dataport

anderScore
trust in competence

Tag 1: Einführung in Git und GitLab, Git-Workflow im Team

17.06.2024, Daniel Krämer & Malte Fischer

Agenda

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
  - Einführung & Kursüberblick
  - Grundlagen von Git
  - Git Rebase und Merge-Strategien
  - Git Remote
  - Grundlagen von GitLab
  - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
  - Gitflow-Workflow
  - Tags, Releases & deren Verwaltung
  - GitLab-Runner
  - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
  - GitOps Grundlagen
  - Lokale Entwicklung mit Docker
  - Container/Docker-Registry
  - Erstellen von Release- und Tagged-Images
  - Möglichkeiten des Deployments & Verwaltung von Konfiguration
  - Abschlussübung & Diskussion

Agenda

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
  - Einführung & Kursüberblick
  - Grundlagen von Git
  - Git Rebase und Merge-Strategien
  - Git Remote
  - Grundlagen von GitLab
  - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
  - Gitflow-Workflow
  - Tags, Releases & deren Verwaltung
  - GitLab-Runner
  - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
  - GitOps Grundlagen
  - Lokale Entwicklung mit Docker
  - Container/Docker-Registry
  - Erstellen von Release- und Tagged-Images
  - Möglichkeiten des Deployments & Verwaltung von Konfiguration
  - Abschlussübung & Diskussion

Inhalt

- Einführung
  - Version Control System
  - Zentrale und dezentrale Versionsverwaltung
- Grundlagen von Git
  - Was ist Git?
  - Konfiguration
  - Projekte und Repositories
  - Commits
  - Branches
  - Zurücksetzen von Commits

Grundlagen und Konzepte eines

Version Control System

Version Control System

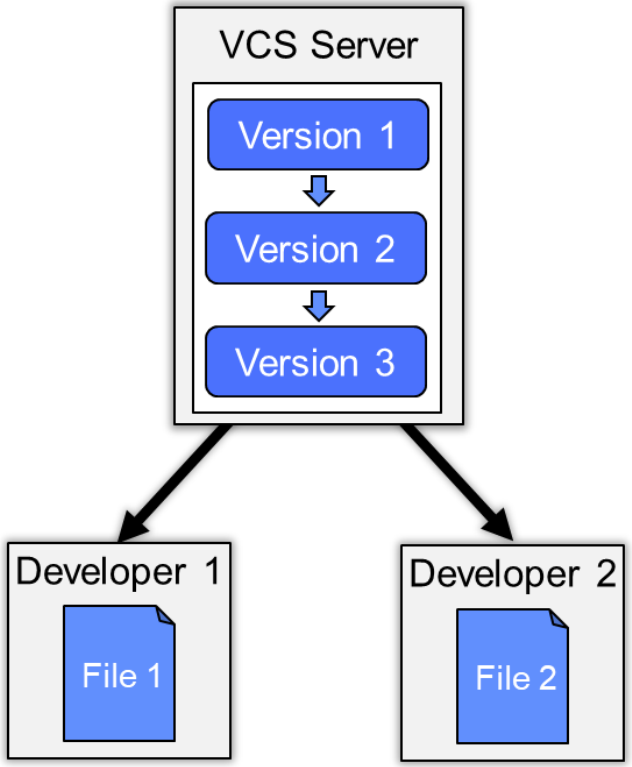
- Dient zur Erfassung von Änderungen an Dateien und Dokumenten
- Speichert Änderung, Zeitstempel und Autor
  - Protokollierung: Wer hat wann welche Änderung vorgenommen
  - Wiederherstellung und Archivierung: Sicherung von Dateien ermöglich Rückkehr zu vorherigen Versionen
  - Ermöglicht gemeinsame Arbeit von mehreren Personen an einem Projekt durch mehrere Entwicklungszweige (Branches)
- Häufig in Softwareentwicklung zur Verwaltung von Quellcode eingesetzt
- Unterscheidung zwischen
  - Lokaler Versionsverwaltung
  - Zentraler Versionsverwaltung
  - Verteilte Versionsverwaltung

Lokale Versionsverwaltung

- Versioniert oft nur eine einzige Datei bzw. Dokument
- Populäre Umsetzungen
  - Source Code Control System (SCCS), 1972
  - Revision Control System (RCS), 1982
- Für Einsatz in ganzen Projekten und Kollaborationen ungeeignet
- Verwendung heutzutage in Büroanwendungen zur Versionierung einzelner Dokumente

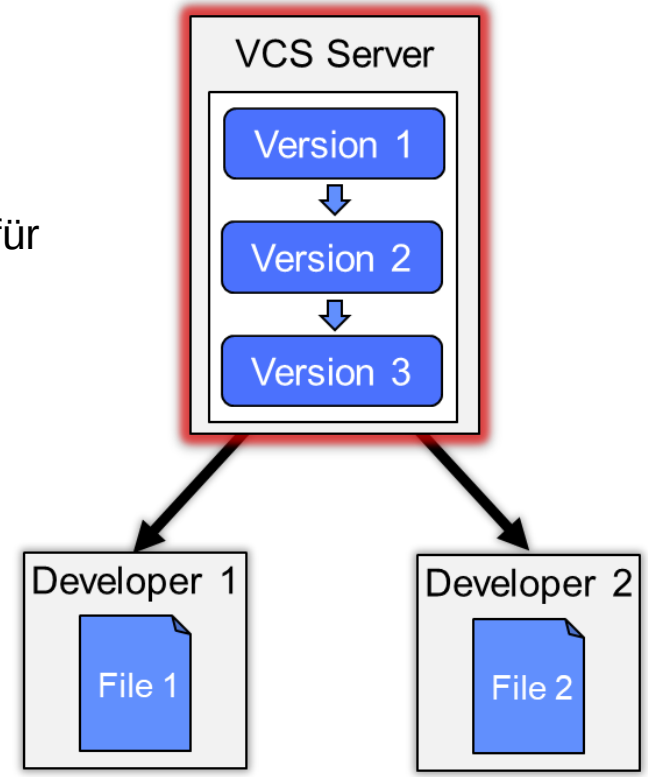
Zentrale Versionsverwaltung

- CVCS (Centralized Version Control System)
- Client-Server Struktur
- Versionierung von vollständigen Projekten
- Zugriff auf zentralen Server über Netzwerk
- Populäre Umsetzungen
  - Concurrent Versions System, 1990 (seit 2008 keine Weiterentwicklung)
  - Apache Subversion, 2000



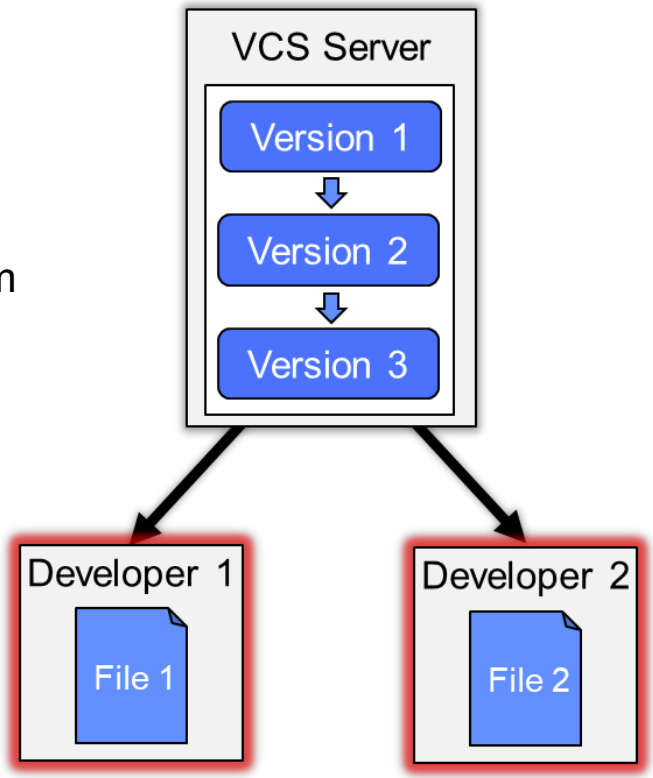
CVCS – Zentraler Server

- Ermöglicht gemeinsame Arbeit an Projekten
- Speicherort der Versionshistorie
  - Entwickler können hier Änderungen nachvollziehen
  - Single Source of Truth - einzige autoritative Quelle für Code und Historie
  - Dadurch aber auch Single Point of Failure
- Ermöglicht Zugriffkontrolle
  - Nur autorisierte Nutzer können Dateien lesen bzw. bearbeiten
  - Vergabe von unterschiedlichen Berechtigungen



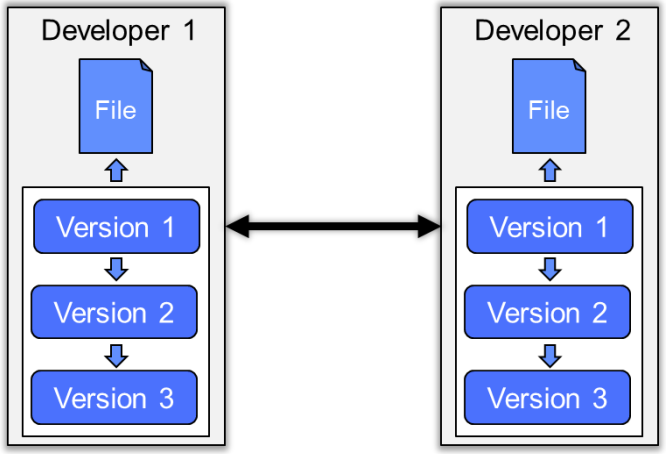
CVCS – Client

- Entwickler können Dateien zur Bearbeitung „auschecken“
  - Datei wird auf lokalen Arbeitsplatz kopiert
  - Änderungen werden lokal vorgenommen
  - Nach Abschluss wird Datei wieder „eingecheckt“, um Änderungen auf den Server zu übernehmen
- Benötigen Netzwerkverbindung, um effektiv arbeiten zu können
- Server gibt Überblick über das Projekt



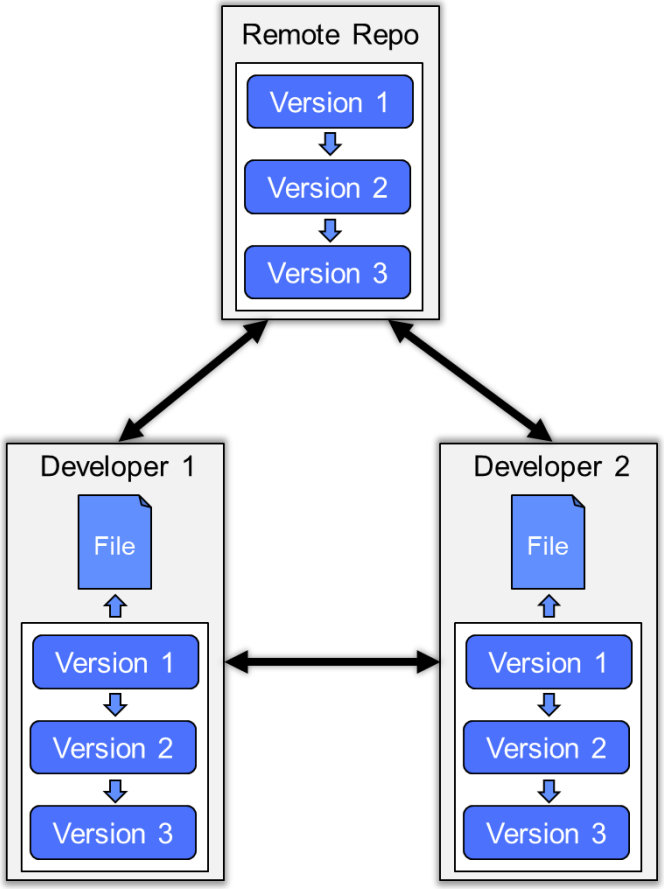
Dezentrale Versionsverwaltung

- DVCS (Distributed Version Control System)
- Kein zentrales Repository
  - Jeder Teilnehmer hat sein eigenes lokales Repository
  - Historie kann ohne Verbindung zu einem Server nachverfolgt werden
- Jedes Repository kann mit jedem anderen abgeglichen werden
  ➔ zentraler Server kann theoretisch entfallen
- Populäre Umsetzungen
  - BitKeeper, 2000
  - Darcs, 2003
  - Mercurial, 2005
  - Git, 2005



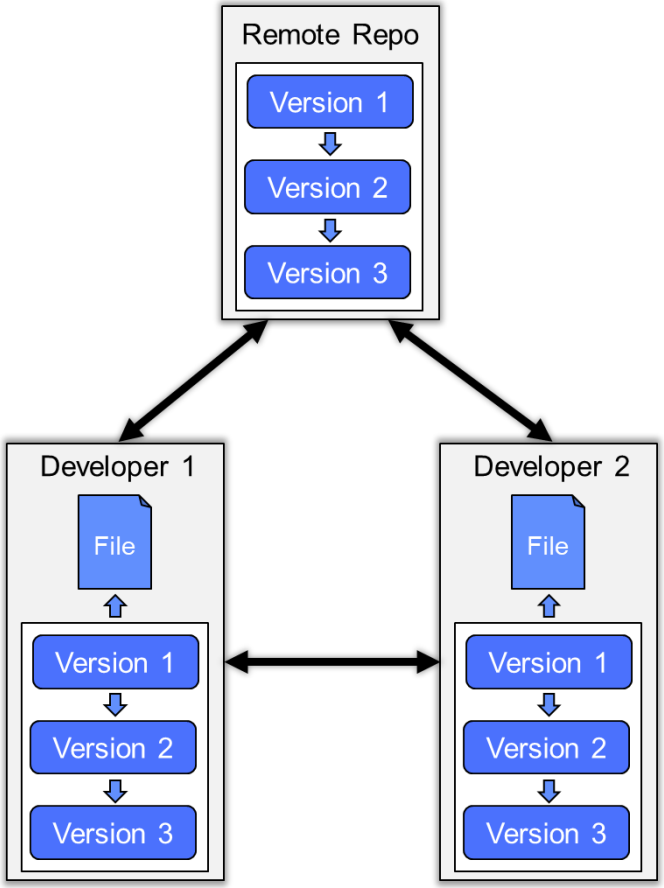
DVCS – Remote Repository

- Oft existiert ein Remote Repository
  - Übernimmt einzelne Konzepte des Servers aus der CVCS
  - Für Betrieb des DVCS nicht unbedingt notwendig
- Vereinfacht Zusammenarbeit
  - Zentraler Ort, von dem Entwickler ihre lokalen Repositories ableiten
  - Abrufen und Hochladen von Änderungen
- Ermöglicht zentrale Zugriffskontrolle



DVCS – Nutzung

- Keine dauerhafte Netzwerkverbindung notwendig
- Entwickler holt such aktuellen Stand vom Remote Repository (*pull*)
- Änderungen werden im eigenen lokalen Workspace vorgenommen und anschließend wieder ans Remote Repository gesendet (*push*)
- Durch lokale Änderungen entstehen zunächst keine Konflikte bei Änderung von mehreren Entwicklern an einer Datei
- Eventuelle Konflikte müssen bei der Zusammenfügung im Remote Repository aufgelöst werden



Einführung und

Konfiguration von Git

Git – Einführung

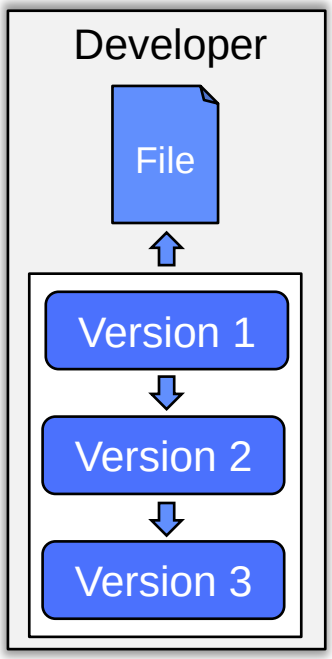
anderScore trust in competence

Was ist Git?

- Freie Software zu dezentralen Versionsverwaltung
- 2005 von Linus Torvalds entwickelt
- Torvalds benötigte durch Lizenzänderungen von BitKeeper neue Quellcode-Management-Software zur Entwicklung des Linux Kernels
- Am weitesten verbreitete Versionsverwaltung
- Bedienung via Command Line
- Vielzahl von GUI-Applikationen und IDE-Integrationen (Intellij, Eclipse uvm.)
- Website: https://git-scm.com/

Git – lokales Repository

- Git lässt sich als DVCS auch ausschließlich lokal Betreiben
  - Remote Repository und andere Mitwirkende entfallen
- Einsatz zur Versionierung auch bei eigenen Projekten ohne andere Mitwirkende sinnvoll
- Zunächst Betrachtung eines lokalen Szenarios, um Begriffe und Basisfunktionen zu verstehen



Installation von Git

- Aktuelle Version von Git prüfen

```
git --version
```

- Bei fehlender Installation Git installieren: https://git-scm.com/downloads

Konfiguration von Git

- Git bietet eine Vielzahl von Konfigurationsmöglichkeiten
- Beinhaltet Tool zur Konfiguration

```
git config
```

- Konfigurationsdateien können auf drei Ebenen gespeichert werden
  - **Systemkonfiguration**
    - /etc/gitconfig (Linux)
    - C:\ProgrammData\Git\config oder C:\Programm Files\Git\etc\gitconfig (Windows, Git Version 2.x oder neuer)
  - **Userkonfiguration**
    - ~/.gitconfig oder ~/.config/gitconfig (Linux)
    - C:\User\<User>\.gitconifg (Windows)
  - **Projektkonfiguration**
    - <Pfad zum Projekt>/.git/config
- Nachfolgende Level überschreiben vorheriges Level

Git – Einführung

- Anzeigen der aktuellen Konfiguration mittels

```
git config --list --show-origin
```

- Identität konfigurieren
  - Notwendig, um mit Git arbeiten zu können
  - Autorinformation, die Git bei Änderungen beifügt

```
git config --global user.name "<Name>"
git config --global user.email <Email>
```

Übungsaufgabe 1: Installation

1. Überprüfen Sie, ob eine aktuelle Version von Git auf Ihrem Rechner installiert ist mit dem Befehl `git --version`

```
$ git --version
git version 2.34.1
```

2. Wenn keine aktuelle Version installiert ist, folgen Sie der Anleitung auf https://git-scm.com/book/de/v2/Erste-Schritte-Git-installieren für Ihr jeweiliges System
3. Falls noch nicht geschehen, setzen mit dem folgenden Befehl Ihren globalen Namen sowie Ihre E-Mail

```
$ git config --global user.name "<Ihr Name>"
$ git config --global user.email "<Ihre E-Mail>"
```

Konzepte und

Grundlagen von Git

Git Projekte

- Sammlung von Dateien, die als Projekt zusammengehören und von Git versioniert werden
- Ein Git Projekt liegt immer in einem Ordner mit beliebig vielen Unterordnern
- Hauptordner wird auch als Workspace bezeichnet
- Um einen Ordner und seinen Inhalt mittels Git versionieren, kann man folgenden Befehl nutzen

```
git init
```

- Dabei wird ein Unterordner **.git** angelegt, indem sich das Repository des Projektes befindet

Beispiel Anlegen eines Git Projektes

```
$ ls -la
total 12
drwxr-xr-x  3 gituser gituser 4096 May  9 22:32 .
drwxr-x--- 15 gituser gituser 4096 May  9 22:32 ..
-rw-r--r--  1 gituser gituser    0 May  9 22:30 file1.txt
-rw-r--r--  1 gituser gituser    0 May  9 22:30 file2.txt
drwxr-xr-x  2 gituser gituser 4096 May  9 22:31 subdir

$ git init
Initialized empty Git repository in /home/gituser/example/.git/

$ ls -la
total 16
drwxr-xr-x  4 gituser gituser 4096 May  9 22:32 .
drwxr-x--- 15 gituser gituser 4096 May  9 22:32 ..
drwxr-xr-x  7 gituser gituser 4096 May  9 22:32 .git
-rw-r--r--  1 gituser gituser    0 May  9 22:30 file1.txt
-rw-r--r--  1 gituser gituser    0 May  9 22:30 file2.txt
drwxr-xr-x  2 gituser gituser 4096 May  9 22:31 subdir
```

Status des Workspaces

- Um den Status eines Workspaces anzuzeigen kann man folgenden Befehl verwenden

```
git status
```

- Zeigt Informationen über Dateien im Workspace
- Beispiel

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt
        file2.txt
        subdir/

nothing added to commit but untracked files present (use "git add" to track)
```

- Dateien müssen explizit zur Versionierung von Git hinzugefügt werden
- Sogenannte „Tracked Files“ werden dann auf Änderungen überwacht

Übungsaufgabe 2: Repository anlegen

1. Erstellen Sie einen Ordner MyRepository und wechseln Sie in diesen.
2. Legen Sie im Ordner ein neues Git-Repository an mit dem Befehl git init
   - Beim ersten Anlegen eines Repositories zeigt Git einen Text bezüglich Benennung des Default Branches an
   - Möchten Sie den Default Branch global umbenennen, so können Sie dies mit dem Befehl `git config --global init.defaultBranch <Name>` machen
   - Im folgenden wird der Default Branch als main bezeichnet
3. Schauen Sie sich über die Ausgabe von `git status` an, ob Ihr Repository erfolgreich angelegt wurde.

Tracked Files

- Um eine Datei in die Git Versionierung aufzunehmen, nutzt man

```
git add <file>
```

- Beispiel

```
$ git add */*.txt

$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt
        new file:   subdir/subdir_file1.txt
```

- `git add */*.txt` tracked beispielweise alle Textdateien im Workspace

Tracking von Ordnern

- Git versioniert keine Ordner an sich
- Ordner werden indirekt über Dateiinformationen verwaltet und aufgenommen
- Dadurch können nur nicht-leere Ordner mit im Repository aufgenommen werden
- Möchte man Ordner ohne Inhalt versionieren, so muss man ein „Dummy File“ in diesem anlegen

Beispiel Kein tracking von leeren Ordnern

```
$ mkdir empty_subdir

$ ls -la
total 20
drwxr-xr-x  5 gituser gituser 4096 May  9 23:07 .
drwxr-x--- 15 gituser gituser 4096 May  9 22:34 ..
drwxr-xr-x  7 gituser gituser 4096 May  9 23:06 .git
drwxr-xr-x  2 gituser gituser 4096 May  9 23:07 empty_subdir
-rw-r--r--  1 gituser gituser 0 May  9 22:30 file1.txt
-rw-r--r--  1 gituser gituser 0 May  9 22:30 file2.txt
drwxr-xr-x  2 gituser gituser 4096 May  9 22:54 subdir

$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt
        new file:   subdir/subdir_file1.txt
```

- Neuer Ordner wird von Git nicht erkannt und taucht nicht im Status auf

Beispiel Tracking mit Dummy File

```
$ touch empty_subdir/.gitkeep

$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt
        new file:   subdir/subdir_file1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        empty_subdir/
```

- Durch Dummy File wird der Ordner erkannt

Git – Grundlagen

anderScore trust in competence

Commits

- Ein Commit speichert den aktuellen Zustand der zum Commit angemeldeten Dateien im Repository
- Änderungen committen mittels

```
git commit
```

- Beispiel

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
        new file:   subdir/subdir_file1.txt

$ git commit -m "Initial commit“
[main (root-commit) c61ef14] Initial commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 subdir/subdir_file1.txt
```

Commit Messages

- `git commit -m` oder `--message` gibt die Option, dem Commit eine Beschreibung hinzuzufügen
- Zu Commit Messages gibt es eine Vielzahl verschiedener Conventions und Guidelines
- Beispiele
  - https://www.baeldung.com/ops/git-commit-messages
  - https://www.gitkraken.com/learn/git/best-practices/git-commit-message
  - https://github.com/angular/angular/blob/main/CONTRIBUTING.md#commit
  - …
- Generell sollte eine Commit Message eine kurze aber aussagekräftige Auskunft über den Commit geben

Staging von Änderungen für neue Commits

- Im vorherigen Beispiel befanden sich nur neu aufgenommene Dateien im Commit
- Wird eine Datei verändert, so muss man diese für den nächsten Commit „stagen“
- Beispiel mit veränderter file1.txt

```
$ echo test > file1.txt

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

- Git zeigt an, dass file1.txt modifiziert wurde
- Ohne Staging würde die Datei beim nächsten Commit nicht beachtet

Git – Grundlagen

- Eine Datei wird über folgenden Command gestaged

  `git add <file>`

- Beispiel

```
$ git add file1.txt

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

$ git commit -m "Update file1.txt"
[main e0b3cf3] Update file1.txt
 1 file changed, 1 insertion(+)
```

- Ermöglicht selektiv Dateien für Commit auszuwählen
- `git commit -a` oder `--all` staged alle geänderten Files für den Commit, ist also eine Kurzform für `git add *` + `git commit`

Inhalt eines Commits

- Änderungen innerhalb eines Commits sollten auf ein einzelnes Feature oder eine spezifische Änderung beschränkt sein
- Commits sollten nur Änderungen für das jeweilige Ziel beinhalten
  - Beim Einfügen eines funktionalen Features sollten zum Beispiel keine Bugs an ganz anderen Stellen gefixt werden
  - Dient der Übersichtlichkeit und ermöglicht Nachverfolgung

Übungsaufgabe 3: Erste Commits

1. Erstellen Sie eine leere Textdatei file1.txt.
2. Führen Sie den Befehl `git status` aus. Git zeigt Ihnen an, dass file1.txt aktuell nicht von Git versioniert wird.
   Lassen Sie Git diese Datei tracken.
3. Committen Sie nun Ihre hinzugefügte Datei mit der Commit-Nachricht „Add file1.txt file“.
4. Füllen Sie nun Ihre file1.txt mit dem Inhalt „file1 content“.
5. Führen Sie erneut den Befehl `git status` aus. Ihnen wird nun angezeigt, dass file1.txt modifiziert wurde.
   Fügen Sie file1.txt zur Staging Area hinzu.
6. Erstellen Sie einen zweiten Commit um die Änderungen an file1.txt zu speichern. Wählen Sie dabei eine geeignete Commit-Message.

Identifier von Commits

- Jeder Commit besitzt einen eindeutigen 40 Zeichen Hex Identifier
- Mittels SHA-1 anhand des Contents, Autor und Commit-Infos errechnet
- Commit IDs können über den Log-Output angezeigt werden

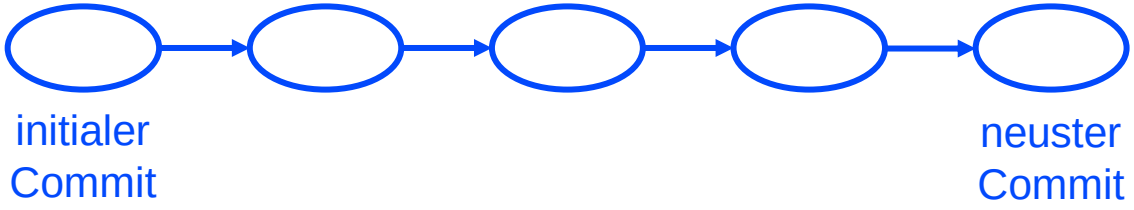
```
git log
```

- Beispiel

```
$ git log
commit e0b3cf300585c6230d1abffafe37bb4b9d540247 (HEAD -> main)
Author: Git User <git.user@example.com>
Date:   Fri May 10 00:04:39 2024 +0200

    Update file1.txt

commit c61ef14e66353d2be1d7a20a3c1eaa73d78ffe71
Author:  Git User <git.user@example.com>
Date:   Thu May 9 23:40:52 2024 +0200

    Initial commit
```

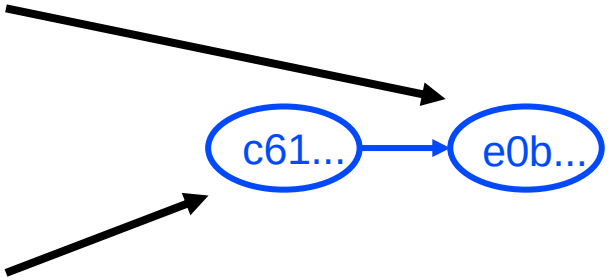
Visualisierung von Commits

- Eine Abfolge von Commits wird häufig als gerichteter Graph dargestellt
- Ein Knoten entspricht dabei einem Commit



- Beispiel

```
$ git log
commit e0b3cf300585c6230d1abffafe37bb4b9d540247
(HEAD -> main)
Author: Git User <git.user@example.com>
Date:   Fri May 10 00:04:39 2024 +0200

    Update file1.txt

commit c61ef14e66353d2be1d7a20a3c1eaa73d78ffe71
Author:  Git User <git.user@example.com>
Date:   Thu May 9 23:40:52 2024 +0200

    Initial commit
```



Grundlegende Git Befehle

`git commit <files>`	Commitet selektiv ausgewählte Dateien, umgeht Staging
`git diff`	Zeigt Unterschiede der aktuellen Dateiinhalte im Vergleich zum letzten Commit
`git diff --staged`	Wie `git diff`, jedoch werden nur gestagete Dateien betrachtet
`git rm`	Entfernt Datei aus Workspace und staged Löschung aus Git für nächsten Commit
`git mv`	Verschiebt Dateien im Workspace, Nutzung auch zum Umbenennen

Git – Grundlagen

`git reset HEAD`	Unstaged alle aktuell gestageten Dateien, Änderungen bleiben erhalten
`git checkout --`	Zurücksetzen aller getrackten Dateien auf den Stand des letzten Commits
`git checkout -- <files>`	Zurücksetzen ausgewählter Dateien auf den Stand des letzten Commits
`git restore --`	Zurücksetzen aller Dateien auf den Stand des letzten Stagings
`git restore --staged` `git restore --staged <files>`	Äquivalent zu `git checkout -- /` `git checkout -- <files>`

Übungsaufgabe 4: Umbenennen von Dateien

1. Erstellen Sie eine Datei new_file.txt mit dem Inhalt „file2 content“, lassen Sie diese durch Git tracken und committen Sie diese.
2. Benennen Sie die Datei new_file.txt über Ihr Betriebssystem in file2.txt um (Beispielsweise `mv new_file.txt file2.txt` unter Linux).
3. Führen Sie den Befehl `git status` aus. Git sollte Ihnen anzeigen, dass new_file.txt gelöscht wurde und file2.txt als untracked file zum Repository hinzugefügt wurde, was nicht direkt unsere Änderungen widerspiegelt.
4. Machen Sie die vorherige Umbenennung rückgängig und führen Sie diese erneut mit dem Befehl `git mv` aus.
   `git status` sollte Ihnen nun anzeigen, dass new_file.txt in file2.txt umbenannt wurde. Committen Sie die Änderungen.

.gitignore Datei

- Definiert von Git ignorierte Ordner oder Dateien
- Auf verschiedenen Ebenen definiert
  - Globale .gitignore in der Git-Konfiguration
  - Respository-spezifische .gitignore
  - Verzeichnis-spezifische .gitignore
- Ordner/Dateien können über Pattern angegeben werden
- Beispielsinhalt

```
# class files
*.class

# log files
*.log

# jar files
*.jar

# build and out dir
**/build
**/out
```

Übungsaufgabe 5: .gitignore

In vielen Softwareprojekten existieren Ordner wie build/ oder bin/, welche typischerweise beim Bauen bzw. beim Kompilieren von Code erzeugt werden. Diese enthalten meist plattformspezifische Binaries und werden dadurch häufig nicht in Git versioniert.

1. Erstellen Sie die beiden Ordner build und bin und legen Sie in beiden Ordnern eine Datei output.class an.
2. Führen Sie den Befehl `git status` aus. Git sollte Ihnen die neuen Ordner und Dateien als untracked anzeigen.
3. Erstellen Sie im Hauptordner eine Datei .gitignore und tragen Sie in diese den folgenden Inhalt ein.

```
**/build/
**/bin/
*.class
```

Übungsaufgabe 5: .gitignore

Durch die Einträge in der .gitignore werden alle build und bin Ordner, sowie alle .class Dateien im gesamten Projekt ignoriert.

4. Committen Sie die .gitignore Datei. Achten Sie darauf, dass die Dateien im bin bzw. build Ordner nicht von Git getracked werden.
5. Nun sollten in der Ausgabe von `git status` weder der build noch der bin Ordner und deren Dateien als untracked aufgeführt werden.
6. Legen Sie im Ordner build eine neue Datei test.txt an. Auch diese sollte nicht durch git getracked werden und damit bei `git status` auch nicht aufgeführt werden.

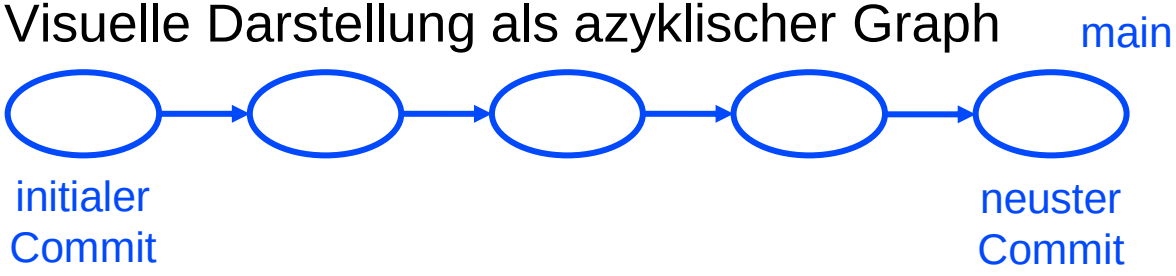
Git

Branches

Git – Branches

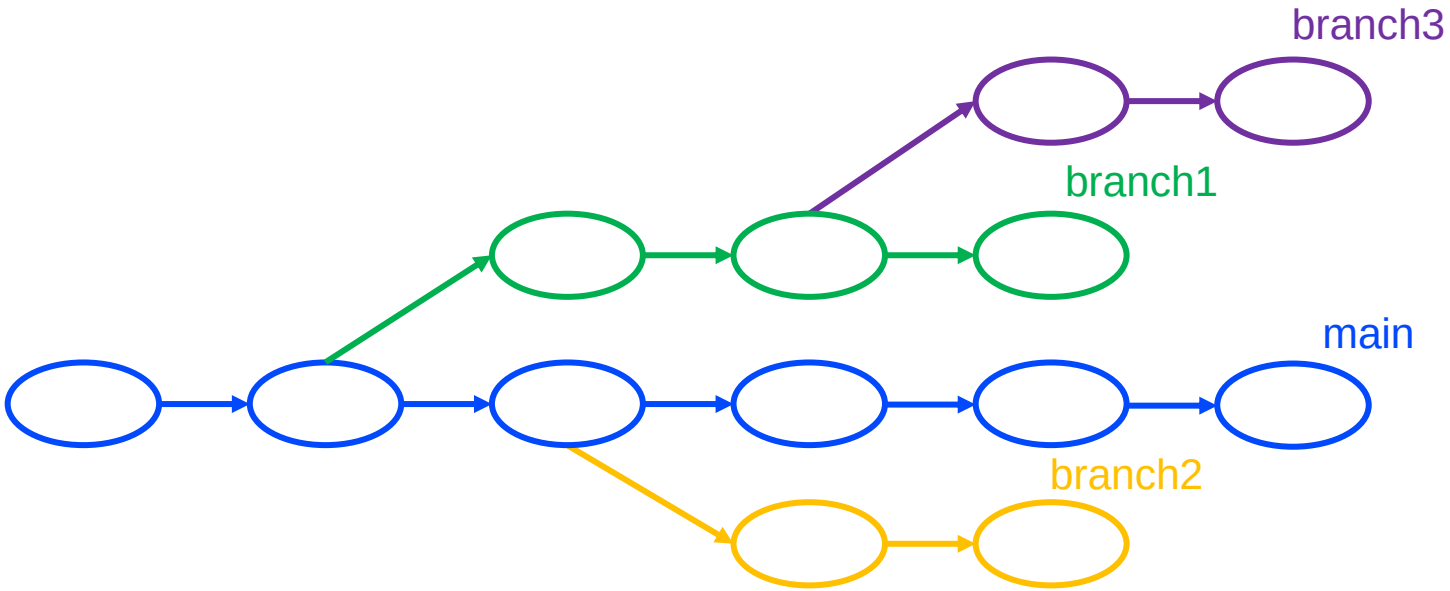
Was sind Branches?

- Branches sind Entwicklungszweige und bestehen aus einer Reihe von einzelnen Commits
- Git arbeitet immer in Branches
- Beim Initialisieren eines Repository mittels `git init` wird automatisch ein initialer *main* Branch angelegt
  - Bis 2020 war der Standardname *master,* aufgrund von offensive Language umbenannt
  - Standardname konfigurierbar mittels
    `git config --global init.defaultBranch <branch-name>`
- Visuelle Darstellung als azyklischer Graph



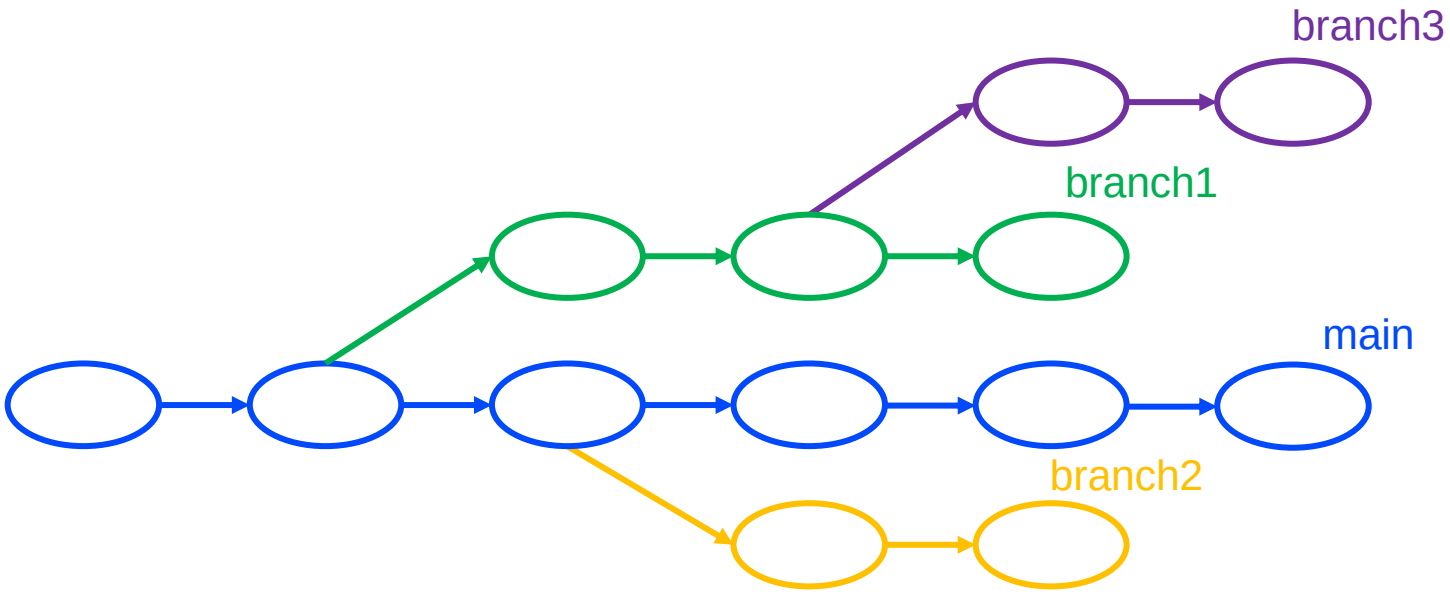
Git – Branches

- Verschiedene Branches ermöglichen paralleles Arbeiten
  - Funktionale Komponenten in separaten Branches entwickeln
  - Bugfixes
  - Hotfixes für spezielle Versionen
  - Versionierung (bspw. Separater Branch für jedes Release/Version)
- Abzweigung von jedem Commit aus möglich
- Beispielgraph



Git – Branches

- Es gibt viele unterschiedliche Möglichkeiten und Ansätze, um Branches zu verwenden
- Aufbau und Nutzung stark von individuellen Bedürfnissen abhängig
- Genaue Abläufe und Beispiele werden im Rahmen von Workflows noch besprochen



Git – Branches

anderScore trust in competence

Grundlegende Befehle für Branches

- Branches des aktuellen Projektes anzeigen mittels

```
git branch
```

- Beispiel

```
$ git branch
  feature1
  feature2
  hotfix1
* Main
```

- Der aktive Branch wird mit einem Stern und ggf. farblich markiert

Git – Branches

anderScore trust in competence

Weitere `git branch` Befehle

`git branch <branch-name>`	Erzeugt neuen Branch ausgehend vom letzten Commit des aktiven Branches
`git branch <branch-name> <source-branch>`	Erzeugt neuen Branch ausgehend vom letzten Commit des angegebenen Source Branches
`git branch <branch-name> <commit-id>`	Erzeugt neuen Branch ausgehend vom spezifzierten Commit

Git – Branches

`git checkout <branch-name>`	Wechselt den aktiven Branch zum angegeben Branch
`git checkout -b <branch-name>`	Erstellt angegebenen Branch falls dieser nicht existiert und setzt ihn als aktiven Branch
`git switch <branch-name>`	Äquivalent zu `git checkout <branch-name>`
`git switch -c <branch-name>`	Äquivalent zu `git checkout -b <branch-name>`
`git log --graph --all`	Ausgabe einer grafischen Repräsentation der Branches auf der Konsole

Git – Branches

anderScore trust in competence

switch / restore

- Die Befehle `git switch` und `git restore` sind aus dem Befehl `git checkout` entstanden
- Ziel ist Mehrdeutigkeit von `git checkout` aufzulösen
  - `git checkout <branch>` ➔ Wechsel von Branches
  - `git checkout --` ➔ Zurücksetzen von Änderungen
- `git checkout` trotzdem immer noch weit verbreitet

Git – Branches

anderScore trust in competence

Beispiel neuen Branch anlegen und direkt wechseln

```
$ git checkout -b feature3
Switched to a new branch 'feature3’

$ git branch
  feature1
  feature2
* feature3
  hotfix
  main

$ git status
On branch feature3
nothing to commit, working tree clean
```

Git – Branches

Übungsaufgabe 6: Branches

1. Erstellen Sie im Projekt einen Ordner features.
2. Erstellen Sie einen neuen Branch feature1 und wechseln Sie in diesen.
3. Legen Sie im Ordner features eine Datei feature1_file1.txt mit beliebigem Inhalt an und committen Sie Ihre Änderungen.
4. Erstellen Sie eine zweite Datei feature1_file2.txt im Ordner an und committen Sie auch diese.
5. Wechseln Sie zurück auf den main Branch. Da Git den Workspace an den aktiven Branch anpasst, sollte der Ordner features hier leer bzw. nicht vorhanden sein.

Git – Branches

anderScore trust in competence

Aktiver Branch

- Im lokalen Workspace gibt es immer einen aktiven Branch
  - Zu Beginn der Default Branch
  - Kann durch `git checkout <branch-name>` gewechselt werden
  - Alle Commits werden auf dem aktiven Branch ausgeführt
- Beim Wechsel des Branches wird der Inhalt sämtlicher Dateien im Workspace auf den Stand des Zielbranches geändert
- Anzeige des aktiven Branches über

  `git status` oder über `git branch`
- Beispiel

```
$ git status
On branch main
nothing to commit, working tree clean
```

Git – Branches

Wechseln zwischen Branches

- Wechseln des Branches mit gestageten Änderungen oder veränderten getrackten Dateien kann zu Problemen führen
- Durch Änderung des Workspaces auf den Zielbranch würden Änderungen verloren gehen
- Git warnt Nutzer, sollten durch den Wechsel Änderungen verloren gehen
- Ausnahme: Existiert Änderung nur im Ausgangsbranch, so kann man den Branch wechseln
  - Änderung schließt Erstellen und Löschen von Dateien mit ein
  - Änderungen werden aus dem alten Branch in den neuen Branch übernommen (Inhaltlich sieht es nach einem Commit im Zielbranch aus, als hätte man die Änderungen direkt im Zielbranch ausgeführt)

Git – Branches

Beispiel unveränderter Zielbranch

```
$ git status
On branch feature1
nothing to commit, working tree clean

$ ls
file1.txt file2.txt subdir

$ git checkout main
Switched to branch 'main’

$ touch main_file1.txt

$ git add main_file1.txt

$ git checkout feature1
A       main_file1.txt
Switched to branch 'feature1’

$ ls
file1.txt file2.txt main_file1.txt subdir
```

- *main_file1.txt* existiert nur auf dem *main* Branch, daher kann der Branch gewechselt werden

Git – Branches

anderScore trust in competence

Beispiel veränderter Zielbranch

```
$ git status
On branch feature1
nothing to commit, working tree clean

$ echo "hello from feature1" > shared_file.txt

$ git add shared_file.txt

$ git commit -m "add share file"
[feature1 d8cb253] add share file
 1 file changed, 1 insertion(+)
 create mode 100644 shared_file.txt

$ git checkout main
Switched to branch 'main’

$ ls
file1.txt  file2.txt  subdir

$ echo "hello from main" > shared_file.txt

$ git add shared_file.txt
```

Git – Branches

anderScore trust in competence

Bespiel veränderter Zielbranch

```
$ git checkout feature1
error: Your local changes to the following files would be overwritten by checkout:
        shared_file.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

- Git verweigert direkten Wechsel
- Mögliche Optionen
  - Änderungen committen
  - Änderungen verwerfen und Checkout mittels `git checkout -f <branch>` (`-f` oder `--force`) erzwingen
  - Änderungen in „stashing area“ zwischenspeichern und diese später wiederherstellen mittels `git stash`

Git – Branches

anderScore trust in competence

HEAD

- HEAD ist eine spezielle Referenz, die immer auf den letzten Commit des aktiven Branch verweist
- Bespiel

```
$ git log
commit e0b3cf300585c6230d1abffafe37bb4b9d540247 (HEAD -> main)
Author: Git User <git.user@example.com>
Date:   Fri May 10 00:04:39 2024 +0200

    Update file1.txt
...
```

- Bei neuen Commits wandert die HEAD Referenz weiter
- Beim Wechsel von Branches wird der HEAD Zeiger auf letzten Commit des neuen Branches verschoben

Detached HEAD

- Detached HEAD bedeutet, dass der HEAD nicht auf den letzten Commit des aktiven Branches verweist
- Eine Möglichkeit ist das Auschecken einzelner Commits

```
$ git checkout c61ef14
Note: switching to 'c61ef14'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at c61ef14 Initial commit
```

Git – Branches

anderScore trust in competence

Übungsaufgabe 7: Branches

Branches können nicht nur vom letzten Commit des aktuellen Branch abzweigen, sondern von jedem beliebigen Commit.

1. Lassen Sie sich über `git log` die Commit-IDs ausgeben und erstellen Sie einen Branch feature2 ausgehend vom vorletzten Commit des main Branches.
2. Legen Sie erneut den Ordner features, sowie eine Datei feature2_file1.txt an und committen Sie Ihre Änderungen.
3. Wechseln Sie zurück auf den main Branch. Der features Ordner ist hier immer noch nicht vorhanden bzw. leer.

Git

Reset & Revert

Git – Reset

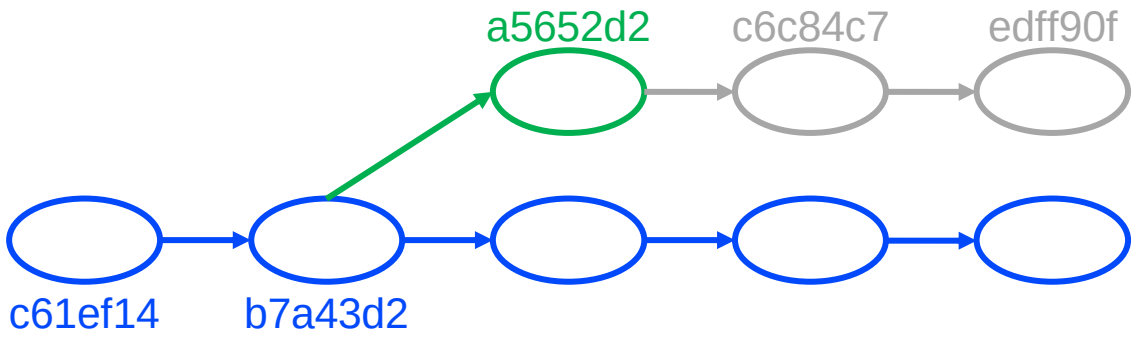
Reset

- Reset auf bestimmte Commits mittels

```
git reset [<mode>] [<commit>]
```

- Setzt den HEAD auf angegebenen Commit
- Wird kein Commit übergeben, wird der letzte Commit verwendet
- Verändert Commit Historie
- Commits werden nicht gelöscht, jedoch nicht mehr referenziert

Git – Reset




```
$ git reset a5652d2

$ git log --oneline
a5652d2 (HEAD -> feature1) Add feature_file1
c61ef14 Initial commit
```

Git – Reset

Reset

- Kann in verschiedenen Modi betrieben werden

```
git reset [<mode>] [<commit>]
```

- --soft
  - Resetet nur die Commit Historie
  - Staging Bereich und Workspace bleiben unverändert
  - Änderungen der vorherigen Commits bleiben erhalten und können neu committet werden
  - Ermöglicht Zusammenfassung der Änderungen vorheriger Commits in einem einzigen, neuen Commit
- --mixed
  - Default Auswahl
  - Resetet die Commit Historie und den Staging Bereich
  - Änderungen im Workspace bleiben erhalten

Git – Reset

- --hard
  - Resetet Commit Historie, Staging Area und Workspaces
  - Führt zum Verlust der Änderungen der übersprungenen Commits
  - Commits werden nur aus Historie entfernt und nicht mehr referenziert, aber nicht gelöscht
- `git reset` kann auch in die entgegengesetzt verwendet werden

```
$ git log --oneline
a5652d2 (HEAD -> feature1) Add feature_file1
c61ef14 Initial commit

$ git reset edff90f

$ git log --oneline
edff90f (HEAD -> feature1) Add feature_file3
c6c84c7 Add feature_file2
a5652d2 Add feature_file1
c61ef14 Initial commit
```

Git –Revert

Revert

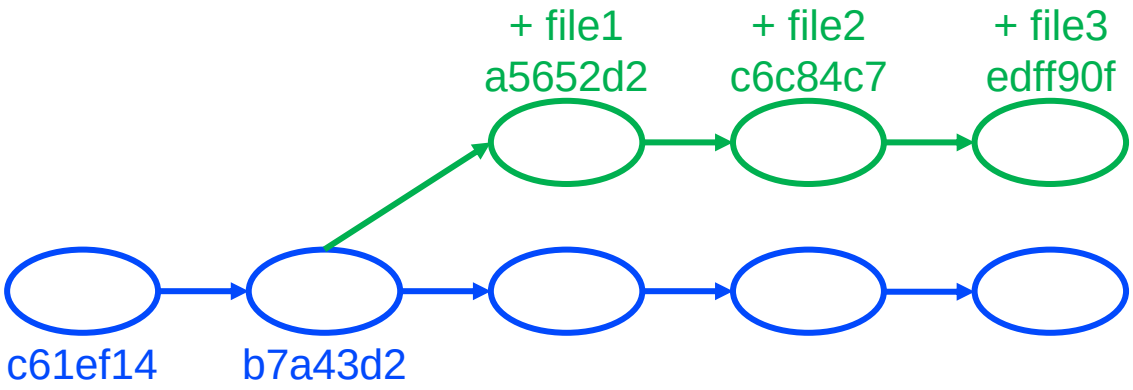
- Andere Möglichkeit ist

  ```
  git revert <commit>…
  ```

- Macht Änderungen der angegebenen Commits rückgängig
- Nutzt dafür für jeden zurückgesetzten Commit einen neuen Commit
- Verändert zurückliegende Commit Historie nicht

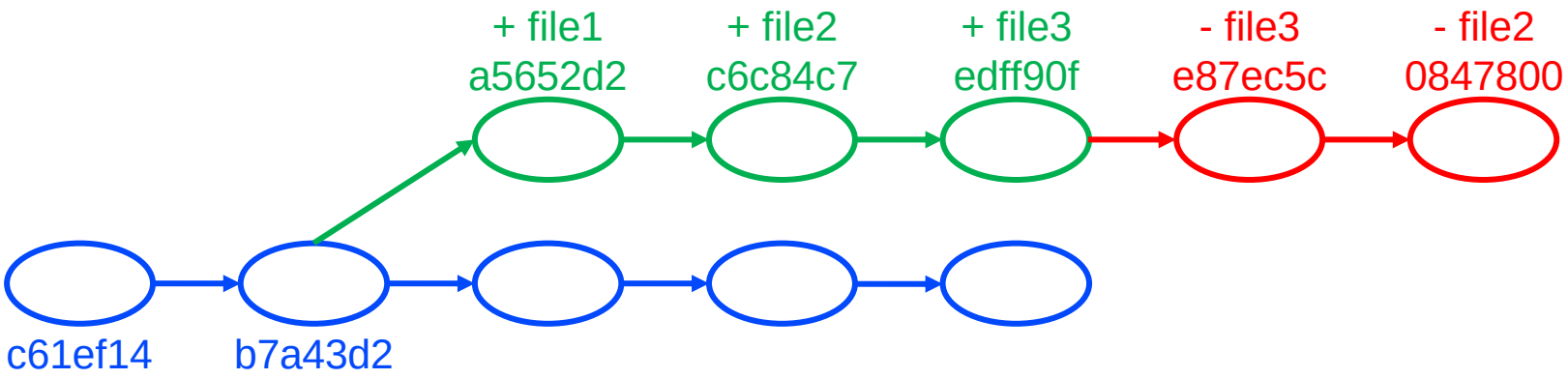
  ➔ Commit Historie bleibt erhalten
- Workspace darf keine uncommitteten Änderungen beinhalten

Git – Revert



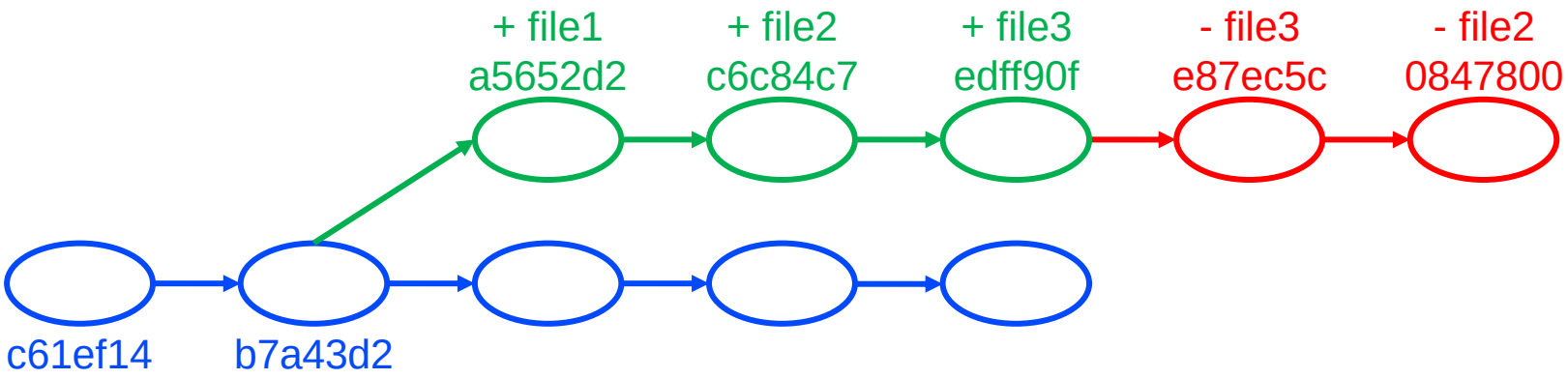

```
$ git log --oneline
edff90f (HEAD -> feature1) Add feature_file3
c6c84c7 Add feature_file2
a5652d2 Add feature_file1
b7a43d2 Add main_file1
c61ef14 Initial commit
```

Git – Revert




```
$ git revert edff90f c6c84c7
[feature1 e87ec5c] Revert "Add feature_file3"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 feature_file3.txt
[feature1 0847800] Revert "Add feature_file2"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 feature_file2.txt
```

Git – Revert




```
$ git log --oneline
0847800 (HEAD -> feature1) Revert "Add feature_file2"
e87ec5c Revert "Add feature_file3"
edff90f Add feature_file3
c6c84c7 Add feature_file2
a5652d2 Add feature_file1
c61ef14 Initial commit
```