A Versionsverwaltung - Einführung

A.1 Überblick

A.1.1 Möglichkeiten

Eine Versionsverwaltung (VCS = Version Control System) bietet folgende Möglichkeiten

- Geschichte
  - speichert alle Änderungen in der Versionsgeschichte
  - (einfach) wiederherstellbar
- Dokumentation
  - Wer hat geändert?
  - Was wurde wann geändert?
- Zweige (Branches)
  - verschiedene Entwicklungspfade bzw. -schwerpunkte oder -varianten
  - verschiedene Alternativen ausprobieren
  - Fehler korrigieren
- Zusammenarbeit mehrerer Entwickler
  - Übernahme der Änderungen
  - Konfliktlösung bei mehreren Entwicklern

A.1.2 Arbeitsweise

Wichtig: Eine Versionsverwaltung erzwingt weder einen bestimmten Arbeitsstil des Entwicklers noch ersetzt es die Kommunikation zwischen den Entwicklern untereinander oder mit dem Management.

A.2 Zentral - Dezentral

A.2.1 Zentrale Versionsverwaltung

Versionsverwaltung **mit** Server

- das Repository dh. alle Versionen liegen *am Server*
- die Entwickler
  - holen sich eine Kopie (*checkout*) der aktuellen Version
  - arbeiten mit der lokalen Arbeitskopie
    - eventuell ist die Datei gesperrt / blockiert
  - spielen ihre Änderungen zurück
- (fast) alle Arbeiten mit dem Archiv benötigen Verbindung zum Server

A.2.2 Dezentrale bzw. Verteilte Versionsverwaltung

Versionsverwaltung **ohne** zentralen Server

- die Entwickler
  - haben jeder das ganze Repository
  - arbeiten mit der lokalen Arbeitskopie

  - übernehmen die Änderungen bei Bedarf aus anderen Archiven
  - nur bei Abgleich der Änderungen ist eine Verbindung zu einem Server nötig
  - push / pull via Netzwerk (verschiedene Protokolle)

A.3 Programme

A.3.1 Zentrale Versionsverwaltung

- cvs – der Klassiker
- svn – ersetzt cvs, Open Source, *Enterprise Level*
- Perforce, SourceSafe – kommerzielle Produkte

A.3.2 Dezentral

- Git – de-facto Standard
- Bazaar
- Mercurial
- BitKeeper, ClearCase – kommerzielle Produkte

A.4 Git

A.4.1 Begriffe

- (lokales) Repository – Sorcecode-Archiv mit "Gedächtnis"
- Remote Repository – Online-Version des lokalen Repos auf einem Server (z.B. GitHub, GitLab, selbst gehosteter Server)
- Cloning – Kopie eines Repositories in einem neuen Verzeichnis anlegen
- Commit – Änderungen in das Archiv übernehmen. Speichert aktuellen Stand des Filesystems ("Snapshot") und erzeugt zu diesem Zeitpunkt eine Version (Projektzustand). Aufeinanderfolgende Commits werden in Commit-History gespeichert
- working directory – das Arbeitsverzeichnis, beinhaltet checkout (lokale Version) der Daten
- staging-area = index – sammelt die zu commitenden Änderungen
- Commit-ID = Hash – eindeutiger SHA1-Hash-Wert eines Commits
- Tag – symbolischer Name für ein Commit
- Branch – Entwicklungszweig (unterschiedlicher Commits): Kopie des Projekts in einer isolierten Umgebung (damit das Main-Projekt nicht verändert wird)
- Merge – Zusammenführen von zwei Branches
- HEAD – symbolischer Name der Commit-ID des letzten Commits. HEAD~5 bezeichnet das 5.-letzte Commit

A.4.2 Git Übersicht

- Project page: http://git-scm.com/
- ursprünglich von Linus Torvalds für den Linux Kernel entwickelt
- verteiltes Versionsverwaltungssystem
- Datentransfer zwischen Repositories: viele verschiedene Protokolle
  - Repository kann Verzeichnis am Fileserver sein – oder auch auf einem USB Stick

- eigenes Protokoll via TCP
- SSH
- HTTP, HTTPS, FTP
- rsync
- Patches (Änderungen) via E-Mail

A.4.3 Vorteile Git

- hohe Effizienz
- gute Skalierbarkeit
  - Linux kernel: 6.9 - 6.10 (2 Monate, Juli 2024)[1]
    - 13,312 commits / 1,918 developers
    - 12392 files changed / 486755 insertions / 251592 deletions
- einfacher Datentransfer zwischen Repositories
- Verzeichnisbasierte Verwaltung
- Branches sind einfach zu erstellen (Leichtgewichtige lokale Branches)
- Kryptographische Sicherheit der Projektgeschichte – nachträgliche Geschichtsfälschung unmöglich

A.4.4 Prinzip

- viele VCS speichern die Änderungen
  - reden aber von den Versionen
- Git speichert alle Versionen
  - redet aber meist von Änderungen (Patches)
  - sehr kompakt durch *zippen*
- Details
  - fälschungssichere Versionshistorie ähnlich wie bei Bitcoin (Blockchain)
  - git speichert `content addressed`
    - im Repository wird der Hash des Dateiinhalts als Adresse (Dateiname) verwendet.
    - jeder Commit enthält eine Verweis auf den `parent`-Commit (see `git cat-file -t hash` oder `git cat-file -p hash`) [https://aboullaite.me/deep-dive-into-git/][https://github.blog/2022-08-29-gits-database-internals-i-packed-object-store/]

B Installation

B.1 Software

siehe http://www.vogella.com/tutorials/Git/article.html und http://git-scm.com/book und http://marklodato.github.io/visual-git-guide/index-en.html

[1] https://kernelnewbies.org/DevelopmentStatistics

B.2 Erste Konfig

Benutzername und Email ändern ↓↓

```
git config --global user.name "Your Name"
git config --global user.email "your.name@htl.rennweg.at"

git config --global push.default simple
```

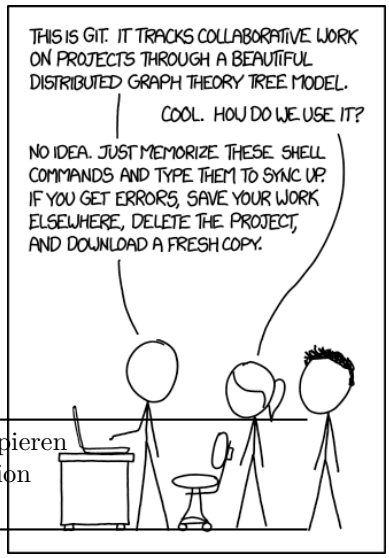
Manchmal *poppt* ein Editor auf, zB. bei einem Merge. Standard ist `vi`, aber man kann sich seinen Lieblingseditor wünschen:

```
git config --global core.editor "nano"
```

B.3 Auf anderen Rechner übertragen/sichern

Backup dieser Dateien:

`$HOME/.ssh`	Keys, einfach den ganzen Ordner kopieren
`$HOME/.gitconfig`	kopieren, enthält globale Konfiguration
`$HOME/conf/git`	Konfiguration unter Mac OS?



https://xkcd.com/1597/

If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as...'; and eventually you'll learn the commands that will fix everything.

C Lokale Repositories erstellen

Mehrere Möglichkeiten:

- Existierendes Repository von einem Remote-Host kopieren (GitHub, GitLab, etc.)

  ```
  $ git clone <remote_repo_url>
  ```

- einen bestimmten Branch aus Remote-Repo holen

  ```
  $ git clone -branch <branch_name> <remote_repo_url>
  ```

- In einem lokalen Verzeichnis ein neues Repository erstellen

  ```
  $ git init
  ```

D Vorbereitung auf GitHub

D.1 SSH-Keys für den Server erstellen

Starte auf einer Linux (oder WSL) -Kommandozeile

```
ssh-keygen
```

Alle Fragen mit Return beantworten, eventuell eine Passphrase vergeben. Bitte nicht Putty verwenden, diese Keys haben ein anderes Format.

In `$HOME` gibt es dann einen Ordner `.ssh/` mit `id_rsa` und `id_rsa.pub` – **nicht umbenennen**!

Für den Zugriff auf die Repositories auf den Github/Gitlab/Bitbucket,... -Server: Inhalt der Datei `id_rsa.pub` als SSH-Key hochladen (siehe unten)

D.2 Login einrichten

- Erstelle auf Github einen User `9999_25_HTL3R` (verwende statt 9999 die 4-stellige Eduvidual-Kennung)
- Auf GitHub anmelden und SSH-Key importieren:
  - Icon klicken / Settings (Zahnrad) / SSH and GPG keys/ New SSH-Key
  - Inhalt der Datei `id_rsa.pub` einfügen

E Neues Repository anlegen

E.1 GitHub

- In GitHub das Dashboard öffnen (klick auf Katzen-Icon)
- Unsere Repo-Namen beginnt immer mit `9999_25_HTL3R`
- Typ = Private (daher dem Lehrer Zugriffsrechte geben)

E.2 Lokales Repo anlegen und mit GitHub verknüpfen

Im IntelliJ/PyCharm Projekt / src eine Kommandozeile aufmachen [2]

```
git init .
```

Jetzt sollte man auch gleich eine Datei `.gitignore` anlegen. Offiziell unterstützten Alternative: direkt in JetBrains-IDE (IntelliJ / PyCharm / …): New Project –> *Greate GIT-Repository* anhaken

Abschließend verknüpfen wir das Repository mit GitHub:

```
git remote add origin https://github.com/9999_25_HTL3R/RepoNameAufGithub.git
```

F Datei in lokale Repository eintragen

Editieren und commiten. Die Commit-Message *muss* mit der Übungsnummer beginnen und danach einen *sinnvollen* beschreibenden Text beinhalten.

```
echo test > myfile
git status
git add myfile
git status
git commit -m "UE00: Initial commit"
git log
git log --oneline
```

Achtung: * weder `git add *` noch `git commit -a ...` sind gute Ideen (werden aber trotzdem häufig verwendet). Schon gar nicht ohne passender `.gitignore` Datei. * sehr gute Ideen sind `git status` und `git log` bzw `git log --oneline`! * *Alias* für Tippfaule: `git config --global alias.lol 'log --oneline --graph --all'`. Jetzt funktioniert `git lol`!

Alternative: * IDE oder * `git gui`

F.1 Repository auf GitHub pushen

Um eine Übung auf GitHub abzugeben, muss man sie pushen:

```
git push -u origin master
```

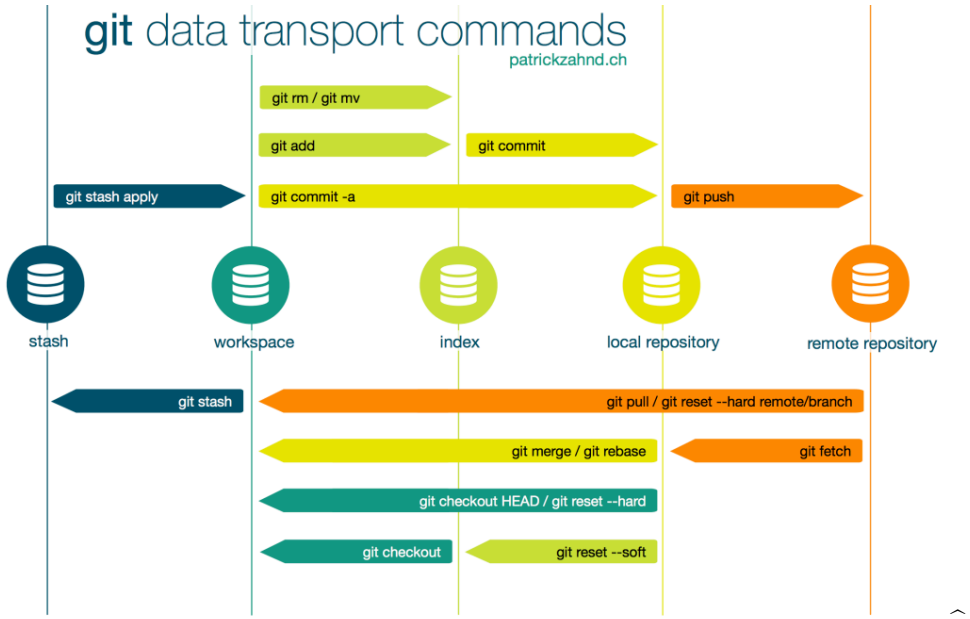
[2] das ist nicht die offiziell empfohlene Variante, aber es geht trotzdem.

G Repository löschen:

- Lokales Repository
  - einfach den Ordner .git löschen.
- Auf GitHub:
  - On GitHub.com, navigate to the main page of the repository.
  - Under your repository name, click Settings. …
  - On the "General" settings page (which is selected by default), scroll down to the "Danger Zone" section and click Delete this repository.
  - Click I want to delete this repository.

H FAQ

H.1 Wann braucht man welches Kommando?



H.2 Änderung an einer Datei vor dem Commit zurücknehmen

Wenn man eine Datei durch editieren zerstört hat und wieder die letzte Version im Repository haben will:

```
git checkout dateiname
```

H.3 Aufräumen

- Eine Datei (die schon einmal *commited* wurde) löschen. Achtung: Datei wird auch *lokal* gelöscht.

  ```
  git rm filename
  git commit ...
  ```

- Datei nur aus Repository löschen (lokale Datei bleibt)

  ```
  git rm  --cached filename
  ```

- Aufräumen

  Alle Dateien löschen die nicht im Repository sind – sehr gefährlich, immer mit `-i` oder `-n` vorher probieren. Mit `-f` wird wirklich gelöscht!

  ```
  git clean -n
  ```

H.4 Rückgängig

- Rückgängig machen – lokale Änderungen

  ```
  git checkout -- filename
  ```

- alte Version holen

  ```
  git checkout versionshash  filename
  ```

- einen Commit rückgängig machen

  ```
  git revert versionshash
  ```

H.5 Merge Probleme

Wenn man im Team arbeitet kommt es bei *gleichzeitigen* Änderungen zu `merge`-commits.

Nachteil:

- *unschön*
- unübersichtlich

Abhilfe

```
git config --global pull.rebase true
git config --global rebase.autoStash true
```

dann schlichtet `git` die commits um.

Wenn es beim pull nicht geht braucht es nach dem erfolglosen `git pull`

```
git rebase  origin/master
```

dann sieht man die merge-Probleme besser (falls es git nicht sowieso schafft)

H.6 Einzelne Dateien nicht mit git verwalten

in der Datei `.gitignore` kann man Muster für Dateinamen angeben – diese werden dann von `git` ignoriert.

Muster:

- https://github.com/github/gitignore
- https://www.gitignore.io/ bzw. https://www.gitignore.io/api/java,intellij
- IntelliJ: Plugin `.ignore`

H.7 *falsche* Dateien entfernen

```
git rm -r --cached .
git add .
```

bzw.

```
git rm --cached `git ls-files -i --exclude-from=.gitignore`
```

Nach einer Überprüfung: *committen*:

```
git commit -am "Remove ignored files"
```

I Jetzt gehts endlich los: Git – Übung

I.1 Installation und Konfiguration

(1) Installiere (falls noch nicht erfolgt) git und GitViz
(2) Erstelle einen Ordner für das Repository: `9999-25-HTL3R-sem1` Ordner: `00_git`

I.2 Dokumentation

Erstelle ein Protokoll und dokumentiere deine Erfolge (GitViz + Output von `git status`) als Screenshots

Erstelle die Datei (ein Office-Dokument) `Familienname_Vorname.doc` und speichere dort deine Erfolge als Screenshots.

Ab sofort machen wir **immer (egal ob Übungstunde, PLF, ITP-Projekt, Diplomarbeit, ...) mindestens ein Commit mit *sinnvollem* Namen nach jeder (Teil)Aufgabe und am Ende der Stunde/PLF** (sonst gibt es Punkteabzug)!

I.3 Git-Übungen: Git-Katas

https://github.com/praqma-training/git-katas

Mache folgende Übungen:

- Basic Commits
- Basic Staging
- Basic Branching
- Fast-forward Merge

I.4 Git Branching

http://learngitbranching.js.org/

Machen folgende Levels:

- *main* – Zeilen 1 und 2
- *remote* – Zeile 1, Aufgaben 1 bis 6

Tipp: Mit dem Befehl `levels` kommt man ins Auswahlmenü zurück.

I.5 Bonus

I.5.1 Git Hug – TODO

https://github.com/Gazler/githug

I.5.2 Git Hardcore

http://gitexercises.fracz.com/

zumindest ein paar sollte man versuchen – die letzten sind echt fies

J Anhang

J.1 Manuelles Entfernen von Files aus Vorgänger-commits in Git[3]

Sollte man unabsichtlich z.B. `.class` Files committed haben, welche vom remote als **forbidden** gelten, kann man diese recht einfach mit wenigen Kommandos wieder entfernen.

J.1.1 Überprüfen der commits

Als erstes müssen wir herausfinden in welchem commit die `.class` Files committed worden sind. Dazu verwenden wir das Kommando:

```
gitk
```

[3] gespendet vom Max Burger

Danach sollte ein Fenster mit allen commits auftauchen.

Mit einem Linksklick auf einen commit kann man ablesen, welche Files jeweils committed worden sind. Hat man nun den fehlerhaften commit gefunden, so muss man sich den Commit Hash des Vorgänger commits kopieren.

J.1.2 Rebase

Mit dem Kommando:

```
git rebase -i <Hash des Vorgängercommits>
```

öffnet sich nun ein VI Editor. Ganz oben sind die Nachfolgercommits, mit jeweils einem Prefix **pick**, eingetragen. Man ersetzt jetzt das Wort **pick** mit dem Wort **edit** (Mit I schaltet man in den Insert mode und mit Escape verlässt man diesen wieder) Schließen erfolgt mit **:wq**

Wichtig: Ausgabe des Befehls lesen, `rebase` ist recht komplex und **muss** immer abgeschlossen oder abgebrochen werden – ansonsten ist das Repo in einem sehr dubiosen Zustand und nicht mehr verwendbar.

J.1.3 Entfernen der Class files mit Hilfe des Git GUIs

Nach dem rebase müssen wir die .class Files herausnehmen. Mit dem

```
git gui
```

ist dies am einfachsten. In dem GUI wählt man nun die Checkbox `'Amend Last Commit'` aus. Jetzt erscheinen auf der linken Seite alle Files welche `committed` worden sind. Mit einem Linksklick auf das Fileicon kann man das File als unstaged markieren. Wenn man alle `.class` Files entfernt hat kann man das Fenster wieder schließen.

J.1.4 Rebase fertigstellen

Mit

```
git rebase --continue
```

schließen wir unseren **rebase** ab, und können nun fehlerfrei auf den remote pushen. Jetzt sollte ein .gitignore File angelegt werden um erneute .class File pushes zu vermeiden.

J.2 Wo ist die GUI?

Mitgeliefert wird:

```
git gui
gitk
git instaweb
```

siehe http://git-scm.com/downloads/guis

Achtung: auf spezielle Anbieter getrimmte Programme (Sourcetree für Gihub) sind nur bedingt brauchbar.

J.3 Eclipse mit EGit

- http://wiki.eclipse.org/EGit/User_Guide/Getting_Started
- http://wiki.eclipse.org/EGit/User_Guide/Remote#Push_Upstream

J.4 Versionen und Archiv erstellen

Git kann

- bestimmte Versionen *benennen* (`Tag`)
- bestimmte Versionen in ein Archiv einpacken (*zippen*)

```
NAME="NameDesProjekts"
DATE=$(date +%Y%m%d)
git tag -fa $NAME-$DATE -m "created $NAME-$DATE.zip"
git archive HEAD --prefix $NAME-$DATE/ -o $NAME-$DATE.zip
#unzip -lv $NAME-$DATE.zip
```

J.5 Erweiterungen

- https://github.com/stevemao/awesome-git-addons
- https://github.com/tj/git-extras bzw. https://vimeo.com/45506445

J.6 Git WorkFlow

- https://www.atlassian.com/git/tutorials/comparing-workflows/
- http://nvie.com/posts/a-successful-git-branching-model/

J.7 Git Graph

Für viele *Dinge* kann man sich das Repo als Graph vorstellen.

Gute Einführung: https://juristr.com/blog/2013/04/git-explained/

J.8 Infos zum Nachlesen

- https://git-scm.com/book/en/v2/
- https://www.git-tower.com/learn/
- http://www.wei-wang.com/ExplainGitWithD3
- oder Nachschauen: https://www.youtube.com/watch?v=1ffBJ4sVUb4
- http://gitready.com/
- https://blog.isquaredsoftware.com/presentations/2019-03-git-internals-rewrite/#/
- https://quickref.me/git

noch mehr Info:

- Git Tutorium von Sujeevan Vijayakumaran aus dem Archiv des ehemaligen freien Magazins 2014/15
  - http://www.freiesmagazin.de/freiesMagazin-2014-12.html Start
  - http://www.freiesmagazin.de/freiesMagazin-2015-01.html Branch
  - http://www.freiesmagazin.de/freiesMagazin-2015-02.html Rebase, Remote
  - http://www.freiesmagazin.de/freiesMagazin-2015-06.html Git-Hub
- https://wiki.fsfw-dresden.de/doku.php/doku/wissenschaftliches_schreiben_mit_git_und_latex
- zum Üben: https://github.com/praqma-training/git-katas
- Buch in Deutsch: http://gitbu.ch/
- Tipps
  - https://dangitgit.com/
  -

Weitere interaktice Kurse

- https://github.com/praqma-training/git-katas
- https://dev.to/ifrah/5-underrated-resources-to-learn-git-and-github-4edi