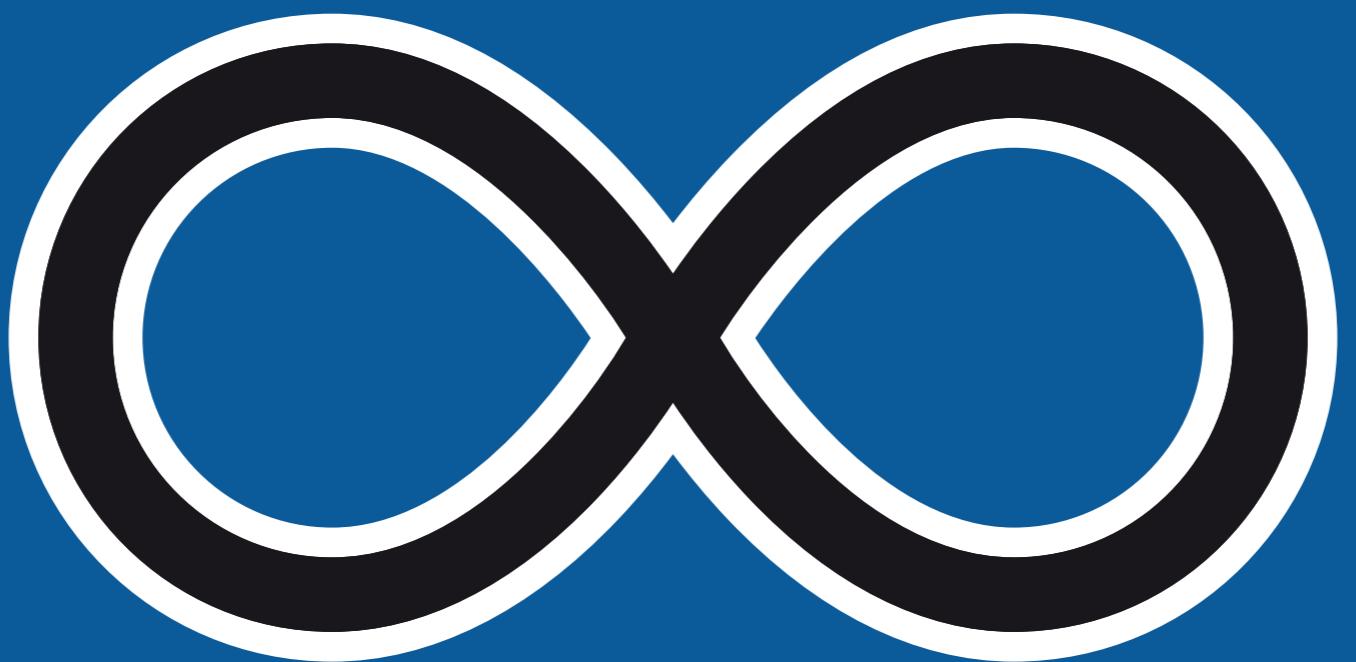


# SOFTWAREENTWICKLUNG

## IM TEAM MIT OPEN-SOURCE-WERKZEUGEN

---

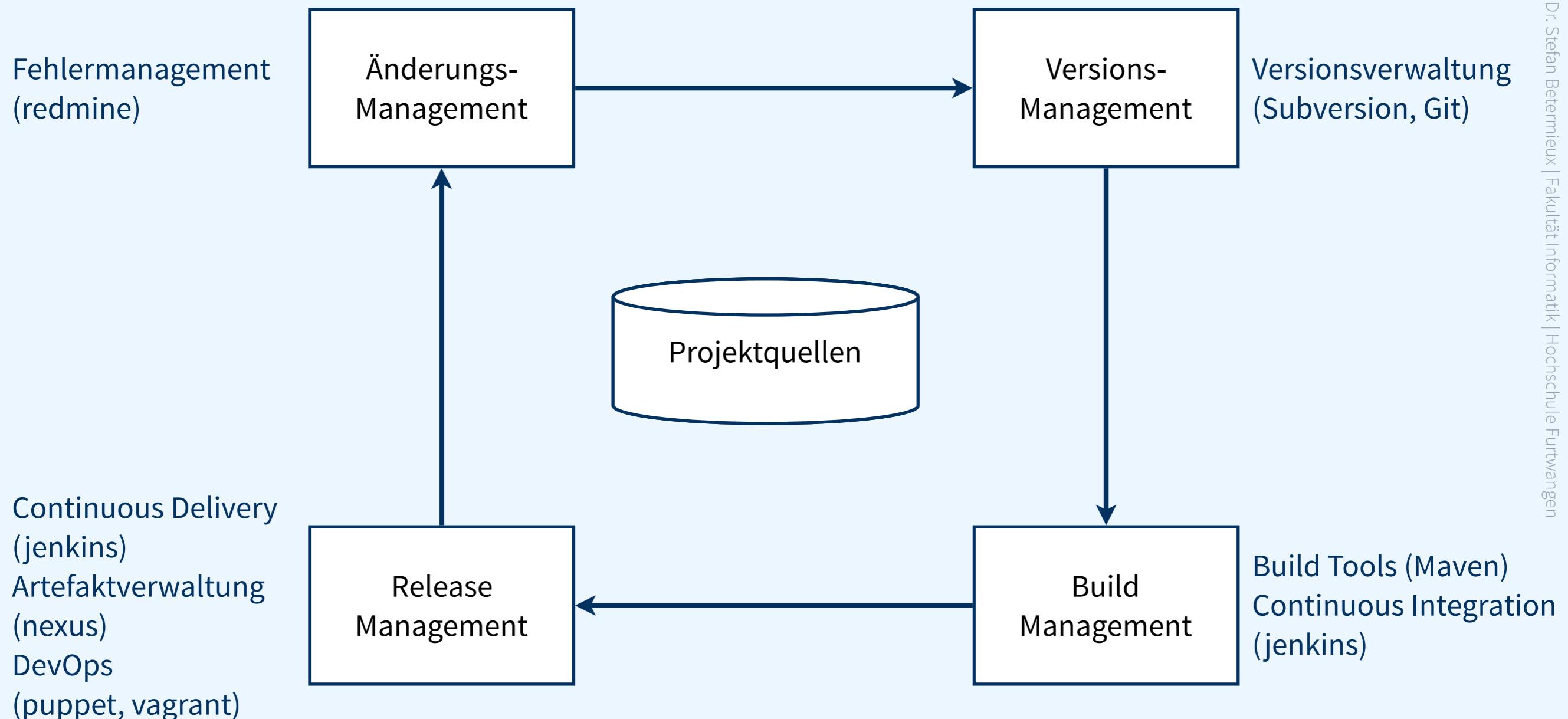
03 - verteiltes Versionsmanagement



# WIEDERHOLUNG

# Konfigurationsmanagement

(pragmatisch, mit Open-Source-Werkzeugen)



# Verzweigung Beispiel

- Branches-Verzeichnis im Repository erzeugen
- Trunk-Verzeichnis in ein neues Branches-Unterverzeichnis kopieren
- Arbeitskopie aus dem Branches-Unterverzeichnis auschecken

```
/$ svn mkdir file:///Users/stefan/Temp/Repository/branches -m "branches erzeugt"
```

Committed revision 6.

```
/$ svn copy file:///Users/stefan/Temp/Repository/trunk file:///Users/stefan/Temp/Repository/branches/alice-zweig -m "alice zweig kopiert"
```

Committed revision 7.

```
/ $ svn checkout file:///Users/stefan/Temp/Repository/branches/alice-zweig ProjektA  
A ProjektA/brief.txt  
Checked out revision 7.
```

Zweig für Alice's langen Absatz

Trunk

Bob arbeitet weiter auf dem trunk

↑r7

4

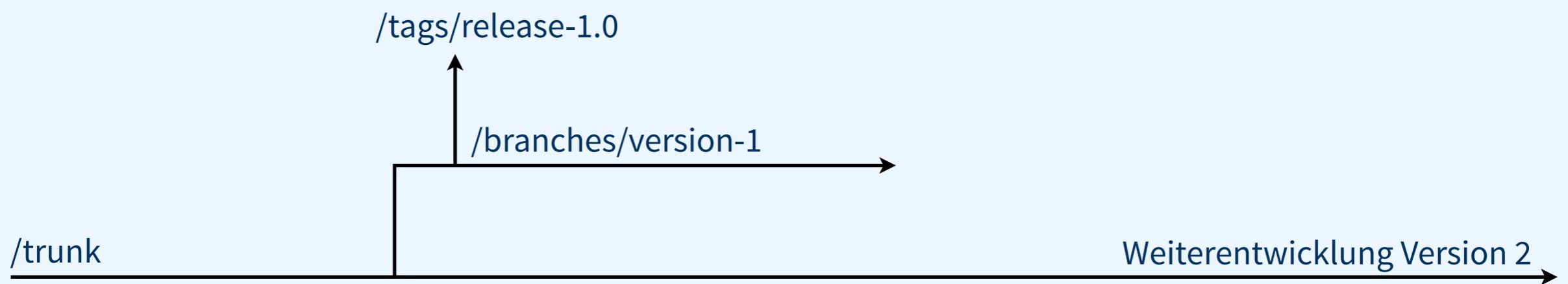
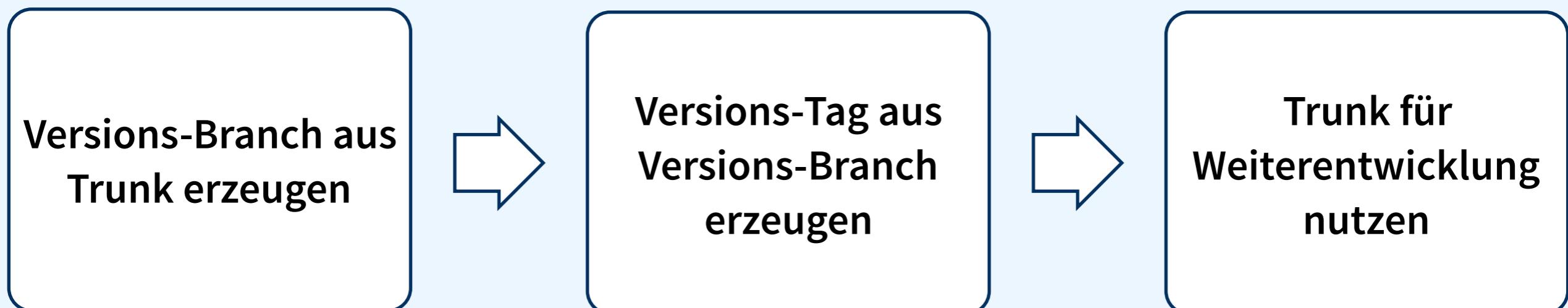
# Re-Integration

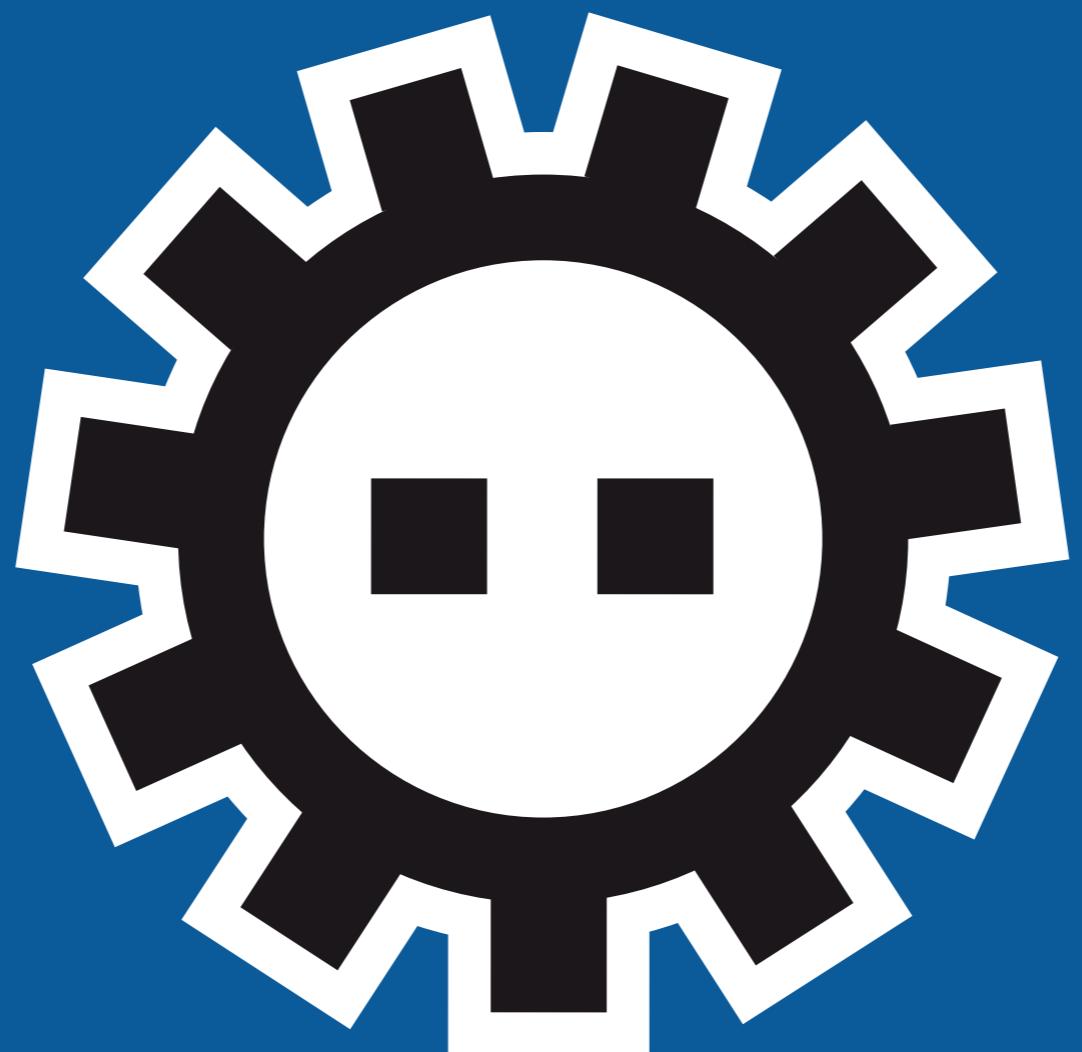
- Zielzweig als Arbeitskopie auschecken
- svn merge mit dem Quellzweig aufrufen
- Änderungen einchecken und branch löschen

```
/$ cd ProjektA
/ProjektA $ emacs brief.txt
/ProjektA $ svn commit -m "Änderungen im Zweig"
Sending      brief.txt
Transmitting file data .
Committed revision 8.
/ProjektA $ cd .. && rm -rf ProjektA
/$ svn checkout file:///Users/stefan/Temp/Repository/trunk ProjektA
A  ProjektA/brief.txt
Checked out revision 8.
/$ cd ProjektA
/ProjektA $ svn merge --reintegrate file:///Users/stefan/Temp/Repository/branches/alice-
zweig
--- Merging differences between repository URLs into '.':
U  brief.txt
/ProjektA $ svn commit -m "Alice Zweig re-integriert"
Sending      .
Sending      brief.txt
Transmitting file data .
Committed revision 9.
```

# Release

Als Release bezeichnet man die Operationen, die im Vorfeld einer Veröffentlichung ausgeführt werden sollten:





# MOTIVATION

# zentrale Versionskontrolle

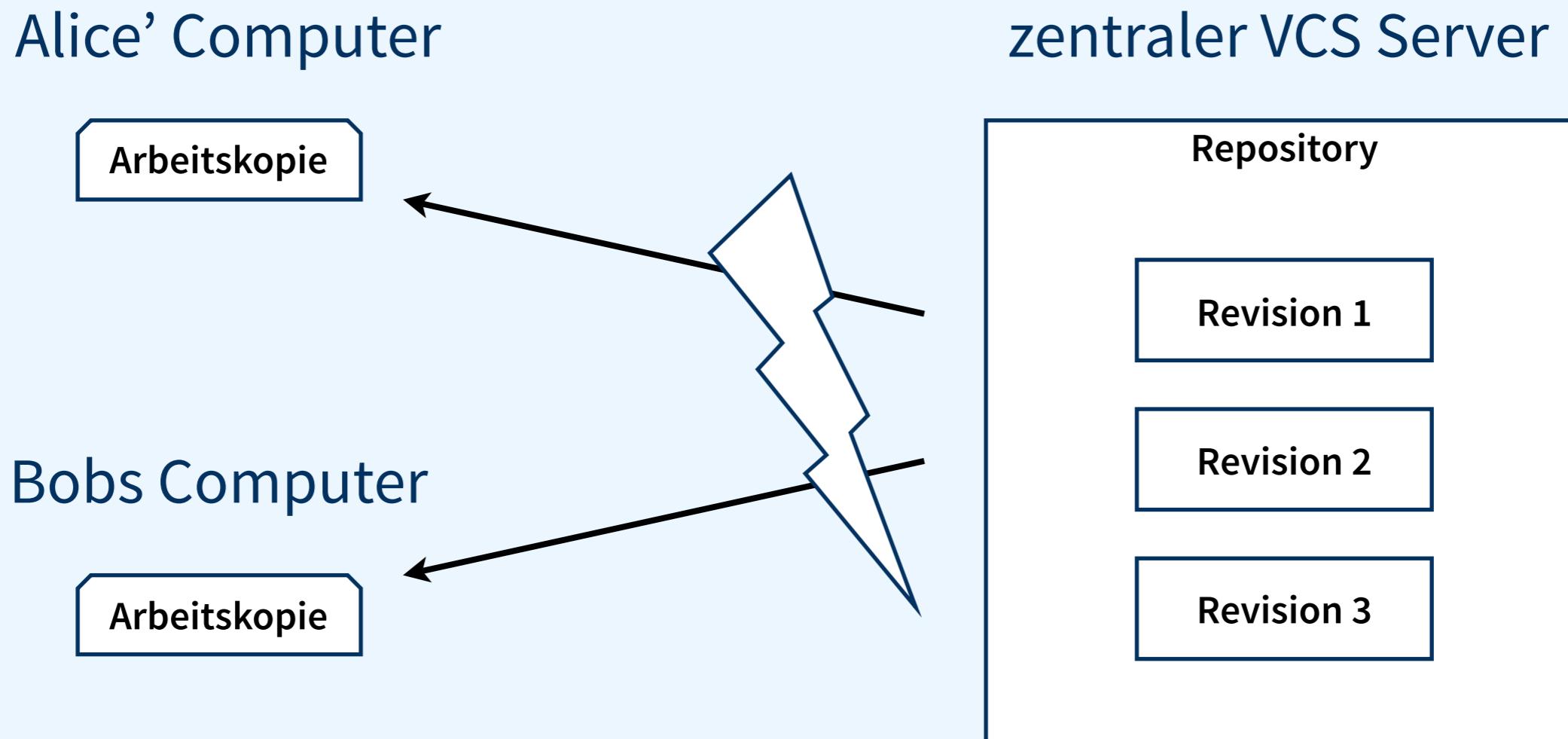
## Stand der Dinge:

- Dateien werden im zentralen Repository versioniert
- Nutzer können Arbeitskopien auschecken
  - ▶ lokale verändern
  - ▶ Änderungen wieder hochladen

## Nachteile:

- Server ist Single Point of Failure
  - ▶ was passiert, wenn der Server einen Tag ausfällt?
  - ▶ was passiert, wenn der Server für immer ausfällt?

# zentrale Versionskontrolle

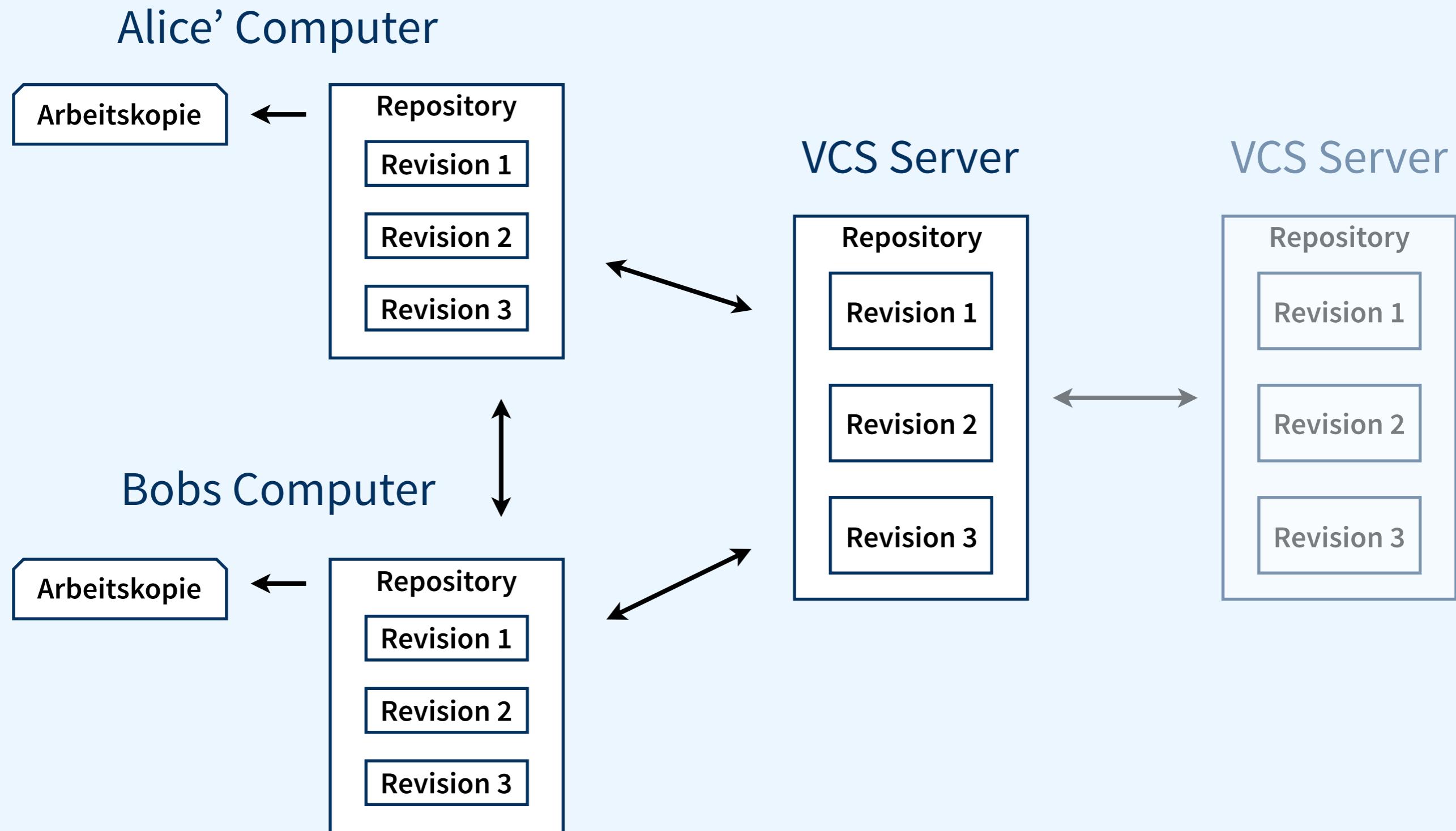


# verteilte Versionskontrolle

## Lösung:

- zentrales VCS durch verteiltes VCS ersetzen
- Nutzer holen sich keine Arbeitskopie
  - ▶ sondern eine Kopie des gesamten Repository
- Nutzer kann auch ohne Netzwerkverbindung arbeiten
- Wenn der Server komplett ausfällt
  - ▶ Repository von einem Nutzer wieder auf den Server kopieren
- Hierarchische Repository-Strukturen sind möglich!

# verteilte Versionskontrolle

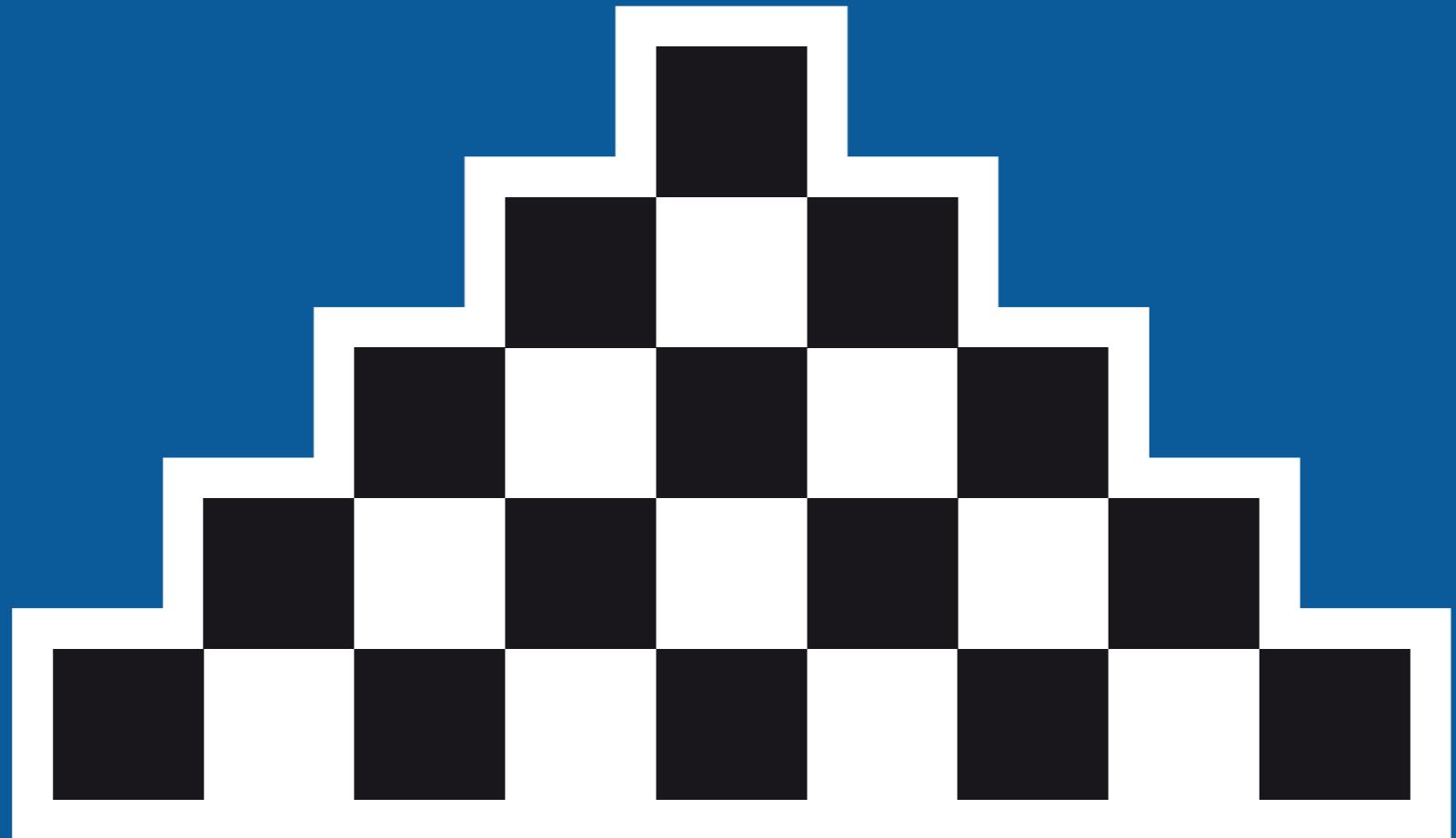


# Konsequenzen

- Das lokale Repository ist ein Klon des Originals
  - ▶ kann man sich auch als Branch vorstellen
  - ▶ welcher allerdings entkoppelt ist

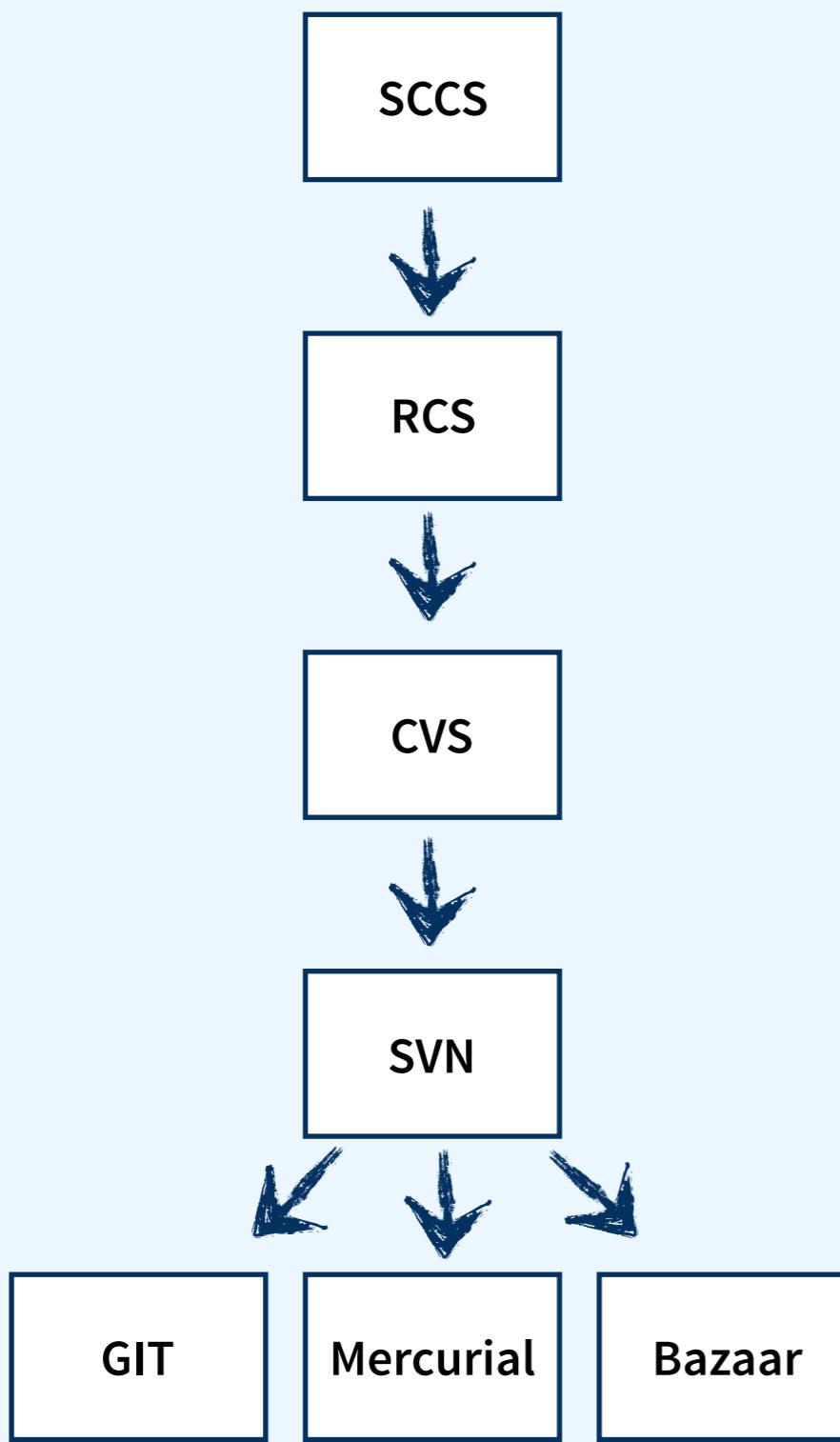
## Vorteile:

- Kein Netzwerkverbindung mehr erforderlich (z.B. auf Reisen)
- Für kleine Teams ist kein Server mehr notwendig
  - ▶ Kommunikation kann auch peer-to-peer erfolgen
- Der Nutzer hat volle Rechte auf dem lokalen Repository
  - ▶ beliebige commits, branches, merges, etc...
- Austausch zwischen Repositories entsprechen einem Merge



# GRUNDLAGEN

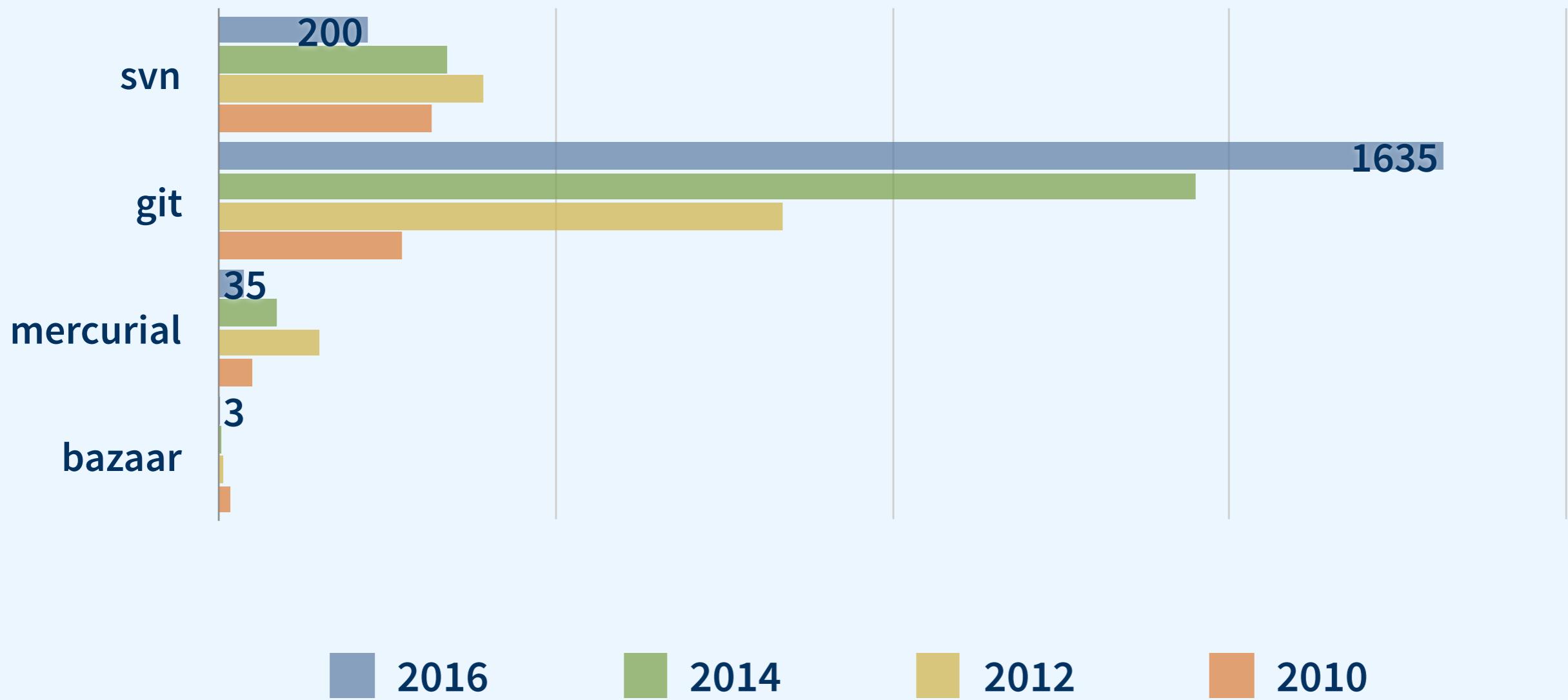
# Historie



- 1972
- Source Code Control System
- Verwaltet nur einzelne Dateien
  
- 1980
- Revision Control System
- hohe Ähnlichkeit mit SCCS
  
- 1989
- Concurrent Versions System
- kann Projekte verwalten
  
- 2000
- Subversion
- bessere Unterstützung für Verzeichnisse
  
- seit 2007
- verteilte Versionsverwaltung
- konkurrierende Ansätze

# Vergleich

Anzahl Fragen pro Monat auf der Entwicklerseite [stackoverflow.com](http://stackoverflow.com):



Ein Vergleich der technischen Fähigkeiten findet sich unter:

<http://www.techtatva.com/2010/09/git-mercurial-and-bazaar-a-comparison/>

# Geschichte von git

»das Ergebnis einer Trotzreaktion«

- Der Linux-Kernel ist ein großes Softwareprojekt
  - ▶ viel Code (>17 Mio. Zeilen)
  - ▶ viele Entwickler (> 1000)
- Von 1991 – 2002 pflegte Linus Torvalds den Quellcode lokal
  - ▶ andere Entwickler schickten Patches
- Von 2002 – 2005 setzte man auf das kommerzielle Bitkeeper
  - ▶ anfangs kostenlos nutzbar
  - ▶ später entzog Bitkeeper die Nutzungslizenz
- Torvalds war pragmatisch und mit Bitkeeper zufrieden
  - ▶ er entwickelte eine Open-Source-Variante → GIT

# Git Funktionsweise

- (Fast) jede Operation ist lokal
  - ▶ hohe Geschwindigkeit, keine Netzwerklatenz
- Git speichert Dokumente im Repository als Snapshots
  - ▶ nächste Folie
- Git stellt Integrität der Dokumente sicher
  - ▶ atomare Commits
  - ▶ Hashwerte der Dokumente

# Repository Organisation

Subversion  
Deltas

brief.txt  
V:1  
komplett

brief.txt  
V:2  
komplett

brief.txt  
V:3  
komplett

Git  
Snapshots

brief.txt  
V:1  
komplett

brief.txt  
V:2  
komplett

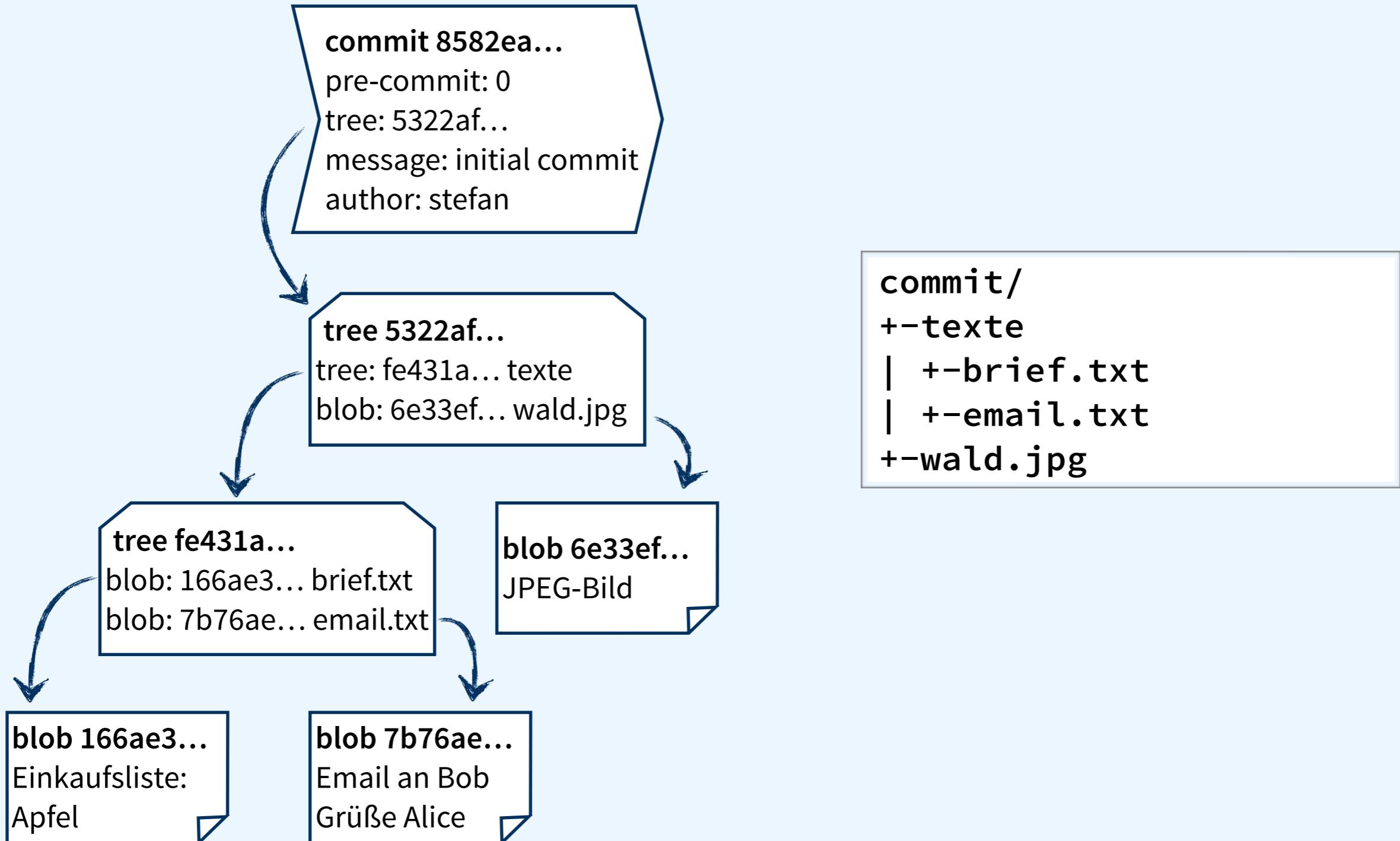
brief.txt  
V:3  
komplett

- Intern speichert Git:
  - ▶ bei Änderungen Datei komplett
  - ▶ ohne Änderungen nur ein Link auf Vorgänger
- Vorteil: Geschwindigkeit
  - ▶ um Revision 3 zu erreichen müssen nicht 3 Dateien kombiniert werden
- Nachteil: größerer Speicherplatzbedarf
  - ▶ git-intern wird aber optimiert
- Git-Repository ist Mini-Dateisystem

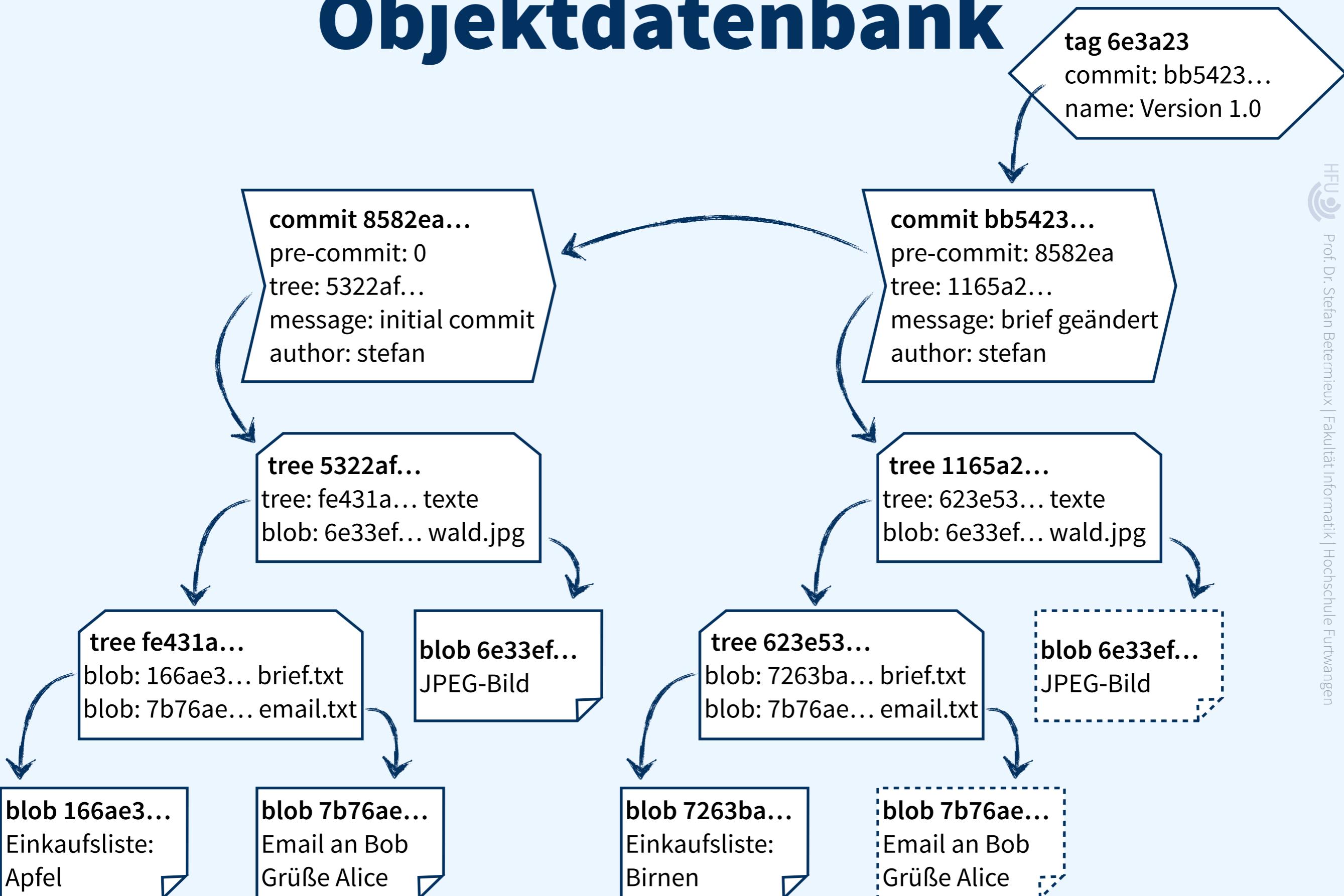
# Hashwerte

- Von jedem Dokument im Repository berechnet Git den Hashwert
- SHA-1 Algorithmus → 40 stellige Hex-Zahl
  - ▶ z.B.: 24b9da6552252987aa493b52f8696cd6d3b00373
- Nur über diesen Hashwert wird das Dokument in der internen Datenbank gespeichert
  - ▶ Dateiname und Pfad sind irrelevant
- Interne Datenbank des Repository besteht aus fünf Klassen:
  - ▶ Blobs → Dokumente
  - ▶ Trees → Verweise auf Blobs und andere Trees
  - ▶ Commits → Ein Verweis auf einen Tree und einen Vorgänger
  - ▶ Branches und Tags → Bezeichnungen für einen Commit

# Objektdatenbank

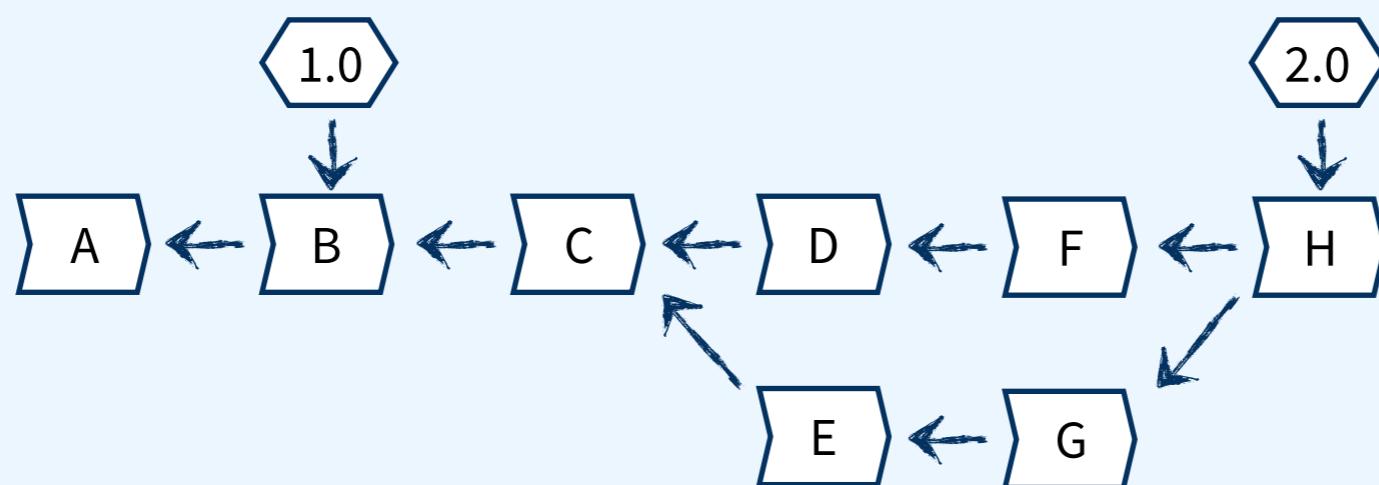


# Objektdatenbank



# Commit-Graph

- Der Graph der Objektdatenbank lässt sich auf die Commits, Tags und Branches reduzieren
- Wird »Commit-Graph« genannt
- Git kennt keine Revisionsnummern, nur Hashwerte der Commits
- Es entsteht ein gerichteter azyklischer Graph (DAG)
- GUI-Tools können diesen Graphen darstellen





# WERKZEUGE

# Kommandozeilen Client

- Ein Projekt der Linux-Kernel-Entwickler
- Kann für alle gängigen Betriebssysteme heruntergeladen werden:
  - ▶ <http://git-scm.com/download>
  - ▶ allerdings nur Kommandozeilenprogramme
- Alle lokalen Operationen von Git werden mit einem Kommando ausgeführt: »git« gefolgt von der Operation
  - ▶ z.B.: »git help«

```
$ git help
usage: git [--version] [--help] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

# GIT konfigurieren

- Bei jedem Commit wird der Name des Autors integriert
- Der Name muss einmal global definiert werden
- Zusätzlich können noch weitere globale Einstellungen vorgenommen werden:
  - ▶ Texteditor
  - ▶ Mergetool
  - ▶ Farben
  - ▶ Passwortspeicher

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com  
$ git config -list  
user.name=John Doe  
user.email=johndoe@example.com
```

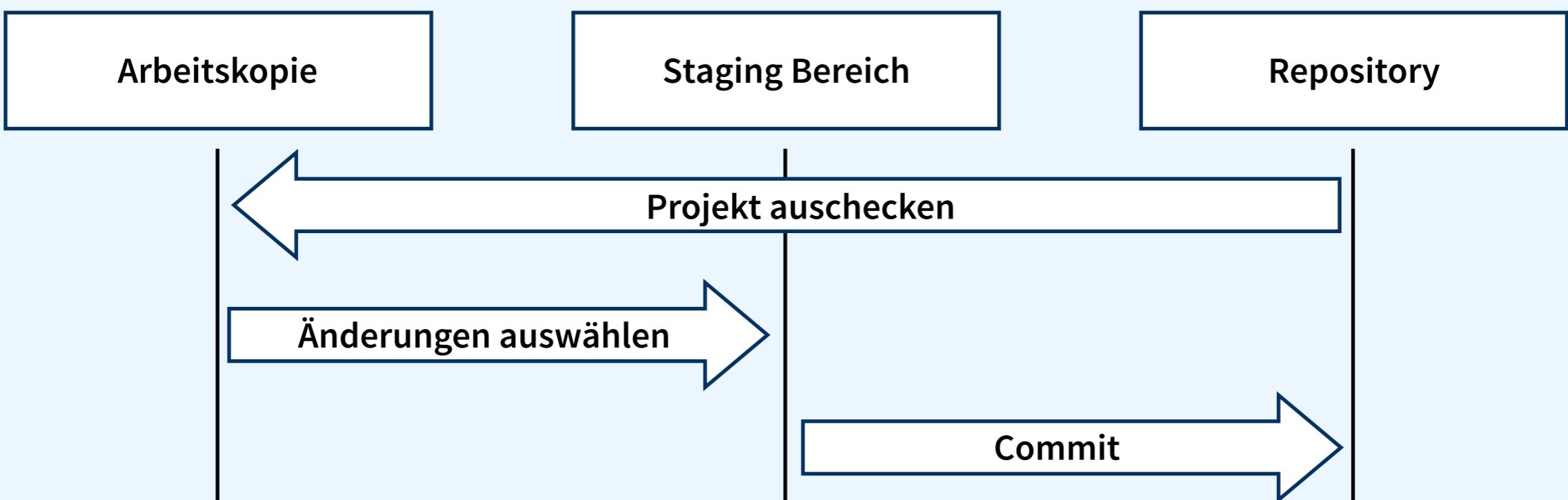
# Repository erstellen

- Mittels »git init« wird im aktuellen Verzeichnis ein neues Git-Repository angelegt
  - ▶ als verstecktes Verzeichnis ».git« im aktuellen Verzeichnis
  - ▶ das Repository ist noch leer
  - ▶ das aktuelle Verzeichnis fungiert als Arbeitsverzeichnis
- Mittels »git clone [url]« kann alternativ ein existierendes Repository kopiert werden

```
/$ mkdir GitProjekt  
/$ cd GitProjekt  
/GitProjekt $ git init  
Initialized empty Git repository in /Users/stefan/Temp/GitProjekt/.git/  
/GitProjekt $ ls -la  
total 0  
drwxr-xr-x  3 stefan  staff  102  6 Dez 14:07 .  
drwxr-xr-x 15 stefan  staff  510  6 Dez 14:07 ..  
drwxr-xr-x 10 stefan  staff  340  6 Dez 14:07 .git
```

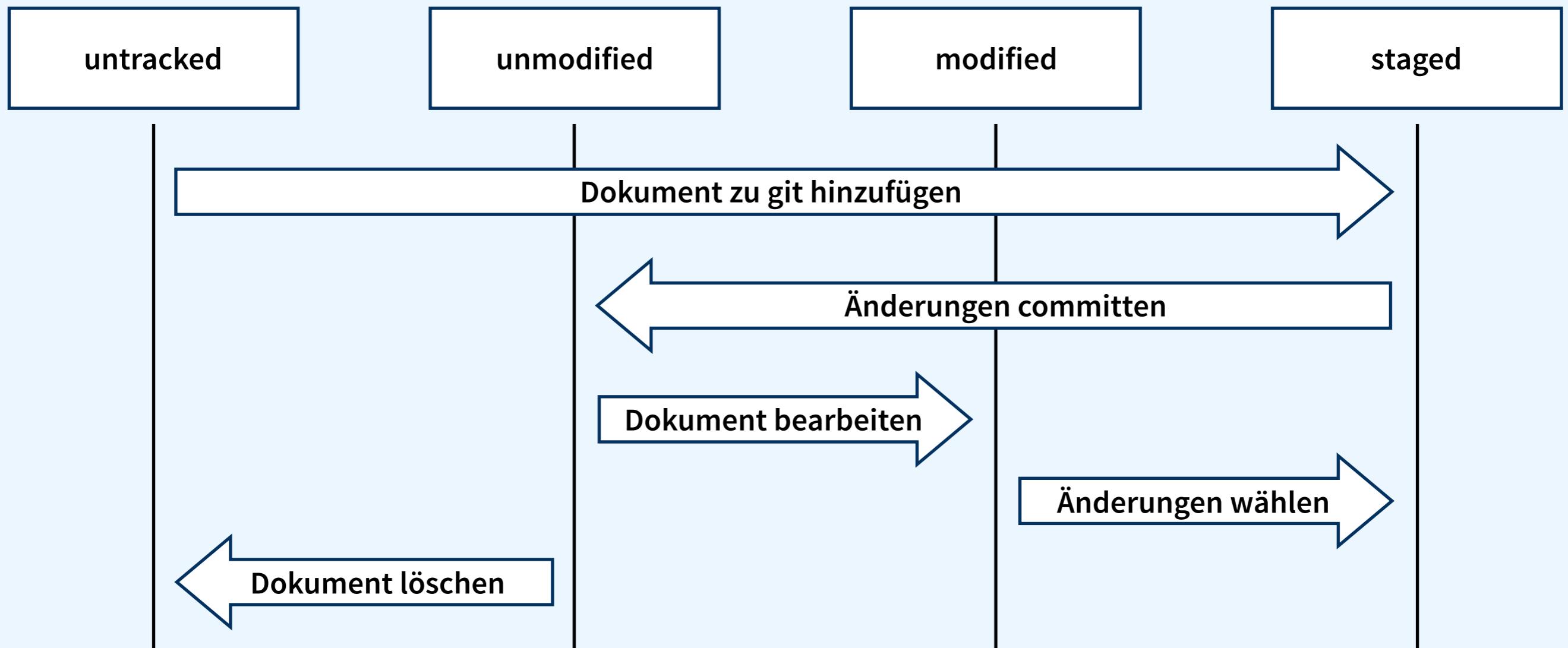
# Staging Bereich

- Anders als in Subversion werden nicht alle lokalen Änderungen bei einem Commit in das Repository übertragen
- Git kennt einen Staging-Bereich
- geänderte Dokumente müssen erst aufgenommen werden
- erlaubt granulare Commits, GUI-Tools erleichtern Arbeit



# Dokumentenstatus

- Mit der Kenntnis des Staging-Bereichs können Dokumente vier verschiedene Zustände annehmen:



# Dokument erstellen

- Dateien im Arbeitsverzeichnis müssen mit »git add [datei]« zum Staging-Bereich hinzugefügt werden
- Mittels »git status« kann der aktuelle Zustand abgefragt werden

```
/GitProjekt $ emacs brief.txt
/GitProjekt $ git status
...
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       brief.txt
/GitProjekt $ git add brief.txt
/GitProjekt $ git status
...
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   brief.txt
```

# Commit erzeugen

- Die Änderungen im Staging-Bereich werden in das Repository übertragen
- Das Kommando »git commit« startet den Vorgang
- Commit erwartet eine kurze Beschreibung der Änderungen »-m [message]«
- In einfachen Fällen kann der Staging-Bereich übersprungen werden, und alle modifizierten Dokumente committed werden:  
commit Parameter »-a«

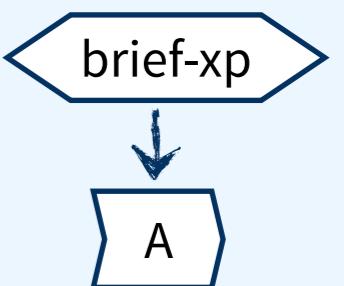
```
/GitProjekt $ git commit -m "brief hinzugefügt"
[master (root-commit) 32b24bc] brief hinzugefügt
 1 file changed, 5 insertions(+)
  create mode 100644 brief.txt
/GitProjekt $ git status
# On branch master
nothing to commit, working directory clean
```

A

# Tag erzeugen

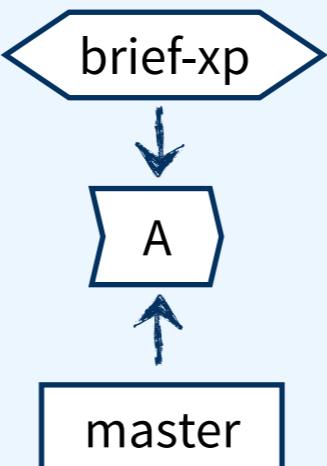
- Tags sind in Git selbständige Objekte
  - ▶ verweisen auf einen Commit
  - ▶ keine Verzeichniskopien wie in Subversion
- Die Operation »git tag -a [version] -m [message]« erstellt einen Tag zu dem letzten Commit

```
/GitProjekt $ git tag -a brief-xp -m "brief.txt als XP  
getaggt"  
/GitProjekt $ git show brief-xp  
tag brief-xp  
Tagger: Stefan Betermieux <stefan@betermieux.de>  
Date:   Fri Dec 6 15:15:47 2013 +0100  
  
brief.txt als XP getaggt  
  
commit 32b24bcfde13da07c28ffaa9b6fe88dbc96f3b66  
Author: John Doe <johndoe@example.com>  
Date:   Fri Dec 6 14:55:13 2013 +0100  
  
brief hinzugefügt
```



# Git Branches

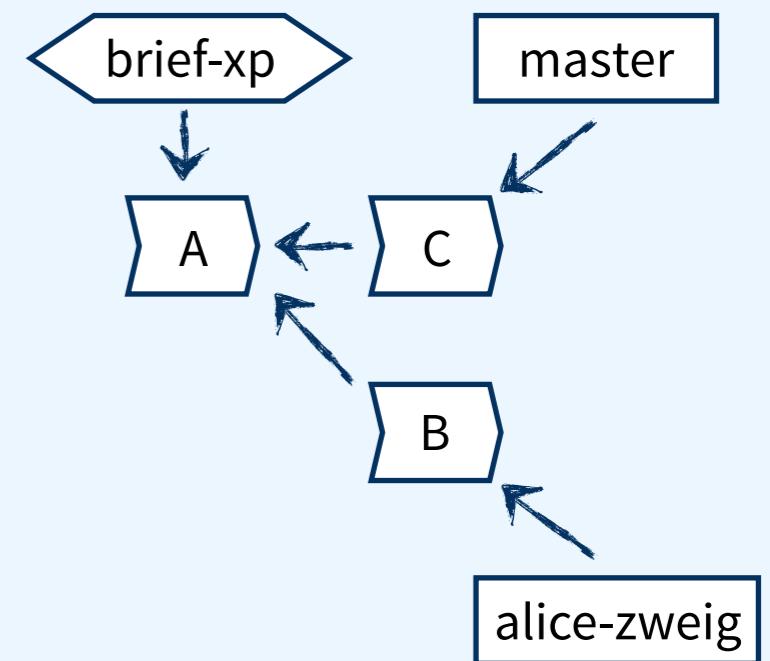
- In Git gibt es einen Standard-Branch »master«
- Vergleichbar mit dem »trunk« in Subversion
- Ähnlich wie ein Tag nur ein Zeiger auf einen Commit
- Der Zeiger wandert bei jedem Commit
  - ▶ zeigt immer auf den letzten Commit des Branches



# Branch erzeugen

- Mit »git branch [zweig]« wird ein neuer Zweig erstellt
  - ▶ neues Branch-Objekt im Repository erzeugt
- Mit »git checkout [zweig]« wird auf den neuen Zweig gewechselt
  - ▶ bestimmt, in welchen Branch Commits geschrieben werden

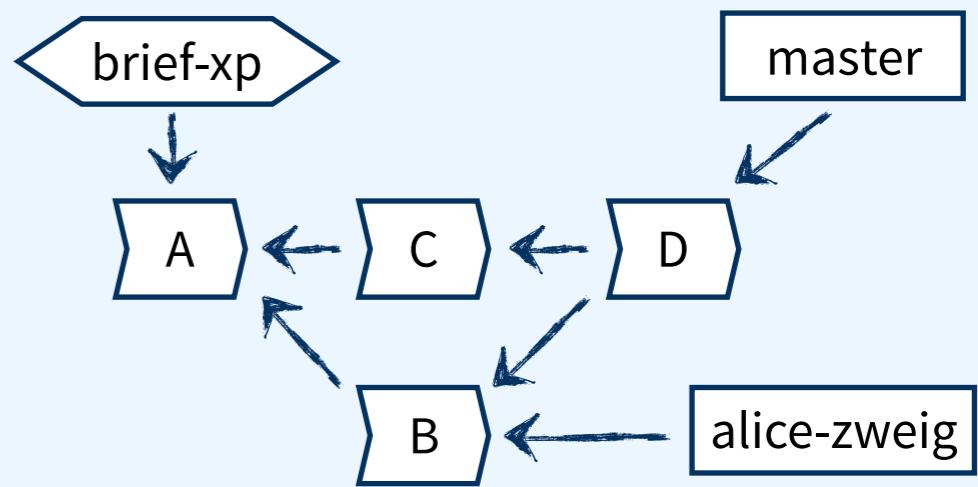
```
/GitProjekt $ git branch alice-zweig
/GitProjekt $ git checkout alice-zweig
Switched to branch 'alice-zweig'
/GitProjekt $ emacs brief.txt
/GitProjekt $ git commit -a -m "Änderungen im Zweig"
[alice-zweig e0a0374] Änderungen im Zweig
 1 file changed, 1 insertion(+)
/GitProjekt $ git checkout master
Switched to branch 'master'
/GitProjekt $ emacs brief.txt
/GitProjekt $ git commit -a -m "Änderungen im Master"
[master 8436b95] Änderungen im Master
 1 file changed, 1 insertion(+), 1 deletion(-)
```



# Branch Zusammenführen

- Das Zusammenführen (Merge) läuft ähnlich wie in Subversion:
- Zielzweig als Arbeitskopie auschecken
- »git merge [zweig]« mit dem Quellzweig aufrufen
  - ▶ Änderungen werden automatisch committed
- Eventuell Quellzweig löschen (entspricht löschen eines Zeigers)
  - ▶ mit »git branch -d [zweig]«

```
/GitProjekt $ git branch
* master
  alice-zweig
/GitProjekt $ git merge alice-zweig
Auto-merging brief.txt
Merge made by the 'recursive' strategy.
  brief.txt | 1 +
  1 file changed, 1 insertion(+)
```



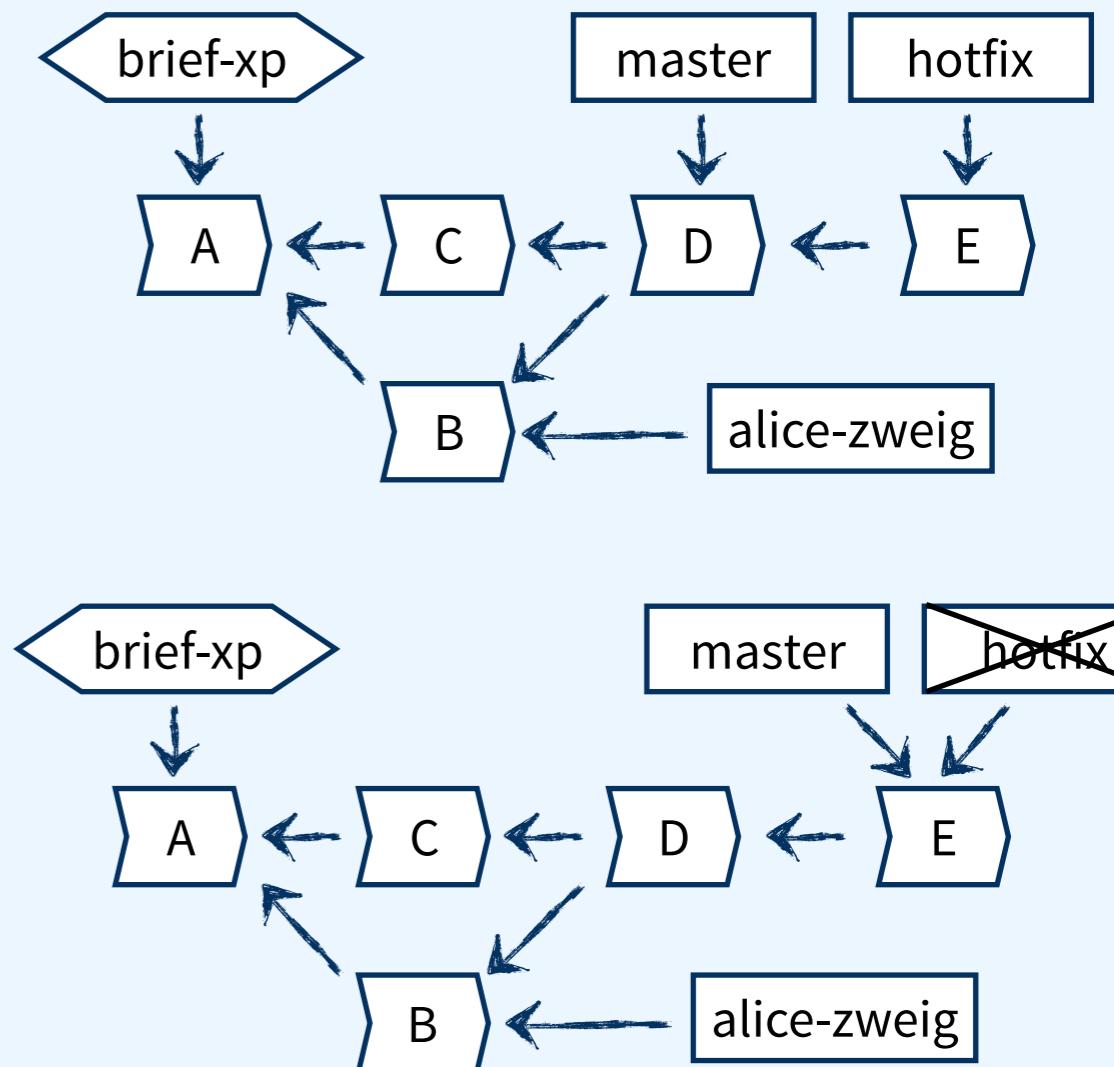
fortgeschrittene Git-Befehle

# Git spezifisches

# Fast-Forward-Merge

- Ein Merge ist besonders einfach, wenn...
  - ▶ ...zwei Branches in direkter Linie miteinander verbunden sind
- Ein (automatischer) Fast-Forward-Merge verschiebt nur den alten Branch-Zeiger auf den neuen.
- ▶ keine Konflikte möglich

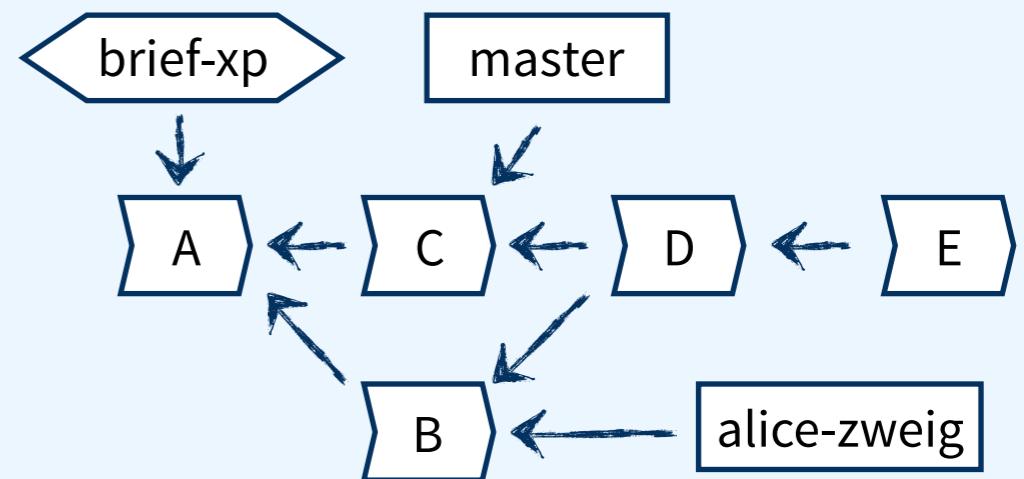
```
/GitProjekt $ git branch hotfix
/GitProjekt $ git checkout hotfix
Switched to branch 'hotfix'
/GitProjekt $ emacs brief.txt
/GitProjekt $ git commit -a -m "brief geändert"
[hotfix 5639bcf] brief geändert
 1 file changed, 1 insertion(+)
/GitProjekt $ git checkout master
Switched to branch 'master'
/GitProjekt $ git merge hotfix
Updating 8aee47c..5639bcf
Fast-forward
  brief.txt | 1 +
  1 file changed, 1 insertion(+)
/GitProjekt $ git branch -d hotfix
```



# Commits rückgängig machen

- Für die nächsten Schritte müssen wir zu einem früheren Zustand zurückkehren → die letzten Commits rückgängig machen
- Mit »git reset --hard HEAD~[Anzahl]« kann der aktuelle Zeiger (in diesem Fall Master) eine Anzahl Commits zurückspringen
- Übersprungene Commits sind zwar nicht gelöscht, können aber nicht mehr über Zeiger erreicht werden
  - ▶ Vorsicht bei der Arbeit im Team, andere könnten noch auf D und E arbeiten!

```
/GitProjekt $ git reset --hard HEAD~2  
HEAD is now at a50851b Änderungen im Master
```

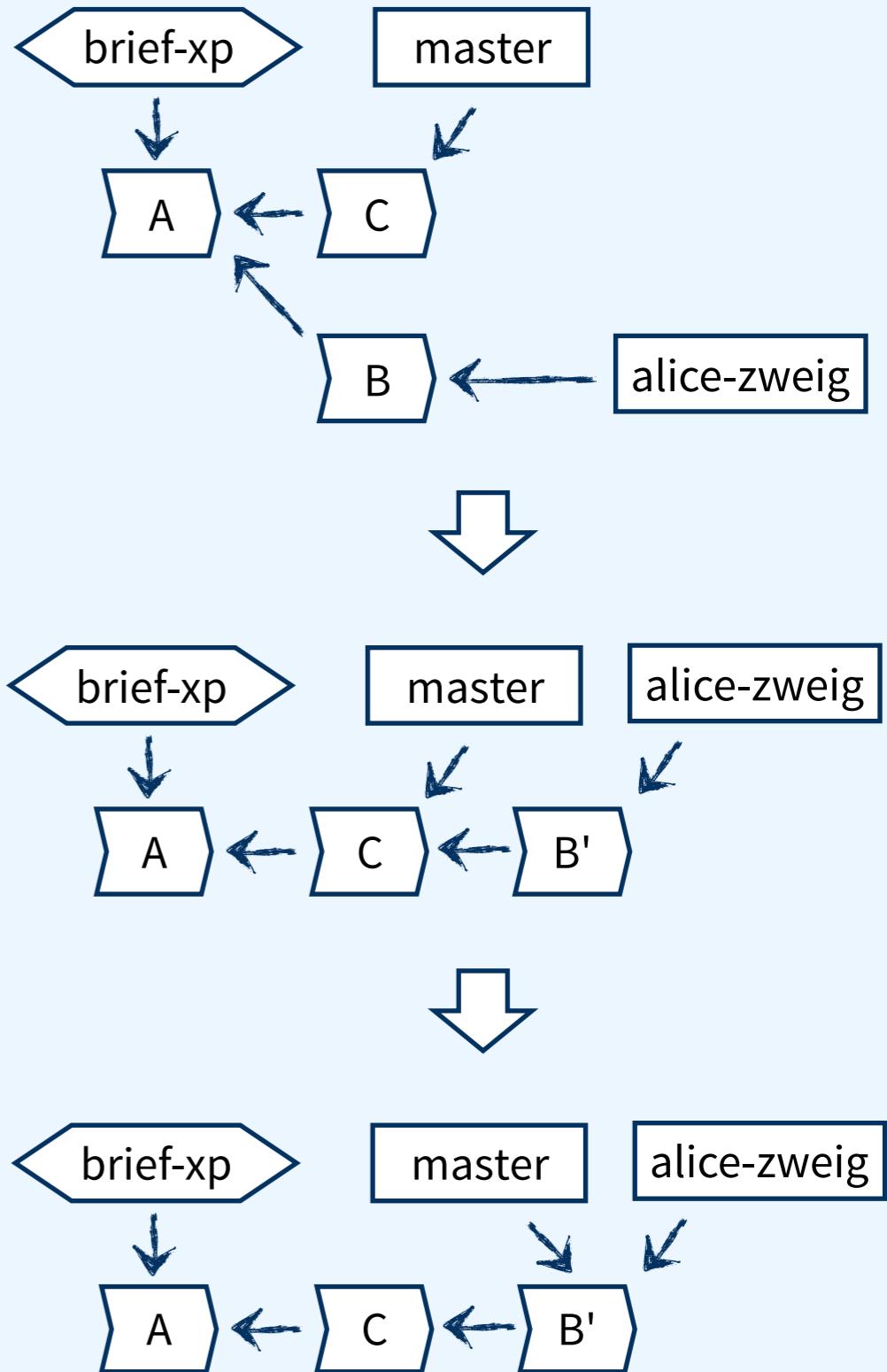


# Rebasing

- Anstelle Branches zusammenzuführen, können Commits auch »rebased« werden:
  - ▶ Commits werden aus dem Ursprungsbranch entfernt und modifiziert als neue Commits im Zielbranch eingefügt (rebase)
- Das Ergebnis eines Merges und eines Rebase ist inhaltlich identisch
  - ▶ nur der Commit-Graph ist am Ende übersichtlicher
- Problem: Originalcommit und modifizierter Commit haben unterschiedlichen Hashwert
  - ▶ kein anderer Commit darf mehr auf den Originalcommit verweisen
  - ▶ schwierig sicherzustellen in verteilten Umgebungen

# Rebasing

```
/GitProjekt $ git checkout alice-zweig
Switched to branch 'alice-zweig'
/GitProjekt $ git branch
* alice-zweig
  master
/GitProjekt $ git rebase master
First, rewinding head to replay your work on
top of it...
Applying: Änderungen im Zweig
/GitProjekt $ git checkout master
Switched to branch 'master'
/GitProjekt $ git merge alice-zweig
Updating a50851b..3f0081e
Fast-forward
brief.txt | 1 +
 1 file changed, 1 insertion(+)
```



# Merge oder Rebase?

- Wenn der Zielzustand identisch ist, wofür sollte man sich entscheiden?
- Merge
  - ▶ alle Commits in einem Quellzweig werden zu einem einzigen neuen Commit im Zielzweig zusammengefasst
- Rebase
  - ▶ alle Commits werden einzeln auf den Zielzweig übertragen (für Fortgeschrittene: Commits können kombiniert werden)
  - ▶ nach einem Rebase muss immer ein Fast-Forward-Merge ausgeführt werden (Vorteil: Bei diesem Merge sind keine Konflikte möglich!)

# Was fehlt?

- Bis jetzt war alles lokal, warum heißt es »verteilte Versionsverwaltung«
  - ▶ externe Branches → spätere Vorlesung
  - ▶ Git hosting im Web
  - ▶ »social coding«
- GUI-Werkzeuge
  - ▶ Integration in Eclipse



# ZUSAMMENFASSUNG

# Zusammenfassung GIT

## Vorteile:

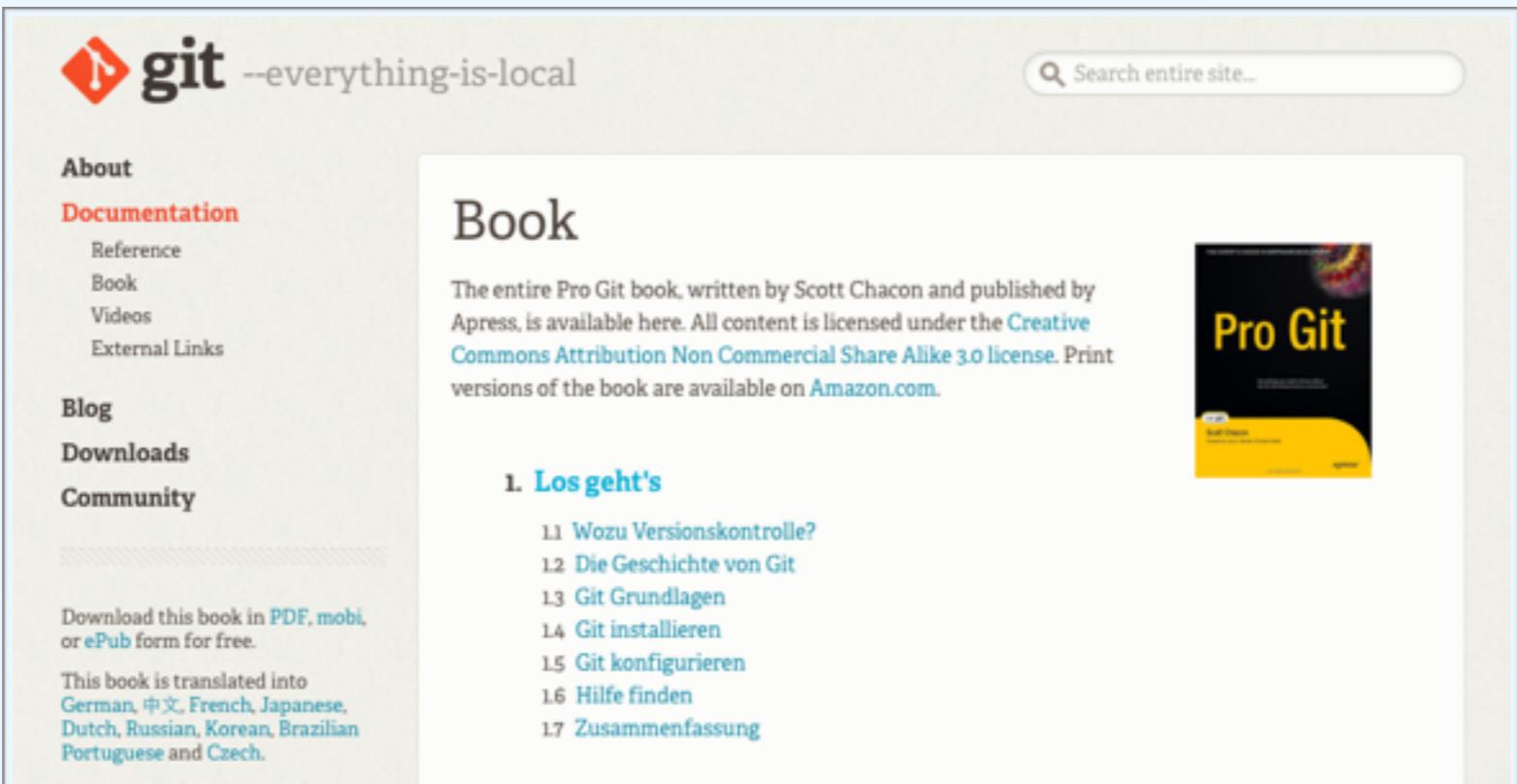
- Es ist schnell
- Kein Netzwerkverbindung mehr erforderlich (z.B. auf Reisen)
- Für kleine Teams ist kein Server mehr notwendig
  - ▶ Kommunikation kann auch peer-to-peer erfolgen
- Branching und Merging wird gut unterstützt
- Jeder hat eine lokale Spielwiese
  - ▶ mit vollen Rechten

## Nachteile:

- Keine zentrale Instanz
  - ▶ regelmäßiges Backup wird wieder wichtiger
- Kein zentraler Entwicklungszweig
  - ▶ jeder Entwickler arbeitet auf eigenen Zweigen
  - ▶ wo ist die aktuellste Version der gemeinsamen Anwendung
- Es gibt keine Revisionsnummern
  - ▶ SHA1-Hashes sind hässlich  
»gib mir mal Änderung 73a9bb!«
- ▶ nicht sequentiell

# Literatur

- Git Buch »Pro Git«
  - ▶ in gedruckter Form von Apress
  - ▶ in elektronischer Form unter der Creative-Commons-Lizenz:  
<http://git-scm.com/book/de>



The screenshot shows the official website for the book "Pro Git". The header features the "git" logo and the tagline "--everything-is-local". On the left sidebar, there are links for "About", "Documentation" (which includes "Reference", "Book", "Videos", and "External Links"), "Blog", "Downloads", and "Community". Below the sidebar, there's a note about downloading the book in PDF, mobi, or ePub format. The main content area is titled "Book" and contains text about the availability of the entire book under a Creative Commons Attribution Non Commercial Share Alike 3.0 license. It also mentions that print versions are available on Amazon.com. To the right of this text is a thumbnail image of the "Pro Git" book cover, which is black with yellow text. Below the book image is a table of contents for chapter 1, titled "Los geht's", listing 17 numbered items from "Wozu Versionskontrolle?" to "Zusammenfassung".

# Git Cheat Sheet

Kurzzusammenfassung der wichtigsten Kommandos:

<http://www.git-tower.com/blog/git-cheat-sheet-detail-de/>

## GIT CHEAT SHEET

presented by TOWER – the most powerful Git client for Mac



REPOS ERSTELLEN	BRANCHES & TAGS	MERGE & REBASE
Clonen eines bestehenden Repos <code>\$ git clone ssh://user@domain.com/repo.git</code>	Alle Branches auflisten <code>\$ git branch -av</code>	Merge von <branch> in aktuellen HEAD <code>\$ git merge &lt;branch&gt;</code>
Neues, lokales Repo erstellen <code>\$ git init</code>	Aktuellen HEAD-Branch wechseln <code>\$ git checkout &lt;branch&gt;</code>	Aktuellen HEAD auf <branch> rebasen <i>Kein Rebase von Commits, die published sind!</i> <code>\$ git rebase &lt;branch&gt;</code>
LOKALE ÄNDERUNGEN	NEUER BRANCH	REBASE
Veränderte Files in der Working Copy <code>\$ git status</code>	Neuen lokalen Branch erstellen (basierend auf dem aktuellen HEAD) <code>\$ git branch &lt;new-branch&gt;</code>	Rebase abbrechen <code>\$ git rebase --abort</code>
Änderungen an versionierten Files <code>\$ git diff</code>	Neuen lokalen Branch auf Basis eines <remote/branch> erstellen <code>\$ git checkout --track &lt;remote/branch&gt;</code>	Rebase nach Konfliktlösung fortsetzen <code>\$ git rebase --continue</code>
Alle lokalen Änderungen zum nächsten Commit hinzufügen <code>\$ git add .</code>	Lokalen Branch löschen <code>\$ git branch -d &lt;branch&gt;</code>	Konflikt in externem Mergetool lösen <code>\$ git mergetool</code>
Teile der Änderungen in <file> zum nächsten Commit hinzufügen <code>\$ git add -p &lt;file&gt;</code>	Tag auf aktuellem Commit erstellen <code>\$ git tag &lt;tag-name&gt;</code>	Konflikt als <resolved> markieren nach manueller Konfliktlösung im Editor <code>\$ git add &lt;resolved-file&gt;</code>
		<b>UPDATE &amp; PUBLISH</b>

# DANKE