FFHS
Fernfachhochschule Schweiz
Mitglied der SUPSI

GIT

Folien zum Modul PyFr

Urs-Martin Künzi

9. März 2024

Inhalt

Versionsverwaltungssysteme

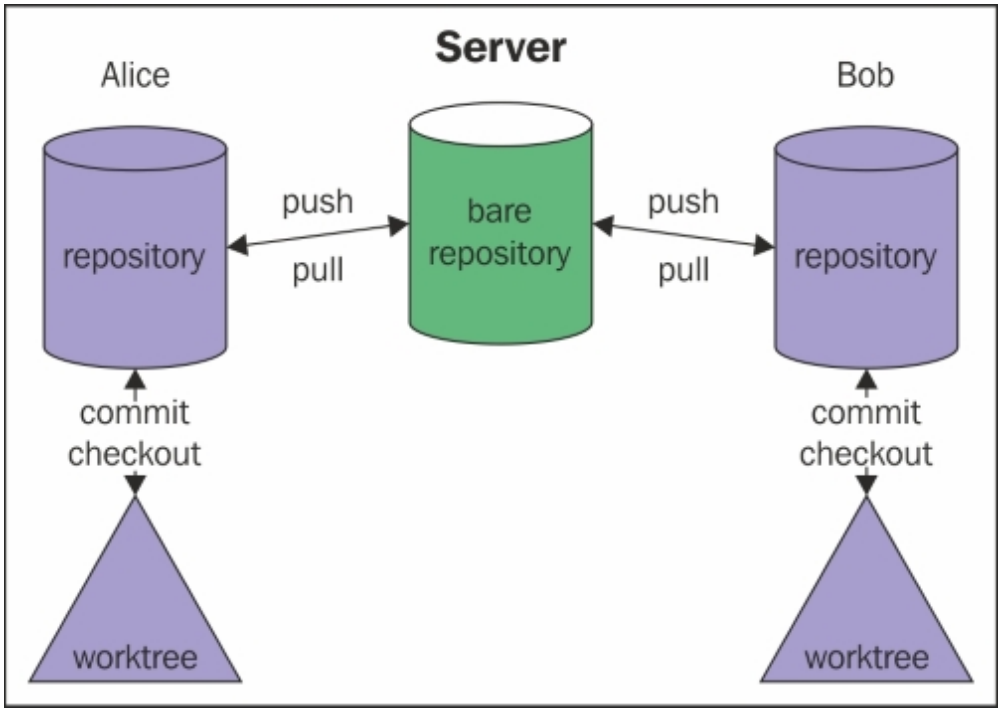
- Versionsverwaltungssysteme erlauben Entwicklung im Team durch Austausch über ein gemeinsames Repository.
- Im Repository ist die Geschichte der Entwicklung abgespeichert; bei Fehlern kann auf alte Versionen zurückgegriffen werden.
- Es ist möglich, verschiedene Zweige eines Projekts zu behandeln.
- Man kann ein Projekt weiter entwickeln, und gleichzeitig eine stabile Version haben.

Resourcen

- Literatur: https://git-scm.com/book/en/v2 (EN)
  https://git-scm.com/book/de/v2 (DE)
- Tutorial: https://www.w3schools.com/git/
- Online-Dokumentation: https://git-scm.com/docs
- Download: https://git-scm.com/downloads

Zentrale und verteilte Versionsverwaltungssysteme

- Zentrale Versionsverwaltungssysteme (CVCS):
  Verwendet ein zentrales Repository für das Projekt.
- Verteilte Versionsverwaltungssysteme (DVCS):
  Alle Benutzenden haben ihr eigenes Repository,
  die Repository werden untereinander abgeglichen.



Verteilte Repositories

- Beispiele:
  - Zentrale Versionsverwaltungssysteme: CVS, Subversion (SVN)
  - Verteilte Versionsverwaltungssysteme: Git, Mercurion
- Vorteile verteilter Systeme:
  - Commit ist offline möglich
  - Bei komplexen Arbeiten ist es möglich, (unfertige) Zwischenzustände lokal zu committen.

Benutzerschnittstellen

- **Primäre Schnittstelle: Kommando-Zeile**
- Integration in Entwicklungsumgebungen
- GUI-Integration ins Betriebssystem
  - Für Windows: TortoiseGit
  - Für OSX: GitFinder
- Angebote für Hosting eines zentralen Repositories bieten Web-Schnittstellen:
  - GitHub
  - Gitolite
  - GitLab
  - Die FFHS unterhält einen GitLab-Server: `https://git.ffhs.ch/`

Git Befehlssyntax

- Allgemeine Befehlssyntax:
  `git` *Kommando* {*Optionen*} {*Parameter*}
- Optionen in der Langform beginnen mit zwei Strichen, in der Kurzform mit einem Strich, z.B.:
  `git help --all` oder `git help -a`

Git Hilfe

- `git help`
  liefert eine Liste der wichtigsten Kommandos.
- `git help -a`
  liefert eine Liste aller Kommandos.
- `git help` *Kommando*
  liefert eine Beschreibung des Kommandos.

- Dokumentation: https://git-scm.com/doc

Konfiguration

- Systemweite Konfigurationsdatei: `$HOME/.gitconfig`
- Repository-lokale Konfigurationsdatei: `.git/config`
- `git config --list`
  Abfragen der Konfiguration
- Initialisierung
  `git config --global user.name` *Name*
  `git config --global user.email` *email-Adresse* `git config --global core.editor` *Editor* (z.B. pico)

Erstellen eines Projekts auf dem Server

- Auf dem Git Server `https://git.ffhs.ch` können Projekte Menu-gesteuert erstellt werden.
- Eine Kopie eines Repositories auf dem Server kann erstellt werden durch
  `git clone` *url*

Erstellen eines lokalen Projekts

- Erzeugung eines neuen lokalen Projekts mit Git:

```
mkdir my_project
cd my_project
git init
```

- Das ist äquivalent zu

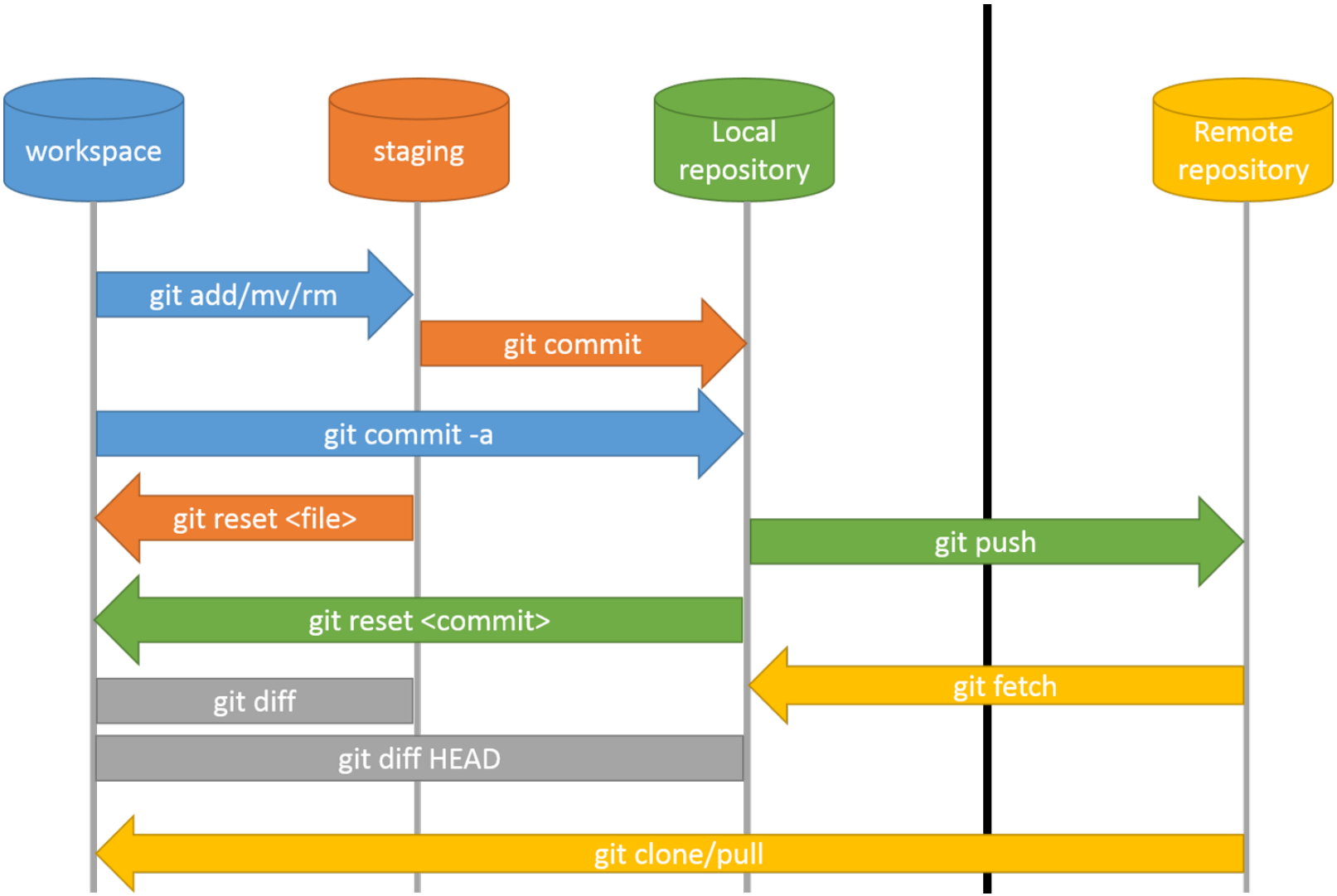
```
git init my_project
cd my_project
```

- Ein lokal erstelltes Projekt kann mit einem Server-Repository verbunden werden durch
  `git remote add origin` *RemoteRepoURL*

Lokale Struktur eines Git Projekts

- In einem Git-Projekt gibt es lokal folgende Bereiche:
  - Working Copy
  - Staging Area
  - Lokales Repository
- Die Staging Area ist ein Index, der Veränderungen enthält, die in der Working Copy erstellt wurden und ins Repository übertragen werden sollen.

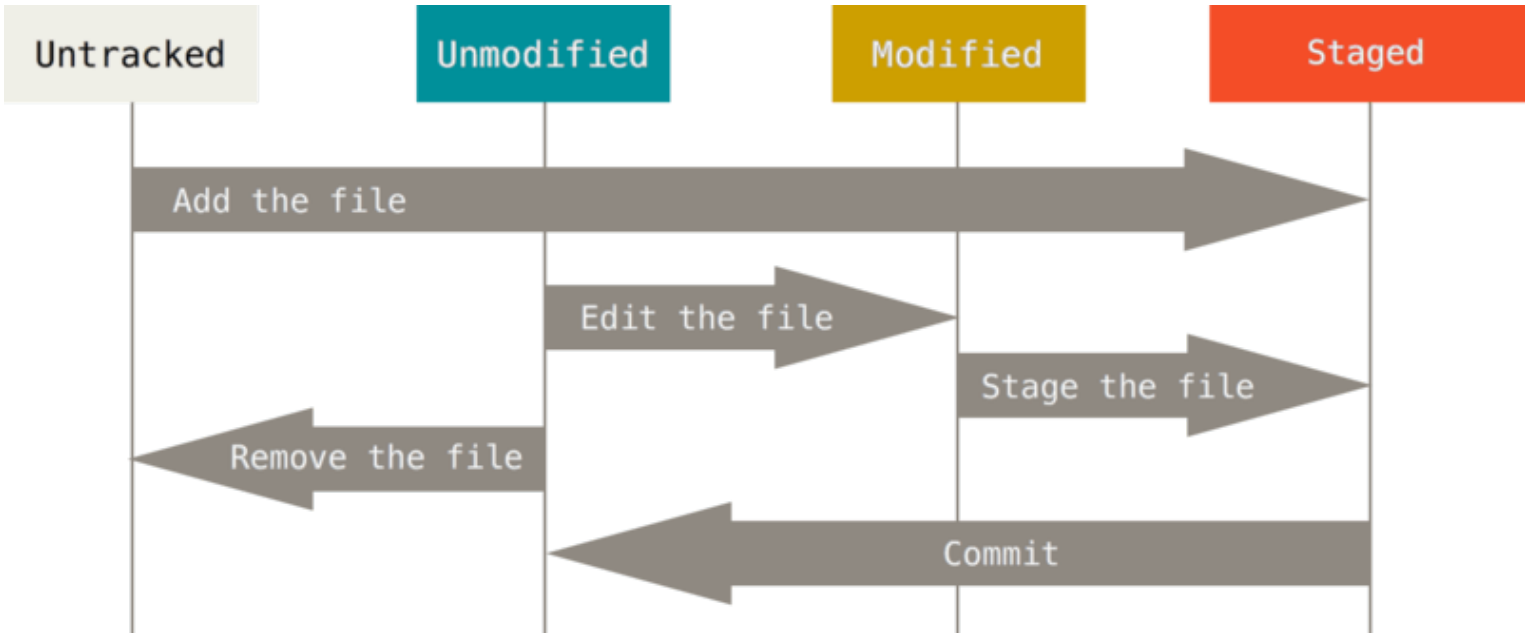
Working Copy ↔ Stage ↔ Local Repository ↔ Remote Repository



Status-Abfrage

- Durch `git status` kann der Zustand der Arbeitskopie abgefragt werden.
- Eine kompaktere Antwort enthält man durch `git status --short` oder `git status -s`

Status eine Datei unter Git



Untracked Die Datei ist zwar im Workspace, aber dem Git-System nicht bekannt.

Unmodified Die Datei der Arbeitskopie ist unverändert gegenüber der entsprechenden Datei im (lokalen) Repository.

Modified Die Datei wurde verändert, aber noch nicht gestaged

Staged Die Datei wurde gestaged

Hinzufügen einer Datei

git add

- Neue Datei erstellen (diese ist dann `untracked`)
- `git add` *neueDatei* (Zustand wird `staged`)
- Mit `git reset HEAD` *zuEntfernendeDatei*
  kann eine Datei aus der Stage Area wieder entfernt werden.

Stagen von Änderungen

git add

- Mit `git add` *dateiName* wird eine veränderte Datei gestaged.
- Eine versehentlich gestagede Datei kann mit
  `git reset HEAD` *Dateiname*
  «ungestaged» werden.

Commits

git commit

- Änderungen der Arbeitskopie werden mit
  `git commit -m "`*Beschreibung des Commits*`"`
  ins Repository übertragen.
- Die Beschreibung ist notwendig. Wird `-m` weggelassen,
  dann erscheint ein Editor für die Beschreibung.
- Es werden nur Änderungen übertragen,
  die durch `git add` gestaged worden sind.
- Durch `git commit -a` oder `git commit --all`
  werden auch Modifikationen von Dateien übertragen,
  die nicht explizit gestaged worden sind.
- Mit `git commit --amend -m "..."`
  wird der letzte Commit verändert
  (bzw. die aktuellen Änderungen werden mit den Änderungen
  des letzten Commits zu einem Commit zusammengefasst).

Verwerfen von Änderungen

- Will man die Änderungen einer Datei im Arbeitsverzeichnis verwerfen und zum letzten Commit zurückkehren,
  so geht das mit
  `git restore` *datei_name*
- Will man alle Änderungen im Arbeitsverzeichnis verwerfen und zum letzten Commit zurückkehren,
  so geht das mit
  `git reset --hard HEAD`
- Um auf den Zustand auf dem entfernten Repositories zurückzusetzen:
  1. Zuerst `git fetch origin`
     (Um das lokale Repo zu aktualisieren.)
  2. Anschließend `git reset --hard HEAD`

Löschen und Verschieben

git rm und git mv

- `git rm DateiName`
  Entfernt Datei
- `git mv DateiName NeuerDateiName`
  Verschiebt Datei
- Die Änderungen müssen committed werden.

Information über das Remote Repository

git remote

- Mit `git remote` kann abgefragt werden, welches das remote Repository ist
- Eine ausführlichere Ausgabe (mit url) erhält man mit `git remote -v` oder `git remote --verbose`

Abgleich mit entferntem Repository

- `git push`
  überträgt Änderungen vom lokalen aufs entfernte Repository.
- `git pull`
  überträgt Änderungen vom entfernten aufs lokale Repository und in die Arbeitskopie.
- `git checkout`
  überträgt Änderungen vom lokalen Repository in die Arbeitskopie

Konflikte

- Falls bei `git push` eine Datei übertragen werden sollte, die im entfernten Repository verändert ist, entsteht ein Konflikt.
  Der Push-Befehl wird nicht ausgeführt.
- Zuerst muss nun `git pull` ausgeführt werden.
- Für eine Datei, die lokal wie auch remote verändert wurde, gilt:
  - Wenn die Datei an verschiedenen Stellen verändert wurde, dann übernimmt git die Veränderungen automatisch.
  - Wenn eine Stelle sowohl lokal wie auch remote verändert wurde, gibt es einen Konflikt.
    Dann wird beides in die Datei geschrieben und die Datei muss zuerst editiert und anschließend gestaged und committed werden.

.gitignore

- In der Datei .gitignore können Muster definiert werden für Dateinahmen, die nicht ins Repository aufgenommen werden sollen.
- Nicht ins Repository aufgenommen werden in der Regel:
  - Generierte Ordner wie
    - `__pycache__` (in einem Python-Projekt)
    - `.jupyter` (in einem Projekt mit Jupyter)
  - Backup-Dateien wie `*.bak`
  - Unter Windows: `Thumbs.db`
  - Unter OSX: `.DS_Store`
- Ein `.gitignore` kann im Home-Directory oder in einem Projekt-Directory stehen.
- Jede Zeile in `.gitignore` steht für ein Muster.

Syntax der Muster in `.gitignore`

(vereinfacht)

- Leerzeilen oder Zeilen, die mit # starten, werden ignoriert.
- ? steht für genau einen Buchstaben (außer Slash).
- * steht für eine beliebige Folge von Buchstaben (außer Slash).
- ** steht für eine beliebige Folge von Buchstaben.
- Ein Muster, das mit einem Slash endet, steht für ein Directory.
- Ein Muster, das mit einem Slash beginnt, steht für einen Pfad relativ zum Projekt-Directory.

.gitignore Beispiel

Beispiel einer .gitignore Datei

```
__pycache__/          # Generierte Dateien von Python
.ipynb_checkpoints # Generierte Dateien von Jupyter
.jupyter              # Konfigurationen für Jupyter
*.bak
*.log
Thumbs.db
.DS_Store/
```

Branches

- Ein Projekt kann verschiedene Branches enthalten.
- Der bei Erstellung vorhandene Branch heißt `main`
  (in früheren Versionen `master`).
- `git branch` zeigt alle vorhandenen lokalen Branches an;
  der aktive Branche wird mit einem Stern markiert.
  (Der aktive Branch wird auch von `git status` angegeben.)
- `git branch -a` zeigt alle inkl remote Branches an.
- `git branch` *NeuerBranchName*
  erstellt einen neuen Branch,
  in der Arbeitskopie bleibt aber der alte.
- `git switch` *BranchName*
  stellt die Arbeitskopie auf den angegebenen Branch um.
- `git switch -c` *NeuerBranchName*
  Erzeugt einen neuen Branch und wechselt auf diesen.

Zusammenführen und Löschen von Branches

- Ein Branch *bra* kann in den aktuellen Branch gemerged werden mit
  `git merge` *bra*
- Ein Branch *bra* kann gelöscht werden durch
  `git branch -d` *bra*
- Beim Mergen von Branches können Konflikte entstehen, die wie bei einem Pull gelöst werden müssen.

Wiederherstellen alter Versionen

git log

- Will man ältere Version wieder herstellen,
  sollte man sich zuerst ein Bild machen über die History:
- `git log`
  zeigt die Geschichte der Commits mit ihrer ID und dem Kommentar.
- `git log --oneline`
  zeigt die Geschichte der Commits in kompakter Form.
- Es ist möglich, die verschiedenen Version auf dem Server im Web-GUI anzusehen.

Wiederherstellen alter Versionen

git diff

- Mit `git diff commit_id` wird gezeigt, wie sich der aktuelle Zustand des Arbeitsverzeichnis vom Zustand des Commits unterscheidet.
- Will man nur die Differenz für eine bestimmte Datei, so geht das mit `git diff commit_id -- datei_name`
- Man braucht nicht die ganze Comic-ID, es genügt ein eindeutiges Anfangsstück.
- Statt die Differenzen mit `git diff` anzusehen, ist es allenfalls einfacher, die Änderungen auf der Server im Browser anzusehen (wenn die aktuelle Version bereits gepushed ist).

Wiederherstellen alter Versionen

`git reset` und `git revert`

- Wenn man zu einem alten Commit zurückkehren will, geht das mit revert oder reset:
  - `git revert` *commit_id*
  - `git reset --hard commit_id`

    (`--hard` bedeutet, dass die Arbeitskopie überschrieben wird)
- Der Unterschied ist, dass bei `git revert` die (fehlerhaften) Versionen im Repository bleiben und weiterhin eingesehen werden können, während bei `git reset` die (fehlerhaften) Commits gelöscht werden.
- Der Unterschied wird deutlich, wenn man nach dem zurücksetzen git log aufruft.
- Bei veröffentlichten (gepusheden) Versionen empfiehlt es sich (insbesondere bei Teamarbeit), `git revert` zu benutzen, während bei lokalen, nicht veröffentlichen Versionen (d.h. nicht gepusheden) Versionen `git reset` verwendet werden kann.

Tags

- Revisionen können getagt werden mit
  `git tag` *TagName*
- `git tag`
  fragt die existierenden Tags ab
- Wie auf getagte Revisionen kann mit checkout und dem Tag-Namen (statt der UID) zugegriffen werden.
- Tags werden bei push nicht mitgepushed. Man benötigt:
  `git push origin` *TagName*.