2 Erste Schritte mit der Kommandozeile

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

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

Erste Schritte mit SourceTree → *Seite 23*

2.1 Git einrichten

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

http://git-scm.com/download

2.2 Ein paar Hinweise für Windows-User

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

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

Installation von Git unter Windows

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

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

Arbeiten mit der Windows-Eingabeaufforderung

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

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

Arbeiten mit der Git-Bash unter Windows

Tipp: Autovervollständigung

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

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

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

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

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

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

2.3 Git einrichten

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

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

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

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

2.4 Das erste Projekt mit Git

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

Abb. 2–1
Unser Beispielprojekt

Tipp: Sicherungskopie nicht vergessen!

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

Projektverzeichnis

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

```
/home/hmustermann/projekte
```

oder

```
C:\Users\hmustermann\projekte
```

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

```
/c/Users/hmustermann/projekte
```

Repository anlegen

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

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

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

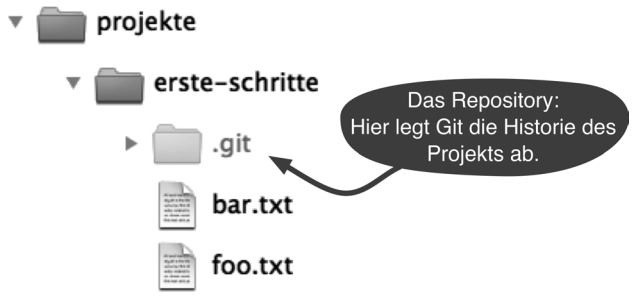

Abb. 2–2
Das Repository-Verzeichnis

Das erste Commit

Als Nächstes können Sie die Dateien `foo.txt` und `bar.txt` ins *Repository* bringen. Eine Projektversion nennt man bei Git ein *Commit*, und sie wird in zwei Schritten angelegt. Als Erstes bestimmt man mit dem `add`-Befehl, welche Dateien in das nächste *Commit* aufgenommen werden sollen. Danach überträgt der `commit`-Befehl die Änderungen ins *Repository* und vergibt einen 40-stelligen sogenannten *Commit-Hash*, der das neue Commit identifiziert. Git zeigt hier nur die ersten Stellen `2f43cf0` an.

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

Status abfragen

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

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

Wenn Sie mehr Details wissen wollen, zeigt Ihnen der `diff`-Befehl jede geänderte Zeile an.

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

Die Ausgabe im diff-Format empfinden viele Menschen als schlecht lesbar, sie kann dafür aber gut maschinell verarbeitet werden. Es gibt glücklicherweise eine ganze Reihe von Tools und Entwicklungsumgebungen, die Änderungen übersichtlicher darstellen können (Abbildung 2–3). Dazu nutzt man statt des `diff`-Befehls den `difftool`-Befehl.

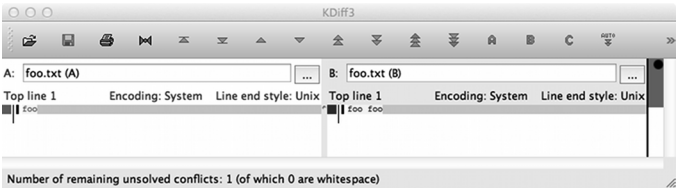

Abb. 2–3
Diff-Darstellung in grafischem Tool (kdiff3)

Ein Commit nach Änderungen

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

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

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

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

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

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

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

Historie betrachten

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

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

    Einiges geändert.

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

    Beispielprojekt importiert.
```

2.5 Zusammenarbeit mit Git

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

[2] *http://www.nongnu.org/cvs/*

[3] *http://subversion.apache.org/*

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

Repository klonen

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

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

Im Verzeichnis `erste-schritte-klon` liegt nun eine Kopie der Projektstruktur wie in Abbildung 2–4 abgebildet.

Änderungen aus einem anderen Repository holen

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

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

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

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

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

```
> cd /projekte/erste-schritte-klon
> git add bar.html
```

Abb. 2–4
Das Beispielprojekt und sein Klon

```
> git commit --message "Eine Änderung im Klon."
> git log --oneline

1fcc06a Eine Änderung im Klon.
7ac0f38 Einiges geändert.
2f43cd0 Beispielprojekt importiert.
```

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

```
> cd /projekte/erste-schritte-klon
> git pull

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

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

Branches zusammenführen → *Seite 73*

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

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

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

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

Änderungen aus beliebigen Repositorys abholen

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

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

Ein Repository für den Austausch erstellen

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

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

```
> git clone --bare /projekte/erste-schritte
                   /projekte/erste-schritte-bare.git

Cloning into bare repository erste-schritte-bare.git...
done.
```

Änderungen mit Push hochladen

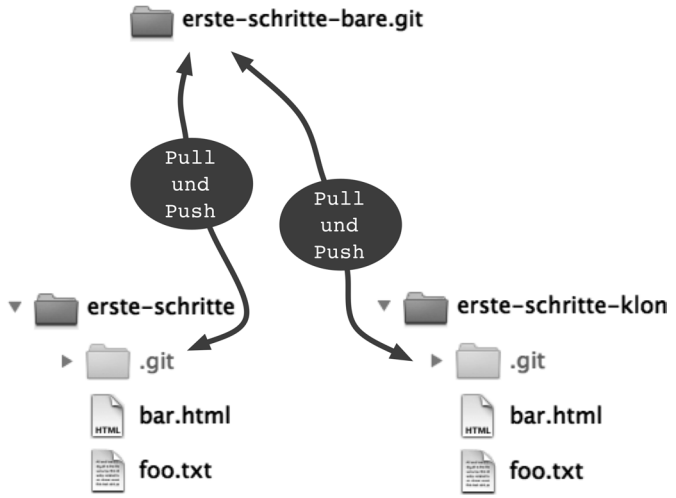

Abb. 2–6
Austausch über ein gemeinsames Repository

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

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

Dieses *Commit* übertragen Sie dann mit dem `push`-Befehl in das zentrale *Repository* (Abbildung 2–6). Dieser Befehl erlaubt, wie der `pull`-Befehl, den Pfad zum *Repository* und einen *Branch* als Parameter anzugeben.

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

Push verweigert! Was tun? → *Seite 109*

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

Pull: Änderungen abholen

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

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

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

2.6 Zusammenfassung

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

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

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

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

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