# Änderungen verwalten mit `git`

PeP et al. Toolbox Workshop

**PeP et al. e.V.**
Physikstudierende und
ehemalige Physikstudierende
der TU Dortmund

2023

# Was ist Versionskontrolle?

- Versionskontrollsoftware speichert Änderungen an Dokumenten / Dateien
- Das kann fast alles sein:
  - Software
  - Rechtliche Dokumente
  - Dokumentation
  - Wissenschaftliche Veröffentlichungen
  - Bilder
  - Baupläne, CAD-Zeichnungen
  - ...
- Ein *Schnappschuss* eines Projektes nennt man *Revision*
- Alle Revisionen zusammen bilden die *Geschichte* des Projekts

# Warum also Versionskontrolle nutzen?

- → Erlaubt, an eine beliebige Revision zurückzukehren
- → Kann die Unterschiede zwischen Revisionen anzeigen
- → Macht Zusammenarbeit an Projekten einfacher
- → Dient auch als Backup

# Warum also Versionskontrolle nutzen?

Versionskontrollsoftware macht die Beantwortung der folgenden Fragen einfach:

**Was?** Was wurde von Revision *A* auf Revision *B* geändert

**Wer?** Wer hat eine Änderung gemacht? Wer hat alles zum Projekt beigetragen?

**Warum?** Warum wurde diese Änderung gemacht?

**Wann?** Wann wurde ein bestimmter Bug eingeführt bzw. behoben?

# Warum also Versionskontrolle nutzen?

Versionskontrollsoftware macht die Beantwortung der folgenden Fragen einfach:

**Was?** Was wurde von Revision *A* auf Revision *B* geändert

**Wer?** Wer hat eine Änderung gemacht? Wer hat alles zum Projekt beigetragen?

**Warum?** Warum wurde diese Änderung gemacht?

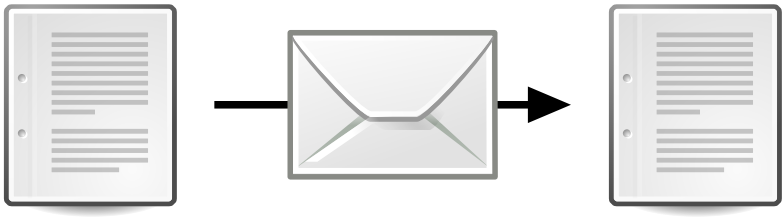
**Wann?** Wann wurde ein bestimmter Bug eingeführt bzw. behoben?

Versionskontrolle ist eine fundamentale Bedingung
für nachvollziehbare, reproduzierbare Wissenschaft.

Wie arbeitet man am besten an einem Protokoll zusammen?

Idee: Austausch über Mails / Messenger

# Mails / Messenger: 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 Cloud Speicher

Dropbox, Google Drive, OneDrive, Nextcloud, iCloud, ...

# Cloud Speicher: Probleme

- → 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`


- 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
- Es gibt auch Plugins für Editoren, z.B. VS Code

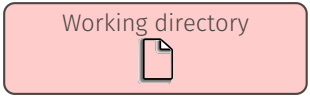
# Was bringt `git` für Vorteile?

- → 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

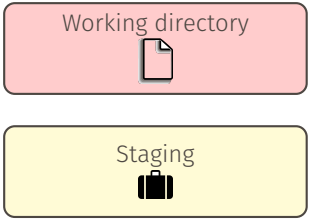
- Erzeugen mit `git init`
- Damit wird der aktuelle Ordner zu einem Repository




Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

# Zentrales Konzept: Das Repository

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

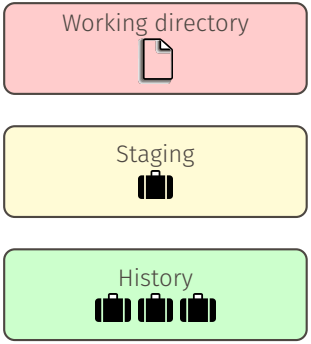



Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

Änderungen, die für einen „commit“ vorgemerkt sind.

# Zentrales Konzept: Das Repository

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



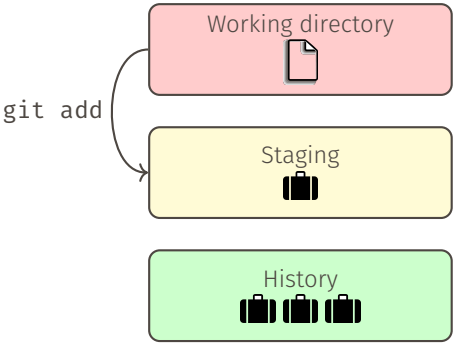

Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

Änderungen, die für einen „commit“ vorgemerkt sind.

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

# Zentrales Konzept: Das Repository

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



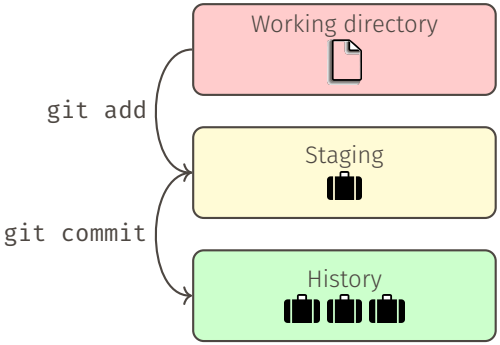

Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

Änderungen, die für einen „commit“ vorgemerkt sind.

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

# Zentrales Konzept: Das Repository

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



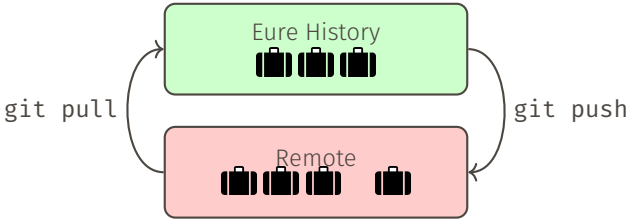

Aktuelles Arbeitsverzeichnis, Inhalt des Ordners im Dateisystem.

Änderungen, die für einen „commit“ vorgemerkt sind.

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

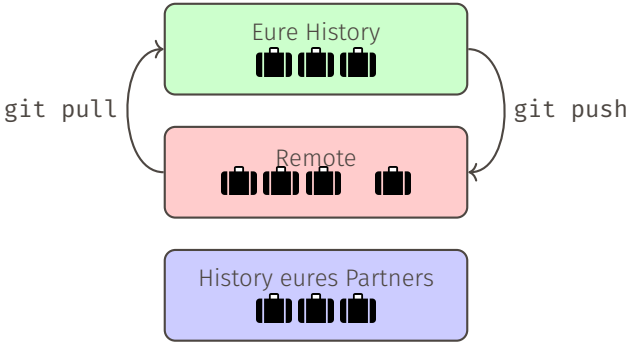
# Remotes

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



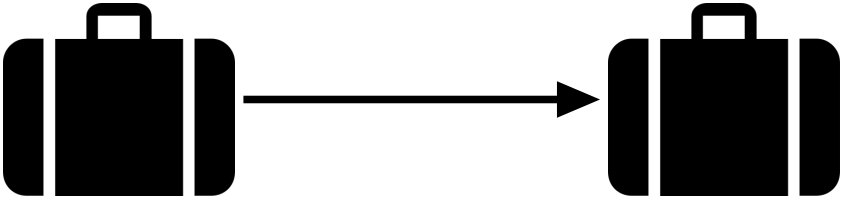
# Remotes

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



# Konzept: Commits

- → Stand des Repositories zu einem Zeitpunkt
- → Erlaubt das Hinzufügen von Kommentaren: Was wurde getan seit dem letzten Commit?
- → Sind die *Versionen* des Repositories

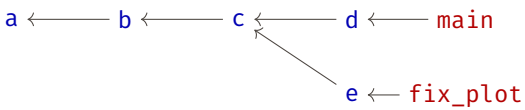


`Commit 1` `Commit 2`

# History

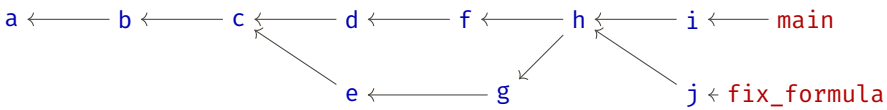

→ Commit: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
  → Enthält Commit-Message (Beschreibung der Änderungen)
  → Wird über einen Hash-Code identifiziert
  → Zeigt immer auf seine(n) Vorgänger

# History



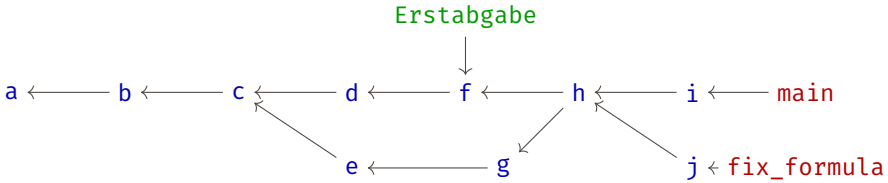

- → Commit: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
  - → Enthält Commit-Message (Beschreibung der Änderungen)
  - → Wird über einen Hash-Code identifiziert
  - → Zeigt immer auf seine(n) Vorgänger
- → Branch: benannter Zeiger auf einen Commit
  - → Entwicklungszweig
  - → Im Praktikum reicht meist der Standard-Branch: `main`
  - → Wandert weiter mit dem aktuellsten Commit

# History




- → 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 meist der Standard-Branch: `main`
  - → Wandert weiter mit dem aktuellsten Commit

# History

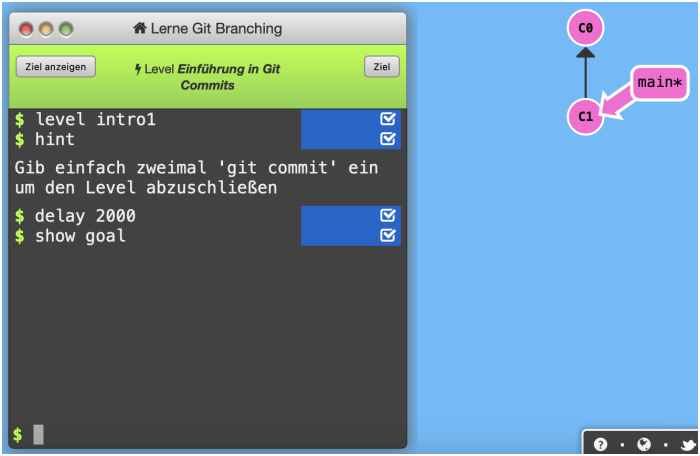



- → Commit: Zustand/Inhalt des Arbeitsverzeichnisses zu einem Zeitpunkt
  - → Enthält Commit-Message (Beschreibung der Änderungen)
  - → Wird über einen Hash-Code identifiziert
  - → Zeigt immer auf seine(n) Vorgänger
- → Branch: benannter Zeiger auf einen Commit
  - → Entwicklungszweig
  - → Im Praktikum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

# Typischer Arbeitsablauf

1. Neues Repo? Repository erzeugen oder klonen: `git init`, `git clone`
   Repo schon da? Änderungen herunterladen: `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`

# Zum selber ausprobieren und Lernen:




https://learngitbranching.js.org/

# Der Start: `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 Repository, nicht reversibel ohne Backup, wird eigentlich nie gebraucht |

# Was passiert in Git: `git status`, `git log`

| | |
|---|---|
| `git status` | zeigt Status des Repos (welche Dateien sind neu, gelöscht, verschoben, bearbeitet) |
| `git status -s` | Kurzform von `git status`, zeigt Liste von geänderten Dateien |
| `git log` | listet Commits in aktuellem Branch. Hat viele Optionen. |

# Staging Bereich: `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 add -u` … | fügt *alle* von Git getrackten und vom User veränderten Dateien 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) |

**Niemals commits ändern, die schon in den `main` branch 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

# Mit der remote History (dem Server) interagieren

`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`

# Zu früheren Versionen zurückkehren

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

# Kurz an was anderem Arbeiten

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

GitHub

- größter Hoster
- viele open-source Projekte
- Unbegrenzt private Repositories für Studenten und Forscher: 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

Weitere Open Source Optionen, auch zum selbst hosten: Gitea, Forgejo

GitHub

- größter Hoster
- viele open-source Projekte
- Unbegrenzt private Repositories für Studenten und Forscher: 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

Weitere Open Source Optionen, auch zum selbst hosten: Gitea, Forgejo

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

# SSH-Keys

Git kann auf mehrere Arten mit einem Server kommunizieren:

**HTTPS** → Mit Nutzername / Passwort: War lange die einfachste Möglichkeit. Wird aber von GitHub aus Sicherheitsgründen nicht mehr einfach unterstützt.
→ Mit „Personal Access Token“. Neues Verfahren für GitHub über HTTPS.

**SSH** : Keys müssen erzeugt und eingestellt werden, Passwort für den Key muss, wenn ein „SSH-Agent“ verwendet wird, nur einmal pro Session eingegeben werden.

SSH-Keys:

1. `ssh-keygen -t ed25519 -C "your_email@example.com"`
2. Passwort wählen
3. `cat ~/.ssh/id_ed25519.pub`
4. Ausgabe ist Public-Key, beim Server eintragen (im Browser)

Für den Agent, falls noch nicht vom Betriebsystem eingerichtet (z. B. Windows mit WSL):

5. `echo 'eval $(ssh-agent -s)' >> ~/.bashrc`
6. `echo 'AddKeysToAgent yes' >> ~/.ssh/config`

Doku: `https://docs.github.com/en/authentication/connecting-to-github-with-ssh`

# Extra Slide für sauberere Projekt Historien: `git pull --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

# Video-Aufzeichnung

Im Rahmen einer Schulung ist 2021 eine Videoaufzeichnung einer ausführlicheren Git-Einführung angefertigt worden, die auf diesem Kurs basiert:

**Teil 1** `https://www.youtube.com/watch?v=R2BCOtPwtXc`

**Teil 2** `https://www.youtube.com/watch?v=ZEcklfIp6Og`