Kurzanleitung Git

1 Grundlagen

Was ist Git? Git ist ein verteiltes Versionskontrollsystem, das Beginn 2005 von Linus Torvalds entwickelt wurde. Git ist Open Source. Durch ein Versionskontrollsystem ist es Entwicklern möglich, Dateien und Verzeichnisse über einen längeren Zeitraum hinweg zu verwalten. Dabei ist der Unterschied zum gewöhnlichem Datenspeicher, dass jede Version einer Datei gespeichert wird und man (falls notwendig) auch auf ältere Versionen einer Datei oder eines Projektes zugreifen kann.

Warum Git? Die Entwicklung einer Software ist in der Regel eine komplexe Aufgabe. Vielleicht wurde eine Änderung gemacht, die sich im Nachhinein als falsch herausstellt oder aber man möchte den Verlauf eines Projektes im Auge behalten. Außerdem wird Software in der Regel arbeitsteilig von verschiedenen Entwicklern oder Entwicklergruppen erstellt.

Mit einem Versionsverwaltungssystem kann man sowohl das zeitliche Aufeinanderfolgen von Änderungen als auch das parallele Arbeiten mehrerer Entwickler koordinieren. Versionsverwaltung stellt somit ein wichtiges Werkzeug im Softwareentwicklungsprozess dar.

Viele Versionsverwaltungssysteme sind *zentralisiert*, d.h. es gibt ein zentrales Repository mit allen Dateien und Versionsständen. Dabei arbeiten die Entwickler normalerweise mit einer lokalen Kopie der aktuellen Version und bringen ihre Änderungen dann in das zentrale Repository ein. Im Unterschied dazu ist Git ein *verteiltes* Versionsverwaltungssystem. D.h. dass jeder Entwickler ein *lokales* Repository hat und damit arbeiten kann. Wenn er eine zusammengehörende Gruppe von Änderungen gemacht und überprüft hat, kann er sein lokales Repository mit entfernten Repositories synchronisieren. Durch dieses verteilte Vorgehen von Git sind bestimmte Arbeitsabläufe in der arbeitsteiligen Entwicklung von Software möglich, die in zentralisierten Versionsverwaltungssystemen nur schwer zu erreichen wären. Insbesondere propagiert die Git-Gemeinde, Branches (siehe Abschnitt 3) einzusetzen, was sich als ein sehr sinnvolles Instrument herausstellt.

1.1 Lokales Arbeiten mit Git

Bei Git hat jeder Entwickler eines Projektes nicht nur die aktuelle Version des Quellcodes, sondern gleich eine Kopie des ganzen Repositories auf seinem lokalem Rechner. Git selbst unterscheidet dabei zwischen drei Bereichen: Dem Arbeitsverzeichnis (*working directory*), dem überwachten Bereich (*staging area*, auch Index genannt) und dem Repository selbst.

- Das *Arbeitsverzeichnis* enthält den aktuellen Stand des Projektes und alle noch nicht erfassten Änderungen.
- Der *überwachte Bereich* beinhaltet alle Änderungen, die bei einem Commit (=Einbringen einer Aktualisierung in das Repository) einer neuen Version dem Repository hinzugefügt werden.
- Das *Repository* beinhaltet alle Versionen des Projektes. Diese werden samt aller Verwaltungsinformationen für das Repository im Ordner .git gespeichert.

Dieses Konzept ist in Abbildung 1 dargestellt.

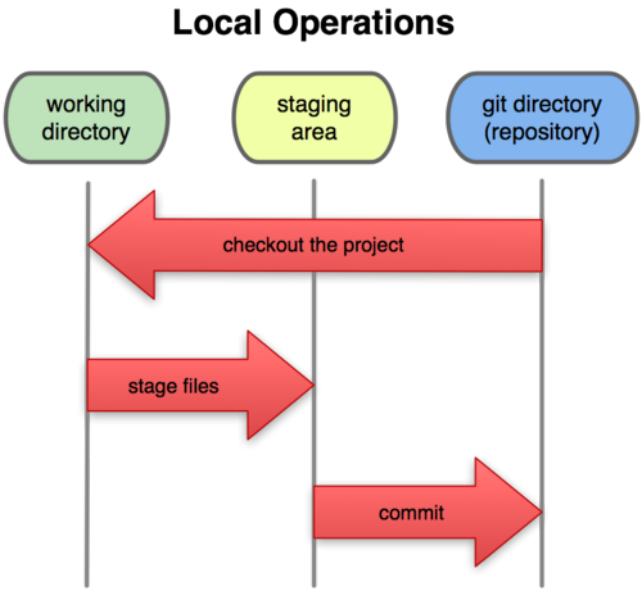

Abbildung 1: Bereiche bei Git [1]

An der Abbildung sieht man, dass es zwei Arbeitsrichtungen gibt.

1. Das *Auschecken* eines Projekts in das Arbeitsverzeichnis geschieht automatisch beim Anlegen des Projektes, sei es lokal durch `git init` oder durch das „Klonen“ eines entfernten Projektes durch `git clone` (mehr dazu später). Dann arbeitet man mit der aktuellen Version, der sogenannten HEAD-Revision.
2. Das Hinzufügen von Neuerungen erfolgt über den Befehl `git add`. Dadurch werden Dateien in den überwachten Bereich verschoben.
3. Das *Einchecken* (auch „denglisch“ *Committen* genannt) der Dateien des überwachten Bereichs erfolgt über den Befehl `git commit`. Dadurch wird eine neue Version des Projekts im Repository erzeugt. Dabei verwendet Git zur Identifikation einer Version einen SHA-Hashschlüssel – im Unterschied zu anderen Codeverwaltungssystemen wie Subversion, die mit Versionsnummern arbeiten.

1.2 Arbeiten mit entfernten Repositories

Um Daten mit anderen Entwicklern auszutauschen, muss ein entferntes Repository erstellt werden. Ein von mehreren Entwicklern verwendetes Repository wird üblicherweise von einem Git-Server verwaltet. Es gibt im Internet solche Plattformen, wie z.B. github.com. An der THM gibt es eine Installation von Gitorious mit der Adresse `https://scm.thm.de/` auf der Studierende der Hochschule Git-Repositories einrichten und verwalten können. Wie man das macht, zeigen wir später, siehe 2.2.

Beim Arbeiten mit entfernten Repositories werden zwei Bereiche unterschieden: zum einen das lokale Repository (oder einfach nur Repository) und das *entfernte Repository*.

Die Arbeit an einem Projekt, das auf etwa `scm.thm.de` verwaltet wird, erfolgt durch folgende Schritte:

1. Mit `git clone` wird ein lokales Repository erzeugt, das dann genau der aktuellen Version des entfernten Repostories auf `scm.thm.de` entspricht.
2. Nun kann mit diesem Repository lokal gearbeitet werden. Alle lokalen Änderungen und Commits sind für andere Entwickler desselben Projekts nicht sichtbar.
3. Möchte man eigene Änderungen für das entfernte Repository zur Verfügung stellen, wird die lokale Kopie „gepusht“ und in das entfernte Projekt mittels einem `git push` eingebracht. Achtung! Es sollte darauf geachtet werden, das bei einem *push* immer nur Code eingepflegt wird, der „compile-clean“ ist, also nicht dazu führt, dass ein Build scheitert.
4. Sollen Änderungen, die andere am entfernten Repository vorgenommen wurden auch lokal verwendet werden, dann holt man sich diese Änderung mit dem Befehl `git pull`.

2 Git anhand eines Fallbeispiels

Information In dieser Kurzanleitung wird die Nutzung von Git anhand eines Fallbeispiels in einer Bash-Konsole erläutert. Eine Installationsanletung zu Git ist zu finden unter `http://book.git-scm.com/2_installing_git.html`, `http://www.macnotes.de/2010/01/06/git-auf-mac-os-x-teil-1-installation` oder `http://cldotde.wordpress.com/2010/12/01/git-fur-windows-installieren-und-nutzen`.

2.1 Git lokal

Erste Einstellungen In unserem Beispiel hier startet der Entwickler „Walter Tichy“[1] auf seinem Rechner das Projekt. Dazu sollten ein paar Grundeinstellungen vorgenommen werden. So werden `user.name` und `user.email` von Git bei Commits verwendet.

```
$ git config --global user.name "Walter Tichy"
$ git config --global user.email "Walter.Tichy@mni.thm.de"
```

Nun ist Git fertig eingerichtet.

Ein Projekt erstellen und die Verwaltung starten Jetzt wird ein neues Projekt „example“ erstellt. Dazu erstellen wir ein leeres Verzeichnis „example“ und initialisieren es mit Git.

```
$ mkdir example
$ cd example
$ git init
Initialized empty git repository in $/example
```

Git hat jetzt im Unterverzeichnis `.git` das lokale Repository angelegt.

[1] Walter Tichy ist der Entwickler des „klassischen“ Versionsverwaltungsystems RCS, sein Internetauftritt: `http://www.ipd.uka.de/Tichy`

Hinzufügen und Einchecken von Dateien Nachdem das Projekt initialisiert wurde, wollen wir eine Datei mit dem Inhalt "Hallo Welt"hinzufügen.

```
$ echo "Hallo Welt">exfile
$ git add exfile
$ git commit -m "Kommentar des ersten Commits"
[master (root-commit) 8b2e852] Kommentar des ersten commits
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 exfile
```

Diese Befehle haben die Datei `exfile` zum überwachten Bereich hinzugefügt und dann beim Commit in eine neue Version des Projektes eingefügt. Die neue Version ist im Ordner „.git“ zu finden.

Achtung! Beim Einchecken werden nur die Versionen der Dateien eingecheckt, die vorher in den überwachten Bereich gebracht wurden. Wir demonstrieren das anhand des folgenden Beispiels. Wir fügen die Datei „exfile“ hinzu und verändern dann kurz vor dem Commit ihren Inhalt.

```
$ git add exfile
$ echo "Hallo">exfile
$ git commit -m "zweiter commit"
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in  working directory)
#
# modified:   exfile
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# file
no changes added to commit (use "git add" and/or "git commit -a")
```

Wir sehen also, dass wir vor einem Commit die Daten zum überwachtem Bereich hinzufügen müssen.

Weitere nützliche Befehle

In diesem Abschnitt werden weitere nützliche Befehle erläutert.

Veränderungen erstellen und überwachen Wer welche Veränderung geschrieben hat, welchen Versions-Hash diese Veränderung hat und wie der entsprechende Entwickler die Änderung dokumentiert hat, findet man durch folgenden Befehl heraus:

```
$ git log
commit 8b2e852bcc716d600343b6a0692c8fb1064e1ec0
Author: Walter <Walter.Tichy@mni.thm.de>
Date:   Wed Jan 18 13:17:37 2012 +0100

    Kommentar des ersten commits
```

```
commit 9a91552bcc716d600343b6a0692c8fb1064e1ec0
Author: Walter <Walter.Tichy@mni.thm.de>
Date:   Wed Jan 18 14:22:32 2012 +0100

  Noch ein Commit
```

Dieser Befehl zeigt alle bisherigen Versionen mit ihrem Versionshash, Kommentar, Datum des Einchecken und dem verantwortlichem Entwickler an.

Git verwendet für die Identifikation von Versionen einen Hashcode, der aus dem Inhalt der Dateien der Version erzeugt wird. Er hat 40 hexadezimale Stellen, wie z.B. oben `8b2e852bcc716d6...`. Um mit Versionshashs in Git zu arbeiten, genügt es die führenden Stellen des Hashs zu verwenden, in der Regel genügen die ersten 5 Stellen.

Löschen von Dateien Das Löschen von Dateien aus einem Repository passiert durch folgenden Befehl

```
$ git rm exfile
rm 'exfile'
```

Änderungen überblicken Um den Unterschied zwischen Versionen eines Projekts zu ermitteln, benutzt man den Befehl

```
$ git diff 8b2e8 9a915
diff --git a/exfile b/exfile
@@ -1 +1 @@
-Hallo Welt
+Hallo
```

Dieser zeigt einem die Unterschiede zwischen den zwei angegebenen Versionen an. Dabei ist a die `exfile` aus der Version `8b2e8` und b die `exfile` aus der Version mit dem Hash `9a915`. Die Zeile `@@ -1 +1 @@` steht dafür das eine Zeile gelöscht und eine Zeile eingefügt wurde. Diese werden dann in den Zeilen danach angegeben.

Alte Versionen wiederherstellen Für das Wiederherstellen einer Datei einer gewissen Version:

```
$ git checkout 9a9155 exfile
```

Falls die Dateien aus dem überwachten Bereich nicht mehr überwacht werden sollen, kann man folgenden Befehl verwenden:

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

Um sein Projekt wieder auf den Stand einer bestimmten Version zurückzusetzen, wird folgender Befehl benutzt:

```
$ git revert 8b2e8
```

Status der Dateien überprüfen Den Überblick über die Dateien im Arbeitsverzeichnis erhält man durch folgenden Befehl:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# filed
# nfile
nothing added to commit but untracked files present (use "git add" to track)
```

Nur wesentliche Dateien berücksichtigen In der Softwareentwicklung entstehen oft Zwischendateien z.B. beim Compilieren, die man nicht durch Git verwalten lassen möchte. Um Git so zu konfigurieren, dass solche Dateien nicht berücksichtigt, also ignoriert werden, gibt es die Datei „.gitignore“. Diese kann z.B. so aussehen:

```
$ cat .gitignore
# Ein Kommentar
*.class   # Keine Dateien die mit .class enden
*.jar   # Keine .jar Dateien
!lib.jar  # Die datei lib.jar wird trozdem gestaged
build/    # Alle Datein aus dem build/ Ordner ignorieren
doc/*.txt # Ignoriere doc/notes.txt, aber nicht doc/server/arch.txt
```

Markieren (*Taggen*) von Versionen Manchmal hat man in dem Verlauf einer Projektentwicklung das Bedürfniss eine Version besonders zu kennzeichnen oder hervorzuheben. Dies macht man z.B. mit einer Version, die man an Kunden ausgeliefert hat.

Dies kann durch „Tagging“ ermöglicht werden. Ein Tag ist einfach eine Markierung, die eine bestimmte Version eines Projektes kennzeichnet.

Tags werden benutzt, um bestimmte Entwicklungsstände, wie zum Beispiel die Fertigstellung eines Moduls oder eine Auslieferung des Projektes zu einem bestimmten Zeitpunkt zu markieren.

```
$ git tag -a v0.1 -m "meine version 0.1"
$ git tag
v0.1
```

2.2 Git in einem Team einsetzen

In den bisherigen Beispielen wurde die lokale Nutzung von Git beschrieben. In den meisten Fällen wird allerdings im Team entwickelt. Um die Entwicklung im Team zu ermöglichen, wird ein entferntes Repository, benötigt welches die zentrale Codebasis darstellt.

Einrichten eines entfernten Repositories Ein entferntes Repository erlaubt es einem Entwickler, Daten mit andern Entwicklern auszutauschen.

Es gibt diverse Internet-Portale, bei denen man kostenlos ein entferntes Repository einrichten kann. Ein Beispiel hierfür ist `www.github.com`. Wir werden in diesem Beispiel die Plattform Gitorius an der THM verwenden, welche unter `https://scm.thm.de/` zu finden ist. Eine Registierung ist hier nicht nötig, da die Nutzerkennung der TH benutzt werden kann.

Nach dem Anmelden in Gitorious wird man auf das Dashboard geleitet. Um ein entferntes Repository einzurichten, muss zunächst ein Projekt angelegt werden. Das Projekt lässt sich direkt im Dashboard anlegen, wie in Abbildung 2 dargestellt.

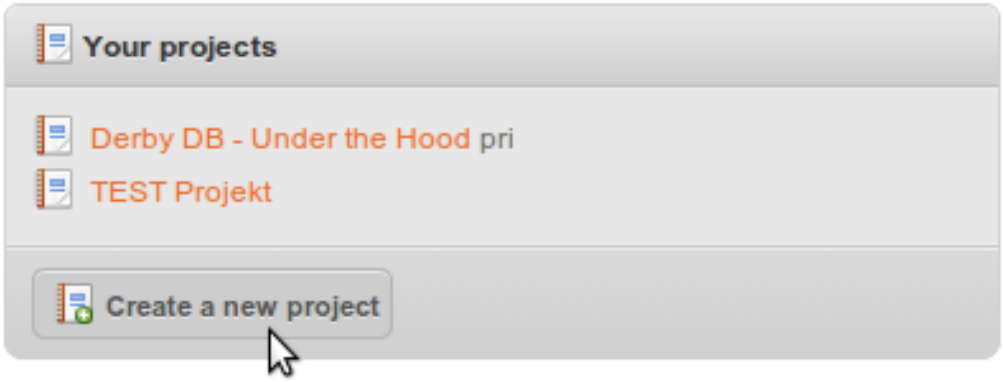

Abbildung 2: Projekt in Gitorious anlegen

Nun tragen wir die nötigen Details ein. Nach Beendigung dieses Schritts ist das Projekt mit dem entfernten Repository angelegt.

Mitarbeiter zum entfernten Repository einladen Um weitere Teammitglieder zum Repository einzuladen, müssen wir im Gitorious (`https://scm.thm.de/`) angemeldet sein und bereits ein Projekt mit einem Repository angelegt haben. Im Dashboard lässt sich das angelegte Repository finden. Nun wählen wir es aus, wie auf Abbildung 3 ersichtlich.

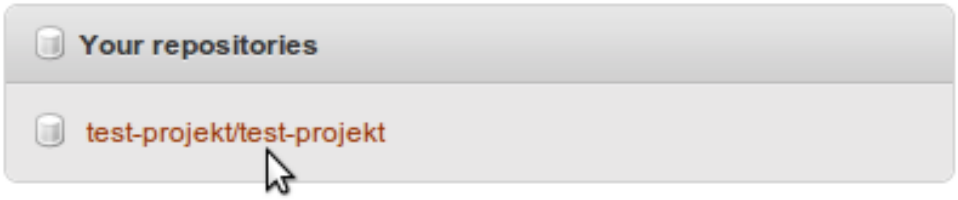

Abbildung 3: Entferntes Repository auswählen

Um die Mitarbeiter zu verwalten, wählen wir den Punkt „Manage collaborators“ aus, wie auf Abbildung 4.

Hier lassen sich Einstellungen zu den bereits bestehenden Mitarbeitern machen und neue anlegen. Um einen neuen Mitarbeiter hinzuzufügen, wählen wir den Punkt „Add collaborators“ (Abbildung 5).

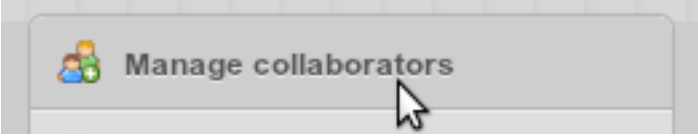

Abbildung 4: Mitarbeiter verwalten

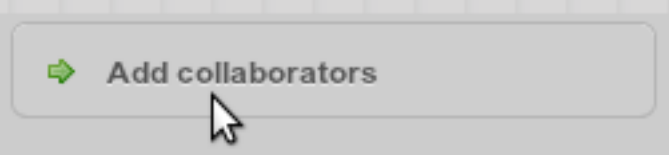

Abbildung 5: Mitarbeiter hinzufügen

Nun können einzelne Mitarbeiter oder auch Teams zum Repository eingeladen werden. Um einen Nutzer einzuladen müssen wir lediglich dessen TH-Benutzerkennung in das Eingabefeld tippen (Abbildung 6).

Abbildung 6: Nutzerkennung eintippen

Anschließend können wir die Rechte des Mitarbeiters festlegen und ihn zum Repository hinzufügen.

Lokales Repository mit entferntem Repository verbinden Git unterstützt die Protokolle SSH, GIT und HTTP zur Synchronisation von lokalen und entfernten Repositories. Im folgenden Beispiel wird erläutert, wie man sein lokales Repository über SSH mit einem entfernten Repository von Gitorious verbindet. Als Voraussetzung für die folgenden Schritte ist es notwendig, bereits zu einem entferntes Repository von Gitorious eingeladen worden zu sein oder selbst ein eigenes angelegt zu haben. Damit wir uns per SSH zum Server verbinden können, müssen wir zunächst ein SSH-Schlüsselpaar erstellen. Folgender Befehl generiert ein Schlüsselpaar:

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/accountname/.ssh/id_dsa):
Created directory '/home/accountname/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/accountname/.ssh/id_dsa.
Your public key has been saved in /home/accountname/.ssh/id_dsa.pub.
The key fingerprint is:
```

```
d3:aa:e3:22:c9:f3:7e:de:e9:10:83:27:5f:6d:fb:63 accountname@host
The key's randomart image is:
+--[ DSA 1024]----+
|                 |
|                 |
|                 |
|     .   o       |
|    o + S +      |
|     + + + .     |
| . .  o . .      |
|  = . o+ . .E    |
|   =o=+o+  ...   |
+-----------------+
```

Der öffentliche Teil des Schlüssels liegt nun im Homeordner im Unterverzeichnis `.ssh/id_sda.pub` und kann mit folgendem Befehl ausgelesen werden:

```
$ cat ~/.ssh/id_dsa.pub
ssh-dss AAAAB3NzaC1kc3MAAACBAPb19FkzSXpjC0RrynjIv/bhS4nV54aR5QQOeMP09GBdgZff1F/5wqwpEQUzw/u8lEN/uojN6Hw
```

Nun müssen wir den öffentlichen Schlüssel in Gitorious eintragen. Dazu loggen wir uns in Gitorious (`https://scm.thm.de/`) ein und klicken im Dashboard auf den Punkt „Manage SSH keys“ (Abbildung 7).

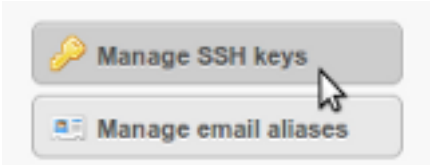

Abbildung 7: SSH-Keys verwalten

Anschließend klicken wir auf den Knopf „Add SSH key“ (Abbildung 8).

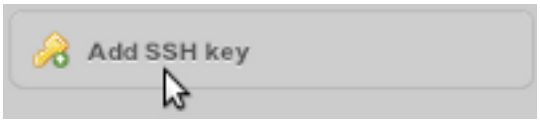

Abbildung 8: Hinzufügen eines SSH-Keys

Nun wird der vorher generierte öffentliche Schlüssel in das Eingabefeld kopiert und bestätigt. Die Serverseite ist nun konfiguriert.

Jetzt beginnen wir mit der Konfiguration des lokalen Rechners. Die folgenden zwei Befehle setzen Namen und E-Mail.

```
$ git config --global user.name "Walter Tichy"
$ git config --global user.email "Walter.Tichy@mni.thm.de"
```

Möchte man Daten in das entfernte Repository einchecken, kann man die folgenden Befehle benutzen:

```
$ git checkout master
$ git remote add origin gitorious@scm.thm.de:test-projekt/repository.git
$ git push origin master
```

Der Befehl `git checkout master` setzt das lokale Repository auf den Hauptenwicklungszweig, auch Masterbranch genannt.

Der Befehl `git remote add origin gitorious@scm.thm.de:test-projekt/repo.git` gibt an, in welches entfernte Repository das lokale Projekt gepushed werden soll.

`git push origin master` schiebt die Daten des lokalen Projekts in das entfernte Repository.

Der Link zum Repository besteht aus dem Prefix: „gitorious@scm.thm.de:“, danach kommt der Projektname: „test-projekt/“ und am Ende der Name des Repositories: „repository.git“. Der Link kann auch direkt aus Gitorious kopiert werden, er ist in der Projektansicht auffindbar.

Abbildung 9: Auswahl des Links zum Repository

Nun können wir die Synchronisation zwischen lokalem und entferntem Repository starten. Wenn bereits Daten im entfernten Repository existieren, welche man ins lokale Repository übertragen möchte, kann man folgenden Befehl ausführen:

```
$ git clone gitorious@scm.thm.de:test-projekt/repository.git
```

3 Arbeiten mit Branches

3.1 Git, Master und Branches

Das Repository wird von Git wie ein Dateisystem mit Historie verwaltet. Jeder Zustand der Dateien in einem Projekt hat einen Versionshash und befindet sich in einer verketteten Liste, die alle Vorgängerversionen in ihrer historischen Reihenfolge enthält.

Eine einfache (lineare) Situation könnte aussehen wie in Abbildung 10:

Die Abbildung zeigt die Versionen `C0` bis `C3`, die durch die Initialisierung `C0` und drei Commits erstellt wurden. Der „Zeiger“ `master` bezeichnet bei Git den Hauptentwicklungszweig und ist auf die aktuelle Version gerichtet. Der Zeiger `HEAD` zeigt stets auf den Zustand, mit dem wir aktuell arbeiten.

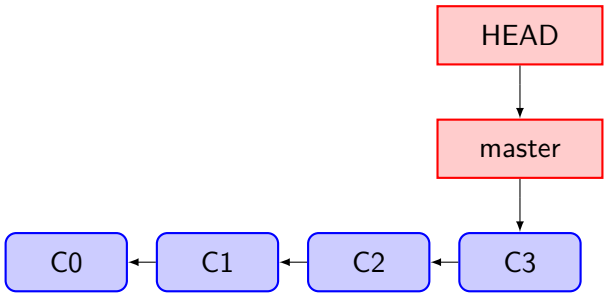

Abbildung 10: Linearer Ablauf eines Projektes

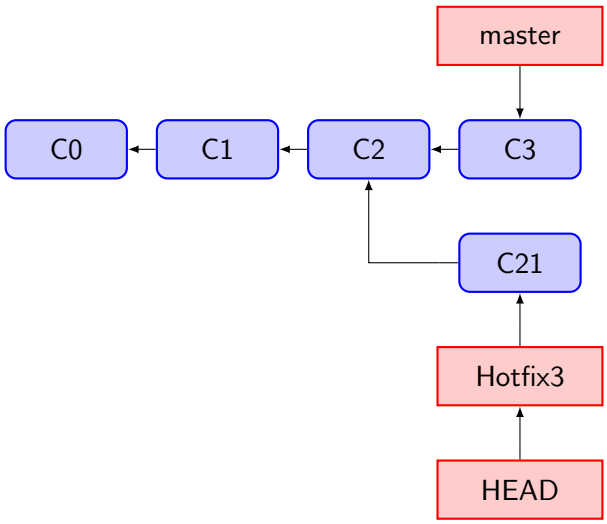

Abbildung 11: Ein Branch bei C2

An jeder Stelle der Versionskette kann eine Abzweigung, ein *Branch* erzeugt werden. Zum Beispiel kann im Zustand `C2` der Branch `Hotfix3` angelegt werden und diesem Branch eine Änderung committed werden. Die Befehle dafür sind:

```
$ git checkout C2
Switched to commit C2
$ git branch "Hotfix3"
$ git checkout "Hotfix3"
$ echo "some fixe">fix.java
$ git commit -ma "Hotfix erstellt"
```

Daraufhin sieht unsere Historie aus wie in Abbildung 11.

Mit git checkout `<branch>` legt man fest, in welchen Branch man arbeiten möchte.

Git bietet diverse Möglichkeiten Branches wieder in den Hauptzweig der Entwicklung zu integrieren. Im einfachstem Fall entstehen dabei keine Konflikte, weil die Änderungen sich nicht überschneiden. Wir werden das Auflösen eines Konfliktes später besprechen (siehe 3.3) und gehen hier davon aus, dass es keinen Konflikt gab.

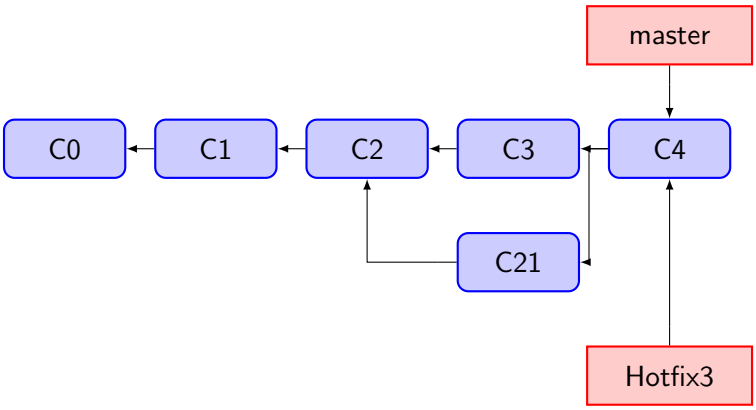

Abbildung 12: Historie nach dem Merge von Hotfix3

In userem Beispiel wird der *Merge*, d.h. die Integration des Branches `Hotfix3` in den Hauptentwicklungszweigs folgendermaßen durchgeführt:

```
$ git checkout master
$ git merge "Hotfix3"
Merges branch "Hotfix3" into master
# 0 conflicts
```

Das führt zu der Historie in Abbildung 12.

3.2 Szenario für das Arbeiten mit Branches

In zentralisierten Versionsverwaltungssystemen wird versucht Branches zu vermeiden. Die Git-Gemeinde allerdings empfiehlt die Verwendung von Branches und propagiert folgende Arbeitsweise:

- Markierung eines bestimmten Standes durch Tags, zum Beispiel „Stand 1.2 für Kunden“
- Weiterentwicklungen in Branches, zum Beispiel „Feature xy“
- Parallele Weiterenwicklungen in anderen Branches, zum Beispiel in Branch „Hotfix 42“

Der Vorteil dieses Vorgehens besteht darin, dass alle Änderungen, die zu *einer* Aufgabe gehören, wie z.B. Hotfix 42 *zusammen* in den Hauptzweig wieder eingebracht werden. In der Regel geht man dabei so vor, dass Branches im jeweils lokalen Repository angelegt werden. Nach Überprüfung der Änderungen werden sie lokal zum master hinzu „gemergt“ und erst dann wird der master in das „offizielle“ entfernte Repository gepushed.

Des Weiteren erlaubt die Trennung des lokalen Repository vom „offiziellem“ Stand eines Projektes das Experimentieren mit Änderungen ohne Angst haben zu müssen, andere Entwickler dadurch zu beeinträchtigen. Git ist deshalb besonders gut für agile Entwicklungsprozesse geeignet.

3.3 Umgang mit Merge-Konflikten

Es kann natürlich sein, dass sich Änderungen in veschiedenen Entwicklungszweigen überschneiden, so dass bei dem Zusammenführen Konflikte auftreten können. Es ist dann die Aufgabe der Entwickler, Konflikte zu lösen, ehe der Merge abgeschlossen werden kann.

In Git wird ein spezieller Commit für den Merge angelegt. Tritt nun ein Konflikt auf, werden diese in der Datei gekennzeichnet. Eine solche Konfliktstelle sieht zum Beispiel so aus:

```
>>>>>> HEAD
  s=s.append("\n\r");
======
  s=s+"\n\r";
<<<<<< 9t43g566
```

Um diese Konfliktstelle zu lösen, muss die richtige Fassung vom Entwickler erkannt werden und der falsche Code mitsamt allen Sonderzeichen gelöscht werden.

Durch einen weiteren Commit kann dann der Merge abgeschlossen werden.

4 Git in Eclipse

Zur Nutzung von Git in Eclipse kann das Plugin „eGit“ installieren. Dazu geht müssen folgende Schritte ausgeführt werden:

- Menüpunkt „Help“ anklicken
- „Install New Software“ auswählen
- In der Eingabefläche „Work with:“ die URL „http://download.eclipse.org/egit/updates“ eintragen und mit Enter bestätigen
- Im Auswahlfeld den „Eclipse Git Team Provider“ wählen wie in Abbildung 13
- Mit „Next >“ bestätigen
- Warten bis die Abhängigkeiten analysiert wurden und anschließend erneut mit „Next >“ bestätigen
- Lizenzbedingungen akzeptieren und mit „Finish“ die Installation starten
- Installation durchlaufen lassen und Eclipse neu starten

Nach dem Neustart von Eclipse ist das eGit Plugin benutzbar.

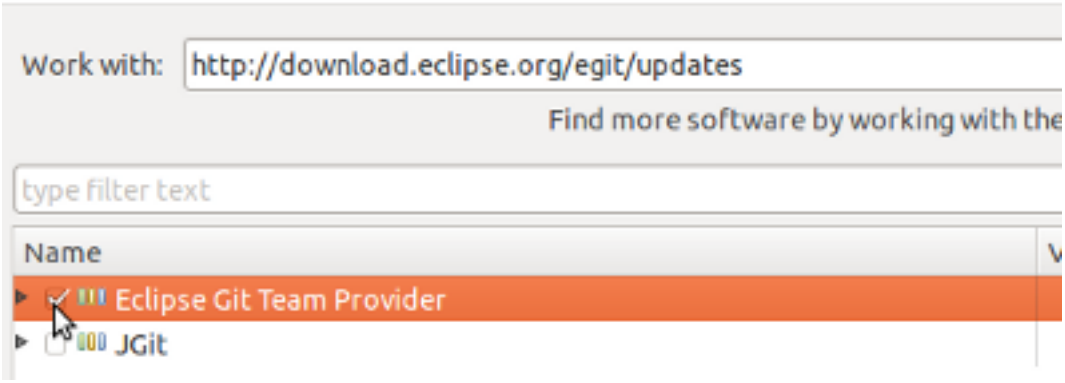

Abbildung 13: eGit in Eclipse installieren

4.1 Projekt unter lokale Versionsverwaltung stellen

Ein Projekt unter lokale Versionsverwaltung geht wie in folgendem Ablauf beschrieben:

- „Rechtsklick“ auf das „Projekt“ im Package Explorer
- Untermenü „Team“ wählen
- „Share Project“ auswählen
- Repositorytyp „Git“ wählen
- Mit Klick auf „Next >“ fortfahren
- Auf „Create“ klicken, um ein neues Repository anzulegen
- Pfad und Namen des Repositories eingeben und anschließend bestätigen
- Nun bestätigen wir erneut

Nach dem Ablauf ist die lokale Versionsverwaltung nutzbar.

4.2 Lokale Versionsverwaltung benutzen

Nachdem das Projekt unter Versionverwaltung steht können wir nun eGit benutzen. Die Funktionen von eGit sind im Teammenü aufgelistet. Um das Teammenü zu öffnen, klickt man auf eine Projektbestandteil mit der rechten Maustaste und geht anschließend über die Fläche Team.

Um eine Ressource zum überwachten Bereich hinzuzufügen, wählt man diese aus und klickt im Teammenü auf „Add to Index“. Die Ressource kann dann durch einen „Commit“ im Teammenü ins Repository gebracht werden. Branching und weitere Funktionen stehen ebenfalls im Teammenü zur Verfügung.

4.3 Arbeiten mit entfernten Repositories

Um mit einem entfernten Repository zu arbeiten, muss bereits ein entferntes Repository eingerichtet sein. Eine Anleitung zur Erstellung gibt es in der Sektion 2.2. In diesem Beispiel benutzen wir ein Repository mit dem SSH-Protokoll, welches im Gitorious (`http://scm.thm.de`) eingerichtet ist. Bevor wir das Repository lokal verfügbar machen können, müssen wir ein SSH-Schlüsselpaar erstellen und den öffentlichen Schlüssel im Gitorious eintragen. Wir können ein Schlüsselpaar wie in der Sektion 2.2 beschrieben anlegen oder auch direkt in Eclipse. Um ein Schlüsselpaar in Eclipse anzulegen, machen wir folgendes:

- Im Menü auf „Window“ klicken
- Dort wählen wir den Punkt „Preferences“
- Nun wählen wir im Seitenmenü „General->Network Connections->SSH2“
- Bereits bestehende Schlüssel lassen sich im Tab „General“ eintragen
- Im Tab „Key Management“ lassen sich neue Schlüssel anlegen
- Wahlweise RSA- oder DSA-Schlüssel generieren und den privaten Schlüssel mit „Save Private Key“ speichern
- Öffentlichen Schlüssel beim entfernten Repository eintragen

Wie der öffentliche Schlüssel in Gitorious eingetragen wird, kann in der Sektion 2.2 nachgelesen werden. Nun können wir mit der Synchronisation beginnen. Wenn wir ein lokales Projekt ins Repository pushen möchten arbeiten wir folgende Schritte ab:

- Rechtsklick auf das Projekt
- Dort wählen wir den Unterpunkt „Team“
- Nun gehen wir auf den Punkt „Commit“ und schieben unseren aktuellen Stand ins lokale Repository
- Anschließend gehen wir erneut ins „Team“-Menü und wählen das Untermenü „Remote“
- Hier wählen wir die Schaltfläche „Push“
- Hier tragen wir die Details des entfernten Repositories ein
  z.B: „gitorious@scm.thm.de:testprojekt/repository.git“
- Bei Protokoll wählen wir „SSH“. Als User tippen wir „gitorious“ und bestätigen die Eingabe.
- In der nächsten Ansicht klicken wir auf den Knopf „Add All Branches Spec“
- Mit einem Klick auf „Finish“ bestätigen wir die Eingabe und pushen das Projekt ins Repository

Wenn bereits ein Eclipseprojekt im Repository liegt arbeiten wir folgende Schritte ab:

- Auf den Menüpunkt „File“ gehen
- Dort wählen wir den Unterpunkt „Import“
- Hier wählen wir „Git -> Projects from Git“
- Hier tragen wir die Details des entfernten Repositories ein
  z.B: „gitorious@scm.thm.de:testprojekt/repository.git“
- Bei Protokoll wählen wir SSH und bestätigen die Eingabe
- Nun wählen wir die Branches aus, die wir synchronisieren wollen und bestätigen
- In der nächsten Ansicht tätigen wir die Einstellungen zum lokalen Zielort und bestätigen mit „Next>“.
- Wir wählen den Punkt „Import Existing Projects“ und klicken erneut auf„Next>“
- Hier wählen wir die Projekte aus, die wir einrichten wollen und bestätigen mit einem Klick auf „Finish“

Nun ist das Projekt eingerichtet und wir können die Repositories synchronisieren. Um den lokalen Stand ins entfernte Repository zu schieben, wählen wir im „Teammenü“ den Punkt „Push to Upstream“. Die Synchronisation vom entfernten Repository auf den lokalen Rechner lässt sich über den Punkt „Fetch from Upstream“ durchführen.

Literatur

[1] Scott Chacon. *Versionskontrolle mit Git*, 2012.

Autor: Rudolf Zimmermann, Artur Klos, Institut für SoftwareArchitektur.