

main_git_leicht_gemacht

June 21, 2024

1 Git Leicht Gemacht

Kurs Methoden der Linguistik (050004-SoSe 2024) Referent Christopher Chandler

Dieses Jupyter-Notebook “Git leicht gemacht” bietet eine Einführung in Git, ein Versionskontrollsystem. Es erklärt 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 zusammenführt (Merging). Außerdem werden grafische Benutzeroberflächen 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. Abschließend werden die wichtigsten Punkte zusammengefasst und weiterführende Quellen angegeben.

Inhaltsverzeichnis

Teil 1: Einführung

1.1 Was ist Versionsverwaltung?

1.1.1 Lokale Versionsverwaltung

1.1.2 Zentrale Versionsverwaltung

1.2 Kurzer Überblick über die Historie von Git

1.3 Was ist Git?

1.3.1 Snapshots statt Unterschiede

1.3.2 Fast jede Funktion arbeitet lokal

1.3.3 Git stellt Integrität sicher

1.3.4 Git fügt im Regelfall nur Daten hinzu

1.3.5 Die drei Zustände

1.4 Die Kommandozeile

1.5 Git installieren

1.5.1 Mac

1.5.2 Linux

1.5.3 Windows

1.5.4 Installation überprüfen

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 Änderungen nach

2.2.1 Zustand

2.2.1 Zustand von Dateien prüfen

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 Überprüfen o

2.2.7 Die Änderungen commiten

2.3 Anzeigen der Commit-Historie

2.4 Ungewollte Änderungen rück

2.4.1 Eine Datei aus der S

2.4.2 Änderung

2.6 Taggen

2.6.1 Annotierte Tags

2.6.2 Tags löschen

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

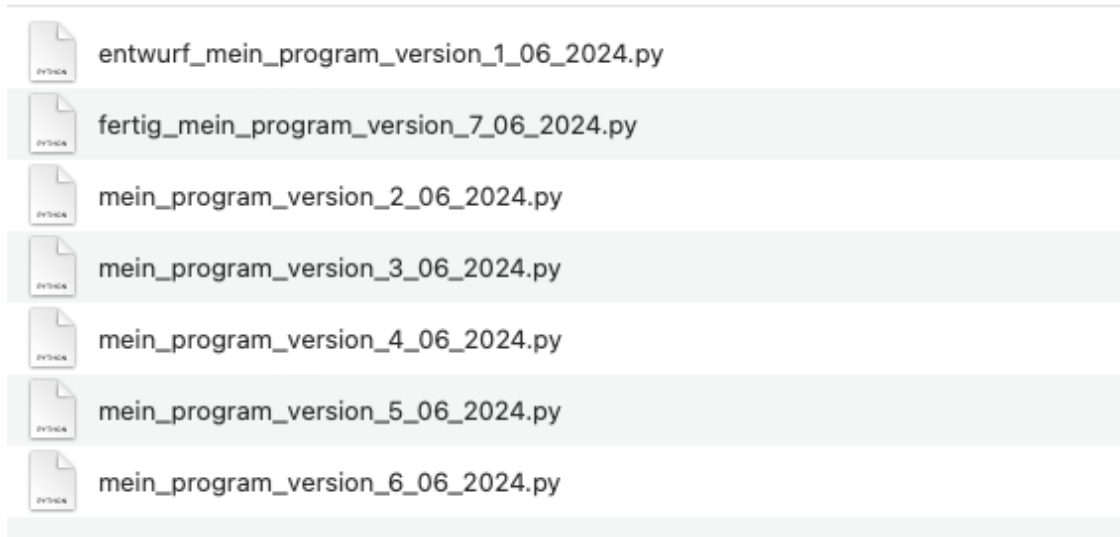
Quellen

2 1. Einfuehrung

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

Damit das Program nicht verloren geht oder man irgendwas aus Versehen überschreibt, 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 Kreativität sehr schnell unübersichtlich 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 21-8-2016	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
- projecttttttt
- aaaaaaaaaaaaaaaaaaaaaaaaaa...



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

Damit die Änderungen an einer Datei gespeichert und in der Zukunft nachvollzogen werden können, 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 dieser Sitzung klarer sein.

2.1 1.1 Was ist Versionsverwaltung?

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

Normalerweise wenn man eine Datei speichert, wird der letzte Stand der Datei überschrieben. Das bedeutet, dass man nicht mehr auf ältere Versionen davon nicht mehr zurückgreifen kann. Noch schlimmer ist es, wenn man ausversehen mehrere Dateien in einem Verzeichnis löscht. Doch mit einer Versionsverwaltungssoftware kann man solche Änderungen und Löschungen rückgängig machen.

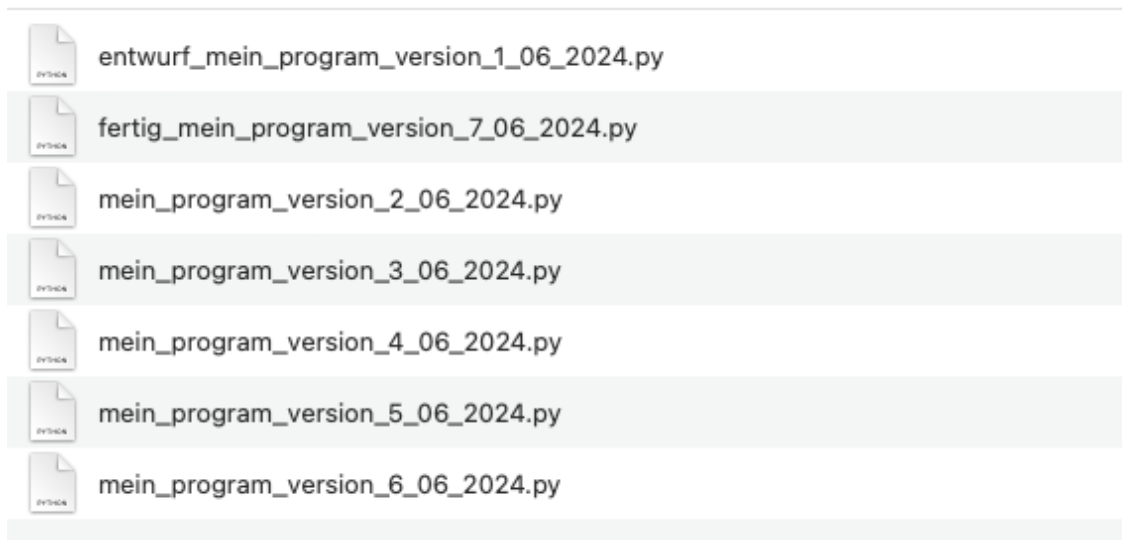
Vereinfacht gesagt, eine Versionsverwaltung ist ein System, welches die Änderungen an einer oder einer Reihe von Dateien über die Zeit hinweg protokolliert, sodass man später auf eine bestimmte Version zurückgreifen kann.

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

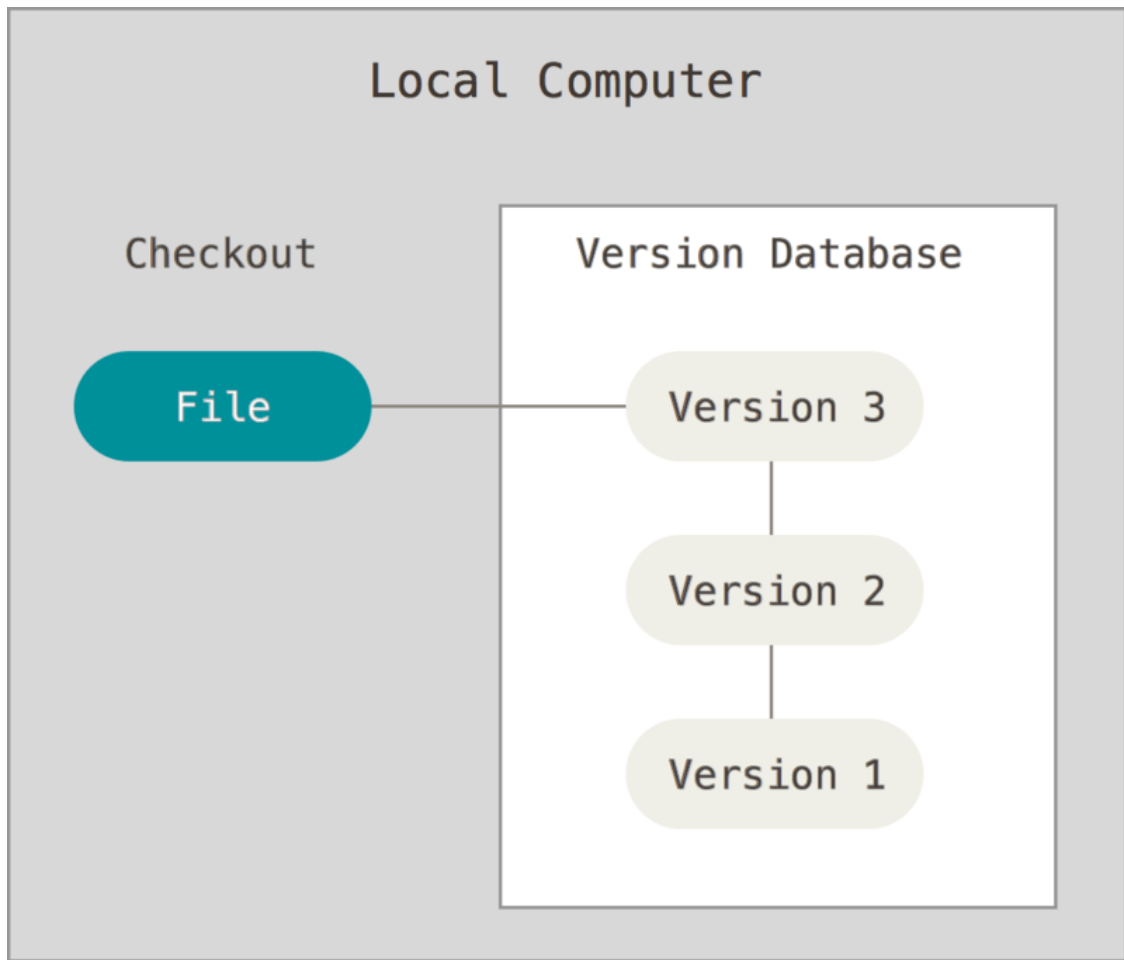
(Es gibt verteilte Versionsverwaltung, aber das ist für diese Präsentation nicht von zentraler Bedeutung)

2.1.1 1.1.1 Lokale Versionsverwaltung

Wir nehmen das Bild von dem letzten Kapitel als Beispiel. Viele speichern dasselbe Programm mehrfach ab. Das Problem bei sowas ist, dass es schnell schnell unübersichtlich viel und evtl. verliert man den Überblick. Es kann auch natürlich vorkommen, dass man ausversehen 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.



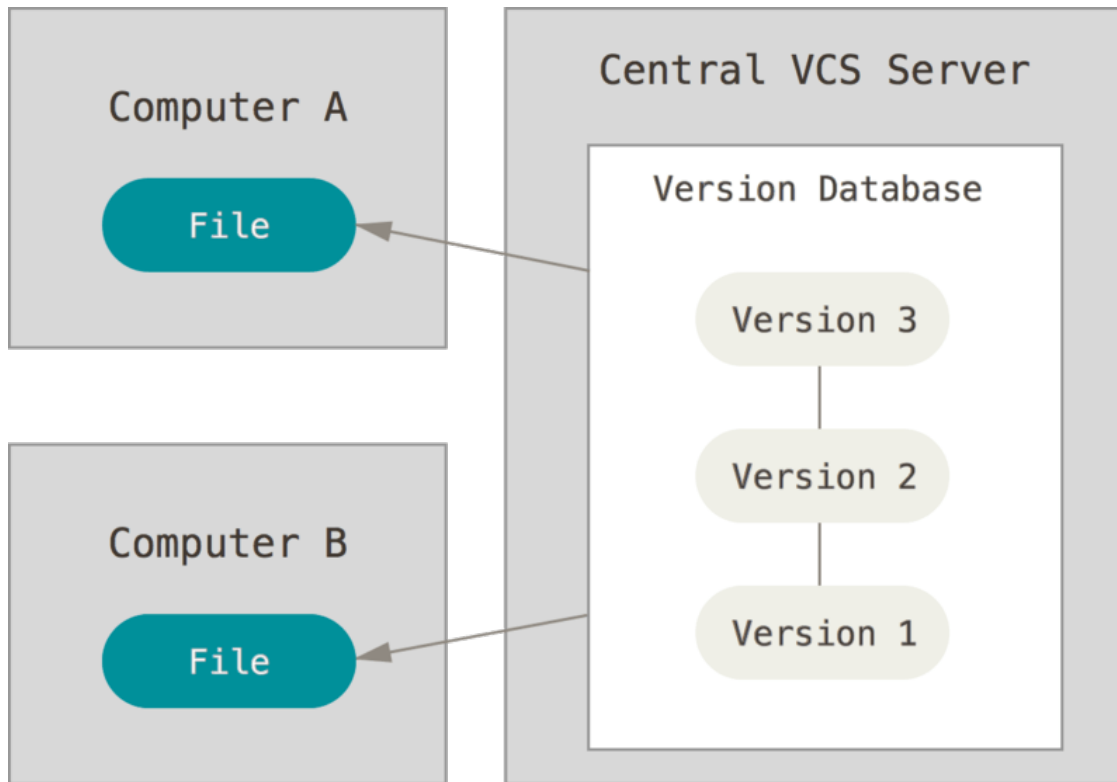
Lokale

Versionsverwaltung

2.1.2 1.1.2 Zentrale Versionsverwaltung

Bei der zentralen Versionsverwaltung geht es darum, die Zusammenarbeit mit anderen Programmieren zu erleichtern. Statt Dateien hin- und her zu schicken, wird ein zentrales Verzeichnis bzw. 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)



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.
- Frühe Entwicklungsjahre (1991-2002): Änderungen wurden als Patches und archivierte Dateien ausgetauscht.
- 2002: Umstieg auf proprietäres DVCS Bitkeeper.
- 2005: Beziehung zwischen Linux-Community und BitKeeper-Unternehmen zerbrach, kostenlose Nutzung von BitKeeper wurde widerrufen.
- Auslöser für 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 für große Projekte und hat ein exzellentes Branching-Konzept für nicht-lineare Entwicklung.

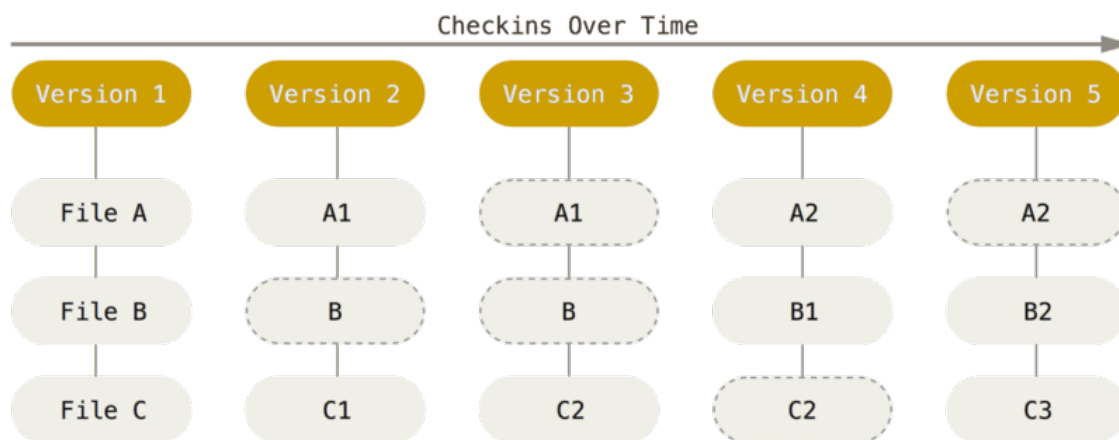
2.3 1.3 Was ist Git?

Git ist eine Sammlung von Dienstprogrammen in der Kommandozeile, die Änderungen in Dateien verfolgen und aufzeichnen (meistens Quellcode, aber du kannst alle möglichen 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 abhängig ist, um alte Versionen deiner Dateien aufzubewahren. Stattdessen funktioniert es vollständig 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 ab (wie Schnappschüsse), im Gegensatz zu anderen System, die lediglich die Differenzen zu der ursprünglichen Hauptdatei abgespeichert. Deswegen können die Änderungen als eine statt **Stapel von Schnappschüssen**



Speichern der Daten-Historie eines Projekts in Form von Schnappschüsse

2.3.2 1.3.2 Fast jede Funktion arbeitet lokal

Die meisten Aktionen in Git benötigen nur lokale Dateien und Ressourcen, um ausgeführt zu werden – im Allgemeinen werden keine Informationen von einem anderen Computer in deinem Netzwerk benötigt. Die allermeisten Operationen können nahezu ohne jede Verzögerung ausgeführt werden, da die vollständige Historie eines Projekts bereits auf dem jeweiligen Rechner verfügbar ist.

Da git immer lokal verfügbar ist, gibt es viele Vorteile: - Git durchsucht die Projekt-Historie lokal, ohne externe Server. - Vollständige Projekthistorie ist sofort verfügbar. - Änderungen einer Datei von vor einem Monat können lokal verglichen werden. - Kein externer Server nötig für Datei-Vergleich oder Historienabfrage. - Offline-Arbeit möglich, z.B. im Flugzeug oder Zug. - Änderungen können später bei Netzwerkverbindung hochgeladen werden. - Arbeit unabhängig von VPN-Verfügbarkeit möglich.

2.3.3 1.3.3 Git stellt Integrität sicher

Von allen zu speichernden Daten berechnet Git Prüfsummen (engl. **checksum**) und speichert diese als Referenz zusammen mit den Daten ab. Das macht es unmöglich, dass sich Inhalte von Dateien oder Verzeichnissen ändern, ohne dass Git das mitbekommt. Git basiert auf dieser Funktionalität

und sie ist ein integraler Teil der Git-Philosophie. Man kann Informationen deshalb z.B. nicht während der Übermittlung verlieren oder unwissentlich beschädigte Dateien verwenden, ohne dass Git in der Lage wäre, dies festzustellen.

2.3.4 1.3.4 Git fügt im Regelfall nur Daten hinzu

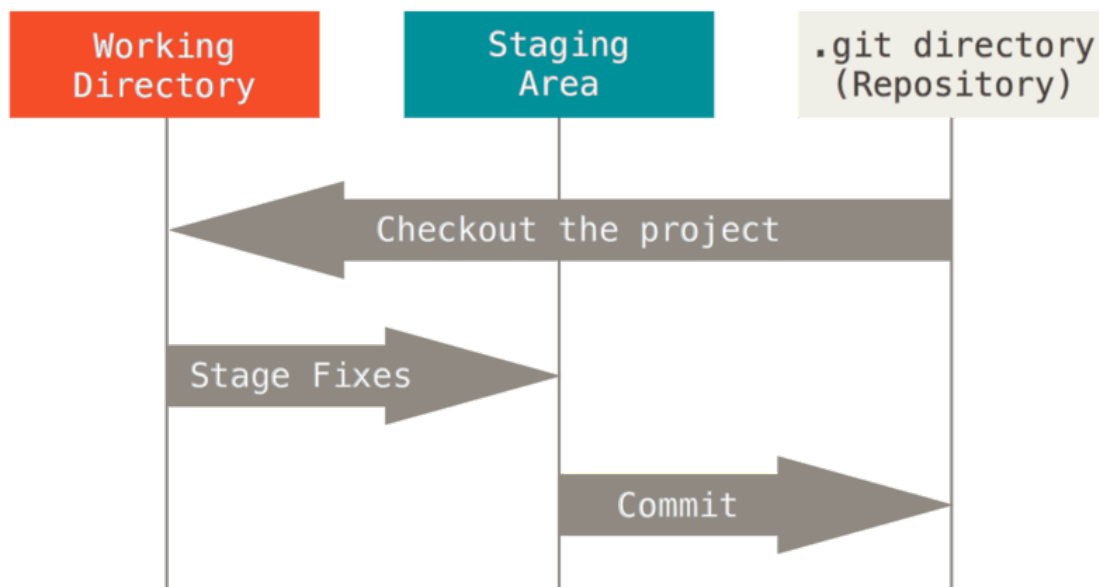
Wenn du Aktionen in Git durchführen willst, werden diese fast immer nur Daten zur Git-Datenbank **hinzufügen**. Deshalb ist es sehr schwer, das System dazu zu bewegen, irgendetwas zu tun, das nicht wieder rückgängig zu machen ist, oder dazu, Daten in irgendeiner Form zu löschen. Unter anderem deshalb macht es so viel Spaß mit Git zu arbeiten. Man weiß genau, man kann ein wenig experimentieren, ohne befürchten zu müssen, irgendetwas zu zerstören oder durcheinander zu bringen.

2.3.5 1.3.5 Die drei Zustände

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

- **Modified** bedeutet, dass eine Datei geändert, aber noch nicht in die lokale Datenbank eingecheckt wurde.
- **Staged** bedeutet, dass eine geänderte Datei in ihrem gegenwärtigen Zustand für den nächsten Commit vorgemerkt ist.
- **Committed** bedeutet, dass die Daten sicher in der lokalen Datenbank gespeichert sind.

Das führt 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 ändern kannst.

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

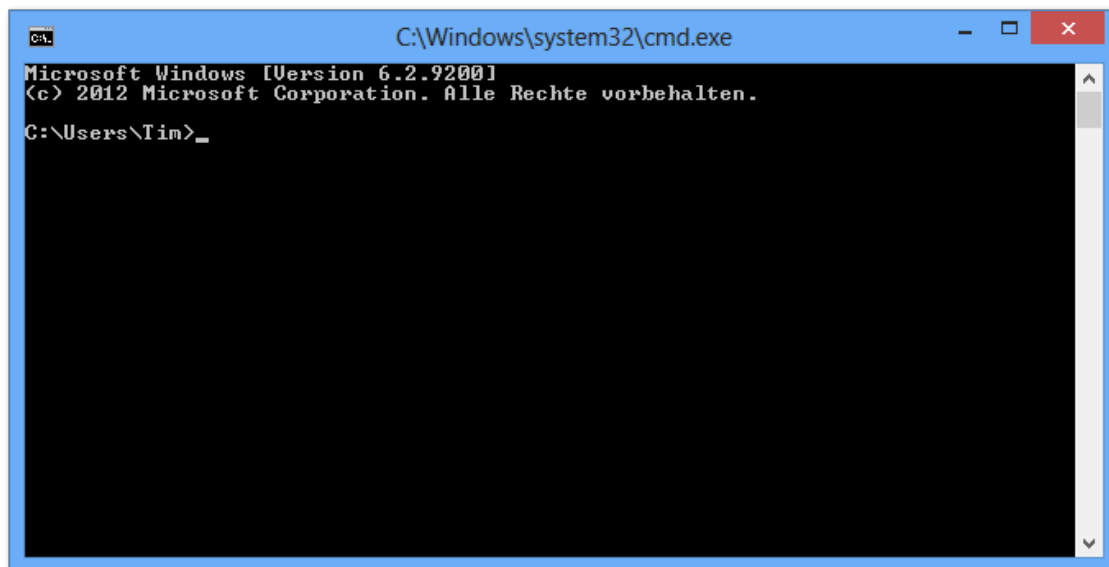
Im Git-Verzeichnis werden die Metadaten und die Objektdatenbank für dein Projekt gespeichert. Das ist der wichtigste Teil von Git, dieser Teil wird kopiert, wenn man ein Repository von einem anderen Rechner **klont**.

Der grundlegende Git-Arbeitsablauf sieht in etwa so aus:

1. Du änderst Dateien in deinem Verzeichnisbaum.
2. Du stellst selektiv Änderungen bereit, die du bei deinem nächsten Commit berücksichtigen möchtest, wodurch nur diese Änderungen in den Staging-Bereich aufgenommen werden.
3. Du führst einen Commit aus, der die Dateien so übernimmt, 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 geändert und in die Staging-Area hinzugefügt wurde, gilt sie als für den Commit **vorgemerkt** (engl. **staged**). Und wenn sie geändert, aber noch nicht zur Staging-Area hinzugefügt wurde, gilt sie als geändert (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.

2.5 1.5 Git installieren

Um die Git-Software benutzen zu können, muss sie erstmal installiert sein.

2.5.1 1.5.1 Mac

Wenn du schon Xcode heruntergeladen hast, wurde git automatisch mit installiert. Falls das nicht der Fall sein soll, bitte dann 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`

2.5.2 1.5.2 Linux

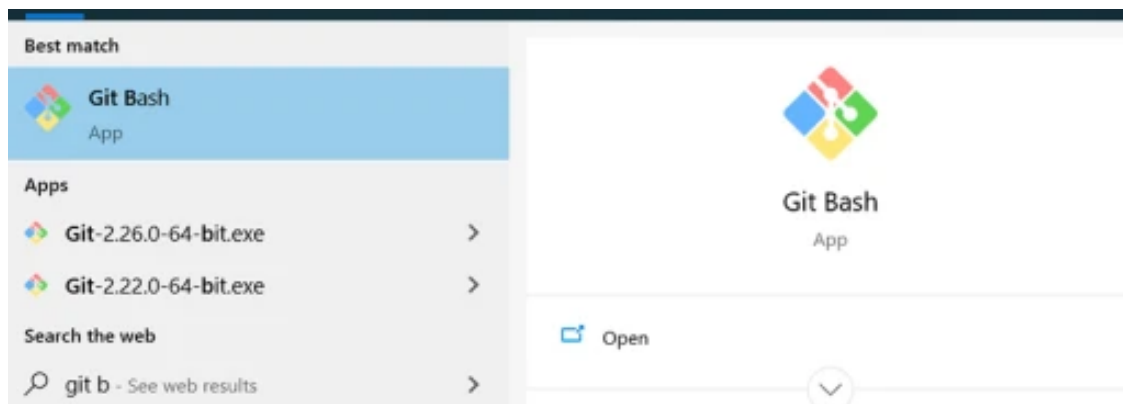
Normalerweise ist Git Teil von Linux-Distributionen und bereits installiert, da es sich um ein Tool handelt, das ursprünglich für die Linux-Kernel-Entwicklung geschrieben wurde. Aber es gibt Situationen, in denen das nicht der Fall ist. Wenn das nicht der Fall sein soll, muss man einen der beiden Befehle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

- `sudo dnf install git-all` oder
- `sudo apt install git-all`

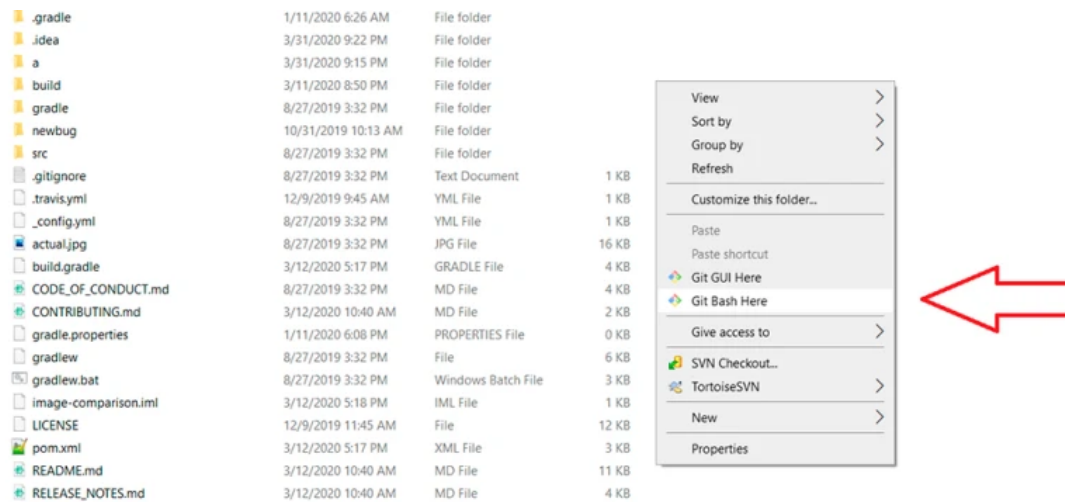
2.5.3 1.5.3 Windows

Auf Windows Git zu installieren ist es ein bisschen anders, denn man muss eine externe Software herunterladen und installieren. Wie üblich muss man eine exe-Datei herunterladen und ausführen. Hier ist alles aber ganz einfach: Klick auf den folgenden [Link](#), führe die Installation durch und fertig. Dazu verwenden wir die von Windows bereitgestellte Bash-Konsole. Unter Windows muss man Git Bash ausführen. So sieht es im Startmenü aus:

Nachdem die .exe heruntergeladen wurde, soll sie nun ausgeführt werden. Die restlichen Schritte werden bei der Installation vom Installationsprogramm erklärt.



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



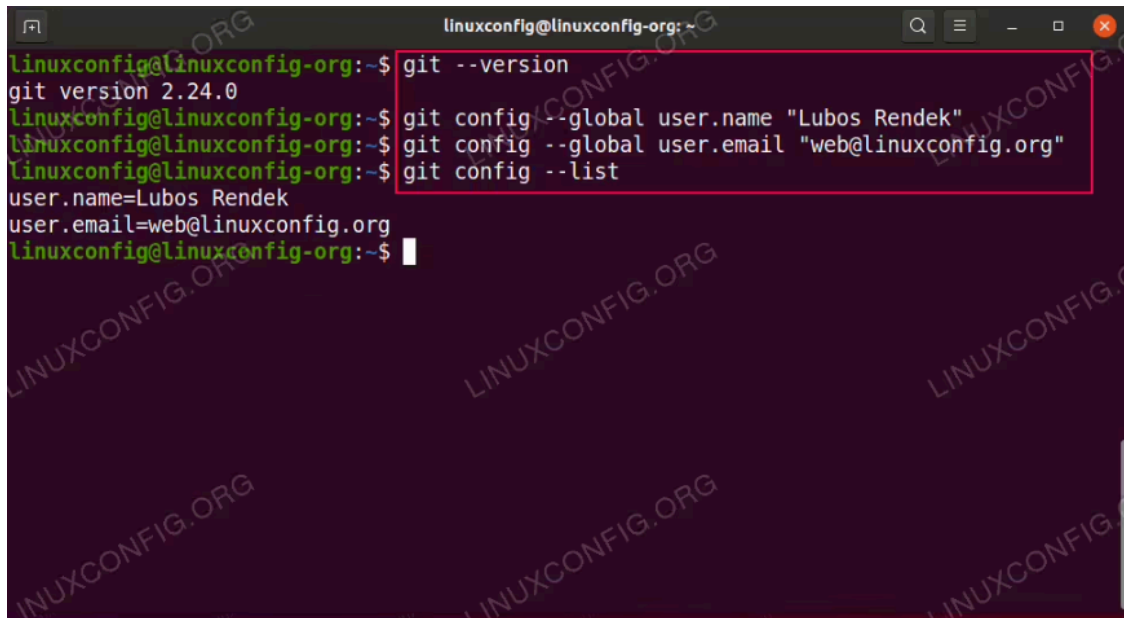
Falls das nicht funktioniert hat, bitte alternativ oder ergänzend dazu die folgende Anleitung benutzen. - <https://www.youtube.com/watch?v=0PWEG6D2MVQ>

2.5.4 1.5.4 Installation ueberpruefen

Um zu überprüfen, dass 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.

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

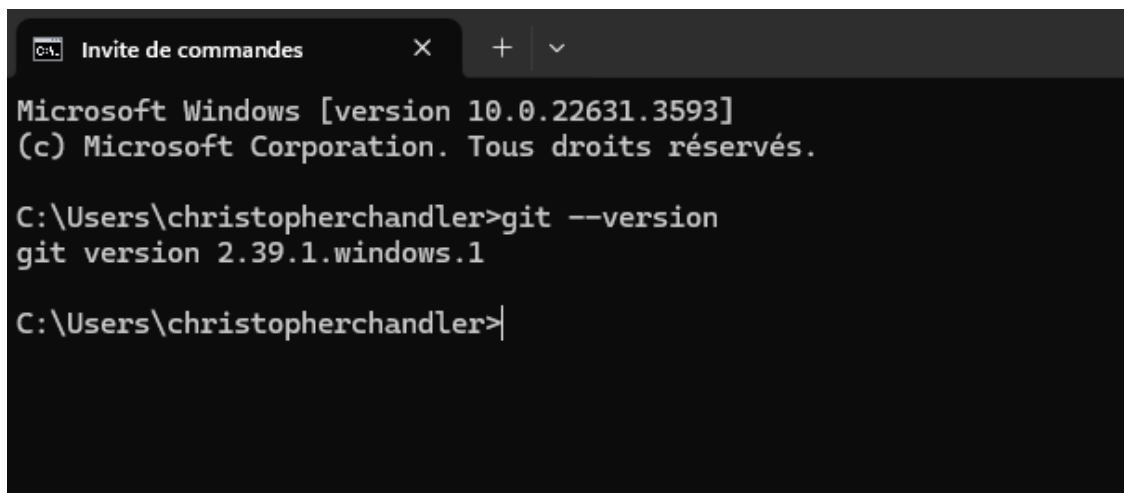
1.5.5 Mac Version

A terminal window titled 'linuxconfig@linuxconfig-org: ~' with a search bar and window controls. The terminal shows the following commands and output:

```
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:~$
```

A red rectangular box highlights the three configuration commands and their output.

1.5.6 Linux Version

A Windows Command Prompt window titled 'Invite de commandes' with a close button, a plus sign, and a dropdown arrow. The prompt shows the following text:

```
Microsoft Windows [version 10.0.22631.3593]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\christopherchandler>git --version
git version 2.39.1.windows.1

C:\Users\christopherchandler>
```

1.5.7 Windows Version

2.6 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 Git-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 können ein lokales Repository mit `git init` anlegen.

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

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 für bessere Lesbarkeit in der Ausgabe

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

```
[ ]: %%%bash

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

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

# Führe 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 möchtest – 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

# Klonen des GitHub-Repositorys 'Hello-World' von Octocat (Beispielrepository) /
  ↳ Zielordener wird mit angegeben.
git clone https://github.com/octocat/Hello-World hello_world_repo
```

3.1.3 2.1.3 Kollaboration

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

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

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

[main eaf2bb2] add calculator.py
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 calculator.py

To github.com:christopher-chandler/simple_calculator.git
0fd7029..eaf2bb2 main -> main
```

Pull: Neueste Änderungen vom Remote-Repository abrufen Mit dem Befehl `git pull` holen Sie die neuesten Änderungen 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.

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

From github.com:christopher-chandler/simple_calculator
eaf2bb2..09689a8 main -> origin/main

Updating eaf2bb2..09689a8
Fast-forward
```

```
calculator.py | 0
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 calculator.py
```

Dieser Befehl holt die neuesten Änderungen 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` ausführst. Dies kannst du mit dem Befehl `git fetch` tun, der die neuesten Änderungen vom Remote-Repository herunterlädt, 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 für die Zusammenarbeit in Git-Projekten zu gewährleisten. Durch regelmäßiges Pushen und Pullen deiner Änderungen kannst du sicherstellen, dass alle Teammitglieder auf dem gleichen Stand sind und Konflikte vermieden werden.

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

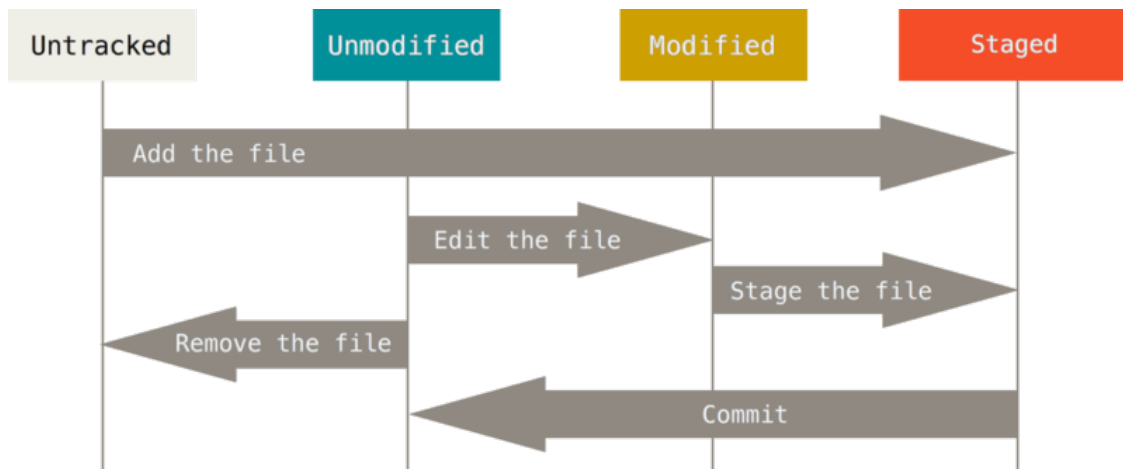
3.2 2.2 Änderungen nachverfolgen und im Repository speichern

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, Änderungen vorzunehmen und Schnappschüsse dieser Änderungen in dein Repository zu committen, wenn das Projekt so weit fortgeschritten ist, dass du es sichern möchtest.

Denke daran, dass sich jede Datei in deinem Arbeitsverzeichnis in einem von zwei Zuständen befinden kann: **tracked** oder **untracked** – Änderungen 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 können entweder unverändert, modifiziert oder für den nächsten 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 unverändert. Nach dem Klonen wurden sie ja ausgecheckt und bis dahin hast du auch noch nichts an ihnen verändert.

Sobald du anfängst versionierte Dateien zu bearbeiten, erkennt Git diese als modifiziert, weil sie sich im Vergleich zum letzten Commit verändert haben. Die geänderten Dateien kannst du dann für den nächsten Commit vormerken und schließlich alle Änderungen, die sich in der Staging-Area befinden, einchecken (engl. committen). Danach wiederholt sich dieser Vorgang.



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

```
[ ]: %%bash

# Wechsle in das übergeordnete Verzeichnis, in dem wir arbeiten möchten
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
  ↳ überprüfen
git status
```

3.2.1 2.2.1 Zustand von Dateien prüfen

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

```
[ ]: %%bash

# Wechsle in das Verzeichnis, in dem die Befehle ausgeführt 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
  ↳ überprüfen
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 unerwünschte Dateien wie generierte Binärdateien nicht hinzuzufügen. Um Änderungen an der README zu verfolgen, müssen wir sie explizit zur Versionsverwaltung hinzufügen.

3.2.2 2.2.2 Neue Dateien zur Versionsverwaltung hinzufügen

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

```

[ ]: %%%bash

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

# Datei readme.me zum Index hinzufügen
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 Versionsverwaltung hinzuzufügen, Dateien für einen Commit vorzumerken und verschiedene andere Dinge – beispielsweise einen Konflikt aus einem Merge als aufgelöst zu kennzeichnen.

3.2.5 2.2.5 Ignorieren von Dateien

Häufig gibt es eine Reihe von Dateien, die Git nicht automatisch hinzufügen oder schon gar nicht als „nicht versioniert“ (eng. *untracked*) anzeigen soll. Dazu gehören in der Regel automatisch generierte Dateien, wie Log-Dateien oder Dateien, die von deinem Build-System erzeugt werden. In solchen Fällen kannst du die Datei `.gitignore` erstellen, die eine Liste mit Vergleichsmustern enthält. 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

# Füge 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

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

# Füge alle anderen geänderten 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 Änderungen 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 commiten

Bis jetzt haben wir die Änderungen hinzugefügt, aber nicht committet. Nachdem deine Staging-Area nun so eingerichtet ist, wie du es wünschst, kannst du deine Änderungen committen. Denke daran, dass alles, was noch nicht zum Commit vorgemerkt ist – alle Dateien, die du erstellt oder geändert hast und für die du seit deiner Bearbeitung nicht mehr `git add` ausgeführt 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

# Führe 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 zurückschauen wollen, um zu erfahren, was geschehen ist. Das wichtigste und mächtigste Werkzeug dafür ist der Befehl `git log`.

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

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

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

```
[ ]: %%%bash

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

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

Wenn du einige gekürzte Statistiken für jeden Commit sehen möchtest, 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 Änderungen rückgängig 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

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

# Führe 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

# Führe einen Commit durch, um die Commit-Nachricht zu ändern (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 geänderten Dateien an
git log --stat
```

3.5 2.4.1 Eine Datei aus der Staging-Area entfernen

Die nächsten beiden Abschnitte erläutern, wie du mit deiner Staging-Area und den Änderungen 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 Änderungen an ihnen rückgängig machen kannst. Nehmen wir zum Beispiel an, du hast zwei Dateien geändert und möchtest sie als zwei separate Änderungen übertragen, 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

# Füge 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 auführt, 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 zurück (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 Änderungen an der Datei nicht behalten willst? Wie kannst du sie in den Ursprungszustand zurücksetzen, so wie sie beim letzten Commit ausgesehen hat (oder anfänglich geklont wurde, oder wie auch immer du sie in dein Arbeitsverzeichnis bekommen hast)? Glücklicherweise 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

# Füge 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
↪zurück
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 Möglichkeit, bestimmte Punkte in der Historie eines Repositorys als wichtig zu markieren. Normalerweise verwenden Leute diese Funktionalität, 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
```

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

3.7.2 2.6.3 Tags auschecken

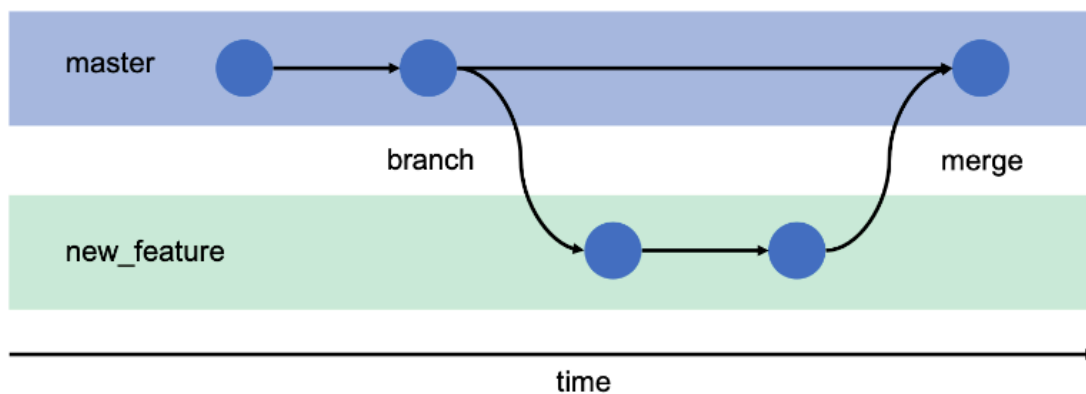
Tags sind hilfreich, da man sie benutzt kann, um bestimmte Zustände des Codes auszuchecken.

```
[ ]: %%%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 standardmäßig ↵
↵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 Neues 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) für 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) für den Branch "main" an
git log

```

4.2 3.4 Merging

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

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

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

- Du wechselst zu deinem Produktions-Branch
- Du erstellst einen Branch, um den Hotfix einzufügen
- Nachdem der Test abgeschlossen ist, mergst du den Hotfix-Branch und schiebst ihn in den Produktions-Branch

- Du wechselst zurück zu deiner ursprünglichen 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 zusammengeführt. Es kommt darauf an, was für einen Arbeitsablauf du hast, aber irgendwann müssen bzw. sollen diese Branches zusammengeführt werden. Dieses Vorgehen nennt sich mergen.

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

5 3.4.2 Testing Branch

In diesem Branch nehmen wir irgendwelche Änderungen 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 füge den Python-Code hinzu
> code.py # Erzeugt die Datei code.py (falls sie noch nicht existiert) oder
↳ löscht ihren Inhalt (falls sie existiert)
cat <<EOL >> code.py # Fügt 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

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

Wir fügen die Änderungen 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
```

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

# Führe 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 führen dann die Branches zusammen.

```
[ ]: %%bash

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

# Führe 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 zusammengeführt. Bevor man Branches zusammenführt 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 enthält, 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 zurück, um so auszusehen, wie es das letzte Mal war, als du in den Branch committed hast. Dateien werden automatisch hinzugefügt, entfernt und verändert, 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 häufiges Problem in der Softwareentwicklung, insbesondere bei der Zusammenarbeit in Teams, die Versionskontrollsysteme wie Git verwenden. Diese Fehler treten auf, wenn Änderungen an Dateien, die in verschiedenen Branches vorgenommen wurden, miteinander in Konflikt stehen und das System nicht automatisch entscheiden kann, welche Änderungen

übernommen werden sollen.

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

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 commiten
git commit -m "Initial commit with greeting"
```

Wir erstellen dann einen ganz neuen Branch und ändern irgendwas an dieser Datei. Wir commiten 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 durchführen
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 für eine Variante und committen diese Änderung.

```
[ ]: %%%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 anhängen
echo "Hello, main branch and feature branch!" >> greeting.txt

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

# Den Merge-Konflikt in greeting.txt auflösen und die Änderungen commiten
git commit -m "Merge-Konflikt in greeting.txt aufgelöst"
```

Git status überprüfen

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

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

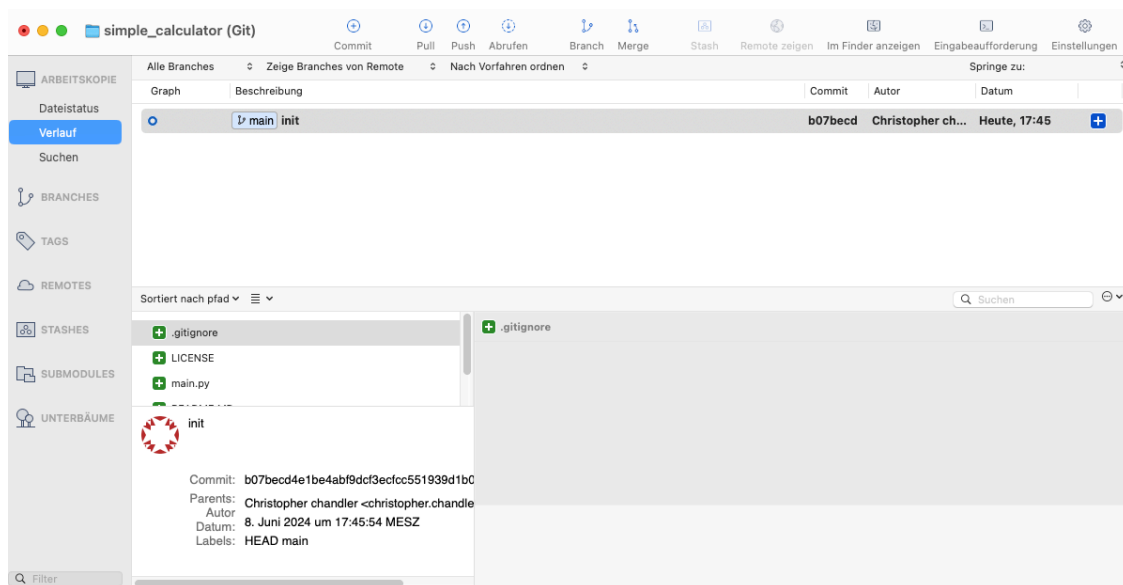
7 Teil 4 Git GUI Software

Bis jetzt haben wir uns nur mit der Kommandozeile befasst. Das ist ganz nützlich, denn man weiß genau, wie die Sachen im Hintergrund funktionieren. Für die Allermeisten ist aber sowas mühsam und umständlich einzusetzen. Zum Glück muss man sich nicht auf sowas beschränken, weil es Programme und Schnittstellen gibt, die uns eine grafische Oberfläche bieten.

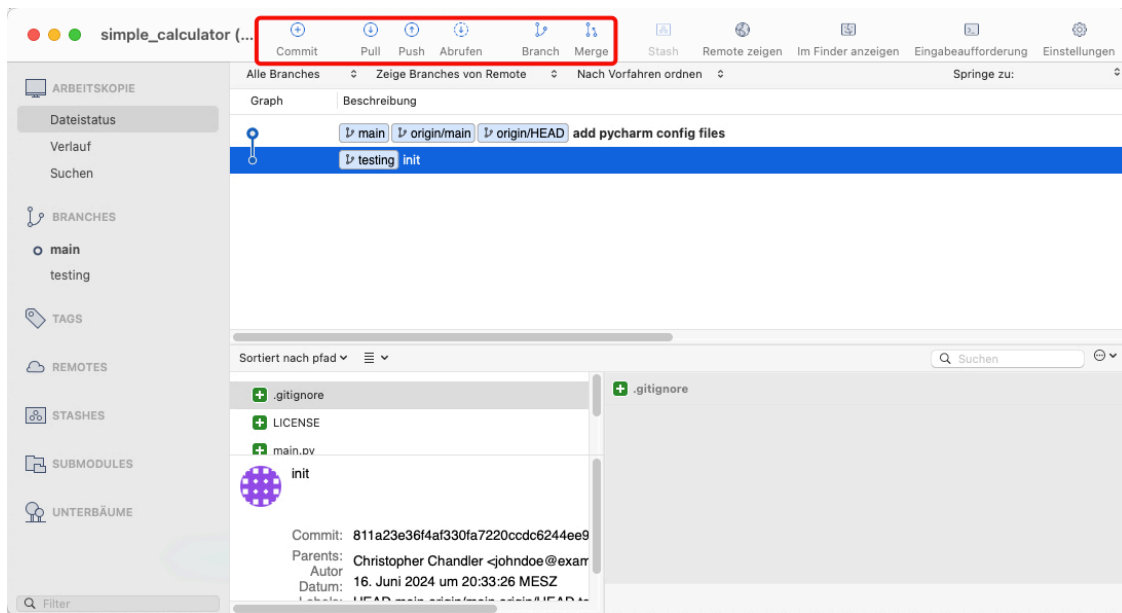
Wir beschäftigen uns aber nur mit ein paar davon: - Source Tree - Git in Pycharm - Github

7.1 4.1 Source Tree

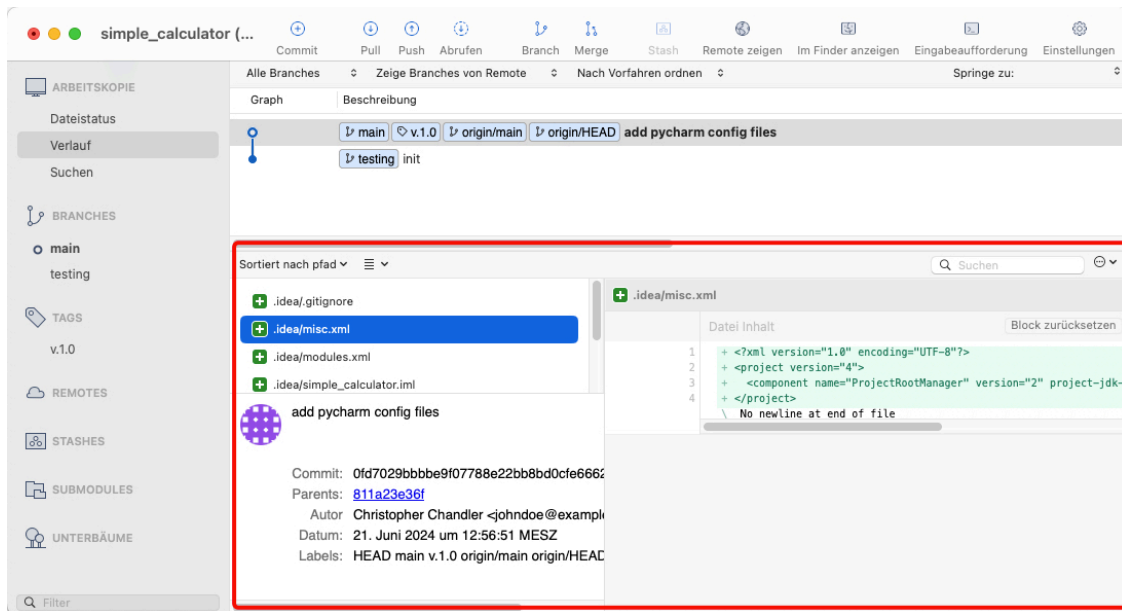
Mit Source Tree haben wir alles auf einen Blick.



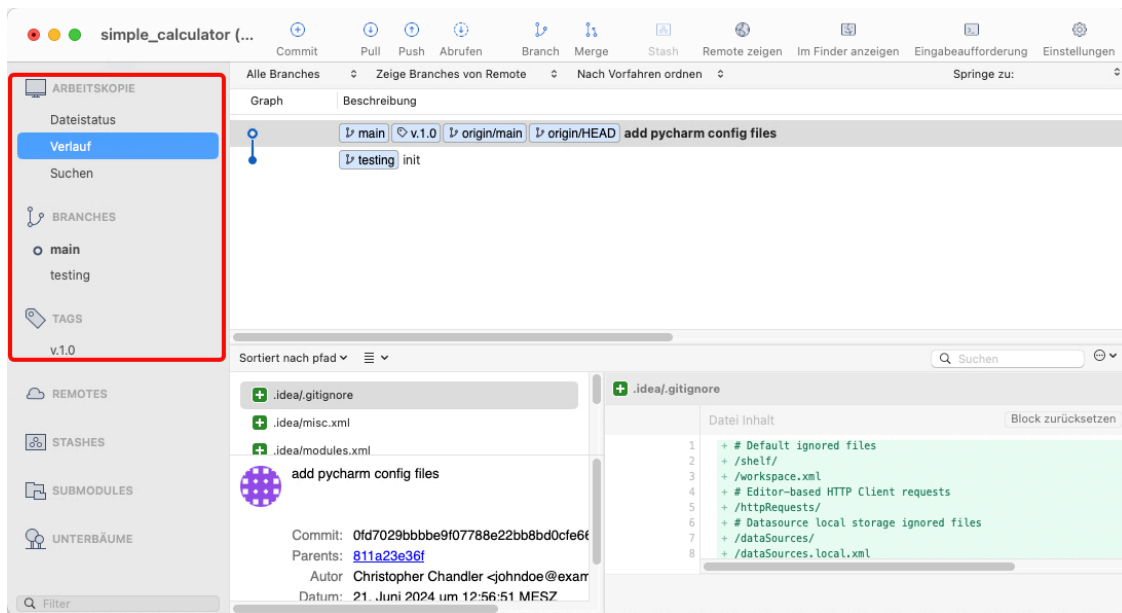
Wir haben die Menüeisten mit den Befehlen, die wir schon aus Git kennen.



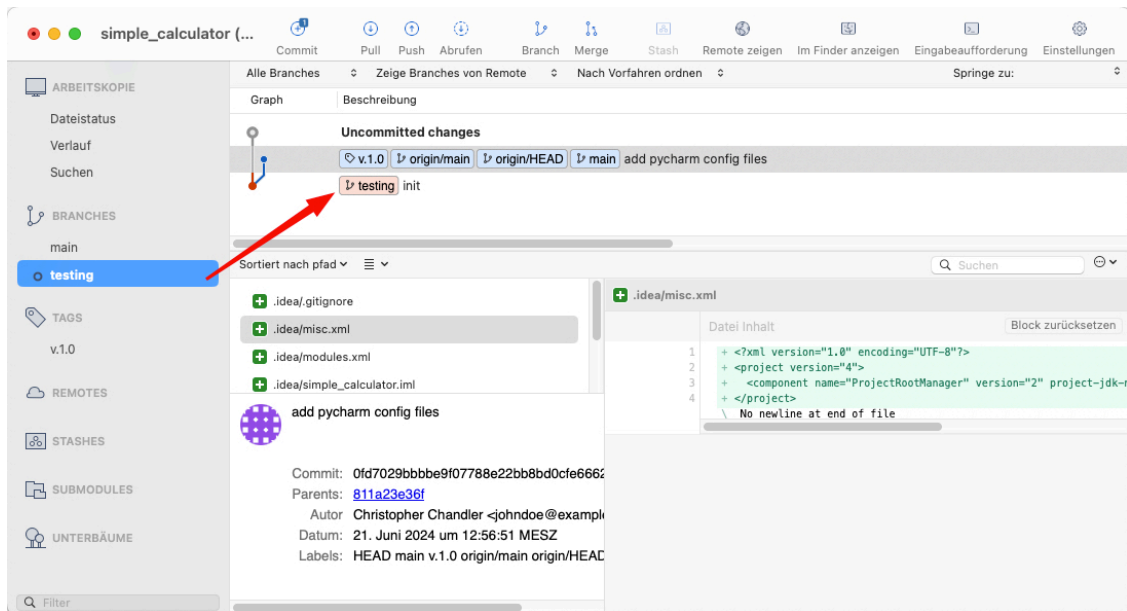
Die Kommentare werden auch auch angezeigt.



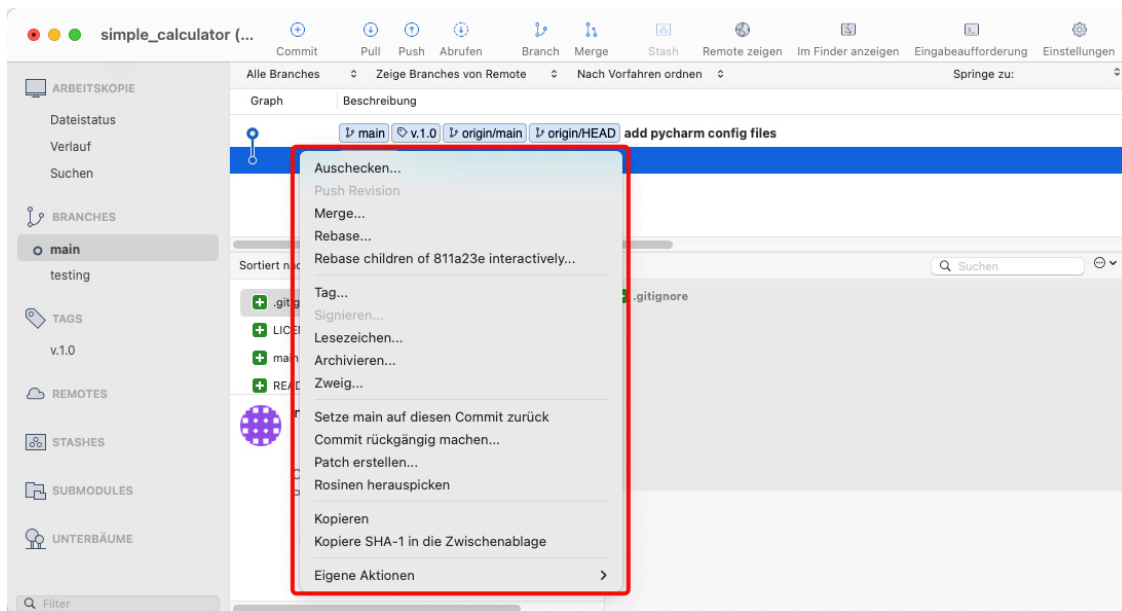
In der Seitenleiste sieht man auch die Branches, Tags und den Verlauf



Wir können auch die Branches anzeigen lassen.

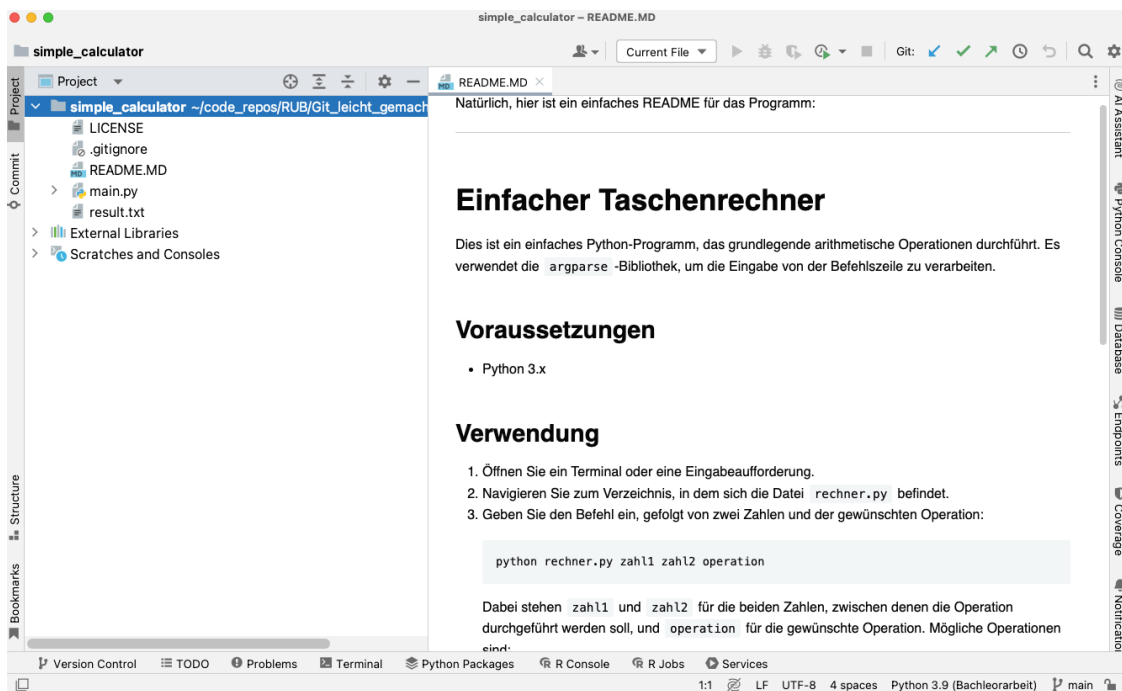


In dem Kontextmenü kann man bestimmte branches auschecken, mergen etc.

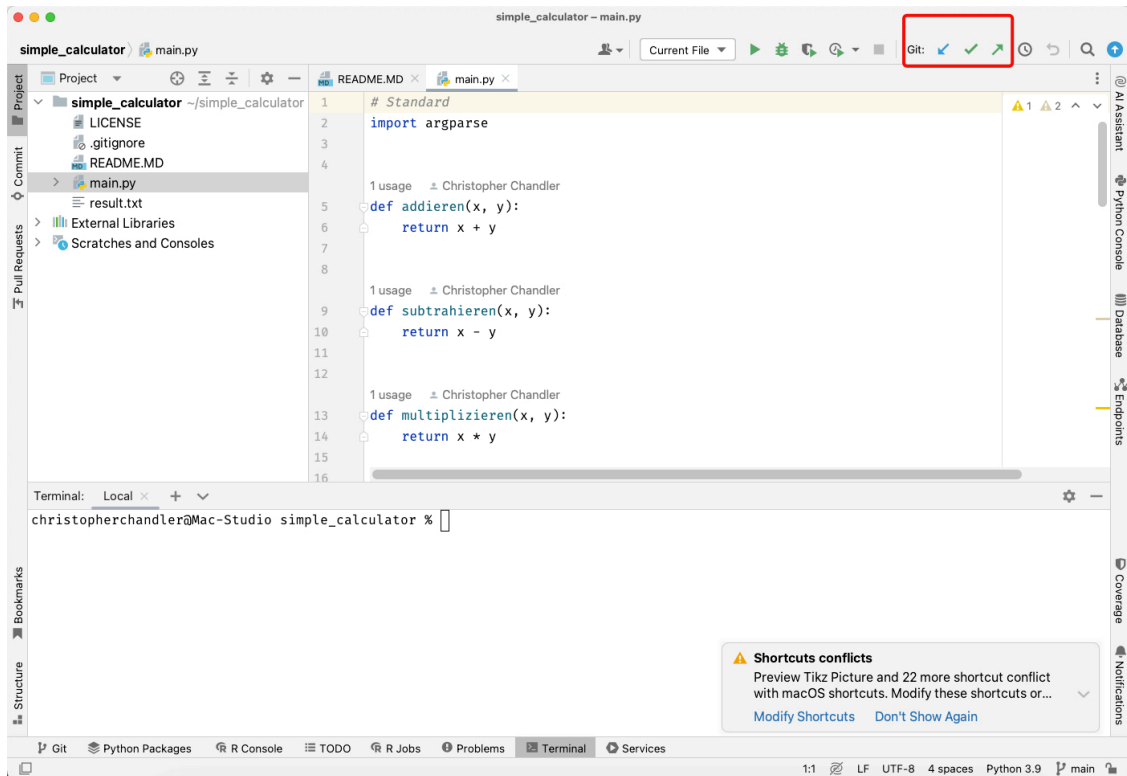


8 4.2 GIT in Pycharm

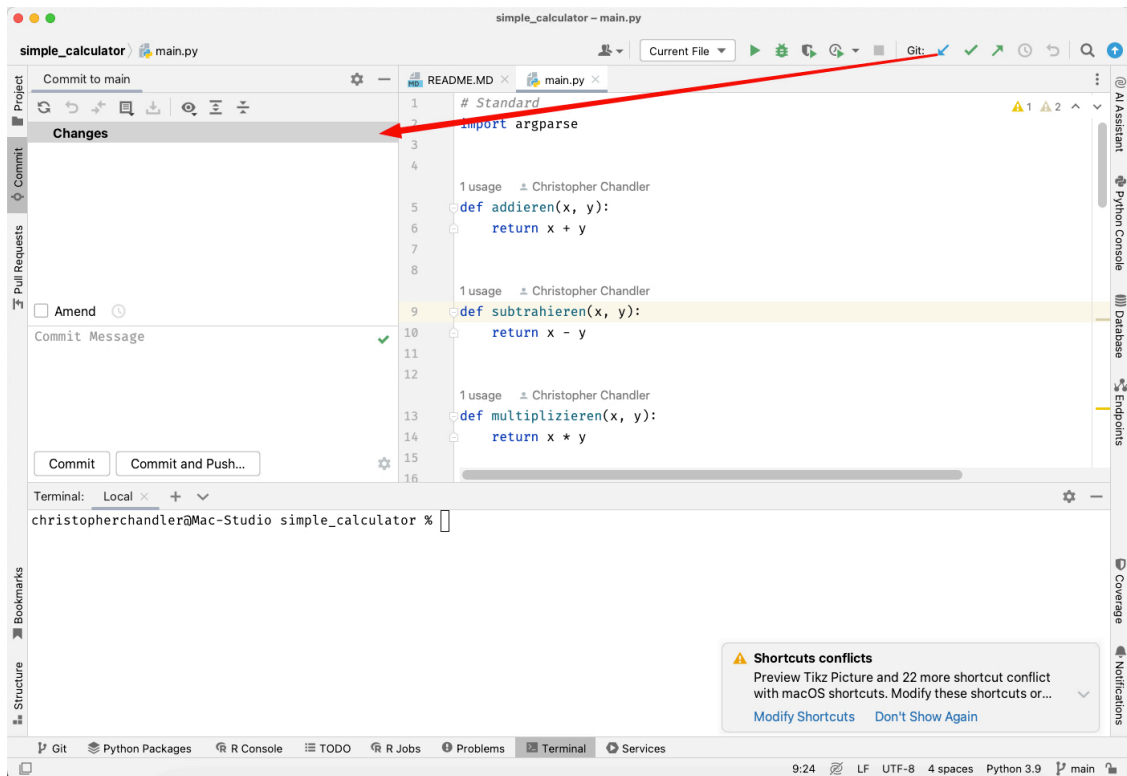
Wenn man den Code direkt in der IDE bearbeiten und mit Git verwalten möchte, geht das auch



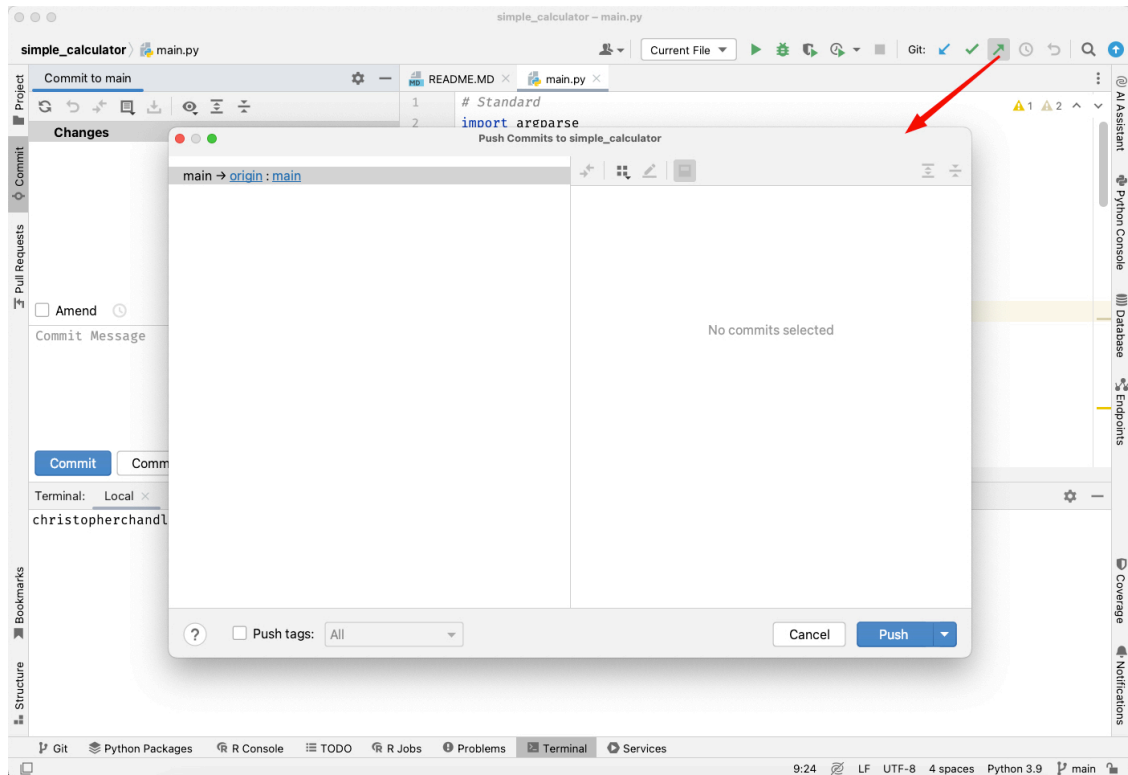
Auch hier gibt es eine Menüleiste



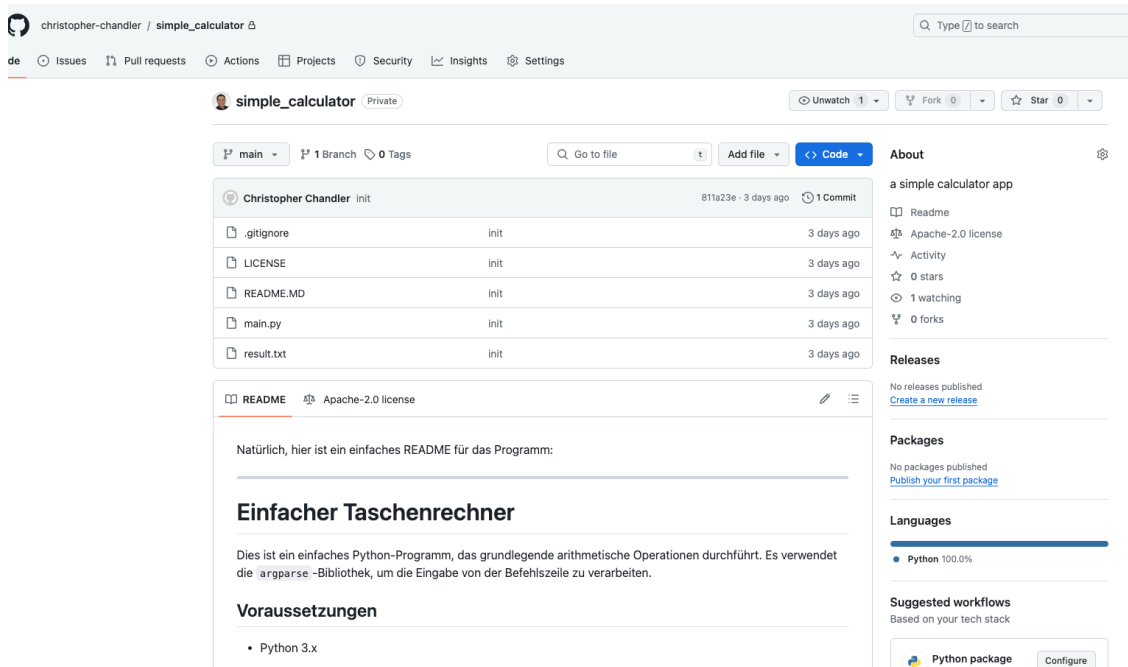
Man kann den Code von Git hub pullen



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 für neulinge. CodeGym. <https://codegym.cc/de/groups/posts/de.379.erste-schritte-mit-git-eine-umfassende-anleitung-fur-neulinge>