

# Inhaltsverzeichnis

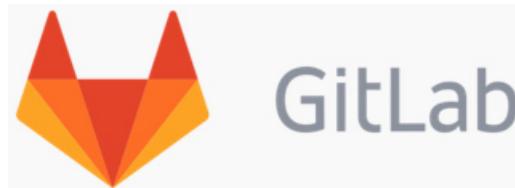
- 01 Einführung
- 02 Prozessmodelle
- 03 Konfigurationsmanagement
- 04 Requirements Engineering
- 05 Modellierung
- 06 Qualitätsmanagement

## Nach dieser Vorlesungseinheit ...

- ... kennen Sie den Zusammenhang von
  - Konfigurationsmanagement (**Konfigurationselemente**, welche nicht)
  - und **Versionskontrolle** (was und wie wird versioniert)
- ... wissen Sie, was ist eine **Konfiguration** ist und was das **Management** von Konfigurationen bedeutet
- ... lernen Sie **KM Werkzeuge** kennen und **Operationen**
  - und speziell: **Git**

# Was hatten Sie schon mit Konfigurationsmanagement zu tun?

03 Konfigurationsmanagement



Bazaar



TortoiseSVN

## ● Konfigurationsmanagement (KM)

„Das KM stellt einen Mechanismus zur **Identifizierung**, **Lenkung** und **Rückverfolgung der Versionen jedes Softwareelements** dar. In vielen Fällen sind auch frühere, nach wie vor in Verwendung befindliche Versionen zu warten und lenken.“

ISO-9000-3

# Konfigurationsmanagement – Definition (2)

## 03 Konfigurationsmanagement

### ● Produktversion

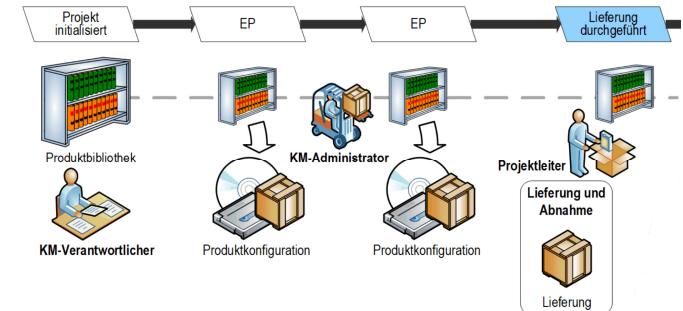
- „.... ist ein **identifizierbarer** und **reproduzierbarer** Bearbeitungsstand eines Produktartefaktes. Eine Produktversion hat genau einen Produktzustand.“

### ● Produktkonfiguration

- „.... ist eine **Menge von Produktversionen**, eine so genannte Baseline. Ihre Aufgabe besteht darin, die Konfigurationseinheiten und deren **strukturellen Zusammenhang** zu definieren. [...]“

### ● Konfigurationsmanagement

- „.... sorgt dafür, dass alle im Projekt erstellten Ergebnisse nachvollziehbar abgelegt und gesichert werden. Zentrales "Produkt" ist die Produktbibliothek, bei der es sich um die Summe aller im Projekt erstellten Produktexemplare in sämtlichen Produktversionen handelt. [...]“
- „... mindestens zu jedem Entscheidungspunkt [wird] eine Produktkonfiguration (Baseline) erstellt. Dabei handelt es sich um eine **Menge aktueller und untereinander konsistenter Produktversionen**, die **nachvollziehbar** gesichert werden, um den Projektstand zu **dokumentieren** und um im Bedarfsfall darauf zurückzugreifen.“



[Quelle: V-Modell XT, Konfigurationsmanagement]



# Wozu Konfigurationsmanagement?

03 Konfigurationsmanagement

- **Kontrolle paralleler Änderungen**

(Konflikte bei parallelen Änderungen am selben Dokument werden entweder vermieden oder erkannt und behoben)

- Kontrollierter Zugriff über **Berechtigungen**

- Jede(r) Berechtigte darf ändern

- **Nachvollziehbarkeit**

- Repository führt über Änderungen Buch

- **Wiederherstellbarkeit** alter Versionen

- **Parallele Pflege mehrerer Versionen** (Branches)

- Vereinfachung der Kommunikation

# Konfigurationselemente – Für welche Dateien ist KM wichtig?

03 Konfigurationsmanagement

- Quelltexte
- Anforderungsdokumente
- Architektur / Designdokumente
- Testspezifikationen und Testdaten
- Build Skripte
- Meta / Konfigurationsdaten
- Benutzerdokumentation
- Installationsanleitung, Release Notes, etc.

Siehe auch: G. Popp: Konfigurationsmanagement

# Konfigurationselemente – Für welche Dateien ist KM irrelevant?

## 03 Konfigurationsmanagement

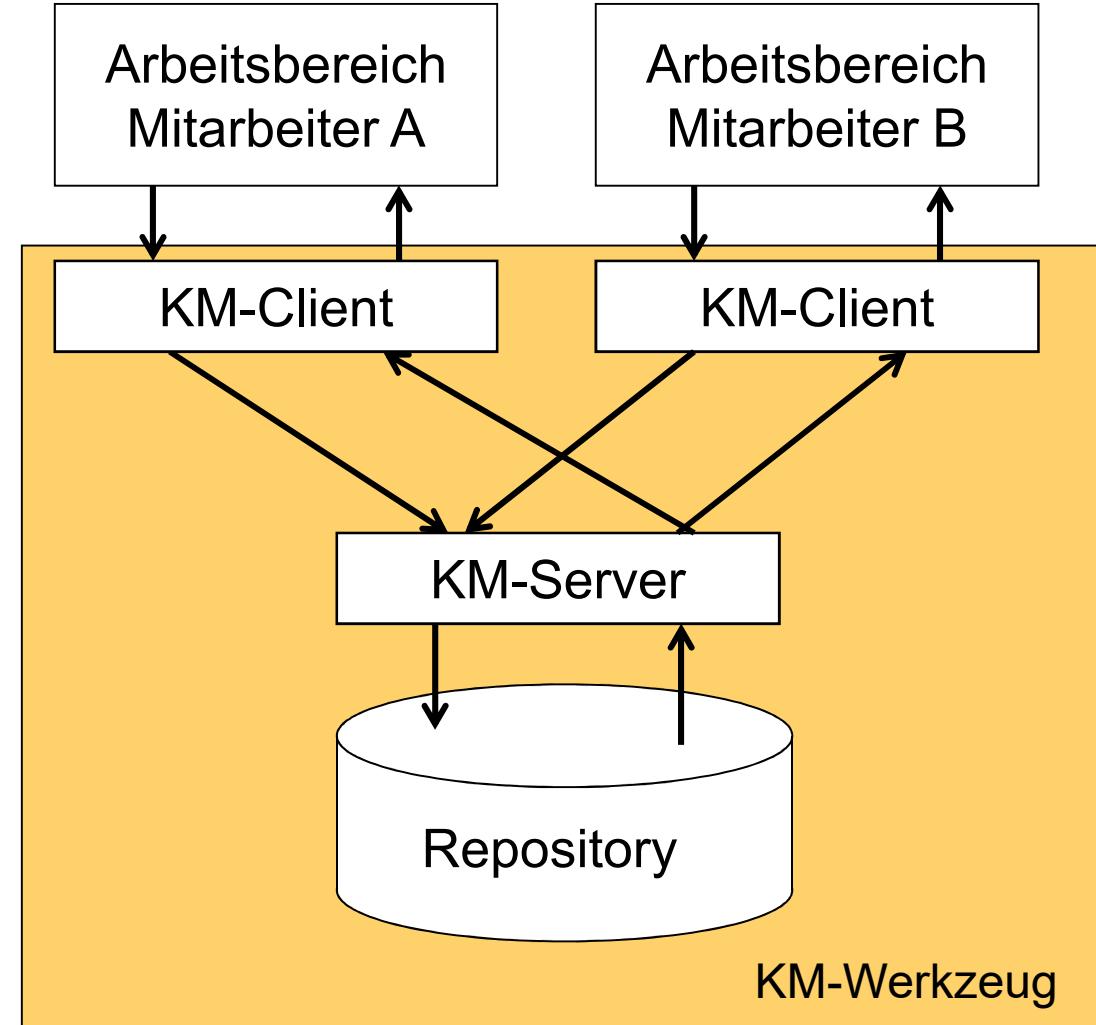
- „Bewegungsdaten“ im Projekt
  - Protokolle von Meetings
  - Binäre Auslieferungsdateien
  - Generierte Dateien
  - Liste offener Punkte (LOP, Risikolisten, etc.)
  - Projektpläne
- Speichern in **Projektablage** (z.B. gem. Verzeichnis)
- Zweifelsfälle (z.B. Bugfix in 10 Jahren?)
  - Werkzeuge (Compiler, Generatoren, MS-Word?, ...)
  - Fremde / eigene Bibliotheken und Frameworks

Siehe auch: G. Popp: Konfigurationsmanagement

# Klassische Grundstruktur KM-Werkzeug

## 03 Konfigurationsmanagement

- **Repository** („Softwarebibliothek“)
  - ist eine Datenbank oder ein Server-Verzeichnis
  - speichert Dokumente, Quelltexte und deren Änderungen
- **KM-Server / KM-Client**
  - führen Austausch durch
  - Operationen: Check Out, Update, Check In, ..., etc.
  - Bildung von Branches, etc.
- **KM-Werkzeug**
  - regelt Austausch von Dateien zwischen Repository und lokalen Arbeitsbereichen
  - kontrolliert Versionen von Dokumenten, Quelltexten

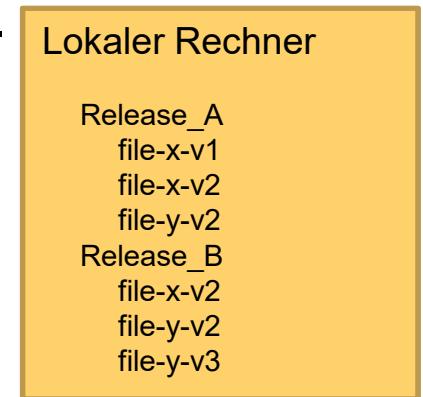


# Arten von Versionskontrollsystmen (1)

## 03 Konfigurationsmanagement

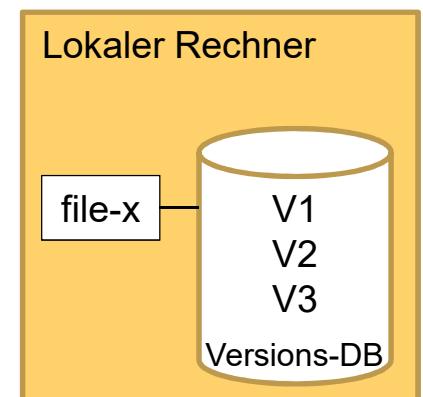
### ● Manuelle Kontrolle (**nicht empfohlen für SW-Entwicklung!**)

- Z.B. Versionsinformation in Dateinamen, unterschiedliche Verzeichnisse bei zusammengehörigen Dateien
- (Leider) oft üblich, da scheinbar einfach – Bordmittel des Betriebssystems
- Sehr fehleranfällig, falsches Kopieren oder Überschreiben



### ● Lokale Kontrolle

- Z.B. rcs (MacOS)
- Lokale Datenbank oder spezielle Dateien (Patches) speichern Änderungen von relevanten Dateien
- Schwierige Zusammenarbeit bei Projekten mit verschiedenen Rechnern und Betriebssystemen

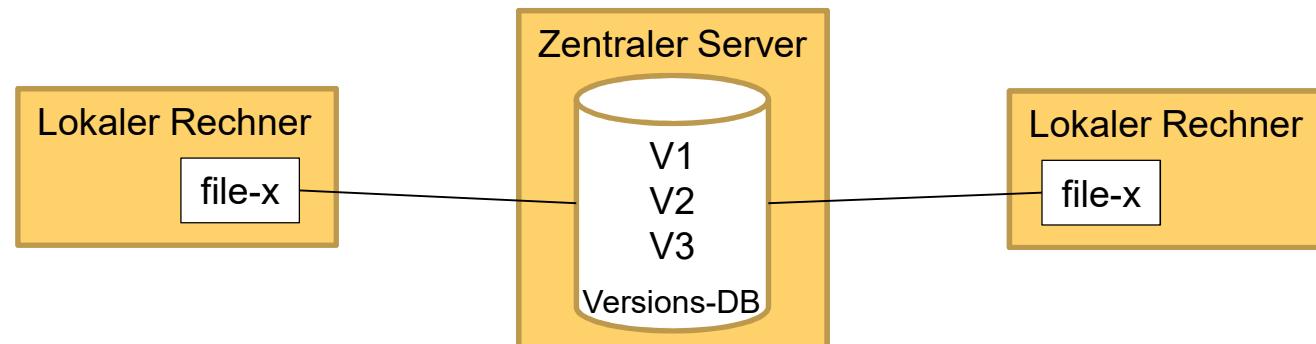


# Arten von Versionskontrollsystmen (2)

## 03 Konfigurationsmanagement

### ● Zentrale Kontrolle

- Z.B. Subversion, CVS
- Zentraler Server verwaltet alle versionierten Dateien, vergleichsweise einfach zu administrieren und einfacher Arbeitsablauf: auschecken → ändern → einchecken
- Nutzer hat nur einen **Snapshot** der Projektdateien – und Server Crash legt Projekt lahm („Single Point of Failure“)

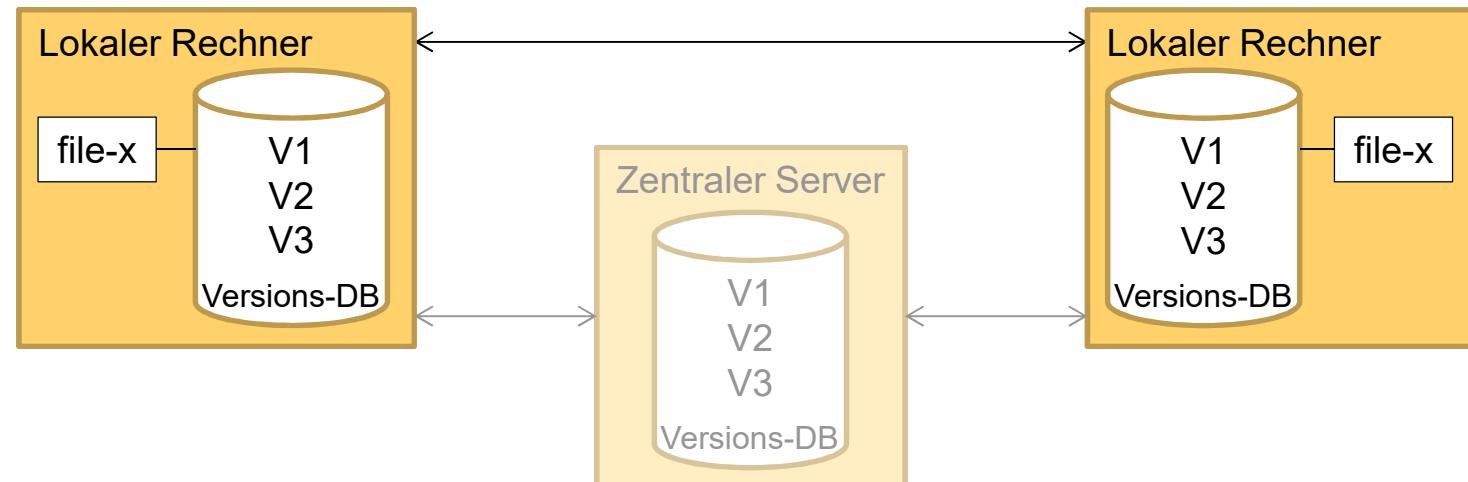


# Arten von Versionskontrollsystmen (3)

## 03 Konfigurationsmanagement

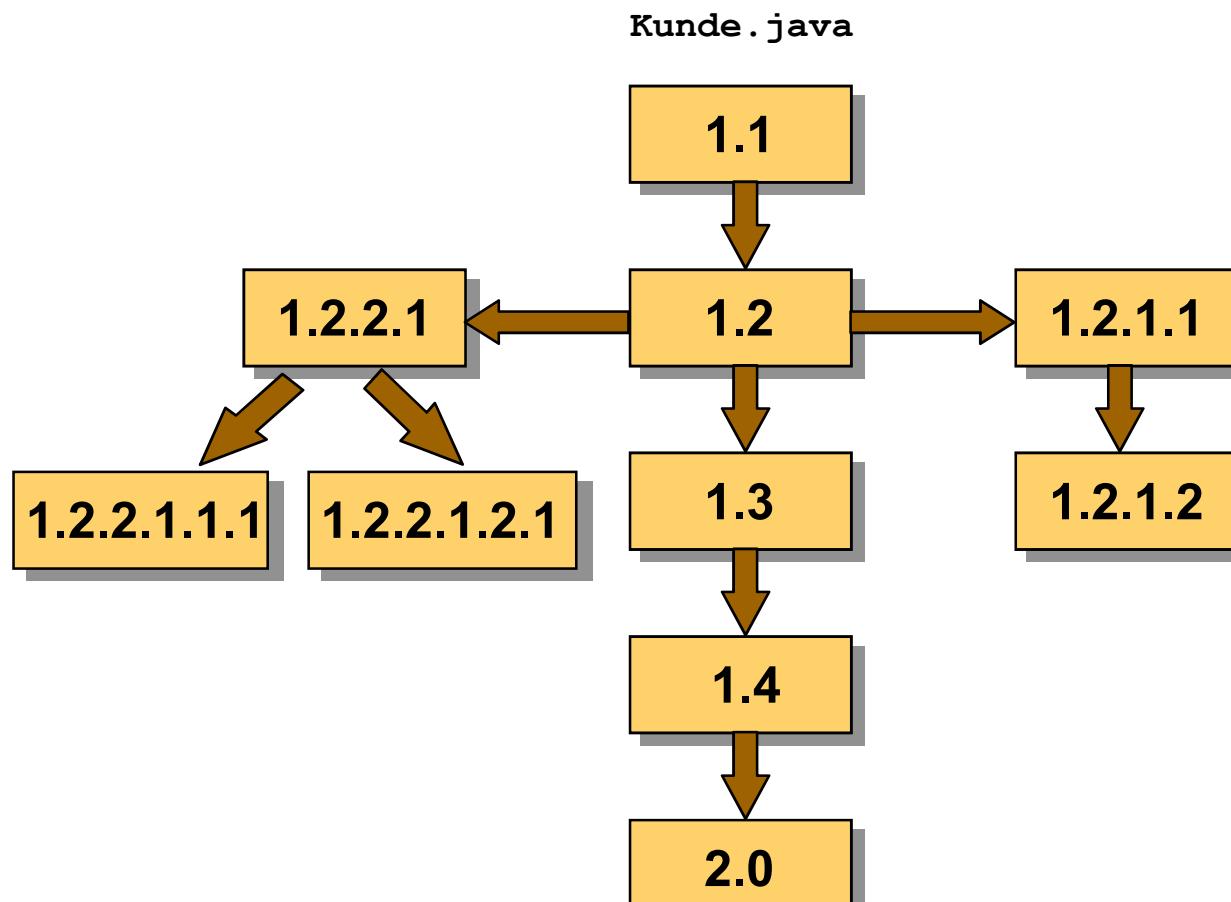
### Verteilte Kontrolle

- Z.B. Git (z.B. Linux), mercurial (z.B. Mozilla), Bazaar (z.B. Ubuntu)
- Jeder Nutzer hat **vollständige Kopie** des Repositories, leichte Server-Wiederherstellung, evtl. sogar Verzicht auf zentrales Repository
- Keine oder kaum Latenz selbst bei global verteilten Teams, ermöglicht unkomplizierte simultane Zusammenarbeit von Teams (unabhängige Arbeitsabläufe), leichtgewichtiges „Branching“



# Wie wird versioniert? – Versionsbäume, hierarchische Nummern

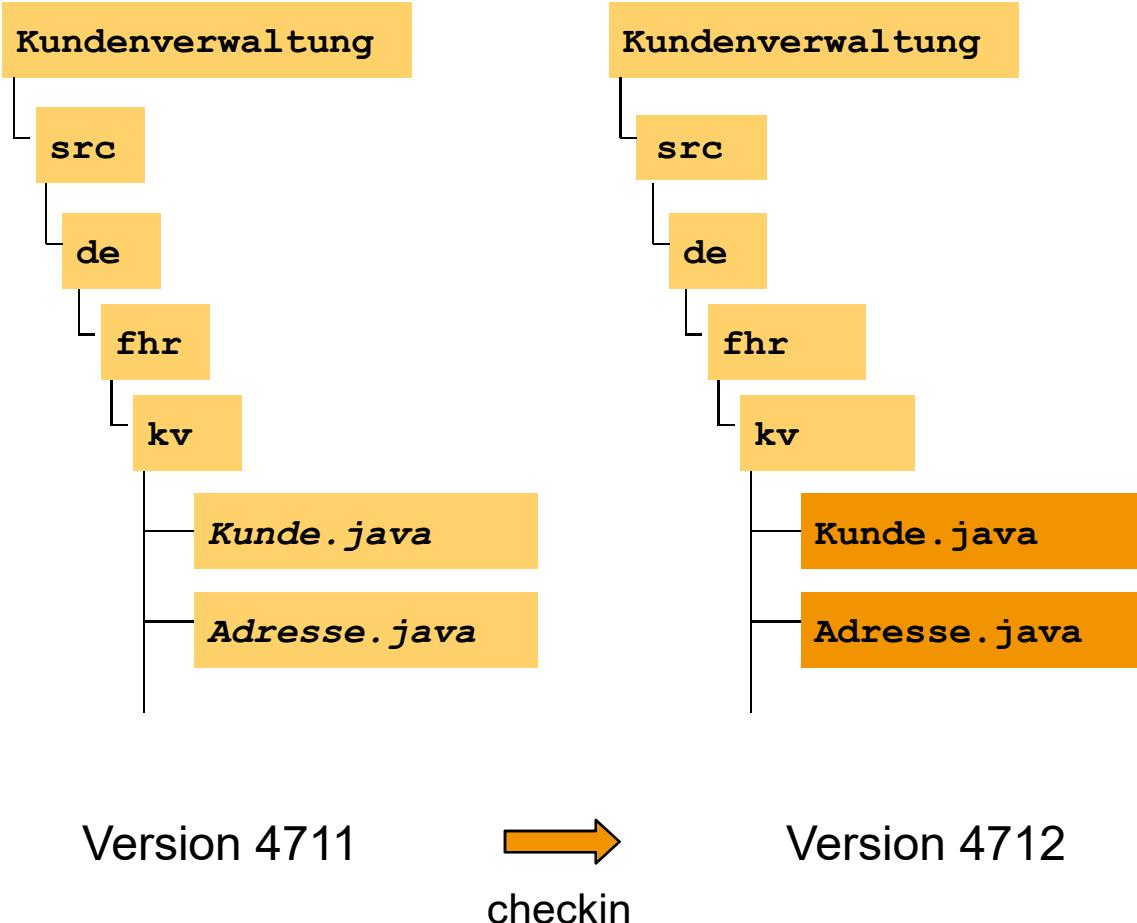
## 03 Konfigurationsmanagement



- Jede Datei hat eigene Versionsnummer
- Versionsnummer
  - Hierarchisch (für Branches)
  - frei wählbar
- Beispiel: CVS
- Problem
  - Keine „Transaktionen“
  - Welche Dateien wurden gleichzeitig geändert?

# Wie wird versioniert? – Nummerierung des Repositories

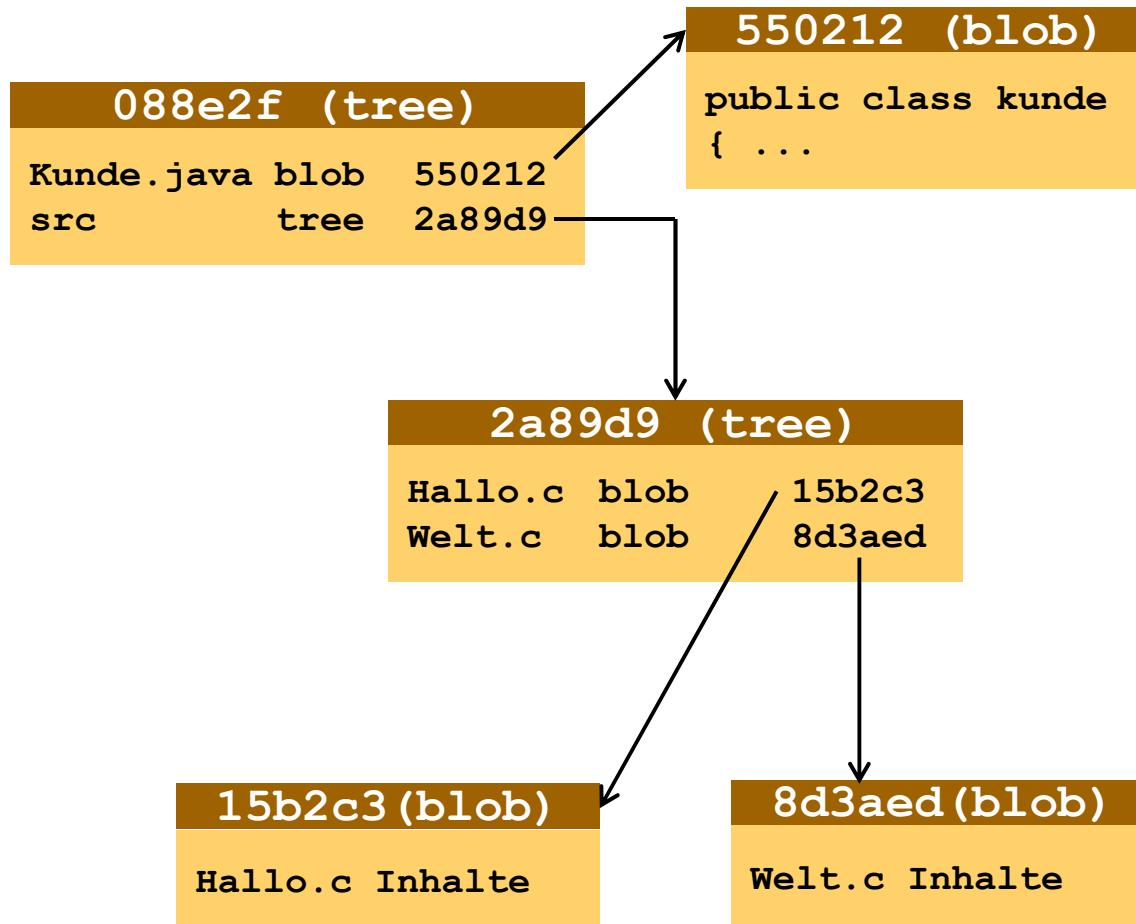
## 03 Konfigurationsmanagement



- Repository hat genau eine Versionsnummer
- Nummer wird bei jedem Commit erhöht
  - Commit = „Transaktion“
  - Erkennung zusammengehörend. Änderungen
- Beispiel: Subversion
- Problem
  - Kopieraufwand
  - Branches über „Lazy Copy“ von Verzeichnissen

# Wie wird versioniert? – Hashing

## 03 Konfigurationsmanagement



- Repository = Object Database
  - Identifikation aller Objekte über SHA-1 Hash (= Prüfsumme)
- Beispiel: Git
  - Objekte sind
    - Dateien (= Blobs)
    - Verzeichnisse (= Trees)
    - Commits
    - Tags (= benannter Stand im Repository)
  - Gespeichert im .git Verzeichnis

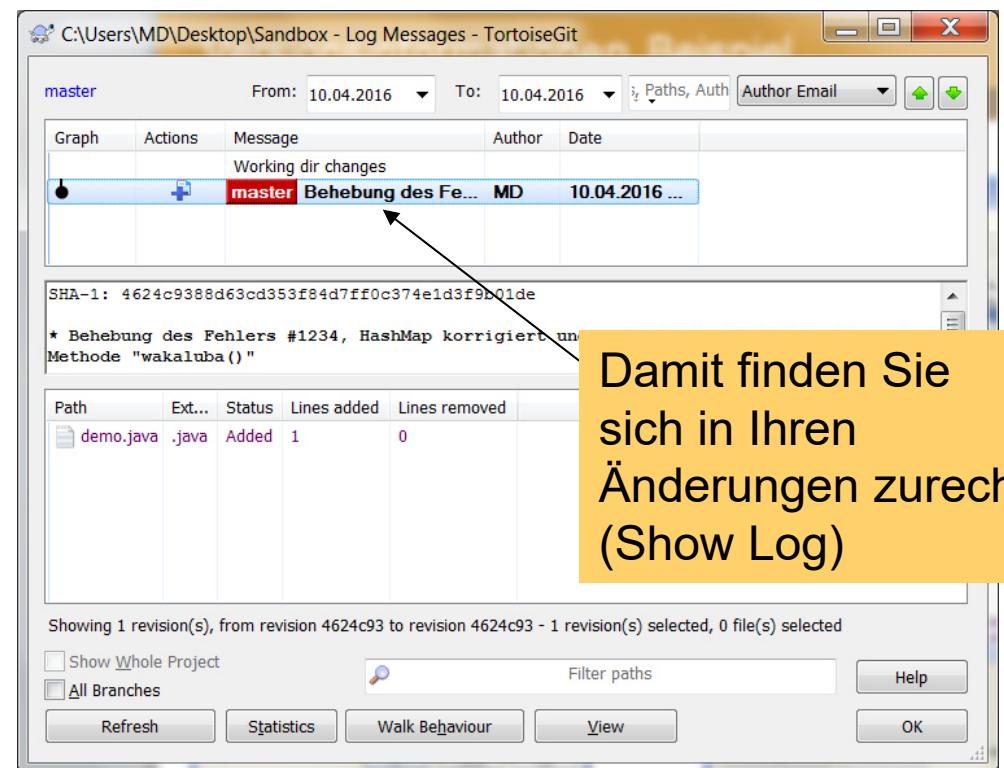
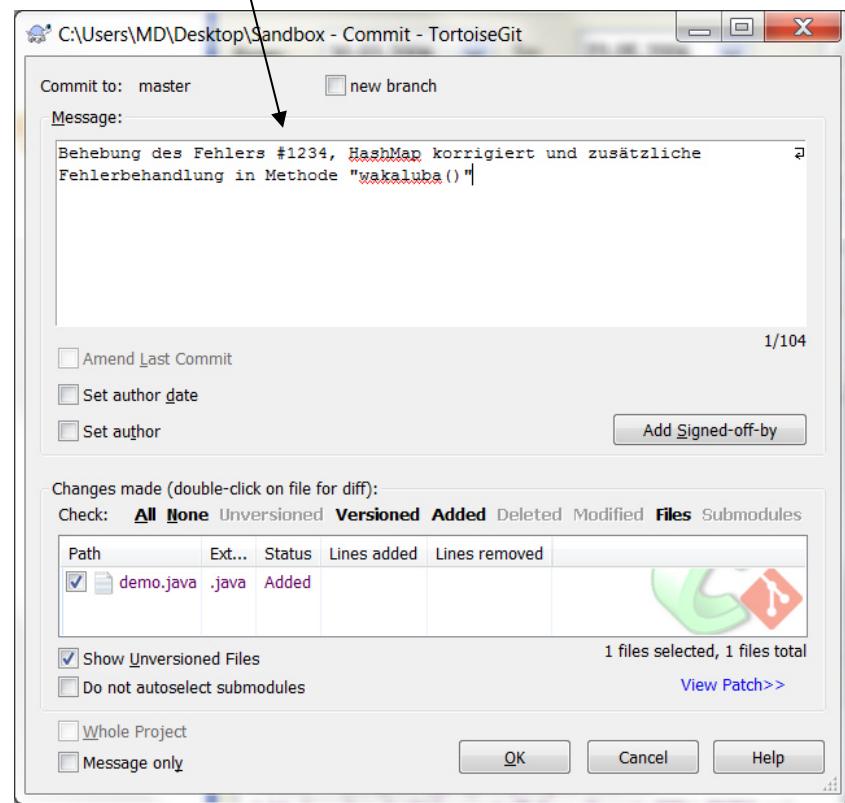
- Automatisch
  - Wer hat Änderung (Commit) durchgeführt?
  - Wann fand der Commit statt?
  - Welche Version der Datei wurde geändert?
  - Ggf. in welchem Status ist die Datei?  
(Working, Released, ...)
  - Versionsinformationen auch innerhalb von (Text-)Dateien mit automatischer Anpassung möglich
- Manuell
  - **Kommentar zur Änderung!**
- Weitere je nach Werkzeug

# Versionsinformationen – Sinnvolle Log Message!

## 03 Konfigurationsmanagement

Bei Commit immer sinnvolle Log  
Message eintragen!

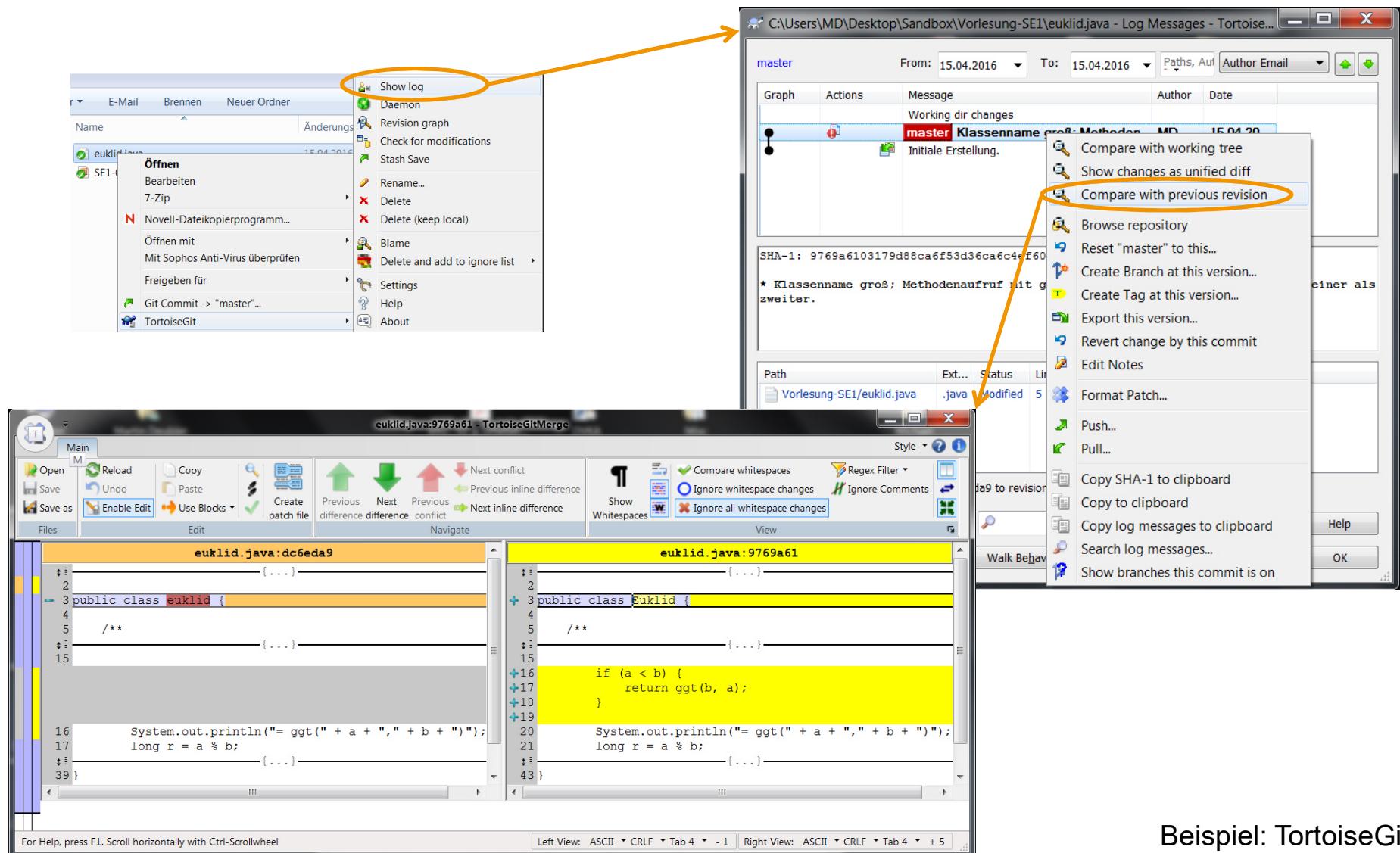
Sinnvoll = Grund für die Änderung  
nennen



Beispiel: TortoiseGit

# Versionsinformationen – Was wurde geändert?

## 03 Konfigurationsmanagement



Beispiel: TortoiseGit



# Grober Ablauf der Nutzung eines Versionskontrollsysteams (1)

## 03 Konfigurationsmanagement

### Initialisierung

1. Lokales Arbeitsverzeichnis herstellen
2. Gegebenenfalls mit einem bestimmten Software-Stand arbeiten

### Eigentliches Arbeiten

3. Synchronisierung der Arbeitskopie mit zentralem Repository (i.d.R. entfernt)
4. Dateien nach Belieben lokal hinzufügen, ändern, ...
5. Dateien (gegebenenfalls erst) unter Versionskontrolle stellen
6. Änderungen an Dateien übernehmen
7. Änderungen auf zentralem Repository zur Verfügung stellen

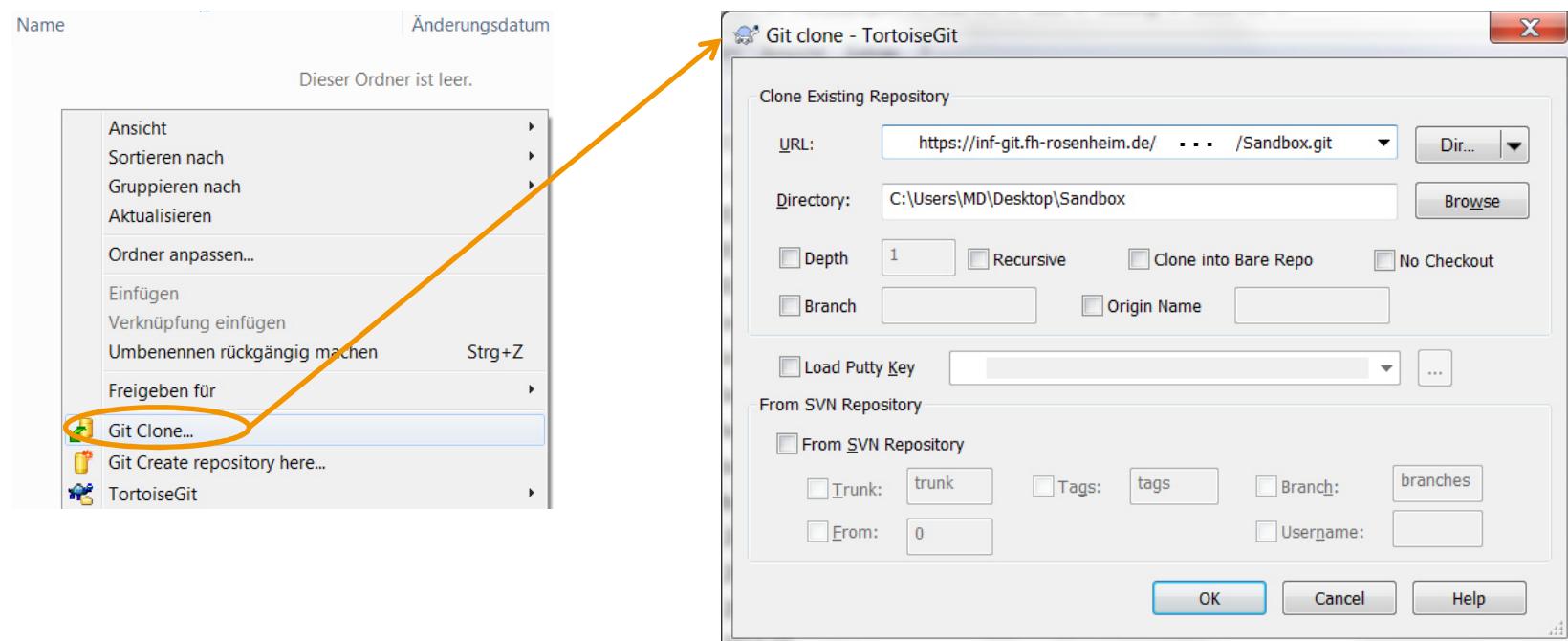
SVN	Git
Checkout	Create (lokal) Clone (entfernt)
Checkout (mit 1.)	Checkout
Update	Pull
...	...
Add	Add
Commit	Commit
Commit (mit 6.)	Push

# Grober Ablauf der Nutzung eines Versionskontrollsystems (2)

## 03 Konfigurationsmanagement

### ● Erstmalige Synchronisation

- Lädt Repository bzw. Dateien aus dem Repository in lokalen Arbeitsbereich
- Dateien: Arbeitskopie („working copy“)



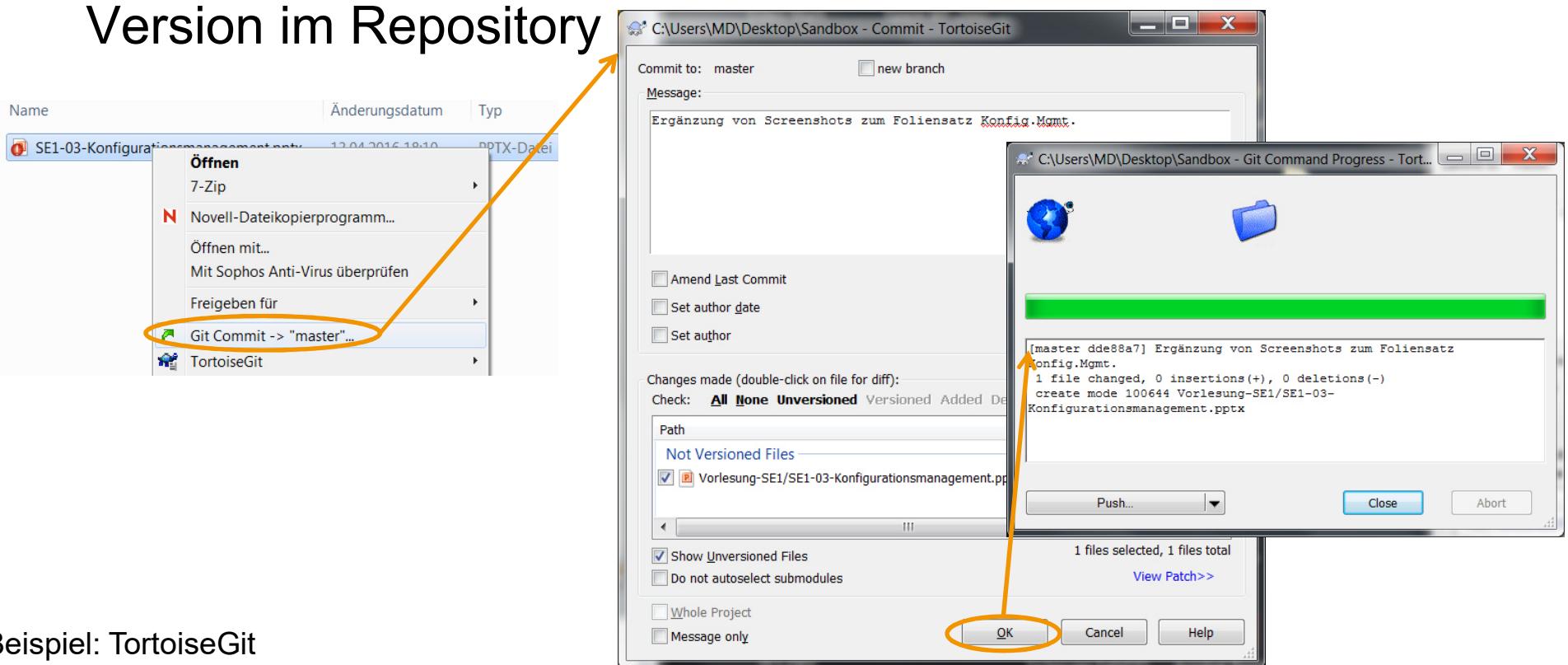
Beispiel: TortoiseGit

# Grober Ablauf der Nutzung eines Versionskontrollsystems (3)

## 03 Konfigurationsmanagement

### Check In / Commit

- Datei wird lokal modifiziert
- Commit speichert die geänderte Arbeitskopie als weitere Version im Repository



Beispiel: TortoiseGit



# Grober Ablauf der Nutzung eines Versionskontrollsystems (4)

## 03 Konfigurationsmanagement

### ● Synchronisation (zentrales) Repository mit Arbeitskopie

- Aktualisierungsfunktion kontrolliert, ob im lokalen Arbeitsbereich eine geänderte Version einer Datei liegt



```
C:\Users\MD\Desktop\Test\Sandbox - Git Command Progress ...
git.exe pull -v --no-rebase --progress "origin"
From C:\Users\MD\Desktop\Sandbox
 * [up to date] master -> origin/master
already up-to-date.

Pulled Diff
```

```
C:\Users\MD\Desktop\Test\Sandbox - Git Command Progress - TortoiseGit
remote: Compressing objects...
git.exe pull -v --no-rebase --progress "origin"
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
From C:\Users\MD\Desktop\Sandbox
88a6789..cabaa77 master -> origin/master
Updating 88a6789..cabaa77
Fast-forward
Vorlesung-SE1/SE1-03-Konfigurationsmanagement.pptx | Bin 896660 -> 897917 bytes
1 file changed, 0 insertions(+), 0 deletions(-)

Pulled Diff
```

```
C:\Users\MD\Desktop\Test\Sandbox - Git Command Progress - TortoiseGit
remote: Compressing objects...
git.exe pull -v --no-rebase --progress "origin"
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
From C:\Users\MD\Desktop\Sandbox
1ba532..88a6789 master -> origin/master
CONFLICT (content): Merge conflict in Vorlesung-SE1/SE1-03-Konfigurationsmanagement.pptx
Automatic merge failed; fix conflicts and then commit the result.
Warning: Cannot merge binary files: Vorlesung-SE1/SE1-03-Konfigurationsmanagement.pptx (HEAD vs.
88a6789c6c4a4dd00900f6c2543747799237c14a)

Pull...
```

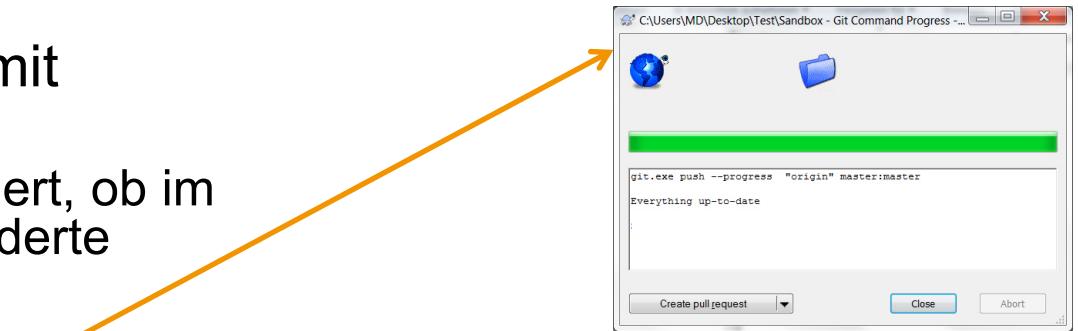
- 3 Möglichkeiten
  - Keine geänderte Version → Kein Update
  - Lokale Arbeitskopie neuer → Kein Update
  - Lokale **Arbeitskopie veraltet** → **Update**
  - Lokale Version und zentrale Version geändert → **Konflikt**  
(anderer Benutzer hat die Datei geändert)

Beispiel: TortoiseGit

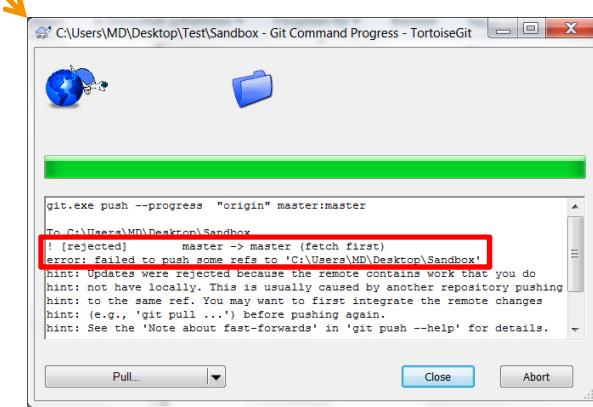
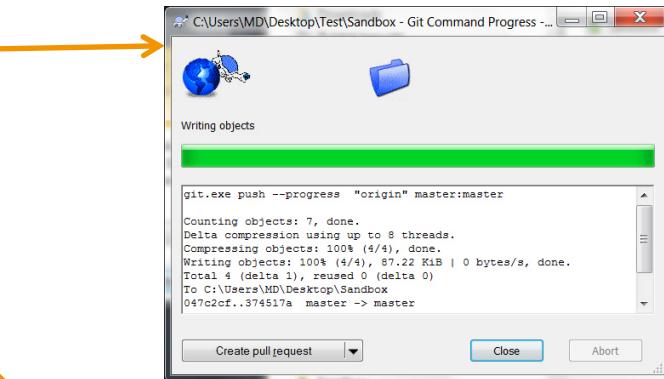
# Grober Ablauf der Nutzung eines Versionskontrollsystems (5)

## 03 Konfigurationsmanagement

- Synchronisation Arbeitskopie mit (zentralem) Repository
  - Aktualisierungsfunktion kontrolliert, ob im zentralen Repository eine geänderte Version einer Datei liegt



- 3 Möglichkeiten
  - Keine geänderte Version → Kein Update
  - Lokale Arbeitskopie älter → Kein Update
  - Lokale **Arbeitskopie neuer** → **Update**
  - Lokale Version und zentrale Version geändert → **Konflikt**  
(anderer Benutzer hat die Datei geändert)



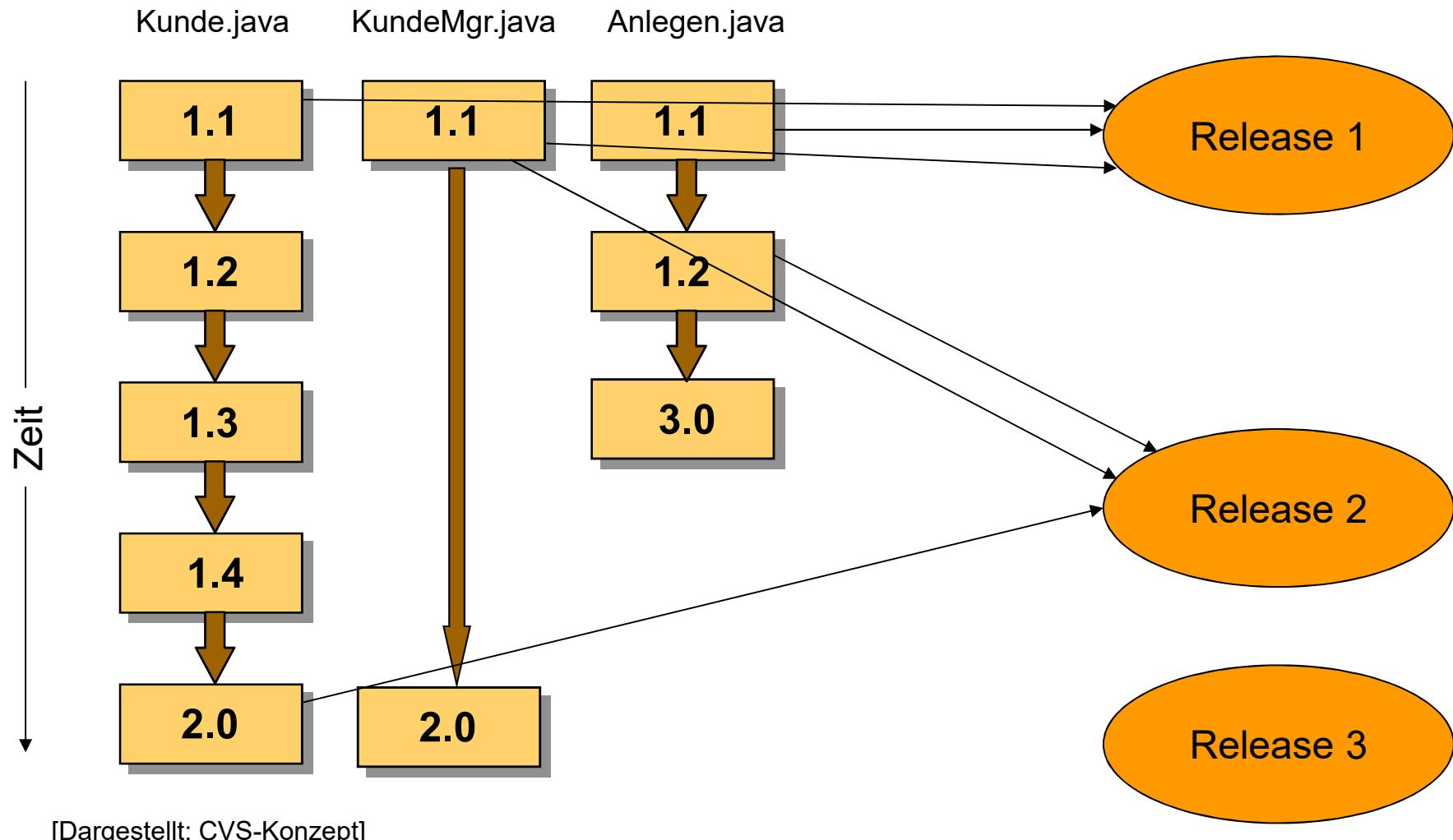
Beispiel: TortoiseGit

## Zwei Strategien, wenn Dateien parallel modifiziert werden

- Konfliktvermeidung (Pessimistisch)
  - = **Lock – Modify – Unlock**
  - Dokument wird beim CheckOut gesperrt
  - Sperre gilt bis zum CheckIn der Änderung
  - Andere Clients dürfen höchstens lesen (Read-only im Dateisystem)
- Konflikterkennung (Optimistisch)
  - = **Copy – Modify – Merge**
  - Dokumente werden niemals gesperrt
  - Beim CheckIn/Synchronisieren wird überprüft, ob inzwischen eine neuere Version gespeichert wurde
    - Wenn ja, wird ein Konflikt gemeldet
    - oder ein automatischer Merge wird durchgeführt
  - Bei Git, aber auch bei Subversion oder CVS

# Konfigurationen (= zusammenpassende Versionen von Dateien)

## 03 Konfigurationsmanagement



# Konfigurationen – Tags, Baselines und Releases

## 03 Konfigurationsmanagement

### Tag

- Bezeichner, kennzeichnet zusammenpassende Versionen von Dateien (= **Konfiguration**)
- Zusammenpassende Versionen können gemeinsam ausgecheckt werden
- Ziel: Wiederherstellbarkeit dieses Standes

### Baseline

- Bezugskonfiguration (zusammenpassende Dateien)
- Bedeutsamer (qualitätsgesicherter) Stand (z.B. Release 1.0)

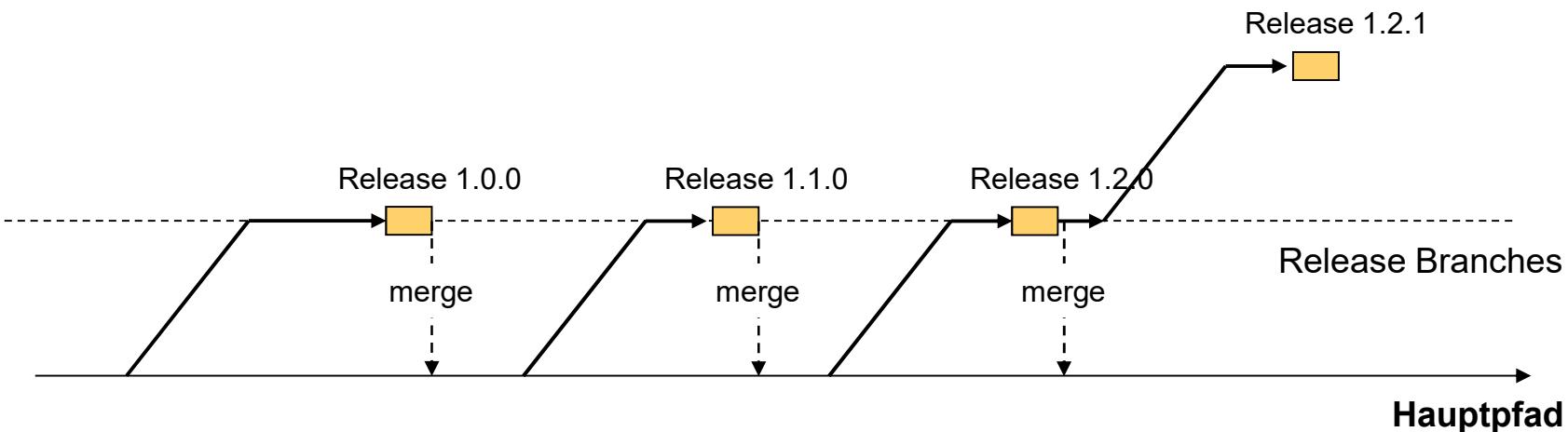
### Release

- Software-Produkt geht in produktiven Betrieb
- Anwender arbeiten damit
- Für jedes Release eine Baseline

# Branches

## 03 Konfigurationsmanagement

- Seltener
  - Linearer Entwicklungspfad
- Häufiger
  - Auslieferung + Bugfixing und parallel Weiterentwicklung
  - Grund: Team auslasten, Entwicklung beschleunigen
- Daher
  - Paralleles Arbeiten an mehreren Versionen
  - **Branch** = Parallel Version der Dateien des Hauptpfades oder eines anderen Branches
  - **Merging** = Zusammenführen von zwei Branches oder eines Branches mit dem Hauptpfad



# Tags und Branches im Repository

## 03 Konfigurationsmanagement

- Dateibasiertes Tagging / Branching (wie in **CVS**)
  - Jede Datei erhält einen Tag / die Branch – Informationen
  - = Sehr aufwändiger Prozess, da jede Datei geändert wird
- Verzeichnisbasiertes Tagging / Branching (**SVN**)
  - Für jeden Tag / Branch wird ein neues Verzeichnis mit einer Kopie (Lazy Copy!) der zu markierenden Dateien angelegt
  - = Einfacher Prozess, da Lazy Copy nur für Änderungen zwischen Original und Kopie Dateien im Repository anlegt
- Objektbasiertes Tagging / Branching (**Git**)
  - Ein neuer Verweis auf ein Commit-Objekt wird erzeugt
  - = Äußert effizienter Prozess, keine Kopien notwendig

- Informationen zu jedem Release
  - Welche Dateien gehören zum Release?
  - Welche Versionen der Dateien gehören zum Release?
  - Grund für das Release?
- Wann wird Release erstellt?
  - Mindestens bei jeder Auslieferung ein Release!
  - Bei konsolidierten Zwischenergebnissen
- Namenskonventionen sind projektabhängig z.B. bei
  - großen Änderungen „Major Release“, z.B. **2.0.0**
  - kleineren Änderungen „Wartungs-Release“, z.B. **1.5.0**
  - „Patch Releases“ bei Bugfixes **1.4.16**

# Konfigurationsmanagement Werkzeuge (nicht vollständig!)

## 03 Konfigurationsmanagement

### ● Kommerzielle Werkzeuge

- Synergy/CM (ex. Continuus, von IBM-Telelogic)
- ClearCase (von IBM-Rational)
- SourceSave/TFS (von Microsoft)
- PVCS
- **Achtung!** Werkzeuge zum Teil sehr aufwändig im Betrieb, ein Vollzeit-Administrator ist notwendig!

### ● Freie Werkzeuge

- Zentralisierte Versionskontrollsysteme
  - CVS
  - Subversion
- Verteilte Versionskontrollsysteme
  - **Git** (hier verwendet)

- Initiator

- Linus Torvalds (= Linux Kernel) in 2005
- Grund: Bitkeeper Lizenz zu teuer

- Wesentliche Problemstellungen

- Sehr viele parallel arbeitende Entwickler, weltweit verteilt
- Auswahl bestimmter Commits sollte möglich sein
- Schnelles Branching / Merging

- Aktuell

- Marktführer bei verteilten Versionskontrollsysteinen
- Viele OpenSource Projekte auf Git umgestiegen
- Große Hosting Plattformen, z.B. GitHub

# Git – Was ist Git?

## 03 Konfigurationsmanagement

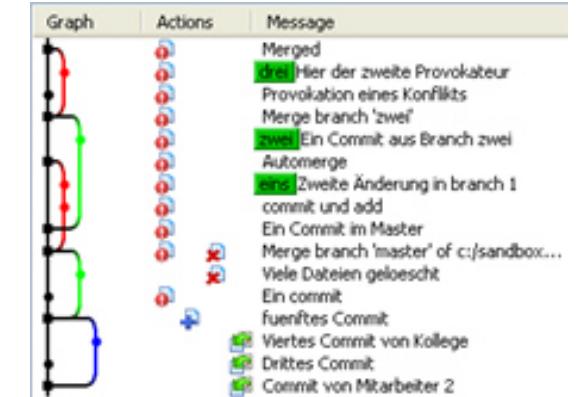
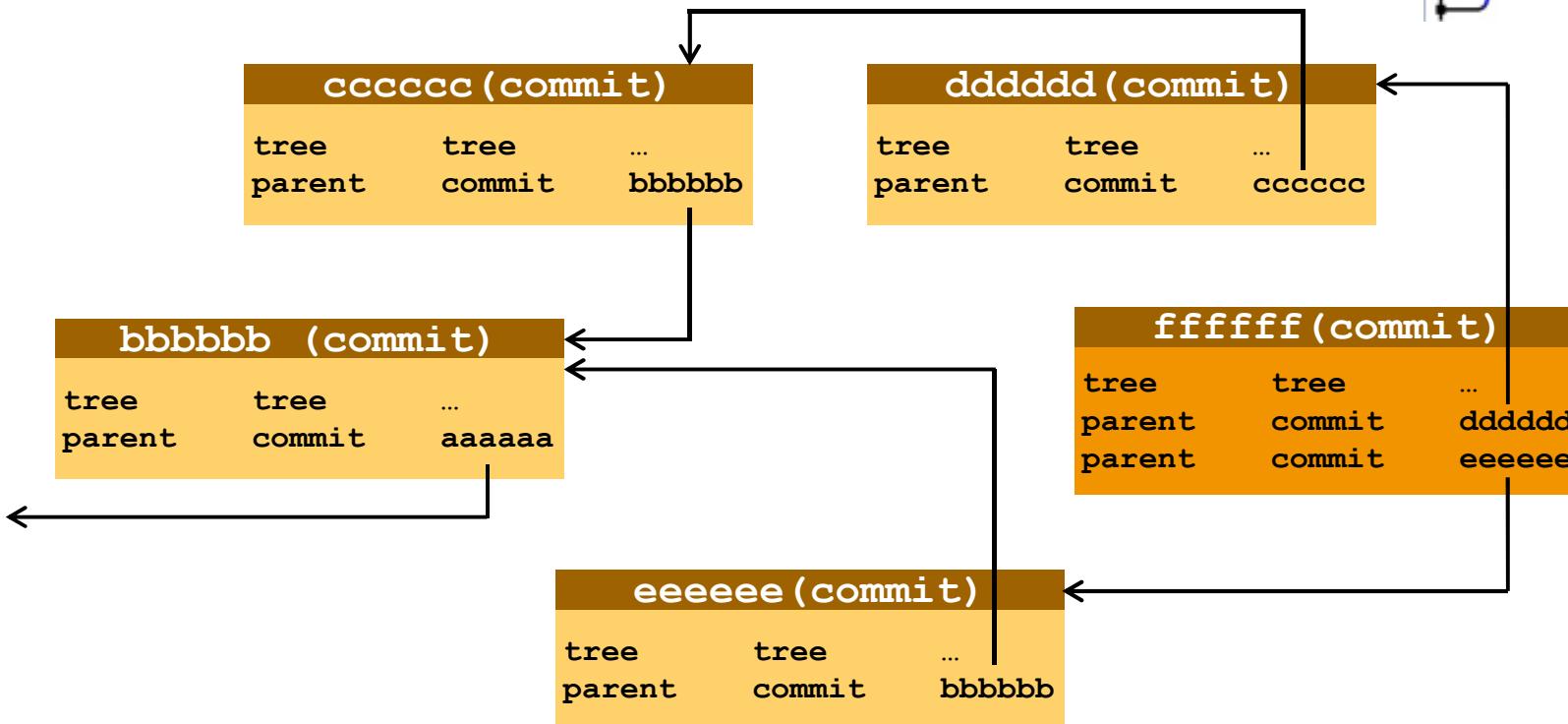
- Git = Dezentrale Versionsverwaltung
  - Keine (dauerhafte) Verbindung zu Versionsverwaltungsserver nötig
  - Lokale Versionierung möglich
  - Synchronisierung mit Server nur bei Bedarf
- Zentrales Element = Commit
  - Repository besteht aus vernetzten Commits und anderen „Objekten“
  - Identifikation durch SHA-1-Hash, 40 Zeichen
    - $2^{160}$  mögliche Schlüssel
    - Sehr geringe Konfliktwahrscheinlichkeit
    - Beispiel: 48c8830f71bcc2b1c083435b0230ce5e2a47b7e1
  - Jedes Commit kennt seine(n) Vorgänger (Parent)

# Git – Commit Vernetzung (1)

## 03 Konfigurationsmanagement

### Commits

- = Gerichteter azyklischer Graph (kein Baum)
- Dadurch leicht darstellbar
  - Branch (Verzweigung)
  - Merge (Zusammenführung)

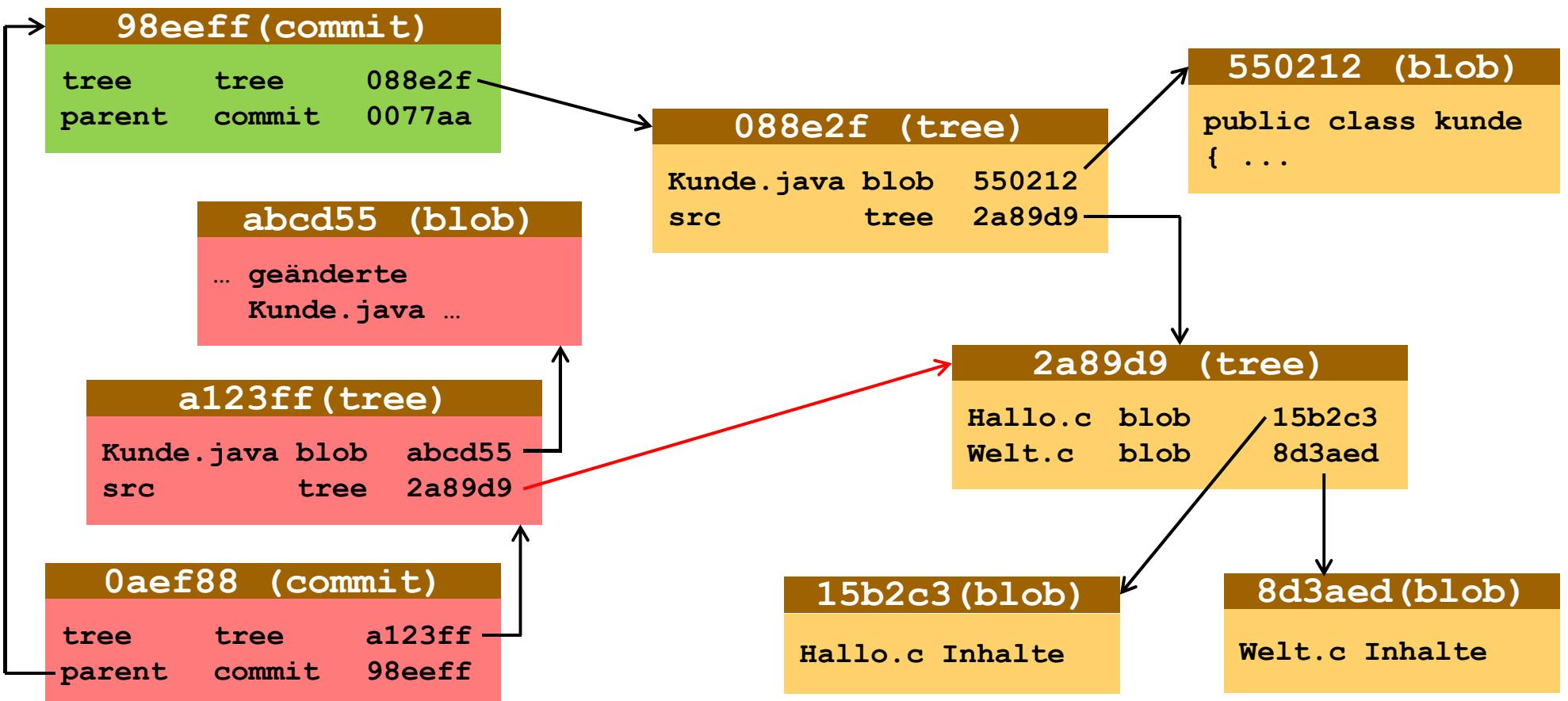


[Commit Vernetzung im Log]

# Git – Commit Vernetzung (2)

## 03 Konfigurationsmanagement

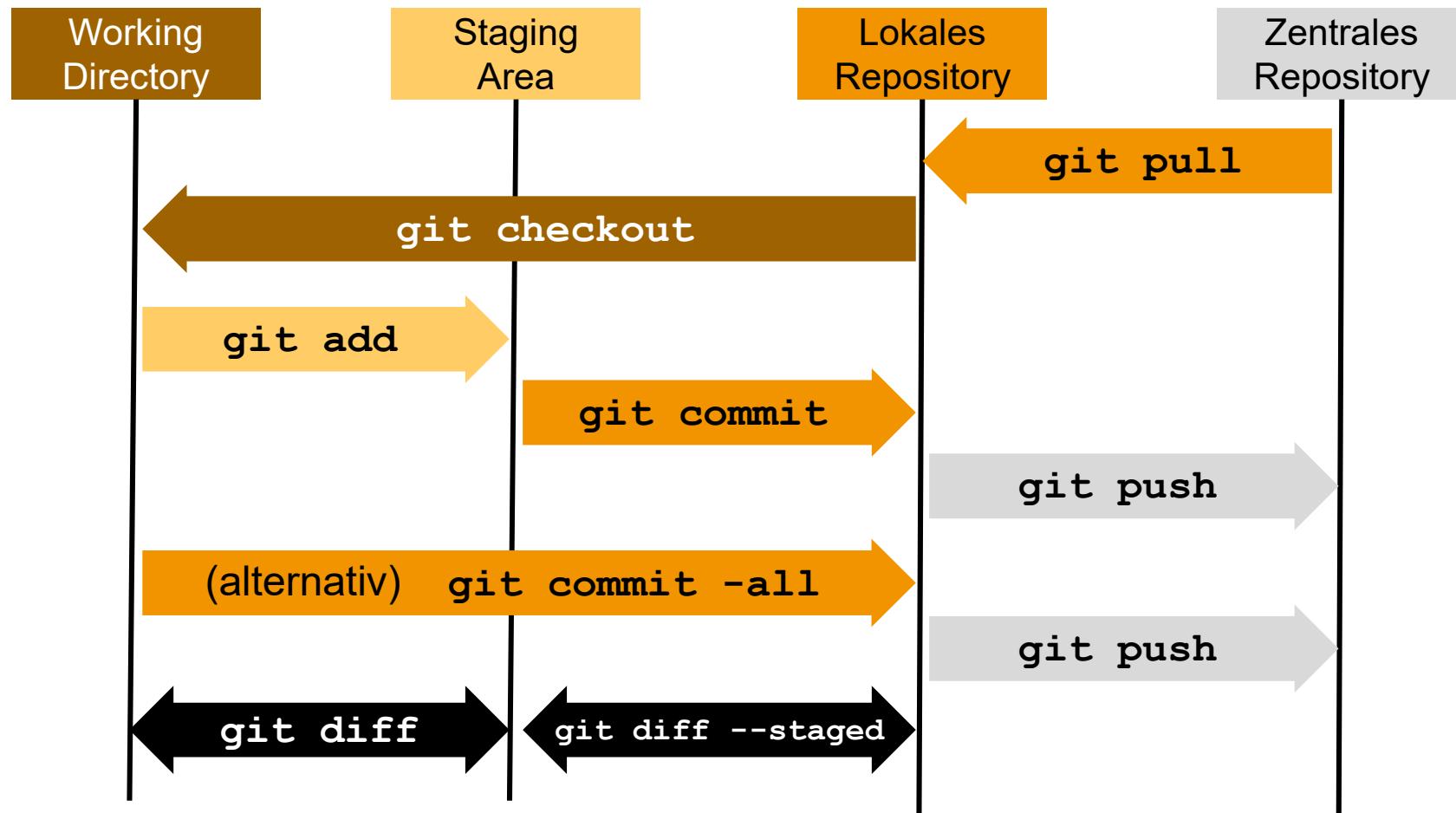
- Commits
  - Wiederverwendung unveränderter Daten



# Git – Hauptbereiche eines Git Projekts

## 03 Konfigurationsmanagement

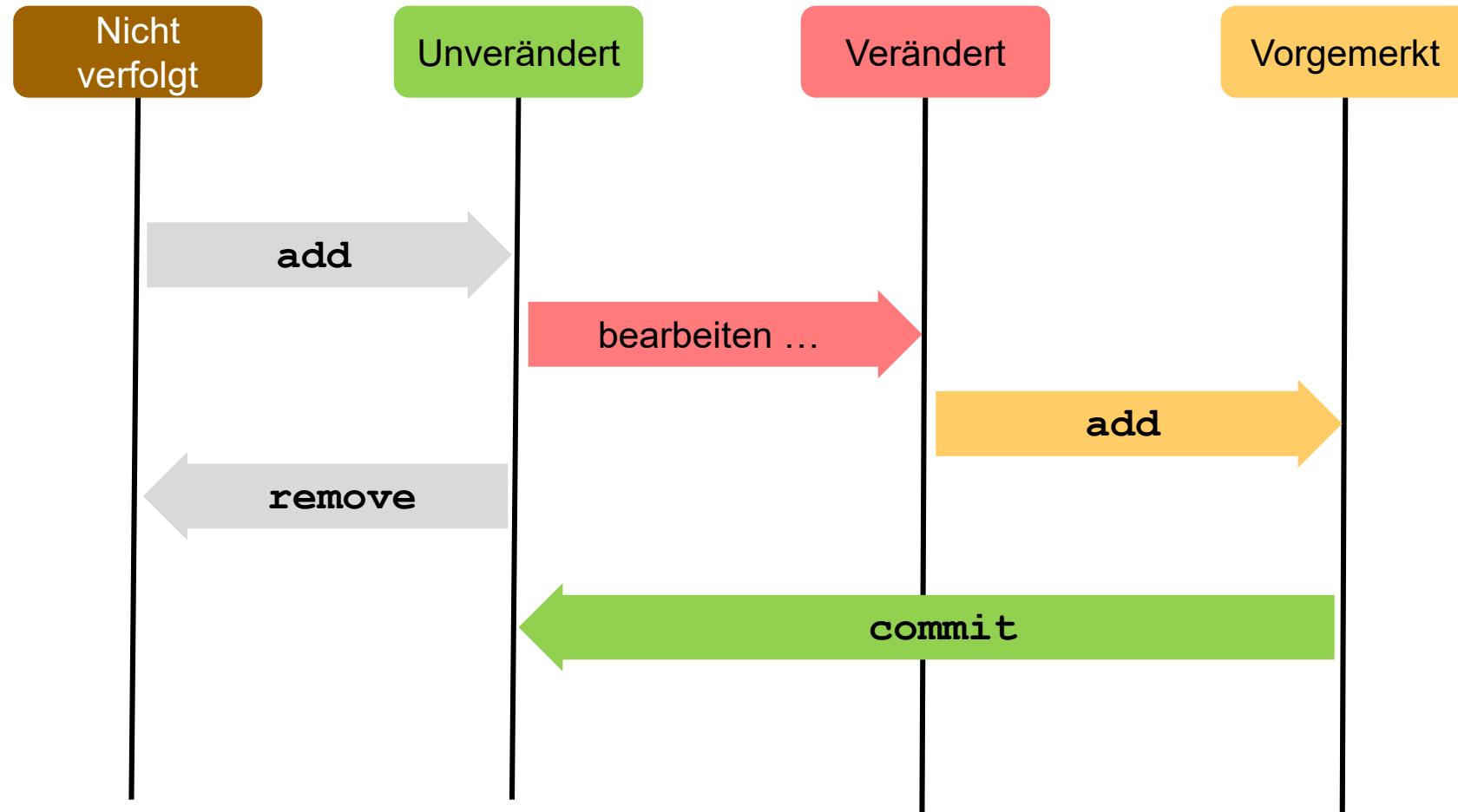
- Arbeiten mit Git
  - Eigene Änderungen sichern und verfügbar machen



# Git – Dateizustände

## 03 Konfigurationsmanagement

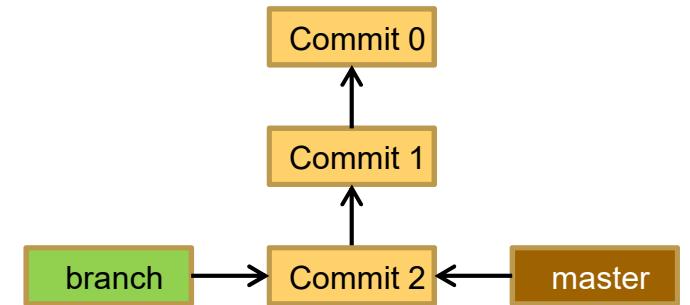
- Arbeiten mit Git
  - Git Dateizustände („File Status Lifecycle“)



# Git – Branching

## 03 Konfigurationsmanagement

- **Branches** = Eigene Code-Zweige mit eigener Änderungshistorie
  - Existieren zunächst nur lokal, können aber auf den Server gepusht werden
- `git branch <branchname>`
  - Legt einen neuen lokalen Branch an, „hinter“ letztem Commit
  - **Vorsicht:** Wird nicht automatisch ausgecheckt
- Wechseln auf Branch
  - `git checkout <branchname>`
- Verfügbare Branches anzeigen
  - `git branch`
- Alle Branches und Änderungen von zentralen Repository holen
  - `git fetch`
  - (nicht alle Branches des zentralen Repositories werden immer vorgehalten)



# Git – Merging

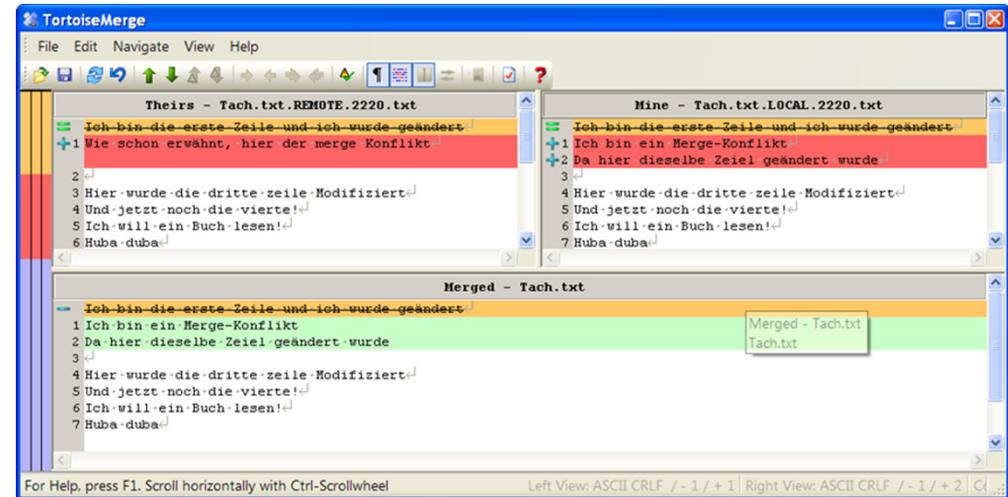
## 03 Konfigurationsmanagement

### ● `git merge <branchname>`

- Zusammenführung von Branch auf den aktuellen
- Weitgehend automatisches Zusammenführen (3 Wege)

### ● Konflikte

- Zum Beispiel bei Änderungen an derselben Stelle
  - Manuell lösen mit `git mergetool`
  - Eigene Änderungen komplett übernehmen  
`git checkout --ours`
  - Die anderen Änderungen komplett übernehmen  
`git checkout --theirs`



# Git – Kompakt

## 03 Konfigurationsmanagement

- **git clone**
  - Repository (zum 1. Mal) kopieren, .git Verzeichnis wird lokal angelegt
- **git init --bare**
  - Leeres Master Repository (ohne Workspace) anlegen
- **git add / git add .**
  - Datei/en unter Versionskontrolle stellen bzw. in Stage Bereich legen
- **git status**
  - Anzeigen von Änderungen
- **git commit –m „Message“**
  - Änderungen (vom Stage Bereich) in lokales Repository speichern
- **git push**
  - Änderungen ins zentrale Repository überspielen (Achtung: vorher git pull)
- **git pull**
  - Holen der aktuellen Änderungen auf dem zentralen Repository
- **git checkout**
  - Anderen Branch verwenden bzw. wechseln auf Branch
- **git branch**
  - Anzeigen von Branches bzw. Anlegen eines neuen lokalen Branch
- **git fetch**
  - Alle Branches + Änderungen von zentralem Repository holen
- **git merge**
  - Zusammenführen von Branches/Dateien

# KM-Handbuch zur Festlegung des Konfigurationsmanagements (1)

## 03 Konfigurationsmanagement

### 1. Einleitung

- 1.1 Inhalt des Dokuments
- 1.2. Leserkreis
- 1.3 Projekthomepage

### 2. Konfigurationselemente

- 2.1 Übersicht
- 2.2 KE: <Name>  
(Abschnitt für jedes Konfigurationselement)

### 3. Projektumgebung

- 3.1 Repository Struktur
- 3.2 Verzeichnisstruktur des Arbeitsbereiches
- 3.3 Werkzeuge
- 3.4 Externe Komponenten

### 4. Verwaltung der Konfigurationselemente **(Versionsverwaltung)**

- 4.1 Erstmaliger Checkout des Arbeitsbereichs
- 4.2. Hinzufügen, Ändern und Löschen von Dateien
- 4.3 Durchführen einer strukturellen Änderung
- 4.4 Erstellen von Tags
- 4.5 Erstellen von Branches

### 5. Änderungs- und Fehlermanagement

- 5.1 Rollen, Rechte und Pflichten
- 5.2 Aufbau des Change Control Boards
- 5.3 Erfassung, Bewertung und Bearbeitung von Fehlermeldungen
- 5.4 Erfassung, Bewertung und Bearbeitung von Änderungswünschen

### 6. Release-Management

- 6.1 Release Plan
- 6.2 Auslieferung eines regulären Releases
- 6.3 Auslieferung eines Patches

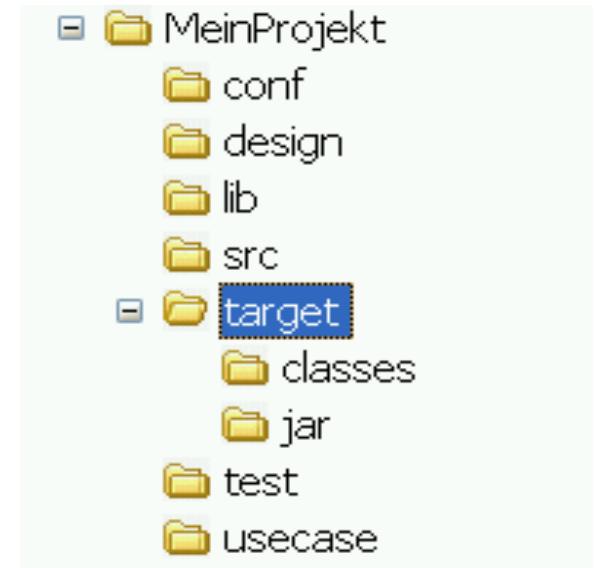
### 7. Audits, Metriken und Berichte

- 7.1 Audit-Plan
- 7.2 Manuelle Metriken
- 7.3 Automatisierte Metriken
- 7.4 Veröffentlichte Berichte

Siehe auch: G. Popp: Konfigurationsmanagement

- Zu Repository Struktur (3.1) und Verzeichnisstruktur des Arbeitsbereiches (3.2)

- Projektstruktur am Anfang des Projektes festlegen und kontinuierlich aufräumen und anpassen
- Projektstruktur im KM-Handbuch oder Projekt-Wiki dokumentieren
- Häufig:  
Pro Konfigurationselement ein Verzeichnis (*usecase*, *design*, *src*, ...)
- Wichtig, da auch die Build-Automatisierung darauf aufsetzt



### ● Zu Werkzeuge (3.3) und Externe Komponenten (3.4)

- Projekt hängt ab von Werkzeugen und externen Komponenten
  - Keine Übersetzung ohne **Compiler** und **Basisbibliotheken!**
  - Keine Änderung des Modells ohne **UML-Werkzeug**
  - Keine Neugenerierung ohne (**selbstgebauten**) Generator
  - Keine Änderung der Doku ohne **Textverarbeitung**
- Risiko
  - Bei Änderungen ist ein notwendiges Werkzeug oder eine bestimmte (Open Source) Komponente nicht mehr verfügbar
- Wichtig
  - **Abhängigkeiten** erkennen und dokumentieren
- Daher
  - Werkzeuge und dazu notwendige Infrastruktur archivieren (Betriebssystem, Hardware, ...)
  - Externe Komponenten (möglichst mit Quellen) archivieren (nicht auf das Internet verlassen!)