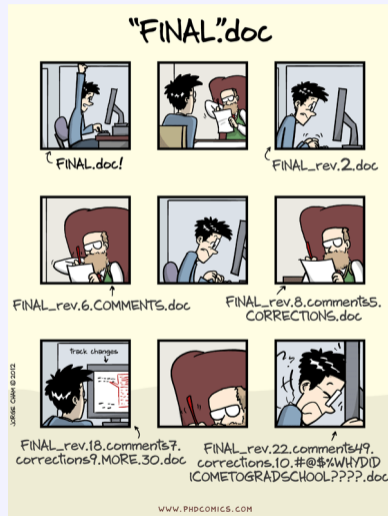


Teil IV

Git

Versionsverwaltung



Versionsverwaltung

- Protokolliert die Änderungen an einer oder mehreren Dateien.
- Ermöglicht es, jederzeit eine alte Version wiederherzustellen.
- Informiert uns, *wann* etwas *von wem* und *warum* geändert wurde.
- Vereinfacht es, mit mehreren Personen an denselben Dateien zusammenzuarbeiten.

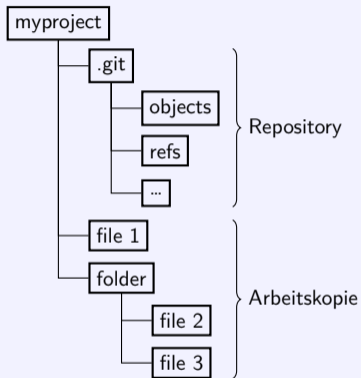
Verteilte Versionsverwaltung

- Alle Mitwirkenden haben eine Kopie der vollständigen Änderungs-Historie.
- Empfohlen: Zusätzliche zentrale Kopie auf einem Server.
- Änderungen offline möglich (von unterwegs, bei Serverausfall, ...).
- Einfache Synchronisation der Änderungen mit anderen Personen (direkt oder via Server).

Git Grundbegriffe

- Ein *Repository* ist eine Sammlung von Dateien mit Änderungs-Historie. Man kann nicht direkt auf die darin enthaltenen Dateien zugreifen.
- Eine *Arbeitskopie* ist ein normaler Ordner auf Ihrem Computer, der eine bestimmte Version der Dateien aus dem Repository enthält. Diese Dateien können Sie beliebig verwenden und auch verändern.
- Ein *Commit* speichert die in der Arbeitskopie vorgenommenen Änderungen ins Repository und erzeugt so eine neue Version. Gespeichert werden:
 - eine Referenz auf die vorherige Version,
 - die an den Dateien vorgenommenen Änderungen,
 - Datum und Uhrzeit des Commits,
 - Autor (Name und Mailadresse) der Änderungen,
 - und ein Kommentar des Autors (z.B. Zusammenfassung der Änderungen oder eine Begründung, warum die Änderungen notwendig waren).

Ordnerstruktur



Der .git Ordner darf nicht von Hand verändert werden!

Git Log



- Commits bilden eine verkettete Liste.
- D ist die aktuelle Version (der neueste Commit).
- A ist der erste Commit im Repository.

git log --stat

```
commit 9ef0941117b3310ec2218604d6b99d7bd2a99048 (HEAD -> master)
Author: Sebastian Schweizer <schweizer@cs.uni-kl.de>
Date: Thu May 2 17:48:41 2019 +0200
```

```
Fehler korrigiert, main Methode muss static sein
```

```
HelloWorld.java | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
commit 5a1ff44ab48bf6c9b2cfb12bd3d89d25d1d72094
Author: Sebastian Schweizer <schweizer@cs.uni-kl.de>
Date: Thu May 2 17:46:23 2019 +0200
```

```
Programm hinzugefügt
```

```
HelloWorld.java | 7 ++++++
1 file changed, 7 insertions(+)
```

```
commit 02077d1341146ae162d107ef5236349fd673a049
Author: Sebastian Schweizer <schweizer@cs.uni-kl.de>
Date: Thu May 2 17:44:06 2019 +0200
```

```
Initialer Commit mit Readme
```

```
README | 3 +++
1 file changed, 3 insertions(+)
```

- Identifikation der Commits durch einen SHA-1 Hash (oft auch nur die ersten 8 Zeichen davon)
- Neueste Version oben
- Log zeigt nur zusammenfassende Daten ohne die jeweiligen Code-Änderungen

Änderungen zwischen (nicht notwendigerweise benachbarten) Versionen:

```
git diff 5a1ff44a 9ef09411
```

```
— a/HelloWorld.java
+++ b/HelloWorld.java
@@ -1,6 +1,6 @@
 public class HelloWorld
 {
-     public void main (String[] args)
+     public static void main (String[] args)
     {
         System.out.println("Hello World!");
     }
 }
```

Repository erstellen

- Neues, leeres Repository erstellen:

```
git init
```

- Bestehendes Repository kopieren:

```
git clone URL
```

Vor unserem ersten eigenen Commit...

Autor für neu generierte Commits konfigurieren:

```
git config --global user.name "Lisa Lista"  
git config --global user.email "lisa.lista@example.com"
```

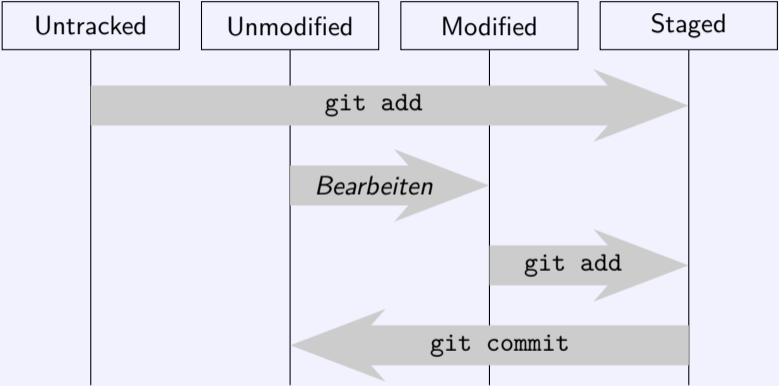
Staging Area

In der *Staging Area* werden Änderungen gesammelt, die für den nächsten Commit vorgesehen sind.

Zustände:

- *Unmodified*: Arbeitskopie stimmt mit Repository überein.
- *Modified*: Änderungen an bestehenden Dateien, die noch nicht für den nächsten Commit vorgesehen sind.
- *Staged*: Für den nächsten Commit vorgesehene Änderungen.
- *Untracked*: Datei in der Arbeitskopie ist nicht Teil des Repositories.

Staging Area



Quelle: <https://git-scm.com/book/de/v2/Git-Grundlagen-%C3%84nderungen-nachverfolgen-und-im-Repository-speichern>

Commit

- Aktuellen Zustand einsehen:

```
git status
```

- Änderungen in Staging Area überprüfen:

```
git diff --cached
```

- Neuen Commit erzeugen:

```
git commit
```

Kommentar im Editor eintragen, oder direkt angeben:

```
git commit -m "Mein Kommentar zum Commit"
```

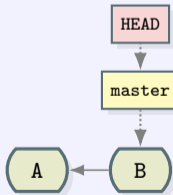
Verzweigungen

Verzweigungen (engl. *branch*) sind nützlich, wenn...

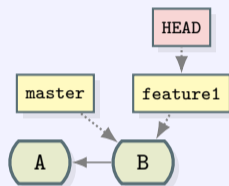
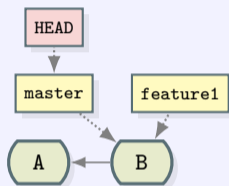
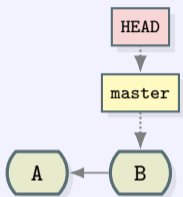
- mehrere Personen gleichzeitig Dateien im Repository bearbeiten,
- man mehrere größere Änderungen parallel bearbeiten möchte,
- oder mehrere Versionen gewartet werden müssen (z.B. Sicherheitsupdates auch für alte Versionen).

Verzweigungen

- Ein Zweig wird durch seinen Namen identifiziert.
- Der Hauptzweig heißt üblicherweise `master`.
- Jeder Zweig zeigt auf einen Commit.
- In der Arbeitskopie ist ein Zweig ausgewählt.
- HEAD zeigt auf den ausgewählten Zweig.



Verzweigungen

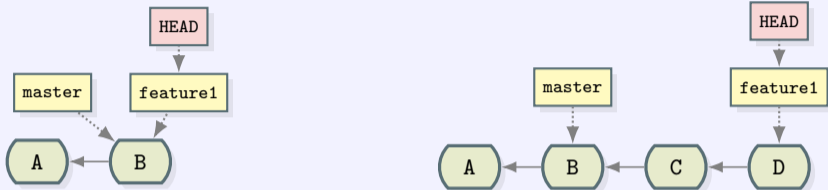


```
git branch feature1
```

```
git checkout feature1
```

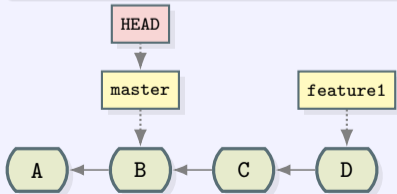
In einem Befehl: `git checkout -b feature1`

Verzweigungen



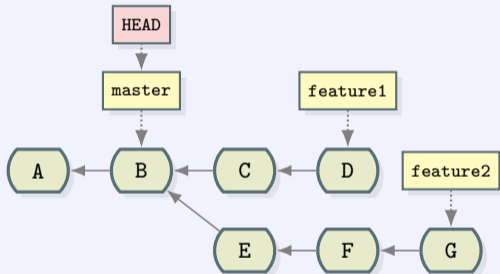
Neue Commits landen in dem Zweig, auf den HEAD zeigt.

```
git checkout master
```



Verzweigung: Merge?

Auf die gleiche Weise erstellen wir einen feature2 Zweig:

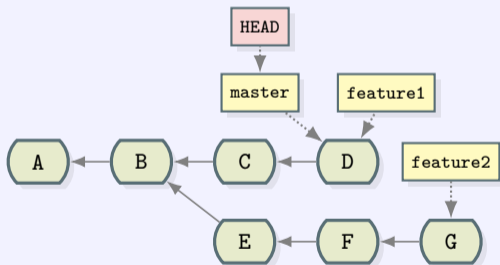


Wie kann master nun diese Änderungen erhalten?

Merge: Fast Forward

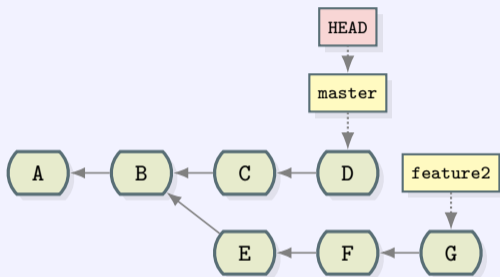
```
git merge feature1
```

Fast Forward: D stammt von B ab.



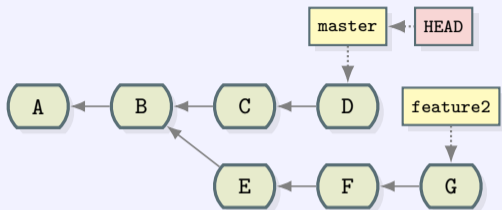
Nicht mehr benötigten Zweig löschen: `git branch -d feature1`

Merge Commit

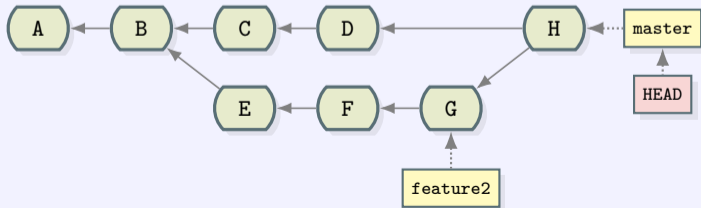


- feature2 soll in master gemerged werden.
- Aber G stammt nicht von D ab!
- Ein Merge Commit ist nötig.

Merge Commit



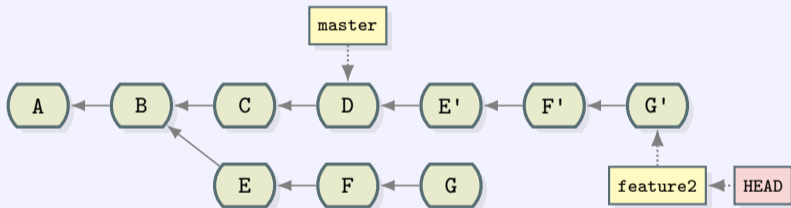
```
git merge feature2
```



Rebase

Falls kein Merge Commit gewünscht ist:

```
git checkout feature2 ; git rebase master
```



E', F' und G' sind neue Commits mit anderen SHA1-Hashes!

Anschließend kann ein Fast Forward Merge durchgeführt werden.

E, F und G sind in keinem Zweig mehr enthalten. Diese Commits werden gelöscht, sobald Git das Repository aufräumt.

Konflikte

- Konflikte sind möglich bei Merge (außer Fast Forward) oder Rebase mit parallelen Änderungen beider Zweige in derselben Datei.
- Konflikte müssen händisch gelöst werden!
- Gute Absprache im Team vermeidet Konflikte (Wer bearbeitet was?).
- Automatische Ergebnisse ohne Konflikte sind nicht immer korrekt!
Beispiel: Zwei neue Funktionen mit demselben Namen an verschiedenen Stellen.


```
git merge feature1
```

```
Auto-merging HelloWorld.java  
CONFLICT (content): Merge conflict in HelloWorld.java  
Automatic merge failed; fix conflicts and then commit the result.
```

```
git status
```

```
On branch master  
You have unmerged paths.  
  (fix conflicts and run "git commit")  
  (use "git merge --abort" to abort the merge)
```

```
Unmerged paths:  
  (use "git add <file>..." to mark resolution)
```

```
    both modified:   HelloWorld.java
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Inhalt der Datei HelloWorld.java:

```
public class HelloWorld
{
    public static void main (String[] args)
    {
<<<<<<< HEAD
        System.out.println("Hallo Welt!");
=====
        System.out.println("Meine erste Testausgabe...");
>>>>>>> feature1
    }
}
```

Konflikt beheben und Konfliktmarkierung löschen. Danach:

```
git add HelloWorld.java
git commit -m "Merge-Konflikt behoben"
```

Konflikte: Mergetool

- Sogenannte *Merge Tools* helfen bei der Behebung von Konflikten.
- Starten auf der Kommandozeile:

```
git mergetool
```

Ein entsprechendes Programm muss installiert sein (z.B. `meld`).

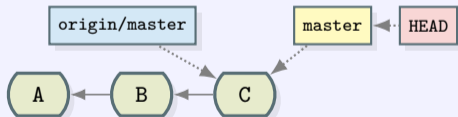
- Unterstützung auch in einigen Editoren, z.B. Visual Studio Code.

Remote

- Repositorys können mit anderen Repositorys verbunden werden:

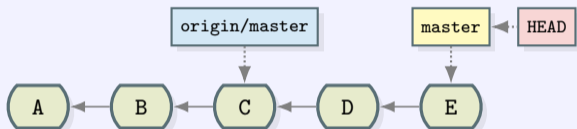
```
git remote add NAME URL
```

- Kopiert man ein Repository mit `git clone`, dann ist in der Kopie automatisch ein remote namens `origin` eingerichtet.

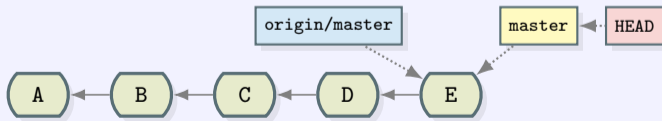


Remote Push

Neue Commits sind vorerst nur im lokalen Zweig enthalten:

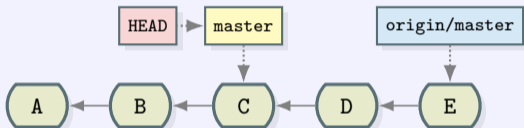


Übertragung der Commits mit `git push`:

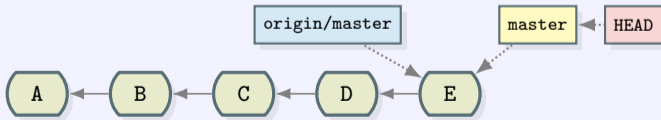


Remote Pull

Der remote Zweig kann neue Commits anderer Personen enthalten:

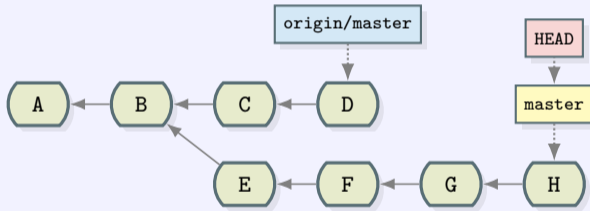


Übertragung der Commits mit `git pull`:



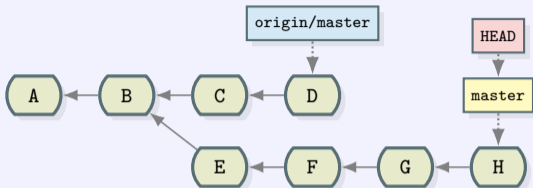
Remote Verzweigung

Verzweigung bei gleichzeitigen lokalen und remote Änderungen:

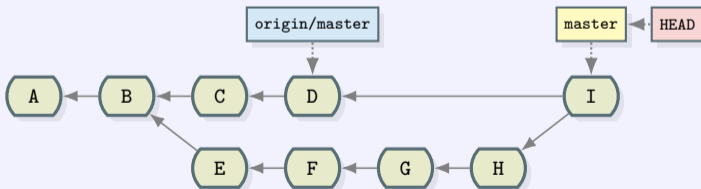


git push schlägt fehl!

Remote Verzweigung



`git pull` erzeugt einen Merge Commit:



Nun kann `git push` ausgeführt werden.

Remote Verzweigung: Rebase statt Merge

- `git pull` lädt zuerst alle neuen Commits herunter und merged dann den remote Zweig in den lokalen Zweig.
- `git fetch` lädt nur die Commits herunter (ohne Merge).
- Danach kann ein Rebase durchgeführt werden:

```
git rebase origin/master
```

GitLab, GitHub und Bitbucket...

...sind Plattformen, die Repositories zentral auf einem Server verwalten.

- Zentrale Kopie des Repositories
- Zugriff über den Webbrowser oder per Git Client
- Verwaltung von Zugriffsberechtigungen: Pull und Push
- Zusätzlich: Plattform zur Planung und Zusammenarbeit
- GitLab ist Open Source - jeder kann es auf seinem Server installieren.

Unser GitLab Server: <https://pl-git.informatik.uni-kl.de/>

URL für Clone in Projektübersicht:

The screenshot shows the GitLab interface for a project named 'pp-2019'. The top navigation bar includes 'GitLab', 'Projects', 'Groups', 'Activity', 'Milestones', 'Snippets', and a search bar. The left sidebar contains a navigation menu with 'Project', 'Details', 'Activity', 'Releases', 'Cycle Analytics', and 'Repository'. The main content area displays the project name 'pp-2019' with 'Project ID: 33'. Below this, it shows 'Add license', '24 Commits', '1 Branch', '0 Tags', and '2.3 MB Files'. The project description is 'Material zum Programmierpraktikum 2019'. A dropdown menu is open, showing cloning options: 'Clone with SSH' with the URL 'git@pl-git.informatik.uni-kl' and 'Clone with HTTPS' with the URL 'https://pl-git.informatik.un'. The current branch is 'master' and the repository path is 'pp-2019 / + v'.

GitLab Projects Groups Activity Milestones Snippets Search or jump to...

pp19 > pp-2019 > Details

P pp-2019 Project ID: 33

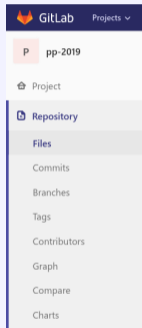
Add license 24 Commits 1 Branch 0 Tags 2.3 MB Files

Material zum Programmierpraktikum 2019

master pp-2019 / + v

Clone with SSH
git@pl-git.informatik.uni-kl

Clone with HTTPS
https://pl-git.informatik.un



Repository Menü:

- Files: Dateien anzeigen
- Commits: Änderungs-Historie
- Branches: Zweige verwalten

Planung: Issue

Planung erfolgt mit Hilfe von *Issues*:

- Eine Aufgabe wird identifiziert (zu behebendes Problem oder ein noch zu entwickelndes Feature).
- Erstelle ein neues Issue:
 - Aussagekräftiger Titel
 - Beschreibung, die die Aufgabe erläutert
 - Das Issue erhält vom System eine Nummer, z.B. #14
- Kommentare zu einem Issue (auch von anderen Personen)
- Nennung der Issue-Nummer im Commit-Kommentar
 - Verlinkung aus Änderungs-Historie zum Issue
 - Commit wird bei den Issue-Kommentaren mit angezeigt
- Nach Erledigung: Issue schließen
 - Issue wird automatisch geschlossen, wenn in `master` ein Commit eingefügt wird, dessen Kommentar `closes #14` enthält.

Zusammenarbeit: Merge Request

Eine Anfrage, Änderungen aus einem Zweig in einen anderen zu mergen.

- Eigene Änderungen in einen separaten Zweig auf den Server pushen.
- Erstelle einen neuen Merge Request (MR):
 - Quelle und Ziel auswählen (Quelle: Neuer Zweig, Ziel: `master`)
 - Titel und Beschreibung
- Kommentare
 - allgemein zum gesamten MR
 - direkt an einer geänderten Code-Zeile
- Neue Commits im Quell-Zweig werden automatisch im MR ergänzt.
- Bei Konflikten: Im lokalen Repository `master` in den Quell-Zweig mergen, dann den Merge Commit pushen.
- Alle zufrieden? → Mergen (Ein Klick in der Weboberfläche)
- Personen ohne Push-Berechtigung in `master` schlagen Änderungen vor, die eine berechtigte Person annehmen kann.

Hinweis: Auf GitHub und Bitbucket heißt es *Pull Request*.

Continuous Integration (CI)

Die Software ständig (bei jedem Push) automatisiert testen.

- Schnelles Feedback, wenn etwas „kaputt“ gegangen ist.
- Wenn Tests fehlschlagen, wird die Person, die den Push durchgeführt hat, per E-Mail benachrichtigt.
- Status und Log der Test-Ausführung sind im Webinterface einsehbar.

Der Test-Status dient nur zur Information, die Commits werden auch bei fehlgeschlagenen Tests ins Repository aufgenommen.

- Definition von Befehlen in der Datei `.gitlab-ci.yml`
- Ausführung in einer isolierten Umgebung auf dem Server

GitLab CI: E-Mail

✖ Your pipeline has failed.

Project Sebastian Schweizer / Testprojekt

Branch `master`

Commit [32bdc58](#)
HelloWorld hinzugefügt

Commit Author  Sebastian Schweizer

Pipeline #1538 triggered by  Sebastian Schweizer
had 1 failed build.

Logs may contain sensitive data. Please consider before forwarding this email.

✖ test [testjob](#)

```
skipping Git submodules setup
$ javac HelloWorld.java
HelloWorld.java:3: error: cannot find symbol
    println("Hello world!");
    ^
  symbol:   method println(String)
  location: class HelloWorld
1 error
ERROR: Job failed: exit code 1
```

Inhalt von `.gitlab-ci.yml`:

```
image: openjdk
testjob:
  script:
  - javac HelloWorld.java
  - java HelloWorld
```

Auf den Namen des Jobs (hier: `testjob`) klicken.

GitLab CI: Im Menü

GitLab Projects Groups Activity Milestones

T Testprojekt

- Project
 - Details
 - Activity
 - Releases
 - Cycle Analytics
- Repository
- Issues 0
- Merge Requests 0
- CI / CD
 - Pipelines
 - Jobs
 - Schedules
 - Charts
- Operations
- Wiki
- Snippets

Sebastian Schweizer > Testprojekt > Jobs

All 1 Pending 0 Running 0 Finished 1 CI lint

Status	Job	Pipeline	Stage	Name	Coverage
failed	#2786 master ↪ 32bcd58	#1538 by [user]	test	testjob	00:00:18 1 minute ago

refresh

Auf den Status des Jobs (hier: failed) klicken.

GitLab CI: Log

```
Sebastian Schweizer > Testprojekt > Jobs > #2786

🔴 failed Job #2786 triggered 2 minutes ago by 🧑 Sebastian Schweizer

Running with gitlab-runner 11.9.0 (692ae235)
  on lampport b03ac405
Using Docker executor with image openjdk ...
Pulling docker image openjdk ...
Using docker image sha256:831a029b6add43a7e2f88608a75b1c924656d9622609fb96898b1f323002ad7 for openjdk ...
Running on runner-b03ac405-project-56-concurrent-0 via lampport...
Initialized empty Git repository in /builds/schweizer/testprojekt/.git/
Fetching changes...
Created fresh repository.
From https://pl-git.informatik.uni-kl.de/schweizer/testprojekt
 * [new branch]      master    -> origin/master
Checking out 32bdcd58 as master...

Skipping Git submodules setup
$ javac HelloWorld.java
HelloWorld.java:3: error: cannot find symbol
    println("Hello World!");
    ^
  symbol:   method println(String)
  location: class HelloWorld
1 error
ERROR: Job failed: exit code 1
```

Best Practices

Was gehört ins Repository:

- Nur von Hand erstellte Dateien: Programmcode, Dokumentation, Konfiguration, ...

Was gehört *nicht* ins Repository:

- Generierte Dateien: .class Dateien, ...
- Heruntergeladene Software-Bibliotheken
(stattdessen: Build-System benutzen → nächste Vorlesung)
- Dateien mit Zugangsdaten oder privaten Schlüsseln
→ lassen sich nicht ohne Weiteres wieder entfernen!
- Benutzerdefinierte Editoreinstellungen: .idea, .vscode, ...
- Absolute Dateipfade: `String appIcon = "C:\Users\HarryHacker\icon.png"`

.gitignore

Eine `.gitignore` Datei definiert, welche Dateien nicht ins Repository aufgenommen werden sollen.

- Verhindert versehentliches Hinzufügen.
- Blendet die Dateien in `git status` aus.
- `.gitignore` selbst sollte ins Repository aufgenommen werden.

Beispiel für eine `.gitignore` Datei:

```
*.class  
*.jar  
out/  
/Test.txt
```

- Alle `.class` und `.jar` Dateien in allen Ordnern
- Alle Ordner mit dem Namen `out`
- Die Datei `Test.txt` im Hauptordner

Zusammenfassung: Git Befehle

- `git init`: Erstellt ein neues, leeres Repository.
- `git clone URL`: Kopiert ein Repository und konfiguriert `remote origin`.
- `git log / git log --stat`: Zeigt die Änderungs-Historie an.
- `git diff version1 version2`: Zeigt den Unterschied zwischen zwei Versionen.
- `git diff --cached`: Zeigt die Änderungen in der Staging Area.
- `git status`: Zeigt den Status aller Dateien an, die nicht Unmodified sind.
- `git add`: Fügt neue Dateien und Änderungen in die Staging Area hinzu.
- `git commit`: Erstellt neuen Commit mit den Änderungen aus der Staging Area.
- `git checkout name-des-zweigs`: Wechselt in einen anderen Zweig.
- `git branch name-des-zweigs`: Erstellt einen neuen Zweig.
- `git branch -d name-des-zweigs`: Löscht einen Zweig.
- `git merge name-des-zweigs`: Merged angegebenen in den aktuellen Zweig.
- `git mergetool`: Startet ein Programm zur Behebung von Konflikten.
- `git rebase name-des-zweigs`: Schreibt die Commits im aktuellen Zweig so um, dass sie von den Commits im angegebenen Zweig abstammen.
- `git push`: Überträgt lokale Änderungen ins remote Repository.
- `git pull`: Überträgt Änderungen aus dem remote Repository ins lokale Repository und erstellt ggf. einen Merge Commit.
- `git fetch`: Lädt Änderungen aus dem remote Repository herunter, ohne sie in einen lokalen Zweig mit aufzunehmen.

Zusammenfassung: Probleme beheben

git push fehlgeschlagen?

- git pull
- ggf. Konflikte beheben, danach erneut git push
- *Niemals* git push -f oder git push --force
(Löscht remote Commits, die nicht Teil des lokalen Zweigs sind!)

Zusammenfassung: GitLab

- Planung in Issues
- Diskussion von Commits in Merge Requests
- GitLab CI Fehler sind fehlgeschlagene Tests
→ Bitte Probleme beheben
Aber die Commits sind trotzdem ins GitLab Repository aufgenommen worden.

Literatur / Vertiefung

- Git Book: <https://git-scm.com/book/en/v2>
- Tutorials von Bitbucket: <https://www.atlassian.com/de/git/tutorials>
- GitLab User Documentation: <https://docs.gitlab.com/ee/user/>
- Praktisches Tool für Kommandozeilen-Liebhaber:
<https://github.com/so-fancy/diff-so-fancy>