Versionsverwaltung

VCS

V ersion C ontrol S ystem

Hier GIT

WARUM VCS BENUTZEN?

BEISPIEL

- `Bachelorarbeit-v0.1.docx`
- `Bachelorarbeit-v0.9.docx`
- `Bachelorarbeit-vFinal.docx`
- `Bachelorarbeit-vFinal-2.docx`
- `Bachelorarbeit-vFinal-FINAL.docx`

GUTE GRÜNDE

1. Zwischenstände Protokollieren
   - Wer - Wann - Was
2. *UnDo* von Änderungen
3. Gruppenarbeit vereinfacht (Synchronisierung)
   - inkl. Berechtigungen
4. gleichzeitiges Arbeit an mehreren Entwicklungszweigen
   - durch schnellen Wechsel zwischen diesen Zweigen

BEGRIFFE

Workcopy
: Dateien, die ich momentan *sehen* und bearbeiten kann (*Arbeitskopie*)

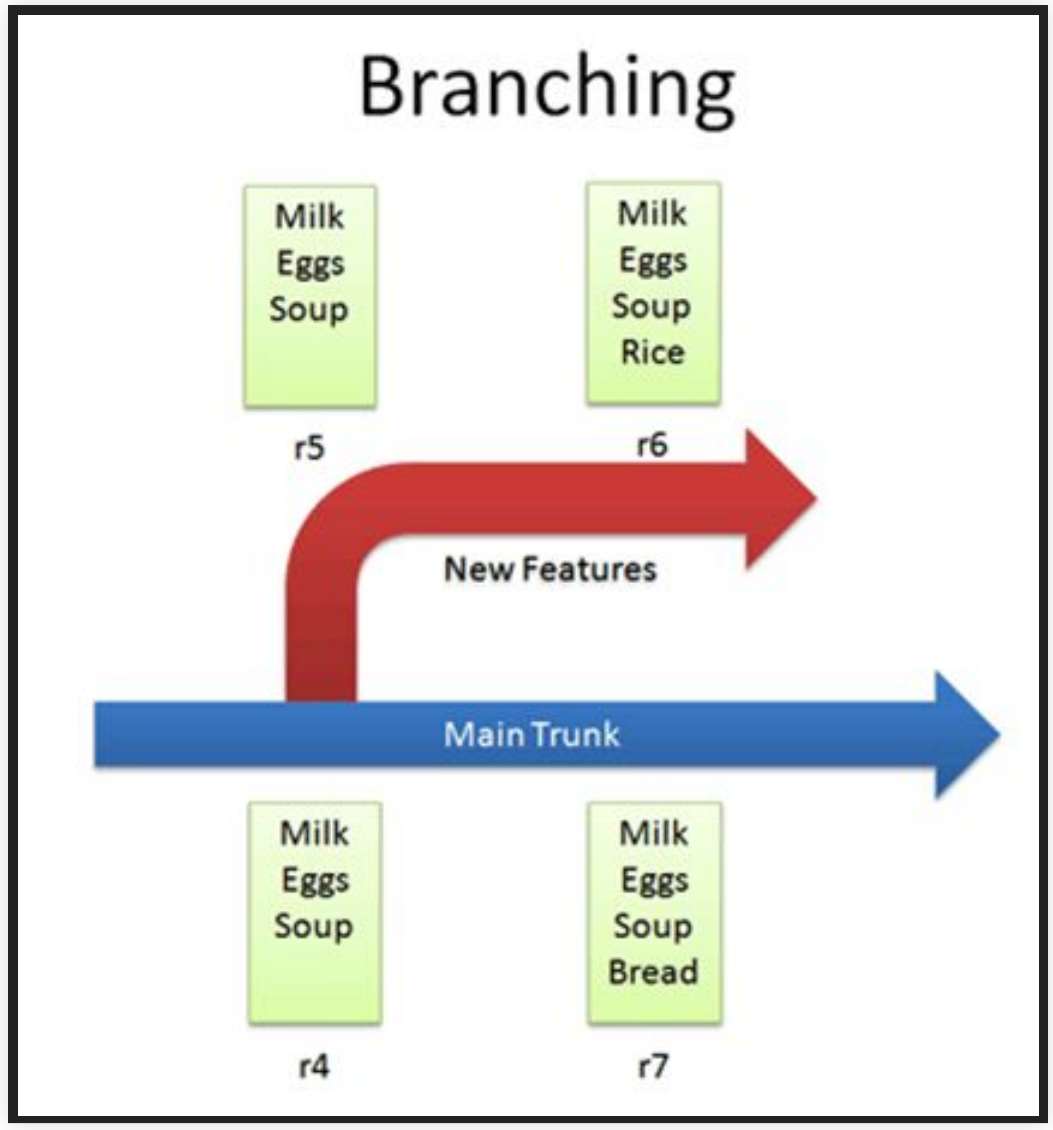
Repository
: Behälter für alle Dateien und deren Versionen, die das VCS kennt

checkout
: Übertragen einer Version aus dem Repository in die Workcopy

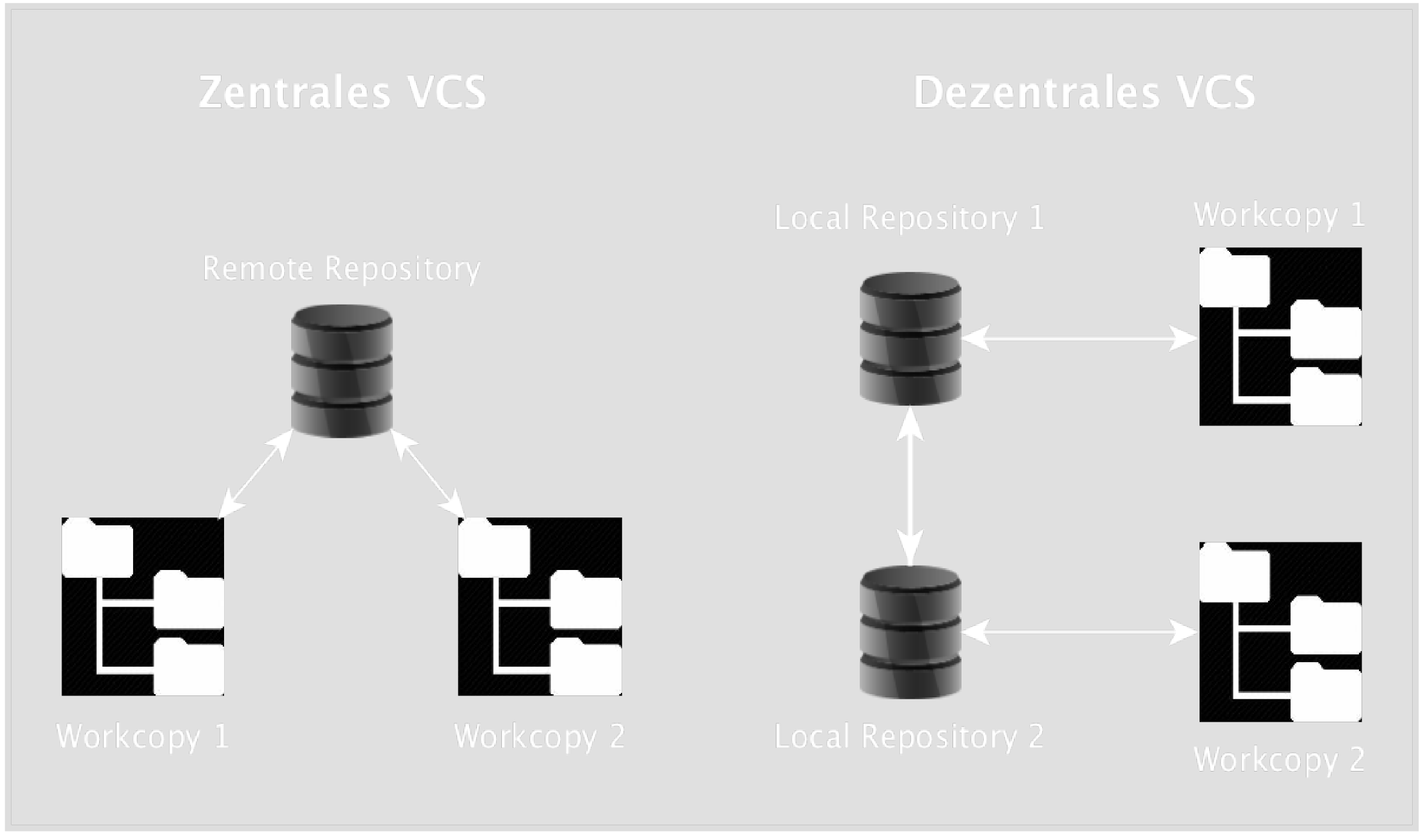
commit
: Übertragen einer Version von der Workcopy in das Repository

Branch

Parallel entwickelte Version



ZENTRAL VS. VERTEILT



GIT

- Verteiltes VCS
- vom Linux Erfinder Linus Torwalds
- seit 2005
- *a stupid content tracker*
- Buch: Pro Git - online

ZENTRAL DEZENTRALISIERT

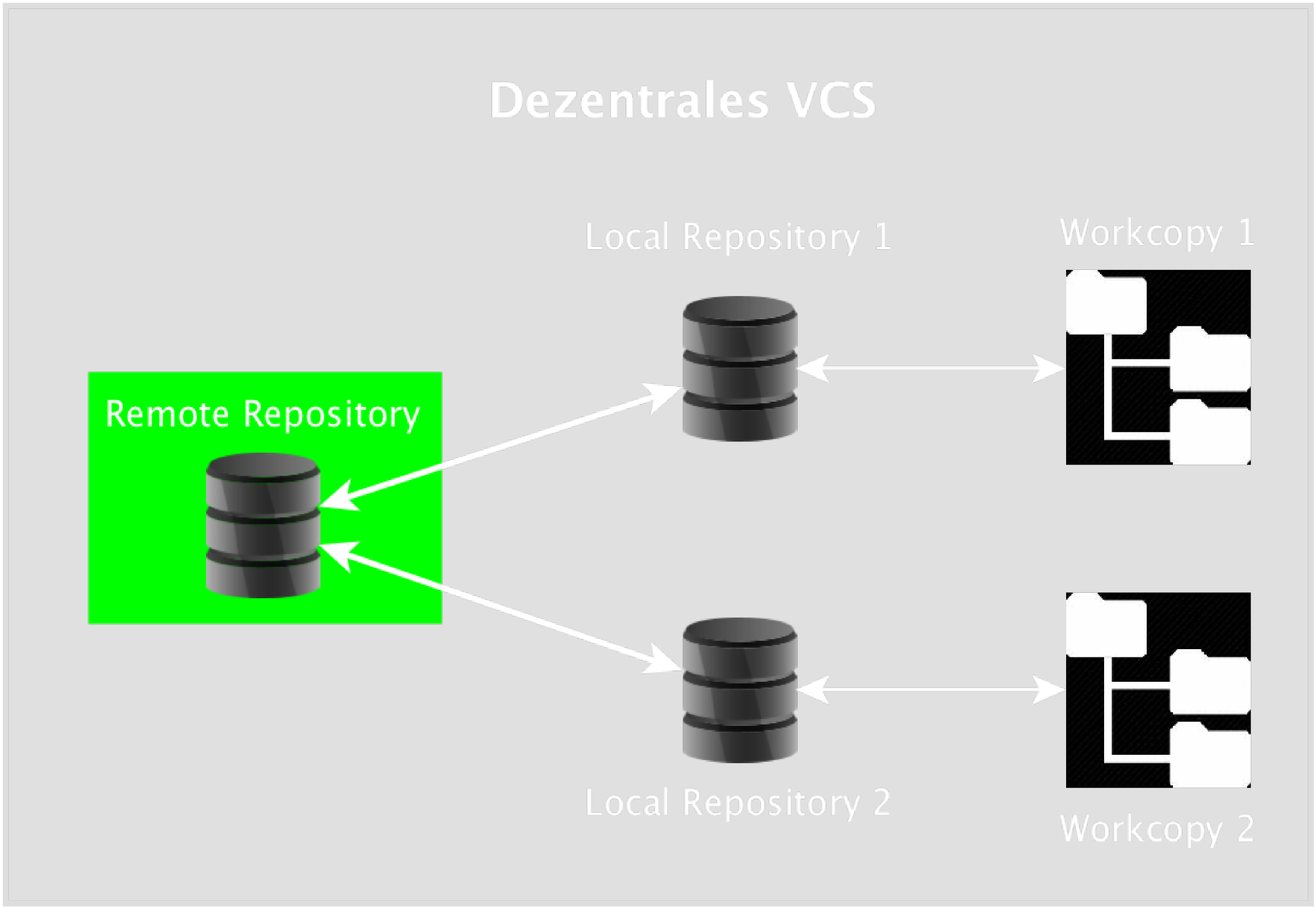
Zentral → Server
Dezentral → kein Server?

ZENTRAL DEZENTRALISIERT

Zusätzlicher zentraler Server hat sich bewährt

blessed Repository

- Zugriffskontrolle
- Gemeinsamer Ursprung für neue Kopien
- Backup
- Basis für Zusatzfunktionen
  - Repo-Browser im Web
  - Konzept: Pull-Requests
  - Web-Editor für Inhalte
  - README.md Rendering


Dezentrales VCS
Local Repository 1
Workcopy 1
Remote Repository
Local Repository 2
Workcopy 2

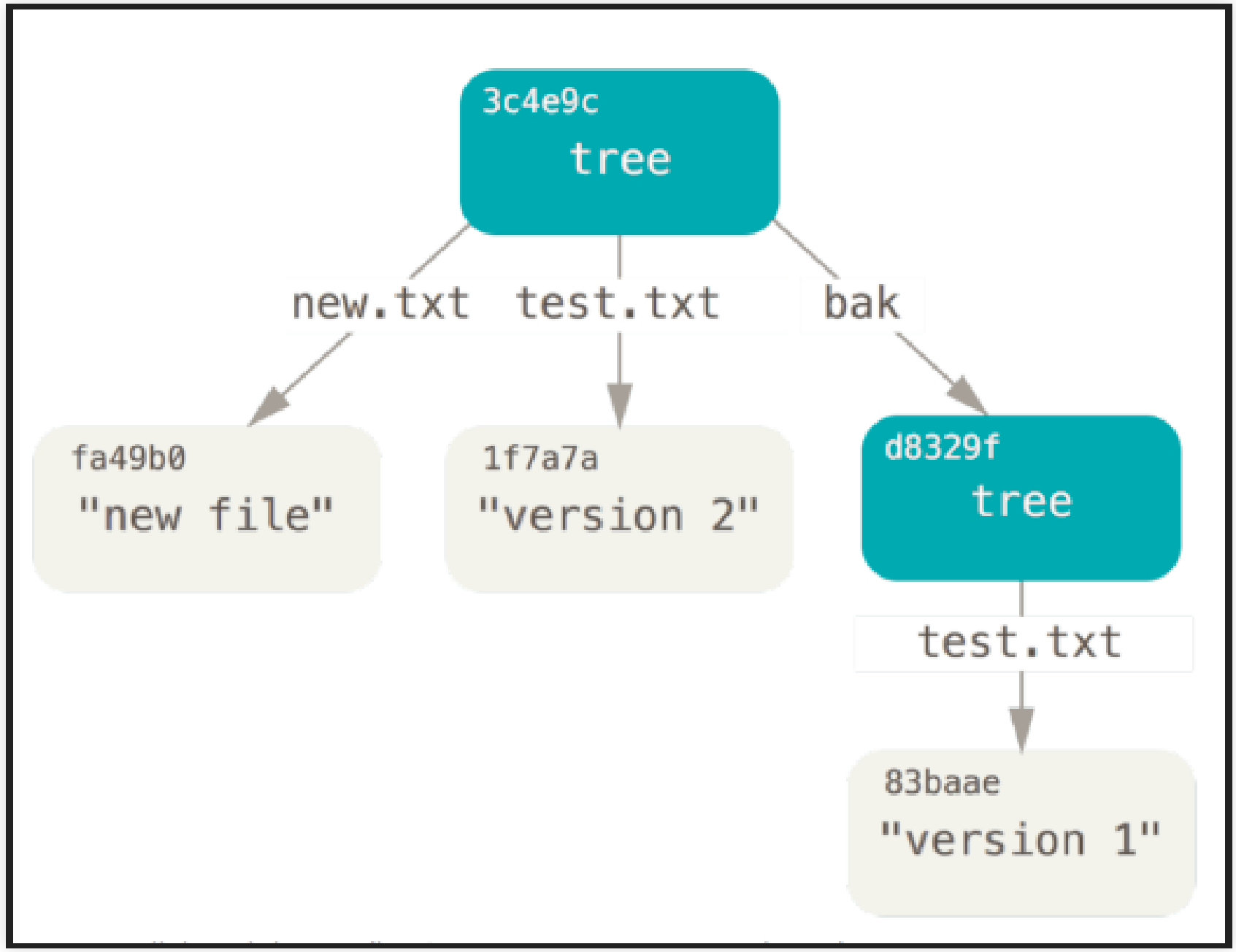
A STUPID CONTENT TRACKER

- Repository
  - == effizienter Objektspeicher
  - für alle Inhalte werden Hash-Werte als Schlüssel berechnet (SHA, 160 bit)
    - Trennung von Dateiinhalt und Dateiname
    - Inhalte werden nur einmal gespeichert (keine Duplikate)
    - Git versioniert immer das ganze Projekt
- HASH Beispiel: a544751ae3de9965c35b88958b0d219e29f7295d

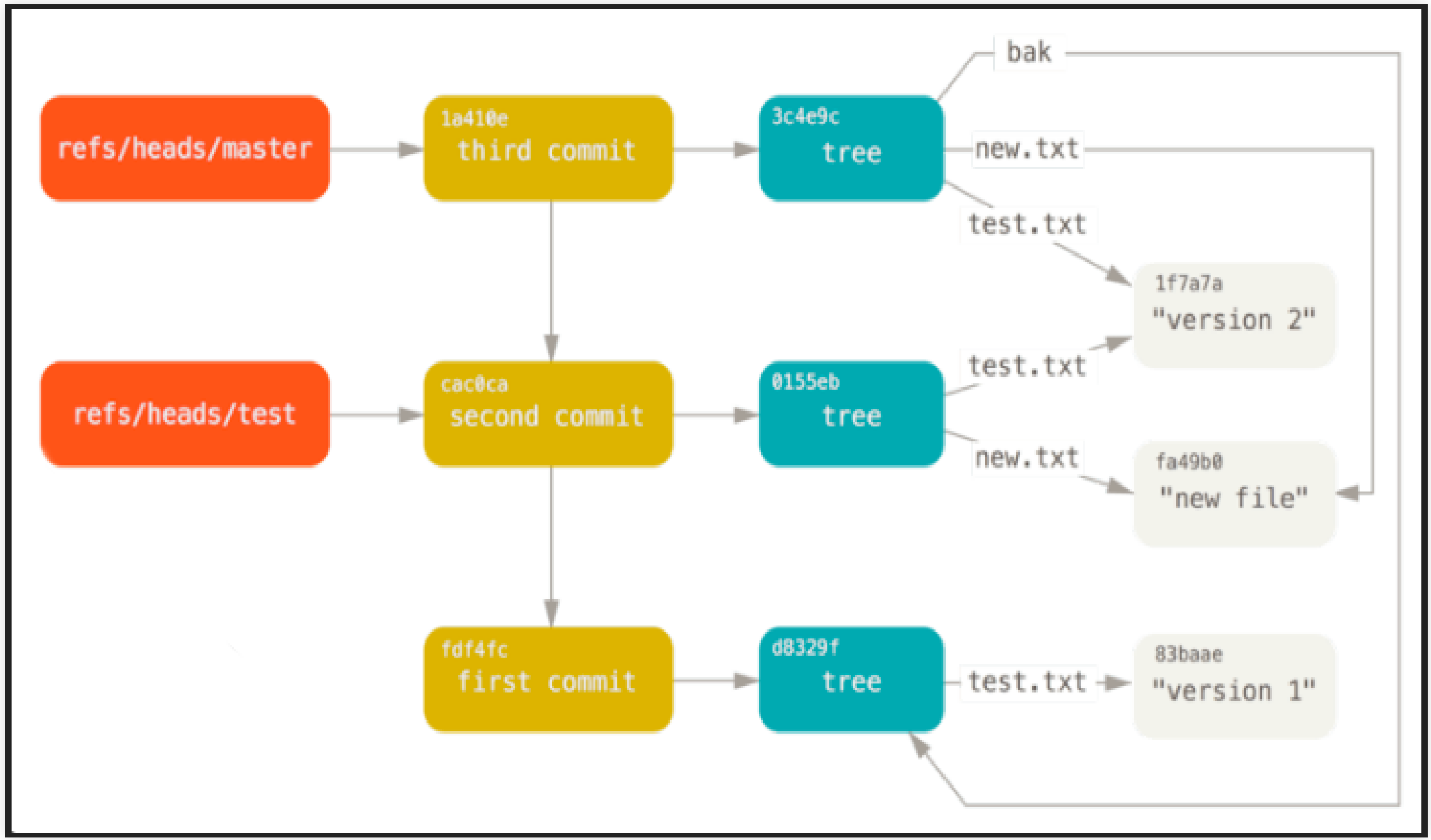
A STUPID CONTENT TRACKER

- Interne Datenstuktur von GIT
  - Blob (sha, packed binary)
  - Tree (sha, Liste von Dateien oder Sub-Trees: sha, Zugriffsrechte, Name)
  - Commit (sha, Liste von Parents: sha, Tree, Author, Datum, Message)
  - Tag (sha, commit-sha, Author, Message)
  - Reference (name, commit-sha)
    - z.B. Branch, HEAD, Tag

A STUPID CONTENT TRACKER



A STUPID CONTENT TRACKER



A STUPID CONTENT TRACKER

- GIT Datenstruktur ist sehr einfach zu verstehen.
- Alle GIT-Kommandos helfen nur, diese Daten zu manipulieren.
- Um mit GIT zu arbeiten ist das Verständnis dieser Struktur PFLICHT.

GIT KOMMANDOS

Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it

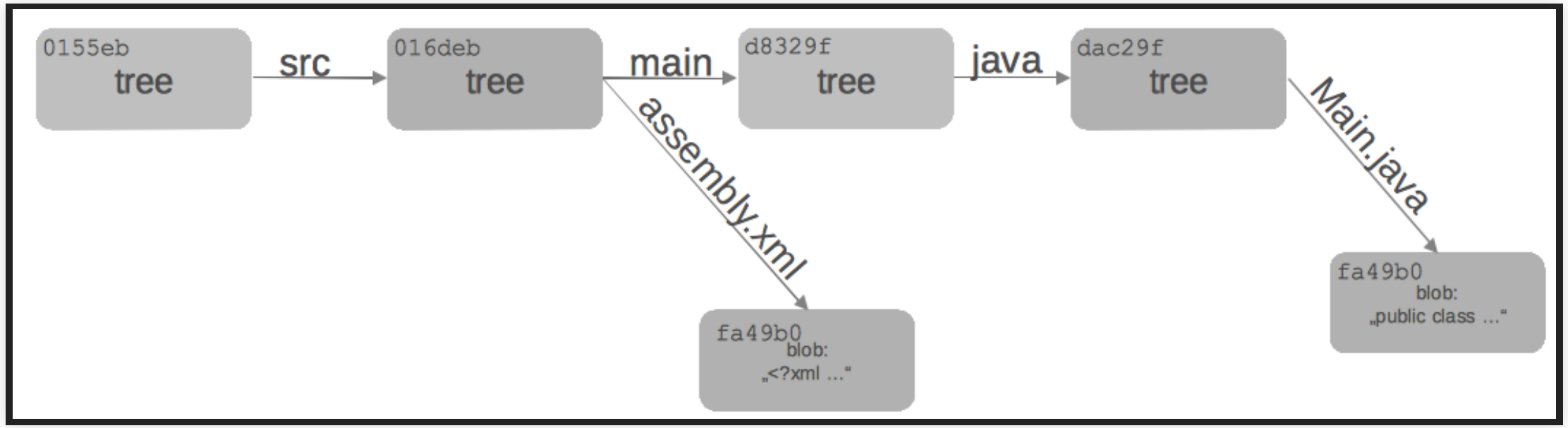
GIT KOMMANDOS

1. Plumbing
   - Low-level Aufgaben
   - Stabile API (Parameter, Output)
   - Designed für UNIX-artige Verkettung (pipes) und Skripte
   - z.B. `git merge-base`, `git ls-tree`, `git cat-file`
2. Porcelain
   - High-Level Aufgaben
   - benutzerfreundliche API (Parameter, Output)
   - z.B. `git merge`, `git status`

Abbildung eines Dateisystems

- tree-Objekt
  - eigener SHA-Schlüssel
  - Liste von Kind-Einträgen ([sub]-tree oder blob) mit jeweils:
    - Datei-Modus (UNIX Benutzerrechte, Executable-Flag)
    - Typ (blob | tree)
    - SHA-Schlüssel
    - Name
- blob-Objekt
  - eigener SHA-Schlüssel
  - Inhalt

ABBILDUNG EINES DATEISYSTEMS



VCS FEATURES - COMMIT

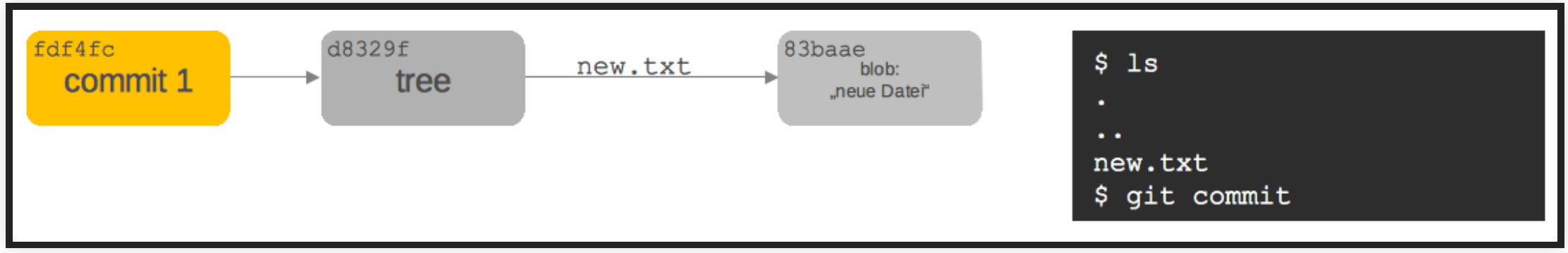
- commit-Objekt
  - eigener SHA-Schlüssel
  - SHA-Schlüssel der Vorgänger-Commits
  - SHA-Schlüssel des root-tree, der den Zustand des Projektes beschreibt
  - Commit-Nachricht
  - Author, Zeitstempel
- SHA kann oft abgekürzt werden

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002d
```

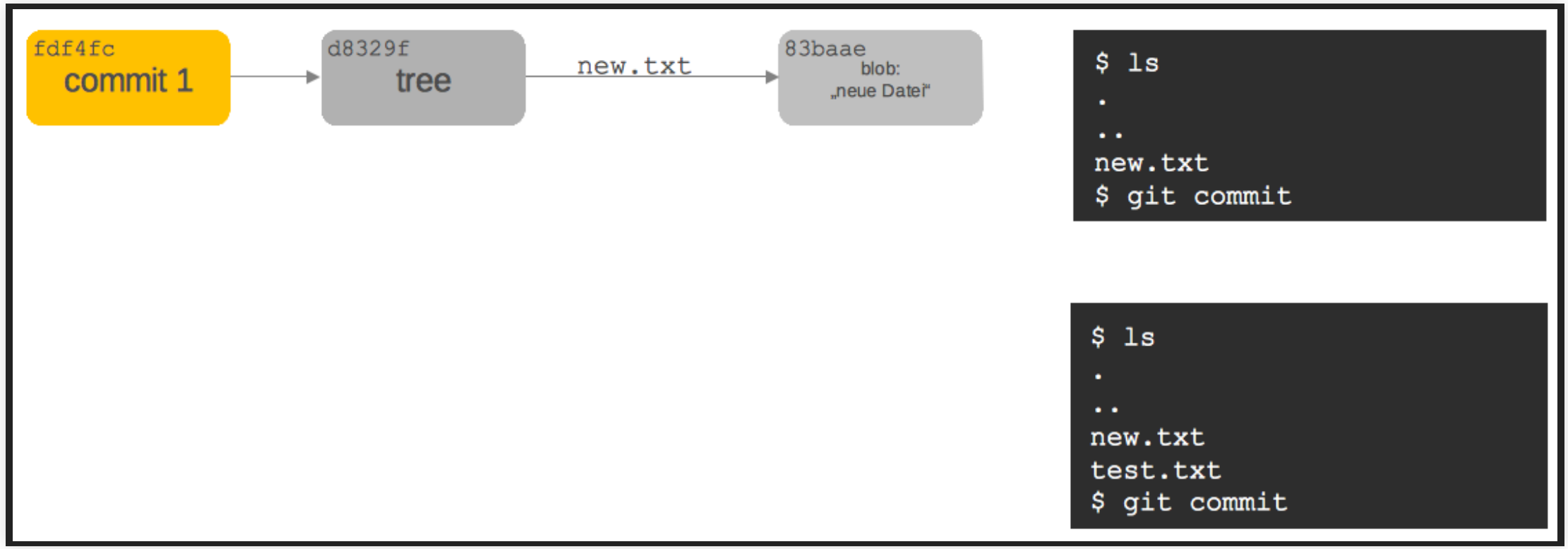
VCS FEATURES - COMMIT

```
$ ls
.
..
new.txt
$ git commit
```

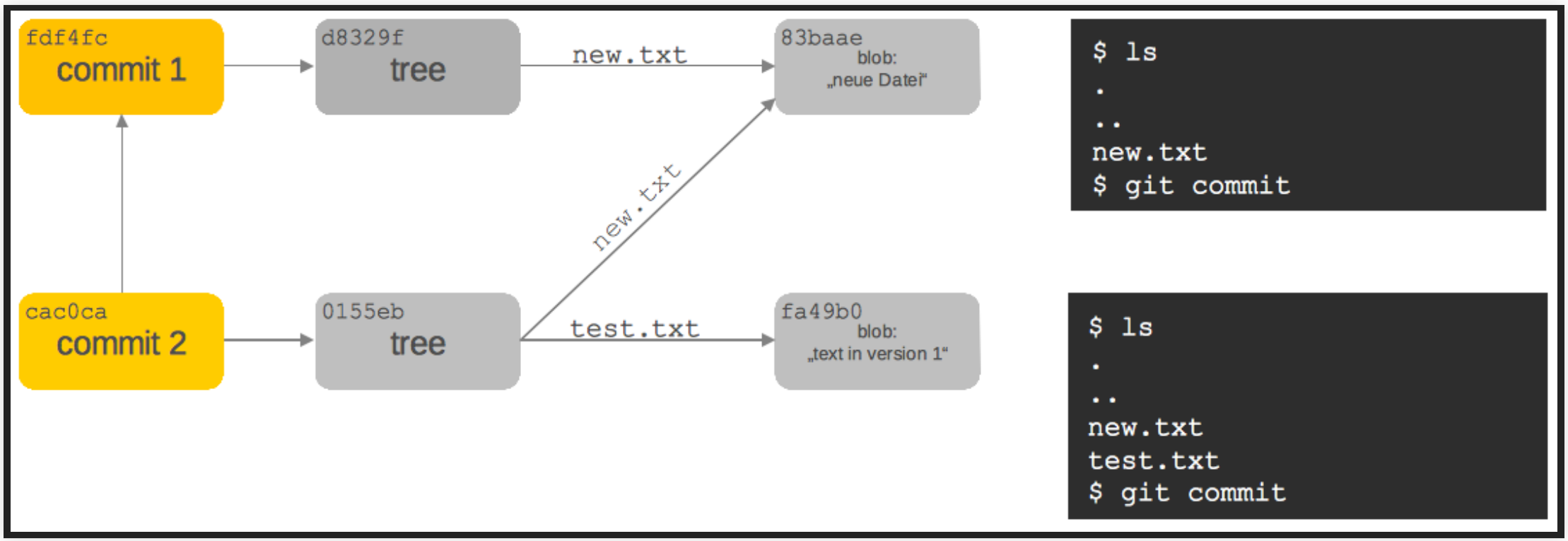
VCS FEATURES - COMMIT



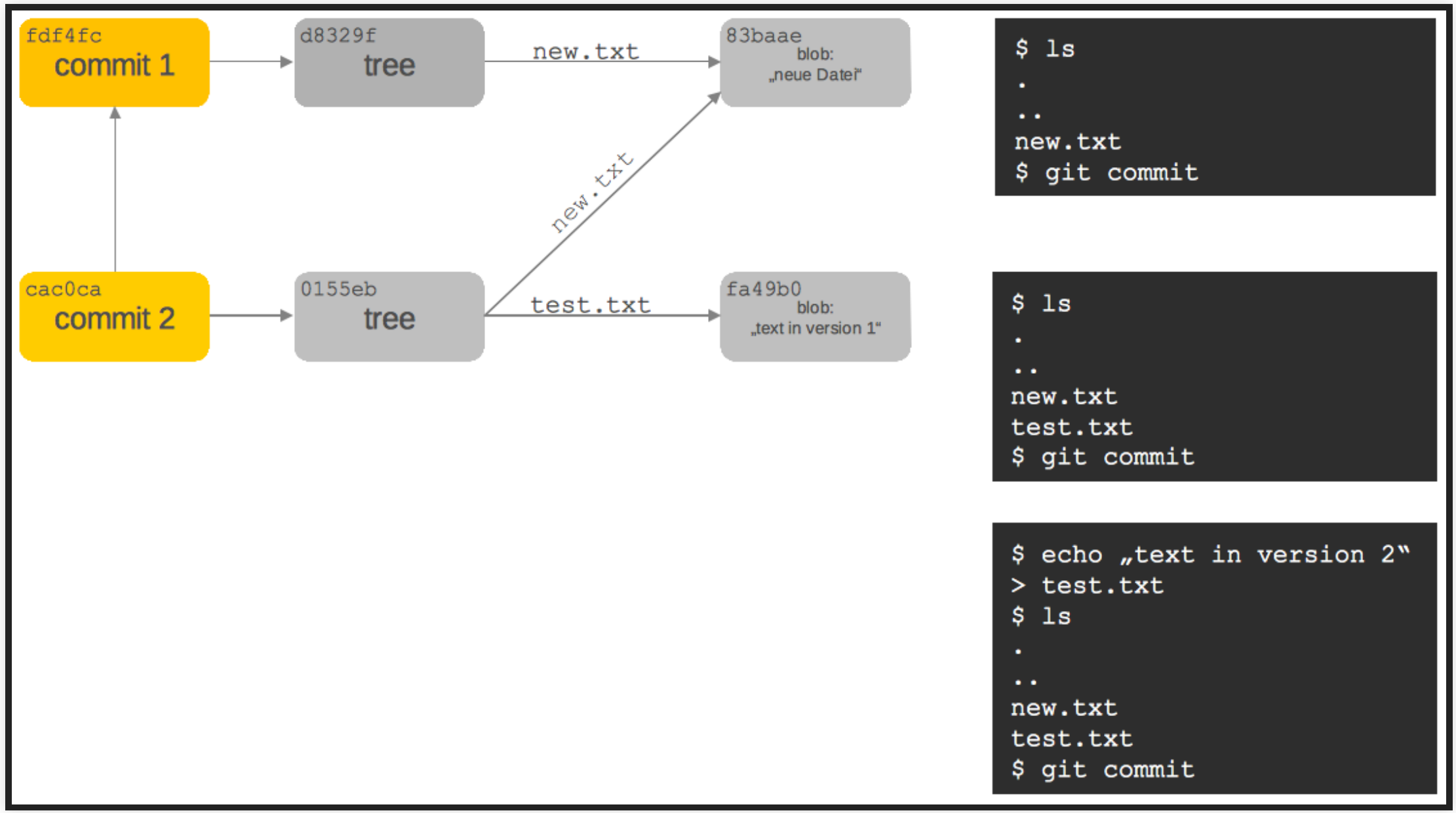
VCS FEATURES - COMMIT



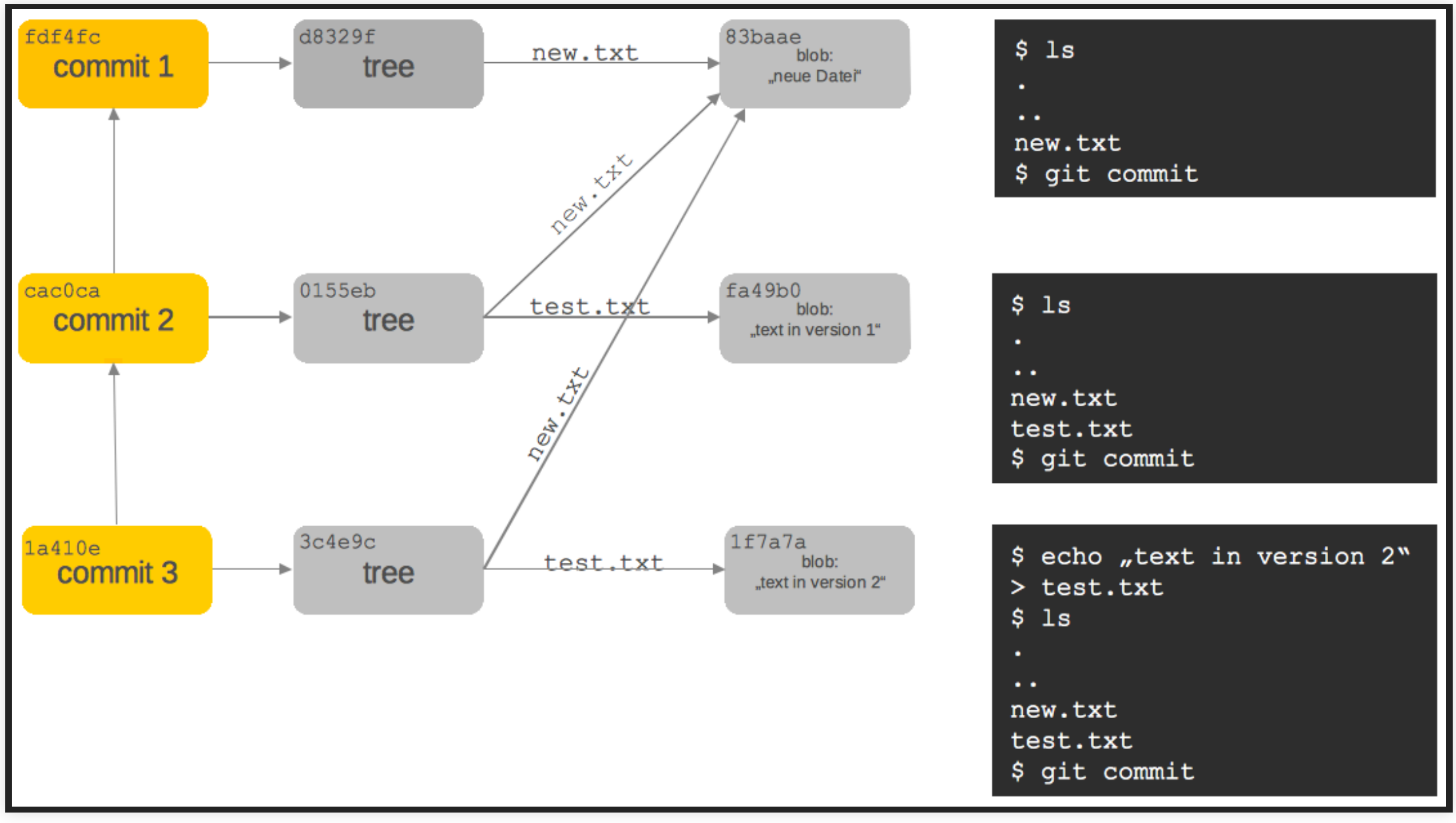
VCS FEATURES - COMMIT



VCS FEATURES - COMMIT



VCS FEATURES - COMMIT



VCS FEATURES - COMMIT

Doppelbedeutung commit

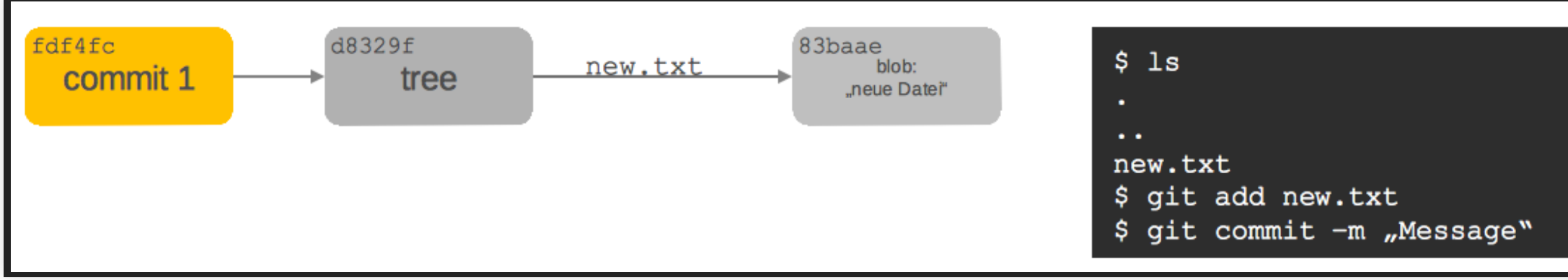
1. das Objekt in der GIT Daten-Struktur
   - stellt den Zustand des gesamten Projektes (== Datei- und Ordner-Struktur) zu einem bestimmten Zeitpunkt dar
2. der Befehl, einen Commit zu erstellen
   - auch als Verb: "Ich committe jetzt"

VCS FEATURES - STAGE | INDEX

VCS FEATURES - STAGE | INDEX

```
$ ls
.
..
new.txt
$ git add new.txt
$ git commit -m „Message“
```

VCS FEATURES - STAGE | INDEX

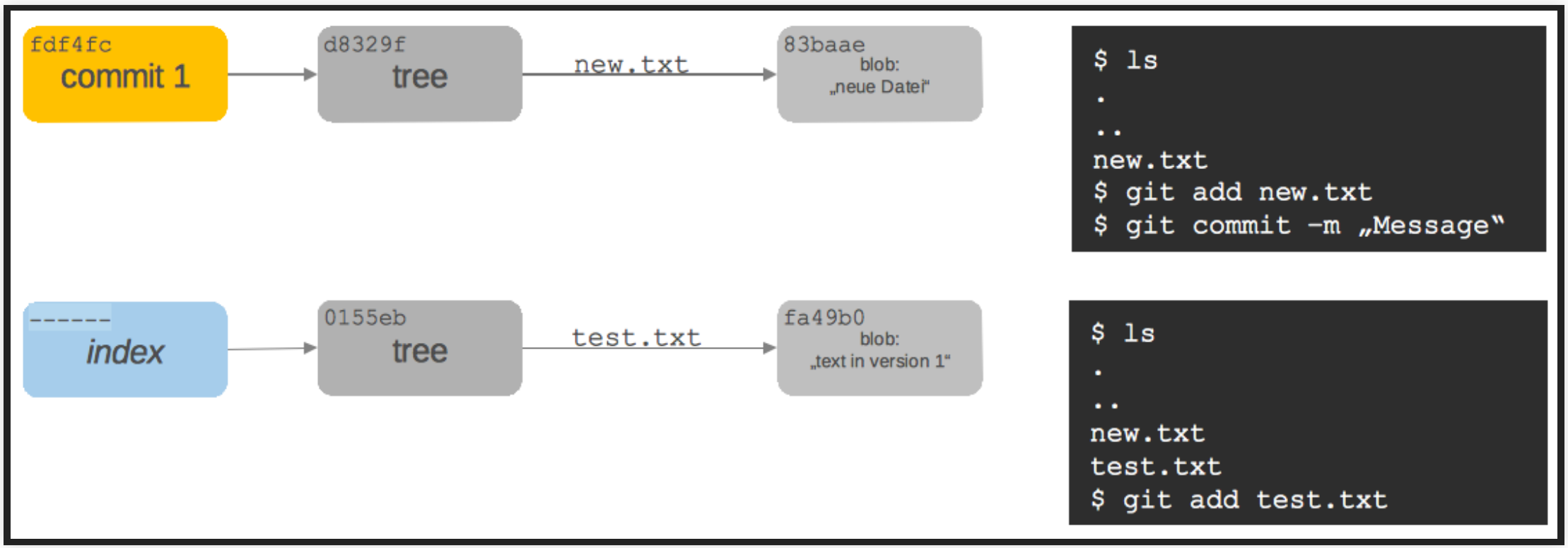


VCS FEATURES - STAGE | INDEX

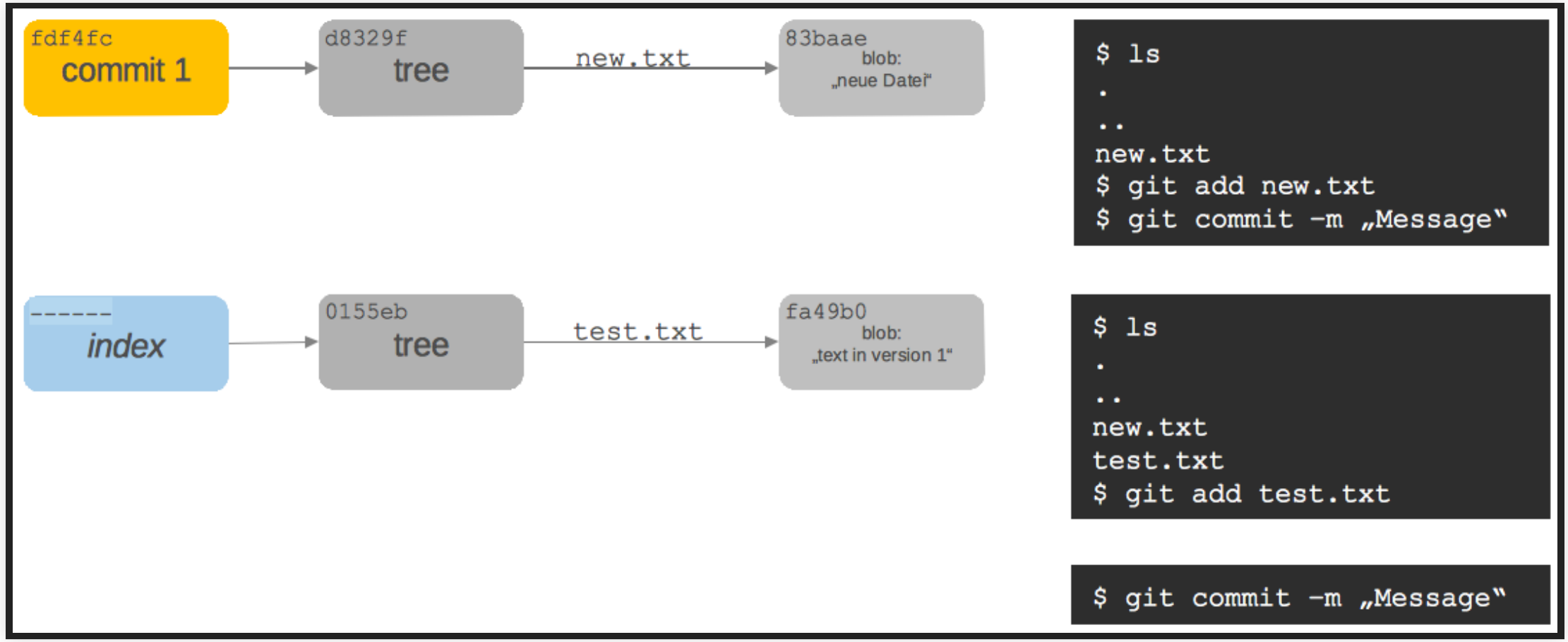

```
$ ls
.
..
new.txt
$ git add new.txt
$ git commit -m „Message"
```

```
$ ls
.
..
new.txt
test.txt
$ git add test.txt
```

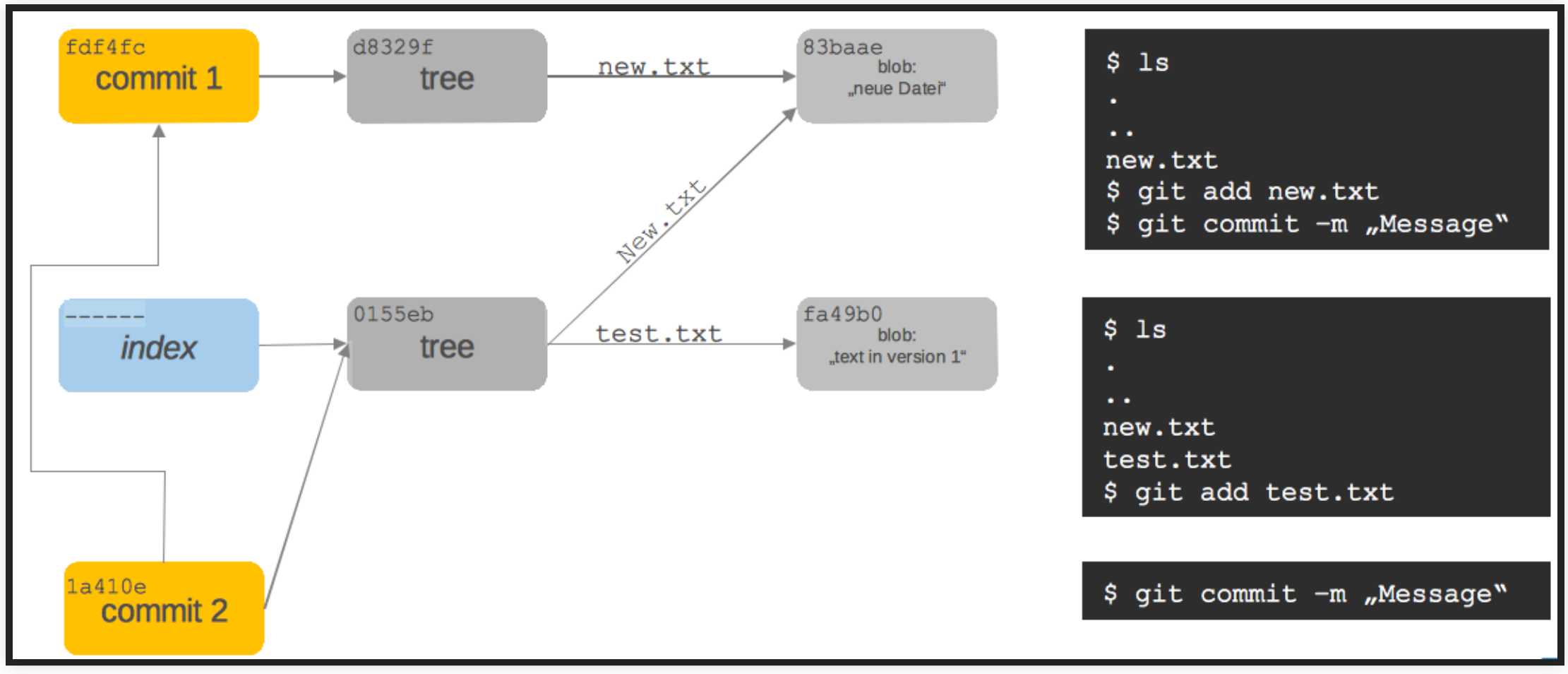
VCS FEATURES - STAGE | INDEX



VCS FEATURES - STAGE | INDEX



VCS FEATURES - STAGE | INDEX



BEFEHLE - ADD/RM

```
## Fügt alle neuen/geänderten vom aktuellen Ordner (rekusiv)
## zum Index hinzu
$ git add .
## Fügt die neue/geänderte Datei zum Index hinzu
$ git add folder-1/file.txt
## Löscht die Datei in der Workcopy und löscht diese Datei im Index
$ rm folder-1/file.txt
$ git rm folder-1/file.txt
## Löscht die Datei in der Workcopy und gleichzeitig im Index
$ git rm folder-1/file.txt
## Fügt alle neuen/geänderten/gelöschten Dateien zum Index hinzu
$ git add -u .
```

BEFEHLE - STATUS

`git status`

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   lectures/02-vcs.adoc

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

        images/02-vcs/git-transport-local.png
        images/02-vcs/staging-flow-1.png
        images/02-vcs/staging-flow-2.png
        images/02-vcs/staging-flow-3.png
        images/02-vcs/staging-flow-4.png
        images/02-vcs/staging-flow-5.png
        images/02-vcs/staging-flow-6.png

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

HEAD, ORIG_HEAD

- Zeiger auf Commits
  - HEAD
    - Referenz auf den Commit, mit dem der aktuelle Working-Tree / Workcopy assoziiert wird
  - ORIG_HEAD
    - Alter Wert von HEAD, der immer dann gesetzt wird, wenn HEAD verändert wird (z.B. `git commit`)
  - Nützlich bei allen Kommandos, die eine commit-ID als Input nehmen , z.B.
    - `git log HEAD`
    - `git reset –hard HEAD`

SPECIFYING REVISIONS

- Zeiger (auf Commits) dereferenzieren
  - (https://git-scm.com/docs/gitrevisions)
  - „Navigation“ von einem Commit ausgehend, auf Basis dessen Vorgängern
    - `HEAD^` → erster Parent von HEAD (unter Windows: `HEAD^^`)
    - `HEAD^1` → erster Parent von HEAD
    - `HEAD~` → erster Parent von HEAD
    - `HEAD~2` → zweiter Parent von HEAD
    - `master~`--> erster Parent von master
    - `HEAD^^` → Parent der zweiten Generation von HEAD (== `HEAD^1^1`)

SPECIFYING REVISIONS

- Zeiger (auf Commits) dereferenzieren
  - (https://git-scm.com/docs/gitrevisions)
  - „Navigation“ von einem Commit ausgehend, auf Basis dessen vorheriger Werte
    - `HEAD@{2}` → zweit-letzter Wert von HEAD
    - `HEAD@{5.minutes.ago}` → Wert von HEAD vor 5 Minuten

ÄNDERUNGEN VERWERFEN

- Der pure `reset`-Befehl entfernt die Änderungen aus dem Stage-Bereich
  - Der Workcopy bleibt unverändert
    - außer bei `--hard`
      - Das Argument HEAD muss angegeben werden
- https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

```
## Änderungen im Stage-Bereich von foo.txt verwerfen
$ git reset HEAD foo.txt
## Alle Änderungen im Stage-Bereich verwerfen
## (Workcopy bleibt unverändert)
$ git reset HEAD
## Alle Änderungen im Stage-Bereich & Workcopy verwerfen
$ git reset --hard HEAD
```

ÄNDERUNGEN VERWERFEN

- Der checkout-Befehl verwirft die Änderungen des Workspace und holt die Version aus dem aktuell gültigen Commit

```
## Änderungen einer Datei verwerfen
$ git checkout -- foo.txt
## Änderungen einer Datei verwerfen - anders
$ git checkout HEAD foo.txt
```

ÄNDERUNGEN VERWERFEN

- Ein bereits erfolgter Commit kann Rückgängig gemacht werden
  - entweder: Commit entfernen & Änderungen behalten
  - oder: Commit entfernen & Änderungen zurücknehmen

```
## Änderung des Commits bleiben im Workspace, aber
## HEAD wird auf seinen Vorgänger gesetzt
$ git reset HEAD^
## Änderungen des Commits werden verworfen
$ git reset –-hard HEAD^
## Änderungen bleiben im Stage-Bereich und im Workspace
## lediglich HEAD wird auf seinen Vorgänger gesetzt
$ git reset –-soft HEAD^
```

COMMITS ANSEHEN

- Liste der Commits
  - Anzeige aller bisherigen Commits
    - `git log`
  - Schönere Anzeige
    - `git log --graph --oneline`
- Einzelnen Commit
  - `git show {commit-sha}`
  - `git cat-file -p {commit-sha}`

TIPPS

LINKS

- https://git-scm.com/book/en/v2
- https://learngitbranching.js.org/
- https://medium.freecodecamp.org/understanding-git-for-real-by-exploring-the-git-directory-1e079c15b807
- https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

EDITOR FÜR COMMIT-NACHRICHTEN

- Commit Messages ohne Vim
  - *erspart* Editor in der Konsole
  - bei `git commit` kann das `-m` nun weggelassen werden

Windows

```
$ git config --global core.editor 'C:\Program Files (x86)\Notepad++\n
```

Mac

```
$ git config --global core.editor "code --wait"
```

ALIAS FÜR HISTORIE

- Folgenden Befehl eingeben, um `git hist` verwenden zu können

```
$ git config --global alias.hist "log --pretty=format:'%C(yellow)[%ad
```

KOMMANDOZEILE

- `dir` → Auflisten aller Dateien in einem Verzeichnis
- `cd ordner1` → Wechsel in des Unterverzeichnis *ordner1*
- `cd ..` → Wechsel in das nächsthöhere Verzeichnis
- `mkdir ordner2` → Erstellen eines neuen Unterverzeichnisses