

Robert Kleinschmager

- Dipl. Ing. (BA) - 2004
- Developer, Projektleiter, Berater
- Folien <https://github.com/barclay-reg/dhbw-slides>

Nicht-Triviale Software Entwicklung

Robert Kleinschmager | github.com/barclay-reg

WIESO NICHT-TRIVIAL?

Coding

Coding

Testing

Design

Coding

Testing

Design

Coding

Testing

Documentation

Analysis

Design

Coding

Testing

Documentation

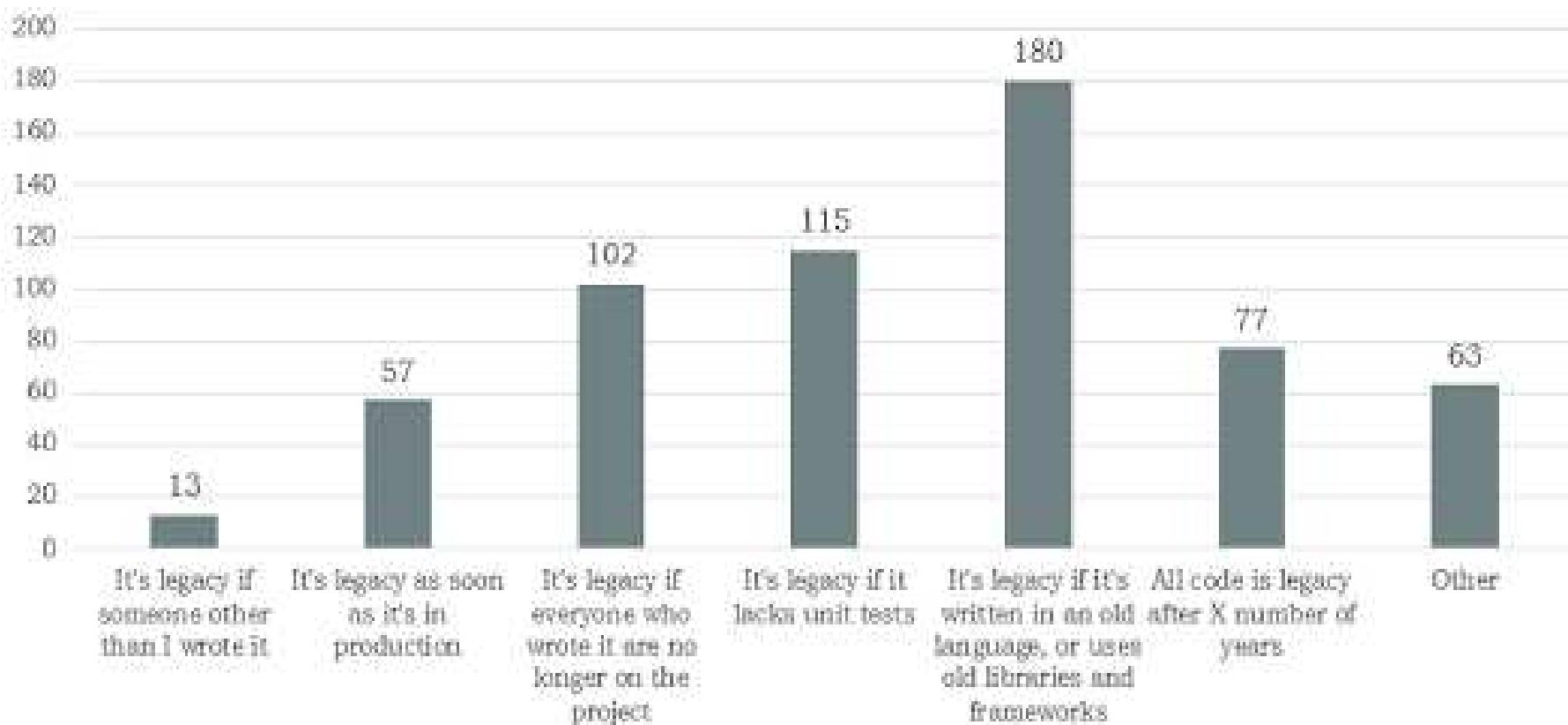
LEGACY CODE

$\mathcal{O}(n)$ Computer Science @CompSciFact · Jun 13
RT @KenScambler: Legacy. Looks easy; should be done in half an hour I reckon.
pic.twitter.com/XYJzKt71

Reply Retweet Favorite

LEGACY CODE - UMFRAGE

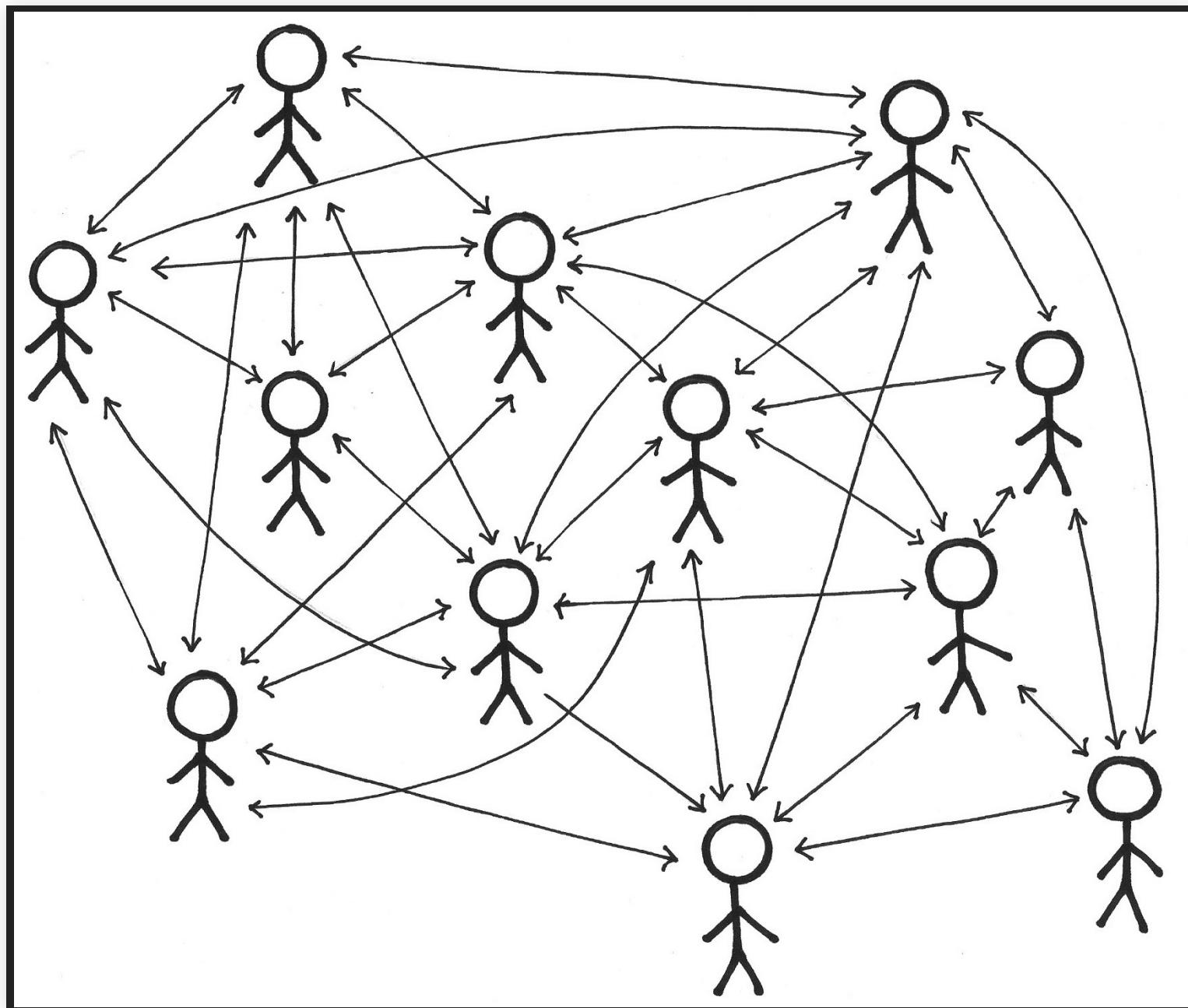
In your opinion, what makes code «legacy»?



RAHMENBEDINGUNGEN

- Funktionale Requirements
- Nicht-Funktionale Requirements
- (alte) Werkzeuge
 - IDE
 - Bibliotheken

TEAM



KONSEQUENZEN

KOMPLIZIERT

- das Maß an Ungewissheit
- Wissen fehlt
- Kompliziertes System
 - zuverlässig, exakt
 - von außen steuerbar
 - konstantes Verhalten

KOMPLEX

- das Maß der Menge an Überraschungen
- Kontext-abhängig
- Komplexes System
 - "lebendig", erzeugt Überraschungen
 - von außen nur beobachtbar
 - dynamisches Verhalten



- Kompliziert oder Komplex?





- Kompliziert





- Kompliziert oder Komplex?





• Komplex



- Kompliziert oder Komplex?





Software wächst in der "freien Wildbahn" in einem Spannungsfeld





- Coding **wird** zunehmen schwerer

ÜBERGANG VON KOMPLIZIERT ZU KOMPLEX

- Schätzungen **werden** ungenauer
- Analysen **werden** umfangreicher
- Test **werden** aufwändiger, vielfältiger
- Dokumentation **wird** umfassender
- Technische Schulden **entstehen**

KOMPLEXITÄT IN DER SOFTWAREENTWICKLUNG

*software systems grow faster in size
and complexity than methods to
handle complexity are invented*

— Wirth's Law

Programming is still a stone-age crafts

GEGENMASSNAHMEN

CLEAN CODE

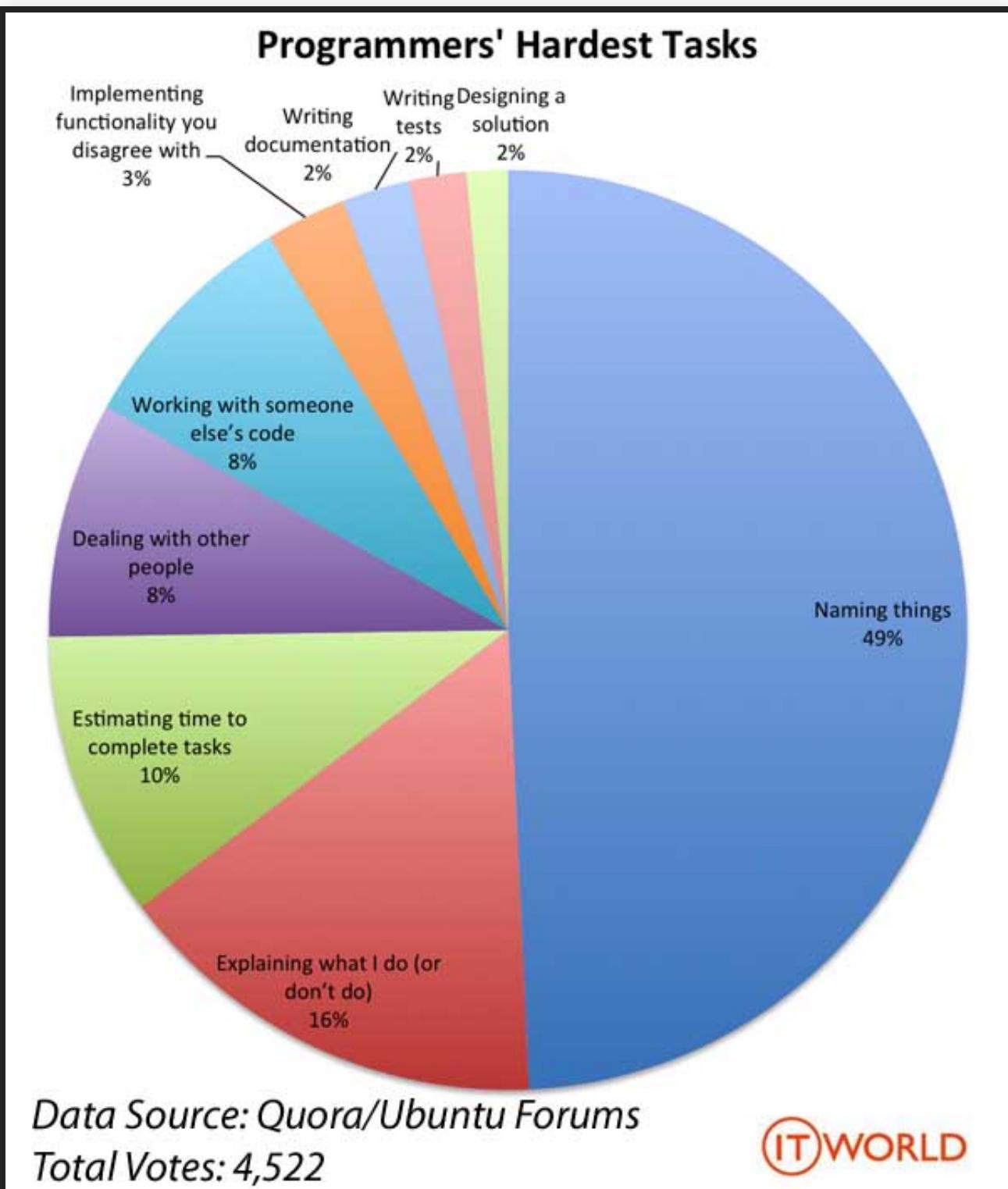
- Code wird nur einmal **geschrieben**, aber viele male **gelesen**
- Keine Code-Magneten schreiben;
 - Single Responsibility Principle (Theorie)
 - Main Purpose Principle (Praxis)
- **Komposition over Inheritance**

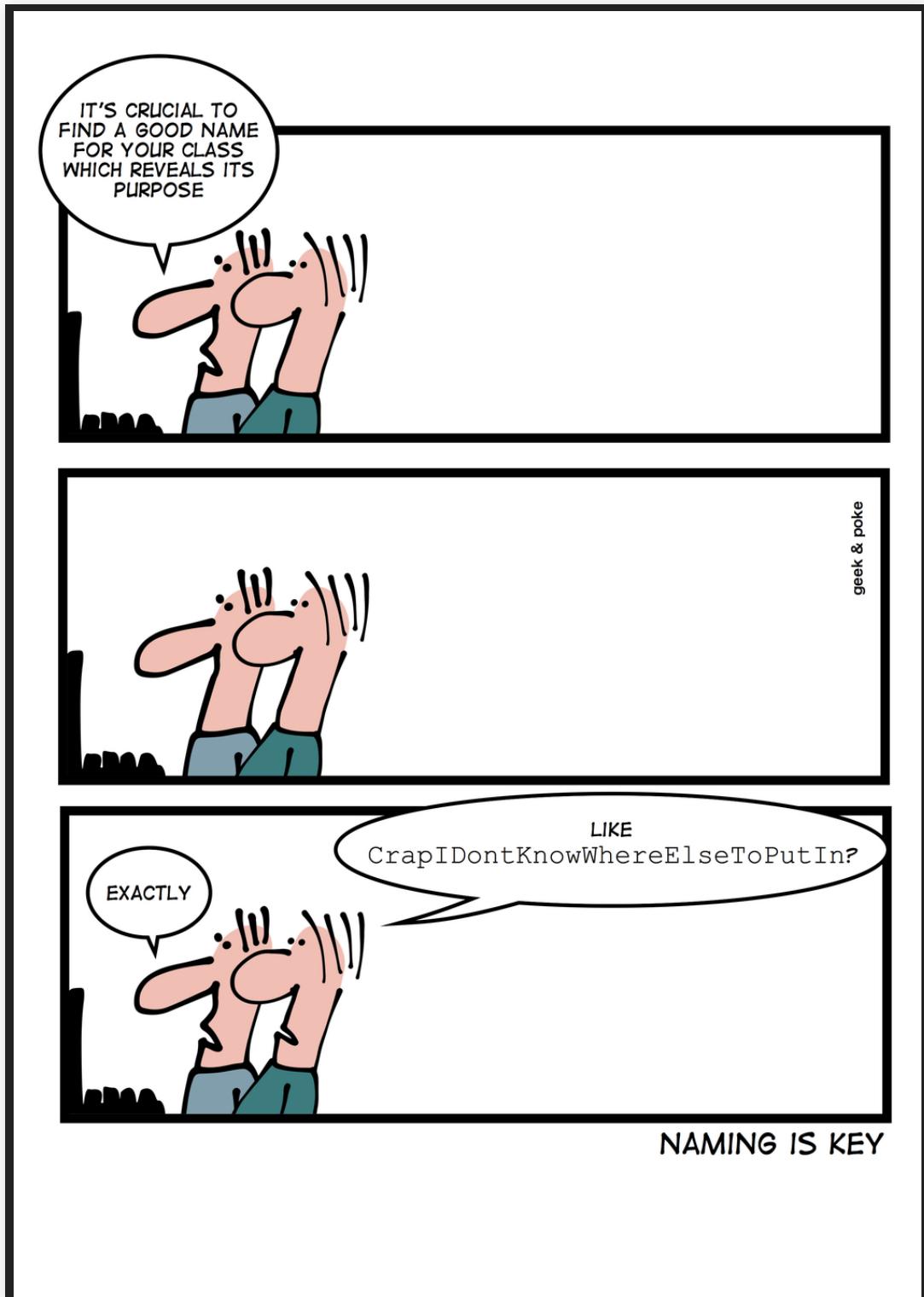
Broken Window Principle

— Robert “Uncle Bob” Martin

*Leave the campground cleaner than
you found it*

— Robert “Uncle Bob” Martin





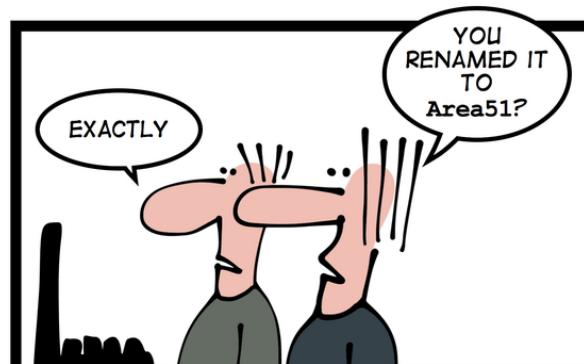
DOKUMENTATION

- Nur Konzepte & Ideen
 - diese möglichst nahe am Code (== gut erreichbar für den Entwickler)
- Nicht WAS, sondern WARUM
- Je expressiver der Code, desto weniger Dokumentation ist nötig

REFACTORING

Struktur verändern
ohne
Verhalten zu ändern

REFACTORING IS KEY



CONTINUOUS INTEGRATION

- Compiliere
- Paketieren
- (automatisiert) Testen
- Validieren
- Verifizieren

NACHVOLLZIEHBARKEIT

Analyse → Anforderung → Aufgabe → Code → Test →
Auslieferung

METRIKEN

- Messen des **Status-Quo** der Software-Komplexität
 - Wird sie schlechter oder besser?
- Identifikation der größten Risiken
 - Wichtig für Priorisierung bei limitiertem Budget/Zeit

The only valid code metric is WTFs per minute.

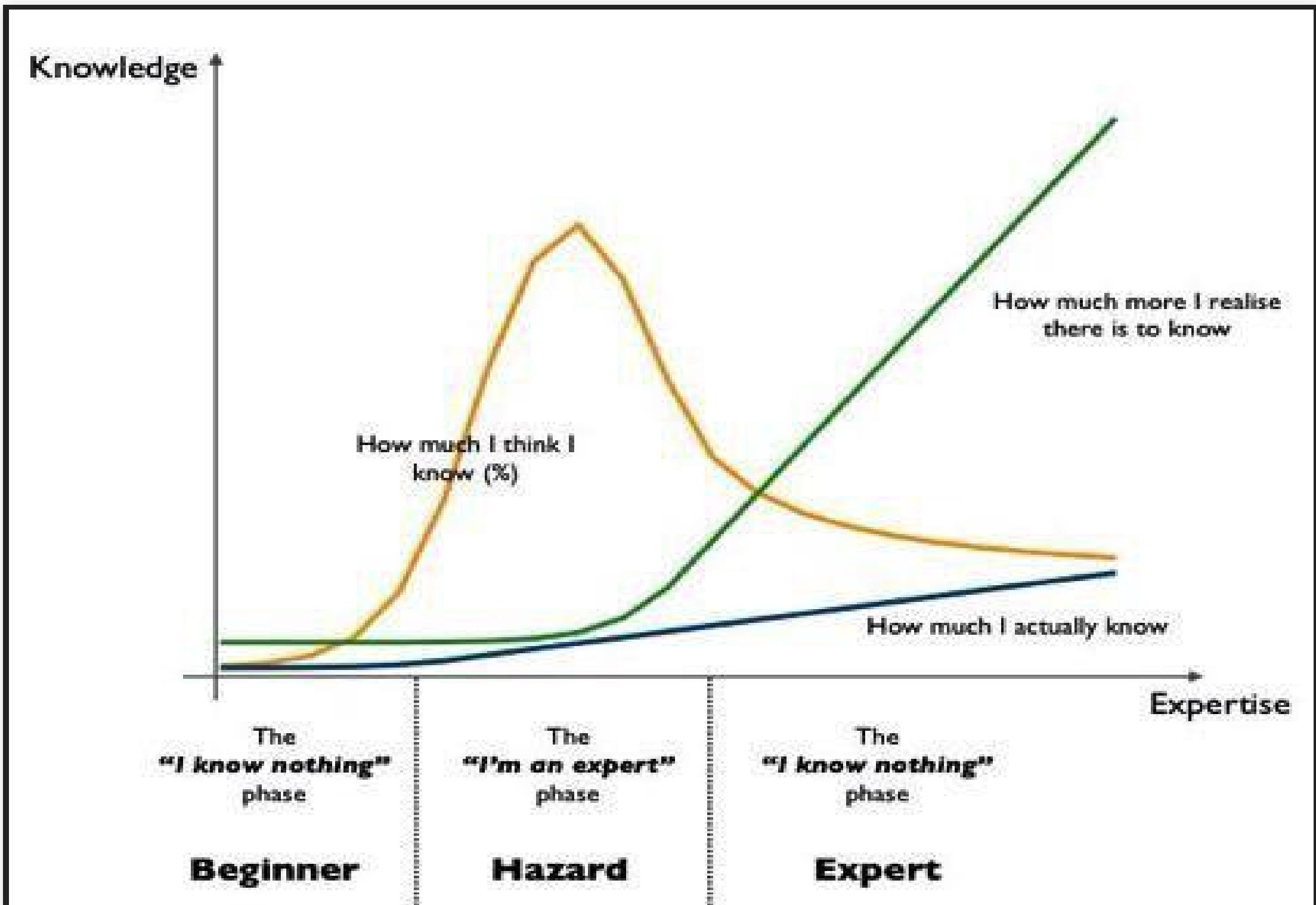
— Robert “Uncle Bob” Martin

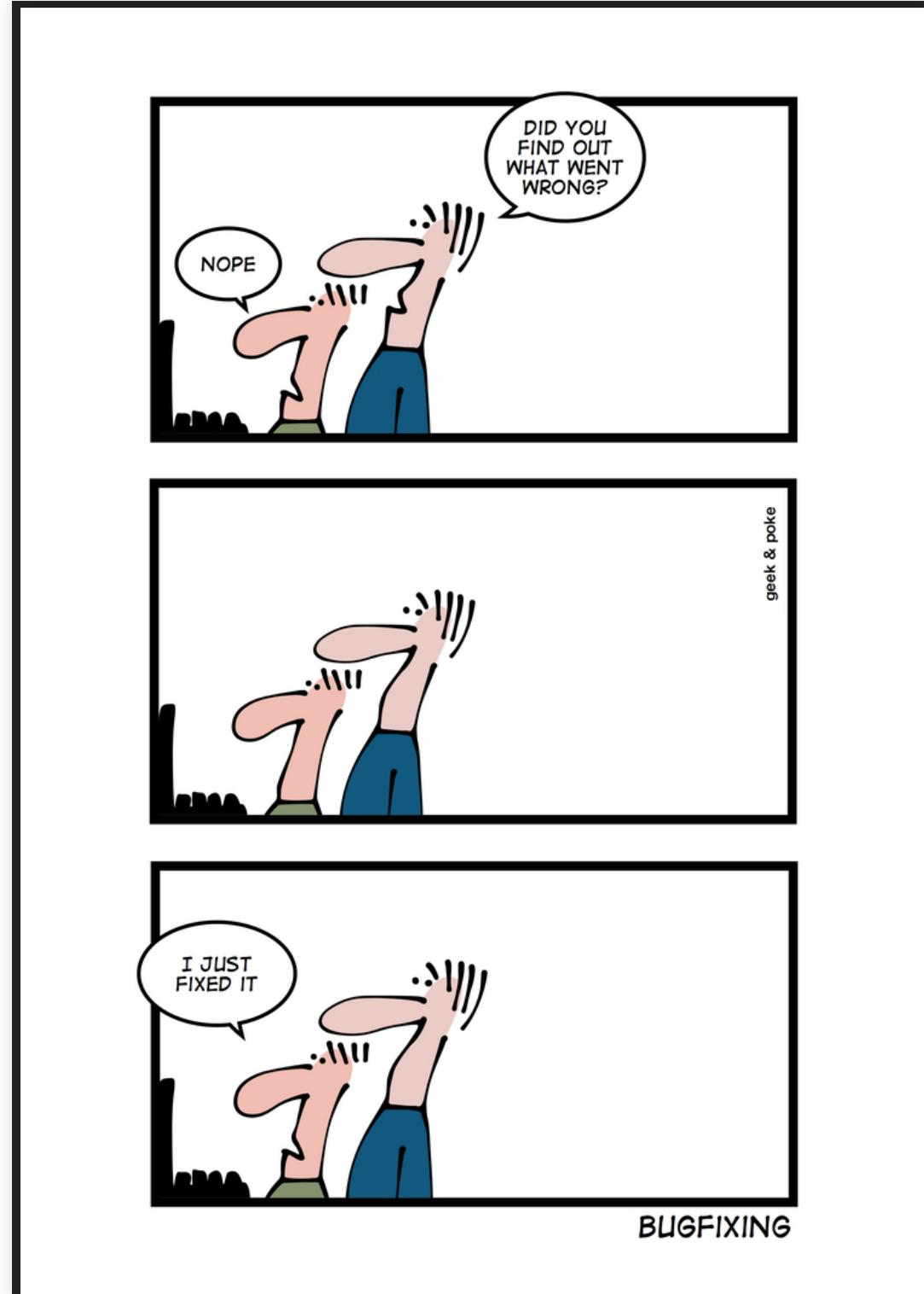
SELBSTKRITIK

- Regelmäßig den eigenen Code überprüfen

*The worst programmer in the world is
you – one year ago.*

— Andy Hunt





ANALYTISCHES TALENT

- Analyse von Kundenwünschen
 - Was will der Kunde und WARUM?
- Analyse von Fehlern/Bugs
- Debugging
- Aufstellen und Verifizieren von Hypothesen
- Bewertung von Hypothesen
 - == Erfahrung
- Schritt-für-Schritt Vorgehen
- Zyklisch verfeinern

FAZIT

Kontinuierliches Anwenden und Verbessern dieser Maßnahmen ist unsere beste Waffe gegen die zunehmende Komplexität

QUELLEN

BILDER

- Legacy Wall

<https://raw.githubusercontent.com/bettercodehub/pitch/master/assets/legacy-code.png>

- Legacy Survey <http://www.karolikl.com/2015/10/your-definition-of-legacy-impacts-how.html>

- Team http://scrumbook.org.datasenter.no/images/SmallTeam_Pre.jpg

- Programmers hardest task

<https://www.itworld.com/article/2833265/cloud-computing/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html>

- Naming is Key <http://geek-and-poke.com/geekandpoke/2013/8/20/naming-is-key>

- Refactoring is Key <http://geek-and-poke.com/geekandpoke/2013/8/26/refactoring-is-key>

BUCHTIPPS

The Pragmatic Programmer, From Journeyman To Master

— Andy Hunt

Clean Coder / Clean Coder

— Robert “Uncle Bob” Martin

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

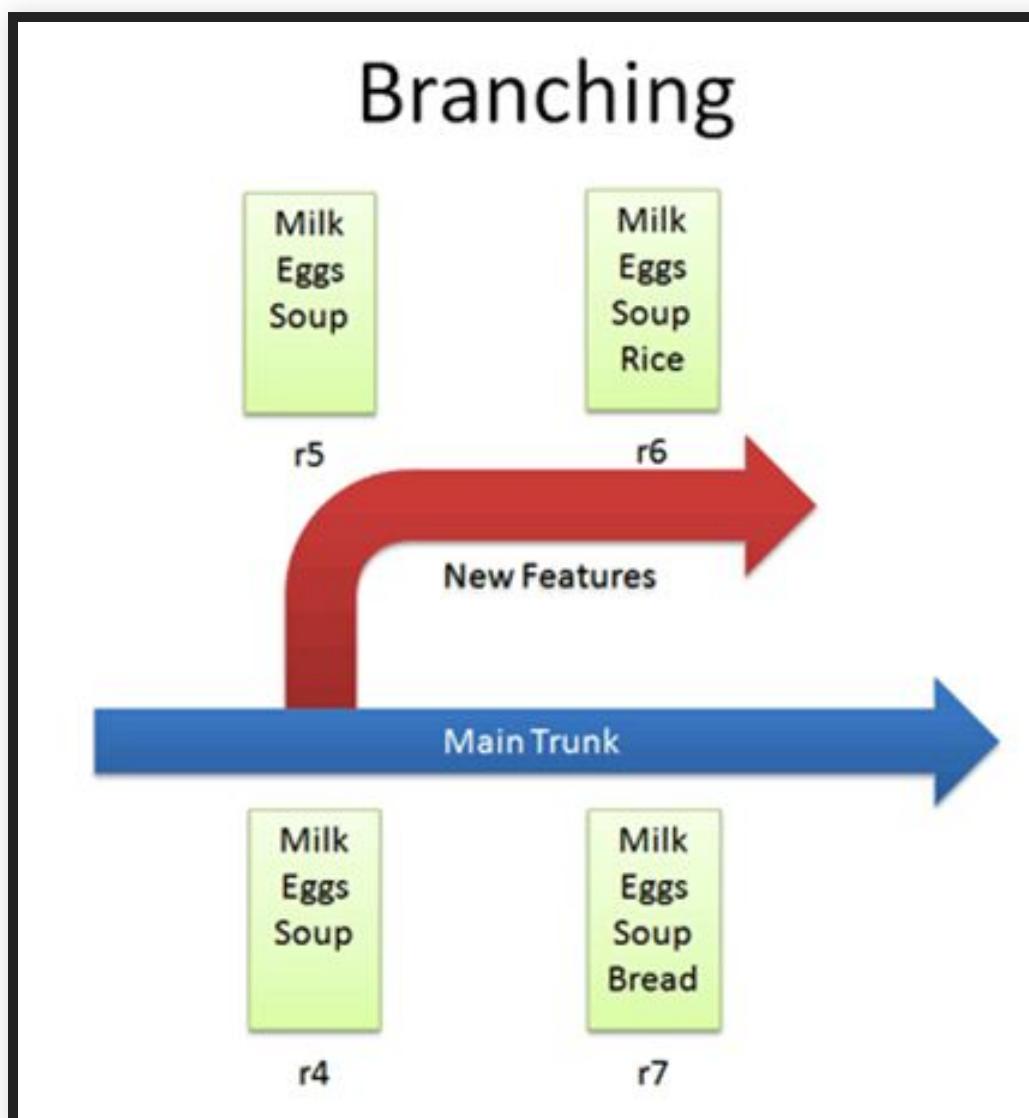
Übertragen einer Version aus dem [Repository](#) in die [Workcopy](#)

commit

Übertragen einer Version von der [Workcopy](#) in das [Repository](#)

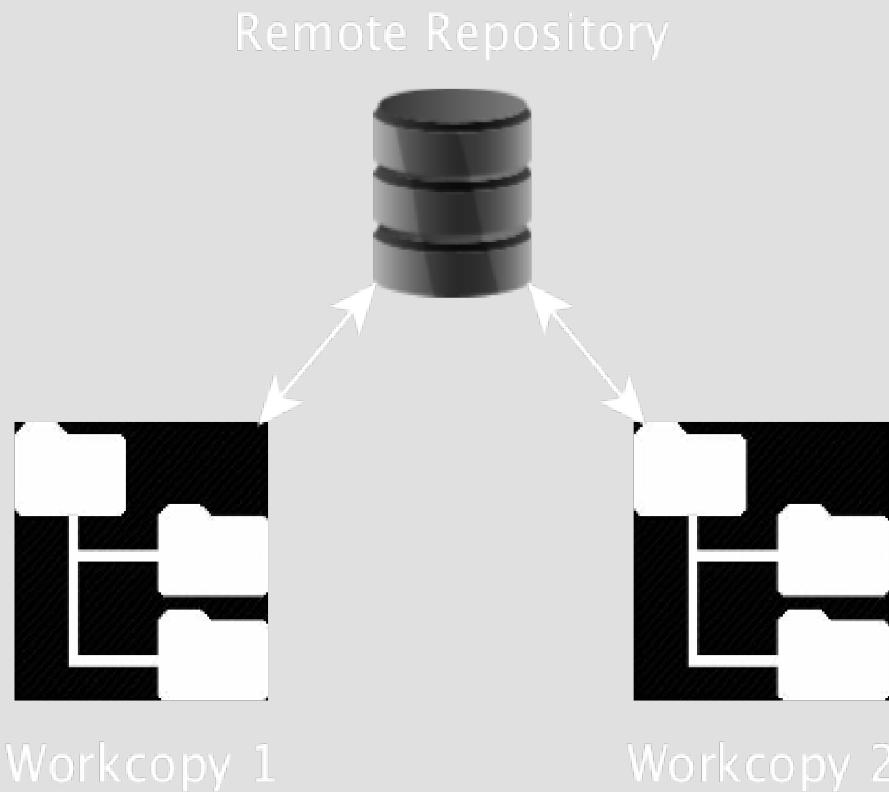
Branch

Parallel entwickelte Version

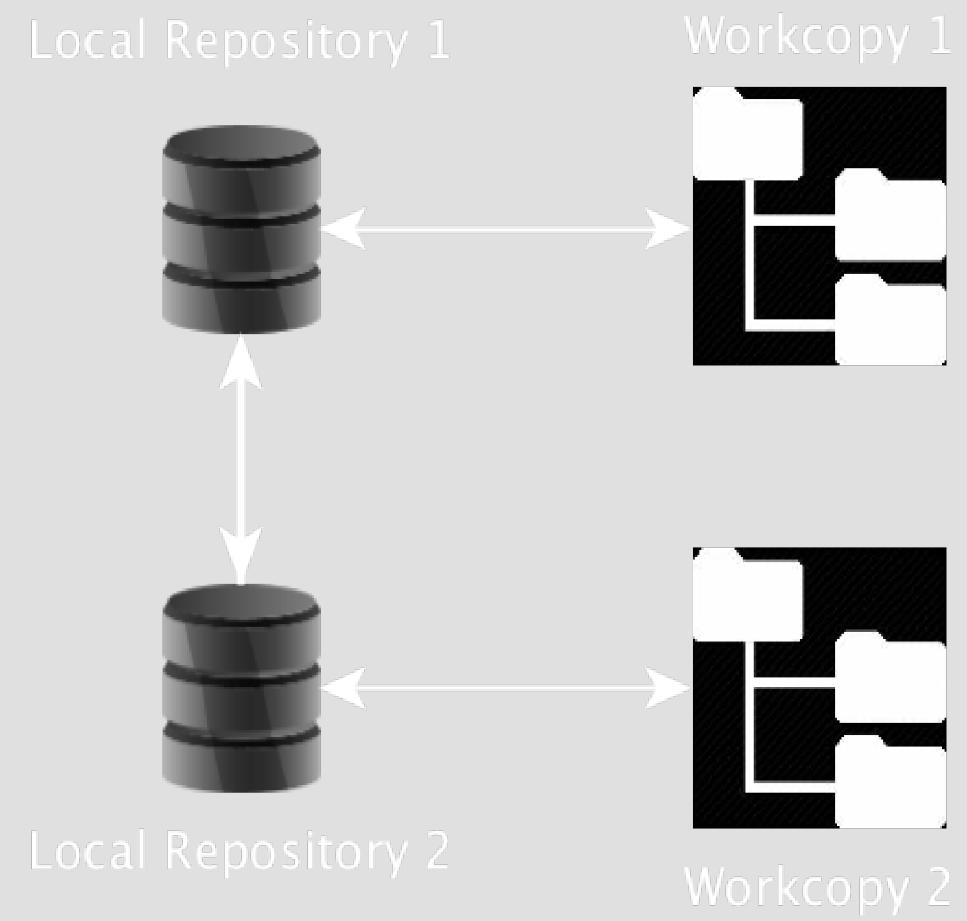


ZENTRAL VS. VERTEILT

Zentrales VCS



Dezentrales VCS



GIT

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

ZENTRAL DEZENTRALISIERT

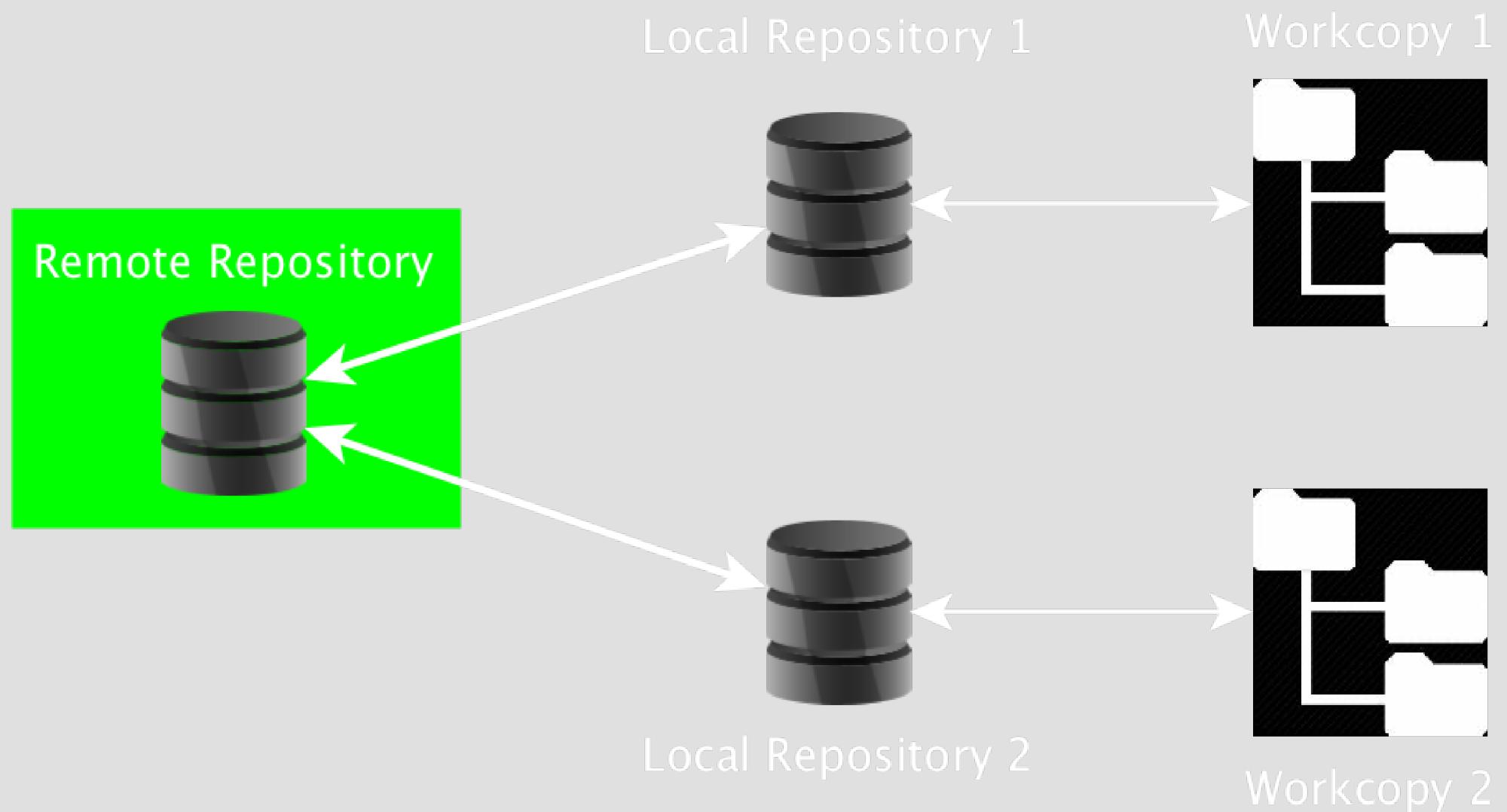
Zentral → Server
Dezentral → kein Server?

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



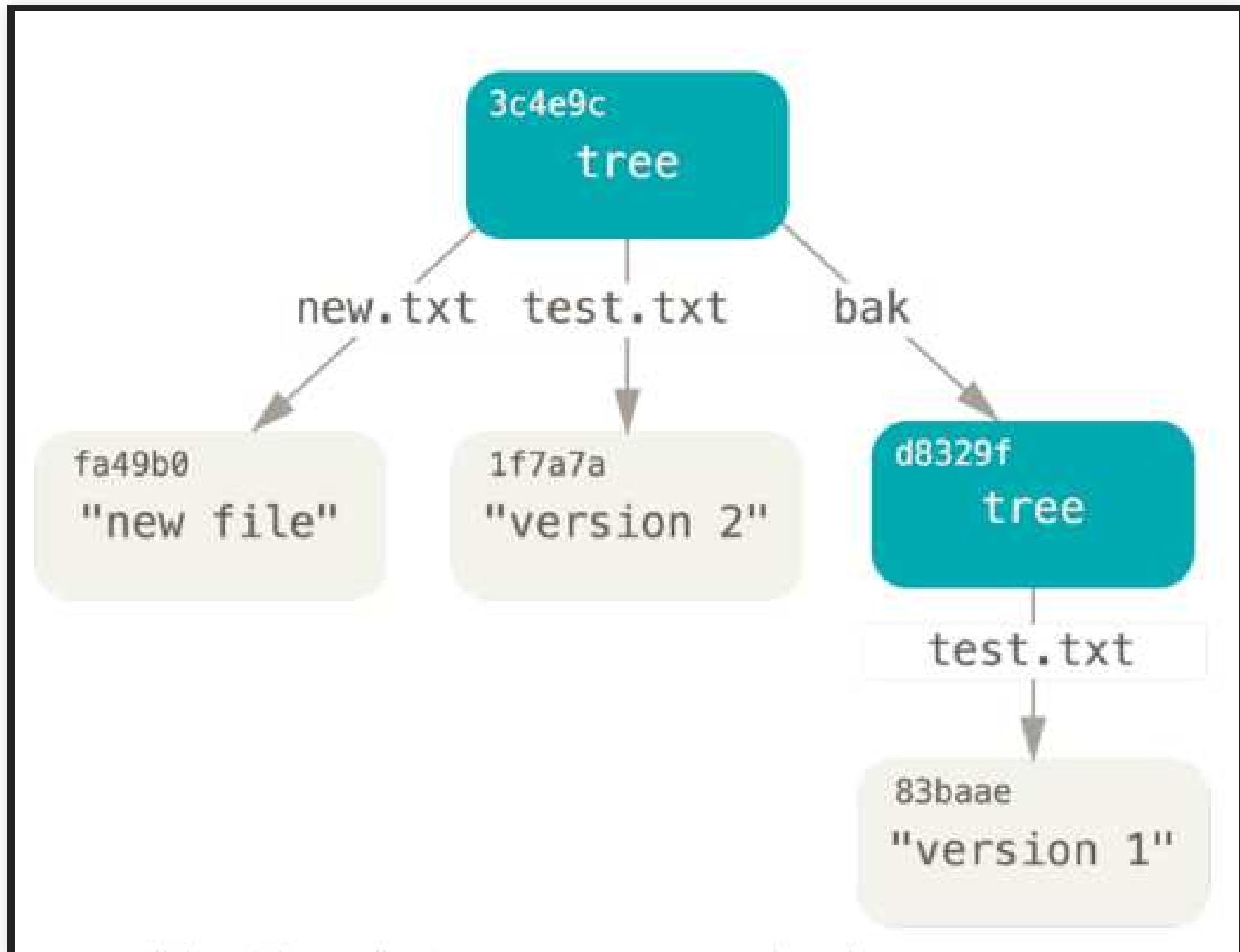
A STUPID CONTENT TRACKER

- 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

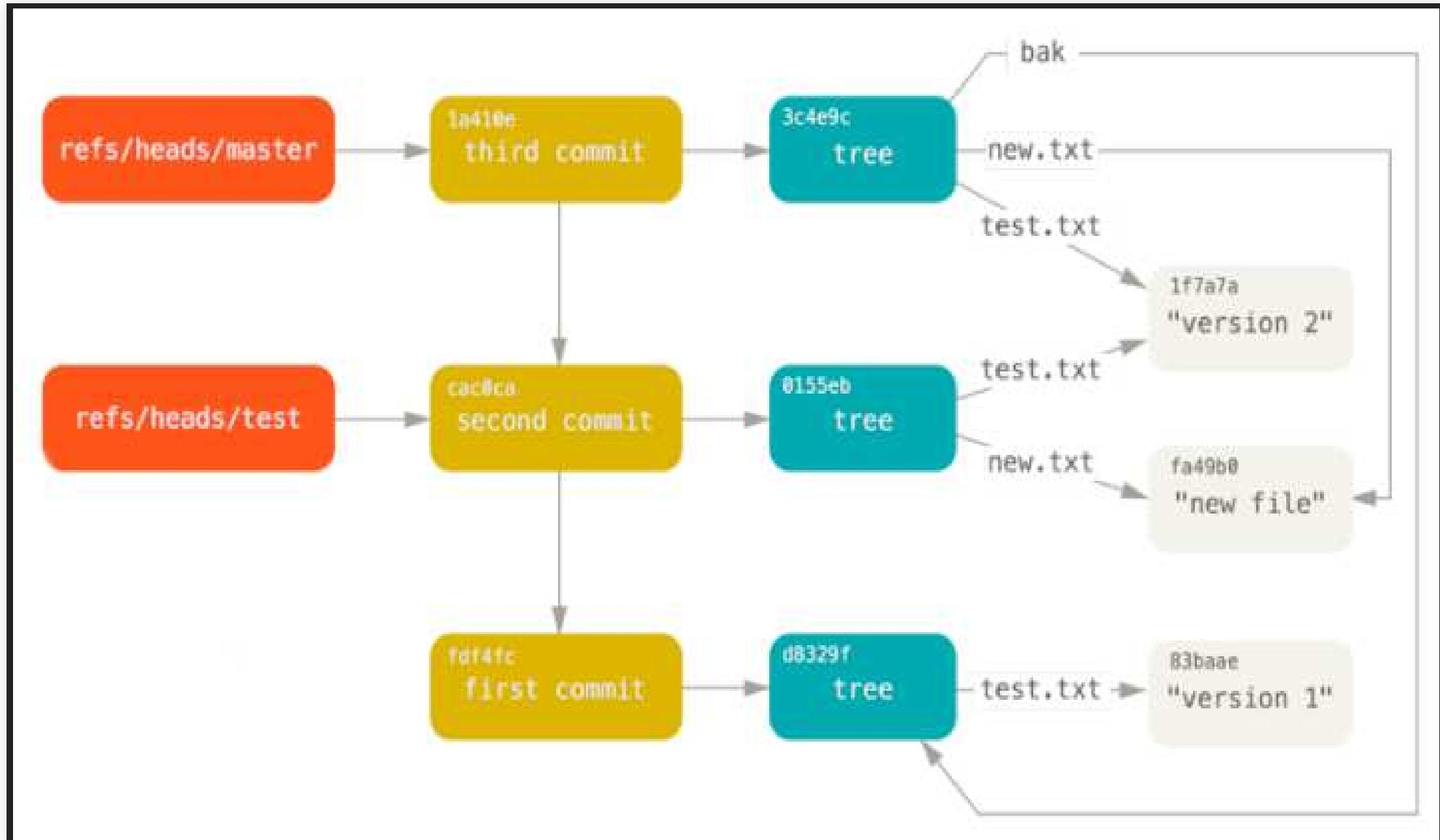
A STUPID CONTENT TRACKER

- Interne Datenstruktur 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`

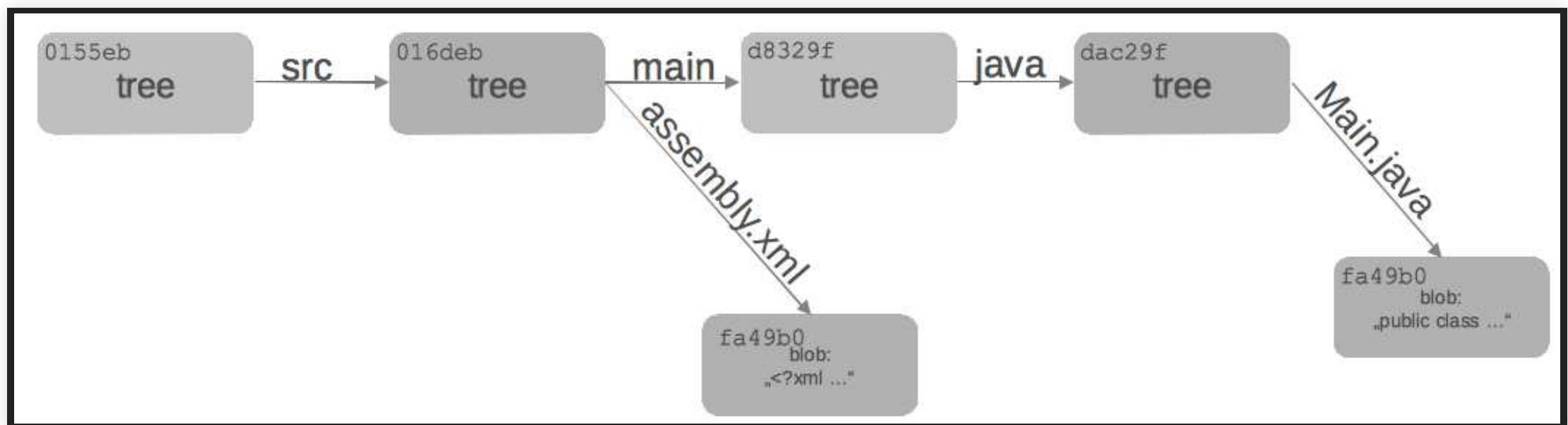
2. Porcelain

- 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

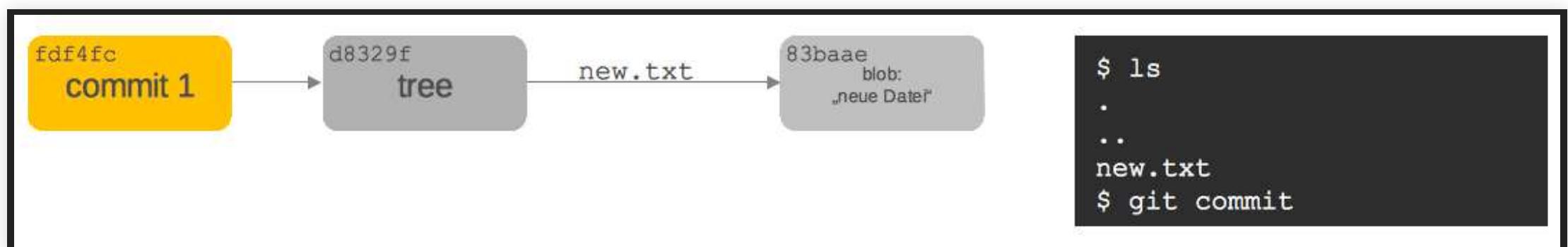
- commit-Objekt
 - eigener SHA-Schlüssel
 - SHA-Schlüssel der Vorgänger-Commits
 - SHA-Schlüssel des root-tree, der den Zustand des Projektes beschreibt
 - Commit-Nachricht
 - Author, Zeitstempel
- SHA kann oft abgekürzt werden

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

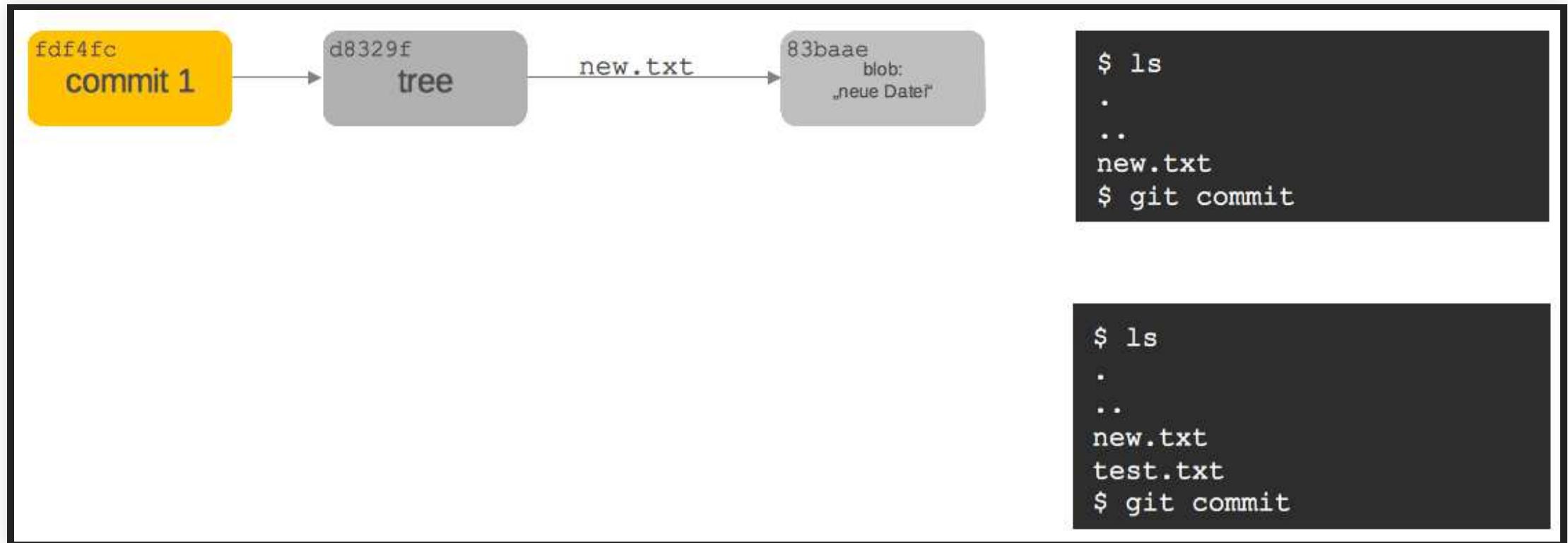
VCS FEATURES - COMMIT

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

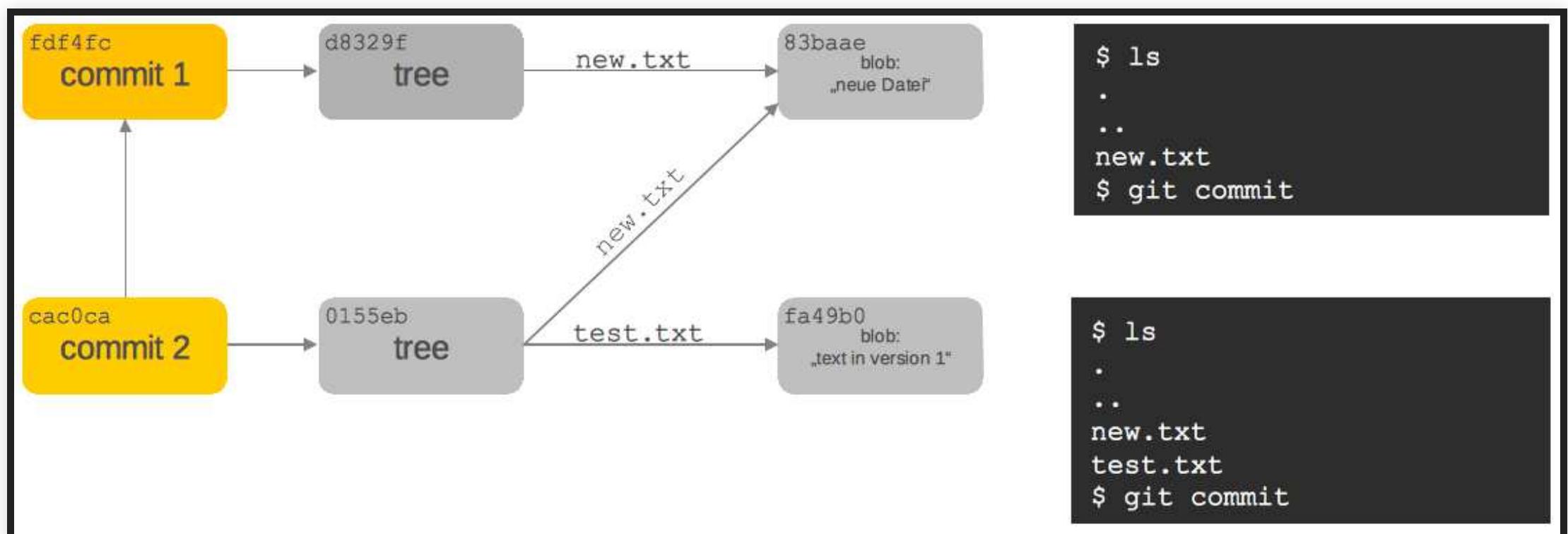
VCS FEATURES - COMMIT



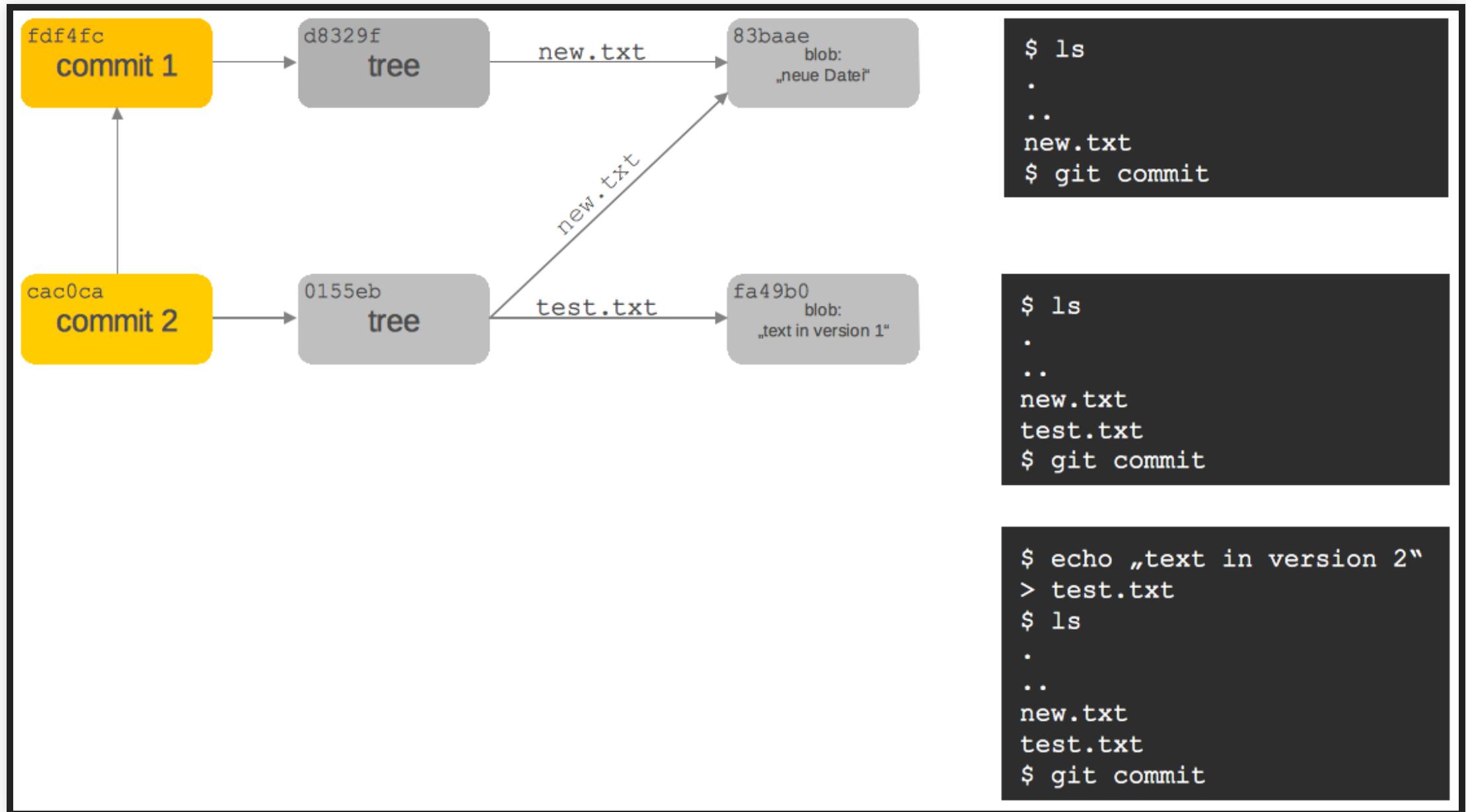
VCS FEATURES - COMMIT



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

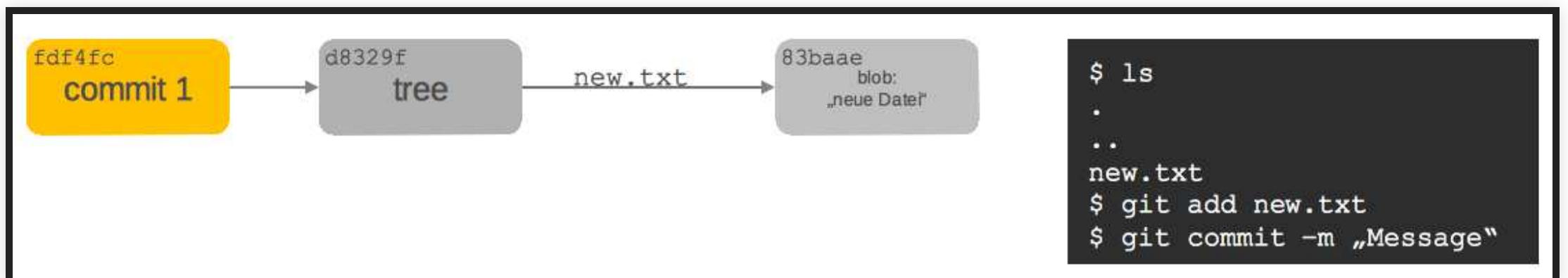
- auch als Verb: "Ich committe jetzt"

VCS FEATURES - STAGE | INDEX

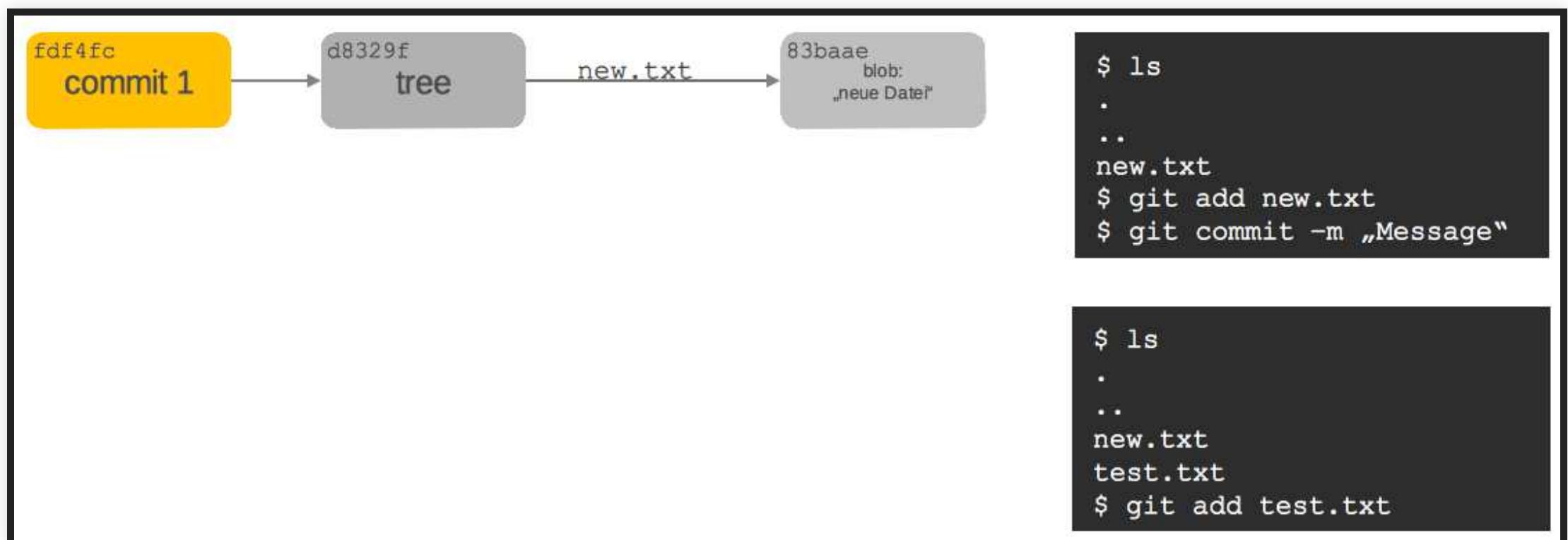
VCS FEATURES - STAGE | INDEX

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

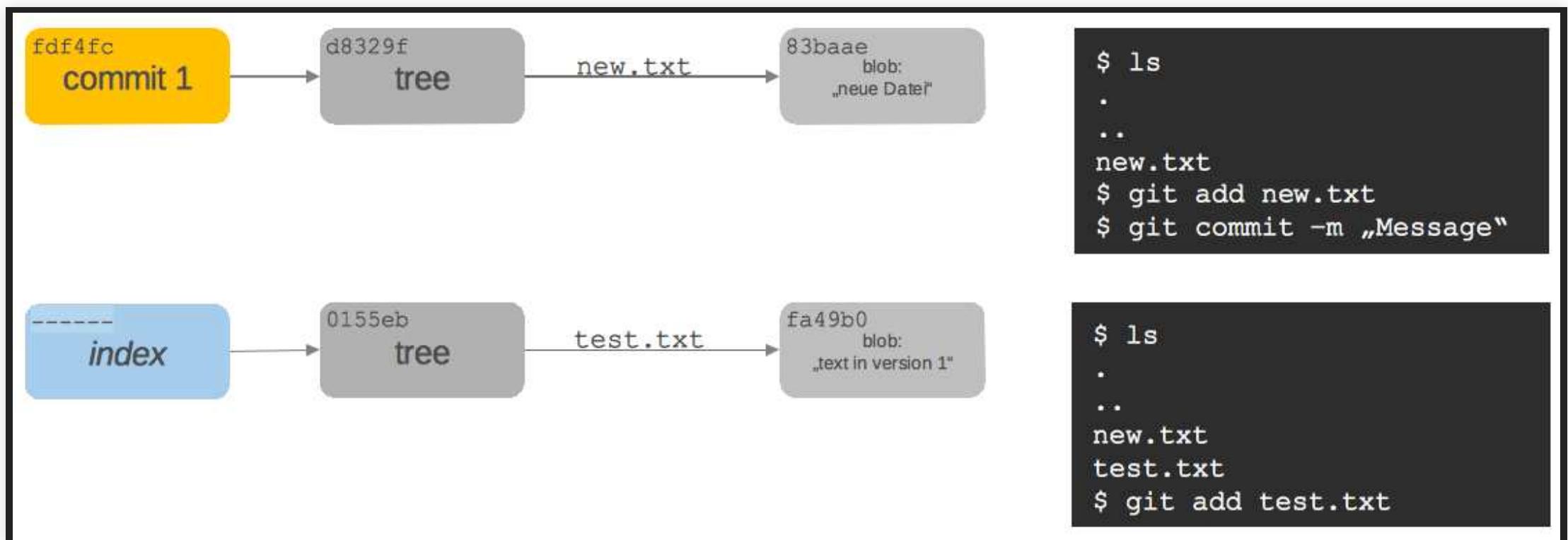
VCS FEATURES - STAGE | INDEX



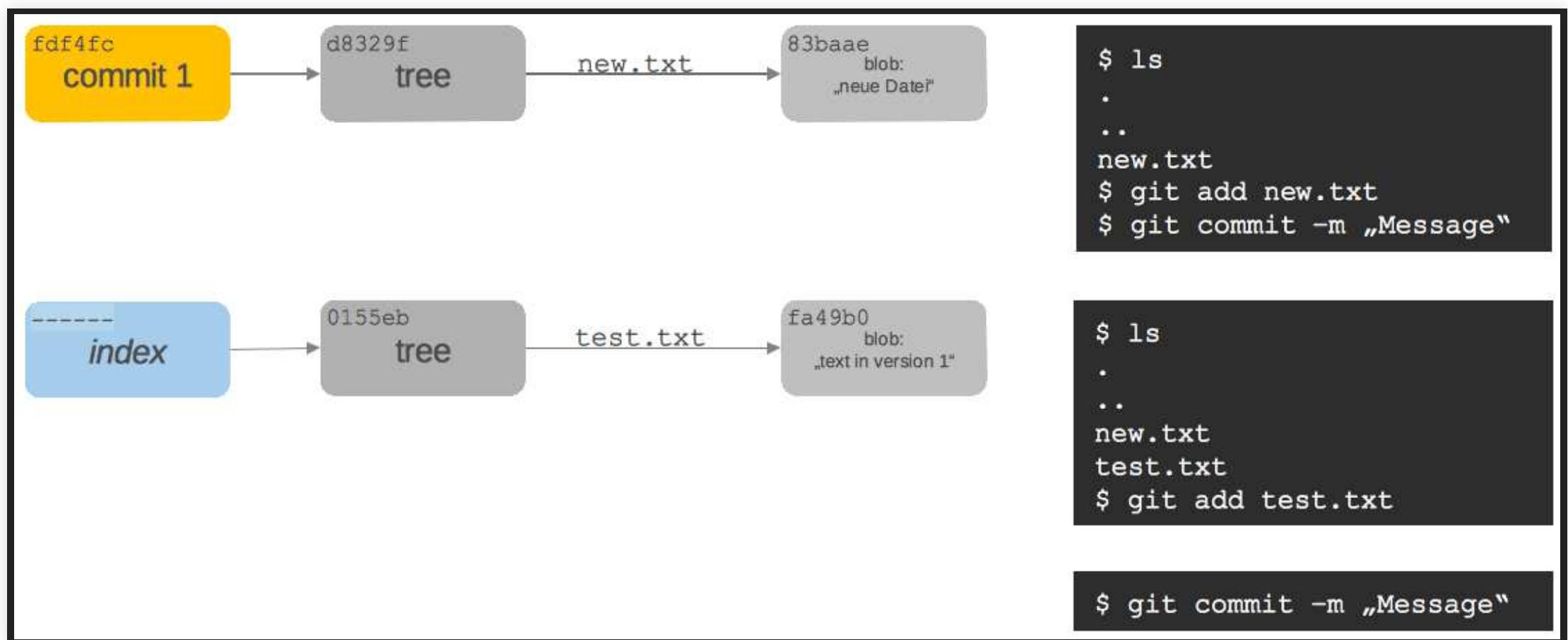
VCS FEATURES - STAGE | INDEX



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, HEAD@{1}

- 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`

HEAD, ORIG_HEAD, HEAD@{1}

- Zeiger dereferenzieren
 - (<https://git-scm.com/docs/gitrevisions>)
 - „Navigation“ von einem Commit ausgehend, z.B
 - `HEAD^` → erster Vorfahr von HEAD (unter Windows: `HEAD^^`)
 - `HEAD^1` → erster Vorfahr von HEAD
 - `HEAD^^` → zweiter Vorfahr von HEAD, ==
`HEAD^2` oder `HEAD^1^1`
 - `HEAD~3` → dritt-letzter Wert von HEAD
 - `HEAD@{2}` → zweit-letzter Wert von HEAD
 - `HEAD@{5.minutes.ago}` → Wert von HEAD vor 5 Minuten

ÄÄNDERUNGEN 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-Bereichs von foo.txt verwerfen
$ git reset HEAD foo.txt
## Alle Änderungen im Stage-Bereichs verwerfen
## (Workcopy bleibt unverändert)
$ git reset HEAD
## Alle Änderungen im Stage-Bereichs & 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`
 - Schöneres Anzeigen
 - `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 %ad %s]'"
```

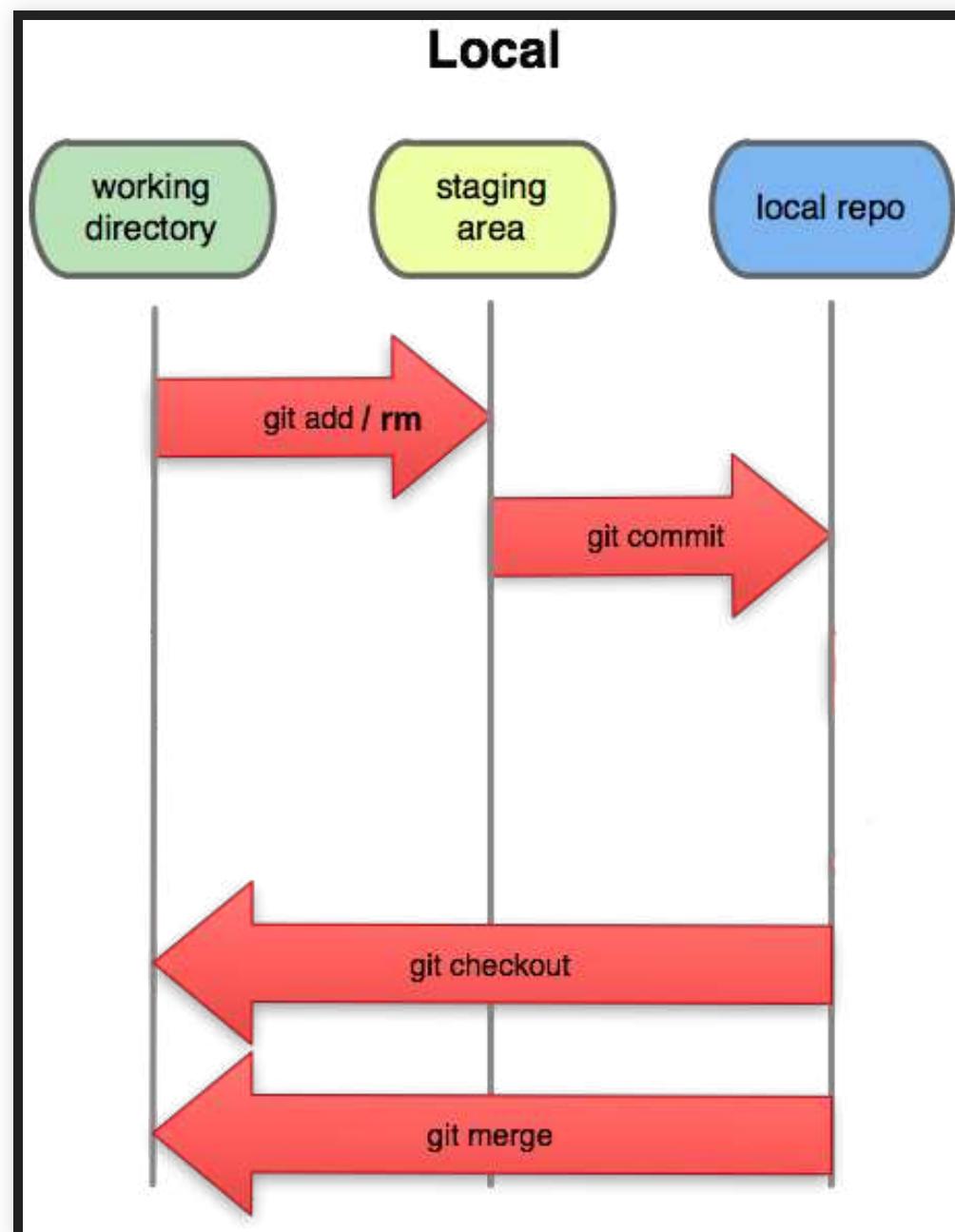
KOMMANDOZEILE

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

VCS 2

REKAPITULATION

GIT KOMMANDOS



GIT KOMMANDOS

1. Interaktion zwischem lokalem Repository und WorkCopy

- `git checkout master`
- `git add README.md`
- `git rm README.md`
- `git commit -m "Neuer Code"`
- `git checkout – README.md`
- `git reset --hard HEAD`

GIT KOMMANDOS 2

1. Hilfe

- `git --help`
- `git commit --help`
- <https://git-scm.com/docs>

2. Graphische Darstellung

- `gitk`
- `log --all --decorate --oneline --graph`
- SourceTree, Fork, GitKraken

GIT SPEICHER

Commit

- Enthält Verweise auf alle Dateien zu einem bestimmten Zeitpunkt
- Enthält einen Verweis auf den Vorgänger-Commit

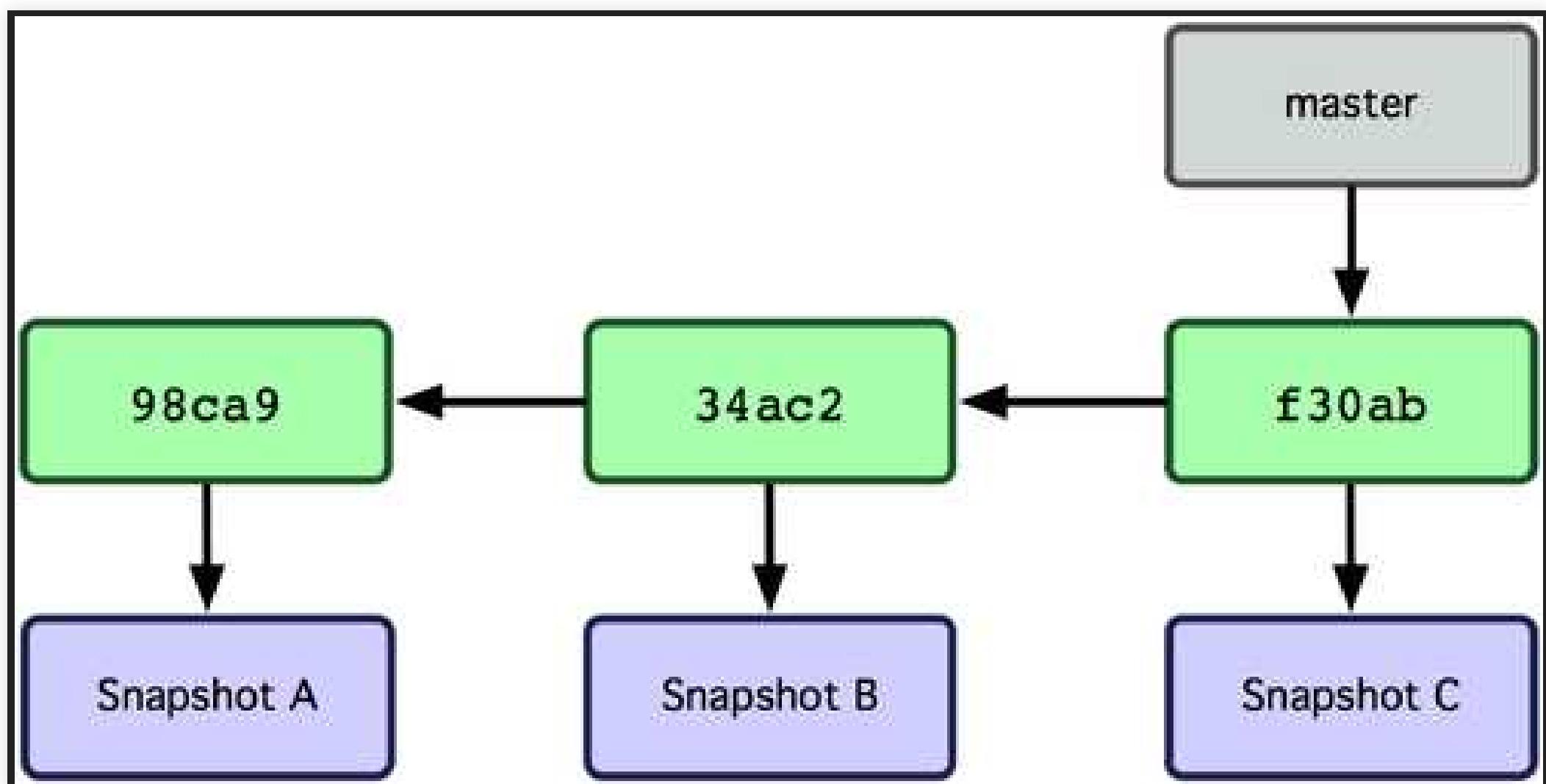
BRANCH

BRANCHING

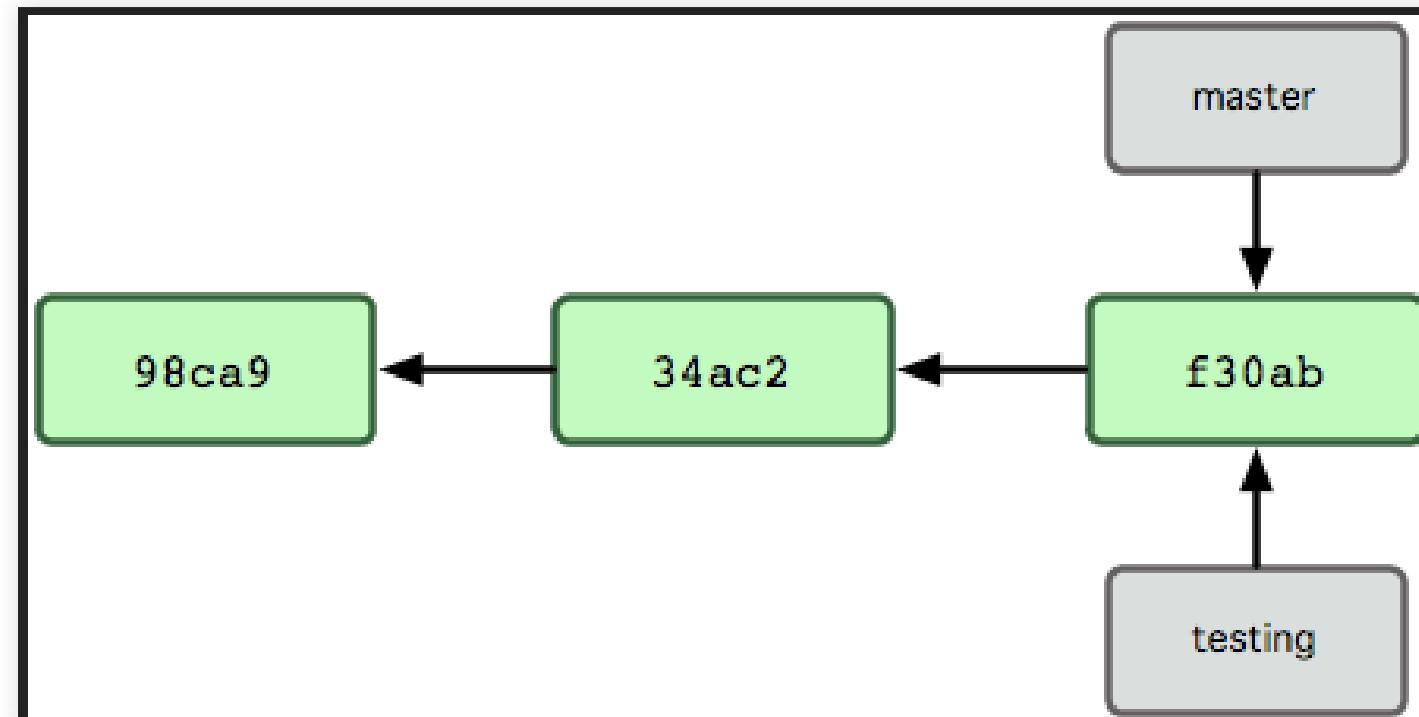
Branching means you diverge from the main line of development and continue to do work without messing with that main line.

BRANCH

- Branch **master** zeigt momentan auf neuesten Commit



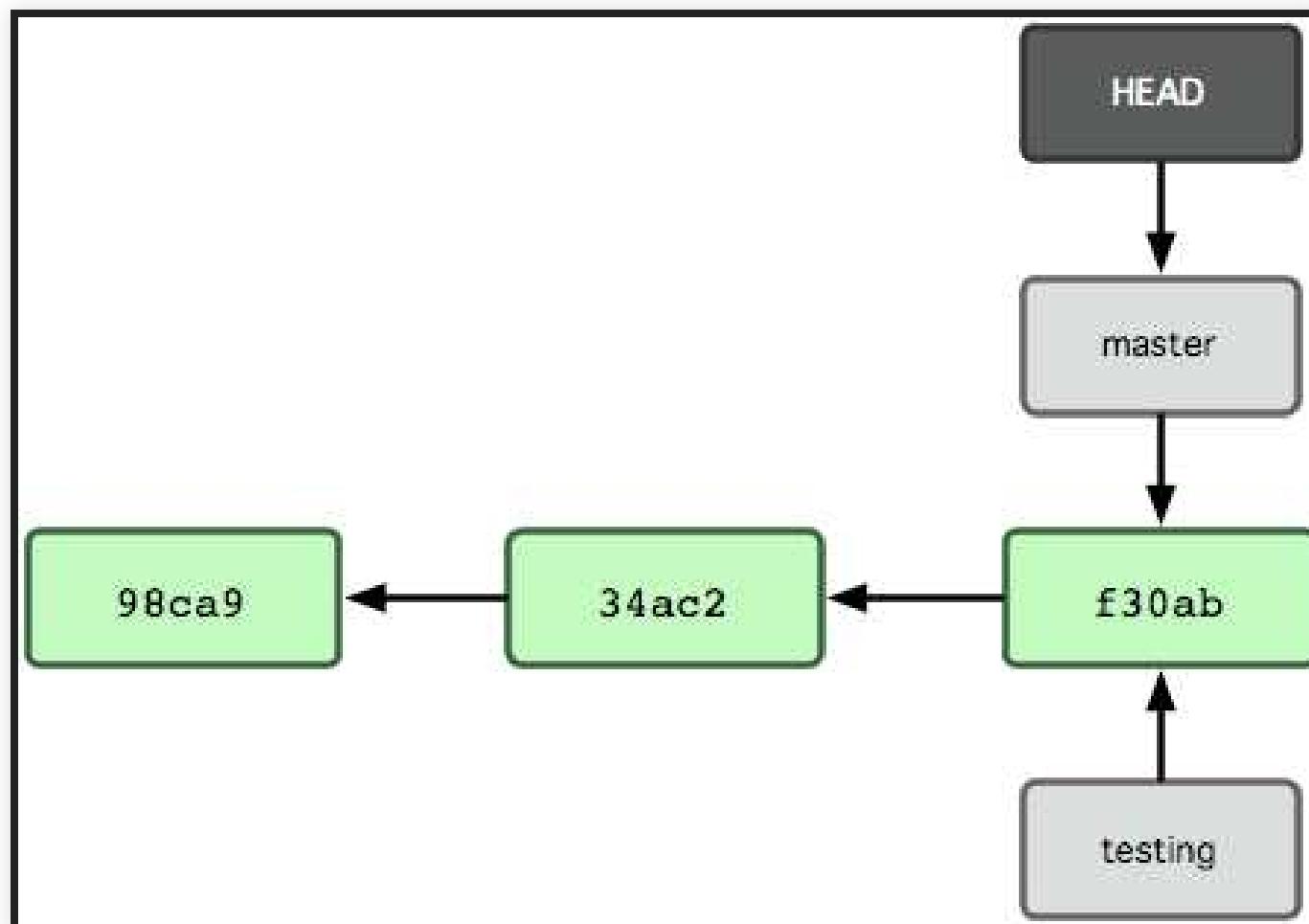
```
## erstellt einen neuen Branch, der auf den gleichen Commit
## wie master zeigt
$ git branch testing master
## erstellt einen neuen Branch, der auf den gleichen Commit
## wie HEAD zeigt
$ git branch testing HEAD
## erstellt einen neuen Branch, der auf den gleichen Commit
## wie HEAD zeigt
$ git branch testing
## erstellt einen neuen Branch, der auf den Commit 23c4fe5 zeigt
$ git branch 23c4fe5
```



HEAD

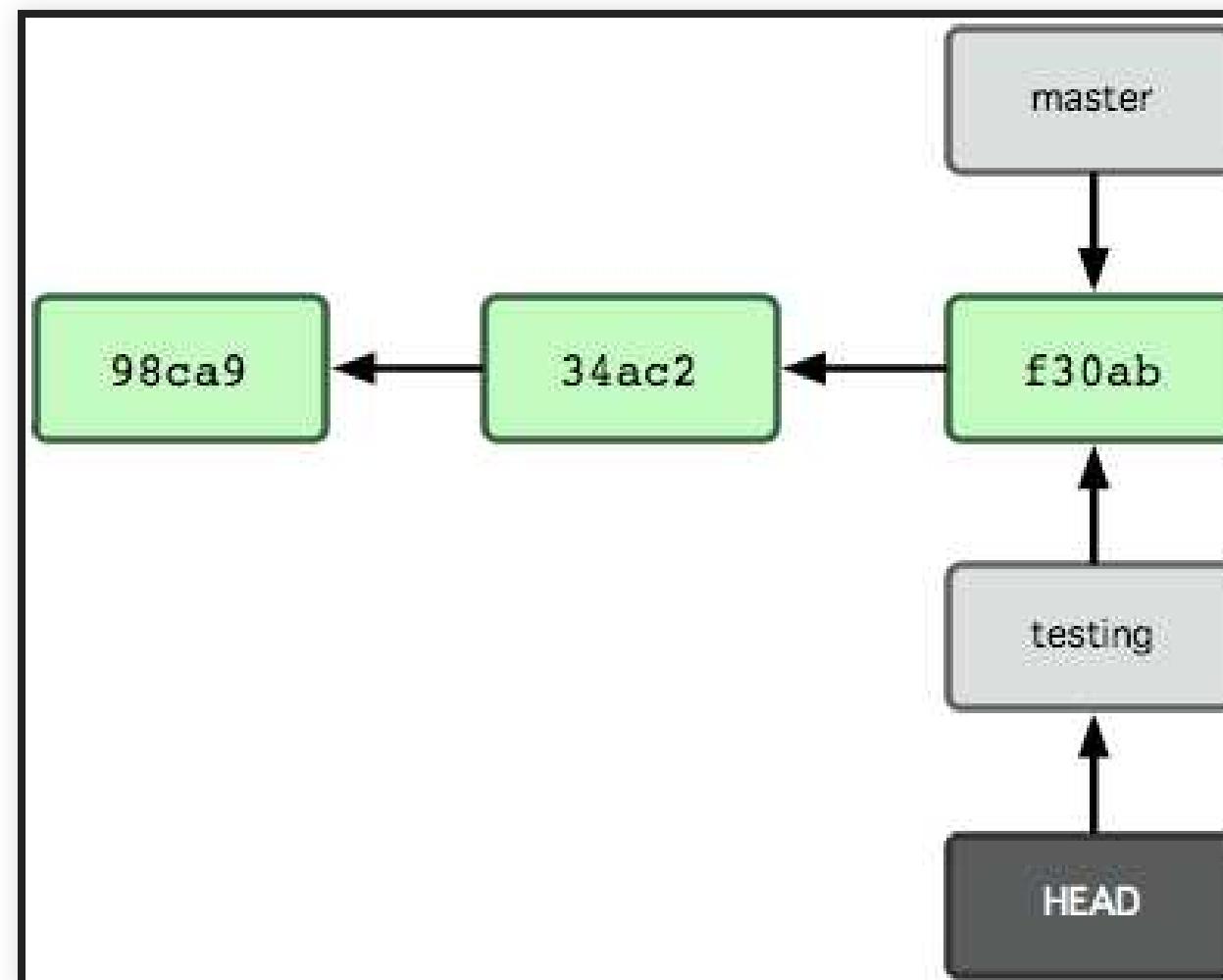
HEAD ist ein spezieller Zeiger

- zeigt auf den Branch, mit dem die Workcopy verbunden ist



Auschecken (aktivieren) eines anderen Branches

```
## aktiviert einen bereits bestehenden Branch  
$ git checkout testing  
## erzeugt und aktiviert einen neuen Branch, der auf den gleichen  
## Commit wie HEAD zeigt  
$ git checkout -b testing
```



(GIT) BRANCH

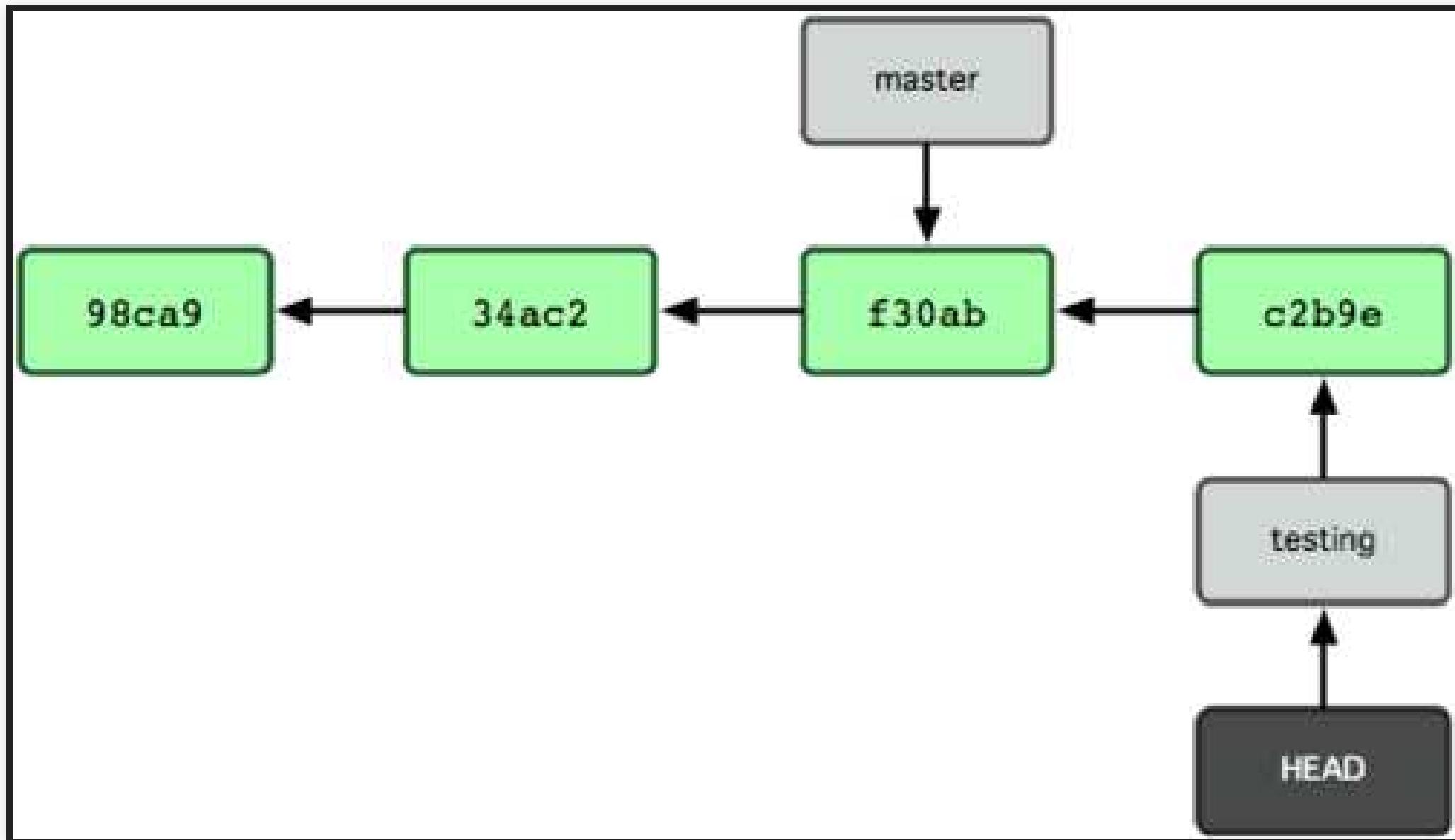
- == Referenz
 - ein (beweglicher) Zeiger auf einen Commit
 - bei `git commit` wird der Zeiger weitergeschoben
 - genauer:
 - der Branch, auf den `HEAD` zeigt, verweist nach dem Committen auf den neuen Commit verschoben
 - `HEAD` zeigt weiter auf diesen Branch
 - im RefLog wird der neue (effektive) Wert von `HEAD` protokolliert

Anzeige aller Branches

```
$ git branch
  feature-24
* master
  my-branch-1
  my-branch-2
$ git status
On branch master
...
...
```

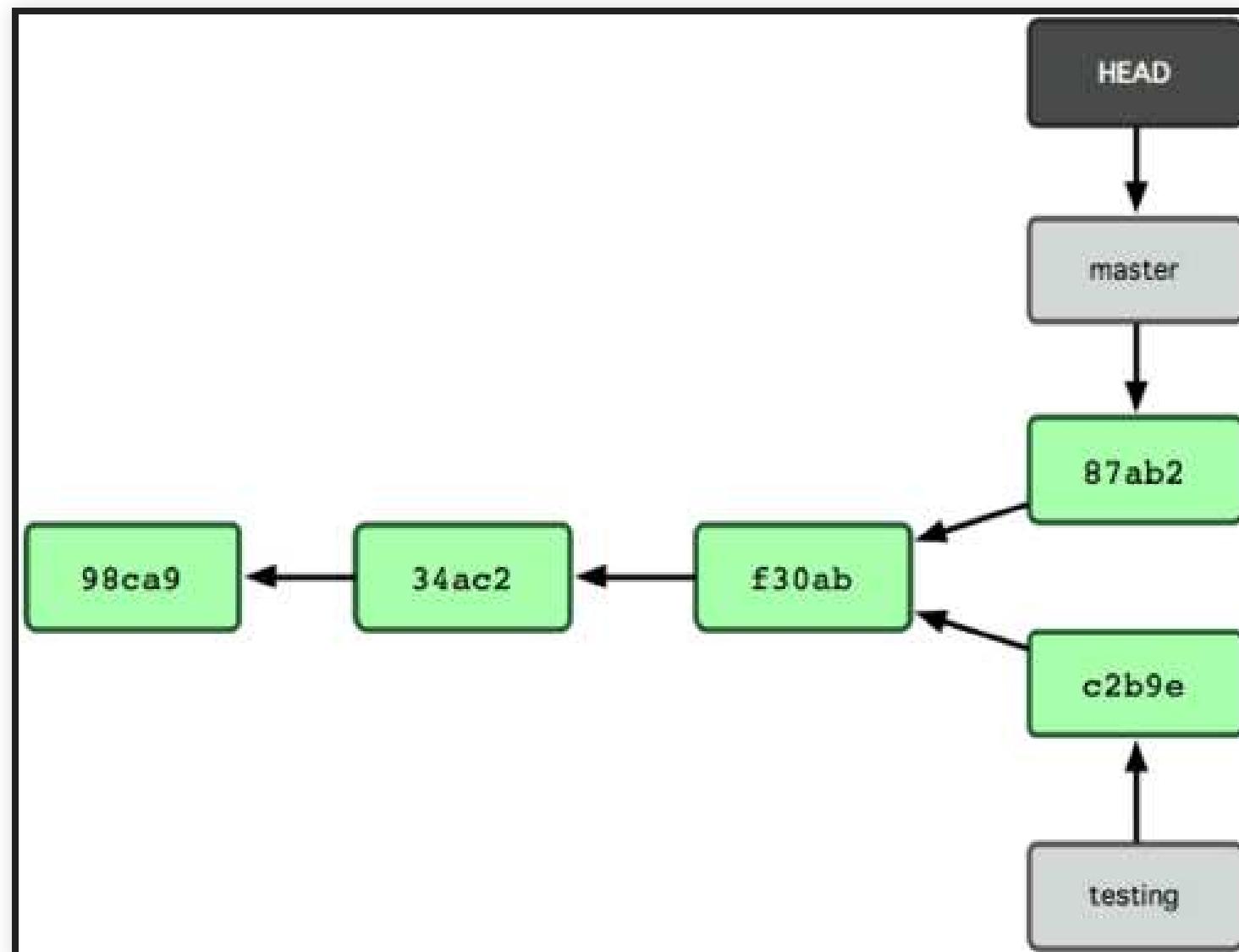
Ein weiterer Commit ...

```
$ echo 'neuer Text' > neue-datei.txt  
$ git commit -a -m 'Neue Datei auf branch testing'
```

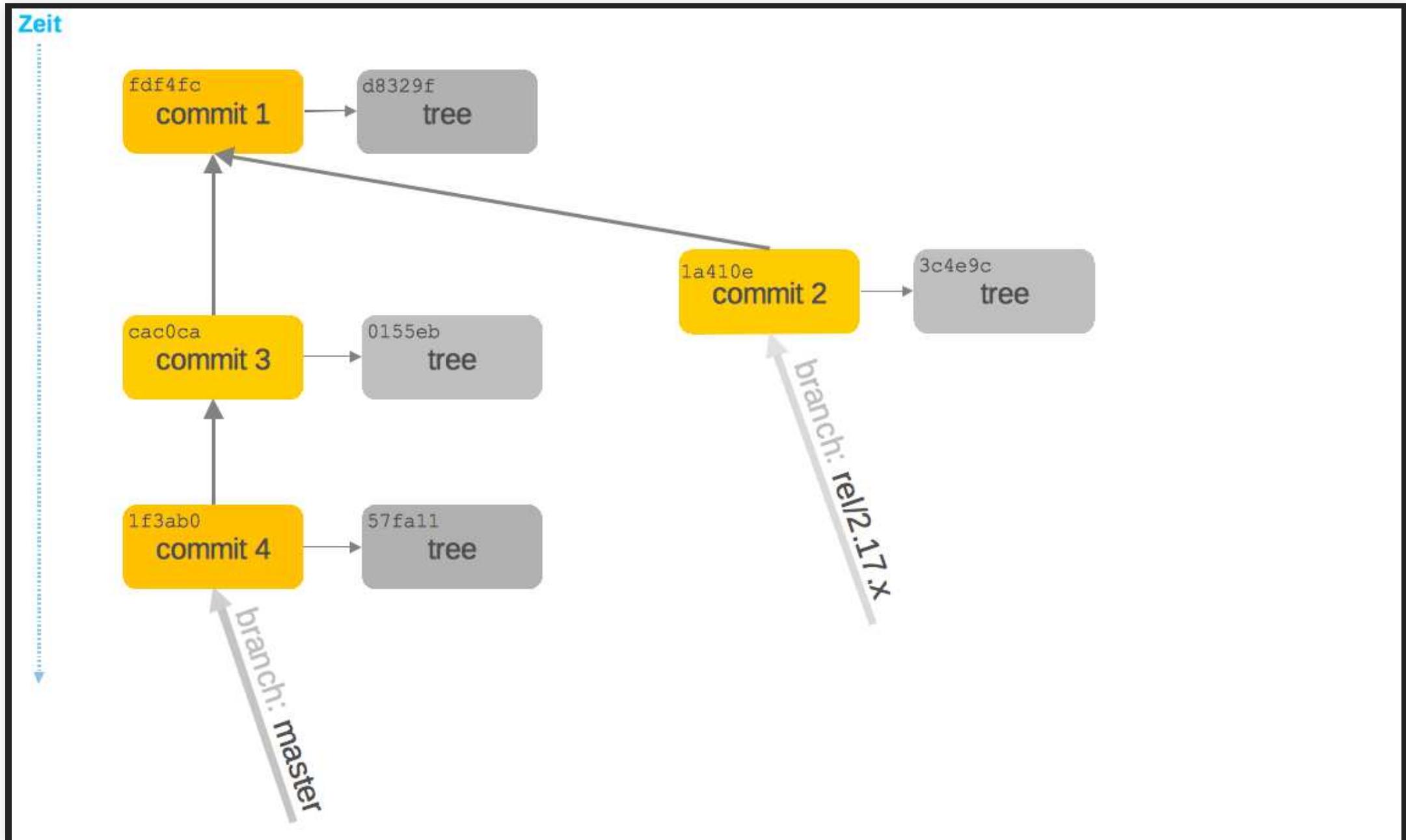


Die Historie läuft auseinander

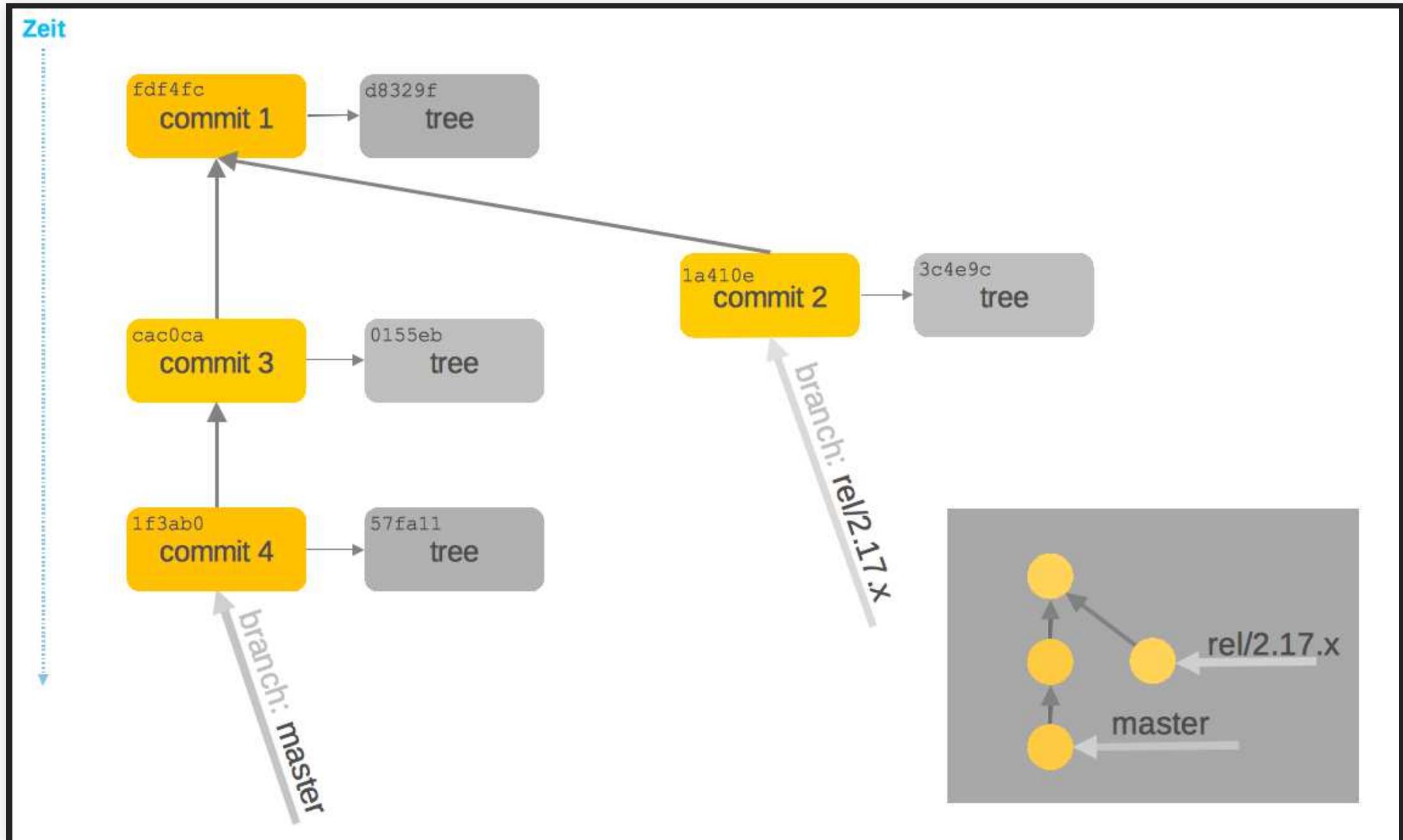
```
$ git co master
$ echo 'neuer anderer Text' > neue-datei-auf-master.txt
$ git commit -a -m 'Neue Datei auf branch master'
```



BRANCH SICHTWEISEN



BRANCH SICHTWEISEN



TAG

- eine *dauerhafte* Markierung / Kennzeichnung
- unveränderlich
- zum *Merken* von bestimmten Zwischenständen

TAG

1. Leichtgewichtiger Tag

- == Referenz (genau wie ein Branch)

2. Annotated Tag

- eigener Objekt-Typ im Git Datenmodell, enthält
 - SHA eines Commits
 - Datum & Author
 - Nachricht
 - ggf. PGP Signatur

TAG ERZEUGEN

```
## leichtgewichtigen Tag erstellen
$ git tag test-tag-1
## alle Tags anzeigen
$ git tag
release-1
release-2
test-tag-1
## annotated Tag erstellen
$ git tag -a -m "Noch ein Test tag" test-tag-2
## Alle Tags inkl. Message anzeigen
$ git tag -n
release-1  Commit-Message ...
release-2  Commit-Message ...
test-tag-1 Commit-Message ...
test-tag-2  Noch ein Test tag
```

REFERENZEN

1. reference

- eine Datei
- Dateiname entspricht dem Namen der Referenz
- Inhalt ist der SHA des Commits, auf den die Referenz verweist

2. symbolic reference

- eine Datei
- Dateiname entspricht dem Namen der Referenz
- Inhalt ist der Name einer anderen Referenz
- eigentlich gibt es hier nur **HEAD**

3. ORIG_HEAD, FETCH_HEAD sind Sonderfälle

REFERENZEN

```
## Auflistung aller Dateien im Ordner .git/refs
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/heads/master
.git/refs/heads/my-branch-1
.git/refs/tags
.git/refs/tags/test-tag-0
.git/refs/tags/test-tag-1
.git/refs/remotes
.git/refs/remotes/origin
.git/refs/remotes/origin/master
```

SYMBOLISCHE REFERENZEN

```
## Ausgabe des Inhalts der Datei .git/HEAD
$ cat .git/HEAD
ref: refs/heads/master
```

1. Branches == Referenzen, die unter `.git/refs/heads` gespeichert werden
2. Tags == Referenzen, die unter `.git/refs/tags` gespeichert werden
 - nur lightweight Tags
3. Ref-Log
 - Protokoll für alle Änderungen, die an den Referenzen gemacht wurden (nur lokal)

GUT ZU WISSEN

- die meisten Git Kommandos haben mind. einen Parameter, der eine Commit ID (SHA) ist

```
## der ganze SHA
$ git show a544751ae3de9965c35b88958b0d219e29f7295d
## der abgekürzte SHA
$ git show a54475
## eine Referenz
$ git show master
## eine symbolische Referenz
$ git show HEAD
## default ist immer HEAD
$ git show
```

REFLOG

```
## zeigt die Historie von HEAD
$ git reflog
## zeigt den 5. letzten Commit beginnend bei HEAD
$ git show HEAD@{5}
## zeigt den letzten Commit von gestern
$ git show HEAD@{yesterday}
## zeigt die Logausgaben mit der reflog Syntax
$ git log -g
```

STASHING

- Verstecken der aktuellen Änderungen
 1. alle Änderungen an der Workcopy
 2. alles im Stage-Bereich (Index)
- Workspace und Stage-Bereich sind danach wieder auf dem Stand des letzten Commits (siehe HEAD)
- Neue Dateien (untracked) werden per default ignoriert
- Man kann unzählig viele Stashes anlegen

STASH KOMMANDOS

```
## Änderungen auf Stash-Stack verschieben
$ git stash
## Änderungen in benannten Stash verschieben
$ git stash push -m „mein zweiter Stash“
## Alle Stashes auflisten
$ git stash list
## Stash Nr 0 auf den aktuellen Workspace anwenden,
## aber Stash nicht löschen
$ git stash apply stash@{0}
## Stash Nr 0 auf den aktuellen Workspace anwenden
## und Stash von Stack löschen
$ git stash pop stash@{0}
```

Painground

PEOPLES KNOWLEDGE

- Verwalten der Profile von Mitarbeiter / Studenten /
...
- Welche Fähigkeiten hat jemand, z.B.
 - Excel
 - Java
 - Matlab
- Soll Kollegen unterstützen, wenn sie Hilfe suchen

USER STORIES

STORY 1

Ich möchte für alle (meine) Mitarbeiter ein Profil anlegen: dies soll den Namen, den Geburtstag und ein Bild beinhalten.

STORY 2

Ich möchte, dass sich jeder Mitarbeiter mit Namen anmelden kann und nur sein eigenes Profile bearbeiten kann.

STORY 3

Ich möchte, dass das Profile auch Felder für die private Adresse, private Telefonnummer und Hobbies enthält.

STORY 4

Die PersonalAbteilung will die Skills jedes Mitarbeiters per REST-Schnittstelle in ihr eigenes System importieren.

STORY 5

Ich möchte, dass ein Mitarbeiter in seinem Profi seine "Skills" pflegen kann, in ein Textfeld - ein Wort == ein skil

STORY 6

*Ich möchte, dass die Skills gewichtet werden können - Gewichtungen:
Novize, Erfahrener, Experte*

STORY 7

Ich möchte, dass beim Erfassen eines weiteren Skills an einem Profile eine Liste an passenden - bereits erfassten Skills angezeigt werden soll (suggest)

STORY 8

Ich möchte eine Liste aller erfasster Skills sehen, dazu die Anzahl der Mitarbeiter, die dieses Skill "haben"

STORY 9

Ich möchte, dass ein Mitarbeiter den Skill (inkl. dessen Gewichtung) eines Kollegen bestätigen kann. Im Profil des Mitarbeiters kann man dann sehen, wie viele "Kollegen" den Skill & Bewertung bestätigt haben.

STORY 10

Ich möchte, dass ein Mitarbeiter seine Bewertung eines Skills ändern kann - diese Änderung soll mit Zeitstempel gespeichert werden sodass später die "Entwicklung" dieses Skills beobachtet werden kann.

STORY 11

Ich möchte, dass die Bewertungen der Kollegen ebenfalls mit Zeitstempel gespeichert werden - so kann man später eine "Entwicklung" der Bewertungen & Bestätigungen aufzeigen.

VCS 3

MERGEN

Doppelbedeutung `merge`

1. Zusammenfügen von zwei Dateiversionen

- ggf. Auflösen von Konflikten

2. Zusammenfügen von zwei (oder mehr) Branches

- alle Änderungen des einen Branches werden auf den andern Branch übertragen

- Beides wird mit `git merge` angestoßen
- Begriff: `gemeinsamer Vorfahr` (most recent common ancestor) Commit
 - `git merge-base` Befehl hilft dabei
- Falls automatische Konfliktlösung nicht funktioniert, muss der User eingreifen

DATEIKONFLIKTE

Git kann die meisten Dateikonflikte automatisch auflösen

```
$ cat datei.txt # Original  
Einfacher Text  
in zwei Zeilen
```

```
$ cat datei.txt # von Branch 1  
Einfacher Text der  
in zwei Zeilen
```

```
$ cat datei.txt # von Branch 2  
Einfacher Text  
in drei Zeilen  
steht
```

```
$ cat datei.txt # nach dem Merge  
Einfacher Text der  
in drei Zeilen  
steht
```

MANUELLE DATEIKONFLIKTE

Wenn Git einen Konflikt nicht automatisch auflösen kann, muss man manuell auflösen:

```
$ git merge branch-1
CONFLICT (content): Merge conflict in datei.txt
Automatic merge failed; fix conflicts and then commit the result.
$ 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:      datei.txt

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

MANUELLE DATEIKONFLIKTE

- Git ändert die Datei und speichert alle nicht-gemergten Stellen innerhalb der Datei ab.
 - mit sog. Standard Konflikt-Markern
- `git add datei.txt` Markiert die Datei als "Konflikte gelöst"
- `git merge --abort` Versetz alles in den Zustand vorher
- `git mergetool` kann eine bessere Sicht auf die Konflikte liefern

MANUELLE DATEIKONFLIKTE

```
$ cat datei.txt # Original  
Einfacher Text  
in zwei Zeilen
```

```
$ cat datei.txt # von Branch 1  
Einfacher Text  
in drei Zeilen
```

```
$ cat datei.txt # von Branch 2  
Einfacher Text  
in vier Zeilen
```

```
$ cat datei.txt # nach dem Merge  
Einfacher Text der  
<<<<< HEAD  
in drei Zeilen  
=====  
in vier Zeilen  
>>>>> branch-2
```

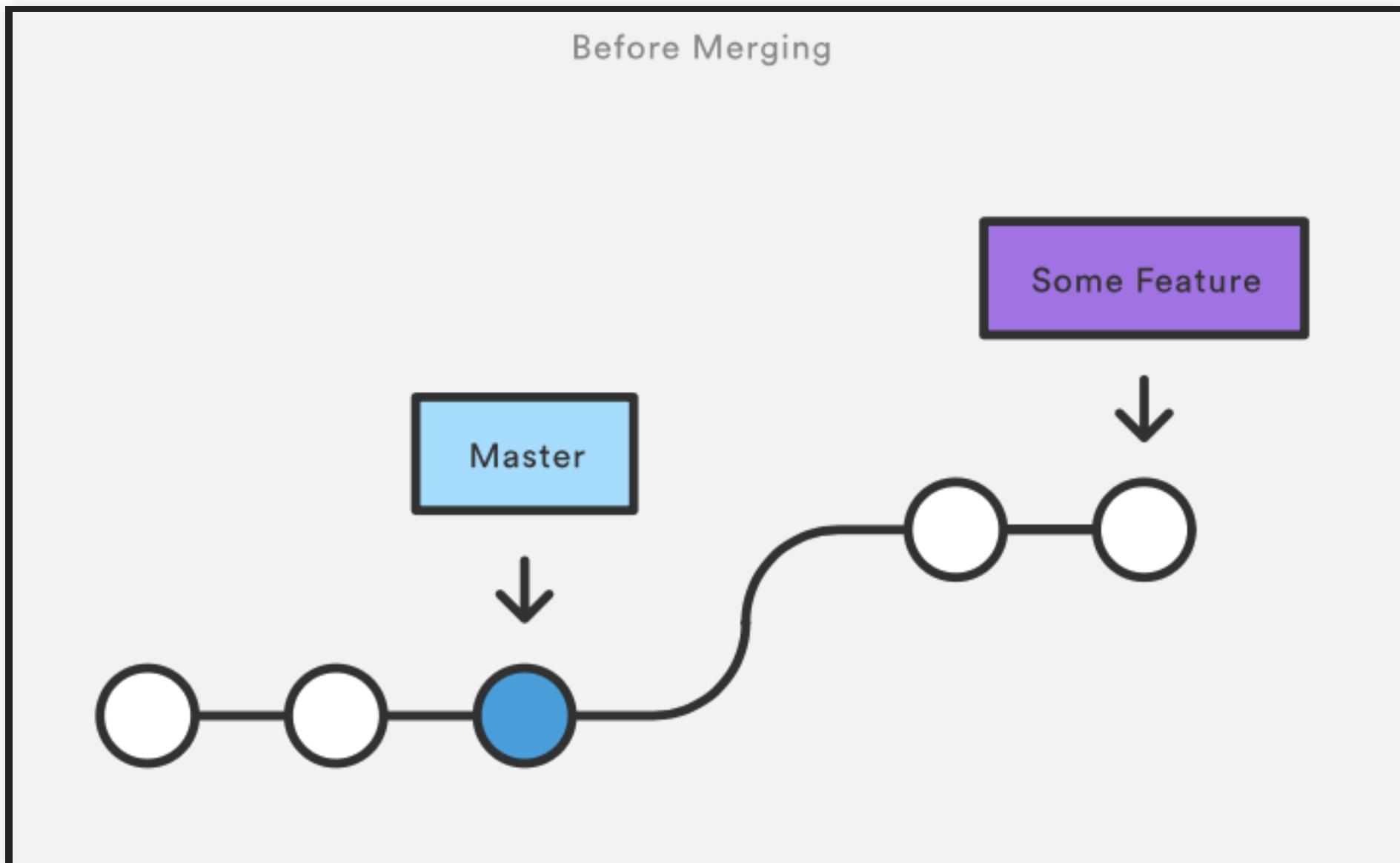
MERGE VON BRANCHES

Zusammenfügen von zwei (oder mehr) Branches

- alle Änderungen des einen Branches werden auf den andern Branch übertragen
- zwei Arten
 1. 3-Way-Merge
 2. Fast-Forward-Merge

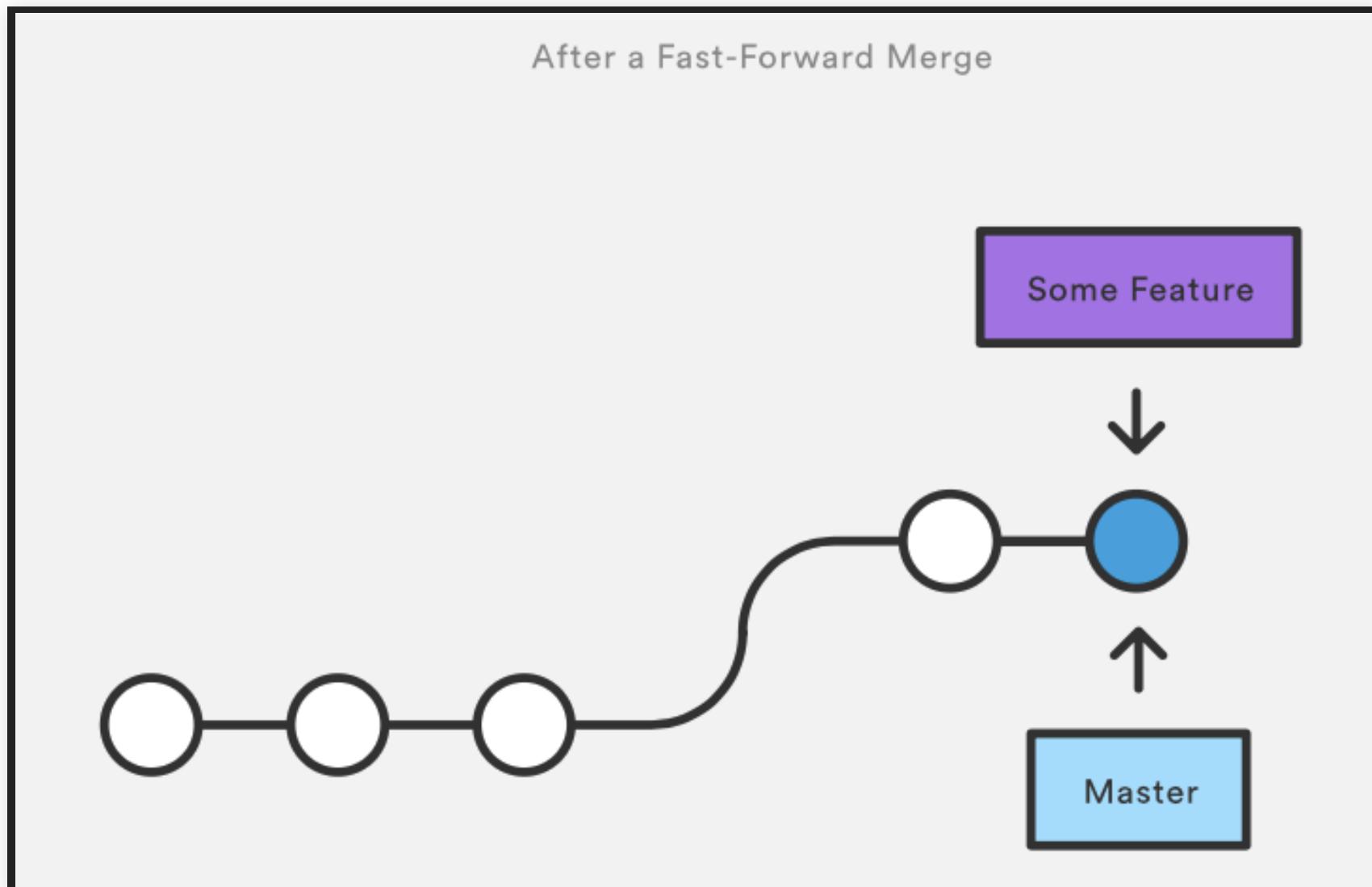
FAST-FORWARD-MERGE

```
$ git checkout master          # Aktivieren des „Ziel“-Branches
```



FAST-FORWARD-MERGE

```
$ git merge feature # Startet merge von allen Commits auf  
$ # Branch feature zu Branch master
```



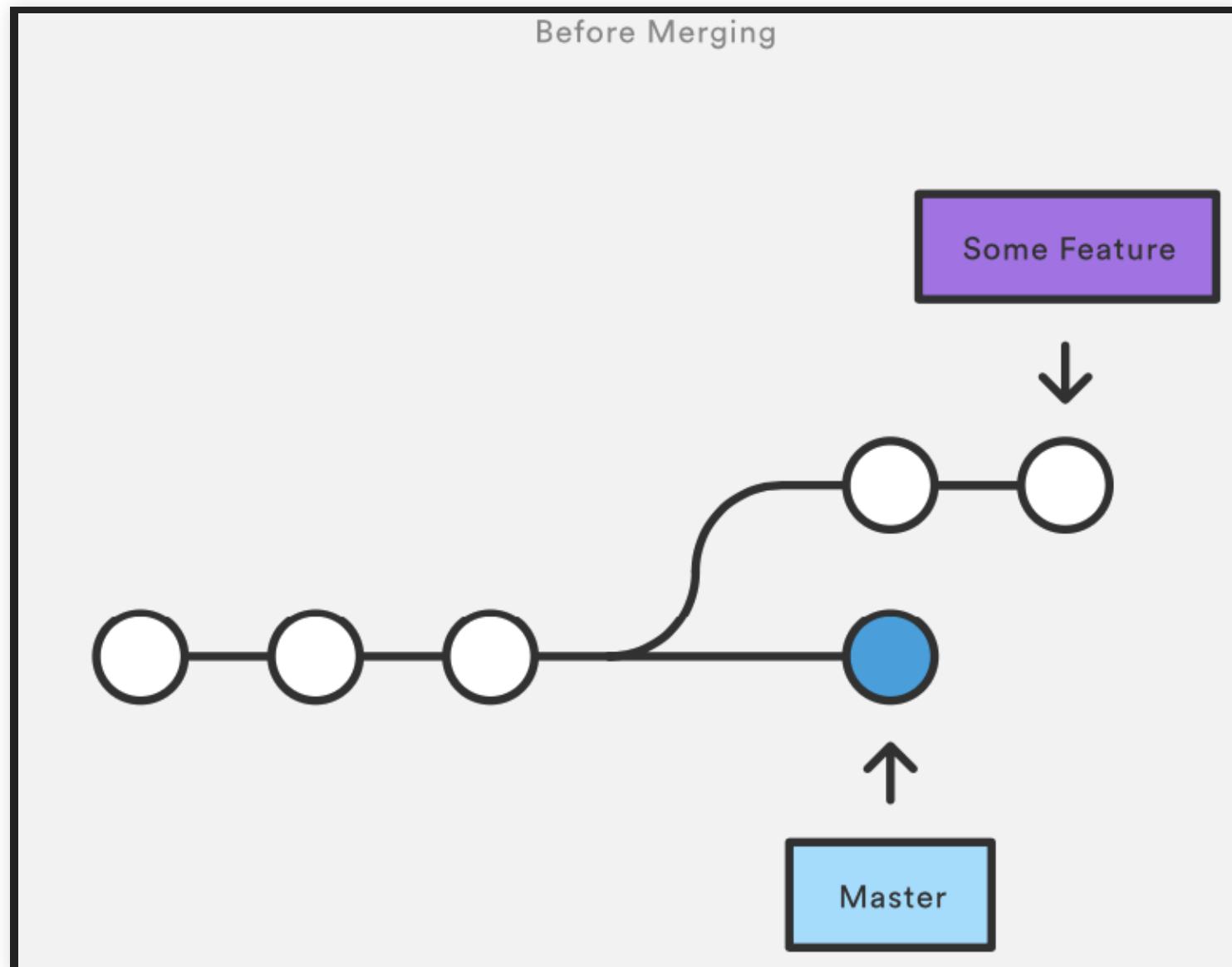
FAST-FORWARD-MERGE

- ein Merge führt zwei Branches zusammen
- ein Merge wird immer zu dem aktiven Branch hin ausgeführt
- Nur wenn es einen **linearen Pfad** von der Spitze des Ziel-Branches zur Spitze des Quell-Branches gibt
- Verändert niemals Dateien

```
$ git checkout master          # Aktivieren des „Ziel“-Branches  
$ git merge feature           # Startet merge von allen Commits auf  
                             # Branch feature zu Branch master  
$ git merge --ff-only feature # Starte merge nur, wenn FF möglich ist
```

3-WAY-MERGE

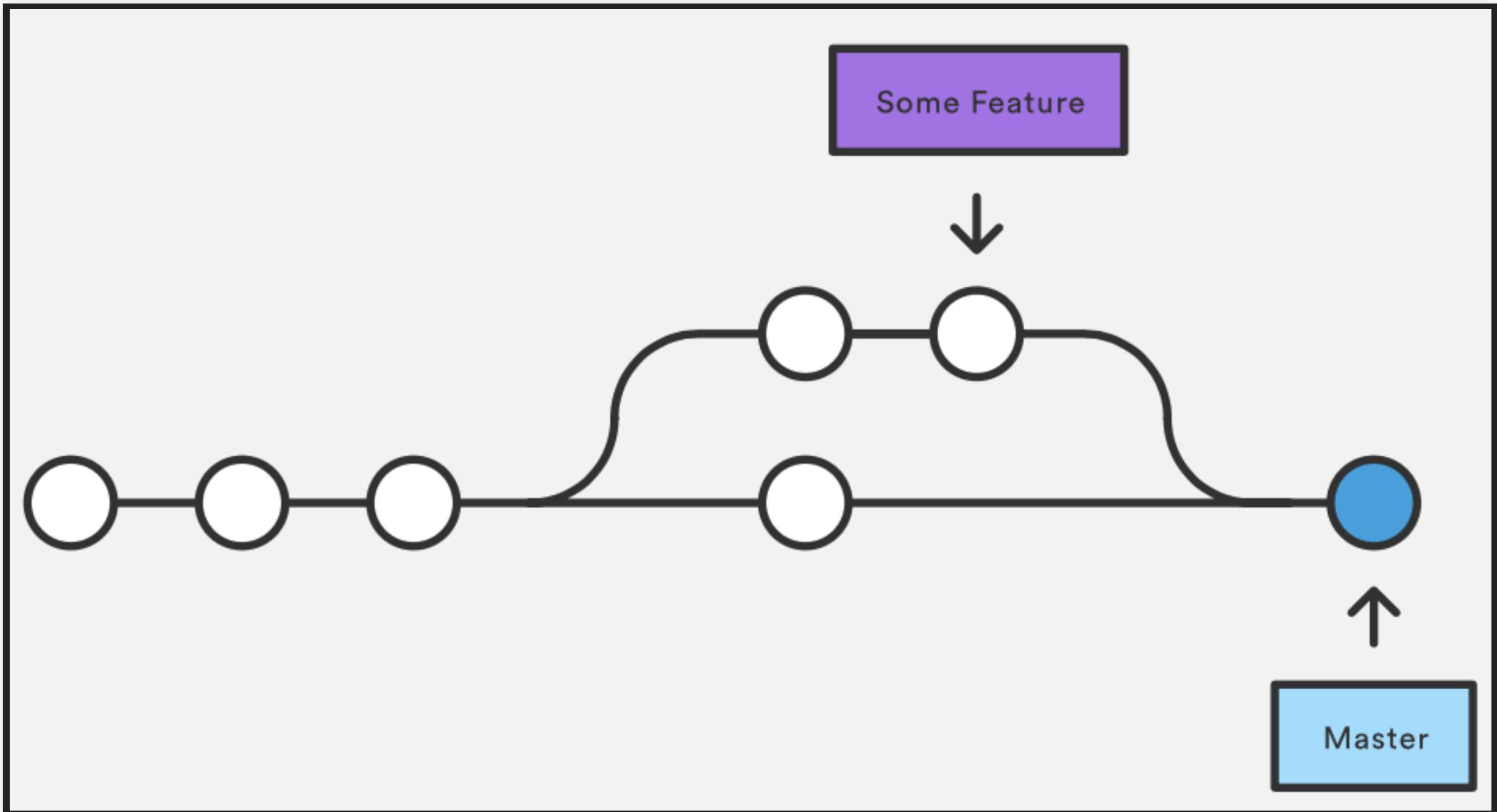
```
$ git checkout master          # Aktivieren des „Ziel“-Branches
```



3-WAY-MERGE

```
$ git merge feature
```

```
# Startet merge von ‚Quell‘-Branch
```



3-WAY-MERGE

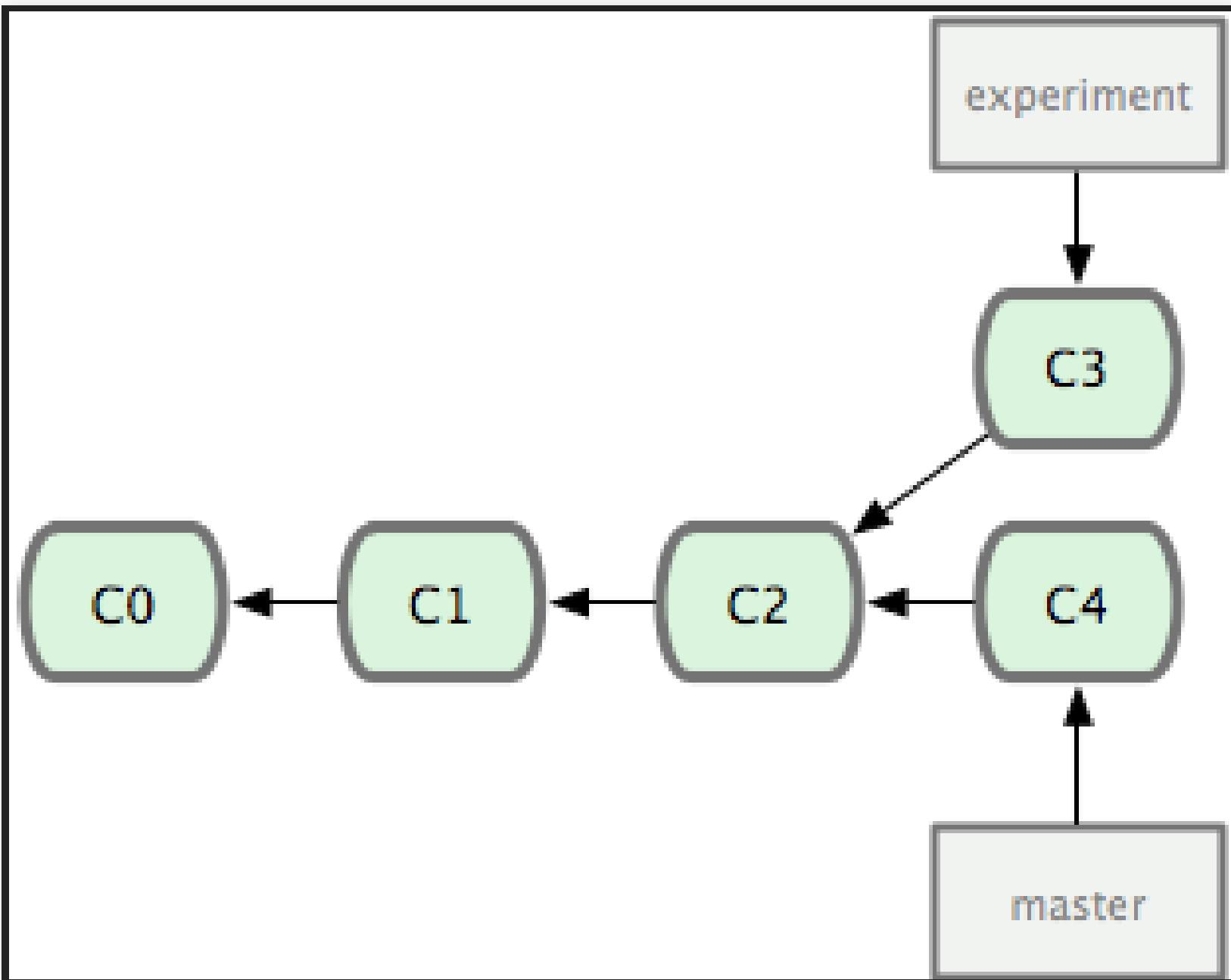
- ein Merge führt zwei Branches zusammen
- ein Merge wird immer zu dem aktiven Branch hin ausgeführt (wie FF)
- es entsteht ein **neuer Commit**

```
$ git checkout master      # Aktivieren des ‚Ziel‘-Branches
$ git merge feature       # Startet merge von ‚Quell‘-Branch
$ git merge --abort        # Abbrechen eines begonnenen Merges
$                                         # (der Konflikte hat)
$ git commit                # abschliessen eines Merges,
$                                         # bei dem Konflikte manuell behoben wurden
$ git reset --hard ORIG_HEAD # Macht ein versehentlichen und
$                                         # abgeschlossenen Merge rückgängig
```

REBASE

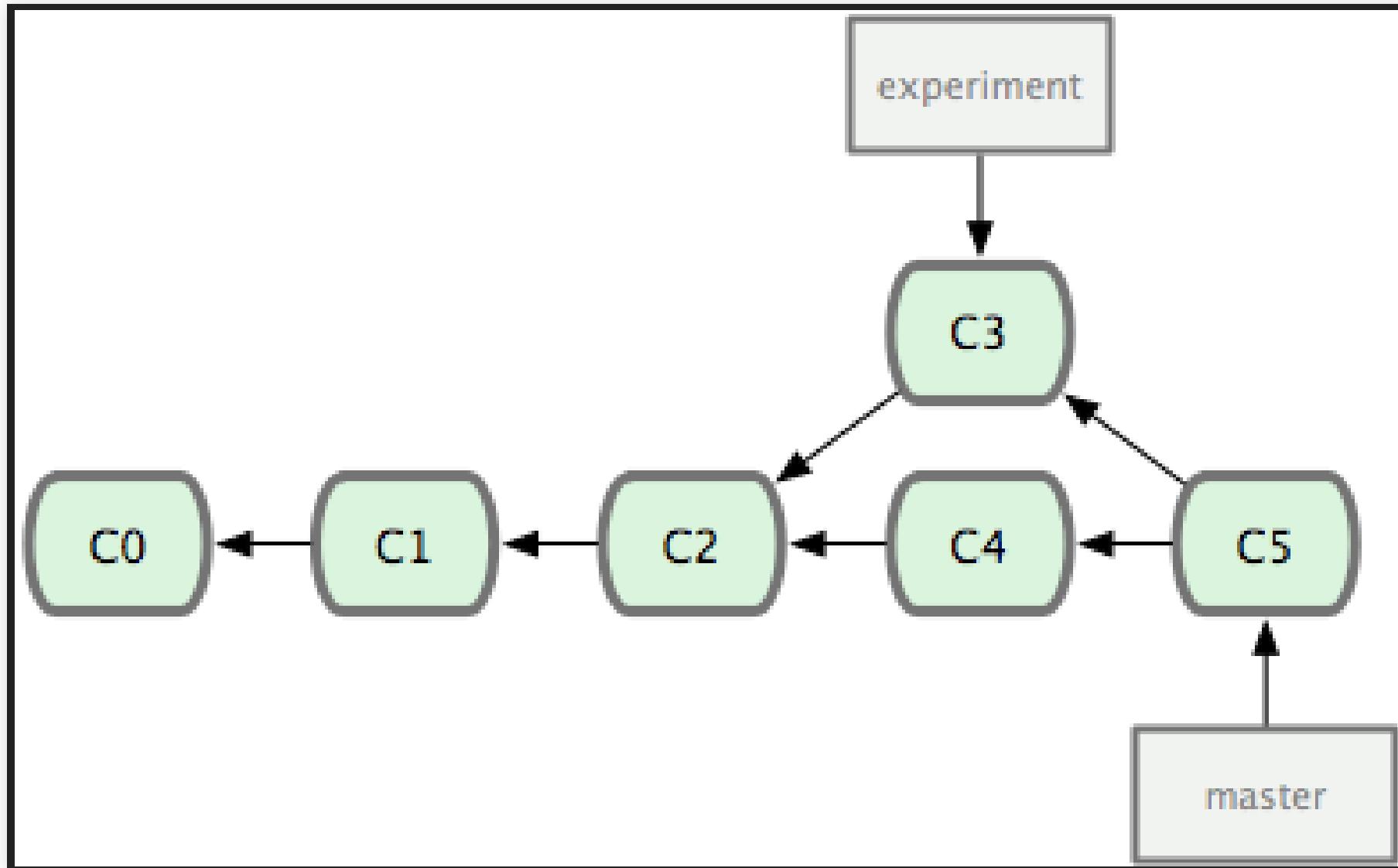
- Ist einer Alternative zum 3-Way-Merge
 - Vermeidet den Merge-Commit, indem die Voraussetzung für ein Fast-Forward geschaffen wird

REBASE START



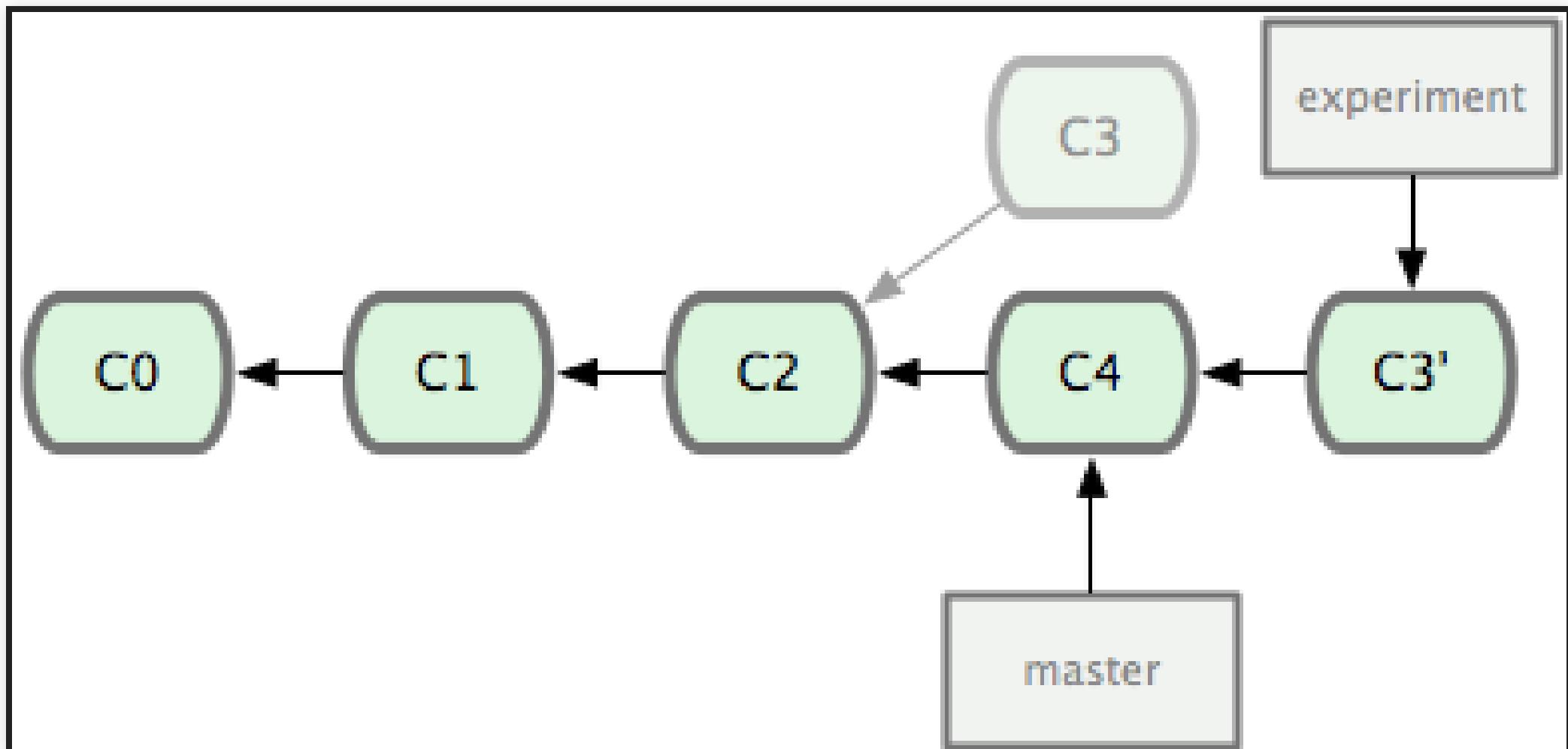
So würde ein 3-Way-Merge aussehen

```
$ git checkout master      # aktivieren des Ziel-Branches  
$ git merge experiment   # Starten des Merge von experiment  
$                                         # in master hinein
```



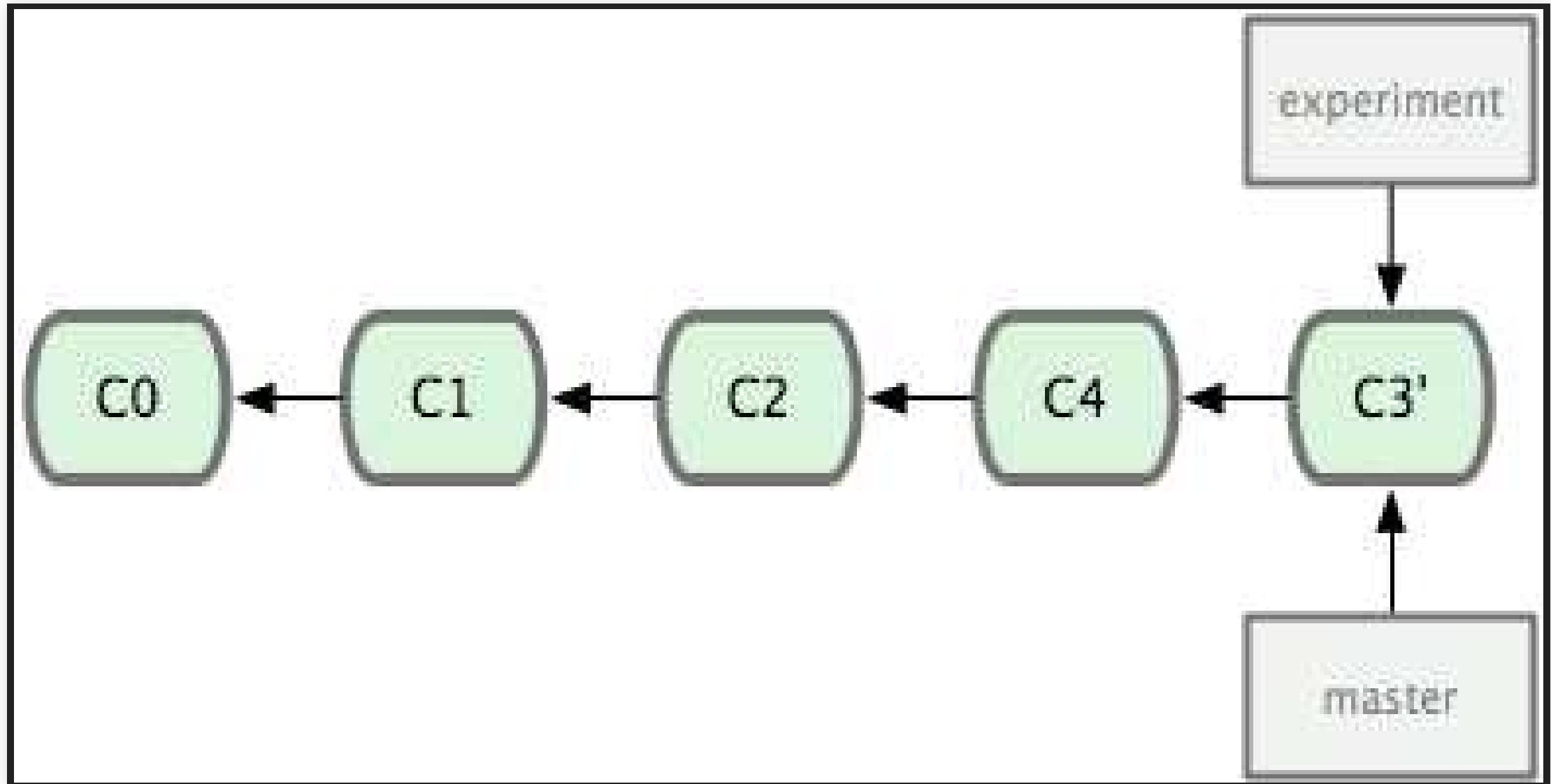
Stattdessen: Vorbereitung per *rebase*

```
$ git checkout experiment # aktivieren des Quell-Branches  
$      # die "Quelle" des Merges, den wir per rebase vorbereiten  
$ git rebase master      # Starten des Rebasing  
$      # "master" als neue Basis für alle Commits  
$      # auf branch "experiment"
```



gefolgt von: *fast-forward* Merge

```
$ git checkout master      # aktivieren des Ziel-Branches  
$ git merge experiment   # Starten des Merge von experiment  
$                                         # in master hinein
```



REBASE

- ein Rebase wird immer auf dem aktiven Branch durchführt
 - Verändert alle Commits, die bisher auf dem aktiven Branch gemacht wurden
 - Ziel und Quelle sind hier anders, als beim (folgenden) Merge
 - ggf. manuelle Konfliktbehebung, wie beim Merge nötig
- Verändert alle Commits des Quelle-Branches

REBASE KOMMANDOS

```
## Aktivieren des Branches, der rebased werden soll
$ git checkout feature
## Startet rebase: neue Basis für den aktiven Branch
$ git rebase master
## Macht ein versehentliches Rebase rückgängig
$ git reset --hard ORIG_HEAD
## fügt Datei, die manuell bereinigt werden musste, zum Rebase hinzu
$ git add former-conflicted.txt
## Fortsetzen des Rebasing, nachdem Konflikte bereinigt wurden
$ git rebase --continue
## Abbruch des Rebasing (jederzeit möglich)
$ git rebase --abort
```

REBASE VORTEILE

- kein unnötiger commit C5
- klar lesbare Historie
- Wenn jmd. anderes deine Änderung integrieren soll, dann ist es einfacher, wenn du einen Rebase machst, anstatt dass er einen 3-Way-Merge machen muss.
 - Verlagern der Verantwortung

QUELLEN

- Atlassian Tutorials <https://www.atlassian.com/git/tutorials/using-branches>
- Git Pro Buch - Was ist ein Branch <https://git-scm.com/book/de/v1/Git-Branching-Was-ist-ein-Branch>
- Git Pro Buch - Rebasing <https://git-scm.com/book/de/v1/Git-Branching-Rebasing>

VCS 4

REMOTES

GIT

Stupid Content Tracker, effizienter Objekt-Speicher mit der Fähigkeit zwei Objektspeicher miteinander zu Synchronisieren.

Synchronisation bei Daten == Replikation

Synchronisation der Git-Objekte == Replikation aller Git-
Objekte

REMOTES

- Das letzte verbliebene Element in der Liste der Git-Objekte
- Ein lokales **Repository** kann mit vielen verschiedenen **entfernten Repositories** verbunden sein
- wenn Repo durch **git clone** entstanden ist, gibt es genau ein Remote mit dem Namen **origin**

```
## Zeigt alle konfigurierten Remotes an
$ git remote -v
origin  https://github.com/barclay-reg/dhw-slides.git (fetch)
origin  https://github.com/barclay-reg/dhw-slides.git (push)
```

BRANCHES SYNCHRONISIEREN

- Sync bei GIT-Objekten relativ einfach
- Sync bei Referenzen etwas komplexer
 - Problem: Referenz wurde auf beiden Seiten verändert
 - Lösung:
 - Speichern aller lokalen und remote Referenzen
 - Speichern der Verbindung zwischen lokaler und remote Referenz

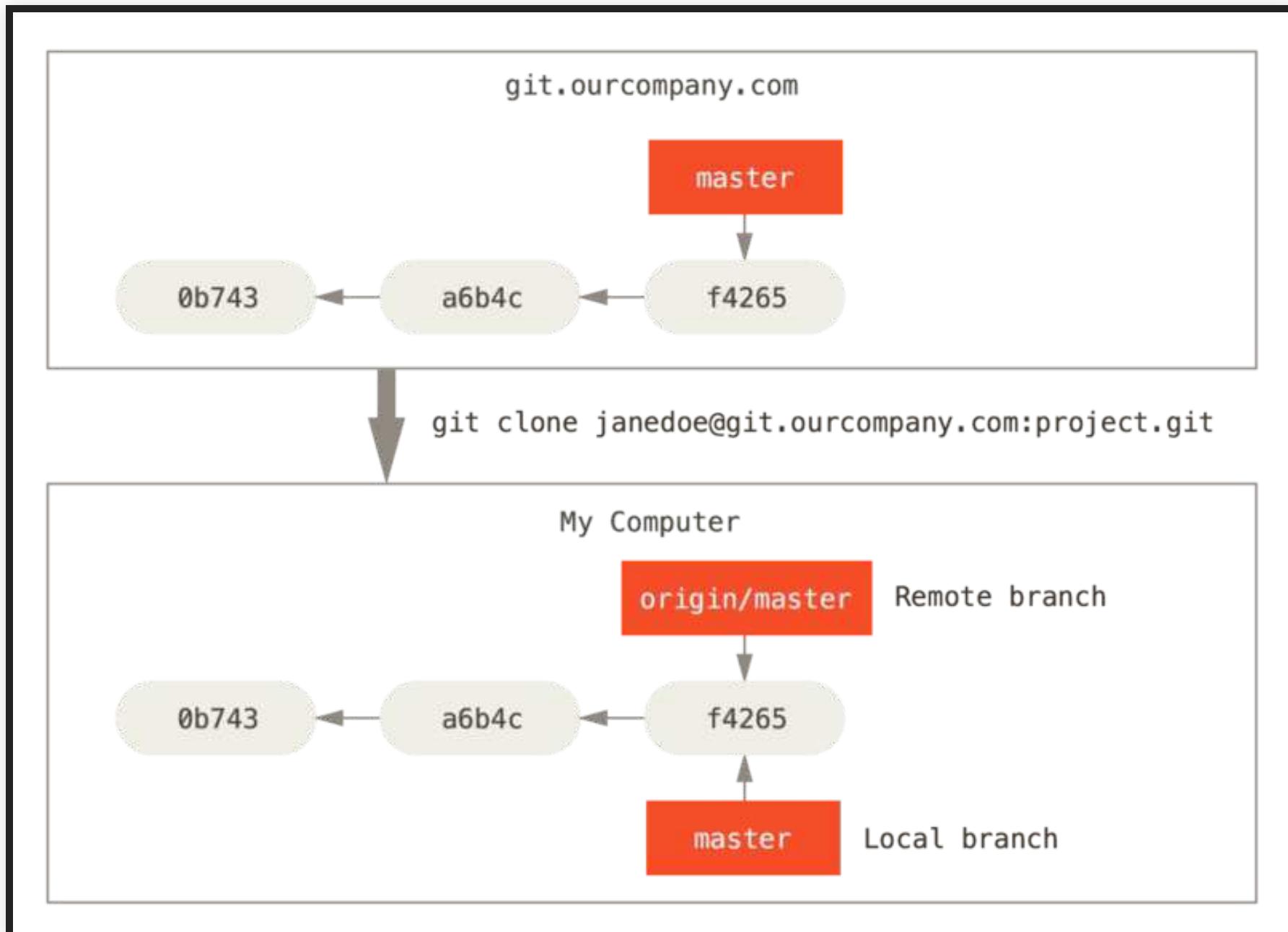
REMOTE REFERENZEN

```
## Auflistung aller Dateien im Ordner .git/refs
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/heads/master
.git/refs/heads/my-branch-1
.git/refs/tags
.git/refs/tags/test-tag-0
.git/refs/tags/test-tag-1
.git/refs/remotes
.git/refs/remotes/origin
.git/refs/remotes/origin/master
.git/refs/remotes/origin/other-branch-2
```

REMOTE TRACKING BRANCHES

- Wenn ein **lokaler** Branch mit einem **remote** Branch verbunden ist, dann spricht man von einem sog.:
 - **Remote-Tracking-Branch**
 - oder **Upstream-Branch**
 - Git weiss, dass der lokale Branch dem remote Branch *folgt*
 - z.B. der Branch **origin/master** wird von **master** getracked
- Name der remote Branches == Namen des Remote Repository plus dem Namen des Branches zusammen
 - z.B. **origin/feature-1**

REMOTE TRACKING BRANCHES

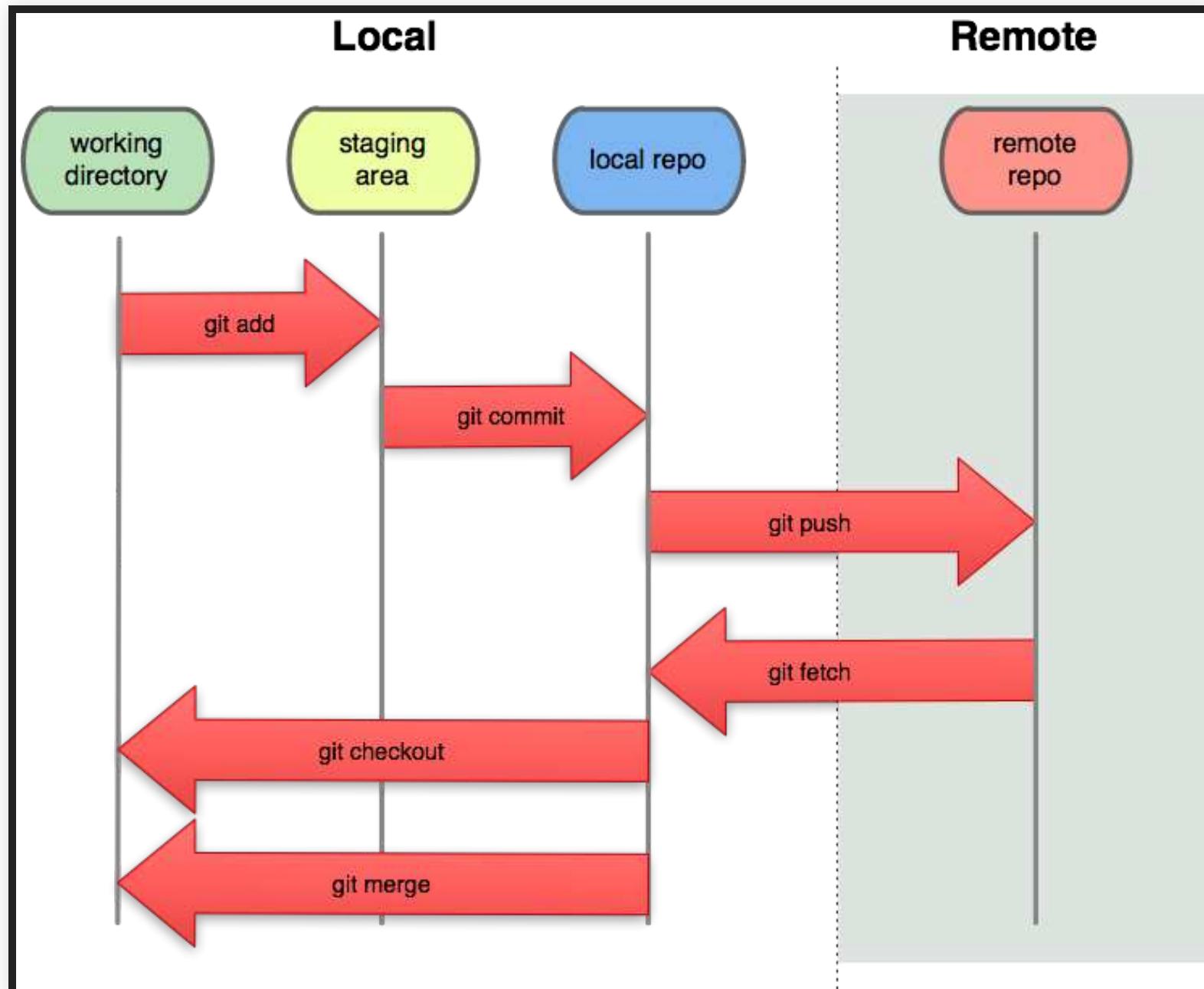


KOMMANDOS

```
## Zeigt alle Remote Branches an
$ git branch -r
origin/master
origin/feature-1
## Legt neuen Branch an, der origin/feature-1 trackt
$ git branch feature-1 origin/feature-1
## Shortcut, legt automatisch lokalen Branch an, falls ein
## Branch `origin/feature-1` existiert
$ git checkout feature-1
## Definiert für den aktuellen lokalen Branch den
## Namen des Remote Branches
$ git branch --set-upstream-to origin/neues-feature
```

ARBEITEN MIT REMOTES

TRANSPORTWEGE

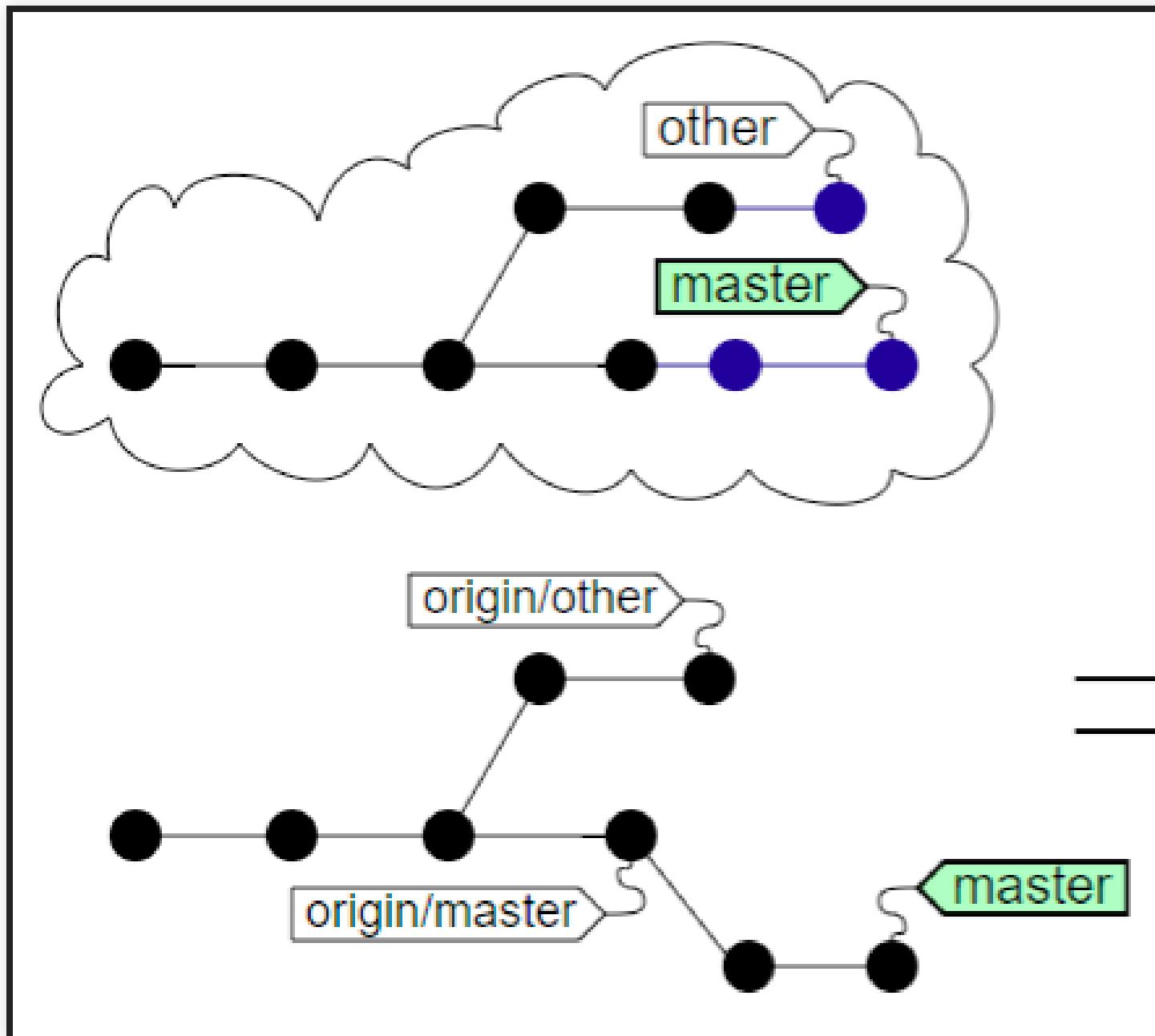


FETCH UND PULL

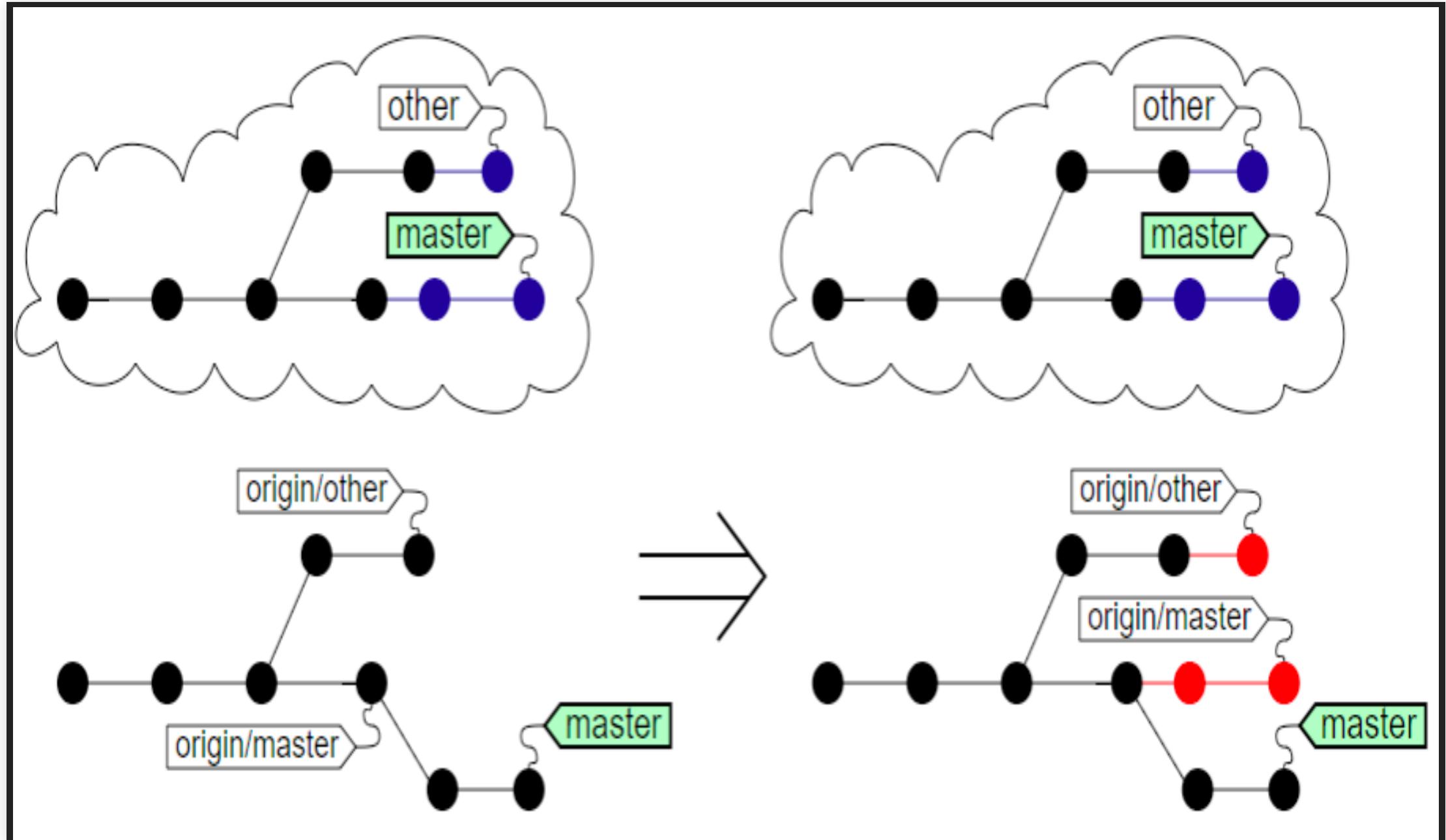
- **git fetch**
 - Holt alle Commits von den remote Repository(ies)
 - keine Änderung an der Workcopy
 - oft ist danach ein Merge notwendig (3WM oder FF)
- **git pull**
 - Shortcut für **git fetch & git merge**

```
## ohne die Angabe von `origin` werden alle Remotes abgefragt
$ git fetch origin
## merge des Remote-Branches auf den aktuellen lokalen Branch
$ git merge origin/master
## Shortcut für die beiden oberen Befehle
## auch hier kann `origin` weggelassen werden
$ git pull origin
```

FETCH UND PULL



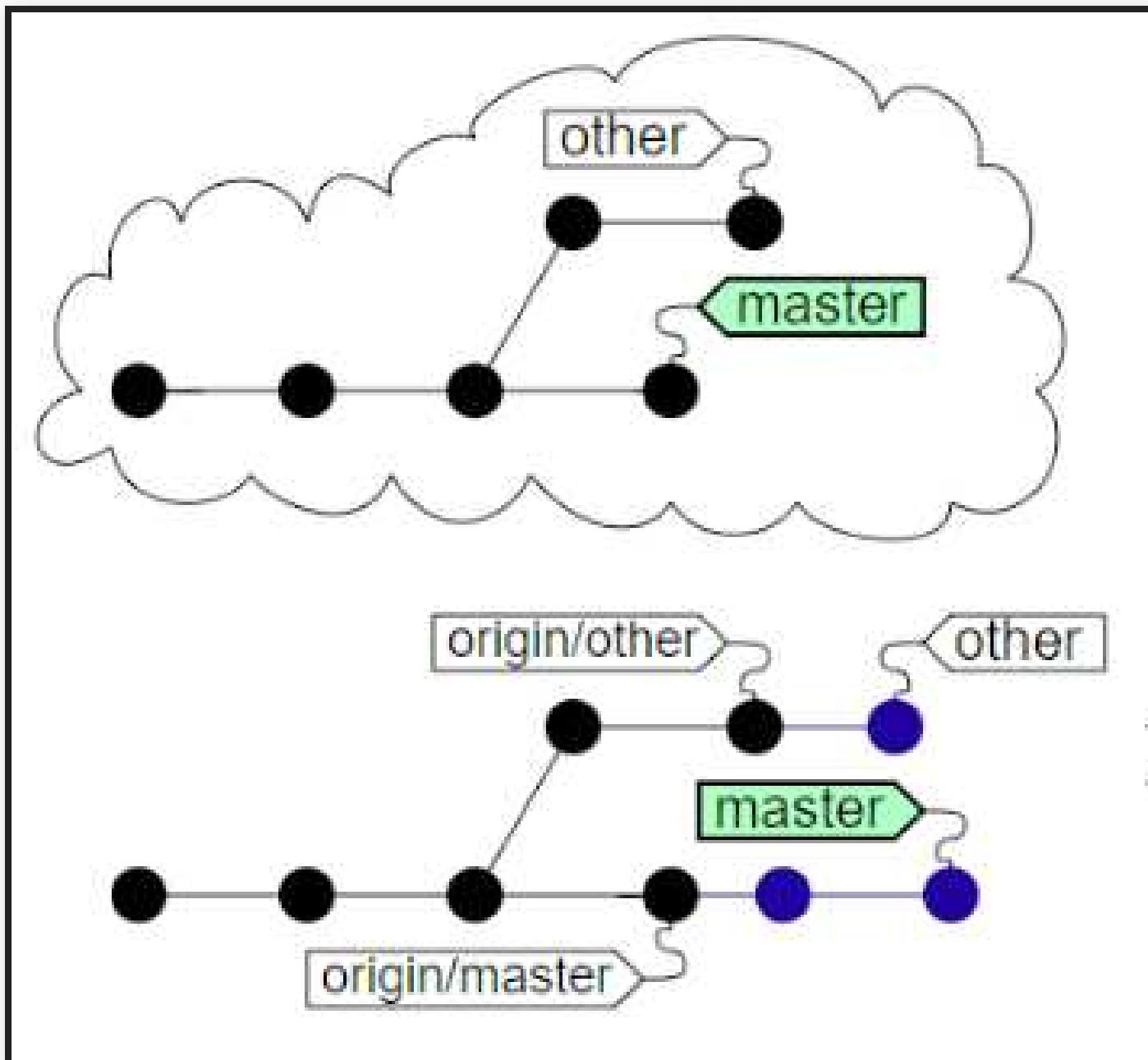
FETCH UND PULL



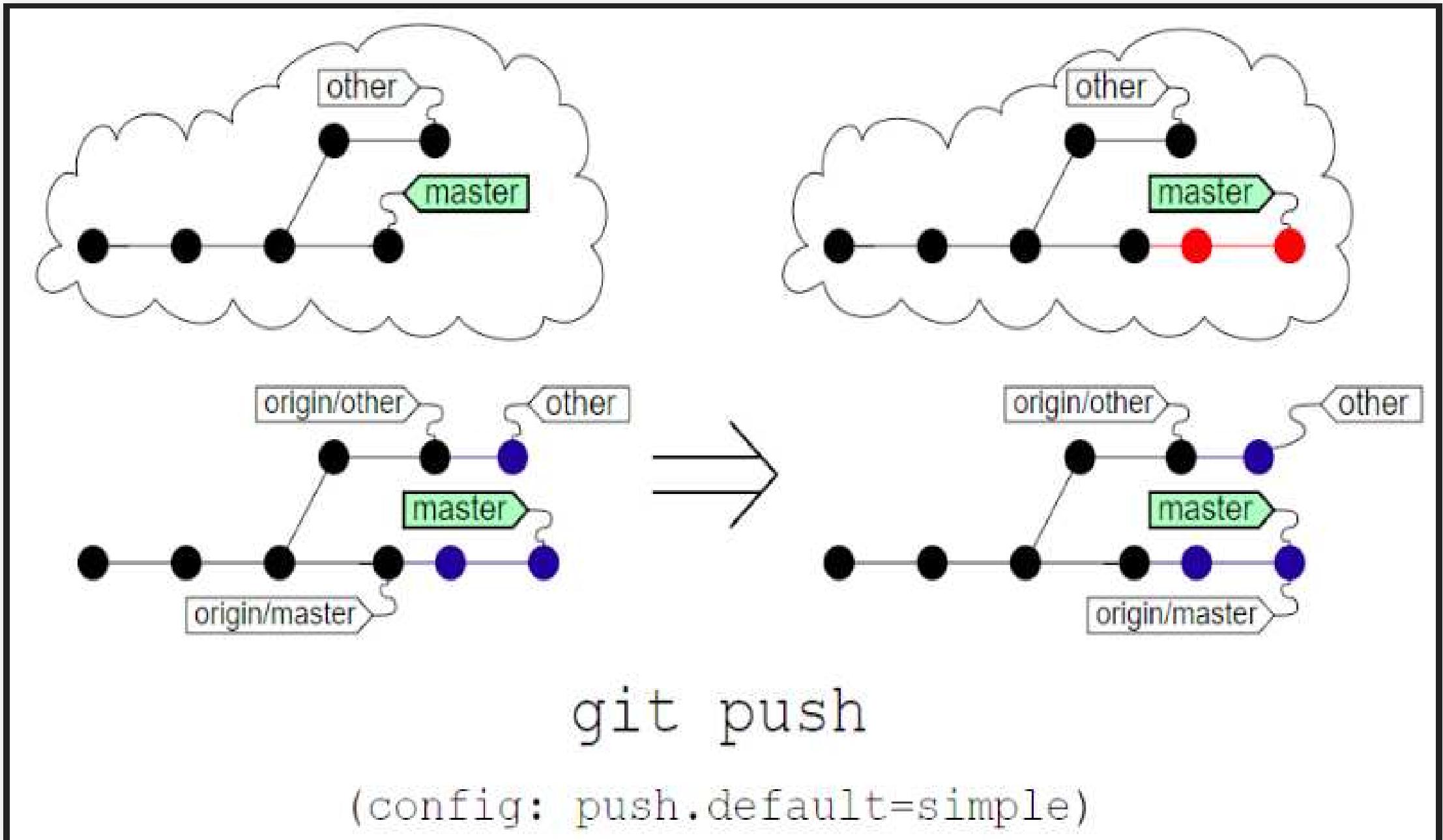
PUSH

- `git push`
 - überträgt alle lokalen Commits zu dem Remote Repository
 - Nur erlaubt, wenn (remote) ein **Fast-Forward-Merge** möglich ist, ansonst vorher `git pull`
 - danach ist **KEIN Ändern** der Historie/Commits empfohlen
 - Kein Commit-Amend, Reset von Branches, Rebasing
 - je nach Konfiguration wird nur der lokale Branch oder alle Branches synchronisiert
 - `config: push.default=simple`

PUSH



PUSH



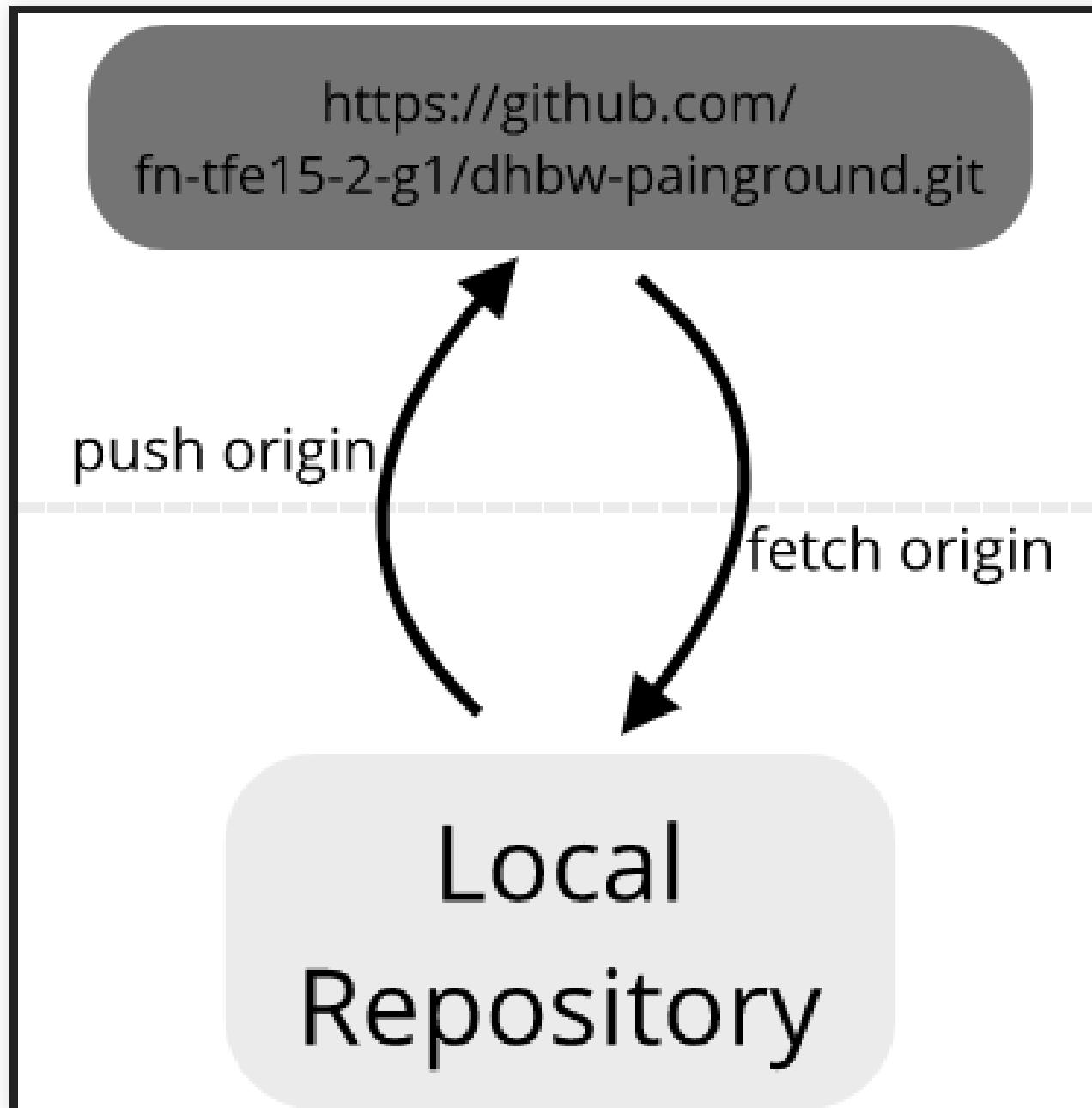
CLONE & FORK

- `git clone`
 - kopieren eines remote Repositories auf den eigenen Rechner
 - "erste Synchronisieren" plus "Checkout"
 - kein `git init` mehr nötig
- `fork`
 - kein Git Befehl
 - Findet auf einem Git-Server statt, z.B. auf <https://github.com>
 - im Hintergrund wird auch `git clone` ausgeführt

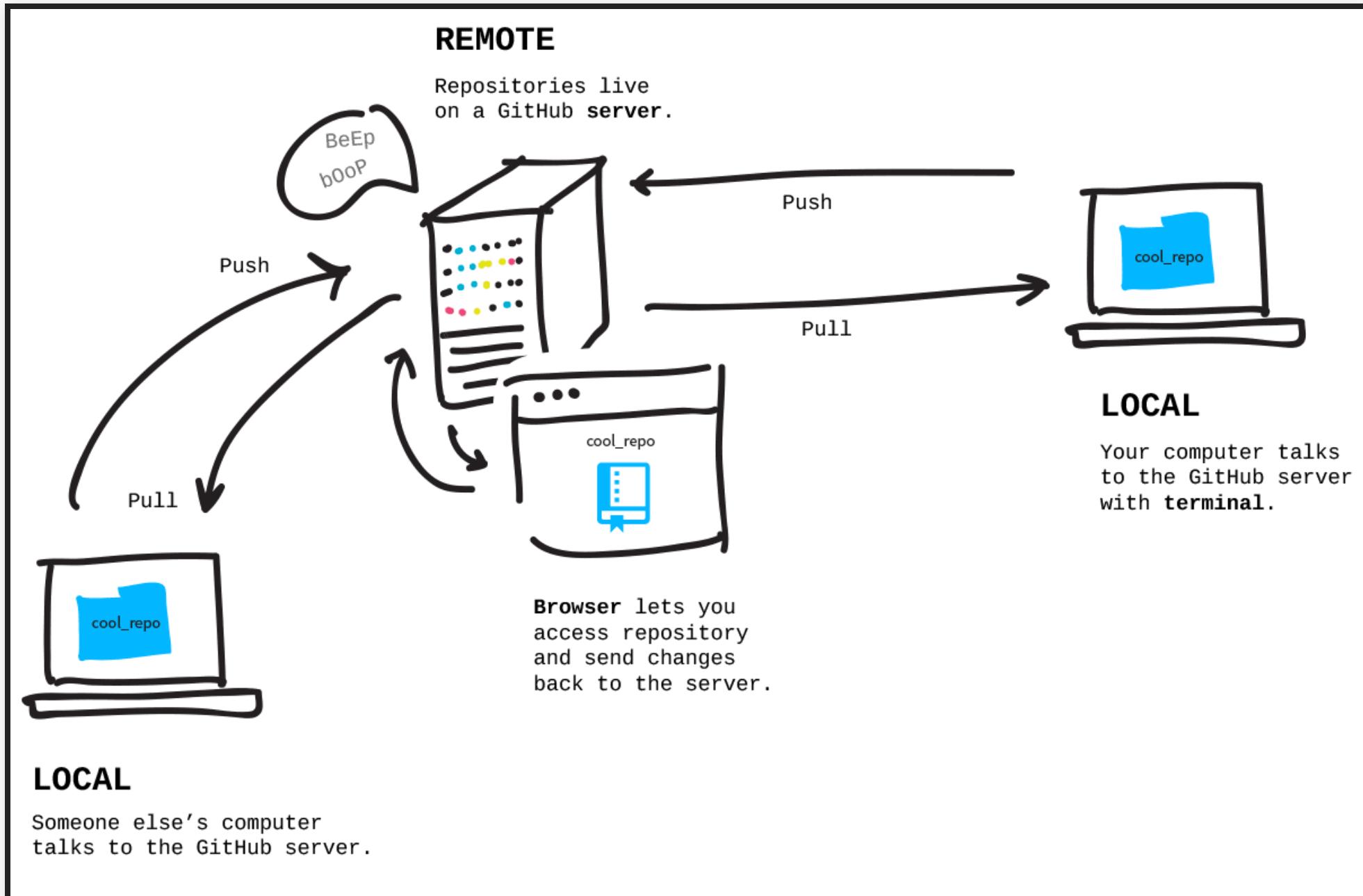
CLONE & FORK

- Problem: Wie kommen Änderungen des Originals zu meinem Fork?
- Lösung: weiteres Remote-Repo

OHNE FORK



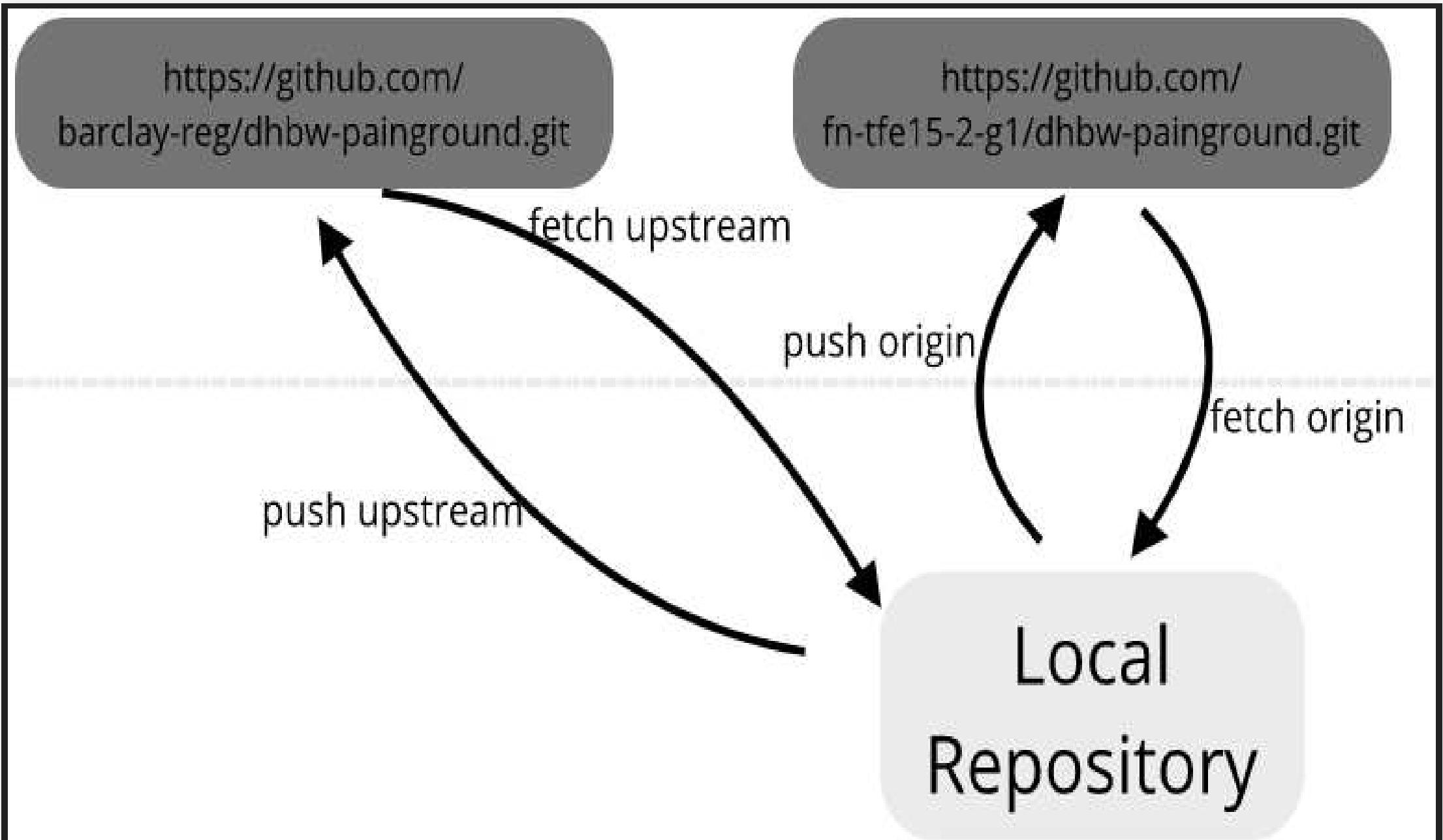
OHNE FORK - ANDERS



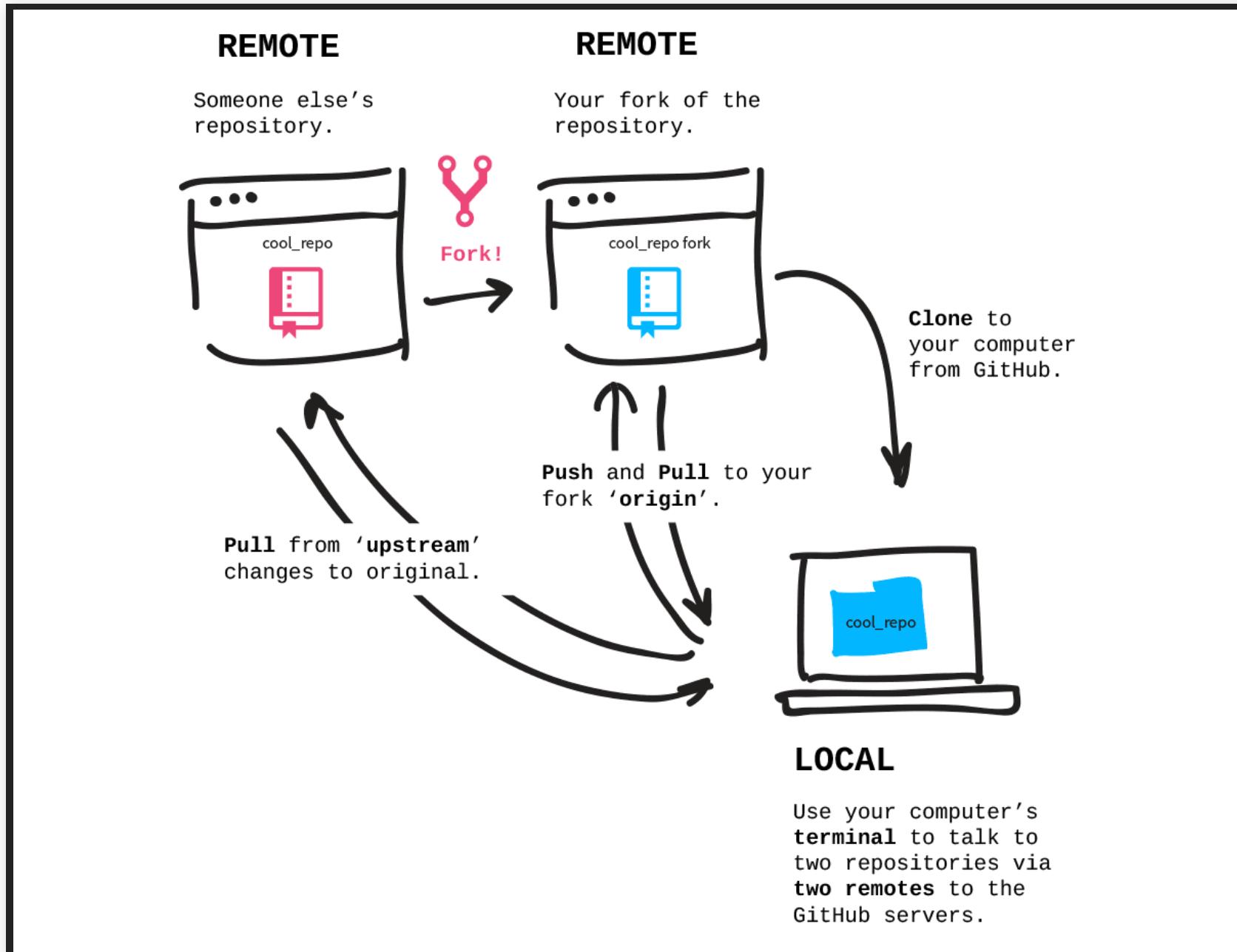
OHNE FORK - REMOTES

```
$ git remote -v
origin  github.com/fn-tfe15-2-g1/dhw-painground.git (fetch)
origin  github.com/fn-tfe15-2-g1/dhw-painground.git (push)
```

MIT FORK



MIT FORK - ANDERS



MIT FORK - REMOTES

```
$ git remote add upstream https://github.com/barclay-reg/dhbw-painground
$ git remote -v
origin    github.com/fn-tfe15-2-g1/dhbw-painground.git (fetch)
origin    github.com/fn-tfe15-2-g1/dhbw-painground.git (push)
upstream   github.com/barclay-reg/dhbw-painground.git (fetch)
upstream   github.com/barclay-reg/dhbw-painground.git (push)
```

MIT FORK - ÄNDERUNGEN ABHOLEN

```
## Änderungen von Remote "upstream" holen
$ git fetch upstream
## auf eigenen Branch "master" wechseln
$ git checkout master
## Alle commits von Branch "master" von Remote "upstream"
## in aktuellen Branch mergen
$ git merge upstream/master
## Änderungen an github senden
$ git push
```

PULL REQUEST

- Antrag, ein oder mehrere Commits von einem Branch in einen anderen Branch zu mergen
- kann jemandem *zugewiesen* werden
- Erlaubt Code-Review, Code-Diskussion
- wenn Antrag akzeptiert ist, wird ein **Pull** (fetch & merge) gemacht
- Kann per `git request-pull` gestartet werden, aber
- besser per Web-Interface (Github, Bitbucket, Gitlab)

WIESO PR

- Warum nicht einfach Mergen?
 - (Feature)-Branches können länger leben
 - Niemand außer dem Author weiß, wann das Feature *fertig* ist
 - Erstellend es PR ist ein eindeutiger Trigger für
 - Start des Code-Reviews
 - Start von aufwändigeren automatisierten Tests

Clean Code

HERKUNFT

TPM

Total Productive Maintenance

- Qualitätsansatz
- ~1960, japanische Autoindustrie
- Konzentration auf Instandhaltung des Arbeitsplatzes
- ähnlich Lean Produktion
- Fundament: 5S-Prinzipien

Seiri

Aussortieren; Übersicht schaffen - wo finde ich Dinge wieder, Namensgebung

Seiton

Ordentlichkeit; Code sollte da stehen, wo ich ihn erwarte

Seiso

Säubern; Abfall und Einzelteile entfernen

Seiketsu

Standardisierung; konsistenter Codierstil

Shutsuke

(Selbst-) Disziplin & ständige Verbesserung

Qualität ist das Ergebnis einer Million selbstloser Akte der Sorgfalt.

— Robert “Uncle Bob” Martin

Wir (Entwickler) sind Autoren. Ein Merkmal von Autoren ist es, dass sie Leser haben.

– Robert “Uncle Bob” Martin

Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden.

– Dave Thomas

CHAOS IM CODE

- je älter ein Projekt, desto höher der Aufwand, neue Funktionen hinzuzufügen
- damit die Produktivität wenigstens annähernd gleich bleibt, wird (leider) der Fokus der Arbeit auf neue Funktionen gelegt
 - Folge: **Code verrottet** - wichtige Basis-Arbeiten werden vernachlässigt
 - keine neuen Tests
 - Konzepte werden durch Ausnahmen aufgeweicht
 - Dokumentation wird nicht nachgezogen
- Gesetz von LeBlanc: **Später heißt niemals**

CHAOS IM CODE

- Schlussfolgerung:
 - es reicht nicht aus, guten Code zu schreiben
 - Code muss auch *sauber gehalten* werden
 - sofort & kontinuierlich

*Leave the campground cleaner than
you found it*

— Robert “Uncle Bob” Martin

AUSSAGEKRÄFTIGE NAMEN

ZWECKBESCHREIBENDE NAMEN

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

- Kontext geht nicht aus dem Code hervor
- Code ist implizit, sollte aber explizit sein

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : this.gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

FEHLINFORMATIONEN VERMEIDEN

- keine irreführenden Hinweise, z.B. für eine Gruppe von Konten:

```
private Map accountList;
```

- zwei Namen sollten sich nicht geringfügig unterscheiden, z.B.

```
XYZControllerForEfficientHandlingofStrings  
XYZControllerForEfficientStorageOfStrings
```

UNTERSCHIEDE DEUTLICH MACHEN

```
public static void copyChars(char c1[], char c2[]) {  
    for (int i=0; i < c1.length; i++) {  
        c2[i] = c1[i];  
    }  
}
```

```
public static void copyChars(char source[], char destination[]) {  
    for (int i=0; i < source.length; i++) {  
        destination[i] = source[i];  
    }  
}
```

- Namen wie **c1** sind nicht **irreführend**, sondern **informationsleer**
- zusammengesetzte Klassennamen können auch **informationsleer** sein
 - **Product**
 - **ProductInfo**
 - **ProductData**

AUSSPRECHBARE NAMEN VERWENDEN

```
class DtaRcrd102 {  
    private Timestamp genymdhms;  
    private Timestamp modymdhms;  
}
```

```
class DtaRcrd102 {  
    private Timestamp genymdhms;  
    private Timestamp modymdhms;  
}
```

ymdhms

Year, Month, Day, Hours ...

```
class DtaRcrd102 {  
    private Timestamp genymdhms;  
    private Timestamp modymdhms;  
}
```

ymdhms

Year, Month, Day, Hours ...

```
class Customer {  
    private Timestamp generationTimestamp;  
    private Timestamp modificationTimestamp;  
}
```

SUCHBARE NAMEN VERWENDEN

```
int s = 0;  
for (int j=0; j<34; j++) {  
    s += (t[j]*4)/5;  
}
```

- Die Länge eines Namens sollte der Größe seines Geltungsbereichs entsprechen
- Suche nach *t* oder *5* ergibt in der gesamten Codebasis viele Treffer

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

CODIERUNGEN VERMEIDEN

```
// Datentypen
private String szVorname;
private Integer nId;
// Geltungsbereich
private String pri_szVorname;
public Integer pub_nId;
```

- Codierung von Informationen in Namen von Variablen
 - Datentyp oder Geltungsbereich
 - Ungarische Notation
- Nachteile
 - Änderungen müssen überall nachgezogen werden
 - Präfixe und Suffixe werden bald vom Entwickler ignoriert

METHODENNAMEN

- Verben verwenden, z.B.
 - `downloadEmailAttachments()`
- nur ein Wort pro Konzept
 - *fetch, retrieve, get ...* sind Synonyme

DOMÄNEN NAMEN

- Problemdomäne
 - Begriffe/Konzepte des Bereichs, für den die Software bestimmt ist
 - z.B. **BeneficialOwner**
 - Bezug auf wirtschaftlich Berechtigten eines Bankkontos
- Lösungsdomäne
 - Begriffe/Konzepte der Informatik, Algorithmen, Pattern
 - z.B. **AccountVisitor**
 - Bezug auf Visitor-Pattern

FUNKTIONEN

BEISPIEL

HtmlUtil.java SetupTeardownIncluder.java

```
public class HtmlUnit {
    public static String testableHtml(
        PageData pageData,
        boolean includeSuiteSetup
    ) throws Exception
    {
        WikiPage wikiPage = pageData.getWikiPage();
        StringBuffer buffer = new StringBuffer();
        if (pageData.hasAttribute("Test")) {
            if (includeSuiteSetup) {
                WikiPage suiteSetup =
                    PageCrawlerImpl.getInheritedPage(
                        SuiteResponder.SUITE_SETUP_NAME, wikiPage
                    );
                ...
            }
        }
    }
}
```

- Beispiel aus **Fitnesse**
 - FitNesse begann als ein HTML und Wiki "front-end" für FIT ("Framework for Integrated Testing")
 - Wiki Seite == Page
 - Test-Suite == Zusammenfassung mehrere Tests
 - Teststruktur
 - ggf. Suite Setup
 - Setup
 - Test (== pageDate)
 - TearDown
 - ggf. Suite TearDown

```
public class HtmlUnit {
    public static String testableHtml(
        PageData pageData,
        boolean includeSuiteSetup
    ) throws Exception
    {
        WikiPage wikiPage = pageData.getWikiPage();
        StringBuffer buffer = new StringBuffer();
        if (pageData.hasAttribute("Test")) {
            if (includeSuiteSetup) {
                WikiPage suiteSetup =
                    PageCrawlerImpl.getInheritedPage(
                        SuiteResponder.SUITE_SETUP_NAME, wikiPage
                    );

```

ERSTE VERBESSERUNG

```
public static String renderPageWithSetupsAndTear downs(
    PageData pageData, boolean isSuite
) throws Exception {

    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTearDownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
}
```

KLEIN

- Funktionen sollten klein sein
Wie kann das erreicht werden?
- keine verschachtelten Strukturen
- die *einzig erlaubte* Einrückungstiefe sollte dann möglichst nur eine Anweisung enthalten

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, bool isSuite
) throws Exception {
    if (isTestPage(pageData)) {
        includeSetupAndTeardownPages(pageData, isSuite)
    }
    return pageData.getHtml();
}
```

EINE AUFGABE ERFÜLLEN

- eine Aufgabe
 - Wenn alle Schritte einer Funktion eine Abstraktionsebene **unter** dem Zweck liegen, der durch den Namen ausgedrückt wird
- Hilfsmittel
 - einen **UM-ZU**-Absatz formulieren

UM

RenderPageWithSetupsAndTear downs ausZUführen, prüfen wir, ob eine Seite eine Testseite ist, und wenn dies der Fall ist, schließen wir die Setups und Tear downs ein. In beiden Fällen stellen wir die Seite in HTML dar.

BESCHREIBENDE NAMEN

- gute Namen für kleine Funktionen finden, die **eine** Aufgabe erledigen
- lange **beschreibende** Namen sind besser als **kurze geheimnisvolle** Namen
- lange **Namen** sind besser als lange **Kommentare**
- mehrere Wörter per Konvention trennen
 - CamelCaseSchreibweise
- verschiedene Namen ausprobieren und Code lesen
 - IDE unterstützt das
- Namen sollten in einem Modul konsistent sein
 - Synonyme vermeiden

FUNKTIONSARGUMENTE

- je weniger Argumente, desto besser
 - jedes Argument erfordert konzeptionelle Kraft beim Lesen
 - Name und Typ des Arguments könnten zu anderer Abstraktionsebene gehören
 - das **Testen** einer Funktion wird aufwändiger
 - die Kombinationen aller Argumente mit allen möglichen Werten

FUNKTIONSARGUMENTE

- Output-Argumente vermeiden, da ungewohnt
 - Input: Argumente
 - Output: Rückgabewert

FUNKTIONSARGUMENTE

- Argument als Output verwendet

```
public static void splitToList(String source, List parameter) {  
    String[] array = source.split(",");  
    parameter.addAll(Arrays.asList(array));  
}
```

FUNKTIONSARGUMENTE

- Argument als Output verwendet

```
public static void splitToList(String source, List parameter) {  
    String[] array = source.split(",");  
    parameter.addAll(Arrays.asList(array));  
}
```

- Rückgabewert als Output

```
public static List splitToList(String source) {  
    String[] array = source.split(",");  
    return Arrays.asList(array);  
}
```

FLAG-ARGUMENTE

- Hinweis darauf, dass mehrere Aufgaben erfüllt werden

```
// Aufruf
    render(true);
// Definition
class Renderer {
    void render(boolean isSuite) {}
}
```

FLAG-ARGUMENTE

- Besser mehrere Methoden

```
// Definition
class Renderer {
    void renderForSuite() {}
    void renderForSingleTest() {}
}
```

DYADISCHE FUNKTIONEN

- Funktionen mit 2 Argumenten
- Verwender muss die Reihenfolge und Bedeutung kennen
 - oder Definition nachschlagen → Aufwand!
- oft unvermeidbar

```
// Aufruf
int result = getResult(); // 24
assertEquals(24, result);
// Definition
class Assert {
    void assertEquals(int expected, int actual) {}
}
```

NEBENEFFEKTE VERMEIDEN

```
public boolean checkPassword(String userName, String password){  
    User user = UserGateway.findByName(userName);  
    if (user != User.NULL) {  
        if (user.password.equals(password)) {  
            Application.loginUser(user);  
            return true;  
        }  
    }  
    return false;  
}
```

ANWEISUNG ODER ABFRAGE

- Funktion sollte entweder
 - etwas tun, oder
 - etwas antworten

```
public boolean set(String attribute, String value){  
    if (internalList.contains(attribute)) {  
        internalList.set(attribute, value);  
        return true;  
    } else {  
        return false;  
    }  
}  
// mögliche Verwendung  
if (set("username", "robkle")) ...
```

FEHLERCODE VS EXCEPTIONS

- Fehlercode
 - muss sofort geprüft werden
- Exception
 - kann am Ende behandelt werden
 - ist ebenfalls eine Aufgabe
 - kann in separate Funktion ausgelagert werden

Beispiel mit Fehlercodes inkl. Behandlung

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (ConfigKeys.deleteKey(page.name.makeKey()) == E_OK) {  
            logger.log("page deleted");  
        } else {  
            logger.log("config key not deleted");  
        }  
    } else {  
        logger.log("deleteReferences from registry failed");  
    }  
} else {  
    logger.log("delete failed");  
}
```

Beispiel mit Exceptionbehandlung

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    ConfigKeys.deleteKey(page.name.makeKey());
}
catch (Exception e)
{
    logger.log(e.getMessage());
}
```

Exceptionsbehandlung auslagern

```
public void delete(Page page) {  
    try {  
        deletePageAndAllReferences();  
    }  
    catch (Exception e)  
    {  
        logError(e);  
    }  
}  
  
public void deletePageAndAllReferences(Page page) {...}  
public void logError(Exception e) {...}
```

DON'T REPEAT YOURSELF

- Viele Innovationen der Software-Entwicklung haben nur ein Ziel
 - Duplizierung zu vermeiden
 - Wiederverwendung fördern
- Duplikate könnten bei einem Umbau vergessen werden
- Beispiel
 - [HtmlUtil.java](#)

KOMMENTARE

ÜBER KOMMENTARE

- Kommentare sind kein Ersatz für schlechten Code
- Kommentare können durch **selbsterklärenden** Code vermieden werden

```
// Check to see, if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    employee.age > 65)
    ...
```

Alternative

```
if (employee.isEligibleForFullBenefits())
    ...
    ...
```

GUTE KOMMENTARE

- Copyright Header
- nicht-triviale Methoden-Beschreibung
- nicht-triviale Klassen-Beschreibung
- Erklärung der Absichten
- Klarstellungen
- Warnung vor Konsequenzen
- TODO-Kommentare

SCHLECHTE KOMMENTARE

- Geraune
- Redundante Kommentare
 - *Wiederholung des Codes*
- irreführende Kommentare
- Positionsbezeichner
- Kommentare hinter schließenden Klammern
- Auskommentierter Code

Continuous Integration

INTEGRATIONS PROBLEME

INTEGRATION

Integration

Zusammenfügen von mehreren Komponenten zu einer Software

- **FALSCH:** dann würde es eher *Continuous Assembly* heißen

INTEGRATION

Integration

Zusammenfügen der (lokalen) Entwicklung
mehrerer Entwicklungszweige

INTEGRATION

Wenn mehrere Entwickler parallel am gleichen Code arbeiten:

Wie stellen wir sicher, dass die Software, die bisher jeder nur lokal erstellt hat, auch funktioniert wenn alle Änderungen zusammenfließen?

PROBLEM BEREICHE

Merge-Konflikte

Entwickler haben **gleichzeitig** die gleiche Datei bearbeitet

Kompilier-Konflikte

keine Merge-Konflikte, aber die **merged** Codebasis kompiliert nicht

Test-Konflikte

keine Merge-Konflikte, keine Kompilier-Konflikte, aber die **Tests** laufen nicht mehr erfolgreich

INTEGRATION

Integration

Code-Basis zweier Entwickler ineinander integrieren um alle Arten von Konflikten zu identifizieren.

AUTOMATISIERUNGS PROBLEME

PUR

HelloWorldApp.java

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Kompilieren

```
javac HelloWorldApp.java
```

Ausführen

```
java -cp . HelloWorldApp  
java -classpath . HelloWorldApp
```

PUR + BIBLIOTHEK

HelloWorldApp.java

```
import org.apache.commons.lang3.StringUtils;
class HelloWorldApp {
    public static void main(String[] args) {
        String msg = "Hello World!";
        msg = StringUtils.substring(msg, 6)
        System.out.println(msg);
    }
}
```

Kompilieren

```
$ javac -cp lib/commons-lang.jar HelloWorldApp.java
```

Ausführen

```
$ java -cp lib/commons-lang.jar;. HelloWorldApp
```

KOMPLIZIERTER

- Woher kommt commons-lang.jar?
- Welche Version von commons-lang.jar wird verwendet?
- Wie viele Bibliotheken verwendet dhw-painground insgesamt?
- Antwort: 112

AUTOMATISIERUNG

KOMPILEREN

Kommandozeile:

```
javac src/main/java/net/kleinschmager/dhbw/tfe15/painground/Paingroun  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/ui/MainUI.java  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/ui/views/Member  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/persistence/mod  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/persistence/rep
```

IDE: Menü *Projekt > Bereinigen*

VERPACKEN

Kommandozeile

```
jar cvmf painground.jar src/main/java/net/kleinschmager/dhbw/tfe15/pa  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/ui/MainUI.java  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/ui/views/Member  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/persistence/mod  
src/main/java/net/kleinschmager/dhbw/tfe15/painground/persistence/rep
```

IDE

- siehe nächste Folie

NACHTEILE

KOMMANDOZEILE IDE

- nicht übersichtlich
- unkomfortabel
- Abhängig von Umgebung
 - javac version
 - Bibliotheken
- *Works on my machine*
- Abhängig von Umgebung
 - javac version
 - Bibliotheken
 - IDE Konfiguration
- *Works on my machine*

WEITERE AUFGABEN

- Testen
- Dokumentation erzeugen
 - Word zu PDF?
 - xyz zu HTML?
- Upload zum Kunden
- Bereitstellen DEMO System

LÖSUNG: AUTOMATISIERUNG

- Build-Tools
 - Ant | Maven | Gradle | CMake
- Continuous Integration
 - Mindset
- Continuous Integration Tools
 - Jenkins
 - Travis-CI
 - Team Foundation Server

CONTINUOUS INTEGRATION

MOTIVATION

*In software, when something is painful,
the way to reduce the pain is to do it
more frequently, not less.*

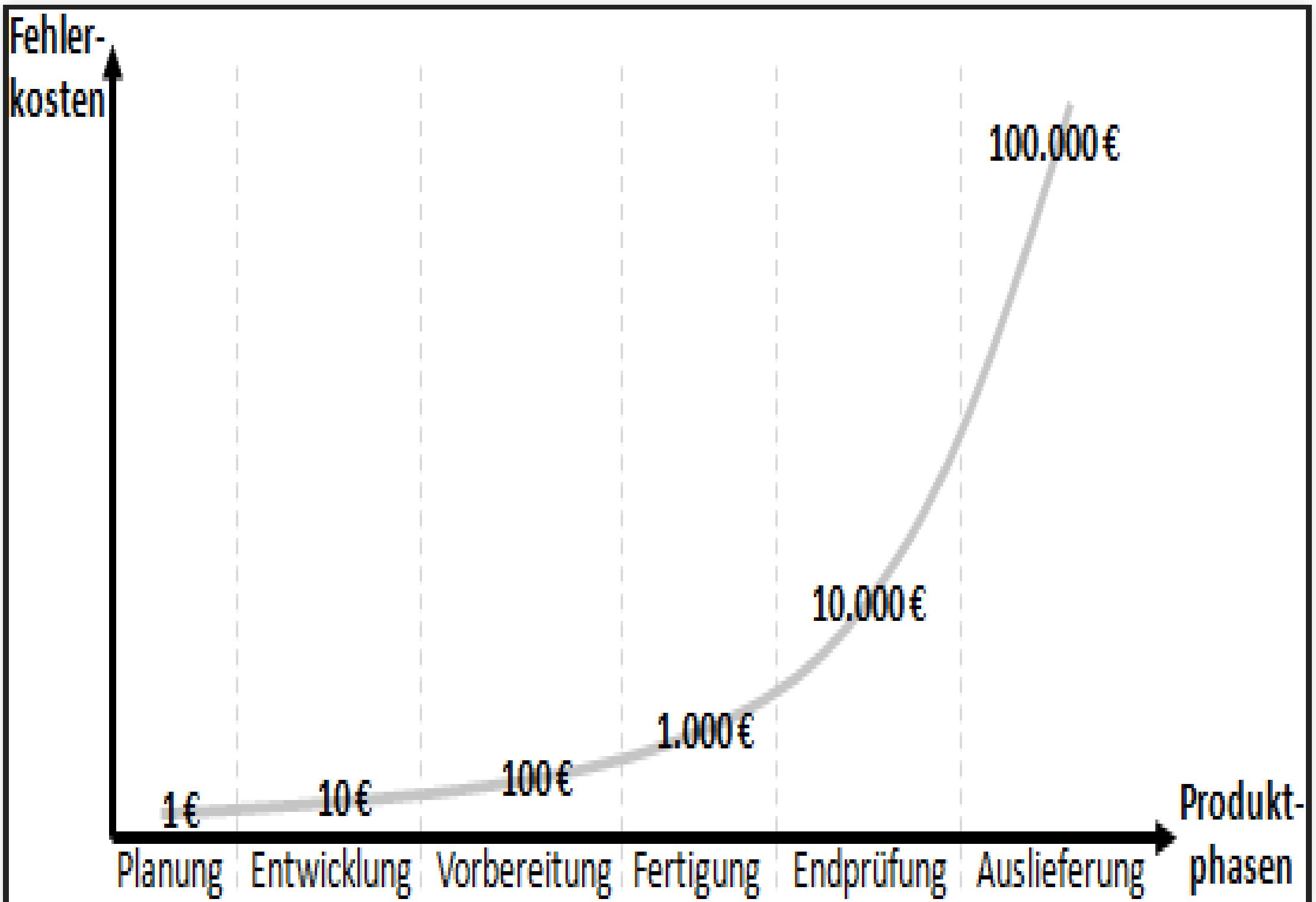
— David Farley

ELEMENTE

1. Code (und Konfiguration) stehen unter **Versionsverwaltung**
2. Build-Prozess ist **automatisiert**
3. Regelmäßiges **einchecken|commit**
 - mind. täglich

ELEMENTE

4. Tests werden gleichzeitig entwickelt (als Code)
 - stehen ebenfalls unter Versionsverwaltung
 - am besten im gleichen Repository wie der Code selbst
5. Wichtige Tests sollten bei jedem commit ausgeführt werden
 - andere wenigstens regelmäßig, z.B. nächtlich
6. eine produktionsnahe Testumgebung steht immer bereit
7. Einfacher Zugriff auf Ergebnisse auch für Nicht-Entwickler



VORTEILE

*Continuous Integration doesn't get rid
of bugs, but it does make them
dramatically easier to find and remove.*

— Martin Fowler

VORTEILE

Continuous Integration

regelmäßiges Kompilieren, Verpacken, Testen,
Bereitstellen einer Software

VORTEILE

- Fehler früher finden (Konflikte vermeiden)
- Feedback für das Entwickler-Team
- Feedback für das Qualitäts-Management
- Feedback für die Tester

CONTINUOUS DELIVERY

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

— Martin Fowler

CONTINUOUS DELIVERY

There should be two tasks for a human being to perform to deploy software into a development, test, or production environment: to pick the version and environment and to press the “deploy” button.

— David Farley

Testen

ARTEN VON TESTS

UNTERSCHIEDUNG

- Größe des Prüflings
- Aufwand für Testumgebung
- Anzahl der Anforderungen
 - Anzahl der Testfälle

DETAILS TESTEN

1. Unit Testing

- *Modul wird isoliert getestet*
 - eine Klasse oder
 - eine Gruppe zusammenhängender Klassen

2. Integrations Testing

- *Service-Test*
- Zusammenspiel mehrerer Module
- z.B. Datenbank & Importer-Modul

SMOKE TESTING

- alle wesentlichen Funktionen kurz ausprobieren
 - **keine** Detail-Tests
- Herkunft: Prüfen, dass das Gerät beim ersten Einschalten nicht brennt.

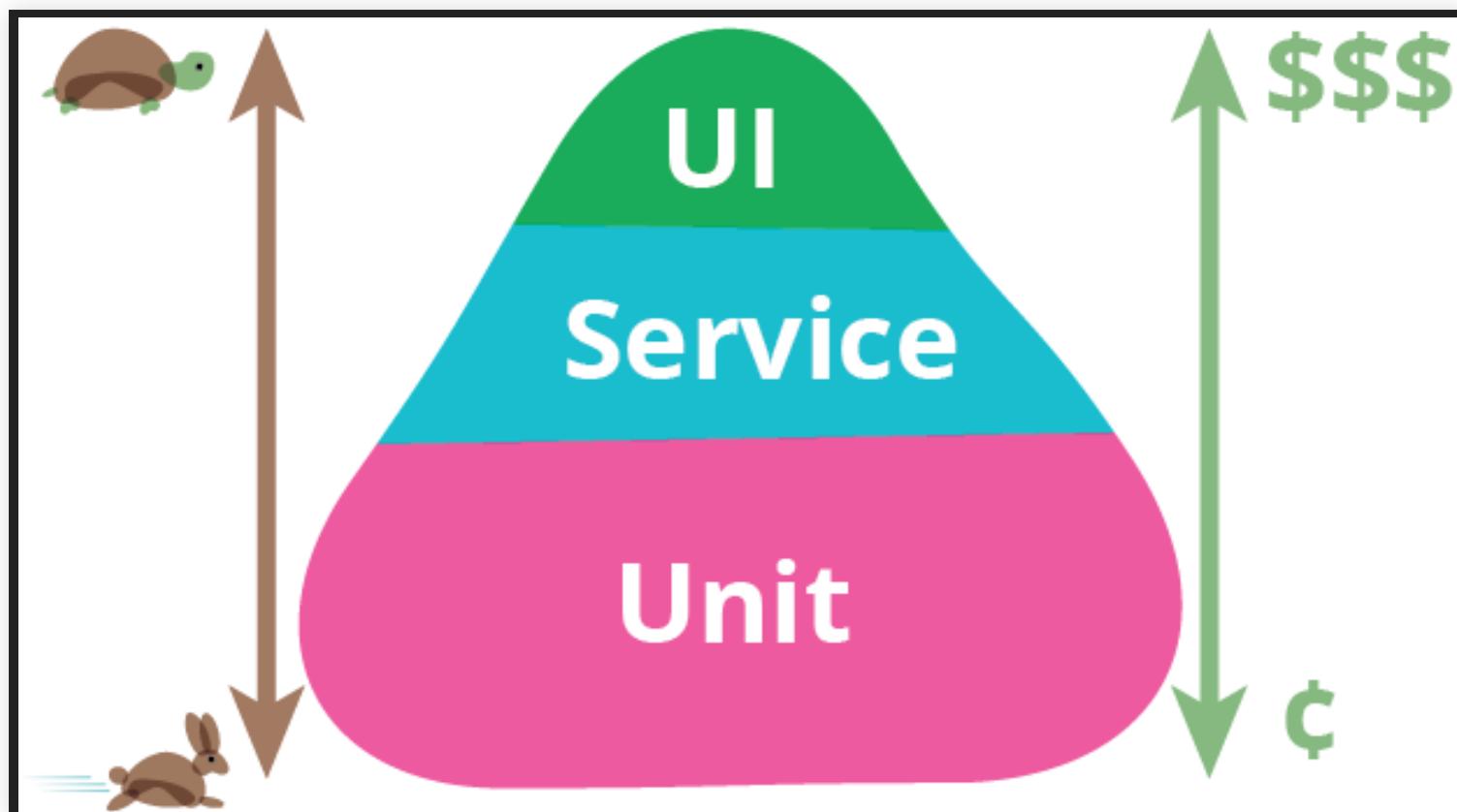
EXPLORATORY TESTING

- Tester lernt die Software beim Testen kennen
 - ad-hoc Entscheidungen, was getestet werden soll
- sinnvoll, wenn
 - keine/schlechte Spezifikation
 - wenig Zeit

END-TO-END TESTING

- Perspektive des Nutzers
- Zusammenspiel aller Module
- ggf. sehr umfangreiche Testumgebung
- Nachteile
 - langsam; spätes Feedback
 - Schwer, zu Pflegen
 - Folge: (meist) instabil
 - Schwer zu automatisieren
 - gefundene Fehler sind schwer zu lokalisieren

TESTPYRAMIDE



TESTTECHNIKEN

BLACK & WHITE

Wie kommen wir zu unseren Testfällen?

- Blackbox Tests
 - aus der Spezifikation/Anforderungen
- Whitebox Tests
 - durch Analyse der Codestruktur

TEST DRIVEN DEVELOPMENT

- Test First
- Anforderungen werden sofort/zuerst in Testfällen ausgedrückt
- es muss nur der Code geschrieben werden, der nötig ist um die Tests zu bestehen

KISS

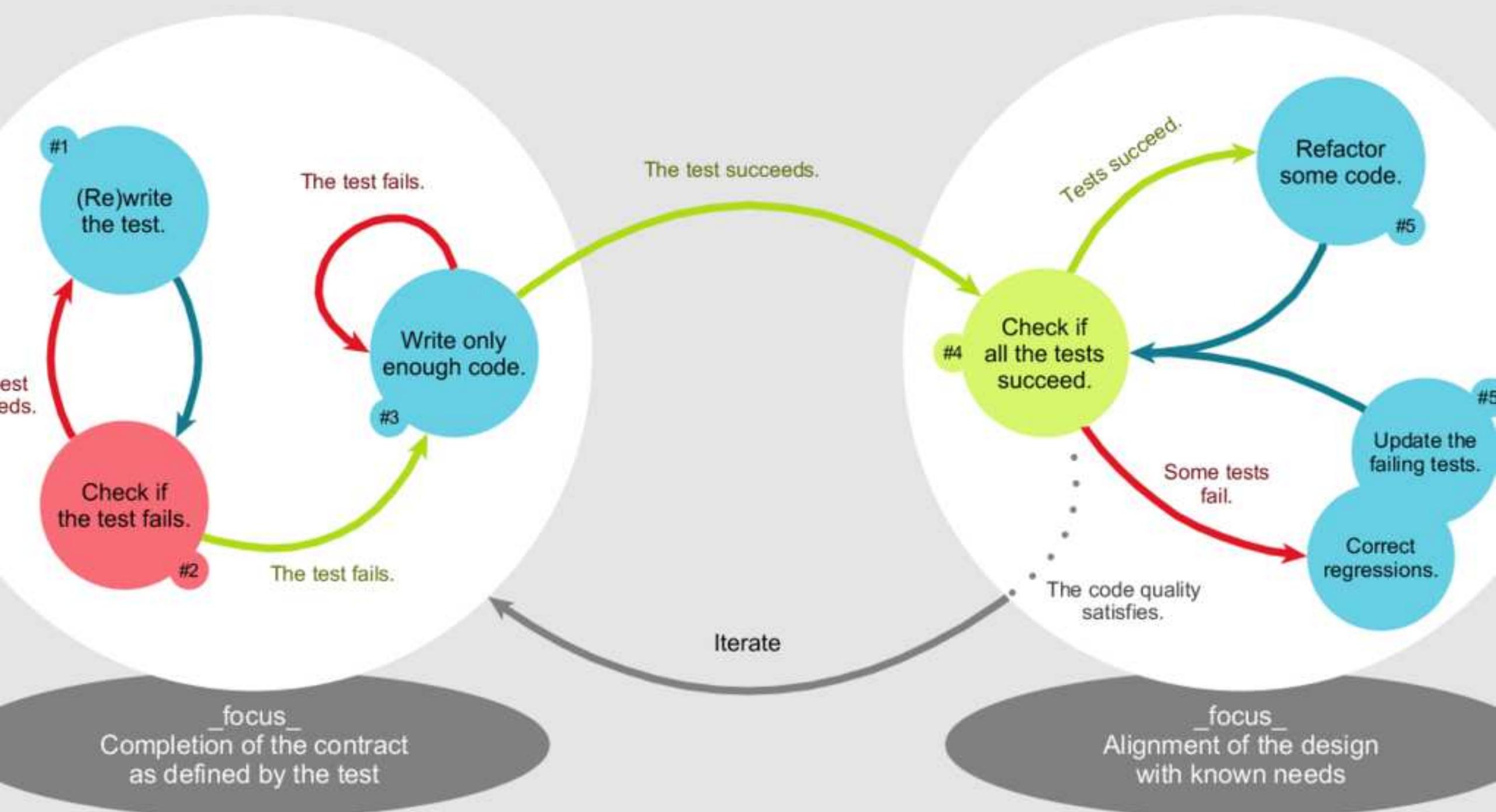
Keep it simple, stupid

YAGNI

You aren't gonna need it

TEST-FIRST DEVELOPMENT

REFACTORING



TEST DRIVEN DEVELOPMENT

- Vorteile
 - 1. TestCode beschreibt den getesteten Code selbst
 - 2. fördert kleine Module
 - 3. fördert **testbare** Software
 - 4. Code-Design wird modularer, flexibler
- Nachteile
 - 1. Blinde Flecken werden vom Coder & Tester nicht gesehen (gleiche Person)
 - 2. spätere Änderungen an Architektur sind aufwendig
 - 3. Testcode muss auch gewartet werden

TESTABDECKUNG

Wie viele Testfälle müssen geschrieben werden?

C0

Durchlauf jeder Anweisung

C1

Durchlauf jedes Zweiges, auch der leeren

```
int z = x;  
if (y > x) {  
    z = y;  
}  
z = z * 2;
```

TESTABDECKUNG

Wie viele Testfälle müssen geschrieben werden?

C2

Durchlauf aller möglichen Pfade; Schwierig bei Schleifen

```
if (y > x) {  
    z = y;  
} else {  
    z = x;  
}  
if (x == 2 || y == 2) {  
    z = z * 2;  
} else {  
    z = z * 4;  
}
```

TESTABDECKUNG

Wie viele Testfälle müssen geschrieben werden?

C3

Durchlauf mit allen möglichen Bedingungen

C3a

Jede atomare Bedingung einer Entscheidung muss einmal mit true und einmal mit false getestet werden.

C3b

Alle Kombinationen der atomare Bedingung einer Entscheidung müssen getestet werden.

TESTABDECKUNG

Wie viele Testfälle müssen geschrieben werden?

C3a

Jede atomare Bedingung einer Entscheidung muss einmal mit true und einmal mit false getestet werden.

```
if (x == 2 | y == 2) {  
    z = z * 2;  
} else {  
    z = z * 4;  
}
```

- zwei Testfälle x,y: 1,1 2,2

TESTABDECKUNG

Wie viele Testfälle müssen geschrieben werden?

C3b

Alle Kombinationen der atomare Bedingung einer Entscheidung müssen getestet werden.

```
if (x == 2 | y == 2) {  
    z = z * 2;  
} else {  
    z = z * 4;  
}
```

- vier Testfälle x,y: 1,2 3,2 3,1 2,2

TESTABDECKUNG

- 100% Coverage kein gutes Ziel
- Coverage allein reicht nicht aus
 - Tests müssen den Rückgabewert verifizieren

QUELLEN

- Bild: TDD Lifecycle; CC BY-SA 4.0

https://en.wikipedia.org/wiki/Test-driven_development

- Bild: test pyramid; Martin Fowler

<https://www.martinfowler.com/bliki/TestPyramid.html>

<https://www.martinfowler.com/bliki/images/testPyramid/test-pyramid.png>

Refactoring

REFACTORING

MOTIVATION

- Erhöhen der Lesbarkeit
- Reduzieren der Komplexität
- Erhöhen der Wartbarkeit
- Erhöhen der Erweiterbarkeit
- Erhöhen der Testbarkeit

WORT-HERKUNFT

Factoring

== De-Komposition; Zerlegen, Aufteilen von komplizierten Problemen in kleine Teile

Re-Factoring

Ändern der Zerlegung

BEDEUTUNG

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

— Martin Fowler

WICHTIGE ELEMENTE

1. Ändern der **internen Struktur**
2. **externes Verhalten bleibt gleich**
3. **diszipliniertes Vorgehen**
4. Viele **kleine Schritte**
 - Mikro-Refactoring
 - es kann weniger kaputt gehen

UNTERSTÜTZUNG

- (automatisierte) Tests
- IDE mit Support für Mikro-Refactorings
- Typsystem der Programmiersprache
 1. strenge Typisierung
 2. dynamische Typisierung

BEISPIELE REFACTORING

UMBENENNEN VON VARIABLEN

Vorher

```
public String getFullName(String s1, String s2) {  
  
    s1 = s1.trim();  
    s2 = s2.trim();  
  
    return s1 + " " + s2;  
}
```

Nachher

```
public String getFullName(String vorname, String nachname) {  
  
    vorname = vorname.trim();  
    nachname = nachname.trim();  
  
    return vorname + " " + nachname;  
}
```

EXTRAHIEREN VON METHODEN

```
@Bean
public CommandLineRunner loadData(MemberProfileRepository repository) {
    return (args) -> {
        /// STEP 1

        // save a couple of profiles
        repository.save(new MemberProfile("robkle", "Kleinschmager"));
        repository.save(new MemberProfile("mickni", "Knight"));
        repository.save(new MemberProfile("geolaf", "Laforge"));

        // STEP 2

        // fetch all profiles
        log.info("MemberProfiles found with findAll()");
        log.info("-----");
        for (MemberProfile profile : repository.findAll()) {
            log.info(profile.toString());
        }
        log.info("");

        // STEP 3

        // fetch an individual customer by ID
        MemberProfile profile = repository.findOne(1L);
        log.info("Profile found with findOne(1L)");
        log.info("-----");
        log.info(profile.toString());
        log.info("");

    };
}
```

EXTRAHIEREN VON METHODEN 2

```
@Bean
public CommandLineRunner loadData(MemberProfileRepository repository) {
    return args -> {
        deleteAllExistingProfiles(repository);
        importProfiles(repository);
        fetchAndPrintAllProfiles(repository);
    };
}
```

VARIABLE IN OBJECT UMWANDELN

Vorher

```
public class Order {  
  
    String customer;  
    List<Item> items;  
}
```

Nachher

```
public class Order {  
  
    Customer customer;  
    List<Item> items;  
}  
  
public class Customer {  
    String name;  
}
```

CODE FORMATTIERUNG

Vorher

```
public boolean equals(Object obj) {  
  
    if (  
        this == obj) return true;  
    if (!(obj instanceof MemberProfile)) {  
        return false; }  
    MemberProfile that = (MemberProfile) obj;  
    EqualsBuilder eb = new EqualsBuilder();  
    eb.append(this.getMemberId(), that.getMemberId());  
  
    return eb.isEquals();  
}
```

CODE FORMATTIERUNG

Nachher

```
public boolean equals(Object obj) {  
  
    if ( this == obj) {  
        return true;  
    }  
  
    if (!(obj instanceof MemberProfile)) {  
        return false;  
    }  
  
    MemberProfile that = (MemberProfile) obj;  
    EqualsBuilder eb = new EqualsBuilder();  
    eb.append(this.getMemberId(), that.getMemberId());  
  
    return eb.isEquals();
```

BESCHREIBENDE VARIABLEN

Vorher

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 )
{
    // do something
}
```

Nachher

```
boolean isMacOs      = platform.toUpperCase().indexOf("MAC") > -1;
boolean isIEBrowser = browser.toUpperCase().indexOf("IE")   > -1;
boolean wasResized  = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
    // do something
}
```

VORHANDENES OBJECT ÜBERGEBEN

Vorher

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();

withinPlan = plan.withinRange(low, high);
```

Nachher

```
withinPlan = plan.withinRange(daysTempRange());
```

QUELLEN

- <https://www.refactoring.com>

Requirements Engineering

BEGRIFFE

Anforderung ist eine Bedingung oder Fähigkeit ...

- die von einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird
- die ein System oder Systemteile erfüllen oder besitzen muss, um einen Vertrag zu erfüllen oder einer Norm, einer Spezifikation oder anderen, formell vorgegebenen Dokumenten zu entsprechen

Funktionale Anforderungen

- Was soll ein Produkt tun
 - Funktionen, Verhalten, Strukturen (Daten, Abhängigkeiten in einem System)

Qualitätsanforderungen

- Wie gut soll ein Produkt seine Leistung erbringen
- non-functional-requirement
- Performance, Verfügbarkeit, Zuverlässigkeit, ...

Rahmenbedingungen

- können (von den Beteiligten) nicht verändert werden
- werden nicht umgesetzt, sondern schränken die Umsetzungsmöglichkeiten ein

Stakeholder

- Projektbetroffener
- Quelle für Anforderungen
- direkt: Nutzer, Administratoren
- indirekt: Management, Hacker, Gesetze

Requirements Engineering

- Hauptaufgaben
 - Ermitteln der Anforderungen
 - Dokumentieren der Anforderungen
 - Prüfen und Abstimmen der Anforderungen
 - Verwalten der Anforderungen
- Vorgehensweise
 - kooperativ, iterativ, inkrementell
 - während des gesamten Lebenszyklus des Systems

ZIELE DES REQUIREMENTS ENGINEERING

- alle relevanten Anforderungen sind **bekannt** und **verstanden**
- alle Stakeholder **stimmen** allen Anforderungen **zu** (Übereinstimmung)
- Alle Anforderungen sind
 - standardkonform **dokumentiert**
 - standardkonform **spezifiziert**

Warum gutes Requirements Engineering?

*60% der Fehler in
Softwareentwicklungsprojekten
enstehen bereits im Requirements
Engineering*

— B. Boehm

URSACHEN FÜR FEHLER

- Anforderungen fehlen
- Anforderungen sind unklar formuliert
- Fehlerhafte Anforderungen erscheinen (trotz der Fehler) subjektiv schlüssig (für den Entwickler) oder werden (vom Entwickler) unbewusst vervollständigt

GEGENMASSNAHMEN

1. Grenzen kennen
2. Fallen vermeiden
3. Detektiv sein
4. gemeinsame Sprache entwickeln
5. strukturiert arbeiten
6. Qualitätsprüfung der Anforderungen
7. ein Rahmen für das Ganze

REQUIREMENTS ENGINEERING

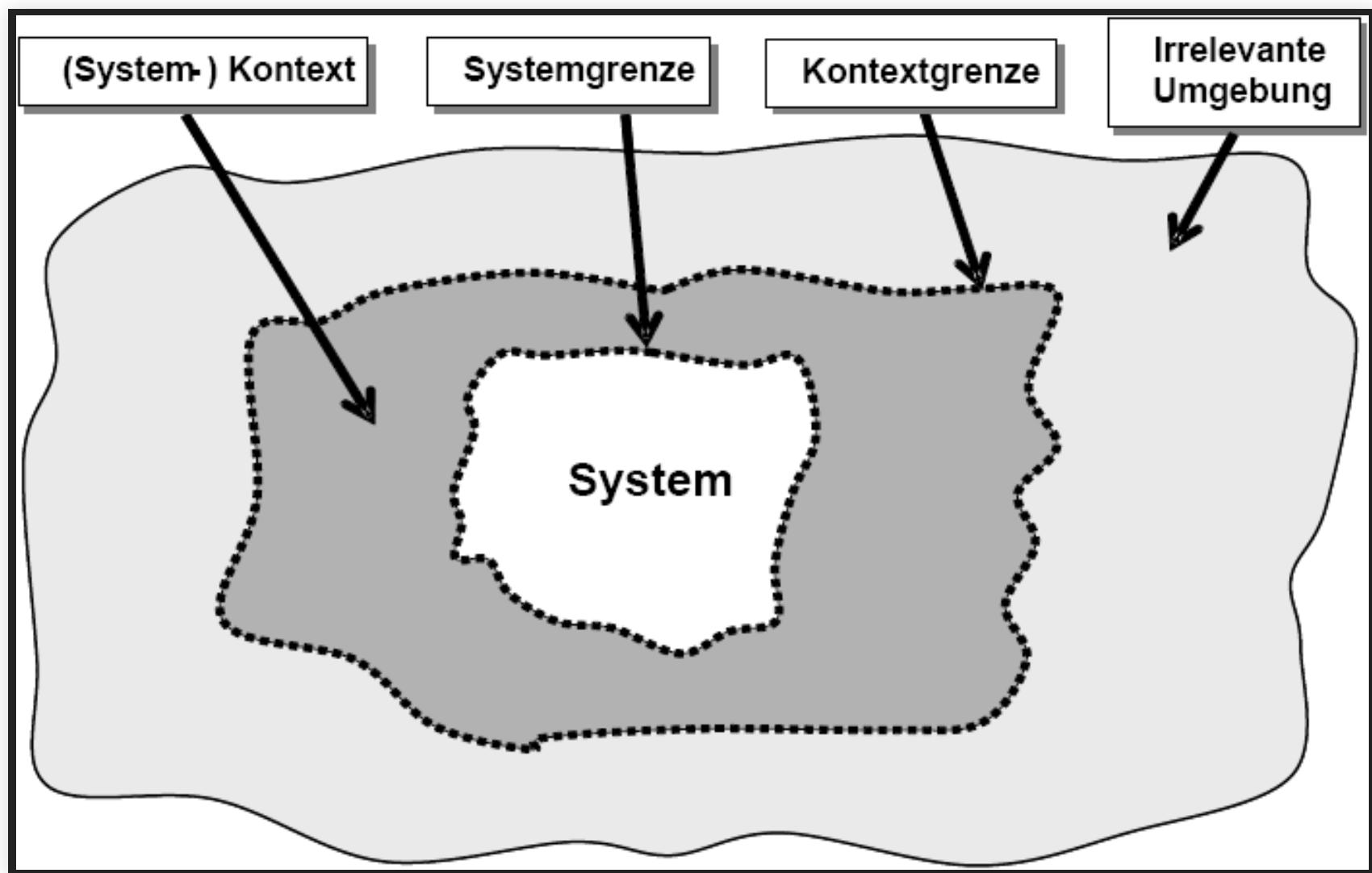
GRENZEN KENNEN

Systemkontext

Alle Aspekte, die eine Beziehung zu dem System haben

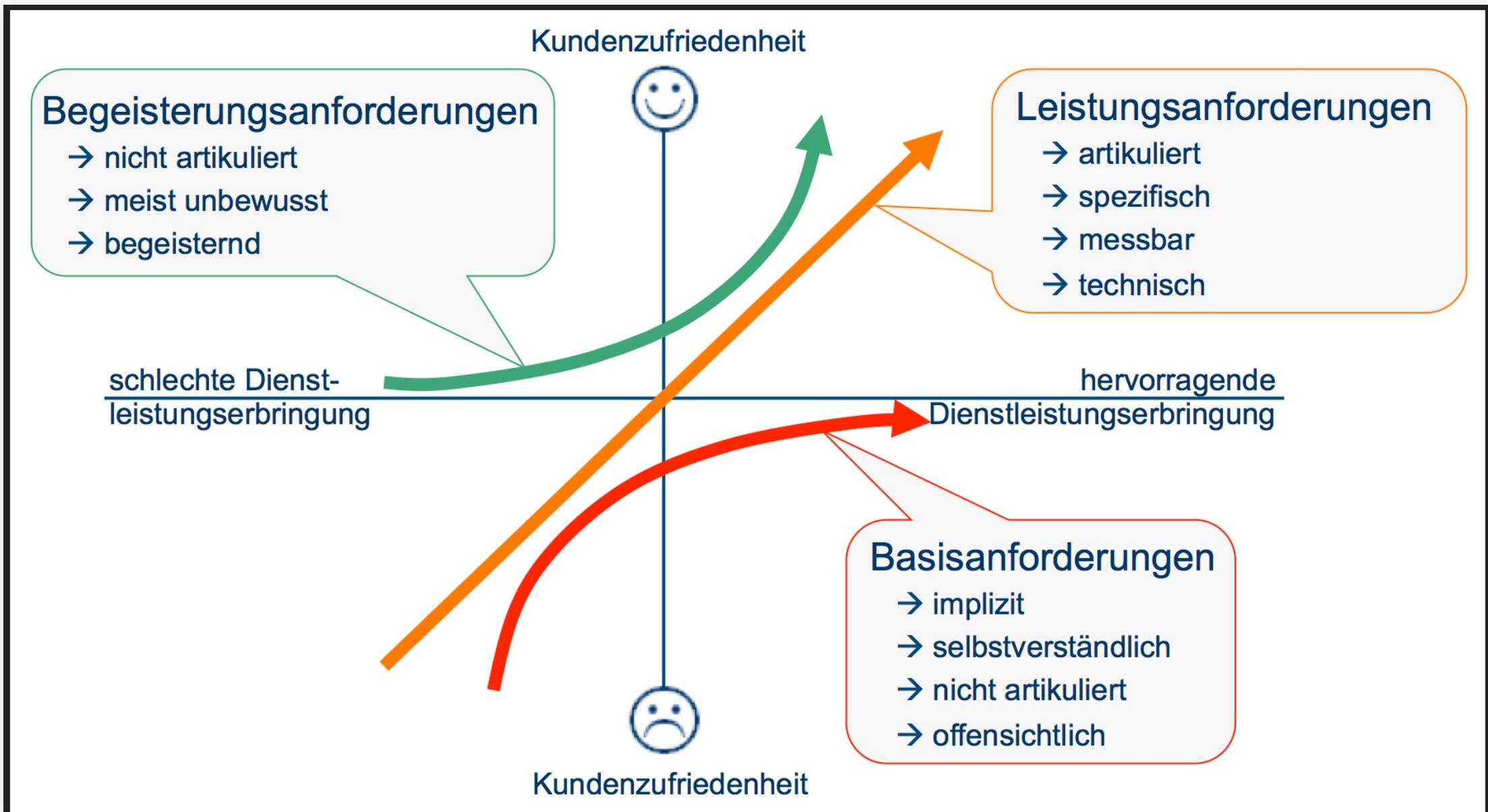
- Personen
- Systeme (Hardware oder andere Software)
- Prozesse, Geschäftsprozesse
- Ereignisse
- Dokumente (Gesetze, Standards)

SYSTEM- UND KONTEXTGRENZEN



FALLEN VERMEIDEN

- Stakeholder systematisch identifizieren und einbeziehen
- aus Projektbetroffenen sollen Projektbeteiligte werden
 - Individuelle "Verträge"



FALLEN VERMEIDEN - KANO

Welche Bedeutung hat ein Anforderung für die Zufriedenheit eines Stakeholders?

- Unterscheidung: unterbewusst, unbewusst, bewusst
- mit der Zeit werden Begeisterungsanforderungen zu Leistungsanforderungen und später zu Basisanforderungen

DETEKTIV SEIN

- Kommunikations-Geschick im Umgang mit dem Stakeholder
- Auswahl der richtigen **Ermittlungstechnik**
 - Befragungstechniken (Interview, Fragebogen)
 - Kreativitätstechniken (Brainstorming, Brainstorming Paradox, Perspektivenwechsel, Analogietechnik/Bisoziation)
 - Beobachtungstechniken (Feldbeobachtung, Apprenticing)

GEMEINSAME SPRACHE

- Erstellung eines Glossars
 - Fachbegriffe, Abkürzungen, Synonyme
 - alltägliche Begriffe, die im Kontext eine andere Bedeutung haben
- Verwalten des Glossars
 - ein Verantwortlicher
 - zentral zugänglich

STRUKTUR & DOKUMENTATION

Was muss dokumentiert werden?

- Stakeholder
- Systemkontext
- Glossar
- Nutzer und Zielgruppen
- Annahmen
- Alle Anforderungen

QUALITÄTSKRITERIEN

Anforderungsdokument muss

- Eindeutig und Konsistent sein
 - jede einzelne Anforderung
 - **kein Widerspruch** zwischen den Anforderungen
 - **identifizierbar** (Dokument & jede Anforderung)
- Klare Struktur haben
- Modifizierbar und Erweiterbar sein
- Vollständig
- Verfolgbar sein

WIE DOKUMENTIEREN ?

Natürliche Sprache

- "Transformation im Kopf" findet immer statt
- ggf. Satzschablonen verwenden
- Kurze Sätze, kurze Absätze
- nur eine Anforderung pro Satz
 - Aktiv formulieren, nur ein Prozesswort (Verb)

NATÜRLICHE SPRACHE - BEISPIEL

Zur Anmeldung des Benutzers werden die Login-Daten eingegeben

Das System soll dem Benutzer ermöglichen,
seinen Usernamen und sein Passwort
über die Tastatur
am Terminal einzugeben.

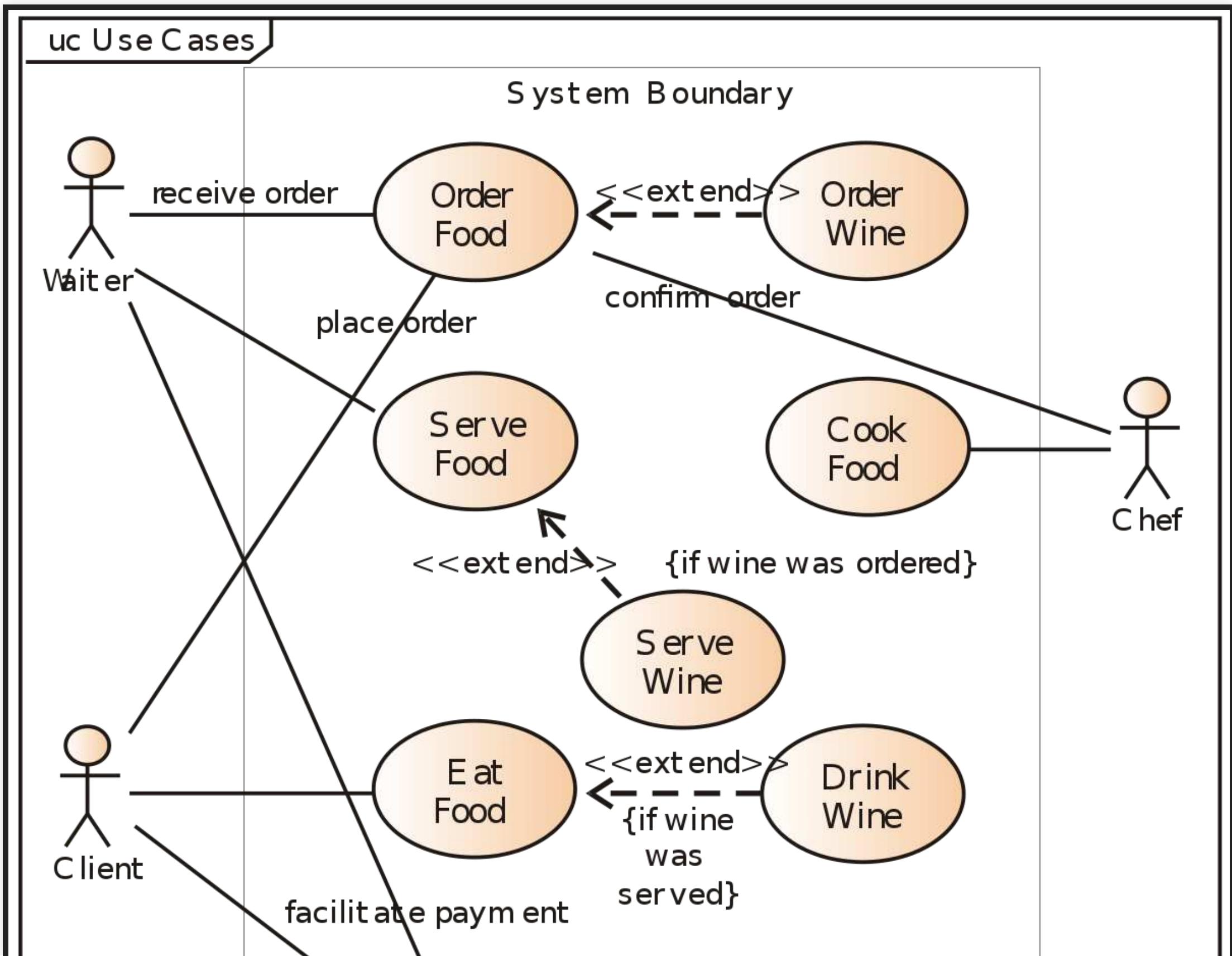
SATZSCHABLOKEN - BEISPIEL

Als <Rolle> möchte ich <Ziel/Wunsch>, um <Nutzen>

WIE DOKUMENTIEREN ?

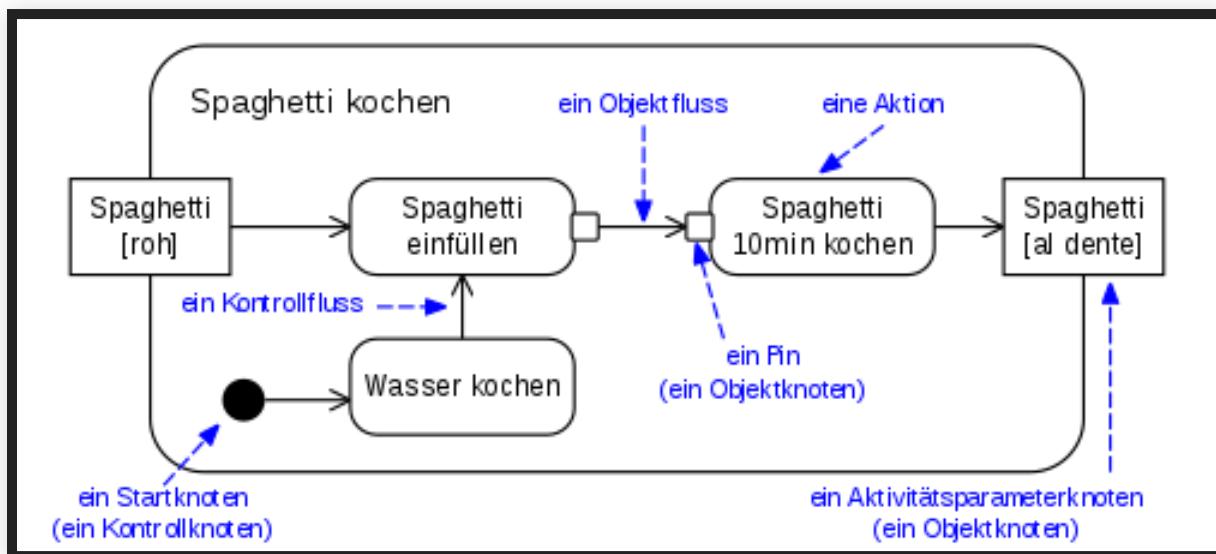
Modellbasiert

- UML
 - Use-Case-Diagramme
 - Datenflussdiagramme
 - Aktivitätsdiagramme



DATENFLUSSDIAGRAM

AKTIVITÄTSGRAMM



QUALITÄTSKRITERIEN FÜR ANFORDERUNGEN

REQUIREMENTS-MANAGEMENT ALS RAHMEN

Verwalten und Kontrollieren aller Aufgaben des Requirements Engineering während der komplette Lebenszeit des Produktes.

- Attributierung der Anforderungen
- Priorisierung der Anforderungen
- Verfolgbarkeit der Anforderungen
- Versionierung der Anforderungen

ATTRIBUTIERUNG BEISPIEL

Attributname	Belegung des Attributs (Attributwert)
Identifikator	Req-10
Name	
Dynamische Stauumfahrung	
Anforderungsbeschreibung	
Das System soll beim Auftreten von Verkehrsbehinderungen, die einen konfigurierbaren Kritikalitätswert übersteigen, selbständig eine Ausweichroute berechnen.	
Stabilität	Verantwortlicher
gefestigt	P. Müller
Quelle	Autor
Produktmanagement	B. Wagner

VERFOLGBARKEIT

Eine Anforderung ist nachvollziehbar, wenn sowohl deren Ursprung als auch deren Umsetzung und die Beziehung zu anderen Dokumenten nachvollziehbar ist.

Andere Dokumente: Commit-Historie, Testplan, Testprotokoll

VORTEILE VERFOLGBARKEIT

- Nachweisbarkeit
- Identifikation von Goldrandlösung
- Auswirkungsanalyse
- Zuordnung von Entwicklungsaufwänden

QUELLEN BILDER

- Kontextabgrenzung <http://docplayer.org/docs-images/24/4428614/images/7-0.png>
- Kano Modell <http://smallthingsmatter.ch/kano/>
- UseCase Diagram https://en.wikipedia.org/wiki/Use_case_diagram
- Datenflussdiagramm <http://www.ritz-dv.de/beratungsangebot/systemanalyseabb.php>
- Aktivitätsdiagramm <https://de.wikipedia.org/wiki/Aktivit%C3%A4tsdiagramm>
- Attributierung von Anforderungen "Basiswissen Requirements Engineering" - Pohl, Rupp

Tipps

GIT

EDITOR FÜR COMMIT-NACHRICHTEN

- Windows & Notepad++
 - erspart Editor in der Konsole
 - bei `git commit` kann das `-m` nun weggelassen werden

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

LÖSUNGEN

für Aufgabe bringt euer Remote-Repo auf den neuesten Stand

```
$ git fetch upstream  
$ git merge upstream/master  
$ git push
```

LÖSUNGEN

falls das `mergen` nicht automatisch funktioniert

- Datei manuell bearbeiten, danach

```
$ git add <dateiname>
$ git commit -m "<informative nachricht>"
```

- Hilfe:

http://genomewiki.ucsc.edu/index.php/Resolving_merg

ECLIPSE

- Suche nach Dateien
 - **strg-shift-r** (Windows, Linux)
 - **command-shift-r** (Mac)