# Änderungen verwalten mit `git`

PeP et al. Toolbox Workshop

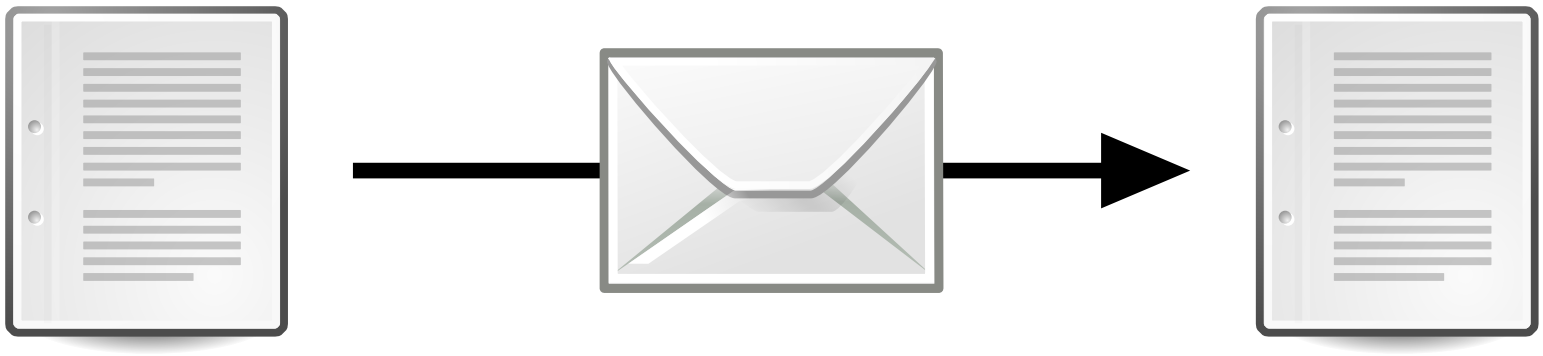
PEP ET AL. E.V.
PHYSIKSTUDIERENDE UND
EHEMALIGE PHYSIKSTUDIERENDE
DER TU DORTMUND

2015

# Wie arbeitet man am besten an einem Protokoll zusammen?

# Idee: Austausch über Mails

# Mails: Probleme



- → 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

# Dropbox: Probleme

- → Man merkt nichts von Änderungen der Anderen
- → Gleichzeitige Änderungen führen zu „In Konflikt stehende Kopie“-Dateien.
- → Änderungen werden nicht zusammengeführt.

**Fazit: Besser, aber hat deutliche Probleme**

# Lösung: Änderungen verwalten mit `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

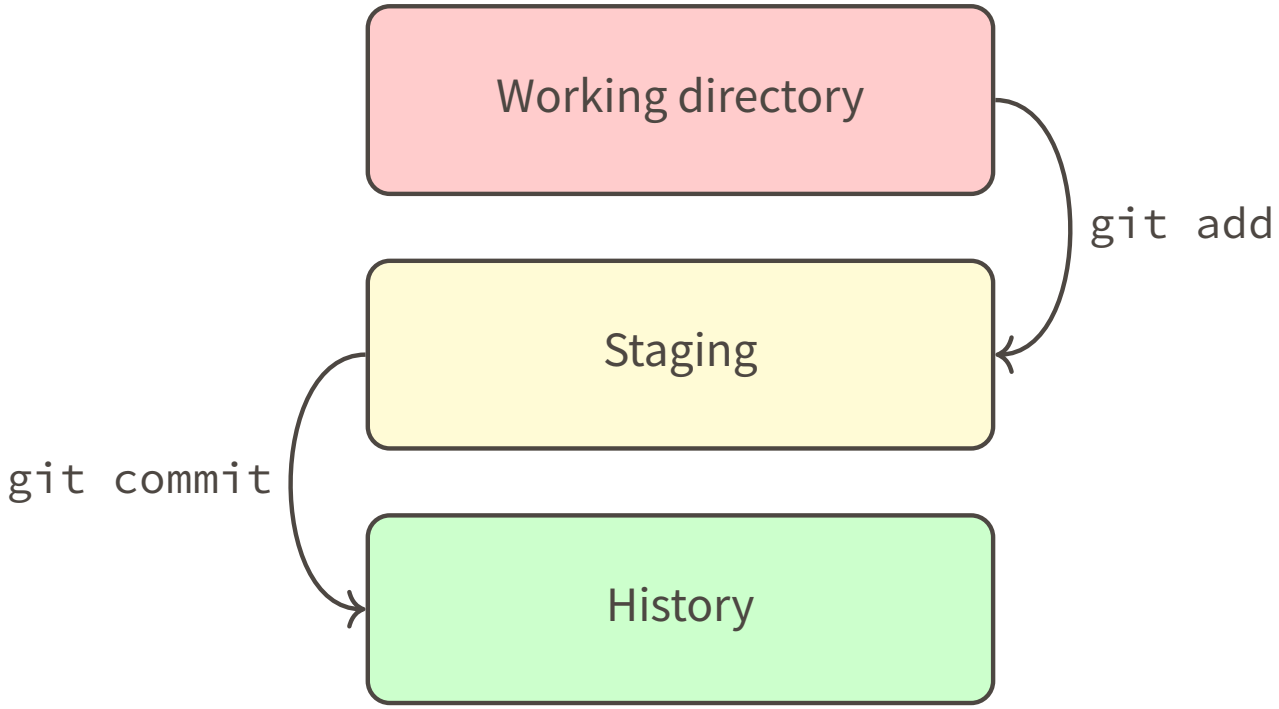
# Was bringt `git` für Vorteile?

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

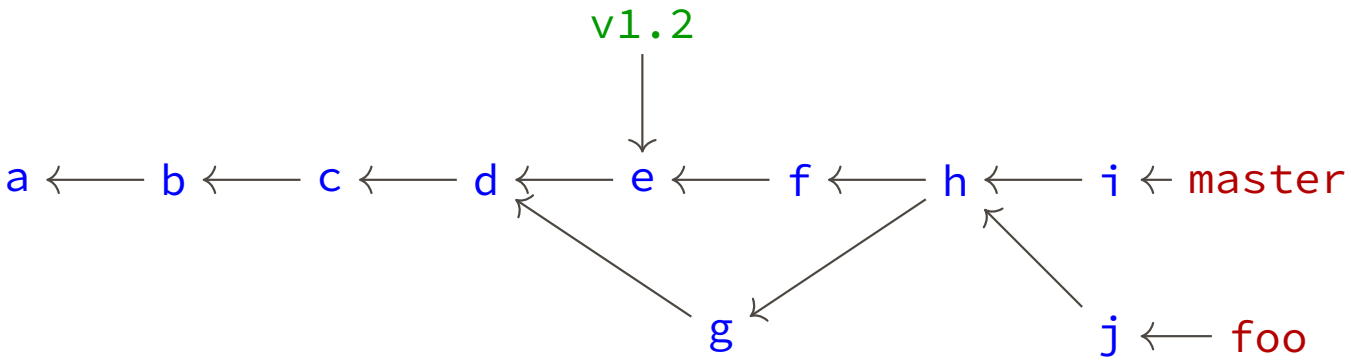
Einzige Herausforderung: Man muss lernen, damit umzugehen

# Zentrales Konzept: Das Repository

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



# History




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

# Typischer Arbeitsablauf

1. Repository erzeugen oder klonen: `git init`, `git clone`
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`

# `git init`, `git clone`

| | |
|---|---|
| `git init` | initialisiert ein `git`-Repo im jetzigen Verzeichnis |
| `git clone` *`url`* | klont das Repo aus *`url`* |
| `rm -rf .git` | löscht alle Spuren von `git` aus dem Repo |

# `git status`, `git log`

| | |
|---|---|
| `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 |

# git diff

| | |
|---|---|
| `git diff` | zeigt Unterschiede zwischen Staging und Arbeitsverzeichnis |
| `git diff --staged` | zeigt Unterschiede zwischen letzten Commit und Staging |
| `git diff commit1 commit2` | zeigt Unterschiede zwischen zwei Commits |

# git commit

| | |
|---|---|
| `git commit` | erzeugt Commit aus jetzigem Staging-Bereich, öffnet Editor für Commit-Message |
| `git commit -m "`*`message`*`"` | Commit mit *`message`* als Message |
| `git commit --amend` | letzten Commit ändern (fügt aktuellen Staging hinzu, Message bearbeitbar) |

- → Wichtig: Sinnvolle Commit-Messages
  - → Erster Satz ist Zusammenfassung
- → 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

→ 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

| | |
|---|---|
| `git checkout` *`commit`* | Commit ins Arbeitsverzeichnis laden |
| `git checkout` *`file`* | Änderungen an Dateien verwerfen (zum letzten Commit zurückkehren) |

# git stash

| | |
|---|---|
| `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__/
```

# SSH-Keys

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, keine Passwörter mehr erforderlich

SSH-Keys:

1. `ssh-keygen`
2. Standardeinstellungen ok (kein Passwort!)
3. `cat ~/.ssh/id_rsa.pub`
4. Ausgabe ist Public-Key, beim Server eintragen (im Browser)

# Hoster

GitHub

- größter Hoster
- viele open-source Projekte
- keine (kostenlosen) privaten Repos

Atlassian Bitbucket

- kostenlose private Repos mit höchstens fünf Leuten
- keine Speicherbegrenzungen

GitLab

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