

1. An wen richtet sich dieses Buch?
2. Wie ist das Buch zu lesen?
3. Komponenten
4. „Installieren und das Git-Repository“
5. Dokumentation und Hilfe
6. Downloads und Kontakt
7. Dankesnungen
8. Vorwort zur 2. Auflage
9. Vorwort zur CreativeCommons-Ausgabe

1. Einführung und erste Schritte

- 1.1. Was ist Git?
- 1.2. Erste Schritte mit Git
- 1.3. Git konfigurieren

2. Grundlagen

- 2.1. Git-Kommandos
- 2.2. Das Objektmodell

3. Praktische Versionsverwaltung

- 3.1. Revertieren, Branches und Tags
- 3.2. Versionen wiederherstellen
- 3.3. Branches zusammenführen: Merges
- 3.4. Merge-Konflikte lösen
- 3.5. Einzelne Commits übernehmen: Cherry-Pick
- 3.6. Wiederherstellung von Repositorys

4. Fortgeschritten Konzepte

- 4.1. Commits verschieben – Rebasing
- 4.2. Die Geschichte umschreiben – Interaktives Rebasing
- 4.3. Wer hat diese Änderungen gemacht? – git blame
- 4.4. Dateien prüfen
- 4.5. Veränderungen auslegen – git stash
- 4.6. Commits annotieren – git notes
- 4.7. Mehrere Root-Commits
- 4.8. Regressionsen finden – git bisect

5. Verteiltes Git

- 5.1. Was funktioniert verteilt Versionsverwaltung?
- 5.2. Repositories klonen
- 5.3. Commits herunterladen
- 5.4. Commits hochladen: git push
- 5.5. Remotes untersuchen
- 5.6. Verteilter Workflow mit mehreren Remotes
- 5.7. Remotes verwäpfen
- 5.8. Tag austauschen
- 5.9. Patch über E-Mail
- 5.10. Ein verteilter, hierarchischer Workflow
- 5.11. Unterprojekte verwalten

6. Workflows

- 6.1. Anwender
- 6.2. Ein Branching-Modell
- 6.3. Ein Pull-Request Management
7. Git auf dem Server

 - 7.1. Einen Git-Server hosten
 - 7.2. Gitolite: Git einfach hosten
 - 7.3. Git-Daemon: Anonymer, lesender Zugriff
 - 7.4. GitHub: Das integrierte Web-Frontend
 - 7.5. CGit – CGI für Git

8. Git überall

 - 8.1. Git-Attribute – Dateien gesondert behandeln
 - 8.2. Hooks
 - 8.3. Eigene Git-Kommandos schreiben
 - 8.4. Versionsgeschichte umschreiben

9. Zusammenspiel mit anderen Versionsverwaltungssystemen

 - 9.1. Subversion
 - 9.2. Eigene Importer

10. Shell-Integration

 - 10.1. Git und die Bash
 - 10.2. Git und die Z-Shell

11. Distributionen

 - A.1. Linux
 - A.2. Mac OS X
 - A.3. Windows

- B. Struktur eines Repositorys

 - B.1. Aufbauen
 - B.2. Performance

Kapitel 4. Fortgeschrittene Konzepte

Das folgende Kapitel behandelt ausgewählte fortgeschrittene Konzepte. Im Vordergrund steht das Rebasing-Kommando mit seinen vielfältigen Anwendungen. Wir finden heraus, wer wann eine Zeile im Quellcode verändert hat (*Blame*) und wie Sie Git anweisen, Dateien und Verzeichnisse zu ignorieren. Außerdem wird darauf eingegangen, wie Sie Änderungen am Working Tree in den Hintergrund stellen (*stash*) und Commits annotieren (*Notes*). Zuletzt zeigen wir Ihnen, wie Sie schnell und automatisiert Commits finden, die einen Bug einführen (*Bisect*).

4.1. Commits verschieben – Rebasing

Im Abschnitt über die Interna von Git wurde bereits erwähnt, dass man Commits in einem Git-Repository (anschaulich: dem Graphen) beliebig verschieben und modifizieren kann. Möglich wird das in der Praxis vor allem durch das Git-Kommando `rebase`. Das Kommando ist sehr mächtig und wichtig, aber zum Teil auch etwas anspruchsvoller in der Anwendung.

Rebase ist ein Kunstwort, was soviel bedeutet wie „etwas auf eine neue Basis stellen“. Gemeint ist damit, dass eine Gruppe von Commits innerhalb des Commit-Graphen verschoben, also Commit für Commit auf Basis eines anderen Knotens aufgebaut wird. Die nachfolgenden Grafiken veranschaulichen die Funktionsweise:

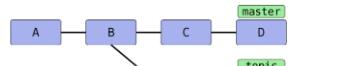


Abbildung 4.1. Vor dem Rebasing

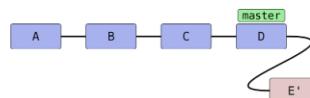


Abbildung 4.2. ... und danach

In der einfachsten Form lautet das Kommando `git rebase <referenz>` (im o.g. Diagramm: `git rebase master`). Damit markiert Git zunächst alle Commits `<referenz>..HEAD`, also die Commits, die von HEAD aus erreichbar sind (dem aktuellen Branch) abzüglich der Commits, die von `<referenz>` aus erreichbar sind – anschaulich gesprochen also alles, was im aktuellen Branch, aber nicht in `<referenz>` liegt. Im Diagramm sind das also E und F.

Die Liste dieser Commits wird zwischengespeichert. Anschließend checkt Git den Commit `<referenz>` aus und kopiert die einzelnen, zwischengespeicherten Commits in der ursprünglichen Reihenfolge als neue Commits in den Branch.

Hierbei sind einige Punkte zu beachten:

- Weil der erste Knoten des `topic`-Branches (E) nun einen neuen Vorgänger (D) hat, ändern sich seine Metadaten und somit seine SHA-1-Summe (er wird zu F). Der zweite Commit (F) hat dann ebenfalls einen anderen Vorgänger (E statt D), dessen SHA-1-Summe ändert sich (er wird zu F') usw. – dies wird auch als *Ripple Effect* bezeichnet. Insgesamt werden *alle* kopierten Commits neue SHA-1-Summen haben – sie sind also im Zweifel gleich (was die Änderungen betrifft), aber nicht identisch.
- Bei einer solchen Aktion können, genau wie bei einem Merge-Vorgang, konfliktbehaftete Änderungen auftreten. Git kann diese teilweise automatisch lösen, bricht aber mit einer entsprechenden Fehlermeldung ab, wenn die Konflikte nicht trivial sind. Der Rebasing-Prozess kann dann entweder „ repariert“ und weitergeführt oder abgebrochen werden (s.u.).
- Sofern keine weitere Referenz auf den Knoten F zeigt, geht dieser verloren, weil die Referenz `HEAD` (und gegebenenfalls die entsprechende Branch) bei einem erfolgreichen Rebasing auf den Knoten F' verschoben wird. Wenn also F keine Referenz mehr hat (und auch keine Vorgänger, die F referenzieren), kann Git den Knoten nicht mehr finden, und der Baum „verschwindet“. Wenn Sie sich nicht sicher sind, ob Sie den Original-Baum noch einmal benötigen, können Sie zum Beispiel mit dem `tag`-Kommando einfach eine Referenz darauf setzen. Dann bleiben die Commits auch nach einem Rebasing erhalten (dann aber in doppelter Form an verschiedenen Stellen im Commit-Graphen).

4.1.1. Ein Beispiel

Betrachten Sie folgende Situation: Der Branch `sqlite-support` zweigt vom Commit „fixed a bug...“ ab. Der `master`-Branch ist aber schon weitergerückt, und ein neues Release 1.4.2 ist erschienen.

```
vit 1.4.2 [master] version bump to 1.4.2
* add -O command line switch
* sqlite-support modify Makefile to support sqlite
  generalize queries
  include sqlite header files, prototypes
  fixed a bug while parsing input
```

Abbildung 4.3. Vor dem Rebasing

Nun wird `sqlite-support` ausgecheckt und neu auf `master` aufgebaut:

```
$ git checkout sqlite-support
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: include sqlite header files, prototypes
Applying: generalize queries
Applying: modify Makefile to support sqlite
```

Rebase wendet die drei Änderungen, die durch Commits aus dem Branch `sqlite-support` eingeführt werden, auf den `master`-Branch an. Danach sieht das Repository in Git wie folgt aus:

```
o  sqlite-support modify Makefile to support sqlite
  generalize queries
  include sqlite header files, prototypes
* v1.4.2 [master] version bump to 1.4.2
* add -O command line switch
  fixed a bug while parsing input
```

Abbildung 4.4. Nach Rebasing

4.1.2. Erweiterte Syntax und Konflikte

Normalerweise wird `git rebase` immer den Branch, auf dem Sie gerade arbeiten, auf einen neuen aufbauen. Allerdings gibt es eine Abkürzung: Wollen Sie `topic` auf `master` aufbauen, befinden sich aber auf einem ganz anderen Branch, können Sie das per

```
$ git rebase master topic
```

Git macht intern Folgendes:

```
$ git checkout topic
$ git rebase master
```

Beachten Sie die (leider wenig intuitive) Reihenfolge:

```
git rebase <worauf> <was>
```

Bei einem Rebasing kann es zu Konflikten kommen. Der Prozess hält dann mit folgender Fehlermeldung an:

```
$ git rebase master
...
CONFLICT (content): Merge conflict in <datei>
Patch failed at 1.
The copy of the patch that failed is found in:
.../.git/rebase-apply/patch
```

```
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".
```

Sie gehen vor wie bei einem regulären Merge-Konflikt (siehe [Abschnitt 3.4., „Merge-Konflikte lösen“](#)) – `git mergetool` ist hier sehr hilfreich. Fügen Sie dann einfach die geänderte Datei per `git add` hinzu und lassen Sie den Prozess per `git rebase --continue` weiterlaufen.^[54]

Alternativ lässt sich der problematische Commit auch überspringen, und zwar mit dem Kommando `git rebase --skip`. Der Commit ist dann aber verloren, sofern er nicht in einem anderen Branch irgendwo referenziert wird! Sie sollten diese Aktion also nur ausführen, wenn Sie sicher wissen, dass der Commit obsolet ist.

Wenn das alles nicht weiterhilft (Sie z.B. den Konflikt nicht an der Stelle lösen können oder gemerkt haben, dass Sie gerade den falschen Baum umbauen), ziehen Sie die Notbremse: `git rebase --abort`. Dies verwirft alle Änderungen am Repository (auch schon erfolgreich kopierte Commits), so dass der Zustand danach genau so ist, wie zu dem Zeitpunkt, als der Rebasing-Prozess gestartet wurde. Das Kommando hilft auch, wenn Sie irgendwann vergessen haben, einen Rebasing-Prozess zu Ende zu führen, und sich andere Kommandos beschweren, dass sie ihre Arbeit nicht verrichten können, weil gerade ein Rebasing im Gang ist.

4.1.3. Warum Rebasing sinnvoll ist

Rebase ist vor allem sinnvoll, um die Commit-Geschichte eines Projekts einfach und leicht verständlich zu halten. Beispielsweise arbeitet ein Entwickler an einem Feature, hat dann aber ein paar Wochen lang etwas anderes zu tun. Währenddessen ist die Entwicklung im Projekt aber schon weiter vorangeschritten, es gab ein neues Release etc. Erst jetzt kommt der Entwickler dazu, ein Feature zu beenden. (Auch wenn Sie Patches per E-Mail verschicken wollen, hilft Rebasing, Konflikte zu vermeiden, siehe dazu [Abschnitt 5.9., „Patches per E-Mail“](#).)

Für die Versionsgeschichte ist es nun viel logischer, wenn sein Feature nicht über einen langen Zeitraum ununterbrochen die Entwicklung „mitgeschleppt“ wurde, sondern wenn die Entwicklung vom letzten stabilen Release abweigt.

Für genau diese Änderung in der Geschichte ist Rebasing gut: Der Entwickler kann nun einfach auf seinem Branch, auf dem er das Feature entwickelt hat, das Kommando `git rebase v1.4.2` eingeben, um seinen Feature-Branch neu auf dem Commit mit dem Release-Tag `v1.4.2` aufzubauen. So lässt sich wesentlich leichter ablesen, welche Unterschiede das Feature wirklich in die Software einbringt.

Auch passiert es jedem Entwickler im Eifer des Gefechts, dass Commits im falschen Branch landen. Da findet sich zufällig ein Fehler, der schnell durch einen entsprechenden Commit behoben wird; aber dann muss direkt noch ein Test geschrieben werden, um diesen Fehler in Zukunft zu vermeiden (ein weiterer Commit), was wiederum in der Dokumentation entsprechend zu vermerken ist. Nachdem die eigentliche Arbeit getan ist, kann man diese Commits mit Rebasing an eine andere Stelle im Commit-Graphen „transplantieren“.

Rebase kann auch dann sinnvoll sein, wenn in einem Branch ein Feature benötigt wird, das erst kürzlich in die Software eingeflossen ist. Ein Merge des `master`-Branches ist semantisch nicht sinnvoll, da dann diese und andere Änderungen un trennbar mit dem Feature-Branch verschmolzen werden. Stattdessen baut man den Branch per Rebasing auf einen neuen Commit auf, in dem das benötigte Feature schon enthalten ist, und kann dieses dann in der weiteren Entwicklung nutzen.

4.1.4. Wann Rebasing *nicht* sinnvoll ist – Rebasing vs. Merge

Das Konzept von Rebasing ist zunächst etwas schwierig zu verstehen. Aber sobald Sie verstanden haben, was damit möglich ist, stellt sich die Frage: Wozu braucht man überhaupt noch ein simples Merge, wenn man doch alles mit Rebasing bearbeiten kann?

Wenn Git-Rebasing nicht oder kaum angewendet wird, entwickelt sich häufig eine Projektgeschichte, die relativ unüberschaubar wird, weil ständig und jeweils für wenige Commits Merges ausgeführt werden müssen.

Wird Rebasing dagegen zu viel angewendet, besteht die Gefahr, dass das gesamte Projekt sinnlos linearisiert wird: Das flexible Branching von Git wird zwar zur Entwicklung genutzt, die Branches werden aber dann reißverschlussartig per Rebasing hintereinander(!) in den Veröffentlichungsbranch integriert. Das stellt uns vor allem vor zwei Probleme:

- Logisch zusammengehörige Commits sind nicht mehr als solche zu erkennen. Da alle Commits linear sind, vermischt sich die Entwicklung mehrerer Features un trennbar.
- Die Integration eines Branches kann nicht mehr ohne weiteres rückgängig gemacht werden, denn diejenigen Commits zu identifizieren, die einmal zu einem Feature-Branch gehörten, ist nur manuell möglich.

So verspielen Sie die Vorteile des flexiblen Branchings von Git. Die Schlussfolgerung ist, dass Rebasing weder zu viel noch zu wenig angewendet werden sollte. Beides macht die Projektgeschichte (auf unterschiedliche Art und Weise) unübersichtlich.

Generell fahren Sie mit den folgenden Faustregeln gut:

1. Ein Feature wird, wenn es fertig wird, per `Merge` integriert. Sinnvollerweise sollte vermieden werden, einen *Fast-Forward-Merge* zu erzeugen, damit der Merge-Commit als Zeitpunkt der Integration erhalten bleibt.
2. Während Sie entwickeln, sollten Sie häufig Rebasing benutzen (besonders interaktives Rebasing, s.u.).
3. Logisch getrennte Einheiten sollten auf getrennten Branches entwickelt werden – logisch zusammengehörige eventuell auf mehreren, die dann per Rebasing verschmolzen werden (wenn das sinnvoll ist). Die Zusammenführung logisch getrennter Einheiten erfolgt dann per Merge.

4.1.5. Ein Wort der Warnung

Wie schon angesprochen, ändern sich bei einem Rebasing zwangsläufig die SHA-1-Summen aller Commits, die „umgebaut“ werden. Wenn diese Änderungen noch nicht veröffentlicht wurden, d.h. bei einem Entwickler im privaten Repository liegen, ist das auch nicht schlimm.

Wenn aber ein Branch (z.B. `master`) veröffentlicht^[55] und später per Rebasing umgeschrieben wird, hat das unschöne Folgen für alle Beteiligten: Alle Branches, die auf `master` aufbauen, referenzieren nun die alte Kopie des mittlerweile umgeschriebenen `master`-Branches. Also muss jeder Branch wiederum per Rebasing auf den neuen `master` aufgebaut werden (wodurch sich wiederum alle Commit-IDs ändern). Dieser Effekt setzt sich fort und kann (je nachdem, wann so ein Rebasing passiert und wie viele Entwickler an dem Projekt beteiligt sind) sehr zeitaufwendig zu beheben sein (das trifft vor allem dann zu, wenn Git-Neulinge dabei sind).

Daher sollten Sie immer an folgende Regel denken:

Warnung

Bearbeiten Sie mit dem Rebasing-Kommando nur unveröffentlichte Commits!

Ausnahmen bilden Konventionen wie persönliche Branches oder `pu`. Letzterer ist ein Kürzel für *Proposed Updates* und ist in der Regel ein Branch, in dem neue, experimentelle Features auf Kompatibilität getestet werden. Auf diesen Branch baut sinnvollerweise niemand seine eigene Arbeit auf, daher kann er ohne Probleme und vorherige Ankündigung umgeschrieben werden.

Eine weitere Möglichkeit bieten private Branches, also solche, die zum Beispiel mit `<user>` starten. Trifft man die Vereinbarung, dass Entwickler auf diesen Branches eigene Entwicklung betreiben, aber ihre Features immer nur auf „offiziellen“ Branches aufbauen, dann dürfen die Entwickler ihre Branches beliebig umschreiben.

4.1.6. Code-Dopplungen vermeiden

Wird über einen langen Zeitraum an einem Feature entwickelt, und Teile des Features fließen schon in ein Mainstream-Release (z.B. per `cherry-pick`), dann erkennt das Rebasing-Kommando diese Commits und lässt sie beim Kopieren bzw. Neuaufbauen der Commits aus, da die Änderung schon in dem Branch enthalten ist.

So besteht der neue Branch nach einem Rebasing nur aus den Commits, die noch nicht in den Basis-Branch eingeflossen sind. Auf diese Weise treten Commits nicht doppelt in der Versionsgeschichte eines Projekts auf. Wäre der Branch einfach nur per Merge integriert worden, so wären mitunter die gleichen Commits mit unterschiedlichen SHA-1-Summen an verschiedenen Stellen im Commit-Graphen vorhanden.

4.1.7. Patch-Stacks verwalten

Es gibt Situationen, in denen es von einer Software eine Vanilla-Version („einfachste Version“) gibt und außerdem eine gewisse Anzahl von Patches, die darauf angewendet werden, bevor die Vanilla-Version ausgeliefert wird. Zum Beispiel baut Ihre Firma eine Software, aber vor jeder Auslieferung an den Kunden müssen (je nach Kunde) einige Anpassungen vorgenommen werden. Oder Sie haben eine Open-Source-Software im Einsatz, diese aber ein wenig an Ihre Bedürfnisse angepasst – jedes Mal, wenn nun eine neue, offizielle Version der Software erscheint, müssen Sie Ihre Änderungen neu anwenden und die Software anschließend neu bauen.^[56]

Um Patch-Stacks zu verwalten, gibt es einige Programme, die auf Git aufbauen, Ihnen aber den Komfort bieten, nicht direkt mit dem Rebbase-Kommando arbeiten zu müssen. Beispielsweise erlaubt *TopGit*^[57] Ihnen, Abhängigkeiten zwischen Branches zu definieren – wenn sich dann in einem Branch etwas ändert und andere Branches hängen davon ab, baut TopGit diese auf Wunsch neu auf. Eine Alternative zu TopGit ist *Stacked Git*^[58].

4.1.8. Rebbase einschränken mit --onto

Sie mögen sich nun gewundert haben: `git rebase <referenz>` kopiert immer alle Commits, die zwischen `<referenz>` und `HEAD` liegen. Was aber, wenn Sie nur einen Teil eines Branches umsetzen, quasi „transplantieren“ möchten? Betrachten Sie folgende Situation:

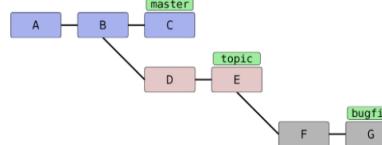


Abbildung 4.5. Vor dem `rebase --onto`

Sie haben gerade auf dem Branch `topic` ein Feature entwickelt, als Ihnen ein Fehler aufgefallen ist; Sie haben einen Branch `bugfix` erstellt und noch einen Fehler gefunden. Rein semantisch hat aber Ihr Branch `bugfix` nichts mit dem `topic`-Branch zu tun. Sinnvollerweise sollte er daher vom `master`-Branch abzweigen.

Wenn Sie nun aber per `git rebase master` den Branch `bugfix` neu aufbauen, passiert Folgendes: Alle Knoten, die in `bugfix` enthalten sind, aber nicht im `master`, werden der Reihe nach auf den `master`-Branch kopiert – das sind also die Knoten D, E, F und G. Dabei gehören jedoch D und E gar nicht zum Bugfix.

Hier kommt nun die Option `--onto` ins Spiel: Sie erlaubt, einen Start- und Endpunkt für die Liste der zu kopierenden Commits anzugeben. Die allgemeine Syntax lautet:

```
git rebase --onto <worauf> <start> <ziel>
```

In diesem Beispiel wollen wir nur die Commits F und G (oder auch: die Commits von `topic` bis `bugfix`) von oben auf `master` aufbauen. Daher lautet das Kommando:

```
$ git rebase --onto master topic bugfix
```

Das Ergebnis sieht aus wie erwartet:

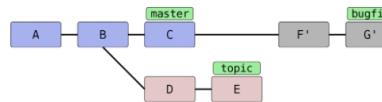


Abbildung 4.6. Nach einem `rebase --onto`

4.1.9. Einen Commit verbessern

Sie haben in [Abschnitt 2.1., „Git-Kommandos“](#) das Kommando `commit --amend` kennengelernt, mit dem Sie einen Commit verbessern. Es bezieht sich aber immer nur auf den aktuellen (letzten) Commit. Mit `rebase --onto` können Sie auch Commits anpassen, die weiter in der Vergangenheit liegen.

Suchen Sie zunächst den Commit heraus, den Sie editieren wollen, und erstellen Sie einen Branch darauf:

```
$ git checkout -b fix-master 21d8691
```

Anschließend führen Sie Ihre Änderungen aus, fügen geänderte Dateien mit `git add` hinzu und korrigieren dann den Commit mit `git commit --amend --no-edit` (die Option `--no-edit` übernimmt Meta-Informationen wie die Beschreibung des alten Commits und bietet diese nicht erneut zum Editieren an).

Nun spielen Sie alle Commits aus dem `master`-Branch von oben auf Ihren korrigierten Commit auf:

```
$ git rebase --onto fix-master 21d8691 master
```

Sie kopieren so alle Commits von `21d8691` (exklusive!) bis `master` (inklusive!). Der fehlerhafte Commit `21d8691` wird nicht mehr referenziert und taucht daher nicht mehr auf. Der Branch `fix-master` ist nun obsolet und kann gelöscht werden.

Eine äquivalente Möglichkeit, einen Commit zu editieren, haben Sie mit der Aktion `edit` im interaktiven Rebbase (siehe [Abschnitt 4.2.2., „Commits beliebig editieren“](#)).

4.1.10. Rebbase feinjustieren

Es gibt Situationen, in denen Sie das Standardverhalten von `git rebase` anpassen müssen. Erstens ist dies der Fall, wenn Sie einen Branch mit Rebbase bearbeiten, der Merges enthält. Rebbase kann versuchen, diese nachzuh主观en statt die Commits zu linearisieren. Zuständig ist die Option `-p` bzw. `--preserve-merges`.^[59]

Mit der Option `-m` bzw. `--merge` können Sie `git rebase` anweisen, Merge-Strategien zu verwenden (siehe dafür auch [Abschnitt 3.3.3., „Merge-Strategien“](#)). Wenn Sie diese Strategien anwenden, bedenken Sie, dass Rebbase intern Commit für Commit per `cherry-pick` auf den neuen Branch aufspielt; daher sind die Rollen von `ours` und `theirs` vertauscht: `theirs` bezeichnet den Branch, den Sie auf eine neue Basis aufbauen!

Ein interessanter Anwendungsfall ist daher die Strategie-Option `theirs` für die Merge-Strategie `recursive`: Falls Konflikte auftreten, wird den Änderungen aus dem Commit, der kopiert wird, Vorrang gegeben. Ein solches Szenario ist also sinnvoll, wenn Sie wissen, dass es konfliktverursachende Änderungen gibt, sich aber sicher sind, dass die Änderungen des neu aufzubauenden Branches „richtiger“ sind als die des Baumes, auf den Sie aufbauen. Wenn Sie `topic` neu auf `master` aufbauen, sahe ein solcher Aufruf so aus:

```
$ git checkout topic
$ git rebase -m -Xtheirs master
```

In den Fällen, in denen die `recursive`-Strategie (Default) den Änderungen aus Commits aus `topic` den Vorzug gibt, werden Sie einen entsprechenden Hinweis `Auto-merging <Commit-Beschreibung>` finden.

Eine kleine, sehr nützliche Option, die von Rebbase direkt an `git apply` weitergeleitet wird, ist `--whitespace=fix`. Sie veranlasst Git, automatisch Whitespace-Fehler (z.B. Trailing-Spaces) zu korrigieren. Falls Sie Merge-Konflikte aufgrund von Whitespace haben (zum Beispiel wegen geänderter Einrückung), können Sie auch die in [Abschnitt 3.3.4., „Optionen für die recursive-Strategie“](#) vorgestellten Strategie-Optionen verwenden, um automatische Lösungen zu lassen (zum Beispiel durch Angabe von `--ignore-space-change`).

4.2. Die Geschichte umschreiben – Interaktives Rebbase

Rebase kennt einen interaktiven Modus; er ist zwar technisch gleich implementiert wie der normale Modus, allerdings ist der typische Anwendungsfall ein ganz anderer, denn der interaktive Rebase erlaubt es, die Geschichte umzuschreiben, d.h. Commits beliebig zu bearbeiten (und nicht nur zu verschieben).

Im interaktiven Rebase können Sie

- die Reihenfolge von Commits verändern
- Commits löschen
- Commits miteinander verschmelzen
- einen Commit in mehrere aufteilen
- die Beschreibung von Commits anpassen
- Commits auf jede sonst erdenkliche Weise bearbeiten

Sie aktivieren den Modus durch die Option `-i` bzw. `--interactive`. Prinzipiell läuft dann der Rebase-Prozess genau so wie vorher, allerdings erhalten Sie eine Liste von Commits, die Rebase umschreiben wird, bevor das Kommando damit anfängt. Das kann zum Beispiel so aussehen:

```
pick e6ec02b6 Fix expected values of setup tests on Windows
pick 95b104c t/README: hint about using $PWD rather than $PWD in tests
pick 91c031d tests: cosmetic improvements to the repo-setup test
pick 786dabe tests: compress the setup tests
pick 4968b2e Subject: setup: officially support --work-tree without
--git-dir
```

Unter dieser Auflistung finden Sie einen Hilfertext, der beschreibt, was Sie nun mit den aufgelisteten Commit tun können. Im Wesentlichen gibt es pro Commit sechs mögliche Aktionen. Die Aktion schreibt Sie einfach statt der Standard-Aktion `pick` an den Anfang der Zeile, vor die SHA-1-Summe. Im Folgenden die Aktionen – Sie können diese auch jeweils durch ihren Anfangsbuchstaben abkürzen, also z.B. `s` für `squash`.

```
pick
    „Commit verwenden“ (Default). Entspricht der Behandlung von Commits im nicht-interaktiven Rebase.
-
    Löschen Sie eine Zeile, dann wird der Commit nicht verwendet (geht verloren).
reword
    Commit-Beschreibung anpassen.
squash
    Commit mit dem vorherigen verschmelzen; Editor wird geöffnet, um die Beschreibungen zusammenzuführen.
fixup
    Wie squash, wirft aber die Beschreibung des Commits weg.
edit
    Freies Editieren. Sie können beliebige Aktionen ausführen.
exec
    Der Rest der Zeile wird als Kommando auf der Shell ausgeführt. Falls das Kommando sich nicht erfolgreich (das heißt mit Rückgabewert 0) beendet, hält der Rebase an.
```

Die Aktion `pick` ist die simpelste – sie besagt einfach, dass Sie den Commit verwenden wollen, Rebase soll diesen Commit so, wie er ist, übernehmen. Das Gegenteil von `pick` ist das simple Löschen einer kompletten Zeile. Der Commit geht dann verloren (wie `git rebase --skip`).

Wenn Sie die Reihenfolge der Zeilen tauschen, dann wird Git die Commits in der neu definierten Reihenfolge anwenden. Zu Anfang sind die Zeilen in der Reihenfolge, in der sie später angewendet werden – also genau anders herum als in der Baumanansicht! Beachten Sie, dass Commits häufig aufeinander aufbauen; daher wird es bei der Verteilung von Commits häufig zu Konflikten kommen, sofern die Commits auf den gleichen Dateien und an den gleichen Stellen Änderungen durchführen.

Das Kommando `reword` ist praktisch, wenn Sie Tippfehler in einer Commit-Nachricht haben und diese korrigieren wollen (oder bisher keine ausführliche verfasst haben und dies nun nachholen wollen). Der Rebase-Prozess wird bei dem mit `reword` markierten Prozess angehalten, und Git startet einen Editor, in dem die Nachricht des Commits bereits angezeigt wird. Sobald Sie den Editor beenden (Speichern nicht vergessen!), wird Git die neue Beschreibung einpflegen und den Rebase-Prozess weiterlaufen lassen.

4.2.1. Kleine Fehler korrigieren: Bug Squashing

Die Kommandos `squash` bzw. `fixup` erlauben es, zwei oder mehr Commits miteinander zu verschmelzen.

Niemand schreibt immer sofort fehlerfreien Code. Häufig gibt es einen großen Commit, in dem Sie ein neues Feature implementiert haben; kurz darauf findet sich kleine Fehler. Was tun? Eine ausführliche Beschreibung, warum Sie aus Unachtsamkeit vergessen haben, eine Zeile hinzuzufügen oder zu entfernen? Nicht wirklich sinnvoll, und vor allem störend für andere Entwickler, die später Ihren Code überprüfen wollen. Schöner wäre es doch, die Illusion zu wahren, dass der Commit gleich beim ersten Mal fehlerfrei war...

Für jeden Fehler, den Sie finden, machen Sie einen kleinen Commit mit einer mehr oder weniger sinnvollen Beschreibung. Das könnte dann zum Beispiel so aussehen:

```
$ git log --oneline master..feature
b5feb7 fix feature 1
34c4453 fix feature 2
ac45c5c fix feature 1
ae65ef0 implement feature 2
cf30ef4 implement feature 1
```

Wenn sich einige solche Commits angesammelt haben, starten Sie einen interaktiven Rebase-Prozess über die letzten Commits. Schätzen Sie dazu einfach ab, auf wie vielen Commits Sie arbeiten wollen, und bearbeiten Sie dann beispielsweise per `git rebase -i HEAD~5` die letzten fünf.

Im Editor erscheinen die Commits nun in umgekehrter Reihenfolge im Vergleich zur Ausgabe von `git log`. Ordnen Sie nun die kleinen Bugfix-Commits so an, dass sie *unter* dem Commit, den sie korrigieren, stehen. Markieren Sie dann die Korrektur-Commits mit `squash` (oder `s`), also z.B. so:

```
* pick cf30ef4 implement feature 1
* ac45c5c fix feature 1
* b5feb7 fix feature 1
* ae65ef0 implement feature 2
* 34c4453 fix feature 2
```

Speichern Sie die Datei und beenden Sie den Editor; der Rebase-Prozess startet. Weil Sie `squash` ausgewählt haben, hält Rebase an, nachdem Commits verschmolzen wurden. Im Editor erscheinen die Commit-Nachrichten der verschmolzenen Commits, die Sie nun geeignet zusammenfassen. Verwenden Sie statt `squash` das Schlüsselwort `fixup` oder kurz `t`, wird die Commit-Nachricht der so markierten Commits weggeworfen – für diese Arbeitsweise also vermutlich praktischer.

Nach dem Rebase sieht die Versionsgeschichte viel aufgeräumter aus:

```
$ git log --oneline master..feature
97fe253 implement feature 2
6329a9a implement feature 1
```

Tipp

Oft kommt es vor, dass man eine kleine Änderung noch in den zuletzt getätigten Commit „schleusen“ möchte. Hier bietet sich folgendes Alias an, das an die `fixup`-Aktion angelehnt ist:

```
$ git config --global alias.fixup "commit --amend --no-edit"
```

Wie oben schon erwähnt, übernimmt die Option `--no-edit` eins zu eins die Metainformationen des alten Commits, insbesondere die Commit-Message.

Wenn Sie die Commit-Nachricht mit `fixup!` bzw. `squash!` beginnen, gefolgt vom Anfang der Beschreibung des Commits, den Sie korrigieren wollen, können Sie das Kommando

```
$ git rebase -i --autosquash master
```

aufrufen. Die wie oben mit `fixup!` bzw. `squash!` markierten Commits werden automatisch an die richtige Stelle verschoben und mit der Aktion `squash` bzw. `fixup` versehen. So können Sie den Editor direkt beenden, und die Commits werden verschmolzen. Falls Sie häufig mit dieser Option arbeiten, können Sie dieses Verhalten durch eine Konfigurationsoption zum Standard bei Rebases-Aufrufen machen: Setzen Sie dafür die Einstellung `rebase.autosquash` auf `true`.

4.2.2. Commits beliebig editieren

Wenn Sie einen Commit mit `edit` markieren, kann er beliebig editiert werden. Dabei geht Rebbase wie in den anderen Fällen auch sequentiell die Commits durch. Bei den Commits, die mit `edit` markiert sind, hält Rebbase an und `HEAD` wird auf den entsprechenden Commit gesetzt. Sie können dann den Commit ändern, als wäre er der aktuellste im Branch. Anschließend lassen Sie Rebbase weiterlaufen:

```
$ vim ...
# Rebase-Kürtze vornehmen
$ git add .
$ git commit --amend
$ git rebase --continue
```

Im Wesentlichen erreichen Sie dabei dasselbe wie im Beispiel `git rebase --onto` in [Abschnitt 4.1.9, „Einen Commit verbessern“](#) – allerdings können Sie auch weit kompliziertere Aktionen ausführen. Einen häufigen Anwendungsfall beschreibt folgendes „Rezept“.

Commits aufteilen

Jeder Programmierer kennt das: Diszipliniert und penibel jede Änderung einzuchecken, ist anstrengend und unterbricht häufig den Arbeitsfluss. Das führt in der Praxis zu Commits, die groß und unübersichtlich sind. Damit die Versionsgeschichte aber für andere Entwickler – und Sie selbst! – nachvollziehbar bleibt, sollten die Änderungen in so kleine logische Einheiten wie möglich aufgeteilt werden.

Im Übrigen ist es nicht nur für Entwickler hilfreich, so vorzugehen. Auch die automatisierte Fehlersuche mittels `git bisect` funktioniert besser und akkurater, je kleiner und sinnvoller die Commits sind (siehe [Abschnitt 4.8, „Regressionen finden – git bisect“](#)).

Mit ein wenig Erfahrung können Sie einen Commit sehr schnell aufteilen. Wenn Sie häufig große Commits produzieren, sollte Ihnen der folgende Arbeitsschritt zur Routine werden.

Zunächst starten Sie den Rebbase-Prozess und markieren den Commit, den Sie aufteilen wollen, mit `edit`. Rebbase hält dort an, `HEAD` zeigt auf diesen Commit.

Anschließend setzen Sie den `HEAD` einen Commit zurück, ohne allerdings die Änderungen von `HEAD` (der aufzuteilende Commit) wegzuvorwerfen. Das passiert durch das Kommando `reset` (siehe auch [Abschnitt 3.2.3, „Reset und der Index“](#); beachten Sie, dass, sofern Sie die Commit-Beschreibung noch brauchen, Sie diese vorher kopieren sollten):

```
$ git reset HEAD^
```

Die Änderungen, die der aufzuteilende Commit verursacht, sind nun noch in den Dateien vorhanden; der Index und das Repository spiegeln aber den Stand des Vorgänger-Commits wider. Sie haben also die Änderungen des aufzuteilenden Commits in den *unstaged*-Zustand verschoben (das können Sie verifizieren, indem Sie `git diff` vor und nach dem `reset`-Aufruf betrachten).

Nun können Sie einige Zeilen hinzufügen, einen Commit erstellen, weitere Zeilen hinzufügen und schließlich einen dritten Commit für die übrigen Zeilen erstellen:

```
$ git add -p
$ git commit -m "Erster Teil"
$ git add -p
$ git commit -m "Zweiter Teil"
$ git add -u
$ git commit -m "Dritter (und letzter) Teil"
```

Was passiert? Durch das `Reset`-Kommando haben Sie den `HEAD` einen Commit zurückgesetzt. Mit jedem Aufruf von `git commit` erstellen Sie einen neuen Commit, aufbauend auf dem jeweiligen `HEAD`. Statt eines großen Commits (den Sie durch den `reset`-Aufruf weggeworfen haben) haben Sie nun drei kleinere Commits an dessen Stelle gesetzt.

Lassen Sie jetzt Rebbase weiterlaufen (`git rebase --continue`) und die übrigen Commits von oben auf `HEAD` (der jetzt der neueste Ihrer drei Commits ist) aufzubauen.

4.3. Wer hat diese Änderungen gemacht? – git blame

Wie andere Versionskontrollsysteme hat auch Git ein Kommando `blame` bzw. `annotate`, das alle Zeilen einer Datei mit Datum und Autor der letzten Änderung versieht. So können Sie z.B. schnell herausfinden, wer der Verantwortliche für eine Zeile Code ist, die ein Problem verursacht, oder seit wann das Problem besteht.

Da bei ist das Kommando `annotate` lediglich für Umsteiger gedacht und hat die gleiche Funktionalität wie das Kommando `blame`, nur ein etwas anderes Ausgabeformat. Sie sollten also im Zweifel immer `blame` verwenden.

Nützliche Optionen sind `-M`, um Code-Verschiebungen, und `-c`, um Code-Kopien anzuzeigen. Anhand des Dateinamens in der Ausgabe können Sie dann erkennen, aus welcher Datei möglicherweise Code kopiert oder verschoben wurde. Wird kein Dateiname angezeigt, konnte Git keine Code-Bewegungen oder -Kopien finden. Wenn Sie diese Optionen verwenden, ist es meist sinnvoll, per `-s` die Angabe von Autor und Datum zu unterdrücken, damit die Anzeige noch ganz auf den Bildschirm passt.

Aus der folgenden Ausgabe erkennt man z.B., dass die Funktion `end_url_with_slash` ursprünglich aus der Datei `http.c` stammte. Die Option `-L<m>,<n>` grenzt die Ausgabe auf die entsprechenden Zeilen ein.

```
$ git blame -C -s -L123,135 url.c
638794cd url.c 123) char *url_decode_parameter_value(const char
**query)
638794cd url.c 124) {
ce83ed url.c 125)     struct strbuf out = STRBUF_INIT;
730220d url.c 126)     return url_decode_internal(query, "&", &out,
1)
638794cd url.c 127) }
d7e92806 http.c 128)
eb9d47cf http.c 129) void end_url_with_slash(struct strbuf *buf, const
char *url)
5ace994f http.c 130) {
5ace994f http.c 131)     strbuf_addstr(buf, url);
5ace994f http.c 132)     if (buf->len && buf->buf[buf->len - 1] != '/')
5ace994f http.c 133)         strbuf_addstr(buf, "/");
5ace994f http.c 134) }
5793a309 url.c 135)
```

4.3.1. Blame grafisch

Eine bequeme Alternative zu `git blame` auf der Konsole bietet das grafische Tool `git gui blame` (hierfür müssen Sie gegebenenfalls das Paket `git-gui` installieren).

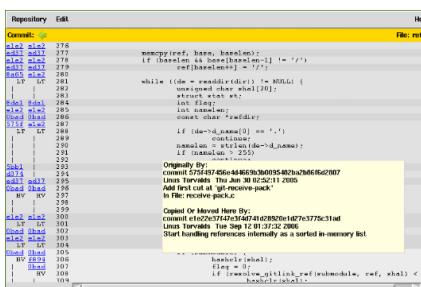


Abbildung 4.7: Ein Stück Code, das aus einer anderen Datei verschoben wurde

Wenn Sie eine Datei per `git gui blame <datei>` untersuchen, werden die unterschiedlichen Blöcke,

die aus verschiedenen Commits stammen, mit Grautönen hinterlegt dargestellt. Links sehen Sie die abgekürzte Commit-ID sowie die Initialen des Autors.

Erst wenn Sie mit der Maus über einen solchen Block fahren, erscheint ein kleines Popup-Fenster mit Informationen zum Commit, der die Zeilen geändert hat, möglicherweise auch mit einer Mitteilung, aus welcher Datei und welchem Commit dieser Codeblock verschoben oder kopiert wurde.

Bei der Code-Review interessiert man sich häufig dafür, wie eine Datei eigentlich vor einer bestimmten Änderung aussah. Dafür bietet das grafische Blame-Tool die folgende Möglichkeit, in der Versionsgeschichte zurückzugehen: Klicken Sie mit der rechten Maustaste auf die Commit-ID eines Code-Blocks und wählen Sie im Kontextmenü *Blame Parent Commit* aus – nun wird der Vorgänger dieser Änderung angezeigt. Sie können auf diese Weise mehrere Schritte zurückgehen. Über den grünen Pfeil links oben können Sie wieder in die Zukunft springen.

4.4. Dateien ignorieren

In fast jedem Projekt fallen Dateien an, die Sie nicht versionieren wollen. Sei es der binäre Output des Compilers, die autogenierte Dokumentation im HTML-Format oder die Backup-Dateien, die Ihr Editor erzeugt. Git bietet Ihnen verschiedene Ebenen, um Dateien zu ignorieren:

- benutzerspezifische Einstellung
- repositoryspezifische Einstellung
- repositoryspezifische Einstellung, die mit eingecheckt wird

Welche Option Sie wählen, hängt ganz von Ihrem Anwendungsfall ab. Die benutzerspezifischen Einstellungen sollten Dateien und Muster enthalten, die für den Benutzer relevant sind, beispielsweise Backup-Dateien, die Ihr Editor erzeugt. Solche Muster werden üblicherweise in einer Datei im `$HOME`-Verzeichnis abgelegt. Mit der Option `core.excludesfile` geben Sie an, welche Datei dies sein soll, z.B. im Fall von `~/.gitignore`:

```
s git config --global core.excludesfile ~/.gitignore
```

Bestimmte Dateien und Muster sind an ein Projekt gebunden und gelten für jeden Teilnehmer, z.B. Compiler-Output und autogenierte HTML-Dokumentation. Diese Einstellung legen Sie in der Datei `.gitignore` ab, die Sie ganz normal einchecken und somit an alle Entwickler ausliefern.

Zuletzt lässt sich die Datei `.git/info/exclude` für repositoryspezifische Einstellungen nutzen, die nicht mit einem Klon ausgeliefert werden sollen, also Einstellungen, die gleichzeitig projekt- und benutzerspezifisch sind.

4.4.1. Syntax für Muster

Die Syntax für Muster ist der Shell-Syntax nachempfunden:

- Leere Zeilen haben keinen Effekt und können zum Gliedern und Trennen verwendet werden.
- Zeilen, die mit einem `#` anfangen, werden als Kommentare gewertet und haben ebenfalls keinen Effekt.
- Ausdrücke, die mit `!` anfangen, werden als Negation gewertet.
- Ausdrücke, die mit einem `/` enden, werden als Verzeichnis gewertet. Der Ausdruck `man/` erfasst das Verzeichnis `man`, nicht aber die gleichnamige Datei oder den Symlink.
- Ausdrücke, die `kein / enthalten`, werden als Shell-Glob für das aktuelle und alle Unterverzeichnisse gewertet. Der Ausdruck `*.zip` in der obersten `.gitignore` etwa erfasst alle Zip-Dateien in der Verzeichnissstruktur des Projekts.
- Der Ausdruck `**` umfasst Null oder mehr Dateien und Verzeichnisse. Sowohl `t/data/set1/store.txt` als auch `t/README.txt` werden durch das Muster `t/**/*txt` erfasst.
- Sonst wird das Muster als Shell-Glob gewertet, genauer als Shell-Glob, das von der Funktion `fnmatch(3)` mit dem Flag `FN_PATHNAME` ausgewertet wird. Das heißt, das Muster `doc/*html` erfasst `doc/index.html`, nicht aber `doc/api/singleton.html`.
- Ausdrücke, die mit einem `/` beginnen, sind an den Pfad gebunden. Der Ausdruck `*/.sh` zum Beispiel erfasst `upload.sh`, nicht aber `scripts/check-for-error.sh`.

Ein Beispiel:[60]

```
s cat ~/.gitignore
# vim swap files
*.sw[nop]
*.pyc

# python bytecode
*.pyc

# documents
*.dvi
*.pdf

# miscellaneous
*.out
```

4.4.2. Nachträglich ignorieren oder versionieren

Dateien, die bereits versioniert sind, werden nicht automatisch ignoriert. Um eine solche Datei trotzdem zu ignorieren, weisen Sie Git explizit an, die Datei zu „vergessen“:

```
s git rm documentation.pdf
```

Um die Datei mit dem nächsten Commit zu löschen, aber trotzdem im Working Tree vorzuhalten:

```
s git rm --cached documentation.pdf
```

Dateien, die bereits ignoriert werden, erscheinen in der Ausgabe von `git status` nicht. Außerdem weigert sich `git add`, die Datei zu übernehmen; mit der Option `--force` bzw. `-f` zwingen Sie Git, die Datei doch zu beachten:

```
s git add documentation.pdf
The following paths are ignored by one of your .gitignore files:
documentation.pdf
Use -f if you really want to add them.
fatal: no files added
s git add -f documentation.pdf
```

4.4.3. Ignorierte und unbekannte Dateien löschen

Das Kommando `git clean` löscht ignorierte sowie unbekannte (sog. *untracked*) Dateien. Da evtl. Dateien unvorderbarlich verlorengehen könnten, verfügt das Kommando über die Option `--dry-run` (bzw. `-n`); sie gibt Auskunft, was gelöscht würde. Als weitere Vorsichtsmaßnahme weigert sich das Kommando, irgendetwas zu löschen, außer Sie übergeben explizit die Option `--force` bzw. `-f`.[61]

Standardmäßig löscht `git clean` nur die unbekannten Dateien, mit `-x` entfernt es nur die ignorierten Dateien und mit `-x` sowohl unbekannte als auch ignorierte. Mit der Option `-d` werden zusätzlich Verzeichnisse gelöscht, die in Frage kommen. Um also unbekannte sowie ignorierte Dateien und Verzeichnisse zu löschen, geben Sie ein:

```
s git clean -dfx
```

4.5. Veränderungen auslagern – git stash

Der Stash (Lager) ist ein Mechanismus, der dazu dient, noch nicht gespeicherte Veränderungen am Working Tree kurzfristig auszulagern. Ein klassischer Anwendungsfall: Ihr Chef bittet Sie, so schnell wie möglich einen kritischen Bug zu beheben. Sie haben aber gerade angefangen, ein neues Feature zu implementieren. Mit dem Kommando `git stash` räumen Sie die unfertigen Zeilen vorübergehend „aus dem Weg“, ohne einen Commit zu erzeugen, und können sich so mit einem sauberen Working Tree dem Fehler zuwenden. Außerdem bietet der Stash Abhilfe, wenn Sie den Branch nicht wechseln können, weil dadurch Veränderungen verlorengehen würden (siehe auch [Abschnitt 3.1.2 „Branches verwalten“](#)).

4.5.1. Grundlegende Benutzung

Mit `git stash` speichern Sie den aktuellen Zustand von Working Tree und Index, sofern diese sich von `HEAD` unterscheiden:

```
s git stash
```

```
 Saved working directory and index state WIP on master: b529e34 new spec  
 how the script should behave  
 HEAD is now at b529e34 new spec how the script should behave
```

Mit der Option `--keep-index` bleibt der Index intakt. Das heißt, alle Veränderungen die bereits im Index sind, bleiben im Working Tree und im Index vorhanden und werden zusätzlich im Stash gespeichert.

Die Veränderungen am Working Tree sowie dem Index werden „beiseite geschafft“, und Git erzeugt keinen Commit auf dem aktuellen Branch. Um den gespeicherten Zustand wieder herzustellen, also um den gespeicherten Patch auf dem aktuellen Working Tree anzuwenden und gleichzeitig den Stash zu löschen, verwenden Sie:

```
 $ git stash pop  
 ...  
 Dropped refs/stash@{0} (d4cc94c37e92390e5fabf184a3b5b7ebd5c3943a)
```

Sie können zwischen dem Abspeichern und dem Wiederherstellen das Repository beliebig verändern, z.B. den Branch wechseln, Commits machen usw. Der Stash wird immer auf den aktuellen Working Tree angewendet.

Das Kommando `git stash pop` ist eine Abkürzung für die zwei Kommandos `git stash apply` (Stash anwenden) und `git stash drop` (Stash verwerfen):

```
 $ git stash apply  
 .  
 $ git stash drop  
 Dropped refs/stash@{0} (d4cc94c37e92390e5fabf184a3b5b7ebd5c3943a)
```

Sowohl `pop` als auch `apply` pflegen die Veränderungen in den Working Tree ein, der Zustand des Index wird nicht wieder hergestellt. Mit der Option `--index` stellen Sie auch den abgespeicherten Zustand des Index wieder her.

Tipp

Mit der Option `--patch` (bzw. kurz `-p`) starten Sie einen interaktiven Modus, d.h. Sie können wie mit `git add -p` und `git reset -p` einzelne Hunks auswählen, um sie dem Stash hinzuzufügen:

```
 $ git stash -p
```

Die Konfigurationseinstellung `interactive.singlekey` (siehe [Abschnitt 2.1.2, „Commits schrittweise erstellen“](#)) gilt auch hier.

4.5.2. Konflikte lösen

Es kann zu Konflikten kommen, wenn Sie einen Stash auf einem anderen Commit anwenden als dem, auf dem er entstanden ist:

```
 $ git stash pop  
 Auto-merging hello.pl  
 CONFLICT (content): Merge conflict in hello.pl
```

In dem Fall verwenden Sie die üblichen Rezepte zum Lösen des Konflikts, siehe [Abschnitt 3.4, „Merge-Konflikte lösen“](#). Wichtig ist aber, dass die Konflikt-Marker die Bezeichnungen `Updated Upstream` (die Version im aktuellen Working Tree) sowie `Stashed Changes` (Veränderungen im Stash) tragen:

```
<<<< Updated upstream  
 # E-Mail: valentin.haenel@gmx.de  
 =====  
 # E-Mail: valentin@gitbu.ch  
>>>> Stashed changes
```

Wichtig

Sollten Sie versucht haben, einen Stash mit `git stash pop` anzuwenden, wird der Stash *nicht* automatisch gelöscht. Sie müssen ihn explizit mit `git stash drop` löschen.

4.5.3. Wenn Sie den Stash nicht anwenden können...

Der Stash wird per Default auf den aktuellen Working Tree angewendet, vorausgesetzt dieser ist sauber – wenn nicht, bricht Git ab:

```
 $ git stash pop  
 Cannot apply to a dirty working tree, please stage your changes
```

Git schlägt zwar vor, dass Sie die Änderungen dem Index hinzufügen, wie Sie aber vorgehen sollten, hängt von Ihrem Ziel ab. Wenn Sie die Änderungen im Stash zusätzlich zu denen im Working Tree haben wollen, bietet sich Folgendes an:

```
 $ git add -u  
 $ git stash pop  
 $ git reset HEAD
```

Zur Erläuterung: Zuerst werden die noch nicht gespeicherten Veränderungen am Working Tree dem Index hinzugefügt; dann die Veränderungen aus dem Stash herausgeholt und auf den Working Tree angewendet, und zuletzt noch der Index zurückgesetzt.

Alternativ dazu können Sie auch einen zusätzlichen Stash erstellen, und die Veränderungen, die Sie haben wollen, auf einen sauberen Working Tree anwenden:

```
 $ git stash  
 $ git stash apply stash@{1}  
 $ git stash drop stash@{1}
```

Bei diesem Rezept verwenden Sie mehrere Stashes. Zuerst lagern Sie die Veränderungen am Working Tree in einen neuen Stash aus, dann holen Sie die Veränderungen, die Sie eigentlich haben wollen, aus dem vorherigen Stash und löschen diesen nach der Anwendung.

4.5.4. Nachricht anpassen

Standardmäßig setzt Git für einen Stash die folgende Nachricht:

```
 WIP: on <branch>: <sha1> <commit-msg>
```

```
<branch>  
 der aktuelle Branch  
<sha1>  
 die Commit-ID des HEAD  
<commit-msg>  
 die Commit-Nachricht des HEAD
```

Meist reicht dies aus, um einen Stash zu identifizieren. Wenn Sie vorhaben, Ihre Stashes länger vorzuhalten (möglich, aber nicht wirklich zu empfehlen), oder wenn Sie mehrere machen wollen, raten wir, diese mit einer besseren Anmerkung zu versehen:

```
 $ git stash save "unfertiges Feature"  
 Saved working directory and index state On master: unfertiges feature  
 HEAD is now at b529e34 new spec how the script should behave
```

4.5.5. Stashes einsehen

Git verwaltet alle Stashes als Stack, d.h. aktuellere Zustände liegen oben auf und werden zuerst verarbeitet. Die Stashes sind mit einer Reflog-Syntax (siehe auch [Abschnitt 3.7, „Reflog“](#)) benannt:

```
 stash@{0}  
 stash@{1}  
 stash@{2}  
 ...
```

Erzeugen Sie einen neuen Stash, wird dieser als `stash@{0}` bezeichnet und die Nummer der anderen wird inkrementiert: Aus `stash@{0}` wird `stash@{1}`, aus `stash@{1}` wird `stash@{2}` usw.

Geben Sie keinen expliziten Stash an, beziehen sich die Kommandos `apply`, `drop` und `show` auf den neuesten, also `stash@{0}`.

Um einzelne Stashes einzusehen, verwenden Sie `git stash show`. Standardmäßig drückt dieses Kommando eine Bilanz der hinzugefügten und entfernten Zeilen aus (wie `git diff --stat`):

```
$ git stash show  
git-stats.sh | 4 +++  
1 files changed, 2 insertions(+), 2 deletions(-)
```

Tipp

Das Kommando `git stash show` akzeptiert zusätzlich allgemeine Diff-Optionen, die das Format beeinflussen, z.B. `-p`, um ein Patch im Diff-Format auszugeben:

```
$ git stash show -p stash@{0}  
diff --git a/git-stats.sh b/git-stats.sh  
index 62f92fe..1235fd3 100755  
--- a/git-stats.sh  
+++ b/git-stats.sh  
@@ -1,6 +1,6 @@  
#!/bin/bash  
-START=18.07.2010  
+END=25.07.2010  
+START=18.07.2000  
+END=25.07.2020  
echo "Number of commits per author:"
```

Das Kommando `git stash list` gibt eine Liste der derzeit angelegten Stashes aus:

```
$ git stash list  
stash@{0}: WIP on master: eae23b6 add number of merge commits to output  
stash@{1}: WIP on master: blee2cf start and end date in one place only
```

4.5.6. Stashes löschen

Einzelne Stashes löschen Sie mit dem Kommando `git stash drop`, alle mit `git stash clear`. Sollten Sie versehentlich einen Stash löschen, finden Sie diesen nicht über die üblichen Reflog-Mechanismen wieder! Jedoch gibt folgender Befehl die ehemaligen Stashes aus:^[62]

```
$ git fsck --unreachable | grep commit | cut -d" " -f3 | \  
xargs git log --merges --no-walk --grep=WIP
```

Tipp

Für den Notfall merken Sie sich, dass Sie den Befehl ganz am Ende der Man-Page von `git-stash(1)` finden.

Außerdem ist wichtig, dass die so gezeigten Einträge nur als unerreichbare Objekte in der Objektdatenbank vorhanden sind und somit auch den normalen Wartungsmechanismen unterliegen – sie werden also nach einiger Zeit gelöscht und nicht dauerhaft vorgehalten.

4.5.7. Wie ist der Stash implementiert?

Git erzeugt für jeden Stash zwei Commit-Objekte, eines für die Veränderungen am Working Tree und eines für die Veränderungen am Index. Beide haben den aktuellen `HEAD` als Vorfahren, das Working-Tree-Objekt hat das Index-Objekt als Vorfahren. Dadurch wird ein Stash in Git als Dreieck angezeigt, was im ersten Moment etwas verwirrend ist:

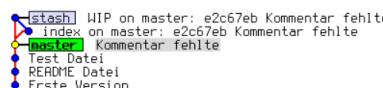


Abbildung 4.8. Ein Stash in Git

Mit dem Alias `git tree` (siehe [Abschnitt 3.6.1. „Revision Parameters“](#)) sieht das so aus:

```
* f1fda63 (refs/stash) WIP on master: e2c67eb Kommentar fehlte  
| * 4faee09 index on master: e2c67eb Kommentar fehlte  
|/  
* e2c67eb (HEAD, master) Kommentar fehlte  
* 8e2ff5f9 Test Datei  
* 308aa1 REHOME Datei  
* b0400b0 Erste Version
```

Da die Stash-Objekte nicht durch einen Branch referenziert sind, wird das Working-Tree-Objekt mit einer besonderen Referenz, `refs/stash`, am Leben erhalten. Dies gilt aber nur für den neuesten Stash. Ältere Stashes werden nur im Reflog (siehe [Abschnitt 3.7. „Reflog“](#)) referenziert und erscheinen deshalb auch nicht in Gitk. Im Gegensatz zu normalen Reflog-Einträgen verfallen gespeicherte Stashes jedoch nicht und werden deshalb auch nicht durch die normalen Wartungsmechanismen gelöscht.

4.6. Commits annotieren – git notes

In der Regel ist es nicht ohne Weiteres möglich, Commits, die einmal veröffentlicht wurden, noch einmal zu ändern oder zu erweitern. Manchmal wünscht man sich jedoch, man könnte Commits im Nachhinein noch Informationen „anhängen“, ohne dass der Commit sich ändert. Das könnten Ticket-Nummern sein, Informationen darüber, ob die Software kompiliert, wer sie getestet hat usw.

Git bietet mit dem Kommando `git notes` eine Möglichkeit, nachträglich Notizen an einen Commit zu heften. Dabei sind die Notizen ein abgekoppelter „Branch“ von Commits, referenziert durch `refs/notes/commits`, auf dem die Entwicklung der Notes gespeichert wird. Auf diesem Branch liegen die Notizen zu einem Commit in einer Datei vor, deren Dateiname der SHA-1-Summe des Commits entspricht, den sie beschreibt.

Diese Interna können Sie aber außer Acht lassen – in der Praxis können Sie die Notizen komplett mit `git notes` verwalten. Wichtig ist nur im Wissen: Pro Commit können Sie nur eine Notiz speichern.^[63] Dafür können Sie die Notizen aber im Nachhinein editieren bzw. erweitern.

Um eine neue Notiz hinzuzufügen: `git notes add <commit>`. Wenn Sie `<commit>` auslassen, wird `HEAD` verwendet. Analog zu `git commit` öffnet sich ein Editor, in dem Sie die Notiz verfassen. Alternativ können Sie diese direkt per `-m "<notiz>"` angeben.

Die Notiz wird dann per Default immer unter der Commit-Nachricht angezeigt:

```
$ git show 0e8a7c1f  
commit 0e8a7c1f4ca6a024acde03e58c2b67fa901f88  
Author: Julius Plenz <julius@plenz.com>  
Date: Sun May 22 15:48:46 2011 +0200  
  
        Schleife optimieren  
  
Notes:  
        Dies verursacht Bug #2319 und wird mit v2.1.3-7-g6dfa88a korrigiert
```

Mit der Option `--no-notes` können Sie Kommandos wie `log` oder `show` explizit anweisen, Notizen nicht anzuzeigen.

Das Kommando `git notes add` beendet sich mit einem Fehler, wenn zu dem angegebenen Commit schon eine Notiz vorliegt. Verwenden Sie dann stattdessen das Kommando `git notes append`, um weitere Zeilen an die Notiz anzuhängen, oder aber direkt `git notes edit`, um die Notiz beliebig zu editieren.

Per Default werden die Notizen nicht hoch- oder runtergeladen, Sie müssen das explizit über die folgenden Kommandos tun:

```
$ git push <remote> refs/notes/commits  
$ git fetch <remote> refs/notes/commits:refs/notes/commits
```

Das Notizen-Konzept ist in Git nicht besonders weit entwickelt. Insbesondere macht es Probleme, wenn mehrere Entwickler parallel Notizen zu Commits erstellen und diese dann zusammengeführt werden müssen. Für weitere Informationen siehe die Man-Page `git-notes(1)`.

Tipp

Wenn Sie Notizen verwenden wollen, bietet sich dies meist nur im Zusammenhang mit Ticket-, Bug-Tracking- oder Continuous-Integration-Systemen an: Diese können automatisiert Notizen erstellen und so möglicherweise hilfreiche Zusatzinformationen im Repository ablegen.

Um die Notizen bei jedem `git fetch` automatisch herunterzuladen, fügen Sie eine Refspec der folgenden Form in die Datei `.git/config` ein (siehe auch [Abschnitt 5.3.1, „git fetch“](#)):

```
fetch = +refs/notes/*:refs/notes/*
```

4.7. Mehrere Root-Commits

Bei der Initialisierung eines Repositorys wird der erste Commit, der sogenannte Root-Commit, erstellt. Dieser Commit ist in der Regel der einzige im ganzen Repository, der keinen Vorgänger hat.

Allerdings ist es auch möglich, mehrere Root-Commits in einem Repository zu haben. Das kann in den folgenden Fällen sinnvoll sein:

- Sie wollen zwei eigenständige Projekte miteinander verbinden, die vorher in getrennten Repositories entwickelt wurden (siehe dafür auch Subtree-Merges in [Abschnitt 5.11.2, „Subtrees“](#)).
- Sie wollen einen vollständig abgekoppelten Branch verwälten, auf dem Sie eine Todo-Liste vorhalten, komplizierte Binaries oder autogenerierte Dokumentation.

Im Falle, dass Sie zwei Repositories zusammenführen wollen, reicht in der Regel dieses Kommando:

```
$ git fetch -n <anderes-repo> master:<anderer-master>
warning: no common commits
...
>From <anderes-repo>
 * [new branch]      master    -> <anderer-master>
```

Der Branch `master` des anderen Repositorys wird als `<anderer-master>` ins lokale Repository kopiert, inklusive aller Commits, bis Git eine Merge-Basis findet oder einen Root-Commit. Die Warnung „no common commits“ deutet schon darauf hin, dass die beiden Versionsgeschichten keinen gemeinsamen Commit haben. Das Repository hat nun zwei Root-Commits.

Beachten Sie, dass ein Merge zwischen zwei Branches, die keine gemeinsamen Commits haben, fehlschlagen wird, sobald eine Datei auf beiden Seiten existiert und nicht gleich ist. Abhilfe schaffen hier möglicherweise Subtree-Merges, siehe [Abschnitt 5.11.2, „Subtrees“](#).

Sie können aber auch, anstatt ein anderes Repository zu importieren, einen komplett abgekoppelten Branch neu erstellen, also einen zweiten Root-Commit. Dafür reichen die folgenden beiden Kommandos aus:

```
$ git checkout --orphan <newroot>
$ git rm --cached -rf .
```

Das erste setzt den `HEAD` auf den (noch nicht existierenden) Branch `<newroot>`. Das `rm`-Kommando löscht alle von Git verwalteten Dateien aus dem Index, lässt sie aber im Working Tree intakt. Sie haben nun also einen Index, der nichts enthält, und einen Branch, auf dem noch kein Commit existiert.

Sie können jetzt mit dem Kommando `git add` Dateien zum neuen Root-Commit hinzufügen und ihn dann mit `git commit` erzeugen.

4.8. Regressionen finden – git bisect

Eine Regression bezeichnet in der Softwareentwicklung den Zeitpunkt, ab dem ein bestimmtes Feature eines Programms nicht mehr funktioniert. Das kann nach einem Update von Bibliotheken sein, nach der Einführung neuer Features, die Seiteneffekte verursachen etc.

Solche Regressionen zu finden, ist mitunter schwer. Wenn Sie eine umfangreiche Test-Suite einsetzen, dann sind Sie relativ gut davor geschützt, trivial erkennbare Regressionen einzubauen (z.B. weil Sie vor jedem Commit ein `make test` laufen lassen).

Wenn die Regression reproduzierbar ist („Mit den Argumenten `<x>` stürzt das Programm ab“, „die Konfigurationseinstellung `<y>` führt zu einem Speicherzugriffsfehler“), dann können Sie mit Git die Suche nach dem Commit, der diese Regression verursacht, automatisieren.

Git stellt dafür das Kommando `bisect` zur Verfügung, dessen Algorithmus nach dem Prinzip „teile und herrsche“ (engl. *divide and conquer*) funktioniert: Zunächst definieren Sie einen Zeitpunkt (also einen Commit), zu dem die Regression noch nicht aufgetreten war (`good`), anschließend einen Zeitpunkt, zu dem sie auftritt (genannt `bad`), lassen Sie diesen weg, nimmt Git `HEAD` an).

Das `bisect`-Kommando geht von der idealisierten Annahme aus, dass die Regression durch *einen* Commit eingeleitet wurde – es gibt also einen Commit, vor dem alles in Ordnung war, und *nach* dem der Fehler auftritt. [64]

Nun wählt Git einen Commit aus der Mitte zwischen `good` und `bad` und checkt ihn aus. Sie müssen dann überprüfen, ob die Regression weiterhin vorhanden ist. Wenn ja, dann setzt Git `bad` auf diesen Commit, wenn nein, wird `good` auf diesen Commit gesetzt. Dadurch fällt circa die Hälfte der zu untersuchenden Commits weg. Git wiederholt den Schritt, bis nur noch ein Commit übrig bleibt.

Die Anzahl der Schritte, die `bisect` benötigt, verhält sich also logarithmisch zur Anzahl der Commits, die Sie untersuchen: Für n Commits benötigen Sie ca. $\log_2(n)$ Schritte. Bei 32 Commits sind das zwar maximal fünf Schritte, für 1024 Commits aber maximal 10 Schritte, weil Sie ja im ersten Schritt schon 512 Commits eliminieren können.

4.8.1. Benutzung

Eine `bisect`-Sitzung starten Sie mit den folgenden Kommandos:

```
$ git bisect start
$ git bisect bad <funktioniert-nicht>
$ git bisect good <funktioniert>
```

Sobald Sie die beiden Punkte definiert haben, checkt Git einen Commit in der Mitte aus. Sie befinden sich also ab jetzt im *Detached-Head*-Modus (siehe [Abschnitt 3.2.1, „Detached HEAD“](#)). Nachdem Sie überprüft haben, ob die Regression noch immer vorhanden ist, können Sie ihn mit `git bisect good` bzw. `git bisect bad` markieren. Git checkt automatisch den nächsten Commit aus.

Möglicherweise können Sie den ausgecheckten Commit nicht testen, z.B. weil das Programm nicht fehlerfrei kompiliert. In diesem Fall können Sie per Git `git bisect skip` einen anderen Commit in der Nähe auswählen lassen und mit diesem wie gewohnt verfahren. Die Fehlersuche können Sie jederzeit abbrechen per `git bisect reset`.

4.8.2. Automatisierung

Idealerweise können Sie automatisiert testen, ob der Fehler auftritt – mit einem Test, der erfolgreich laufen muss, wenn die Regression nicht auftritt.

Sie können dann wie oben die Punkte `good` und `bad` definieren. Danach geben Sie `git bisect run` `<pfad/zum/test>` ein.

Anhand des Rückgabewerts entscheidet `bisect`, ob der überprüfte Commit `good` ist (wenn das Skript sich erfolgreich, d.h. mit Rückgabewert 0 beendet) oder `bad` (Werte 1–127). Ein Spezialfall ist der Rückgabewert 125, der ein `git bisect skip` bewirkt. Wenn Sie also ein Programm haben, das kompiliert werden muss, sollten Sie es erstes ein Kommando wie `make || exit 125` einbauen, so dass der Commit übersprungen wird, wenn das Programm nicht richtig kompiliert.

Bisect kann dann ganz automatisch den problematischen Commit identifizieren. Das sieht z.B. so aus:

```
$ git bisect run ./t.sh
Bisecting: 9 revisions left to test after this (roughly 3 steps) ...
Bisecting: 4 revisions left to test after this (roughly 2 steps) ...
Bisecting: 2 revisions left to test after this (roughly 1 step) ...
Bisecting: 0 revisions left to test after this (roughly 0 steps) ...
d29758fffc080d0da8ee9e5266fd75fcb98076 is the first bad commit
```

Tipp

Mit kleinen Commits und sinnvollen Beschreibungen können Sie sich durch das `bisect`-Kommando bei der Suche nach obskuren Fehlern viel Arbeit sparen.

Achten Sie daher besonders darauf, dass Sie keine Commits erzeugen, die die Software in einem „kaputten“ Zustand lassen (kompliert nicht etc.), was ein späterer Commit repariert.

[[54](#)] Wenn Sie Patch-Stacks mit Git verwalten, bei denen potentiell Konflikte auftreten können, sollten Sie sich in jedem Fall das Feature `Reuse Recorded Resolution` ansehen, kurz `rere`. `Rere` speichert Konfliktlösungen und korrigiert Konflikte automatisch, wenn schon eine Lösung gespeichert wurde, siehe auch [Abschnitt 3.4.2., „rere: Reuse Recorded Resolution“](#).

[[55](#)] Indem zum Beispiel der Branch in ein öffentlich verfügbares Repository hochgeladen wird, siehe [Abschnitt 5.4., „Commits hochladen: git push“](#).

[[56](#)] Im letzteren Fall machen Sie z.B. einfach ein `git remote update` (die neuen Commits werden in den Branch `origin/master` geladen) und bauen anschließend Ihren eigenen Branch von neuem auf `origin/master` auf. Siehe auch [Abschnitt 5.1., „Wie funktioniert verteilte Versionsverwaltung?“](#).

[[57](#)] Den Quellcode finden Sie unter <http://repo.or.cz/w/topgit.git>.

[[58](#)] Kurz `stg` oder StGit, erreichbar unter <http://www.procode.org/stgit/>.

[[59](#)] Das funktioniert auch problemlos, sofern alle Abzweigungen und Zusammenführungen *oberhalb* der neuen Referenz sind (also nur Commits enthalten sind, von denen aus man die neue Basis erreichen kann). Sonst schlägt Rebase bei jedem Commit fehl, der schon in der Geschichte enthalten ist (Fehlermeldung: „nothing to commit“); diese müssen dann stets mit einem `git rebase --continue` übersprungen werden.

[[60](#)] Weitere Beispiele finden Sie auf der Man-Page zu `gitignore(5)` und unter <http://help.github.com/git-ignore/>.

[[61](#)] Das Verhalten wird unterbunden, indem Sie die Einstellung `clean.requireForce` auf `false` setzen.

[[62](#)] Das Kommando sucht zuerst alle Commit-Objekte heraus, die nicht mehr erreichbar sind, und schränkt die Liste dann auf diejenigen ein, die Merge-Commits sind und deren Commit-MESSAGE die Zeichenkette `WIP` enthält – die Eigenschaften, die ein Commit-Objekt aufweist, das als Stash erstellt wurde, vgl. [Abschnitt 4.5.7., „Wie ist der Stash implementiert?“](#).

[[63](#)] Das stimmt nicht ganz; Sie können unter `refs/notes/commits` nur eine Notiz pro Commit speichern, zusätzlich aber z.B. unter `refs/notes/bts` noch weitere Notizen, die sich auf das Bug-Tracking-System beziehen – dort aber auch nur eine pro Commit.

[[64](#)] Dieser Commit muss natürlich nicht den Kern der Regression ausmachen, sie wurde möglicherweise durch einen ganz anderen Commit vorbereitet.

[Zurück](#)

[Weiter](#)



Lizenziert unter der [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).