Git Magic = Ben Lynn August 2007

Vorwort

Git ist eine Art "Schweizer Taschenmesser" für Versionsverwaltung. Ein zuverlässiges, vielseitiges Mehrzweck-Versionsverwaltungswerkzeug, dessen außergewöhnliche Flexibilität es schwierig zu erlernen macht, ganz zu schweigen davon, es zu meistern.

Wie Arthur C. Clarke festgestellt hat, ist jede hinreichend fortschrittliche Technologie nicht von Magie zu unterscheiden. Das ist ein großartiger Ansatz, um an Git heranzugehen: Anfänger können seine inneren Mechanismen ignorieren und Git als ein Ding ansehen, dass mit seinen erstaunlichen Fähigkeiten Freunde verzückt und Gegner zur Weißglut bringt.

Anstatt die Details aufzudecken, bieten wir grobe Anweisungen für die jeweiligen Funktionen. Bei regelmäßiger Anwendung wirst Du allmählich verstehen, wie die Tricks funktionieren und wie Du die Rezepte auf Deinen Bedarf zuschneiden kannst.

- Vereinfachtes Chinesisch: von JunJie, Meng und JiangWei. Zu Traditionellem Chinesisch konvertiert via `cconv -f UTF8-CN -t UTF8-TW`.
- Französich: von Alexandre Garel, Paul Gaborit, und Nicolas Deram. Auch gehostet unter itaapy.
- Deutsch: von Benjamin Bellee und Armin Stebich; Auch gehostet unter Armin's Website.
- Italienisch: von Mattia Rigotti.
- Portugiesisch: von Leonardo Siqueira Rodrigues [ODT-Version].
- Russisch: von Tikhon Tarnavsky, Mikhail Dymskov, und anderen.
- Spanisch: von Rodrigo Toledo und Ariset Llerena Tapia.
- Vietnamesisch: von Trần Ngọc Quân; Auch gehostet unter seiner Website.
- Einzelne Webseite: reines HTML, ohne CSS.
- PDF Datei: druckerfreundlich.

- Debian Packet, Ubuntu Packet: Für eine schnelle und lokale Kopie dieser Seite. Praktisch, wenn dieser Server offline ist.
- Gedrucktes Buch [Amazon.com]: 64 Seiten, 15.24cm x 22.86cm, schwarz/weiß. Praktisch, wenn es keinen Strom gibt.

Danke!

Ich bin erstaunt, dass so viele Leute an der Übersetzung dieser Seiten gearbeitet haben. Ich weiß es zu würdigen, dass ich, dank der Bemühungen der oben genannten, einen größeren Leserkreis erreiche.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin, und Tyler Breisacher haben Korrekturen und Verbesserungen beigesteuert.

François Marier unterhält das Debian Packet, das ursprünglich von Daniel Baumann erstellt wurde.

Meine Dankbarkeit gilt auch vielen anderen für deren Unterstützung und Lob. Ich war versucht, Euch hier alle aufzuzählen, aber das könnte Erwartungen in unermesslichem Umfang wecken.

Wenn ich Dich versehentlich vergessen habe, sag' mir bitte Bescheid oder schicke mir einfach einen Patch!

- http://repo.or.cz/ hostet freie Projekte. Die allererste Git Hosting Seite. Gegründet und betrieben von einem der ersten Git Entwickler.
- http://gitorious.org/ ist eine andere Git Hosting Seite, bevorzugt für Open-Source Projekte.
- http://github.com/ hostet Open-Source Projekte kostenlos und geschlossene Projekte gegen Gebühr.

Vielen Dank an alle diese Seiten für das Hosten dieser Anleitung.

Lizenz

 Natürlich wird der Quelltext in einem Git *Repository* gehalten und kann abgerufen werden durch:

```
$ git clone git://repo.or.cz/gitmagic.git # Erstellt "gitmagic" Verzeichnis.
```

oder von einem der Mirrorserver:

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
```

Einleitung

Ich benutze eine Analogie, um in die Versionsverwaltung einzuführen. Für eine vernünftigere Erklärung siehe den Wikipedia Artikel zur Versionsverwaltung.

Arbeit ist Spiel

Ich spiele Computerspiele schon fast mein ganzes Leben. Im Gegensatz dazu habe ich erst als Erwachsener damit begonnen, Versionsverwaltungssysteme zu benutzen. Ich vermute, dass ich damit nicht alleine bin, und der Vergleich hilft vielleicht dabei, die Konzepte einfacher zu erklären und zu verstehen.

Stelle Dir das Bearbeiten deines Codes oder Deiner Dokumente wie ein Computerspiel vor. Wenn Du gut vorangekommen bist, willst Du das Erreichte sichern. Um das zu tun, klickst Du auf auf Speichern in Deinem vertrauten Editor.

Aber das überschreibt die vorherige Version. Das ist wie bei den Spielen der alten Schule, die nur Speicherplatz für eine Sicherung hatten: sicherlich konntest Du speichern, aber Du konntest nie zu einem älteren Stand zurück. Das war eine Schande, denn vielleicht war Deine vorherige Sicherung an einer außergewöhnlich spannenden Stelle des Spiels, zu der Du später gerne noch einmal zurückkehren möchtest. Oder noch schlimmer, Deine aktuelle Sicherung ist in einem nicht lösbaren Stand, dann musst Du von ganz vorne beginnen.

Versionsverwaltung

Beim Editieren kannst Du Deine Datei durch *Speichern unter …* mit einem neuen Namen abspeichern oder Du kopierst sie vor dem Speichern irgendwo hin, um die alte Version zu erhalten. Außerdem kannst Du sie komprimieren, um Speicherplatz zu sparen. Das ist eine primitive und mühselige Form der Versionsverwaltung. Computerspiele machten das lange Zeit so, viele von ihnen hatten automatisch erstellte Sicherungspunkte mit Zeitstempel.

Jetzt lass uns das Problem etwas komplizierter machen. Sagen wir, Du hast einen Haufen Dateien, die zusammen gehören, z.B. Quellcodes für ein Projekt oder Dateien einer Website. Wenn Du nun eine alte Version erhalten willst, musst Du den ganzen Ordner archivieren. Viele Versionen auf diese Art zu archivieren, ist mühselig und kann sehr schnell teuer werden.

Bei einigen Computerspielen bestand ein gesicherter Stand wirklich aus einem Ordner voller Dateien. Diese Spiele versteckten die Details vor dem Spieler und

präsentierten eine bequeme Oberfläche, um verschiedene Versionen des Ordners zu verwalten.

Versionsverwaltungen sind nicht anders. Sie alle haben bequeme Schnittstellen, um Ordner voller Dateien zu verwalten. Du kannst den Zustand des Ordners sichern, so oft Du willst, und du kannst später jeden Sicherungspunkt wieder herstellen. Im Gegensatz zu den meisten Computerspielen sind sie aber in der Regel dafür ausgelegt, sparsam mit dem Speicherplatz umzugehen. Normalerweise ändern sich immer nur wenige Dateien zwischen zwei Versionen, und die Änderungen selbst sind oft nicht groß. Die Platzersparnis beruht auf dem Speichern der Unterschiede an Stelle einer Kopie der ganzen Datei.

Verteilte Kontrolle

Nun stell Dir ein ganz kompliziertes Computerspiel vor. So schwierig zu lösen, dass viele erfahrene Spieler auf der ganzen Welt beschließen, sich zusammen zu tun und ihre gespeicherten Spielstände auszutauschen, um das Spiel zu beenden. *Speedruns* sind Beispiele aus dem echten Leben: Spieler, die sich in unterschiedlichen Spielebenen des selben Spiels spezialisiert haben, arbeiten zusammen, um erstaunliche Ergebnisse zu erzielen.

Wie würdest Du ein System erstellen, bei dem jeder auf einfache Weise die Sicherungen der anderen bekommt? Und eigene Sicherungen bereitstellt?

Früher nutzte jedes Projekt eine zentralisierte Versionsverwaltung. Irgendwo speicherte ein Server alle gespeicherten Spiele, sonst niemand. Jeder Spieler hatte nur ein paar gespeicherte Spiele auf seinem Rechner. Wenn ein Spieler einen Fortschritt machen wollte, musste er den aktuellsten Stand vom Hauptserver herunterladen, eine Weile spielen, sichern und den Stand dann wieder auf den Server laden, damit irgendjemand ihn nutzen kann.

Was, wenn ein Spieler aus irgendeinem Grund einen alten Spielstand will? Vielleicht ist der aktuell gesicherte Spielstand nicht mehr lösbar, weil jemand in der dritten Ebene vergessen hat, ein Objekt aufzunehmen und sie versuchen, den letzten Spielstand zu finden, ab dem das Spiel wieder lösbar ist. Oder sie wollen zwei Spielstände vergleichen, um festzustellen, wie viel ein Spieler geleistet hat.

Es gibt viele Gründe, warum man einen älteren Stand sehen will, aber das Ergebnis ist das selbe. Man muss vom Hauptserver das alte gespeicherte Spiel anfordern. Je mehr gespeicherte Spiele benötigt werden, desto mehr Kommunikation ist erforderlich.

Die neue Generation der Versionsverwaltungssysteme, zu denen Git gehört, werden verteilte Systeme genannt und können als eine Verallgemeinerung der zentralisierten Systeme verstanden werden. Wenn Spieler vom Hauptserver herunterladen, erhalten sie jedes gespeichertes Spiel, nicht nur das zuletzt gespeicherte. Es ist, als ob der Hauptserver gespiegelt wird.

Dieses erste *Cloning* kann teuer sein, vor allem, wenn eine lange Geschichte existiert, aber auf Dauer wird es sich lohnen. Ein unmittelbarer Vorteil ist, wenn aus irgendeinem Grund ein älterer Stand benötigt wird, ist keine Kommunikation mit dem Hauptserver notwendig.

Ein dummer Aberglaube

Ein weit verbreitetes Missverständnis ist, dass verteilte System ungeeignet sind für Projekte, die ein offizielles zentrales *Repository* benötigen. Nichts könnte weiter von der Wahrheit entfernt sein. Jemanden zu fotografieren, stiehlt nicht dessen Seele. Genauso wenig setzt das *Clonen* des zentralen *Repository* dessen Bedeutung herab.

Eine gute erste Annäherung ist, dass alles, was eine zentralisierte Versionsverwaltung kann, ein gut durchdachtes verteiltes System besser kann. Netzwerkressourcen sind einfach teurer als lokale Ressourcen. Auch wenn wir später Nachteile beim verteilten Ansatz sehen werden, ist man mit dieser Faustregel weniger anfällig für falsche Vergleiche.

Ein kleines Projekt mag nur einen Bruchteil der Möglichkeiten benötigen, die so ein System bietet. Aber deshalb ein einfacheres, schlecht erweiterbares System zu benutzen, ist wie römische Ziffern zum Rechnen mit kleinen Zahlen zu verwenden.

Außerdem könnte Dein Projekt weit über die ursprünglichen Erwartungen hinauswachsen. Git von Anfang an zu benutzen, ist wie ein Schweizer Taschenmesser mit sich zu tragen, auch wenn damit meistens nur Flaschen geöffnet werden. Eines Tages brauchst Du vielleicht dringend einen Schraubendreher, dann bist du froh, mehr als nur einen einfachen Flaschenöffner bei dir zu haben.

Merge Konflikte

Für diesen Punkt ist unsere Computerspielanalogie ungeeignet. Stattdessen stellen wir uns wieder vor, wir editieren ein Dokument.

Stell dir vor, Alice fügt eine Zeile am Dateianfang hinzu und Bob eine am Dateiende. Beide laden ihre Änderungen hoch. Die meisten Systeme wählen automatisch eine vernünftige Vorgehensweise: akzeptiere beide Änderungen und füge sie zusammen, damit fließen beide Änderungen in das Dokument mit ein.

Nun stell Dir vor beide, Alice und Bob, machen Änderungen in der selben Zeile. Dann ist es unmöglich, ohne menschlichen Eingriff fortzufahren. Die zweite Person, welche die Datei hoch lädt, wird über einen *'Merge' Konflikt* informiert und muss entscheiden, welche Änderung übernommen wird oder die ganze Zeile überarbeiten.

Es können noch weitaus kompliziertere Situationen entstehen. Versionsverwaltungssysteme behandeln die einfacheren Fälle selbst und überlassen die schwierigen uns Menschen. Üblicherweise ist deren Verhalten einstellbar.

Erste Schritte

Bevor wir uns in ein Meer von Git-Befehlen stürzen, schauen wir uns ein paar einfache Beispiele an. Trotz ihrer Einfachheit sind alle davon wichtig und nützlich. Um ehrlich zu sein, in meinen ersten Monaten mit Git brauchte ich nicht mehr, als in diesem Kapitel beschrieben steht.

Stand sichern

Hast Du gravierende Änderungen vor? Nur zu, aber speichere Deinen aktuellen Stand vorher lieber nochmal ab:

```
$ git init
$ git add .
$ git commit -m "Meine erste Sicherung"
```

Falls deine Änderungen schief gehen, kannst du jetzt die alte Version wiederherstellen:

```
$ git reset --hard
```

Um den neuen Stand zu sichern:

```
$ git commit -a -m "Eine andere Sicherung"
```

Hinzufügen, Löschen, Umbenennen

Bisher kümmert sich Git nur um Dateien, die existierten, als Du das erste Mal **git add** ausgeführt hast. Wenn Du Dateien oder Verzeichnisse hinzufügst, musst du Git das mitteilen:

```
$ git add readme.txt Dokumentation
```

Ebenso, wenn Git Dateien vergessen soll:

```
$ git rm ramsch.h veraltet.c
$ git rm -r belastendes/material/
```

Git löscht diese Dateien für Dich, falls Du es noch nicht getan hast.

Eine Datei umzubenennen, ist das selbe, wie sie zu löschen und unter neuem Namen hinzuzufügen. Git benutzt hierzu die Abkürzung **git mv**, welche die gleiche Syntax wie **mv** hat. Zum Beispiel:

```
$ git mv fehler.c feature.c
```

Fortgeschrittenes Rückgängig machen/Wiederherstellen

Manchmal möchtest Du einfach zurückgehen und alle Änderungen ab einem bestimmten Zeitpunkt verwerfen, weil sie falsch waren. Dann:

```
$ git log
```

zeigt Dir eine Liste der bisherigen *Commits* und deren SHA1 Hashwerte:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

    Ersetze printf() mit write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

Die ersten paar Zeichen eines Hashwert reichen aus, um einen *Commit* zu identifizieren; alternativ benutze kopieren und einfügen für den kompletten Hashwert. Gib ein:

```
$ git reset --hard 766f
```

um den Stand eines bestimmten *Commits* wieder herzustellen und alle nachfolgenden Änderungen für immer zu löschen.

Ein anderes Mal willst Du nur kurz zu einem älteren Stand springen. In diesem Fall, gib folgendes ein:

```
$ git checkout 82f5
```

Damit springst Du in der Zeit zurück, behältst aber neuere Änderungen. Aber, wie bei Zeitreisen in einem Science-Fiction-Film, wenn Du jetzt etwas änderst und *commitest*, gelangst Du in eine alternative Realität, denn Deine Änderungen sind anders als beim früheren *Commit*.

Diese alternative Realität heißt *Branch*, und wir kommen später darauf zurück. Für jetzt merke Dir,

```
$ git checkout master
```

bringt Dich wieder in die Gegenwart. Um zu verhindern, dass sich Git beschwert, solltest Du vor einem *Checkout* alle Änderungen *commiten* oder *reseten*.

Um wieder die Computerspielanalogie anzuwenden:

- `git reset --hard`: Lade einen alten Stand und lösche alle Spielstände, die neuer sind als der jetzt geladene.
- `git checkout`: Lade einen alten Spielstand, aber wenn Du weiterspielst, wird der Spielstand von den früher gesicherten Spielständen abweichen. Jeder Spielstand, der ab jetzt gesichert wird, entsteht in dem separaten *Branch*, welcher der alternative Realität entspricht. dazu kommen wir später.

Du kannst auch nur einzelne Dateien oder Verzeichnisse wiederherstellen, indem Du sie an den Befehl anhängst:

```
$ git checkout 82f5 eine.datei andere.datei
```

Sei vorsichtig, diese Art des ***Checkout*** kann Dateien überschreiben, ohne dass Du etwas merkst. Um Unfälle zu vermeiden, solltest Du immer *commiten* bevor Du ein *Checkout* machst, besonders am Anfang, wenn Du Git noch erlernst. Allgemein gilt: Wenn Du unsicher bist, egal, ob ein Git Befehl oder irgendeine andere Operation, führe zuerst **git commit -a** aus.

Du magst Kopieren und Einfügen von Hashes nicht? Dann nutze:

```
$ git checkout :/"Meine erste Si"
```

um zu einem *Commit* zu springen, dessen Beschreibung so anfängt. Du kannst auch nach dem 5. letzten *Commit* fragen:

```
$ git checkout master~5
```

Rückgängig machen

In einem Gerichtssaal können Ereignisse aus den Akten gelöscht werden. Ähnlich kannst Du gezielt *Commits* rückgängig machen.

```
$ git commit -a
$ git revert 1b6d
```

wird den *Commit* mit dem angegebenen Hashwert rückgängig machen. Das Rückgängig machen wird als neuer *Commit* erstellt, was mit **git log** überprüft werden kann.

Changelog erstellen

Verschiedene Projekte benötigen ein Änderungsprotokoll. Das kannst du mit folgendem Befehl erstellen:

```
$ git log > ChangeLog
```

Dateien herunterladen

Eine Kopie eines mit Git verwalteten Projekts bekommst du mit:

```
$ git clone git://server/pfad/zu/dateien
```

Um zum Beispiel alle Dateien zu bekommen, die ich zum Erzeugen dieser Seiten benutze:

```
$ git clone git://git.or.cz/gitmagic.git
```

Es gibt gleich noch viel mehr über den *clone* Befehl zu sagen.

Das Neueste vom Neuen

Wenn Du schon eine Kopie eines Projektes hast, kannst Du es auf die neuste Version aktualisieren mit:

```
$ git pull
```

Einfaches Veröffentlichen

Angenommen Du hast ein Skript geschrieben und möchtest es anderen zugänglich machen. Du könntest sie einfach bitten, es von Deinem Computer herunterzuladen, aber falls sie das tun, während Du experimentierst oder das Skript verbesserst, könnten sie in Schwierigkeiten geraten. Genau deswegen gibt es Releasezyklen. Entwickler arbeiten regelmäßig an einem Projekt, veröffentlichen den Code aber nur, wenn sie ihn für vorzeigbar halten.

Um dies in Git zu tun, gehe ins Verzeichnis in dem das Skript liegt:

```
$ git init
$ git add .
$ git commit -m "Erster Stand"
```

Dann sage Deinen Nutzern:

```
$ git clone dein.computer:/pfad/zum/skript
```

um Dein Skript herunterzuladen. Das setzt voraus, dass sie einen SSH-Zugang haben. Falls nicht, führe **git deamon** aus und sage den Nutzern folgendes:

```
$ git clone git://dein.computer/pfad/zum/skript
```

Ab jetzt, immer wenn Dein Skript reif für eine Veröffentlichung ist:

```
$ git commit -a -m "Nächster Stand"
```

und Deine Nutzer können ihr Skript aktualisieren mit:

```
$ git pull
```

Deine Nutzer werden nie mit Versionen in Kontakt kommen, von denen Du es nicht willst. Natürlich funktioniert der Trick für fast alles, nicht nur Skripts.

Was habe ich getan?

Finde heraus, was Du seit dem letzten *Commit* getan hast:

```
$ git diff
```

Oder seit Gestern:

```
$ git diff "@{gestern}"
```

Oder zwischen irgendeiner Version und der vorvorletzten:

```
$ git diff 1b6d "master~2"
```

Jedes mal ist die Ausgabe ein *Patch*, der mit **git apply** eingespielt werden kann. Versuche auch:

```
$ git whatchanged --since="2 weeks ago"
```

Um mir die Geschichte eines *Repositories* anzuzeigen, benutze ich häufig qgit da es eine schicke Benutzeroberfläche hat, oder tig, eine Konsolenanwendung, die sehr gut über langsame Verbindungen funktioniert. Alternativ kannst Du einen Webserver installieren mit **git instaweb**, dann kannst Du mit jedem Webbrowser darauf zugreifen.

Übung

A, B, C, D sind 4 aufeinander folgende *Commits*. B ist identisch mit A, außer, dass einige Dateien gelöscht wurden. Wir möchten die Dateien in D wieder hinzufügen, aber nicht in B. Wie machen wir das?

Es gibt mindestens 3 Lösungen. Angenommen, wir sind bei D:

1. Der Unterschied zwischen A und B sind die gelöschten Dateien. Wir können einen *Patch* erstellen, der diesen Unterschied darstellt und diesen dann auf D anwenden:

   ```
   $ git diff B A | git apply
   ```

2. Da die Dateien im *Repository* unter dem *Commit* A gespeichert sind, können wir sie wieder herstellen:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Wir können den *Commit* von A auf B als Änderung betrachten, die wir rückgängig machen wollen:

   ```
   $ git revert B
   ```

Welche Lösung ist die beste? Die, welche Dir am besten gefällt. Es ist einfach, mit Git das zu bekommen, was Du willst und oft führen viele Wege zum Ziel.

Rund ums *Clonen*

In älteren Versionsverwaltungssystemen ist *checkout* die Standardoperation, um Dateien zu bekommen. Du bekommst einen Haufen Dateien eines bestimmten Sicherungsstands.

In Git und anderen verteilten Versionsverwaltungssystemen ist *clone* die Standardaktion. Um Dateien zu bekommen, erstellst Du einen *Clone* des gesamten *Repository*. Oder anders gesagt, Du spiegelst den zentralen Server. Alles, was man mit dem zentralen *Repository* tun kann, kannst Du auch mit deinem *Clone* tun.

Computer synchronisieren

Es ist akzeptabel, für Datensicherungen und einfaches Synchronisieren mit *tarball* Archiven oder **rsync** zu arbeiten. Aber manchmal arbeite ich an meinem Laptop, dann an meinem Desktop-PC, und die beiden haben sich inzwischen nicht austauschen können.

Erstelle ein Git *Repository* und *commite* Deine Dateien auf dem einen Rechner. Dann auf dem anderen:

```
$ git clone anderer.computer:/pfad/zu/dateien
```

um eine zweite Kopie der Dateien und des Git *Repository* zu erstellen. Von jetzt an wird

```
$ git commit -a
$ git pull anderer.computer:/pfad/zu/dateien HEAD
```

den Zustand der Dateien des anderen Computer auf den übertragen, an dem Du gerade arbeitest. Solltest Du kürzlich konkurrierende Änderungen an der selben Datei vorgenommen haben, lässt Git Dich das wissen, und Du musst erneut *commiten*, nachdem Du die Konflikte aufgelöst hast.

Klassische Quellcodeverwaltung

Erstelle ein Git *Repository* für Deine Dateien:

```
$ git init
$ git add .
$ git commit -m "Erster Commit"
```

Auf dem zentralen Server erstelle ein *bare Repository* in irgendeinem Ordner:

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$ touch proj.git/git-daemon-export-ok
```

Wenn nötig, starte den Git-Dämon:

```
$ git daemon --detach  # er könnte schon laufen
```

Für Git Hostingdienste folge den Anweisungen zum Erstellen des zunächst leeren Git *Repository*. Normalerweise füllt man ein Formular auf einer Website aus.

Übertrage (*push*) dein Projekt auf den zentralen Server mit:

```
$ git push zentraler.server/pfad/zu/proj.git HEAD
```

Um die Quellcodes abzurufen, gibt ein Entwickler folgendes ein:

```
$ git clone zentraler.server/pfad/zu/proj.git
```

Nach dem Bearbeiten sichert der Entwickler die Änderungen lokal:

```
$ git commit -a
```

Um auf die aktuelle Server-Version zu aktualisieren:

```
$ git pull
```

Irgendwelche *Merge*-Konflikte sollten dann aufgelöst und erneut *commitet* werden:

```
$ git commit -a
```

Um die lokalen Änderungen in das zentrale *Repository* zu übertragen:

```
$ git push
```

Wenn inzwischen neue Änderungen von anderen Entwicklern beim Hauptserver eingegangen sind, schlägt Dein *push* fehl. Aktualisiere das lokale *Repository* erneut mit *pull*, löse eventuell aufgetretene *Merge*-Konflikte und versuche es nochmal.

Entwickler brauchen SSH Zugriff für die vorherigen *pull* und *push* Anweisungen. Trotzdem kann jedermann die Quelltexte einsehen, durch Eingabe von:

```
$ git clone git://zentraler.server/pfad/zu/proj.git
```

Das ursprüngliche Git-Protokoll ähnelt HTTP: Es gibt keine Authentifizierung, also kann jeder das Projekt abrufen. Folglich ist standardmäßig das *Pushen* per Git-Protokoll verboten.

Geheime Quellen

Für ein Closed-Source-Projekt lasse die *touch* Anweisung weg und stelle sicher, dass niemals eine Datei namens `git-daemon-export-ok` erstellt wird. Das *Repository* kann nun nicht mehr über das Git-Protokol abgerufen werden; nur diejenigen mit SSH Zugriff können es einsehen. Wenn alle *Repositories* geschlossen sind, ist es unnötig den Git Dämon laufen zu lassen, da jegliche Kommunikation über SSH läuft.

Nackte Repositories

Ein nacktes (*bare*) *Repository* wird so genannt, weil es kein Arbeitsverzeichnis hat. Es enthält nur Dateien, die normalerweise im *.git* Unterverzeichnis versteckt sind. Mit anderen Worten, es verwaltet die Geschichte eines Projekts, enthält aber niemals einen Auszug irgendeiner beliebigen Version.

Ein *bare Repository* übernimmt die Rolle des Hauptserver in einem zentralisierten Versionsverwaltungssystem: Das Zuhause Deines Projekts. Entwickler *clonen* Dein Projekt davon und *pushen* die letzten offiziellen Änderungen dort hin. Meistens befindet es sich auf einem Server, der nicht viel tut außer Daten zu verbreiten. Die Entwicklung findet in den *Clonen* statt, so kann das Heim-*Repository* ohne Arbeitsverzeichnis auskommen.

Viele Git Befehle funktionieren nicht in *bare Repositories*. Es sei denn, die `GIT_DIR` Umgebungsvariable wird auf das Arbeitsverzeichnis gesetzt, oder die `--bare` Option wird übergeben.

Push oder *Pull*

Warum haben wir den *push*-Befehl eingeführt, anstatt bei dem vertrauten *pull*-Befehl zu bleiben? Zuerst, *pull* funktioniert nicht mit *bare Repositories*: stattdessen benutze *fetch*, ein Befehl, den wir später behandeln. Aber auch wenn wir ein normales *Repository* auf dem zentralen Server halten würden, wäre das *pullen* eine mühselige Angelegenheit. Wir müssten uns zuerst in den Server einloggen und dem *pull*-Befehl die Netzwerkadresse des Computer übergeben, von dem aus wir die Änderungen *pullen*, also abholen wollen. Firewalls könnten uns stören und was, wenn wir gar keine Berechtigung für eine Serverkonsole haben.

Wie auch immer, abgesehen von diesem Fall, raten wir vom *Pushen* in ein *Repository* ab. Falls das Ziel nämlich ein Arbeitsverzeichnis hat, können Verwirrungen entstehen.

Kurzum, während du lernst mit Git umzugehen, *pushe* nur, wenn das Ziel ein *bare Repository* ist, andernfalls benutze *pull*.

Fork eines Projekts

Hast Du es satt, wie sich ein Projekt entwickelt? Du denkst, Du kannst das besser? Dann mache folgendes auf deinem Server:

```
$ git clone git://haupt.server/pfad/zu/dateien
```

Dann erzähle jedem von Deinem *Fork* des Projekts auf Deinem Server.

Zu jedem späteren Zeitpunkt kannst du die Änderungen des Originalprojekts *mergen* mit:

```
$ git pull
```

Ultimative Datensicherung

Du willst zahlreiche, vor Manipulation geschützte, redundante Datensicherungen an unterschiedlichen Orten? Wenn Dein Projekt viele Entwickler hat, musst Du nichts tun! Jeder *Clone* Deines Codes ist eine vollwertige Datensicherung. Nicht nur des aktuellen Stand, sondern die gesamte Geschichte. Wird irgendein *Clone* beschädigt, wird dies dank des kryptographischen *Hashing* sofort erkannt, sobald derjenige versucht, mit anderen zu kommunizieren.

Wenn Dein Projekt nicht so bekannt ist, finde so viele Server, wie Du kannst, um dort einen *Clone* zu platzieren.

Die wirklich Paranoiden sollten immer den letzten 20-Byte SHA1 Hash des *HEAD* aufschreiben und an einem sicheren Ort aufbewahren. Er muss sicher sein, aber nicht privat. Zum Beispiel wäre es sicher, ihn in einer Zeitung zu veröffentlichen, denn es ist schwer für einen Angreifer jede Zeitungskopie zu manipulieren.

Multitasking mit Lichtgeschwindigkeit

Nehmen wir an Du willst parallel an mehreren Funktionen arbeiten. Dann *commite* Dein Projekt und gib ein:

```
$ git clone . /irgendein/neuer/ordner
```

Harten Links ist es zu verdanken, dass ein lokaler Klon weniger Zeit und Speicherplatz benötigt als eine herkömmliche Datensicherung.

Du kannst nun an zwei unabhängigen Funktionen gleichzeitig arbeiten. Zum Beispiel kannst Du an einen Klon bearbeiten, während der andere kompiliert wird. Zu jeder Zeit kannst Du *comitten* und die Änderungen des anderen Klon *pullen*.

```
$ git pull /der/andere/clone HEAD
```

Versionsverwaltung im Untergrund

Arbeitest Du an einem Projekt, das ein anderes Versionsverwaltungssystem nutzt und vermisst Du Git? Dann erstelle ein Git *Repository* in deinem Arbeitsverzeichnis:

```
$ git init
$ git add .
$ git commit -m "Erster Commit"
```

dann *Clone* es:

```
$ git clone . /irgendein/neuer/ordner
```

Nun gehe in das neue Verzeichnis und arbeite dort mit Git nach Herzenslust. Irgendwann wirst Du dann mit den anderen synchronisieren wollen, dann gehe in das Originalverzeichnis, aktualisiere mit dem anderen Versionsverwaltungssystem und gib ein:

```
$ git add .
$ git commit -m "Synchronisation mit den anderen"
```

Dann gehe wieder ins neue Verzeichnis und gib ein:

```
$ git commit -a -m "Beschreibung der Änderungen"
$ git pull
```

Die Vorgehensweise, wie Du Deine Änderungen den anderen übergibst, hängt vom anderen Versionsverwaltungssystem ab. Das neue Verzeichnis enthält die Dateien mit deinen Änderungen. Führe die Anweisungen des anderen Versionsverwaltungssystems aus, die nötig sind, um die Dateien ins zentrale *Repository* zu übertragen.

Subversion, vielleicht das beste zentralisierte Versionsverwaltungssystem, wird von unzähligen Projekten benutzt. Der **git svn**-Befehl automatisiert den zuvor genannten Ablauf für Subversion *Repositories* und kann auch benutzt werden, um ein Git Projekt in ein Subversion *Repository* zu exportieren.

Mercurial

Mercurial ist ein ähnliches Versionsverwaltungssystem, das fast nahtlos mit Git zusammenarbeiten kann. Mit der `hg-git`-Erweiterung kann ein Benutzer von Mercurial verlustfrei in ein Git *Repository pushen* und daraus *pullen*.

Beschaffe Dir die `hg-git`-Erweiterung mit Git:

```
$ git clone git://github.com/schacon/hg-git.git
```

oder Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

Leider kenne ich keine solche Erweiterung für Git. Aus diesem Grund plädiere ich für Git statt Mercurial für ein zentrales *Repository*, auch wenn man Mercurial bevorzugt. Bei einem Mercurial Projekt gibt es gewöhnlich immer einen Freiwilligen, der parallel dazu ein Git *Repository* für die Git Anwender unterhält, wogegen, Dank der `hg-git`-Erweiterung, ein Git Projekt automatisch die Benutzer von Mercurial mit einbezieht.

Die Erweiterung kann auch ein Mercurial *Repository* in ein Git *Repository* umwandeln, indem man in ein leeres *Repository pushed*. Einfacher geht das mit dem `hg-fast-export.sh` Skript, welches es hier gibt:

```
$ git clone git://repo.or.cz/fast-export.git
```

Zum Konvertieren gib in einem leeren Verzeichnis ein:

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

nachdem Du das Skript zu deinem `$PATH` hinzugefügt hast.

Bazaar

Wir erwähnen auch kurz Bazaar, weil es nach Git und Mercurial das bekannteste freie verteilte Versionsverwaltungssystem ist.

Bazaar hat den Vorteil des Rückblicks, da es relativ jung ist; seine Entwickler konnten aus Fehlern der Vergangenheit lernen und kleine historische Unwegbarkeiten umgehen. Außerdem waren sich die Entwickler der Popularität und Interoperabilität mit anderen Versionsverwaltungssystemen bewusst.

Eine `bzr-git`-Erweiterung lässt Anwender von Bazaar einigermaßen mit Git *Repositories* arbeiten. Das `tailor` Programm konvertiert Bazaar *Repositories* zu Git *Repositories* und kann das forlaufend tun, während `bzr-fast-export` für einmalige Konvertierungen besser geeignet ist.

Warum ich Git benutze

Ich habe ursprünglich Git gewählt, weil ich gehört habe, dass es die unvorstellbar unüberschaubaren Linux Kernel Quellcodes verwalten kann. Ich hatte noch keinen Grund zu wechseln. Git hat mir bewundernswert gedient und hat mich bis jetzt noch nie im Stich gelassen. Da ich in erster Linie unter Linux arbeite, sind Probleme anderer Plattformen bedeutungslos.

Ich bevorzuge auch C-Programme und *bash*-Skripte gegenüber Anwendungen wie zum Beispiel Python Skripts: Es gibt weniger Abhängigkeiten und ich bin süchtig nach schellen Ausführungszeiten.

Ich dachte darüber nach, wie Git verbessert werden könnte, ging sogar so weit, dass ich meine eigene Git-Ähnliche Anwendung schrieb, allerdings nur als akademische Übungen. Hätte ich mein Projekt fertig gestellt, wäre ich trotzdem bei Git geblieben, denn die Verbesserungen wären zu gering gewesen, um den Einsatz eines Eigenbrödler-Systems zu rechtfertigen.

Natürlich können Deine Bedürfnisse und Wünsche ganz anders sein und vielleicht bist Du mit einem anderen System besser dran. Wie auch immer, mit Git kannst Du nicht viel falsch machen.

Branch-Magie

Unverzügliches *Branchen* und *Mergen* sind die hervorstechenden Eigenschaften von Git.

Problem: Externe Faktoren zwingen zum Wechsel des Kontext. Ein schwerwiegender Fehler in der veröffentlichten Version tritt ohne Vorwarnung auf. Die Frist für ein bestimmtes Leistungsmerkmal rückt näher. Ein Entwickler, dessen Unterstützung für eine Schlüsselstelle im Projekt wichtig ist, verlässt das Team. In allen Fällen musst Du alles stehen und liegen lassen und Dich auf eine komplett andere Aufgabe konzentrieren.

Den Gedankengang zu unterbrechen ist schlecht für die Produktivität und je komplizierter der Kontextwechsel ist, desto größer ist der Verlust. Mit zentraler Versionsverwaltung müssen wir eine neue Arbeitskopie vom Server herunterladen. Bei verteilen Systemen ist das viel besser, da wir die benötigt Version lokal *clonen* können.

Doch das *Clonen* bringt das Kopieren des gesamten Arbeitsverzeichnis wie auch die ganze Geschichte bis zum angegebenen Punkt mit sich. Auch wenn Git die Kosten durch Dateifreigaben und Verknüpfungen reduziert, müssen doch die gesamten Projektdateien im neuen Arbeitsverzeichnis erstellt werden.

Lösung: Git hat ein besseres Werkzeug für diese Situationen, die wesentlich schneller und platzsparender als *clonen* ist: **git branch**.

Mit diesem Zauberwort verwandeln sich die Dateien in Deinem Arbeitsverzeichnis plötzlich von einer Version in eine andere. Diese Verwandlung kann mehr als nur in der Geschichte vor und zurück gehen. Deine Dateien können sich verwandeln, vom aktuellsten Stand, zur experimentellen Version, zum neusten Entwicklungsstand, zur Version Deines Freundes und so weiter.

Die Chef-Taste

Hast Du schon einmal ein Spiel gespielt, wo beim Drücken einer Taste („der Chef-Taste“), der Monitor sofort ein Tabellenblatt oder etwas anderes angezeigt

hat? Dass, wenn der Chef ins Büro spaziert, während Du das Spiel spielst, Du es schnell verstecken kannst?

In irgendeinem Verzeichnis:

```
$ echo "Ich bin klüger als mein Chef" > meinedatei.txt
$ git init
$ git add .
$ git commit -m "Erster Stand"
```

Wir haben ein Git *Repository* erstellt, das eine Textdatei mit einer bestimmten Nachricht enthält. Nun gib ein:

```
$ git checkout -b chef  # scheinbar hat sich danach nichts geändert
$ echo "Mein Chef ist klüger als ich" > meinedatei.txt
$ git commit -a -m "Ein anderer Stand"
```

Es sieht aus, als hätten wir unsere Datei überschrieben und *commitet*. Aber es ist eine Illusion. Tippe:

```
$ git checkout master  # wechsle zur Originalversion der Datei
```

und Simsalabim! Die Textdatei ist wiederhergestellt. Und wenn der Chef in diesem Verzeichnis herumschnüffelt, tippe:

```
$ git checkout chef  # wechsle zur Version die der Chef ruhig sehen kann
```

Du kannst zwischen den beiden Versionen wechseln, so oft du willst und du kannst unabhängig voneinander in jeder Version Änderungen *commiten*

Schmutzarbeit

Sagen wir, Du arbeitest an einer Funktion und Du musst, warum auch immer, drei Versionen zurückgehen, um ein paar print Anweisungen einzufügen, damit Du siehst, wie etwas funktioniert. Dann:

```
$ git commit -a
$ git checkout HEAD~3
```

Nun kannst Du überall wild temporären Code hinzufügen. Du kannst diese Änderungen sogar *commiten*. Wenn Du fertig bist,

```
$ git checkout master
```

um zur ursprünglichen Arbeit zurückzukehren. Beachte, dass alle Änderungen, die nicht *commitet* sind, übernommen werden.

Was, wenn Du am Ende die temporären Änderungen sichern willst? Einfach:

```
$ git checkout -b schmutzig
```

und *commite*, bevor Du auf den *Master Branch* zurückschaltest. Wann immer Du zu Deiner Schmutzarbeit zurückkehren willst, tippe einfach:

```
$ git checkout schmutzig
```

Wir sind mit dieser Anweisung schon in einem früheren Kapitel in Berührung gekommen, als wir das Laden alter Stände besprochen haben. Nun können wir die ganze Geschichte erzählen: Die Dateien ändern sich zu dem angeforderten Stand, aber wir müssen den *Master Branch* verlassen. Jeder *Commit* ab jetzt führt Deine Dateien auf einen anderen Weg, dem wir später noch einen Namen geben können.

Mit anderen Worten, nach dem Abrufen eines alten Stands versetzt Dich Git automatisch in einen neuen, unbenannten *Branch*, der mit **git checkout -b** benannt und gesichert werden kann.

Schnelle Fehlerbehebung

Du steckst mitten in der Arbeit, als es heißt, alles fallen zu lassen, um einen neu entdeckten Fehler in *Commit* `1b6d...` zu beheben:

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

Dann, wenn Du den Fehler behoben hast:

```
$ git commit -a -m "Fehler behoben"
$ git checkout master
```

und fahre mit Deiner ursprünglichen Arbeit fort. Du kannst sogar die frisch gebackene Fehlerkorrektur auf Deinen aktuellen Stand übernehmen:

```
$ git merge fixes
```

Mergen

Mit einigen Versionsverwaltungssystemen ist das Erstellen eines *Branch* einfach, aber das Zusammenfügen (*Mergen*) ist schwierig. Mit Git ist *Mergen* so einfach, dass Du gar nicht merkst, wenn es passiert.

Tatsächlich sind wir dem *Mergen* schon lange begegnet. Die **pull** Anweisung holt (*fetch*) eigentlich die *Commits* und verschmilzt (*merged*) diese dann mit dem aktuellen *Branch*. Wenn Du keine lokalen Änderungen hast, dann ist *merge* eine *schnelle Weiterleitung*, ein Ausnahmefall, ähnlich dem Abrufen der letzten Version eines zentralen Versionsverwaltungssystems. Wenn Du aber Änderungen hast, wird Git diese automatisch *mergen* und Dir Konflikte melden.

Normalerweise hat ein *Commit* genau einen Eltern-*Commit*, nämlich den vorhergehenden *Commit*. Das *Mergen* mehrerer *Branches* erzeugt einen *Commit* mit mindestens zwei Eltern. Das wirft die Frage auf: Welchen *Commit*

referenziert `HEAD~10` tatsächlich? Ein *Commit* kann mehrere Eltern haben, welchem folgen wir also?

Es stellt sich heraus, dass diese Notation immer den ersten Elternteil wählt. Dies ist erstrebenswert, denn der aktuelle *Branch* wird zum ersten Elternteil während eines *Merge*; häufig bist Du nur von Änderungen betroffen, die Du im aktuellen *Branch* gemacht hast, als von den Änderungen, die von anderen *Branches* eingebracht wurden.

Du kannst einen bestimmten Elternteil mit einem Caret-Zeichen referenzieren. Um zum Beispiel die Logs vom zweiten Elternteil anzuzeigen:

```
$ git log HEAD^2
```

Du kannst die Nummer für den ersten Elternteil weglassen. Um zum Beispiel die Unterschiede zum ersten Elternteil anzuzeigen:

```
$ git diff HEAD^
```

Du kannst diese Notation mit anderen Typen kombinieren. Zum Beispiel:

```
$ git checkout 1b6d^^2~10 -b uralt
```

beginnt einen neuen *Branch* „uralt“, welcher den Stand 10 *Commits* zurück vom zweiten Elternteil des ersten Elternteil des *Commits*, dessen Hashwert mit 1b6d beginnt.

Kontinuierlicher Arbeitsfluss

In Herstellungsprozessen muss der zweite Schritt eines Plans oft auf die Fertigstellung des ersten Schritt warten. Ein Auto, das repariert werden soll, steht unbenutzt in der Garage bis ein Ersatzteil geliefert wird. Ein Prototyp muss warten, bis ein Baustein fabriziert wurde, bevor die Konstruktion fortgesetzt werden kann.

Bei Softwareprojekten kann das ähnlich sein. Der zweite Teil eines Leistungsmerkmals muss warten, bis der erste Teil veröffentlicht und getestet wurde. Einige Projekte erfordern, dass Dein Code überprüft werden muss, bevor er akzeptiert wird; Du musst also warten, bis der erste Teil geprüft wurde, bevor Du mit dem zweiten Teil anfangen kannst.

Dank des schmerzlosen *Branchen* und *Mergen* können wir die Regeln beugen und am Teil II arbeiten, bevor Teil I offiziell freigegeben wurde. Angenommen Du hast Teil I *commitet* und zur Prüfung eingereicht. Sagen wir, Du bist im `master` *Branch*. Dann *branche* zu Teil II:

```
$ git checkout -b teil2
```

Du arbeitest also an Teil II und *commitest* Deine Änderungen regelmäßig. Irren ist menschlich und so kann es vorkommen, dass Du zurück zu Teil I willst, um

einen Fehler zu beheben. Wenn Du Glück hast oder sehr gut bist, kannst Du die nächsten Zeilen überspringen.

```
$ git checkout master  # Gehe zurück zu Teil I.
$ fix_problem
$ git commit -a        # 'Commite' die Lösung.
$ git checkout teil2   # Gehe zurück zu Teil II.
$ git merge master     # 'Merge' die Lösung.
```

Schließlich, Teil I ist zugelassen:

```
$ git checkout master  # Gehe zurück zu Teil I.
$ submit files         # Veröffentliche deine Dateien!
$ git merge teil2      # 'Merge' in Teil II.
$ git branch -d teil2  # Lösche den Branch "teil2"
```

Nun bist Du wieder im `master` *Branch*, mit Teil II im Arbeitsverzeichnis.

Es ist einfach, diesen Trick auf eine beliebige Anzahl von Teilen zu erweitern. Es ist genauso einfach, rückwirkend zu *branchen*: angenommen, Du merkst zu spät, dass vor sieben *Commits* ein *Branch* erforderlich gewesen wäre. Dann tippe:

```
$ git branch -m master teil2  # Umbenennen des 'Branch' "master" zu "teil2".
$ git branch master HEAD~7   # Erstelle neuen "master", 7 Commits voraus
```

Der `master` Branch enthält nun Teil I, und der `teil2` Branch enthält den Rest. Wir befinden uns in letzterem Branch; wir haben `master` erzeugt, ohne dorthin zu wechseln, denn wir wollen im `teil2` weiterarbeiten. Das ist unüblich. Bisher haben wir unmittelbar nach dem Erstellen in einen *Branch* gewechselt, wie in:

```
$ git checkout HEAD~7 -b master  # erzeuge einen Branch, und wechsle zu ihm.
```

Mischmasch Reorganisieren

Vielleicht magst Du es, alle Aspekte eines Projekts im selben *Branch* abzuarbeiten. Du willst deine laufenden Arbeiten für Dich behalten und andere sollen Deine *Commits* nur sehen, wenn Du sie hübsch organisiert hast. Beginne ein paar *Branches*:

```
$ git branch sauber        # Erzeuge einen Branch für gesäuberte Commits.
$ git checkout -b mischmasch  # Erzeuge und wechsle in den Branch zum Arbeiten.
```

Fahre fort, alles zu bearbeiten: Behebe Fehler, füge Funktionen hinzu, erstelle temporären Code und so weiter und *commite* Deine Änderungen oft. Dann:

```
$ git checkout bereinigt
$ git cherry-pick mischmasch^^
```

wendet den Urahn des obersten *Commit* des „mischmasch“ *Branch* auf den „bereinigt“ *Branch* an. Durch das Herauspicken der Rosinen kannst Du einen

Branch konstruieren, der nur endgültigen Code enthält und zusammengehörige *Commits* gruppiert hat.

Branches **verwalten**

Ein Liste aller *Branches* bekommst Du mit:

```
$ git branch
```

Standardmäßig beginnst Du in einem *Branch* namens „master“. Einige plädieren dafür, den „master“ *Branch* unangetastet zu lassen und für seine Arbeit einen neuen *Branch* anzulegen.

Die **-d** und **-m** Optionen erlauben dir *Branches* zu löschen und zu verschieben (umzubenennen). Siehe **git help branch**.

Der „master“ *Branch* ist ein nützlicher Brauch. Andere können davon ausgehen, dass Dein *Repository* einen *Branch* mit diesem Namen hat und dass er die offizielle Version enthält. Auch wenn Du den „master“ *Branch* umbenennen oder auslöschen könntest, kannst Du diese Konvention aber auch respektieren.

Temporäre ***Branches***

Nach einer Weile wirst Du feststellen, dass Du regelmäßig kurzlebige *Branches* erzeugst, meist aus dem gleichen Grund: jeder neue *Branch* dient lediglich dazu, den aktuellen Stand zu sichern, damit Du kurz zu einem alten Stand zurück kannst, um eine vorrangige Fehlerbehebung zu machen oder irgendetwas anderes.

Es ist vergleichbar mit dem kurzzeitigen Umschalten des Fernsehkanals, um zu sehen, was auf dem anderen Kanal los ist. Doch anstelle ein paar Knöpfe zu drücken, machst du *create*, *checkout*, *merge* und *delete* von temporären *Branches*. Glücklicherweise hat Git eine Abkürzung dafür, die genauso komfortabel ist wie eine Fernbedienung:

```
$ git stash
```

Das sichert den aktuellen Stand an einem temporären Ort (*stash*=Versteck) und stellt den vorherigen Stand wieder her. Dein Arbeitsverzeichnis erscheint wieder exakt in dem Zustand, wie es war, bevor Du anfingst zu editieren. Nun kannst Du Fehler beheben, Änderungen vom zentralen *Repository* holen (*pull*) und so weiter. Wenn Du wieder zurück zu Deinen Änderungen willst, tippe:

```
$ git stash apply  # Es kann sein, dass Du Konflikte auflösen musst.
```

Du kannst mehrere *stashes* haben und diese unterschiedlich handhaben. Siehe **git help stash**. Wie Du Dir vielleicht schon gedacht hast, verwendet Git *Branches* im Hintergrund, um diesen Zaubertrick durchzuführen.

Arbeite, wie Du willst

Du magst Dich fragen, ob *Branches* diesen Aufwand wert sind. Immerhin sind *Clone* fast genauso schnell, und Du kannst mit **cd** anstelle von esoterischen Git Befehlen zwischen ihnen wechseln.

Betrachten wir Webbrowser. Warum mehrere Tabs unterstützen und mehrere Fenster? Weil beides zu erlauben, eine Vielzahl an Stilen unterstützt. Einige Anwender möchten nur ein Browserfenster geöffnet haben und benutzen Tabs für unterschiedliche Webseiten. Andere bestehen auf dem anderen Extrem: mehrere Fenster, ganz ohne Tabs. Wieder andere bevorzugen irgendetwas dazwischen.

Branchen ist wie Tabs für dein Arbeitsverzeichnis, und *Clonen* ist wie das Öffnen eines neuen Browserfenster. Diese Operationen sind schnell und lokal, also warum nicht damit experimentieren, um die beste Kombination für sich selbst zu finden? Git lässt Dich genauso arbeiten, wie Du es willst.

Geschichtsstunde

Eine Folge von Git's verteilter Natur ist, dass die Chronik einfach verändert werden kann. Aber, wenn Du an der Vergangenheit manipulierst, sei vorsichtig: verändere nur den Teil der Chronik, den Du ganz alleine hast. So wie Nationen ewig diskutieren, wer welche Greueltaten vollbracht hat, wirst Du beim Abgleichen in Schwierigkeiten geraten, falls jemand einen *Clone* mit abweichender Chronik hat und die Zweige sich austauschen sollen.

Einige Entwickler setzen sich nachhaltig für die Unantastbarkeit der Chronik ein, mit allen Fehlern, Nachteilen und Mängeln. Andere denken, dass Zweige vorzeigbar gemacht werden sollten, bevor sie auf die Öffentlichkeit losgelassen werden. Git versteht beide Gesichtspunkte. Wie *Clonen*, *Branchen* und *Mergen* ist das Umschreiben der Chronik lediglich eine weitere Stärke, die Git Dir bietet. Es liegt an Dir, diese Weise zu nutzen.

Ich nehme alles zurück

Hast Du gerade *commitet*, aber Du hättest gerne eine andere Beschreibung eingegeben? Dann gib ein:

```
$ git commit --amend
```

um die letzte Beschreibung zu ändern. Du merkst, dass Du vergessen hast, eine Datei hinzuzufügen? Führe **git add** aus, um sie hinzuzufügen, und dann die vorhergehende Anweisung.

Du willst noch ein paar Änderungen zu deinem letzten *Commit* hinzufügen? Dann mache diese Änderungen und gib ein:

```
$ git commit --amend -a
```

... und noch viel mehr

Nehmen wir jetzt an, das vorherige Problem ist zehnmal schlimmer. Nach einer längeren Sitzung hast Du einen Haufen *Commits* gemacht. Aber Du bist mit der Art der Organisation nicht glücklich, und einige *Commits* könnten etwas umformuliert werden. Dann gib ein:

```
$ git rebase -i HEAD~10
```

und die letzten zehn *Commits* erscheinen in Deinem bevorzugten $EDITOR. Auszug aus einem Beispiel:

```
pick 5c6eb73 Link repo.or.cz hinzugefügt
pick a311a64 Analogien in "Arbeite wie Du willst" umorganisiert
pick 100834f Push-Ziel zum Makefile hinzugefügt
```

Dann:

- Entferne *Commits* durch das Löschen von Zeilen.
- Organisiere *Commits* durch verschieben von Zeilen.
- Ersetze `pick` mit:
  - `edit`, um einen *Commit* für *amends* zu markieren.
  - `reword`, um die Log-Beschreibung zu ändern.
  - `squash`, um einen *Commit* mit dem vorhergehenden zu vereinen (*merge*).
  - `fixup`, um einen *Commit* mit dem vorhergehenden zu vereinen (*merge*) und die Log-Beschreibung zu verwerfen.

Speichere und Beende. Wenn Du einen *Commit* mit *edit* markiert hast, gib ein:

```
$ git commit --amend
```

Ansonsten:

```
$ git rebase --continue
```

Also *commite* früh und oft: Du kannst später mit *rebase* aufräumen.

Lokale Änderungen zum Schluss

Du arbeitest an einem aktiven Projekt. Über die Zeit haben sich einige lokale *Commits* angesammelt und dann synchronisierst Du mit einem *Merge* mit dem offiziellen Zweig. Dieser Zyklus wiederholt sich ein paar Mal, bevor Du zum *Pushen* in den zentralen Zweig bereit bist.

Aber nun ist die Chronik in deinem lokalen Git-*Clone* ein chaotisches Durcheinander deiner Änderungen und den Änderungen vom offiziellen Zweig. Du willst alle Deine Änderungen lieber in einem fortlaufenden Abschnitt und hinter den offiziellen Änderungen sehen.

Das ist eine Aufgabe für **git rebase**, wie oben beschrieben. In vielen Fällen kannst Du den **--onto** Schalter benutzen, um Interaktion zu vermeiden.

Siehe auch **git help rebase** für ausführliche Beispiele dieser erstaunlichen Anweisung. Du kannst auch *Commits* aufteilen. Du kannst sogar *Branches* in einem *Repository* umorganisieren.

Chronik umschreiben

Gelegentlich brauchst Du Versionsverwaltung vergleichbar dem Wegretuschieren von Personen aus einem offiziellen Foto, um diese in stalinistischer Art aus der Geschichte zu löschen. Stell Dir zum Beispiel vor, Du willst ein Projekt veröffentlichen, aber es enthält eine Datei, die aus irgendwelchen Gründen privat bleiben muss. Vielleicht habe ich meine Kreditkartennummer in einer Textdatei notiert und diese versehentlich dem Projekt hinzugefügt. Die Datei zu löschen, ist zwecklos, da über ältere *Commits* auf sie zugegriffen werden könnte. Wir müssen die Datei aus allen *Commits* entfernen:

```
$ git filter-branch --tree-filter 'rm sehr/geheime/Datei' HEAD
```

Siehe **git help filter-branch**, wo dieses Beispiel erklärt und eine schnellere Methode vorstellt wird. Allgemein, **filter-branch** lässt Dich große Bereiche der Chronik mit einer einzigen Anweisung verändern.

Danach beschreibt der Ordner `.git/refs/original` den Zustand der Lage vor der Operation. Prüfe, ob die *filter-branch* Anweisung getan hat, was Du wolltest, dann lösche dieses Verzeichnis, bevor Du weitere *filter-branch* Operationen durchführst.

Zuletzt, ersetze alle *Clones* Deines Projekts mit Deiner überarbeiteten Version, falls Du später mit ihnen interagieren möchtest.

Geschichte machen

Du möchtest ein Projekt zu Git umziehen? Wenn es mit einem der bekannteren Systeme verwaltet wird, besteht die Möglichkeit, dass schon jemand ein Skript geschrieben hat, das die gesamte Chronik für Git exportiert.

Anderenfalls, sieh dir **git fast-import** an, das Text in einem speziellen Format einliest, um eine Git Chronik von Anfang an zu erstellen. Normalerweise wird ein Skript, das diese Anweisung benutzt, hastig zusammengeschustert und einmalig ausgeführt, um das Projekt in einem einzigen Lauf zu migrieren.

Erstelle zum Beispiel aus folgendem Listing eine temporäre Datei, z.B. `/tmp/history`:

```
commit refs/heads/master committer Alice <alice@example.com> Thu, 01 Jan
1970 00:00:00 +0000 data <<EOT Initial commit.  EOT

M 100644 inline hello.c data <<EOT #include <stdio.h>

int main() {
  printf("Hallo, Welt!\n");
  return 0;
}
EOT


commit refs/heads/master committer Bob <bob@example.com> Tue, 14 Mar 2000
01:59:26 -0800 data <<EOT Ersetze printf() mit write().  EOT

M 100644 inline hello.c data <<EOT #include <unistd.h>

int main() {
  write(1, "Hallo, Welt!\n", 14);
  return 0;
}
EOT
```

Dann, erstelle ein Git *Repository* aus dieser temporären Datei, durch Eingabe von:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

Die aktuellste Version des Projekts kannst Du abrufen (*checkout*) mit:

```
$ git checkout master .
```

Die Anweisung **git fast-export** konvertiert jedes *Repository* in das **git fast-import** Format, dessen Ausgabe Du studieren kannst, um Exporte zu schreiben und außerdem, um *Repositories* im menschenlesbaren Text zu übertragen.

Tatsächlich können diese Anweisungen Klartext-*Repositories* über reine Textkanäle übertragen.

Wo ging alles schief?

Du hast gerade eine Funktion in Deiner Anwendung entdeckt, die nicht mehr funktioniert und Du weißt sicher, dass sie vor ein paar Monaten noch ging.

Argh! Wo kommt dieser Fehler her? Hättest Du nur die Funktion während der Entwicklung getestet.

Dafür ist es nun zu spät. Wie auch immer, vorausgesetzt Du hast oft *comittet*, kann Git Dir sagen, wo das Problem liegt:

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git ruft einen Stand ab, der genau dazwischen liegt. Teste die Funktion und wenn sie immer noch nicht funktioniert:

```
$ git bisect bad
```

Wenn nicht, ersetzte "bad" mit "good". Git versetzt Dich wieder auf einen Stand genau zwischen den bekannten Versionen "good" und "bad" und reduziert so die Möglichkeiten. Nach ein paar Durchläufen wird Dich diese binäre Suche zu dem *Commit* führen, der die Probleme verursacht. Wenn Du deine Ermittlungen abgeschlossen hast, kehre zum Originalstand zurück mit:

```
$ git bisect reset
```

Anstatt jede Änderung per Hand zu untersuchen, automatisiere die Suche durch Ausführen von:

```
$ git bisect run mein_skript
```

Git benutzt den Rückgabewert der übergebenen Anweisung, normalerweise ein Skript für einmalige Ausführung, um zu entscheiden, ob eine Änderung gut (*good*) oder schlecht (*bad*) ist: Das Skript sollte 0 für *good* zurückgeben, 125 wenn die Änderung übersprungen werden soll und irgendetwas zwischen 1 und 127 für *bad*. Ein negativer Rückgabewert beendet die *bisect*-Operation sofort.

Du kannst noch viel mehr machen: die Hilfe erklärt, wie man *bisect*-Operationen visualisiert, das *bisect*-Log untersucht oder wiedergibt und sicher unschuldige Änderungen ausschließt, um die Suche zu beschleunigen.

Wer ist verantwortlich?

Wie viele andere Versionsverwaltungssysteme hat Git eine *blame* Anweisung:

```
$ git blame bug.c
```

das jede Zeile in der angegebenen Datei kommentiert, um anzuzeigen, wer sie zuletzt geändert hat und wann. Im Gegensatz zu vielen anderen Versionsverwaltungssystemen funktioniert diese Operation offline, es wird nur von der lokalen Festplatte gelesen.

Persönliche Erfahrungen

In einem zentralisierten Versionsverwaltungssystem ist das Bearbeiten der Chronik eine schwierige Angelegenheit und den Administratoren vorbehalten. *Clonen*, *Branchen* und *Mergen* sind unmöglich ohne Netzwerkverbindung. Ebenso grundlegende Funktionen wie das Durchsuchen der Chronik oder das *comitten* einer Änderung. In manchen Systemen benötigt der Anwender schon eine Netzwerkverbindung, nur um seine eigenen Änderungen zu sehen oder um eine Datei zum Bearbeiten zu öffnen.

Zentralisierte Systeme schließen es aus, offline zu arbeiten und benötigen teurere Netzwerkinfrastruktur, vor allem, wenn die Zahl der Entwickler steigt. Am wichtigsten ist, dass alle Operationen bis zu einem gewissen Grad langsamer sind, in der Regel bis zu dem Punkt, wo Anwender erweiterte Anweisungen scheuen, bis sie absolut notwendig sind. In extremen Fällen trifft das auch auf die grundlegenden Anweisungen zu. Wenn Anwender langsame Anweisungen ausführen müssen, sinkt die Produktivität, da der Arbeitsfluss unterbrochen wird.

Ich habe diese Phänomen aus erster Hand erfahren. Git war das erste Versionsverwaltungssystem, das ich benutzt habe. Ich bin schnell in die Anwendung hineingewachsen und betrachtete viele Funktionen als selbstverständlich. Ich habe einfach vorausgesetzt, dass andere Systeme ähnlich sind: die Auswahl eines Versionsverwaltungssystems sollte nicht anders sein als die Auswahl eines Texteditors oder Internetbrowser.

Ich war geschockt, als ich später gezwungen war, ein zentralisiertes System zu benutzen. Eine unzuverlässige Internetverbindung stört mit Git nicht sehr, aber sie macht die Entwicklung unerträglich, wenn sie so zuverlässig wie ein lokale Festplatte sein sollte. Zusätzlich habe ich mich dabei ertappt, bestimmte Anweisungen zu vermeiden, um die damit verbundenen Wartezeiten zu vermeiden, und das hat mich letztendlich davon abgehalten, meinem gewohnten Arbeitsablauf zu folgen.

Wenn ich eine langsame Anweisung auszuführen hatte, wurde durch die Unterbrechung meiner Gedankengänge dem Arbeitsfluss ein unverhältnismäßiger Schaden zugefügt. Während des Wartens auf das Ende der Serverkommunikation tat ich etwas anderes, um die Wartezeit zu überbrücken, zum Beispiel E-Mails lesen oder Dokumentation schreiben. Wenn ich zur ursprünglichen Arbeit zurückkehrte, war die Operation längst beendet und ich vergeudete noch mehr Zeit beim Versuch, mich zu erinnern, was ich getan habe. Menschen sind nicht gut im Kontextwechsel.

Da war auch ein interessanter Tragik-der-Allmende Effekt: Netzwerküberlastungen erahnend, verbrauchten einzelne Individuen für diverse Operationen mehr Netzwerkbandbreite als erforderlich, um zukünftige Engpässe zu vermeiden. Die Summe der Bemühungen verschlimmerte die Überlastungen, was einzelne wiederum ermutigte, noch mehr Bandbreite zu verbrauchen, um noch längere

Wartezeiten zu verhindern.

Multiplayer Git

Anfangs benutzte ich Git bei einem privaten Projekt, bei dem ich der einzige Entwickler war. Unter den Befehlen im Zusammenhang mit Git's verteilter Art, brauchte ich nur **pull** und **clone**, damit konnte ich das selbe Projekt an unterschiedlichen Orten halten.

Später wollte ich meinen Code mit Git veröffentlichen und Änderungen von Mitstreitern einbinden. Ich musste lernen, wie man Projekte verwaltet, an denen mehrere Entwickler aus aller Welt beteiligt waren. Glücklicherweise ist das Git's Stärke und wohl auch seine Daseinsberechtigung.

Wer bin ich?

Jeder *Commit* enthält Name und eMail-Adresse des Autors, welche mit **git log** angezeigt werden. Standardmäßig nutzt Git Systemeinstellungen, um diese Felder auszufüllen. Um diese Angaben explizit zu setzen, gib ein:

```
$ git config --global user.name "Max Mustermann"
$ git config --global user.email maxmustermann@beispiel.de
```

Lasse den -global Schalter weg, um diese Einstellungen für das aktuelle *Repository* zu setzen.

Git über SSH, HTTP

Angenommen, Du hast einen SSH-Zugang zu einem Webserver, aber Git ist nicht installiert. Wenn auch nicht so effizient wie mit dem systemeigenen Protokoll, kann Git über HTTP kommunizieren.

Lade Git herunter, compiliere und installiere es unter Deinem Benutzerkonto und erstellen ein *Repository* in Deinem Webverzeichnis:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Bei älteren Git Versionen funktioniert der *copy*-Befehl nicht, stattdessen gib ein:

```
$ chmod a+x hooks/post-update
```

Nun kannst Du Deine letzten Änderungen über SSH von jedem *Clone* aus veröffentlichen.

```
$ git push web.server:/pfad/zu/proj.git master
```

und jedermann kann Dein Projekt abrufen mit:

```
$ git clone http://web.server/proj.git
```

Git über alles

Willst Du *Repositories* ohne Server synchronisieren oder gar ohne Netzwerkverbindung? Musst Du während eines Notfalls improvisieren? Wir haben gesehen, dass man mit **git fast-export** und **git fast-import** *Repositories* in eine einzige Datei konvertieren kann und zurück. Wir können solche Dateien hin und her schicken, um Git *Repositories* über jedes beliebige Medium zu transportieren, aber ein effizienteres Werkzeug ist **git bundle**.

Der Absender erstellt ein *Bundle*:

```
$ git bundle create einedatei HEAD
```

und transportiert das *Bundle* `einedatei` irgendwie zum anderen Beteiligten: per eMail, USB-Stick, einen **xxd** Hexdump und einen OCR Scanner, Morsecode über Telefon, Rauchzeichen usw. Der Empfänger holt sich die *Commits* aus dem *Bundle* durch Eingabe von:

```
$ git pull einedatei
```

Der Empfänger kann das sogar mit einem leeren *Repository* tun. Trotz seiner Größe, `einedatei` enthält das komplette original Git *Repository*.

In größeren Projekten vermeidest Du Datenmüll, indem Du nur Änderungen *bundlest*, die in den anderen *Repositories* fehlen. Zum Beispiel, nehmen wir an, der *Commit* „1b6d…“ ist der aktuellste, den beide Parteien haben:

```
$ git bundle create einedatei HEAD ^1b6d
```

Macht man das regelmäßig, kann man leicht vergessen, welcher *Commit* zuletzt gesendet wurde. Die Hilfeseiten schlagen vor, *Tags* zu benutzen, um dieses Problem zu lösen. Das heißt, nachdem Du ein *Bundle* gesendet hast, gib ein:

```
$ git tag -f letztesbundle HEAD
```

und erstelle neue Aktualisierungsbundles mit:

```
$ git bundle create neuesbundle HEAD ^letztesbundle
```

Patches: Das globale Zahlungsmittel

Patches sind die Klartextdarstellung Deiner Änderungen, die von Computern und Menschen gleichermaßen einfach verstanden werden. Dies verleiht ihnen eine universelle Anziehungskraft. Du kannst einen *Patch* Entwicklern schicken,

ganz egal, was für ein Versionsverwaltungssystem sie benutzen. Solange Deine Mitstreiter ihre eMails lesen können, können sie auch Deine Änderungen sehen. Auch auf Deiner Seite ist alles was Du brauchst ein eMail-Konto: es gibt keine Notwendigkeit ein Online Git *Repository* aufzusetzen.

Erinnere Dich an das erste Kapitel:

```
$ git diff 1b6d > mein.patch
```

gibt einen *Patch* aus, der zur Diskussion einfach in eine eMail eingefügt werden kann. In einem Git *Repository* gib ein:

```
$ git apply < mein.patch
```

um den *Patch* anzuwenden.

In einer offizielleren Umgebung, wenn Autorennamen und eventuell Signaturen aufgezeichnet werden sollen, erstelle die entsprechenden *Patches* nach einem bestimmten Punkt durch Eingabe von:

```
$ git format-patch 1b6d
```

Die resultierenden Dateien können an **git-send-email** übergeben werden oder von Hand verschickt werden. Du kannst auch eine Gruppe von *Commits* angeben:

```
$ git format-patch 1b6d..HEAD^^
```

Auf der Empfängerseite speichere die eMail in eine Datei, dann gib ein:

```
$ git am < email.txt
```

Das wendet den eingegangenen *Patch* an und erzeugt einen *Commit*, inklusive der Informationen wie z.B. den Autor.

Mit einer Webmail Anwendung musst Du eventuell ein Button anklicken, um die eMail in ihrem rohen Originalformat anzuzeigen, bevor Du den *Patch* in eine Datei sicherst.

Es gibt geringfügige Unterschiede bei mbox-basierten eMail Anwendungen, aber wenn Du eine davon benutzt, gehörst Du vermutlich zu der Gruppe Personen, die damit einfach umgehen können, ohne Anleitungen zu lesen.!

Entschuldigung, wir sind umgezogen.

Nach dem *Clonen* eines *Repositories*, wird **git push** oder **git pull** automatisch auf die original URL zugreifen. Wie macht Git das? Das Geheimnis liegt in der Konfiguration, die beim *Clonen* erzeugt wurde. Lasst uns einen Blick riskieren:

```
$ git config --list
```

Die `remote.origin.url` Option kontrolliert die Quell-URL; „origin“ ist der Spitzname, der dem Quell-*Repository* gegeben wurde. Wie mit der „master“

Branch Konvention können wir diesen Spitznamen ändern oder löschen, aber es gibt für gewöhnlich keinen Grund, dies zu tun.

Wenn das original *Repository* verschoben wird, können wir die URL aktualisieren mit:

```
$ git config remote.origin.url git://neue.url/proj.git
```

Die `branch.master.merge` Option definiert den Standard-Remote-*Branch* bei einem **git pull**. Während des ursprünglichen *Clonen* wird sie auf den aktuellen *Branch* des Quell-*Repository* gesetzt, so dass selbst dann, wenn der *HEAD* des Quell-*Repository* inzwischen auf einen anderen *Branch* gewechselt hat, ein späterer *pull* treu dem original *Branch* folgen wird.

Diese Option gilt nur für das *Repository*, von dem als erstes *gecloned* wurde, was in der Option `branch.master.remote` hinterlegt ist. Bei einem *pull* aus anderen *Repositories* müssen wir explizit angeben, welchen *Branch* wir wollen:

```
$ git pull git://beispiel.com/anderes.git master
```

Das obige erklärt, warum einige von unseren früheren *push* und *pull* Beispielen keine Argumente hatten.

Entfernte *Branches*

Wenn Du ein *Repository clonst*, *clonst* Du auch alle seine *Branches*. Das hast Du vielleicht noch nicht bemerkt, denn Git versteckt diese: Du musst speziell danach fragen. Das verhindert, dass *Branches* vom entfernten *Repository* Deine lokalen *Branches* stören, und es macht Git einfacher für Anfänger.

Zeige die entfernten *Branches* an mit:

```
$ git branch -r
```

Du solltes etwas sehen wie:

```
origin/HEAD
origin/master
origin/experimentell
```

Diese Liste zeigt die *Branches* und den HEAD des entfernten *Repository*, welche auch in regulären Git Anweisungen verwendet werden können. Zum Beispiel, angenommen Du hast viele *Commits* gemacht und möchtest einen Vergleich zur letzten abgeholten Version machen. Du kannst die Logs nach dem entsprechenden SHA1 Hashwert durchsuchen, aber es ist viel einfacher, folgendes einzugeben:

```
$ git diff origin/HEAD
```

Oder Du kannst schauen, was auf dem *Branch* „experimentell“ los war:

```
$ git log origin/experimentell
```

Mehrere *Remotes*

Angenommen, zwei andere Entwickler arbeiten an Deinem Projekt, und wir wollen beide im Auge behalten. Wir können mehr als ein *Repository* gleichzeitig beobachten mit:

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

Nun haben wir einen *Branch* vom zweiten *Repository* eingebunden und wir haben einfachen Zugriff auf alle *Branches* von allen *Repositories*:

```
$ git diff origin/experimentell^ other/some_branch~5
```

Aber was, wenn wir nur deren Änderungen vergleichen wollen, ohne unsere eigene Arbeit zu beeinflussen? Mit anderen Worten, wir wollen ihre *Branches* untersuchen, ohne dass deren Änderungen in unser Arbeitsverzeichnis einfließen. Anstatt *pull* benutzt Du dann:

```
$ git fetch        # Fetch vom origin, der Standard.
$ git fetch other  # Fetch vom zweiten Programmierer.
```

Dies holt lediglich die Chroniken. Obwohl das Arbeitsverzeichnis unverändert bleibt, können wir nun jeden *Branch* aus jedem *Repository* in einer Git Anweisung referenzieren, da wir eine lokale Kopie besitzen.

Erinnere Dich, dass ein *Pull* hinter den Kulissen einfach ein **fetch** gefolgt von einem **merge** ist. Normalerweise machen wir einen **pull**, weil wir die letzten *Commits* abrufen und einbinden wollen. Die beschriebene Situation ist eine erwähnenswerte Ausnahme.

Siehe **git help remote**, um zu sehen, wie man Remote-*Repositories* entfernt, bestimmte *Branches* ignoriert und mehr.

Meine Einstellungen

Für meine Projekte bevorzuge ich es, wenn Unterstützer *Repositories* vorbereiten, von denen ich *pullen* kann. Verschiedene Git Hosting Anbieter lassen Dich mit einem Klick deine eigene *Fork* eines Projekts hosten.

Nachdem ich einen Zweig abgerufen habe, benutze ich Git Anweisungen, um durch die Änderungen zu navigieren und zu untersuchen, die idealerweise gut organisiert und dokumentiert sind. Ich *merge* meine eigenen Änderungen und führe eventuell weitere Änderungen durch. Wenn ich zufrieden bin, *pushe* ich in das zentrale *Repository*.

Obwohl ich nur unregelmäßig Beiträge erhalte, glaube ich, dass diese Methode sich auszahlt. Siehe diesen Blog Beitrag von Linus Torvalds (englisch).

In der Git Welt zu bleiben, ist etwas bequemer als *Patch*-Dateien, denn es erspart mir, sie in Git *Commits* zu konvertieren. Außerdem kümmert sich Git um die Details wie Autorname und eMail-Adresse, genauso wie um Datum und Uhrzeit und es fordert den Autor zum Beschreiben seiner eigenen Änderungen auf.

Git für Fortgeschrittene

Mittlerweile solltest Du Dich in den **git help** Seiten zurechtfinden und das meiste verstanden haben. Trotzdem kann es langwierig sein, den exakten Befehl zur Lösung einer bestimmten Aufgabe herauszufinden. Vielleicht kann ich Dir etwas Zeit sparen: Nachfolgend findest Du ein paar Rezepte, die ich in der Vergangenheit gebraucht habe.

Quellcode veröffentlichen

Bei meinen Projekten verwaltet Git genau die Dateien, die ich archivieren und für andere Benutzer veröffentlichen will. Um ein tarball-Archiv des Quellcodes zu erzeugen, verwende ich den Befehl:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Commite Änderungen

Git mitzuteilen, welche Dateien man hinzugefügt, gelöscht und umbenannt hat, ist für manche Projekte sehr mühsam. Stattdessen kann man folgendes eingeben:

```
$ git add .
$ git add -u
```

Git wird sich die Dateien im aktuellen Verzeichnis ansehen und sich die Details selbst erarbeiten. Anstelle des zweiten Befehl kann man auch `git commit -a` ausführen, falls man an dieser Stelle ohnehin *comitten* möchte. Siehe **git help ignore** um zu sehen, wie man Dateien definiert, die ignoriert werden sollen.

Man kann das aber auch in einem einzigen Schritt ausführen mit:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Die **-z** und **-0** Optionen verhindern unerwünschte Nebeneffekte durch Dateinamen mit ungewöhnlichen Zeichen. Da diese Anweisung aber auch zu ignorierende Dateien hinzufügt, kann man noch die `-x` oder `-X` Option hinzufügen.

Mein *Commit* ist zu groß!

Hast Du es zu lange versäumt zu *comitten*? Hast Du so versessen programmiert, dass Du darüber die Quellcodeverwaltung vergessen hast? Machst Du eine Serie von unabhängigen Änderungen, weil es Dein Stil ist?

Keine Sorge, gib ein:

```
$ git add -p
```

Für jede Änderung, die Du gemacht hast, zeigt Git Dir die Codepassagen, die sich geändert haben und fragt, ob sie Teil des nächsten *Commit* sein sollen. Antworte mit "y" für Ja oder "n" für Nein. Du hast auch noch andere Optionen, z.B. den Aufschub der Entscheidung; drücke "?" um mehr zu erfahren.

Wenn Du zufrieden bist, gib

```
$ git commit
```

ein, um exakt die ausgewählten Änderungen zu *comitten* (die "inszenierten" Änderungen). Achte darauf, nicht die Option **-a** einzusetzen, anderenfalls wird Git alle Änderungen *comitten*.

Was ist, wenn Du viele Dateien an verschiedenen Orten bearbeitet hast? Jede Datei einzeln nachzuprüfen, ist frustrierend und ermüdend. In diesem Fall verwende **git add -i**, dessen Bedienung ist nicht ganz einfach, dafür aber sehr flexibel. Mit ein paar Tastendrücken kannst Du mehrere geänderte Dateien für den *Commit* hinzufügen (*stage*) oder entfernen (*unstage*) oder Änderungen einzelner Dateien nachprüfen und hinzufügen. Alternativ kannst Du **git commit --interactive** verwenden, was dann automatisch die ausgewählten Änderungen *commited*, nachdem Du fertig bist.

Der Index: Git's Bereitstellungsraum

Bis jetzt haben wir Git's berühmten *Index* gemieden, aber nun müssen wir uns mit ihm auseinandersetzen, um das bisherige zu erklären. Der Index ist ein temporärer Bereitstellungsraum. Git tauscht selten Daten direkt zwischen Deinem Projekt und seiner Versionsgeschichte aus. Vielmehr schreibt Git die Daten zuerst in den Index, danach kopiert es die Daten aus dem Index an ihren eigentlichen Bestimmungsort.

Zum Beispiel ist **commit -a** eigentlich ein zweistufiger Prozess. Der erste Schritt erstellt einen Schnappschuss des aktuellen Status jeder überwachten Datei im Index. Der zweite Schritt speichert dauerhaft den Schnappschuss, der sich nun im Index befindet. Ein *Commit* ohne die **-a** Option führt nur den zweiten Schritt aus und macht nur wirklich Sinn, wenn zuvor eine Anweisung angewendet wurde, welche den Index verändert, wie zum Beispiel **git add**.

Normalerweise können wir den Index ignorieren und so tun, als würden wir direkt aus der Versionsgeschichte lesen oder in sie schreiben. In diesem Fall wollen wir aber mehr Kontrolle, also manipulieren wir den Index. Wir erstellen einen Schnappschuss einiger, aber nicht aller unser Änderungen im Index und speichern dann diesen sorgfältig zusammengestellten Schnappschuss permanent.

Verliere nicht Deinen KOPF

Der HEAD Bezeichner ist wie ein Cursor, der normalerweise auf den jüngsten *Commit* zeigt und mit jedem neuen *Commit* voranschreitet. Einige Git Anweisungen lassen Dich ihn manipulieren. Zum Beispiel:

```
$ git reset HEAD~3
```

bewegt den HEAD Bezeichner drei *Commits* zurück. Dadurch agieren nun alle Git Anweisungen, als hätte es die drei letzten *Commits* nicht gegeben, während Deine Dateien unverändert erhalten bleiben. Siehe auf der Git Hilfeseite für einige Anwendungsbeispiele.

Aber wie kannst Du zurück in die Zukunft? Die vergangenen *Commits* wissen nichts von der Zukunft.

Wenn Du den SHA1 Schlüssel vom originalen HEAD hast, dann:

```
$ git reset 1b6d
```

Aber stell Dir vor, Du hast ihn niemals notiert? Keine Sorge: Für solche Anweisungen sichert Git den original HEAD als Bezeichner mit dem Namen ORIG_HEAD und Du kannst gesund und munter zurückkehren mit:

```
$ git reset ORIG_HEAD
```

KOPF-Jagd

Möglicherweise reicht ORIG_HEAD nicht aus. Vielleicht hast Du gerade bemerkt, dass Du einen kapitalen Fehler gemacht hast, und nun musst Du zu einem uralten *Commit* in einem länst vergessenen *Branch* zurück.

Standardmäßig behält Git einen *Commit* für mindesten zwei Wochen, sogar wenn Du Git anweist, den *Branch* zu zerstören, in dem er enthalten ist. Das Problem ist, den entsprechenden SHA1-Wert zu finden. Du kannst Dir alle SHA1-Werte in `.git/objects` vornehmen und ausprobieren ob Du den gesuchten *Commit* findest. Aber es gibt einen viel einfacheren Weg.

Git speichert jeden errechneten SHA1-Wert eines *Commits* in `.git/logs`. Das Unterverzeichnis `refs` enthält den Verlauf aller Aktivitäten auf allen *Branches*, während `HEAD` alle SHA1-Werte enthält, die jemals diese Bezeichnung hatten.

Die letztere kann verwendet werden, um SHA1-Werte von *Commits* zu finden, die sich in einem *Branch* befanden, der versehentlich gestutzt wurde.

Die reflog Anweisung bietet eine benutzerfreundliche Schnittstelle zu diesen Logdateien. Versuche

```
$ git reflog
```

Anstatt SHA1-Werte aus dem reflog zu kopieren und einzufügen, versuche:

```
$ git checkout "@{10 minutes ago}"
```

Oder rufe den fünftletzten *Commit* ab, mit:

```
$ git checkout "@{5}"
```

Siehe in der „Specifying Revisions“ Sektion von **git help rev-parse** für mehr.

Vielleicht möchtest Du eine längere Gnadenfrist für todgeweihte *Commits* konfigurieren. Zum Beispiel:

```
$ git config gc.pruneexpire "30 days"
```

bedeutet, ein gelöschter *Commit* wird nur dann endgültig verloren sein, nachdem 30 Tage vergangen sind und **git gc** ausgeführt wurde.

Du magst vielleicht auch das automatische Ausführen von **git gc** abstellen:

```
$ git config gc.auto 0
```

wodurch *Commits* nur noch gelöscht werden, wenn Du **git gc** manuell aufrufst.

Auf Git bauen

In echter UNIX Sitte erlaubt es Git's Design, dass es auf einfache Weise als Low-Level-Komponente von anderen Programmen benutzt werden kann, wie zum Beispiel grafischen Benutzeroberflächen und Internetanwendungen, alternative Kommandozeilenanwendungen, Patch-Werkzeugen, Import- und Konvertierungswerkzeugen und so weiter. Sogar einige Git Anweisungen selbst sind nur winzige Skripte, wie Zwerge auf den Schultern von Riesen. Mit ein bisschen Handarbeit kannst Du Git anpassen, damit es Deinen Anforderungen entspricht.

Ein einfacher Trick ist es, die in Git integrierte Aliasfunktion zu verwenden, um die am häufigsten benutzten Anweisungen zu verkürzen:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # display current aliases
alias.co checkout
$ git co foo                            # same as 'git checkout foo'
```

Etwas anderes ist der aktuelle *Branch* im Prompt oder Fenstertitel. Die Anweisung

```
$ git symbolic-ref HEAD
```

zeigt den Namen des aktuellen *Branch*. In der Praxis möchtest Du aber das "refs/heads/" entfernen und Fehler ignorieren:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Das `contrib` Unterverzeichnis ist eine Fundgrube von Werkzeugen, die auf Git aufbauen. Mit der Zeit können einige davon zu offiziellen Anweisungen befördert werden. Auf Debian und Ubuntu, findet man dieses Verzeichnis unter `/usr/share/doc/git-core/contrib`.

Ein beliebter Vertreter ist `workdir/git-new-workdir`. Durch cleveres verlinken erzeugt dieses Skript ein neues Arbeitsverzeichis, das seine Versionsgeschichte mit dem original *Repository* teilt:

```
$ git-new-workdir ein/existierendes/repo neues/verzeichnis
```

Das neue Verzeichnis und die Dateien darin kann man sich als *Clone* vorstellen mit dem Unterschied, dass durch die gemeinschaftliche Versionsgeschichte die beiden Versionen automatisch synchron bleiben. Eine Synchronisierung mittels *merge*, *push* oder *pull* ist nicht notwendig.

Gewagte Kunststücke

Heutzutage macht es Git dem Anwender schwer, versehentlich Daten zu zerstören. Aber, wenn man weiß, was man tut, kann man die Schutzmaßnahmen der häufigsten Anweisungen umgehen.

Checkout: Nicht versionierte Änderungen lassen *checkout* scheitern. Um trotzdem die Änderungen zu zerstören und einen vorhandenen *Commit* abzurufen, benutzen wir die *force* Option:

```
$ git checkout -f HEAD^
```

Auf der anderen Seite, wenn Du einen speziellen Pfad für *checkout* angibst, gibt es keine Sicherheitsüberprüfungen mehr. Der angegebene Pfad wird stillschweigend überschrieben. Sei vorsichtig, wenn Du *checkout* auf diese Weise benutzt.

Reset: Reset versagt auch, wenn unversionierte Änderungen vorliegen. Um es zu erzwingen, verwende:

```
$ git reset --hard 1b6d
```

Branch: *Branches* zu löschen scheitert ebenfalls, wenn dadurch Änderungen verloren gehen. Um das Löschen zu erzwingen, gib ein:

```
$ git branch -D dead_branch  # instead of -d
```

Ebenso scheitert der Versuch, einen *Branch* durch ein *move* zu überschreiben, wenn das einen Datenverlust zur Folge hat. Um das Verschieben zu erzwingen, gib ein:

```
$ git branch -M source target  # instead of -m
```

Anders als bei *checkout* und *reset* verschieben diese beiden Anweisungen das Zerstören der Daten. Die Änderungen bleiben im .git Unterverzeichnis gespeichert und können wieder hergestellt werden, wenn der entsprechende SHA1-Wert aus `.git/logs` ermittelt wird (siehe "KOPF-Jagd" oben). Standardmäßig bleiben die Daten mindestens zwei Wochen erhalten.

Clean: Verschiedene git Anweisungen scheitern, weil sie Konflikte mit unversionierten Dateien vermuten. Wenn Du sicher bist, dass alle unversionierten Dateien und Verzeichnisse entbehrlich sind, dann lösche diese gnadenlos mit:

```
$ git clean -f -d
```

Beim nächsten Mal werden diese lästigen Anweisung gehorchen!

Verhindere schlechte *Commits*

Dumme Fehler verschmutzen meine *Repositories*. Am schrecklichsten sind fehlende Dateien wegen eines vergessenen **git add**. Kleinere Verfehlungen sind Leerzeichen am Zeilenende und ungelöste *merge*-Konflikte: obwohl sie harmlos sind, wünschte ich, sie würden nie in der Öffentlichkeit erscheinen.

Wenn ich doch nur eine Trottelversicherung abgeschlossen hätte, durch Verwendung eines *hook*, der mich bei solchen Problemen alarmiert.

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Older Git versions: chmod +x pre-commit
```

Nun bricht Git einen *Commit* ab, wenn es überflüssige Leerzeichen am Zeilenende oder ungelöste *merge*-Konflikte entdeckt.

Für diese Anleitung hätte ich vielleicht am Anfang des **pre-commit** *hook* folgendes hinzugefügt, zum Schutz vor Zerstreutheit:

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Viele Git Operationen unterstützen *hooks*; siehe **git help hooks**. Wir haben den Beispiel *hook* **post-update** aktiviert, weiter oben im Abschnitt Git über HTTP. Dieser läuft immer, wenn der *HEAD* sich bewegt. Das Beispiel *post-update* Skript aktualisiert Dateien, welche Git für die Kommunikation über *Git-agnostic transports* wie z.B. HTTP benötigt.

Aufgedeckte Geheimnisse

Wir werfen einen Blick unter die Motorhaube und erklären, wie Git seine Wunder vollbringt. Ich werde nicht ins Detail gehen. Für tiefer gehende Erklärungen verweise ich auf das englischsprachige Benutzerhandbuch.

Unsichtbarkeit

Wie kann Git so unauffällig sein? Abgesehen von gelegentlichen *Commits* und *Merges* kannst Du arbeiten, als würde die Versionsverwaltung nicht existieren. Das heißt, bis Du sie brauchst. Und das ist, wenn Du froh bist, dass Git die ganze Zeit über Dich gewacht hat.

Andere Versionsverwaltungssysteme zwingen Dich ständig, Dich mit Verwaltungskram und Bürokratie herumzuschlagen. Dateien können schreibgeschützt sein, bis Du einem zentralen Server mitteilst, welche Dateien Du gerne bearbeiten möchtest. Die einfachsten Befehle werden bis zum Schneckentempo verlangsamt, wenn die Anzahl der Anwender steigt. Deine Arbeit kommt zum Stillstand, wenn das Netzwerk oder der zentrale Server weg sind.

Im Gegensatz dazu hält Git seinen Verlauf einfach im `.git` Verzeichnis von Deinem Arbeitsverzeichnis. Das ist Deine eigene Kopie der Versionsgeschichte, damit kannst Du so lange offline bleiben, bis Du mit anderen kommunizieren willst. Du hast die absolute Kontrolle über das Schicksal Deiner Dateien, denn Git kann jederzeit einfach einen gesicherten Stand aus `.git` wiederherstellen.

Integrität

Die meisten Leute verbinden mit Kryptographie die Geheimhaltung von Informationen, aber ein genau so wichtiges Ziel ist es, Informationen zu sichern. Die richtige Anwendung von kryptographischen Hash-Funktionen kann einen versehentlichen oder bösartigen Datenverlust verhindern.

Einen SHA1-Hash-Wert kann man sich als eindeutige 160-Bit Identitätsnummer für jegliche Zeichenkette vorstellen, welche Dir in Deinem ganzen Leben begegnen wird. Sogar mehr als das: jegliche Zeichenfolge, die alle Menschen über mehrere Generationen verwenden.

Ein SHA1-Hash-Wert selbst ist eine Zeichenfolge von Bytes. Wir können SHA1-Hash-Werte aus Zeichenfolgen generieren, die selbst SHA1-Hash-Werte enthalten. Diese einfache Beobachtung ist überraschend nützlich: suche nach *hash chains*. Wir werden später sehen, wie Git diese nutzt um effizient die Dateninteigrität zu garantieren.

Kurz gesagt, Git hält Deine Daten in dem `.git/objects` Unterverzeichnis, wo Du anstelle von normalen Dateinamen nur Identitätsnummern findest. Durch

die Verwendung von Identitätsnummern als Dateiname, zusammen mit ein paar Sperrdateien und Zeitstempeltricks, macht Git aus einem einfachen Dateisystem eine effiziente und robuste Datenbank.

Intelligenz

Woher weiß Git, dass Du eine Datei umbenannt hast, obwohl Du es ihm niemals explizit mitgeteilt hast? Sicher, Du hast vielleicht **git mv** benutzt, aber das ist exakt das selbe wie **git rm** gefolgt von **git add**.

Git stöbert Umbenennungen und Kopien zwischen aufeinander folgenden Versionen heuristisch auf. Vielmehr kann es sogar Codeblöcke erkennen, die zwischen Dateien hin und her kopiert oder verschoben wurden! Jedoch kann es nicht alle Fälle abdecken, aber es leistet ordentliche Arbeit und diese Eigenschaft wird immer besser. Wenn es bei Dir nicht funktioniert, versuche Optionen zur aufwendigeren Erkennung von Kopien oder erwäge einen Upgrade.

Indizierung

Für jede überwachte Datei speichert Git Informationen wie deren Größe, ihren Erstellzeitpunkt und den Zeitpunkt der letzten Bearbeitung in einer Datei die wir als *Index* kennen. Um zu ermitteln, ob eine Datei verändert wurde, vergleicht Git den aktuellen Status mit dem im Index gespeicherten. Stimmen diese Daten überein, kann Git das Lesen des Dateiinhalts überspringen.

Da das Abfragen des Dateistatus erheblich schneller ist als das Lesen der Datei, kann Git, wenn Du nur ein paar Dateien verändert hast, seinen Status im Nu aktualisieren.

Wir haben früher festgestellt, dass der Index ein Bereitstellungsraum ist. Warum kann ein Haufen von Dateistatusinformationen ein Bereitstellungsraum sein? Weil die *add* Anweisung Dateien in die Git Datenbank befördert und die Dateistatusinformationen aktualisiert, während die *commit* Anweisung, ohne Optionen, einen *Commit* nur auf Basis der Dateistatusinformationen erzeugt, weil die Dateien ja schon in der Datenbank sind.

Git's Wurzeln

Dieser *Linux Kernel Mailing List* Beitrag beschreibt die Kette von Ereignissen, die zu Git geführt haben. Der ganze Beitrag ist eine faszinierende archäologische Seite für Git Historiker.

Die Objektdatenbank

Jegliche Versionen Deiner Daten wird in der Objektdatenbank gehalten, welche im Unterverzeichnis `.git/objects` liegt; Die anderen Orte in `.git/` enthalten weniger wichtige Daten: den Index, *Branch* Namen, Bezeichner (*tags*), Konfigurationsoptionen, Logdateien, die Position des aktuellen *HEAD Commit* und so weiter. Die Objektdatenbank ist einfach aber trotzdem elegant, und sie ist die Quelle von Git's Macht.

Jede Datei in `.git/objects` ist ein *Objekt*. Es gibt drei Arten von Objekten, die uns betreffen: *Blob*-, *Tree*- und *Commit*-Objekte.

Blobs

Zuerst ein Zaubertrick. Suche Dir irgendeinen Dateinamen aus. In einem leeren Verzeichnis:

```
$ echo sweet > DEIN_DATEINAME
$ git init
$ git add .
$ find .git/objects -type f
```

Du wirst folgendes sehen: `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

Wie konnte ich das wissen, ohne den Dateiname zu kennen? Weil der SHA1-Hash-Wert von:

```
"blob" SP "6" NUL "sweet" LF
```

aa823728ea7d592acc69b36875a482cdf3fd5c8d ist. Wobei SP ein Leerzeichen ist, NUL ist ein Nullbyte und LF ist ein Zeilenumbruch. Das kannst Du durch die Eingabe von

```
$ printf "blob 6\000sweet\n" | sha1sum
```

kontrollieren.

Git ist *assoziativ*: Dateien werden nicht nach Ihren Namen gespeichert, sondern eher nach dem SHA1-Hash-Wert der Daten, welche sie enthalten, in einer Datei, die wir als *Blob*-Objekt bezeichnen. Wir können uns den SHA1-Hash-Wert als eindeutige Identnummer des Dateiinhalts vorstellen, was sinngemäß bedeutet, dass die Dateien über ihren Inhalt adressiert werden. Das führende `blob 6` ist lediglich ein Vermerk, der sich aus dem Objekttyp und seiner Länge in Bytes zusammensetzt; er vereinfacht die interne Verwaltung.

So konnte ich einfach vorhersagen, was Du sehen wirst. Der Dateiname ist irrelevant: nur der Dateiinhalt wird zum Erstellen des *Blob*-Objekt verwendet.

Du wirst Dich fragen, was mit identischen Dateien ist. Versuche Kopien Deiner Datei hinzuzufügen, mit beliebigen Dateinamen. Der Inhalt von `.git/objects`

bleibt der selbe, ganz egal wieviele Dateien Du hinzufügst. Git speichert den Dateiinhalt nur ein einziges Mal.

Übrigens, die Dateien in `.git/objects` sind mit zlib komprimiert, Du solltest sie also nicht direkt anschauen. Filtere sie durch zpipe -d, oder gib ein:

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

was Dir das Objekt im Klartext anzeigt.

Trees

Aber wo sind die Dateinamen? Sie müssen irgendwo gespeichert sein. Git kommt beim *Commit* dazu, sich um die Dateinamen zu kümmern:

```
$ git commit  # Schreibe eine Bemerkung.
$ find .git/objects -type f
```

Du solltest nun drei Objekte sehen. Dieses mal kann ich Dir nicht sagen, wie die zwei neuen Dateien heißen, weil es zum Teil vom gewählten Dateiname abhängt, den Du ausgesucht hast. Fahren wir fort mit der Annahme, Du hast eine Datei „rose“ genannt. Wenn nicht, kannst Du den Verlauf so umschreiben, dass es so aussieht, als hättest Du es:

```
$ git filter-branch --tree-filter 'mv DEIN_DATEINAME rose'
$ find .git/objects -type f
```

Nun müsstest Du die Datei `.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9` sehen, denn das ist der SHA1-Hash-Wert ihres Inhalts:

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

Prüfe, ob diese Datei tatsächlich dem obigen Inhalt entspricht, durch Eingabe von:

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

Mit zpipe ist es einfach, den SHA1-Hash-Wert zu prüfen:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

Die SHA1-Hash-Wert Prüfung mit *cat-file* ist etwas kniffliger, da dessen Ausgabe mehr als die rohe unkomprimierte Objektdatei enthält.

Diese Datei ist ein *Tree*-Objekt: eine Liste von Datensätzen, bestehend aus dem Dateityp, dem Dateinamen und einem SHA1-Hash-Wert. In unserem Beispiel ist der Dateityp 100644, was bedeutet, dass `rose` eine normale Datei ist, und der SHA1-Hash-Wert entspricht dem *Blob*-Objekt, welches den Inhalt von `rose` enthält. Andere mögliche Dateitypen sind ausführbare Programmdateien, symbolische Links oder Verzeichnisse. Im letzten Fall zeigt der SHA1-Hash-Wert auf ein *Tree*-Objekt.

Wenn Du *filter-branch* aufrufst, bekommst Du alte Objekte, welche nicht länger benötigt werden. Obwohl sie automatisch über Bord geworfen werden, wenn ihre Gnadenfrist abgelaufen ist, wollen wir sie nun löschen, damit wir unserem Beispiel besser folgen können.

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Für reale Projekte solltest Du solche Anweisungen üblicherweise vermeiden, da Du dadurch Datensicherungen zerstörst. Wenn Du ein sauberes *Repository* willst, ist es am besten, einen neuen Klon anzulegen. Sei auch vorsichtig, wenn Du `.git` direkt manipulierst: was, wenn zeitgleich ein Git Kommando ausgeführt wird oder plötzlich der Strom ausfällt? Generell sollten Referenzen mit **git update-ref -d** gelöscht werden, auch wenn es gewöhnlich sicher ist `refs/original` von Hand zu löschen.

Commits

Wir haben nun zwei von drei Objekten erklärt. Das dritte ist ein *Commit*-Objekt. Sein Inhalt hängt von der *Commit*-Beschreibung ab, wie auch vom Zeitpunkt der Erstellung. Damit alles zu unserem Beispiel passt, müssen wir ein wenig tricksen:

```
$ git commit --amend -m Shakespeare  # Ändere die Bemerkung.
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"' # Manipuliere Zeitstempel und Autor.
$ find .git/objects -type f
```

Du solltest nun `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` finden, was dem SHA1-Hash-Wert seines Inhalts entspricht:

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

Wie vorhin kannst Du *zpipe* oder *cat-file* benutzen, um es für Dich zu überprüfen.

Das ist der erste *Commit* gewesen, deshalb gibt es keine Eltern-*Commits*. Aber

spätere *Commits* werden immer mindestens eine Zeile enthalten, die den Eltern-*Commit* identifiziert.

Von Magie nicht zu unterscheiden

Git's Geheimnisse scheinen zu einfach. Es sieht so aus, als müsste man nur ein paar Kommandozeilenskripte zusammenmixen, einen Schuß C-Code hinzufügen und innerhalb ein paar Stunden ist man fertig: eine Mischung von grundlegenden Dateisystemoperationen und SHA1-Hash-Berechnungen, garniert mit Sperrdateien und Synchronisation für Stabilität. Tatsächlich beschreibt dies die früheste Version von Git. Nichtsdestotrotz, abgesehen von geschickten Verpackungstricks, um Speicherplatz zu sparen, und geschickten Indizierungstricks, um Zeit zu sparen, wissen wir nun, wie Git gewandt ein Dateisystem in eine Datenbank verwandelt, das perfekt für eine Versionsverwaltung geeignet ist.

Angenommen, wenn irgendeine Datei in der Objektdatenbank durch einen Laufwerksfehler zerstört wird, dann wird sein SHA1-Hash-Wert nicht mehr mit seinem Inhalt übereinstimmen und uns sagen, wo das Problem liegt. Durch Bilden von SHA1-Hash-Werten aus den SHA1-Hash-Werten anderer Objekte, erreichen wir Integrität auf allen Ebenen. *Commits* sind elementar, das heißt, ein *Commit* kann niemals nur Teile einer Änderung speichern: wir können den SHA1-Hash-Wert eines *Commits* erst dann berechnen und speichern, nachdem wir bereits alle relevanten *Tree*-Objekte, *Blob*-Objekte und Eltern-*Commits* gespeichert haben. Die Objektdatenbank ist immun gegen unerwartete Unterbrechungen wie zum Beispiel einen Stromausfall.

Wir können sogar den hinterhältigsten Gegnern widerstehen. Stell Dir vor, jemand will den Inhalt einer Datei ändern, die in einer älteren Version eines Projekt liegt. Um die Objektdatenbank intakt aussehen zu lassen, müssten sie außerdem den SHA1-Hash-Wert des korrespondierenden *Blob*-Objekt ändern, da die Datei nun eine geänderte Zeichenfolge enthält. Das heißt auch, dass sie jeden SHA1-Hash-Wert der *Tree*-Objekte ändern müssen, welche dieses Objekt referenzieren und demzufolge alle SHA1-Hash-Werte der *Commit*-Objekte, welche diese *Tree*-Objekte beinhalten, zusätzlich zu allen Abkömmlingen dieses *Commits*. Das bedeutet auch, dass sich der SHA1-Hash-Wert des offiziellen HEAD von dem des manipulierten *Repository* unterscheidet. Folgen wir dem Pfad der differierenden SHA1-Hash-Werte, finden wir die verstümmelte Datei, wie auch den *Commit*, in dem sie erstmals auftauchte.

Kurz gesagt, so lange die 20 Byte, welche den SHA1-Hash-Wert des letzen *Commit* repräsentieren sicher sind, ist es unmöglich ein Git *Repository* zu fälschen.

Was ist mit Git's berühmten Fähigkeiten? *Branching*? *Merging*? *Tags*? Nur Kleinigkeiten. Der aktuelle HEAD wird in der Datei `.git/HEAD` gehalten, welche den SHA1-Hash-Wert eines *Commit*-Objekts enthält. Der SHA1-Hash-Wert wird während eines *Commit* aktualisiert, genauso bei vielen anderen Anweisungen. *Branches* sind fast das selbe: sie sind Dateien in `.git/refs/heads`. *Tags*

ebenso: sie stehen in `.git/refs/tags` aber sie werden durch einen Satz anderer Anweisungen aktualisiert.

Git's Mängel

Ein paar Git-Probleme habe ich bisher unter den Teppich gekehrt. Einige lassen sich einfach mit Skripten und *Hooks* lösen, andere erfordern eine Reorganisation oder Neudefinition des gesamten Projekt und für die wenigen verbleibenden Beeinträchtigungen kannst Du nur auf eine Lösung warten. Oder noch besser, anpacken und mithelfen.

SHA1 Schwäche

Mit der Zeit entdecken Kryptographen immer mehr Schwächen an SHA1. Schon heute wäre für finanzkräftige Unternehmen es technisch machbar, Hash-Kollisionen zu finden. In ein paar Jahren hat vielleicht schon ein ganz normaler Heim-PC ausreichend Rechenleistung, um ein Git *Reopsitory* unbemerkt zu korrumpieren.

Hoffentlich stellt Git auf eine bessere Hash Funktion um, bevor die Forschung SHA1 komplett unnütz macht.

Microsoft Windows

Git unter Microsoft Windows kann frustrierend sein:

- Cygwin, eine Linux ähnliche Umgebung für Windows, enthält eine Windows Portierung von Git.
- Git für Windows ist eine Alternative, die sehr wenig Laufzeitunterstützung erfordert, jedoch bedürfen einige Kommandos noch einer Überarbeitung.

Dateien ohne Bezug

Wenn Dein Projekt sehr groß ist und viele Dateien enthält, die in keinem direkten Bezug stehen, trotzdem aber häufig geändert werden, kann Git nachteiliger sein als andere Systeme, weil es keine einzelnen Dateien überwacht. Git überwacht immer das ganze Projekt, was normalerweise schon von Vorteil ist.

Eine Lösung ist es, Dein Projekt in kleinere Stücke aufzuteilen, von denen jedes nur die in Beziehung stehenden Dateien enthält. Benutze **git submodule** wenn Du trotzdem alles in einem einzigen *Repository* halten willst.

Wer macht was?

Einige Versionsverwaltungssysteme zwingen Dich explizit, eine Datei auf irgendeine Weise für die Bearbeitung zu kennzeichnen. Obwohl es extrem lästig ist, wenn es die Kommunikation mit einem zentralen Server erfordert, so hat es doch zwei Vorteile:

1. Unterschiede sind schnell gefunden, weil nur die markierten Dateien untersucht werden müssen.
2. Jeder kann herausfinden wer sonst gerade an einer Datei arbeitet, indem er beim zentralen Server anfragt, wer die Datei zum Bearbeiten markiert hat.

Mit geeigneten Skripten kannst Du das auch mit Git hinkriegen. Das erfordert aber die Mitarbeit der Programmierer, denn sie müssen die Skripte auch aufrufen, wenn sie eine Datei bearbeiten.

Dateihistorie

Da Git die Änderungen über das gesamte Projekt aufzeichnet, erfordert die Rekonstruktion des Verlaufs einer einzelnen Datei mehr Aufwand als in Versionsverwaltungssystemen, die einzelne Dateien überwachen.

Die Nachteile sind üblicherweise gering und werden gern in Kauf genommen, da andere Operationen dafür unglaublich effizient sind. Zum Beispiel ist `git checkout` schneller als `cp -a`, und projektweite Unterschiede sind besser zu komprimieren als eine Sammlung von Änderungen auf Dateibasis.

Der erster Klon

Einen Klon zu erstellen ist aufwendiger als in anderen Versionsverwaltungssystemen, wenn ein längerer Verlauf existiert.

Der initiale Aufwand lohnt sich aber auf längere Sicht, da die meisten zukünftigen Operationen dann schnell und offline erfolgen. Trotzdem gibt es Situationen, in denen es besser ist, einen oberflächlichen Klon mit der `--depth` Option zu erstellen. Das geht wesentlich schneller, aber der resultierende Klon hat nur eingeschränkte Funktionalität.

Unbeständige Projekte

Git wurde geschrieben um schnell zu sein, im Hinblick auf die Größe der Änderungen. Leute machen kleine Änderungen von Version zu Version. Ein einzeiliger Bugfix hier, eine neue Funktion da, verbesserte Kommentare und

so weiter. Aber wenn sich Deine Dateien zwischen aufeinanderfolgenden Versionen gravierend ändern, dann wird zwangsläufig mit jedem *Commit* Dein Verlauf um die Größe des gesamten Projekts wachsen.

Es gibt nichts, was irgendein Versionsverwaltungssystem dagegen machen kann, aber der Standard Git Anwender leidet mehr darunter, weil normalerweise der ganze Verlauf geklont wird.

Die Ursachen für die großen Unterschiede sollten ermittelt werden. Vielleicht können Dateiformate geändert werden. Kleinere Bearbeitungen sollten auch nur minimale Änderungen an so wenig Dateien wie möglich bewirken.

Vielleicht ist eher eine Datenbank oder Sicherungs-/Archivierungslösung gesucht, nicht ein Versionsverwaltungssystem. Ein Versionsverwaltungssystem zum Beispiel ist eine ungeeignete Lösung um Fotos zu verwalten, die periodisch von einer Webcam gemacht werden.

Wenn die Dateien sich tatsächlich konstant verändern, und sie wirklich versioniert werden müssen, ist es eine Möglichkeit, Git in zentralisierter Form zu verwenden. Jeder kann oberflächliche Klone erstellen, die nur wenig oder gar nichts vom Verlauf des Projekts enthalten. Natürlich sind dann viele Git Funktionen nicht verfügbar, und Änderungen müssen als *Patches* übermittelt werden. Das funktioniert wahrscheinlich ganz gut, wenn auch unklar ist, warum jemand die Versionsgeschichte von wahnsinnig instabilen Dateien braucht.

Ein anderes Beispiel ist ein Projekt, das von Firmware abhängig ist, welche die Form einer großen Binärdatei annimmt. Der Verlauf der Firmware interessiert den Anwender nicht und Änderungen lassen sich schlecht komprimieren, so blähen Firmwarerevisionen die Größe des *Repository* unnötig auf.

In diesem Fall sollte der Quellcode der Firmware in einem Git *Repository* gehalten werden und die Binärdatei außerhalb des Projekts. Um das Leben zu vereinfachen, könnte jemand ein Skript erstellen, das Git benutzt, um den Quellcode zu klonen und *rsync* oder einen oberflächlichen Klon für die Firmware.

Globaler Zähler

Verschiedene Versionsverwaltungssysteme unterhalten einen Zähler, der mit jedem *Commit* erhöht wird. Git referenziert Änderungen anhand ihres SHA1-Hash, was in vielen Fällen besser ist.

Aber einige Leute sind diesen Zähler gewöhnt. Zum Glück ist es einfach, Skripte zu schreiben, sodass mit jedem Update das zentrale Git *Repository* einen Zähler erhöht. Vielleicht in Form eines *Tags*, der mit dem SHA1-Hash des letzten *Commit* verknüpft ist.

Jeder Klon könnte einen solchen Zähler bereitstellen, aber der wäre vermutlich nutzlos, denn nur der Zähler des zentralen *Repository* ist für alle relevant.

Leere Unterverzeichnisse

Leere Unterverzeichnisse können nicht überwacht werden. Erstelle eine Dummy-Datei, um dieses Problem zu umgehen.

Die aktuelle Implementierung von Git, weniger sein Design, ist verantwortlich für diesen Pferdefuß. Mit etwas Glück, wenn Git's Verbreitung zunimmt und mehr Anwender nach dieser Funktion verlangen, wird sie vielleicht implementiert.

Initialer *Commit*

Ein klischeehafter Computerwissenschaftler zählt von 0 statt von 1. Leider, bezogen auf *Commits*, hält sich Git nicht an diese Konvention. Viele Kommandos sind mürrisch vor dem intialen *Commit*. Zusätzlich müssen verschiedene Grenzfälle speziell behandelt werden, wie der *Rebase* eines *Branch* mit einem abweichenden initialen *Commit*.

Git würde davon provitieren, einen Null-*Commit* zu definieren: sofort nach dem Erstellen eines *Repository* wird der *HEAD* auf eine Zeichenfolge von 20 Null-Bytes gesetzt. Dieser spezielle *Commit* repräsentiert einen leeren *Tree*, ohne Eltern, irgendwann vielleicht der Vorfahr aller Git *Repositories*.

Würde dann zum Beispiel **git log** ausgeführt, würde der Anwender darüber informiert, dass noch keine *Commits* gemacht wurden, anstelle mit einem fatalen Fehler zu beenden. Das gilt stellvertretend für andere Anweisungen.

Jeder initiale *Commit* ist dann stillschweigend ein Abkömmling dieses Null-*Commits*.

Leider gibt es noch ein paar Problemfälle. Wenn mehrere *Branches* mit unterschiedlichen initialen *Commits* zusammengeführt und dann ein *Rebase* gemacht wird, ist ein manuelles Eingreifen erforderlich.

Eigenarten der Anwendung

Für die *Commits* A und B, hängt die Bedeutung der Ausdrücke "A..B" und "A…B" davon ab, ob eine Anweisung zwei Endpunkte erwartet oder einen Bereich. Siehe **git help diff** und **git help rev-parse**.

Diese Anleitung übersetzen

Ich empfehle folgende Schritte, um diese Anleitung zu übersetzen, damit meine Skripte einfach eine HTML- und PDF-Version erstellen können. Außerdem können so alle Übersetzungen in einem *Repository* existieren.

Clone die Quelltexte, dann erstelle ein Verzeichnis mit dem Namen des IETF Sprachkürzel der übersetzten Sprache: siehe den W3C Artikel über Internationalisierung. Zum Beispiel, Englisch ist "en", Japanisch ist "ja". Kopiere alle `txt`-Dateien aus dem "en"-Verzeichnis in das neue Verzeichnis und übersetze diese.

Um zum Beispiel die Anleitung auf Klingonisch zu übersetzen, musst du folgendes machen:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" ist das IETF Sprachkürzel für Klingonisch.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # übersetze diese Datei.
```

und das machst du für jede txt-Datei.

Bearbeite das Makefile und füge das Sprachkürzel zur Variable `TRANSLATIONS` hinzu. Nun kannst Du Deine Arbeit jederzeit wie folgt überprüfen:

```
$ make tlh
$ firefox book-tlh/index.html
```

Committe Deine Änderungen oft, und wenn Du fertig bist, gib bitte Bescheid. GitHub hat eine Schnittstelle, die das erleichtert: Erzeuge Deinen eigene *Fork* vom "gitmagic" Projekt, *pushe* Deine Änderungen, dann gib mir Bescheid, Deine Änderungen zu *mergen*.