BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Versionsverwaltung mit **git**

Christoph Knabe
FB VI
14.04.2017

Inhalt

- Probleme bei Software-Entwicklung
- Begriffe in git
- Geschichte von git
- Was ist verteilt an git?
- Installation
- Projekt auf Hoster einrichten
- Beispiel-Benutzung
- Mischen verteilter Änderungen und Konflikte
- Entwicklungszweige (Branching)
- Integration Manager Workflow, Einzelentwickler-Workflows
- Nur Quelltexte versionieren!
- Fazit

Probleme bei der Software-Wartung/Entwicklung

- Produkt(teile) entwickeln sich
  - beim Kunden ältere Versionen im Einsatz,
  - Reproduktion und Wartung dieser muss möglich sein ⇒
  - **Version**sverwaltung nötig! Machbar mit **CVS**, **svn**, **git**.

- Ähnliche, aber nicht gleiche Produkte
  - Bsp. **Varianten** für Windows, Unix, … ⇒
  - **Konfiguration**sverwaltung nötig!
  - Machbar mittels **Maven**-Profilen.

- Global verteilte Entwicklung,
  - Bsp.: Open Source (Linux)
  - Machbar mit **verteilten Versionierung**ssystemen (**git**).

Begriffe in **git**

- **Changeset**:
  - Menge an Änderungen von Projektdateien
  - durch ein `git commit` festgehalten
  - mit einem **Revision**sstempel (SHA1) identifiziert, Bsp. `98ca9`...
  - Bsp. https://www.assembla.com/code/lehrkraftnews/git/changesets
  - hat einen Eltern-Changeset.

- **Version**: Änderungszustand einer Gesamtauslieferung
  - politisch festgesetzt
  - Bsp. 2.0 in JPA 2.0

- **Repository**: speichert Historie
  - auch die Arbeitskopie ist ein Repository
  - Beliebige Repos können Ziel oder Quelle von `push`/`pull` sein!

Geschichte

- **SCCS**: Source Code Control System
  - Ca. 1980, mit Unix verbreitet, Vorwärts-Diffs
- **RCS**: Revision Control System
  - ca. 1985: Rückwärts-Diffs, Sperren-Modifizieren-Freigeben
- **CVS**: Concurrent Versions System
  - ca. 1990: Kopieren-Modifizieren-Konfliktlösung-Freigeben
  - Verbreitung mit Open-Source-Bewegung
- **SVN**: Subversion
  - ca. 2004: Umbenennen (Refactoring) unterstützt
  - moderne Zugänge (https)
- **git**: (verteiltes Versionierungssystem)
  - 2005: von Linus Torvalds
  - für Linux-Entwicklung
  - ohne zentrales Repository, ermöglicht vielfältige Workflows
  - Konkurrenz **mercurial**

Was ist verteilt an git?

- **Bis Subversion zentrales Repository**
  + platzsparend
  - geht nur online
  - Flaschenhals vorprogrammiert
  - Verlust der Historie bei Verlust des Providers
- **Verteilte Versionierung** :
  + viele Repositories können zusammenarbeiten
  + Workflow frei wählbar
  + Commit und Geschichtssuche gehen auch offline
  + skalierbar
- **Workflowmodelle**

  http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows
  * **Centralized**: Ein Repo, mehrere Entwickler pushen dahin
  * **Integration-Manager**: Er sammelt aus von Entwicklern veröffentlichten Entwickler: Server-Fork, clone, develop, push. I-Mgr.: clone, test, pull.
  * **Dictator-Lieutenants**: Diktator sammelt von Leutnants, diese von Entw.

Installation

- **Klientarten**
  - Kommandozeile: standardisiert, leicht dokumentierbar, automatisierbar
  - GUI-Applikation: Bequemer
  - IDE-Plugins: Am bequemsten

- **Einstieg in Installation:**
  https://help.github.com/articles/set-up-git
  Wichtig: Identität festlegen:

```
git config --global user.name "Your Name Here"
git config --global user.email "your_email@example.com"
```

- **Eigene Installationserfahrungen**:
  https://app.assembla.com/spaces/lehrkraftnews/wiki/Versionsverwaltung_mit_GIT

Projekt auf Hoster einrichten

- Auf Hoster (wie https://gitlab.beuth-hochschule.de/ mit HRZ-Daten) anmelden.
- Für einen Kommandoshell sorgen. Auf Unix-ähnlichen Systemen Default. Bei Git for Windows im Explorer mittels MausRechts > *Git Bash Here* auf einem Verzeichnis.
- Secure-Shell-Schlüsselpaar erzeugen:
  ssh-keygen
  Generating public/private rsa key pair.
  Enter file in which to save the key (~/.ssh/id_rsa):
  Akzeptieren.
- Beim Hoster unter Profile > SSH Keys (o.ä.) den Inhalt der Datei `~/.ssh/id_rsa.pub` eintragen und mit einem leicht merkbaren Namen versehen.
- Beim Hoster ein Projekt über die Web-Oberfläche leer anlegen.
- Dort die Repo-Clone-URL kopieren.
- Projekt wie auf Folgeseite klonen.

Beispiel-Benutzung

git clone repoURL
holt sich Repository vom Provider

echo "# Mein Projekt" >README.md
erzeugt Datei

git add README.md
markiert diese als in Commit aufzunehmen

git commit -m "Projekt-README angelegt."
Fixiert alle markierten Dateien in einem Changeset lokal mit Message.

git pull
Holt Änderungen vom Server und mischt diese ein. Jetzt testen!

git push
Schiebt die eigenen Commit-Changesets zum Server.

Mischen verteilter Änderungen

- **Push-Konflikt**:
  * ***`push`*** nur erlaubt, wenn die eigenen Commits auf dem letzten Changeset des Repositorys beruhen.
  * Ansonsten noch einmal ***`pull`*** nötig. Beinhaltet auch ***`merge`***.

- **Pull-Konflikt**:

* Änderungen durch Andere in anderen Dateien oder Zeilenbereichen werden stillschweigend eingemischt.
* Änderungen in derselben Datei im selben Zeilenbereich werden als Konfliktmarkierung geholt:

```
<<<<<<< HEAD:Gruss.java
System.out.println("Guten Abend");     }klientseitig
=======
System.out.println("Guten Morgen");    }serverseitig, zuvor durch anderen
>>>>>>> origin/master:Gruss.java       }Klienten commitet.
```

* Konflikt muss manuell korrigiert werden. Wie z.B.?

Branching

- **Ein “Branch” ist ein Entwicklungszweig.**
  * Standardmäßig im Branch **`master`**
  * Erzeugen eines getrennten Zweiges sehr billig (speichert 41 Bytes)
  * Kommando-Bsp.: `git branch ticket47`
  * `git checkout ticket47` macht `ticket47` zum aktuellen Branch.
  * Alle Kommandos wirken auf aktuellen Branch.
  * `git checkout master` schaltet zum Hauptzweig zurück.

- **Zusammenführen von Zweigen**:
  * `git checkout ticket47` #Auf Branch umschalten
  * Entwickeln, `mvn test` #Ticket lösen und testen
  * `git merge master` #Einmischen von Änderungen des Hauptzweiges
  * `mvn test` #Erneut testen
  * `git commit -m "Ticket 47 gelöst"` #lokal freigeben
  * `git checkout master; git pull` #Aktuellsten Stand einmischen
  * `git merge ticket47; mvn test` #Lösung einmischen und testen
  * `git push` #Zum origin-Repository schieben

http://www.git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging

Integration Manager Workflow

- **Integration Manager**
  * Pflegt eigenes öffentliches und privates Repo.
- **Entwickler**:
  * Forkt das öffentliche Integrations-Repo auf Serverseite.
  * Erhält dadurch eigenes öffentliches Repo mit gleichen Inhalten.
  * `git clone` *Adresse* #Klont es zu privater Arbeitskopie
  * Entwickelt, testet.
  * `git push` #Zum eigenen öffentlichen Repo schieben
  * Pull-request an Integrationsmanager senden..
- **Integration Manager**
  * Klont oder pullt vom öffentlichen Repo des Entwicklers.
  * Testet.
  * Mergt in eigenes privates Repo. testet.
  * Pusht in eigenes öffentliches repo.
- **Entwickler**:
  * Pullt vom öffentlichen Repo des Integrations-Managers.

Einzelentwickler-Workflows

- **Für Datensicherung**
  * regelmäßig **git commit**; **git push**:
    sichert ins entfernte **origin**-Repository, selbst wenn noch fehlerhaft.
- **Mehrrechner-Koordination**:
  * Vor Arbeitsbeginn auf jedem Rechner: **git pull**
  * Vor Arbeitsende auf jedem Rechner: **git commit**; **git push**
  * Konflikte möglich, wenn diese Reihenfolge nicht eingehalten wird und in derselben Datei im selben Zeilenbereich editiert wird.
- **Meilenstein-Branches**
  * Abzuliefernde, vorführbare Version als Branch festhalten
  * Im Master für nächsten Meilenstein weiterentwickeln
  * Zur Vorführung umschalten: **git checkout** *meilenstein*
  * Danach wieder zum Master-Zweig: **git checkout master**
  * Wenn es Änderungen gab, integrieren: **git merge *meilenstein***

Nur Quellen versionieren

- **Echte Quelltexte**:
  * git versioniert standardmäßig alle Dateien
  * passend für Orginale (`.scala`, `.java`, `.odt`, `.docx`, ...)

- **Erzeugbare Dateien nicht versionieren**
  * Datei `.gitignore` kann Namensmuster für unversionierte Dateien enthalten. Bsp. für Maven:

```
/target/
*.pdf
*.class
```

Fazit

git ist ein leistungsfähiges Werkzeug zur verteilten Entwicklung mit vielen Teilnehmern.

- **Verteilte Repos:**
  * Ermöglichen komplexe Workflows.
  * Ermöglichen Nutzung eines eigenen Repos als Datensicherung.
- **Leichtes Branching:**
  * Entwicklung nicht linear wie bei Subversion,
  * sondern leichtes Verzweigen und Zusammenführen.
- **Groß im Kommen**
  + Linux-Entwicklung läuft damit seit 2005.
  + Adaptiert z.B. von Spring, Lift, jQuery, Ruby on Rails, SWE-Labor

Vielen Dank

Quellen:
Eigene Erfahrungen, Google, angegebene Links