Einführung in Git

Versionskontrolle ohne Angst

Lino Haupt

Forum Digitale Uni Jena 2026

Übersicht

Motivation und Beispielfälle

Die Idee hinter `Git`

Commits, Diffs und Branches

Kooperation mit anderen

Auflösung der Beispielfälle

Wie lerne ich `Git` weiter?

Motivation und Beispielfälle

Warum überhaupt `Git`?

Die Grundidee

Das Problem beim Schreiben eines Textes ist oft nicht nur das Schreiben selber.
Das Verwalten und Zusammenführen von Versionen usw. nimmt viel Zeit ein.

Gleich kommen vier Situationen

Die Personen sind fiktiv, aber sie stellen grundsätzliche Schwierigkeiten beim Arbeiten dar.

Interaktive Frage

In welcher Situation erkennt ihr euch am ehesten wieder?
Schreibt gern die Nummer in den Chat.

1. Susi: ein längeres Schreibprojekt

Situation

Susi ist Geisteswissenschaftlerin. Neben ihrer Arbeit an der Universität in Lehre und Forschung hat sie ein **längeres Prosa-Projekt**.
Sie hat nur sehr **unregelmäßig Zeit**, an ihrem Text zu schreiben. Lehre, Forschung, Verwaltung und das Chaos des Lebens holen sie immer wieder ein.

Konkrete Probleme

- Nach längeren Pausen weiß Susi nicht mehr genau, was sie zuletzt geändert hat.
- Sie findet alte Formulierungen manchmal besser, aber nicht mehr wieder.
- Sie möchte verstehen, warum sie bestimmte Abschnitte umgeschrieben hat.
- Ihre Dateien heißen irgendwann `kapitel-final-neu-wirklich.tex`.

2. Ahmet: ein Paper mit riskanten Ideen

Situation

Ahmet arbeitet an einem wissenschaftlichen Paper. Die **Grundfassung steht schon**. Plötzlich hat er zwei größere **experimentelle und riskante Ideen**:

- Er könnte die Einleitung didaktisch völlig anders aufziehen.
- Außerdem könnte er in der Analyse eine andere Methode verwenden.

Konkrete Probleme

- Ahmet möchte experimentieren, ohne die stabile Fassung zu zerstören.
- Er möchte gute Teile der Experimente später übernehmen können oder auch nicht.
- Jede Entscheidung ob ein Experiment übernommen wird, soll unabhängig sein.
- Auch während des Experiments wird er Tippfehler usw. korrigieren, aber wie kriegt er das ohne Divergenz in die experimentellen und stabilen Versionen rein?

3. Olena und Li Wei: gemeinsames Schreiben

Situation

Olena und Li Wei arbeiten **gemeinsam an einem Antrag**. Sie schreiben beide an Textteilen, Literaturüberblick und Zeitplan.
Manchmal **arbeiten sie fast gleichzeitig**. Beide schicken sich gelegentlich **Dateien oder Kommentare per E-Mail** hin und her.

Konkrete Probleme

- Beide ändern manchmal den Text fast gleichzeitig, das führt zu Verwirrung.
- Sie müssen immer wieder händisch Versionen zusammenpflegen.
- Manchmal übersieht Olena, dass Li Wei auf Seite 12 eine Änderung gemacht hat. Diese fehlt dann in der Version, die Olena an Li Wei schickt.
- Manchmal suchen sie sehr lange, um zu finden, wann ein Vorschlag verloren gegangen ist.

4. Diego: ein Projekt auf mehreren Geräten

Situation

Diego arbeitet an seiner Dissertation. Zuhause nutzt er seinen **Desktop-Rechner**. Im Büro arbeitet er am Uni-Rechner. Er reist viel und **in der Bahn schreibt er auf dem Laptop** weiter.
Unterwegs ist das **Internet nicht immer zuverlässig**. Manchmal hat er **vergessen die neueste Version zu laden** und will keine Arbeit doppelt machen.

Konkrete Probleme

- Diego weiß nicht immer, welche Version gerade die neueste ist.
- Er hat Angst, Arbeit doppelt zu machen oder zu überschreiben.
- Manchmal macht er Änderungen und stellt dann fest, dass es gar nicht die neueste Version war. Dann muss er diese Änderungen händisch übertragen.
- Automatische Cloud-Synchronisation hilft nicht mehr, wenn Versionen divergieren.

Wie ihr heute mitmachen könnt

Dieser Workshop ist leider kein vollständiger Mitmachkurs

Bei vielen unterschiedlichen Geräten und Installationen können wir heute nicht alle technischen Probleme live debuggen.
Das Ziel ist zuerst: die Konzepte verstehen.

Trotzdem: gern mittippen

Wenn eure Installation steht, dürft ihr bei der Live-Demo gern Befehle mittippen.
Das ist aber nicht notwendig.

Empfehlung

Lieber heute die Grundideen von `Git` verstehen und danach in einem kleinen Testprojekt ausprobieren, in dem auch etwas schiefgehen darf.

Eine wichtige Einschränkung

Git kann grundsätzlich jede Datei speichern, aber nicht jede Datei gleich gut verstehen.

Wichtig für heute

Heute konzentrieren wir uns auf Dateien, die Git gut versteht: **reine Textdateien**!
Zum Beispiel:

- Text ggf. mit explizit beschriebener Formatierung: `.txt` (beliebiger Text), `.md` (Markdown), `.tex` (LaTeX), ...
- reine Tabellen: `.csv` (Kommaseparierte Daten), ...
- Code: `.py` (Pythoncode), `.R` (R-Code), `.sh` (Shell-Code), ...

Was ist mit Word, LibreOffice und PDFs?

Dateien wie `.docx`, `.odt`, `.pptx` oder `.pdf` kann Git grundsätzlich versionieren.
Sie sind also nicht verboten, aber Git kann sie nur schwer vergleichen und kaum sinnvoll zusammenführen.

Warum erklären wir Git im Terminal?

Terminal ist nicht notwendig

Viele Programme haben Git-Integration.

Terminals zeigen das Grundmodell klarer

Im Terminal sehen wir direkt:

- Was ist ein Repository?
- Was wird gespeichert?
- Was ist ein Commit?
- Was passiert beim Wechseln von Versionen?

Außerdem sind die Befehle überall gleich und damit **übertragbar** (und auch automatisierbar)!

Die Idee hinter `Git`

Fehlkonzeptionen und Arbeitsdefinition

`Git` ist

- **kein** automatischer Speichermechanismus,
- **kein** Hintergrundprozess, der heimlich alles überwacht,
- **keine** KI, die eure Arbeit versteht,
- **keine** Magie,
- **keine** einfache Cloud-Synchronisation wie Dropbox oder Nextcloud.

`Git` macht nur etwas, wenn wir `Git` ausdrücklich aufrufen.

Arbeitsdefinition

`Git` ist ein Werkzeug, mit dem wir die **Geschichte eines Projekts** bewusst in Versionen (Commits) gliedern, ansehen, vergleichen und zusammenführen können.

Drei Ebenen auf dem Weg zum Commit

1. Speichern

Das aktuelle Manuskript liegt bei mir auf dem Schreibtisch.
Die Tinte ist trocken.

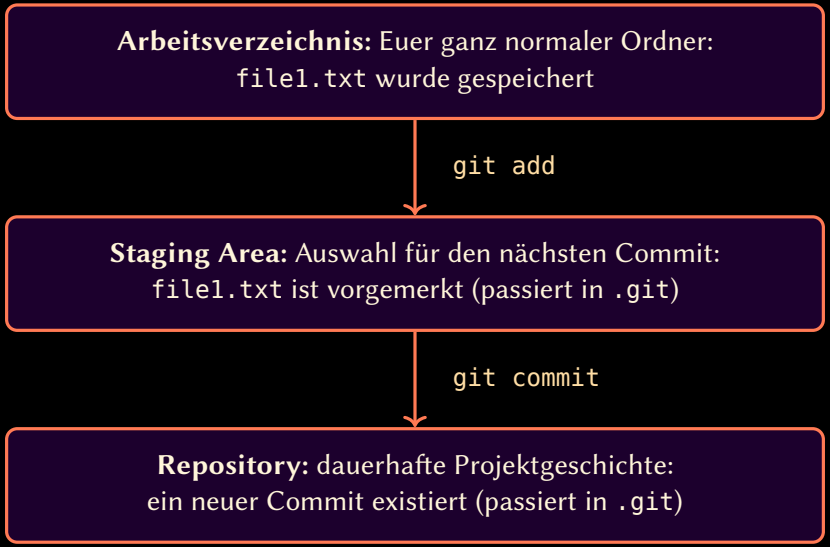
2. Staging

Ich lege fest, welche Änderungen in den nächsten Brief an den Archivar soll.
Nicht alles muss mit.

3. Commit

Der Archivar legt eine neue Akte an und fragt mich was meine Idee bei der Änderung war.
Dieser Projektstand ist jetzt Teil der Geschichte.

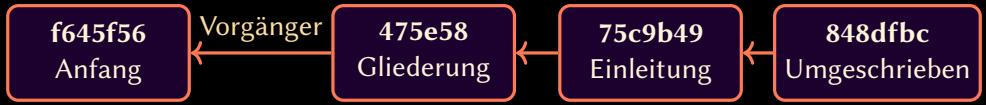
Die drei Ebenen als Bild



Was ist ein Commit?

Das Netzwerk der Commits

Ein Commit bündelt eine Version und verweist auf den Vorgänger.



Nützliche Metadaten

Bestandteil	Bedeutung	Entstehung
Projektstand	Welche Dateien wie aussehen	**durch Staging bestimmt**
Vorgänger	Worauf dieser Commit aufbaut	automatisch
Autor	Wer den Commit erstellt hat	automatisch **(Einstellung)**
Zeitpunkt	Wann der Commit erstellt wurde	automatisch
Nachricht	Was und warum geändert wurde	**vom Nutzer eingegeben**
Hash	Eindeutige Kennung des Commits	automatisch

Was ist ein Diff?

Idee

Ein Diff beschreibt den Unterschied (*Differenz*) zwischen zwei Versionen.
Der Computer vergleicht zwei Dateien und sucht eine möglichst kleine Änderung, die die eine Version in die andere überführt.

Diffs gibt es auch ohne `Git` und sind sehr nützlich.

Anwendungen bei `Git`

Man kann sich den Diff zwischen Commits oder zwischen Arbeitsstand und letztem Commit anzeigen lassen.

Der Diff zeigt die Details. Daher kann sich die Commit-Nachricht darauf beschränken, die Rolle der Änderung zu erklären.

Die Befehle zum Modell

Ebene	Was passiert?	Git-Befehl
Arbeitsverzeichnis	Datei existiert	kein `Git`-Befehl nötig
Status prüfen	`Git` zeigt die Lage	`git status`
Staging	Änderung vormerken	`git add <Dateiname>`
Commit	Version anlegen	`git commit -m <Nachricht>`
Geschichte ansehen	Commits anzeigen	`git log`
Änderungen sehen	Diff anzeigen	`git diff` oder `git diff <v1> <v2>`

Wichtig

`git add` speichert noch keinen Commit.
`git commit` nimmt nur das, was vorher gestaged wurde.

Wir schauen uns das mal im Terminal an!

Der wichtigste Diagnosebefehl

Wenn ihr unsicher seid

```
git status
```

Warum?

`git status` sagt meistens sehr genau:

- Was ist gestaged?
- Was ist geändert, aber noch nicht gestaged?
- Welche Dateien kennt Git noch gar nicht?
- Auf welchem Branch bin ich? (lernen wir später kennen)

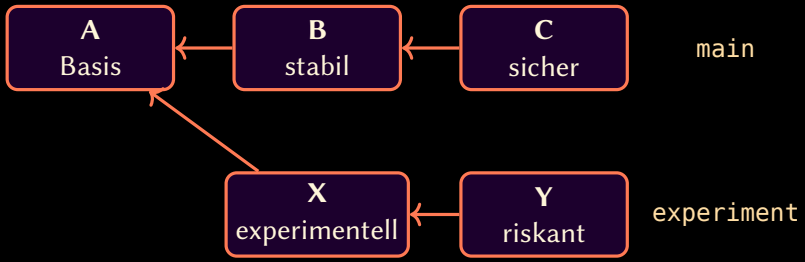
Commits und Branches

Branches: Eine Geschichte kann sich verzweigen

Arbeitsdefinition

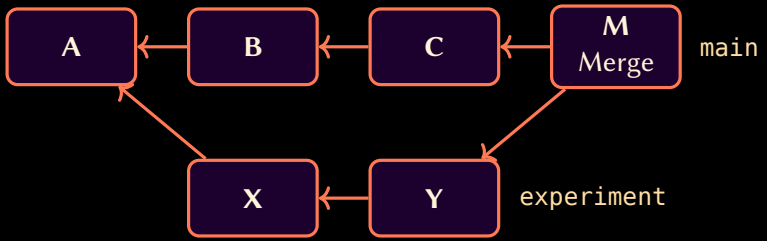
Nach einem Commit kann es mehrere Fortsetzungen, also eine **Verzweigung**, geben. Wir nennen diese Zweige **Branches**. Technisch in `Git` ein Verweis auf den **letzten Commit in einem Zweig**.

Hier sehen wir z.B. eine stabile Entwicklungslinie und einen experimentellen Zweig:



Merging: Zweige müssen nicht getrennt bleiben

Ein Branch erlaubt ein Experiment. Nützlich ist es erst so richtig, wenn man die Änderungen aus diesem Experiment wieder in eine andere Linie einpflegen kann.



Was macht ein Merge?

`Git` sucht den gemeinsamen Vorgänger zweier Branches, vergleicht beide Entwicklungen und versucht die Änderungen auf dem einen Branch auf dem anderen auch zu vollziehen.

Merge-Konflikte sind keine Katastrophe

Was ist ein Merge-Konflikt?

Ein Merge-Konflikt tritt auf, wenn zwei Zweige Änderungen an derselben Stelle enthalten und `Git` Wissen und Autorität fehlen, um zu entscheiden, welche Änderung richtig ist.

Das heißt nicht „`Git` ist kaputt“, sondern „Hier muss jemand entscheiden“.

Merksatz

`Git` entscheidet nicht kreativ über eure Arbeit. Wenn es unsicher ist, hält es an.

Merge-Konflikte lösen

`Git` baut an die Konfliktstellen in die Dateien sogenannte **Konfliktmarker** sowie beide konkurrierenden Versionen ein. Wir müssen dann manuell (oder mit Tools) die Konfliktstellen durch die gewünschte Version ersetzen.

Alternativ kann man einen `Merge` immer abbrechen! `git merge --abort`

Befehle zum Reisen zwischen Commits und Branches

Branch-Befehle

Im Repository reisen

i) Bereits bestehenden Branch besuchen:
`git switch <Name des Branches>`

ii) Neuen Branch erstellen und besuchen:
`git switch -c <Name des Branches>`

iii) Einen ausgewählten alten Commit besuchen:
`git switch --detach <Commit-Hash>`

Eine Kombination aus iii) und ii) erlaubt aus alten Commits neue Branches zu machen!

Wichtig

Wenn man einen neuen Branch erstellt, dann entspricht er genau dem Projektstand (und der Historie), die man gerade sieht.

Branches zusammenführen

Terminal

```
$ git switch main
Switched to branch 'main'

$ git merge experiment
Merge made by the 'ort' strategy.
 text.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Lesart

Wir stehen auf `main`.
Dann holen wir die Änderungen aus `experiment` in `main`.

Kleine Befehlsübersicht

Ziel	Befehl	Text	Blob/Office
Branches anzeigen	`git branch`	✅	✅
Branch wechseln	`git switch <Branchname>`	✅	✅
Branch erstellen und wechseln	`git switch -c <Branchname>`	✅	✅
Alten Commit ansehen	`git switch --detach <Commithash>`	✅	✅
Zurück zur Hauptlinie	`git switch main`	✅	✅
Branch zusammenführen	`git merge <Branch mit Änderungen>`	✅	❌
Unterschiede ansehen	`git diff`	✅	⚠
Änderungen verstehen		✅	❌
Konflikte lösen		✅	❌

✅ gut ⚠ schwierig / Zusatzwerkzeuge ❌ nicht sinnvoll direkt in Git

Kooperation mit anderen

Bisher: Kooperation mit mir selbst

Was wir bisher gesehen haben

Bisher war `Git` vor allem ein Werkzeug, um mit sich selber in Vergangenheit und Zukunft zu kooperieren und Branches zu verwalten.

Was wir jetzt machen

Mehrere Kopien desselben Repositories können Änderungen austauschen.
Das kann bedeuten: mehrere Personen oder auch ich selbst auf verschiedenen Geräten.
Wichtig: Jede Kopie eines Repositories hat ihre eigene lokale Geschichte.
Das gilt für jede Person, jeden Computer und jede Kopie auf diesem Computer.

Remote

Ein Remote ist ein anderes Repository, mit dem ich Commits austauschen kann.
Typische Befehle: `git push` und `git pull`

Ein neues Werkzeug: `clone` und `Git`-Server

Häufig kommt vor: Ein Projekt gibt es schon, zum Beispiel auf dem GitLab der Universität, einem USB-Stick, einem anderen Rechner usw.

Idee

`git clone` erstellt eine lokale Kopie eines vorhandenen Repositories.

Ein `Git`-Server

Ein Server bewahrt Repositories auf und entscheidet nach Regeln unter anderem:

- Wer darf Änderungen herunterladen?
- Wer darf Änderungen hochschieben?

Daher muss man sich bei Servern um **Authentifizierung** kümmern.

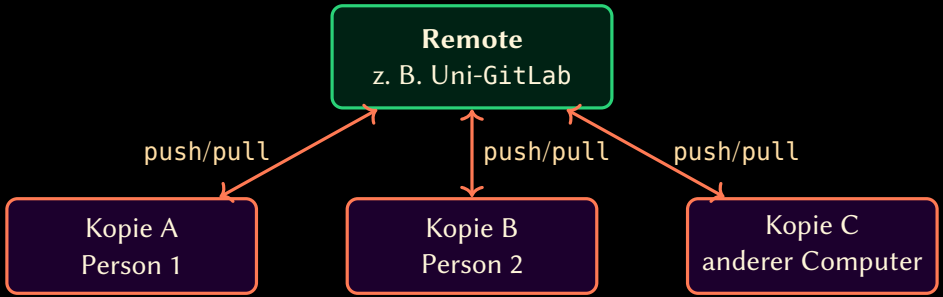
Merksatz

`GitLab` und `GitHub` sind nicht `Git` selbst. Sie sind Dienste, die `Git`-Repositories hosten, eine Weboberfläche und Zusatzfunktionen anbieten.

Kooperation als Bild

Push und Pull

Der Befehl `git push` versucht die lokale Historie auf den Server zu schreiben.
Der Befehl `git pull` versucht die Historie vom Server zu holen und lokal einzupflegen.



`git pull`: zwei Schritte auf einmal

Grundidee

`git pull` bedeutet: **`git fetch` und danach `git merge`**.
`git fetch` fragt den Remote: **„Welche Commits und Branches kennst du?“**
Danach kennt meine lokale Kopie den Stand des Remotes als Branch unter Namen wie `origin/main`.

Nach einem `fetch` müssen wir also wieder über Branches und Merges nachdenken.

Konflikte bei `pull` und `push`

- Bei einem `git pull` wird nach dem `fetch` ein `merge` gemacht. Es können also auch Konflikte auftreten. Auch hier sind diese nicht schlimm.
- `git push` soll dem remote nur die neuen Commits mitteilen. Sollte der Remote aber Commits haben, die wir lokal noch nicht kennen sind die Historien inkompatibel. Dann müssen wir erst `pull`en und die Geschichte vereinen.

Auflösung der Beispielfälle

Auflösung 1: Susi

Problem

Susi arbeitet unregelmäßig an einem langen Text. Nach Pausen möchte sie wissen:
Was habe ich geändert? Warum habe ich es geändert? Wie sah die alte Fassung aus?

Git-Werkzeuge

- Kleine Commits speichern Arbeitsschritte, Commit-Nachrichten erklären die Absicht.
- `git diff` zeigt aktuelle Änderungen.
- `git log` hilft, alte Stände wiederzufinden.
- `git switch --detach <Commit>` erlaubt einen Blick in alte Fassungen.

Workflow-Idee

Susi ersetzt Dateinamen wie `final-neu-wirklich.md` durch gute Commit-Nachrichten und erhält dadurch eine lesbare Projektgeschichte.

Auflösung 2: Ahmet

Problem

Ahmet hat eine stabile Paper-Fassung.
Jetzt möchte er riskante Ideen ausprobieren, ohne die stabile Version zu beschädigen.

Git-Werkzeuge

- Branches erlauben mehrere Experimente neben der stabilen Linie.
- Commits dokumentieren die Schritte im Experiment.
- Merges übernehmen gelungene Experimente in den Hauptzweig.
- Ein verworfenes Experiment kann einfach als Branch liegen bleiben.

Workflow-Idee

Ahmet arbeitet nicht mit Kopien wie `paper-riskant-v2.tex`.
Er gibt jedem Experiment eine eigene Entwicklungslinie.

Ahmet: stabile Korrekturen ins Experiment holen

Beispielbefehle

```
$ git switch main # Zurück auf den Hauptzweig
# Tippfehler in der stabilen Fassung korrigieren
$ git add paper.tex # Stage
$ git commit -m "Korrigiere Tippfehler in Abschnitt 2" # Commit
$ git switch experiment-einleitung # Zurück zum experimentellen Zweig
$ git merge main # oder cherry-pick für spezifische Wahl
```

Deutung

Manchmal entstehen gute kleine Korrekturen auf `main`, während ein Experiment noch läuft. Dann kann Ahmet die stabile Linie in sein Experiment hineinmergen.

Auflösung 3: Olena und Li Wei

Problem

Olena und Li Wei arbeiten gemeinsam und beide ändern Text oft fast gleichzeitig.

Git-Werkzeuge

- Ein Remote-Repository dient als gemeinsamer Treffpunkt.
- Jede Person arbeitet lokal in einer eigenen Kopie.
- Branches trennen parallele Arbeit.
- `git pull` holt neue Änderungen; `git push` teilt eigene Commits.
- Merge-Konflikte markieren Stellen, die gemeinsam entschieden werden müssen.

Workflow-Idee

Nicht Dateien per Mail hin und her schicken.
Stattdessen Commits über ein gemeinsames Repository austauschen.

Auflösung 4: Diego

Problem

Diego arbeitet auf mehreren Geräten. Er möchte nicht raten, welche Datei gerade die neueste ist.

Git-Werkzeuge

- Das Remote-Repository ist Bezugspunkt.
- Jedes Gerät hat aber eine eigene lokale Kopie!
- Vor dem Gerätewechsel: committen und pushen
- Auf dem nächsten Gerät: pullen.
- Falls push/pull vergessen oder offline: Erst lokal committen und später mergen.

Workflow-Idee

`Git` macht nachvollziehbar, welcher Projektstand auf welchem Gerät angekommen ist.

Offline oder vergessen zu pushen?

>_ Terminal

```
$ git status # Auch ohne Internet möglich
$ git add kapitel-4.tex
$ git commit -m "Skizziere Beispiel in Kapitel 4"
# Später, wenn Internet wieder da ist:
$ git pull # Fetch + Merge (Möglicherweise ein Merge-Konflikt)
$ git push # Teile Fortschritt mit Remote
```

Deutung

`Git` funktioniert lokal. Diego kann also auch offline sinnvolle Arbeitsschritte speichern.

Merge-Konflikte

Auch hier bedeutet das nur, dass `Git` einfach nicht weiß, was genau die gewollte Änderung ist. Das muss man `Git` dann mitteilen.

Weiterlernen

Wie lerne ich Git weiter?

Die gute Nachricht

Man muss Git nicht vollständig verstehen, bevor man es benutzen darf.
Man lernt Git am besten an kleinen echten Projekten.

Am Anfang reicht ein Werkzeugkasten

Ein paar sichere Grundbefehle.
Ein mentales Modell.
Und die Fähigkeit, Hilfe gezielt nachzuschlagen.

Zum Autofahren muss man auch nicht mit LKW-Anhänger rangieren können!
Ziel ist **Learning by doing!** und nicht „alles können“.

Ein guter Lernweg

Mini-Trainingsplan

- Ein Repository für ein kleines Textprojekt anlegen und nur wenige Befehle regelmäßig benutzen.
- Eine Datei ändern. Änderungen mit `git diff` ansehen.
- Eine konkrete Datei mit `git add` vormerken.
- Mit `git commit` als Version ablegen.
- Vor und nach jedem Befehl kurz `git status` lesen.
- Regelmäßig fragen: Was hat sich geändert? Habe ich das erwartet?
- Mit `git log` die Geschichte ansehen.
- Fragen notieren und gezielt nachschlagen.

Hilfe direkt in `Git`

Die `--help` Flag

Man kann `--help` an Befehle dran hängen. Dann werden sie nicht ausgeführt sondern Hilfe ausgegeben.

- `git --help` zeigt eine kurze Übersicht über alle wichtigen Befehle.
- `git <subcommand> --help` zeigt eine Übersicht zu dem Befehl:
  `git merge --help` erklärt den `merge`-Befehl.

Nicht erschrecken!

Es öffnet sich dann manchmal ein „Pager“, diesen beendet man mit der Taste `q` (`q` für „quit“). Navigation im Pager ist über Pfeiltasten. Mit der Taste `h` zeigt der Pager selbst Hilfe an.

Welche Art von Quelle brauche ich gerade?

Von der Git-Community selbst:

- Offizielle Referenz: `https://git-scm.com/docs`
- Offizieller Spickzettel: `https://git-scm.com/cheat-sheet`
- Offizielles Buch: `https://git-scm.com/book/de/v2`

Gute externe Materialien

- Git für offene und reproduzierbare Forschung (Buch):
  `https://book.the-turing-way.org/reproducible-research/vcs`
- Git für Anfänger (Tutorial): `https://swcarpentry.github.io/git-novice`