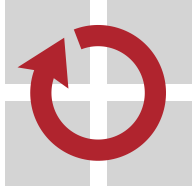
Praktikum angewandte Systemsoftwaretechnik (PASST)

Git-Internals

16. November 2020

Dustin Nguyen, Tobias Langer, Jonas Rabenstein, Phillip Raffeck

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



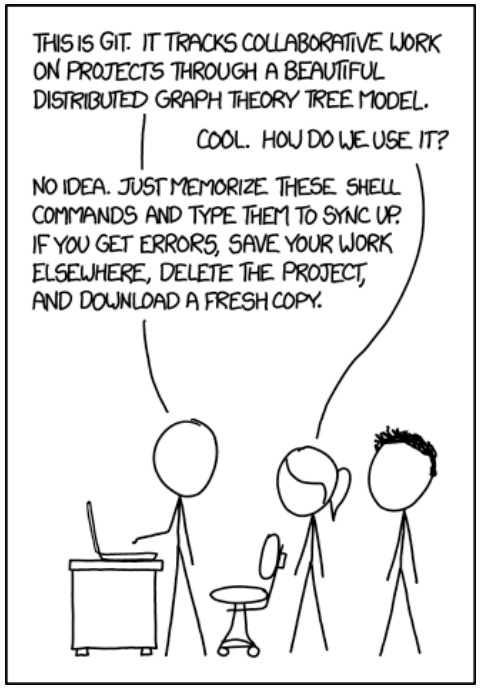
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Rückblick

Rückblick



https://imgs.xkcd.com/comics/git.png

- Versionsverwaltung mit **Git**
  - Grundlegende Versionsverwaltungskonzepte mit **Git**
  - Arbeiten am eigenen Repository
  - Kollaboratives Arbeiten mit anderen

Motivation

Motivation

- Entwicklungsprozesse sind selten linear
  - Experimentelle Features werden getestet
  - Änderungen führen Fehler ein
  - Änderungen müssen zusammengeführt werden
- Versionierung unterstützt Entwicklungsprozesse
  - Dokumentation von Änderungen
  - Dokumentation von Versionsständen
  - Verfahren zum Zusammenführen von Versionsständen

$\Rightarrow$ Manuelle Versionierung ist mühsam & fehleranfällig

Lernziele

Lernziele

Im Anschluss an diesen Vortrag solltet Ihr...

- grundlegend die interne Funktionsweise des Versionskontrollsystems **Git** beschreiben
- fortgeschrittene **Git-Konzepte** erklären

...können.

Agenda

Fakten & Infos rund um Git

Funktionsweise von Git

Fortgeschrittene Git-Konzepte

Zusammenfassung

Git++

Fakten & Infos rund um Git

Das Versionskontrollsystem des Linux-Kernels

Git

- Entwicklung 2005 von Linus Torvalds initiiert
- „I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'" – Linus Torvalds
- Ausgelegt zur Unterstützung der Linux-Kernel Entwicklung

Zentrale Eigenschaften:

- Unterstützung für dezentrale & parallele Entwicklung
- Visualisierung von Entwicklungszweigen
- Ausgelegt für Umgang mit großen Patchmengen
- Integrität der Historie durch SHA-1 Hash

Statistik zur Git Repo des Kernels

Ein paar Zahlen zur Kernel-Entwicklung:

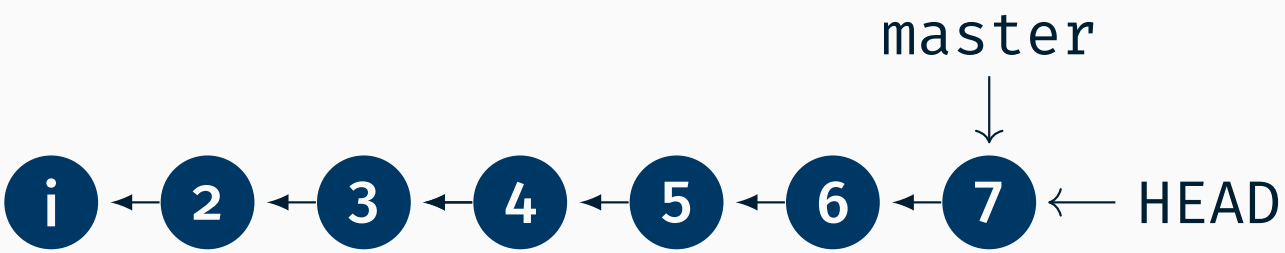
Die Git-Historie des Linux-Kernels (Stand: Kernel v5.9.8) umfasst

- `1183010 Commits`
- `71520 Merge-Commits` (inklusiv)
- `6842 Tags`
- `22280 Contributors`

Funktionsweise von Git

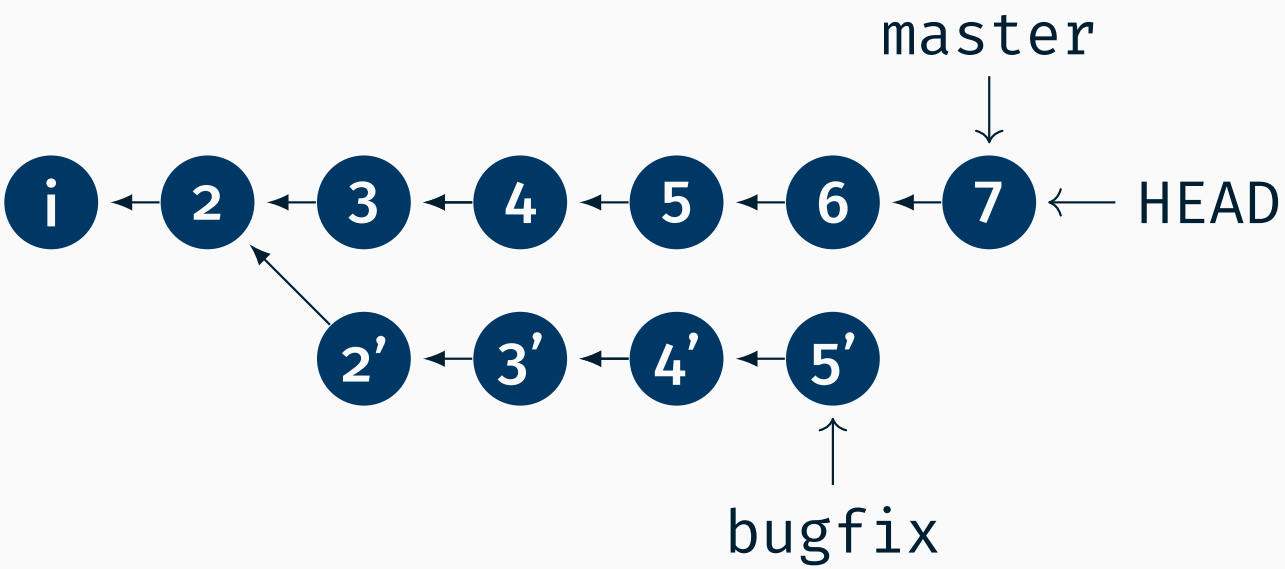
Wie funktioniert Git?

- Sicherung vollständiger Daten jedes Versionsstandes („commit“)
- Eindeutige SHA1-Hashes für jede Version
- Jede Version kennt ihren Vorgänger („parent“)
- Jede Versionsserie („branch“) bekommt einen Namen
- HEAD verweist auf den aktuellen Commit im Workspace

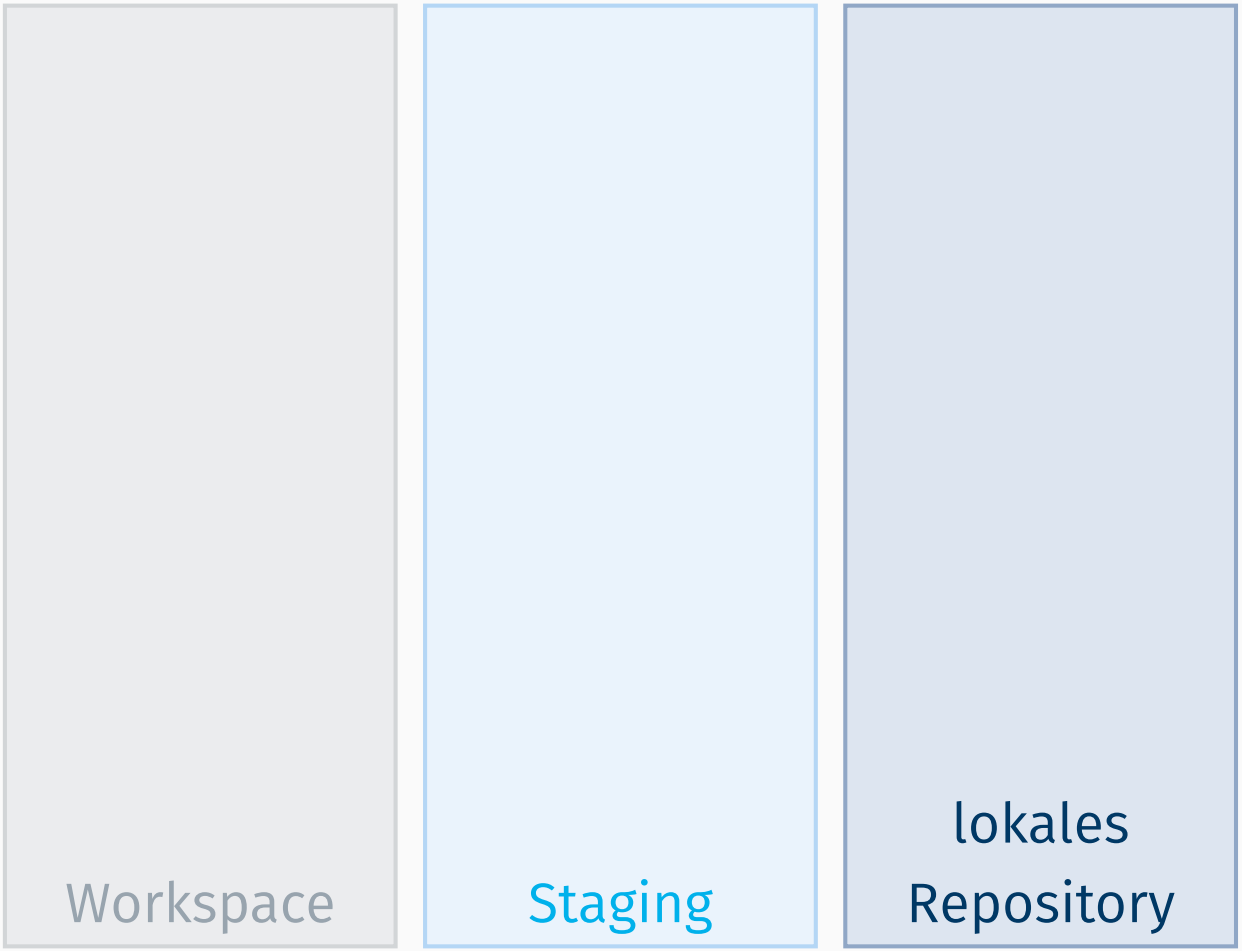


Wie funktioniert Git?

- Sicherung vollständiger Daten jedes Versionsstandes („commit“)
- Eindeutige SHA1-Hashes für jede Version
- Jede Version kennt ihren Vorgänger („parent“)
- Jede Versionsserie („branch“) bekommt einen Namen
- HEAD verweist auf den aktuellen Commit im Workspace

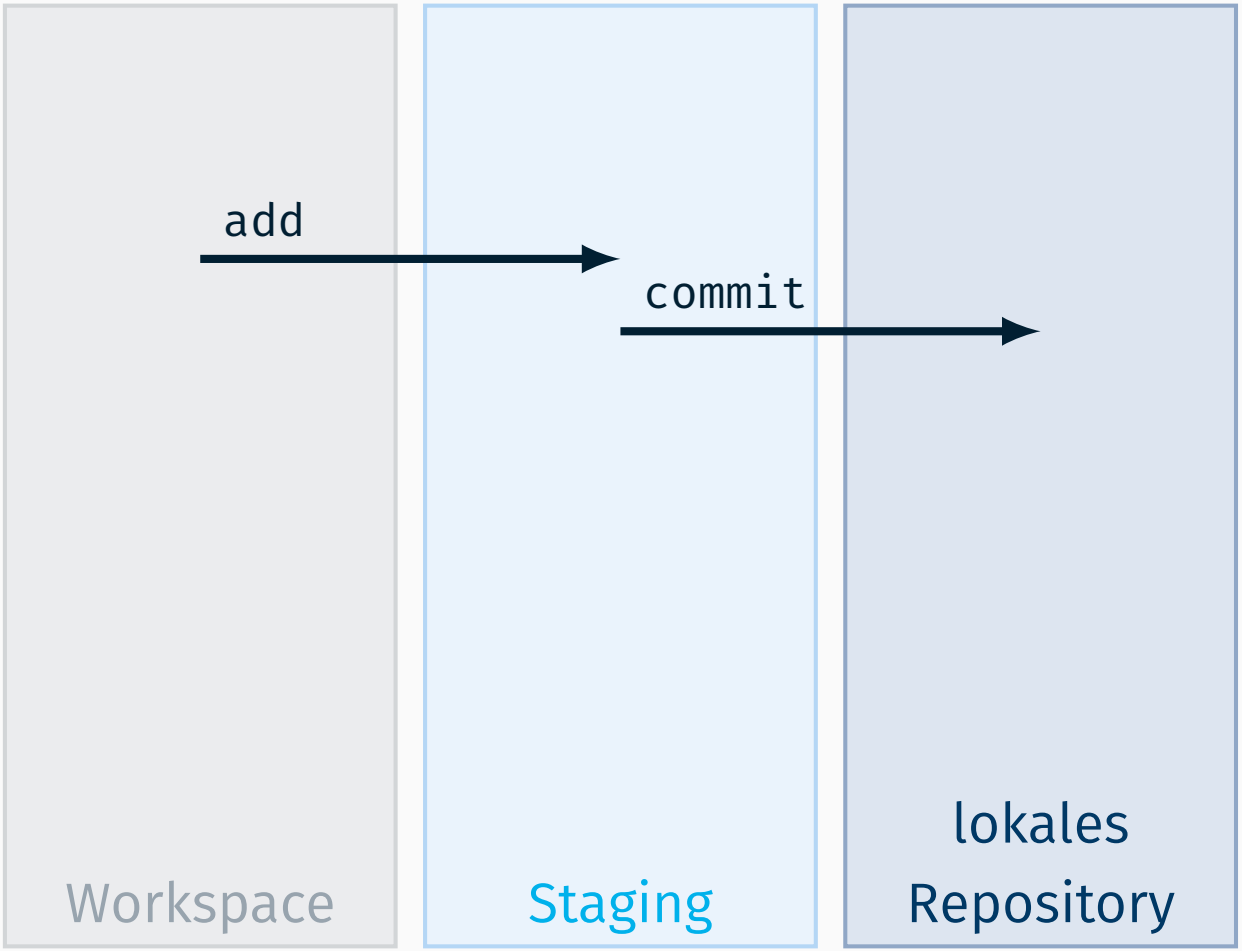


Git Workflow



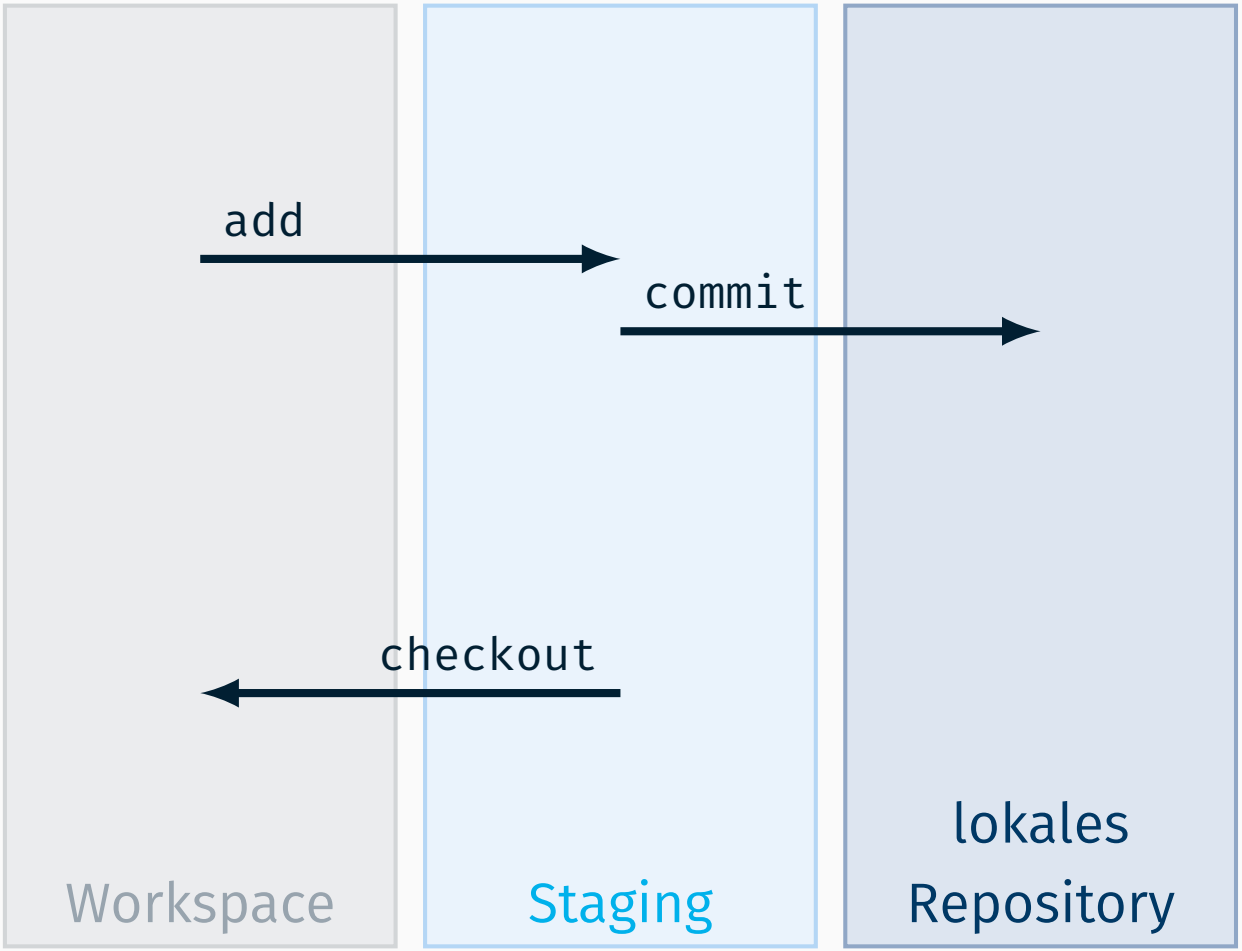
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



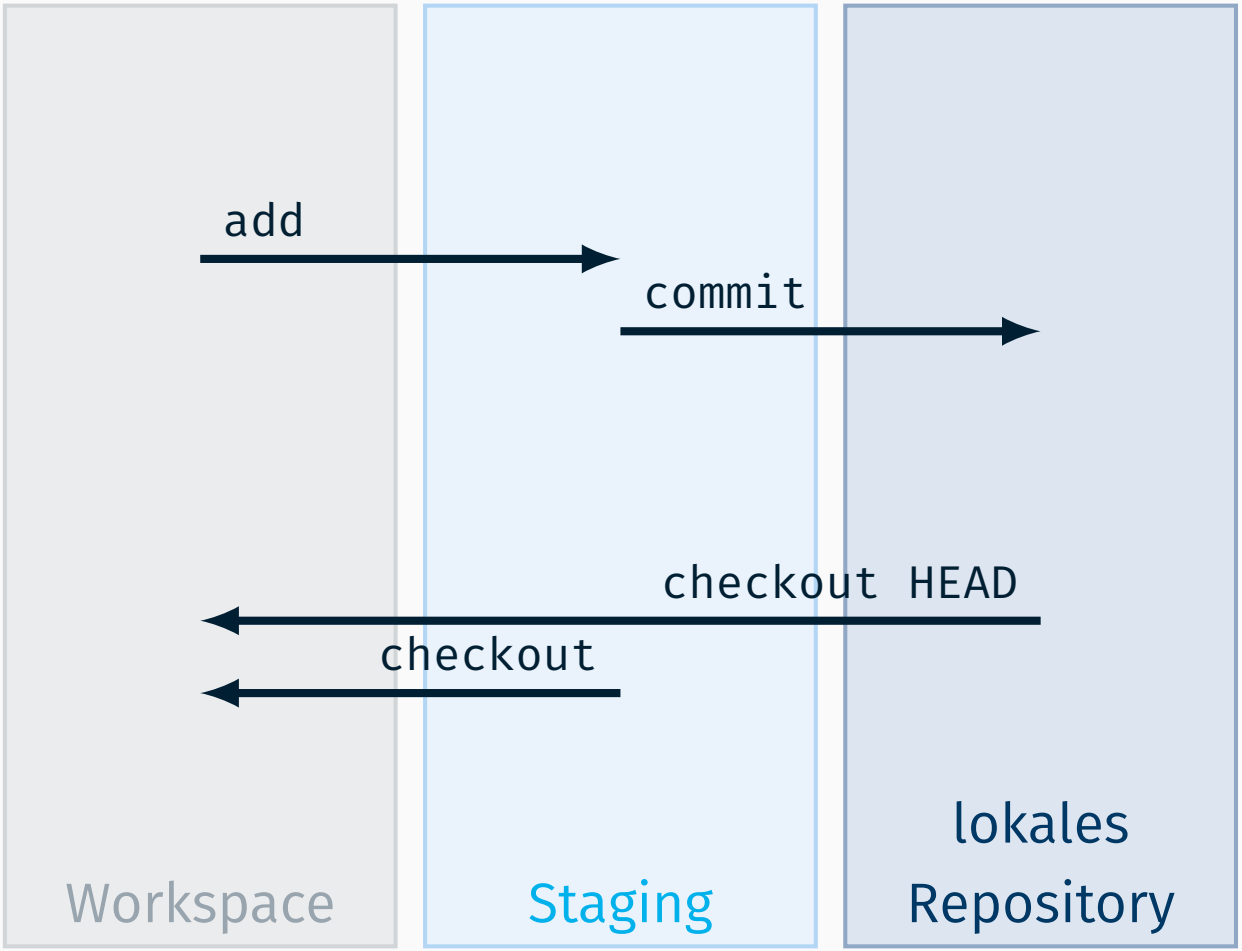
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



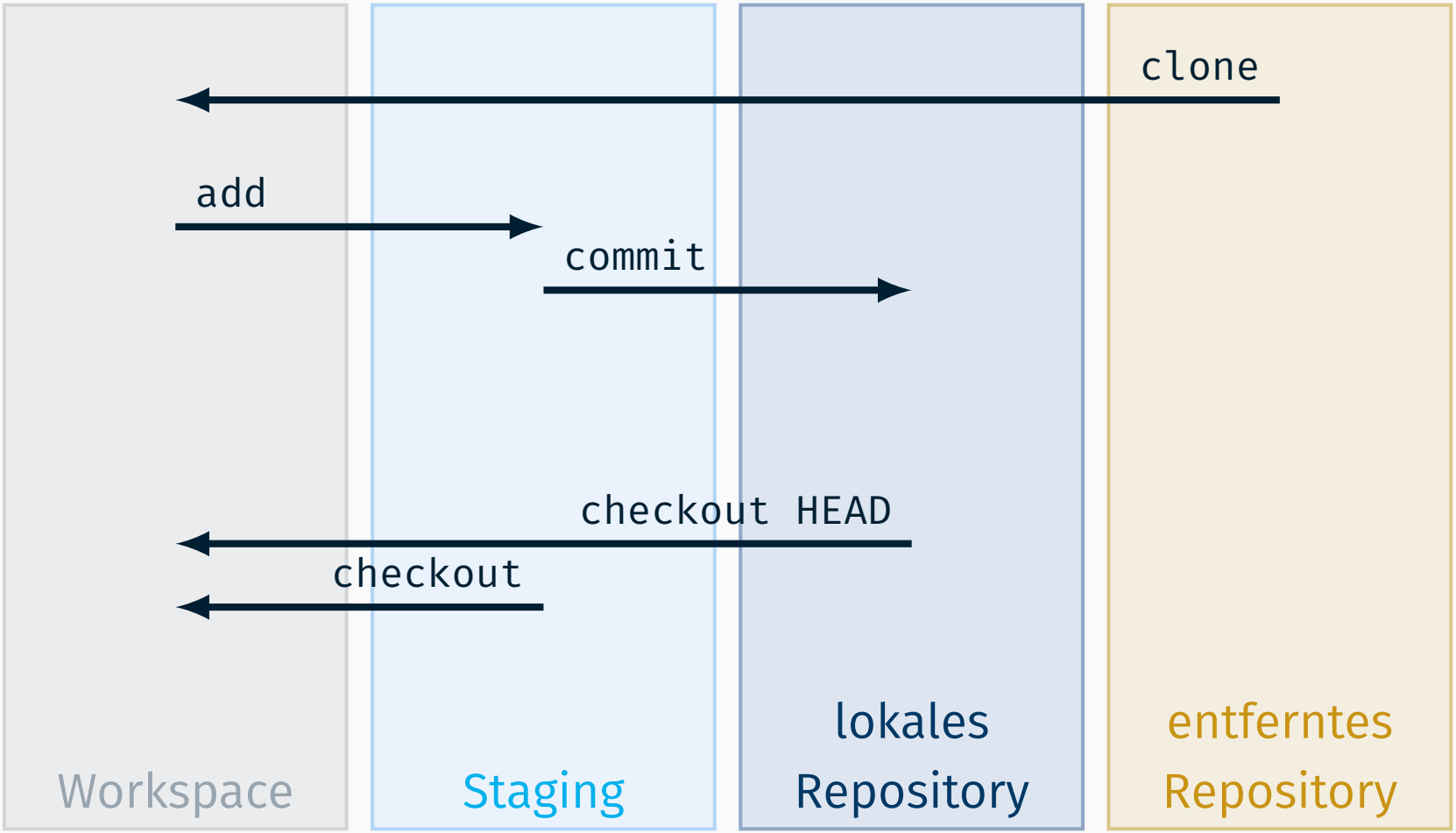
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



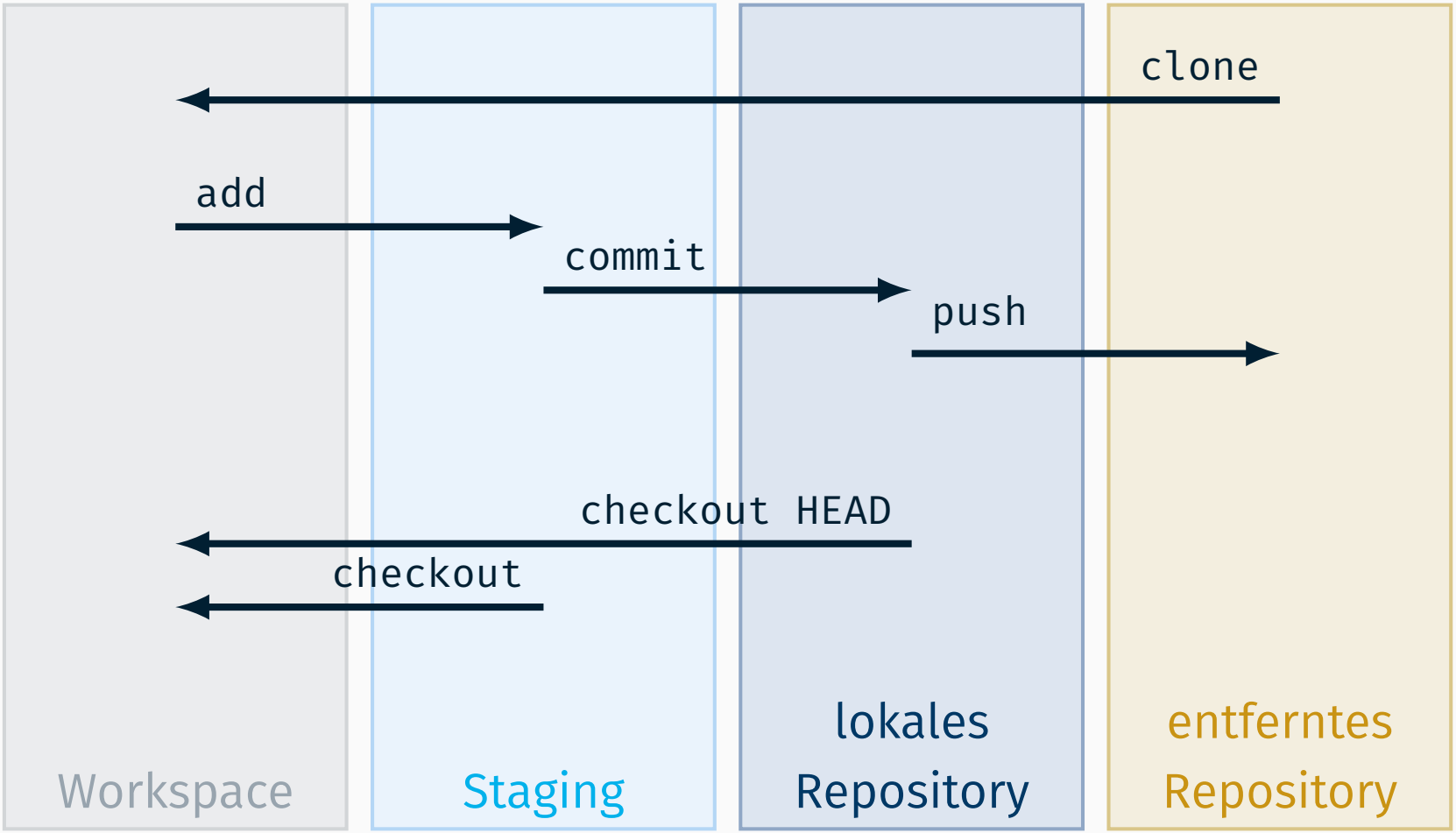
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



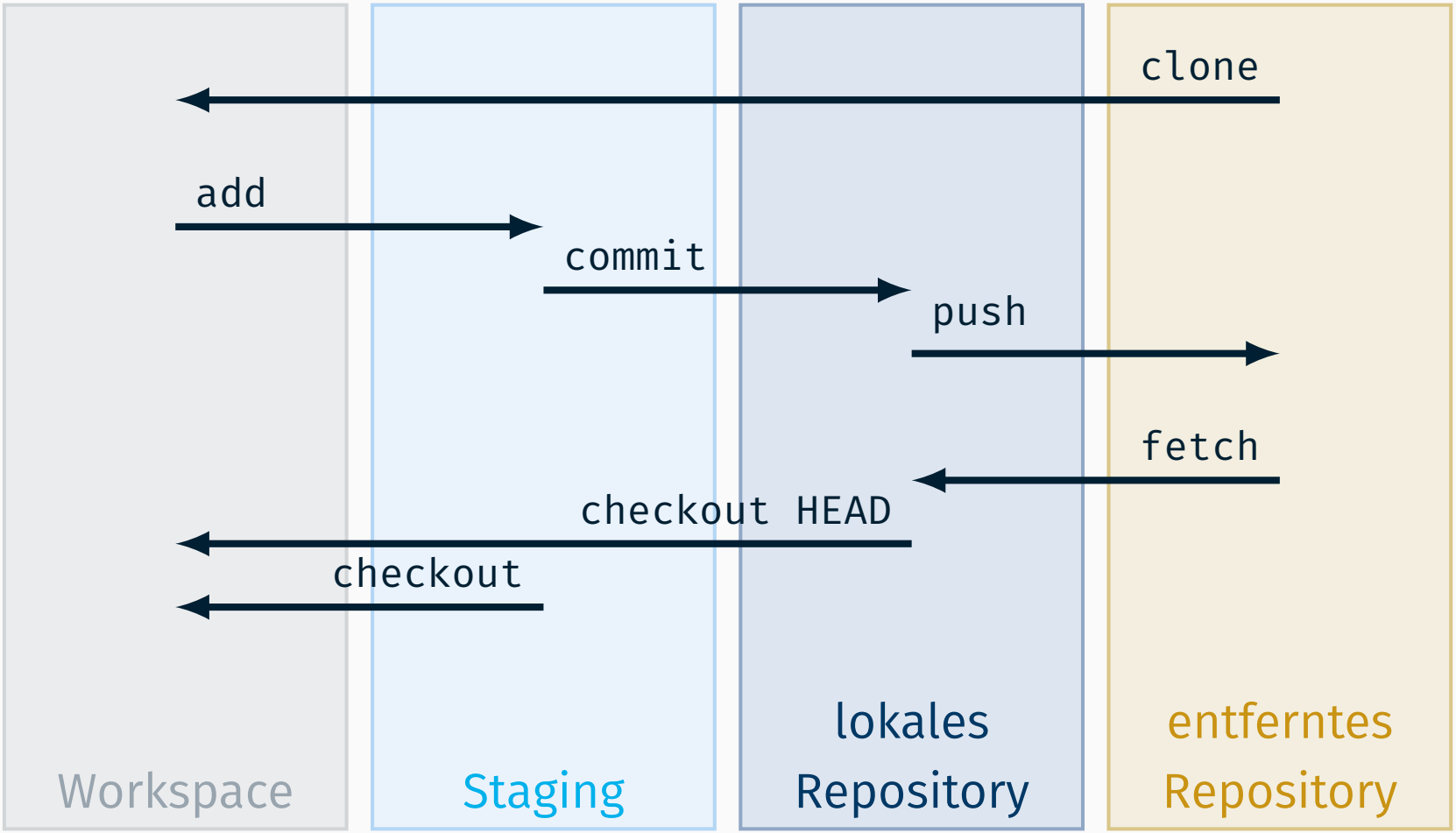
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



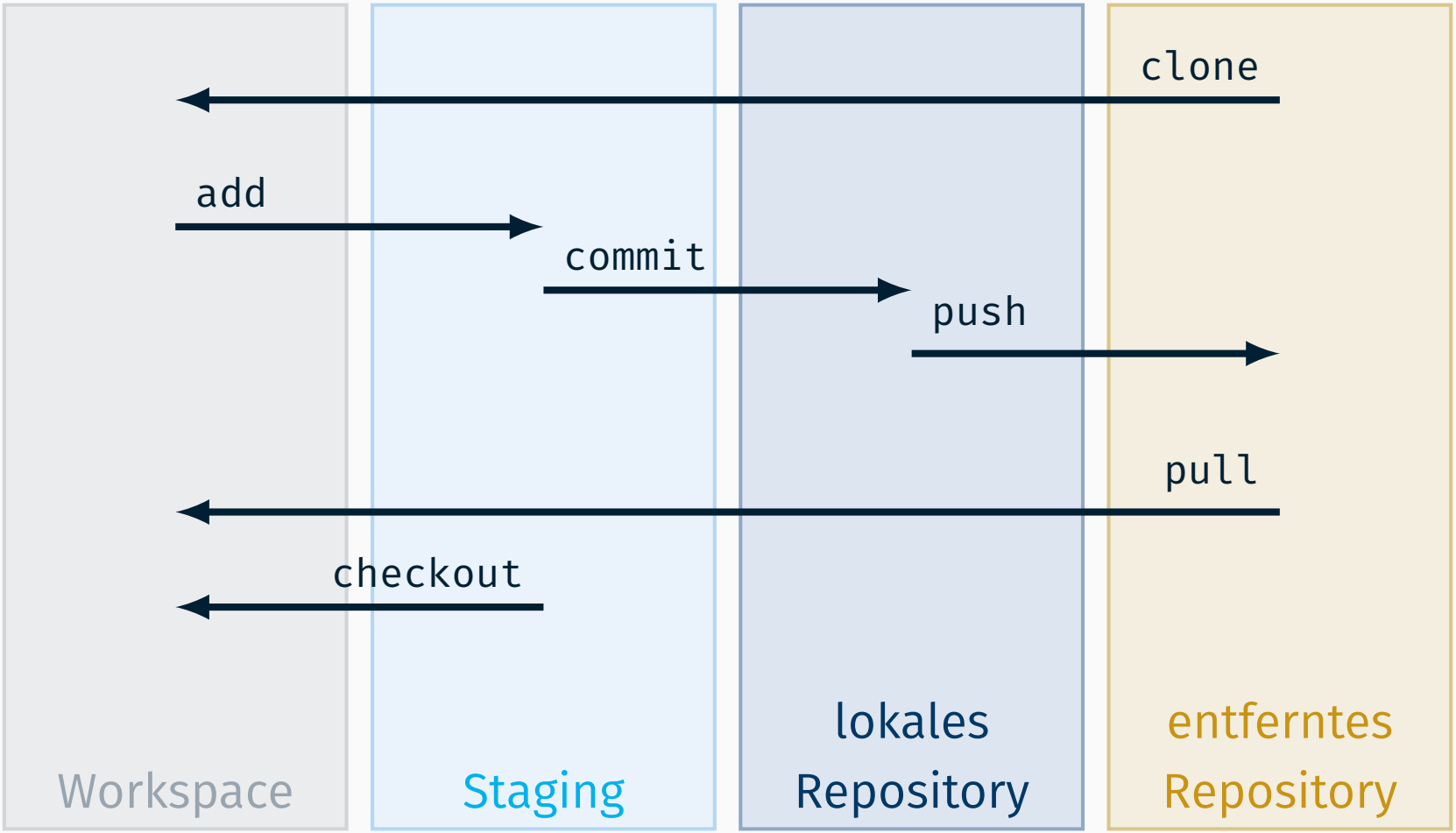
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



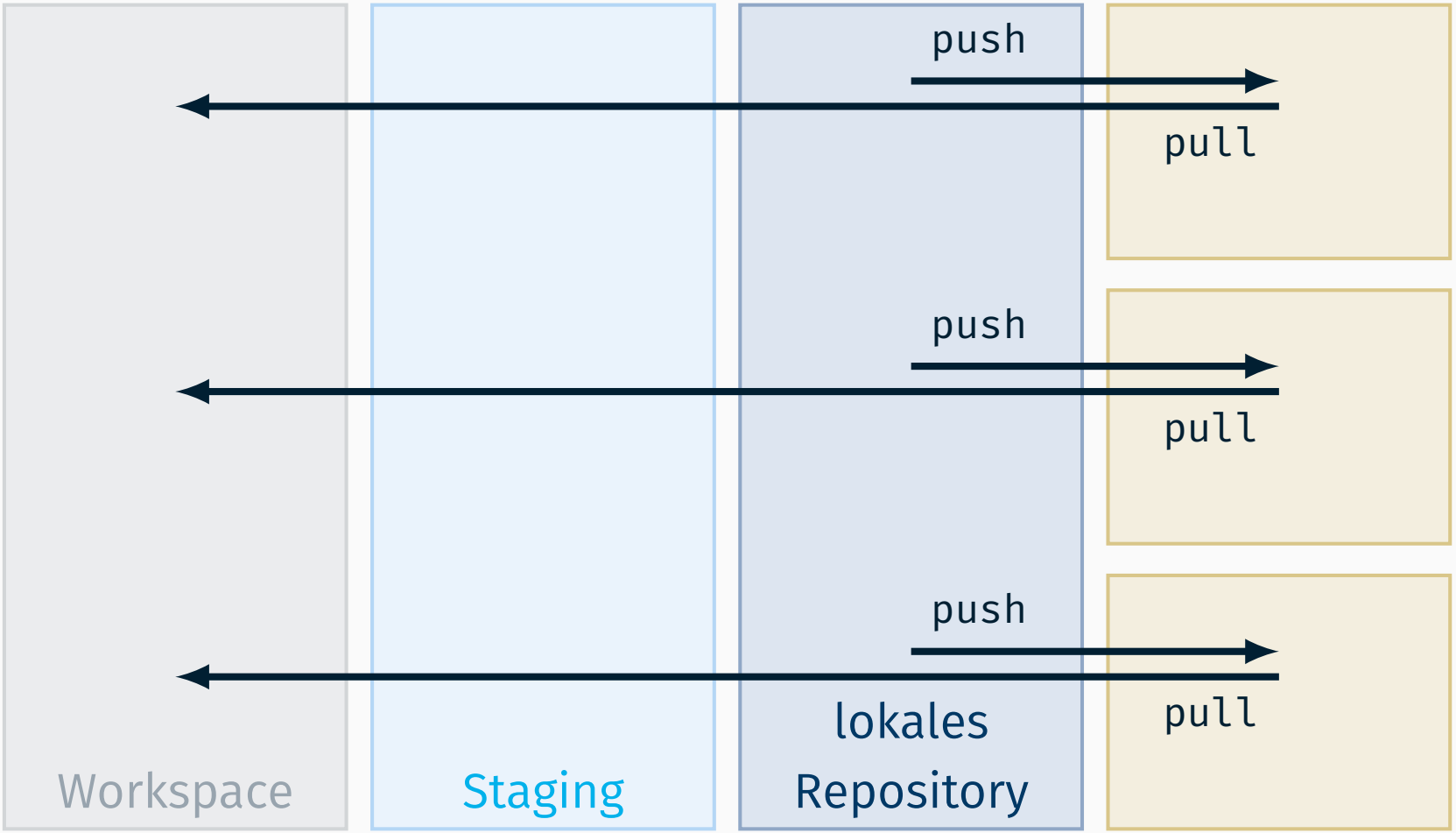
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



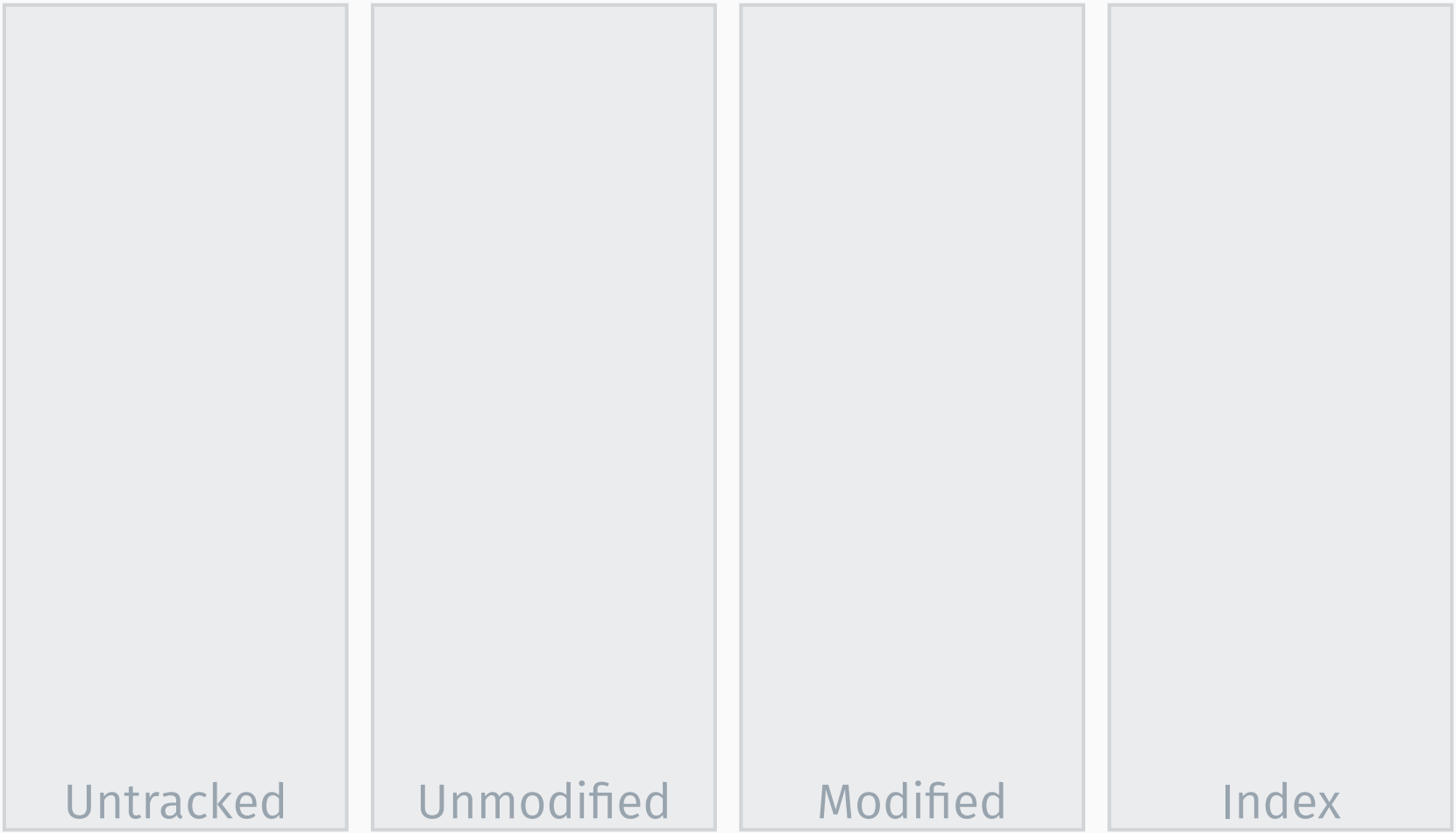
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workflow



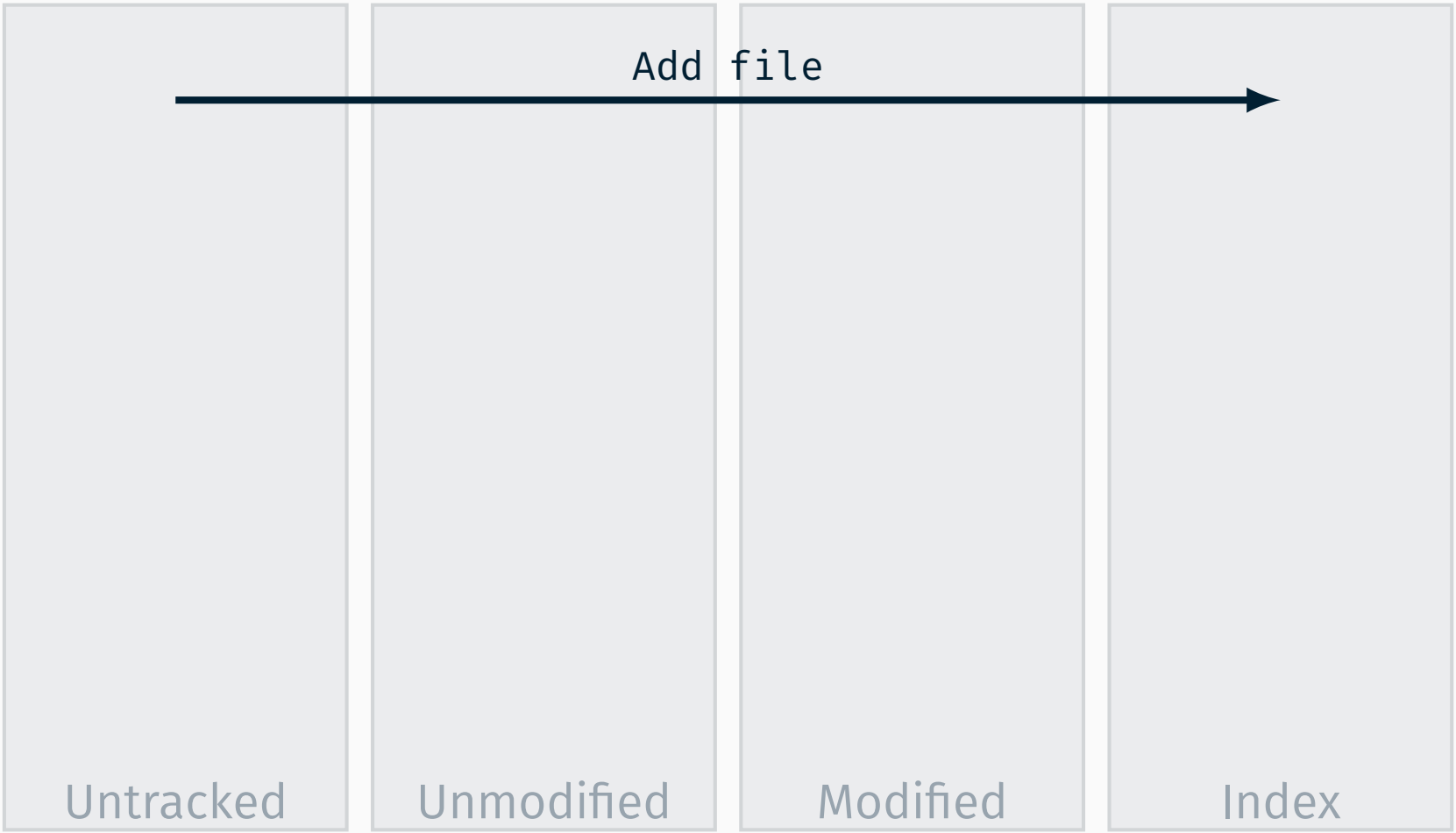
Für weitere Informationen:
`https://blog.osteele.com/2008/05/my-git-workflow`

Git Workspace



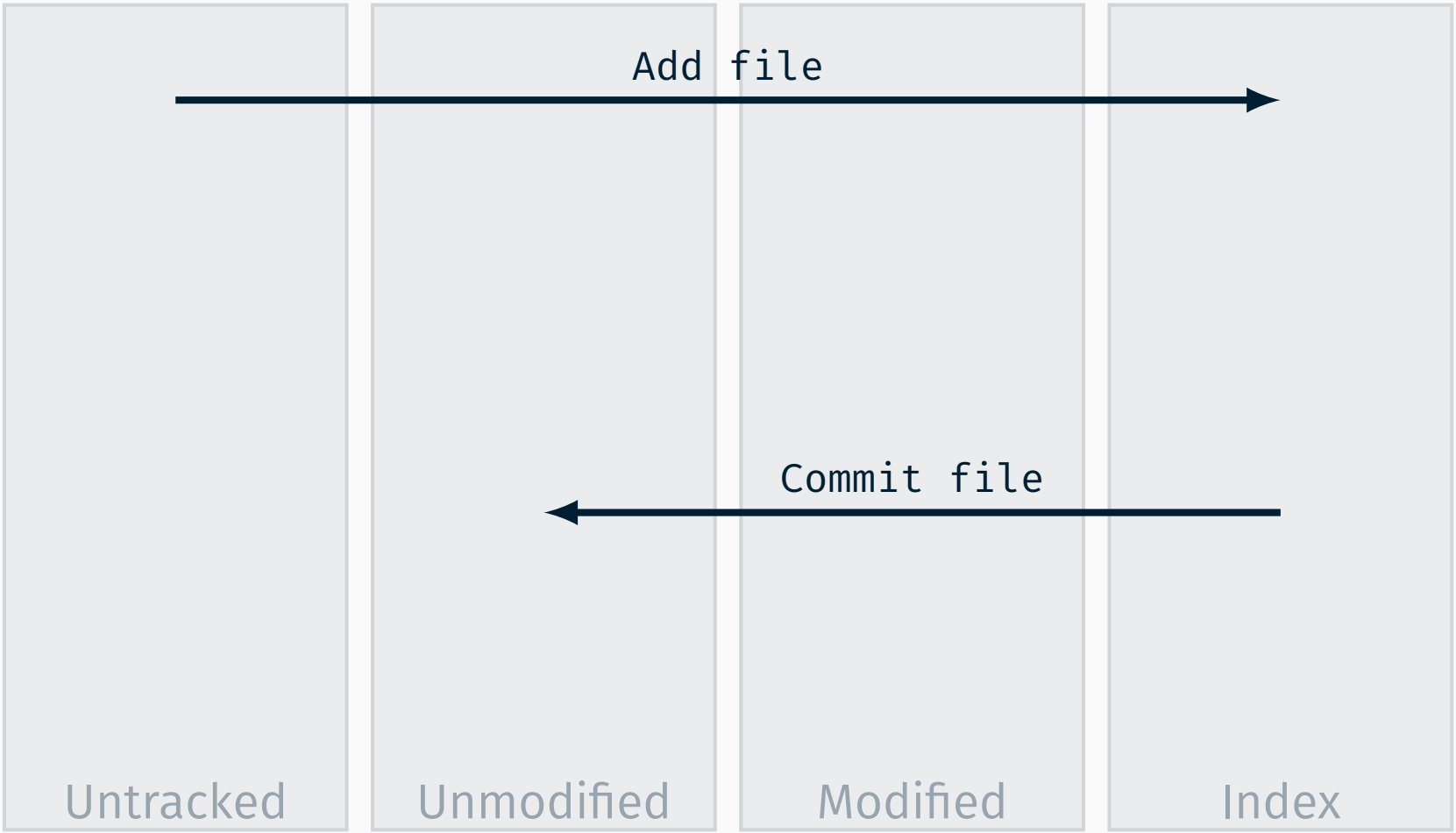
Für weitere Informationen: `https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository`

Git Workspace



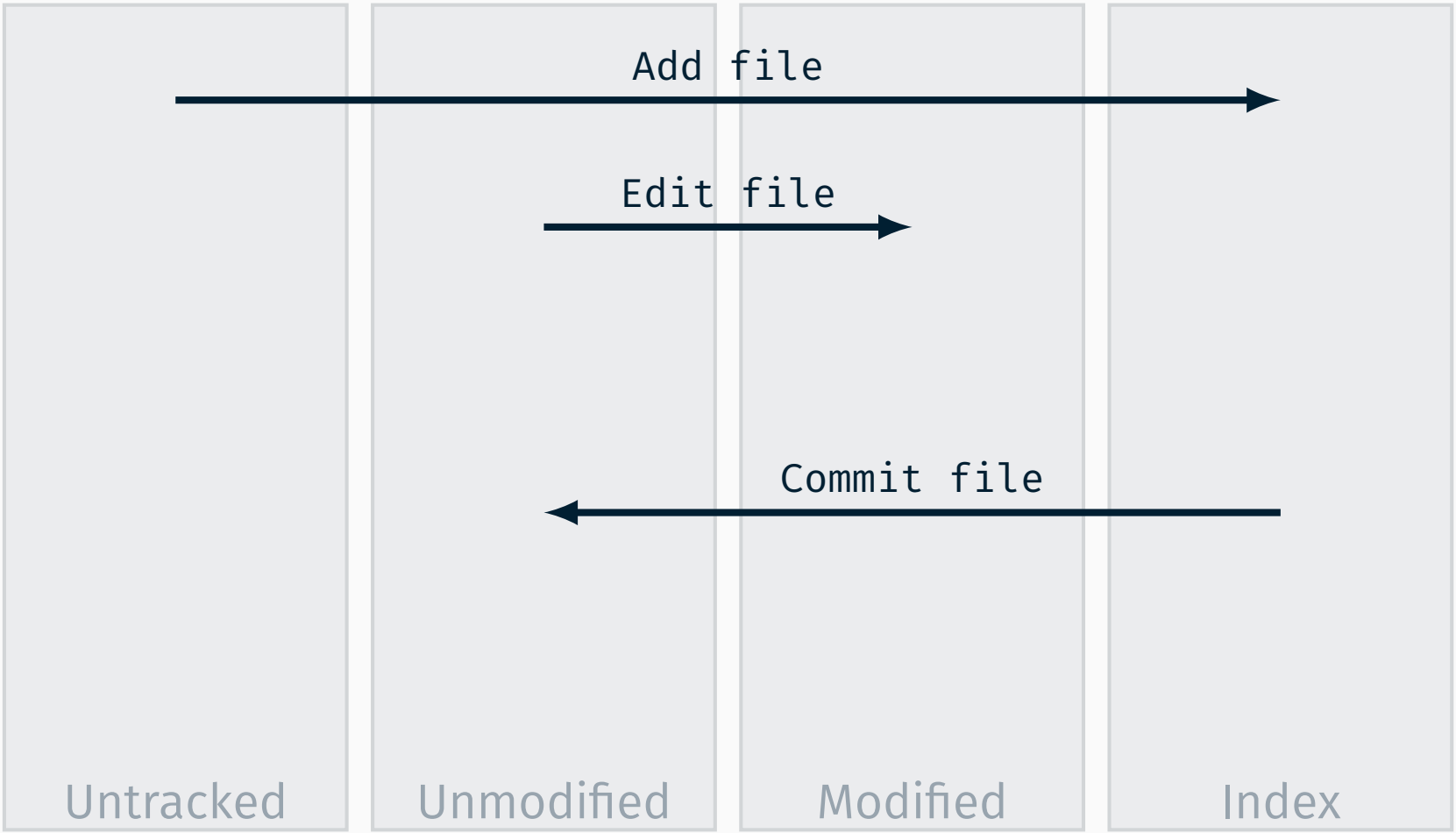
Für weitere Informationen: `https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository`

Git Workspace



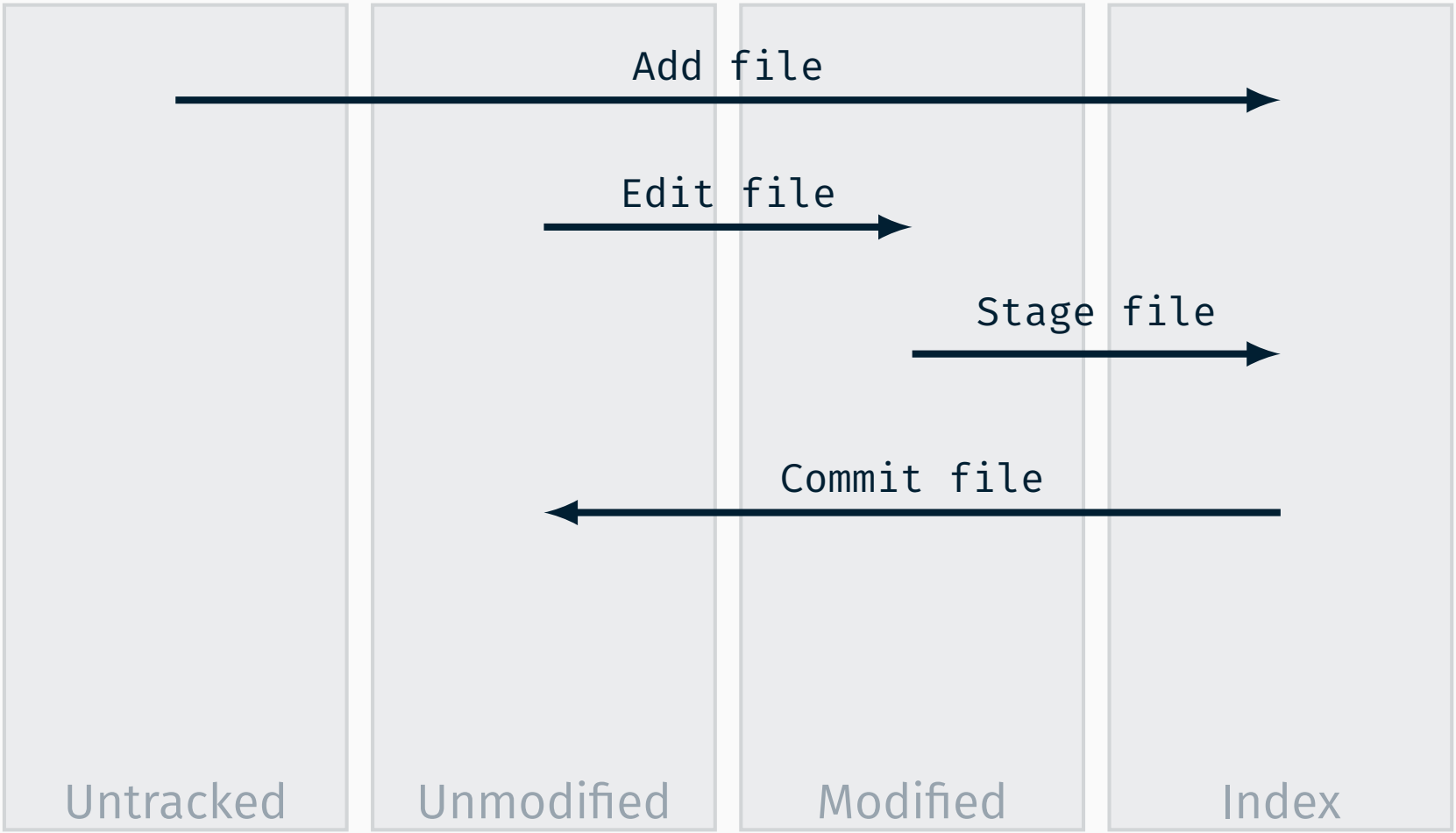
Für weitere Informationen: `https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository`

Git Workspace



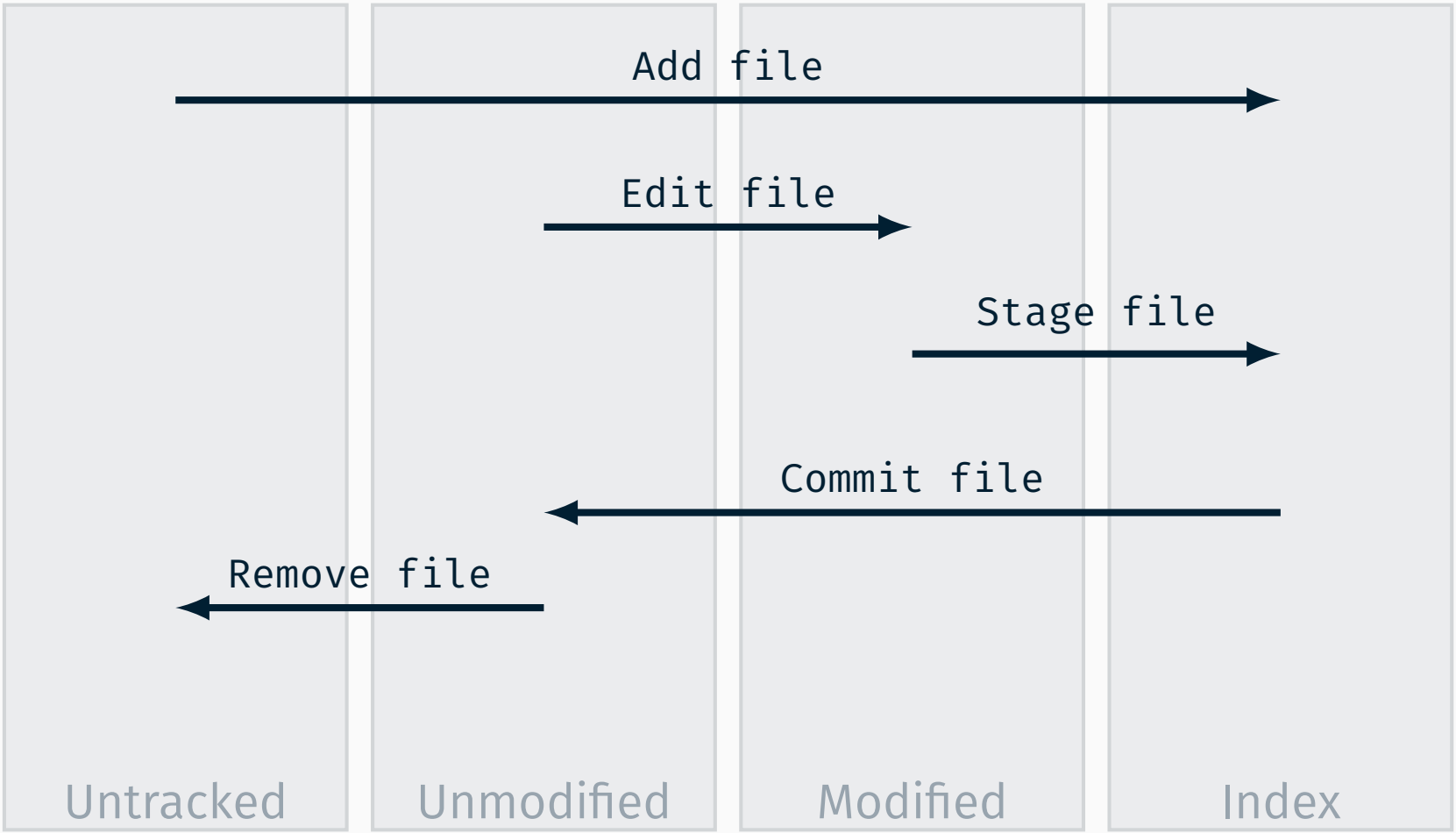
Für weitere Informationen: `https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository`

Git Workspace



Für weitere Informationen: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Git Workspace



Für weitere Informationen: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

Wie funktioniert Git *intern*? (1/5)

Git Interna

Git arbeitet intern mit sogenannten **Git Objects**

Diese Objekte können ein...

- **Blob**: Inhalt einer (oder mehrerer) Dateien
- **Tree**: Verzeichnis mit Referenzen auf dessen Unterverzeichnisse/Dateien
- **Commit Object**: Vorhandener Commit
- **Tag Object**: Vorhandener Tag

... sein.

Wie funktioniert Git *intern*? (2/5)

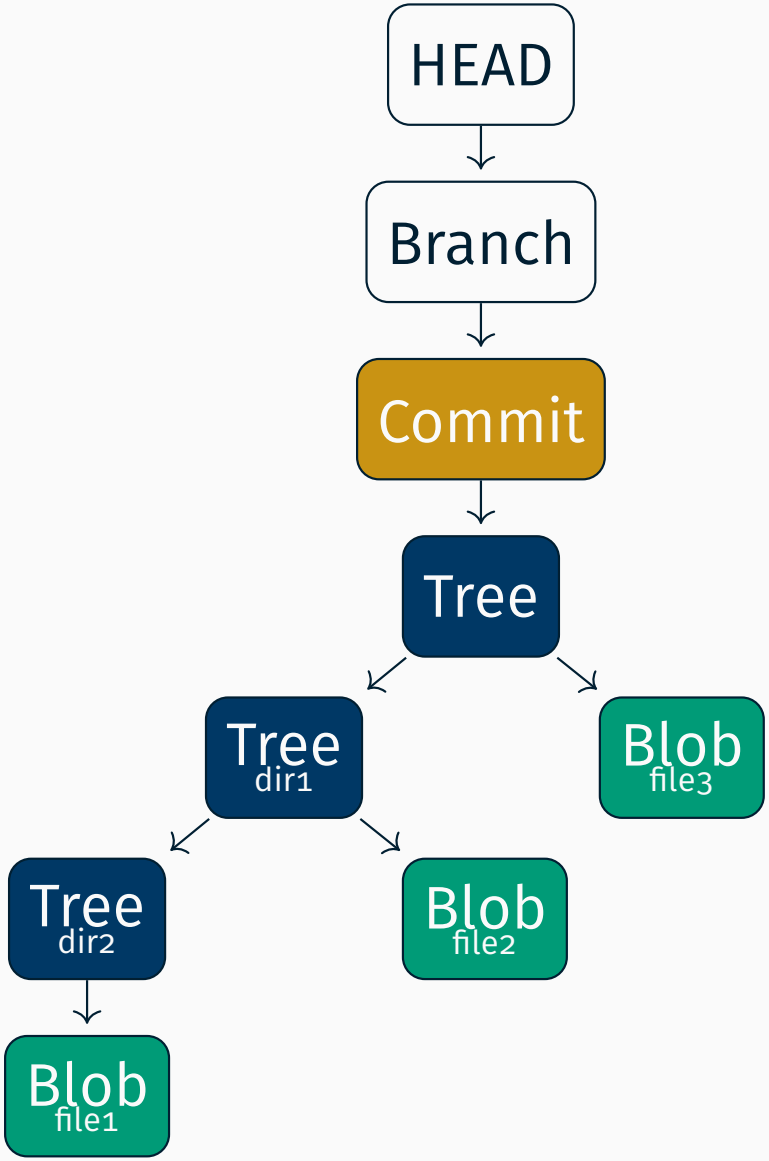
Aufbau des Repos:

```
$ tree .
.
├── dir1
│   ├── dir2
│   │   └── file1
│   └── file2
└── file3

2 directories, 3 files
```

Wie funktioniert Git *intern*? (2/5)

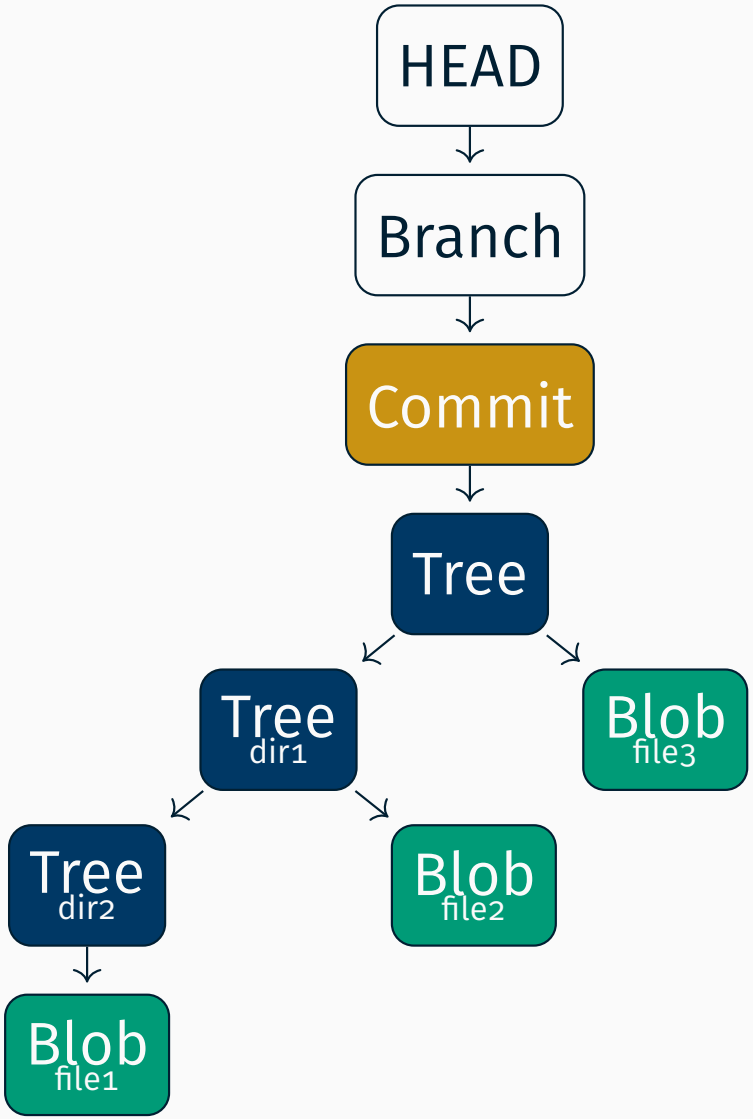
Aufbau des Repos:

```
$ tree .
.
├── dir1
│   ├── dir2
│   │   └── file1
│   └── file2
└── file3

2 directories, 3 files
```



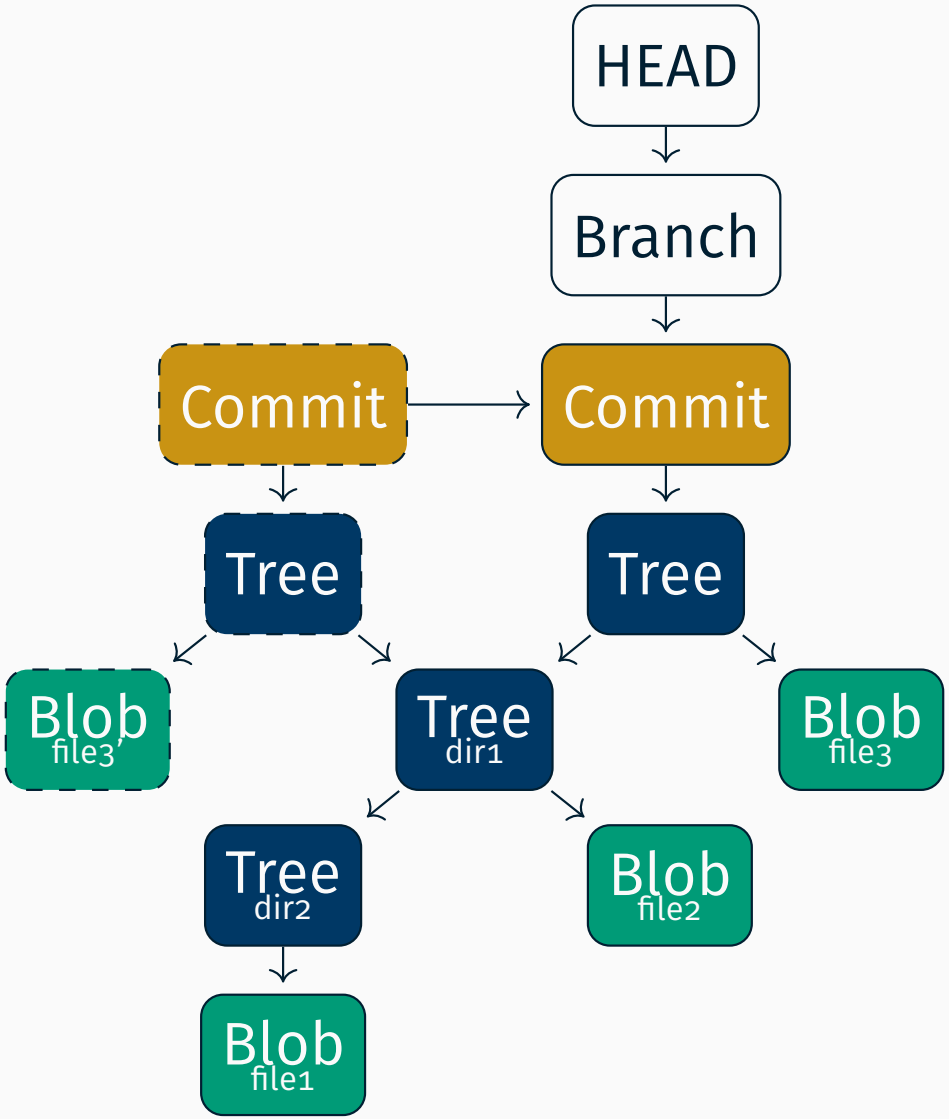
Wie funktioniert Git *intern*? (3/5)

Wie wird die Commit-Historie abgebildet?



Wie funktioniert Git *intern*? (3/5)

Wie wird die Commit-Historie abgebildet?



Wie funktioniert Git *intern*? (4/5)

Wie sieht ein Commit von innen aus?

```
$ git show HEAD --format=raw

commit d6a2895739eb22997408685c22ddc7ee0107aabf
tree aa24ec3f184b2f53cc48c1126f729f7c903a6fcc
parent 312be25d2ea89e78cbbe373947e897792bcaa896
author Phillip Raffeck <phillip.raffeck@fau.de> 1605183929 +0100
committer Phillip Raffeck <phillip.raffeck@fau.de> 1605184746 +0100
```

Wie funktioniert Git *intern*? (5/5)

Wie werden die restlichen Daten gespeichert?

- **Branches**:

  ```
  $ cat .git/refs/heads/master
  81794d4539e8e1797d49862555b1d77c47be7114
  ```

- **HEAD**:

  ```
  $ cat .git/HEAD
  ref: refs/heads/master
  ```

Für weitere Informationen:

`https://github.com/pluralsight/git-internals-pdf`

Wie funktioniert Git *intern*? (5/5)

Wie werden die restlichen Daten gespeichert?

- **Branches**:

  ```
  $ cat .git/refs/heads/master
  81794d4539e8e1797d49862555b1d77c47be7114
  ```

- **HEAD**:

  ```
  $ cat .git/HEAD
  ref: refs/heads/master
  ```

Git Objects

Head, Branches, Tags, ... verhalten sich wie Pointer

Für weitere Informationen:

`https://github.com/pluralsight/git-internals-pdf`

Fortgeschrittene Git-Konzepte

Werkzeugkasten Git

Git bringt eine Vielzahl mächtiger Befehle mit sich

Fortgeschrittene Git-Konzepte

Werkzeugkasten Git

Git bringt eine Vielzahl mächtiger Befehle mit sich

Im Anschluss werden Folgende besprochen:

- `git checkout`
- `git reset`
- `git reflog`
- `git stash`
- `git rebase`
- `git cherry-pick`
- `git bisect`

git checkout (1/2)

`git checkout`

Wechselt Branches und stellt Dateien wieder her

git checkout (1/2)

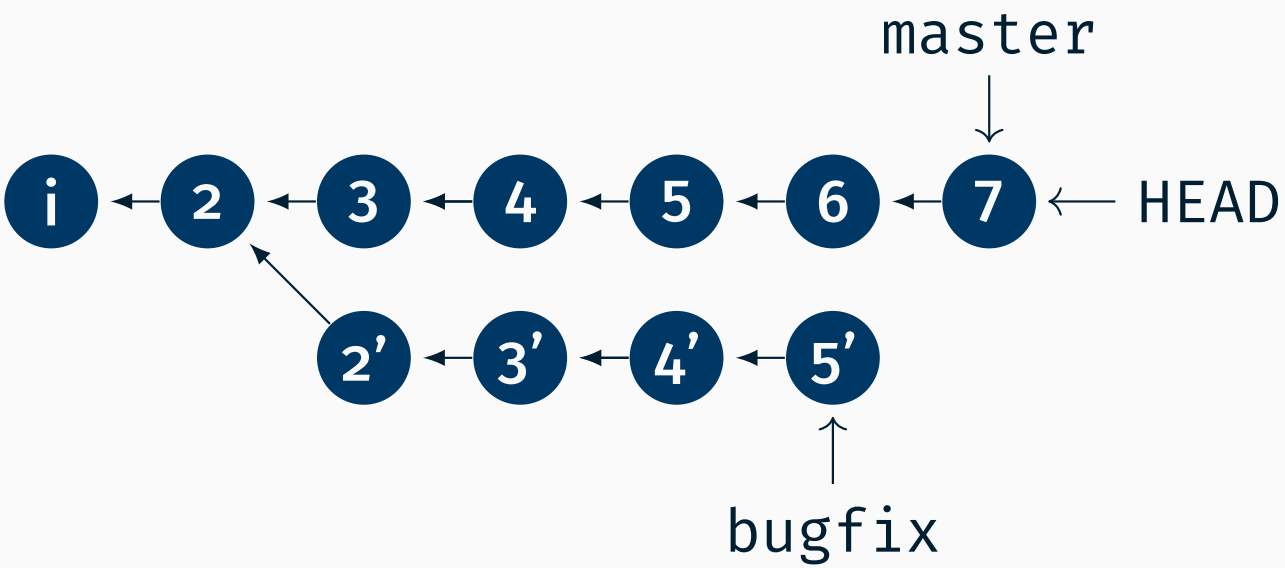
`git checkout`

Wechselt Branches und stellt Dateien wieder her

git checkout (1/2)

`git checkout`

Wechselt Branches und stellt Dateien wieder her

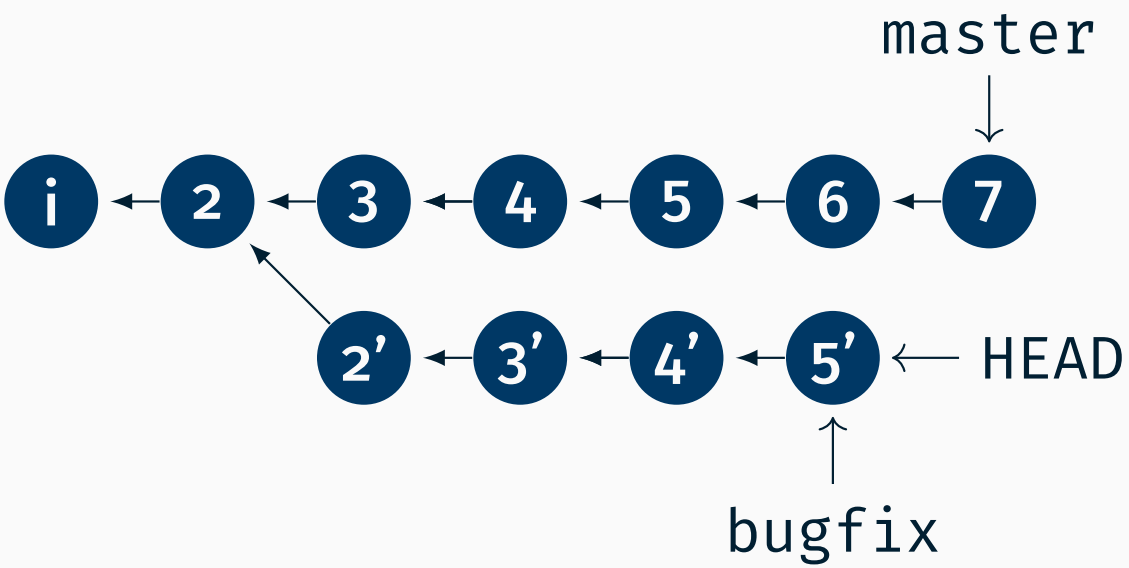



```
$ git checkout bugfix
```

git checkout (1/2)

`git checkout`

Wechselt Branches und stellt Dateien wieder her




```
$ git checkout bugfix
```

git checkout (2/2)

`git checkout` kann auch benutzt werden, um Änderungen zu verwerfen:

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

$ echo "42" >> test

$ git status
On branch master
Changes not staged for commit:
    modified:   test

$ git checkout test    # Restore from working tree

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

git reset (1/5)

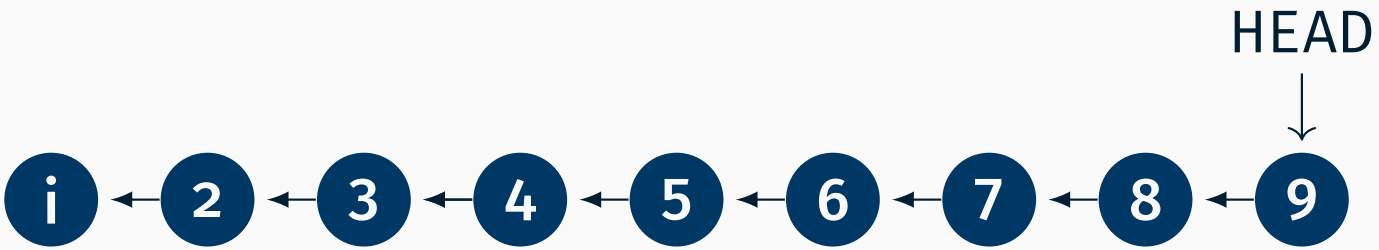
`git reset`

Setzt HEAD Zeiger um

git reset (1/5)

`git reset`

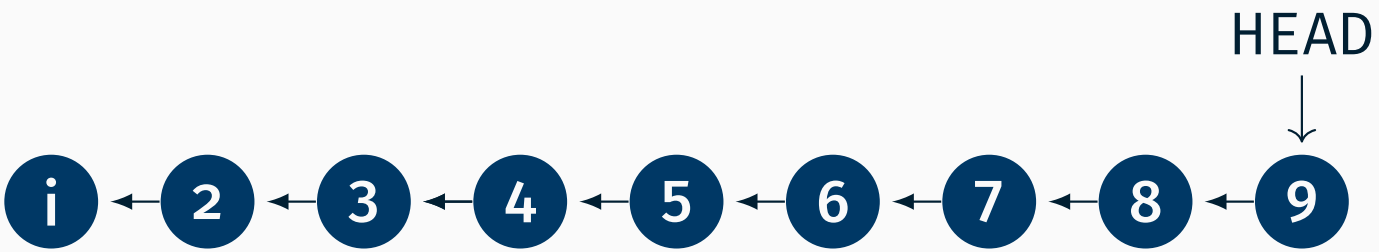
Setzt HEAD Zeiger um



git reset (1/5)

`git reset`

Setzt HEAD Zeiger um

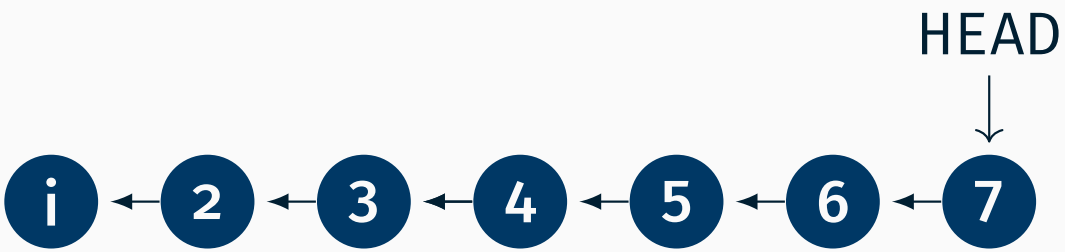



```
$ git reset HEAD~~
```

git reset (1/5)

`git reset`

Setzt HEAD Zeiger um




```
$ git reset HEAD~~
```

git reset (1/5)

`git reset`

Setzt HEAD Zeiger um


```
$ git reset HEAD~~
```

Was passiert mit Commits 8 & 9?

git reset (2/5)

`git reset --soft`

Setzt HEAD zurück ohne Index & Workspace zu verändern

Ausgangsitation:

```
$ git status
Changes to be committed:
    new file:   test1

Changes not staged for commit:
    modified:   test0
Untracked files:
    test2
```

Nach soft-Reset:

```
$ git status
Changes to be committed:
    modified:   test0
    new file:   test1
Changes not staged for commit:
    modified:   test0
Untracked files:
    test2
```

git reset (3/5)

`git reset --mixed`

Setzt HEAD zurück ohne Workspace zu verändern

Ausgangsitation:

```
$ git status
Changes to be committed:
    new file:   test1
Changes not staged for commit:
    modified:   test0
Untracked files:
    test2
```

Nach mixed-Reset:

```
$ git status
Changes not staged for commit:
    modified:   test0

Untracked files:
    test1
    test2
```

Index wird zurückgesetzt!

git reset (4/5)

`git reset --hard`

Setzt HEAD und den Workspace zurück

Ausgangsitation:

```
$ git status
Changes to be committed:
    new file:   test1
Changes not staged for commit:
    modified:   test0
Untracked files:
    test2
```

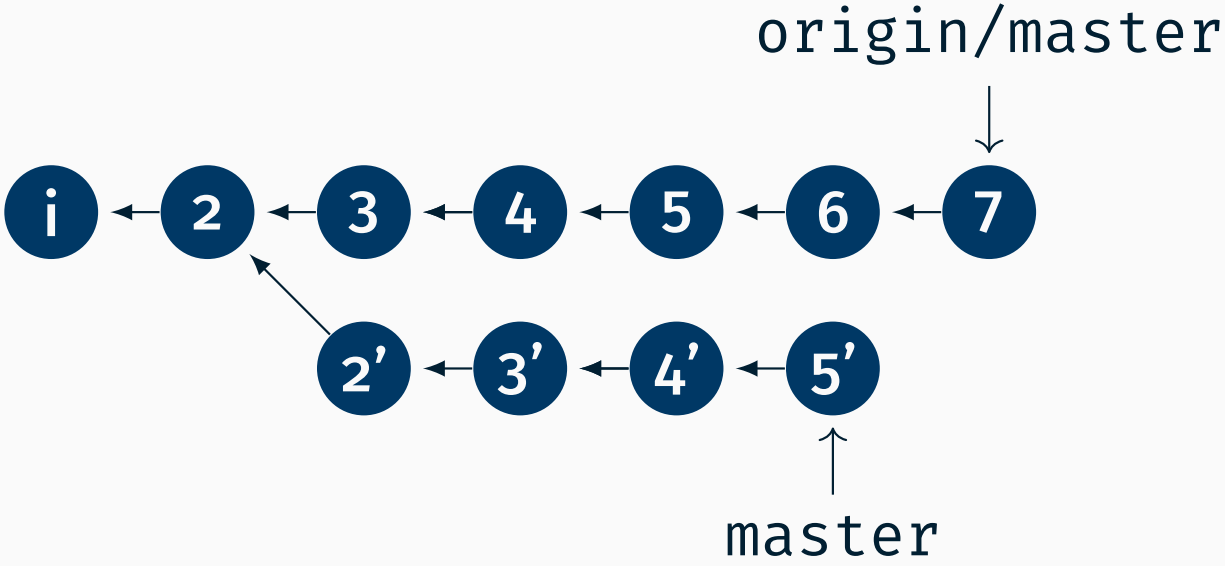
Nach hard-Reset:

```
$ git status
Untracked files:
    test2
```

Alles wird zurückgesetzt!

git reset (5/5)

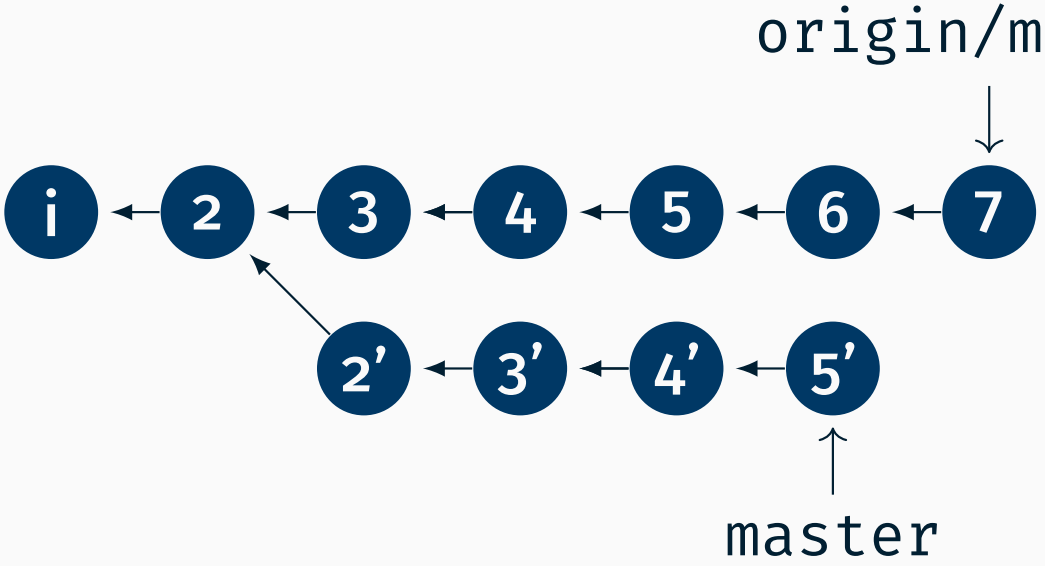
`git reset` kann mehr als nur die Commithistory *zurücksetzen*:




```
$ git checkout master
```

git reset (5/5)

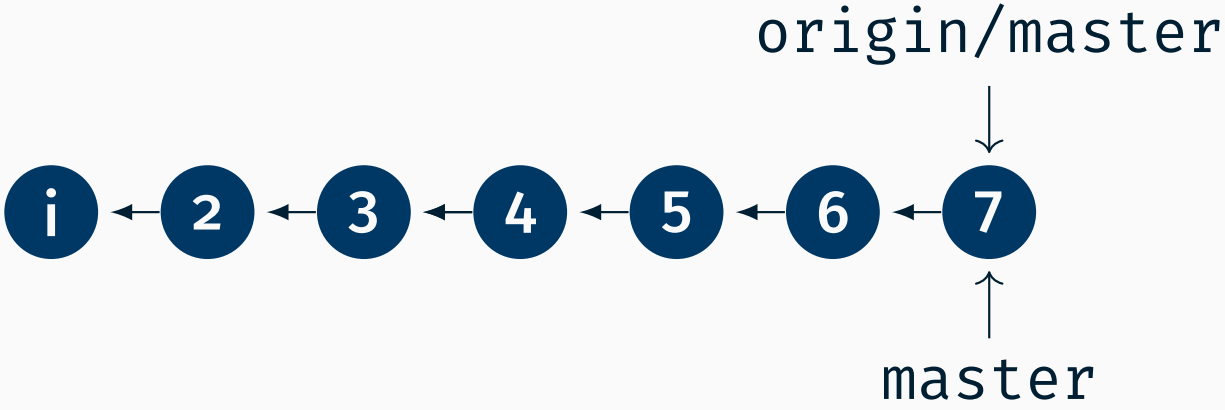
`git reset` kann mehr als nur die Commithistory *zurücksetzen*:




```
$ git checkout master
$ git reset origin/master
```

git reset (5/5)

`git reset` kann mehr als nur die Commithistory *zurücksetzen*:




```
$ git checkout master
$ git reset origin/master
```

git reflog

`git reflog`

Zeigt vergangene HEAD Pointer an

- `git reflog` zeigt den sogenannten **Reference Log** an
- Dieser enthält ehemalige (lokal) vorhandene HEAD Positionen
- Der Reference Log enthält ebenfalls gecachte/referenzlose Commits und Stashes

git reflog

`git reflog`

Zeigt vergangene HEAD Pointer an

- `git reflog` zeigt den sogenannten **Reference Log** an
- Dieser enthält ehemalige (lokal) vorhandene HEAD Positionen
- Der Reference Log enthält ebenfalls gecachte/referenzlose Commits und Stashes

Damit können (*versehentlich*) „gelöschte“ Commits wiederhergestellt werden!

git stash (1/3)

`git stash`

Verbirgt vorrübergehend Änderungen

Häufiges Problem bei Arbeiten mit vielen Branches:

```
$ git checkout bugfix
error: Your local changes to the following files would be overwritten
    test0
Please commit your changes or stash them before you switch branches.
Aborting
```

Lösung: Temporäres Verbergen

```
$ git stash     # Stash changes
$ git stash pop # Remove and apply
```

git stash (2/3)

Stashes sind **stackbar**:

```
$ git stash                    # First stash
$ vim foo
[...]
$ git stash                    # Second stash
$ git stash list               # List stashes
stash@{0}: WIP on master: eb60234 Feature: add manatee
stash@{1}: WIP on master: eb60234 Feature: add manatee
$ git stash apply stash@{0} # Select first stash
```

git stash (3/3)

`git stash` erzeugt temporäre Commits auf eigenem Branch:

- Darstellung eines Stashes:

```
$ cat .git/refs/stash
7faa4ca073c65c6df81e0839264506e3d4f2ef13
```

- Inhalt eines Stashes:

```
$ git show 7faa4ca073c65c6df81e0839264506e3d4f2ef13
commit 7faa4ca073c65c6df81e0839264506e3d4f2ef13 (refs/stash)
Merge: c0d0e39 e12d139
Date:   Sat Mar 14 12:34:56 2020 +0200

    WIP on master: c0d0e39 0

[...]
```

git rebase

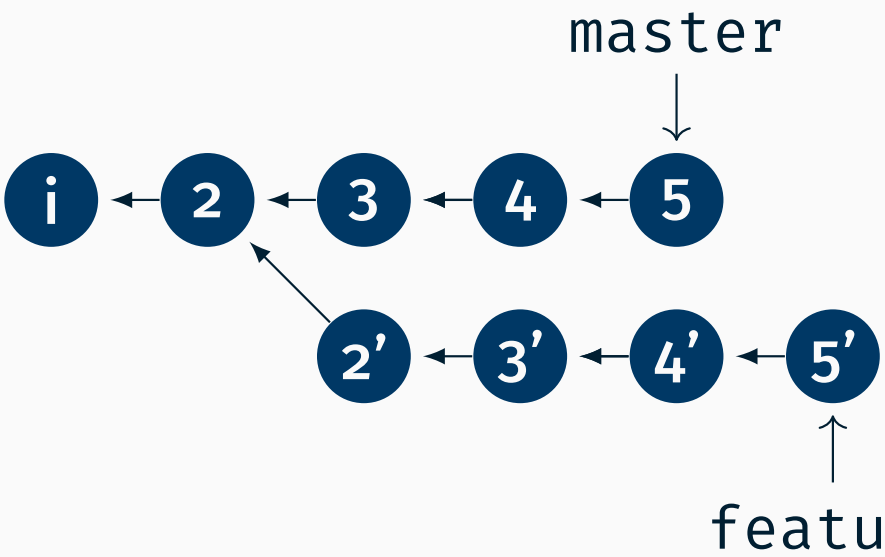
`git rebase`

Setzt unterschiedliche Branches aufeinander auf

git rebase

`git rebase`

Setzt unterschiedliche Branches aufeinander auf

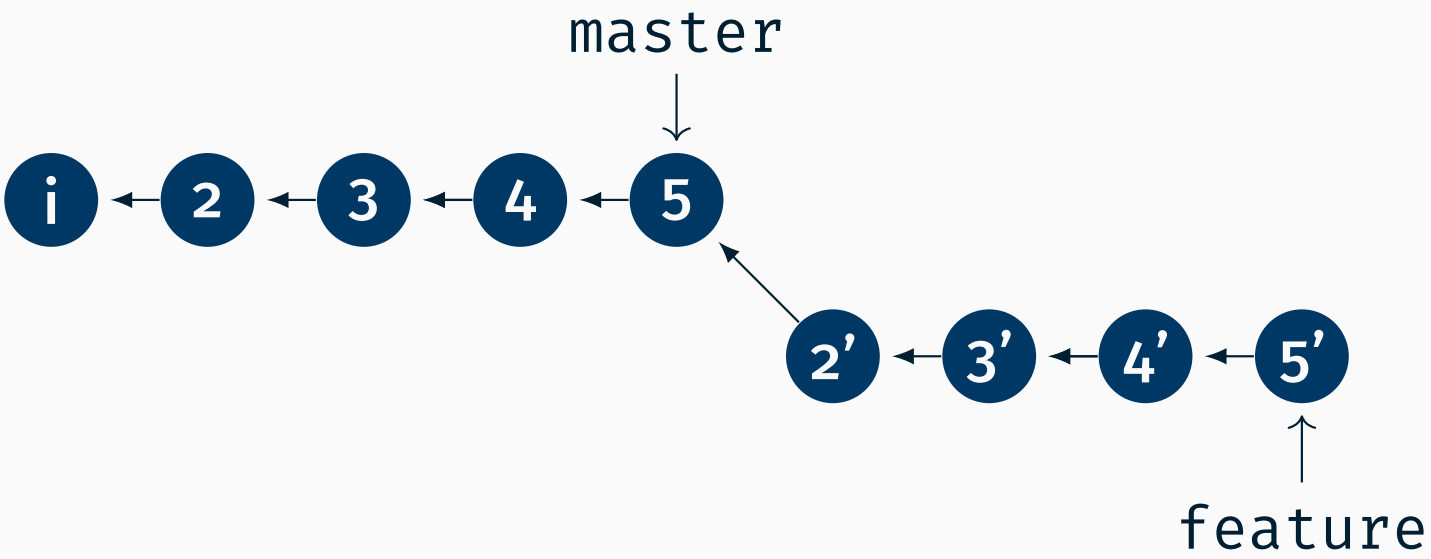



```
$ git checkout feature
$ git rebase master
```

git rebase

`git rebase`

Setzt unterschiedliche Branches aufeinander auf

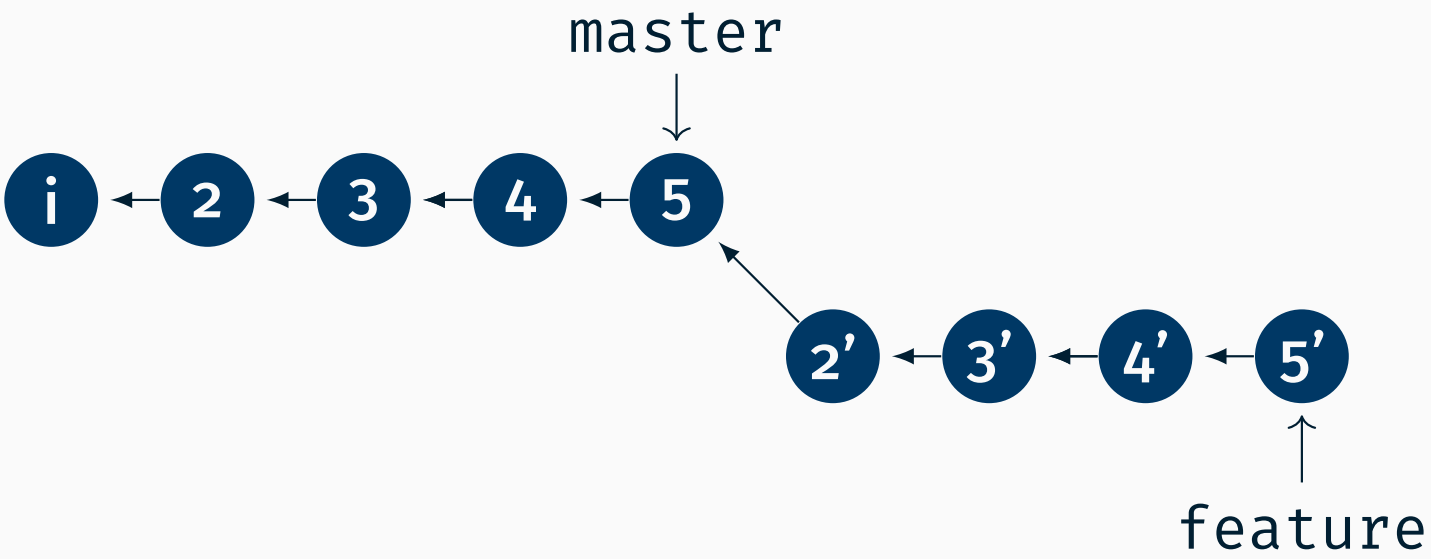



```
$ git checkout feature
$ git rebase master
```

git rebase

`git rebase`

Setzt unterschiedliche Branches aufeinander auf




```
$ git checkout feature
$ git rebase master
```

Vorsicht: Bei Konflikten ist *manuelles* Eingreifen notwendig!

git rebase -i (1/2)

```
git rebase -i
```

Ändert Git-Historie interaktiv

git rebase -i (1/2)

`git rebase -i`

Ändert Git-Historie interaktiv

Erlaubt die Versionsgeschichte „neu“ zu schreiben, also Commits

- umsortieren
- verschmelzen
- aufteilen
- nachträglich ändern
- …

git rebase -i (2/2)

```
$ git rebase --interactive HEAD~4

[vim]
pick 0b9bf3812ad1 afs: Split wait from afs_make_call()
pick a690f60a2ba3 afs: Calculate lock extend timer from...
pick 68ce801ffd82 afs: Fix AFS file locking to allow fine...

# Rebase 149e703cb8bf..a9fbcd672883 onto 149e703cb8bf (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# ...
```

git cherry-pick

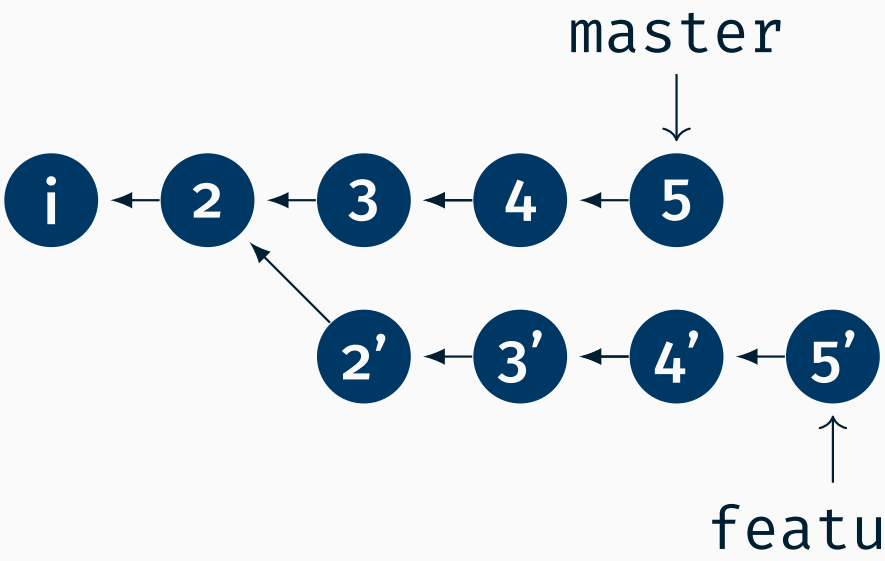
`git cherry-pick`

„Pflückt“ bestehende Commits aus der Git-Historie

git cherry-pick

`git cherry-pick`

„Pflückt“ bestehende Commits aus der Git-Historie

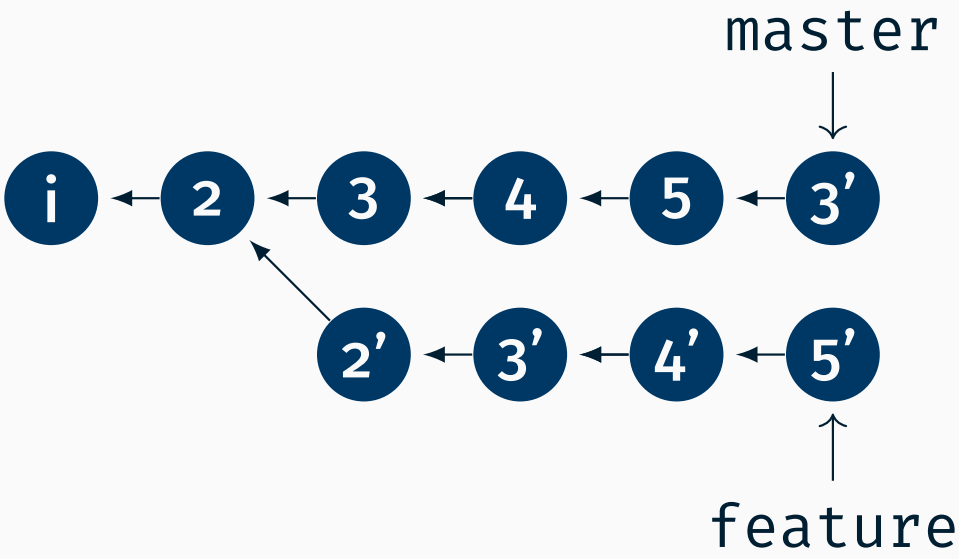



```
$ git checkout master
$ git cherry-pick 3’
```

git cherry-pick

`git cherry-pick`

„Pflückt“ bestehende Commits aus der Git-Historie




```
$ git checkout master
$ git cherry-pick 3’
```

git bisect

`git bisect`

Sucht nach fehlerhaften Revisionen

Voraussetzungen:

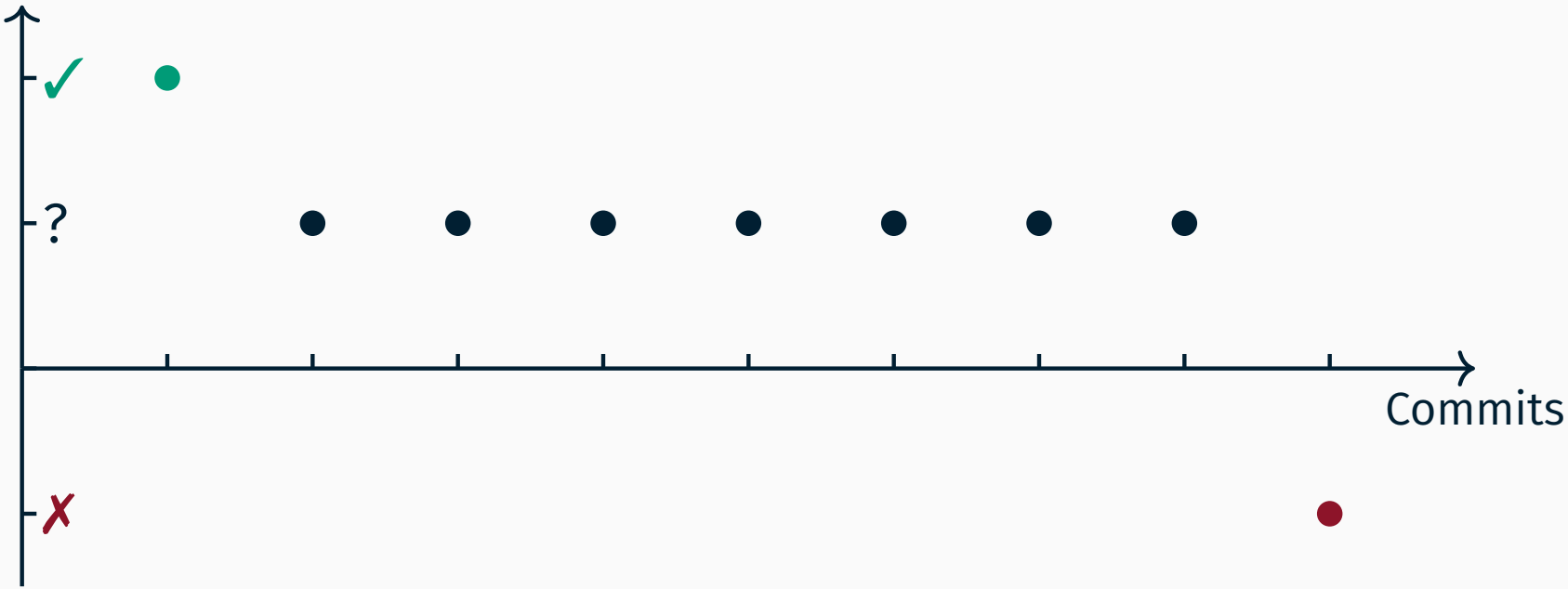
- Eine Revision, in der der Fehler auftritt
- Eine Revision, in der der Fehler (noch nicht) auftritt
- Manueller oder automatischer Test für den Fehler
- Möglichst viele testbare (d.h. übersetzbare) Revisionen

Bisektionsalgorithmus

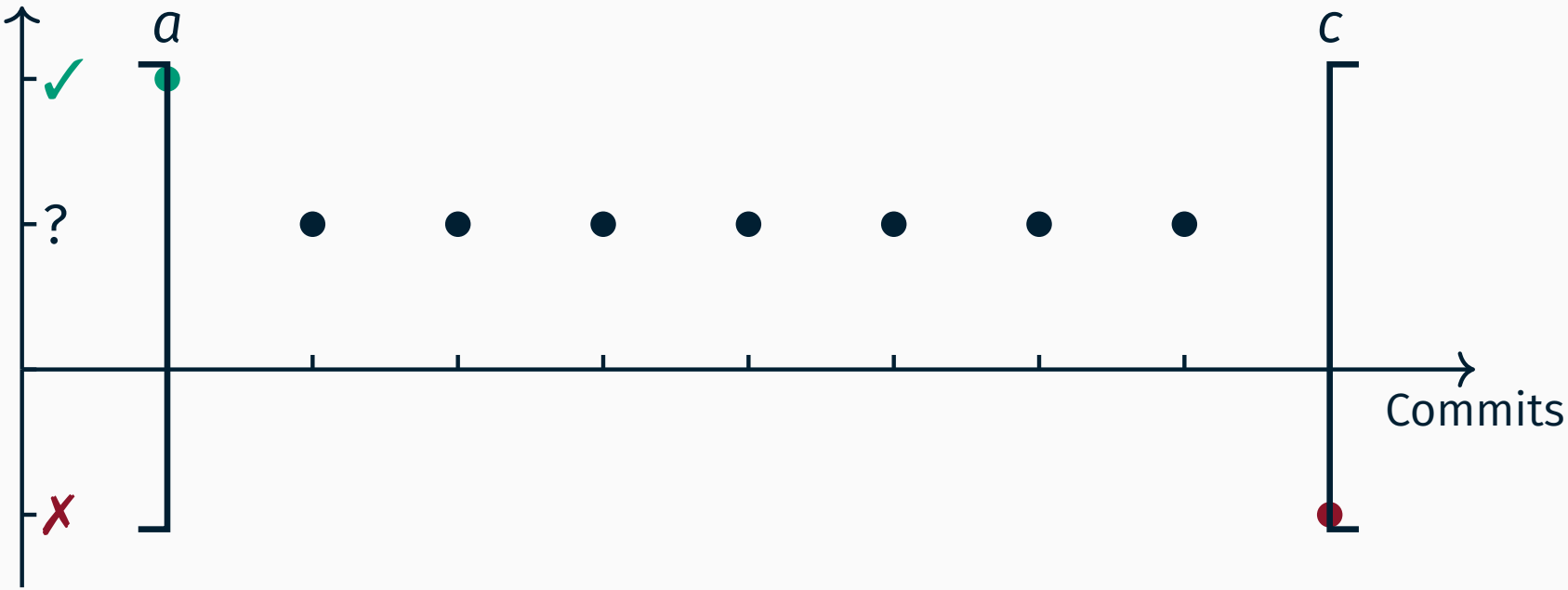
Verfahren (beinahe binäre Suche)

1. wähle Intervall $]a; c[$, so dass Fehler in c, kein Fehler in a
2. wähle b „in der Mitte“ zwischen a und c
3. prüfe b auf Fehler
   - Fehler: Wiederhole mit mit Intervall $]a; b[$
   - kein Fehler: Wiederhole mit mit Intervall $]b; c[$
4. fertig, wenn Intervall leer

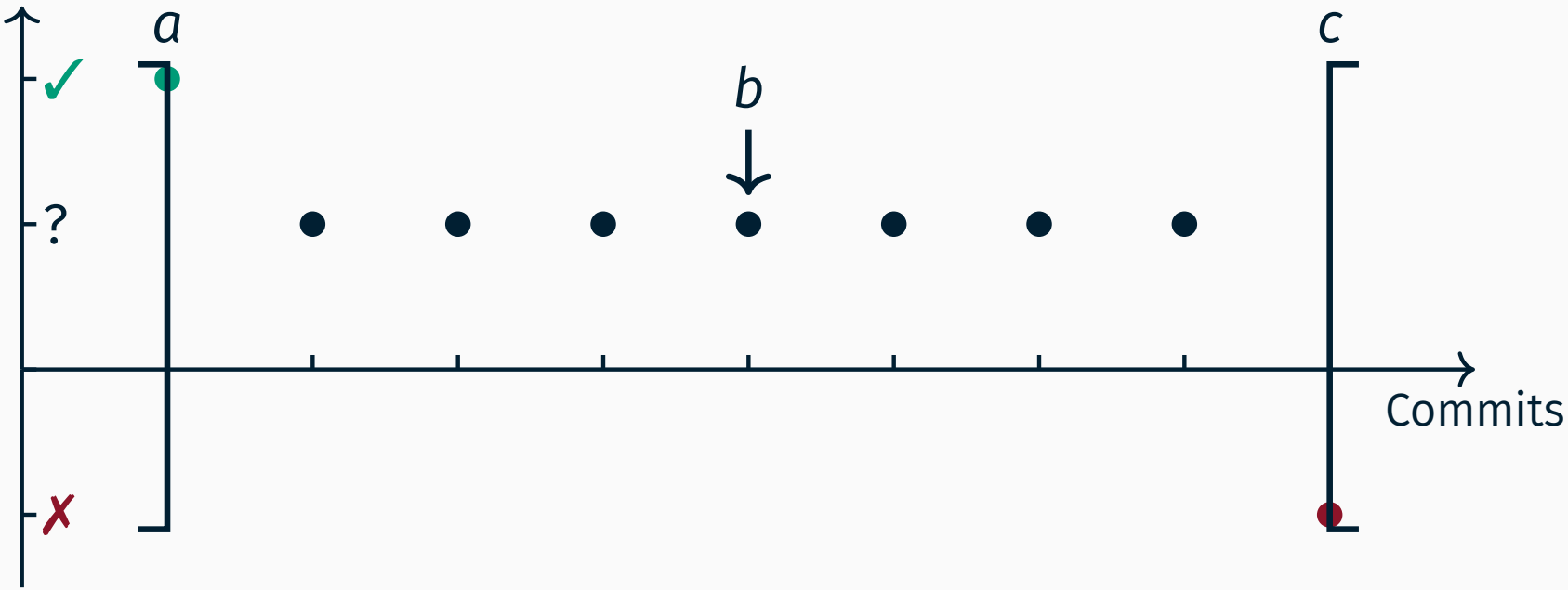
Bisektionsalgorithmus am Beispiel



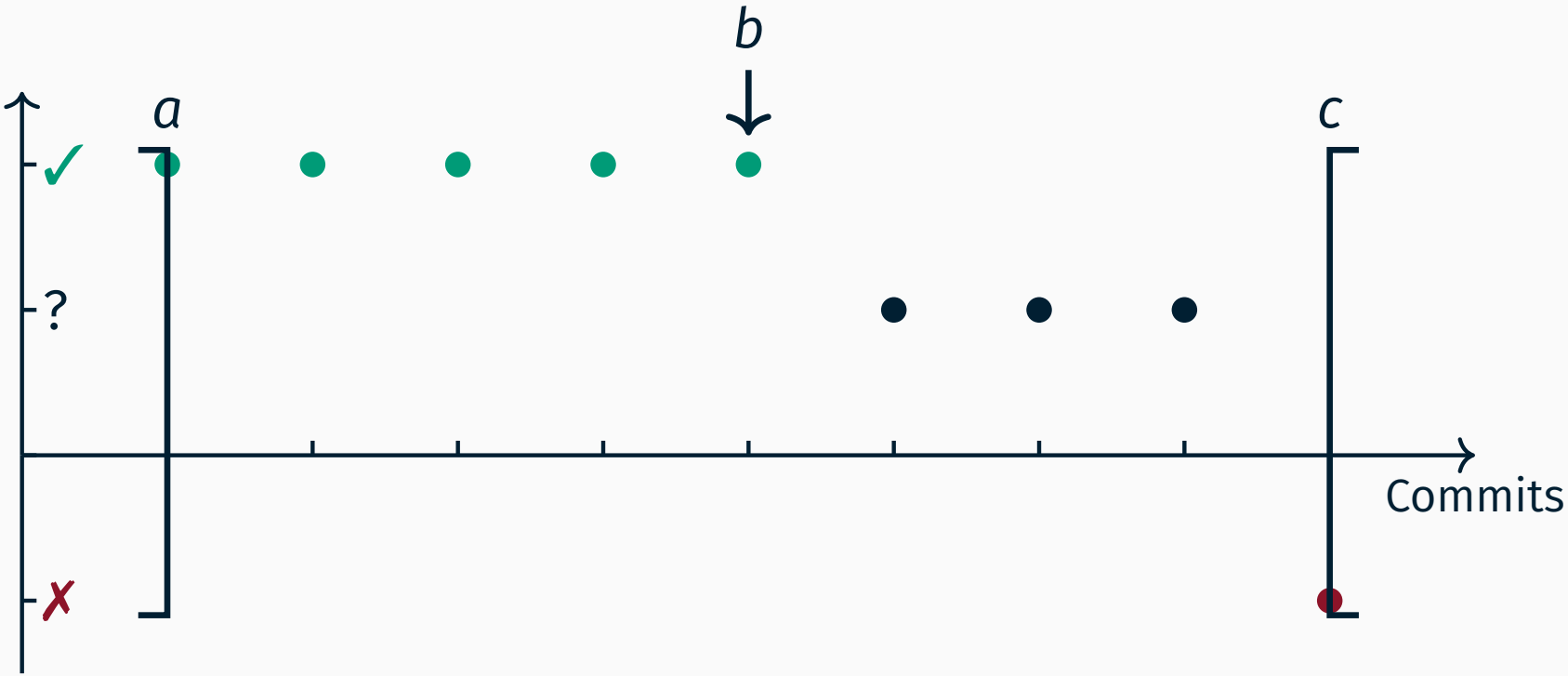
Bisektionsalgorithmus am Beispiel



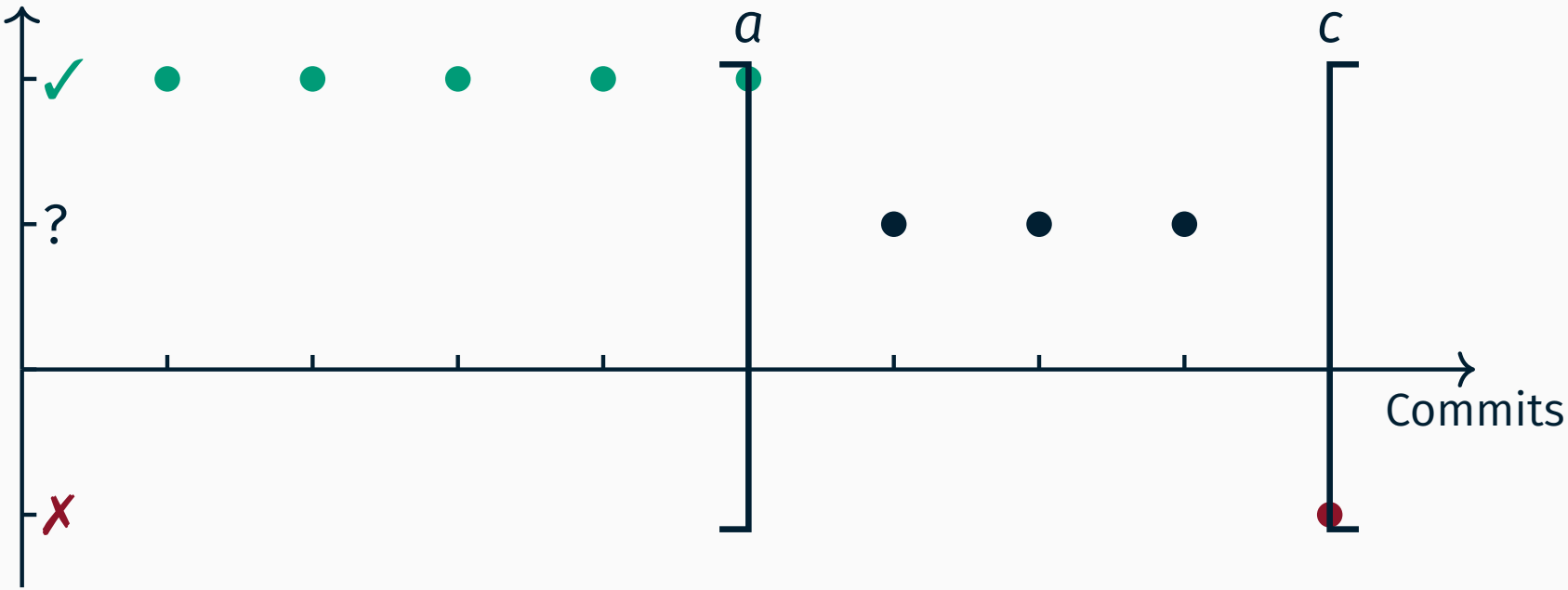
Bisektionsalgorithmus am Beispiel



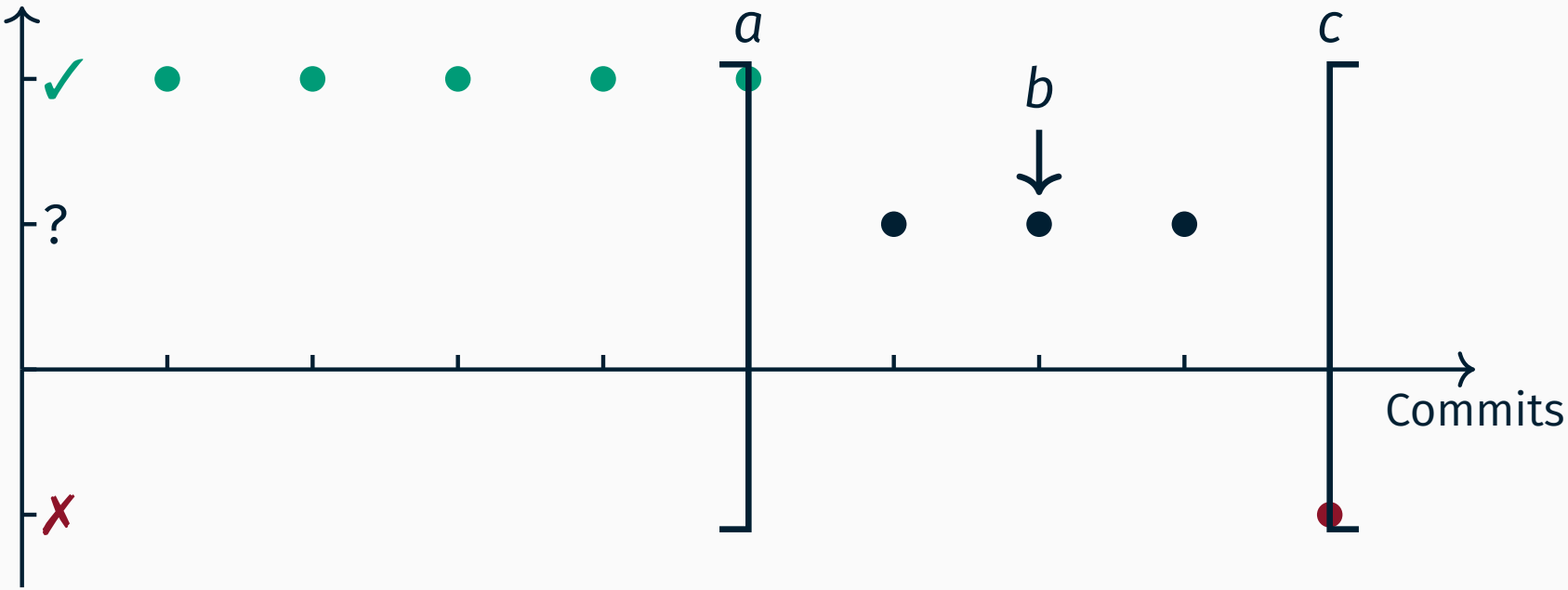
Bisektionsalgorithmus am Beispiel



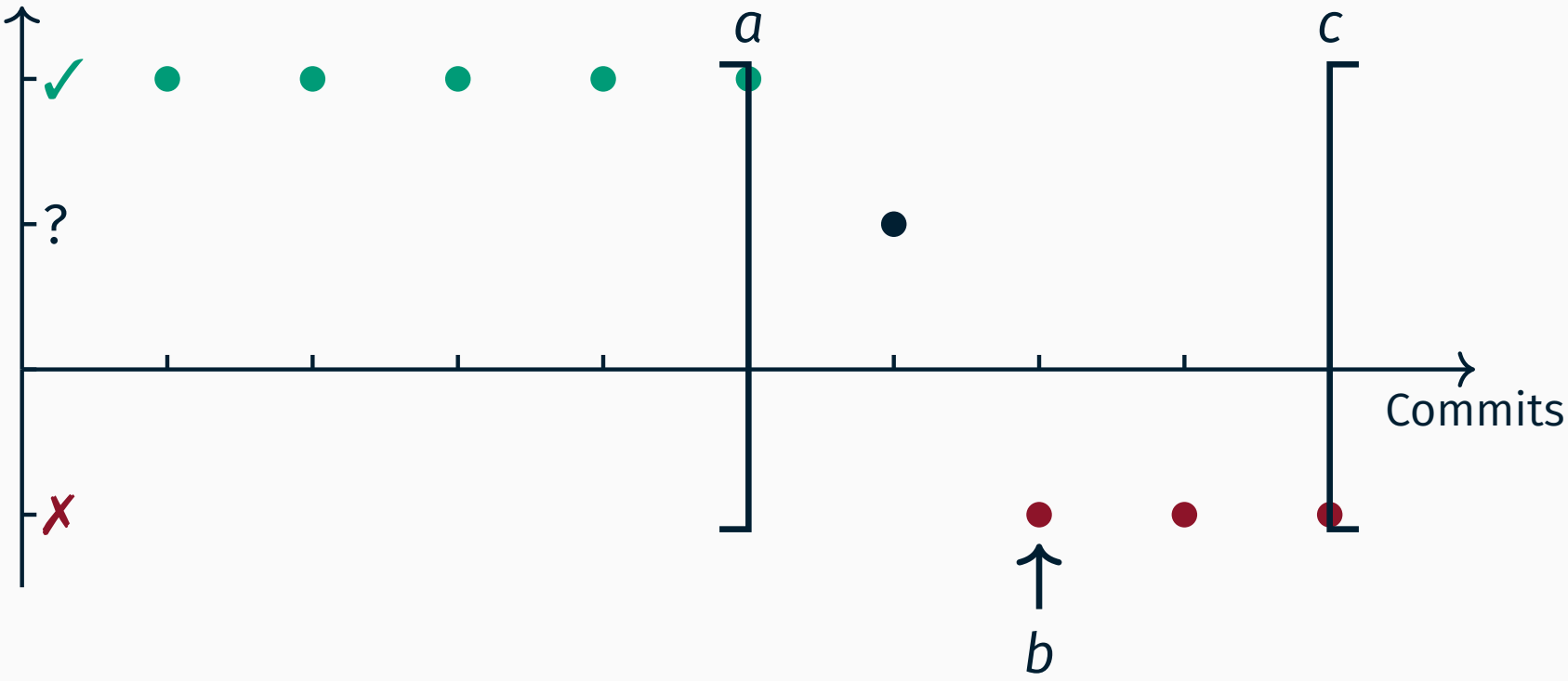
Bisektionsalgorithmus am Beispiel



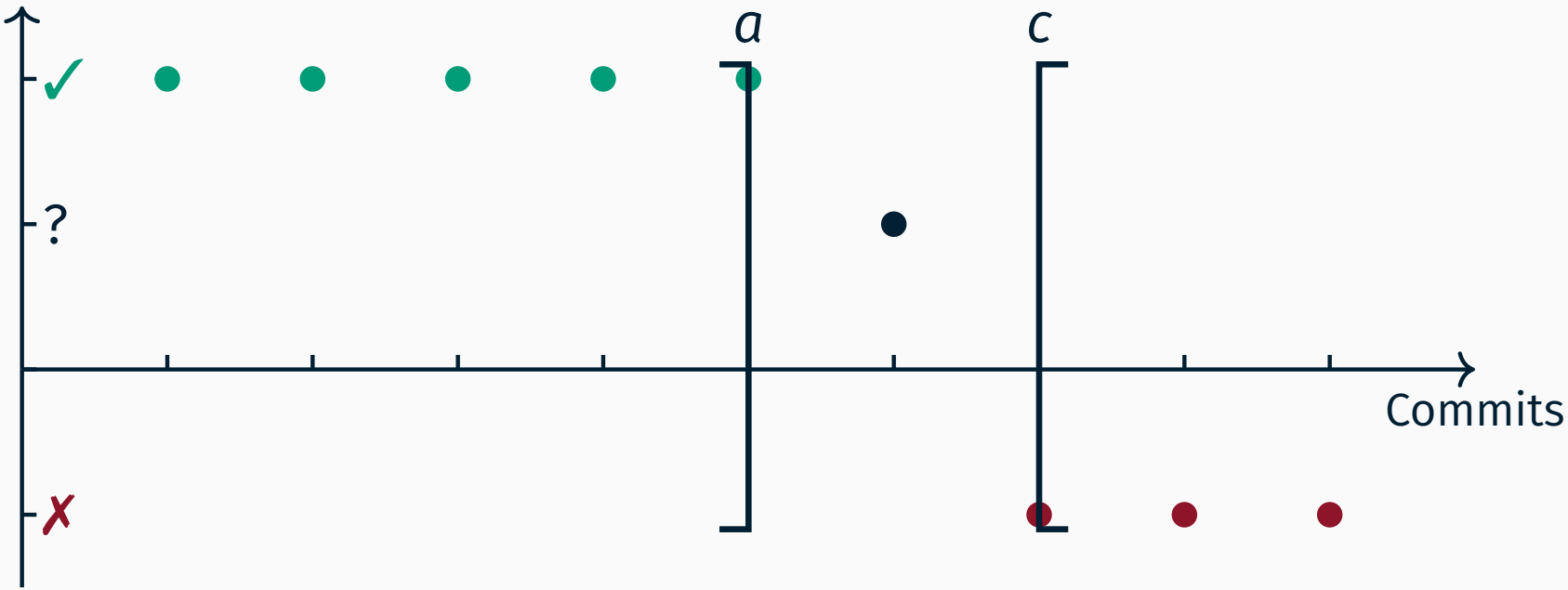
Bisektionsalgorithmus am Beispiel



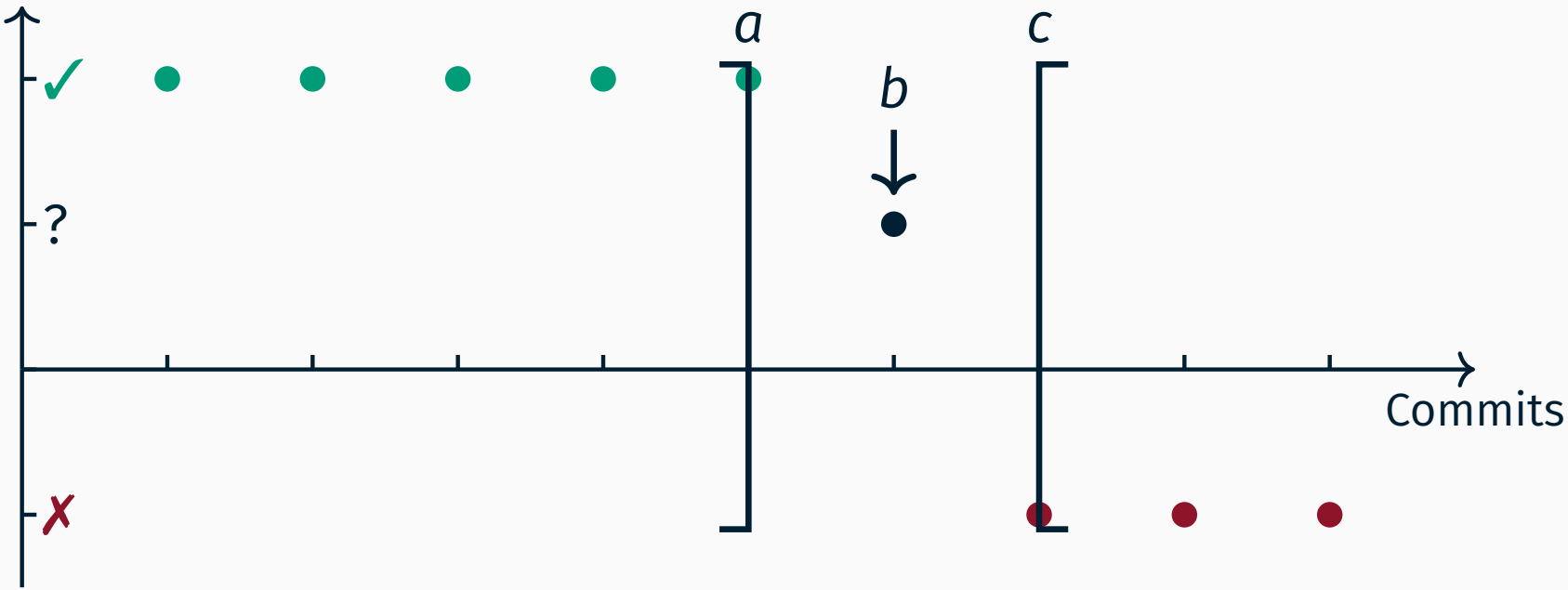
Bisektionsalgorithmus am Beispiel



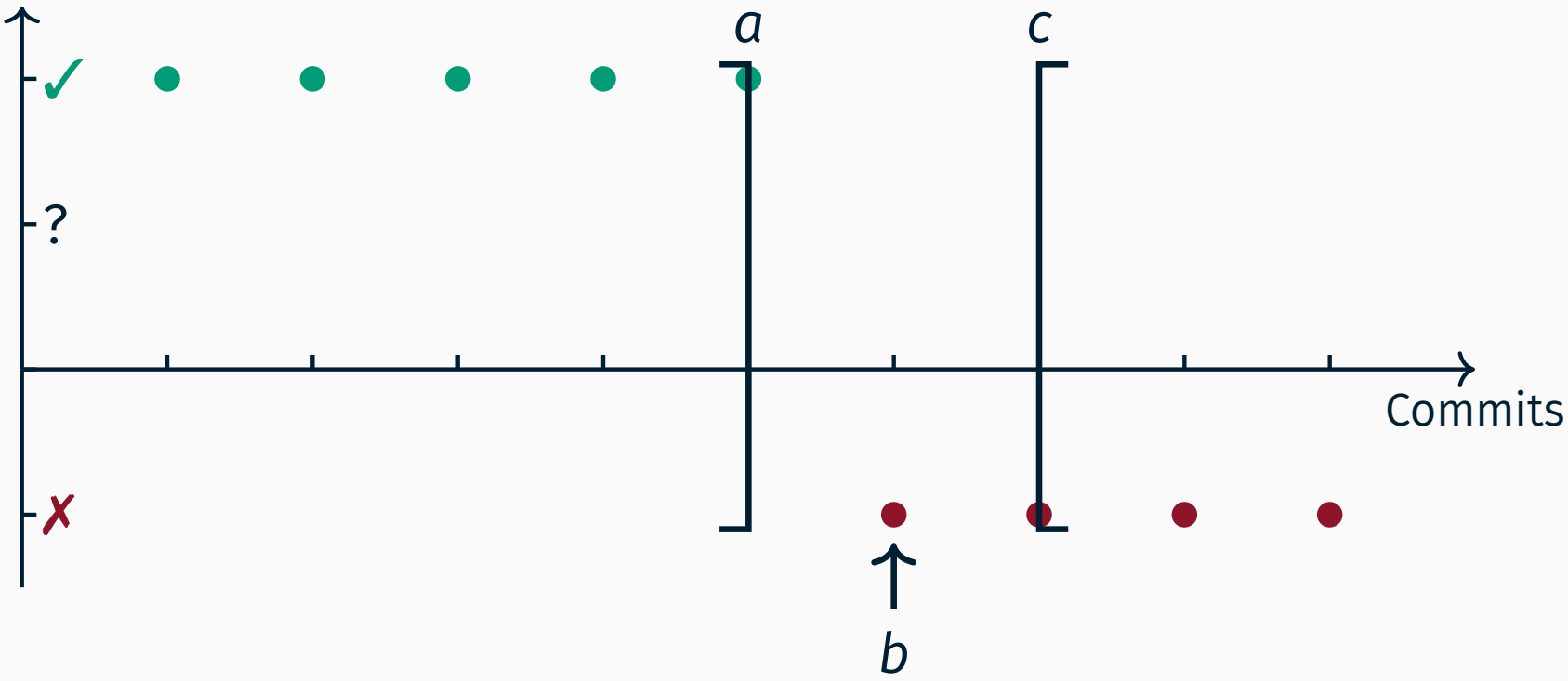
Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus am Beispiel



Bisektionsalgorithmus in der Praxis

Verfahren mit Git

1. Bisektion starten & fehlerhaften (bad) / guten (good) Commit markieren

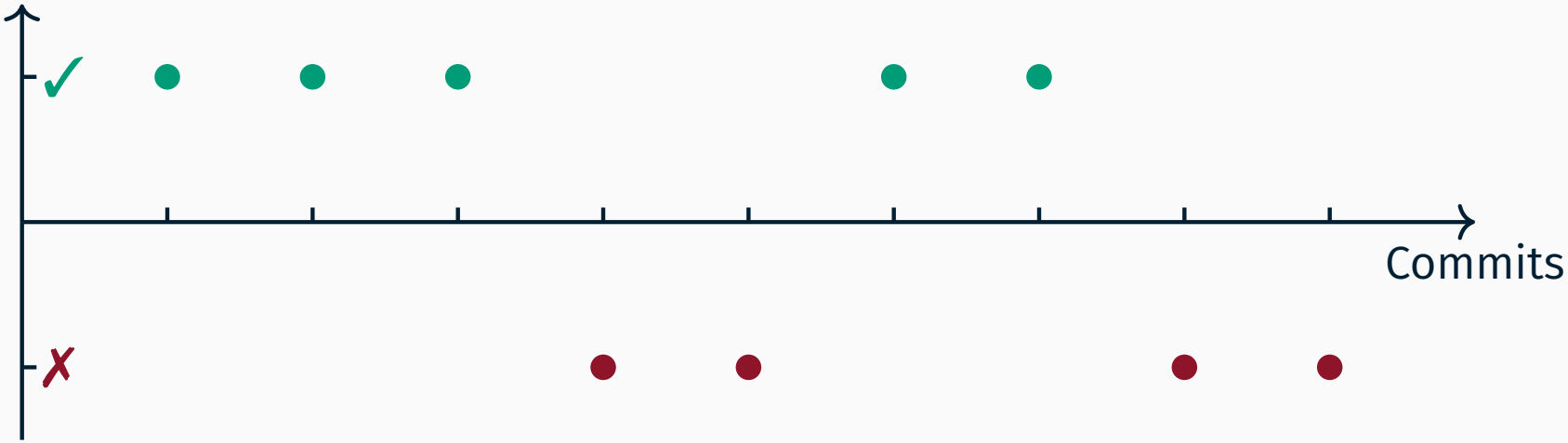
```
$ git bisect start
$ git bisect bad
$ git bisect good HEAD~5
```

2. Testen auf Fehler
   - Bisektionsschritte werden automatisch ausgecheckt
   - Markieren mit `git bisect [good|bad|skip]`
3. Git zeigt an wenn Fehlerhafter Commit identifiziert

- Visualisierung des aktuellen Stands

```
$ git bisect log
$ git bisect visualize
```

Bisektionsalgorithmus in der Praxis



Die Commithistory kann mehrere Übergänge enthalten!

spart Zeit…

- Binäre Suche nach Fehler auf Intervall von n Revsionen
- Jeder Schritt halbiert Intervall (außer bei „skip“)
- Bester Fall: $\lceil \log_2 n \rceil$
  - ~10 Schritte für 1000 Revisionen
  - ~20 Schritte für 1 Mio. Revisionen
- Schlechtester Fall: $n - 1$ Schritte
  - Nur wenn keine Revision baut/testbar

Moral

Nur übersetzbaren Code in den Hauptentwicklungszweig!

spart noch mehr Zeit...

- Wenn bekannt ist, dass Fehler nur Teilmodul betrifft
  - können Commits zu anderen Modulen ignoriert werden
  - spart zusätzlich Bisektionsschritte

$\Rightarrow$ Nur Teilbäume betrachten

```
$ git bisect start -- src/subsystem/subsubsystem
```

Moral

Nur logisch zusammenhängende Änderungen in Commits!

und noch mehr Zeit...

- Nonplusultra:
  - $+$ automatischen Test
  - $+$ funktionierendes Build-Skript
  - $+$ kleines Skript das beides aufruft
  - $=$ automagisches `git bisect`

```
$ git bisect run ./test.sh
```

Moral

automatische & schnelle Testsuites und schnelle Build-Skripte sind toll

siehe gitlab.cs.fau.de/i4/git-bisect-demo.git

Zusammenfassung

Zusammenfassung

- Überblick über interne Funktionsweise von **Git**
  - Zusammenspiel von **Git** Objects
  - Umsetzung von Branches, Commit-Historie, ...
- Fortgeschrittene **Git-Konzepte**
  - Verwendung von ausgewählte Befehle

Git++

git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen

git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen

git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen


```
$ git add bugfix
```

git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen


```
$ git add bugfix
$ git commit --fixup 4
```

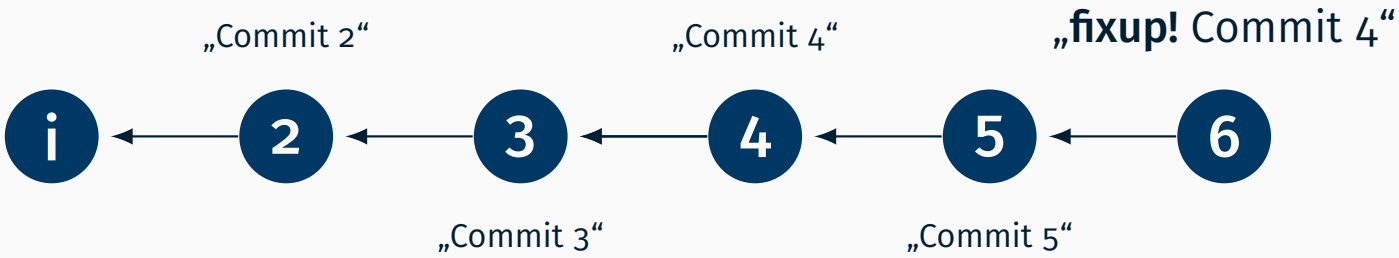
git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen




```
$ git add bugfix
$ git commit --fixup 4
```

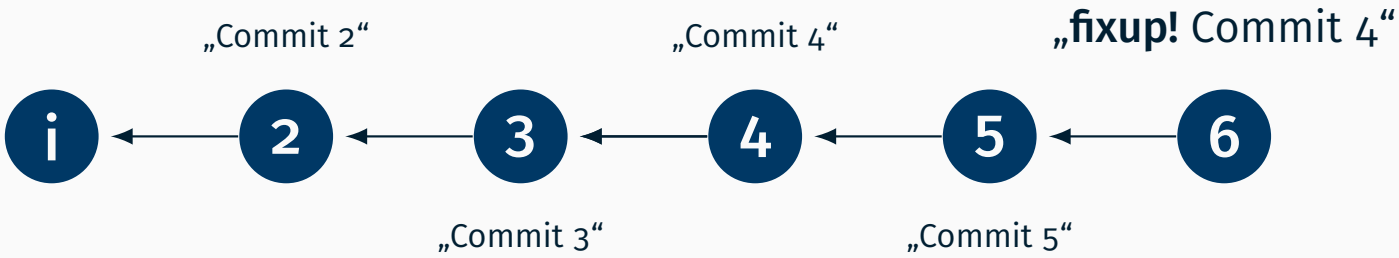
git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen




```
$ git add bugfix
$ git commit --fixup 4
$ git rebase -i --autosquash 3
```

git commit + git rebase

`git commit --fixup`

markiert Commit als Verbesserung/Korrektur

`git rebase -i --autosquash`

Führt markierte Commits zusammen


```
$ git add bugfix
$ git commit --fixup 4
$ git rebase -i --autosquash 3
```

git clean

`git clean`

Löscht nicht-versionierte Dateien

git clean

`git clean`

Löscht nicht-versionierte Dateien

- Häufig entstehen temporäre Dateien beim Kompilieren:

```
$ git status
Untracked files:
roms/vgabios/
storage-daemon/qapi/qapi-commands.c
storage-daemon/qapi/qapi-commands.h
virtiofsd
```

- Mittels `git clean` lassen sich diese schnell entfernen

git tag

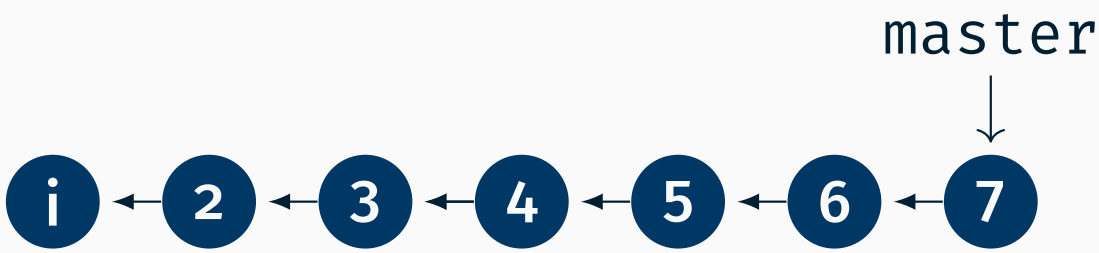
`git tag`

Erstellt, löscht und listet Tags auf

git tag

`git tag`

Erstellt, löscht und listet Tags auf

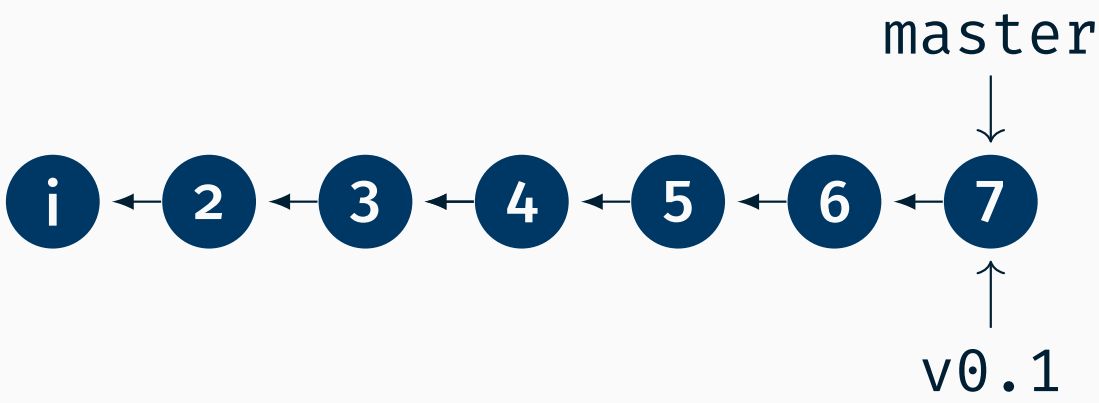



```
$ git tag "v0.1"
```

git tag

`git tag`

Erstellt, löscht und listet Tags auf

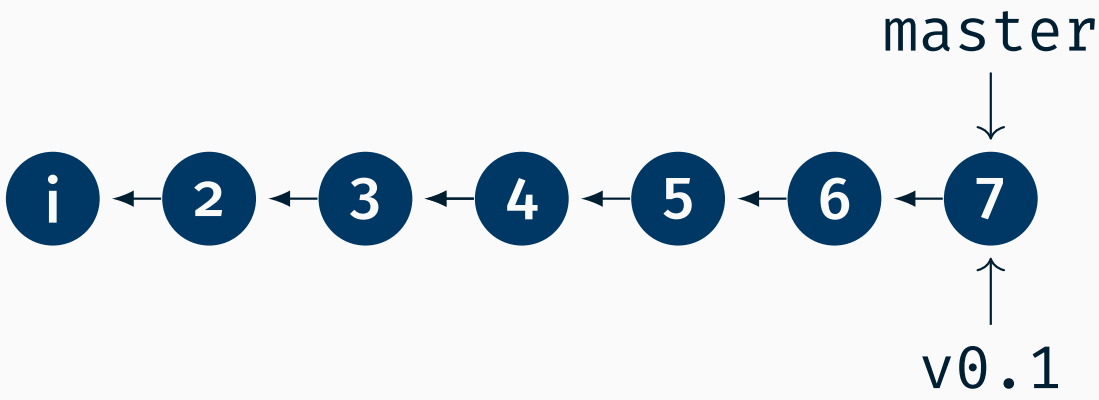



```
$ git tag "v0.1"
```

git tag

`git tag`

Erstellt, löscht und listet Tags auf




```
$ git tag "v0.1"
```

- Linux-Kernel benutzt Tags als Versionmarkierung (z.B. `v5.9`)
- Tags können wie Branches ausgecheckt werden

git gc

`git gc`

Startet Garbage Collection von **Git**

- Vielzahl von **Git** Befehlen (`rebase`, `reset`, ...) erzeugen referenzlose Commits
- **Git** stellt dafür Garbage Collecetion bereit
  - Kompression von Versionsständen
  - Löschen von referenzlosen Git Objects
- Bestimmte **Git** Befehle stoßen Garbage Collecetion automatisch an
- `git gc` startet manuelle Garbage Collecetion

Fragen?