

# Änderungen verwalten mit `git`

PeP et al. Toolbox Workshop



**PeP et al. e.V.**

Physikstudierende und  
ehemalige Physikstudierende  
der TU Dortmund

2020

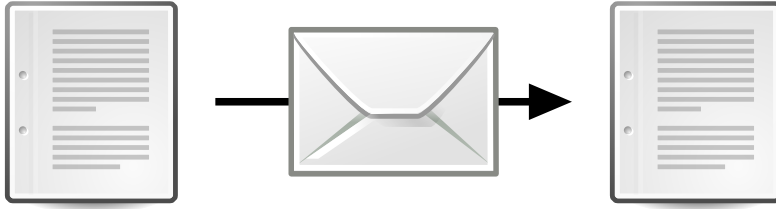
# Versionskontrolle

- 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?

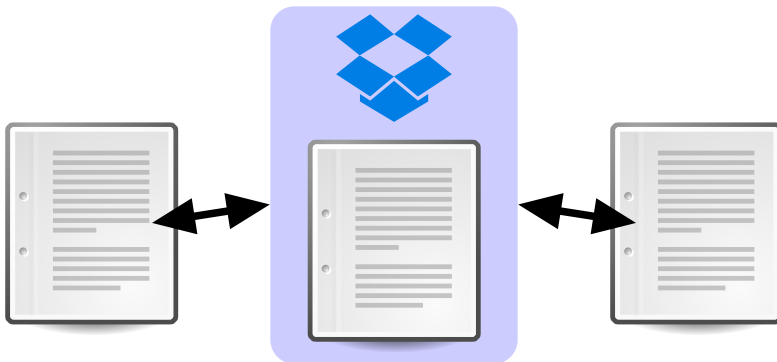
Idee: Austausch über Mails



- 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?

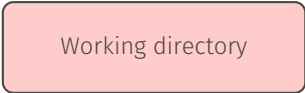
- Arbeit wird für andere sichtbar protokolliert
- Erlaubt Zurückspringen an einen früheren Zeitpunkt
- Kann die meisten Änderungen automatisch zusammenfügen
- Wirkt nebenbei auch als Backup

Einzigste Herausforderung: Man muss lernen, damit umzugehen

# Das Repository

# Zentrales Konzept: Das Repository

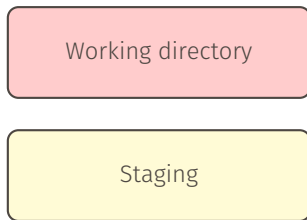
- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository



Working directory

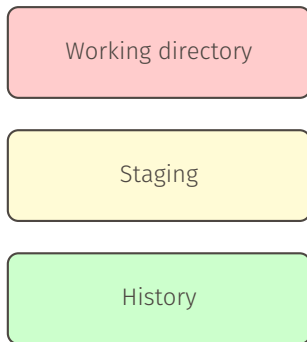
# Zentrales Konzept: Das Repository

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



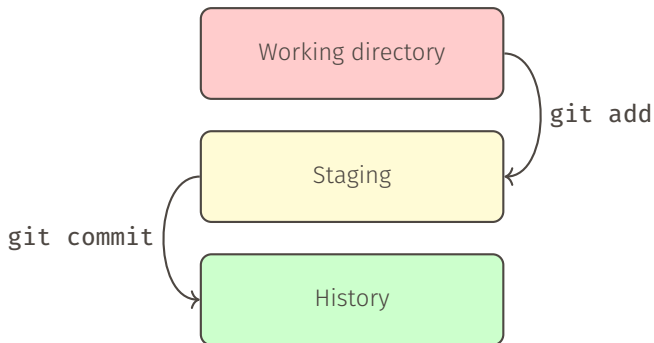
# Zentrales Konzept: Das Repository

- 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





# Zentrales Konzept: Das Repository

Working directory

Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

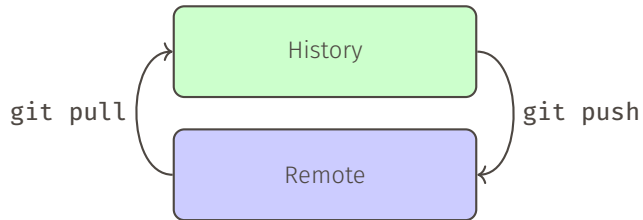
Staging

Änderungen, die für den nächsten commit vorgemerkt sind.

History

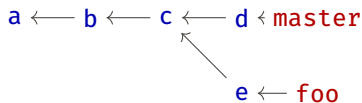
Gespeicherte Historie des Projekts. Alle jemals gemachten Änderungen. Ein Baum von Commits.

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

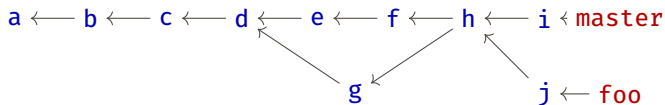


a ← b ← c ← d ← master

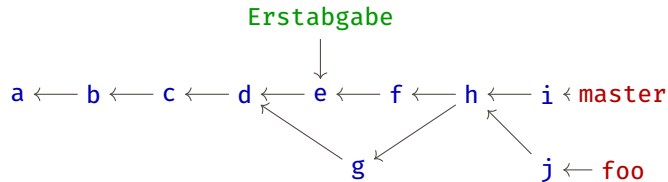
- **Commit**: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
  - Enthält Commit-Message (Beschreibung der Änderungen)
  - Wird über einen Hash-Code identifiziert
  - Zeigt immer auf seine(n) Vorgänger



- **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** (Auf github ab Okt. 2020: **main**)
  - 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** (Auf github ab Okt. 2020: **main**)
  - 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** (Auf github ab Okt. 2020: **main**)
  - 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:
- 2.3 Änderungen als commit speichern:

```
git add  
git commit
```

3. Commits anderer herunterladen und integrieren:

```
git pull
```

4. Eigene Commits hochladen:

```
git push
```

# git init, git clone

<code>git init</code>	initialisiert ein <b>git</b> -Repo im jetzigen Verzeichnis
<code>git clone url</code>	klont das Repo aus <b>url</b>
<code>rm -rf .git</code>	löscht alle Spuren von <b>git</b> 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 commit1 commit2</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 <b>message</b> 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, git push

`git pull` Commits herunterladen

`git push` Commits hochladen

## 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 ...** (Files mit behobenen Konflikten)
  - 3.2 **git commit** → Editor wird geöffnet
  - 3.3 Vorgeschlagene Nachricht kann angenommen werden (In vim "**:wq**" eintippen)

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



# .gitignore

- 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__/
```

## --rebase (optional)

Vielfaches Merging und Merge Konflikte erzeugen eine etwas nichtlineare Projekt-Historie, denn:  
`git pull` entspricht `git fetch origin; git merge ...` (→ gemergter Branch bleibt erhalten)

Alternativ kann man `git pull --rebase` ausführen, welches (in etwa) äquivalent ist zu  
`git fetch origin; git rebase ...` (→ lokale Commits werden auf neue Commits angewendet).

Achtung: Um einen Merge Konflikt bei `git pull --rebase` abzuschließen, muss  
`git rebase --continue` anstelle von `git commit -m " ... "` ausgeführt werden! Also einfach genau  
lesen was Git empfiehlt ;)

Dies hat Vorteile:

- Die Projekt-Historie ist linearer
- Es gibt weniger merge-commits

aber auch (kleinere) Nachteile:

- Es ist hinterher nicht mehr sichtbar, wer einen Merge Konflikt wie behoben hat
- Die Abfolge der Commits entspricht nicht mehr der wahren Entwicklungshistorie

Entscheidet man sich für pulls mit Rebase als Standard, muss Git anders konfiguriert werden:

`git config --global pull.rebase true`, dann wird bei allen folgenden `git pull` Befehlen ein  
Rebase gemacht

## GitHub

- größter Hoster
- viele open-source Projekte
- Unbegrenzt private Repositories für Studenten und Forscher:  
[education.github.com](https://education.github.com)

## Bitbucket

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

## GitLab

- 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](https://education.github.com)

## Bitbucket

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

## GitLab

- 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)