5. Auflage

René Preißel · Bjørn Stachmann

Git

Dezentrale Versionsverwaltung im Team
Grundlagen und Workflows

dpunkt.verlag

René Preißel arbeitet als freiberuflicher Softwarearchitekt, Entwickler und Trainer. Er beschäftigt sich seit vielen Jahren mit der Entwicklung von Anwendungen und dem Coaching von Teams. Seine Arbeitsschwerpunkte liegen im Bereich Softwarearchitektur, Java-Entwicklung und Build-Management.

Mehr Informationen unter *www.eToSquare.de*.

Bjørn Stachmann arbeitet als Software Developer für die Otto GmbH & Co KG in Hamburg. Seine Schwerpunkte liegen in den Bereichen Java-Entwicklung, Softwarearchitektur und Hadoop. Sein aktuelles Arbeitsfeld ist der Hadoop-basierte Big-Data-Stack für die BI-Plattform BRAIN.

René Preißel · Bjørn Stachmann

Git

Dezentrale Versionsverwaltung im Team
Grundlagen und Workflows

5., aktualisierte und erweiterte Auflage

dpunkt.verlag

René Preißel
Bjørn Stachmann
git@eToSquare.de

Lektorat: René Schönfeldt
Lektoratsassistenz/Projektkoordinierung: Anja Weimer
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Da-TeX Gerd Blumenstein, Leipzig, *www.da-tex.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions 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 978-3-86490-649-7
PDF 978-3-96088-730-0
ePub 978-3-96088-731-7
mobi 978-3-96088-732-4

5., aktualisierte und erweiterte Auflage 2019

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

5 4 3 2 1 0

Dieses Buch wurde selbstverständlich mit Git erstellt:

```
Version (Commit-Hash):
commit ff46fbf39089cd6435367cda6038bf07d54075c6
Author: bstachmann <git@eToSquare.de>

    Add section about push --force

Anzahl Commits:

2169

Änderungsstatistik:

 789 files changed, 341725 insertions(+), 228 deletions(-)

Status:

On branch master
Your branch is up to date with 'fork/master'.

nothing to commit, working tree 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 Frühjahr 2019, 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 Beitragen 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 173).

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 korrumpieren.

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 Plugins 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 fünften Auflage

Git ist nun schon seit vielen Jahren im Einsatz. Die Projekte werden immer größer, die Entwickler werden mehr, und daraus ergeben sich neue Herausforderungen. Deshalb haben wir dem Umgang mit sehr großen Repositorys ein eigenes Kapitel gewidmet, in dem Konzepte wie Sparse Checkout und Shallow Clone sowie Tools wie Watchman vorgestellt werden.

Große Repositorys *→ Seite 285*

Ein Abschnitt über den im Alltag sehr nützlichen `show`-Befehl, den wir bisher übersehen hatten, ist jetzt mit an Bord.

Das Konzept des HEAD-Commits, welches bisher in verstreuten Anmerkungen nur angerissen wurde, hat nun einen eigenen Abschnitt erhalten. Ebenso erläutern wir jetzt in einem Abschnitt gebündelt die wichtigsten Schreibweisen für Commit-Parameter.

Das HEAD-Commit *→ Seite 35*
Schreibweisen für Commits *→ Seite 40*

Von Lesern und Seminarteilnehmern gab es immer mal wieder Fragen zur Bedeutung des Master-Branch. Deshalb haben wir einen Abschnitt hierzu ergänzt.

***Der* Master-Branch** *→ Seite 68*

Auch für das `origin`-Repository haben wir einen erläuternden Abschnitt hinzugefügt.

***Das* Origin-Repository** *→ Seite 99*

Dringend nötig war auch die Überarbeitung des Kapitels über Jenkins, da sich sowohl beim Build-Server als auch bei den Hosting-Plattformen für Git in den letzten Jahren einiges bewegt hat.

Integration mit Jenkins *→ Seite 273*

Dank des Feedbacks von Lesern konnten wir darüber hinaus in den Grundlagen-Kapiteln (1 bis 12) viele Dinge präziser und hoffentlich auch verständlicher formulieren.

Feedback zum Buch

Falls Sie Fehler entdecken, Fragen oder Vorschläge haben, können Sie Feedback über unsere Website geben:

http://kapitel26.github.io/feedback

Oder Sie schreiben uns eine E-Mail:

`git@eToSquare.de`

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. Dies 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.

Ein Projekt aufsetzen → *Seite 133*
Release durchführen → *Seite 195*

Die Zutaten

Gleich ausprobieren! → *Seite 9*
Was sind Commits? → *Seite 31*
Tipps und Tricks → *Seite 117*
Workflow-Verzeichnis → *Seite 325*

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

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. Darüber 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 195). 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 66), haben wir in Schritt-für-Schritt-Anleitungen beschrieben.

Schritt-für-Schritt-Anleitungen
→ *Seite 323*

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 besonderen Belange der 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 320 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 momentan 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?

Zusammenfassung am Ende der Einführungskapitel

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:

Überspringen Sie diese Kapitel, wenn Sie es eilig haben.

Kapitel 6, Das Repository
Kapitel 9, Mit Rebasing die Historie glätten
Kapitel 12, Versionen markieren
Kapitel 30, Abhängigkeiten zwischen Repositorys
Kapitel 13, Tipps und Tricks

Wo finde ich was?

Workflow-Verzeichnis *→ Seite 325*

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

Anleitungsverzeichnis *→ Seite 323*

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

Index *→ Seite 330*

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.

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

Index → Seite 330

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 330

Viele Beispiele in diesem Buch beschreiben wir mit Kommandozeilenaufrufen. 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[1], TortoiseGit[2], SmartGit[3], GitX[4], Git Extensions[5], tig[6], qgit[7]), einige Entwicklungsumgebungen, die Git von Haus aus unterstützen (IntelliJ[8], Xcode 4[9]), und viele Plugins für Entwicklungsumgebungen (z. B. EGit für Eclipse[10], NBGit für NetBeans[11], Git Extensions für Visual Studio[12]). Wir haben uns trotzdem für die Kommandozeilenbeispiele entschieden, weil

***Grafische Werkzeuge für Git** → Seite 310*

- Git-Kommandozeilenbefehle 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 Kommandozeile für viele Anwendungsfälle unschlagbar effizient ist.

[1] *http://www.sourcetreeapp.com/*
[2] *http://code.google.com/p/tortoisegit/*
[3] *http://www.syntevo.com/smartgit/*
[4] *http://gitx.frim.nl/*
[5] *http://gitextensions.github.io/*
[6] *https://jonas.github.io/tig/*
[7] *http://sourceforge.net/projects/qgit/*
[8] *http://www.jetbrains.com/idea/*
[9] *https://developer.apple.com/xcode/*
[10] *http://eclipse.org/egit/*
[11] *https://netbeans.org/kb/docs/ide/git.html*
[12] *http://gitextensions.github.io/*

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 Kommandozeilenaufrufe stellen wir wie folgt dar:

```
> git commit
```

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

```
> 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 bekanken wir uns bei Maurice Kowalski, Jochen Schlosser, Oliver Zeigermann, Ralf Degner, Michael Schulze-Ruhfus und einem halben Dutzend anonymer 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 Bjørn 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 Fragen, Korrektur- 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, Herbert Feichtinger, Jason Stäuble (@shadyhh), Udo Heyn (@UdoHeyn) und Herrn Böckler. Vielen Dank dafür!

Danksagungen zur fünften Auflage

Ein ganz besonderer Dank geht an Dr. Oliver Thilmann, der uns eine lange E-Mail mit sehr vielen Anmerkungen und Verbesserungsvorschlägen geschrieben hat, die uns geholfen hat, diese Auflage präziser und verständlicher zu gestalten. Nützliche Hinweise haben wir außerdem erhalten von: Gerald Schroll (@gschroll), @TheDet und Hampa Brügger (@hampa-git).

»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

Mehr über Git

Anhang

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 *Repositorys* 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.

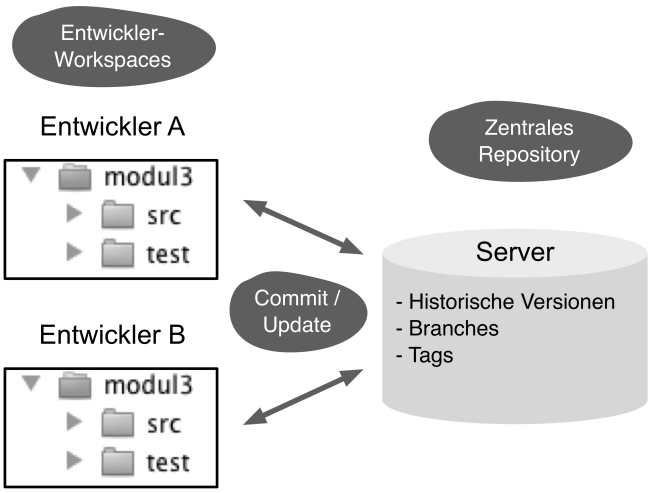

Abb. 1–1
Zentrale Versionsverwaltung

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*). Parallele Entwicklungsstränge (*Branches*) und benannte Versionen (*Tags*) werden auch zentral verwaltet.

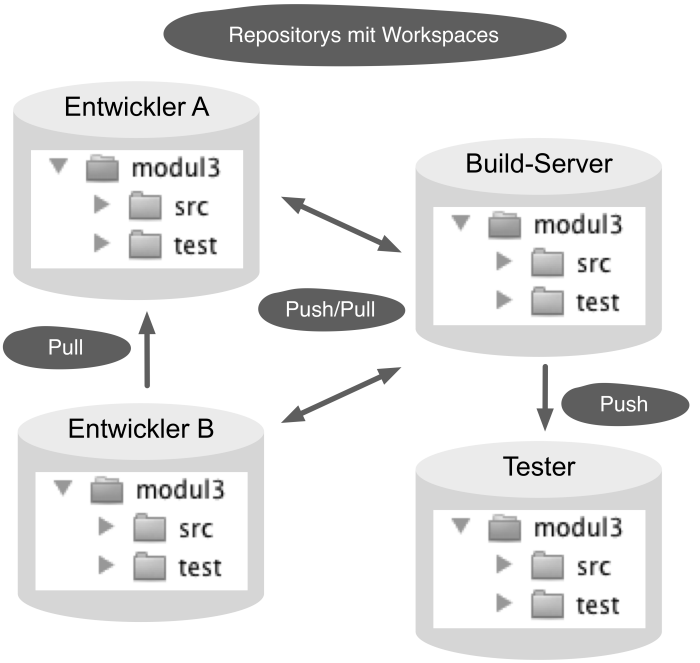

Abb. 1–2
Dezentrale Versionsverwaltung

Das Repository
→ *Seite 53*

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 101*

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

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.

Ein Projekt aufsetzen
→ Seite 133

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 *Repositorys* 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 *Repositorys* 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 *Repositorys* erproben, bevor man sie in das Original-*Repository* überträgt.

1.2 Das Repository – die Grundlage dezentralen Arbeitens

In zentralen Versionsverwaltungen werden alle Projekte, selbst dann, wenn sie inhaltlich nichts miteinander zu tun haben, in einem gemeinsam genutzten Repository abgelegt. In Git hingegen bekommt jedes Projekt[1] sein eigenes Repository.

Das Repository
→ Seite 53

Kern des *Repositorys* ist ein effizienter Datenspeicher: die *Object Database*. Dort speichert Git Dateiinhalte, Verzeichnisstruktur und Versionshistorie des Projekts, in Form von sogenannten *Objekten*, das sind unter anderem:

Versionen (Commits): Ein Commit in Git beschreibt einen wiederherstellbaren Gesamtzustand des Projekts. Es ist eine Momentaufnahme des Verzeichnisbaumes (Tree) mit allen Dateiinhalten (Blobs), ggf. mitsamt Unterverzeichnissen (ebenfalls Trees). Außerdem werden der Autor, die Uhrzeit, ein Kommentar und die Vorgängerversion festgehalten.

Verzeichnisse (Trees): Ein Tree ist eine Liste mit allen enthaltenen Dateien und Unterverzeichnissen. Jeder Datei ist ein Blob zugeordnet, jedem Unterverzeichnis ein Tree.

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

Für jedes *Objekt* wird ein hexadezimaler *Hashwert* berechnet, z. B. `1632acb65b01c6b621d6e1105205773931bb1a41`. Dieser dient 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:

---

[1] Sie ahnen es sicher schon: Mit *Projekt* meinen wir hier nicht der Begriff aus der Projektmanagementlehre. In der Git-Community und in vielen Git-Tools verwendet man den Begriff *Projekt* schlicht und einfach für eine Menge von Dateien, die man gemeinsam bearbeiten und versionieren möchte. So tun wir es auch in diesem Buch.

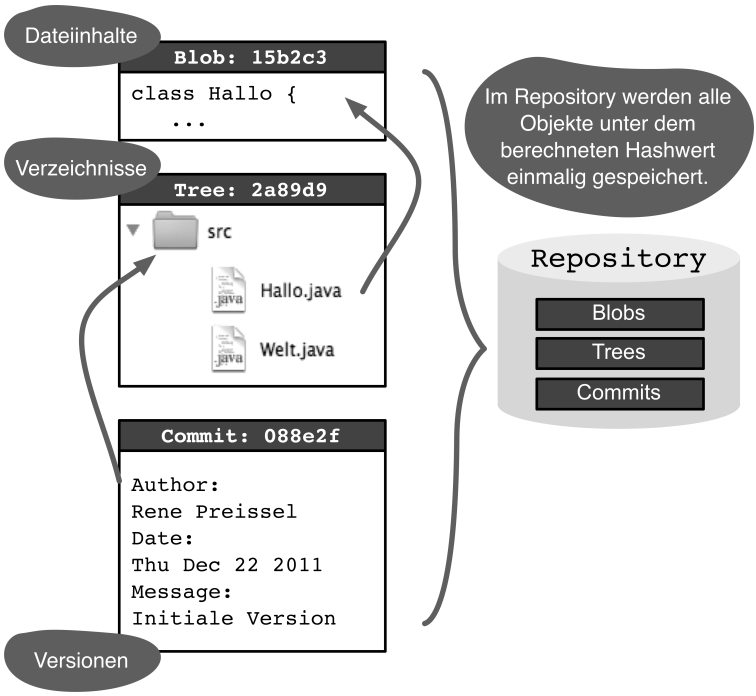

Abb. 1–3
Ablage von Objekten im Repository

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 Objekte: Der *Hashwert* wird aus dem Inhalt der *Objekte* 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!

Angenommen zwei Entwickler bearbeiten jeweils eine Kopie desselben Projekts, so entstehen zwei Versionen des Projekts, die sich an manchen Stellen in einigen Dateien unterscheiden. Eine solche Verzweigung nennt man *Branch*.

Interessant wird es, wenn man die Ergebnisse der beiden zu einer neuen Version zusammenführt. Man spricht dann von einem *Merge*. Einen Merge kann man manuell durchführen, indem man geeignete Abschnitte aus beiden Versionen zusammenkopiert. Das ist leider mühsam, fehleranfällig und später oft nicht mehr nachvollziehbar. Deshalb zählt es zu den wichtigen Fähigkeiten einer Versionsverwaltung, den Merge-Vorgang zu unterstützen und die Zusammenführung in der Historie zu dokumentieren.

Branches verzweigen
→ *Seite 63*

Das Verzweigen (Branching) und das Zusammenführen (Merging) wird von vielen Versionsverwaltungen als Sonderfall behandelt und gehört 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 Einzelergebnisse ist dabei eine große Herausforderung. Deshalb wurde Git von vornherein 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, führen Änderungen durch und erstellen jeweils ein neues *Commit*. 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 in sein *Repository* importiert, kann er Git die Versionen zusammenführen lassen (*Merge*). Ist dies erfolgreich, so erstellt Git ein neues *Commit*, das beide Änderungen enthält: das *Merge-Commit*. Wenn der andere Entwickler dieses *Commit* abholt, sind beide wieder auf einem gemeinsamen Stand.

Mit Feature-Branches entwickeln
→ *Seite 153*

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*).

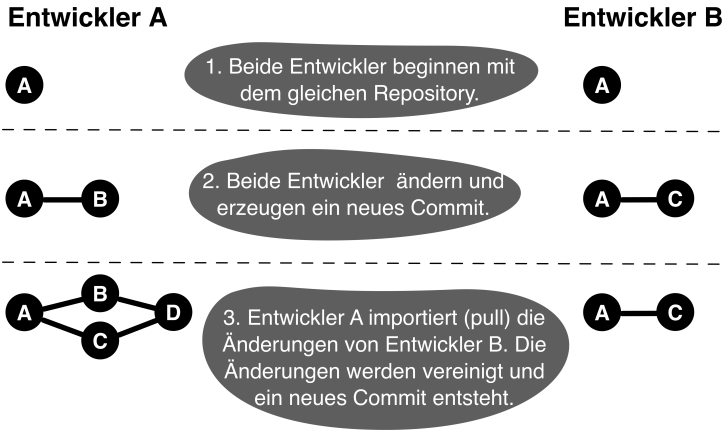

Abb. 1–4
Branches entstehen durch paralleles Arbeiten.

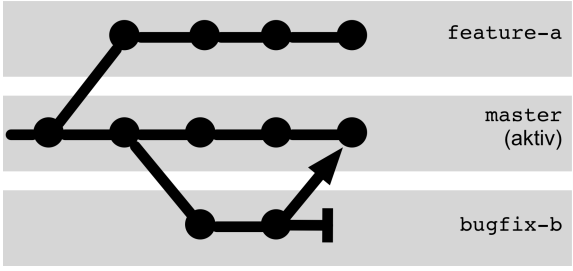

Abb. 1–5
Explizite Branches für unterschiedliche Aufgaben

Beim Austausch zwischen *Repositorys* (*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; dies nennt man *Rebasing*.

Mit Rebasing die Historie glätten
→ *Seite 87*

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

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

Interaktives Rebasing
→ *Seite 308*

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 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 127

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

Falls Sie ein vielbeschäftigter Manager sind und noch nach Gründen 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 315 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.
- 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 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

Tipp: Autovervollständigung

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=
```

Achtung! In der Git-Bash müssen Sie »/« als Pfadtrenner 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 »kaputt zu machen«, 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.

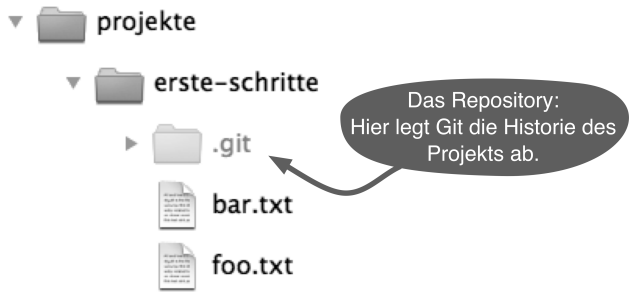

Abb. 2–2
Das Repository-Verzeichnis

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 40-stelligen sogenannten *Commit-Hash*, der das neue Commit identifiziert. Git zeigt hier nur die ersten Stellen `2f43cf0` an.

```
> 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.

```
> 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
```

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). Dazu nutzt man statt des `diff`-Befehls den `difftool`-Befehl.

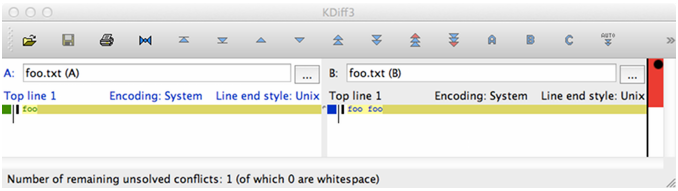

Abb. 2–3
Diff-Darstellung in grafischem Tool (kdiff3)

Ein Commit nach Änderungen

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

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