5. Auflage

René Preißel · Bjørn Stachmann

Git

Dezentrale Versionsverwaltung im Team
Grundlagen und Workflows

dpunkt.verlag

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.

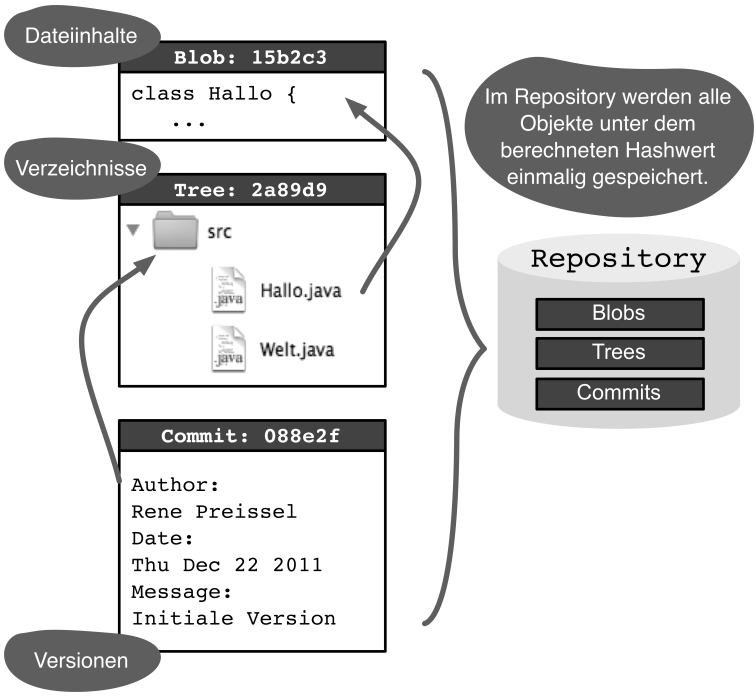

Abb. 1–3
Ablage von Objekten im Repository

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*).

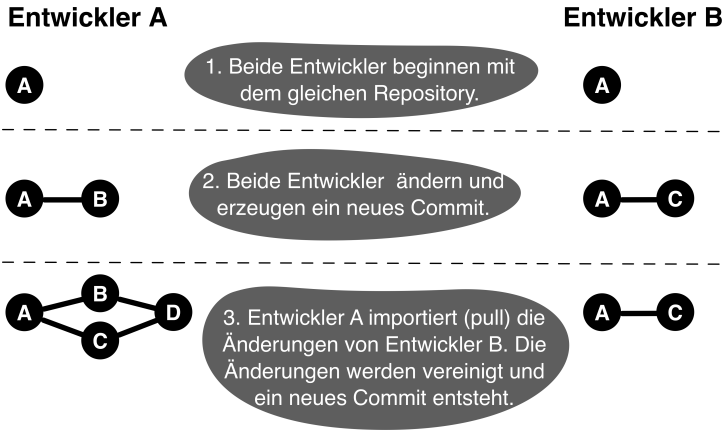

Abb. 1–4
Branches entstehen durch paralleles Arbeiten.

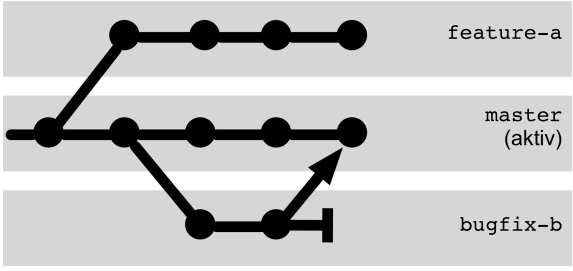

Abb. 1–5
Explizite Branches für unterschiedliche Aufgaben

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*