
> git add --all  
> git commit -m "RELEASE3 aus legacy-vcs geholt"
```

### Schritt 4: Generierte Dateien unversioniert lassen

Der jetzige Softwarestand wird gebaut und getestet. Dabei entstehen wahrscheinlich neue Dateien, die nicht in das Repository übernommen werden sollen. Die .gitignore-Datei muss ergänzt werden:

```
> git commit .gitignore -m "ignore build artifacts"
```

*Mit .gitignore  
Dateien unversioniert  
lassen → Seite 46*

### Schritt 5: Git-Branch anlegen

Jetzt wird der Branch angelegt, auf dem später in Git weiterentwickelt werden soll:

```
> git branch release3
```

### Schritt 6: Ergebnis prüfen

Es empfiehlt sich, das Ergebnis noch einmal zu überprüfen. Damit die Metadateien der Versionsverwaltungen dabei nicht stören, wird der Vergleich in temporären Verzeichnissen außerhalb der Workspaces durchgeführt. Hierbei hilft der archive-Befehl, der den Dateibaum eines beliebigen Commits als Archivdatei (tar oder zip) exportiert. Der aktuelle Stand des Branch in Git, z. B. in release3, wird in ein temporäres Verzeichnis git-vcs geschrieben:

```
> git archive release3 | tar -x -C /tmp/git-vcs/
```

Dann exportiert man den Stand, z. B. RELEASE3, aus der alten Versionsverwaltung, zum Beispiel nach /tmp/legacy-vcs. Jetzt kann man den Vergleich durchführen, zum Beispiel mit kdiff3. Bis auf die .gitignore-Datei sollte es keine Unterschiede geben.

```
> kdiff3 /tmp/git-vcs/ /tmp/legacy-vcs
```

## Repository in Betrieb nehmen

Unser Ziel ist es, einen möglichst reibungsfreien Übergang zu schaffen.

### Schritt 1: Ankündigung

Kündigen Sie die Migration rechtzeitig an. Die Ankündigung sollte folgende Informationen enthalten:

**Einführung:** Laden Sie zu einem Termin ein, bei dem das normale Arbeiten mit Git gezeigt wird.

**Einrichtung der Entwicklungsumgebung:** Beschreiben Sie kurz, wie man die Entwicklungsumgebung einrichtet (Git und IDE-Plug-ins installieren und konfigurieren) und wie man sich ein Projekt klonen kann.

**Freeze-Zeitpunkt:** Fordern Sie die Mitarbeiter auf, bis zu einem genannten Zeitpunkt alle lokalen Änderungen in die alte Versionsverwaltung zu bringen und ab dann keine neuen Änderungen einzupflegen.

**Continue-Zeitpunkt:** Ab wann kann mit dem neuen Git-Repository weitergearbeitet werden?

**Notfallplan:** Hotfix-Releases können auch während der Umstellungsphase aus der alten Versionsverwaltung heraus durchgeführt werden. Die Änderungen müssen dann aber später in Git nachgezogen werden. Es ist wichtig, deutlich darauf hinzuweisen, dass das dann auch unbedingt gemacht werden muss. Andernfalls könnte ein bereits behobener Fehler beim Git-Release erneut ausgeliefert werden.

### Schritt 2: Einführung

Nun zeigen Sie, wie man mit Git arbeitet. Im normalen Alltag benötigt man nur wenige Befehle. Sie können sich zum Beispiel am Workflow »Gemeinsam auf einem Branch entwickeln« (Seite 135) orientieren. Zum Vorführen kann man einfach einen Klon des neuen Repositorys verwenden. Man kann damit nach Herzenslust herumexperimentieren und später den Klon einfach wegwerfen.

### Schritt 3: Letzte Änderungen abholen

Nach dem Freeze-Zeitpunkt müssen Sie alle Änderungen aus dem alten Repository nachziehen, die seit der Erstellung des Git-Repositories vorgenommen wurden. Dies erfolgt im *Dual Workspace* für jeden *Legacy-Branch*. Zuerst wechselt man in Git auf den *Legacy-Branch*:

```
> git checkout legacy-release3
```

Danach wechselt man in der alten Versionsverwaltung auf den entsprechenden Branch/Tag, z. B. RELEASE3, und prüft, ob sich Änderungen ergeben haben:

```
> git status
```

Ist das der Fall, werden diese in den Legacy-Branch übernommen:

```
> git add -all  
> git commit -m "updating legacy-release3 from old vcs"
```

Danach werden die Änderungen in den neuen Branch für die Weiterentwicklung in Git, z. B. release3, übernommen:

```
> git checkout  
> git merge legacy-release3
```

Wenn noch keine Weiterentwicklung in Git stattgefunden hat, wird es hier keinen Merge-Konflikt geben.

Auf diese Weise können Änderungen übrigens auch dann noch nachgezogen werden, wenn die Weiterentwicklung in Git längst begonnen hat, zum Beispiel wenn ein Entwickler den Freeze-Zeitpunkt verschlafen hat oder weil ein eiliges Hotfix noch in der alten Versionsverwaltung durchgeführt werden musste. In so einem Fall kann es allerdings Merge-Konflikte geben, die man manuell auflösen muss.

*Tipp: Änderungen können auch später noch nachgezogen werden.*

#### Schritt 4: Neues Repository bereitstellen

Nachdem alle Branches nachgezogen wurden, kann man das Repository auf dem Server ablegen. Dann gibt man die URL dafür bekannt und fordert die Entwickler auf, das Repository zu klonen und mit der Entwicklung fortzufahren (Continue-Zeitpunkt).

#### Schritt 5: Produkt bauen bzw. Release durchführen

Jetzt geht es darum, sobald wie möglich ein Release durchzuführen bzw. die aktuelle Version Ihres Produkts zu bauen, um sicherzugehen, dass Sie jetzt ohne die alte Versionsverwaltung auskommen können.

#### Schritt 6: Altes Repository auf »Read only« schalten

Sobald man in der Lage ist, aus dem neuen Repository heraus Releases durchzuführen (bzw. Produkte zu bauen), sollte man das alte Repository in einen »Read only«-Modus versetzen. Es dient dann nur noch als Archiv für die Historie des Projekts.

### Schritt 7: Entwickler unterstützen

Vergessen Sie nicht, etwas Zeit einzuplanen, um die Entwickler während der ersten Wochen zu unterstützen. Insbesondere sollten Sie darauf vorbereitet sein, Merge-Konflikte aufzulösen und lokale Änderungen rückgängig machen zu müssen, z. B. mit dem `reset`-Befehl oder durch *interaktives Rebasing*.

### Aufräumen

Nachdem das alte Repository abgeklemmt ist, kann man die *Legacy-Banches* löschen. Am besten tut man dies in einem frisch geklonten Workspace und nicht im *Dual Workspace*, weil dort `origin` nicht verknüpft ist.

```
> git branch -d legacy-release3  
> git push origin :legacy-release3
```

## Warum nicht anders?

### Warum nicht die ganze Historie übernehmen?

In diesem Workflow werden nur einzelne Softwarestände übernommen, die man weiterentwickeln möchte. Dies hat den Nachteil, dass man im neuen Git-Repository die alte Historie nicht sieht. Sie verbleibt im alten Repository.

Es gibt verschiedene Werkzeuge (in Git, aber auch aus eigenständigen Projekten), die eine Historienübernahme grundsätzlich ermöglichen. Zum Beispiel kann der `cvsimport`-Befehl CVS-Repository-Inhalte in ein Git-Repository übertragen. Da aber die Struktur im Repository von CVS eine ganz andere ist als in Git, ist die Übersetzung nicht trivial, und die Qualität des Ergebnisses kann variieren, je nachdem, in welcher Weise CVS vorher genutzt wurde. Auf jeden Fall sollte man sich das Importergebnis sehr genau ansehen, bevor man damit weiterarbeitet. Eventuell muss man nacharbeiten, damit es passt.

Es ist zum einen der damit verbundene Aufwand, der uns davon abgehalten hat, diesen Weg zu gehen. Zum anderen wollten wir einen Migrationsweg zeigen, der gangbar ist, egal von welcher Versionsverwaltung man kommt.

## Könnte man auf die *Legacy-Banches* verzichten?

Im Workflow werden anfangs sogenannte *Legacy-Banches* erstellt, die den jeweiligen Stand von Branches und Tags in der alten Versionsverwaltung widerspiegeln. Am Ende des Workflows werden sie gelöscht. Sie dienen nur einem einzigen Zweck: nämlich dem Abholen von nachträglichen Änderungen aus dem alten Repository. Wenn Sie die Entwicklung für ein paar Tage unterbrechen können (zum Beispiel wenn das Team eine Schulung besucht oder an etwas anderem arbeiten kann), dann können Sie durchaus auf *Legacy-Banches* verzichten und den Workflow für sich ein wenig vereinfachen.

## Kann man auf den *Dual Workspace* verzichten?

Im *Dual Workspace* kann man mit Git und der alten Versionsverwaltung gleichzeitig arbeiten. Das erleichtert das Austauschen von Softwareständen: Man wechselt in der alten Versionsverwaltung auf den gewünschten Stand und macht dann ein Commit in Git.

Ein *Dual Workspace* ist aber nicht mit jeder anderen Versionsverwaltung möglich. In solchen Fällen könnte man mit zwei separaten Workspaces arbeiten. Dann müsste man Änderungen hin- und herübertragen, z. B. mit Shell-Skripten oder mit rsync. Das ist möglich, aber deutlich aufwendiger.



# 28 Integration mit Jenkins

Viele Projekte nutzen einen Build-Server wie »Jenkins«<sup>1</sup>, »JetBrains Teamcity«<sup>2</sup> oder »Bamboo«<sup>3</sup>, um regelmäßig automatische Builds durchzuführen. Git kann dabei natürlich als Quelle für die zu bauenden Softwarestände dienen.

Nachfolgend zeigen wir am Beispiel von Jenkins die Konfiguration und die Integrationsmöglichkeiten. Wir haben uns für Jenkins als Beispielserver entschieden, da

- dieser in sehr vielen Projekten benutzt wird,
- als Open-Source-Server von Ihnen sofort verwendet werden kann und
- die Autoren sich damit am besten auskennen.

## 28.1 Vorbereitungen

Jenkins kann auf verschiedenen Plattformen installiert werden. Die genauen Voraussetzungen und Schritte finden Sie auf der Jenkins-Webseite. Nachfolgend gehen wir von einem installierten Jenkins<sup>4</sup> mit deutscher Oberfläche unter folgender URL aus:

<http://myjenkins:8080/>

Damit Jenkins mit Git-Repositorys arbeiten kann, müssen zwei zusätzliche Plug-ins vorhanden sein: das »Git Client Plug-In«<sup>5</sup> und das »Git Plug-In«<sup>6</sup>. Diese können unter JENKINS VERWALTEN / PLUG-INS VERWALTEN / VERFÜGBAR installiert werden. Zusätzlich muss auf dem System, auf dem Jenkins ausgeführt wird, noch Git installiert werden. Un-

---

<sup>1</sup> <http://jenkins-ci.org/>

<sup>2</sup> <http://www.jetbrains.com/teamcity/>

<sup>3</sup> <https://www.atlassian.com/software/bamboo/>

<sup>4</sup> getestet mit Version 2.26

<sup>5</sup> getestet mit Version 2.0.0

<sup>6</sup> getestet mit Version 3.0.0

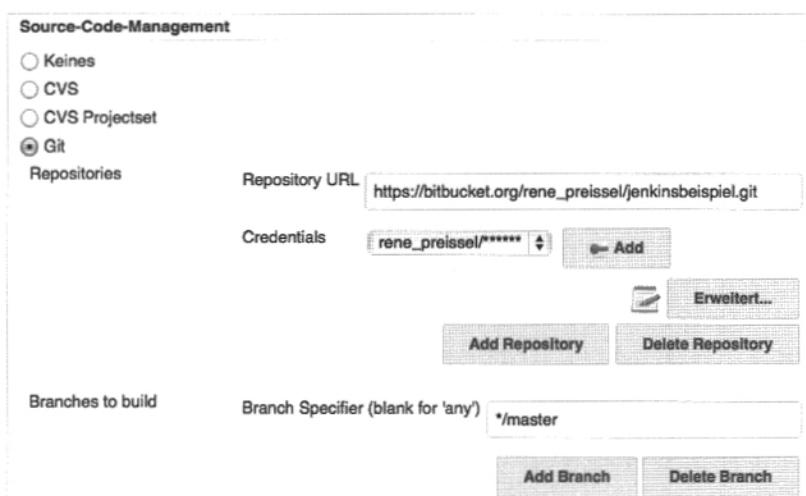
ter JENKINS VERWALTEN / SYSTEM KONFIGURIEREN muss man dann im Abschnitt GIT den Pfad zur Git-Installation angeben.

## 28.2 Ein einfaches Git-Projekt einrichten

Nachdem die Git-Plug-ins installiert sind, ist das Einrichten eines Jenkins-Jobs für ein Git-Repository sehr einfach. Erzeugen Sie als Erstes einen neuen Jenkins-Job mit ELEMENT ANLEGEN und wählen Sie z. B. »FREE STYLE«-SOFTWARE-PROJEKT BAUEN aus.

Im Abschnitt SOURCE-CODE-MANAGEMENT muss GIT ausgewählt werden. Dadurch erscheint der entsprechende Git-Abschnitt für die Konfiguration (Abbildung 28-1).

*Abb. 28-1  
Jenkins-Konfiguration  
des Repositorys*



Tragen Sie im Feld REPOSITORY URL eine gültige Git-URL ein. Im Feld CREDENTIALS können Sie den Benutzernamen und das Passwort auswählen bzw. über den Knopf ADD neu hinzufügen.

Im Abschnitt BRANCHES TO BUILD können Sie festlegen, welche Branches durch den Job gebaut werden. Als Standard ist nur der master-Branch ausgewählt. Sie können in dem Feld allerdings auch mit Wildcards arbeiten oder über den Knopf ADD BRANCH weitere Branches hinzufügen.

Als Letztes muss noch der Build-Auslöser festgelegt werden. Die einfachste Variante besteht darin, Git regelmäßig abzufragen und bei Änderungen in den angegebenen Branches einen Build auszulösen. Dazu konfigurieren Sie im Abschnitt BUILD-AUSLÖSER den Eintrag SOURCE CODE MANAGEMENT SYSTEM ABFRAGEN. Im Feld ZEITPLAN definieren Sie nun das Intervall, in dem geprüft wird, ob Änderungen

vorliegen, z. B. bedeutet H/15 \* \* \* alle 15 Minuten, wobei durch Angabe des Buchstabens H der eigentliche Startzeitpunkt zufällig festgelegt wird (Abbildung 28–2).

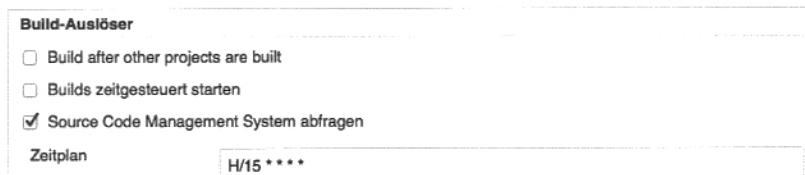


Abb. 28–2  
Build-Auslöser konfigurieren

## Beispiel-Repository

Wir haben auf GitHub ein Beispiel-Repository für Sie bereitgestellt, das die nachfolgenden Jenkins-Beispiele automatisch anlegt: <https://github.com/kapitel26/jenkins-templates>

Dazu nutzt es das Job DSL PLUG-IN<sup>a</sup>. Dieses Plug-in muss genauso wie die anderen Plug-ins installiert werden. Erzeugen Sie so, wie in diesem Abschnitt beschrieben wird, einen einfachen Jenkins-Job und konfigurieren Sie die Git-URL auf:

<https://github.com/kapitel26/jenkins-templates.git>

Sie benötigen keinen Build-Trigger, da der Job nur manuell gestartet wird. Über die Auswahlbox BUILD-SCHRITT HINZUFÜGEN wählen Sie den Eintrag PROCESS JOB-DSLs aus. Daraufhin erscheint der entsprechende Abschnitt. Tragen Sie in dem Feld DSL SCRIPTS den Wert `*.groovy` ein.

Wenn Sie nun den neuen Jenkins-Job starten, wird das Beispiel-Repository geklont und alle darin enthaltenen Job-Groovy-Skripte werden ausgeführt. Das wiederum führt dazu, dass alle Beispiel-Jenkins-Projekte angelegt werden.

<sup>a</sup> getestet mit Version 1.52

## 28.3 Hook als Build-Auslöser

Im vorigen Abschnitt haben Sie gesehen, wie man einen einfachen Jenkins-Job einrichtet und diesen zeitgesteuert startet. Das regelmäßige Pollen von Repositorys kann bei vielen Jenkins-Jobs und kleinem Intervall unnötige Last auf dem System und im Netzwerk erzeugen. Außerdem startet der Build oft verzögert, wenn das Build-Intervall groß eingestellt ist.

**Hooks – Git erweitern**

→ Seite 294

Mit der Kombination von Hooks und dem »Git Plug-In« ist es möglich, den Build direkt durch den push-Befehl auszulösen.

Hooks erlauben es, eigene Skripte vor oder nach Git-Befehlen auszuführen. Dabei ist es möglich, Hooks sowohl im lokalen Repository als auch im Remote-Repository zu konfigurieren. Für das Auslösen des Jenkins-Builds eignet sich der »Post-Receive-Hook« im Remote-Repository. Er wird immer dann aufgerufen, wenn neue Daten durch ein Push übertragen wurden.

Der zweite Bestandteil dieses Ablaufs ist ein Feature des Git-Plug-ins. Es reagiert auf Abrufe der folgenden URL, deren Parameter <repo-url> der Repository-URL eines oder mehrerer Jenkins-Jobs entsprechen muss:

```
http://myjenkins:8080/git/notifyCommit?url=<repo-url>
```

Wird die Notify-URL abgerufen, löst das Plug-in bei diesen Jenkins-Jobs eine Überprüfung aus. Sie holen dann den letzten Stand des Repositorys (per fetch-Befehl) und stoßen einen Build an, falls Änderungen vorliegen.

Nachdem die technischen Details bekannt sind, zeigen wir als Nächstes die konkrete Umsetzung des Hooks und die Einrichtung des Jenkins-Jobs.

Der »Post-Receive-Hook« wird als Shell-Skript mit den Namen `post-receive` im Verzeichnis `<remote-repo>/hooks` abgelegt. Die einfachste Form, die auf Änderungen aller Branches reagiert, ist hier zu sehen:

```
#!/bin/sh
curl http://myjenkins:8080/git/notifyCommit\
?url=<repo-url>\
> /dev/null 2>&1
```

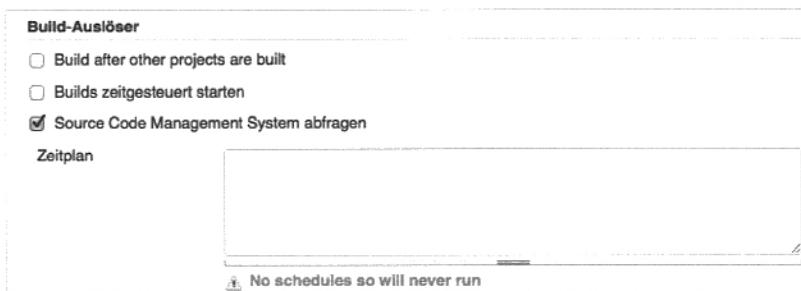
Die Variable <repo-url> ist durch die konkrete Repository-URL zu ersetzen. Immer wenn der Hook ausgeführt wird, ruft der `curl`-Befehl die Notify-URL am Jenkins-Server auf. Mit `> /dev/null 2>&1` werden alle Ausgaben des `curl`-Befehls unterdrückt. In den zugehörigen Jenkins-Jobs muss im Abschnitt **BUILD-AUSLÖSER** der Eintrag **SOURCE CODE MANAGEMENT SYSTEM ABFRAGEN** aktiviert sein. Das Feld **ZEITPLAN** muss leer bleiben (Abbildung 28–3).<sup>7</sup>

Im Beispiel-Repository gibt es in der Datei `notifier-post-receive-hook` noch eine erweiterte Variante, die nur auf einen bestimmten Branch reagiert:

```
https://github.com/kapitel26/jenkins-templates
```

---

<sup>7</sup> Die Warnung **NO SCHEDULES SO WILL NEVER RUN** kann ignoriert werden.



**Abb. 28–3**  
Zeitplan für  
Build-Auslöser leer  
lassen

## 28.4 Ein Tag für jeden erfolgreichen Build

In den vorangegangenen Abschnitten haben wir Jenkins mit Git kombiniert, allerdings werden bisher keine Informationen von Jenkins nach Git übertragen. Es kann aber sinnvoll sein, jeden erfolgreichen Build mit einem Tag, z. B. `build/<build-number>`, zu versehen. Eine weitere Möglichkeit ist, einen Branch, z. B. `build/latest`, zu pflegen, der immer auf den letzten erfolgreichen Build zeigt. Es ist dann möglich, jederzeit den Unterschied zwischen dem aktuellen HEAD und dem letzten erfolgreichen Build zu ermitteln.

Mit dem »Git Plug-In« ist das alles nur eine Frage der Konfiguration. Dazu muss man im Abschnitt POST-BUILD-AKTIONEN den GIT PUBLISHER-Schritt hinzufügen. Über die Knöpfe ADD TAG und ADD BRANCH können dann das Tag und der Branch konfiguriert werden (Abbildung 28–4).

Aktivieren Sie das Feld PUSH ONLY IF BUILD SUCCEEDS, damit Tag und Branch nur bei erfolgreichen Builds in das Remote-Repository übertragen werden.

Für das Build-Tag wollen wir die Jenkins-Build-Nummer als eigentlichen Tag-Namen wählen. Dafür stellt Jenkins die Variable `$BUILD_NUMBER` bereit. Damit die Build-Tags und der Build-Branch gut von anderen Tags und Branches zu unterscheiden sind, versehen wir diese mit einem Präfix: `build`. Im Feld TAG TO PUSH muss der vollständige Tag-Name konfiguriert werden. Das Feld CREATE NEW TAG muss aktiviert werden, da ansonsten das Tag nicht in das Remote-Repository zurückgeschrieben werden kann. Im Feld TARGET REMOTE NAME muss der Remote-Name, normalerweise `origin`, konfiguriert werden (Abbildung 28–4 oben).

Der Build-Branch kann genauso konfiguriert werden. In unserem Beispiel soll dieser den Namen `build/latest` bekommen. Dieser Name ist im Feld BRANCH TO PUSH einzutragen. Im Feld TARGET REMOTE NAME muss wie beim Tag der Wert `origin` konfiguriert werden (Abbildung 28–4 unten).

**Abb. 28–4**  
Konfiguration von Tags  
und Branches für  
erfolgreiche Builds

**Post-Build-Aktionen**

**Git Publisher**

Push Only If Build Succeeds

Merge Results   
If pre-build merging is configured, push the result back to the origin

Force Push   
Add force option to git push

**Tags**

Tag to push

Tag message

Create new tag

Update new tag

Target remote name

**Delete Tag**

**Add Tag**

Tags to push to remote repositories

**Branches**

Branch to push

Target remote name

## 28.5 Pull-Requests bauen

Wenn man Pull-Requests einsetzt, um Feature-Banches zu integrieren, dann will man meistens vor dem Merge überprüfen, ob der Build funktionieren würde.

Für die Arbeit mit Pull-Requests wird eine zentrale Repository-Verwaltung benötigt, die dieses Feature explizit unterstützt, z. B. Atlassian Bitbucket Server<sup>8</sup>, GitHub<sup>9</sup> oder GitLab<sup>10</sup>. Diese ermöglichen das Anlegen, Reviewen und Abschließen von Pull-Requests über eine Weboberfläche.

Dabei werden für jeden Pull-Request zwei speziell benannte Branches angelegt: ein Branch, der auf das aktuellste Commit des Feature-Branch zeigt, und zusätzlich ein Branch, der auf das temporäre Merge-Commit mit dem Ziel-Branch zeigt. Die Benennung der beiden Branches ist abhängig von der eingesetzten Repository-Verwaltung. Beim Bitbucket Server würde der erste Branch `refs/pull-requests/<id>/from` und der zweite Branch `refs/pull-requests/<id>/merge` heißen<sup>11</sup>.

Man kann sich jetzt entscheiden, welche Version man bauen möchte: entweder nur den aktuellsten Commit des Feature-Branch oder das Merge-Commit. Nach dem Build kann man das Ergebnis an die Repository-Verwaltung melden. Dadurch wird die Build-Meldung im Pull-Request angezeigt.

Die genaue Integration mit Jenkins hängt dabei von der benutzten Repository-Verwaltung ab. Typischerweise braucht man:

- einen Trigger innerhalb der zentralen Repository-Verwaltung, um das Build nach dem Anlegen oder Ändern eines Pull-Requests zu starten,
- einen Job im Build-Server, um das Pull-Request-Build zu bauen, ggf. gibt es dafür auch spezielle Plug-Ins,
- ein Plug-In im Build-Server, um die zentrale Repository-Verwaltung über das Ergebnis des Build zu informieren.

Nachfolgend zeigen wir als Beispiel die Integration mit dem Atlassian Bitbucket Server.

---

<sup>8</sup> [www.atlassian.com/software/bitbucket/server](http://www.atlassian.com/software/bitbucket/server)

<sup>9</sup> <https://github.com>

<sup>10</sup> <https://gitlab.com/>

<sup>11</sup> Wenn der Merge zu einem Konflikt führt, dann gibt es nur den ersten Branch, und in der Weboberfläche wird eine entsprechende Fehlermeldung stehen.

## Zusammenspiel mit Atlassian Bitbucket Server

Beim Atlassian Bitbucket Server gibt es die oben beschriebene Namenskonvention für die Pull-Request-Banches: `refs/pull-requests/*`. Dabei wird der `*` durch eine fortlaufende Pull-Request-Nummer ersetzt. Da die Branches nicht mit `refs/head/` beginnen, holt der normale `fetch`-Befehl diese nicht automatisch in das lokale Repository. Deswegen muss man in der Jenkins-Konfiguration unter **ERWEITERT** noch eine explizite *Refspec* angeben (Abbildung 28–5).

**Abb. 28–5**  
Konfiguration der  
Refspec für den  
Pull-Request

The screenshot shows the Jenkins job configuration interface. In the 'Refspec' section, the 'Repository URL' is set to `http://localhost:7990/scm/git/jenkinsexample.git`. The 'Credentials' dropdown is empty ('- leer -'). The 'Name' field is set to 'origin'. The 'Refspec' field contains the value `+refs/pull-requests/*:refs/remotes/origin/pull-requests/*`.

Sie können das auch in Ihrem eigenen lokalen Repository ausprobieren, wenn Sie zum Beispiel vorab den Pull-Request eines Kollegen anschauen wollen:

```
> git fetch origin \
    +refs/pull-requests/*:refs/remotes/origin/pull-requests/*
```

Der Atlassian Bitbucket Server zeigt das Build-Ergebnis im Pull-Request nur an, wenn es mit dem letzten Commit des Feature-Branch (`refs/pull-requests/*/from`) verknüpft ist<sup>12</sup>. Deswegen kann man den `refs/pull-requests/*/merge`-Branch nicht einfach bauen.

Möchte man aber wissen, ob der Feature-Branch nach dem Merge fehlerfrei bauen würde, kann man in Jenkins unter **ADDITIONAL BEHAVIOURS** angeben, dass vor dem Build einen Merge mit dem Ziel-Branch durchzuführen ist. Das heißt, man gibt den `refs/pull-requests/*/from`-Branch in der Job-Konfiguration an und spezifiziert zusätzlich, dass vor dem Build ein Merge mit dem `master`-Branch stattfinden soll (Abbildung 28–6).

<sup>12</sup> Es gab für ältere Versionen des Atlassian Bitbucket Server das Stash-Build-Status-Plug-In, welches die Build-Ergebnisse vom temporären Merge-Commit auf das aktuellste Commit des Feature-Branch kopiert hat. Dieses Plug-in ist allerdings zum aktuellen Zeitpunkt nicht mehr verfügbar. Deswegen kann man nicht einfach den `refs/pull-requests/*/merge` bauen.

Branch Specifier (blank for 'any')	origin/pull-requests/*/from
<b>Merge before build</b>	
Name of repository	origin
Branch to merge to	master
Merge strategy	default
Fast-forward mode	--no-ff

**Abb. 28-6**  
Konfiguration des Merge für den Pull-Requests

Den Jenkins-Job kann man zeitlich ansteuern. Wie das geht, ist im Abschnitt »Ein einfaches Git-Projekt einrichten« ab Seite 264 beschrieben. Alternativ ist das Auslösen über einen Hook möglich. Dafür gibt es in Atlassian Bitbucket Server ein eigenes Plug-in: WEBHOOK TO JENKINS FOR BITBUCKET<sup>13</sup>.

Damit der Jenkins-Job das Ergebnis des Build wiederum an Atlassian Bitbucket Server zurückmeldet, muss ein weiteres Plug-In *Stash Notifier Plug-In*<sup>14</sup> installiert werden. Im Jenkins-Job muss als POST-BUILD-AKTION die Aktion Notify Stash Instance hinzugefügt und konfiguriert werden (Abbildung 28-7).

<b>Post-Build-Aktionen</b>	
<b>Notify Stash Instance</b>	
Stash base URL	http://localhost:7990
Credentials	jenkins/*********

**Abb. 28-7**  
Konfiguration des Stash Notifiers für Pull-Requests

Sobald nun ein neuer Pull-Request angelegt oder ein vorhandener geändert wird, startet der Bitbucket-Server den Jenkins-Job. Der Job führt als Erstes einen Merge des Pull-Requests-Branch mit dem master-Branch durch. Das Merge-Ergebnis wird gebaut. Nach dem Bauen informiert das Stash-Notifier-Plug-In den Bitbucket-Server über das Build-Ergebnis.

<sup>13</sup> <http://marketplace.atlassian.com/Plug-Ins>

<sup>14</sup> getestet mit Version 1.11.4

## 28.6 Automatischer Merge von Branches

Im Workflow »Periodisch Releases durchführen« (Seite 185) haben wir gezeigt, wie man verschiedene Branches nutzt, um ein Release vorzubereiten. Eine wichtige Aufgabe dabei ist es, die Änderungen von einem Branch auf einen anderen Branch zu übertragen. Das GIT PLUG-IN von Jenkins kann dies automatisch. Es erstellt vor dem Bauen ein Merge-Commit und bringt dieses nach erfolgreichem Build auf den Ziel-Branch.

Im Folgenden wird die Jenkins-Konfiguration für ein Merge vom master-Branch auf den develop-Branch gezeigt.

Als Erstes wird ein normaler Job mit einem Repository konfiguriert. Als BRANCH TO BUILD wird hierbei der master ausgewählt. Das heißt, immer, wenn auf dem master neue Commits auftauchen, soll der Build ausgeführt werden (Abbildung 28-8).

**Abb. 28-8**

Konfiguration des Repository und des Branch

Repository URL	<input type="text" value="https://bitbucket.org/rene_preissel/jenkinsbeispiel.git"/>
Credentials	<input type="text" value="rene_preissel/*****"/> <input type="button" value="Add"/>
Branch Specifier (blank for 'any')	<input type="text" value="master"/>

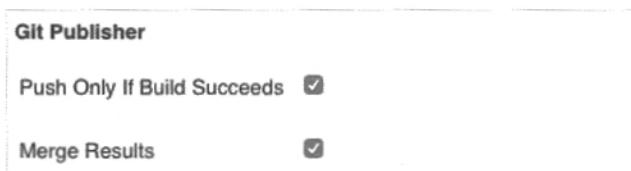
Das Entscheidende ist, dass zusätzlich im Bereich ADDITIONAL BEHAVIOURS das Element MERGE BEFORE BUILD ausgewählt wird (Abbildung 28-9). Im Feld BRANCH TO MERGE TO muss der Ziel-Branch (in unserem Fall develop) angegeben werden. Will man sichergehen, dass es keinen Fast-Forward-Merge gibt, muss im Feld FAST-FORWARD MODE noch --no-ff ausgewählt werden.

**Abb. 28-9**

Merge vor dem Build konfigurieren

<b>Merge before build</b>	
Name of repository	origin
Branch to merge to	develop
Merge strategy	default
Fast-forward mode	--no-ff

Als Letztes muss als POST-BUILD-AKTION der GIT PUBLISHER konfiguriert werden. Dabei sind die Felder PUSH ONLY IF BUILD SUCCEEDS und MERGE RESULTS zu aktivieren (Abbildung 28–10). Dies sorgt dafür, dass nach einem erfolgreichen Build das Merge-Commit in den Ziel-Branch, in unserem Fall in den develop-Branch, zurückgeschrieben wird.



**Abb. 28–10**  
Erfolgreichen Merge  
zurückschreiben

Die vorangegangenen Abschnitte sollten Ihnen nur einen Überblick über die Integration von Git und Jenkins geben. Das »Git Plug-In« kann noch einiges mehr. Sehen Sie sich die entsprechende Dokumentation an.



# 29 Abhangigkeiten zwischen Repositorys

In Git ist das Repository die Release-Einheit, d.h., Versionen, Branches und Tags konnen nur auf dem gesamten Repository angelegt werden. Besteht ein Projekt aus Subprojekten mit jeweils eigenem Release-Zyklus und somit eigenen Versionen, muss es auch fur jedes Subprojekt ein Repository geben.

Die Beziehungen zwischen dem Gesamtprojekt und den Subprojekten konnen in Git mit dem `submodule`-Befehl oder mit dem `subtree`-Befehl<sup>1</sup> umgesetzt werden.

Der Hauptunterschied zwischen dem Submodule- und dem Subtree-Konzept ist, dass bei Submodulen im Gesamt-Repository nur Verweise auf die Modul-Repositories eingebunden sind, wahrend beim Subtree die Inhalte der Modul-Repositories in das Gesamt-Repository importiert werden.

## 29.1 Abhangigkeiten mit Submodulen

Mit *Submodulen* konnen in einem Git-Repository (Gesamt-Repository) andere Repositorys (Modul-Repositories) eingebunden werden. Dazu wird im Gesamt-Repository ein Verzeichnis mit einem Commit eines Modul-Repository verknüpft.

In Abbildung 29-1 ist die Grundstruktur dargestellt. Es gibt zwei Repositorys: `main` und `sub`. Im Gesamt-Repository wurde das `sub`-Verzeichnis mit dem Modul-Repository verknüpft. Im Workspace des Gesamt-Repository liegt unter dem `sub`-Verzeichnis ein vollstandiges Modul-Repository. Im eigentlichen Gesamt-Repository wird jedoch nur auf das Modul-Repository verwiesen. Dazu gibt es die Datei

**Repositorys konnen nur vollstandig verwendet werden**

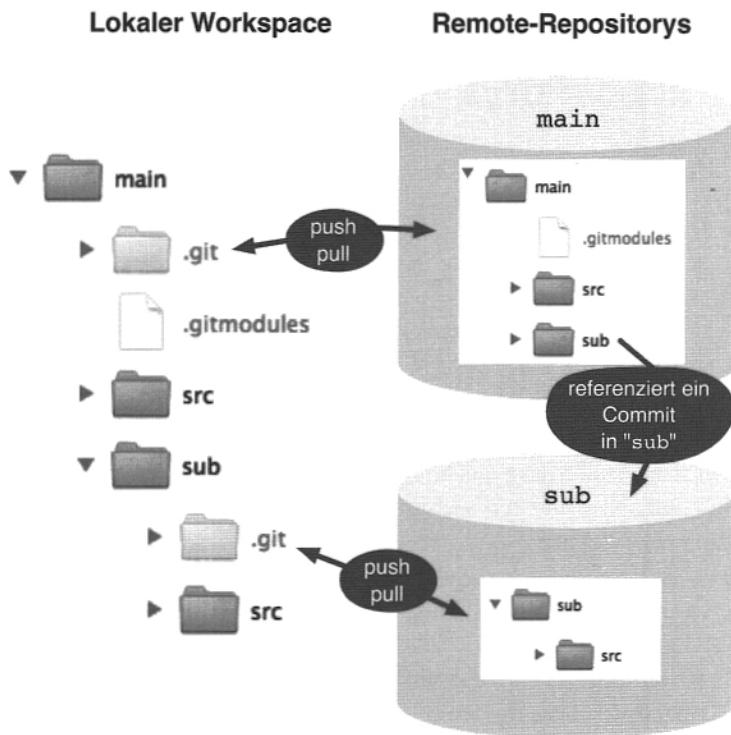
→ Seite 300

**Groe Projekte aufteilen** → Seite 221

---

<sup>1</sup>Der `Subtree`-Befehl ist seit Version 1.7.11 offiziell in der Git-Distribution vorhanden. Allerdings ist er auch nur als optionaler Bestandteil im `contrib`-Verzeichnis zu finden. Einige Installationspakete liefern den `Subtree`-Befehl bereits automatisch mit, bei anderen ist das manuelle Nachinstallieren notwendig.

**Abb. 29-1**  
Submodule:  
Grundlagen



.gitmodules, in der für jedes verknüpfte Verzeichnis die URL zu dem Modul-Repository definiert ist.

```
[submodule "sub"]
path = sub
url = <sub-repo-url>
```

Außer in der .gitmodules-Datei werden die Verweise auf die Submodule noch in der .git/config-Datei abgelegt. Das passiert mit dem submodule init-Befehl, der die .gitmodules-Datei liest und die Informationen in die .git/config-Datei schreibt. Diese indirekte Konfiguration erlaubt es, die Pfade zu den Modul-Repositorys lokal in der .git/config-Datei anzupassen.

```
[submodule "sub"]
url = <sub-repo-url>
```

Mit den bisherigen Informationen wäre es nicht möglich, für jedes Commit des Gesamt-Repository den zugehörigen Zustand des Modul-Repository zu reproduzieren. Dazu wird noch das Commit des Modul-Repository benötigt. Dieses wird am Verzeichnis-Objekt (Tree) im Gesamt-Repository abgelegt. Nachfolgend ist ein Tree-Objekt dargestellt. Der dritte Eintrag, sub, ist ein Submodul, was man am commit-

Typ erkennt. Der nachfolgende Hashwert referenziert das Commit im Modul-Repository.

```
100644 blob 1e2b1d1d51392717a479eaaaa79c82df1c35d442 .gitmodules  
100644 tree 19102815663d23f8b75a47e7a01965dc96468c src  
160000 commit 7fa7e1c1bd6c920ba71bd791f35969425d28b91b sub
```

Schritt für Schritt

## Submodule einbinden

*Ein bereits vorhandenes Git-Projekt wird als Submodul in ein anderes Projekt eingebunden.*

### 1. Mit Verzeichnis verbinden

Um ein Submodul einzubinden, muss man dieses mit dem submodule add-Befehl anlegen und mit einem Verzeichnis verknüpfen. Dabei müssen die URL zu dem Modul-Repository und der Verzeichnisname angegeben werden:

```
> git submodule add <sub-repo-url> sub
```

Als Ergebnis wird das Modul-Repository vollständig geklont und die Workspace-Dateien werden in das angegebene Verzeichnis abgelegt<sup>2</sup>. Zusätzlich wird die .gitmodules-Datei im Gesamt-Repository angelegt bzw. erweitert.

### 2. Submodul-Version auswählen

Der Workspace des Modul-Repository zeigt initial auf den HEAD des Standard-Branch. Um ein anderes Commit des Submoduls zu benutzen, verwendet man den checkout-Befehl und wählt die gewünschte Version:

```
> cd sub  
> git checkout v1.0
```

### 3. .gitmodules-Datei und Subverzeichnis zum Commit hinzufügen

Durch das Hinzufügen eines Submoduls wird die Datei .gitmodules angelegt bzw. verändert und muss zum Commit hinzugefügt werden. Auch das neue Verzeichnis des Submoduls muss hinzugefügt werden:

```
> cd ..  
> git add .gitmodules  
> git add sub
```

<sup>2</sup>Das .git-Verzeichnis des Modul-Repository wird unterhalb des .git-Verzeichnisses des Gesamt-Repository abgelegt.

#### 4. Commit durchführen

Als Letztes muss man im Gesamt-Repository ein Commit durchführen:

```
> git commit -m "Submodul hinzugefügt"
```

Wird ein Repository geklont, das Submodule enthält, muss der submodule init-Befehl ausgeführt werden. Dieser überträgt die URLs der Submodule in die .git/config-Datei. Anschließend werden mit dem submodule update-Befehl die Verzeichnisse der Modul-Repositories geklont. Alternativ kann auch der clone-Befehl mit der Option --recursive aufgerufen werden, der die einzelnen Schritte zusammenfasst:

```
> git clone --recursive main.git
```

Fügt man nachträglich ein Submodul zu einem Repository hinzu, müssen alle bisherigen Klone mit dem submodule update-Befehl und den Optionen --init --recursive aktualisiert werden. Dadurch wird das neue Submodul geklont und auf das richtige Commit initialisiert:

```
> git submodule update --init --recursive
```

Mit dem submodule status-Befehl kann man sich die referenzierten Commit-Hashwerte der Submodule ansehen. Dabei steht das referenzierte Tag, falls vorhanden, am Ende der Ausgabe in Klammern:

```
> git submodule status
```

```
091559ec65c0ded42556714c3e6936c3b1a90422 sub (v1.0)
```

Git referenziert bei Submodulen immer genau ein Commit aus dem Modul-Repository. Dessen Commit-Hashwert ist auch Bestandteil jedes Commits des Gesamt-Repository. Daraus ergibt sich, dass neue Commits im Modul-Repository nicht automatisch im Gesamt-Repository eingespielt werden. Dieses Verhalten ist explizit erwünscht, damit beim Wiederherstellen einer Projektversion des Gesamt-Repository auch immer die dazu passende Projektversion des Modul-Repository benutzt wird.

Will man im Gesamt-Repository eine neue Version des Modul-Repository benutzen, muss man diese explizit ändern.

Schritt für Schritt

## Neue Version eines Submoduls verwenden

*Es soll eine andere Version eines bereits eingebundenen Submoduls verwendet werden.*

### 1. Submodul aktualisieren

Als Erstes bringt man den lokalen Workspace des Submoduls auf den gewünschten Stand. Typischerweise beginnt man mit einem `fetch`-Befehl, um die neuesten Commits des Modul-Repository zu erhalten:

```
> cd sub  
> git fetch
```

Als Nächstes legt man mit dem `checkout`-Befehl das gewünschte Commit fest:

```
> git checkout v2.0
```

### 2. Neue Version verwenden

Als Nächstes bereitet man ein neues Commit mit dem Submodul-Verzeichnis vor und führt das Commit aus:

```
> cd ..  
> git add sub  
> git commit -m "Neue Version des Submoduls"
```

Wenn man im Workspace gleichzeitig im Gesamt-Repository und im Modul-Repository arbeitet, dann muss man sowohl die Änderungen des Modul-Repository als auch die Änderungen des Gesamt-Repository durch ein Commit abschließen. Hat man ein zentrales Repository, müssen auch beide Repositorys separat durch den `push`-Befehl übertragen werden.

Schritt für Schritt

## Mit Submodulen arbeiten

*In einem Workspace werden Änderungen an Dateien des Gesamt-Repository und an Dateien des Modul-Repository durchgeführt. Das Gesamt-Repository soll anschließend auf das neue Commit des Modul-Repository zeigen.*

### 1. Änderungen des Modul-Repository abschließen und übertragen

Als Erstes werden die Änderungen des Modul-Repository durch einen `commit`-Befehl abgeschlossen und ggf. mit dem `push`-Befehl in das zentrale Repository übertragen:

```
> cd sub  
> git add foo.txt  
> git commit -m "Submodul-Änderungen"  
> git push
```

### 2. Änderungen des Gesamt-Repository abschließen und übertragen

Als Nächstes werden die Änderungen des Gesamt-Repository inklusive des Verweises auf das neue Commit des Modul-Repository festgeschrieben und ggf. übertragen:

```
> cd ..  
> git add bar.txt  
> git add sub  
> git commit -m "Neue Version des Submoduls"
```

Nach jeder Aktualisierung eines Workspace, der Submodule enthält, sollte der `submodule update`-Befehl benutzt werden, um die richtigen Versionen der Submodule zu holen.

Wenn ein ganz neues Submodul hinzugekommen ist, muss man bei dem `submodule update`-Befehl noch die Option `--init` angeben.

Ganz sicher gehen Sie als Entwickler, wenn Sie nach jeder Aktualisierung des Workspace (Checkout, Merge, Rebase, Reset, Pull) den `submodule update`-Befehl mit den Optionen `init` und `recursive` ausführen.

**Submodule aktualisieren**  
→ Seite 281

Schritt für Schritt

## Submodule aktualisieren

*Wenn eine neue Version eines Submoduls von einem anderen Entwickler eingespielt wurde, dann muss diese explizit im eigenen lokalen Klon und Workspace aktualisiert werden.*

Dazu dient der submodule update-Befehl:

```
> git submodule update --init --recursive
From /projekte/sub
 091559e..4722848 master      -> origin/master
 * [new tag]      v1.0        -> v1.0
 * [new tag]      v2.0        -> v2.0
Submodule path 'sub':
checked out '472284843ce4c0b0bb503bc4921ab7...1e51'
```

## 29.2 Abhängigkeiten mit Subtrees

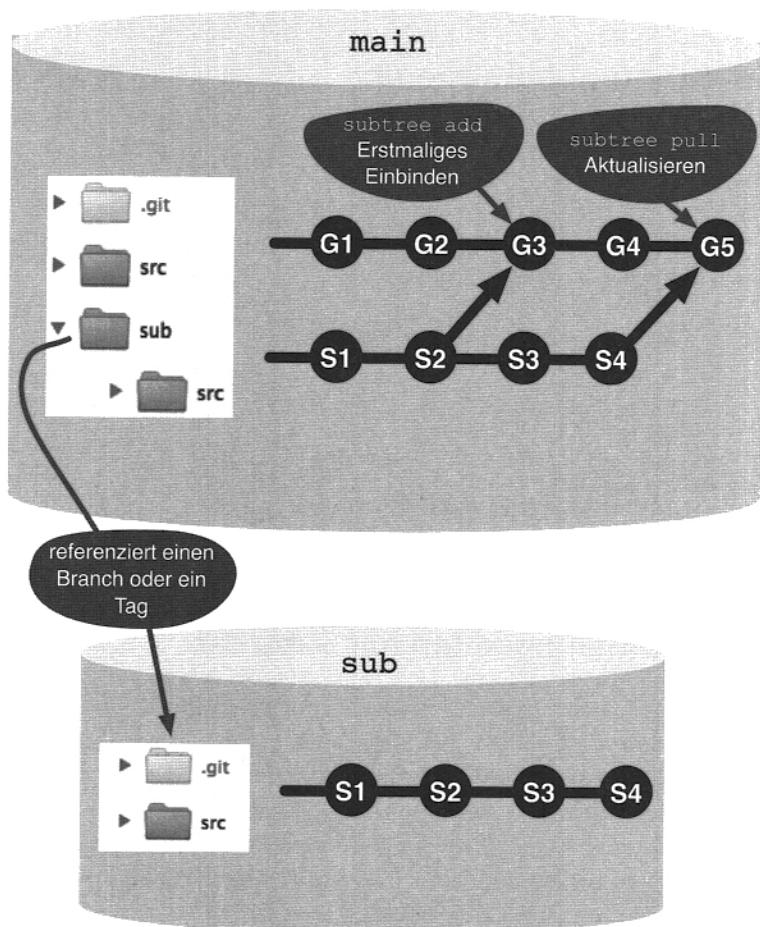
Mit Subtrees können in einem Git-Repository (Gesamt-Repository) andere Repositorys (Modul-Repositorys) eingebunden werden. Dazu wird im *Gesamt-Repository* ein Verzeichnis mit einem Commit/Tag/Branch eines Modul-Repository verknüpft. Anders als bei Submodulen wird jedoch im Gesamt-Repository der gesamte relevante Inhalt des Modul-Repository importiert und nicht nur referenziert. Das erlaubt das autarke Arbeiten mit dem Gesamt-Repository.

In Abbildung 29–2 ist die Grundstruktur dargestellt. Es gibt zwei Repositorys: main und sub. Im Gesamt-Repository wurde das sub-Verzeichnis mit dem Modul-Repository verknüpft (subtree add-Befehl). Im Gesamt-Repository liegen unter dem sub-Verzeichnis die Dateien einer Version des Modul-Repository.

Technisch holt der subtree add-Befehl alle Commits des Modul-Repository in das Gesamt-Repository (siehe die Commits s1 und s2). Anschließend wird der aktuelle Branch des Gesamt-Repository mit dem spezifizierten Commit des Modul-Repository vereinigt (siehe Merge-Commit g3). Dabei wird intern die Subtree-Merge-Strategie (--strategy=subtree) verwendet. Diese führt einen Merge in einem definierten Verzeichnis durch, sodass der Inhalt des Modul-Repository unter dem sub-Verzeichnis landet.

Abb. 29-2

Subtree: Grundlagen



Schritt für Schritt

## Subtree einbinden

*Ein bereits vorhandenes Git-Projekt wird als Subtree in ein anderes Projekt eingebunden.*

Um ein Modul-Repository einzubinden, muss man dieses im Gesamt-Repository mit dem subtree add-Befehl einmalig hinzufügen. Dabei ist das Unterverzeichnis als --prefix anzugeben. Zusätzlich müssen die URL<sup>3</sup> des Modul-Repository und das gewünschte Tag bzw. der gewünschte Branch angegeben werden:

```
> git subtree add \
    --prefix=sub <sub-repo-url> v2.0
```

Ist die Historie des Modul-Repository im Kontext des Gesamt-Repository nicht relevant, kann mit der Option --squash nur der Inhalt des gewünschten Commit geholt werden:

```
> git subtree add --squash --prefix=sub <sub-repo-url> master
```

Als Ergebnis wird ein neues Merge-Commit angelegt, das im Commit-Kommentar die Hash-ID des geholten Commit vermerkt hat, um beim nächsten Update wieder das richtige Commit im Modul-Repository zu finden.

Anders als bei Submodulen muss beim Klonen eines Gesamt-Repository mit Subtrees nichts Spezielles beachtet werden. Der normale clone-Befehl holt das Gesamt-Repository und alle enthaltenen Modul-Repositorys:

```
> git clone <main-repo-url>
```

---

<sup>3</sup> Natürlich ist es auch möglich, ein Remote zu konfigurieren. Dazu müssen Sie den remote add-Befehl und den Remote-Namen verwenden.

Schritt für Schritt

## Neue Version eines Subtree verwenden

*Es soll eine andere Version eines bereits eingebundenen Subtree verwendet werden.*

Der subtree `pull`-Befehl aktualisiert einen eingebundenen Subtree. Dabei sind die gleichen Parameter wie beim subtree `add`-Befehl zu übergeben. Falls beim Add ein Tag übergeben wurde, muss ein neueres Tag verwendet werden. Falls ein Branch benutzt wurde, kann der gleiche oder ein anderer Branch spezifiziert werden. Falls es keine Änderungen auf dem Branch gab, wird der subtree `pull`-Befehl auch nichts tun.

```
> git subtree pull --prefix=sub <sub-repo-url> v2.1
```

Auch beim Pull kann mit der Option `--squash` auf die Historie des Modul-Repository verzichtet werden. Dann werden keine Zwischen-Commits geholt, sondern nur das spezifizierte Commit. Bei der Verwendung von `--squash` kann auch auf ältere Versionen des Modul-Repository zurückgegangen werden, z. B. von v2.0 auf v1.5.

```
> git subtree pull --squash  
--prefix=sub <sub-repo-url> master
```

Auch bei Subtrees ist es möglich, Änderungen in den eingebundenen Modulverzeichnissen vorzunehmen. Dabei ist nichts Besonderes zu beachten. Es wird einfach der normale `commit`-Befehl verwendet. Es können in einem Commit gleichzeitig Änderungen des Gesamt-Repository und eines oder mehrerer Modulverzeichnisse versioniert werden.

Erst das Zurückübertragen der Moduländerungen in das jeweilige Repository verlangt spezielle Vorkehrungen.

Schritt für Schritt

## Änderungen in das Modul-Repository übertragen

*Änderungen in Modulverzeichnissen sollen in das Modul-Repository übertragen werden.*

### 1. Änderungen im Modulverzeichnis separieren

Als Erstes werden die Änderungen des Modulverzeichnisses von den anderen Änderungen mit dem subtree `split`-Befehl getrennt. Dabei erzeugt der subtree `split`-Befehl für jedes Commit, in dem eine Moduldatei geändert wurde, ein neues Commit auf Basis des letzten bekannten Modul-Repository-Commit. Das Ergebnis ist ein lokaler Branch, der auf die neuen Commits zeigt (z. B. `sub/master`). Falls Sie ohne Squash beim subtree `add`-Befehl und subtree `pull`-Befehl arbeiten, verwenden Sie den Parameter `--rejoin`. Dieser vereinfacht den wiederholten Aufruf von Split:

```
> git subtree split --rejoin  
      --prefix sub --branch sub/master
```

### 2. Änderungen mit Modul-Repository vereinigen

Die lokalen Änderungen müssen mit den entfernten Änderungen des Modul-Repository zusammengeführt werden. Dazu ist als Erstes der neu angelegte Branch zu aktivieren und dann die aktuellste Version des Ziel-Branch zu holen.<sup>4</sup> Anschließend müssen beide Branches vereinigt werden:

```
> git checkout sub/master  
> git fetch <sub-repo-url> master  
> git merge FETCH_HEAD
```

---

<sup>4</sup>Das Fetch mit URL erzeugt eine temporäre Referenz `FETCH_HEAD`, die auf das aktuellste Commit des geholten Branch zeigt. Wenn man mit Remotes arbeitet, kann natürlich auch der Remote-Name anstelle der URL verwendet werden. Außerdem ist dann der Ziel-Branch direkt vorhanden und nicht nur der `FETCH_HEAD`.

### 3. Änderungen an ein Modul-Repository übertragen und den temporären Branch löschen

Die lokalen Änderungen auf dem temporären Branch müssen in das entfernte Modul-Repository übertragen werden (`push`-Befehl).<sup>5</sup> Anschließend kann wieder auf den Branch des Gesamt-Repository gewechselt und der temporäre Branch gelöscht werden:

```
> git push <sub-repo-url> HEAD:master
> git checkout master
> git branch -d sub/master
```

Wie an den obigen Ausführungen gut zu erkennen ist, sind die meisten Operationen mit Subtrees einfacher als mit Submodulen. Nur das Extrahieren der Änderungen ist ähnlich komplex.

In vielen Szenarien wird das Extrahieren aber gar nicht benötigt, da im Gesamt-Repository nicht in den Modulverzeichnissen gearbeitet wird. Diese dienen nur dazu, die aktuellste Version des Modul-Repository einzubinden.

## 29.3 Zusammenfassung

**Submodule einbinden:** Mit dem `submodule add`-Befehl werden Submodule eingebunden.

**Projekt mit Submodulen klonen:** Nach dem Klonen den `submodule init`-Befehl und den `submodule update`-Befehl benutzen. Alternativ kann man beim Klonen die `--recursive` benutzen.

**Neue Version des Submoduls wählen:** Erst im Submodul-Verzeichnis das neue Commit wählen (`checkout`-Befehl) und dann im Gesamt-Repository durch ein Commit festschreiben.

**Modul-Repository und Gesamt-Repository gleichzeitig bearbeiten:** Es muss erst das Commit im Modul-Repository und dann das Commit im Gesamt-Repository durchgeführt werden. Beide Repositorys müssen mit dem `push`-Befehl übertragen werden.

**Subtrees einbinden:** Mit dem `subtree add`-Befehl werden Subtrees eingebunden.

**Neue Version des Subtree wählen:** Der `subtree pull`-Befehl aktualisiert das Modulverzeichnis auf den gewünschten Branch oder das gewünschte Tag.

---

<sup>5</sup> Auch hier kann der Remote-Name verwendet werden. Im Beispiel wird das letzte Commit des aktuellen Branch (HEAD) auf den entfernten Master-Branch übertragen.

**Änderungen im Modulverzeichnis extrahieren:** Der `subtree split`-Befehl erstellt einen separaten Branch mit den Änderungen an dem Modulverzeichnis. Dieser kann anschließend durch den `merge`-Befehl mit den anderen Änderungen zusammengeführt und mit dem `push`-Befehl übertragen werden.



# 30 Was gibt es sonst noch?

Wir haben uns in diesem Buch auf die Git-Konzepte und -Befehle beschränkt, die in typischen Projektsituationen in Unternehmen benutzt werden. Das folgende Kapitel soll Ihnen einen Überblick geben, was es sonst noch für Möglichkeiten in Git gibt. Die Befehle werden nicht ausführlich beschrieben, sondern nur so weit, wie es für das Verständnis notwendig ist.

## 30.1 Worktrees – mehrere Workspaces mit einem Repository

Normalerweise gehört zu einem Git-Repository genau ein Workspace, bzw. zu einem Workspace genau ein Git-Repository. Da Git sehr schnell beim Wechseln von Branches ist, kann man trotzdem bequem zwischen verschiedenen Entwicklungszenarien umschalten. Das Ändern der Dateien im Workspace ist aber oft nur ein Teil des Branch-Wechsels. Häufig muss danach wieder ein vollständiges Build mit den neuen Quelldateien durchgeführt werden. Dabei hängt es von der jeweiligen Programmiersprache und dem Umfang der Quellen ab, wie schnell das geht. Im ungünstigsten Fall kann das mehrere Stunden dauern.

Die Alternative zum Branch-Wechsel im Workspace ist, für jeden Branch einen eigenen Workspace zu benutzen. Also für alle wichtigen Branches legt man einen Klon inklusive Workspace an. Der Kontextwechsel besteht nun darin, das Arbeitsverzeichnis zu wechseln.

Ein Problem mit dieser Lösung ist, dass es auch zwei Repositorys gibt. Jedes Repository benötigt Platz, und jedes Repository muss separat aktualisiert (fetch-Befehl) werden.

In dem Szenario hilft der `worktree`-Befehl. Mit dem ist es möglich, zusätzliche Workspaces zu einem Repository anzulegen. Der folgende Befehl legt einen neuen Workspace `myproject-release1` parallel zum aktuellen Workspace an und aktiviert den `release1`-Branch:

```
> git worktree add ../myproject-release1 release1
```

Dieser neue Workspace hat kein eigenes Repository (.git-Verzeichnis), sondern holt sich die Daten aus dem Repository des Ursprungs-Workspace.

Git unterbindet, dass es zwei Workspaces mit demselben aktivierten Branch gibt.

## 30.2 Interaktives Rebasing – Historie verschönern

Sie haben an verschiedenen Stellen dieses Buches Rebasing kennengelernt. Es dient dazu, die Änderungen von Commits nochmals anzuwenden und neue Commits zu erzeugen, z. B. wenn man einen Branch umpflanzen will.

*Mit Rebasing die Historie glätten*

→ Seite 81

Wenn man sehr viel Wert auf die Commit-Historie legt, dann kann man das Rebasing auch dazu benutzen, um Commits zusammenzufassen (squash), aufzuspalten (edit) oder neu zu sortieren. Dazu wird der rebase-Befehl mit dem Parameter --interactive aufgerufen. Als Parameter wird das Commit angegeben, ab dem die Historie verändert werden soll. Um zum Beispiel die letzten drei Commits zu verändern, wäre folgender Befehl abzusetzen:

```
> git rebase --interactive HEAD~3
```

Als Ergebnis zeigt ein Texteditor drei Commits:

```
pick 927d33a commit 3
pick 7d343d0 commit 4
pick fbe58cb commit 5

# Rebase 940d0db..fbe58cb onto 940d0db
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

In dieser Textdatei können die Commits neu sortiert oder mit den aufgeführten Befehle verändert werden. Nach dem Schließen des Editors wird Git die Commits entsprechend den Befehlen verarbeiten.

**Achtung!** Die Commit-Historie sollte niemals nach dem Ausführen des push-Befehls verändert werden. Andere Teammitglieder könnten schon von den »alten« Commits abhängen.

### 30.3 Umgang mit Patches

Insbesondere in der Unix-Welt werden Änderungen häufig per Patch-Datei übertragen. In reinen Git-Umgebungen besteht selten die Notwendigkeit, direkt mit Patches zu arbeiten. Hier werden Änderungen per Commit ausgetauscht. Falls es doch mal notwendig ist, unterstützt Git das Erzeugen und Einspielen von Patches.

Patches können mit dem diff-Befehl erzeugt werden:

```
> git diff rel-1.0.0 HEAD >local.patch
```

Zum Einspielen der Änderungen in ein anderes Repository gibt es den apply-Befehl:

```
> git apply local.patch  
> git commit -m "applied patch"
```

### 30.4 Archive erstellen

Mit dem archive-Befehl kann man den Projektinhalt eines beliebigen Commits als Archiv ohne Git-Metadaten (.git-Verzeichnis) exportieren. Dabei werden das tar- und das zip-Format unterstützt.

```
> git archive HEAD --format=tar > archive.tar  
> git archive HEAD --format=zip > archive.zip  
> git archive HEAD --format=tar | gzip > archive.tar.gz
```

Es ist auch möglich, nur einzelne Unterverzeichnisse in ein Archiv zu packen:

```
> git archive HEAD subdir --format=tar > archive.tar
```

Mithilfe des Parameters --remote können auch Archive von entfernten Repositorys angelegt werden.

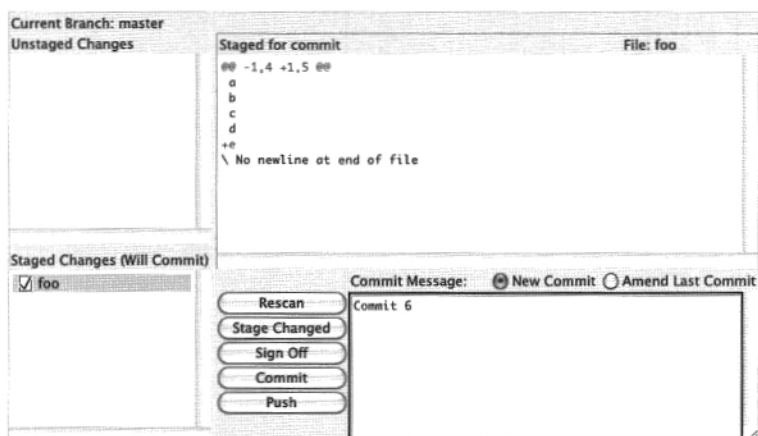
## 30.5 Grafische Werkzeuge für Git

Wir haben in diesem Buch hauptsächlich mit der Kommandozeile gearbeitet. Doch Git bringt auch schon zwei grafische Werkzeuge mit.

Das ist zum einen das Git-GUI (gui-Befehl): ein grafisches Werkzeug, um Commits zu erstellen (Abbildung 30-1).

```
> git gui
```

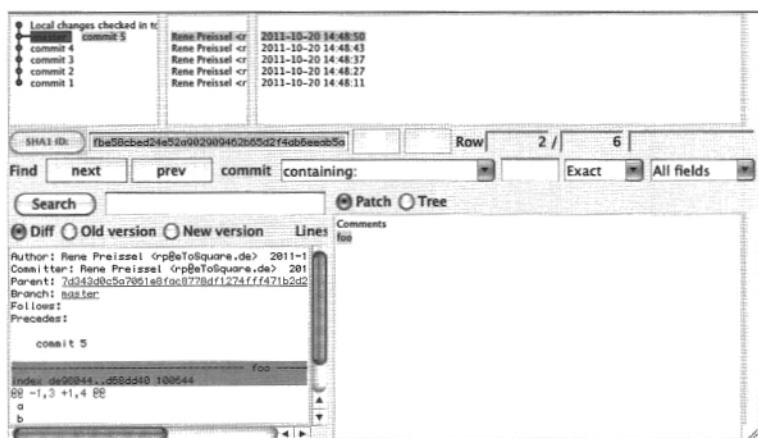
**Abb. 30-1**  
Grafisches Werkzeug,  
um Commits zu  
erzeugen



Zum anderen gibt es noch das grafische Werkzeug GitK (gitk-Befehl), mit dem die Historie betrachtet werden kann (Abbildung 30-2):

```
> git gitk --all
```

**Abb. 30-2**  
Grafisches Werkzeug,  
um die Historie zu  
betrachten



Die Git-Entwickler haben sich sehr viel Mühe gegeben, die Oberflächen mit deutschen Begriffen zu versehen. Wenn es Ihnen allerdings wie uns geht und Ihnen »Diese Version pflücken« für »Cherry-pick this commit« ungewohnt erscheint, dann können Sie auch die englischen Namen wählen. Das geht leider nicht per Optionen, sondern indem man die Übersetzungsdateien löscht. Diese finden Sie unter <GIT-HOME>/share/gitk/lib/msgs/de.msg und unter <GIT-HOME>/share/git-gui/lib/msgs/de.msg. Alternativ können Sie auch in der Git-Shell die Sprache auf Englisch setzen. Diese Einstellung wird von beiden Werkzeugen respektiert:

```
> export LANG=en
```

## 30.6 Repository im Webbrowser anschauen

GitWeb ist ein Werkzeug, um ein Git-Repository im Browser anzuschauen und zu durchsuchen (Abbildung 30-3). GitWeb zeigt eine Übersicht aller Commits und aller Branches. Man kann sich die Dateien des jeweiligen Commit anschauen und die enthaltenen Änderungen untersuchen. Über eine Suchfunktion können Dateien und Commits gefunden werden.

*Tipp: Englische Oberfläche*

The screenshot shows a web browser displaying the GitWeb interface for a repository. The title bar says "projects / .git / shortlog". The main content area is a table of commits, each with author, date, message, and a link to the commit details. The commits listed are:

Date	Author	Message
3 hours ago	Rene Preisel	"Was gibt es sonst noch" hinzugefügt
5 hours ago	Rene Preisel	Workflow-Intro vervollständigt
2 days ago	Björn Stachmann	Mindmap aktualisiert.
2 days ago	Björn Stachmann	Vorwort verbessert.
2 days ago	Björn Stachmann	Status in MindMap aktualisiert.
3 days ago	Björn Stachmann	Vorwort: Workflows besser beschreiben.
3 days ago	Björn Stachmann	fix: liste der workflows wird wieder erzeugt.
3 days ago	Björn Stachmann	Vorwort: Zutaten-Abschnitt.
3 days ago	Björn Stachmann	genericRef-Macro
4 days ago	Björn Stachmann	Überarbeitung vorwort.
4 days ago	Björn Stachmann	Darstellung für Querverweise überarbeitet.
4 days ago	Björn Stachmann	remove whitespace from titles
4 days ago	Björn Stachmann	chapterRef
4 days ago	Björn Stachmann	Vorwort Überarbeitung.

**Abb. 30-3**  
Git-Weboberfläche im Browser

Zum Starten und Stoppen wird der `instaweb`-Befehl benutzt:

```
> git instaweb start
> git instaweb stop
```

## 30.7 Zusammenarbeit mit Subversion

Git ermöglicht es, ein Subversion<sup>1</sup>-Repository zu klonen (`svn clone`-Befehl). Dabei wird die gesamte Historie importiert. In dem Git-Repository können lokale Commits angelegt werden. Es ist jederzeit möglich, den neuesten Stand aus Subversion zu holen (`svn rebase`-Befehl). Werden die lokalen Commits zurück nach Subversion gespielt (`svn dcommit`-Befehl), dann wird für jedes Commit eine eigene Subversion-Revision angelegt:

```
> git svn clone http://localhost/projecta/trunk  
> git svn rebase  
> git svn dcommit
```

## 30.8 Hooks – Git erweitern

Git stellt einen Mechanismus »Hooks« bereit, um mittels Skripten in die Verarbeitung von bestimmten Befehlen einzugreifen. So ist es möglich, vor dem eigentlichen Commit bestimmte Überprüfungen vorzunehmen, z. B. ob gewisse Konventionen bei dem Commit-Kommentar eingehalten wurden (`commit-msg`-Hook).

Im Verzeichnis `.git/hooks` eines jeden Repository sind Beispiele für die möglichen Hooks hinterlegt.

## 30.9 Mit Bisection Fehler suchen

Während der Entwicklung passiert es häufig, dass plötzlich ein Fehler in bereits erfolgreich getesteten Funktionalitäten auftaucht, der in früheren Versionen nicht vorhanden war. Eine Erfolg versprechende Strategie bei der Fehlersuche besteht darin, das Commit zu suchen, in dem der Fehler zum ersten Mal beobachtet werden kann.

Git unterstützt einen solchen Suchprozess nach fehlerhaften Commits mittels *Bisection*.

Bisection beruht auf einer binären Suche. Ausgehend von einem bekannten fehlerfreien Commit und einem bekannten fehlerbehafteten Commit wird die Historie »halbier« und der »mittlere« Commit im Workspace aktiviert. Das nun aktuelle Commit kann auf das Vorhandensein des Fehlers hin untersucht werden. Je nachdem, ob der Fehler darin vorhanden ist oder nicht, wird der verbliebene Bereich der Historie, in dem sich der Fehler verstecken muss, wieder »halbier« und

---

<sup>1</sup><http://subversion.apache.org/>

das neue »mittlere« Commit ausgewählt. Am Ende wird es normalerweise ein Commit geben, in dem der Fehler zum ersten Mal beobachtet werden kann.

Bisection wird mit dem `bisect start`-Befehl gestartet. Dabei ist als erster Parameter das fehlerhafte Commit und als zweiter Parameter das fehlerfreie Commit anzugeben:

```
> git bisect start 202d25d 87ac59e
```

Nachdem man den ausgewählten Commit untersucht hat, kann man diesen Commit entweder als gut (»good«) oder schlecht (»bad«) markieren:

```
> git bisect bad
```

Bisection wird automatisch beendet, wenn der erste fehlerhafte Commit gefunden wurde.



# 31 Die Grenzen von Git

Nachdem wir in den bisherigen Kapiteln die Vorzüge von Git und das effiziente Arbeiten mit einer dezentralen Versionsverwaltung in den Mittelpunkt gestellt haben, setzt sich dieses Kapitel mit den Problembereichen auseinander.

## 31.1 Hohe Komplexität

Der Umgang mit zentralen Versionsverwaltungen gehört mittlerweile zum Standardwissen jedes Entwicklers. Häufig beschränkt dieses Wissen sich jedoch auf die grundlegenden Funktionen, wie »Neue Versionen holen« und »Eigene Änderungen einspielen«. Das Branching und die Repository-Administration werden häufig von Build-Verantwortlichen mit Spezialkenntnissen durchgeführt.

Bei Git ist das Branching jedoch ein elementares Konzept, das bei jedem Commit, Pull und Push verstanden sein muss. Auch ist jeder Entwickler der Administrator seines eigenen Repository. Der Umgang mit Remotes und der Austausch zwischen Repositorys muss von jedem Teammitglied selbst durchgeführt werden.

Daneben bringen dezentrale Versionsverwaltungen einen zusätzlichen Push-Schritt in den normalen Ablauf. Während es bei zentralen Versionsverwaltungen reicht, ein Commit durchzuführen, damit die Änderungen für alle sichtbar sind, müssen bei Git noch die Commits mit dem push-Befehl in das zentrale Repository übertragen werden. Insbesondere in der Umstiegsphase werden Sie sehr häufig den Satz hören: »Aber das habe ich doch schon gefixt ... ähem ... Moment, ich pu... she ... jetzt noch mal pullen.«

Die genannten Punkte basieren auf der Komplexität einer dezentralen Versionsverwaltung und treffen auch auf jedes andere dezentrale Werkzeug, z. B. Mercurial<sup>1</sup>, zu. Mit großer Wahrscheinlichkeit werden Softwareentwickler diese Konzepte aber bald als Standardwissen parat haben.

---

<sup>1</sup> <http://mercurial.selenic.com/>

Daneben bringt Git auch noch ein paar Eigenarten mit. Der Ursprung von Git liegt in der Linux-Kernel-Entwicklung. Dort ist man es gewohnt, viel mit der Kommandozeile zu arbeiten, und entsprechend mächtig ist diese auch bei Git. Es gibt eine Unmenge von Befehlen und Parametern. Wenn man sich die Hilfeseiten von Git-Befehlen anschaut, wird man von den Möglichkeiten schier erschlagen. Die Ausführlichkeit der Hilfeseiten ist zwar gut, um alle Details zu verstehen, doch helfen sie wenig, wenn es darum geht, zwischen dem Wichtigen und dem Unwichtigen zu unterscheiden.

Zu guter Letzt wurde bei der Namensgebung von Befehlen häufig mehr der technische Aspekt als der Anwendungsaspekt hervorgehoben. Zum Beispiel wird zum Verwerfen von lokalen Änderungen in Git der folgende Befehl benutzt:

```
> git checkout--- DATEI
```

Wären Sie darauf gekommen?

Einige Git-Befehlsnamen haben bei anderen bekannten Versionsverwaltungen auch eine andere Bedeutung. Zum Beispiel heißt in Subversion<sup>2</sup> der Befehl zum Verwerfen von lokalen Änderungen folgendermaßen:

```
> svn revert DATEI
```

Einen revert-Befehl (Seite 179) gibt es auch in Git, doch dient er dort dazu, die Änderungen eines bereits durchgeföhrten Commit zu entfernen, indem ein neues Commit mit den »entgegengesetzten« Änderungen erzeugt wird.

Die beschriebenen Punkte führen zu der hohen Komplexität von Git, und entsprechend lange dauert das Erlernen. Deswegen ist es wichtig, bei der Einföhrung von Git die Entwickler gut vorzubereiten und für die Standard-Workflows klare Abläufe zu definieren.

Belohnt wird man dafür mit einem sehr leistungsfähigen Werkzeug, das einen nicht in der eigenen Arbeitsweise einschränkt.

---

<sup>2</sup> <http://subversion.apache.org/>

## 31.2 Komplizierter Umgang mit Submodulen

Das Submodulkonzept wurde im Kapitel »Abhängigkeiten zwischen Repositorys« ab Seite 275 beschrieben. Submodule sind eigenständige Repositorys, die in ein anderes Repository (Gesamt-Repository) eingebunden werden.

Dabei ist schon das Klonen eines Gesamt-Repository mit Submodulen kompliziert und erfordert extra Schritte (`submodule init`-Befehl und `submodule update`-Befehl). Man erkennt deutlich, dass das Submodulkonzept nachträglich eingebaut wurde.

Git geht bei Submodulen den sehr konsequenten Weg, dass im Gesamt-Repository immer genau ein spezifiziertes Commit des Submodul-Repository eingebunden wird. Das führt dazu, dass man immer einen reproduzierbaren Stand seines Projekts inklusive Submodulen wiederherstellen kann.

Leider macht das auch die Arbeit kompliziert. Änderungen an Submodulen müssen zunächst durch ein eigenes Commit abgeschlossen werden. Danach muss das neue Commit des Submoduls im Gesamt-Repository ausgewählt und anschließend durch ein zweites Commit festgeschrieben werden.

**Mit Submodulen  
arbeiten** → Seite 280

In vielen Entwicklungsprojekten will man allerdings während der Entwicklungsphase immer den aktuellsten Stand von Submodulen integrieren (z. B. den `HEAD` im `master`-Branch). Dieses Vorgehen wird nicht von Git-Submodulen unterstützt. Es muss immer explizit ein Commit ausgewählt werden.

Dadurch, dass Submodule eigenständige Repositorys sind, ist es auch nicht möglich, zwischen ihnen Dateien inklusive Historie zu verschieben.

Das alles führt dazu, dass man in Git häufig auf Submodule verzichtet. Wenn fachliche Module nur als Strukturierungseinheit innerhalb eines Projekts dienen, dann arbeitet man am besten mit einem großen Repository, in dem alle Module enthalten sind. Damit kann man immer auf dem aktuellsten Stand aller Module arbeiten und Dateien können inklusive Historie verschoben werden. Separate Release-Zyklen, Branches und Tags für einzelne Module sind mit dieser Lösung allerdings nicht möglich.

Alternativ, wenn die Module nicht so eng gekoppelt sind und eigene Release-Zyklen erfordern, greift man auf ein externes Komponenten-Repository inklusive Abhängigkeitsmanagement zurück (z. B. Maven<sup>3</sup> oder Ivy<sup>4</sup> in Java) und versioniert in Git nur die Definition der Abhängigkeiten zu den Modulen (in Maven mit der `pom.xml`).

<sup>3</sup> <http://maven.apache.org/>

<sup>4</sup> <http://ant.apache.org/ivy/>

### 31.3 Ressourcenverbrauch bei großen binären Dateien

Git hat eine sehr effiziente Speicherverwaltung. So wird der Inhalt von Dateien nur einmal gespeichert, auch wenn es mehrere Kopien einer Datei gibt. Das funktioniert auch Commit-übergreifend. Das heißt, so lange sich der Inhalt einer Datei nicht ändert, gibt es nur ein Git-Objekt für alle Commits.

Zusätzlich werden die Git-Objekte zu Paketen vereinigt und diese werden komprimiert. Das alles führt zu einer sehr ressourcenschonenden Ablage von Dateien.

Allerdings werden immer alle Versionen aller Dateien im lokalen Repository gehalten. Sobald man in Git große binäre Dateien ablegt (z. B. Filme, Bilder oder virtuelle Maschinen), führt das bei jedem Entwickler zu mehr Ressourcenverbrauch. Werden jetzt diese großen binären Dateien geändert und wird eine neue Version erstellt, dann wird sowohl die alte wie auch die neue Datei im lokalen Repository liegen.

Hier haben zentrale Versionsverwaltungen den Vorteil, dass nur die aktuellste Version bei den Entwicklern lokal vorhanden ist. Ältere Versionen liegen nur auf dem Server.

Als Konsequenz daraus sollte man versuchen, im eigentlichen Git-Entwicklungs-Repository die Anzahl von großen binären Dateien zu minimieren. »Kleine« binäre Dateien (z. B. Java-Bibliotheken) sind kein Problem für heutige Festplatten und Netzwerkbandbreiten.

Alternativ kann man den Git Large File Storage verwenden. Dieser sorgt dafür, dass die Dateiinhalte außerhalb des Repository abgelegt werden und nur kleine Link-Dateien innerhalb des Repository. Im Workflow »Ein Projekt mit großen binären Dateien versionieren« (Seite 213) ist das genaue Vorgehen beschrieben.

Falls ein Repository bereits sehr groß geworden ist, kann man mit dem Workflow »Lange Historien auslagern« (Seite 235) die älteren Versionen von Dateien entfernen.

### 31.4 Repositorys können nur vollständig verwendet werden

**Was sind Commits?**

→ Seite 31

Git versioniert in einem Commit immer das gesamte Projekt bzw. Verzeichnis. Im Gegensatz dazu verwalten die meisten zentralen Versionsverwaltungen einzelne Dateien. Deswegen unterstützen zentrale Versionsverwaltungen auch Teil-Checkouts, d. h., man kann einzelne Unterdateien separat aus der Versionierung holen und Änderungen wieder zurückspielen.

In Git sind Teil-Checkouts nicht vorgesehen, da alle Dateien sowieso schon lokal vorhanden sind.<sup>5</sup> Das Bedürfnis nach Teil-Checkouts weist häufig auf eine fehlende Modularisierung des Projekts hin, d. h., man sollte mehrere Repositorys anlegen.

Häufig werden Teil-Checkouts in zentralen Versionsverwaltungen auch benutzt, um der Langsamkeit der Systeme entgegenzuwirken – das Problem hat Git ganz sicher nicht.

Möchte man wirklich nur einzelne Dateien anschauen, dann kann man einen GitWeb-Server aufsetzen (siehe `instaweb`-Befehl). Dieser ermöglicht den direkten Zugriff auf bestimmte Dateien und Versionen.

Alternativ kann man auch den `archive`-Befehl benutzen, um nur Teile des Repository zu exportieren.

**Große Projekte aufteilen**  
→ Seite 221

**Repository im Webbrowser anschauen**  
→ Seite 293  
**Archive erstellen**  
→ Seite 291

## 31.5 Autorisierung nur auf dem ganzen Repository

Im vorigen Abschnitt wurde beschrieben, dass Git mit einem Repository nur in der Gesamtheit umgehen kann. Das trifft auch auf die Autorisierung zu.

Es ist mit Git nicht möglich, Leserechte für einzelne Verzeichnisse zu spezifizieren. Entweder hat ein Nutzer vollständigen Zugriff auf das Repository, oder er kann auf kein Verzeichnis zugreifen. Es ist möglich, zwischen lesendem und schreibendem Zugriff zu unterscheiden, d. h., ein Nutzer darf entweder nur lesen oder lesen und schreiben.

Einige zentrale Repository-Verwaltungen unterstützen auch die Schreibrechtevergabe auf einzelnen Verzeichnissen und Dateien. Dadurch ist es möglich, den schreibenden Zugriff beim Push für einzelne Nutzer zu unterbinden. Genauso ist es möglich, den schreibenden Zugriff auf Branches einzuschränken, d. h., nicht jeder hat das Recht, einen Push für einen Branch durchzuführen.

Problematisch wird es, wenn innerhalb eines Repository einzelne Verzeichnisse vor Nutzergruppen versteckt werden sollen. Dafür gibt es keine Lösung mit Git. Man muss dann die geschützten Teile in ein anderes Repository auslagern.

In Open-Source-Projekten löst man das Rechte-Problem häufig ganz anders: durch den Forking-Workflow.

Der Forking-Workflow erlaubt keine Push-Zugriffe auf das offizielle Repository, sondern benutzt Pull-Requests. Dabei erzeugen Entwick-

**Ein Projekt aufsetzen**  
→ Seite 123

**Mit Forks entwickeln**  
→ Seite 163

<sup>5</sup> Git unterstützt sogenannte *Sparse Checkouts*. Dabei enthält der Workspace nur die ausgewählten Verzeichnisse. Im Repository sind weiterhin alle Dateien vorhanden.

ler im eigenen Fork lokale Commits und schicken Pull-Requests an die Integratoren.

Die Integratoren akzeptieren die Pull-Requests erst nach dem Review der inhaltlichen Änderungen.

So wird beim Forking-Workflow die starre Rechtevergabe für Verzeichnisse durch einen Reviewprozess abgelöst. Bei großen Open-Source-Projekten (z. B. dem Linux-Kernel) gibt es mehrere Ebenen von Integratoren. Erst nach mehreren Schritten landet die Änderung vielleicht im offiziellen Repository. Dabei muss die oberste Ebene nicht noch einmal alle Commits kontrollieren, da ja bereits vertrauenswürdige Entwickler die Änderungen überprüft haben.

Eine Abwandlung des Forking-Workflow wird durch das Werkzeug *Gerrit*<sup>6</sup> unterstützt. Dabei müssen alle Codeänderungen durch einen Reviewprozess, bevor die Änderungen im offiziellen Branch sichtbar sind.

## 31.6 Mäßige grafische Werkzeuge für die Historienauswertung

Wenn es in Projekten zu Merge-Konflikten kommt oder wenn nach einem Merge Fehler auftauchen, dann hilft die Commit-Historie, die Ursachen zu finden. Dabei ist häufig die Frage zu klären, warum eine Änderung eingebaut wurde. Bei aktiver Entwicklungstätigkeit und damit vielen Commits und Merges ist das nicht trivial.

Git bietet sehr mächtige Kommandozeilenwerkzeuge (`log`-Befehl, `blame`-Befehl, `annotate`-Befehl) für die Analyse der Historie. Doch das mitgelieferte grafische Werkzeug `gitk` und auch die Plug-ins für Entwicklungsumgebungen (z. B. »EGit«<sup>7</sup>) bieten bisher nur eine unübersichtliche Darstellung. Es ist mühsam, mit dem Finger auf dem Monitor Pfade nachzuvollziehen. Da bieten kommerzielle Versionsverwaltungen klarere Darstellungsmöglichkeiten.

---

<sup>6</sup> <http://code.google.com/p/gerrit/>

<sup>7</sup> <http://eclipse.org/egit/>

# **Anhang**



# Schritt-für-Schritt-Anleitungen

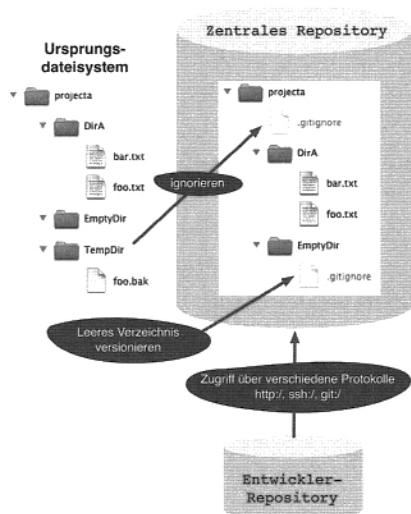
Alle Änderungen in einem Commit übernehmen .....	32
Unterschiede zwischen Commits .....	35
Commit-Historie zeigen .....	36
Selektives Commit – Änderungen auswählen .....	41
Add und Commit in einem Schritt .....	42
Was steht im Stage-Bereich? Was nicht? .....	44
Zurücknehmen von Änderungen aus dem Stage-Bereich .....	45
Änderungen zwischenspeichern .....	47
Änderungen aus dem Zwischenspeicher zurückholen .....	47
Umbenennungen und Verschiebungen verfolgen .....	54
Kopien aufspüren .....	55
Herkunft von Codeabschnitten bestimmen .....	56
Branch erstellen .....	62
Checkout verweigert. Was nun? .....	63
Branch löschen .....	64
Verschentlich gelöschten Branch wiederherstellen .....	65
Merge manuell durchführen .....	72
Push verweigert! Was tun? .....	103
Ein Commit mit einem Tag markieren .....	105
Versionierte Dateien ignorieren .....	111
Commits auf einen anderen Branch verschieben .....	114
Offene Feature-Banches anzeigen .....	158
Integrierte Features anzeigen .....	158
Alle Änderungen eines integrierten Features anzeigen .....	159
Alle Commits eines Feature-Branch finden .....	160
Submodule einbinden .....	277
Neue Version eines Submoduls verwenden .....	279
Mit Submodulen arbeiten .....	280
Submodule aktualisieren .....	281
Subtree einbinden .....	283
Neue Version eines Subtree verwenden .....	284
Änderungen in das Modul-Repository übertragen .....	285



# Workflow-Verzeichnis

## Ein Projekt aufsetzen

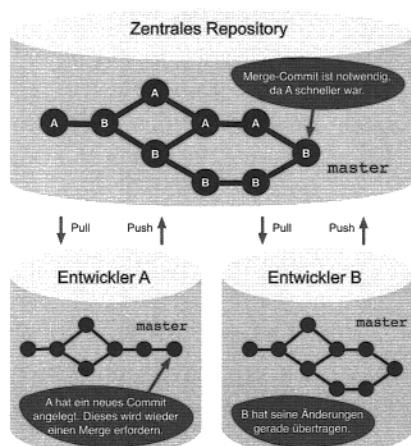
Seite 123



Ein Projektverzeichnis wird in ein neues Repository importiert. Dieses Repository wird als zentrales Repository für die Entwicklung im Team zur Verfügung gestellt.

## Gemeinsam auf einem Branch entwickeln

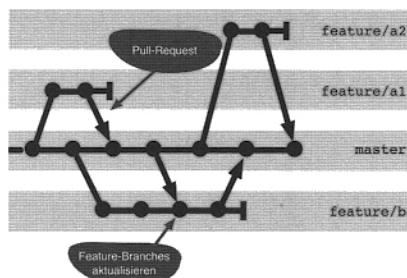
Seite 135



Alle Entwickler arbeiten auf dem gleichen Branch in ihren lokalen Repositorys und integrieren die Ergebnisse in den Haupt-Branch des zentralen Repositorys.

## Mit Feature-Branches entwickeln

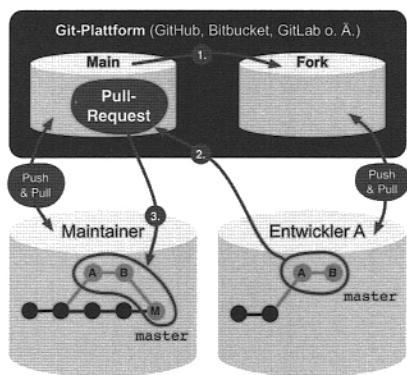
Seite 143



Jedes Feature oder jeder Bugfix wird in einem separaten Branch entwickelt. Nach der Fertigstellung wird das Feature oder der Bugfix mithilfe eines Pull-Request in den master-Branch integriert.

## Mit Forks entwickeln

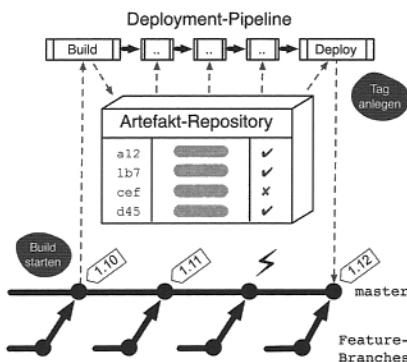
Seite 163



Entwicklung findet nur auf *Forks* statt, nie im eigentlichen Repository des Projekts. *Forks* sind unabhängige Klone des Main-Repository. Im *Fork* wird auf dem master-Branch entwickelt. Sind die Entwickler fertig, stellen sie einen Pull-Request. Der Maintainer holt dann die Änderungen vom *Fork* ab und integriert sie in den master-Branch des Main-Repository.

## Kontinuierlich Releases durchführen

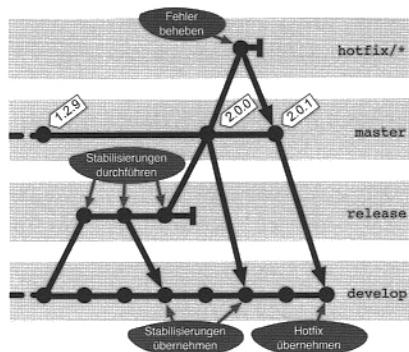
Seite 175



Für ein Projekt, das Continuous Delivery umsetzt, wird ein Release erstellt. Jedes Commit auf dem master-Branch ist ein potenzieller Release-Kandidat. Erfolgreiche Releases werden durch ein Tag markiert.

## Periodisch Releases durchführen

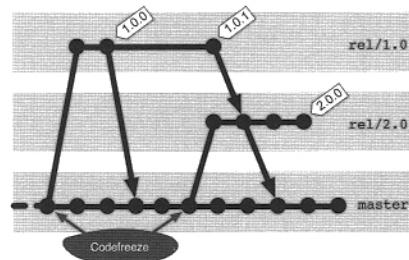
Seite 185



Für ein Projekt wird regelmäßig ein neues *Release* erstellt. Die Vorbereitung des Release findet in einem separaten Branch statt. Auf dem aktuell produktiven Release können Hotfixes durchgeführt werden.

## Mit mehreren aktiven Releases arbeiten

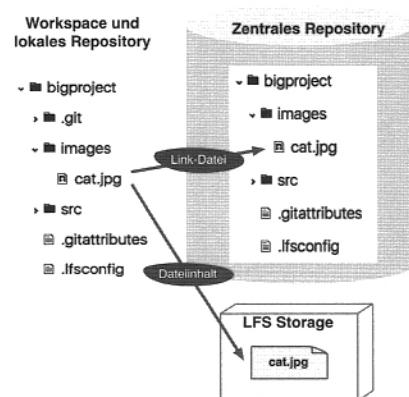
Seite 199



Für ein Softwareprodukt werden mehrere Releases parallel betreut. Jedes Release wird auf einem separaten Branch vorbereitet. Hotfixes können zwischen Releases ausgetauscht werden.

## Ein Projekt mit großen binären Dateien versionieren

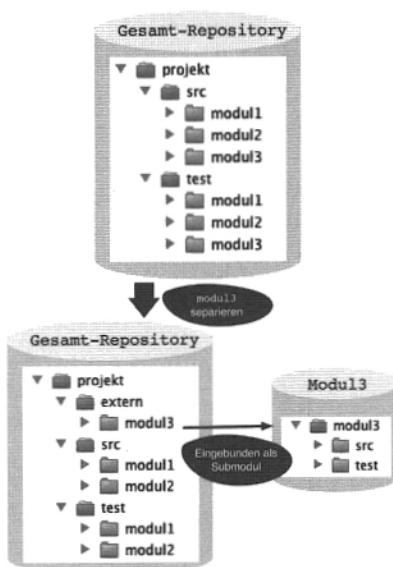
Seite 213



Das Large File Storage (*LFS*) wird benutzt, um große binäre Dateien separat vom Git-Repository zu speichern und zu versionieren.

## Große Projekte aufteilen

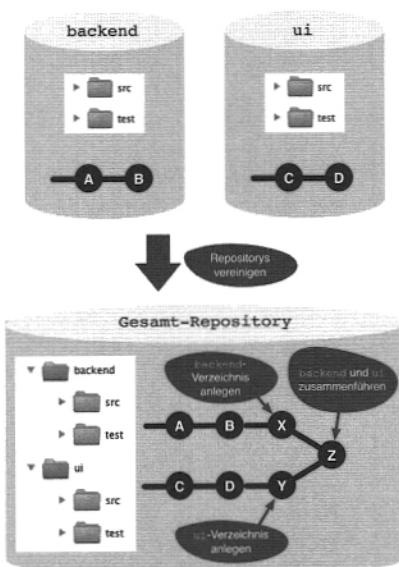
Seite 221



Ein Modul wird aus einem Projekt entfernt und in ein eigenes Repository migriert. Die Commit-Historie bleibt erhalten, unnötige Dateien und Commits werden entfernt. Das separierte Modul wird als externes Submodul wieder integriert.

## Kleine Projekte zusammenführen

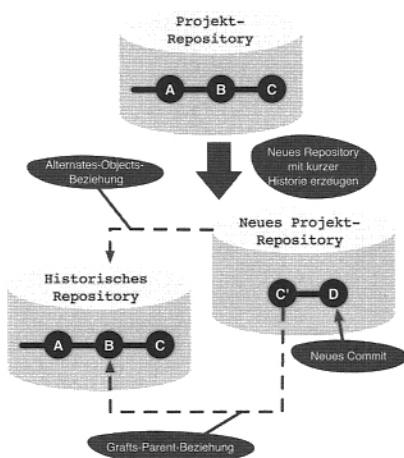
Seite 229



Mehrere Projekte mit eigenem Repository werden in einem gemeinsamen Repository vereinigt. Die Commit-Historien der Projekte bleiben erhalten.

## Lange Historien auslagern

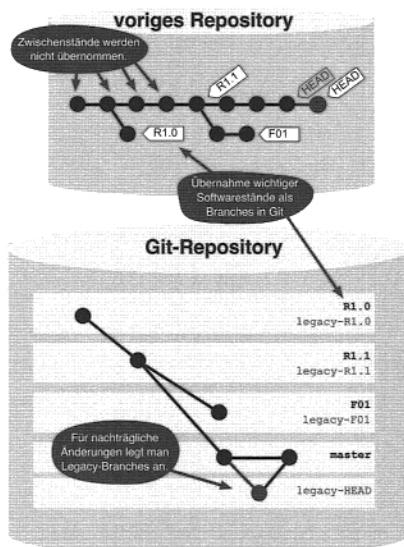
Seite 235



Ein Repository, das eine sehr lange Commit-Historie mit vielen und großen Dateien enthält, wird verkleinert. Die älteren Commits werden in ein separates Repository ausgelagert. Recherchen in der Historie sind weiterhin möglich.

## Ein Projekt nach Git migrieren

Seite 247



Ein Projekt aus einer anderen Versionsverwaltung wird nach Git migriert. Alle Softwarestände, die weiterentwickelt werden sollen, werden in das Git-Repository übernommen. Danach kann mit dem neuen Repository weitergearbeitet werden. Bei Bedarf können »nachtröpfelnde« Änderungen aus der alten Versionsverwaltung nachgezogen werden.



# Index

## Symbolen

.gitignore 32, 46  
.Notation 76  
.gitattributes 128

## A

-a (Option von tag) 106  
--abort (Option von merge) 73  
--abort (Option von rebase) 84  
add  
  -A 32  
  --all 32  
  Befehl 13, 14, 21, 32, 39, 41, 42, 44,  
    46, 48, 111  
  -i 41, 48  
  --interactive 41, 48

aktiver Branch 61

Alias 112

--all, -A (Option von add) 32  
--all, -a (Option von commit) 42  
--all (Option von lsfs-fetch) 220  
--all (Option von push) 102

annotate  
  Befehl 235, 302

apply

  Befehl 291

archive

  Befehl 257, 291, 301  
  --remote 291

--assume-unchanged (Option von  
  update-index) 111  
--assume-unchanged (Option von  
  update-index) 46, 111

## B

-b (Option von checkout) 62, 233  
Backport 172, 199, 200  
--bare (Option von clone) 19, 91, 133  
--bare (Option von init) 90, 93  
Bare-Repository 19, 90, 110, 126, 129,  
  132

Bash 9, 11

Bearbeitungskonflikte 69

Benutzeroberfläche, grafische xiv

bisect start

  Befehl 295

bisection 294

blame

  Befehl 56, 57, 235, 302  
  -C 56  
  -M 56

Blessed Repository 3, 36

Blob 3, 50, 52, 56

Branch 2, 3, 5–7, 60, 64, 66–69, 73, 74,  
  98, 115

  aktiver 61

branch

  Befehl 61, 62, 62, 66, 96–98, 158  
  -D 64  
  -d 64, 151  
  -I 96, 204  
  --list 96, 204  
  --no-merged master 158  
  -r 96  
  --remote 96  
  -v 98

Branch-Head 51

## C

-C (Option von blame) 56

-C (Option von log) 55

cat-file

  Befehl 50, 52  
  -p 50

Changeset 34, 35, 81, 88

Checkout 31, 32

checkout

  -b 62, 233  
  Befehl 62, 66, 72, 147, 220, 277,  
    279, 286  
  -f 63  
  --force 63

- ours 72, 220
- theirs 72, 220
- cherry**
  - Befehl 210, 211
- cherry-pick**
  - Befehl 78, **88**, 110, 200, 205, 207–209
  - mainline 208
  - x 208
- Cherry-Picking** 7, 42, 107
- clone**
  - bare 19, 91, 133
  - Befehl 16, 19, 21, **91**, 91–93, 133, 244, 278, 283
  - depth 244
  - no-hardlinks 224
  - recursive 278, 286
- Commit** 2–5, 7, 13, 21, 24, 30, **31**, 39, 59, 61
- commit**
  - a 42
  - all 42
  - Befehl 13, 15, 21, **31**, 37–39, 42, 44, 214, 280, 284
- Commit-Graph** 61, 81
- Commit-Hash** 13, **31**, 106
- Commit-Historie** 59, 68, 81, 82, 237, 311
- Commit-Objekt** 52, 106
- config**
  - Befehl 11, 55, 71, 218, 220
  - file 218
- contains (Option von tag) 107, 108
- Content-Tracker** 126
- continue (Option von rebase) 83
- Continuous Delivery** 171, 175
- CVS** 40
- cvsimport**
  - Befehl 260
- D**
  - D (Option von branch) 64
  - d (Option von branch) 64, 151
  - d (Option von replace) 243
  - decorate (Option von log) 107
  - delete (Option von push) 102
- Deployment-Pipeline** 175
- depth (Option von clone) 244
- dereference (Option von show-ref) 106
- Detached Head State** 97
- dezentrale Architektur** 36
- Diff** 34, 164
- diff**
  - Befehl 14, 21, **35**, 35, 44, 219, 291
  - staged 44
  - stat 35
  - word-diff 35, 112
- Diff-Algorithmus** 111
- diff-Format** 14
- difftool**
  - Befehl 77
- digitale Unterschrift** 106
- dirstat (Option von log) 37
- Dual Workspace** 255, 258, 260, 261
- E**
- EGit** 302
- Eingabeaufforderung** 9
- F**
  - f (Option von push) 190
- Fast-Forward** 102, 116
- Fast-Forward-Merge** **74**, 74, 76, 197
- Feature-Branch** 6, 75, 87, 88, **143**
- Feature-Merge** 76
- Feature-Toggle** 148
- fetch**
  - Befehl **99**, 99, 101, 116, 147, 151, 213, 230, 233, 243, 266, 270, 279
  - Refspec 115
- ff-only (Option von pull) 146
- file (Option von config) 218
- File Permissions** 32
- file-Protokoll** 92
- filter-branch**
  - Befehl 222, 224, 225, 228, 236, 240, 241
  - index-filter 228
  - subdirectory-filter 228
  - tag-name-filter 240
- find-copies-harder (Option von log) 55
- First-Parent** **75**
  - first-parent (Option von log) 75, 158, 188, 189
- First-Parent-History** 75, 76, 135, 143, 158, 192
- Floating Tag** 108
- follow (Option von log) 55
- follow-tags (Option von push) 101
- force, -f (Option von checkout) 63
- force, -f (Option von push) 102

- 
- force, -f (Option von tag) 107
  - force (Option von push) 156
  - Fork 87, 91, **163**, 163–173, 308
  - Fork Repository 3
  - format (Option von log) 37
  - fsck
    - Befehl 33
  - ftp(s)-Protokoll 92
  - G**
    - Garbage Collection 83, 84, 109
  - gc
    - Befehl 51, **66**, 66, 83, 84, 109, 225, 241
      - prune 241
  - Gerrit 302
  - Gesamt-Repository 281
  - Git Large File Storage **213**, 300
  - Git-Bash 12
  - Git-Flow 153, 196
  - git-Protokoll 92
  - gitignore **46**
  - gitk
    - Befehl 292
  - GitWeb 293
  - Glob-Syntax 46
  - Globs 41
    - graft (Option von replace) 240
    - graph (Option von log) 38
    - grep (Option von log) 159, 160, 207
  - große binäre Dateien 235
  - gui
    - Befehl 292
  - H**
    - hard (Option von reset) 64
  - Hash 106
  - hash-object
    - Befehl **49**
      - w 49
  - Hashcode 50, 51
  - Hashkollision 52
  - Hashwert 4, 5
  - HEAD 44, 75
  - HEAD-Revision **37**, 37
  - Hooks 89, 294
  - Hotfix 81, 185, **193**, 197, 199, 203
  - http(s)-Protokoll 92
  - Hunk 48
- I**
    - Ignore 248
    - Ignorieren von Dateien 46
  - Index **39**
    - index-filter (Option von filter-branch) 228
  - Inhaltliche Konflikte 69
  - init (Option von submodule update)
    - 278, 280
  - init
    - bare 90, 93
    - Befehl 12, 13, 21, **89**, 90, 93
  - instaweb
    - Befehl 293, 301
  - interactive, -i (Option von add) 41, 48
  - interactive, -i (Option von rebase)
    - 114, 290
  - interaktives Rebasing 7, 260
  - Issue Tracker 170
  - K**
    - Klon 2, 16, 26, 91, 92
    - Kommandozeile xiv
    - Konflikt **69**, 81, 83, 103, 139
    - Konfliktmarkierung **70**
    - Kopieren von Dateien 54
  - L**
    - Legacy-Branch **254**, 258, 260, 261
    - LFS **213**, 213–220, 309
    - lfs-clone
      - Befehl 219
    - lfs-fetch
      - all 220
      - Befehl 220
    - lfs-install
      - Befehl **216**, 216
    - lfs-ls-files
      - Befehl **218**
    - lfs-prune
      - Befehl 220
    - lfs-status
      - Befehl **219**
    - lfs-track
      - Befehl **217**, 217, 218
    - lfs-untrack
      - Befehl **217**, 218
  - Lightweight Tag **106**
  - Link-Datei **214**, 214, 216, 218–220
    - list, -l (Option von branch) 96, 204
    - list (Option von tag) 205

- log**
- Befehl 15, 18, 21, 36, 38, 54, 55, 75–77, 107, 188, 207, 235, 243, 302
  - C 55
  - decorate 107
  - dirstat 37
  - find-copies-harder 55
  - first-parent 75, 158, 188, 189
  - follow 55
  - format 37
  - graph 38
  - grep 159, 160, 207
  - M 54, 55
  - merge 77
  - n 37
  - oneline 37, 158, 188, 189, 238
  - pretty 86
  - pretty=oneline 238
  - shortstat 37
  - stat 37
  - summary 54
- M**
- M (Option von blame) 56
  - M (Option von log) 54, 55
  - m (Option von tag) 106
- Main Line 254
- Main-Repository 36, 163, 163–169, 173, 308
- mainline (Option von cherry-pick) 208
  - mainline (Option von revert) 179
- Maintainer 87, 163–165, 166, 166, 168–172, 308
- Merge 6, 17, 18, 28, 67, 68, 69, 74, 75, 78, 83, 84, 100, 110
- merge (Option von log) 77
- merge
- abort 73
  - Befehl 67, 73, 78, 82, 83, 88, 100, 101, 115, 287
  - no-ff 74, 192
  - s 281
  - s ours 206
  - strategy=subtree 281
- Merge-Base 77, 113
- merge-base
- Befehl 77, 160
- Merge-Commit 6, 67, 73–75, 77, 135, 164
- Merge-Konflikt 87, 169
- Merge-Request 148
- Merge-Tool 83
- mergetool
- Befehl 71, 103, 220
- Migration 247, 249, 311
- mv
- Befehl 232
- N**
- n (Option von log) 37
  - no-assume-unchanged (Option von update-index) 111
  - no-ff (Option von merge) 74, 192
  - no-hardlinks (Option von clone) 224
  - no-merged master (Option von branch) 158
  - no-tags (Option von pull) 108
- Notation xiii
- O**
- Object Database 49, 52, 89
  - Octopus-Merge 73, 76
  - oneline (Option von log) 37, 158, 188, 189, 238
- Open-Source-Projekt 163
- Origin 92
- Original-Repository 91, 92
- ours (Option von checkout) 72, 220
- P**
- p (Option von cat-file) 50
- Pack Files 51, 52
- Parent-Objekt 52, 53
- Patch 166, 166
- Plumbing 49
- Porcelaine 49
- prefix (Option von subtree add) 283
  - pretty (Option von log) 86
  - pretty=oneline (Option von log) 238
- Protokoll 92
- prune (Option von gc) 241
- Pull 2, 4, 6, 102
- pull
- Befehl 17–21, 83, 100, 136, 138, 140, 141, 146, 152, 167, 213, 214
  - ff-only 146
  - no-tags 108
  - rebase 83, 140
  - Refspec 115
- Pull-Request 87, 143, 146, 148, 163, 168, 170, 178, 302
- Push 2, 6, 30

push  
  --all 102  
  Befehl 19–21, 101–103, 105, 108,  
    136, 140, 146, 147, 152, 156,  
    190, 214, 266, 279, 280, 286,  
    287, 291, 297  
  --delete 102  
  -f 102, 190  
  --follow-tags 101  
  --force 102, 156  
  Refspec 115  
    --set-upstream 102, 131, 146  
  --tags 101, 105, 108  
  -u 102

**Q**  
Quality-Gate 168, 170

**R**  
Remote-Repository 101  
--really-refresh (Option von  
  update-index) 111  
Rebase 87, 95  
--rebase (Option von pull) 83, 140  
rebase  
  --abort 84  
  Befehl 81, 82–84, 86, 88, 110, 290  
  --continue 83  
  -i 114, 290  
  --interactive 114, 290  
  --skip 84  
Rebasing 81, 83–85, 87  
--recursive (Option von clone) 278, 286  
--recursive (Option von submodule  
  update) 278  
Ref 104, 115  
Reflog 65, 90, 109, 110  
reflog  
  Befehl 65, 109  
Refspec 104, 116  
  angeben 270  
Refspecs 104  
--rejoin (Option von subtree pull) 285  
Release 187, 309  
Release-Historie 188  
Remote 92  
--remote, -r (Option von branch) 96  
--remote (Option von archive) 291  
remote  
  Befehl 92  
  -v 92

  --verbose 92  
  remote add  
    Befehl 283  
  remote rm  
    Befehl 92  
  remote show  
    Befehl 92, 98  
  Remote-Branch 97, 98  
  Remote-Tracking-Branch 96, 98, 99, 101,  
    115, 255  
  Rename Detection 54, 54, 55  
  replace  
    Befehl 236, 239, 240, 242, 243  
    -d 243  
    --graft 240  
  Repository 1–7, 12, 24, 31, 43, 44, 49  
    anlegen 12, 24  
    klonen 16, 26  
  Repository-URL 92  
  reset  
    Befehl 45, 45, 64, 153, 190, 260  
    --hard 64  
  rev-parse  
    Befehl 179  
    --verify 179  
  revert  
    Befehl 179, 182, 191, 203, 298  
    --mainline 179  
  Review 168  
rm  
  Befehl 42, 42, 225, 232  
rsync-Protokoll 92

**S**  
-s (Option von tag) 106  
-s ours (Option von merge) 206  
selektive Commits 110  
--set-upstream, -u (Option von push) 102  
--set-upstream (Option von push) 131,  
  146  
Shallow-Klon 244  
Shared Repository 3  
Shell xiv  
  --short, -s (Option von status) 40  
  --shortstat (Option von log) 37  
show  
  Befehl 72  
show-ref  
  Befehl 106, 116  
  --dereference 106  
  --tags 106

- skip (Option von rebase) 84
- SourceTree 23
- Sparse Checkouts 301
- squash (Option von subtree add) 283, 284
- ssh-Protokoll 92
- Stable Release Version 171, 172
- Stage-Bereich 39, 41–45, 62, 68, 71, 111
- staged (Option von diff) 44
- Staging 25, 39
- stash
  - Befehl 45, 47, 48, 62, 63, 68
  - stash list
    - Befehl 47
  - stash pop
    - Befehl 47, 63
  - Stashing 42
  - stat (Option von diff) 35
  - stat (Option von log) 37
  - status
    - Befehl 13–15, 21, 39–41, 46, 48, 70, 111, 217, 257
    - s 40
    - short 40
  - strategy=subtree, -s (Option von merge) 281
  - subdirectory-filter (Option von filter-branch) 228
- Submodul 221, 227, 275
- submodule
  - Befehl 275
  - submodule add
    - Befehl 227, 277, 286
  - submodule init
    - Befehl 276, 278, 286, 299
  - submodule status
    - Befehl 278
  - submodule update
    - Befehl 278, 280, 281, 286, 299
    - init 278, 280
    - recursive 278
- Subtree 281
- subtree
  - Befehl 275
- subtree add
  - Befehl 281, 283–286
  - prefix 283
  - squash 283, 284
- subtree pull
  - Befehl 284–286
  - rejoin 285
- subtree split
  - Befehl 285, 287
- Subversion 40, 54
- summary (Option von log) 54
- svn clone
  - Befehl 294
- svn dcommit
  - Befehl 294
- svn rebase
  - Befehl 294
- T**
- Tag 2, 33, 99, 101, 105, 105, 106, 113, 115, 171, 172
- tag
  - a 106
  - Befehl 105, 106–108, 181
  - contains 107, 108
  - f 107
  - force 107
  - list 205
  - m 106
  - s 106
  - u 106
  - tag-name-filter (Option von filter-branch) 240
- Tag-Objekt 106
- tags (Option von push) 101, 105, 108
- tags (Option von show-ref) 106
- Texteditor 11
- theirs (Option von checkout) 72, 220
- TODO-Kommentare 170, 173
- Topic 143
- Topic-Branch 143
- Tree 3, 50, 52, 56
- Trunk 254
- U**
- u (Option von tag) 106
- Umbenennen von Dateien 54
- Update 40
- update-index
  - assume-unchanged 111
  - assume-unchanged 46, 111
  - Befehl 46, 111
  - no-assume-unchanged 111
  - really-refresh 111
- Upstream-Branch 64, 97, 98, 101, 102

**V**

-v (Option von branch) 98  
--verbose, -v (Option von remote) 92  
--verify (Option von rev-parse) 179  
Verschieben von Dateien 54  
Vorgänger-Commit 53

**W**

-w (Option von hash-object) 49  
Windows 9  
--word-diff (Option von diff) 35, 112  
Workflow xi, **117**  
Workflow Repository 3  
Workspace 1, 2, 4, 12, 24, 32, 43, 44, **89**,  
91  
worktree  
    Befehl **289**

**X**

-x (Option von cherry-pick) 208

**Z**

Zeilenenden 127  
Zugriffsberechtigungen 32