Grundlagen der Versionskontrolle

mit Git

Inhalte

1. Ziele
2. Datensicherung
3. Git
    - Web UI
    - Grundlegende Befehle
    - Lifecycle
    - Branching
    - Paralleles Arbeiten

Am Ende dieses Moduls ...

... kennen Sie die Vorteile von Versionskontrollsystemen
... haben Sie Git eingerichtet
... können Sie Git Projekte anlegen
... können Sie Konflikte auflösen

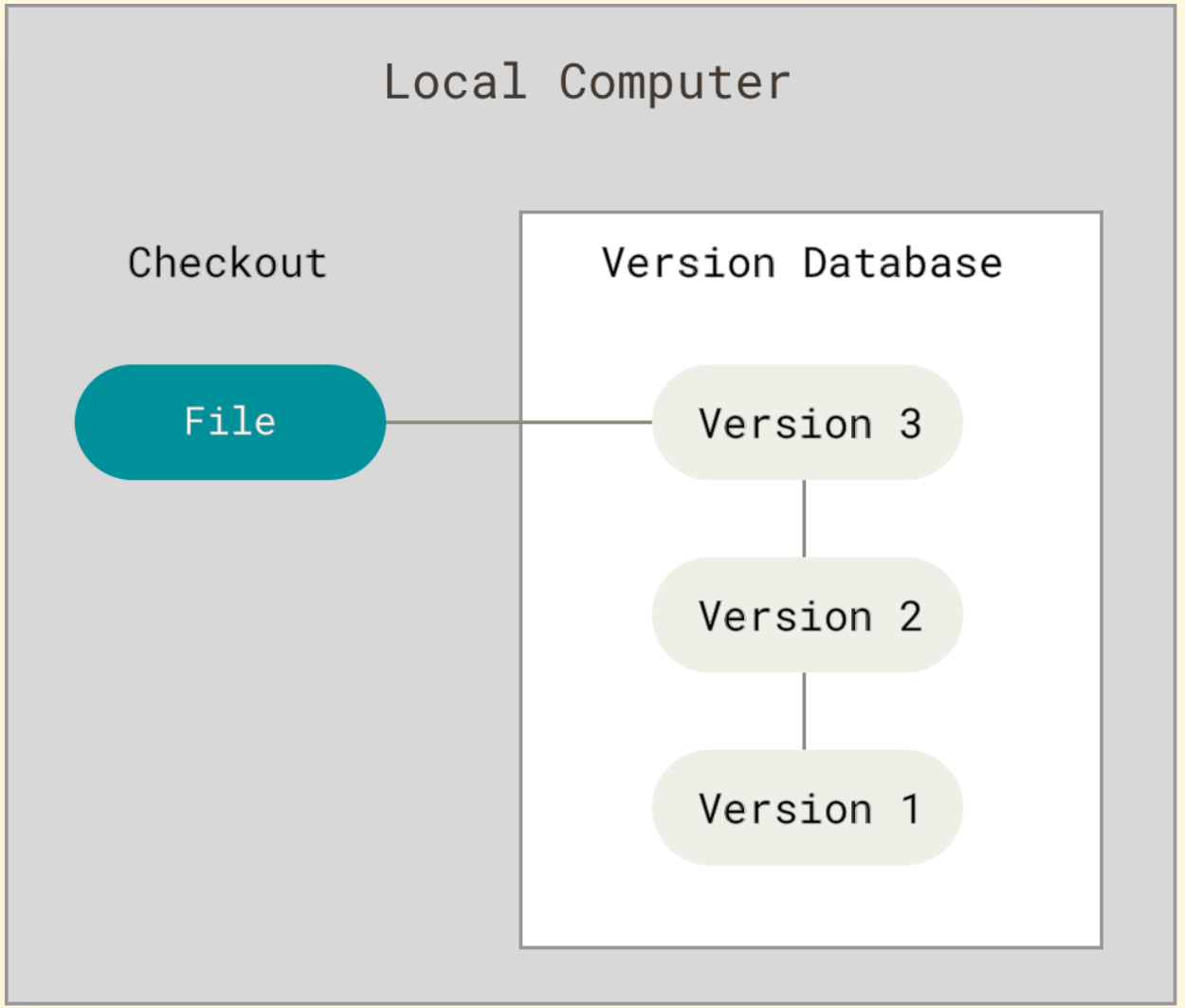
Warum ist Versionskontrolle wichtig?

Ziele der Versionskontrolle

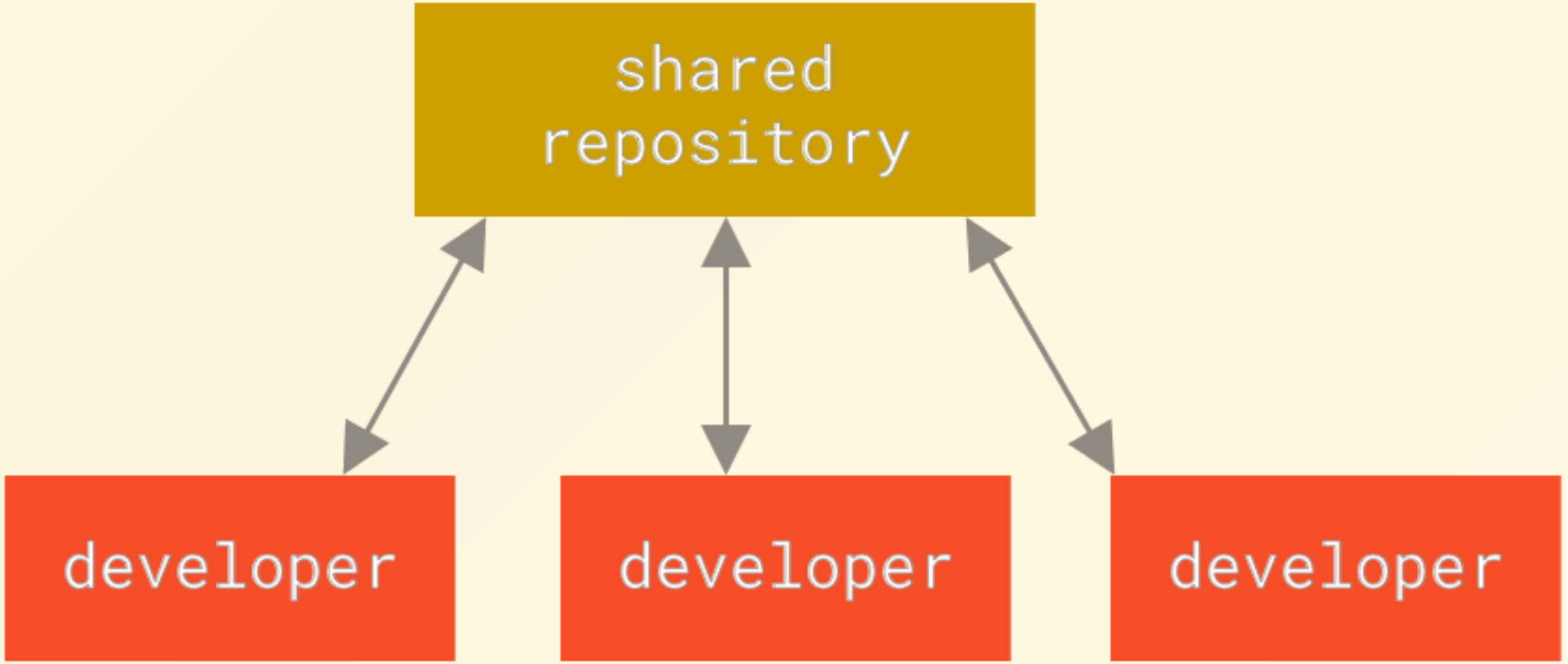
1. Eindeutigkeit bzgl. der aktuellen Version
2. Austausch von Daten
3. Datensicherung
4. Nachvollziehbarkeit
5. Wiederherstellung eines Softwarestandes

Verteilte Versionskontrolle

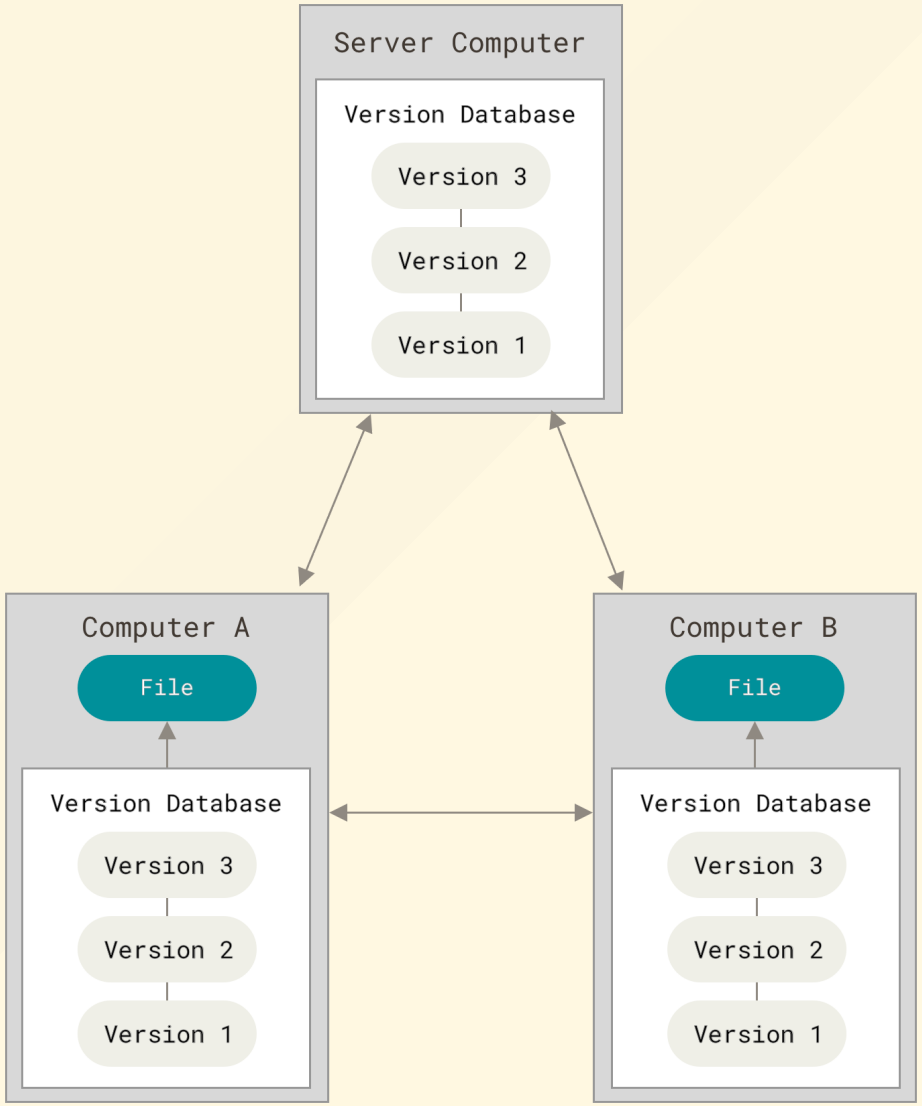
Lokale Versionskontrolle



Zentrale Versionskontrolle



Verteilte Versionskontrolle



Git

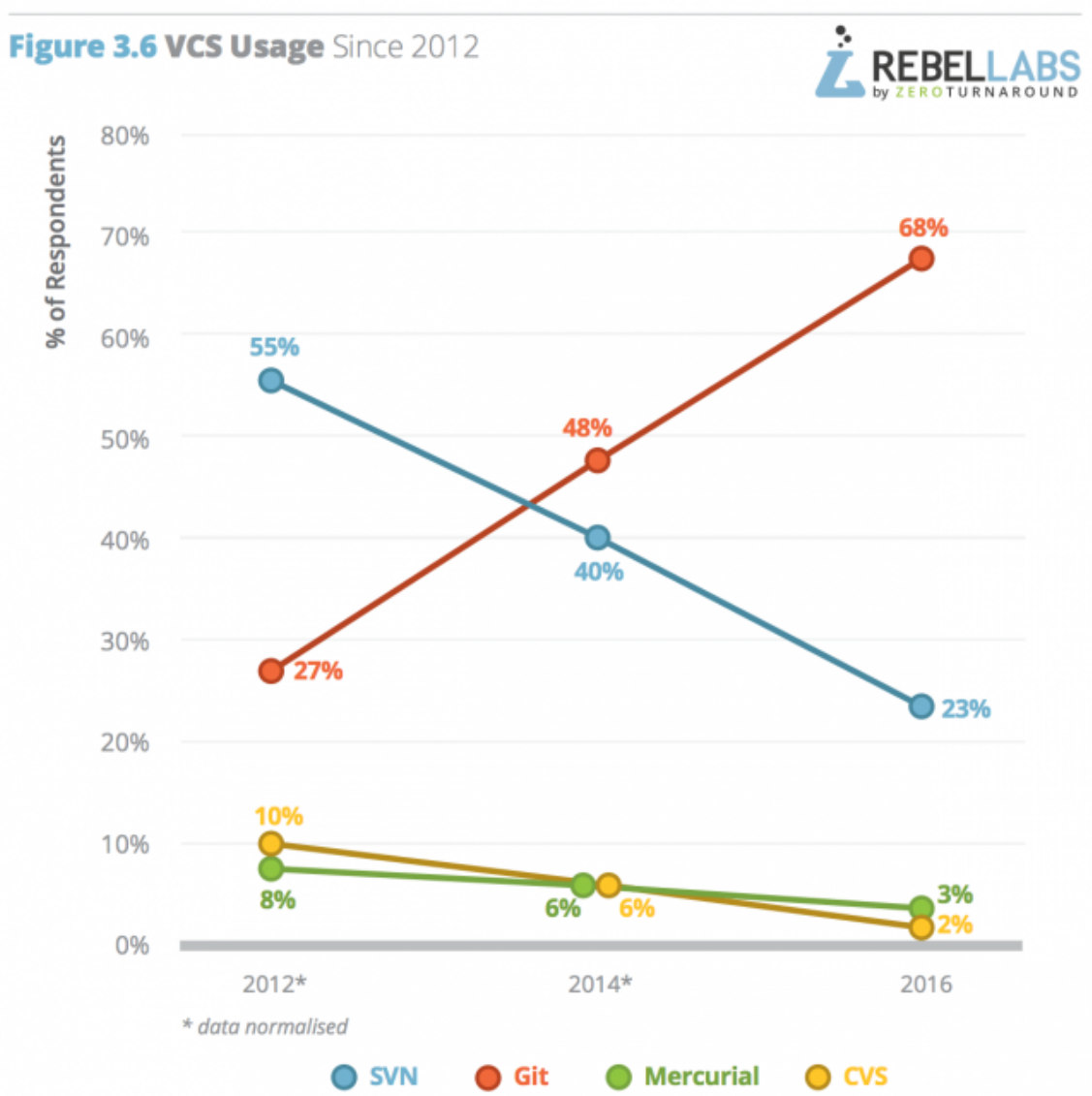
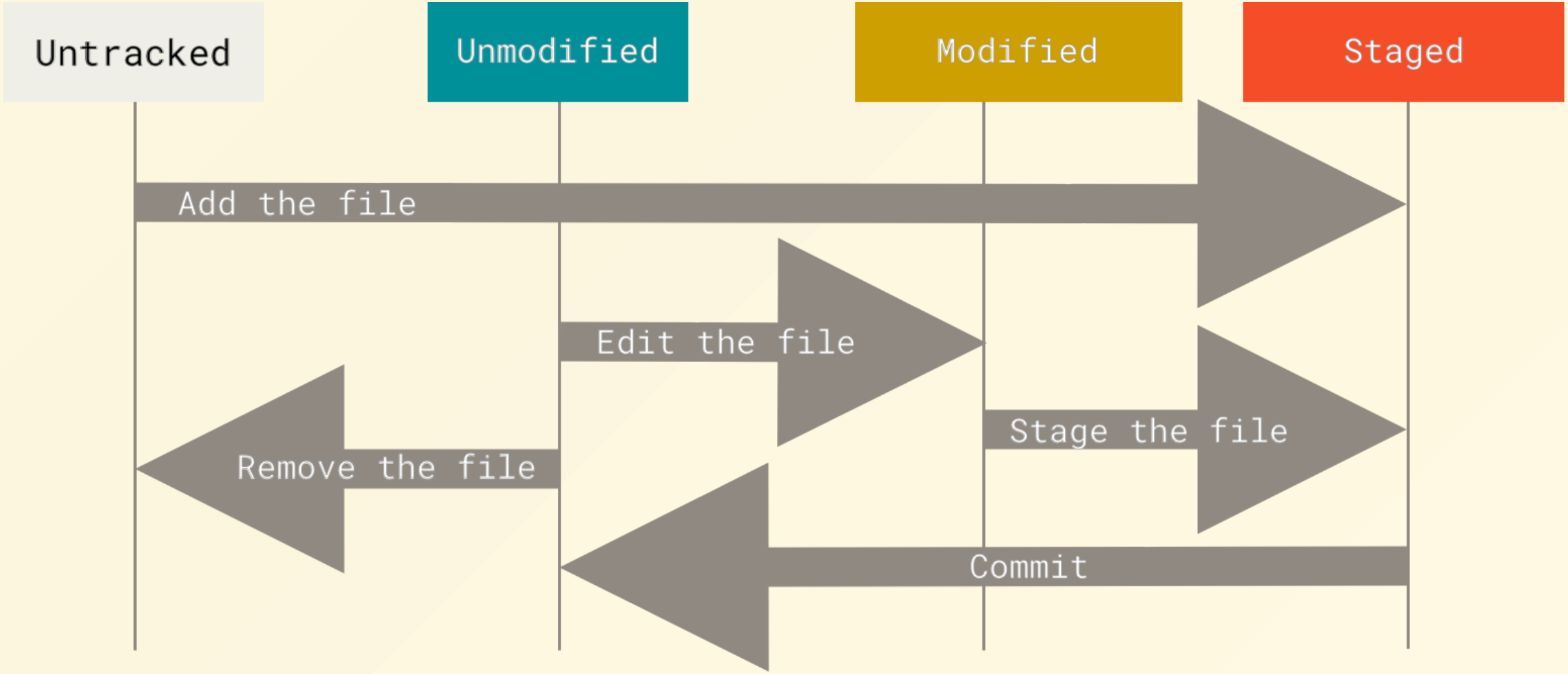


Figure 3.6 VCS Usage Since 2012
REBELLABS by ZEROTURNAROUND
% of Respondents
80%
70%
60%
50%
40%
30%
20%
10%
0%
55%
40%
23%
68%
48%
27%
10%
8%
6%
6%
3%
2%
2012*
2014*
2016
* data normalised
SVN
Git
Mercurial
CVS

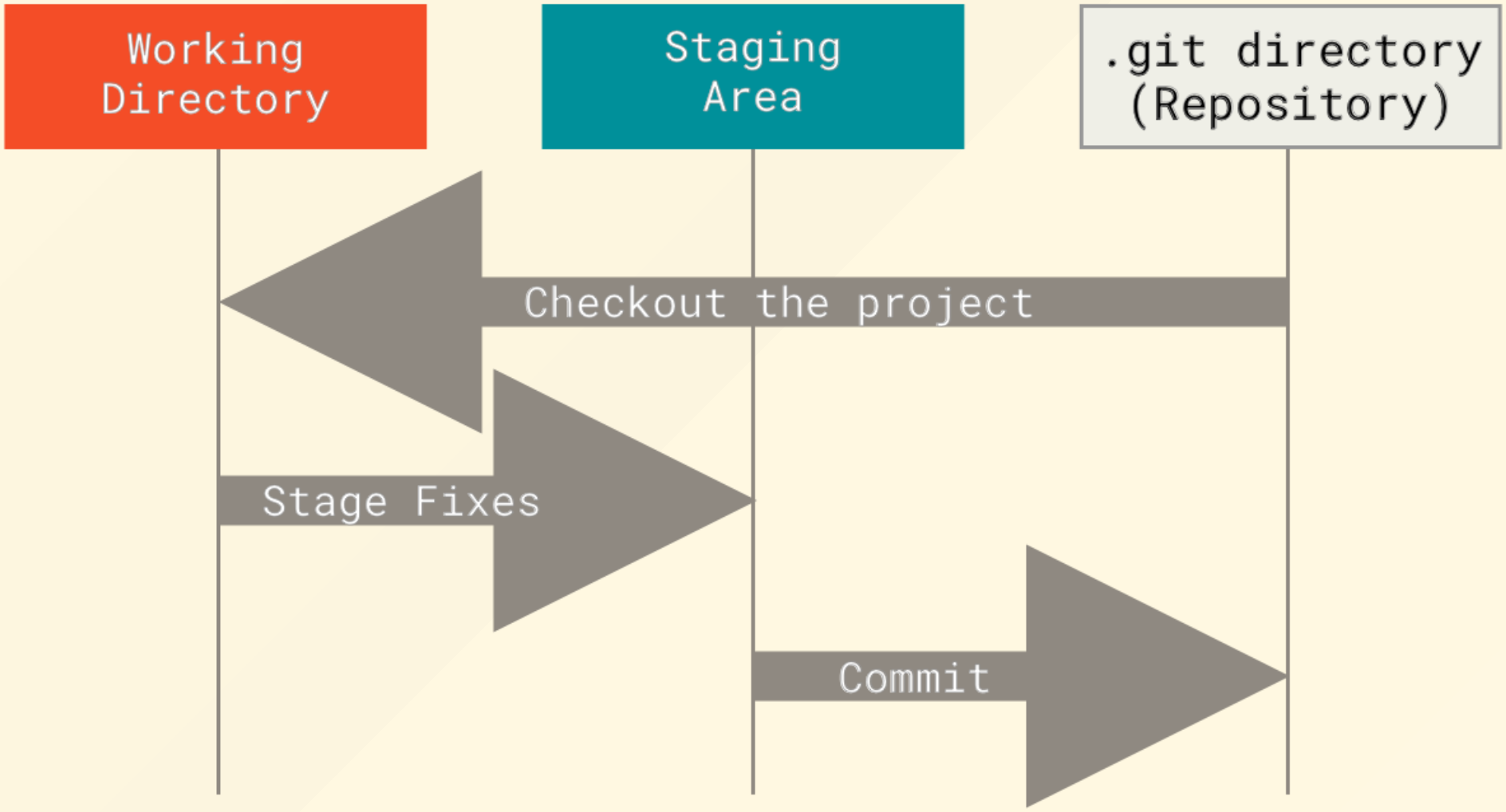
Web UIs

Produkt	Hersteller	Features
Gerrit	Google	Review System, Changesets, Outdated
GitHub	Microsoft	Marktführer, Issues, Wiki, Editor, Pull Requests
GitLab	Open-Source / GitLab Inc.	CI, Container Registry, Issues, WebIDE, Wiki, Merge Requests

Lifecycle einer Datei



Git Arbeitsbereiche



Git Befehle

`git --help`

`--help` am Ende eines Befehls zeigt die Hilfe an. Unter anderem werden die möglichen Parameter aufgelistet und erläutert.

`git config`

Ermöglicht die Konfiguration von Git, dabei gibt es:

- Lokale Einstellungen (je Repository)
- Globale Einstellungen

```
git config --global user.name "Mona Lisa"
git config --global user.email "mona.lisa@example.com"
```

```
git init
```

Initialisiert ein neues lokales Repository im angegebenen Ordner.

`git status`

Zeigt den Status des lokalen Repositories an:

- Aktueller Branch
- Status veränderter Dateien
- Inhalt des Staging Bereichs

```
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   Readme.md
```

`git add`

Fügt eine Datei dem Staging Bereich des Versionskontrollsystems hinzu.

```
git add Readme.md
```

`git commit`

Sichert den aktuellen Stand des Staging Bereichs ins Repository. Dabei wird für diesen eine eindeutige ID generiert. Der Staging Bereich wird geleert.

```
git commit -m "My first commit"
[master (root-commit) 562fbb8] My first commit
 1 file changed, 2 insertions(+)
 create mode 100644 Readme.md
```

`git remote`

Ermöglicht den Umgang mit Remote Repositories, indem es einen Alias hinzufügt. Dadurch kann leicht auf diesen verwiesen werden.

`git remote add <alias> <URL>`

```
git remote add origin https://github.com/<Nutzer>/myfirstrepo.git
```

`git push`

Aktualisiert das angegebene Remote Repository.

```
git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 279 bytes | 139.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/<Nutzer>/myfirstrepo/pull/new/master
remote:
To github.com:<Nutzer>/myfirstrepo.git
 * [new branch]      master -> master
```

Demo

Basics: Shell Commands

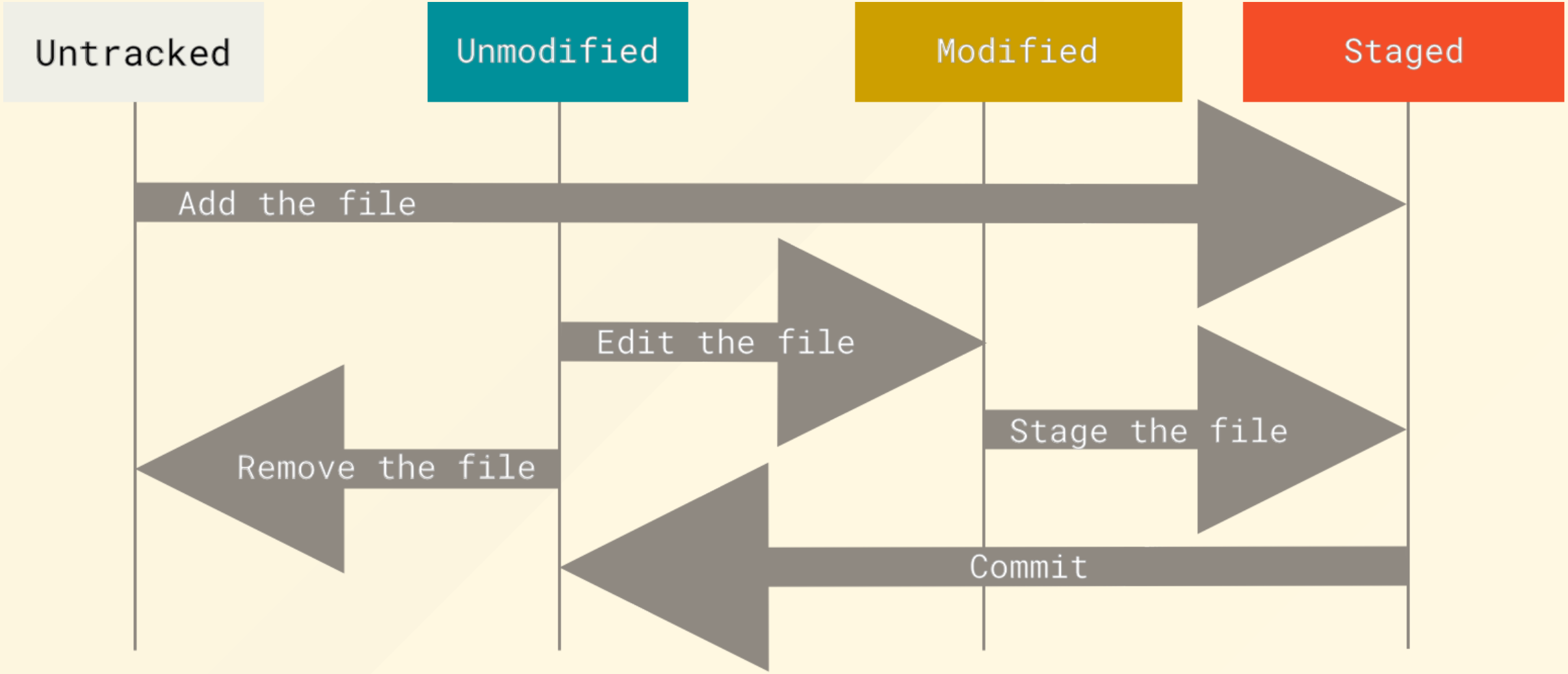
- `mkdir <folder>`
  Legt einen neuen Ordner an.
- `cd <folder>`
  Navigiert in das angegeben Verzeichnis.
  Sonderfall: "`cd ..`" verlässt den Unterordner.
- `ls -al`
  Listet alle Dateien und Ordner im aktuellen Verzechnis.
- `cat <file>`
  Gibt den Inhalt der Datei auf die Konsole aus.

Basics: VIM

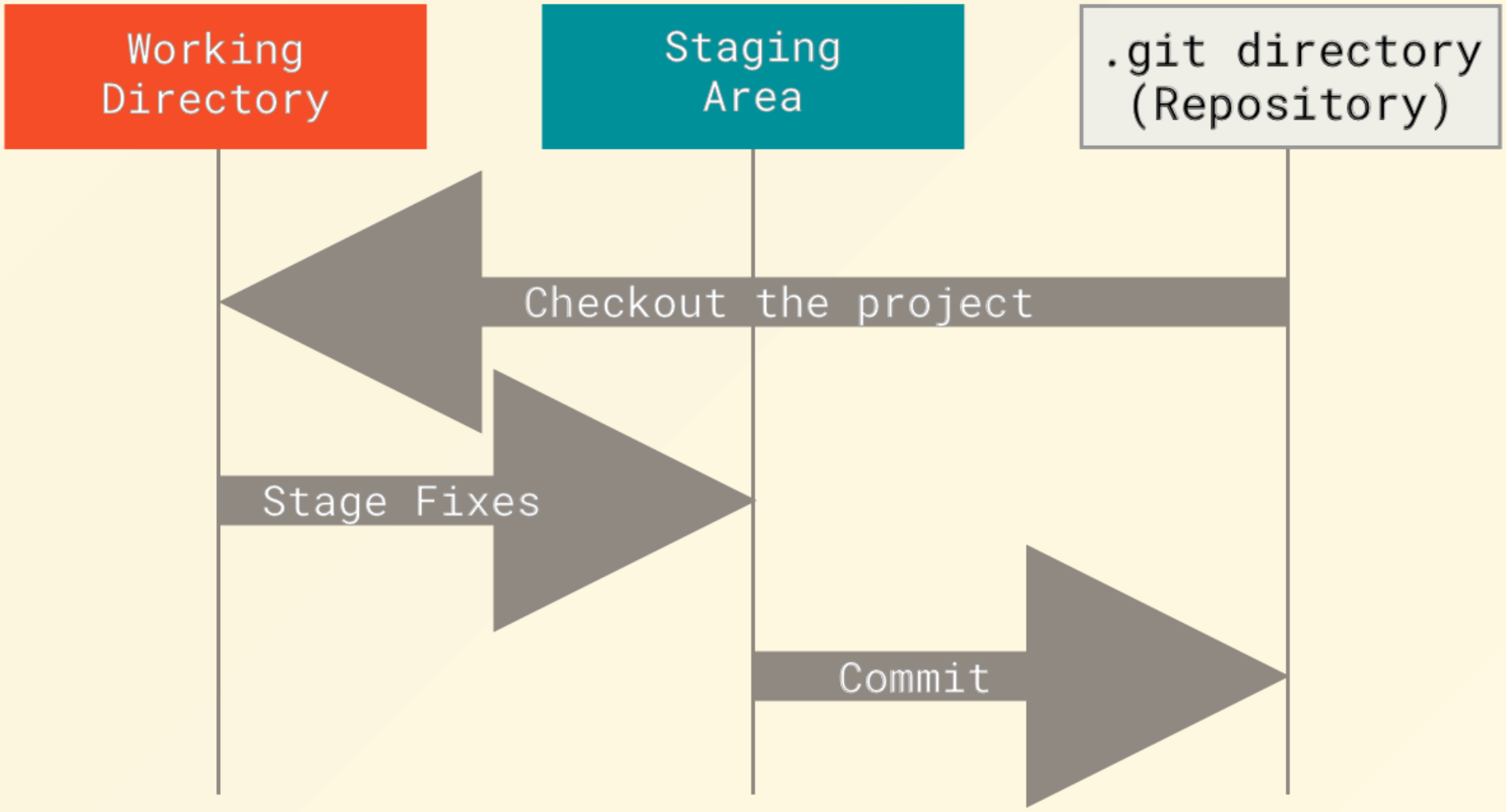
- `i` für interactive mode
- `esc` für normal mode
- `:wq!` beenden *mit* speichern
- `:q!` beenden *ohne* speichern

Übung 1: Git Setup

Wiederholung: Lifecycle einer Datei



Wiederholung: Git Arbeitsbereiche



`git clone`

Erstellt eine lokale Kopie eines bestehenden Repositories in einem neuen Verzeichnis.

```
git clone https://github.com/<Nutzer>/myfirstrepo.git/
Cloning into 'myfirstrepo'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Branching

In Git gibt es Branches, die auf Commits bzw. Snapshots verweisen. Eine Branch verweist auf einen Commit. Auf einen Commit kann von mehreren Branches gleichzeitig verwießen werden. Die momentan aktive Branch wird durch den HEAD bestimmt.

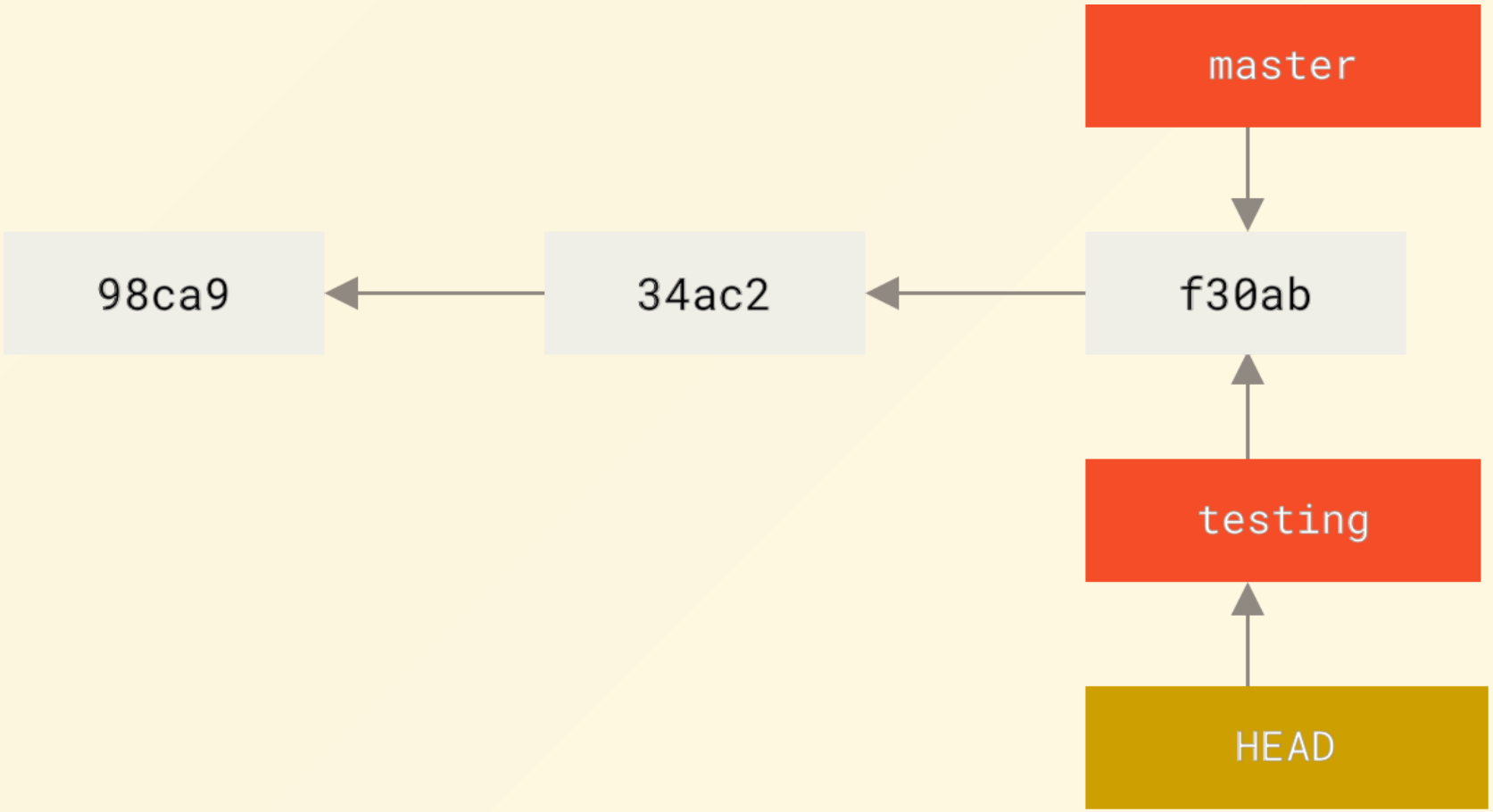
`git checkout`

Ermöglicht es, einen Branch auszuchecken.

```
git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

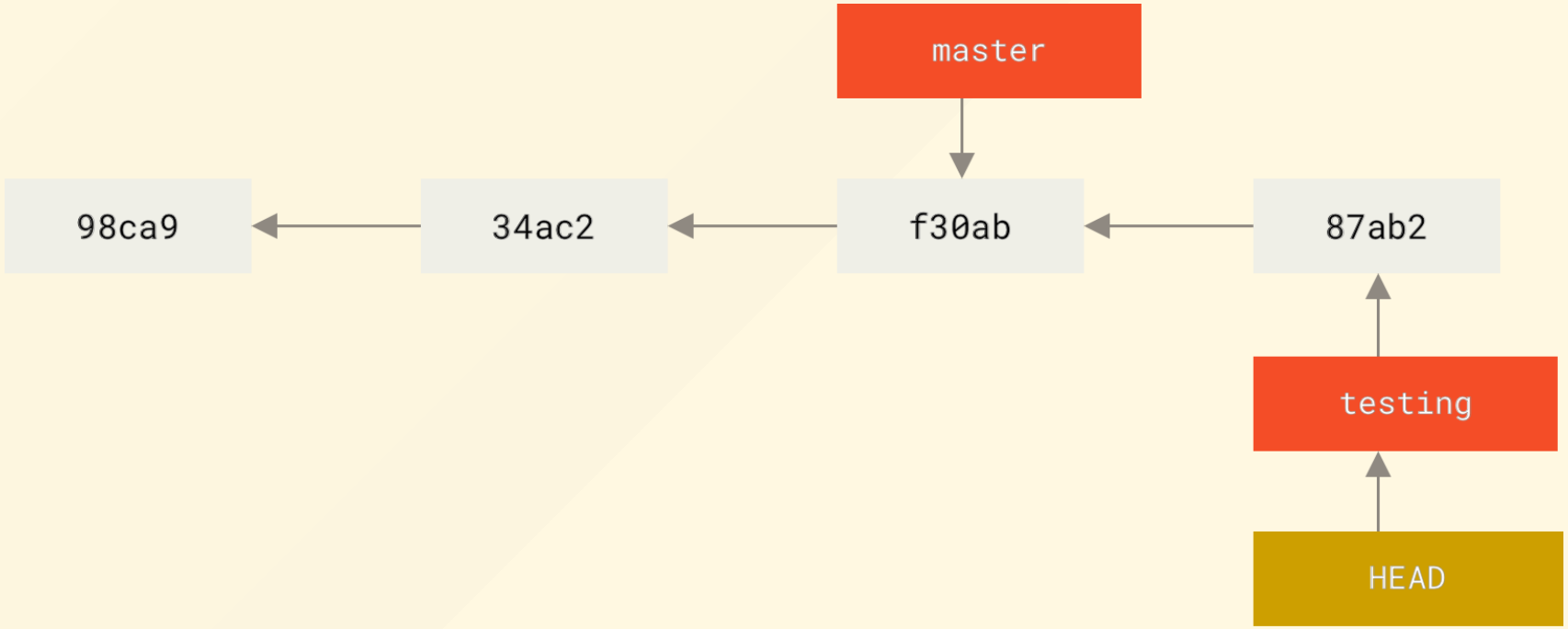
Kann auch genutzt werden um neue Branches zu erstellen

```
git checkout -b testing
Switched to a new branch 'testing'
```



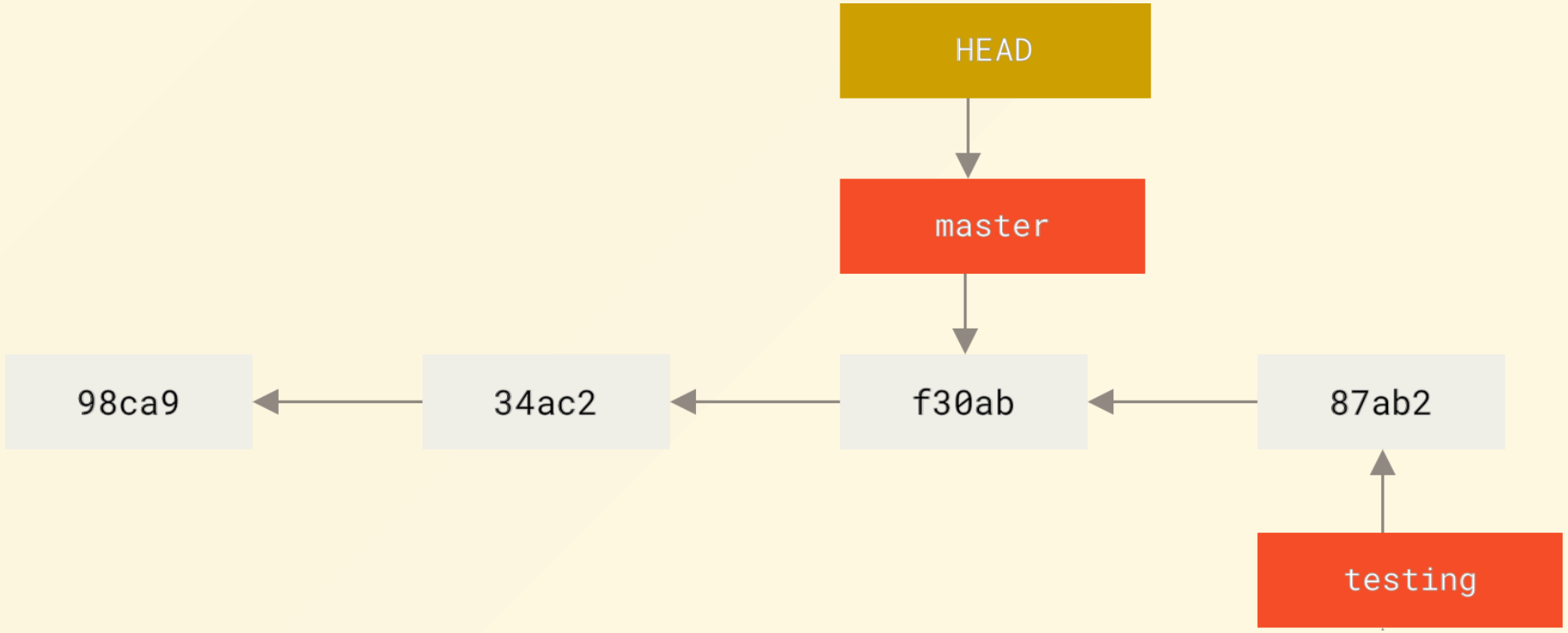
Wie würde sich der Graph bei einem commit in die `testing` Branch verändern?

```
vim neuedatei.txt
git add neuedatei.txt
git commit -m 'Added neuedatei.txt'
```



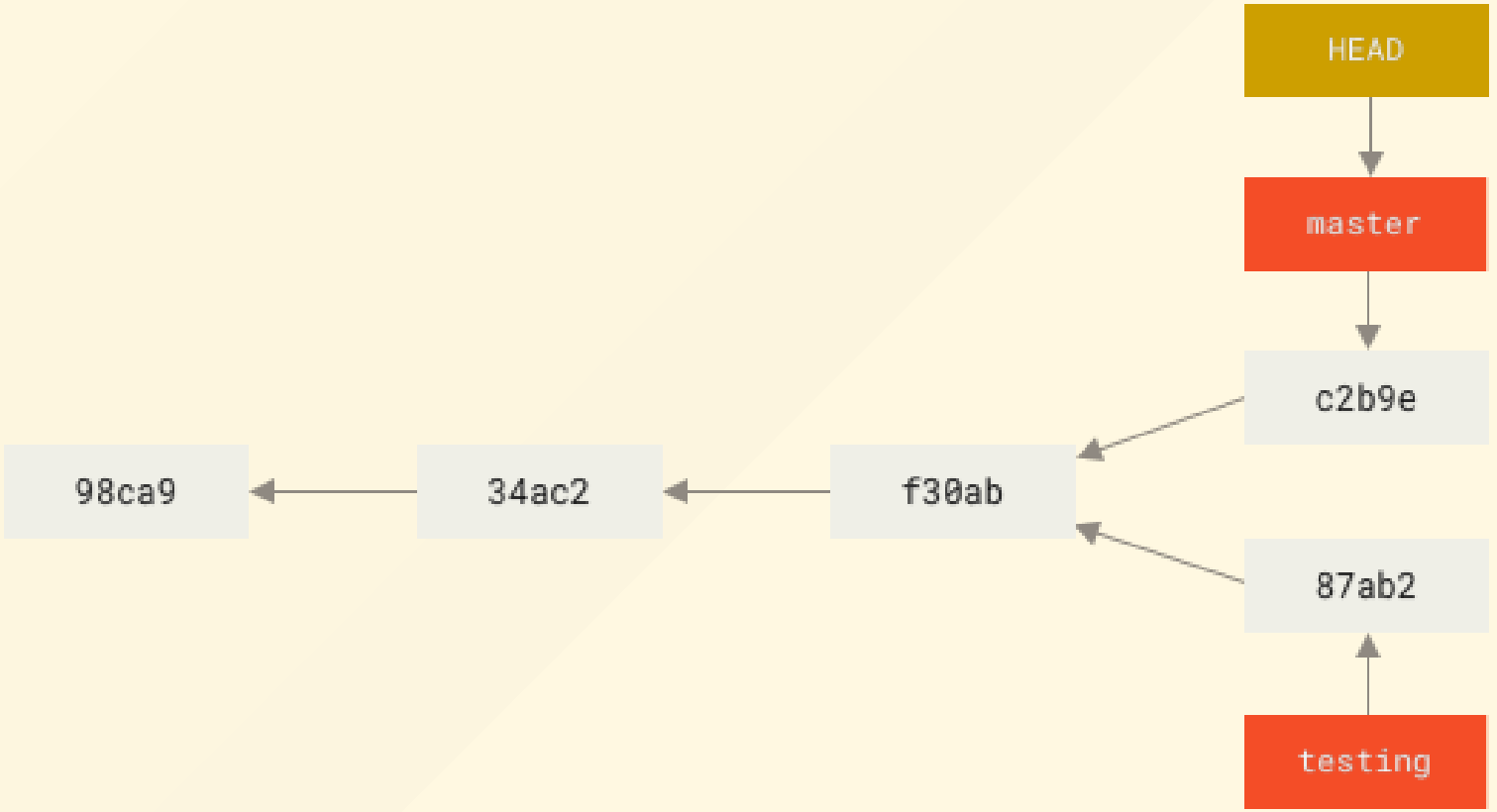
Was würde bei einem Checkout der `master` Branches passieren?

```
git checkout master
```

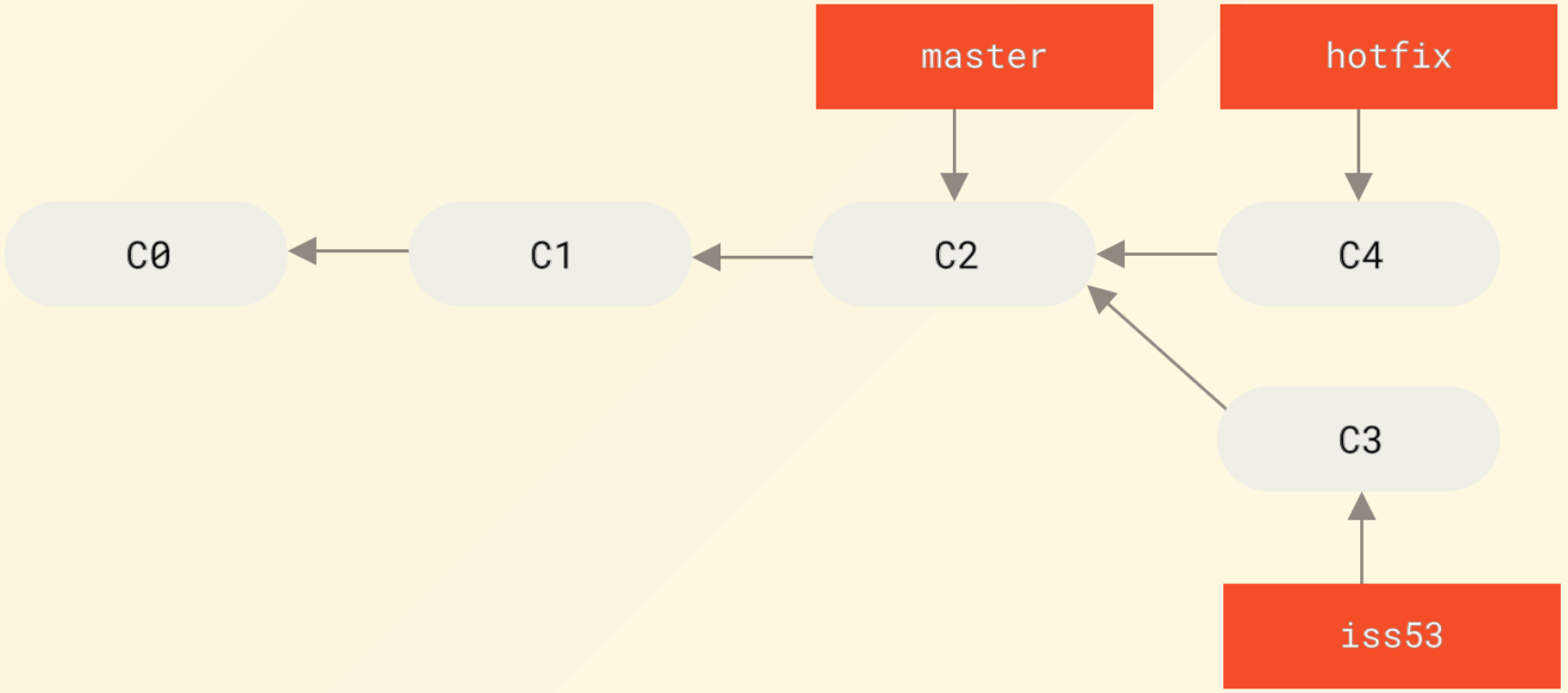


Wie würde ein Commit im `master` Branch aussehen?

```
vim readme.md
git commit -m 'Fix some minor issues'
```



Änderungen zusammenführen?



- Änderungen können in Git mit den Befehlen `git rebase` und `git merge` zusammengeführt werden.

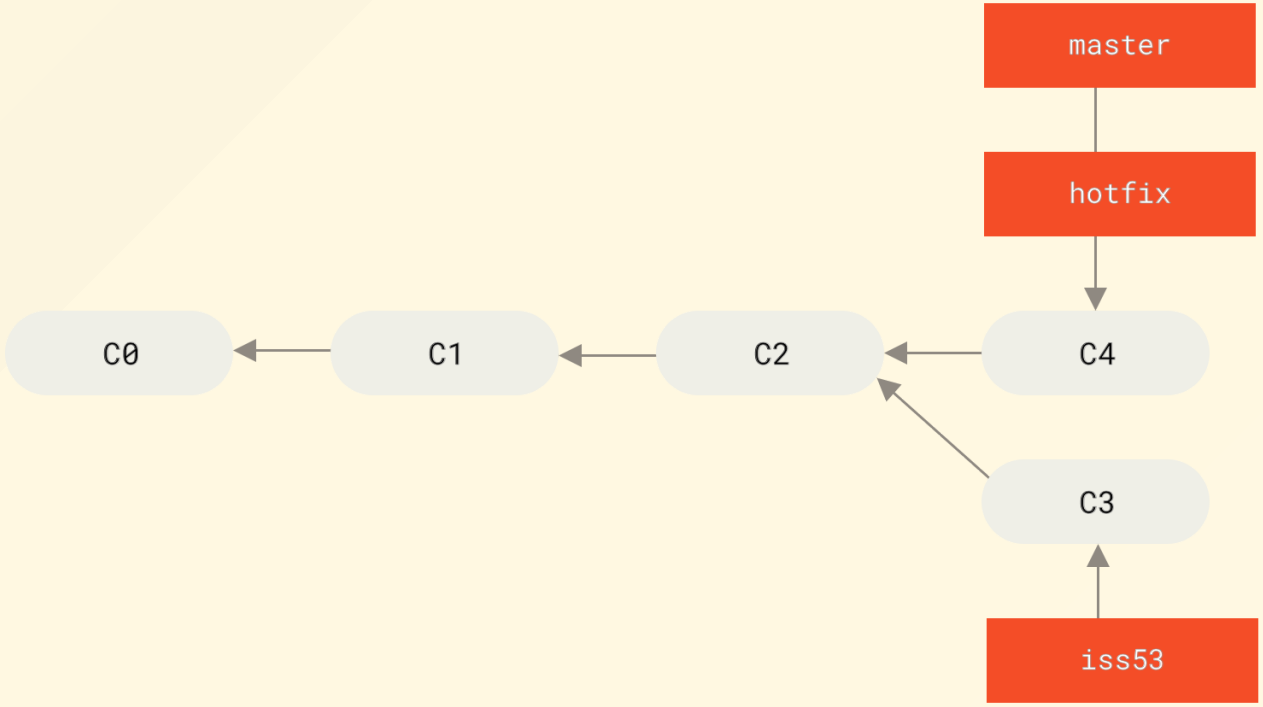
`git merge`

Fügt Commit Historien zusammen. So können z.B. Branches zusammengeführt werden. Je nach Situation verhält sich der Befehl unterschiedlich:

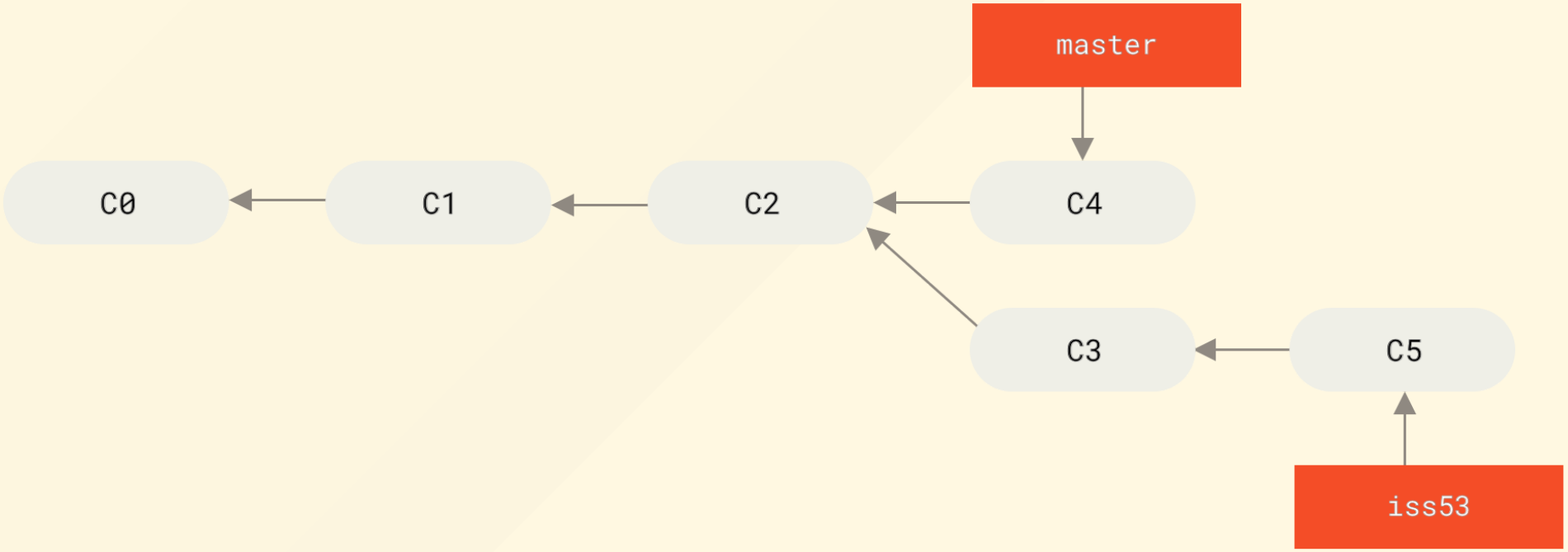
- Fast Forward: Eine der Historien hat keine Änderung
- Merge Commit: Beide Historien haben Änderungen
  - Automatisch: Änderungen sind in unterschiedlichen Dateien
  - Manuell: Änderungen in gleicher Datei

`git merge` - Fast-Forward

```
git merge hotfix
Updating f42c575..3a0874d
Fast-forward
```



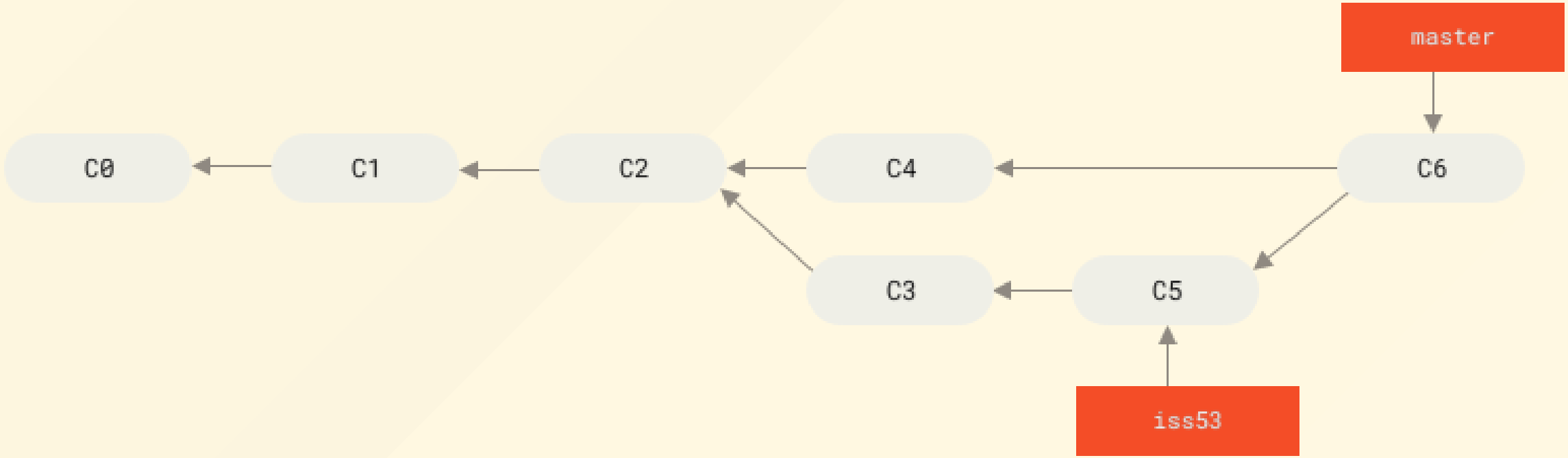
git merge



In dieser Situation muss der Merge einen Commit durchführen, der die Änderungen zusammenführt.

git merge

```
git checkout master
git fetch
git merge iss53
```



`git merge` - Konflikt

Falls Git nicht in der Lage ist die Änderungen selbständig zusammenzuführen entsteht ein Konflikt. Die Meldung sieht wie folgt aus:

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

Diesen Konflikt müssen Sie manuell auflösen.

`git merge` - Konflikt

Git hat Ihnen in den betreffenden Dateien Kommentare hinzugefügt. Diese sehen wie folgt aus:

```
echo Hello World
<<<<<<< HEAD
echo Zusaetzlicher Commit aus meiner Branch
=======
echo Zusaetzlicher Commit aus einer anderen Branch
>>>>>>>
```

- "`<<<<<<< HEAD`" bis "`=======`": Änderungen aus der lokalen Branch.
- "`=======`" bis "`>>>>>>>`": Änderungen aus der entfernten Branch.

`git merge` - Konflikt

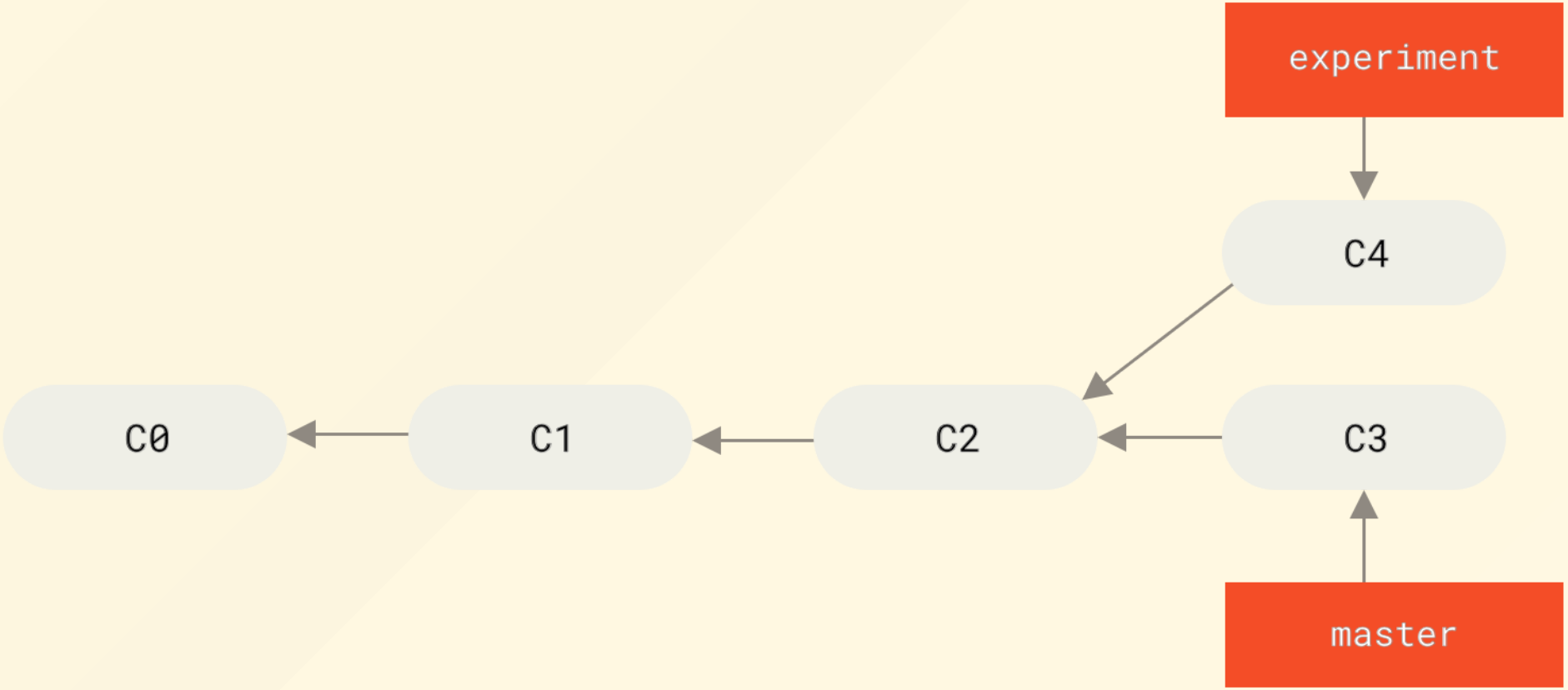
1. Führen Sie die Änderungen zusammen.
2. Entfernen Sie die Sonderzeichen von Git.
3. Fügen Sie die Änderung mit `git add` hinzu.
4. Überprüfen Sie den Status mit `git status`.

Wenn alle Dateien mit Konflikten zusammengeführt sind beenden Sie den Vorgang mit

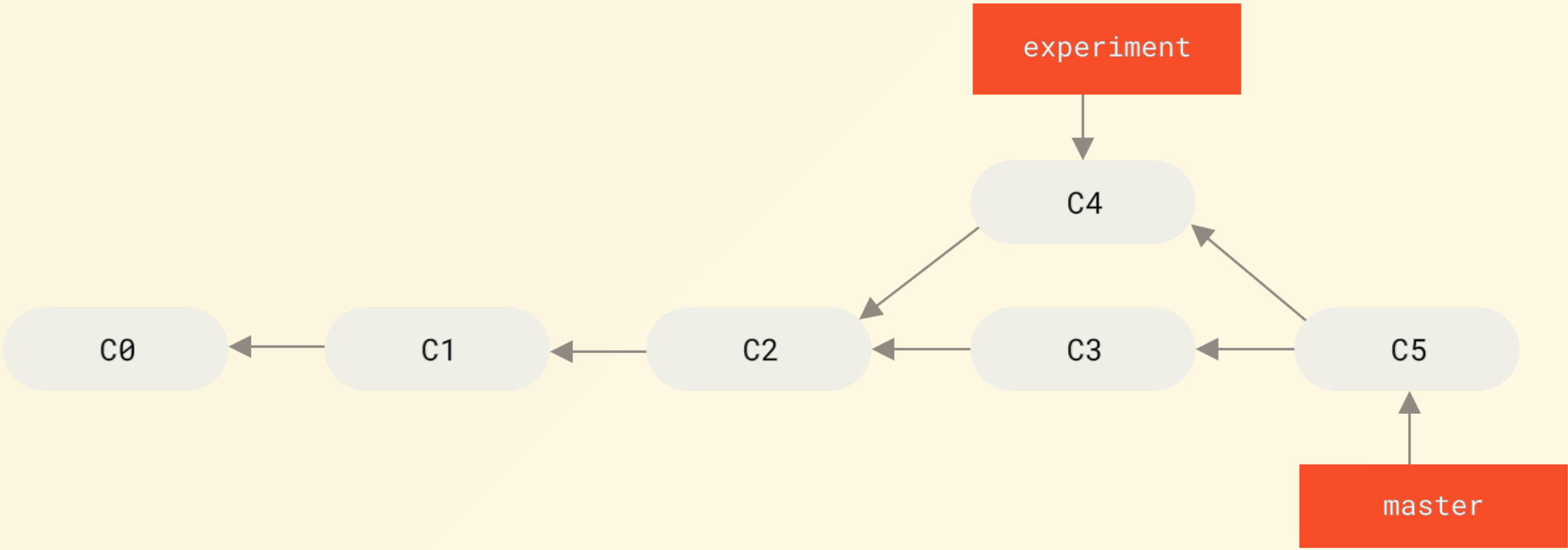
```
git merge --continue
```

`git merge` vs `git rebase`

Wie würde das Ergebnis eines `git merge` bei folgenden Ausgangszustand aussehen?

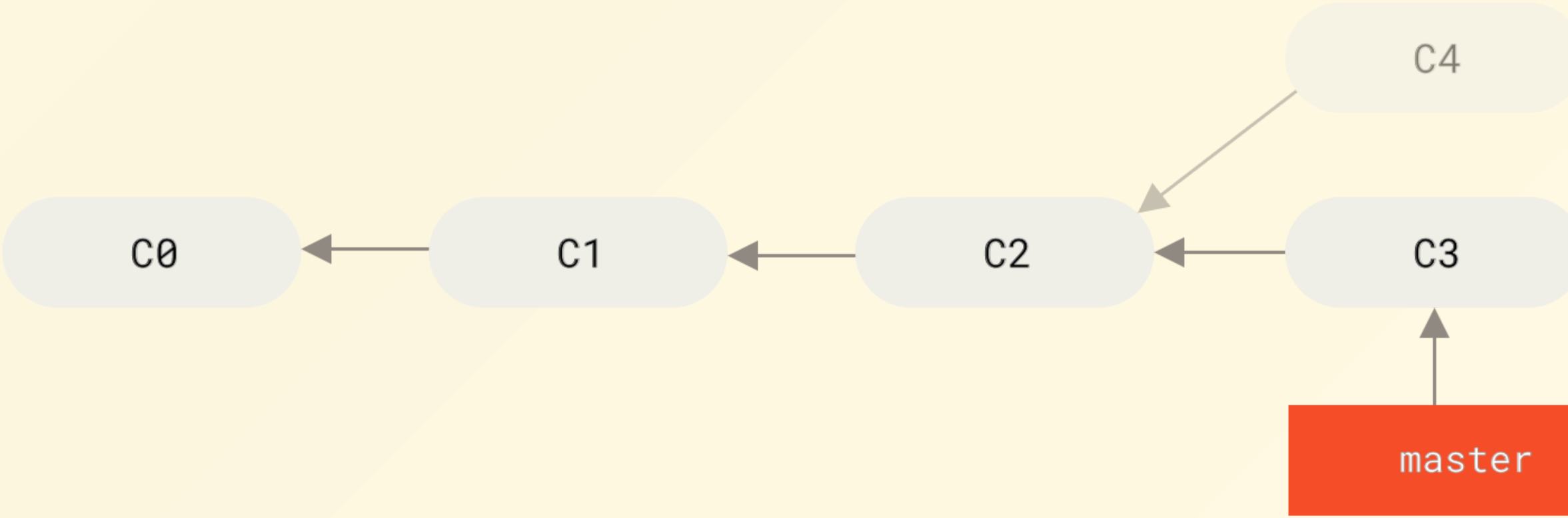


`git merge` - Ergebnis



`git rebase`

`git rebase` wendet die Veränderung eines Commits auf einen anderen Softwarestand an.



`git rebase` Vorsicht!!!

`git rebase` verändert die Historie. D.h. Sie sollten Git Rebase niemals auf Commits anwenden, die Sie bereits mit anderen geteilt haben.

`git fetch`

Empfängt Informationen aus dem Remote Repository. Folgenden Entitäten (refs) werden abgeholt:

- Branch
- Tag
- Commit

Der HEAD wird noch nicht verändert.

Übung 2 & 3 - Paralleles Arbeiten

(mit Konflikt)

.gitignore

In dieser Datei können Dateien aufgenommen werden, sodass diese bei Änderungen nicht berücksichtigt werden. Auch Dateiendungen bzw. Wildcard Patterns sind möglich.

```
# ignore all .log files
*.log

# but do track lib.log, even though you're ignoring .log files above
!lib.log

# ignore the target directory
target/
```

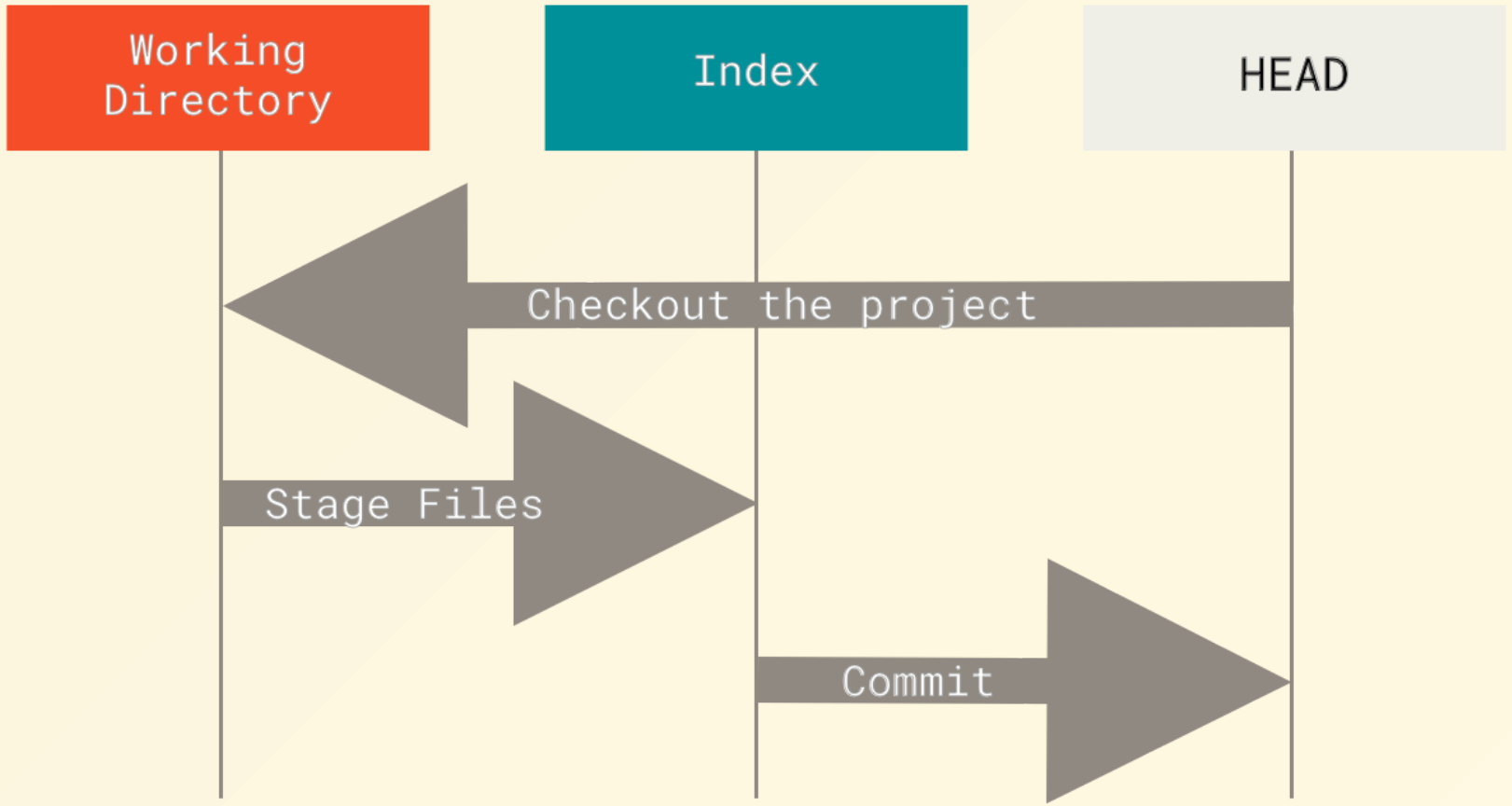
`git log`

Zeigt die Historie von Git an. Die Option `--graph --oneline` verbessert die Lesbarkeit.

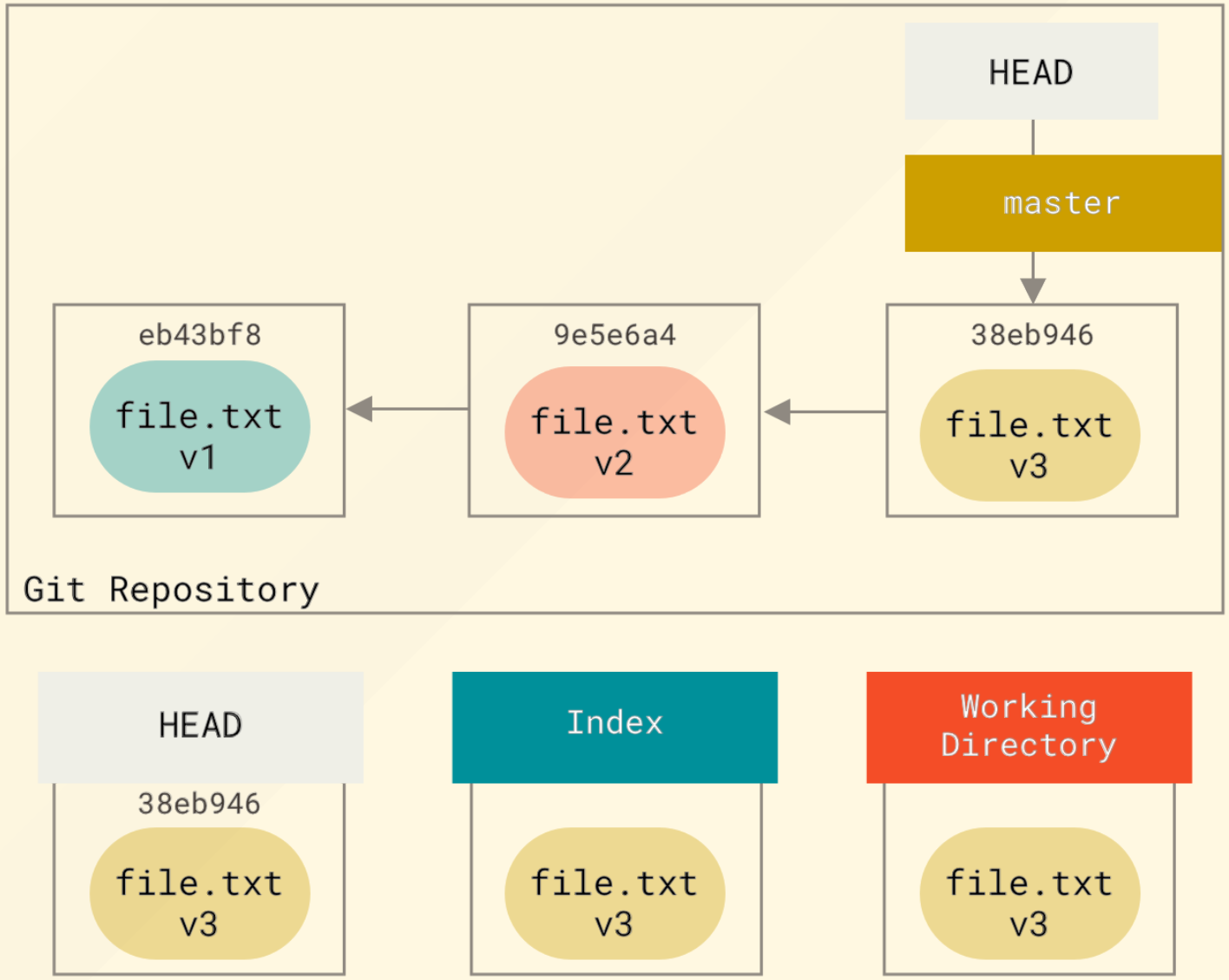
```
git log --graph --oneline
*   92e03be (HEAD -> master, sshorigin/master) Merge remote-tracking branch 'refs/remotes/sshorigin/master'
|\
| * 02ac5df Meine Erwartungen
* | 3930dff Erwartungen 3
|/
* 1b72bf0 Meine Erwartungen 2
* 774da85 (origin/master, origin/HEAD) Feature: Contributer.md added
* 562fbb8 (newBranch) My first commit
```

`git reset`

Manchmal muss man den *Arbeitsbereich*, den *Index* oder den *HEAD* verändern.

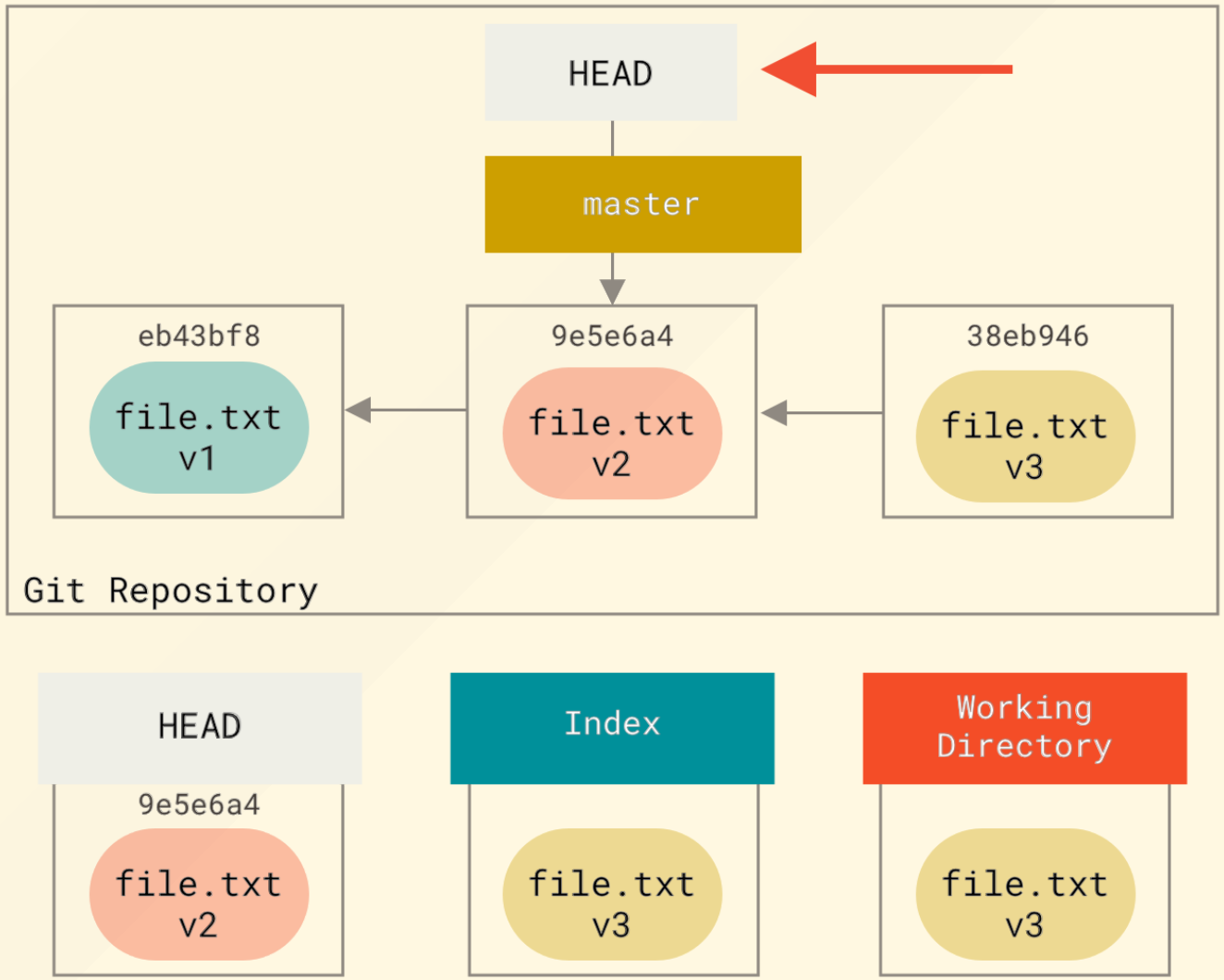


git reset


HEAD
master
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
38eb946
file.txt
v3
Index
file.txt
v3
Working
Directory
file.txt
v3

`git reset --soft`

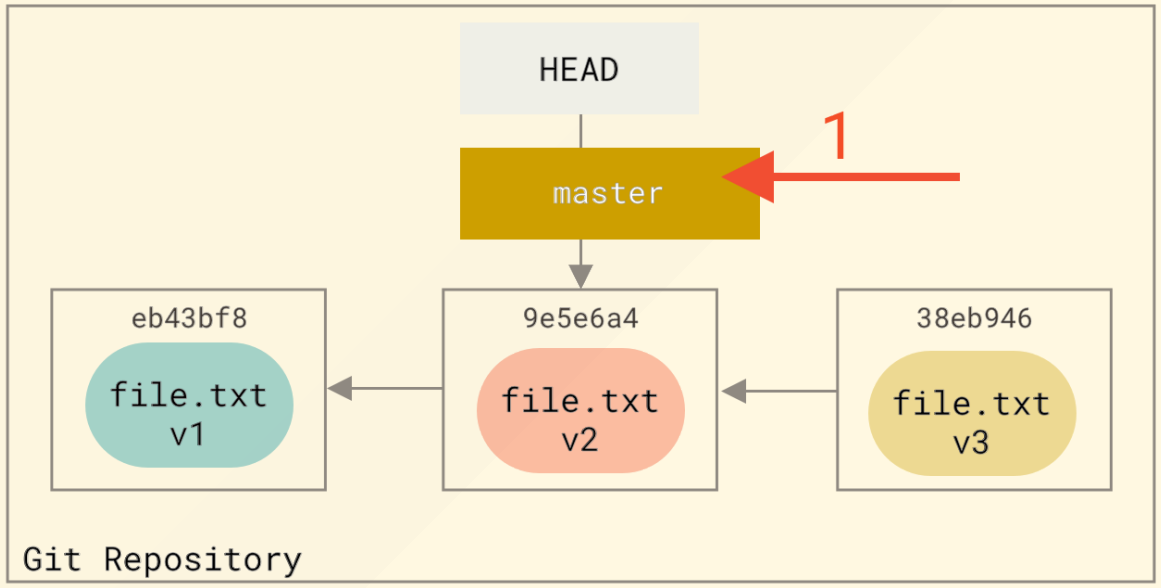
Setzt lediglich HEAD zurück, Index und Arbeitsbereich bleiben gleich.

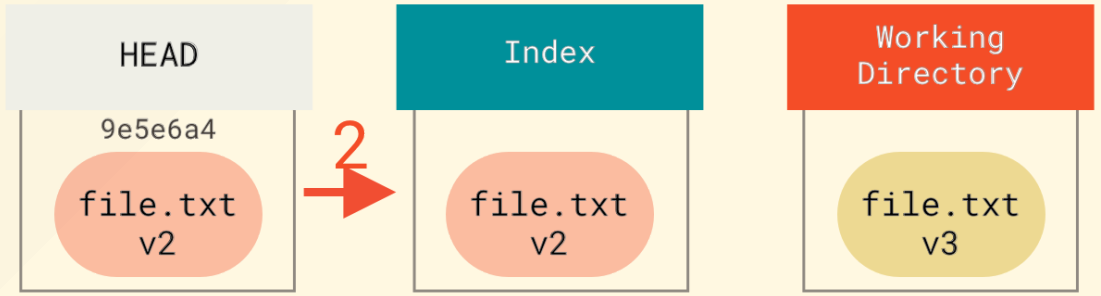


`git reset --soft HEAD~`

`git reset --mixed`

Setzt HEAD und Index zurück, der Arbeitsbereich bleibt gleich.

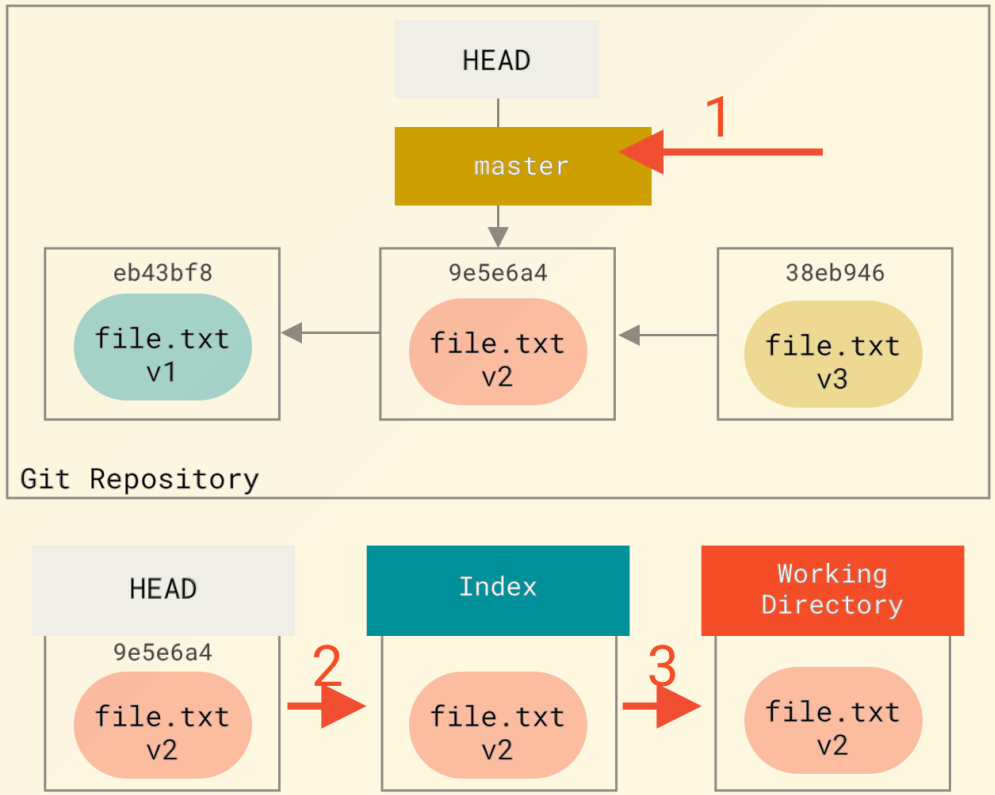




`git reset [--mixed] HEAD~`

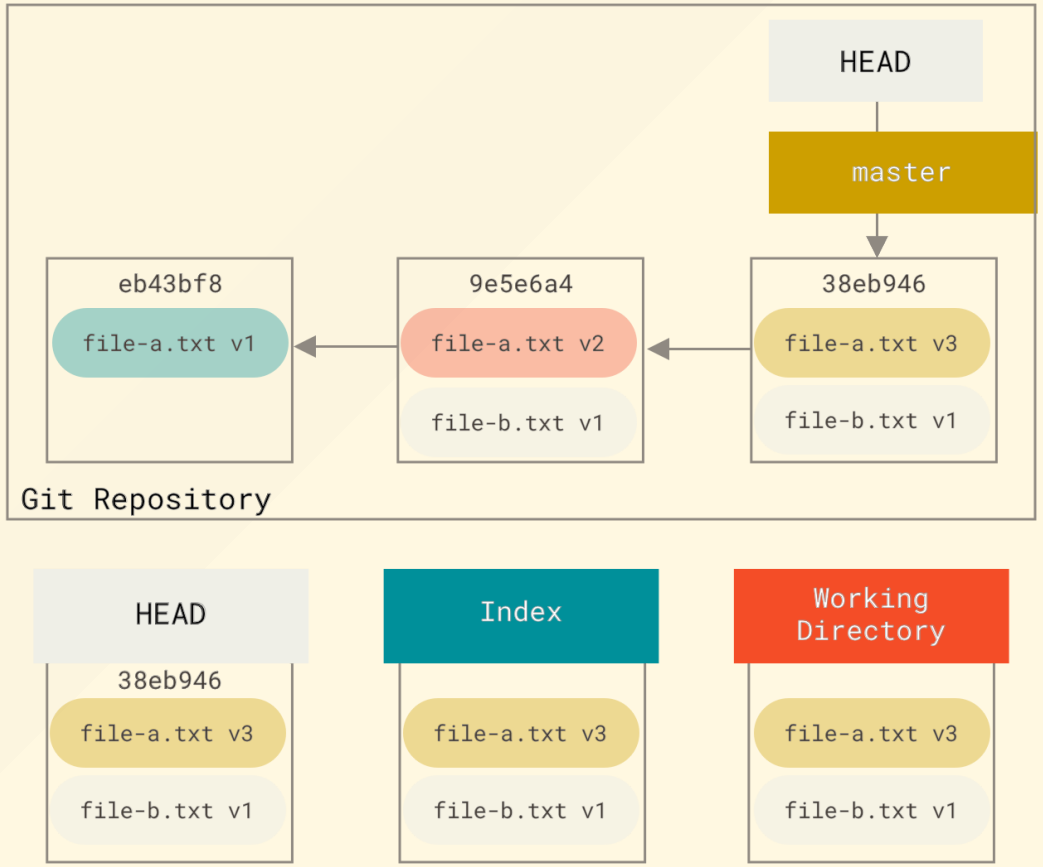
`git reset --hard`

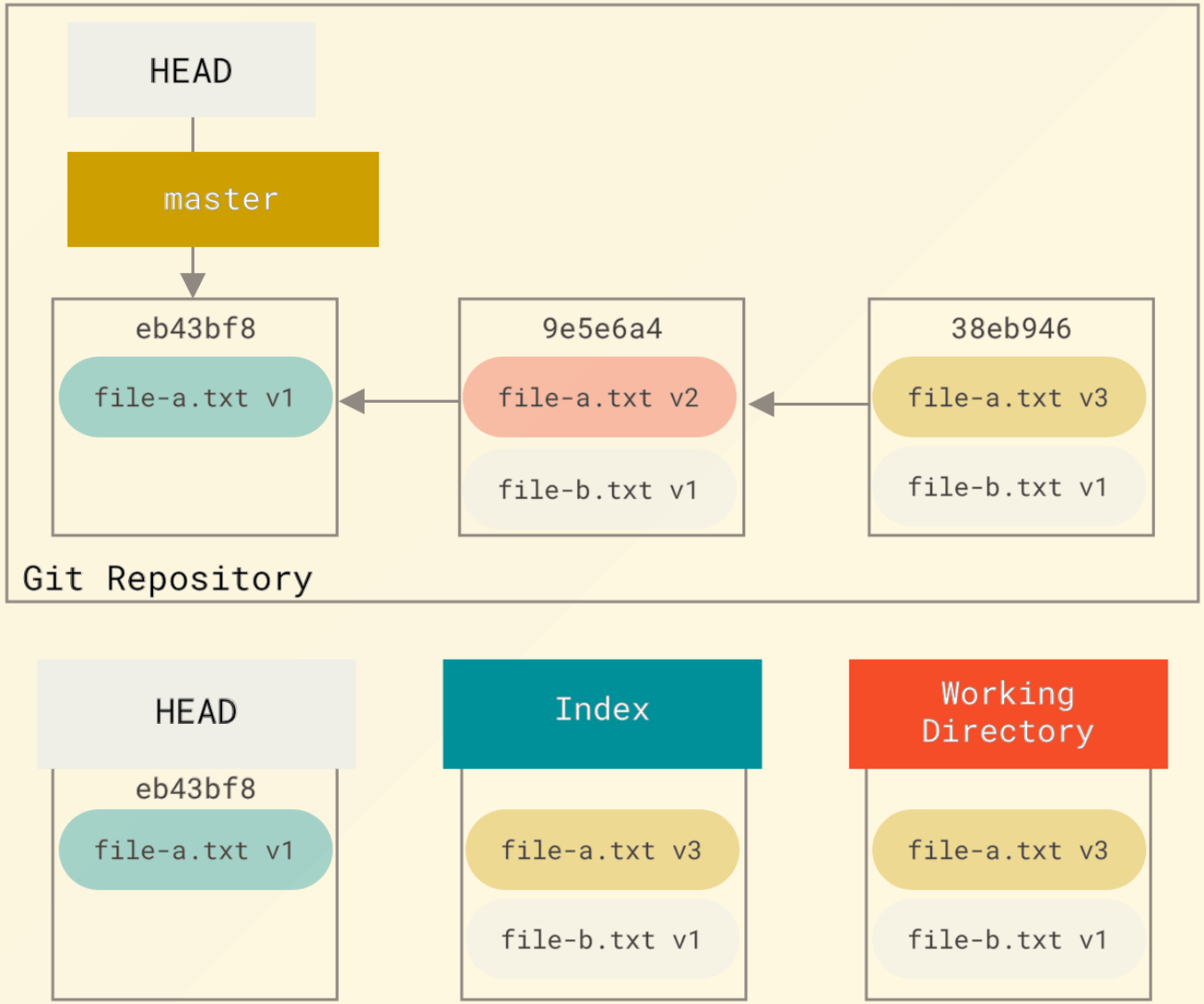
Setzt HEAD, Index und Arbeitsbereich (Vorsicht!) zurück.



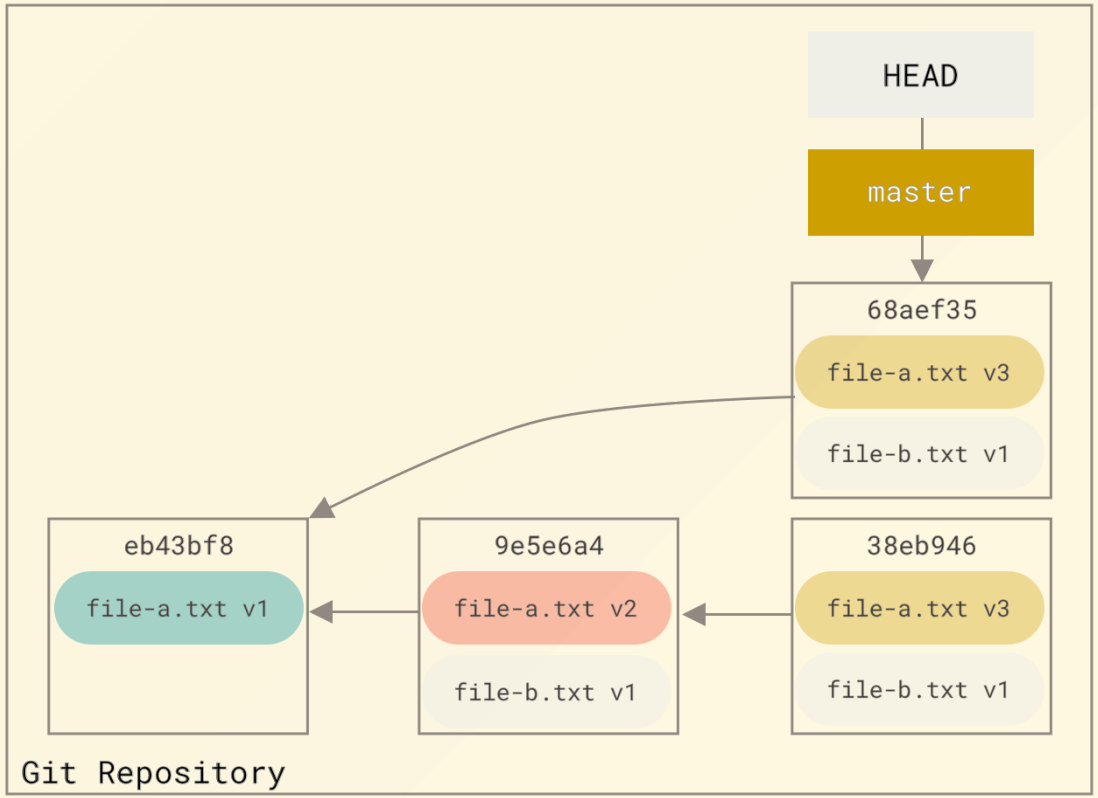
`git reset` squash commits

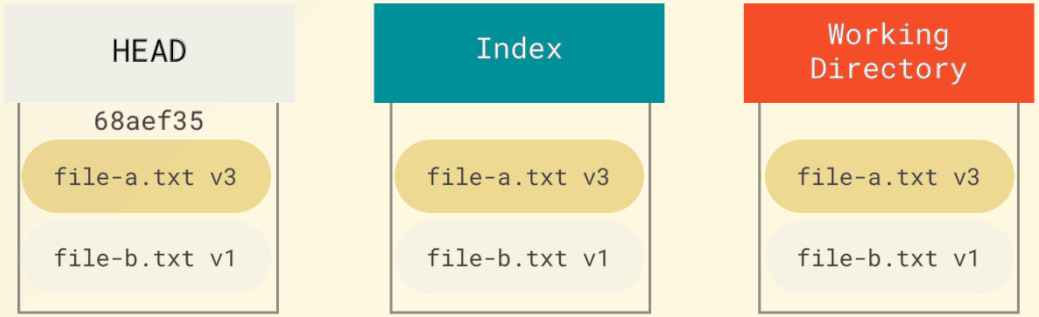
Wenn man Commits wie "WIP", "noch nicht fertig" oder "Buggy" hat, möchte man diese in der Regel zusammenführen.





git reset --soft HEAD~2


HEAD
master
68aef35
file-a.txt v3
file-b.txt v1
eb43bf8
file-a.txt v1
9e5e6a4
file-a.txt v2
file-b.txt v1
38eb946
file-a.txt v3
file-b.txt v1
Git Repository


HEAD
68aef35
file-a.txt v3
file-b.txt v1
Index
file-a.txt v3
file-b.txt v1
Working Directory
file-a.txt v3
file-b.txt v1

git commit

Links

- Git Cheat Sheet
- Git Book
- Source Tree (Git GUI)