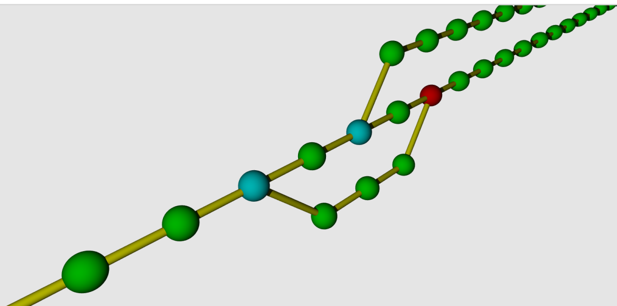
Karlsruher Institut für Technologie

# Versionsverwaltung mit GIT und gitlab

Andreas Poenicke | 10. April 2024



# Inhaltsverzeichnis

Karlsruher Institut für Technologie


## 1. Versionsverwaltung

- Aufgaben von Versionverwaltungssystemen
- Arten von Verwaltungssystemen

## 2. Das Git VCS

- Repositorium erstellen
- Git konfigurieren
- Erste Schritte

## 3. Arbeiten mit Revisionen (Commits)

- Historie - Git-Verlauf (Log)
- „Undo“ - Die Historie Nutzen

## 4. Remote Repositories - GitLab

# Wozu Versionverwaltung?

Karlsruher Institut für Technologie

## Szenario 1

- Seminarvortrag, Diplomarbeit
- Abschnitt entfernt („brauch ich nicht“)
- Zwei Wochen später: Wie bekomme ich den Abschnitt wieder?

# Wozu Versionverwaltung?

Karlsruher Institut für Technologie

## Szenario 1

- Seminarvortrag, Diplomarbeit
- Abschnitt entfernt („brauch ich nicht“)
- Zwei Wochen später: Wie bekomme ich den Abschnitt wieder?

## Szenario 2

- Fehler in der Numerik:
- „Was hab ich letzte Nacht noch geändert“?
- „Wann hat sich der Fehler eingeschlichen und welche Ergebnisse sind betroffen?“

# Wozu Versionverwaltung?

Karlsruher Institut für Technologie

## Szenario 1

- Seminarvortrag, Diplomarbeit
- Abschnitt entfernt („brauch ich nicht“)
- Zwei Wochen später: Wie bekomme ich den Abschnitt wieder?

## Szenario 2

- Fehler in der Numerik:
- „Was hab ich letzte Nacht noch geändert“?
- „Wann hat sich der Fehler eingeschlichen und welche Ergebnisse sind betroffen?“

## Szenario 3

- Synchronisierung der Arbeit auf dem Notebook mit Pool/Arbeitsplatz
- Zusammenführung der jeweiligen Änderungen?

# Der steinige Weg, von Hand...



## Lösung?

- Sicherungskopien mit unterschiedlichen Namen
- Regelmäßige automatische Snapshots
- Manuelle Vergleiche

# Der steinige Weg, von Hand...

Karlsruher Institut für Technologie

## Lösung?

- Sicherungskopien mit unterschiedlichen Namen
- Regelmäßige automatische Snapshots
- Manuelle Vergleiche

## Aber...

- Umkopieren vergessen?
- Fehleranfällig! (Gerade nach langer Arbeit)
- Keine weitere Information über die Versionen
- Kann der Rechner nicht die stupide Arbeit übernehmen?

# Versionsverwaltungssystem / Version Control System (VCS)

Karlsruher Institut für Technologie

- Protokollierung der Änderungen an Dateien und Daten
- Wiederherstellung beliebiger vorheriger Versionen
- Vergleich verschiedener Versionen
- Gruppenarbeit: Unabhängiges bearbeiten der selben Dateien/Daten
- Zusammenführen (Merging) der Änderungen mehrerer Personen

# Nachteile eines Versions Control Systems

Karlsruher Institut für Technologie

- Zusätzliche Komplexität
- Man muss es wirklich regelmässig benutzen
- Automatisierte Änderungen an den Files ungünstig
  (z.B. Automatische Formatierung durch Editoren)

# Arten von Verwaltungssystemen

Karlsruher Institut für Technologie

- Zentrale Systeme
  - CVS
  - Subversion
  - uvm. . . .
- Verteilte Systeme
  - GIT
  - Mercurial (Hg)
  - Bazaar NG (bzr)

# Zentrale Versionsverwaltung

Karlsruher Institut für Technologie

- Ein zentrales „repository“
- Checkout, edit, checkin
- Großer Aufwand für kleine Projekte (Protokolle, Programmieraufgaben, Masterarbeit etc.) (Eigenes Repository aufsetzen, . . . )
- Zugang zum Repository nötig. Zum Arbeiten immer online

# Verteilte Versionsverwaltung

Karlsruher Institut für Technologie

- Jedes Arbeitsverzeichnis ist auch ein Repository
- Neues Verzeichnis unter Versionsverwaltung nehmen: trivial
- Remote Repositories: Gleichberechtigt, keine „Authorities“
- Austausch per Mail möglich, Versionierungsdaten werden beibehalten

# Verzeichnis unter Versionskontrolle stellen

# Verzeichnis unter Versionskontrolle stellen

Karlsruher Institut für Technologie

## Leeres Repository im aktuellen Verzeichnis erstellen

```
~$ cd Git-example
~/Git-example$ git init
~/Git-example$ git status
```

## Das Repositorium

Neu hinzugekommen: Ein Verzeichnis `.git` in dem das eigentlich Repositorium liegt.

# Verzeichnis unter Versionskontrolle stellen

Karlsruher Institut für Technologie

## Leeres Repository im aktuellen Verzeichnis erstellen

```
~$ cd Git-example
~/Git-example$ git init
~/Git-example$ git status
```

## Das Repositorium

Neu hinzugekommen: Ein Verzeichnis `.git` in dem das eigentlich Repositorium liegt.

## Vorsicht

Durch Löschen des `.git` Verzeichnisses

- werde ich das Repositorium wieder los ...
- ... aber verliere auch alle Versionsinformationen etc.

# Konfiguration von Git

Karlsruher Institut für Technologie

## Übersicht über die bestehende Konfiguration

```
~/Git-example$ git config --list
~/Git-example$ git config --list --show-scope
```

Git wendet nacheinander drei Konfigurationen an:

- **system** alle Repositorien
- **global** alle Repositorien von mir
- **local** nur dieses Repositorium

## Wo liegt die Konfiguration?

```
~/Git-example$ git config --list --show-origin
```

# Konfiguration von Git

Karlsruher Institut für Technologie

## Wichtige globale Einstellungen

```
$ git config --global user.name "Vlad Dracula"
$ git config --global user.email "vlad@tran.sylvan.ia"
```

## Sinnvolle globale Einstellungen

```
$ git config --global core.editor "code --wait"   # Editor
$ git config --global init.defaultBranch "main"   #
```

## Achtung

Ohne `user.name` oder `user.email` können keine „`commits`“ (s.u.) durchgeführt werden.

## Alle Dateien aufnehmen

```
~/Git-example $ git add .
~/Git-example $ git status
~/Git-example $ git commit -m "Initial import"
```

## git commit

Mit `git commit` werden die Änderungen an das Repositorium übertragen.
Wichtige Optionen:

**-m** Logeintrag für die Änderung direkt angeben. (Nur bei einfachen Änderungen sinnvoll)

**-a** alle geänderten Dateien übertragen, fügt aber keine neuen Dateien hinzu. (recht unkontrolliert, eher vermeiden)

## Tipp

Git hat eine eingebaute Hilfe: `git help commit`
Für alle Befehle gibt es auch eine Kurzübersicht: `git commit -h`

# Dateien ändern

Karlsruher Institut für Technologie

## Änderungen mit dem persöhnlichen Lieblingseditor

```
~/Git-example $ vi thesis.tex
```

## änderungen im Versionssystem speichern

```
~/Git-example $ git add thesis.tex
~/Git-example $ git status
~/Git-example $ git commit -m "Started chapter 2"
```

## Moment!

`git add` nochmal? Sicher?

# Die Staging-Area

Karlsruher Institut für Technologie

## Zentrales Konzept von Git: Commit in zwei Schritten

- `git add` überträgt Änderungen in Zwischenspeicher (Staging Area)
- Staging Area kann weiter verändert werden.
- `git commit` überträgt Änderungen aus Staging Area ins Repositorium
- Staging-Area wird dabei wieder geleert.
- Ermöglicht den „Commit“ in kleinen Einzelschritten aufzubauen

Mächtig um Änderungen an verschiedenen Stellen um Filesystem zusammenzufassen

# Dateien ignorieren - `.gitignore`

Karlsruher Institut für Technologie

## Dateien ignorieren

- `git status` zeigt alle Änderungen und neue Dateien an.
- Aber nicht alle Dateien sollen überwacht oder aufgenommen werden. Z.B. `.aux`-Dateien bei LaTeX
- Lösung: Datei mit Suchmuster die ignoriert werden sollen
- Dateien können nicht versehentlich aufgenommen werden

## `.gitignore`

```
~/Git-example$ echo "Test" > test.tmp
~/Git-example$ git status
~/Git-example$ vi .gitignore      # Füge *.tmp hinzu
~/Git-example$ git status
~/Git-example$ git status --ignored
~/Git-example$ git add test.tmp
```

# Dateien wieder löschen

Karlsruher Institut für Technologie

## Datei entfernen

```
~/Git-example$ rm test.txt
~/Git-example$ git add test.txt
~/Git-example$ git commit -m "test.txt gelöscht"
```

## Folge

- Datei ist nicht mehr im Arbeitsverzeichnis
- Datei wird nicht mehr von Git auf Änderungen überwacht
- Alte Version immer noch im Repositorium vorhanden

## Abkürzung

```
~/Git-example$ git rm test.txt
~/Git-example$ git commit -m "test.txt gelöscht"
```

# Git und Verzeichnisse

Karlsruher Institut für Technologie

## Versuche Verzeichnis hinzunehmen

```
~/Git-example$ mkdir NewDir
~/Git-example$ git add NewDir
~/Git-example$ git status
```

- Git speichert keine leeren Verzeichnisse
- Verzeichnisse nur Orte der Dateien
- `git add DIR` übernimmt alle Dateien in `DIR`

# Stöbern in der Vergangenheit

Karlsruher Institut für Technologie

## Auf der Konsole

```
~/Git-example$ git log
~/Git-example$ git log -3
~/Git-example$ git log --oneline
~/Git-example$ git log --graph
```

## Graphisch

```
~/Git-example$ gitk
```

# Stöbern in der Vergangenheit

Karlsruher Institut für Technologie

## Commit ansehen

```
~/Git-example$ git show a6c9c1f
```

## Ältere Versionen ansehen

```
~/Git-example$ git show a6c9c1f thesis.tex
```

# Stöbern in der Vergangenheit

Karlsruher Institut für Technologie

## Versionen vergleichen

```
~/Git-example$ git diff a6c9c1f..HEAD
~/Git-example$ git diff HEAD~1..HEAD
~/Git-example$ git diff --cached      # Unterschied zwischen Staging Area und HEAD
~/Git-example$ git diff               # Unterschied zwischen Filesystem und Staging Area/HEAD
~/Git-example$ git diff --color-words
~/Git-example$ gitk
```

# `git add` zurücknehmen, Unstaging

Karlsruher Institut für Technologie

## Dateien aus der Staging-Area entfernen

Wurden Dateien irrtümlich mit `git add` zum Index hinzugefügt, können sie dort wieder entfernt werden. Änderungen an den Dateien im Arbeitsverzeichnis selbst bleiben erhalten.

`git status` schlägt dies auch vor:

```
  Changes to be committed:
    (use "git restore --staged <file>..." to unstage)
        modified: ...
```

## Unstaging

```
~/Git-example $ git restore --staged file1 file2
```

# Zeitreisen

Karlsruher Institut für Technologie

## Ältere Dateiversionen wiederherstellen

```
~/Git-example $ git log
~/Git-example $ git checkout a6c9c1f thesis.tex
~/Git-example $ git checkout HEAD~1 thesis.tex
```

Datei im Workspace wird zurückgesetzt **und** gleich in die Staging Area aufgenommen.

# Zeitreisen

Karlsruher Institut für Technologie

## Ältere Dateiversionen wiederherstellen

```
~/Git-example $ git log
~/Git-example $ git checkout a6c9c1f thesis.tex
~/Git-example $ git checkout HEAD~1 thesis.tex
```

Datei im Workspace wird zurückgesetzt **und** gleich in die Staging Area aufgenommen.

## Vorsicht

Ohne Angabe einer Datei wird alles auf die alte Version gesetzt

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
```

# Zeitreisen

Karlsruher Institut für Technologie

## Alte Version ansehen

```
~/Git-example $ git checkout HEAD~1
```

## detached HEAD

```
HEAD (refers to commit 'b')
|
v
a---b---c---d  branch 'main' (refers to commit 'd')
```

Nur sinnvoll zum Ansehen, oder um einen neuen Branch (s.u.) zu starten.

## Zurück zur aktuellen Version

```
    ~/Git-example $ git checkout main   # oder master
```

# Kennzeichner (Tags)

Karlsruher Institut für Technologie

## a6c9c1f kann sich kein Mensch merken → Tags

```
~/Git-example $ git tag second
~/Git-example $ git tag first HEAD~1
~/Git-example $ git tag
~/Git-example $ gitk
```

## Verwendung von Tags

```
~/Git-example $ git checkout first
~/Git-example $ git diff first..second
~/Git-example $ gitk
```

## Etiketten entfernen

```
~/Git-example $ git tag -d second
```

# Änderungen zurücknehmen(1)

Karlsruher Institut für Technologie

## Letzten Commit zurücknehmen

```
~/Git-example $ git revert HEAD
```

Generiert automatisch einen neuen Commit, der alle Änderungen rückgängig macht.

## Einen bestimmten Commit zurücknehmen

```
~/Git-example $ git revert a6c9c1f
```

Kann Konflikte auslösen, die einen „merge“ nötig macht

## Bemerkung

Die Arbeitskopie darf keine Änderungen enthalten!

# Änderungen zurücknehmen(2)

Karlsruher Institut für Technologie

**Warnung**

Dieser Befehl birgt die Gefahr von Datenverlust!

**Letzte Commits verwerfen**

```
~/Git-example $ git reset --soft HEAD~1     # Nur Repository wird zurückgesetzt
~/Git-example $ git reset --mixed HEAD~1    # Arbeitskopie wird beibehalten
~/Git-example $ git reset --hard HEAD~1     # Alles wird zurückgesetzt
```

**Warnung**

Nie Commits verwerfen, die schon an ein Remote-Repository übertragen wurden!
Besser: -> `git revert`

# Remote Repositories - Gitlab/GitHub

Karlsruher Institut für Technologie

## Repositorien

**GitHub** (Microsoft) Standardort für OpenSource Projekte, Kooperation mit Externen

**GitLab** (GitLab) Alternative zu github

**KIT-GitLab** (KIT) Sichtbarkeit: Pr ivat, KIT-Intern oder öffentlich, Kooperation nur KIT-intern

**+ ...** Diverse rein kommerzielle Angebote, z.B. Atlassian Bitbucket

## Authentifizierung mit git-Befehl

- HTTPS: Username + Password
- HTTPS: Username + Token
- ssh: Key

# Remote Repositories - Gitlab/GitHub

## Repositorien

- **GitHub** (Microsoft) Standardort für OpenSource Projekte, Kooperation mit Externen
- **GitLab** (GitLab) Alternative zu github
- **KIT-GitLab** (KIT) Sichtbarkeit: Pr ivat, KIT-Intern oder öffentlich, Kooperation nur KIT-intern
- **+ . . .** Diverse rein kommerzielle Angebote, z.B. Atlassian Bitbucket

## Authentifizierung mit git-Befehl

- ~~HTTPS: Username + Password~~ (bei KIT-GitLab nicht möglich!)
- HTTPS: Username + Token
- ssh: Key

# Token Authentifizierung – eher unter Windows

Karlsruher Institut für Technologie

## git-Konfiguration:

```
> git config --global credential.helper=manager   # Sollte default sein
> git config --global credential.https://gitlab.kit.edu.username YOURUSERNAME
> git config --global credential.https://gitlab.kit.edu.helper=manager-core
```

## In Gitlab

- Account → Edit profile
- Acess Tokens → Personal Access Tokens → Add new token
- Create personal access token → Copy (Achtung später nicht mehr abrufbar)

## via git Authentifizieren, z.B.

```
> git clone git@gitlab.kit.edu:andreas.poenicke/git-example.git Git-example1
```

Token ist jetzt im Password-Manager!

# SSH-Key Authentifizierung - eher unter Linux

Karlsruher Institut für Technologie

## ssh-key generieren

```
~$ ssh-keygen -t ed25519   # Immer Passwort vergeben!
```

## In Gitlab

- Account → Edit profile
- SSH Keys → Add new key
- Inhalt von ~/.ssh/id_ed25519.pub als Key einfügen

## Testen:

```
$ ssh -T git@gitlab.kit.edu   # Password von Key
$ ssh-add                     # Password von Key
$ ssh -T git@gitlab.kit.edu   # kein Password
```

# Repositorium übertragen – Clone

Karlsruher Institut für Technologie

## In GitLab

- "+"$\rightarrow$ New project/repository
- Create blank project
- Project name: Git-example
- Visibility Level: Private
- Create project $\rightarrow$ öffnet Projekt
- Rechte Seite $\rightarrow$ Code $\rightarrow$ Clone with SSH / Clone with HTTPS (kopieren)

## git clone

```
~$ git clone git@gitlab.kit.edu:andreas.poenicke/git-example.git Git-example1
~$ cd Git-example1
~Git-example1$ git remote -v
~Git-example1$ git config -l
```

# Konflikte

Karlsruher Institut für Technologie

## pull -Strategie konfigurieren

- Git versucht Konflikte automatisch zu lösen.
- dazu muss es wissen wie:

Änderung im Repositorium und Änderung lokal:

## Rechner/Verzeichnis 2

```
~Git-example2$ git pull
hint: You have divergent branches and need to specify how to reconcile them.
...
~Git-example2$ git config pull.rebase false
~Git-example2$ git pull
```

## Gleichzeitigen Änderungen - Pull

Beide Änderungen werden zusammgefasst (gemerged) und ein neuer Commit erstellt

# Konflikte - merge per Hand

Karlsruher Institut für Technologie

## Rechner/Verzeichnis 1

```
~Git-example1$ vi file1.txt
~Git-example1$ git commit -a -m "edited File1"
~Git-example1$ git push
```

## Rechner/Verzeichnis 2

```
~Git-example2$ vi file1.txt
~Git-example2$ git commit -a -m "also edited File1"
~Git-example2$ git push
~Git-example2$ git pull
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

# Arbeit:

Karlsruher Institut für Technologie

## Rechner/Verzeichnis 2

```
~Git-example2$ git status
    Unmerged paths:
    (use "git add <file>..." to mark resolution)
          both modified:   file1.txt
~Git-example2$ vi file1.txt
~Git-example2$ git status
~Git-example2$ git commit -m "Merge changes from GitLab"
~Git-example2$ git push
```

## Rechner/Verzeichnis 1

```
~Git-example1$ git pull
~Git-example1$ git log
```

## Weiterführendes

Karlsruher Institut für Technologie

### Das waren nur die Grundlagen . . .

- Branching
- CI\CD - Continuous Integration and Continuous Delivery
- uvm. . . .

### Weiterführende Tutorials

- Pro Git
- GitLab Docs

# Fragen ?!

# Commits nachträglich ergänzen

Karlsruher Institut für Technologie

## Mist! Ich hab was vergessen!

```
~/Git-example$ git add thesis.tex
~/Git-example$ git status
~/Git-example$ git commit --amend
```

## Achtung

Gefährlich, wenn Änderungen schon auf ein Remote-Repository gepusht wurden.