Wissenschaftliches Schreiben mit git und LaTeX

siehe auch: Versionsverwaltung mit git

Diese Anleitung ist noch unvollständig und wird weiter ausgebaut.

LaTeX ist als Textsatzprogramm zum Erstellen wissenschaftlicher Arbeiten, Artikel und Bücher im akademischen Umfeld weit verbreitet. Vorteile sind einfacher Mathematiksatz, Erweiterbarkeit und Anpassbarkeit durch Pakete und eigene Makros, sowie die unübertroffene Satzqualität. Außerdem erlaubt das Textformat Automatisierung mit einfachen Skripten.

In diesem Tutorial soll es um einen anderen Vorteil gehen, den LaTeX für das akademische Schreiben hat: Das Textformat erlaubt die Verwendung eines Versionskontrollsystems. Die Zielgruppe dieser Anleitung sind alle, die LaTeX schon verwenden (oder gerade erlernen) und keine Angst vor der Shell haben (und am besten ein Unix-System verwenden).

Eine Kurze Einführung in Verteilte Versions- & Variantenverwaltung und git

Die Idee von Versionskontrollsystem ist, dass man zu einem Dokument (oder einem Verzeichnisbaum) auch auf alle vorigen Versionen zugriff hat und Werkzeuge hat um Änderungen zu verwalten (z. B. abgleichen mit den Änderungen die jemand anderes an den Dateien vorgenommen hat, paralleles Bearbeiten verschiedener Versionen). Außerdem machen Versionskontrollsysteme es leicht die gesamten Daten auf mehrere Computer zu verteilen und synchron zu halten. Ein Verzeichnisbaum zusammen mit seiner Bearbeitungsgeschichte nennt man *Repository* (kurz Repo).

Die Variantenverwaltung ergänzt dieses Konzept der linear aufeinander folgenden Versionen durch Branches (auf deutsch etwa: Äste oder Zweige). Dies wird aber erst später im Abschnitt über Branches behandelt.

git gehört zu der (modernen) Klasse der Verteilten Versionskontrollsysteme, das heißt, dass jede Kopie der Daten technisch gleichberechtigt ist (es ist lediglich eine Frage der Konvention, welche Version die “offizielle” ist). Es können also mehrere Nutzer *gleichzeitig* an den Dokumenten arbeiten, und wenn sie das abgeschlossen haben, können sie git verwenden um die Versionen zu einer zusammenzuführen (*mergen*) die die Änderungen beider enthält. Falls beide die gleiche Stelle bearbeitet haben sollten, dann gibt es einen *Konflikt*, der manuell aufgelöst werden muss.

Im nächsten Abschnitt wird allerdings zunächst gezeigt, warum git auch für einen einzeln Arbeitenden von Vorteil ist.

Weiterführende Links:

- Themenabend Git Basics beim c3d2

Beispiel: git für Einen

Um dem Tutorial im weiteren zu folgen müssen die Programme git und (optional aber empfohlen) gitk installiert werden.

Szenario

Bob schreibt eine Hausarbeit. Da er in der Vergangenheit schon oft durch Missgeschicke Änderungen verloren hat oder nach fehlgeschlagenen Verbesserungen an einem Absatz diese nicht wieder loswerden konnte ohne die Verbesserungen an einem anderen Absatz mühsam manuell in seine Sicherheitskopie zu übertragen, möchte er nun git für die Versionsverwaltung einsetzen.

Konfiguration

Erstmal Konfiguriert Bob seine git Installation. Dies erfolgt am besten mit Kommando `git config --global`. Durch das `--global` werden die Konfigurationsvariablen in der Datei `~/.gitconfig` als Bobs Voreinstellungen ablegt. Er setzt seinen Namen und seine E-Mail-Adresse, die beide in seinen Commits im Repository verzeichnet werden:

```
$ git config --global user.name "Bob Penguin"
$ git config --global user.email "bob@penguin.example"
```

Erste Schritte

Zunächst legt Bob einen Ordner an und intialisiert diesen als git Repository:

```
$ mkdir Hausarbeit
$ cd Hausarbeit
$ git init
```

Als nächstes legt er eine Datei `.gitingore` mit folgendem Inhalt an:

```
# ignore emacs backup files
*~
# ignore TeX auxiliary files
*.aux
*.toc
# ignore compiled pdf files
*.pdf
```

(TODO: Ist es sinnvoll, das hier jetzt schon zu sagen, ohne vorher `git status` erwähnt zu haben? Macht das die Sache nicht unnötig kompliziert?)

Damit wird festgelegt dass Dateien die den aufgeführten globs entsprechen von bestimmten git

Kommandos ignoriert werden. Außerdem legt er die Datei `arbeit.tex` an und fügt die Präambel und eine leere `document`-Umgebung ein. Dann tippt er:

```
$ git add .gitingore arbeit.tex
```

Damit fügt er die Änderungen an den beiden Dateien zu dem *Index* hinzu.

```
$ git commit -m 'erster Commit'
```

Damit ist der jetzige Zustand des Index im git Repository verzeichnet und kann jederzeit wieder hergestellt werden.

Anlegen einer Sicherheitskopie

Da Bob auch eine Sicherheitskopie seines Arbeitsfortschritts haben will, steckt er seinen USB-Stick an und wechselt in ein Verzeichnis auf dem USB-Stick, dort tippt er:

```
$ git clone /pfad/zu/Hausarbeit
```

Damit ist eine vollständige Kopie des aktuellen Standes auf dem Stick (er *hat das Repository geklont*). Wenn er später wieder den Stand aktualisieren will, wechselt er in das Verzeichnis `Hausarbeit` auf dem USB-Stick und tippt:

```
$ git pull
```

TODO: Vielleicht doch lieber ein bare-Repo klonen und dahin pushen, sind Auto-Mount-Points stabil (dann ist das kein Problem)? Auch noch andere Varianten erläutern (ein git server, per ssh auf eine andere Maschine).

Bearbeiten der Dateien und Übernehmen der Änderungen

Bob schreibt nun eine Einleitung für seine Arbeit. Nachdem dies geschehen ist, tippt er:

```
$ git add arbeit.tex
$ git commit -m "Einleitung geschrieben"
```

Alternativ könnte er auch nur `git commit` (ohne die `-m` -Option) eingeben und seine *Commit Message*, also eine kurz Beschreibung seiner Änderungen, in einem Texteditor eingeben, der dann automatisch geöffnet würde (in der Regel vi, das ist in der Umgebungsvariable `$EDITOR` festgelegt). Dort tippt Bob nun eine *Commit Message*, die eine Zusammenfassung seiner Änderungen enthält:

```
Einleitung geschrieben
```

Sobald er diese speichert und den Editor verlässt (im vi: Esc-Taste und dann `:wq` eintippen) ist die aktuelle Version der Arbeit gesichtert.

Um von den Änderungen auch eine Sicherheitskopie zu haben muss er nur in das Verzeichnis auf dem USB-Stick wechseln und

```
$ git pull
```

ausführen. Um eine Übersicht über die Änderungen und die Dateien im Verzeichnis zu erhalten kann Bob jederzeit:

```
$ git log
```

und

```
$ git status
```

verwenden. (TODO: Erklären was git einem damit sagt).

Branches, Merges und ''git revert''

Zunächst wird kurz erklärt was ein git Repsository eigentlich ist.

Die Geschichte in einem git Repository ist ein azyklischer, gerichteter Graph von Revisionen (Zustände der Ordnerstruktur samt Inhalt). Jeder Knoten des Graphen stellt einen “Snapshot” des Zustandes dar, der mit Metadaten dekoriert ist (Autor, Commit-Nachricht, ...). Wir repräsentieren die Graphen wie folgt, hier ein Repo mit drei Commits (A, B, C):

```
A    B    C
o<---o<---o
```

Aus dem Graphen und dem Inhalt des Commits wird die Commit-ID per kryptographischer Checksumme abgeleitet und identifiziert den Commit eindeutig (mit astronomisch kleiner Chance einer Kollision).

Zusätzlich dazu enthält das Repository den `working tree` also die Ordner- und Dateistruktur mit den Dateiinhalten an denen gearbeitet wird und den Index (staging area), den man als vorläufigen Commit verstehen kann.

Die Graphenstruktur erlaubt mehrere konkurrente Änderungen über einer Basisversion.

```
       D    E
       o<---o
A     /
o<---o<---o
     B    C
```

Branches funktionieren wie “Lesezeichen” im Commit-Graphen.

```
       D    E <- bugfix
       o<---o
A     /
o<---o<---o
```

```
      B    C <- master <- HEAD
```

Der Commit der aktuell der Bezugspunkt für die git-Operationen ist wird mit HEAD bezeichnet.

Um einen Branch zu wechseln wird das Kommando `git checkout` verwendet:

```
$ git checkout bugfix
```

Danach sieht der Graph wie folgt aus:

```
       D    E <- bugfix <- HEAD
       o<---o
A     /
o<---o<---o
      B    C <- master
```

Das Arbeitsverzeichnis und der Index werden auf den Zustand der im Commit verzeichnet ist gesetzt.

Um am aktuellen Commit einen neuen Branch zu erzeugen verwendet man:

```
$ git branch neuer-branch
```

Danach kann man ihn mit `git checkout` auschecken. Kurz geht das mit einer Operation:

```
$ git checkout -b neuer-branch
```

Nach dieser Operation sieht der Graph wie folgt aus:

```
               <- neuer-branch <- HEAD
       D    E <- bugfix
       o<---o
A     /
o<---o<---o
      B    C <- master
```

Nun können wir unsere Dateien bearbeiten und neue Commits erzeugen. Diese werden dann im aktuellen Branch verzeichnet:

```
              <- bugfix    <- neuer-branch <- HEAD
       D    E      F     G
       o<---o<-----o<----o
A     /
o<---o<---o
      B    C <- master
```

Nun haben wir mehrere Branches, nun ist das Ziel diese wieder zusammenzuführen, dies funktioniert mit `git merge`:

```
$ git checkout bugfix
$ git merge neuer-branch
```

Danach sieht unser Graph so aus:

```
                          <- bugfix  <- HEAD
                          <- neuer-branch
       D    E      F     G
       o<---o<-----o<----o
  A    /
  o<---o<---o
       B    C <- master
```

Diese Form des merges nennt man fast-forward (Vorspulen). Darüber hinaus gibt es “echte Merges”, wenn wir nun `bugfix` in `master` mergen:

```
$ git checkout master
$ git merge bugfix
```

Dann sieht der Graph wie folgt aus:

```
                          <- bugfix
                          <- neuer-branch
       D    E      F     G
       o<---o<-----o<----o<-
  A    /                   \
  o<---o<---o<--------------o
       B    C                H <- master <- HEAD
```

Dabei wurde der merge-Commit H angelegt, darin sind die Änderungen in `master` und `bugfix` Zusammengeführt. Dabei kann es zu Konflikten kommen, die manuell Aufgelöst werden müssen.

Kollaboration mit git

git kann seine Vorteile noch besser ausspielen, wenn man mit anderen kollaboriert: Jeder kann Unabhängig an seiner Version arbeiten und man kann die Änderungen dann automatisiert zusammenführen.

Damit das geht braucht man ein git Repository. Da öffentlich github Repositories für wissenschaftliche Projekte nicht zu empfehlen sind, sollte man seinen eigenen Server mit gitolite3 betreiben, oder von der Uni bereitgestellte Infrastruktur nutzen (an der TU Dresden kann man z. B. Fusionforge verwenden).

Man sollte einen SSH schlüssel erzeugen (… TODO …) und den öffentlichen Schlüssel beim Server eintragen.

Wenn man dort einen Account hat und ein Repo und die entsprechenden Berechtigungen eingerichtet hat, kann man es mit

```
$ git clone user@server/repo
```

klonen, dabei läuft die Authentifizierung über das SSH-public-key-Verfahren.

Danach kann man mit

```
$ git fetch
```

oder

```
$ git pull
```

Änderungen vom repository holen. Mit

```
$ git push
```

kann man seine eigenen Änderungen verbreiten.

Das erlaubt es gemeinsam mit anderen am Repository zu arbeiten und Änderungen auszutauschen.

Automatisch erstellter Inhalt und Makefiles

TODO:

- Mit Skripten automatisch Daten auswerten und daraus TeX-Source erzeugen
- Makefiles verwenden um den build-Prozess zu automatisieren
- bibtex/biber, makeglossaries

Siehe auch

- https://wiki.zum.de/wiki/Verfassen_von_wissenschaftlichen_Arbeiten_mit_LibreOffice

Wissenschaftliches Schreiben, git, LaTeX, Anleitung