

main_git_leicht_gemacht

June 26, 2024

1 Git Leicht Gemacht

Kurs Methoden der Linguistik (050004-SoSe 2024)

Referent: Christopher Chandler

Dieses Jupyter-Notebook “Git leicht gemacht” bietet eine Einfuehrung in Git, ein Versionskontrollsystem. Es erklart die grundlegenden Konzepte wie Repository und Branch und beschreibt wichtige Befehle wie `git init`, `git clone`, `git add`, `git commit`, `git push` und `git pull`.

Es zeigt, wie man Branches erstellt, wechselt und zusammenfuehrt (Merging). Ausserdem werden grafische Benutzeroberflaechen wie Source Tree, Git in Pycharm und Github vorgestellt, die die Verwaltung von Repositories erleichtern.

Praktische Beispiele und Beispielcode illustrieren die Anwendung der Git-Befehle.

Inhaltsverzeichnis

Teil 1: Einfuehrung

1.1 Was ist Versionsverwaltung?

1.1.1 Lokale Versionsverwaltung

1.1.2 Zentrale Versionsverwaltung

1.2 Kurzer Ueberblick ueber die Historie von Git

1.3 Was ist Git?

1.3.1 Snapshots statt Unterschiede

1.3.2 Fast jede Funktion funktioniert lokal

1.3.3 Git stellt Integritaet sicher

1.3.4 Git fuegt im Regelfall nur Daten hinzu

1.3.5 Die drei Zustaende

1.4 Die Kommandozeile

1.5 Git installieren

1.5.1 Mac

1.5.2 Linux

1.5.3 Windows

1.5.4 Installation ueberpruefen

1.5.5 Mac Version

1.5.6 Linux Version

1.5.7 Windows Version

1.6 Hilfe finden

Teil 2: Git-Grundlagen

2.1 Ein Git-Repository anlegen

2.1.1 Lokal

2.1.2 Remote

2.1.3 Kollaboration

2.2 Aenderungen nach

2.2.1 Zustand

2.2.1 Zustand von Dateien pruefen

2.2.2 Neue Dateien z

2.2.3 Kompakter Status

2.2.4 Geaenderte Da

2.2.5 Ignorieren von Dateien

2.2.6 Ueberpruefer

2.2.7 Die Aenderungen commiten

2.3 Anzeigen der Commit-Historie

2.4 Ungewollte Aenderungen rue

2.4.1 Eine Datei aus der S

2.4.2 Aenderung

2.6 Taggen

2.6.1 Annotierte Tags

2.6.2 Tags loeschen

2.6.3 Tags auschecken

Teil 3: - Git Branching

3.1 Branches auf einen Blick

3.2 Erzeugen eines neuen Branches

3.3 Wechseln des Branches

3.4 Merging

3.4.1 Einfaches Merging

3.4.2 Testing Branch

3.5 Merge-Fehler

Teil 4: Git GUI Software

4.1 Source Tree

4.2 GIT in PyCharm

4.3 Github

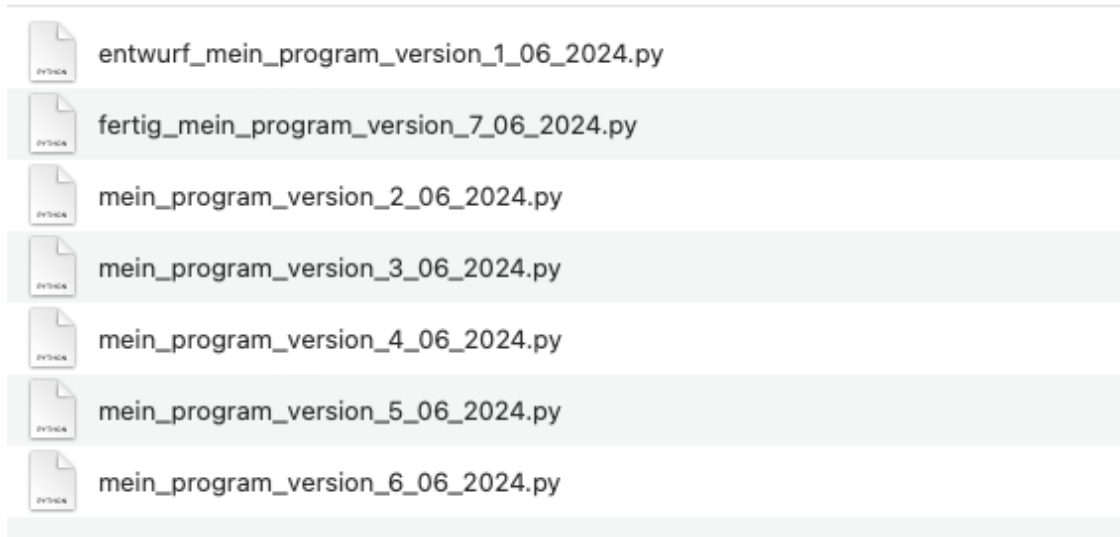
Quellen

2 1. Einfuehrung

Es ist selten der Fall, dass man ein Programm oder ein Stueck Code erstellt und letztendlich vollkommen damit zufrieden ist. Noch seltener ist es, dass man ein Programm entwickelt und nie wieder etwas daran aendert.

Damit das Program nicht verloren geht oder man irgendwas aus Versehen ueberschreibt, erstellt man gerne verschiedene Versionen davon. Oft wird eine Mischung aus Datum und Ziffern benutzt, um festzulegen, was die aktuellste Version ist.

Deswegen hast du wahrscheinlich schon sowas mal gemacht:



Das ist noch relativ harmlos, aber so ein Vorgehen kann wegen der eigenen Kreativitaet sehr schnell unuebersichtlich werden und dann sieht es am Ende so aus.

Who would win

Most advanced version control system used to keep track of changes in any set of files. Aimed at speed, data integrity, and support for distributed, non-linear workflows.



Me

Development > Creations > Android > My Color Manager All Saves >				
Name	Date modified	Type	Size	
ColorManager	13-03-2017 12:47	File folder		
ColorManager 9-5-2016	13-03-2017 12:47	File folder		
ColorManager 11-5-2016	13-03-2017 12:48	File folder		
ColorManager save 1	13-03-2017 12:48	File folder		
MyColorManager 03-06-16	13-03-2017 12:48	File folder		
MyColorManager 4-7-2016	13-03-2017 12:49	File folder		
MyColorManager 4-7-2016 2nd	13-03-2017 12:49	File folder		
MyColorManager 5-6-16	13-03-2017 12:50	File folder		
MyColorManager 5-8-16	13-03-2017 12:51	File folder		
MyColorManager 6-7-16	13-03-2017 12:51	File folder		
MyColorManager 10-6-2016	13-03-2017 12:52	File folder		
MyColorManager 10-7-2016	13-03-2017 12:52	File folder		
MyColorManager 10-8-2016	13-03-2017 12:53	File folder		
MyColorManager 13-7-2016	13-03-2017 12:54	File folder		
MyColorManager 15-5-16	13-03-2017 12:54	File folder		
MyColorManager 17-5-16	13-03-2017 12:55	File folder		
MyColorManager 18-7-16	13-03-2017 12:56	File folder		
MyColorManager 19-05-16	13-03-2017 12:56	File folder		
MyColorManager 20-5-16 13-24	13-03-2017 12:57	File folder		
MyColorManager 20-8-2016	13-03-2017 12:58	File folder		
MyColorManager 21-7-2016	13-03-2017 12:58	File folder		
MyColorManager 23-5-16 2	13-03-2017 12:59	File folder		
MyColorManager 23-7-2016	13-03-2017 12:59	File folder		
MyColorManager 24-5-16	13-03-2017 13:00	File folder		
MyColorManager 25-8-2016	13-03-2017 13:01	File folder		
MyColorManager 25-9-2016	13-03-2017 13:02	File folder		
MyColorManager 26-5-16	13-03-2017 13:02	File folder		
MyColorManager 26-7-2016	13-03-2017 13:03	File folder		
MyColorManager 27-7-2016	13-03-2017 13:04	File folder		
MyColorManager 27-09-2016	13-03-2017 13:05	File folder		
MyColorManager 30-7-2016	17-09-2017 01:06	File folder		
MyColorManager 2016-9-21	13-03-2017 13:06	File folder		

oder so



Name

- v1.0
- v2.0
- v2.1
- v2.2

- project
- project-revised
- project-final
- project-final-for-real

Name

- project
- projectt
- projectttttttt
- aaaaaaaaaaaaaaaaaaaaaaaaaa...



Bei sowas verliert man sehr schnell den Ueberblick und man kann hier nicht wirklich von einer effektiven Versionierung sprechen.

Damit die Aenderungen an einer Datei gespeichert und in der Zukunft nachvollzogen werden koennen, sollte man eine Versionsverwaltungssoftware wie GIT benutzen:

```
commit 79baada5e16080cc7e9289319c6d2f4289bb94d2 (HEAD -> main)
Author: Christopher chandler <christopher.chandler@outlook.de>
Date:   Fri Jun 7 22:31:39 2024 +0200

    update text

hello_world_01.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)

commit bd431cfd35bbef818f71140b62a459fe76fefb16
Author: Christopher chandler <christopher.chandler@outlook.de>
Date:   Fri Jun 7 22:31:04 2024 +0200

    add text

hello_world_01.txt | 1 +
1 file changed, 1 insertion(+)

commit 28348fbe71c5d100cd2b0554cb6d46dd7ca5e281
Author: Christopher chandler <christopher.chandler@outlook.de>
Date:   Fri Jun 7 18:34:30 2024 +0200

    hello_world angepasst - anderer Name

hello_world_01.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
~
christopherchandler@Mac-Studio test_git_repo %
```

Das Bild sieht jetzt etwas kryptisch aus, aber hoffentlich wird das am Ende dieses Notebooks klarer sein.

2.1 1.1 Was ist Versionsverwaltung?

Was ist „Versionsverwaltung“, und warum sollte man sich dafuer interessieren?

Normalweise wenn man eine Datei speichert, wird der letzte Stand der Datei ueberschrieben. Das bedeutet, dass man nicht mehr auf aeltere Versionen davon nicht mehr zurueckgreifen kann. Noch schlimmer ist es, wenn man ausversehen mehrere Dateien in einem Verzeichnis loescht. Doch mit einer Versionsverwaltungssoftware kann man solche Aenderungen und Loeschungen rueckgaengig machen.

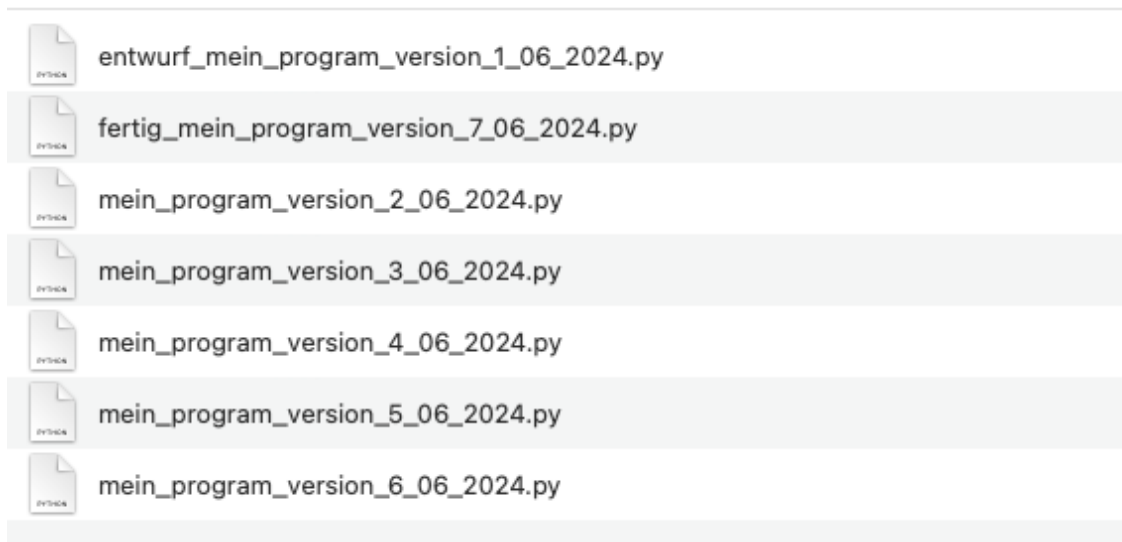
Vereinfacht gesagt, eine Versionsverwaltung ist ein System, welches die Änderungen an einer oder einer Reihe von Dateien über die Zeit hinweg protokolliert, sodass man später auf eine bestimmte Version zurückgreifen kann.

Versionsverwaltung kann dann wiederum hauptsächlich in zwei Kategorien unterteilt werden: *Lokal* und *Zentral*.

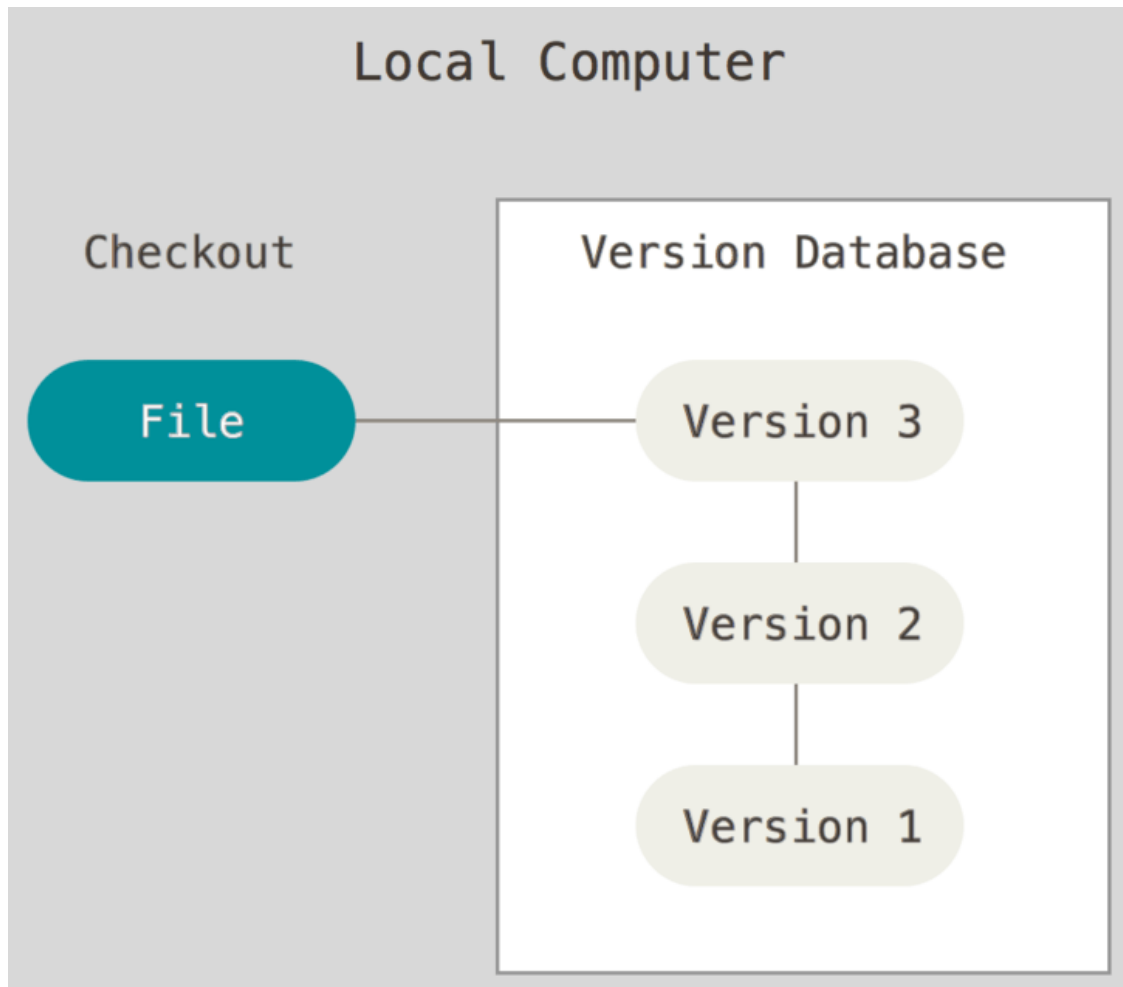
(Es gibt verteilte Versionsverwaltung, aber das ist für diese Präsentation nicht weiter relevant)

2.1.1 1.1.1 Lokale Versionsverwaltung

Wir nehmen uns das Bild von dem letzten Kapitel als Beispiel. Viele speichern dasselbe Programm mehrfach ab. Das Problem bei sowas ist, dass es sehr schnell unübersichtlich wird und evtl. verliert man den Überblick, wie vorhin erwähnt. Es kann auch natürlich vorkommen, dass man aus Versehen in der falschen Datei arbeitet.

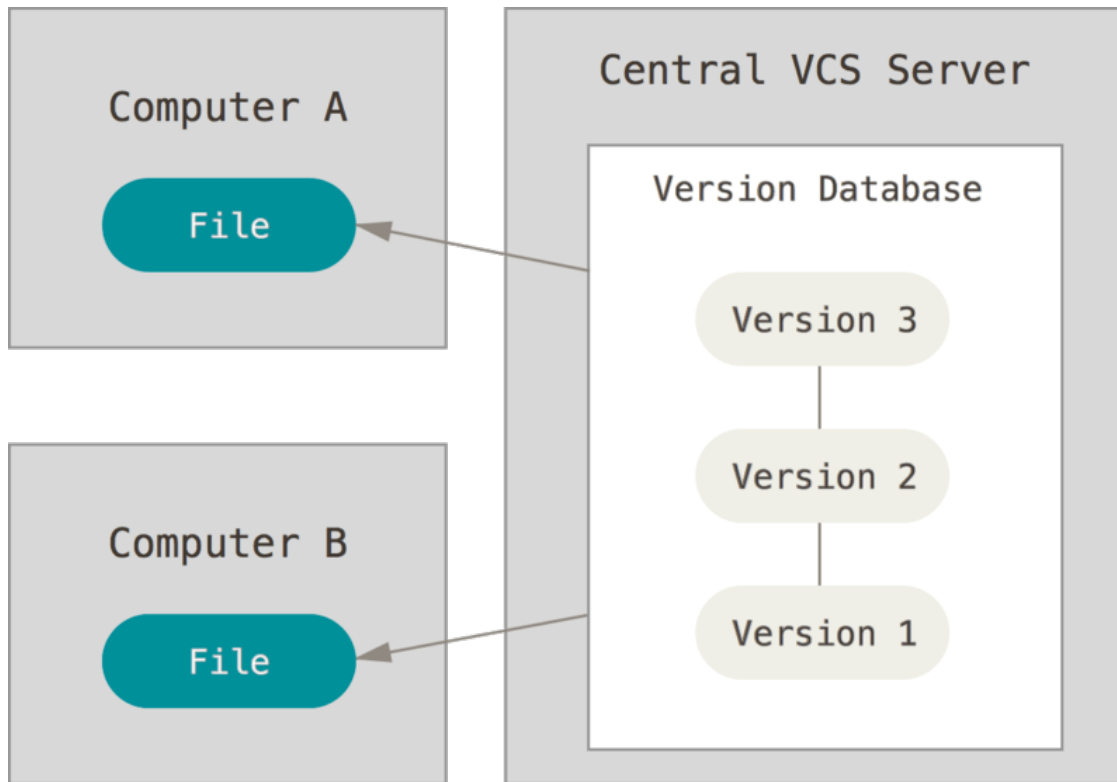


Um dies zu verhindern, benutzt man Git diese Änderungen zu tracken bzw. zu verfolgen. Die Datei wird zwar überschrieben, aber jede Änderung, die an dieser Datei vollzogen wurde, ist sichtbar.



Lokale Versionsverwaltung

2.1.2 1.1.2 Zentrale Versionsverwaltung



Bei der *zentralen Versionsverwaltung* geht es darum, die Zusammenarbeit mit anderen Programmierern zu erleichtern. Statt Dateien hin- und her zu schicken, wird ein zentrales Verzeichnis bzw. ein sogenanntes *Repository* (auch manchmal Repo) angelegt. Von dort aus wird der aktuellste Stand des Programms heruntergeladen oder .ggf dahin hochgeladen.

Beim Herunterladen spricht man von **auschecken** (eng. *to checkout*) Beim Hochladen spricht man von **pushen** (eng. *to push*)

Es gibt den Befehl **pullen** (eng. *to pull*), der später thematisiert wird.

2.2 1.2 Kurzer Ueberblick ueber die Historie von Git

- Git entstand aus kreativem Chaos und hitziger Diskussion.
- Der Linux-Kernel ist ein grosses Open-Source-Projekt.
- Fruehe Entwicklungsjahre (1991-2002): Aenderungen wurden als Patches und archivierte Dateien ausgetauscht.
- 2002: Umstieg auf proprietäeres DVCS Bitkeeper.
- 2005: Beziehung zwischen Linux-Community und BitKeeper-Unternehmen zerbrach, kostenlose Nutzung von BitKeeper wurde widerrufen.
- Ausloeser fuer Linus Torvalds, ein eigenes Tool zu entwickeln.
- Ziele des neuen Systems:

- Geschwindigkeit
 - Einfaches Design
 - Unterstützung nicht-linearer Entwicklung (tausende parallele Entwicklungs-Banches)
 - Vollständig dezentrale Struktur
 - Effektives Management grosser Projekte wie dem Linux Kernel (Geschwindigkeit und Datenumfang)
- Seit 2005 kontinuierliche Weiterentwicklung und Reifung von Git.
 - Git ist schnell, effizient fuer grosse Projekte und hat ein exzellentes Branching-Konzept fuer nicht-lineare Entwicklung.

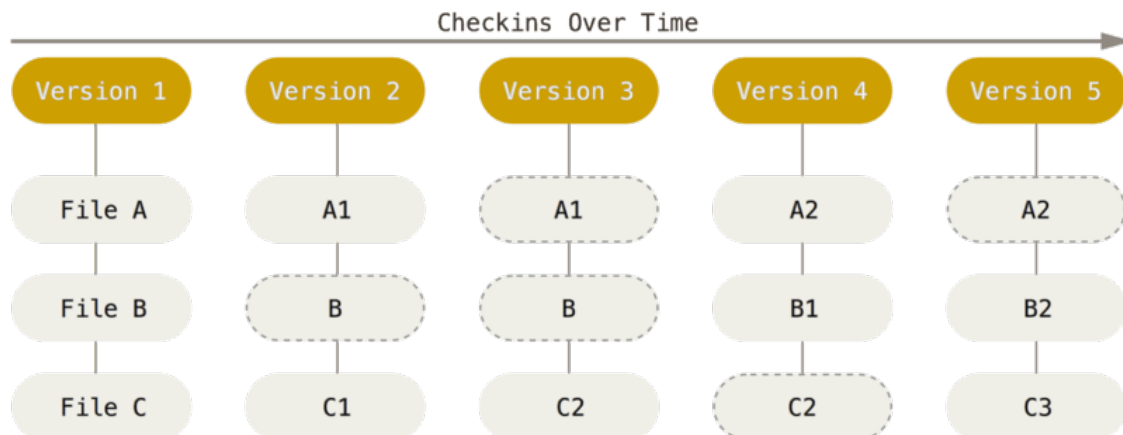
2.3 1.3 Was ist Git?

Git ist eine Sammlung von Dienstprogrammen in der Kommandozeile, die Aenderungen in Dateien verfolgen und aufzeichnen (meistens Quellcode, aber du kannst alle moeglichen Dateien wie Textdateien und sogar Bild-Dateien “tracken”. Dieser Prozess wird als *Versionskontrolle* bezeichnet.

Git ist dezentralisiert, das bedeutet, dass es nicht von einem zentralen Server abhaengig ist, um alte Versionen deiner Dateien aufzubewahren. Stattdessen funktioniert es vollstaendig lokal, indem es diese Daten als Ordner auf deiner Festplatte speichert. Das nennen wir auch *Repository*.

2.3.1 1.3.1 Snapshots statt Unterschiede

Git speichert die verschiedenen Versionen des Programms bzw. der Datei(en) ab (wie Schnappschuesse), im Gegensatz zu anderen System, die lediglich die differenzen zu der urspruenglichen Hauptdatei abgespeichert. Deswegen koennen die Aenderungen als eine start *Stapel von Schnappschuessen*



Speichern der Daten-Historie eines Projekts in Form von Schnappschuesse

2.3.2 1.3.2 Fast jede Funktion funktioniert lokal

Die meisten Aktionen in Git benoetigen nur lokale Dateien und Ressourcen, um ausgefuehrt zu werden – im Allgemeinen werden keine Informationen von einem anderen Computer in deinem Netzwerk benoetigt. Die allermeisten Operationen koennen nahezu ohne jede Verzoegerung aus-

gefuehrt werden, da die vollstaendige Historie eines Projekts bereits auf dem jeweiligen Rechner verfuegbar ist.

Da git immer lokal verfuegbar ist, gibt es viele Vorteile: - Git durchsucht die Projekt-Historie lokal, ohne externe Server. - Vollstaendige Projekthistorie ist sofort verfuegbar. - Aenderungen einer Datei von vor einem Monat koennen lokal verglichen werden. - Kein externer Server noetig fuer Datei-Vergleich oder Historienabfrage. - Offline-Arbeit moeglich, z.B. im Flugzeug oder Zug. - Aenderungen koennen spaeter bei Netzwerkverbindung hochgeladen werden. - Arbeit unabhaengig von VPN-Verfuegbarkeit moeglich.

2.3.3 1.3.3 Git stellt Integritaet sicher

Von allen zu speichernden Daten berechnet Git Pruefsummen (engl. **checksum**) und speichert diese als Referenz zusammen mit den Daten ab. Das macht es unmoeglich, dass sich Inhalte von Dateien oder Verzeichnissen aendern, ohne dass Git das mitbekommt. Git basiert auf dieser Funktionalitaet und sie ist ein integraler Teil der Git-Philosophie. Man kann Informationen deshalb z.B. nicht waehrend der Uebermittlung verlieren oder unwissentlich beschaedigte Dateien verwenden, ohne dass Git in der Lage waere, dies festzustellen.

2.3.4 1.3.4 Git fuegt im Regelfall nur Daten hinzu

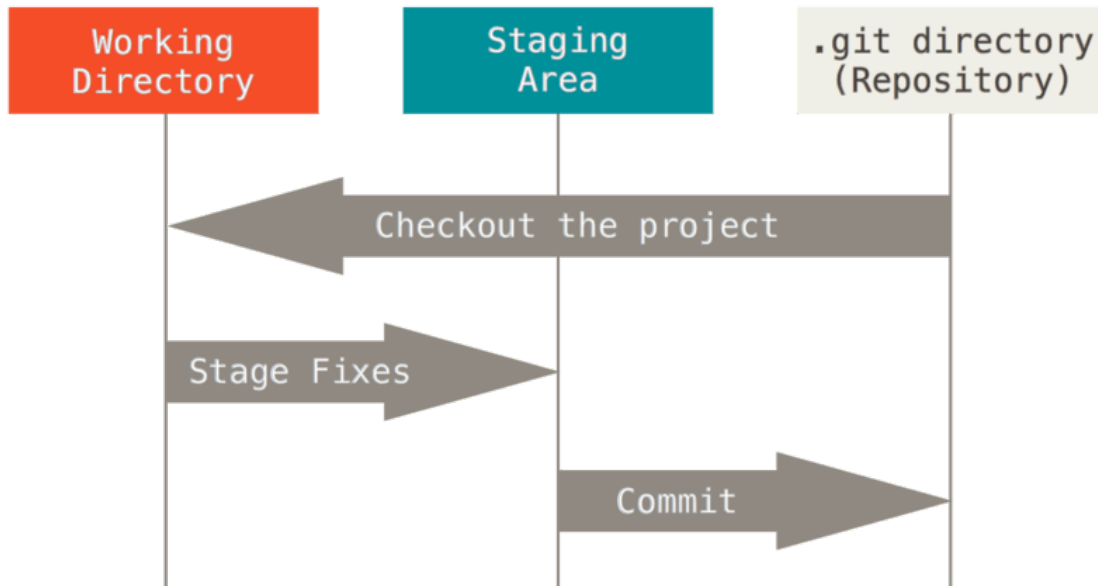
Wenn du Aktionen in Git durchfuehren willst, werden diese fast immer nur Daten zur Git-Datenbank **hinzufuegen**. Deshalb ist es sehr schwer, das System dazu zu bewegen, irgendetwas zu tun, das nicht wieder rueckgaengig zu machen ist, oder dazu, Daten in irgendeiner Form zu loeschen. Unter anderem deshalb macht es so viel Spass mit Git zu arbeiten. Man weiss genau, man kann ein wenig experimentieren, ohne befuerchten zu muessen, irgendetwas zu zerstoenen oder durcheinander zu bringen.

2.3.5 1.3.5 Die drei Zustaende

Es folgt die wichtigste Information, die man sich merken muss, wenn man Git erlernen und dabei Fallstricke vermeiden will. Git definiert drei Hauptzustaende, in denen sich eine Datei befinden kann: **committed** (engl. **committed**), **geaendert** (engl. **modified**) und fuer Commit vorgemerkt (engl. **staged**).

- **Modified** bedeutet, dass eine Datei geaendert, aber noch nicht in die lokale Datenbank eingecheckt wurde.
- **Staged** bedeutet, dass eine geaenderte Datei in ihrem gegenwaertigen Zustand fuer den naechsten Commit vorgemerkt ist.
- **Committed** bedeutet, dass die Daten sicher in der lokalen Datenbank gespeichert sind.

Das fuehrt uns zu den drei Hauptbereichen eines Git-Projekts: dem Verzeichnisbaum (engl. Working Tree), der sogenannten Staging-Area und dem Git-Verzeichnis.



Verzeichnisbaum, Staging-Area und Git-Verzeichnis

Der Verzeichnisbaum ist ein einzelner **Checkout** einer Version des Projekts. Diese Dateien werden aus der komprimierten Datenbank im Git-Verzeichnis geholt und auf der Festplatte so abgelegt, damit du sie verwenden oder aendern kannst.

Die Staging-Area ist in der Regel eine Datei, die sich in deinem Git-Verzeichnis befindet und Informationen darueber speichert, was in deinem naechsten Commit einfließen soll. Der technische Name im Git-Sprachgebrauch ist **Index**, aber der Ausdruck **Staging-Area** funktioniert genauso gut.

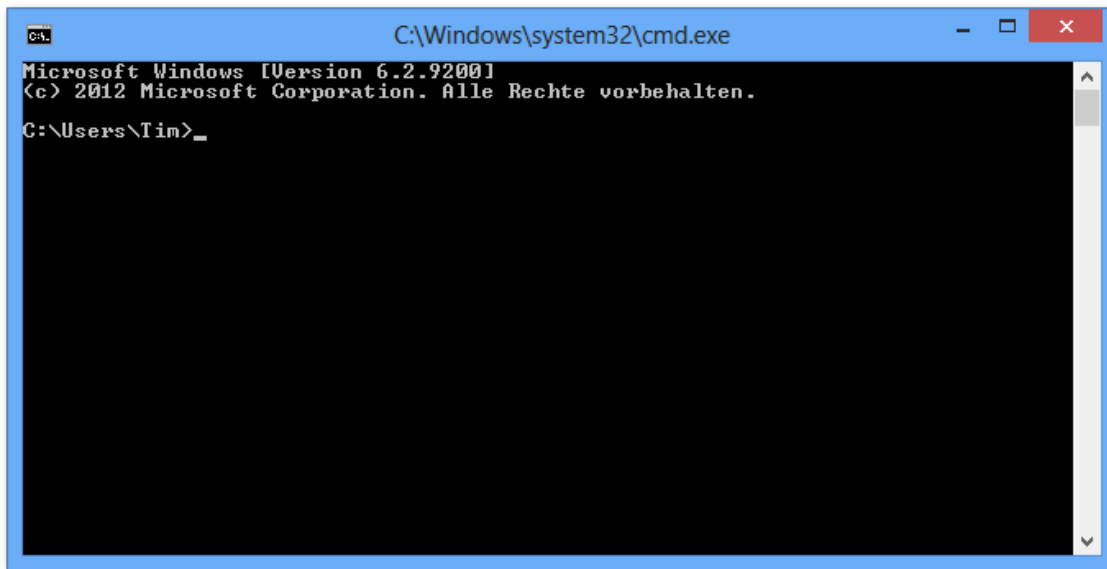
Im Git-Verzeichnis werden die Metadaten und die Objektdatenbank fuer dein Projekt gespeichert. Das ist der wichtigste Teil von Git, dieser Teil wird kopiert, wenn man ein Repository von einem anderen Rechner **klont**.

Der grundlegende Git-Arbeitsablauf sieht in etwa so aus:

1. Du aenderst Dateien in deinem Verzeichnisbaum.
2. Du stellst selektiv Aenderungen bereit, die du bei deinem naechsten Commit beruecksichtigen moechtest, wodurch nur diese Aenderungen in den Staging-Bereich aufgenommen werden.
3. Du fuehrt einen Commit aus, der die Dateien so uebernimmt, wie sie sich in der Staging-Area befinden und diesen Snapshot dauerhaft in deinem Git-Verzeichnis speichert.

Wenn sich eine bestimmte Version einer Datei im Git-Verzeichnis befindet, wird sie als **committed** betrachtet. Wenn sie geaendert und in die Staging-Area hinzugefuegt wurde, gilt sie als fuer den Commit **vorgemerkt** (engl. **staged**). Und wenn sie geaendert, aber noch nicht zur Staging-Area hinzugefuegt wurde, gilt sie als geaendert (engl. **modified**).

2.4 1.4 Die Kommandozeile



Es gibt viele verschiedene Möglichkeiten Git einzusetzen. Auf der einen Seite gibt es die Werkzeuge, die per Kommandozeile bedient werden und auf der anderen, die vielen grafischen Benutzeroberflächen (engl. graphical user interface, GUI), die sich im Leistungsumfang unterscheiden.

Zuerst verwenden wir die Kommandozeile, denn - Alle Git-Befehle können in der Kommandozeile ausgeführt werden. - Grafische Oberflächen bieten oft nur einen Teil der Git-Funktionalitäten. - Kenntnisse in der Kommandozeilenversion helfen beim Umgang mit GUI-Versionen. - Umgekehrt ist das nicht unbedingt der Fall. - Wahl der GUI ist Geschmackssache. - **Kommandozeilenversion ist auf jedem Rechner verfügbar.**

1.5 Git installieren Um die Git-Software benutzen zu können, muss man sie erstmal installieren. Keine Sorge, das ist gar nicht mal so schwierig, wie es klingt.

2.4.1 1.5.1 Mac

Wenn du schon *Xcode* heruntergeladen hast, wurde git vermutlich automatisch mit installiert. Falls das nicht der Fall sein sollte, dann bitte wie folgt vorgehen. Bei Mac kann man das ganz leicht machen, indem man [homebrew](#) installiert und dann den entsprechenden Befehl für Git in der Kommandozeile ausführt.

```
[ ]: /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install git
```

1.5.2 Linux Normalerweise ist Git Teil von Linux-Distributionen und bereits installiert, da es sich um ein Tool handelt, das ursprünglich für die Linux-Kernel-Entwicklung geschrieben wurde. Aber es gibt Situationen, in denen das nicht der Fall ist. Wenn das nicht der Fall sein sollte, muss man einen der beiden Befehle eingeben, um Git zu installieren:

```
[ ]: sudo dnf install git-all
```

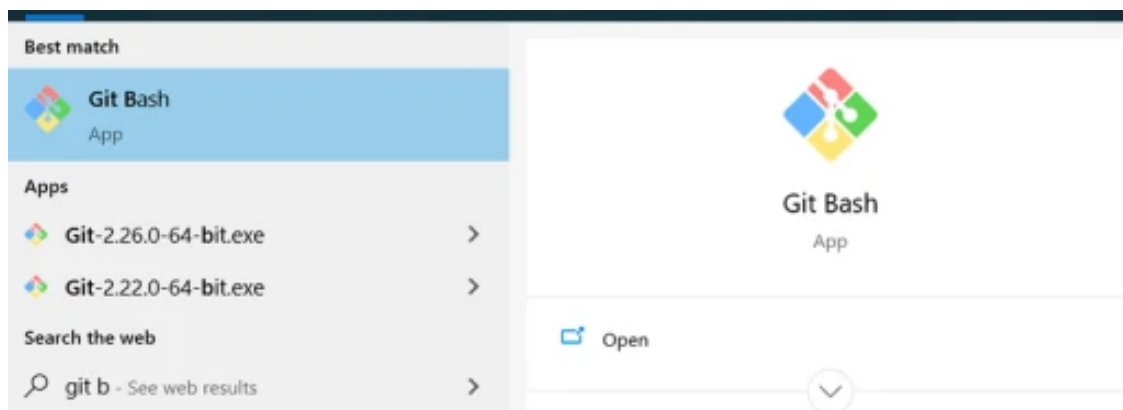
oder

```
[ ]: sudo apt install git-all
```

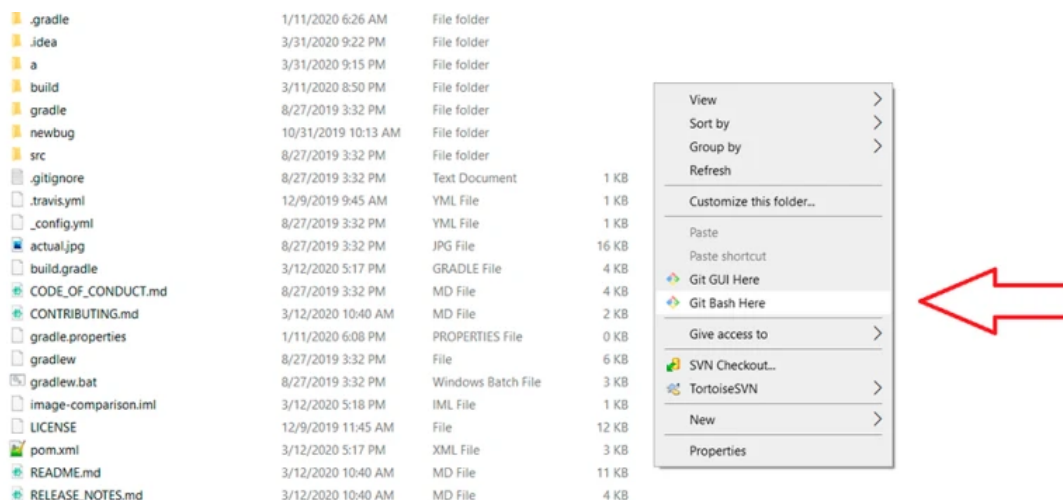
2.4.2 1.5.3 Windows

Auf Windows git zu installieren ist ein bisschen anders, denn man muss eine externe Software herunterladen und installieren. Das geht leider nicht mit der Kommandozeile. Wie ueblich muss man eine *exe-Datei* herunterladen und ausfuehren. Hier ist alles aber ganz einfach: 1. Klick auf den folgenden [Link](#) 2. Fuehre die Installation durch und fertig. 3. Dazu verwenden wir die von Windows bereitgestellte Bash-Konsole. Unter Windows muss man Git Bash ausfuehren. So sieht es im Startmenue aus:

Nachdem die .exe Datei heruntergeladen wurde, soll sie nun ausgefuehrt werden. Die restlichen Schritte werden bei der Installation vom Installationsprogramm erklart.



Dies ist nun eine Eingabeaufforderung, mit der man arbeiten kann. Um nicht jedes Mal in den Ordner mit dem Projekt gehen zu muessen, um Git dort zu oeffnen, kann man mit der rechten Maustaste die Eingabeaufforderung im Projektordner mit dem von uns benoetigten Pfad oeffnen:



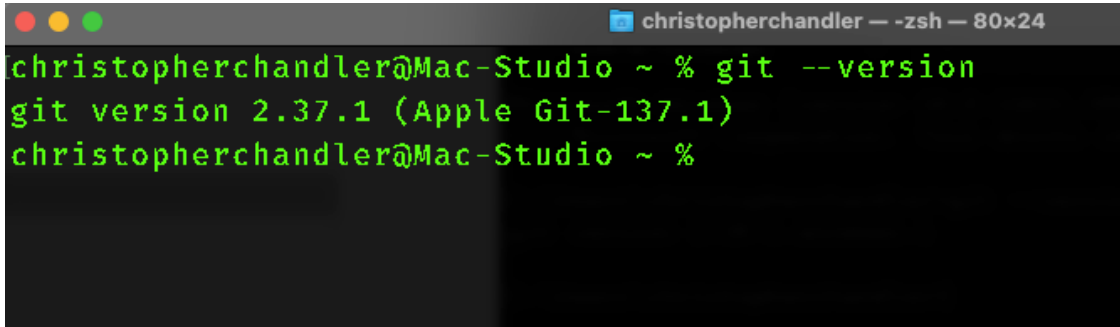
Falls das nicht funktioniert hat, bitte alternativ oder ergaenzend dazu die [folgende Anleitung](#) be-

nutzen.

2.4.3 1.5.4 Installation ueberpruefen

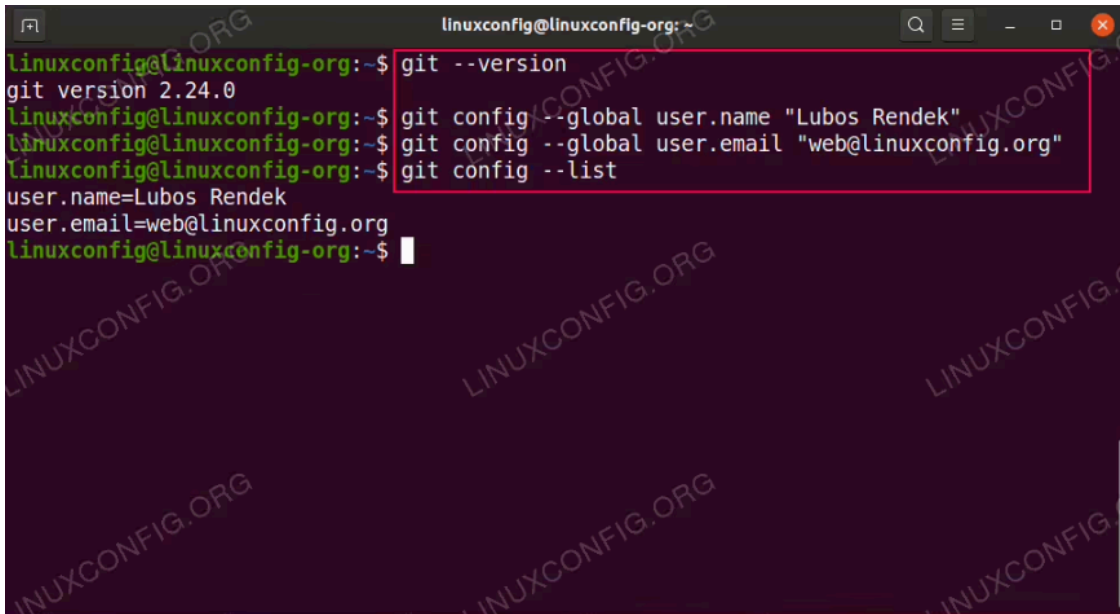
Um zu ueberpruefen, ob git erfolgreich installiert wurde bzw. vorhanden ist, kann man `git --version` in der Kommandozeile eingeben. Wenn `git` installiert ist, soll sowas in der Konsole erscheinen.

2.4.4 1.5.5 Mac Version

A screenshot of a terminal window titled 'christopherchandler — zsh — 80x24'. The prompt is 'christopherchandler@Mac-Studio ~ %'. The user enters 'git --version', and the output is 'git version 2.37.1 (Apple Git-137.1)'. The prompt then returns to 'christopherchandler@Mac-Studio ~ %'.

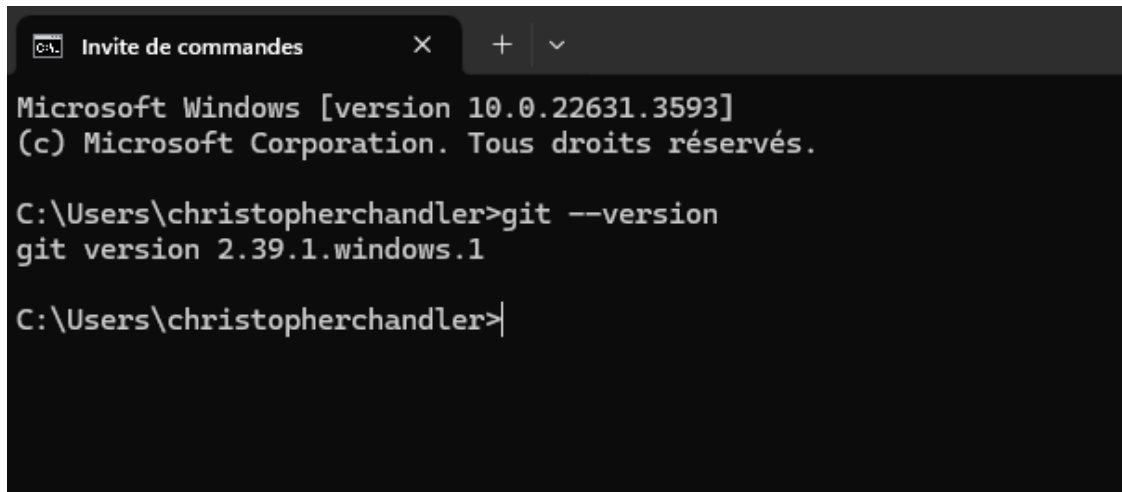
```
christopherchandler@Mac-Studio ~ % git --version
git version 2.37.1 (Apple Git-137.1)
christopherchandler@Mac-Studio ~ %
```

2.4.5 1.5.6 Linux Version

A screenshot of a terminal window titled 'linuxconfig@linuxconfig-org: ~'. The prompt is 'linuxconfig@linuxconfig-org:~\$'. The user enters 'git --version', and the output is 'git version 2.24.0'. The user then enters 'git config --global user.name "Lubos Rendek"', 'git config --global user.email "web@linuxconfig.org"', and 'git config --list'. The output of 'git config --list' is 'user.name=Lubos Rendek' and 'user.email=web@linuxconfig.org'. The prompt then returns to 'linuxconfig@linuxconfig-org:~\$'.

```
linuxconfig@linuxconfig-org:~$ git --version
git version 2.24.0
linuxconfig@linuxconfig-org:~$ git config --global user.name "Lubos Rendek"
linuxconfig@linuxconfig-org:~$ git config --global user.email "web@linuxconfig.org"
linuxconfig@linuxconfig-org:~$ git config --list
user.name=Lubos Rendek
user.email=web@linuxconfig.org
linuxconfig@linuxconfig-org:~$
```

2.4.6 1.5.7 Windows Version



2.5 1.6 Hilfe finden

Wie beim Programmieren geht es nicht darum alle Befehle auswendig zu kennen. Wenn man nicht weiter weiss oder einen Befehl vergessen hat bzw. nicht kennt, kann man ganz einfach in der Konsole folgendes eingeben: - `git help <verb>` - `git <verb> --help` - `man git-<verb>`

```
[ ]: !git help
```

```
[ ]: !git help config
```

```
[ ]: !git help commit
```

3 Teil 2 Git Grundlagen

3.1 2.1 Ein Git-Repository anlegen

Du hast zwei Möglichkeiten, ein Repository auf deinem Rechner anzulegen. - Du kannst ein lokales Verzeichnis, das sich derzeit nicht unter Versionskontrolle befindet, in ein Git-Repository verwandeln, oder - Du kannst ein bestehendes Git-Repository von einem anderen Ort aus klonen.

3.1.1 2.1.1 Lokal

Wir koennen ein lokales Repository mit `git init` anlegen.

```
[ ]: %%%bash
rm -rf /Users/christopherchandler/code_repos/RUB/test_repo
# rmdir ist der Befehl zum Entfernen von Verzeichnissen
# -rf sind Optionen fuer den Befehl 'rm', nicht 'rmdir'
# Es sollte rm -rf verwendet werden, aber das koennte gefaehrlich sein, da es
↪alles in dem angegebenen Pfad loescht

mkdir /Users/christopherchandler/code_repos/RUB/test_repo
```



```
# mkdir erstellt ein neues Verzeichnis mit dem angegebenen Pfad

cd /Users/christopherchandler/code_repos/RUB/test_repo
# cd wechselt das aktuelle Verzeichnis zum angegebenen Pfad

git init
# git init initialisiert ein neues leeres Git-Repository im aktuellen
↳ Verzeichnis

git status
# git status zeigt den Status des Git-Repositories an (z. B. unversionierte
↳ Dateien, Änderungen usw.)
```

Der Befehl erzeugt ein Unterverzeichnis `.git`, in dem alle relevanten Git-Repository-Daten enthalten sind, also so etwas wie ein Git-Repository Grundgerüst. Normalerweise kann man dieses Unterverzeichnis nicht sehen, weil es versteckt ist. Das geht aber mit dem Befehl `ls -a`

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem die Befehle ausgeführt werden sollen
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Zeige alle Dateien und Verzeichnisse (einschliesslich versteckter) im
↳ aktuellen Verzeichnis an
ls -a
```

Unser Repo ist leer. Erzeugen wir eine Text-Datei.

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem die Befehle ausgeführt werden sollen
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Definiere die Commit-Nachricht als Variable
msg="Text zur test.txt hinzufügen"

# Schreibe den Inhalt der Commit-Nachricht in die Datei 'test.txt'
echo "$msg" > test.txt

# Zeige den Inhalt der 'test.txt'-Datei an
cat test.txt
echo "" # Leere Zeile fuer bessere Lesbarkeit in der Ausgabe

# Zeige den Status des Git-Repositorys an
git status
```

```
[ ]: %%bash
```

```
# Wechsle in das Verzeichnis, in dem die Git-Befehle ausgefuehrt werden sollen
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Fuege die Datei 'test.txt' der Staging Area hinzu
git add test.txt

# Fuehre einen Commit durch mit der Commit-Nachricht "add test.txt"
git commit -m "add test.txt"
```

3.1.2 2.1.2 Remote

Wenn du eine Kopie eines existierenden Git-Repositorys aus dem Internet oder lokal anlegen moechtest – um beispielsweise an einem Projekt mitzuarbeiten – kannst du den Befehl `git clone` verwenden. Du klonst ein Repository mit dem Befehl: `git clone [url]`

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem das Repository geklont werden soll
cd /Users/christopherchandler/code_repos/RUB

# Klonen des GitHub-Repositorys 'Hello-World' von Octocat (Beispielrepository) /
  ↳ Ordner wird nicht angegeben und der Ordner wird in dem lokalen Verzeichnis
  ↳ gespeichert.
git clone https://github.com/octocat/Hello-World
```

Du kannst auch den Ort bestimmt, wo das geklonte Repo abgelegt werden soll.

```
git clone URL ORDNER_NAME
```

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem das Repository geklont werden soll
cd /Users/christopherchandler/code_repos/RUB/Hello-World

ls -a
```

3.1.3 2.1.3 Kollaboration

Bei Git spielen die Befehle `push` und `pull` eine zentrale Rolle fuer die Zusammenarbeit und den Austausch von Aenderungen an einem Projekt. Diese Befehle ermoeeglichen es dir, deine lokalen Aenderungen mit einem zentralen Remote-Repository zu synchronisieren, das als Kopie deines Projekts auf einem Server wie GitHub oder GitLab fungiert.

Push: Lokale Aenderungen mit dem Remote-Repository teilen Der Befehl `git push` sendet deinen lokalen Commits und Branches an das Remote-Repository. Dies ermoeeglicht es dir, deine Arbeit mit anderen Teammitgliedern zu teilen und sicherzustellen, dass alle auf dem gleichen Stand sind.

```
[31]: %%%bash
cd /Users/christopherchandler/code_repos/RUB
rm -rf /Users/christopherchandler/code_repos/RUB/simple_calculator
git clone git@github.com:christopher-chandler/simple_calculator.git
ls -a
```

Cloning into 'simple_calculator'...

```
.
..
.DS_Store
Hello-World
hello_world_repo
simple_calculator
test_repo
```

```
[32]: %%%bash
cd /Users/christopherchandler/simple_calculator
touch calculator.py
git add calculator.py
git commit -m "add calculator.py"
git push
```

On branch main

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

nothing to commit, working tree clean

Everything up-to-date

Pull: Neueste Aenderungen vom Remote-Repository abrufen Mit dem Befehl `git pull` holen Sie die neuesten Aenderungen und Branches vom Remote-Repository in Ihr lokales Repository. Dies ist wichtig, um auf dem Laufenden zu bleiben und die Arbeit anderer Teammitglieder in Ihr Projekt zu integrieren.

```
[33]: %%%bash
cd /Users/christopherchandler/simple_calculator
git pull
```

```
From github.com:christopher-chandler/simple_calculator
   f553675..9ff3173  main    -> origin/main
```

Updating f553675..9ff3173

Fast-forward

```
calculator.py | 1 +
```

```
1 file changed, 1 insertion(+)
```

Dieser Befehl holt die neuesten Aenderungen vom Main-Branch des Remote-Repositorys auf GitHub und merget sie in deinen lokalen main-Branch ein.

3.1.4 Hinweis

Wichtig: Sicherstellen, dass dein lokales Repository auf dem neuesten Stand ist, bevor du den Befehl `git pull` ausfuehrst. Dies kannst du mit dem Befehl `git fetch` tun, der die neuesten Aenderungen vom Remote-Repository herunterlaedt, ohne sie jedoch mit deinem lokalen Repository zu mergen.

Zusammenspiel von Push und Pull Die Befehle `git push` und `git pull` arbeiten zusammen, um einen reibungslosen Workflow fuer die Zusammenarbeit in Git-Projekten zu gewaehrleisten. Durch regelmassiges Pushen und Pullen deiner Aenderungen kannst du sicherstellen, dass alle Teammitglieder auf dem gleichen Stand sind und Konflikte vermieden werden.

Best Practices: - Push deine Aenderungen regelmassig, um anderen Entwicklern Zugang zu deiner Arbeit zu ermoeglichen. - Pull sie vor Beginn der Arbeit an einem neuen Feature, um die neueste Codebasis zu erhalten.

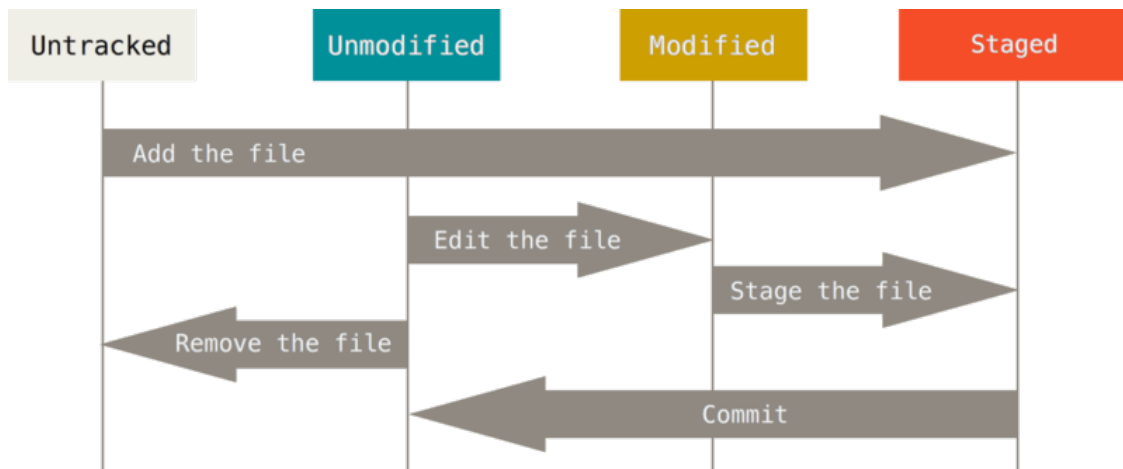
3.2 2.2 Aenderungen nachverfolgen und im Repository speichern

An dieser Stelle solltest du ein **angemessenes** Git-Repository auf deinem lokalen Computer und eine Checkout- oder Arbeitskopie aller deiner Dateien vor dir haben. Normalerweise wirst du damit beginnen wollen, Aenderungen vorzunehmen und Schnappschuesse dieser Aenderungen in dein Repository zu committen, wenn das Projekt so weit fortgeschritten ist, dass du es sichern moechten.

Denke daran, dass sich jede Datei in deinem Arbeitsverzeichnis in einem von zwei Zustaenden befinden kann: **tracked** oder **untracked** – Aenderungen an der Datei werden verfolgt (engl. **tracked**) oder eben nicht (engl. **untracked**). Tracked Dateien sind Dateien, die im letzten Snapshot enthalten sind. Genauso wie alle neuen Dateien in der Staging-Area. Sie koennen entweder unveraendert, modifiziert oder fuer den naechsten Commit vorgemerkt (staged) sein. Kurz gesagt, nachverfolgte Dateien sind Dateien, die Git kennt.

Alle anderen Dateien in deinem Arbeitsverzeichnis dagegen, sind nicht versioniert: Das sind all diejenigen Dateien, die nicht schon im letzten Schnappschuss enthalten waren und die sich nicht in der Staging-Area befinden. Wenn du ein Repository zum ersten Mal klonst, sind alle Dateien versioniert und unveraendert. Nach dem Klonen wurden sie ja ausgecheckt und bis dahin hast du auch noch nichts an ihnen veraendert.

Sobald du anfaengst versionierte Dateien zu bearbeiten, erkennt Git diese als modifiziert, weil sie sich im Vergleich zum letzten Commit veraendert haben. Die geaenderten Dateien kannst du dann fuer den naechsten Commit vormerken und schliesslich alle Aenderungen, die sich in der Staging-Area befinden, einchecken (engl. committen). Danach wiederholt sich dieser Vorgang.



Das wichtigste Hilfsmittel, um den Zustand zu ueberpruefen, in dem sich deine Dateien gerade befinden, ist der Befehl `git status`. Wenn du diesen Befehl unmittelbar nach dem Klonen eines Repositories ausfuehren, sollte er in etwa folgende Ausgabe liefern:

```
[2]: %%bash

# Wechsle in das uebergeordnete Verzeichnis, in dem wir arbeiten moechten
cd /Users/christopherchandler/code_repos/RUB

# Entferne das vorhandene Verzeichnis 'test_repo' und alle seine Inhalte
rm -rf /Users/christopherchandler/code_repos/RUB/test_repo

# Erstelle ein neues Verzeichnis namens 'test_repo'
mkdir test_repo

# Wechsle in das neu erstellte 'test_repo'-Verzeichnis
cd test_repo

# Initialisiere ein neues Git-Repository im aktuellen Verzeichnis
git init

# Zeige den Status des Git-Repositories an, um den aktuellen Zustand zu ueberpruefen
git status
```

```
bash: line 3: cd: /Users/christopherchandler/code_repos/RUB: No such file or directory
```

```
mkdir: test_repo: File exists
```

```
Reinitialized existing Git repository in
/Users/christopherchandler/code_repos/christopher-
chandler/RUB/Git_leicht_gemacht/test_repo/.git/
On branch main
```

No commits yet

nothing to commit (create/copy files and use "git add" to track)

3.2.1 2.2.1 Zustand von Dateien pruefen

Nehmen wir einmal an, du fuegst eine neue Datei mit dem Namen README zu deinem Projekt hinzu. Wenn die Datei zuvor nicht existiert hat und du jetzt git status ausfuehrst, zeigt Git die bisher nicht versionierte Datei wie folgt an:

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem die Befehle ausgefuehrt werden sollen
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Erstelle eine neue README-Datei mit dem Namen 'readme.me' und schreibe
  ↳ mehrere Zeilen hinein
cat > readme.me <<EOL
Hallo, ich bin eine Readme.MD datei.
Ich bin ganz klein, aber ich kann grosse Sachen bewirken.
EOL

# Zeige den Inhalt der README-Datei an, um sicherzustellen, dass die Zeilen
  ↳ korrekt geschrieben wurden
cat readme.me

# Zeige den Status des Git-Repositorys an, um den aktuellen Zustand zu
  ↳ ueberpruefen
git status
```

Dateien, die im Abschnitt „Untracked files“ aufgelistet werden, sind noch nicht versioniert. Die Datei README erscheint dort, weil sie im letzten Schnappschuss nicht enthalten war und noch nicht gestaged wurde. Git nimmt solche Dateien nicht automatisch in die Versionsverwaltung auf, um unerwuenschte Dateien wie generierte Binaerdateien nicht hinzuzufuegen. Um Aenderungen an der README zu verfolgen, muessen wir sie explizit zur Versionsverwaltung hinzufuegen.

3.2.2 2.2.2 Neue Dateien zur Versionsverwaltung hinzufuegen

Um eine neue Datei zu versionieren, kannst du den Befehl git add verwenden. Fuer deine neue README Datei, kannst du folgendes ausfuehren: git add readme

```
[ ]: %%bash

# Wechseln zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Datei readme.me zum Index hinzufuegen
git add readme.me
```

```
# Zeige den aktuellen Status des Git-Repositories an
git status
```

Leider wird der Befehl `git add` oft missverstanden. Viele assoziieren damit, dass damit Dateien zum Projekt hinzugefügt werden. Wie du aber gerade gelernt hast, wird der Befehl auch noch für viele andere Dinge eingesetzt. Wenn du den Befehl `git add` einsetzt, solltest du das eher so sehen, dass du damit einen bestimmten Inhalt für den nächsten Commit vormerkst.

3.2.3 2.2.3 Kompakter Status

Die Ausgabe von `git status` ist sehr umfassend und auch ziemlich wortreich. Git hat auch ein Kurzformat, mit dem du deine Änderungen kompakter sehen kannst. Wenn du `git status -s` oder `git status --short` ausführst, erhältst du eine kürzere Darstellung des Befehls:

```
[ ]: %%bash

# Wechsel zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Git-Status abrufen und verkürzte Ausgabe anzeigen
git status --short
```

Neue Dateien, die nicht versioniert werden, werden mit `??` markiert. Neue Dateien, die der Staging-Area hinzugefügt wurden, haben ein `A`, geänderte Dateien haben ein `M` usw.

3.2.4 2.2.4 Geänderte Dateien zur Staging-Area hinzufügen

Las uns jetzt eine bereits versionierte Datei ändern. Wenn du zum Beispiel eine bereits unter Versionsverwaltung stehende Datei mit dem Dateinamen `CONTRIBUTING.md` änderst und danach den Befehl `git status` erneut ausführst, erhältst du in etwa folgende Ausgabe:

```
[ ]: %%bash

# Wechseln zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Variable mit der Zeichenkette definieren
VAR='Hello, World! How are you?'

# Die Variable zur readme.me Datei hinzufügen (append)
echo $VAR >> readme.me

# Git-Status abrufen, um Änderungen anzuzeigen
git status
```

Die Datei erscheint im Abschnitt „Changes not staged for commit“. Das bedeutet, dass eine versionierte Datei im Arbeitsverzeichnis verändert worden ist, aber noch nicht für den Commit vorgemerkt wurde. Um sie vorzumerken, führst du den Befehl `git add` aus.

Um die Veränderungen vorzumerken, führst du den Befehl `git add` aus. Der Befehl `git add` wird zu vielen verschiedenen Zwecken eingesetzt. Man verwendet ihn, um neue Dateien zur Versionsver-

waltung hinzuzufuegen, Dateien fuer einen Commit vorzumerken und verschiedene andere Dinge – beispielsweise einen Konflikt aus einem Merge als aufgeloesst zu kennzeichnen.

3.2.5 2.2.5 Ignorieren von Dateien

Haeufig gibt es eine Reihe von Dateien, die Git nicht automatisch hinzufuegen oder schon gar nicht als „nicht versioniert“ (eng. untracked) anzeigen soll. Dazu gehoeren in der Regel automatisch generierte Dateien, wie Log-Dateien oder Dateien, die von deinem Build-System erzeugt werden. In solchen Faellen kannst du die Datei .gitignore erstellen, die eine Liste mit Vergleichsmustern enthaelt. Hier ist eine .gitignore Beispieldatei:

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Erzeuge eine neue Datei .gitignore
touch .gitignore

# Erzeuge eine neue Datei hello_world.txt
touch hello_world.txt

# Erzeuge eine neue Datei code.py
touch code.py

# Fuege der .gitignore Datei den Eintrag "*.txt" hinzu
echo "*.txt" >> .gitignore
```

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Fuege die Datei code.py zur Staging-Area von Git hinzu
git add code.py

# Fuege alle anderen geaenderten Dateien (inklusive .gitignore) zur
↪Staging-Area von Git hinzu
git add .gitignore

# Zeige den aktuellen Status des Git-Repositories an
git status
```

3.2.6 2.2.6 Ueberpruefen der Staged- und Unstaged-Aenderungen

Um die Aenderungen zu sehen, die du noch nicht zum Commit vorgemerkt hast, gibst du den Befehl `git diff` ohne weitere Argumente, ein:

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
```



```
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Zeige die Unterschiede zwischen dem aktuellen Arbeitsverzeichnis und der
↳ Staging-Area an
git diff
```

3.2.7 2.2.7 Die Aenderungen committen

Bis jetzt haben wir die Aenderungen hinzugefuegt, aber nicht comittet. Nachdem deine Staging-Area nun so eingerichtet ist, wie du es wuenscht, kannst du deine Aenderungen committen. Denke daran, dass alles, was noch nicht zum Commit vorgemerkt ist – alle Dateien, die du erstellt oder geaendert hast und fuer die du seit deiner Bearbeitung nicht mehr `git add` ausgefuehrt hast – nicht in diesen Commit einfließen werden.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Fuehre einen Commit durch mit der Commit-Nachricht "initial commit"
git commit -m "initial commit"

# Zeige den aktuellen Status des Git-Repositories an
git status
```

3.3 2.3 Anzeigen der Commit-Historie

Nachdem du mehrere Commits erstellt hast, oder wenn du ein Repository mit einer bestehenden Commit-Historie geklont hast, wirst du wahrscheinlich zurueckschauen wollen, um zu erfahren, was geschehen ist. Das wichtigste und maechtigste Werkzeug dafuer ist der Befehl `git log`.

```
[ ]: %%bash
# Wechseln zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Git-Log anzeigen fuer den gesamten Verlauf (alle Commits)
git log
```

Eine der hilfreichsten Optionen ist `-p` oder `--patch`. Sie zeigt die Aenderungen (die patch-Ausgabe) an, die bei jedem Commit durchgefuehrt wurden. Du kannst auch die Anzahl der anzuzeigenden Protokolleintraege begrenzen, z.B. mit `-2` werden nur die letzten beiden Eintraege dargestellt.

```
[ ]: %%bash

# Wechseln zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Git-Log anzeigen fuer die letzten beiden Commits mit Commit-Diffs
git log -p -2
```

Wenn du einige gekuerzte Statistiken fuer jeden Commit sehen moechtest, kannst du die Option `--stat` verwenden:

```
[ ]: %%bash

# Wechseln zum Verzeichnis des Git-Repositorys
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Git-Log mit Dateistatistiken anzeigen
git log --stat
```

3.4 2.4 Ungewollte Aenderungen rueckgaengig machen

```
[ ]: %%bash

# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Erzeuge eine neue Datei hello.md
touch hello.md

# Fuege die Datei hello.md zur Staging-Area von Git hinzu
git add hello.md

# Fuehre einen Commit durch mit der Nachricht "add hello.md"
git commit -m "add hello.md"
```

```
[ ]: %%bash

# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Fuehre einen Commit durch, um die Commit-Nachricht zu aendern (amend) und
↪ setze die Nachricht auf "add the hello.md as a new file"
git commit --amend -m "add the hello.md as a new file"

# Zeige die Commit-Historie und die geaenderten Dateien an
git log --stat
```

3.5 2.4.1 Eine Datei aus der Staging-Area entfernen

Die naechsten beiden Abschnitte erlaeuern, wie du mit deiner Staging-Area und den Aenderungen des Arbeitsverzeichnisses arbeitest. Der angenehme Nebeneffekt ist, dass der Befehl, mit dem du den Zustand dieser beiden Bereiche bestimmst, dich auch daran erinnert, wie du Aenderungen an ihnen rueckgaengig machen kannst. Nehmen wir zum Beispiel an, du hast zwei Dateien geaendert und moechtest sie als zwei separate Aenderungen uebertragen, aber du gibst versehentlich `git add *` ein und stellst sie dann beide in der Staging-Area bereit. Wie kannst du eine der beiden aus der Staging-Area entfernen? Der Befehl `git status` meldet:

```
[ ]: %%%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Erzeuge eine neue Datei new_code.py
touch new_code.py

# Fuege die Datei new_code.py zur Staging-Area von Git hinzu
git add new_code.py

# Zeige den Status des Git-Repositories an
git status
```

Wenn man den Befehl aufuehrt, sieht man, dass das Staging area leer ist.

```
[ ]: %%%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Setze die Datei new_code.py im Staging-Bereich zurueck (entfernt sie aus dem
↪Staging-Bereich)
git reset HEAD new_code.py

# Zeige den Status des Git-Repositories an
git status
```

3.5.1 2.4.2 Aenderung in einer modifizierten Datei zuruecknehmen

Was ist, wenn du feststellst, dass du deine Aenderungen an der Datei nicht behalten willst? Wie kannst du sie in den Ursprungszustand zuruecksetzen, so wie sie beim letzten Commit ausgesehen hat (oder anfaenglich geklont wurde, oder wie auch immer du sie in dein Arbeitsverzeichnis bekommen hast)? Gluecklicherweise sagt dir git status, wie du das machen kannst. Im letzten Beispiel sieht die Unstaged-Area so aus:

```
[ ]: %%%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Schreibe "Das ist eine Zeile." in die Datei hello_world.md
echo "Das ist eine Zeile." > hello_world.md

# Fuege die Datei hello_world.md zur Staging-Area von Git hinzu
git add hello_world.md

# Zeige den Inhalt der Datei hello_world.md an
cat hello_world.md
```

```
[ ]: %>%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Schreibe "Das ist eine zweite Zeile." in die Datei hello_world.md
echo "Das ist eine zweite Zeile." > hello_world.md

# Ausgabe vor dem Auschecken
echo "vor dem Auschecken"

# Zeige den Inhalt der Datei hello_world.md an
cat hello_world.md

# Setze die Datei hello_world.md auf die letzte im Git gespeicherte Version.
↪zurueck
git checkout -- hello_world.md

# Ausgabe nach dem Auschecken
echo "nach dem Auschecken"

# Zeige den Inhalt der Datei hello_world.md an
cat hello_world.md
```

3.6 2.6 Taggen

Wie die meisten VCSs hat Git die Moeglichkeit, bestimmte Punkte in der Historie eines Repositorys als wichtig zu markieren. Normalerweise verwenden Leute diese Funktionalitaet, um Releases zu markieren.

```
[ ]: %>%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Erstelle einen neuen Tag mit der Bezeichnung "v1.0"
git tag v1.0
```

```
[ ]: %>%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Zeige eine Liste aller Tags im aktuellen Git-Repository an
git tag -l
```

3.7 2.6.1 Annotierte Tags

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo
git status
# Erstelle einen annotierten Tag mit der Bezeichnung "v1.4" und der Nachricht
↪ "meine Version 1.4"
git tag -a v1.3 -m "meine Version 1.4"
git tag -l
```

3.7.1 2.6.2 Tags loeschen

Wenn bestimmte tags nicht mehr benoetigt werden, koennen sie geloescht werden.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Loesche den Tag mit der Bezeichnung "v1.0"
git tag -d v1.0
```

3.7.2 2.6.3 Tags auschecken

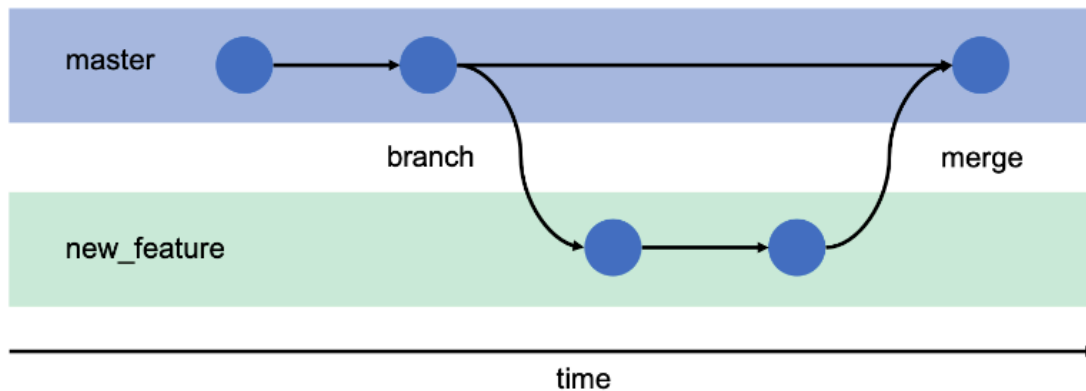
Tags sind hilfreich, da man sie benutzt kann, um bestimmte Zustaende des Codes auszuchecken.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/code_repos/RUB/test_repo
cd /Users/christopherchandler/code_repos/RUB/test_repo

# Versuche, zum Tag v1.1 zu wechseln (Achtung: Tags sind in Git standardmaessig
↪ nicht direkt wechselbar)
git checkout v1.1
```

4 Teil 3 - Git Branching

4.1 3.1 Branches auf einen Blick



Nahezu jedes VCS unterstützt eine Form von Branching. Branching bedeutet, dass du von der Hauptlinie der Entwicklung abzweigen und deine Arbeit fortsetzen kannst, ohne die Hauptlinie durcheinanderzubringen. In vielen VCS-Tools ist das ein etwas aufwändiger Prozess, bei dem du oft eine neue Kopie deines Quellcode-Verzeichnisses erstellen musst, was bei grossen Projekten viel Zeit in Anspruch nehmen kann.

Wenn man ein neues Feature entwickelt, wird es als gute Praxis angesehen, an einer Kopie des Originalprojekts zu arbeiten, die als Branch bezeichnet wird. Branches haben ihre eigene Historie und isolieren ihre Änderungen voneinander, bis man sich entscheidet, sie wieder zusammenzuführen. Dies geschieht aus verschiedenen Gründen:

- Eine bereits funktionierende, stabile Version des Codes wird nicht von ungewünschten Fehlern beeinträchtigt
- Viele Funktionen können sicher von mehreren Entwicklern gleichzeitig entwickelt werden
- Die Entwickler können an ihrem eigenen Branch arbeiten, ohne das Risiko, dass sich ihre Codebasis durch die Arbeit eines anderen Entwicklers ändert.
- Wenn man sich nicht sicher ist, was das Beste ist, können mehrere Versionen desselben Features auf verschiedenen Branches entwickelt und dann miteinander verglichen werden.

4.1.1 3.2. Erzeugen eines Neuen Branches

Bevor man Branches benutzen kann, muss man sie erstmal erzeugen.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Erstelle einen neuen Branch mit dem Namen "testing"
git branch testing
```

Dieser Befehl erzeugt einen neuen Zeiger, der auf denselben Commit zeigt, auf dem du dich gegenwärtig befindest.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Wechsle zum Branch "main"
git checkout main

# Zeige die Commit-Historie kompakt (einzeilige Ausgabe) mit Dekorationen ↪
↪(Branch- und Tag-Namen) an
git log --oneline --decorate
```

4.1.2 3.3. Wechseln des Branches

Um zu einem existierenden Branch zu wechseln, führe die Anweisung `git checkout` aus. Lass uns zum neuen testing Branch wechseln.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Wechsle zum Branch "testing"
git checkout testing

# Füge die Zeile "b=23" der Datei code.py hinzu
echo "b=23" >> code.py
```

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Füge die Änderung (Zeile "a=23" hinzufügen) in die Datei code.py zur ↪
↪Staging-Area von Git hinzu
echo "a=23" >> code.py
git add code.py

# Führe einen Commit durch mit der Nachricht "add variable"
git commit -m "add variable"

# Zeige den aktuellen Status des Git-Repositories an
git status
```

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Wechsle zum Branch "testing"
git checkout testing
```

```
# Zeige die Commit-Historie (Git-Log) fuer den Branch "testing" an
git log
```

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Wechsle zum Branch "main"
git checkout main

# Zeige die Commit-Historie (Git-Log) fuer den Branch "main" an
git log
```

4.2 3.4 Merging

Lass uns ein einfaches Beispiel fuer das Verzweigen und Zusammenfuehren (engl. branching and merging) anschauen, wie es dir in einem praxisnahen Workflow begegnen koennte. Stell dir vor, du fuehrst folgende Schritte aus:

- Du arbeitest an einer Website
- Du erstellst einen Branch fuer eine neue Anwendergeschichte (engl. User Story), an der du gerade arbeitest
- Du erledigst einige Arbeiten in diesem Branch

In diesem Moment erhaeltst du einen Anruf, dass ein anderes Problem kritisch ist und ein Hotfix benoetigt wird. Dazu wirst du folgendes tun:

- Du wechselst zu deinem Produktions-Branch
- Du erstellst einen Branch, um den Hotfix einzufuegen
- Nachdem der Test abgeschlossen ist, mergst du den Hotfix-Branch und schiebst ihn in den Produktions-Branch
- Du wechselst zurueck zu deiner urspruenglichen Anwenderstory und arbeitest daran weiter

4.2.1 3.4.1 Einfaches Merging

In unserem vorherigen Beispiel haben wir verschiedene Branches, aber wir haben diese Branches nicht zusammengefuehrt. Es kommt darauf an, was fuer einen Arbeitsablauf du hast, aber irgendwann muessen bzw. sollen diese Branches zusammengefuehrt werden. Dieses Vorgehen nennt sich mergen.

Wir machen das ganz einfach, indem wir die Aenderungen auf unserem Branch commit

5 3.4.2 Testing Branch

In diesem Branch nehmen wir irgendwelche Aenderungen in einem neuen Branch vor.


```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Erstelle die Datei code.py und fuege den Python-Code hinzu
> code.py # Erzeugt die Datei code.py (falls sie noch nicht existiert) oder
↳loescht ihren Inhalt (falls sie existiert)
cat <<EOL >> code.py # Fuegt den folgenden Python-Code in die Datei code.py ein
# Einfache "Hello World" Funktion
def hello_world():
    print('Hello, World!!!')

hello_world()
EOL

# Zeige den Inhalt der Datei code.py an
cat code.py

# Fuehre den Python-Code in der Datei code.py aus
python code.py
```

Wir fuegen die Aenderungen zu unserem Branch hinzu und kommentieren sie entsprechend.

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Zeige alle Branches im Repository an, markiere den aktuellen Branch mit einem
↳Stern (*)
git branch

# Zeige den aktuellen Status des Git-Repositories an
git status

# Fuege die Datei code.py zur Staging-Area von Git hinzu
git add code.py

# Fuehre einen Commit durch mit der Nachricht "add hello world"
git commit -m "add hello world"
```

Wir wechseln dann wieder zu dem Hauptbranch

```
[ ]: %%bash
# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Wechsle zum Branch "main"
git checkout main
```

```
# Zeige alle Branches im Repository an, markiere den aktuellen Branch mit einem ↪ Stern (*)
git branch
```

Wir fuehren dann die Branches zusammen.

```
[ ]: %%bash

# Wechsle in das Verzeichnis /Users/christopherchandler/test_repo
cd /Users/christopherchandler/test_repo

# Fuehre den Merge des Branches "testing" in den aktuellen Branch durch
git merge testing

# Zeige den aktuellen Status des Git-Repositories an
git status
```

Wir haben also die Branches Main und Testing zusammengefuehrt. Bevor man Branches zusammenfuehrt oder wechselt, gibt es jedoch Einiges zu beachten. - Beachten dabei, dass Git das Wechseln zu einem anderen Branch blockiert, falls dein Arbeitsverzeichnis oder dein Staging-Bereich nicht committete Modifikationen enthaelt, die Konflikte verursachen. Generell ist es am besten, einen sauberen Zustand des Arbeitsbereichs anzustreben, bevor du Branches wechselst. - Wenn du die Branches wechselst, setzt Git dein Arbeitsverzeichnis zurueck, um so auszusehen, wie es das letzte Mal war, als du in den Branch committed hast. Dateien werden automatisch hinzugefuegt, entfernt und veraendert, um sicherzustellen, dass deine Arbeitskopie auf demselben Stand ist wie zum Zeitpunkt deines letzten Commits auf diesem Branch.

6 3.5 Merge Fehler

Merge-Konfliktfehler sind ein haeufiges Problem in der Softwareentwicklung, insbesondere bei der Zusammenarbeit in Teams, die Versionskontrollsysteme wie Git verwenden. Diese Fehler treten auf, wenn Aenderungen an Dateien, die in verschiedenen Branches vorgenommen wurden, miteinander in Konflikt stehen und das System nicht automatisch entscheiden kann, welche Aenderungen uebernommen werden sollen.

Hier ist ein einfaches Merge-Fehler-Beispiel und eine entsprechende Loesung.

Wir erstellen ein neues Git-Repo mit einer Datei und commiten sie.

```
[ ]: %%bash

# Verzeichnis entfernen, falls es existiert
rm -rf /Users/christopherchandler/merge_conflict_example

# Verzeichnis erstellen
mkdir /Users/christopherchandler/merge_conflict_example

# Zum neuen Verzeichnis wechseln
```

```

cd /Users/christopherchandler/merge_conflict_example

# Neues Git-Repository initialisieren
git init

# Eine Datei mit einer Begrüßungsnachricht erstellen
echo "Hello world" > greeting.txt

# Die Datei dem Staging-Bereich hinzufügen
git add greeting.txt

# Die Datei mit einer Nachricht im Repository committen
git commit -m "Initial commit with greeting"

```

Wir erstellen dann einen ganz neuen Branch und ändern irgendwas an dieser Datei. Wir committen diese Änderung ebenfalls.

```

[ ]: %%bash
# Change directory to where the Git repository is located
cd /Users/christopherchandler/merge_conflict_example

# Create a new branch and switch to it
git checkout -b feature_branch

# Make a change in the greeting.txt file in feature_branch
echo "Hello, feature branch!" > greeting.txt

# Add the modified file to the staging area
git add greeting.txt

# Commit the change in feature_branch
git commit -m "Update greeting in feature_branch"

```

Wir wechseln zurück zu dem Hauptbranch und nehmen Änderung vor. Wir committen sie.

```

[ ]: %%bash
# Change directory to where the Git repository is located
cd /Users/christopherchandler/merge_conflict_example

# Switch to the main branch
git checkout main

# Make a change in the greeting.txt file in the main branch
echo "Hello, main branch!" > greeting.txt

# Add the modified file to the staging area
git add greeting.txt

```

```
# Commit the change in the main branch
git commit -m "Update greeting in main branch"
```

Ein Merge fehler tritt auf, weil die Dateien zu stark von einander abweichen.

```
[ ]: %%bash
# Ins Verzeichnis wechseln, wo sich das Git-Repository befindet
cd /Users/christopherchandler/merge_conflict_example

# Merge des feature_branch in den aktuellen Branch durchfuehren
git merge feature_branch
```

Wir zeigen den Fehler in der Console an.

```
[ ]: %%bash
# Ins Verzeichnis wechseln, wo sich die Datei greeting.txt befindet
cd /Users/christopherchandler/merge_conflict_example

# Inhalt der Datei greeting.txt anzeigen
cat greeting.txt
```

Wir entscheiden uns fuer eine Variante und committen diese Aenderung.

```
[ ]: %%bash
# Ins Verzeichnis des Git-Repositories wechseln
cd /Users/christopherchandler/merge_conflict_example

# Den kombinierten Gruss "Hello, main branch and feature branch!" an die Datei_
↪greeting.txt anhaengen
echo "Hello, main branch and feature branch!" >> greeting.txt

# Die geaenderte Datei dem Staging-Bereich hinzufuegen
git add greeting.txt

# Den Merge-Konflikt in greeting.txt aufloesen und die Aenderungen commiten
git commit -m "Merge-Konflikt in greeting.txt aufgeloeset"
```

Git status ueberpruefen

```
[ ]: %%bash
# Ins Verzeichnis des Git-Repositories wechseln
cd /Users/christopherchandler/merge_conflict_example

# Den Status des Git-Repositorys pruefen
git status
```

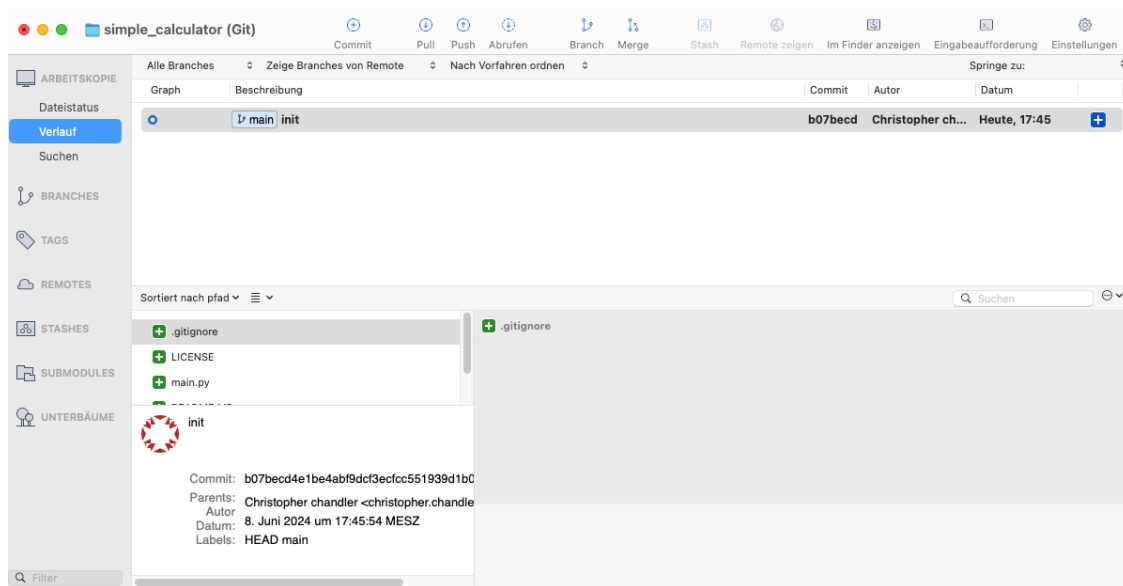
7 Teil 4 Git GUI Software

Bis jetzt haben wir uns nur mit der Kommandozeile befasst. Das ist ganz nuetzlich, denn man weiss genau, wie die Sachen im Hintergrund funktionieren. Fuer die Allermeisten ist aber sowas muehsam und umstaendlich einzusetzen. Zum Glueck muss man sich nicht auf sowas beschraenken, weil es Programme und Schnittstellen gibt, die uns eine grafische Oberflaeche bieten.

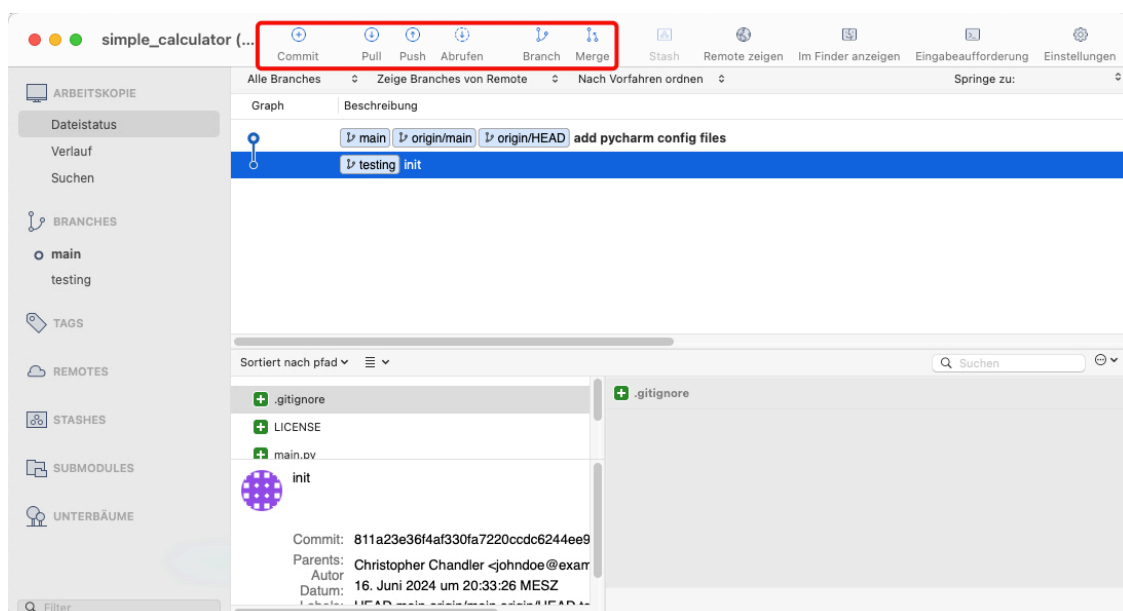
Wir beschaeftigen uns aber nur mit ein paar davon: - Source Tree - Git in Pycharm - Github

7.1 4.1 Source Tree

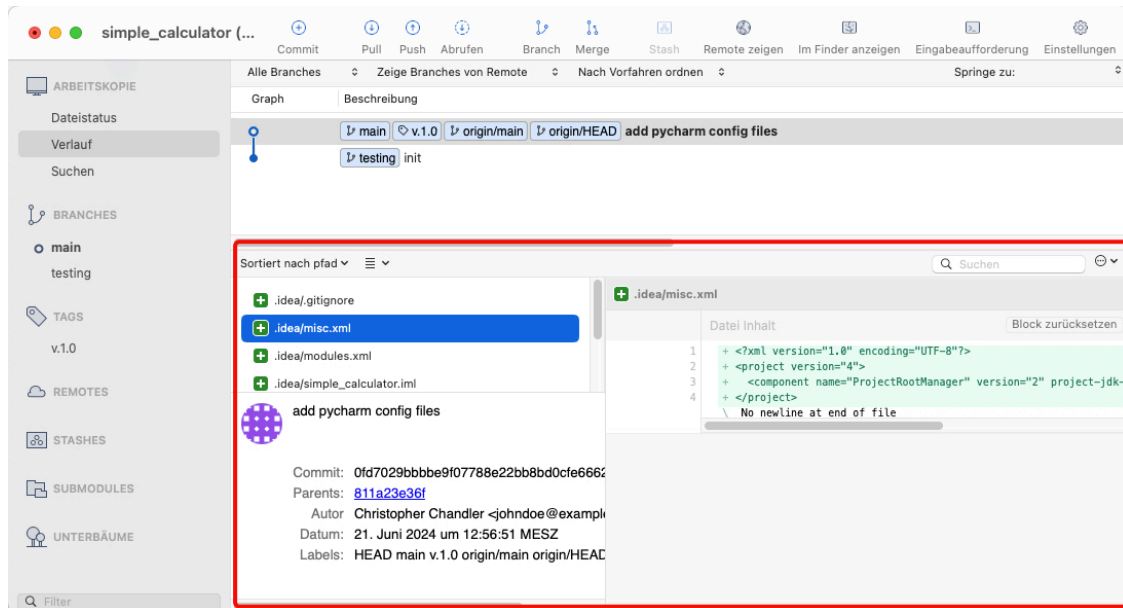
Mit Source Tree haben wir alles auf einen Blick.



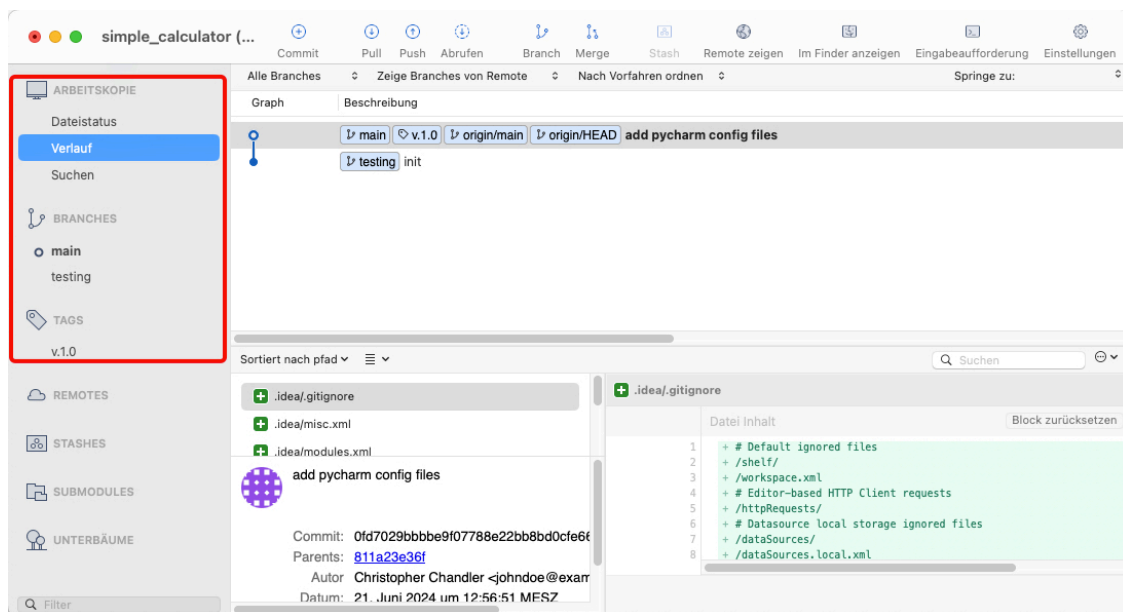
Wir haben die Menueeeisten mit den Befehlen, die wir schon aus Git kennen.



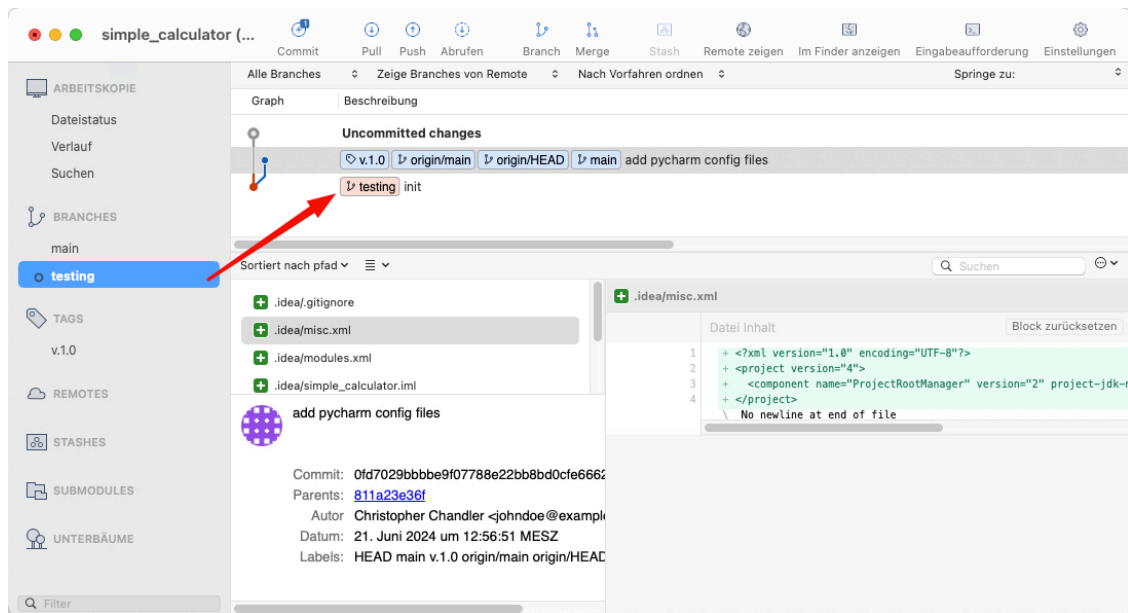
Die Kommentare werden auch auch angezeigt.



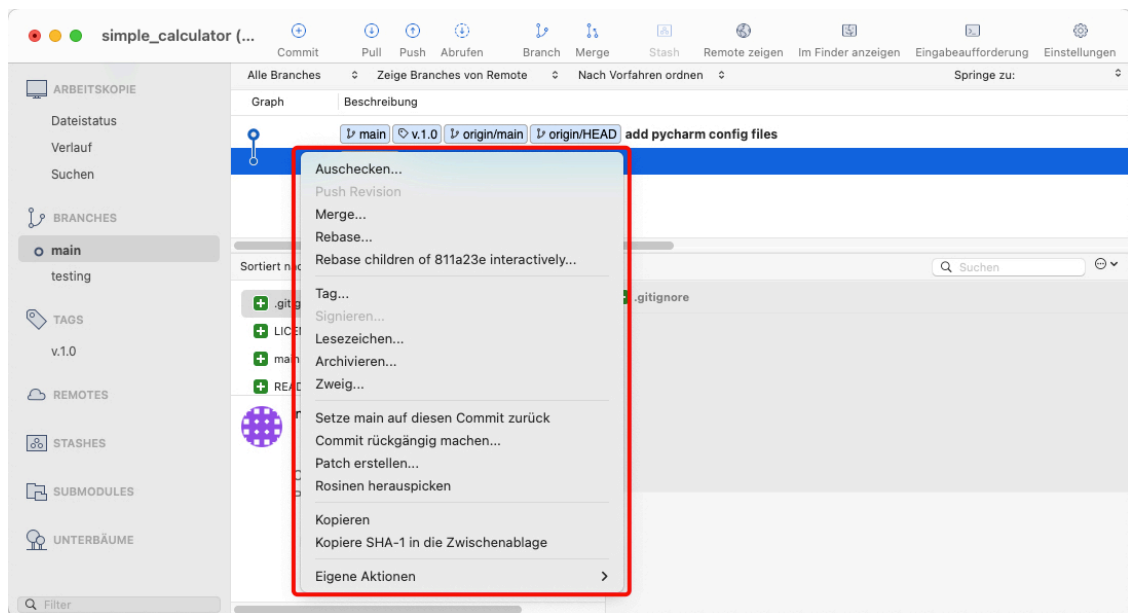
In der Seitenleiste sieht man auch die Branches, Tags und den Verlauf



Wir koennen auch die Branches anzeigen lassen.

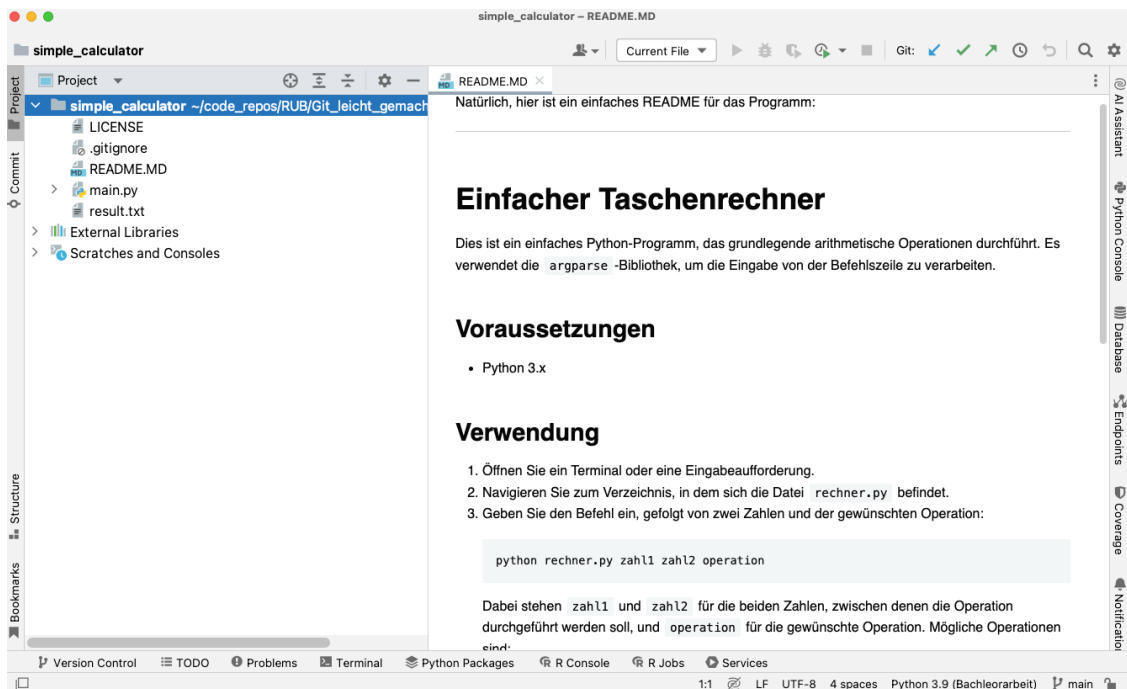


In dem Kontextmenue kann man bestimmte branches auschecken, mergen etc.

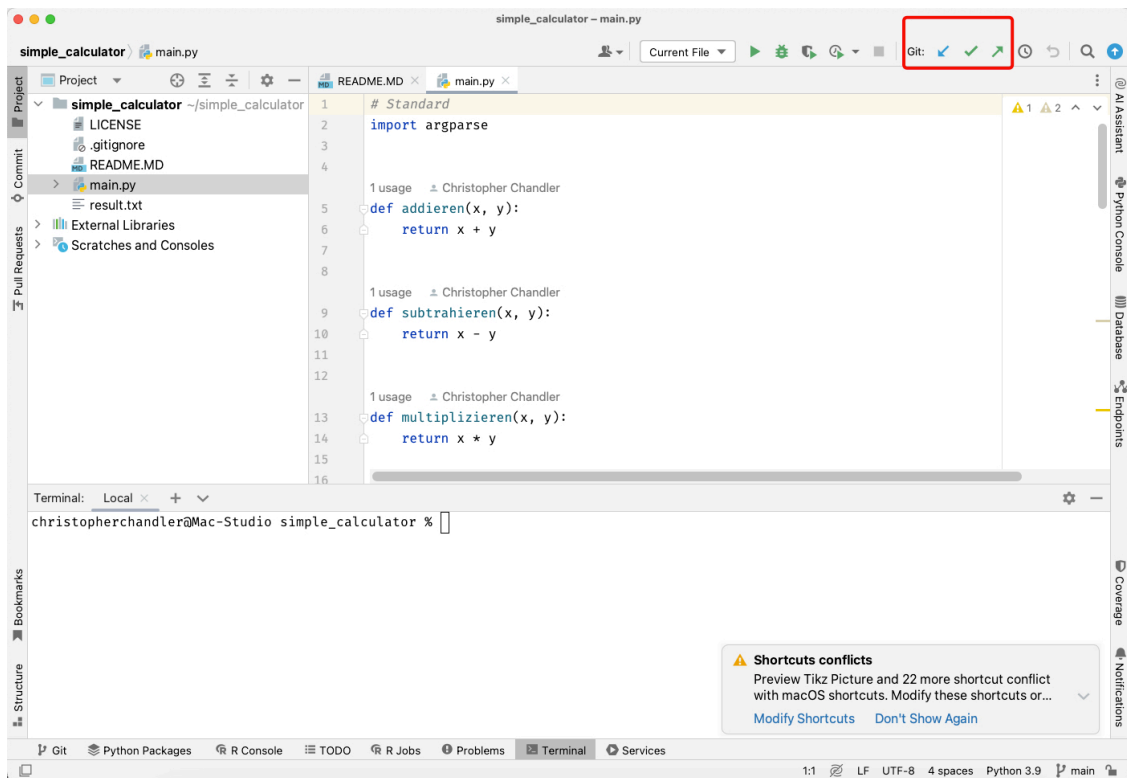


8 4.2 GIT in Pycharm

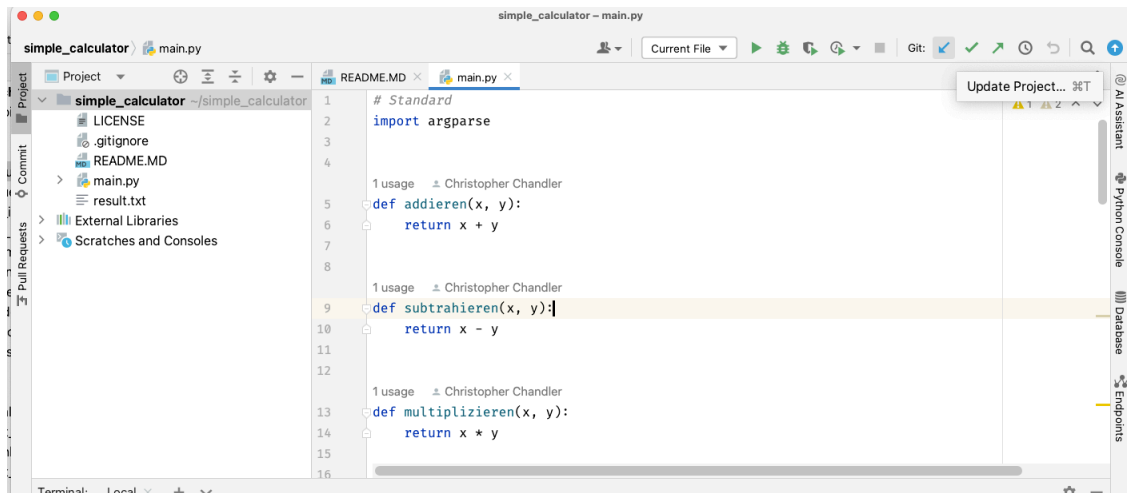
Wenn man den Code direkt in der IDE bearbeiten und mit Git verwalten moechte, geht das auch



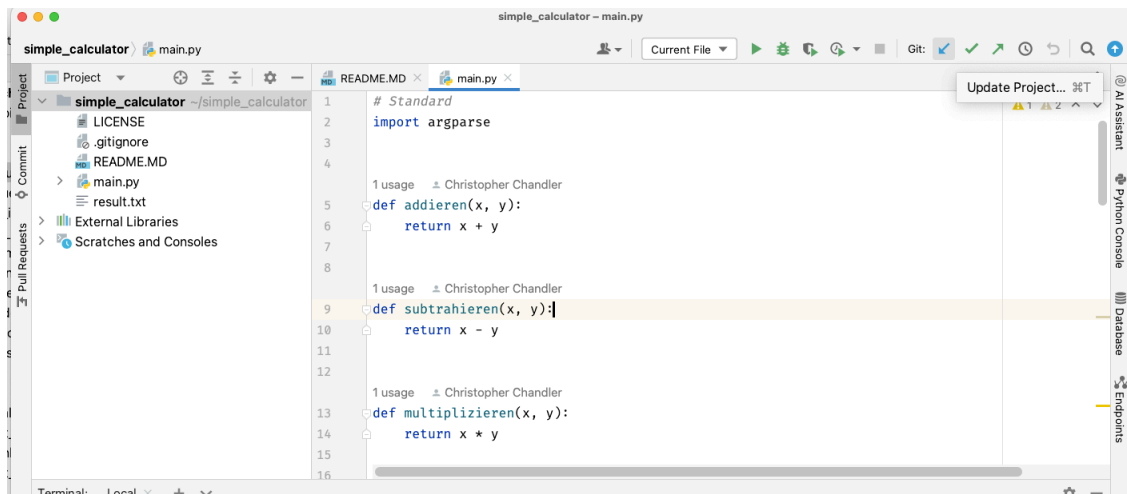
Auch hier gibt es eine Menueleiste



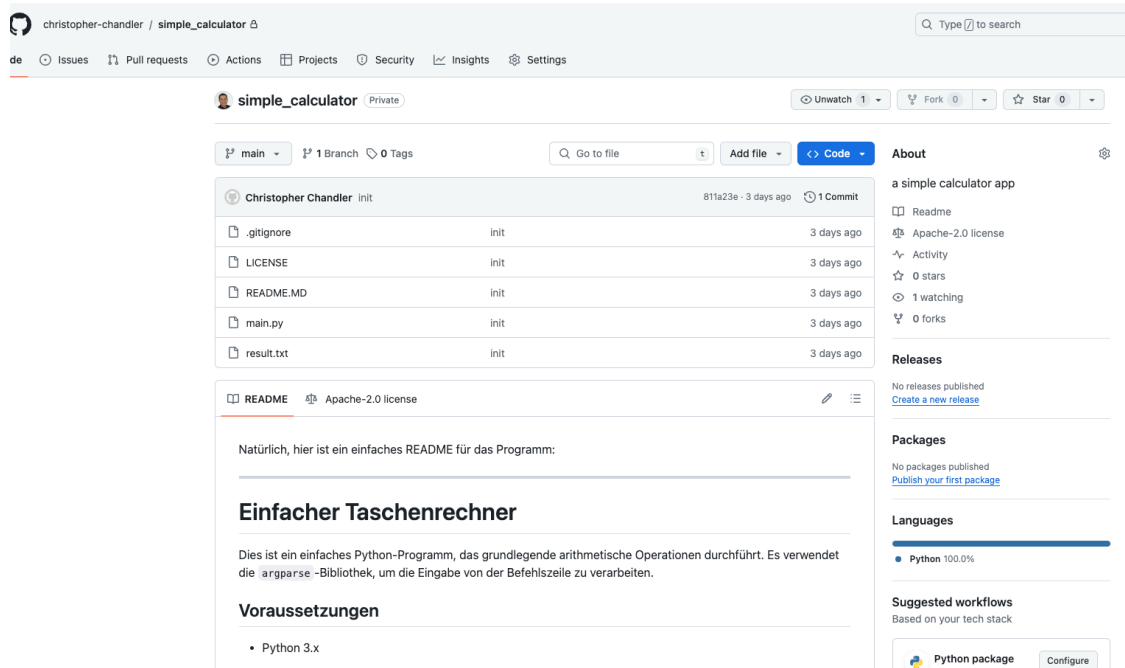
Man kann den Code von Git hub pullen



Man kann den Code nach Github pushen



8.1 4.3 Github



9 Quellen

Arek. (2020, August 15). Git Lernen in 30 Minuten - Anfänger Tutorial (2022). <https://lerneprogrammieren.de/git/>

Chacon, S., & Long, J. (n.d.). Book. Git. <https://git-scm.com/book/de/v2>

Rhoenerlebnis. (n.d.). Git cheat sheet. https://rhoenerlebnis.de/_upl/de/_pdf-seite/git_cheatsheet_de_white.pdf

Squirrels, J. (2023, July 21). Erste Schritte mit git: Eine Umfassende Anleitung fuer neulinge. CodeGym. <https://codegym.cc/de/groups/posts/de.379.erste-schritte-mit-git-eine-umfassende-anleitung-fur-neulinge>