

# Codespaces

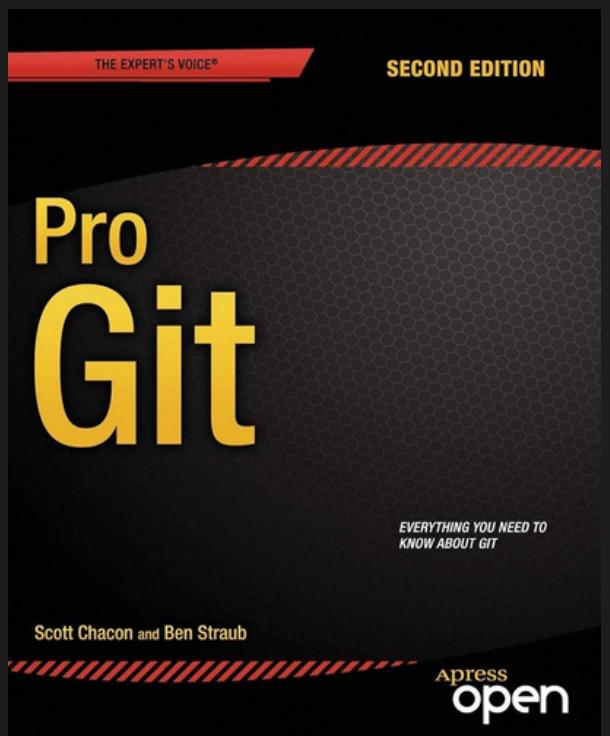
In Code We Trust.



the stupid content tracker



# Git Universum





# Agenda



Hinweise: Bei diesem Symbol findest du Hinweise zum Inhalt der Folie.

## *Symbol Legende*



open end  
60min

# Inhalt

Intro-Themen

---

Theorie

---

Praxis - Opensource VSCode

---

Diskussion / Fragen

---

# Was ist das Problem?

— git ist die Lösung

# Was ist das Problem?

- Überschreiben statt Nachvollziehen – „script\_final\_v3\_reallyFINAL.js“-Chaos
- Kein sicheres Zurück – Fehler landen direkt im Produktiv-Code
- „Wer hat das geändert?“ – Verantwortliche und Gründe bleiben im Dunkeln
- Geteilte ZIP-Backups – verstreut, uneinheitlich, oft veraltet
- Gleichzeitig am selben File – manuelle Kopien → Konflikt-Hölle
- Unscharfe Releases – unklar, welche Version in Test/Prod läuft
- Single Point of Failure – zentraler Server oder Laptop fällt aus → Daten weg

# Lösung: Version Control System (VCS)

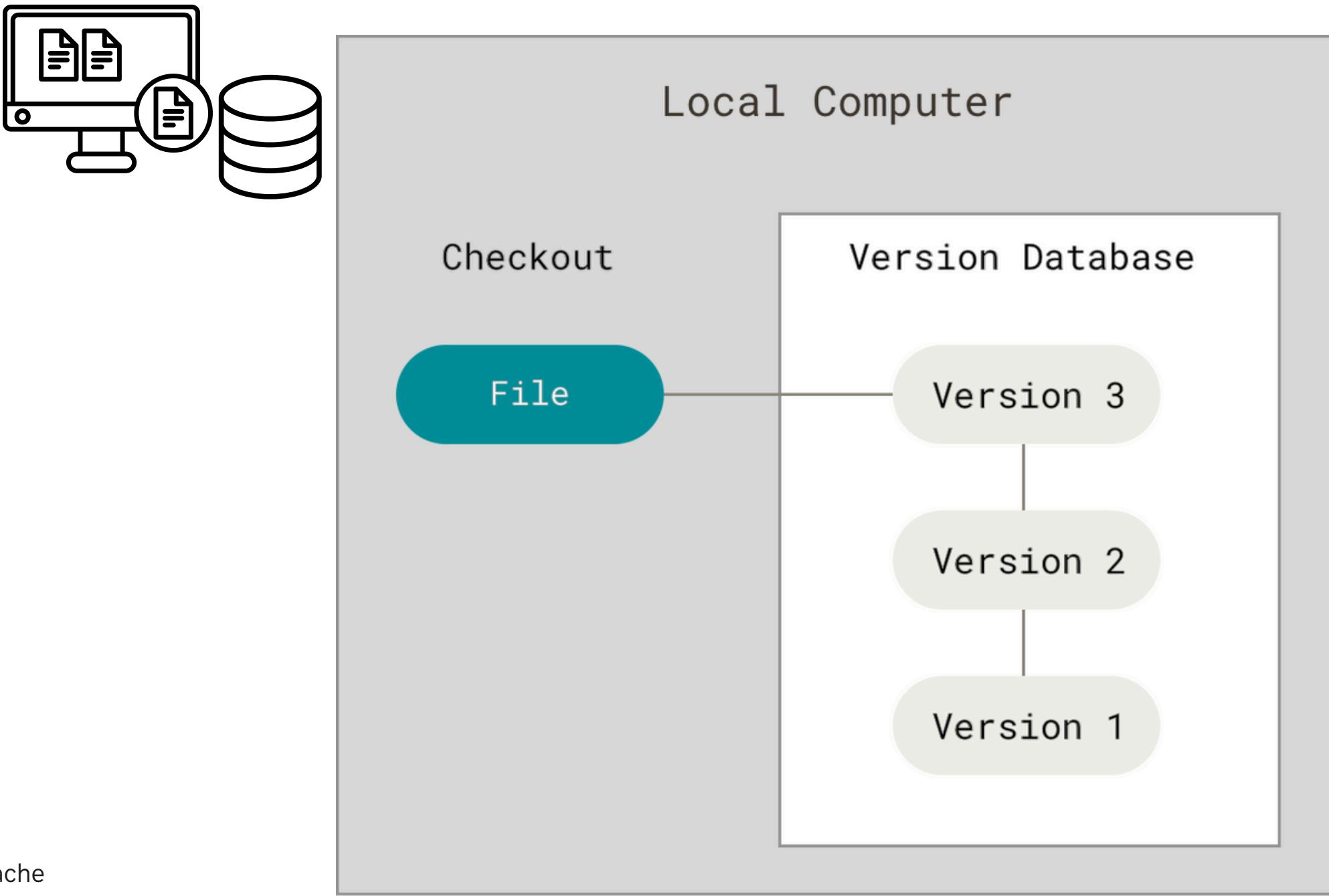


Kennst du noch **Subversion**,  
**Mercurial** oder **Concurrent**  
**Version System**?

# Evolution 'VCS'

— Typen von Version Control Systems

# Local Version Control System



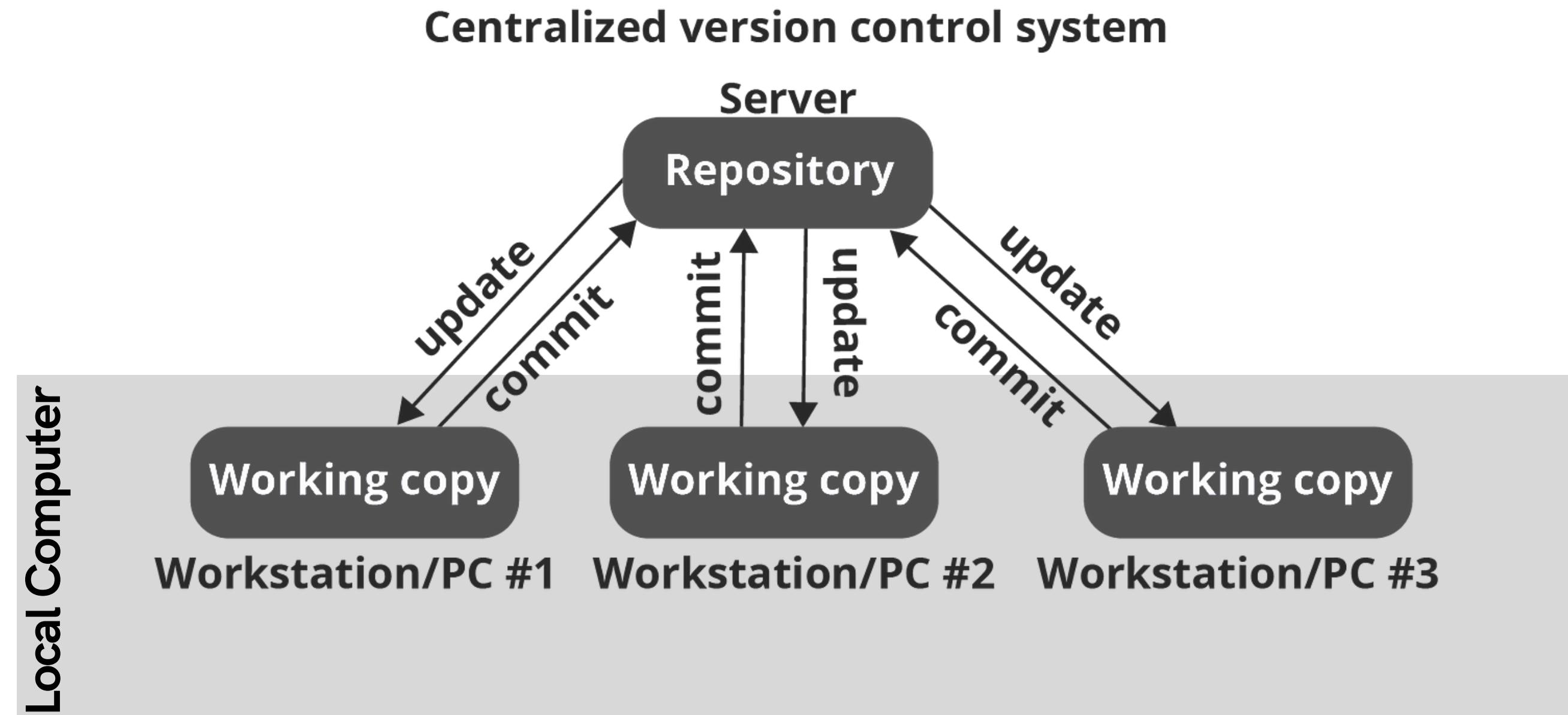
Nutzung: Einzelperson, einfache  
Projekte.  
Schnell und einfach aber kein Team-  
Workflow, kein Backup



💡 Repository = Projektordner +  
Versionsgeschichte  
„Ablage“, „Speicher“ oder „Archiv“



# Central Version Control System



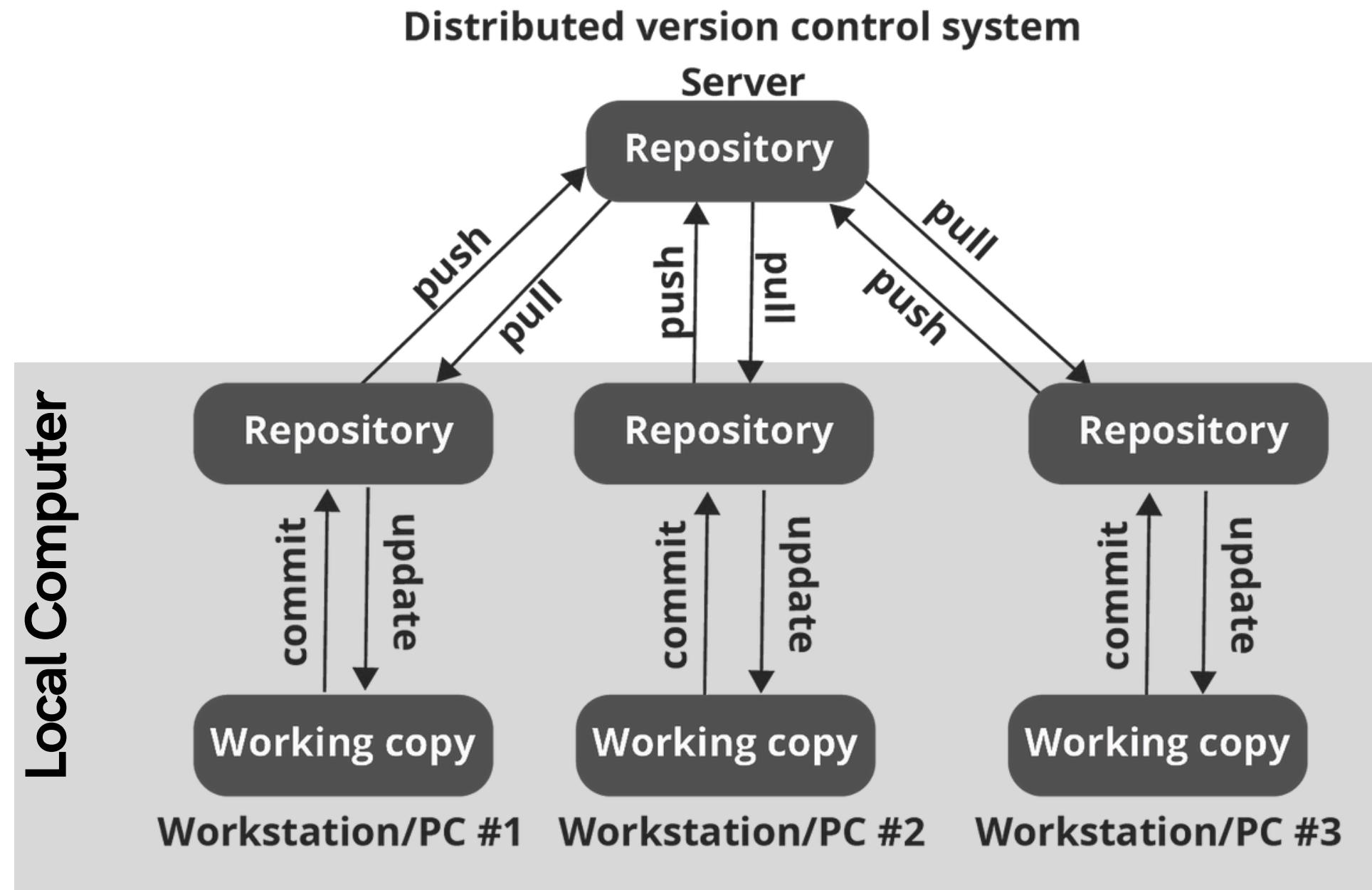
Nutzung: Teams mit zentralem Server.  
Einfache Verwaltung, zentraler Zugriff  
dafür Single Point of Failure,  
eingeschränkte Offline-Arbeit



Die History existiert nur auf dem Server;  
somit sind offline keine commits, diffs oder  
die Wiederherstellung alter Versionen  
möglich



# Distributed Version Control System



Nutzung: Teams, OpenSource, Offline  
Offline möglich, keine zentrale Abhängigkeit,  
schneller  
dafür Komplexer und mehr  
Speicherverbrauch



Offline arbeiten: Voll möglich – dank  
lokaler Projekt-History. Committen,  
Branches erstellen, vergleichen,  
zurückrollen: alles geht ohne Server. Nur  
Push/Pull braucht Verbindung.



# Was ist Git?

— Wie unterscheidet sich git und was sind die Vorteile

# Git in a Nutshell

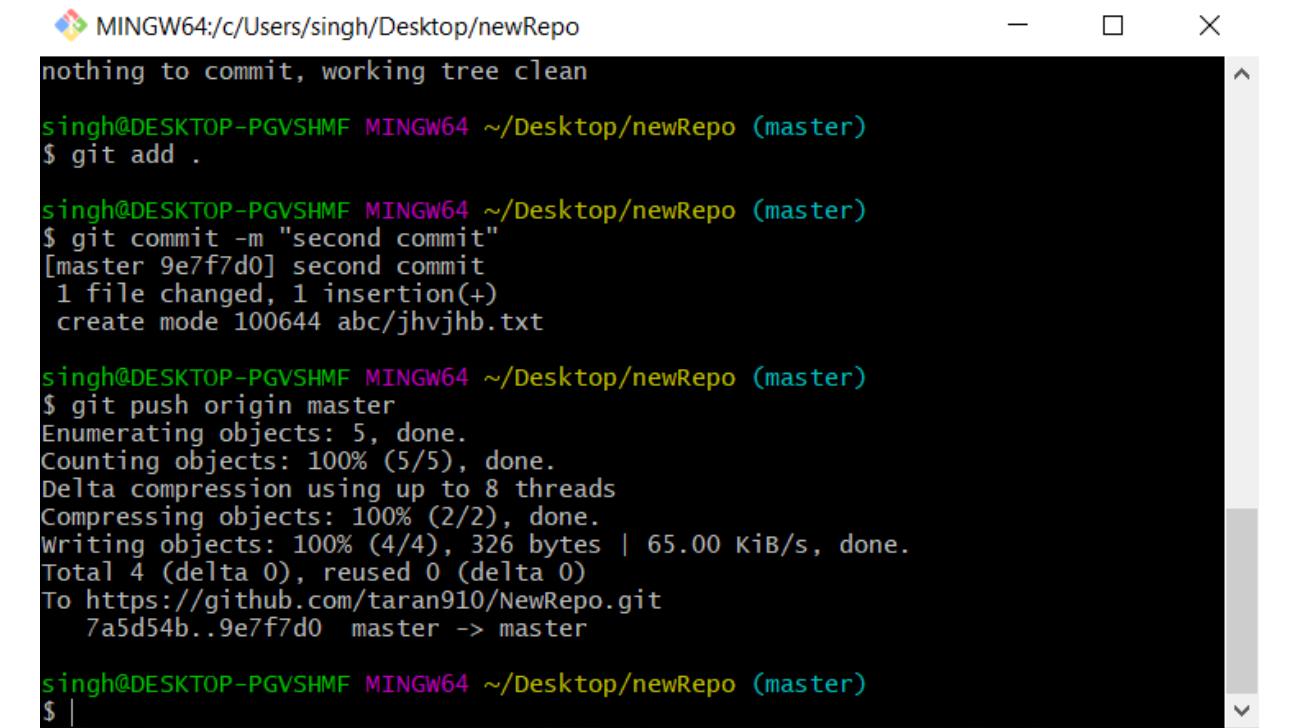
Distributed Version Control System

CLI-Tool mit optionalen GUIs

Open Source

Offline Modus möglich

“Guenstige, lokale, wegwerfbare” Branches



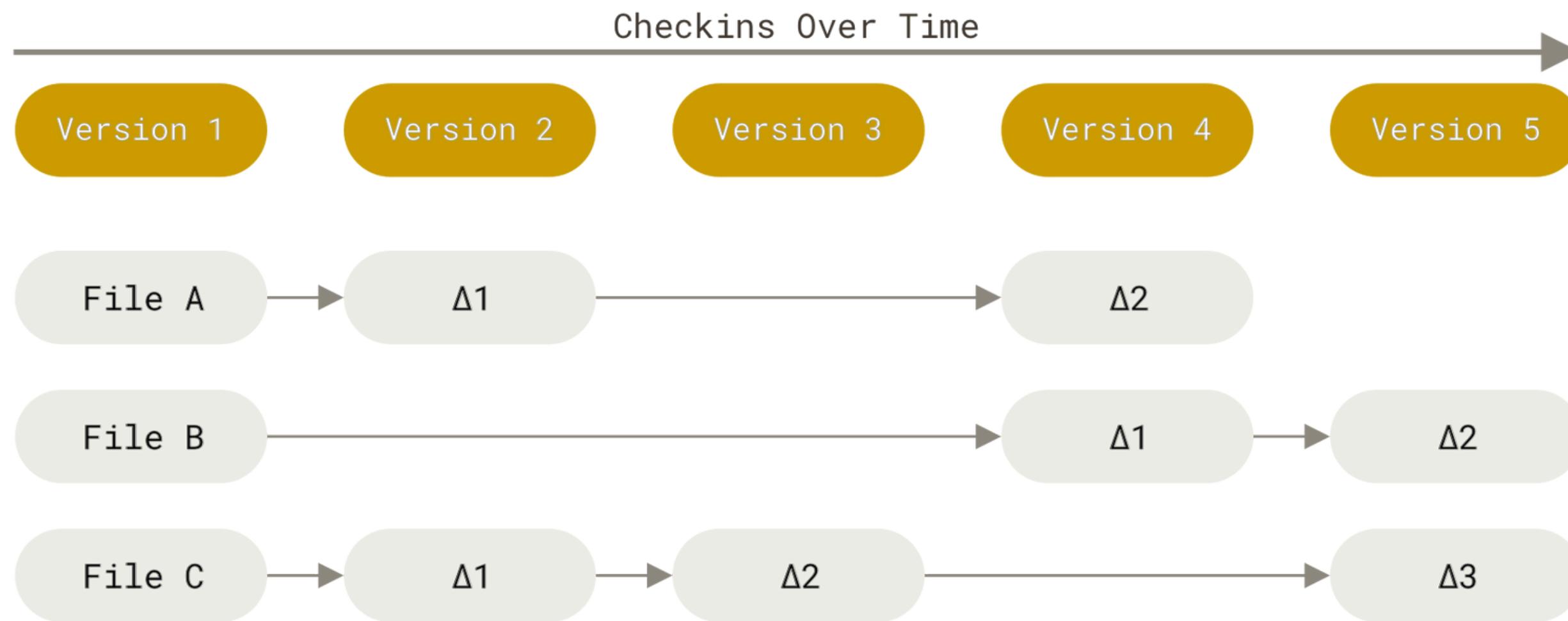
```
MINGW64:/c/Users/singh/Desktop/newRepo
nothing to commit, working tree clean
singh@DESKTOP-PGVSHMF MINGW64 ~/Desktop/newRepo (master)
$ git add .

singh@DESKTOP-PGVSHMF MINGW64 ~/Desktop/newRepo (master)
$ git commit -m "second commit"
[master 9e7f7d0] second commit
 1 file changed, 1 insertion(+)
 create mode 100644 abc/jhvjhbc.txt

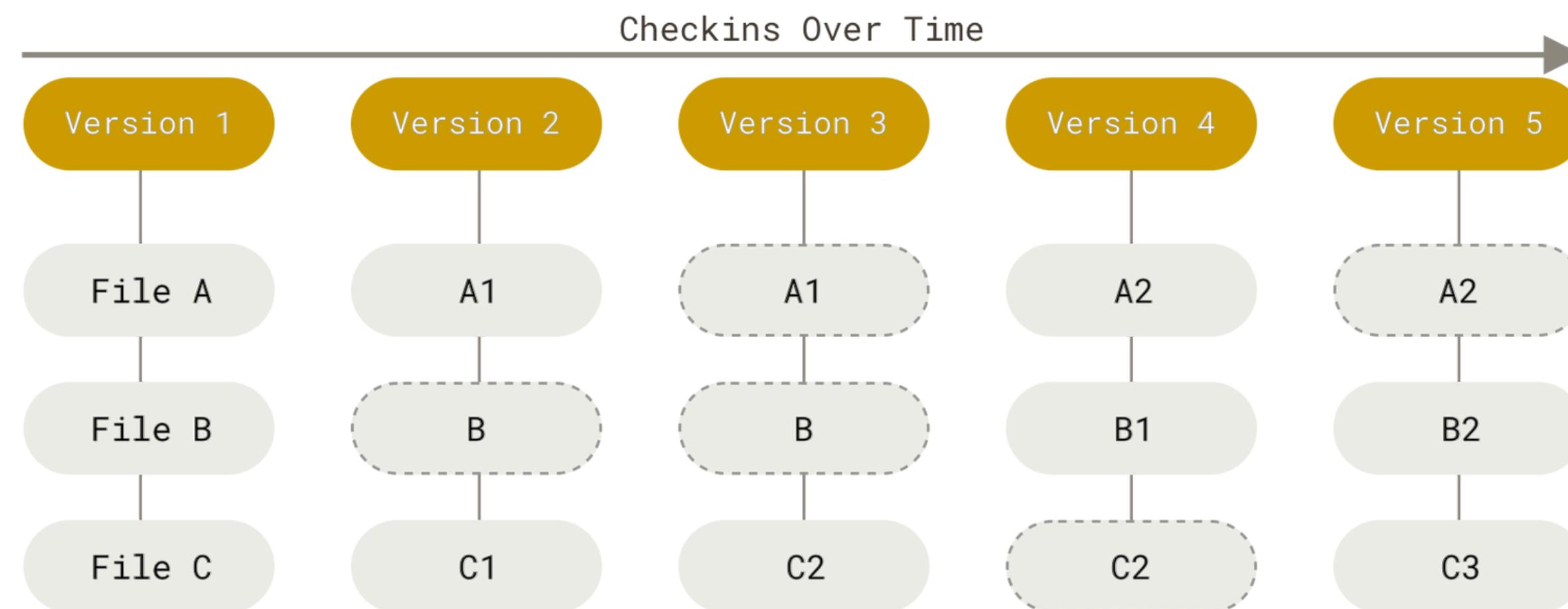
singh@DESKTOP-PGVSHMF MINGW64 ~/Desktop/newRepo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 326 bytes | 65.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/taran910/NewRepo.git
 7a5d54b..9e7f7d0 master -> master

singh@DESKTOP-PGVSHMF MINGW64 ~/Desktop/newRepo (master)
$ |
```

# Git VS die Anderen



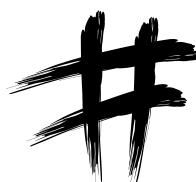
# Git VS die Anderen



# Drei Vorteile



Fast jede Funktion arbeitet lokal



Git stellt Integrität sicher



Git fügt im Regelfall nur Daten hinzu

Git sichert Integrität durch SHA-1-Hashes für

alle Dateien und Verzeichnisse. Somit kann

Git Änderungen oder Datenschaedigung

erkennen. Gilt beim Speichern, Uebertragen

(Push/Pull) und Abrufen (Checkout).

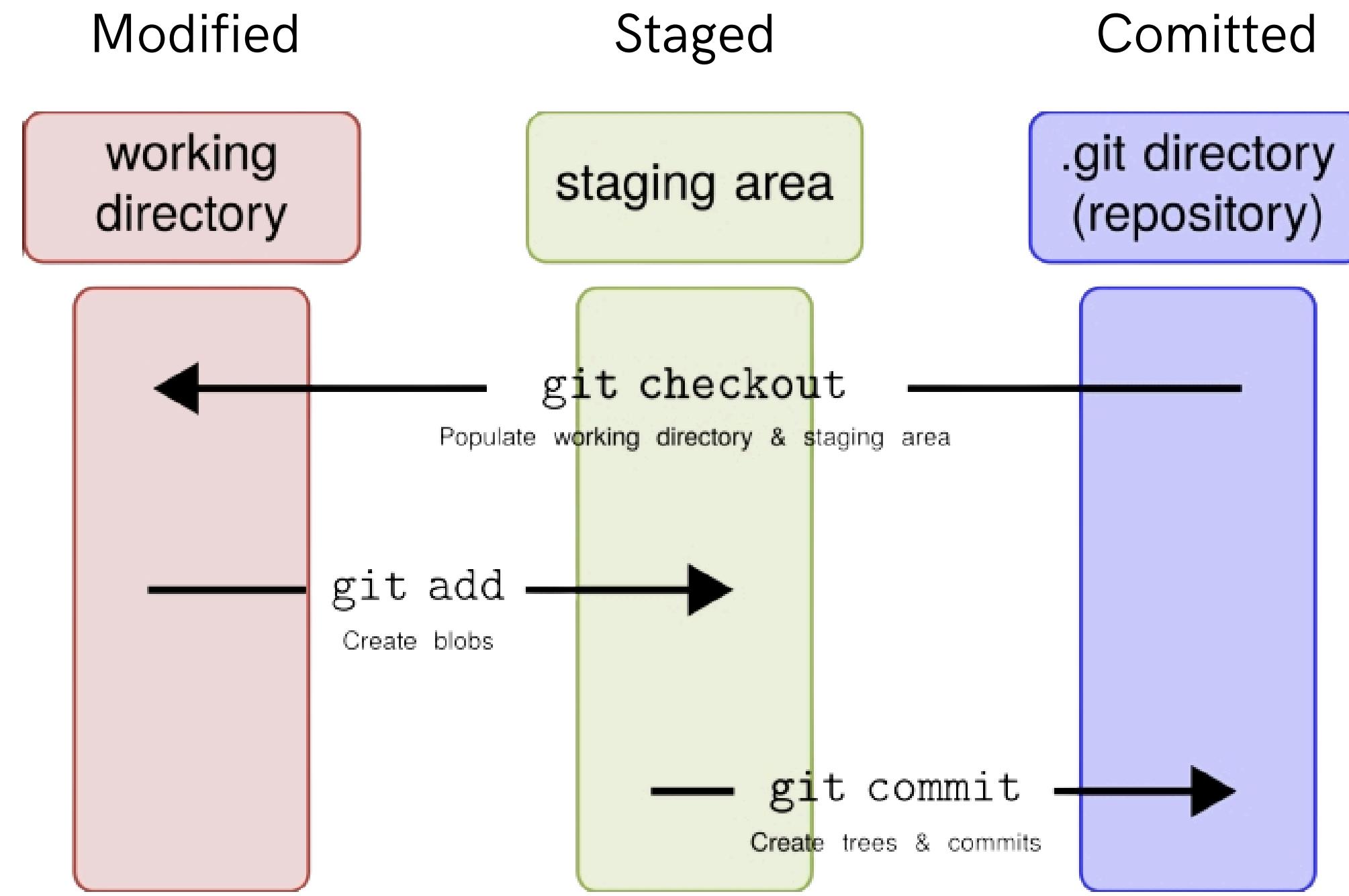


**Wichtig:** Git identifiziert Inhalte anhand ihres

Hashes, nicht ihres Namens - das verhindert

inkonsistente Zustände

# Drei Zustände



git ist der CLI von git.

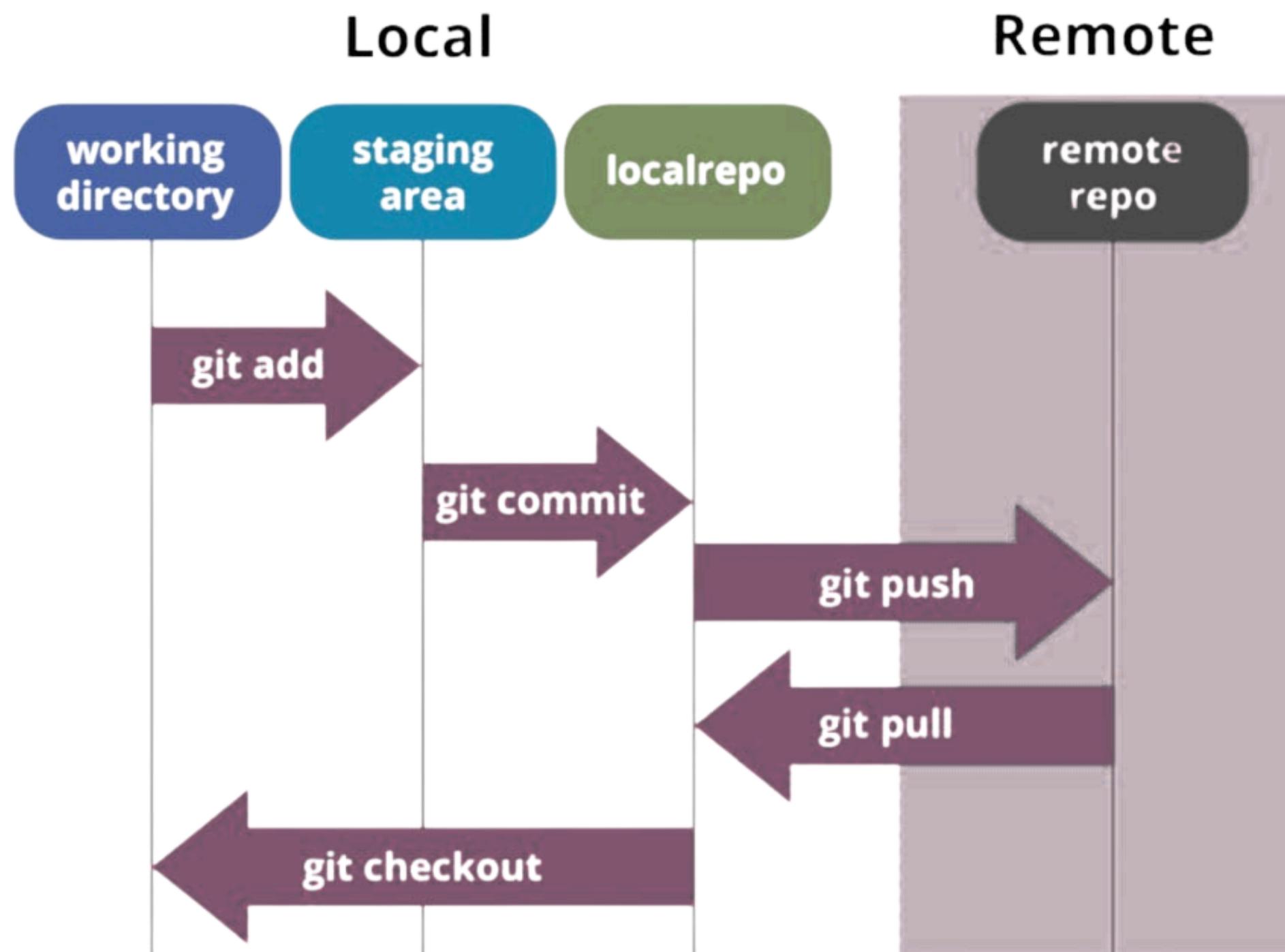
git checkout wechselt den Branch.

git add fügt Datei/Dateien zum Commit vormerken

git commig fügt Änderung hinzu



# + Remote Zustand



# Server

— Git Server

# Git auf dem Server

Kein spezieller "Git-Server" notig

"bare" Projekt Ordner

Zugriffsprotokolle: SSH/HTTPS



GitLab



**Web-UI, Rechteverwaltung, Issues, Merge Request / Pull Request, CI/CD**

# Geschichte

— Es ist Zeit für eine Gutenachtgeschichte. Es war einmal Linus Torvalds...

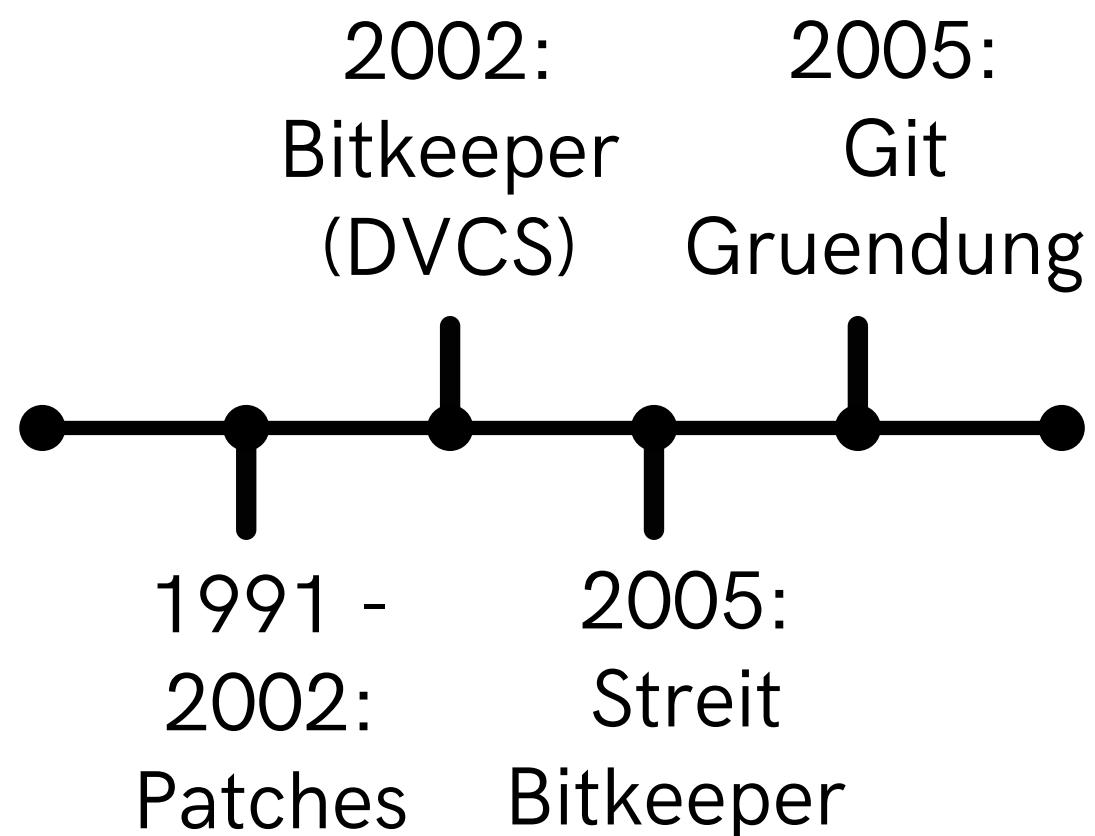
# Linus Torvalds

„Git“ ist kein Akronym – der Name hat keine offizielle Bedeutung.



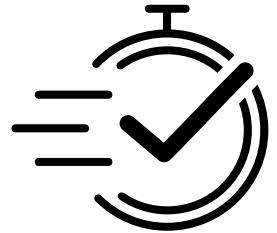
Linus sagte selbstironisch, „git“ könne auch „Idiot“ oder „Blödmann“ bedeuten (britischer Slang) – in Anspielung auf sich selbst.

I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'

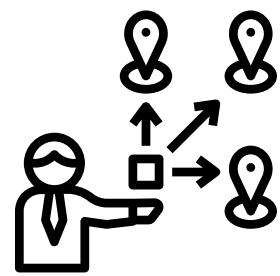


Erfinder des Linux-Kernels (1991) und Schöpfer von Git (2005).

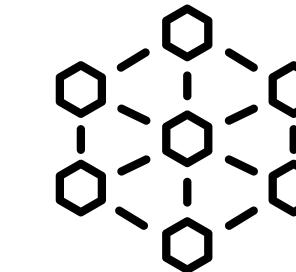
# Eigenschaften von Git



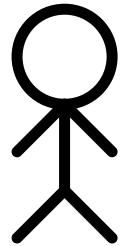
Geschwindigkeit



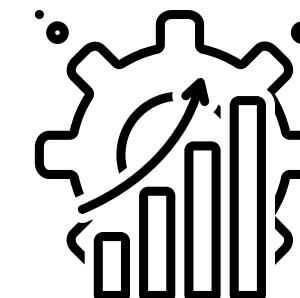
parallele Entwicklungs-Branches



Vollständig dezentrale Struktur



Einfaches Design



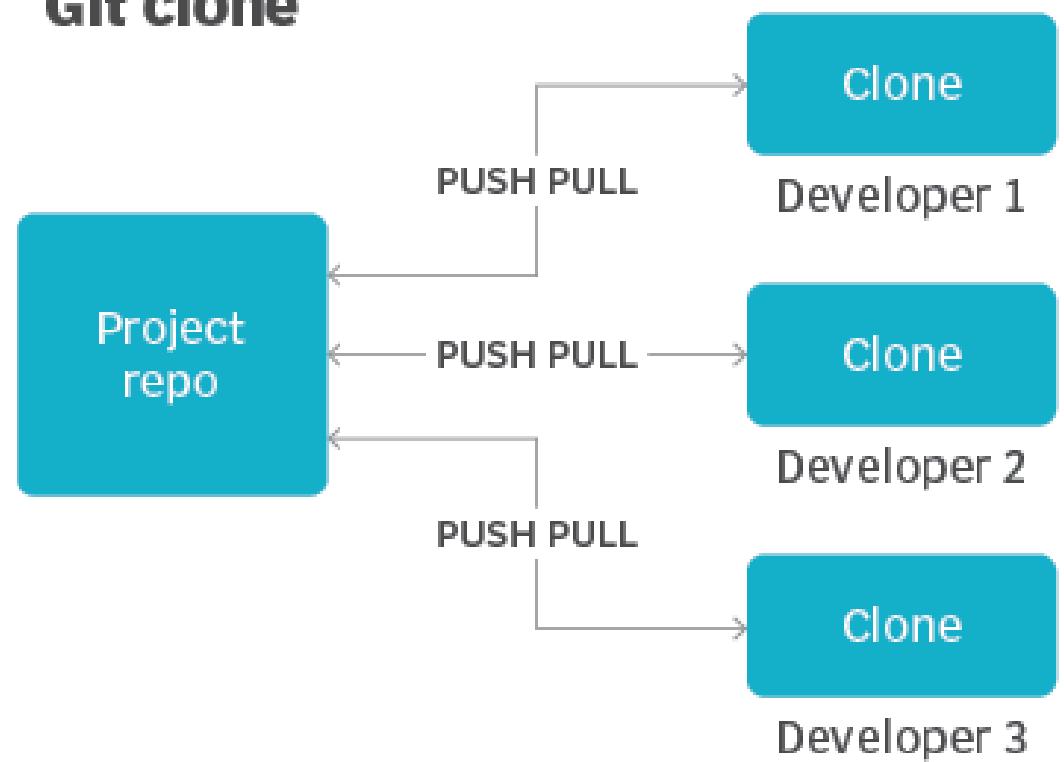
Fähigkeit, grosse Projekte, wie den Linux Kernel, effektiv zu verwalten  
(Geschwindigkeit und Datenumfang)

# Fork vs Clone

— Kopie: Serverseitig vs Lokal

# Fork vs Clone

## Git clone

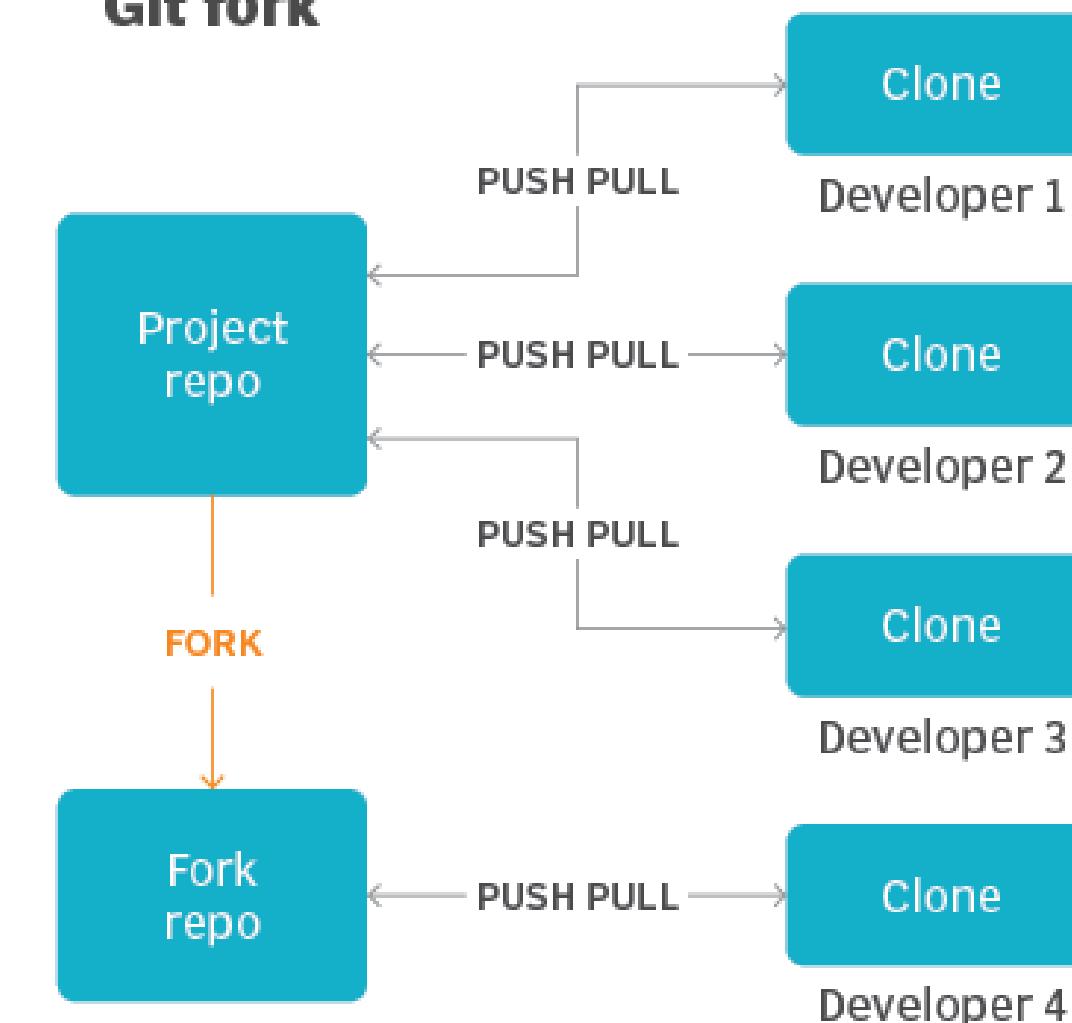


Ein Fork erstellt eine Kopie eines Repositories auf dem Remote-Server, um unabhängig zu arbeiten und Änderungen per Pull Request zurückzugeben. Ein Clone lädt das Repository lokal auf den Rechner, um Dateien zu bearbeiten und zu entwickeln.

Forks nutzt man bei fremden Projekten ohne Schreibrechte, Clone für lokale Arbeit. Fork ist die Remote-Kopie, Clone die lokale. Zusammen ermöglichen sie effiziente Zusammenarbeit.



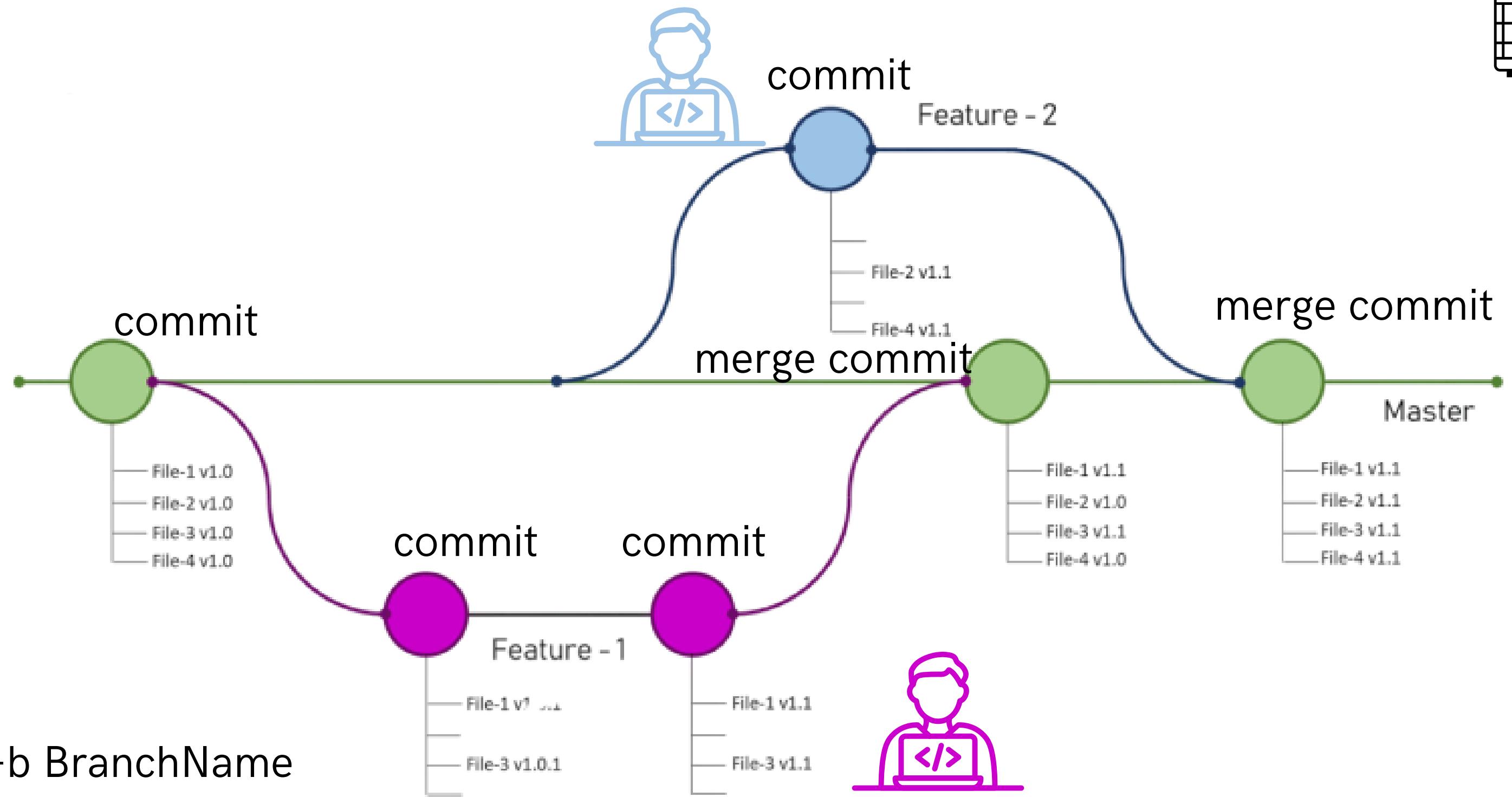
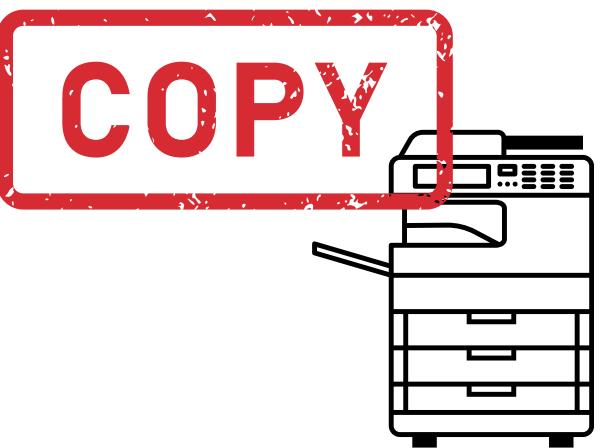
## Git fork



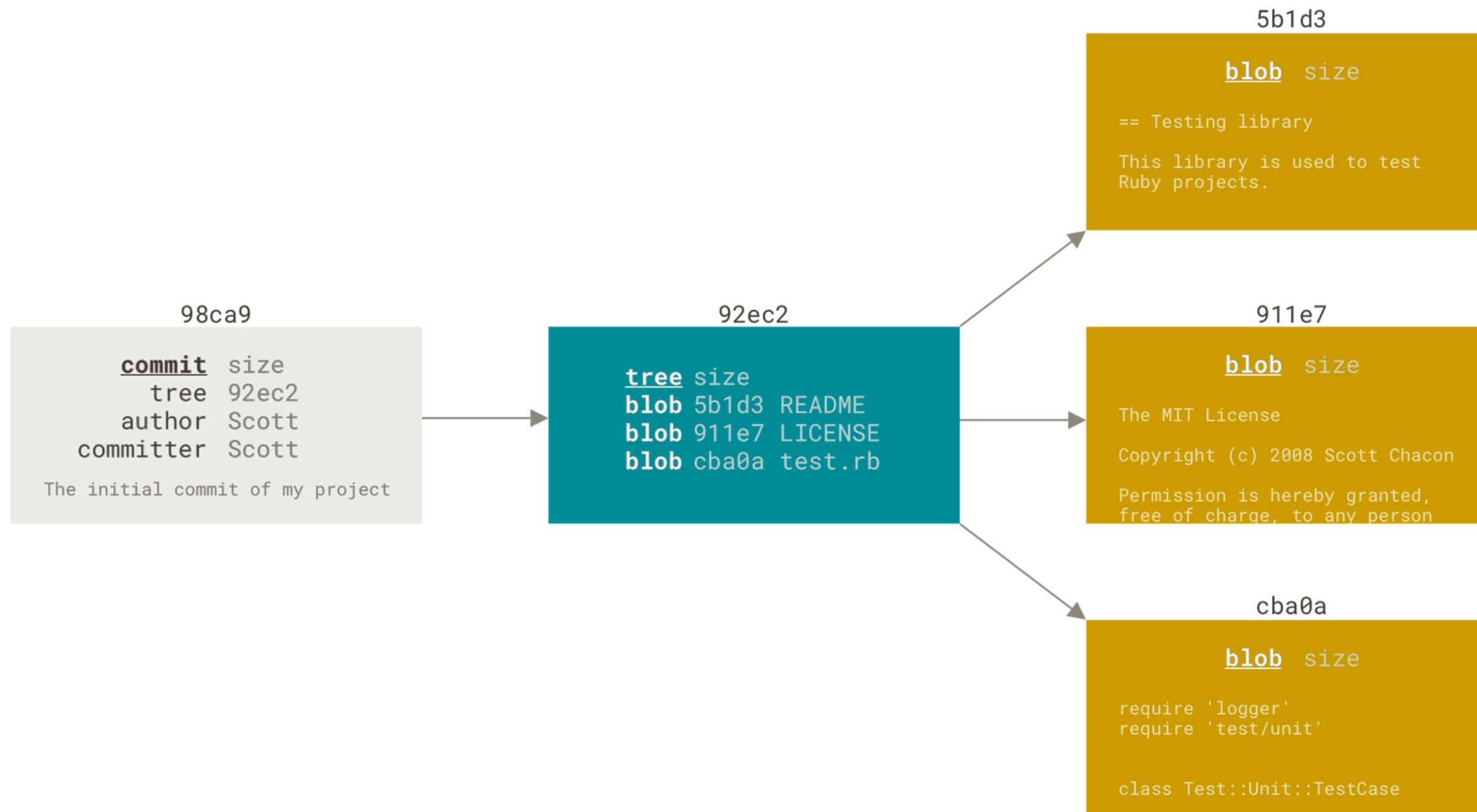
# Branches

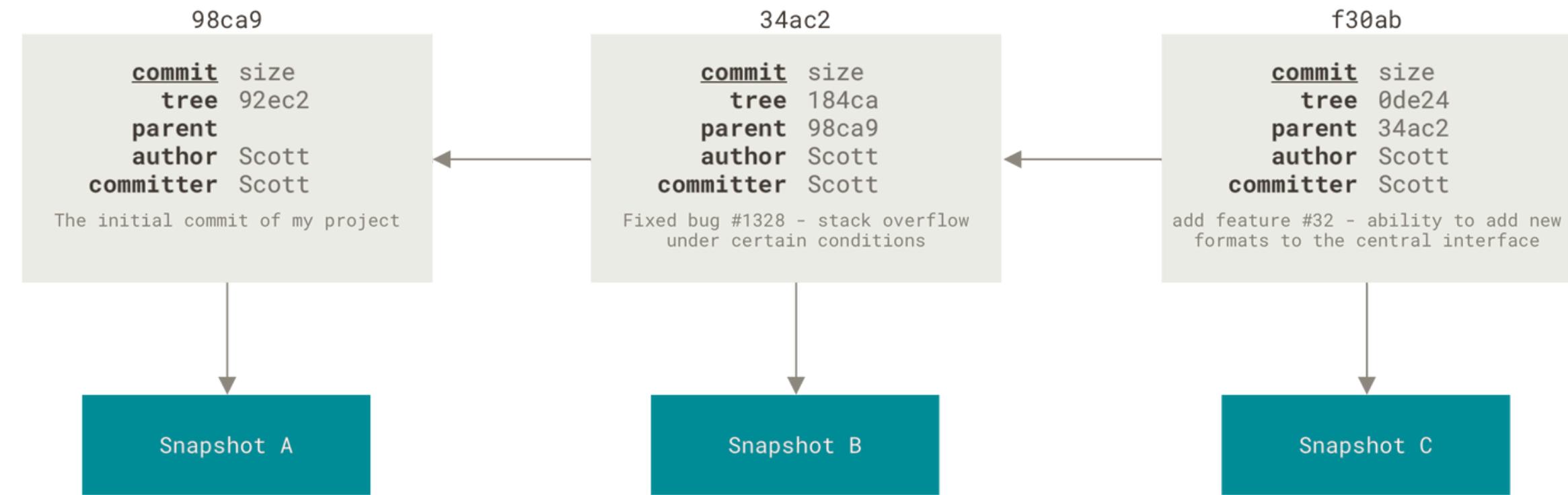
— Parallel Entwicklung einfach gemacht

# Die Idee

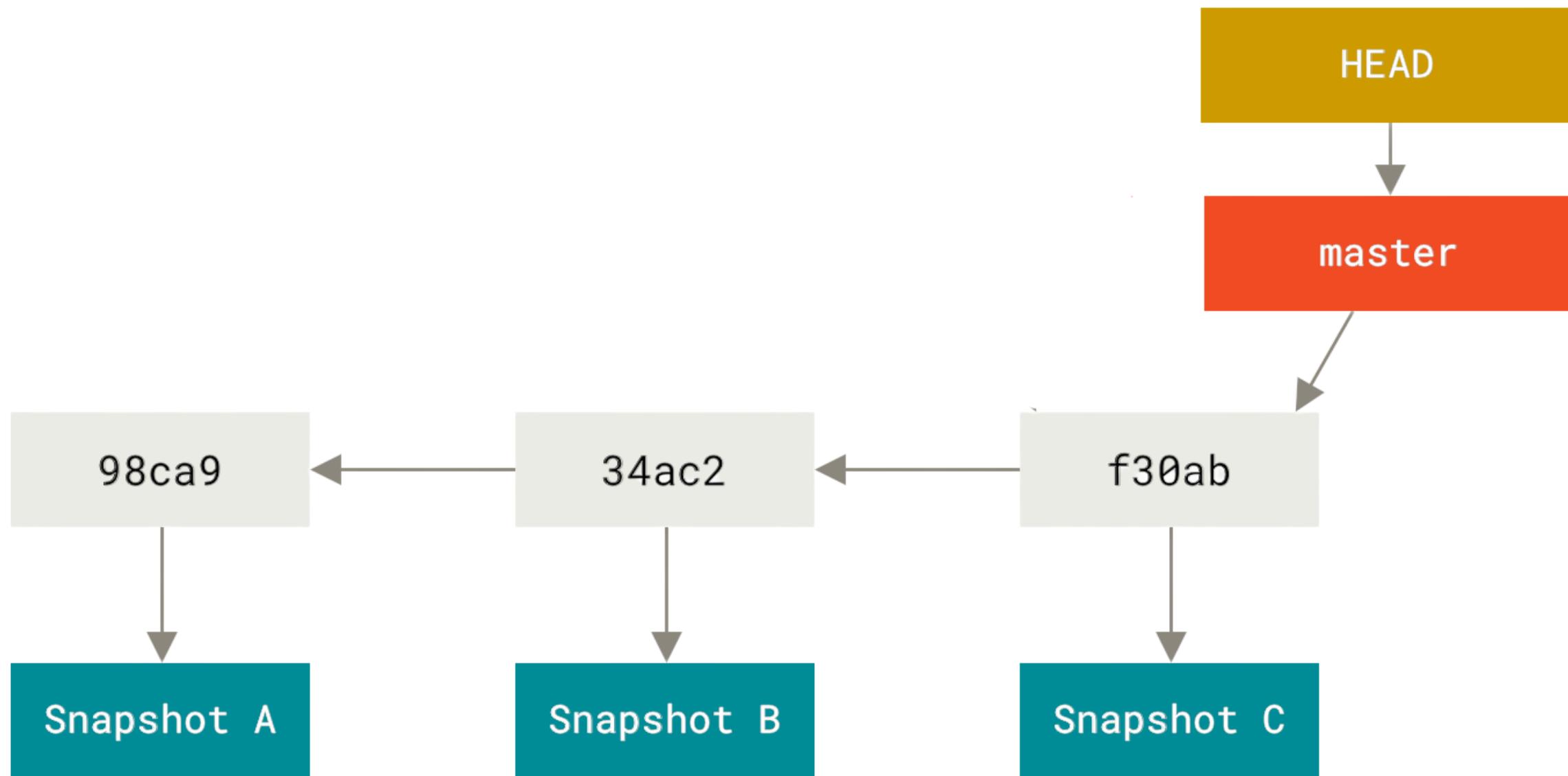


# Wie funktioniert das?





# HEAD + Master



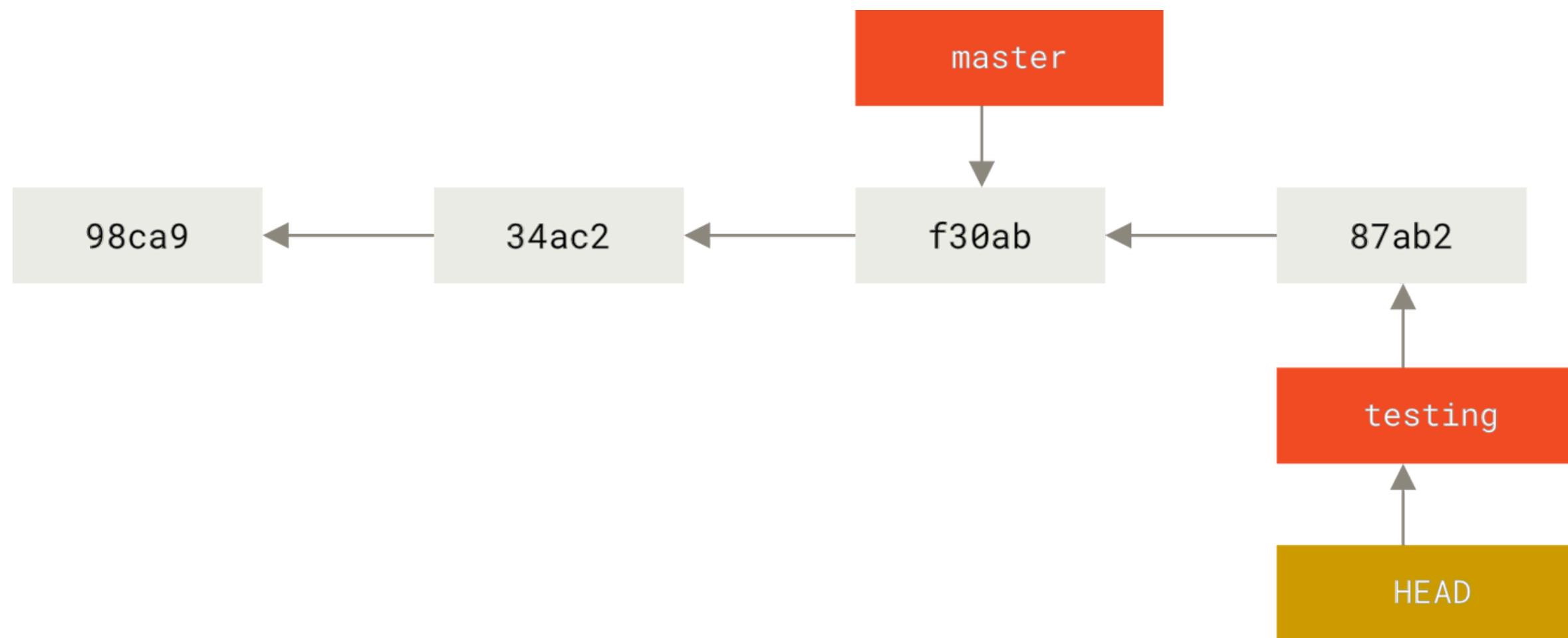
# git branch testing



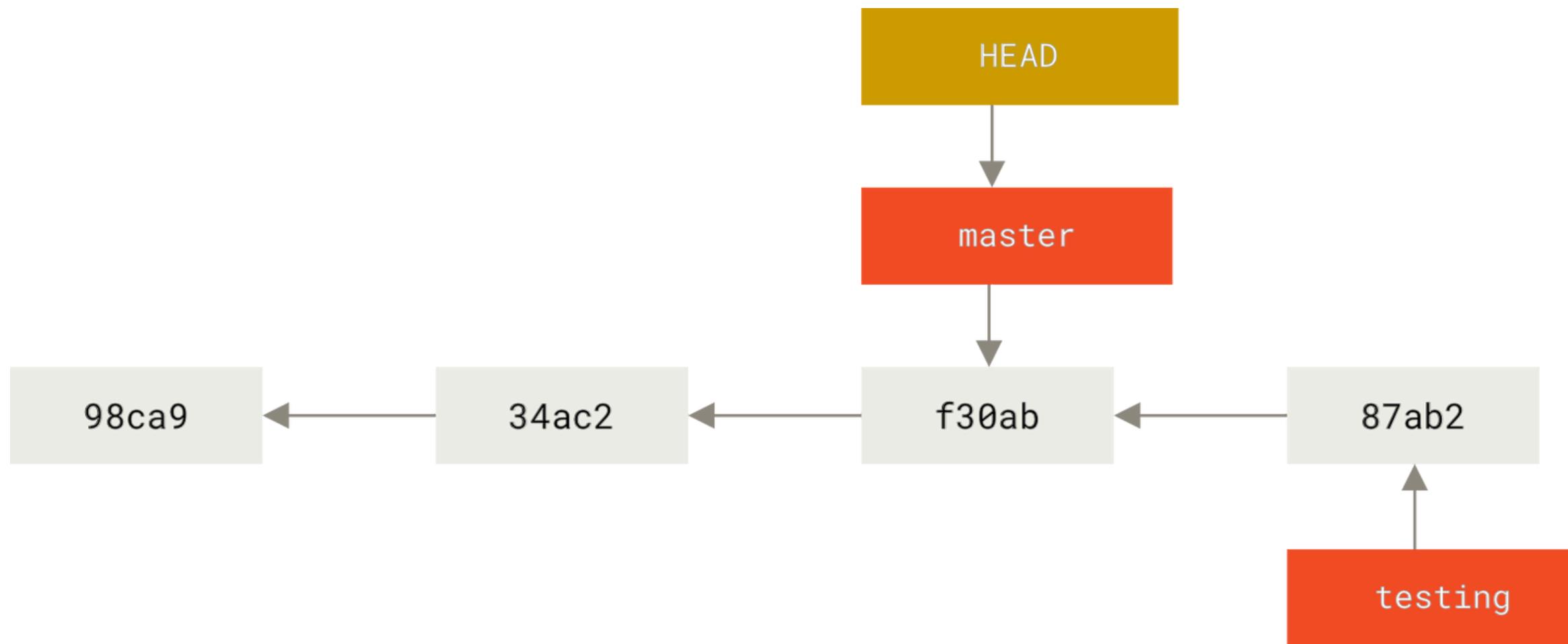
# git checkout testing



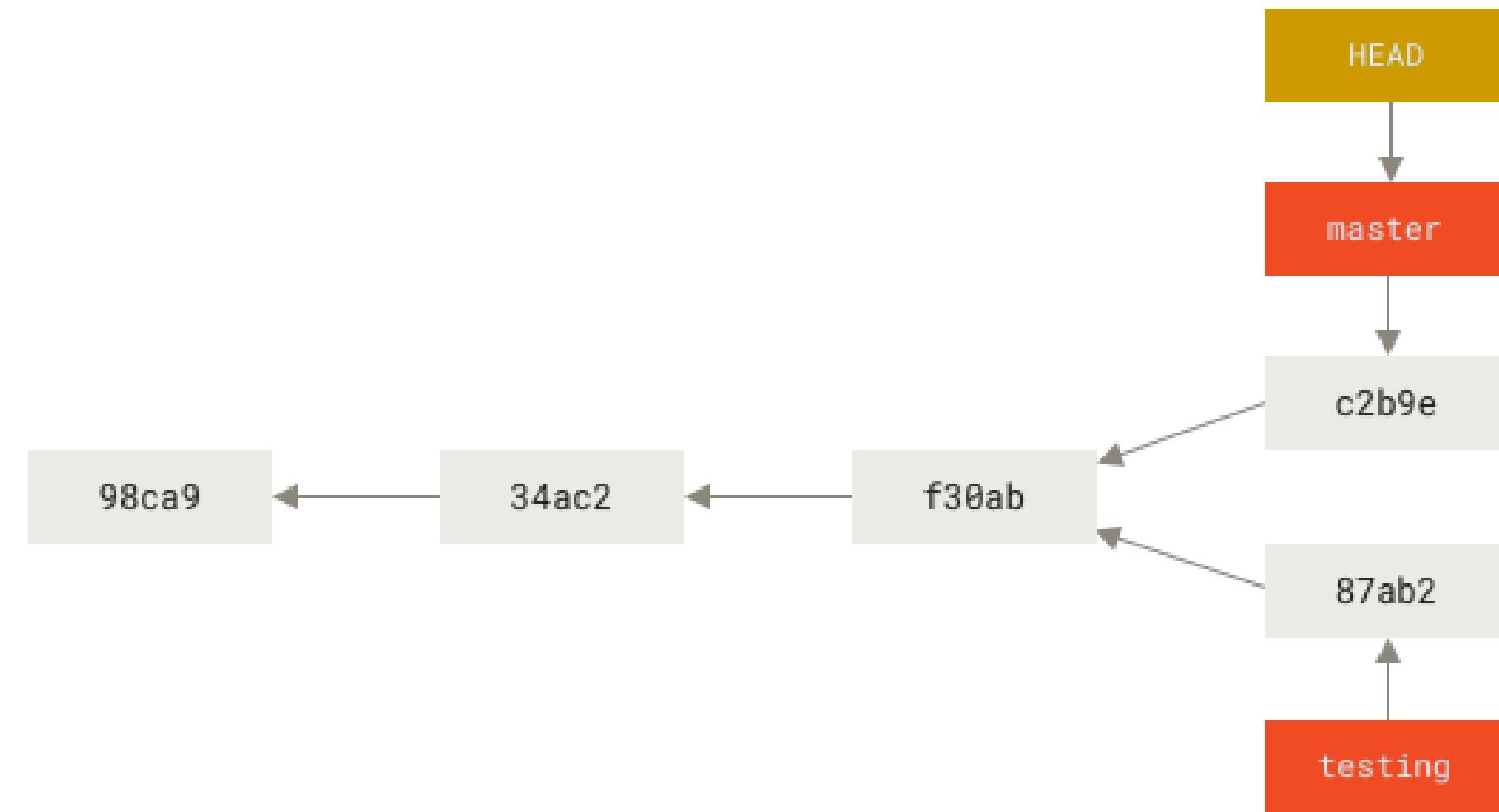
# Neuer Commit auf neuem Branch



# git checkout -b testing



# Veraestelung



# Commits & Commit- Nachrichten

— How to Nachrichten

# Commit Regeln

Atomic Commits



Commit Frequency

Vermeide unnötige Commits (update/fix)

Signed Commits (optional, aber oft bei Open Source) (GPG)

Commit Hooks

Commit Guidelines lesen

Rebase statt Merge bei Feature Branches



[Commit message templates | GitLab Docs](#)

Use commit message templates to ensure commits to your GitLab project contain all necessary...

[gitlab.com](https://gitlab.com)

# How To Nachrichten

COMMENT	DATE
CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
ENABLED CONFIG FILE PARSING	9 HOURS AGO
MISC BUGFIxes	5 HOURS AGO
CODE ADDITIONS/EDITS	4 HOURS AGO
MORE CODE	4 HOURS AGO
HERE HAVE CODE	4 HOURS AGO
AAAAAAA	3 HOURS AGO
ADKFJSLKDFJSOKLFJ	3 HOURS AGO
MY HANDS ARE TYPING WORDS	2 HOURS AGO
HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

**How to Write a Git Commit Message**  
Commit messages matter. Here's how to write them well.  
cbeams / May 27, 2023

Kurz & prägnant

Beschreibend

Imperativform verwenden

Bezug zu Issue/Feature

Kein „ich“ oder „wir“

Keine Mehrfachänderungen in einem Commit: Ein Commit = eine logische Änderung.

Fix editor crash when opening large files  
  
The editor now properly handles large files by increasing the memory limit.  
This fixes issue #5678 reported by multiple users.  
  
Signed-off-by: Max Mustermann <[max@example.com](mailto:max@example.com)>

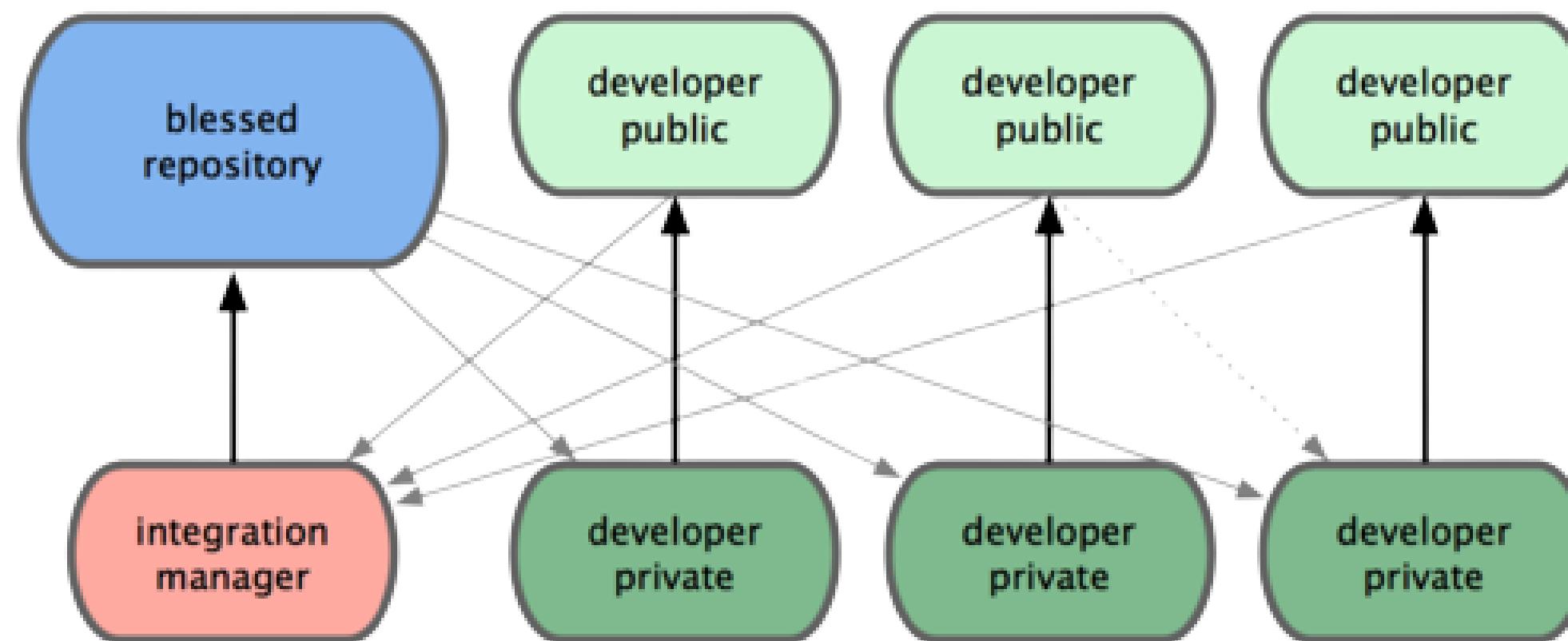


- Open Source Projekte wie VSCode haben viele Mitwirkende weltweit. Klare, konsistente Commit Messages erleichtern Reviews, Integration und Rückverfolgung.
- Viele Projekte nutzen automatisierte Tools, die Commit Messages auswerten (z.B. für Changelogs).
- Das Einhalten der Regeln zeigt Professionalität und erleichtert deine Annahme von Pull Requests.

# Synchronisation mit upstream

— Bleib up-to-date

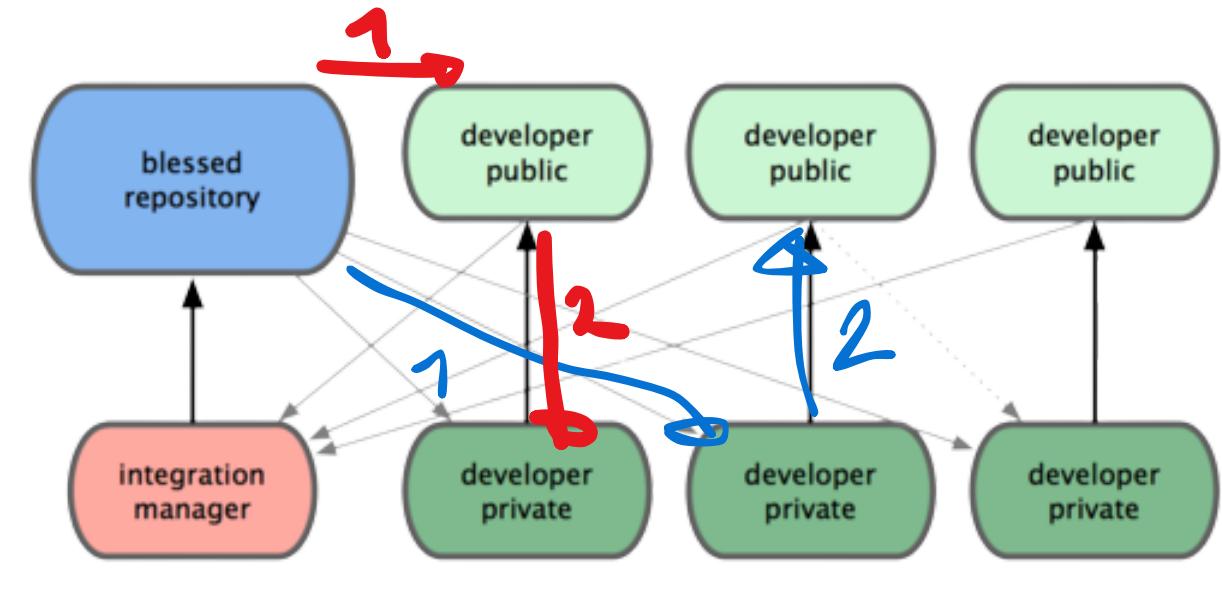
# Integration-Manager Workflow



# Synchronisiere upstream

✓1

```
# 1. Upstream-Remote hinzufügen (nur einmal)  
git remote add upstream https://github.com/original-owner/original-repo.git  
  
# 2. Änderungen vom Original-Repo holen  
git fetch upstream  
  
# 3. Auf den Hauptbranch wechseln  
git checkout main  
  
# 4. Upstream-Änderungen in den lokalen Branch integrieren (Merge oder Rebase)  
  
# Merge Variante:  
git merge upstream/main  
  
# Oder Rebase Variante (sauberere Historie):  
git rebase upstream/main  
  
# 5. Aktualisierten Hauptbranch zum eigenen Fork pushen  
git push origin main
```

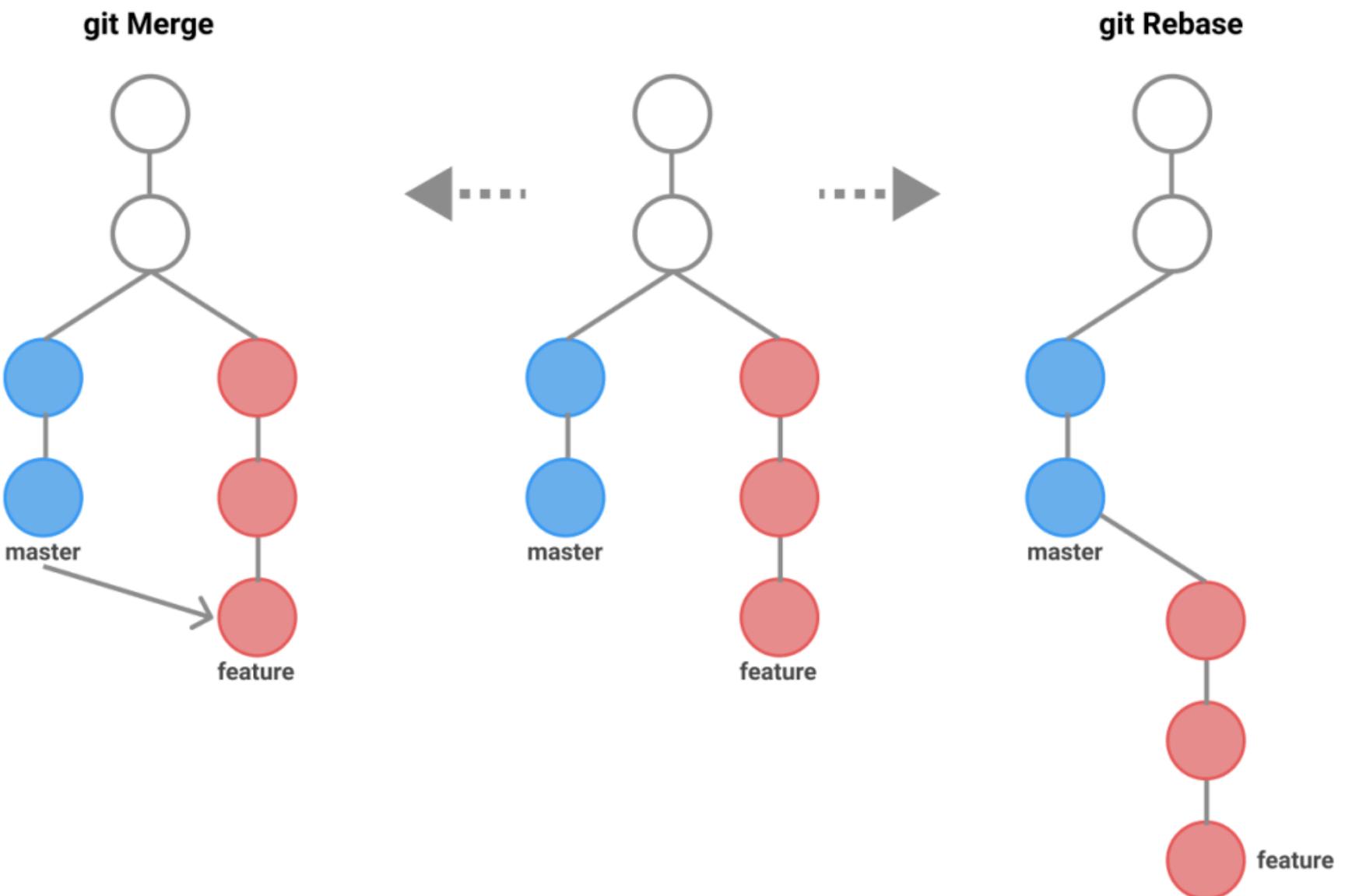


✓2

# Rebase vs. Merge

— Halte deine History sauber

# Rebase vs Merge



## 3-Way Merge:

Git verbindet zwei Branches, indem es den gemeinsamen Basis-Commit plus die Änderungen aus beiden Branches vergleicht und zusammenführt.



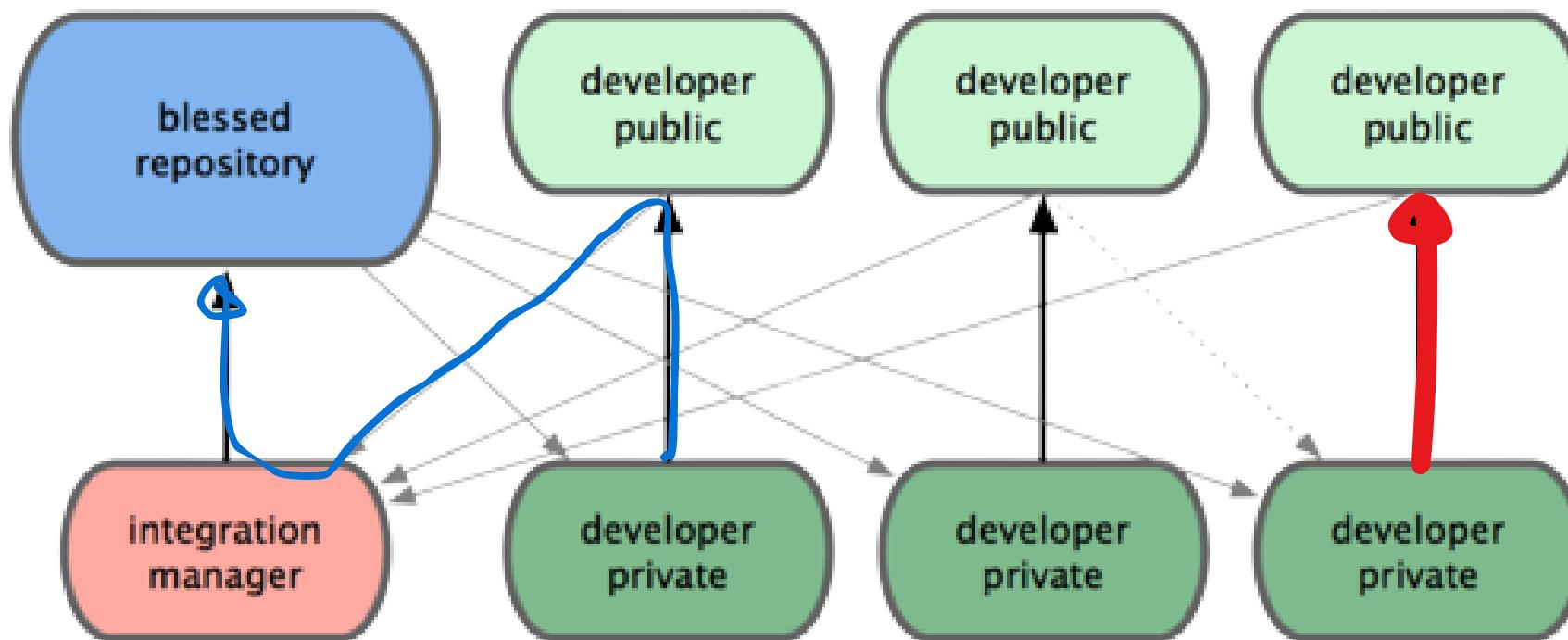
## Fast Forward:

Git verschiebt einfach den Zeiger des Ziel-Branches vorwärts, wenn keine neuen Commits seit der Abzweigung existieren, ohne einen Merge-Commit zu erstellen.

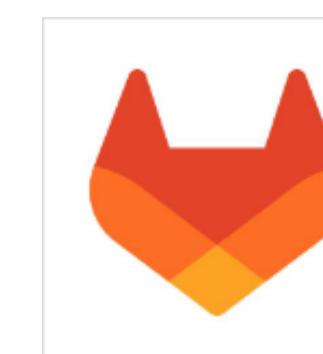
# Pull Request (PR) / Merge Request (MR)

— Anfrage zur Änderung der Codebasis

# Merge Request (GitLab)



 GitLab nennt die Anfrage fuer eine Änderung an der Codebasis 'Merge Request'. Im Github Jargon heisst es 'Pull Request'.



**Add columns to jobs table (!117424) · Merge requests · GitLab.org / GitLab ...**

What does this MR do and why? This is one step to refactor the jobs view for admins from haml to...

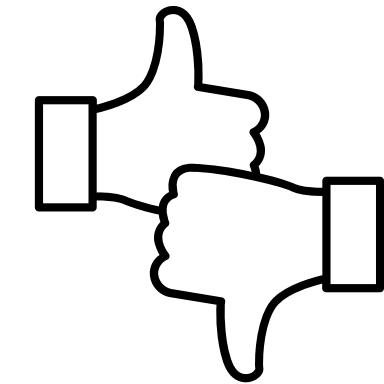
 GitLab

# Code-Review & Feedback integrieren

— Ein guter Review spart viel Zeit und Geld `in the long run`

# Was ist Review & Feedback?

Überprüfung von Code oder Arbeitsergebnissen



Gegenseitiges Feedback zur Verbesserung

Qualitätssicherung durch Team

Lern- und Verbesserungsprozess

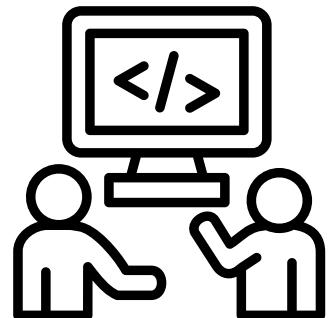
# Wann findet Review & Feedback statt?

Bei Pull/Merge-Requests in Versionskontrollsystmen

Nach Fertigstellung eines Arbeitspakets

Regelmässig in festgelegten Meetings

Während Pair Programming oder gemeinsamen Sessions



# Regeln für Feedback

Konstruktiv und respektvoll kommunizieren

Konkrete Verbesserungsvorschläge geben

Nicht nur Fehler suchen, auch Lob aussprechen

Fokus auf den Code, nicht die Person

Zeitnah und regelmässig durchführen

Coder sind extrem stolze Menschen wenn es um ihren Code geht.



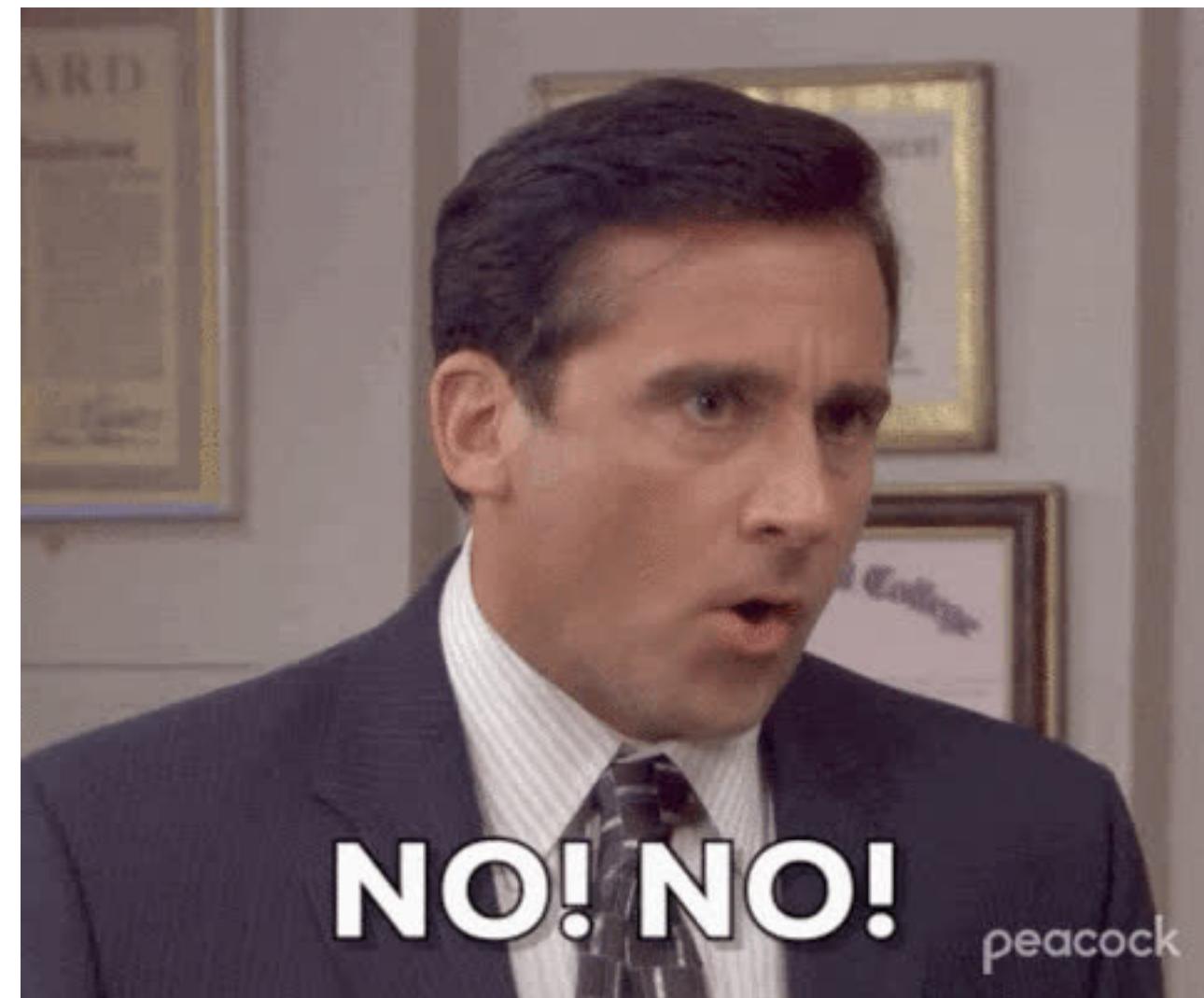
Fun-Fact: Schau dir deinen Code von vor einem halben Jahr an - du wirst erstaunt sein.

Offen für Feedback sein und daraus lernen

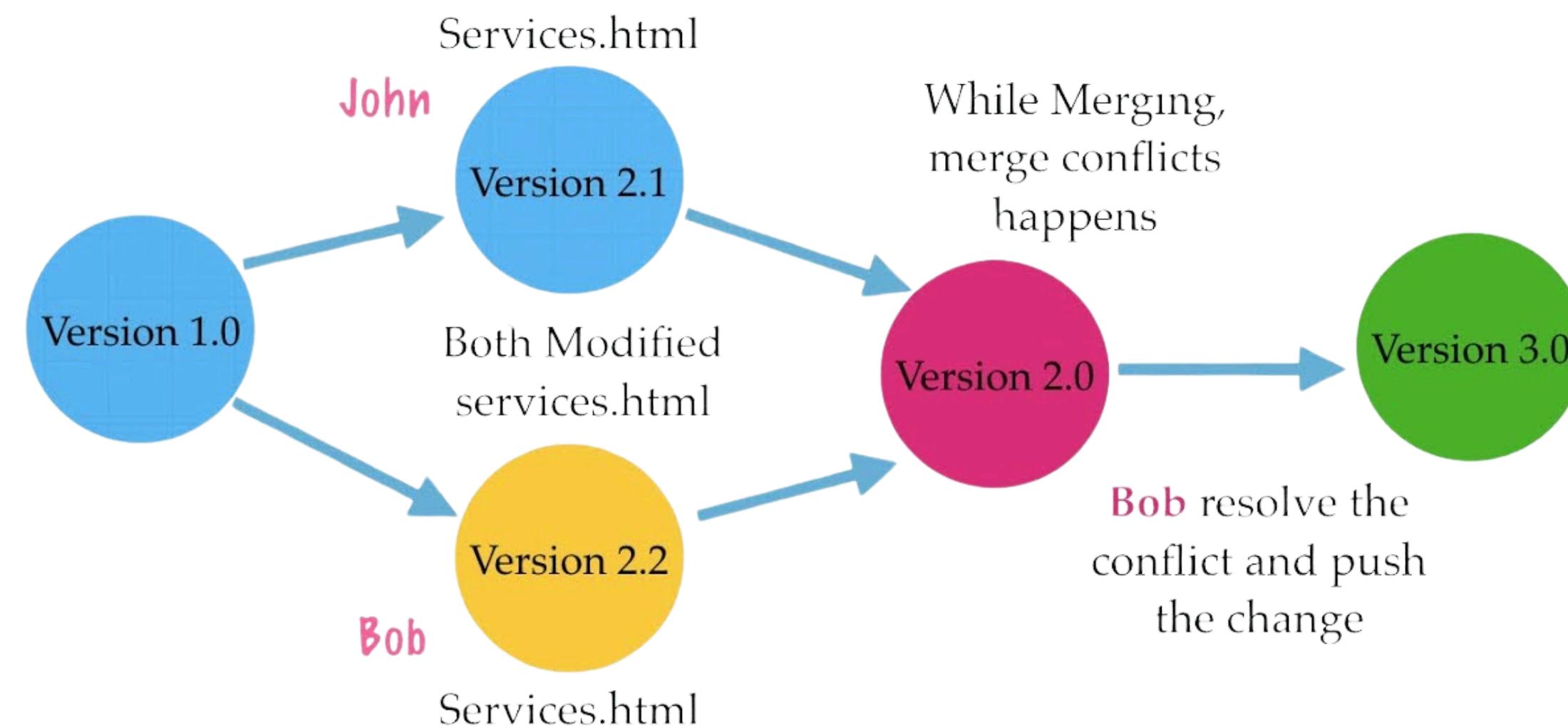
# Merge-Konflikt lösen

— Konflikte erkennen und lösen

# MERGE KONFLIKT!



# Was ist ein Merge Konflikt?



# Abbrechen

```
// Abbruch des merges/rebase. Der urspruengliche  
// Status wird wiederhergestellt.  
git merge --abort
```

```
git rebase --abort
```

# Fehler beim Merge Versuch

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

# Betroffene Dateien

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

# Konflikt Markers

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

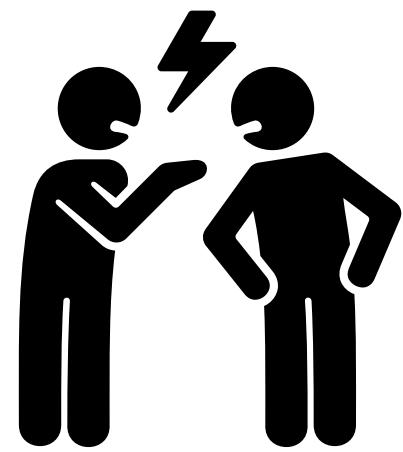
Frage den Autor des Codes wenn du nicht weiter weisst.



Pair Programming mit dem Autor kann Zeit sparen.

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

# Vermeide Konflikte



Regelmässig updateen (rebase/merge)

Kleine, gezielte Änderungen machen

Arbeitsbereiche klar aufteilen und kommunizieren

Früh zusammenführen und nicht zu lange in Branches arbeiten

Automatische Formatierung nutzen

# Diff

— Was sich ändert – schnell sichtbar gemacht.

# Was ändert sich?

```
git diff  
git diff --staged
```

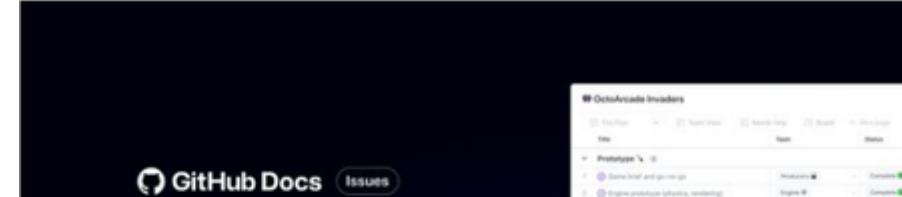
```
# Diff Ausgabe in eine Datei speichern (Linux/macOS)  
diff file1.txt file2.txt > changes.diff  
  
# Patch anwenden  
git apply changes.diff  
  
# Diff Ausgabe direkt in die Zwischenablage kopieren (Linux, mit xclip)  
diff file1.txt file2.txt | xclip -selection clipboard  
  
# Windows PowerShell: Diff Ausgabe in die Zwischenablage kopieren  
diff file1.txt file2.txt | clip
```

```
git diff  
diff --git a/Dockerfile b/Dockerfile  
index 24877b0..97227b5 100644  
--- a/Dockerfile  
+++ b/Dockerfile  
@@ -1,5 +1,5 @@  
 # The base image we'll be using for our container  
-FROM golang:1.18  
+FROM golang:1.19  
  
# The directory on the container we'll be copying all of our application code into  
WORKDIR /app  
diff --git a/README.md b/README.md  
index fb65379..6fc463a 100644  
--- a/README.md  
+++ b/README.md  
@@ -1,3 +1,8 @@  
+# Changelog  
+  
+3.21.24: Updated golang image  
+  
+  
# Getting started with minikube
```

# Issue-Verknüpfungen

— Pull Request und Issues verknuepfen

# Verknüpfung von Pull-Request und Issues



The screenshot shows a GitHub project interface for "OctoArcade Invaders". On the left, there's a sidebar with sections like "Project", "Issues", "Pull Requests", and "Branches". The main area displays a board with three columns: "Prototype", "Develop", and "Test". Under "Prototype", there's an issue titled "Define draft art and prototypes" with status "In Progress". Under "Develop", there are two issues: "Engine prototype (graphics, rendering)" and "Initial concept art", both also in "In Progress" status. Below the board, there are sections for "Beta" and "Launch" with their respective tasks.

**Einen Pull Request zu einem Issue verknüpfen - GitHub-Dokumentation**

GitHub Docs

Du kannst einen Pull Request oder Branch mit einem Issue verknüpfen, um zu zeigen, dass ein Fix in Arbeit ist und das Issue automatisch zu schließen, wenn der Pull Request oder Branch gemitgt wird.

# Blame & History verstehen

— Spurensuche im Code: Änderungen im Blick



Skip aus Zeitgründen

# Hands-On



# Jetzt kenne ich die Basics



Danke fürs zuhören