

Verteilte Systeme – Übung

Grundlagen: Git

Sommersemester 2025

Harald Böhm, Christian Berger, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Informatik 4 (Systemsoftware)

<https://sys.cs.fau.de>



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Versionsverwaltung mit Git

- Grundlagen

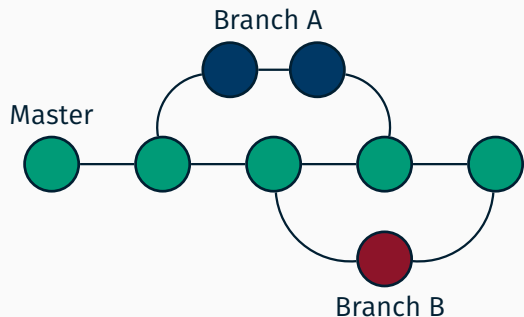
- Branches

- Konflikte

- Git in Eclipse

Git – Warum eigentlich?

- Vorteile eines Versionskontrollsystems
 - Ermöglicht Zusammenarbeit mit mehreren Entwicklern
 - Einfaches Zusammenführen von Code und Erkennen von Konflikten
 - Parallele Entwicklung mehrerer Features
 - Fehlersuche uvm. durch „zurückspringen“ zu alten Versionen
- Weit verbreitet und öffentliche Hosting Plattformen (z.B. Gitlab, GitHub)



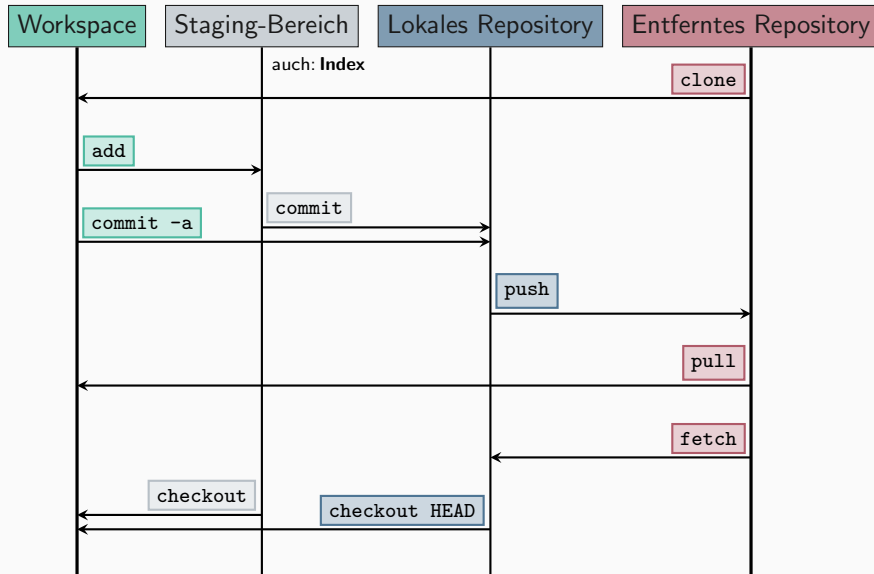
- Übungsaufgaben sollten im Git bearbeitet werden
- Benötigt ein Benutzerkonto bei <https://gitlab.cs.fau.de>
 - Konto werden automatisch mit dem IdM Single-Sign-On verknüpft
→ "Sign-in with *FAU Single Sign-On*"
 - Öffentliche(n) SSH-Schlüssel hinzufügen:
 - Oben rechts auf das Profil-Logo und auf „Profile Settings“ klicken
 - Reiter „SSH Keys“ auswählen
 - Einen oder mehrere SSH-Schlüssel hinzufügen
(siehe auch: <https://gitlab.cs.fau.de/help/ssh/README>)
- Repositories werden von uns **anhand der Übungsgruppen** erstellt.
Jeder Gruppenteilnehmer erhält automatisch Zugriff zum Repository seiner Gruppe.



Versionsverwaltung mit Git

Grundlagen

Überblick über den Git-Arbeitsablauf



- Erstellen einer **lokalen** Arbeitskopie über ein **entferntes** Repository

- Befehl: `> git clone <URL>`

- Beispiel: `git clone` über SSH (SSH-Schlüssel nötig!)

```
> git clone git@gitlab.cs.fau.de:i4-exercise/vs/ss21/vs-gruppe-42.git
```

(URL des GitLab-Repository steht auf der jeweiligen Projektübersichtsseite)

```
> git clone git@gitlab.cs.fau.de:i4-exercise/vs/ss21/vs-gruppe-42.git
Cloning into 'vs-gruppe-42'...
X11 forwarding request failed
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.
Resolving deltas: 100% (1/1), done.

> ls vs-gruppe-42/
README.md
```

- Dateien werden zunächst nur dem Staging-Bereich hinzugefügt oder davon entfernt
 - Es wird nur der **aktuelle** Zustand hinzugefügt
 - Änderungen haben erst beim nächsten Commit Auswirkungen auf das Repository
 - Einzelne Änderungen durch Option `-p` bzw. `--patch` auswählbar
- Änderung(en) zu Staging-Bereich hinzufügen (bzw. Datei(en) entfernen)

```
> git add [-p] <file(s)-to-add>  
> git rm <file(s)-to-remove>
```

- Änderung(en) aus Staging-Bereich entfernen

```
> git reset HEAD [-p] <file(s)-to-reset>
```

- Änderung(en) im **Workspace** verwerfen

```
> git checkout -- [-p] <file(s)-to-checkout>
```

Seit Version 2.23 gibt es 'git restore', das 'git reset HEAD' und 'git checkout --' ersetzt.
Laden der neuen git Version im CIP: 'load module git'

■ Auswirkungen des nächsten Commits überprüfen

```
> git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: src/Application.java

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

Untracked files:

(use "git add <file>..." to include in what will be committed)

Makefile

Unterschiedliche Ausprägungen von diff:

- *Standardverhalten*: Diff zwischen Workspace und Staging-Bereich

```
> git diff [<filename>]
```

- Diff zwischen Staging-Bereich und aktuellem Commit

```
> git diff --cached [<filename>]
```

- Diff zwischen Workspace und einem bestimmten Commit

```
> git diff <commit> [<filename>]
```

- Unterschiede zu Dateien in einem Remote-Branch

```
> git diff <local_branch> <remote_branch>
```

Zum Beispiel:

Unterschied von lokalem Branch 'master' zu Zustand von 'master' im entfernten Repository

(local_branch := master und remote_branch := origin/master)

→  Vorheriges git fetch (siehe Folie 2-13) ratsam.

- Änderungen vom Staging-Bereich ins lokale Repository übernehmen

```
> git commit [<file(s)-to-commit>]
```

Nützliche Parameter:

- a Alle modifizierten Dateien übernehmen
- m <message> message als Commit-Nachricht verwenden
- amend Vorherigen Commit modifizieren

- Commits vom lokalen in das **entfernte** Repository einprüfen

```
> git push [[remote_name] [branch_name]]
```

Wenn das entfernte Repository **zusätzliche, noch nicht lokal vorhandene** Commits enthält, muss das lokale Repository **zuerst** aktualisiert werden.

- Zustand aus entferntem Repository holen und integrieren

```
> git pull [[remote_name] [branch_name]]
```

Eventuell Konfliktauflösung notwendig, siehe „Konflikte“ ab Folie 19

- Aktualisierung der lokalen Sicht auf das entfernte Repository

```
> git fetch --all
```

- Änderungen werden nur gelesen, noch nicht eingespielt
- Ermöglicht Vergleich von lokalem und entferntem Stand, z.B.

```
> git diff master origin/master
```

■ Betrachten von Commits im lokalen Repository

```
> git log

commit f8ceebed8d581cab736350c055b072db148987cd
Author: Laura Lawniczak <lawniczak@cs.fau.de>
Date:   Fri Oct 25 13:11:11 2019 +0200

Add initial README file

[...]
```

- Aufbau: Commit-ID, Autor, Datum, Commit-Nachricht
- Ausgeben der Änderungen eines Commits: `> git log -p [<commit-id>]`

■ Graphische Aufarbeitung im Terminal

```
> tig [<file(s)-to-view-log-for>]
```

■ Git-GUIs mit graphischer Darstellung

- git-cola
- gitk

Kompilierte Dateien (z. B. `.class`-Dateien) sollten nicht ins Repository!

- Zu ignorierende Dateien in `.gitignore` eintragen

```
# Ignore class files
*.class
```

- Sollte in das Repository eingecheckt werden
- Greift nicht für bereits eingecheckte Dateien
→ ggf. die entsprechende Datei explizit mit `git rm <file>` löschen

- Lokale Änderungen inklusive ignorerter Dateien anzeigen

```
> git status --ignored

[...]
```

Ignored files:
(use "`git add -f <file>...`" to include in what will be committed)
application.class

- E-Mail-Adresse und Name für Commits festlegen

```
> git config --global user.email max@mustermann.de  
> git config --global user.name "Max_Mustermann"
```

- Alle gesetzten Variablen ansehen

```
> git config --list  
  
user.name=Max Mustermann  
user.email=max@mustermann.de  
[...]
```

- Dokumentation: `man 1 git-config`

Versionsverwaltung mit Git

Branches

- Für jedes neue Feature wird üblicherweise ein neuer Branch erstellt

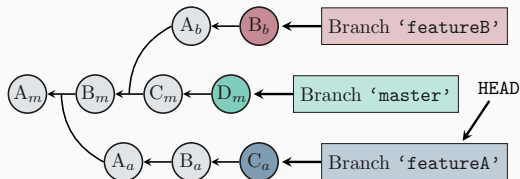
```
> git checkout -b <new_branch_name>
```

- Wechseln zwischen Branches (Workspace und Staging-Bereich bleiben erhalten)

```
> git checkout <branch_name>
```

- Anzeigen aller Branches (-a inkludiert entfernte Branches)

```
> git branch [-a]
master
* featureA
featureB
```



- Irgendwann müssen verschiedene Zweige vereint werden

- Prinzipiell zwei unterschiedliche Wege

- Klassischer Merge:

→ Mergen von <branch> in <other_branch>:

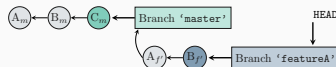
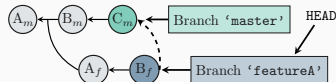
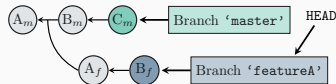
```
> git checkout <other_branch>
> git merge <branch>
```

- Einfacher Fall: *fast-forward merge*
 - Fall mit eventuell notwendiger Konfliktauflösung: *3-way merge*

- Rebase:

```
> git checkout <other_branch>
> git rebase [-i] <branch>
```

- Interaktives Rebase (-i): Historie neu schreiben
 -  Sollte nicht auf öffentlichem Branch angewendet werden



Versionsverwaltung mit Git

Konflikte

- Es gibt Konflikte, die git nicht selbstständig auflösen kann

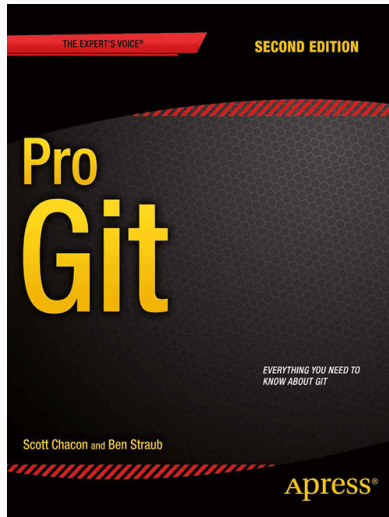
```
> git pull
[...]  
1b09b5d..39efa77  master -> origin/master  
Auto-merging README.md  
CONFLICT (content): Merge conflict in README.md  
Automatic merge failed; fix conflicts and then commit the result.  
  
> cat README.md  
Das ist meine README Datei!  
  
<<<<<< HEAD  
Meine neue Änderung  
=====  
Änderung aus dem gepullten Commit  
>>>>>> 39efa77d814d4aebfec37da8d252cfc80091907
```

- Konflikt muss manuell gelöst werden und Ergebnis committed werden

```
> git add README.md  
> git commit
```



<https://xkcd.com/1597/>



<https://git-scm.com/book/en/v2>

Versionsverwaltung mit Git

Git in Eclipse

- Eclipse enthält Unterstützung für Git

- Schritte zum Einrichten

1. Lokale Kopie des Repositories erstellen (wenn nicht schon per 'git clone')

- „File“ → „Import...“ → „Git“ → „Projects from Git“
- Anschließend „Clone URI“ auswählen und URL aus Gitlab einfügen
- Bei „Branch Selection“ auf weiter klicken
- Bei „Local Destination“ ggf. **Pfad** anpassen
- „Import using the New Project wizard“ auswählen

2. Als Projekt in Eclipse einfügen

- Neues „Java“ → „Java Project“ auswählen
- **„Use default location“ deaktivieren**
- **Pfad des lokalen Repositories eingeben**
 - ⇒ Eclipse erkennt das Git-Repository automatisch
- Rest wie ohne Git

- Git-Befehle sind nach Rechtsklick auf das Projekt über das „Team“-Untermenü verfügbar