

Änderungen verwalten mit `git`

PeP et al. Toolbox Workshop



PeP et al. e.V.

Physikstudierende und
ehemalige Physikstudierende
der TU Dortmund

2018

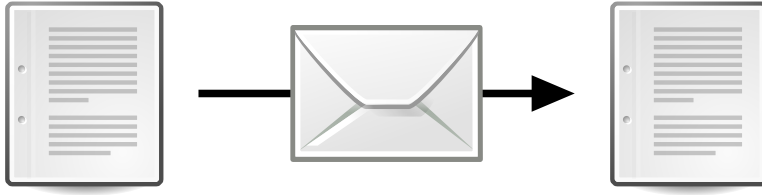
Versionskontrolle

- Verwaltung von Versionen
- Speicherung der „Geschichte“ eines Projekts
- Es ist jederzeit möglich auf eine ältere Version zurückzukehren
- Es ist möglich, sich die Unterschiede zwischen Versionen anzeigen zu lassen
- Backup

**Wichtige Voraussetzungen für korrektes wissenschaftliches Arbeiten,
auch wenn man alleine arbeitet**

Wie arbeitet man am besten an einem Protokoll
zusammen?

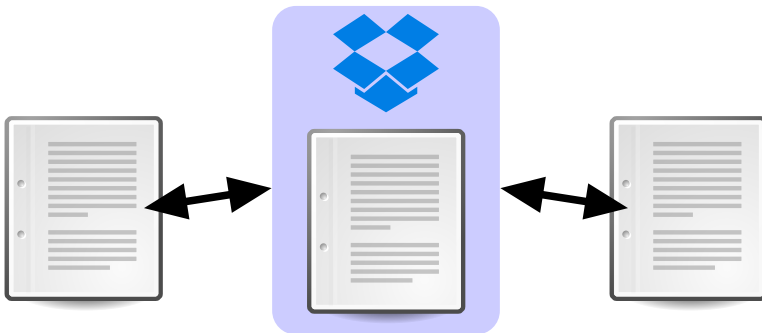
Idee: Austausch über Mails



- Risiko, dass Änderungen vergessen werden, ist groß
- Bei jedem Abgleich muss jemand anders aktiv werden
 - Stört
 - Es kommt zu Verzögerungen

Fazit: Eine sehr unbequeme / riskante Lösung

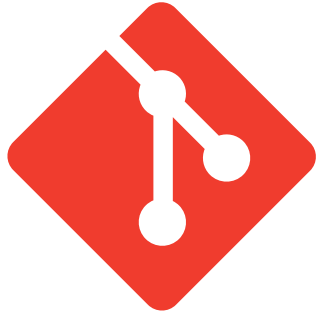
Idee: Austausch über Dropbox



- Man merkt nichts von Änderungen der Anderen
- Gleichzeitige Änderungen führen zu „In Konflikt stehende Kopie“-Dateien
- Änderungen werden nicht zusammengeführt
- Keine echte Historie des Projekts

Fazit: Besser, aber hat deutliche Probleme

Lösung: Änderungen verwalten mit `git`



git

- Ein Versionskontrollsystem
- Ursprünglich entwickelt, um den Programmcode des Linux-Kernels zu verwalten (Linus Torvalds)
- Hat sich gegenüber ähnlichen Programmen (SVN, mercurial) durchgesetzt
- Wird in der Regel über die Kommandozeile benutzt

Was bringt git für Vorteile?

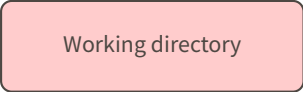
- Arbeit wird für andere sichtbar protokolliert
- Erlaubt Zurückspringen an einen früheren Zeitpunkt
- Kann die meisten Änderungen automatisch zusammenfügen
- Wirkt nebenbei auch als Backup

Einzigste Herausforderung: Man muss lernen, damit umzugehen

Das Repository

Zentrales Konzept: Das Repository

- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository



Working directory

Zentrales Konzept: Das Repository

- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository

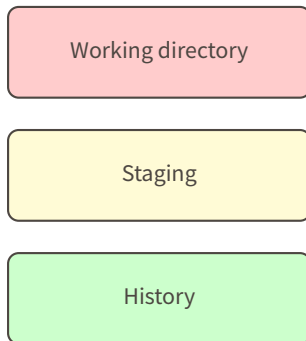
The diagram consists of two vertically stacked rounded rectangular boxes. The top box is light red and contains the text 'Working directory'. The bottom box is light yellow and contains the text 'Staging'. There are no arrows or other graphical elements connecting the two boxes.

Working directory

Staging

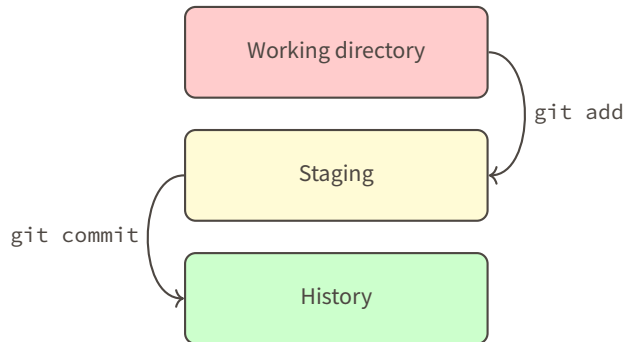
Zentrales Konzept: Das Repository

- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository



Zentrales Konzept: Das Repository

- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository



Working directory

Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

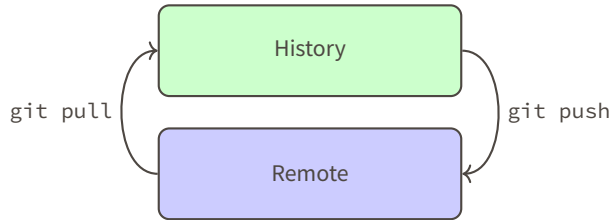
Staging

Änderungen, die für den nächsten commit vorgemerkt sind.

History

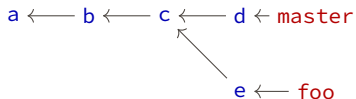
Gespeicherte Historie des Projekts. Alle jemals gemachten Änderungen. Ein Baum von Commits.

Remotes sind zentrale Stellen, z. B. Server auf denen die History gespeichert wird.

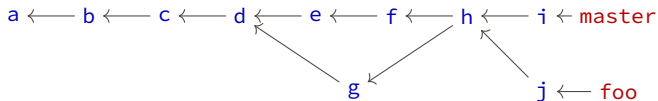


a ← b ← c ← d ← master

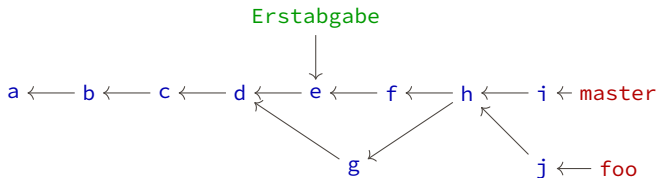
- **Commit**: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
 - Enthält Commit-Message (Beschreibung der Änderungen)
 - Wird über einen Hash-Code identifiziert
 - Zeigt immer auf seine(n) Vorgänger



- **Commit**: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
 - Enthält Commit-Message (Beschreibung der Änderungen)
 - Wird über einen Hash-Code identifiziert
 - Zeigt immer auf seine(n) Vorgänger
- **Branch**: benannter Zeiger auf einen Commit
 - Entwicklungszweig
 - Im Praktikum reicht bereits die Standard-Branch: master
 - Wandert weiter



- **Commit**: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
 - Enthält Commit-Message (Beschreibung der Änderungen)
 - Wird über einen Hash-Code identifiziert
 - Zeigt immer auf seine(n) Vorgänger
- **Branch**: benannter Zeiger auf einen Commit
 - Entwicklungszweig
 - Im Praktikum reicht bereits die Standard-Branch: master
 - Wandert weiter



- **Commit**: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
 - Enthält Commit-Message (Beschreibung der Änderungen)
 - Wird über einen Hash-Code identifiziert
 - Zeigt immer auf seine(n) Vorgänger
- **Branch**: benannter Zeiger auf einen Commit
 - Entwicklungszweig
 - Im Praktikum reicht bereits die Standard-Branch: master
 - Wandert weiter
- **Tag**: unveränderbarer Zeiger auf einen Commit
 - Wichtiges Ereignis, z.B. veröffentlichte Version

- 1.** Neues Repo? Repository erzeugen oder klonen:
Repo schon da? Änderungen herunterladen:

```
git init, git clone  
git pull
```
- 2.** Arbeiten
 - 2.1 Dateien bearbeiten und testen
 - 2.2 Änderungen vorbereiten:

```
git add
```
 - 2.3 Änderungen als *commit* speichern:

```
git commit
```
- 3.** Commits anderer herunterladen und integrieren:

```
git pull
```
- 4.** Eigene Commits hochladen:

```
git push
```

<code>git init</code>	initialisiert ein git-Repo im jetzigen Verzeichnis
<code>git clone url</code>	klont das Repo aus <i>url</i>
<code>rm -rf .git</code>	löscht alle Spuren von git aus dem Repo

`git status` zeigt Status des Repos (welche Dateien sind neu, gelöscht, verschoben, bearbeitet)

`git log` listet Commits in aktuellem Branch

git add,git mv,git rm,git reset

- `git add file ...` fügt Dateien/Verzeichnisse zum Staging-Bereich hinzu
- `git add -p ...` fügt Teile einer Datei zum Staging-Bereich hinzu
- `git mv` wie mv (automatisch in Staging)
- `git rm` wie rm (automatisch in Staging)
- `git reset file` entfernt Dateien/Verzeichnisse aus Staging

<code>git diff</code>	zeigt Unterschiede zwischen Staging und Arbeitsverzeichnis
<code>git diff --staged</code>	zeigt Unterschiede zwischen letzten Commit und Staging
<code>git diff <i>commit1</i> <i>commit2</i></code>	zeigt Unterschiede zwischen zwei Commits

<code>git commit</code>	erzeugt Commit aus jetzigem Staging-Bereich, öffnet Editor für Commit-Message
<code>git commit -m "message"</code>	Commit mit <i>message</i> als Message
<code>git commit --amend</code>	letzten Commit ändern (fügt aktuellen Staging hinzu, Message bearbeitbar)

Niemals commits ändern, die schon gepusht sind!

- Wichtig: Sinnvolle Commit-Messages
 - Erster Satz ist Zusammenfassung (ideal < 50 Zeichen)
 - Danach eine leere Zeile lassen
 - Dann längere Erläuterung des commits
- Logische Commits erstellen, für jede logische Einheit ein Commit
 - `git add -p` ist hier nützlich
- Hochgeladene Commits sollte man nicht mehr ändern

`git pull` Commits herunterladen

`git push` Commits hochladen

→ Aus der Installationsanleitung:

```
git config --global pull.rebase true
```

Don't Panic

Entstehen, wenn `git` nicht automatisch mergen kann (selbe Zeile geändert, etc.)

1. Die betroffenen Dateien öffnen
2. Markierungen finden und die Stelle selbst mergen (meist wenige Zeilen)

```
<<<<<< HEAD
foo
||||||| merged common ancestors
bar
=====
baz
>>>>>> Commit-Message
```

3. Merge abschließen:

3.1 `git add ...`

3.2 `git rebase --continue`

Nützlich: `git config --global merge.conflictstyle diff3`

`git checkout commit` Commit ins Arbeitsverzeichnis laden

`git checkout file` Änderungen an Dateien verwerfen (zum letzten Commit zurückkehren)

`git stash` Änderungen kurz zur Seite schieben
`git stash pop` Änderungen zurückholen aus Stash

- Man möchte nicht alle Dateien von git beobachten lassen
- z.B. build-Ordner

Lösung: .gitignore-Datei

- einfache Textdatei
- enthält Regeln für Dateien, die nicht beobachtet werden sollen

Beispiel:

```
build/  
*.pdf  
__pycache__/
```

GitHub

- größter Hoster
- viele open-source Projekte
- Unbegrenzt private Repositories für Studenten und Forscher:
education.github.com



- kostenlose private Repos mit höchstens fünf Leuten
- keine Speicherbegrenzungen
- Hängt was Oberfläche und Funktionen angeht, den beiden anderen weit hinterher



- open-source
- keine Begrenzungen an privaten Repos
- kann man selbst auf einem eigenen Server betreiben

GitHub

- größter Hoster
- viele open-source Projekte
- Unbegrenzt private Repositories für Studenten und Forscher:
education.github.com



- kostenlose private Repos mit höchstens fünf Leuten
- keine Speicherbegrenzungen
- Hängt was Oberfläche und Funktionen angeht, den beiden anderen weit hinterher



- open-source
- keine Begrenzungen an privaten Repos
- kann man selbst auf einem eigenen Server betreiben

„Now, everybody sort of gets born with a GitHub account“ – Guido van Rossum

Git kann auf mehrere Arten mit einem Server kommunizieren:

- HTTPS: funktioniert immer, keine Einstellungen erforderlich, Passwort muss für jede Kommunikation eingegeben werden
- SSH: Keys müssen erzeugt und eingestellt werden, Passwort für den Key muss nur einmal pro Session eingegeben werden.

SSH-Keys:

- 1.** `ssh-keygen -t rsa -b 4096 -o -a 100`
- 2.** Passwort wählen
- 3.** `cat ~/.ssh/id_rsa.pub`
- 4.** Ausgabe ist Public-Key, beim Server eintragen (im Browser)