

# Kapitel 1

# Software Configuration Management

**Stand: 6.10.2019**

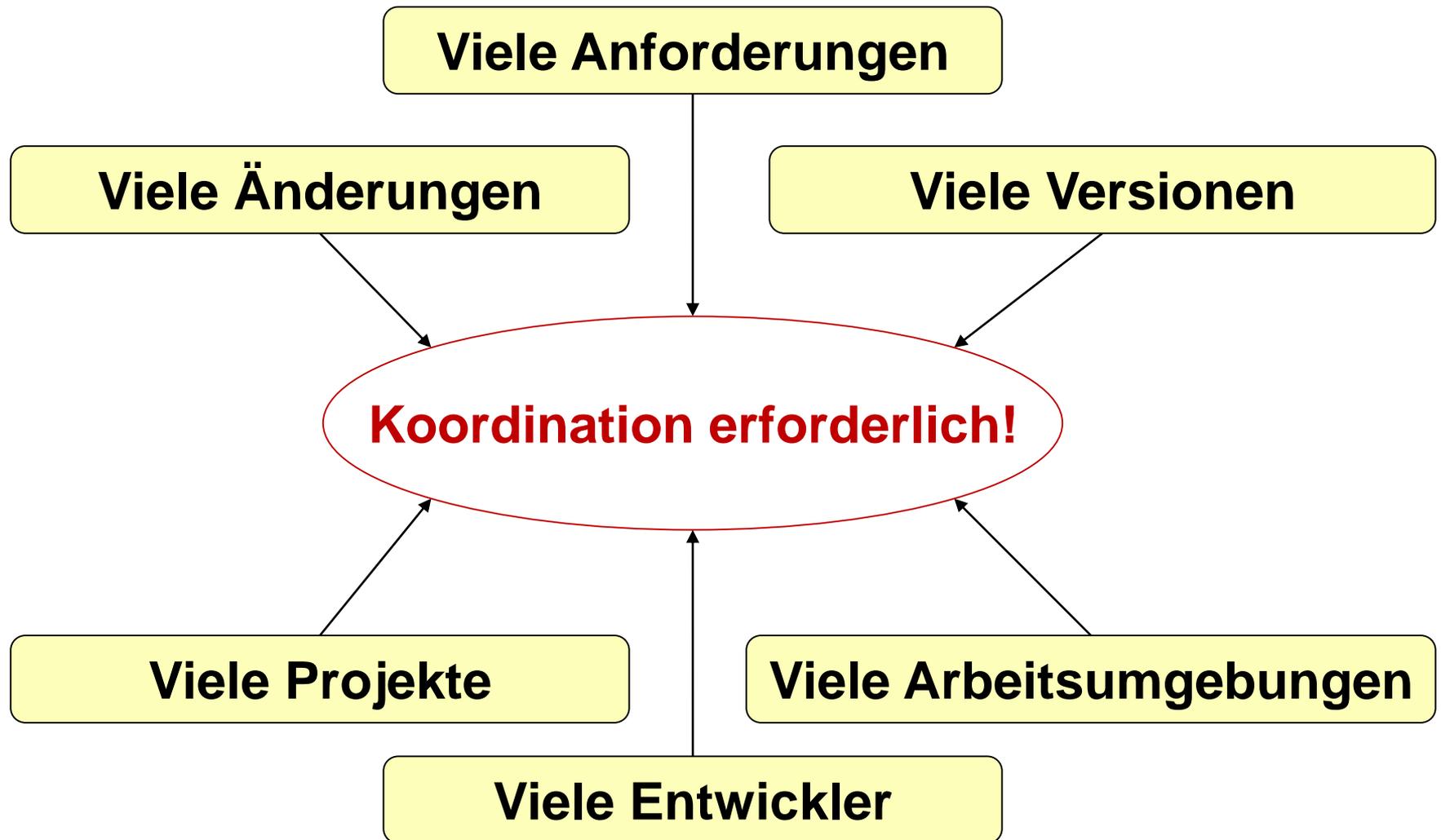
Motivation

Grundlagen

Zentrale Ansätze

Dezentrale Ansätze

# Herausforderungen während der Softwareentwicklung



# Warum Software Configuration Management?

## Problem

- Software-Entwicklung ist nicht linear
  - ◆ Man macht Programmierfehler
  - ◆ Man trifft falsche Entwurfsentscheidungen
- Software-Entwicklung ist Teamarbeit
  - ◆ Sie und andere arbeiten parallel
  - ◆ ... auf vielen verschiedenen Dateien
- Verwaltung verschiedener Versionen
  - ◆ Kunde erhält stabile Version, während Entwicklung weitergeht
  - ◆ Fehlerkorrekturen müssen in alle Versionen integriert werden
  - ◆ Verschiedene Kunden erhalten verschiedene Varianten des Produkts

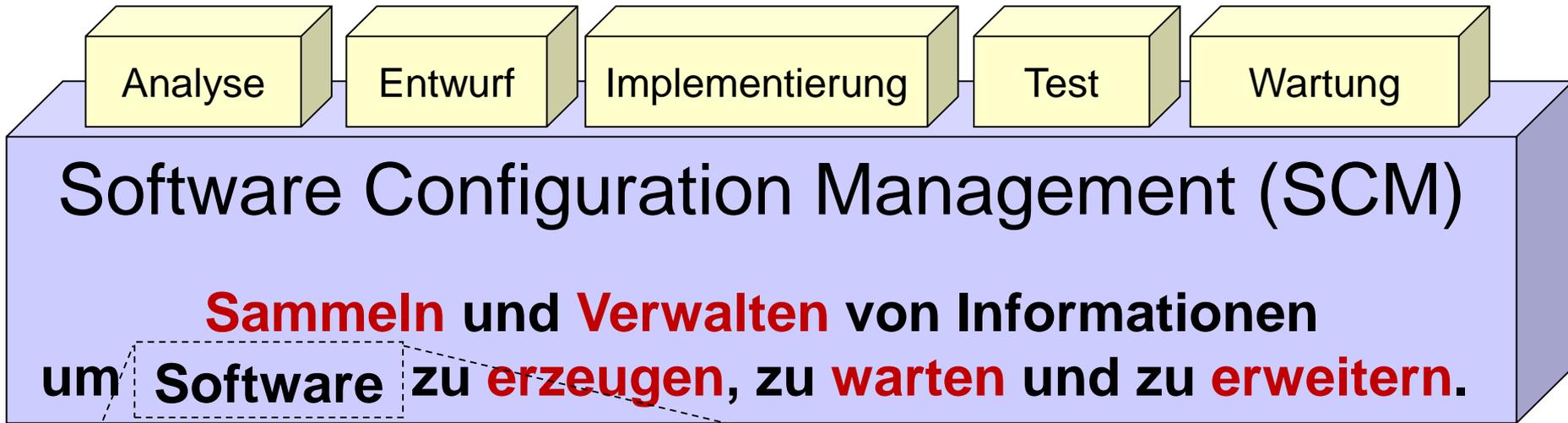
## Sie möchten

→ wissen, was Sie wann warum getan haben  
→ zu alter Version zurückgehen

→ wissen, wer was wann warum getan hat  
→ Änderungen gemeinsam nutzen

→ parallele Entwicklung und Integration kontrollieren  
→ Varianten kontrollieren

# SCM: Das Fundament des Entwicklungsprozesses



- ✓ Quellcode
- ✓ Binärcode
- ✓ Build Skripte
- ✓ Konfigurationen
- ✓ Bitmaps & JPEGs
- ✓ HTML/XML Dateien
- ✓ CGI, Javascript
- ✓ CSS
- ✓ Anforderungen
- ✓ Entwürfe
- ✓ Test Skripte
- ✓ Dokumentation

# SCM Funktionalitäten

---

- Verwaltet alle Komponenten eines Projekts in einem „Repository“
  - ◆ Keine redundanten Kopien
  - ◆ Sicher
- Macht Unterschiede zwischen Versionen sichtbar
  - ◆ „Was hat sich verglichen mit der gestrigen Version verändert?“
- Erlaubt Änderungen zu identifizieren, auszuwerten, zu diskutieren (!) und sie schließlich anzunehmen oder zu verwerfen.
- Verwaltet Metadaten: Wer hat was, wann, warum und wo getan?
  - ◆ „Wer hat was in Klasse X geändert?“
  - ◆ „Warum hat er es verändert?“
- Ermöglicht die Wiederherstellung voriger Zustände
  - ◆ Vollständig oder selektiv
- Erlaubt die Definition von Referenzversionen („Tags“)
  - ◆ Release (Veröffentlichte Version)
  - ◆ Milestone (interne Version, Demo beim Kunden, ...)
  - ◆ Wichtiger Zwischenstand (fehlerfrei getestete Version, Stand vor ...)

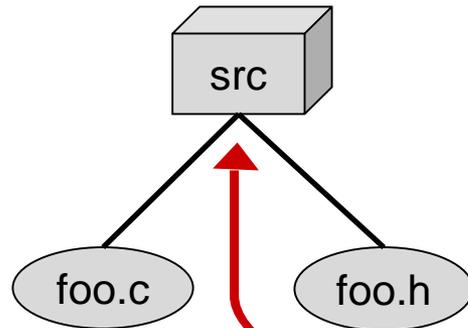
# Grundlegende Ansätze

Vault Model  
Virtual File System

# Arbeitsumgebung und Repository

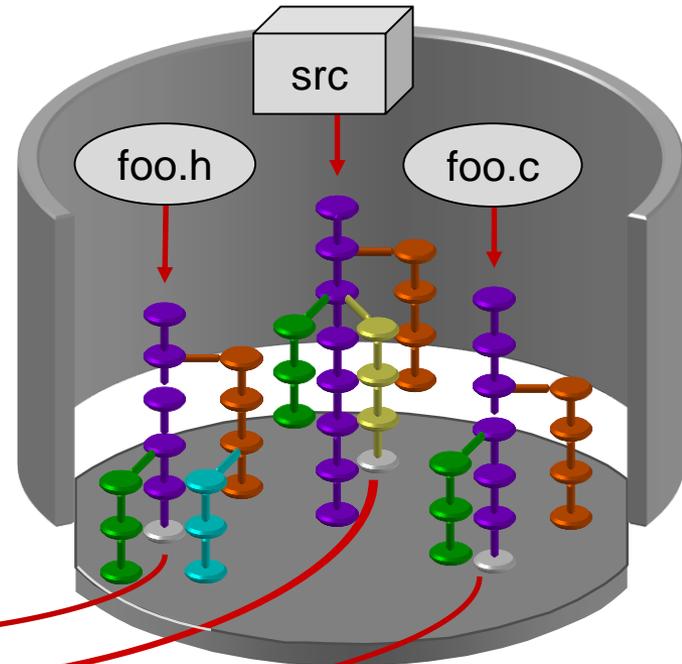
## Arbeitsumgebung (“Sandbox”)

- Enthält nur die aktuell relevante Version eines jeden Artefaktes aus dem Repository
  - ◆ die Aktuellste
  - ◆ die letzte Funktionierende
  - ◆ ...

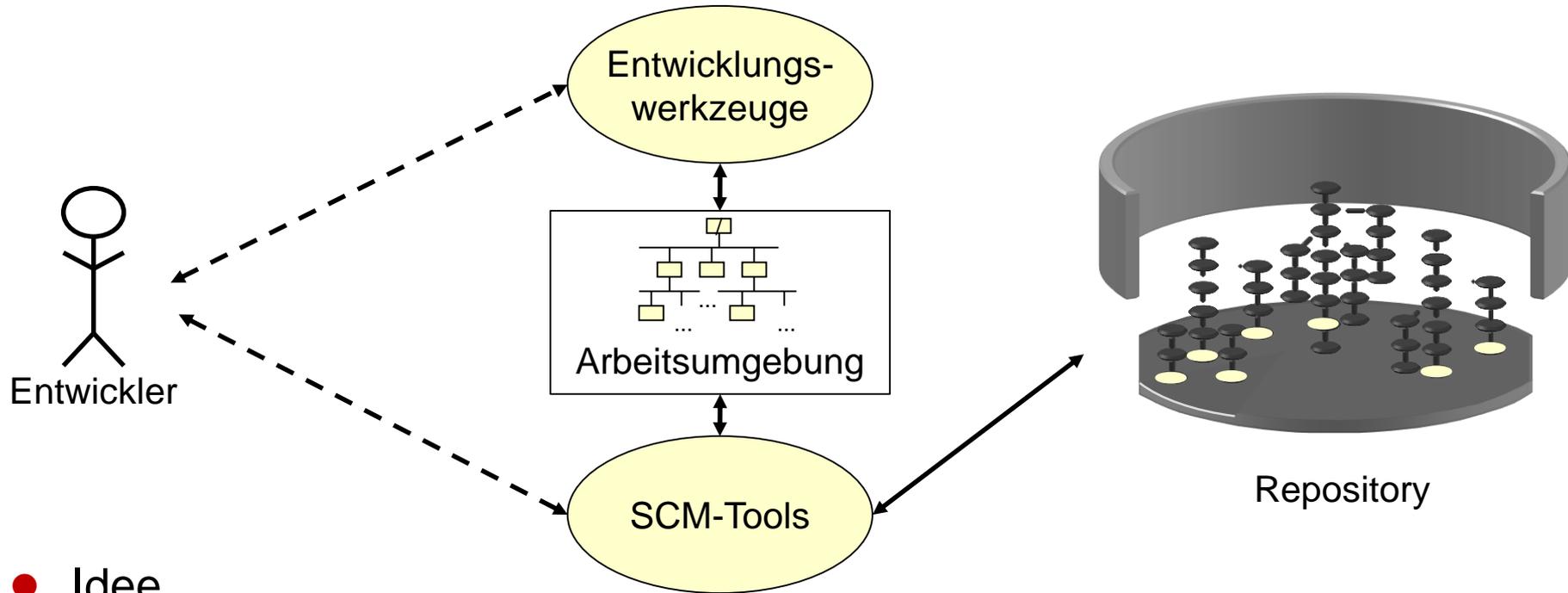


## Repository

- Enthält alle Versionen (incl. “branches”) eines jeden Artefaktes, das unter SCM-Kontrolle stehen



# SCM-Ansätze: „Vault Model“



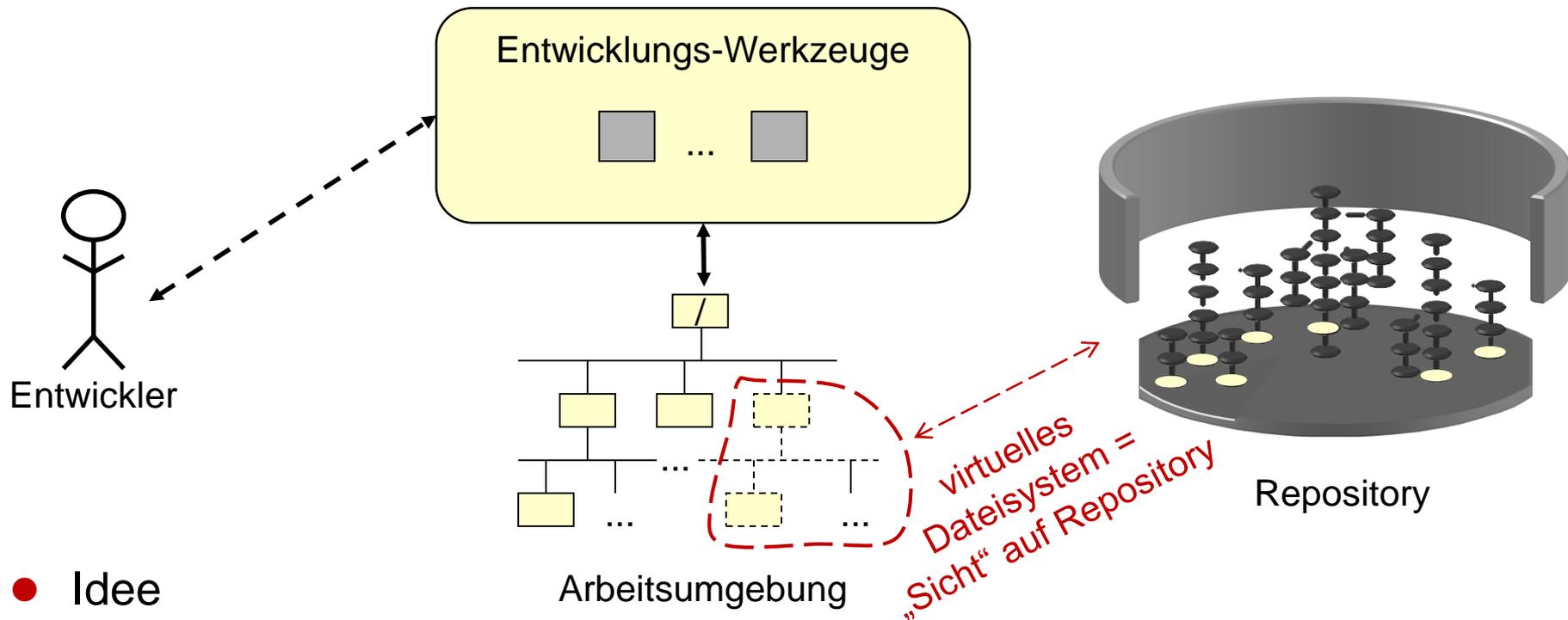
- Idee

- ◆ Dateien kopieren: Arbeitsumgebung  $\leftarrow \rightarrow$  Repository

- Probleme

- ◆ Evtl. **viele private Kopien** in Arbeitsumgebungen außerhalb des Repository
- ◆ Entwickler arbeitet evtl. auf **veralteten** Dateien
- ◆ Möglicher Datenverlust durch **gleichzeitige** oder **abgebrochene** Updates

# SCM-Ansätze: "Virtual File System" (VFS)



- Idee

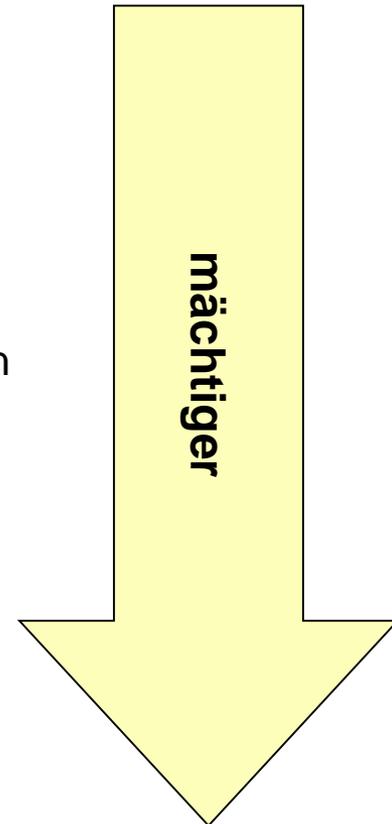
- ◆ Abfangen von I/O-Operationen des Betriebssystems (open, read, write) und Umleiten zum Repository

- Vorteile

- ◆ Transparenz, da das Repository als **normaler Verzeichnisbaum** erscheint
  - ⇒ nahtlose Integration bestehender Standardsoftware
  - ⇒ Benutzer kann wie gewohnt arbeiten

# Verbreitete SCM-Tools die obige Ansätze umsetzen

- ◆ CVS (simpel, kostenlos)
  - ⇒ **Repository = Dateisystem**
  - ⇒ Versionierung von Dateien
  - ⇒ Hauptsächlich Textdateien, Binärdateien nur eingeschränkt
- ◆ SVN (besser, kostenlos)
  - ⇒ **Repository = Datenbank** ⇒ Transaktionsunterstützung
  - ⇒ Versionierung von Dateien und Ordnern ⇒ Umbenennungen
  - ⇒ Text und Binärdateien
  - ⇒ Komfortable graphische Benutzeroberflächen
- ◆ ClearCase (high-end, kommerziell)
  - ⇒ **Virtuelles Dateisystem** basierend auf **Datenbank**
  - ⇒ Alle Eigenschaften von SVN plus...
    - Regelbasierte, dynamische Sichten auf Repository
    - „Derived Object Sharing“
    - Multiple Server, Replikation
    - Prozessmodellierung (Event-Condition-Action Regeln)



# Arbeiten mit SVN (und CVS)

Check-in und Check-out

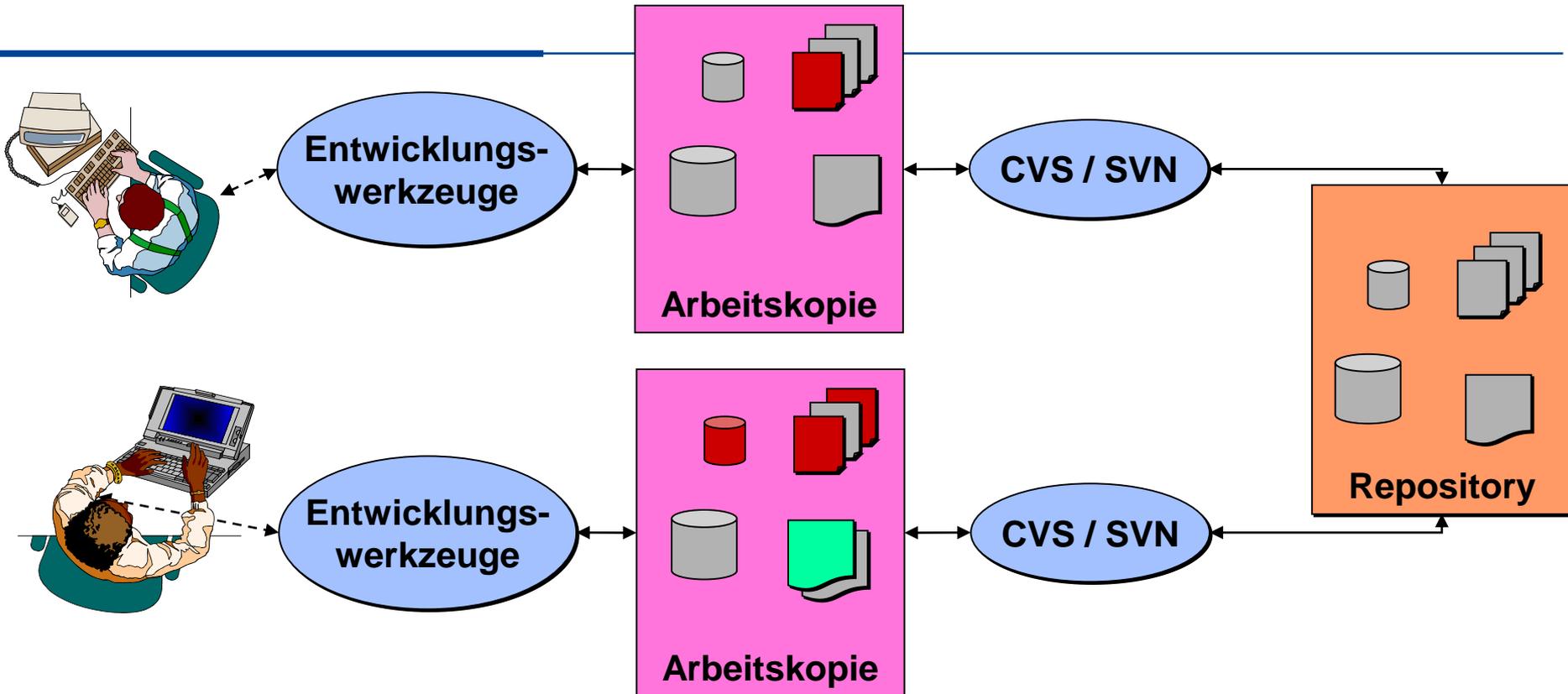
Commit und Update

Synchronisierung

Konfliktbeurteilung durch Dateivergleich

Manuelle Konfliktauflösung

# CVS und SVN: "Vault Model"

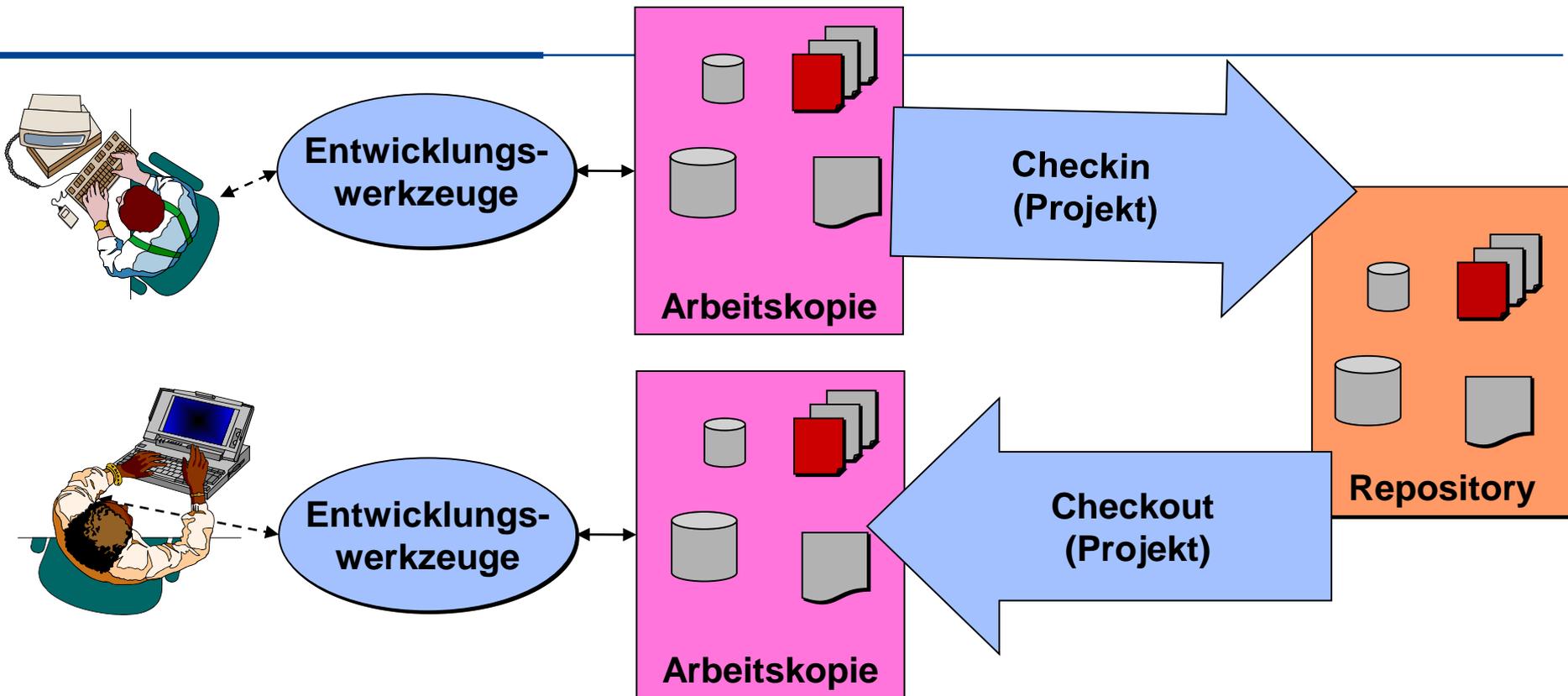


## □ Prinzip

- Ein zentrales Sammelbecken („Repository“) aller relevanter Dateien
  - Nur „offizielle“ Versionen
- Viele private Arbeitsumgebungen („Sandbox“) mit Kopien von Dateien
  - Auch temporäre, inkonsistente, unfertige, ... Versionen

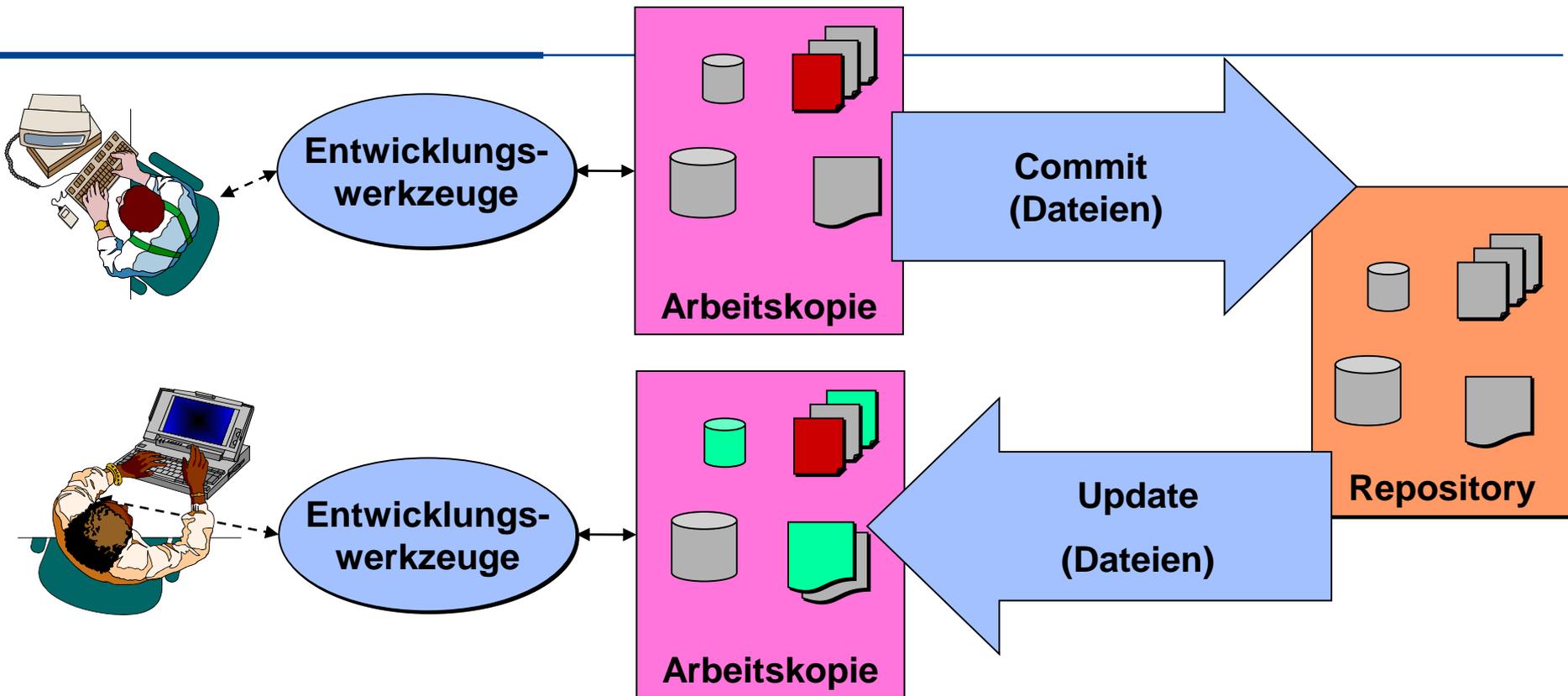


# CVS und SVN: Checkin und Checkout



- Checkin
  - Fügt **Projekt** dem Repository hinzu
- Checkout
  - Erstellt eine Arbeitskopie des **Projekts** vom Repository

# CVS und SVN: Commit und Update



## □ Commit

- Transferiert vom Programmierer geänderte **Dateien** in das Repository

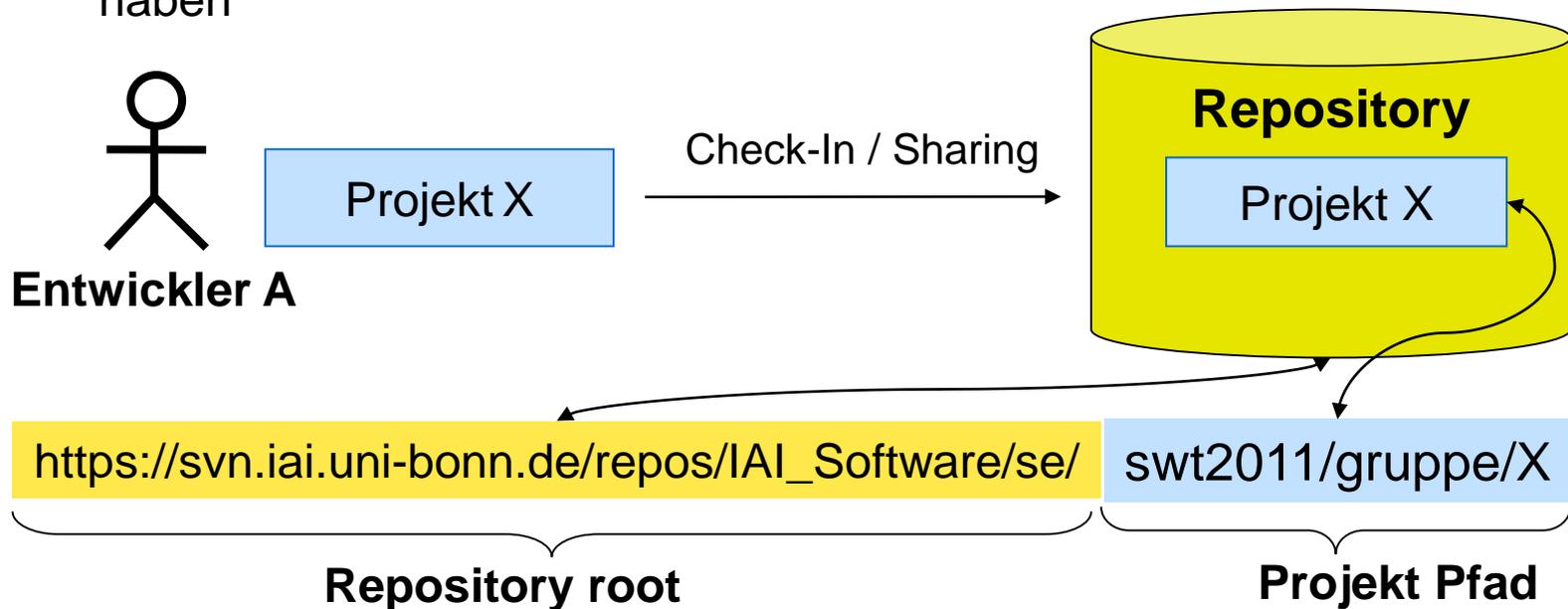
## □ Update

- Transferiert geänderte **Dateien** vom Repository in die Arbeitskopie

# Check-In:

## Projekt unter Versionskontrolle stellen

- Ausgangssituation
  - ◆ Ein Repository existiert
  - ◆ Ein Projekt existiert, wird aber noch nicht gemeinsam genutzt.
- Check-In / Sharing / Promotion des Projektes
  - ◆ Das Projekt wird dem Repository hinzugefügt
  - ◆ Es ist nun für alle Entwickler verfügbar die Zugriff auf das Repository haben

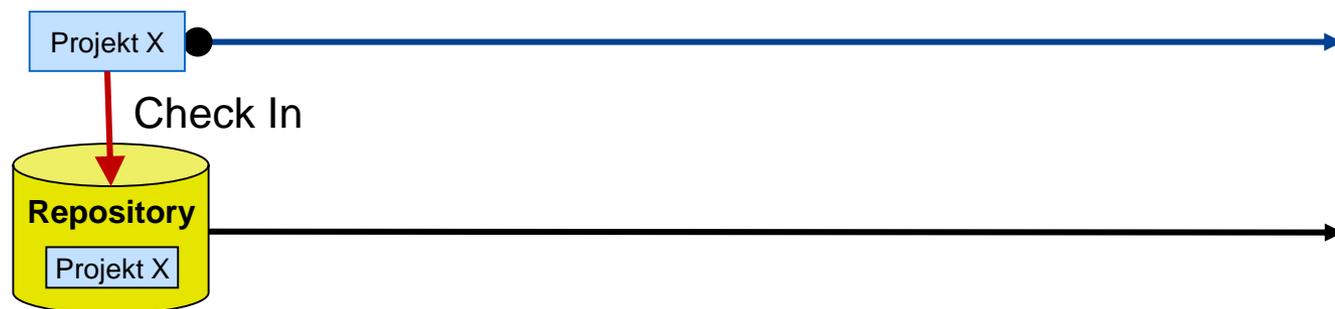


# Check-In: Projekt unter Versionskontrolle stellen

- Ausgangssituation
  - ◆ Ein Repository existiert
  - ◆ Ein Projekt existiert, wird aber noch nicht gemeinsam genutzt.
- „Check-In“ / „Sharing“ / „Promotion“ des Projektes
  - ◆ Das Projekt wird dem Repository hinzugefügt
  - ◆ Es ist nun für alle Entwickler verfügbar die Zugriff auf das Repository haben

Arbeitskopie von  
Entwickler A

Projekt im Repository



# Check-Out:

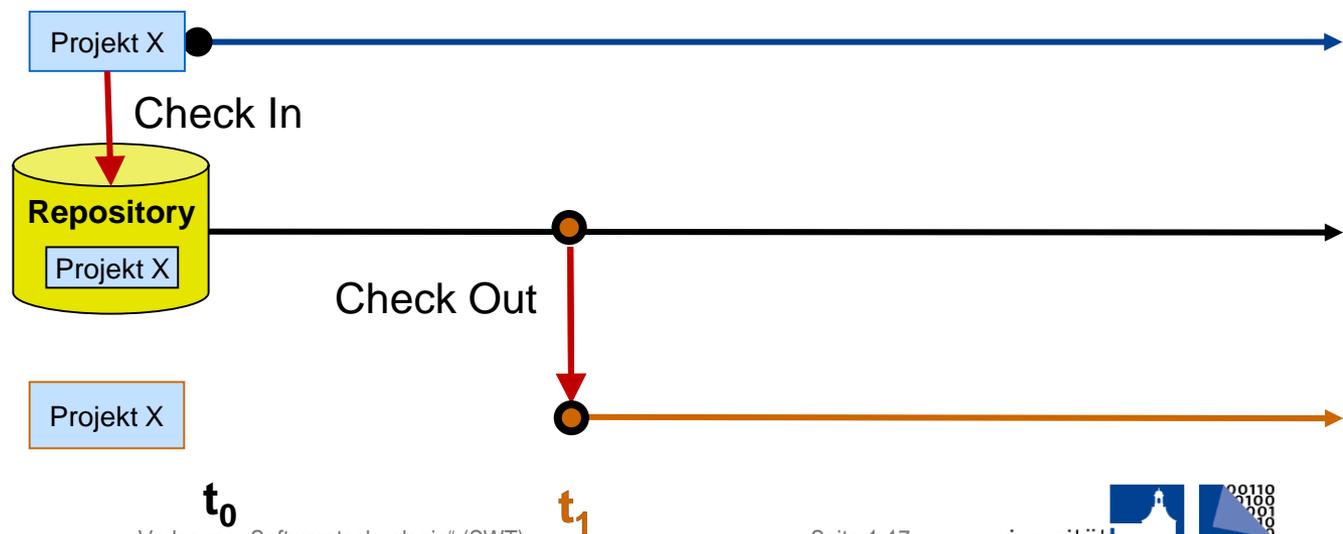
## Initialer Projektdownload aus Repository

- Entwickler bekommt eine lokale Arbeitskopie eines Projektes
- Auschecken wird nur einmal pro Projekt gemacht!  
(Neue Version aus Repository holen → siehe „Update“)

Arbeitskopie von  
Entwickler A

Projekt im Repository

Arbeitskopie von  
Entwickler B



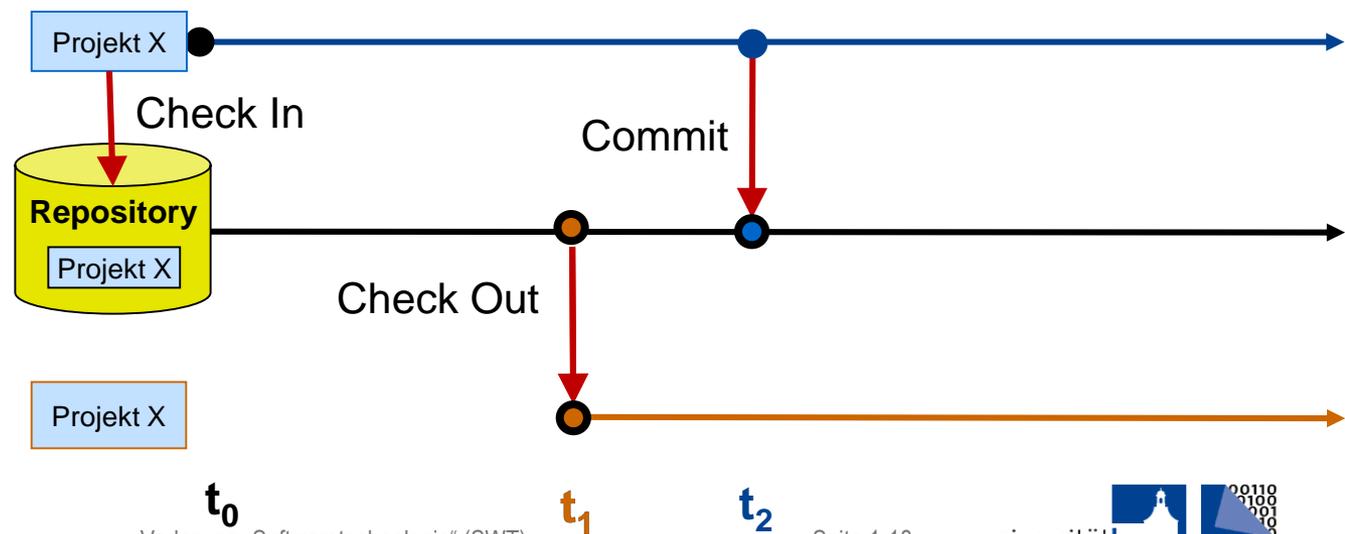
# Commit: Änderungen in das Repository übertragen

- Ausgangslage: Entwickler A hat seine Arbeitskopie geändert
- Mittels „Commit“ fügt er seine Änderungen dem Repository hinzu
- Mit einem „Commit Kommentar“ teilt er dem Team mit was er warum geändert hat: „NullPointerException behoben, die auftrat wenn ...“

Arbeitskopie von  
Entwickler A

Projekt im Repository

Arbeitskopie von  
Entwickler B



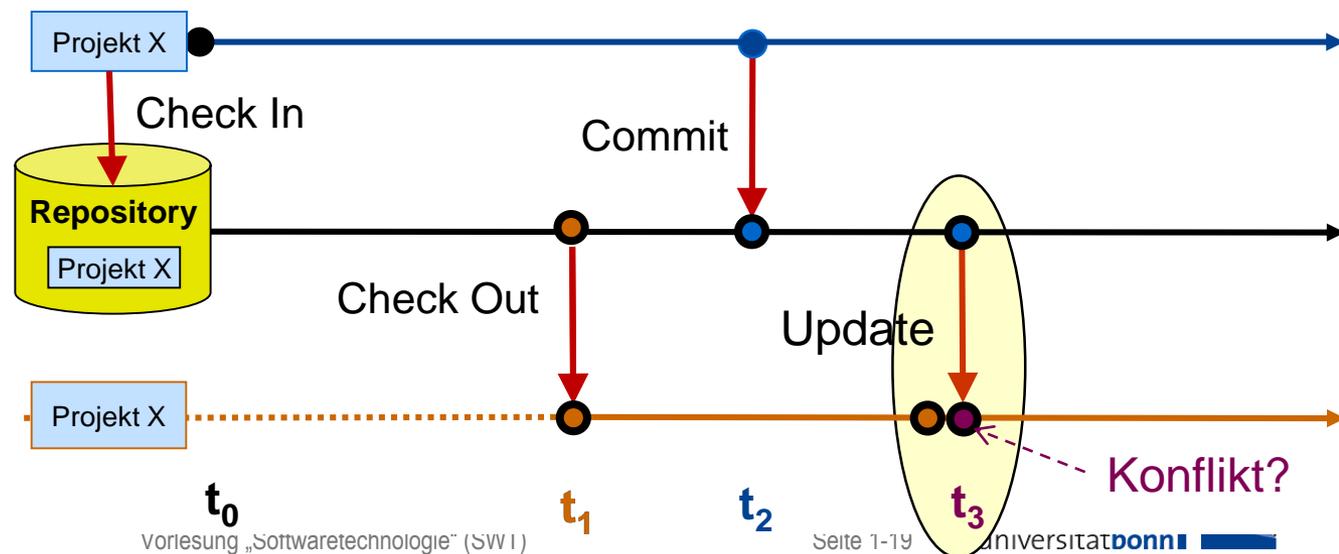
# Update: Änderungen vom Repository übernehmen

- Ausgangslage: Repository hat sich geändert
  - ◆ Entwickler B ist sich **sicher(!)**, dass er die Änderungen übernehmen will
- Update
  - ◆ B aktualisiert seine Arbeitskopie mit dem aktuellen Zustand des Repository
- Problem
  - ◆ Bei Konflikten, wird automatisch ein „merge“ durchgeführt – evtl. unerwünscht
  - ◆ Besser: Zuerst „Synchronize“ benutzen und selbst entscheiden!

Arbeitskopie von  
Entwickler A

Projekt im Repository

Arbeitskopie von  
Entwickler B



# Synchronisation: Versionsvergleich mit Konflikterkennung

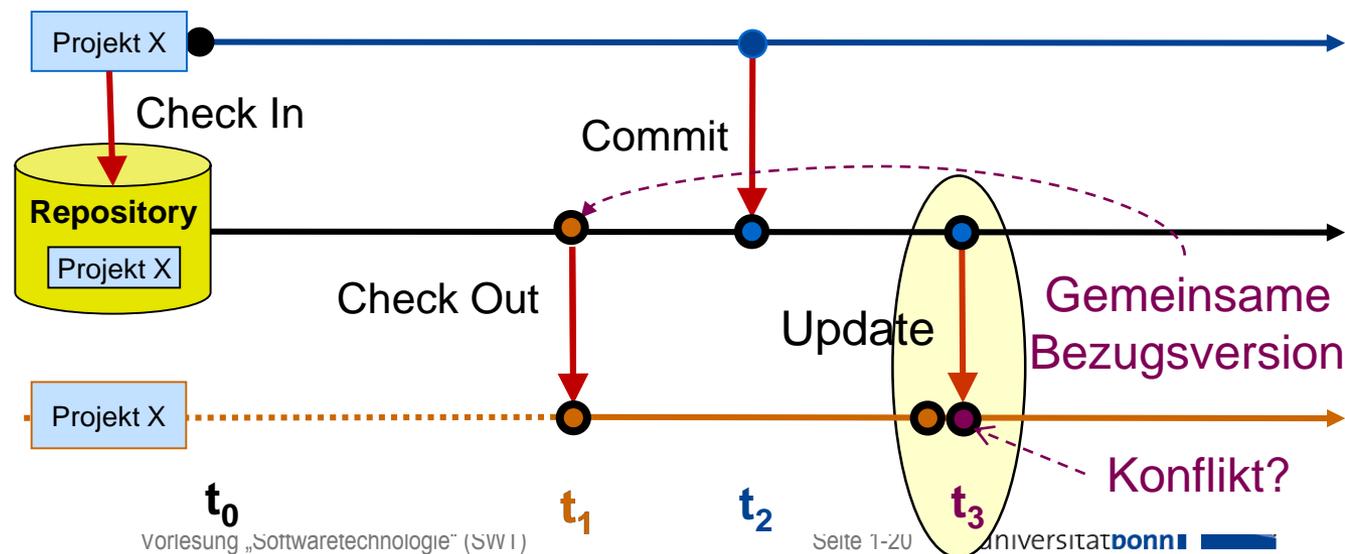
- Vergleich des Projektes in Arbeitskopie mit Repository bezogen auf Stand beim letzten Abgleich (check-in / commit / check-out / update)
  - ◆ Automatisierter Vergleich **aller Dateien** im Projekt
  - ◆ Automatisierter Vergleich **einzelner Dateiinhalte**
  - ◆ Der Entwickler entscheidet selbst was aktuell ist
  - ◆ ... und führt Updates oder Commits explizit durch (eventuell selektiv)



Arbeitskopie von  
Entwickler A

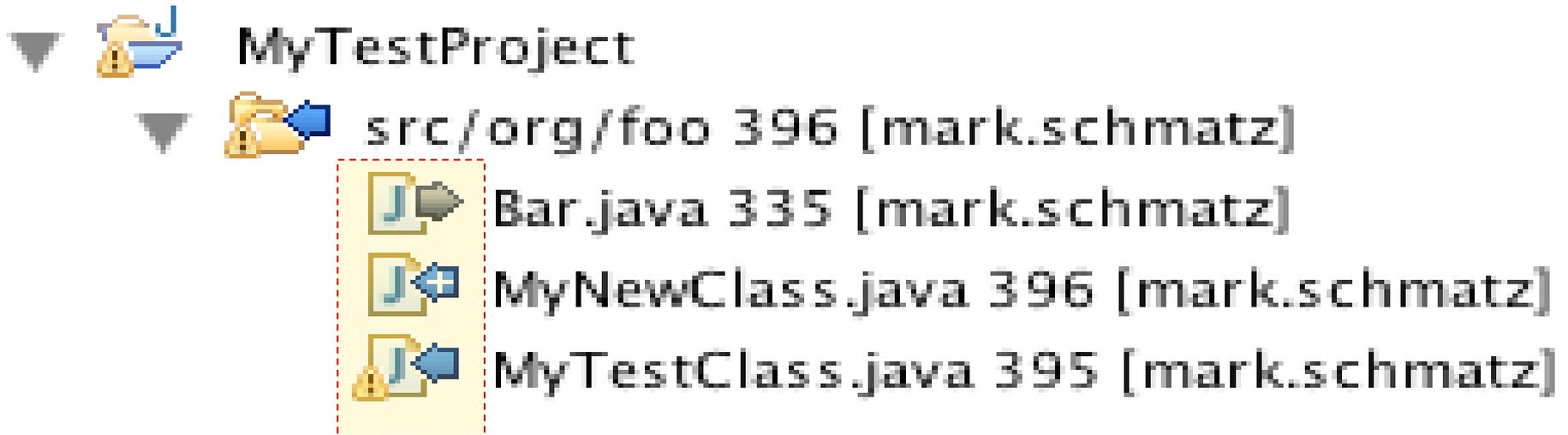
Projekt im Repository

Arbeitskopie von  
Entwickler B



# “Synchronize View” in Eclipse: Automatisierte Gesamtübersicht

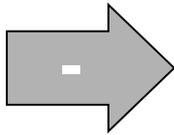
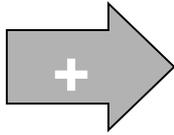
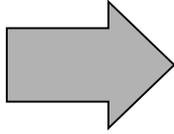
- Vergleich aller Dateien eines Projektes (oder selektierten Ordners)
- Anzeige aller **ein- und ausgehende Änderungen sowie Konflikte** einer jeden Datei



# Symbole im „Synchronize View“

**Ausgehende Änderung**

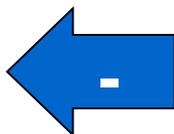
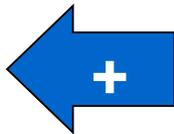
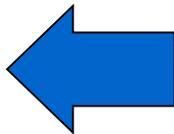
(lokale Änderung)



- Lokale Datei ist neuer als ihre Version im Repository  
→ Überschreibe Version im Repository mit lokaler Version
- Neue lokale Datei existiert nicht im Repository  
→ Füge Datei zum Repository hinzu
- Datei aus Repository wurde lokal gelöscht  
→ Datei aus (nächster Version in) dem Repository löschen

**Eingehende Änderung**

(Änderung im Repository)

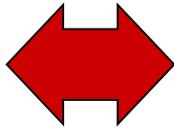


- Datei im Repository ist neuer als ihre lokale Version  
← Überschreibe lokale Kopie mit der aus dem Repository
- Neue Datei aus Repository existiert nicht lokal  
← Füge die Datei der lokalen Arbeitskopie hinzu
- Lokale Datei wurde im Repository gelöscht  
← Lösche die Datei aus der lokalen Arbeitskopie

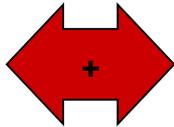
# Symbole im „Synchronize View“ (Fortsetzung)

- Vorherige Folie: Fälle, wenn eine Datei nur auf einer Seite verändert wurde
  - ◆ Änderung nur lokal → Übernahme ins Repository (ausgehende Änderung)
  - ◆ Änderung nur in Repository → Übernahme in lokale Kopie (eingehende Änderung)
- Nun: **Konflikte**, wenn eine Datei auf beiden Seiten verändert wurde gegenüber des Stands beim letzten Abgleich:

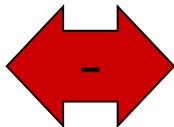
Konflikt



- Datei**inhalt** wurde lokal und im Repository verändert  
↔ Manuelle Konfliktlösung notwendig



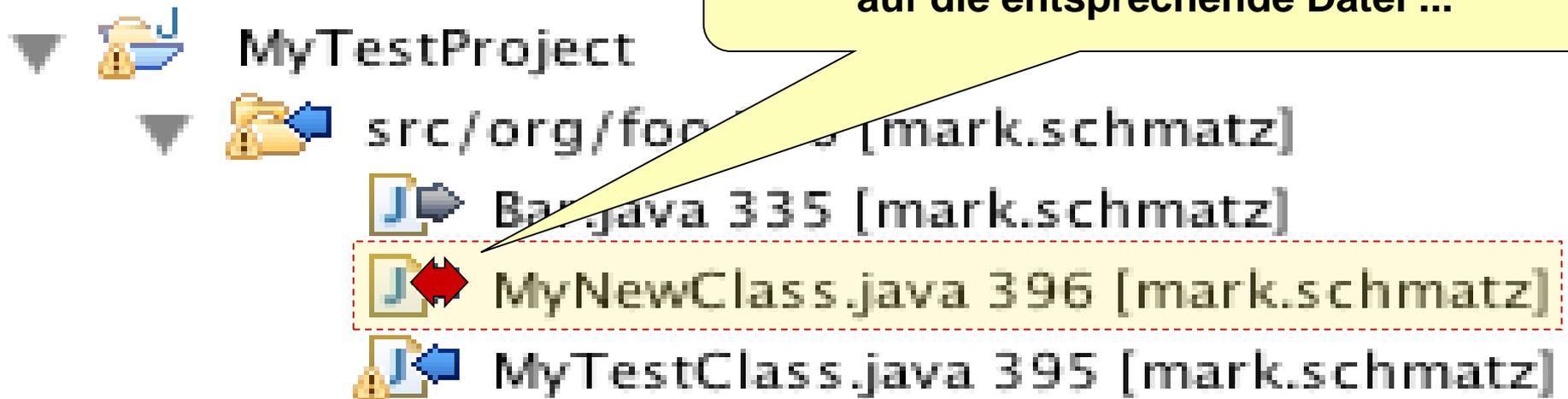
- Lokale veränderte Datei wurde im Repository gelöscht  
↔ Manuelle Konfliktlösung notwendig



- Lokale gelöschte Datei wurde im Repository verändert  
↔ Manuelle Konfliktlösung notwendig

# Synchronisieren: Konfliktauflösung

- Konfliktauflösung erfordert Inhaltsvergleich der Dateien
  - ◆ Angezeigt in “Side-by-side View” / “Compare Editor” von Eclipse
  - ◆ Zeigt Versionsvergleich von Dateien („diff“) übersichtlich an



# Synchronisieren: Inhaltsvergleich im „Compare Editor“ Fenster

- Hier die Anzeige eines Konfliktes

Local File	Remote File (394 [mark.schmatz])
<pre>public void myTestMethod() {     log("Test method entered.");      boolean keepOnRunning = true;     int i=0;      while( keepOnRunning )     {         int r1 = doSomething1();         int r2 = doSomething3();          if( r1+r2 &gt; MAX_THRESHOLD )         {             log("Threshold exceeded.");             log("Iterations: " + i);             keepOnRunning = false;         }     } }</pre>	<pre>public void myTestMethod() {     log("Test method entered.");      boolean keepOnRunning = true;     int i=0;      while( keepOnRunning )     {         int r1 = doSomething1();         int r2 = doSomethingElse();          if( r1+r2 &gt; MAX_THRESHOLD )         {             log("Threshold exceeded.");             log("Iterations: " + i);             keepOnRunning = false;         }     } }</pre>

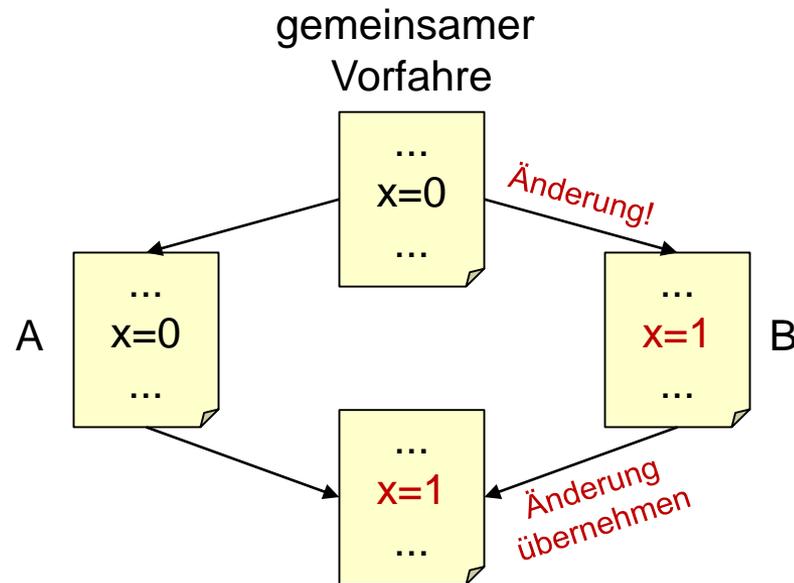
- Bei Bedarf kann auch die gemeinsame Bezugsversion angezeigt werden (“Show Ancestor Pane”).
  - ◆ So kann man selbst entscheiden, welches die relevante Änderung ist

# 3-Wege-Konfliktauflösung

Prinzip:

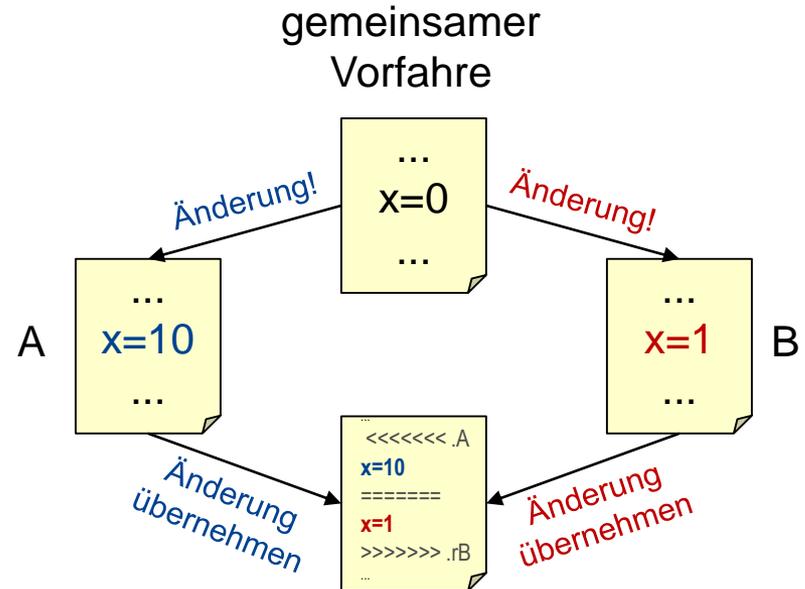
Anhand des gemeinsamen Vorfahren feststellen was sich geändert hat!

## Automatischer „Merge“



abgegliche Version (nach „Merge“)  
(der Benutzer muss nichts mehr tun, die einzige Änderung wurde übernommen)

## „Merge“-Konflikt



Version mit „Merge“-Konflikt  
(der Konflikt wird von SVN in die Datei übernommen, der Benutzer muss sie anschließend editieren)



# Visuelle Anzeige von Unterschieden bei Merge-Konflikt (Beispiel)

```
return CONTENT_URI.buildUpon().
}

@Override
public void createSchema(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE " + TABLE_NAME + " (
        BaseColumns._ID + " INTEGER PRIMARY KEY,
        Columns.ID + " TEXT NOT NULL,
        Columns.PARENT_ID + " TEXT NOT NULL,
        Columns._TYPE + " TEXT NOT NULL,
        Columns.TITLE + " TEXT NOT NULL,
        Columns.LATITUDE + " REAL NOT NULL,
        Columns.LONGITUDE + " REAL NOT NULL,
        Columns.LATITUDE_SPAN + " REAL NOT NULL,
        Columns.LONGITUDE_SPAN + " REAL NOT NULL,
        Columns.ADDRESS_LINES + " TEXT NOT NULL);");
}

public static Uri getLocationUri(Uri uri) {
    return CONTENT_URI.buildUpon().appendPathSegments(uri.getPathSegments()).build();
}

public static String getLocationId(Uri uri) {
    return uri.getPathSegments().get(0);
}

public static Uri getLocationsUri(Uri uri) {
    return CONTENT_URI;
}

@Override
public int hashCode() {
    return TABLE_NAME.hashCode();
}

@Override
public void createSchema(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE " + TABLE_NAME + " (
        BaseColumns._ID + " INTEGER PRIMARY KEY,
        Columns.ID + " TEXT NOT NULL,
        Columns.PARENT_ID + " TEXT NOT NULL,
        Columns._TYPE + " TEXT NOT NULL,
        Columns.TITLE + " TEXT NOT NULL,
        Columns.LATITUDE + " REAL NOT NULL,
        Columns.LONGITUDE + " REAL NOT NULL,
        Columns.LATITUDE_SPAN + " REAL NOT NULL,
        Columns.LONGITUDE_SPAN + " REAL NOT NULL,
        Columns.ADDRESS_LINES + " TEXT NOT NULL);");
}

public static Uri getLocationUri(Uri uri) {
    return CONTENT_URI.buildUpon().appendPathSegments(uri.getPathSegments()).build();
}

public static String getLocationId(Uri uri) {
    return uri.getPathSegments().get(0);
}

public static Uri getLocationsUri(Uri uri) {
    return CONTENT_URI;
}

@Override
public int hashCode() {
    return TABLE_NAME.hashCode();
}

@Override
public String getCreateSql() {
    return "CREATE TABLE " + TABLE_NAME + " (
        BaseColumns._ID + " INTEGER PRIMARY KEY,
        Columns.ID + " TEXT NOT NULL,
        Columns.PARENT_ID + " TEXT NOT NULL,
        Columns._TYPE + " TEXT NOT NULL,
        Columns.TITLE + " TEXT NOT NULL,
        Columns.LATITUDE + " REAL NOT NULL,
        Columns.LONGITUDE + " REAL NOT NULL,
        Columns.LATITUDE_SPAN + " REAL NOT NULL,
        Columns.LONGITUDE_SPAN + " REAL NOT NULL,
        Columns.ADDRESS_LINES + " TEXT NOT NULL);";
}

public static Uri getLocationUri(Uri uri) {
    return CONTENT_URI.buildUpon().appendPathSegments(uri.getPathSegments()).build();
}

public static String getLocationId(Uri uri) {
    return uri.getPathSegments().get(0);
}

public static Uri getLocationsUri(Uri uri) {
    return CONTENT_URI;
}

@Override
public int hashCode() {
    return TABLE_NAME.hashCode();
}
```

INS : Ln 142, Col 1

# Grenzen der Änderungserkennung und darauf basierender Konfliktauflösung

- Textuelles Vorgehen

- ◆ **Kommentare** werden nicht von Code unterschieden
- ◆ **Formatierungsänderungen** werden angezeigt
- ◆ **Bruchstücke** von Namen werden als unverändert angezeigt

- Zeilenweises Vorgehen

- ◆ **Haupt- und Folgeänderungen** werden nicht unterschieden (z.B. bei Umbenennung)
- ◆ **Verschiebungen** werden nicht erkannt (als **Löschung** und **Hinzufügung** angezeigt)

<pre>1 public class Person { 2   &gt; 3   private String vorname; 4   private String nachname; 5   private int alter; 6   &gt; 7   public String getName() { 8     return this.vorname + " " + this.nachname; 9   } 10  &gt; 11  public void setVorname(String vorname) { 12    &gt; this.vorname = vorname; 13  } 14  &gt; 15  public void setNachname(String nachname) { 16    &gt; this.nachname = nachname; 17  } 18  &gt; 19  public void increaseAlter() { 20    alter++; 21  } 22 }</pre>		<pre>1 public class Person { 2   &gt; 3   private String firstName; 4   private String lastName; 5   private int age; 6   &gt; 7   public String getName() { 8     return this.firstName + " " + this.lastName; 9   } 10  &gt; 11  public void increaseAlter() { 12    age++; 13  } 14  &gt; 15  public void setFirstName(String firstName) { 16    this.firstName = firstName; 17  } 18  &gt; 19  public void setLastName(String lastName) { 20    this.lastName = lastName; 21  } 22 }</pre>
--	--	--

**Beachte: Das Verhalten des obigen Codes hat sich nicht geändert (nur Umbenennungen und Verschiebungen)!**

# Vergleich von Subversion (SVN) und CVS

Versionierung von Verzeichnissen

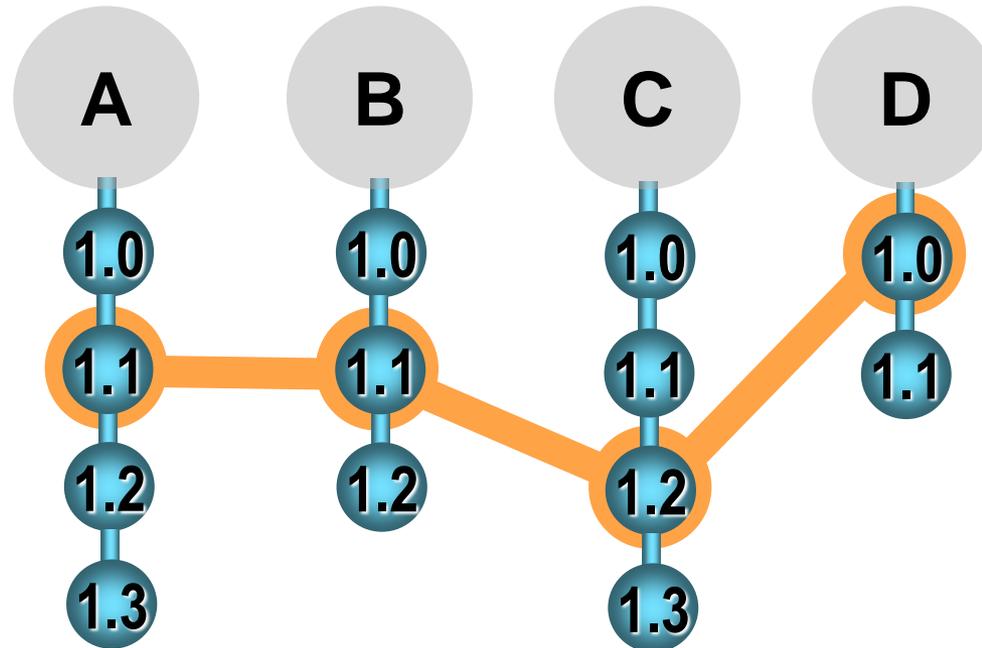
Globale Commit-Nummerierung

Atomare Commit-Aktionen

# Subversion kann mehr als CVS:

## 1. Globale Versions-Nummerierung

- CVS weist jeder Datei ihre **eigenen** Versions-Nummern zu

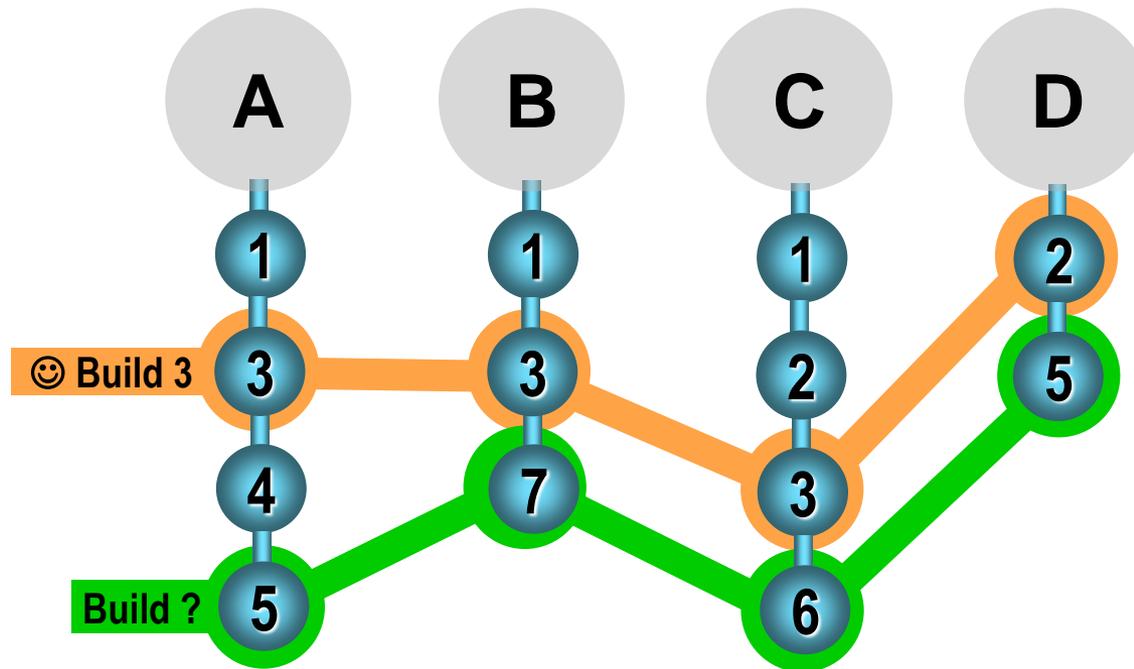


- Problem: Es ist nicht möglich zu sehen Dateiversionen zusammengehören
  - ◆ „Build 3 = A1.1 + B1.1 + C1.2 + D1.0“ selbst merken ☹

# Subversion kann mehr als CVS:

## 1. Globale Versions-Nummerierung

- SVN weist jeder Datei die bei einem Commit verändert wurde die selbe Versions-Nummer zu.

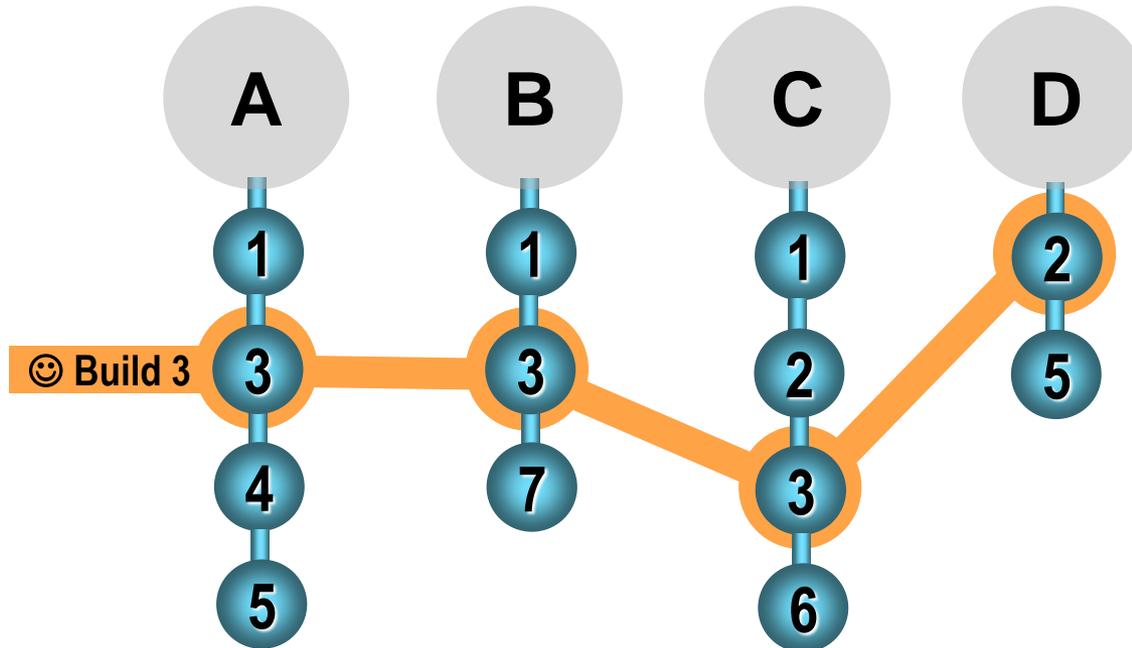


- Vorteil: Dateiversionen die zusammengehören sind auf einen Blick zu erkennen: Zu einer Projektversion V gehören alle Elemente mit Versionsnummer V oder kleiner als V

# Subversion kann mehr als CVS:

## 1. Globale Versions-Nummerierung

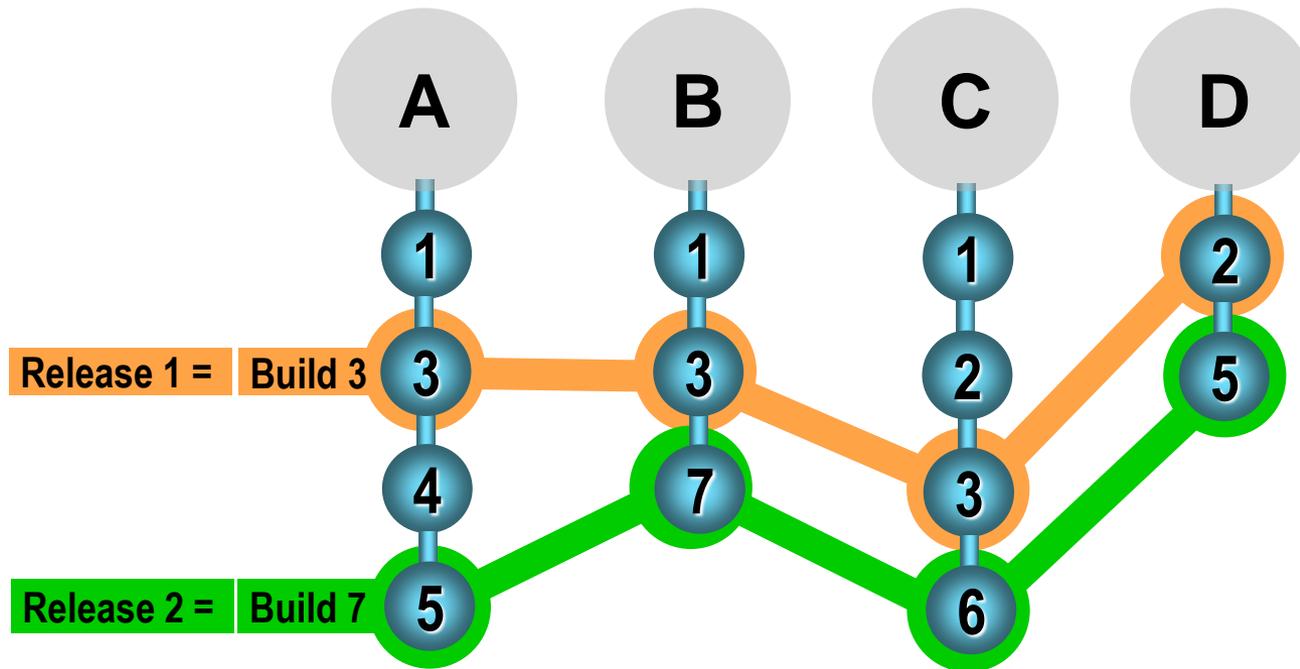
- SVN: „Roll back“ auf eine alte Version ist einfach
  - ◆ „Gehe zurück zu Build 3“



- CVS:
  - ◆ Zuordnungen Dateiversion  $\leftrightarrow$  Build selbst verwalten 😞
  - ◆ Jede Datei einzeln zurücksetzen 😞

# Randbemerkung: Tagging (= Baselineing)

- Tag = Name den man jedem Build zusätzlich geben kann.



- Vorteil:
  - ◆ Semantische Information explizit machen → „Release 1.0 alpha“
  - ◆ Leichter zu merken → „1.0 alpha“ versus „1293“

# Subversion kann mehr als CVS:

## 2. Atomare commits

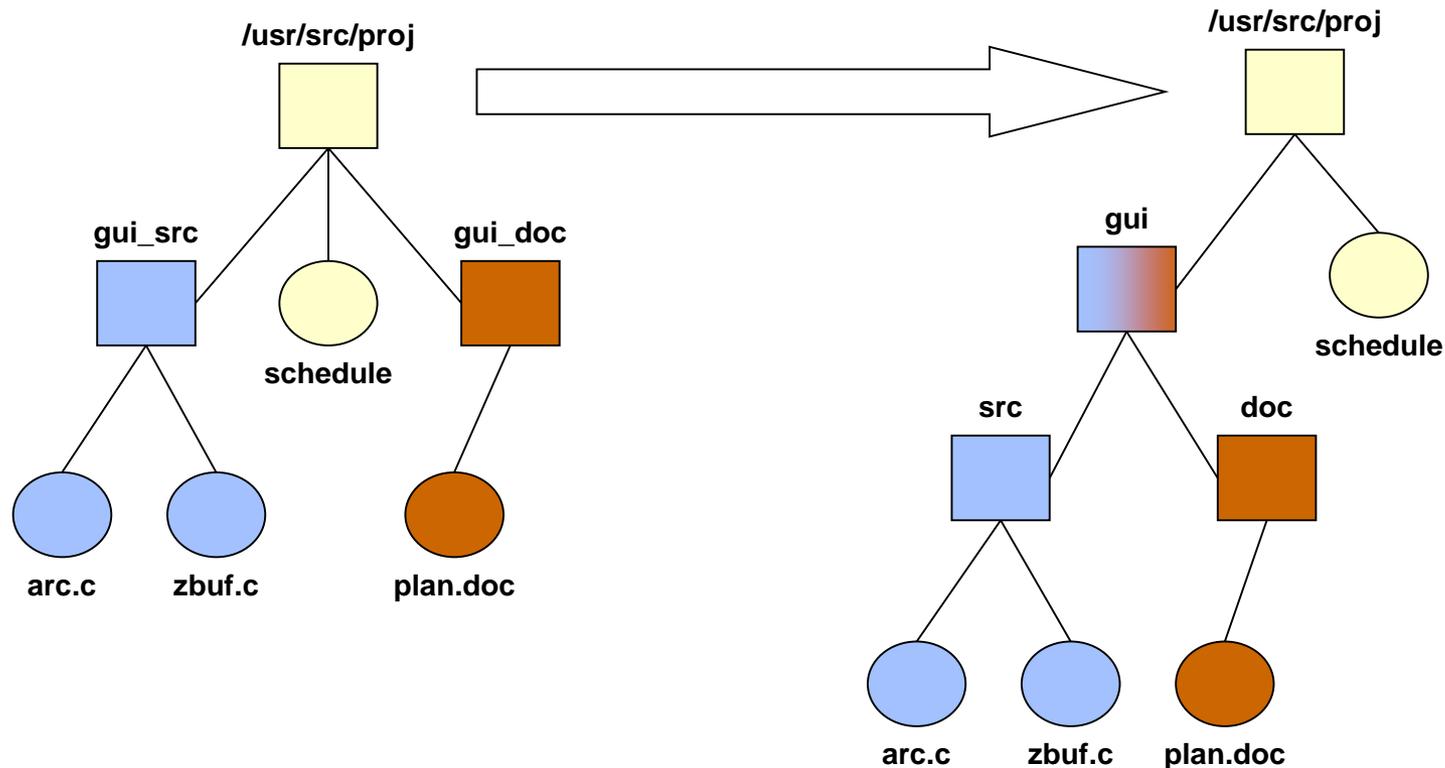
---

- Netzwerkfehler oder andere Probleme können zu unvollständigen Commits führen (nicht alle Dateien wurden „commitet“)
- Wenn dies passiert, hinterlässt CVS das Repository in einem inkonsistenten Zustand.
- SVN führt bei unvollständigen Commits einen „Roll back“ durch.
  - ◆ SVN benutzt ein Datenbanksystem um das Repository zu speichern und kann daher Commits als atomare Transaktionen implementieren!

# Subversion kann mehr als CVS:

## 3. Versionierung von Verzeichnissen

- Dateien und Verzeichnisse zu löschen, umzubenennen und zu verschieben entspricht einer neuen Version des umgebenden Verzeichnisses
- Hier eine Versionsänderung von /usr/src/proj



# Subversion kann mehr als CVS:

## 3. Versionierung von Verzeichnissen

---

- Es ist in Subversion möglich Dateien und Verzeichnisse zu **löschen**, **umzubenennen** und zu **verschieben**

### Warnung 1: Umbenennung und Verschieben

- Man muss diese Operationen mit einem „Subversion Client“ durchführen! → Siehe Folie „Wichtige Links“
- Führt man es selbst auf Systemebene durch wird Subversion nichts vom Umbenennen oder Verschieben erfahren. Es wird annehmen eine Datei wurde gelöscht und eine andere hinzugefügt!

### Warnung 2: .svn-Ordner

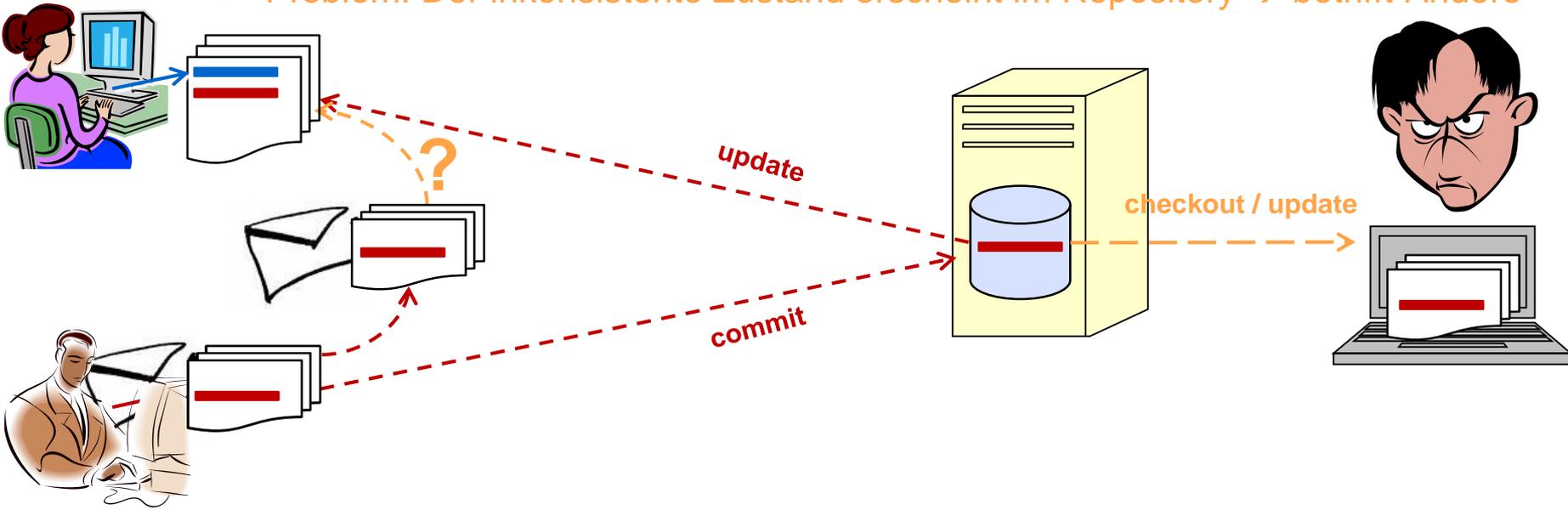
- „.svn“ Ordner innerhalb der Arbeitskopie nicht verändern oder löschen! Sie enthalten die Meta-Informationen für SVN über die stattgefundenen Änderungen

# Von zentraler zu verteilter Versionskontrolle

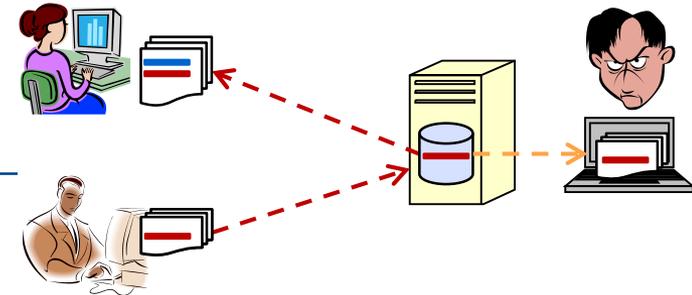
- ▶ **Motivation:** Nachteile zentralisierter Ansätze
- ▶ **Prinzip:** Was bedeutet verteilte Versionskontrolle?
- ▶ **Lokal arbeiten:** add, commit, diff, checkout, merge, reset
- ▶ **Verteilt arbeiten:** clone, fetch, pull, push
- ▶ **Gesamtüberblick:** Wie alles zusammenspielt

# Austausch halbfertiger Zwischenstände via zentralem SCM-System

- Peter und Lisa haben unabhängig am gleichen Projekt gearbeitet
- Peter will Lisa seine noch nicht ganz ausgegorenen Änderungen geben, damit sie sie integriert
  - ◆ Ohne SCM: Via e-mail, etc.
    - ⇒ Problem: Keine Unterstützung für Abgleich (Synchronize & Merge)
  - ◆ Mit SCM: Via commit und update
    - ⇒ Problem: Der inkonsistente Zustand erscheint im Repository → betrifft Andere



# Probleme



- Falls Commit von inkonsistenten Zuständen
  - ◆ inkonsistenter Zustand betrifft Andere ☹️
  - ◆ Regel: Kein commit von inkonsistenten Zustände! Niemals!!!
- Commit inkonsistenter Zustände verhindern durch spezielle „committer“ Rolle
  - ◆ Nur sehr erfahrene Entwickler haben das Recht zum „Commit“
  - ◆ Sie erhalten jeden Änderungsvorschlag als „patch“ den sie erst auf Ihrer eigenen Arbeitskopie testen und gegebenenfalls ablehnen
  - ◆ Folgeproblem: Engpass
- Falls kein commit inkonsistenter Zustände
  - ◆ Synchronize & Merge nicht für Abgleich von Zwischenzuständen nutzbar
    - ⇒ Fehleranfälliger manueller Abgleich von Zwischenzuständen (Peter → Lisa)
  - ◆ Commit & Revert nicht für Zwischenzustände nutzbar
    - ⇒ Es sammeln sich umfangreiche Änderungen an
    - ⇒ Rücksetzen auf Repository-Stand entsprechend verlustreich und aufwendig

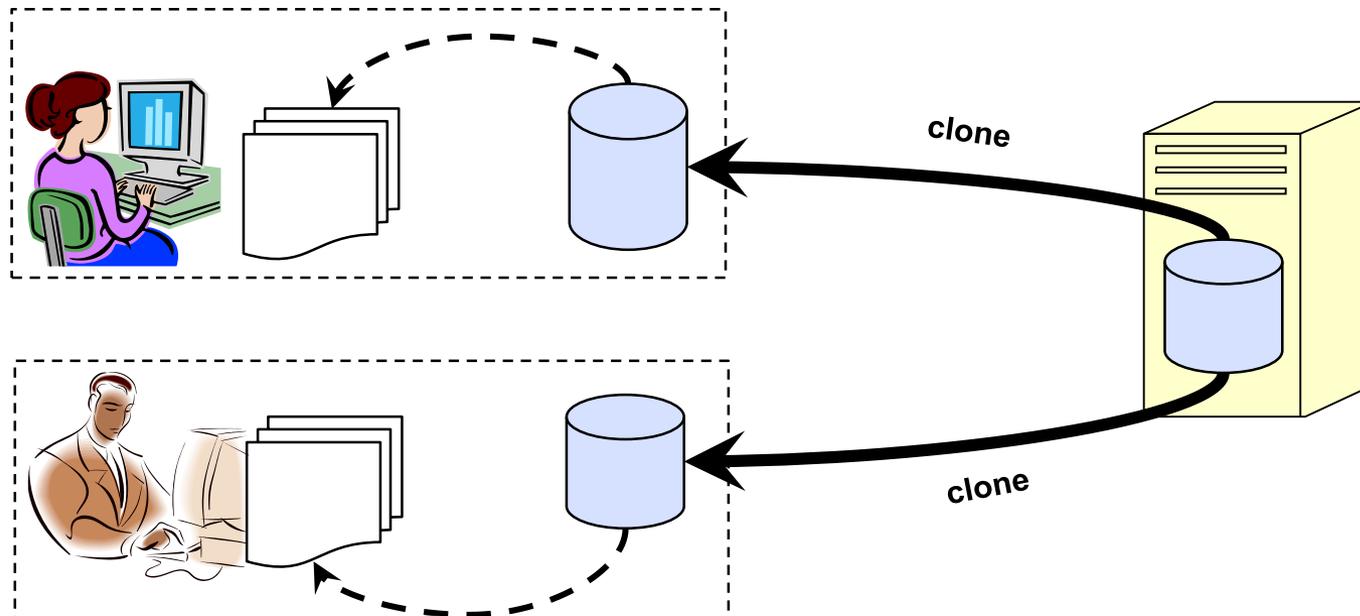
# Grundlagen des verteilten Arbeitens

- ▶ Operationen
- ▶ Remotes
- ▶ Zugriffskontrolle
- ▶ Organisatorische Zentralisierung

# Verteilte Versionskontrolle ►

## Multiple, verteilte Repositories

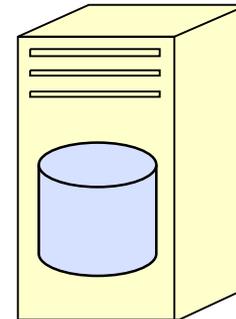
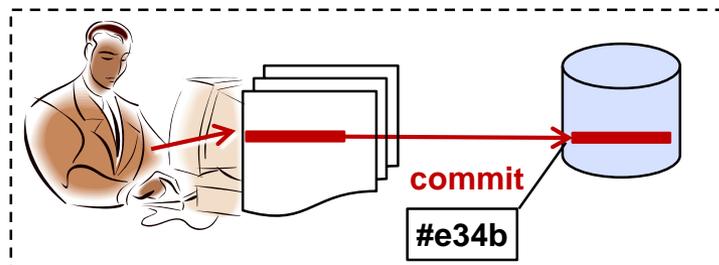
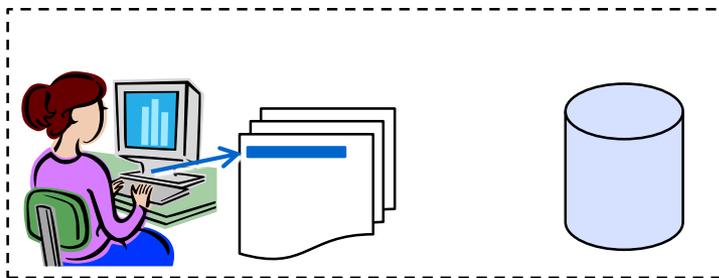
- Jede(r) hat ein oder mehrere lokale Repositories
  - ◆ Mehrere → Verschieden Projekte oder verschiedene Aufgaben im Projekt
- Repository kann geklont werden → **inkl. der gesamten Versionshistorie!**
  - ◆ Ab jetzt kann man lokal weiterarbeiten
  - ◆ Mit allen Funktionalitäten (commit, branch, switch brach, merge, ...)
  - ◆ Man kann diese Funktionen lokal auch für „Unfertiges“ verwenden!



# Verteilte Versionskontrolle ▶

## Multiple, verteilte Repositories

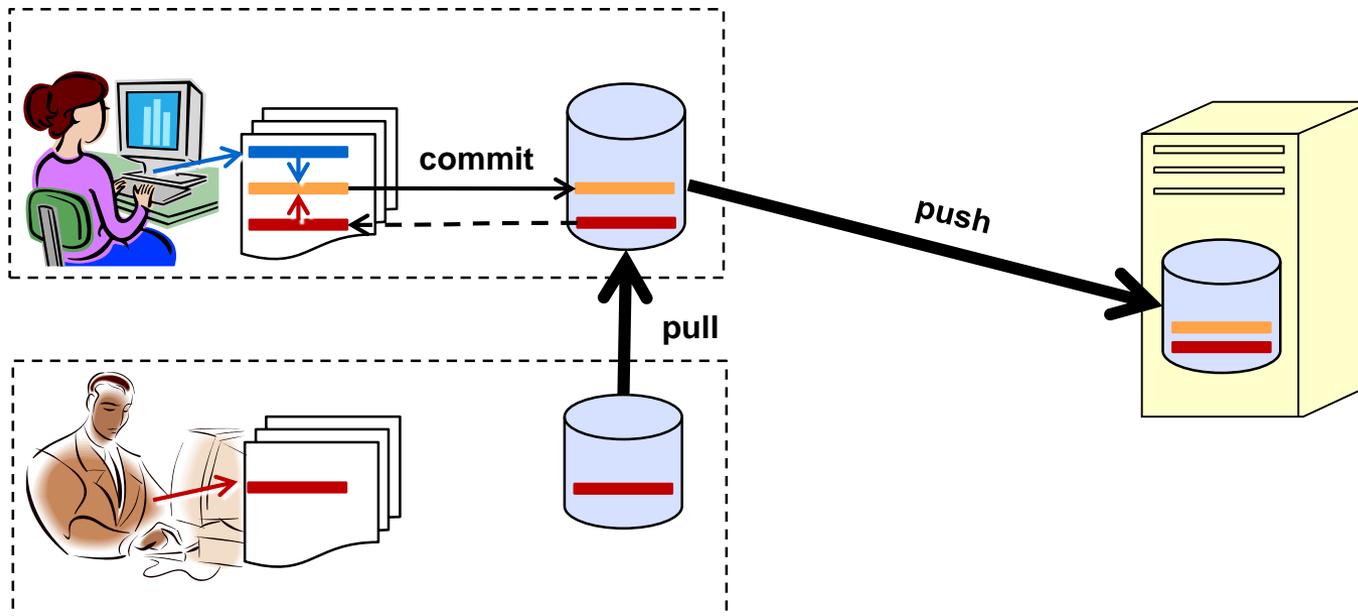
- Checkout, Commit, Branch, Switch Branch, ... geschehen lokal
  - ◆ Sehr schnell
  - ◆ Unabhängig von Netzverbindung und Server
- Fokus auf Gesamt-Repository statt einzelnen Artefakten
  - ◆ Commit = Menge von Änderungen („Change set“ / „Patch“)
  - ◆ Jedes Commit hat eine eindeutige ID
  - ◆ Repository-Zustand = ID einer Änderung = Zustand nach dieser Änderung



# Verteilte Versionskontrolle ►

## Multiple, verteilte Repositories

- Repositories können miteinander abgeglichen werden
  - ◆ Pull: Holen von Änderungen aus anderem Repository
    - ⇒ beinhaltet bei Bedarf automatisches Merge
    - ⇒ Ergebnis des Merge wird anschließend evtl. manuell weiter bearbeitet und schließlich explizit committed
  - ◆ Push: Übertragen von Änderungen in ein anderes Repository



# Verteilte Versionskontrolle ▶

## Multiple, verteilte Repositories

- Technische Gleichberechtigung

- ◆ Jedes Repository kann geklont werden

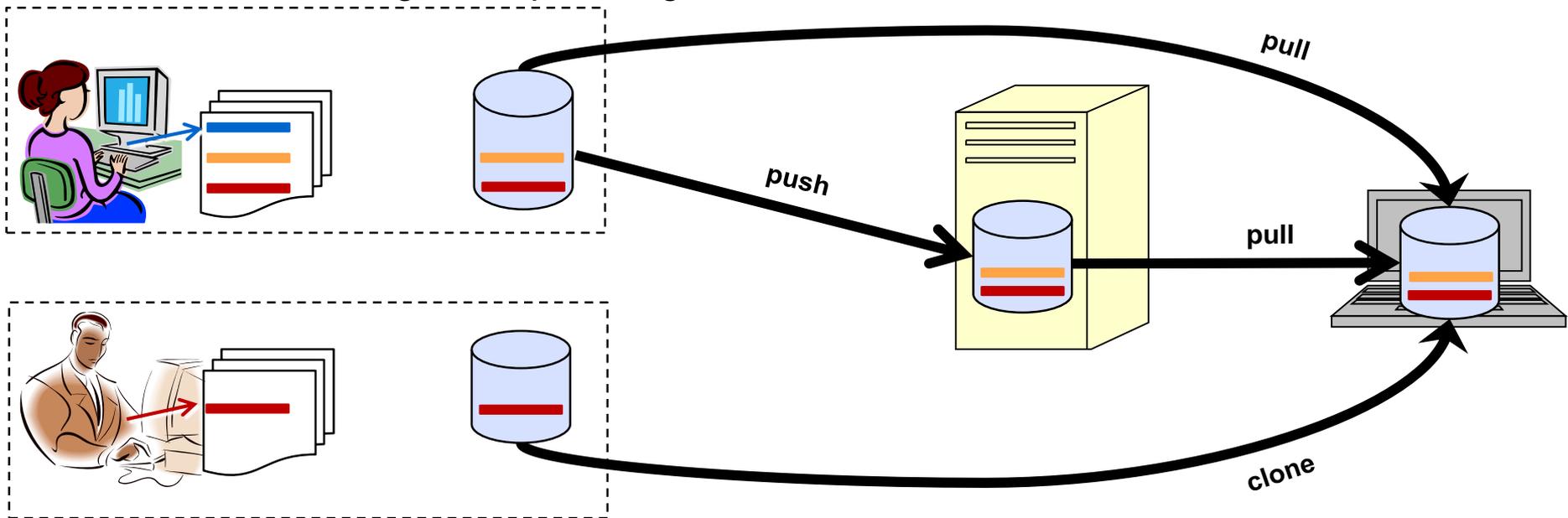
- ◆ Zu jedem kann „push“ und von jedem kann „pull“ durchgeführt werden

- ◆ Voraussetzungen

- ⇒ „remote“ setzen = anderes repository dem eigenen bekannt machen

- ⇒ „fetch“ = Entfernte „branches“ holen / bekannt machen

- ⇒ „tracking“ = Entsprechungen von lokalen und entfernten „branches“ definieren



# Verteilungsspezifische Git Terminologie

---

- **remote** = anderes bekanntes repository
  - ◆ Spezifiziert durch URL oder Pfad im lokalen Dateisystem
- **origin** = Repository von dem dieses geklont worden ist
  - ◆ ausgezeichnete Rolle unter den „remotes“
- **tracking** = Beziehung zwischen lokalen und entfernten „branches“
  - ◆ Push / pull bezieht sich immer auf die branches zu denen ein tracking definiert ist
  - ◆ Push / pull kann für **einen** lokalen branch und seinen tracking branch in einem remote durchgeführt werden
  - ◆ ... oder für **alle** lokalen branches und ihre jeweiligen tracking branches in dem aktuell ausgewählten remote.

# Remotes im vorherigen Beispiel

1. Nach dem clonen haben die lokale Repos von Lisa und Peter den Server als einzigen remote mit Standardnamen „**origin**“

```
git clone ... <repo> <dir>
```

← Mit der Option `--origin <name>` kann man den Namen des geklonten Repos explizit auf etwas anderes als “origin” setzen.

2. Um das „pull“ von Peter durchzuführen muss Lisa erst sein Repo als zusätzlichen remote eintragen

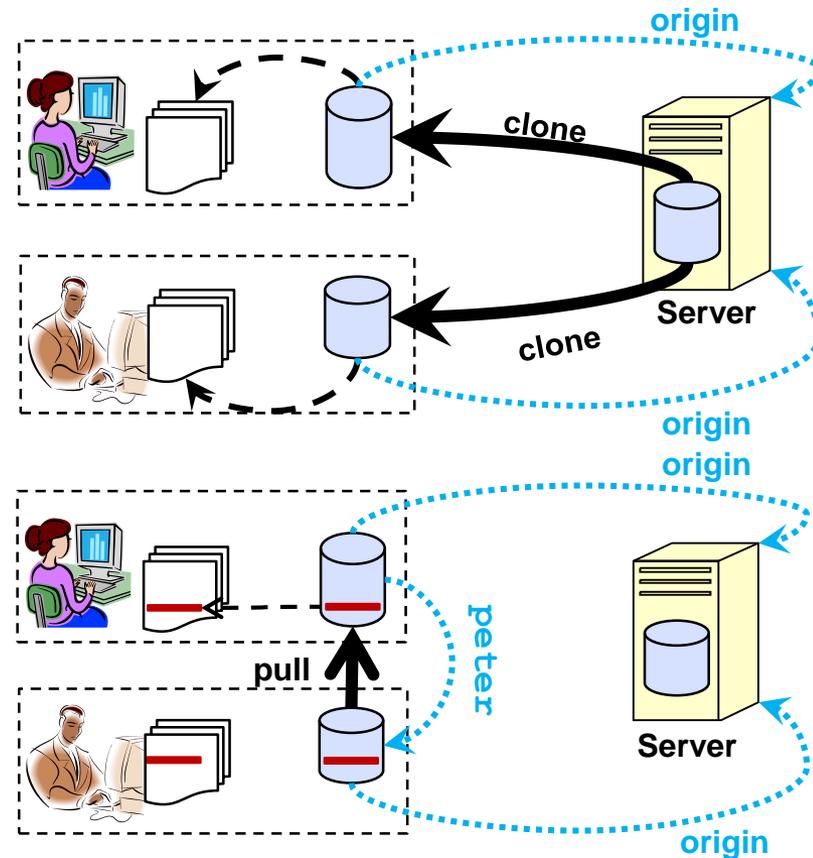
```
git remote add ... <name> <url>
```

```
git remote add ... peter <url>
```

3. ... und den Namen des neuen remotes beim „pull“ als Quelle angeben

```
git pull ... <name>
```

```
git pull ... peter
```



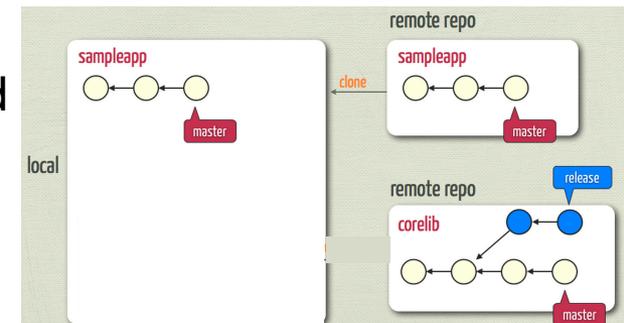
Details siehe <https://git-scm.com/docs/git-remote>,  
[.../git-pull](#) und [.../git-fetch](#)



# Zentralisierung trotz Dezentralisierung

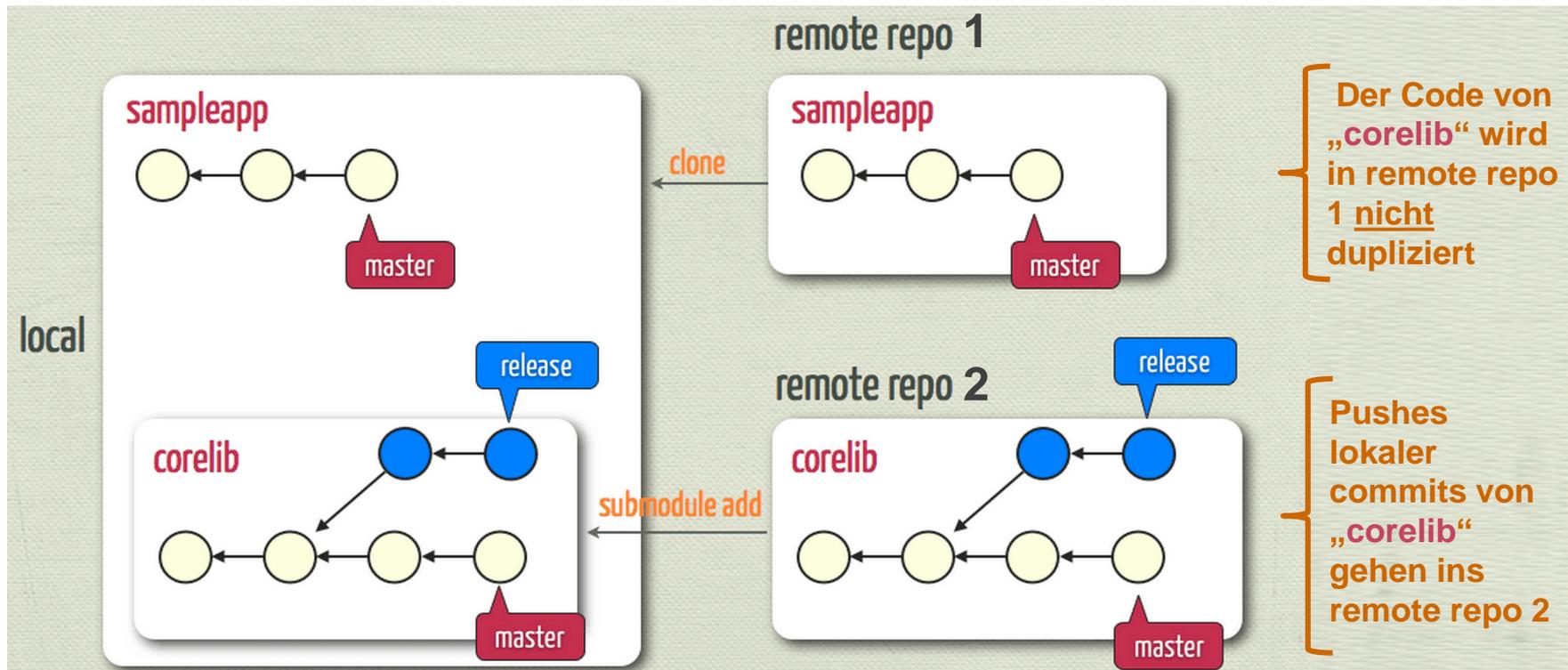
## Zentrales Referenz-Repository ist organisatorisch möglich

- Team vereinbart, welcher „remote“ die „offizielle“ Bezugsversion ist
  - ◆ So muss man nicht alle existierenden Repositories kennen, um dennoch an den aktuellsten Stand zu kommen
- Beliebig skalierbar
  - ◆ Jedes Team hat seinen eigenen Bezugsserver
    - ⇒ Z.B. für Linux Filesystem, Prozessverwaltung, ...
  - ◆ Integration kann mehrstufig erfolgen bis hin zum Server der alles enthält
- Submodul-Konzept hilfreich
  - ◆ Eigenständiges Git-Repository, das lokal als Teil eines übergeordneten Repositories verwaltet wird
  - ◆ So können in einem Repository Teil-Repositories für verschiedene Subsysteme oder externe Libraries existieren
  - ◆ Grundidee → nächste Folie,  
Details → <https://git-scm.com/book/en/v2/Git-Tools-Submodules>



# Submodule ► einheitliche lokale Gesamtsicht auf dezentrale Repositories

- Submodul = Eigenständiges Git-Repository, das lokal als Teil eines übergeordneten Repositories verwaltet wird
- Beispiel ► „corelib“ duplikationsfrei ins eigene Projekt „sampleapp“ integriert:



# Zugriffskontrolle

Wie kontrolliert Peter wer auf sein Repository zugreifen darf?

- GIT hat keine eigene Zugriffskontrolle
- Es wird stattdessen der Mechanismus genutzt, der gerade verfügbar ist:

Zugriff auf	über	Zugriffskontrolle
Lokales Repository	Betriebssystem	Zugriffskontrolle des BS
Remote Repository	FTP	Username + Passwort
	HTTPS	Username + Passwort
	SSH	Private Key (lokal) + Public Key (remote)
	GIT	Read only

# Grundlagen des lokalen Arbeitens

- ▶ **Speicherbereiche:** Arbeitsbereich, Stage / Index, Repository
- ▶ **Der Weg ins Repository:** add, commit
- ▶ **Der Weg aus dem Repository:** checkout, reset

# Ehre wem Ehre gebührt

## Danksagung



Der Inhalt der folgenden Folien ist zum Teil angelehnt an

<https://marklodato.github.io/visual-git-guide/index-de.html>

Unser Dank gilt Mark Lodato für die ursprüngliche Website und insbesondere all seine tollen Grafiken und an Martin Funk für die deutsche Übersetzung

## Acknowledgement

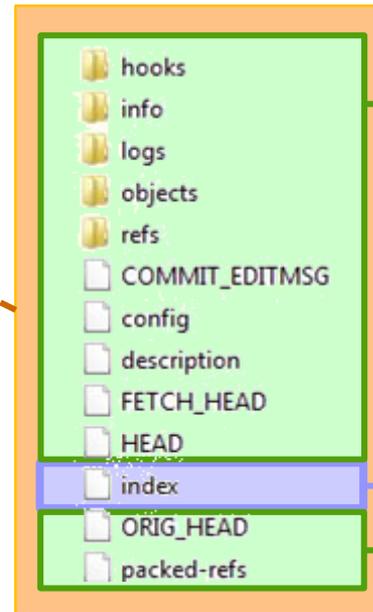
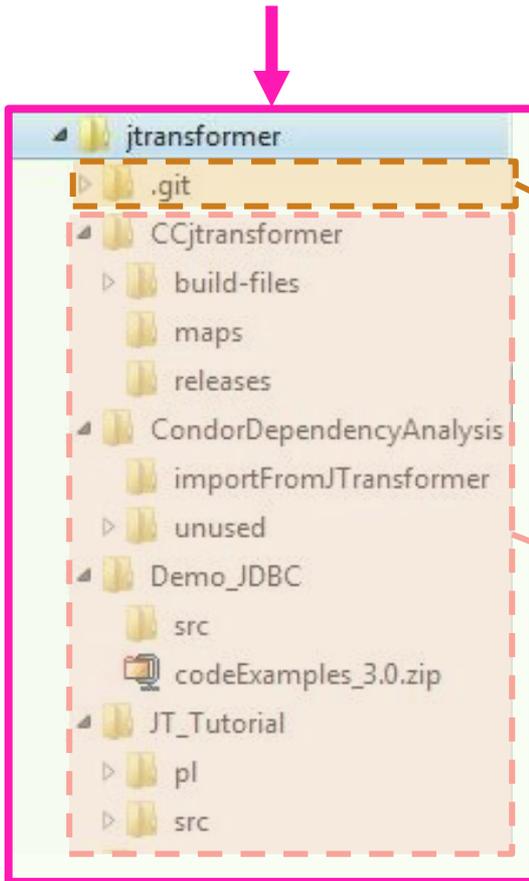


The following slides are inspired by

Our thanks go to Mark Lodato for the original website and especially for all his great graphics and to Martin Funk for the German translation

# Der GIT-Ordner und der .git Ordner

**GIT-Ordner** = Ordner der das GIT-Repository und die Arbeitskopie enthält:



**Konzeptuelle lokale Arbeitsbereiche** für git-Operationen:

History  
(repository)

Index  
(stage)

Sandbox  
(working directory)



# Lokale Arbeitsbereiche

**History (repository)** = Kopien aller unter Versionskontrolle stehenden Dateien in all ihren Versionen und **komplette Historie der durchgeführten Veränderungen**, Branches, Commits, Tags, ...

**Index (Stage)** = Kopien aller seit letztem „commit“ durchgeführten und **zur Übernahme ins Repository „vorgemerkten“ Änderungen**.  
Das nächste „commit“ wird sie ins Repo übertragen und die Stage leeren.

**Sandbox (Working directory)** = Dem Entwickler aktuell zugängliche Dateien. Manche sind evtl. noch nicht unter Versionskontrolle. Veränderungen derer die versioniert sind können im Index gespeichert sein oder auch nicht (= sind noch nicht zu committen).

**Konzeptuelle lokale Arbeitsbereiche**  
für git-Operationen

History  
(repository)

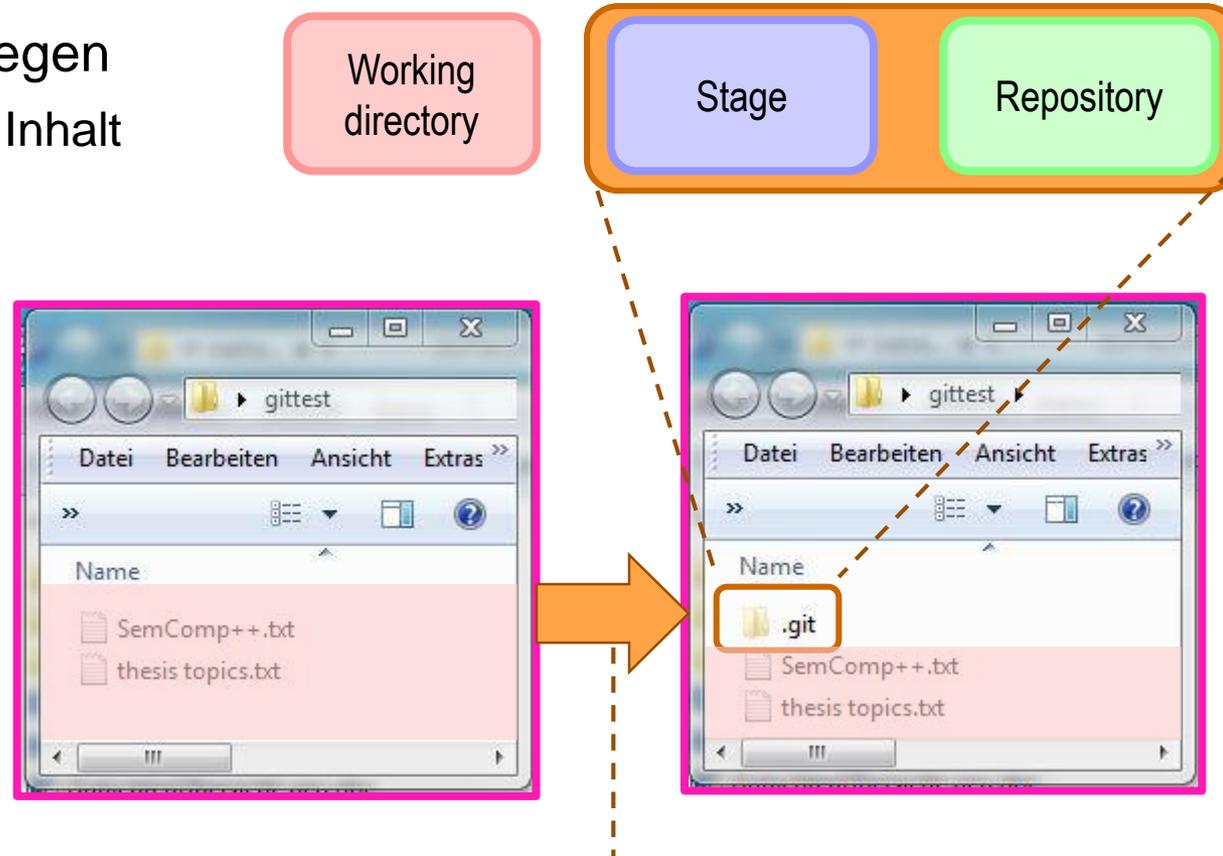
Index  
(stage)

Sandbox  
(working directory)

# Lokale Operationen ► Initialisierung

## git init

- Leeres Repository anlegen
  - ◆ .git-Ordner mit allem Inhalt

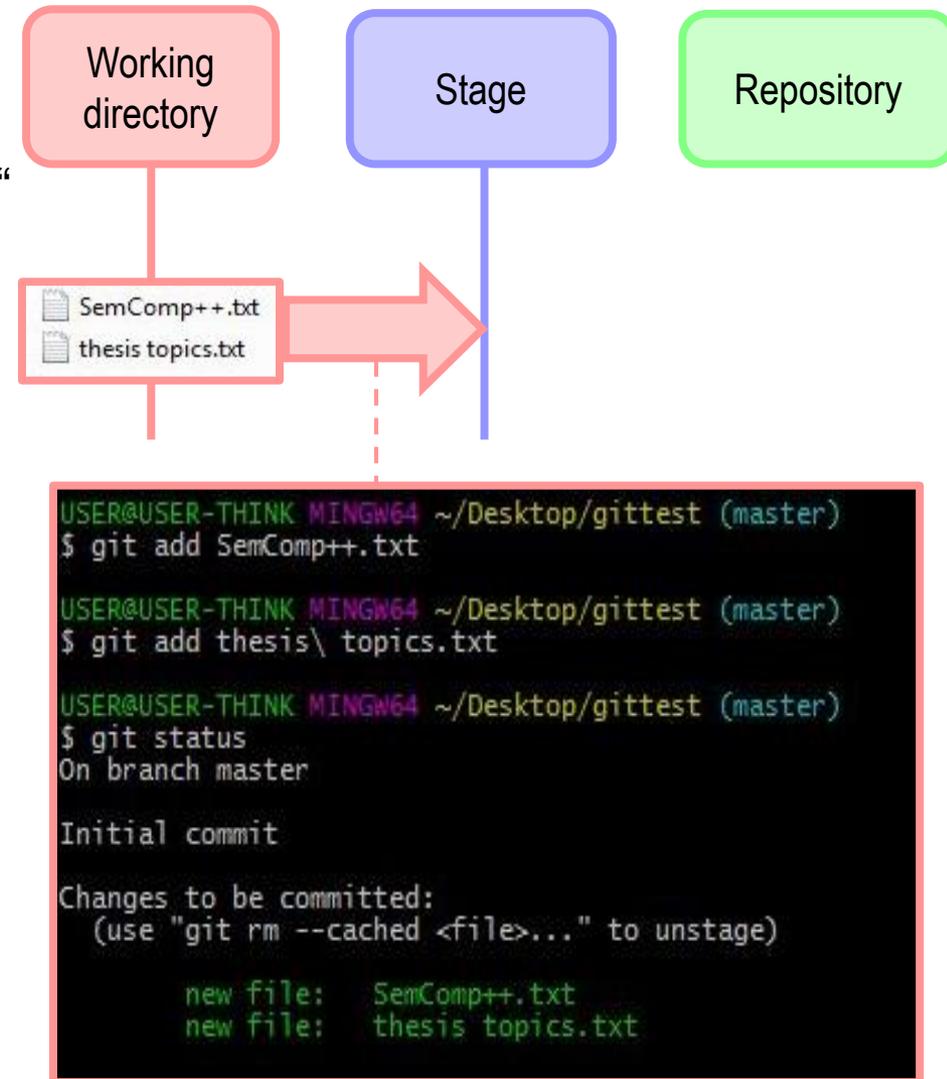


```
USER@USER-THINK MINGW64 ~/Desktop/gittest
$ git init
Initialized empty Git repository in C:/Users/USER/Desktop/gittest/.git/
```

# Lokale Operationen ► Add („Staging“)

## git add files

- Änderungen zusammenzutragen, die logisch zusammengehören
  - ◆ Absicht: „Feingranulare Commits“  
→ jeder Commit soll nur eine *konzeptuelle* Änderung enthalten
  - ◆ Nutzen: Gezieltes, selektives Rücksetzen oder Übernehmen von konzeptuellen Änderungen
- Dazu werden in der „**Stage**“ (= „**Index**“) ausgewählte geänderte Dateizustände zwischengespeichert
- Es können auch nur Teile einer geänderten Datei gespeichert werden → „hunks“
  - ◆ „hunk“ = zusammenhängender Zeilenbereich

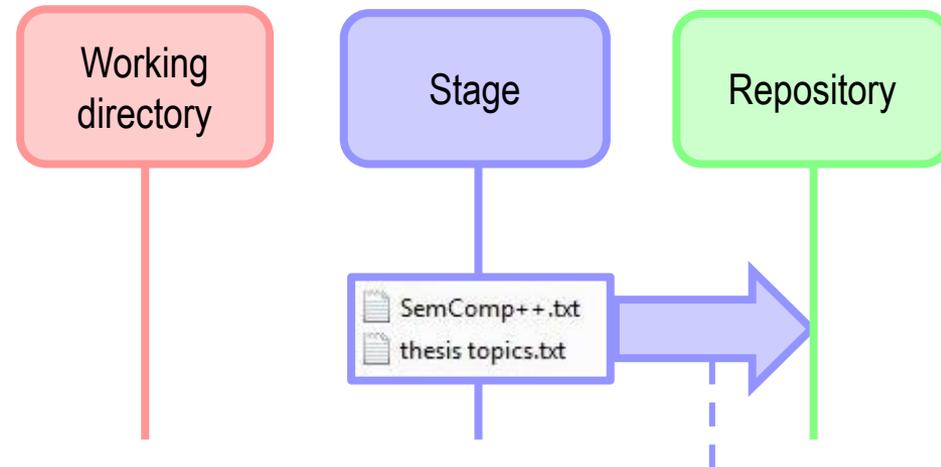


# Lokale Operationen ► Commit

## git commit

### git commit

- Gesamter Stage-Inhalt wird ins Repository übernommen



```
USER@USER-THINK MINGW64 ~/Desktop/gittest (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   SemComp++.txt
   new file:   thesis topics.txt

USER@USER-THINK MINGW64 ~/Desktop/gittest (master)
$ git commit
[master (root-commit) 7f2868f] 2 Dateien hinzugefügt
2 files changed, 144 insertions(+)
 create mode 100644 SemComp++.txt
 create mode 100644 thesis topics.txt

USER@USER-THINK MINGW64 ~/Desktop/gittest (master)
$
```

# Lokale Operationen ► Commit

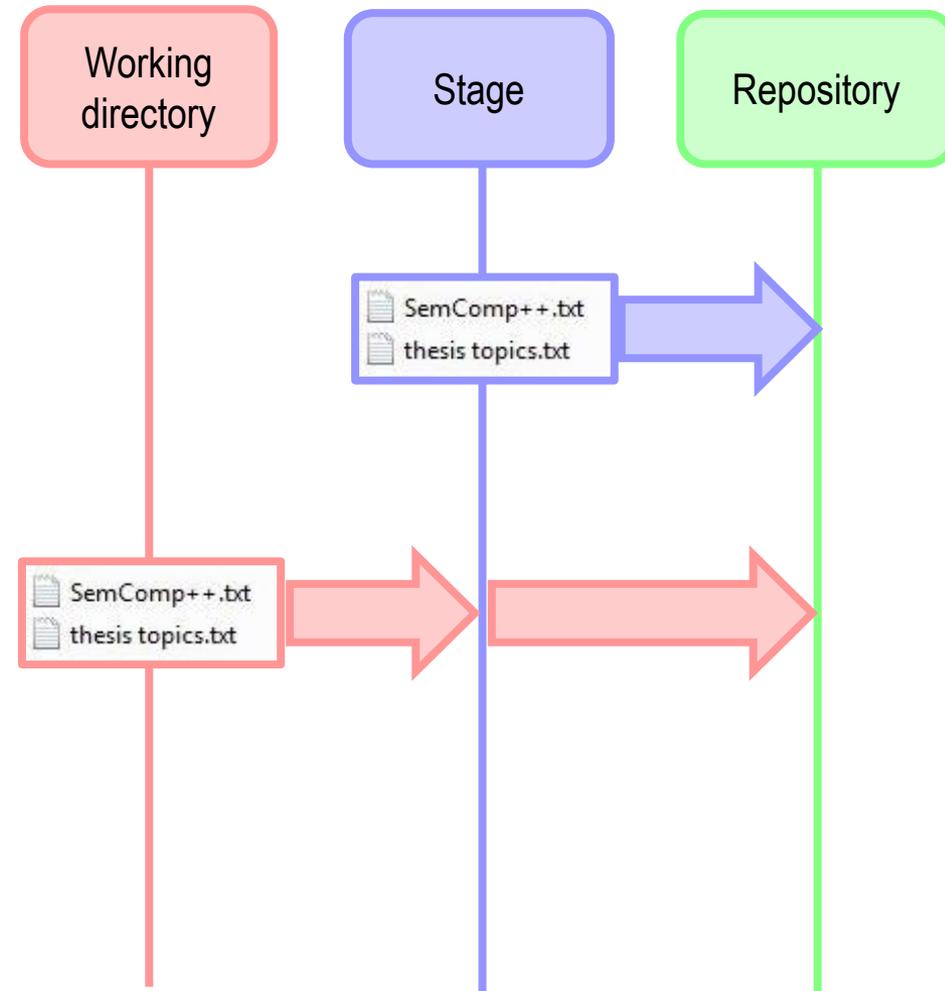
## git commit

### git commit

- Gesamter Stage-Inhalt wird ins Repository übernommen

### git commit files

- **files** werden aus working directory in die Stage und sofort anschließend ins Repository übernommen



# Lokale Operationen ▶ Checkout

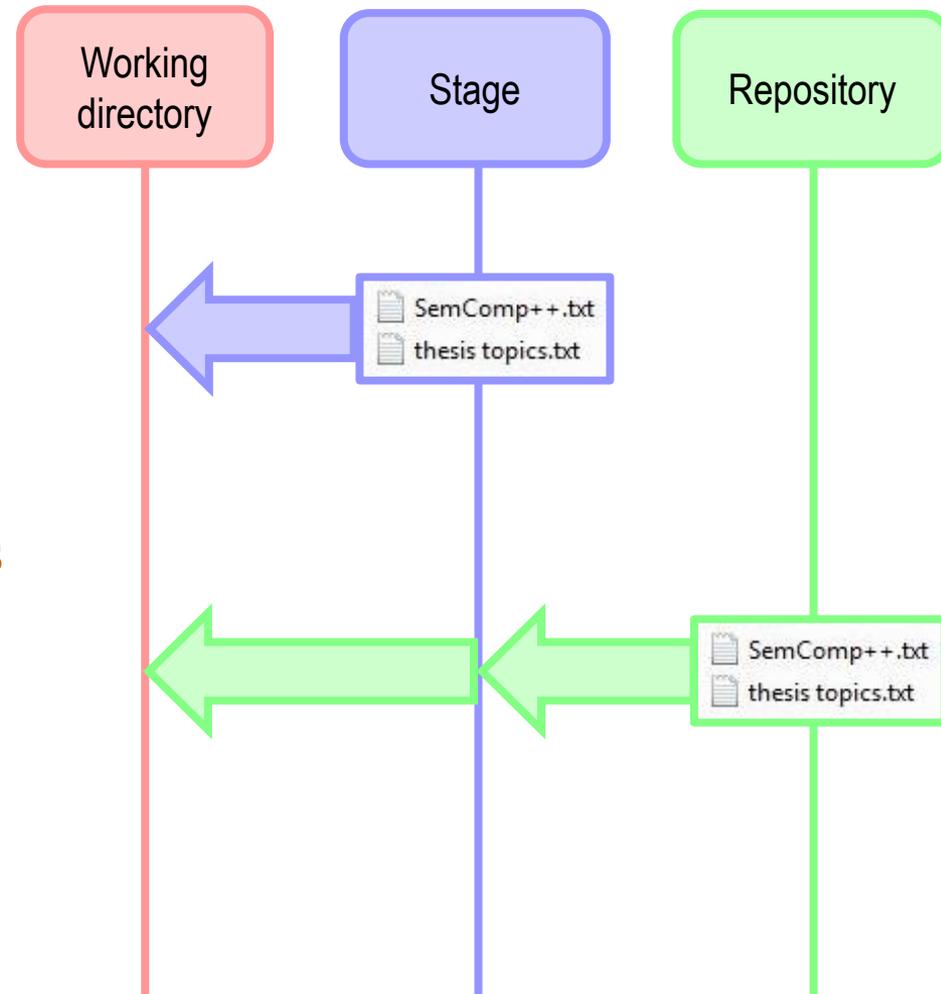
## git checkout

### git checkout -- files

- **files** werden aus Stage ins working directory kopiert
- **files** im working directory werden überschrieben

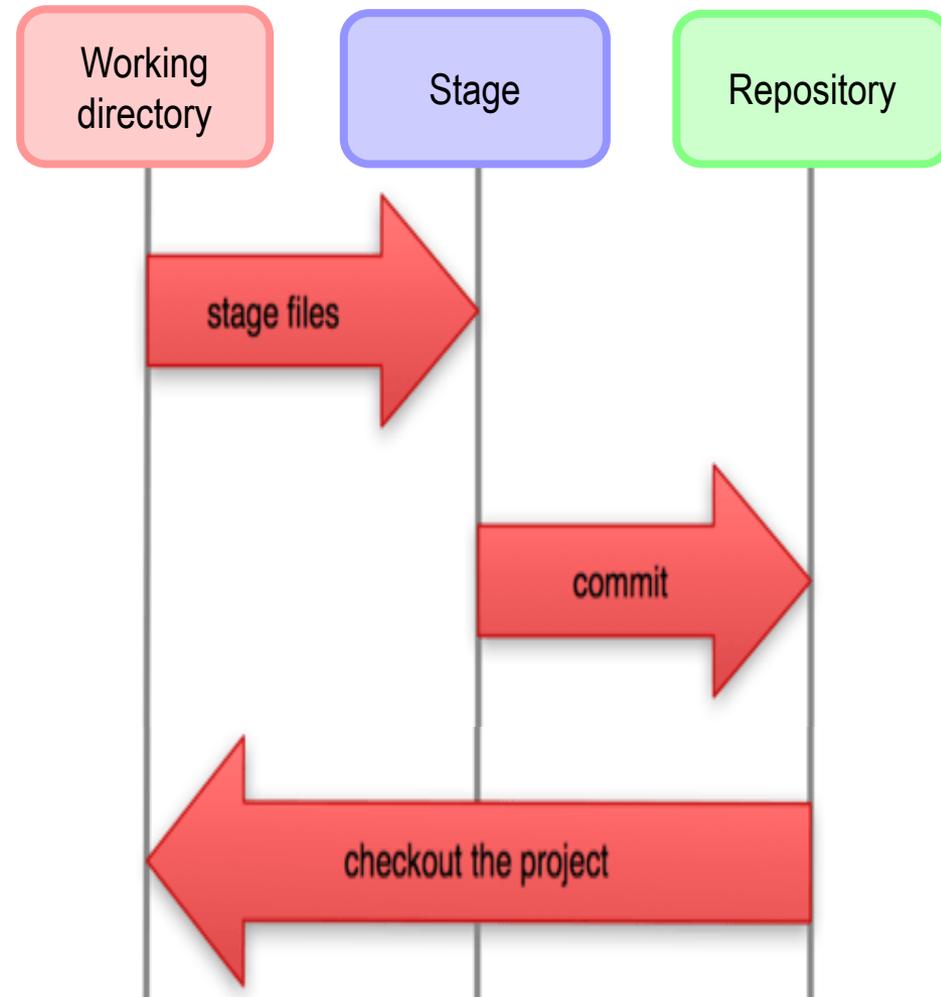
### git checkout commit -- files

- **files** werden aus angegebenem **commit** im aktuellen Branch des Repositories in die Stage und ins working directory kopiert
- **files** in Stage und working directory werden überschrieben



# Lokale Operationen ► Zusammenfassung

- Staging
  - ◆ Speichert in der „**Stage**“ ausgewählte geänderte Dateizustände zwischen
- Commit
  - ◆ Überträgt Inhalt der Stage als neuen Commit ins Repository
- Checkout
  - ◆ Überschreibt Stage und Working directory mit Inhalt eines bestimmten Commits aus Repo



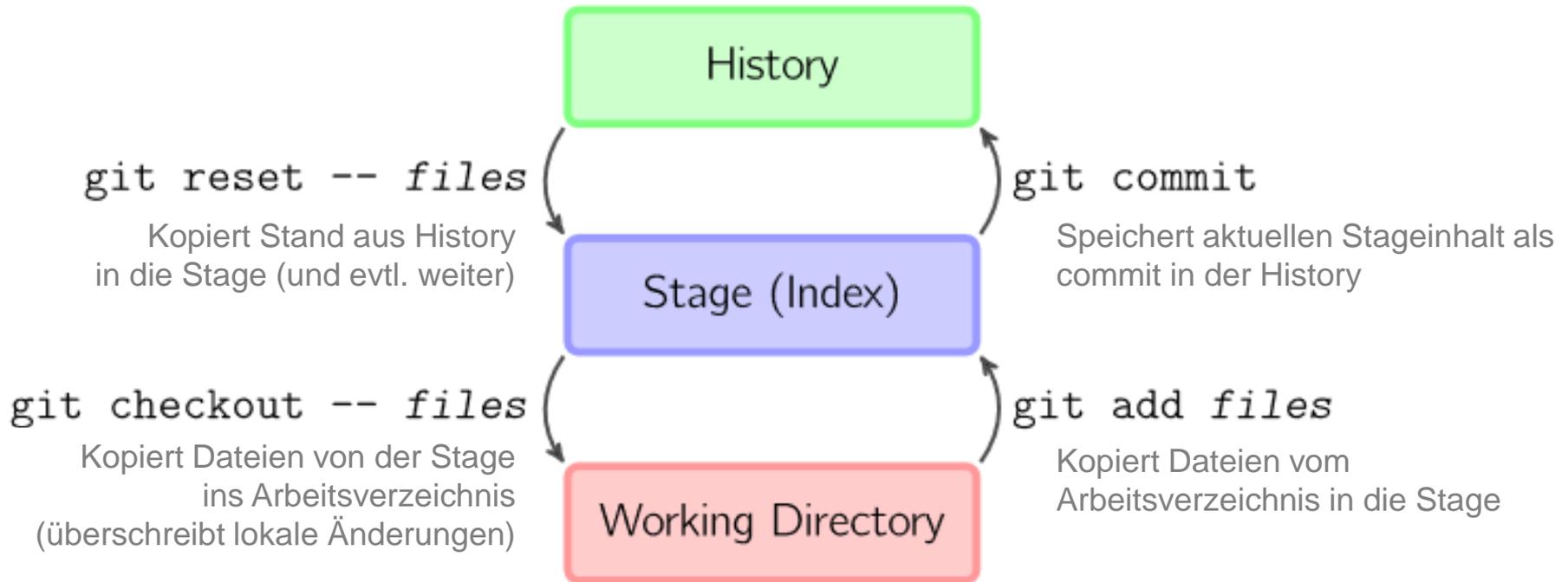
# Änderungen rückgängig machen

- ▶ **Des Pudels Kern:** Fehler beseitigen!!!

# Was wollen wir evtl. rückgängig machen?

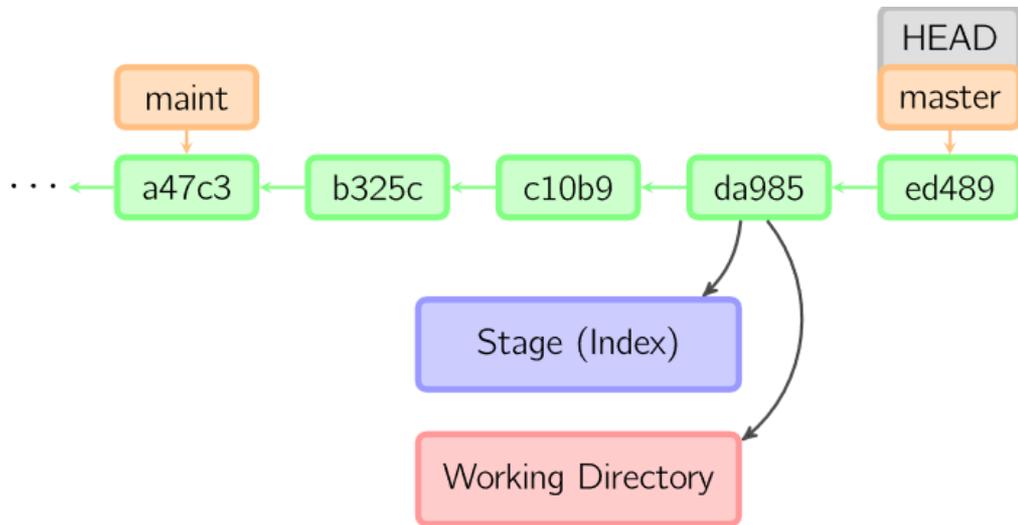
- Änderungen im working tree → Überschreiben mit altem Inhalt
  - ◆ Hinzufügen, Löschen oder Editieren von Dateien wird rückgängig gemacht
  - ◆ Zurücksetzen auf Stand eines bestimmten commit
  - ◆ **Achtung, hier kann es Datenverlust geben ☹, denn die neuesten Änderungen im Working Tree sind nirgends sonst gesichert (wenn sie es vorher nicht selbst getan haben)!!!**
- Stage → Unstage
  - ◆ Für nächsten commit vorgesehene Änderungen aus der Stage entfernen
  - ◆ Arbeitsbereich wird nicht verändert
- Commit → Uncommit
  - ◆ Aktueller Branch wird zurückgesetzt auf einen bestimmten commit
  - ◆ Neuere commits werden vergessen
  - ◆ Stage und Arbeitsbereich werden nicht verändert

# Grundoperationen



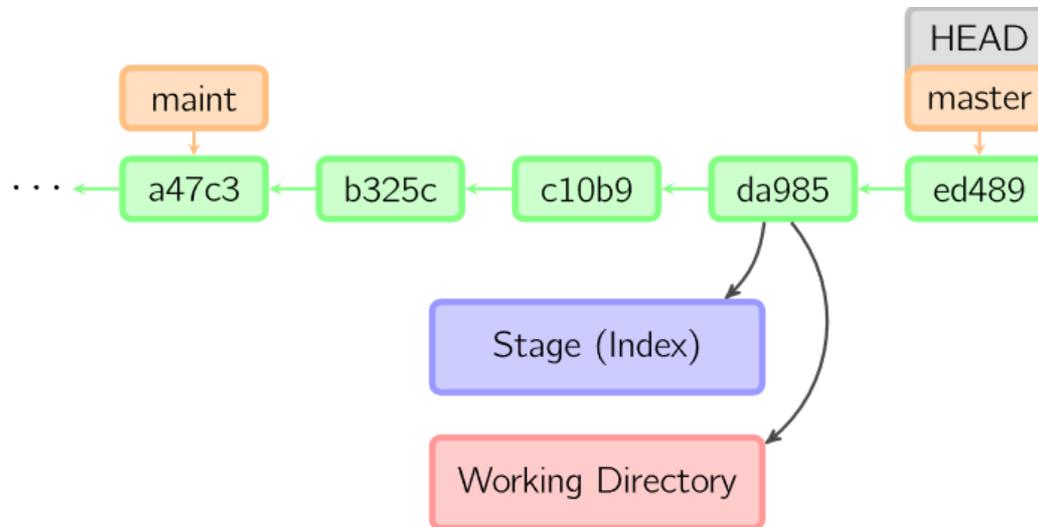
# git checkout **commit** [paths]

- Beispiel: Checkout aller Dateien im letzten commit vor HEAD (= HEAD~)
- ◆ git checkout HEAD~



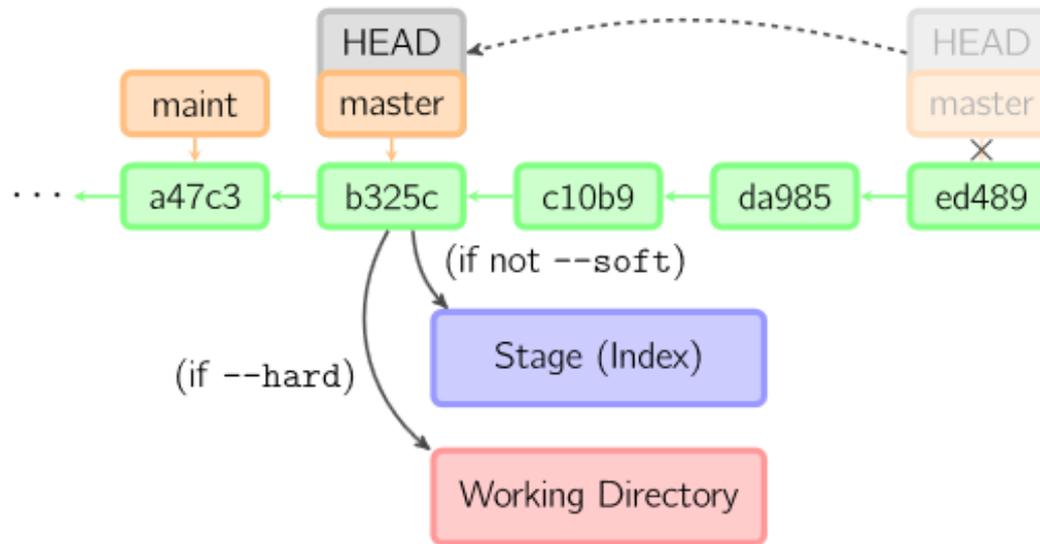
- **commit** angegeben
  - ◆ → Inhalt des **commit** überschreibt Stage und Arbeitsbereich
- Kein **commit** angegeben
  - ◆ → Inhalt der Stage überschreibt Arbeitsbereich
- Pfad angegeben → git überschreibt nur diese Datei(en)

# git checkout **commit** [paths]



# git reset [--option] commit

- Verschiebt aktuellen Branch zu **commit**
  - ◆ Im aktuellen Branch alle commits vergessen die neuer sind als **commit**
  - ◆ Implementierung: Aktueller Branch-Zeiger wird auf **commit** zurückgesetzt
    - “uncommit”
- Kann zusätzlich den Inhalt von commit in den Index (**--mixed**) oder in Index und Arbeitsverzeichnis (**--hard** ← **Achtung, Datenverlust!**) kopieren:



**git reset HEAD~ 3**

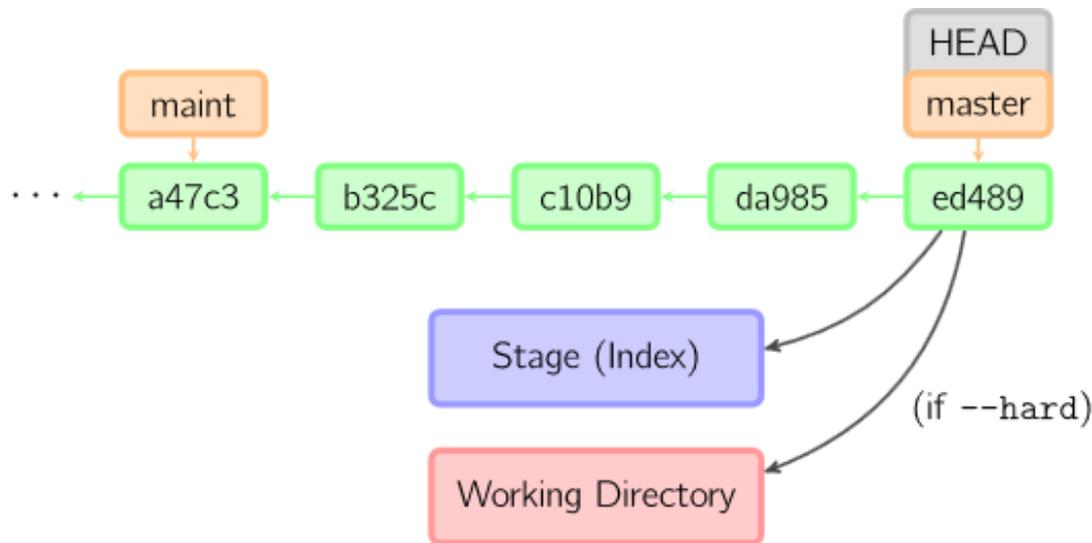
# git reset [--option] commit

Je nach Option:

- **--soft**
  - ◆ Im aktuellen Branch alle commits vergessen die neuer sind als **commit**
  - ◆ Implementierung: Aktueller Branch-Zeiger wird auf **commit** zurückgesetzt
    - “uncommit”
- **--mixed** (*default wenn keine Option angegeben*)
  - ◆ Zusätzlich zu 1. Inhalt des **commit** in Stage übertragen
  - ◆ Somit in Stage vorgemerkte Änderungen rückgängig machen
    - “uncommit and unstage”
- **--hard**
  - ◆ Zusätzlich zu 2. Inhalt des Working Tree mit **commit**-Zustand überschreiben
  - Alle Änderungen seit **commit** rückgängig machen ← **Achtung, Datenverlust möglich!!!** ☹

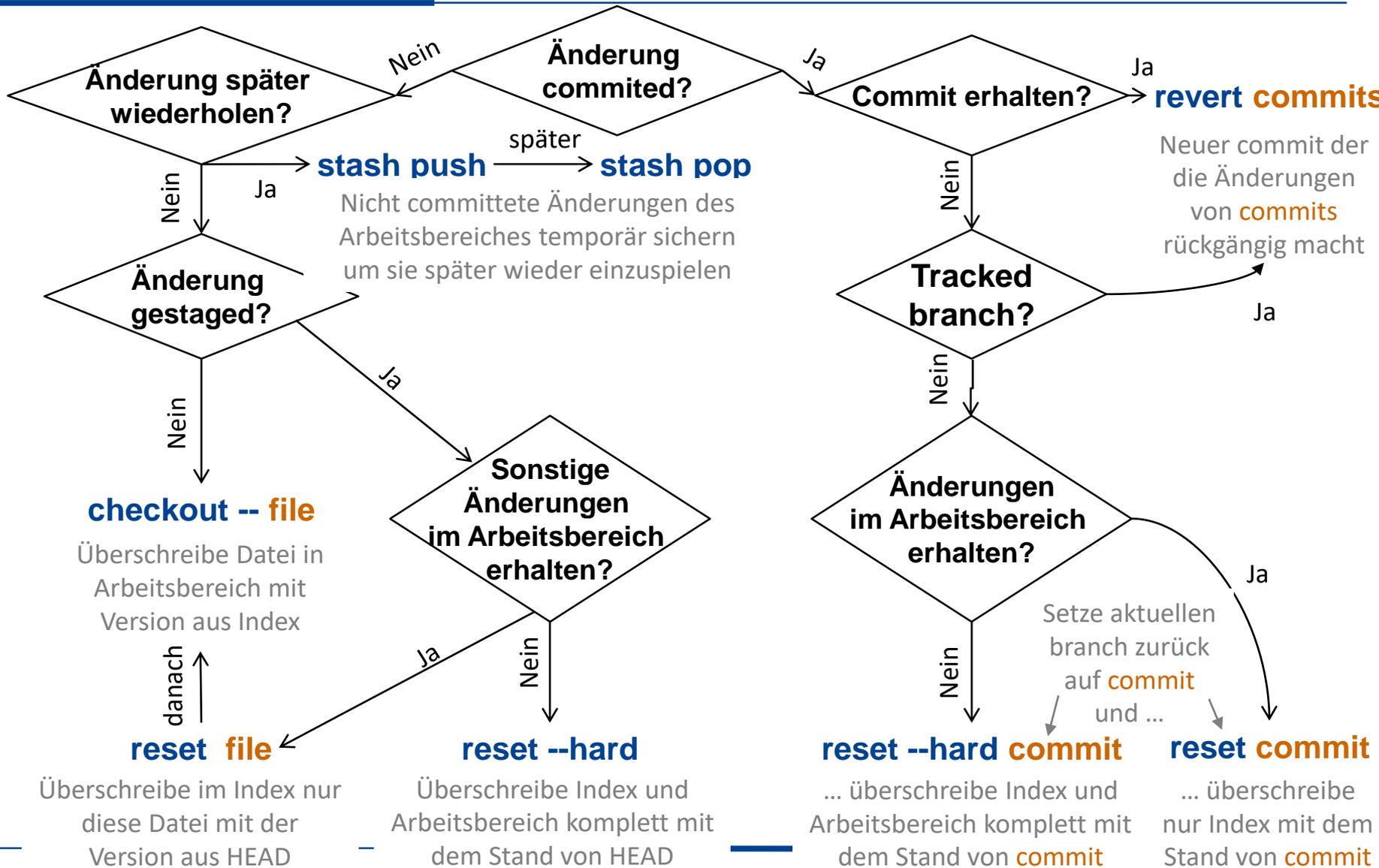
# git reset filePath

- Gegenstück zu **git add filePath (= stage)**
  - ◆ Überschreib den Zustand einer Datei in der Stage/Index mit ihrem Zustand aus dem letzten Commit des aktuellen Branches (HEAD)
  - ◆ Damit wird die Stage-Operation Rückgängig gemacht (**= unstage**)
- Mit der Option **-- hard** wird auch der Zustand der Datei im Arbeitsbereich überschrieben ← **Achtung, Datenverlust!**



**git reset myBrokenFile.c**

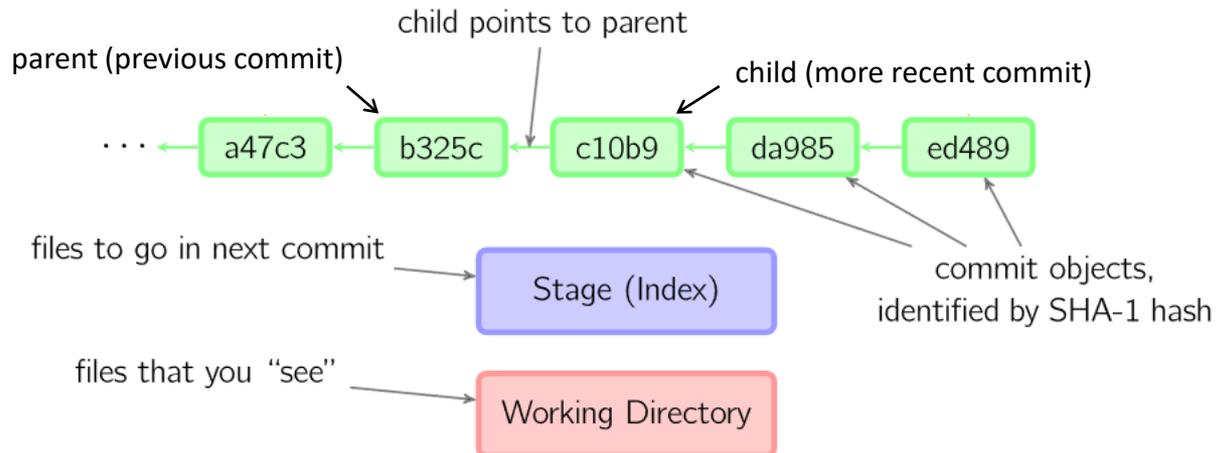
# Zusammenfassung: Änderung rückgängig machen



# **Interne Darstellung und Implementierung der Operationen**

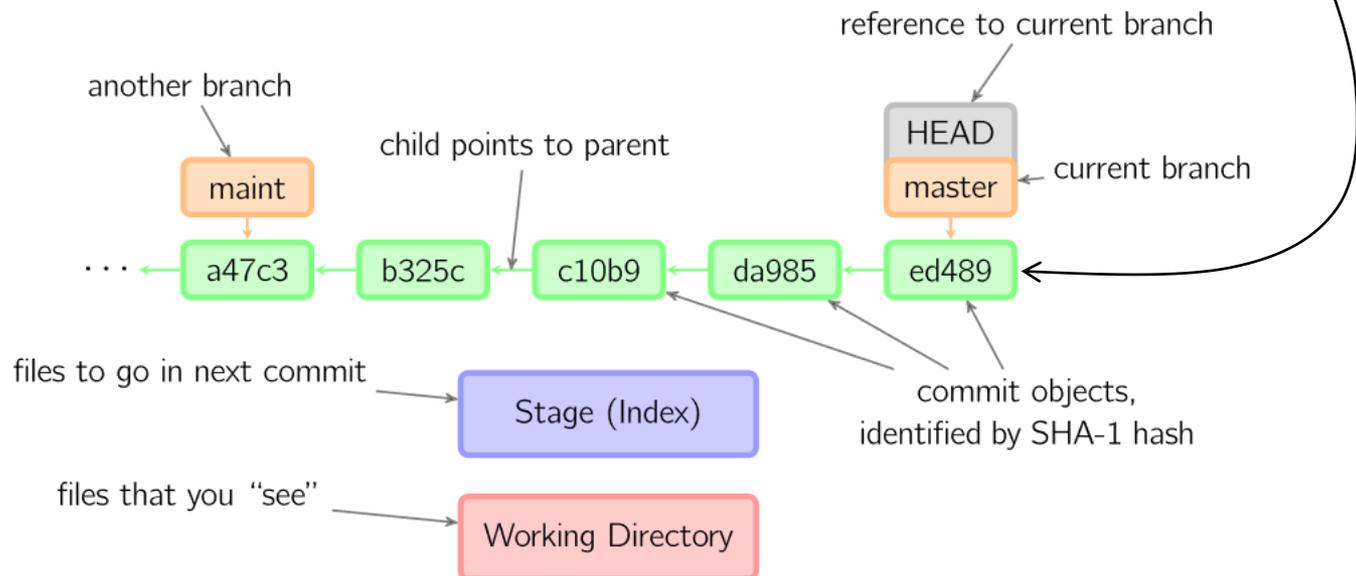
# Interne Darstellung ► Commits

- Ein Commit ist eine Momentaufnahme („snapshot“) der unter Versionskontrolle stehenden Dateien zu einem bestimmten Zeitpunkt
  - ◆ Commits werden im Folgenden grün mit 5-Buchstaben-IDs dargestellt
- Jeder Commit verweist auf seine Eltern (Vorhänger)
  - ◆ Normalfall: Jeder commit hat genau ein Vorgänger
  - ◆ Sonderfall: Commit für eine Merge-Operation hat zwei Vorgänger



# Interne Darstellung ► Branches

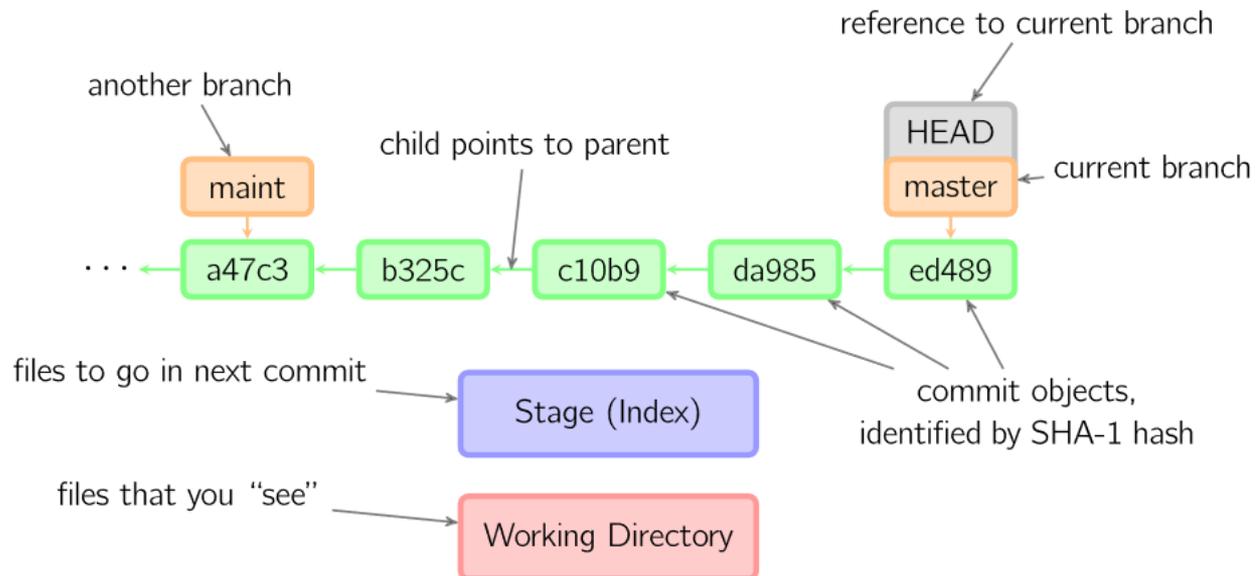
- Ein Branch ist konzeptuell eine lineare Folge von Commits
- Ein Branch ist implementiert als Zeiger auf das neueste Commit in der Folge (im Bild in orange dargestellt)
- HEAD = der aktuelle Branch
  - ◆ meint oft implizit den aktuellsten Commit des aktuellen Branches
- HEAD~n = n commits zurück im aktuellen Branch



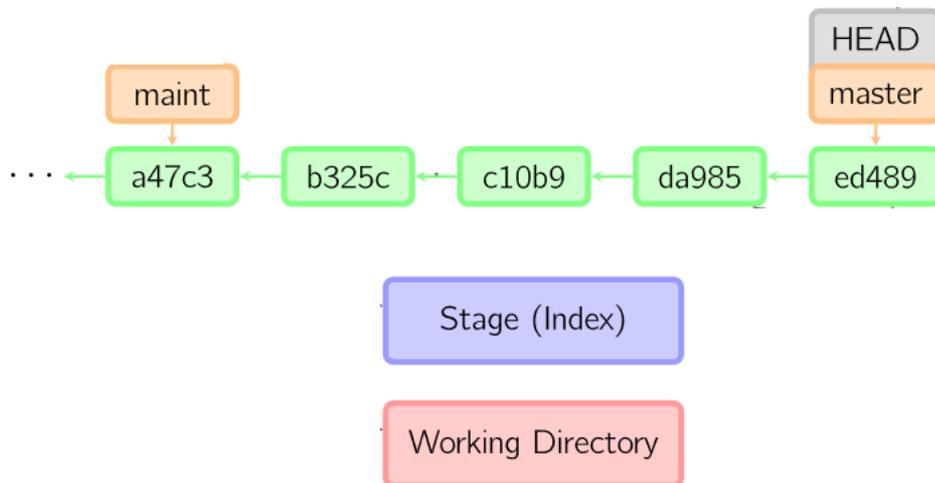
# Interne Darstellung ► Branches

In diesem Bild werden die fünf aktuellsten Commits gezeigt, wobei ed489 der jüngste ist.

- **master** zeigt auf genau diesen Commit
- **maint** zeigt auf den 4-ten Vorfahren von master
- **HEAD** (=der aktuelle Branch) ist **master**.



# Interne Darstellung ▶ Branches

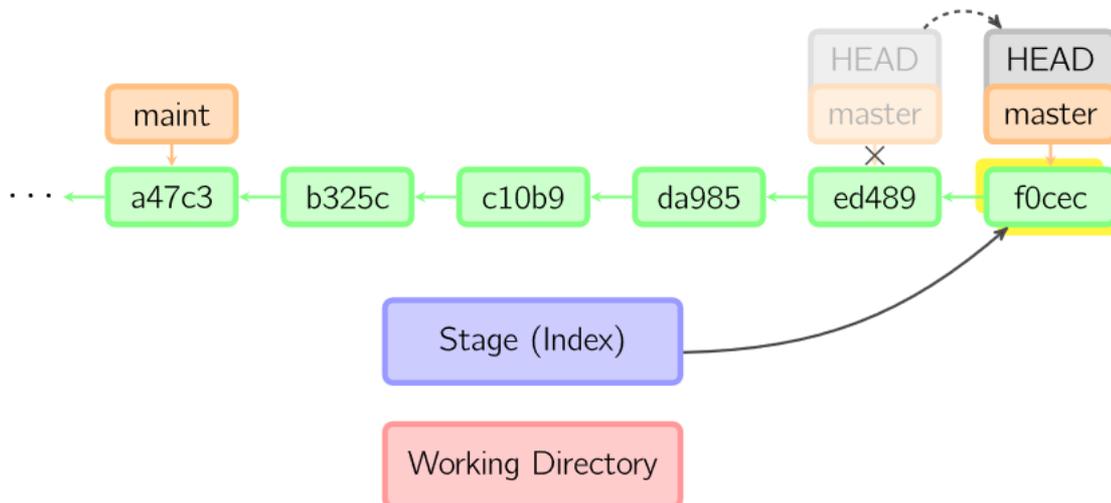


# git commit

- Erzeugt ein neues Commit Objekt mit den Dateien der Stage und setzt den aktuellen Commit als Vorfahre ein.
- Verschiebt den Zeiger des aktuellen Branches auf den neuen Commit.

In dem folgenden Bild ist der aktuelle Branch *master*:

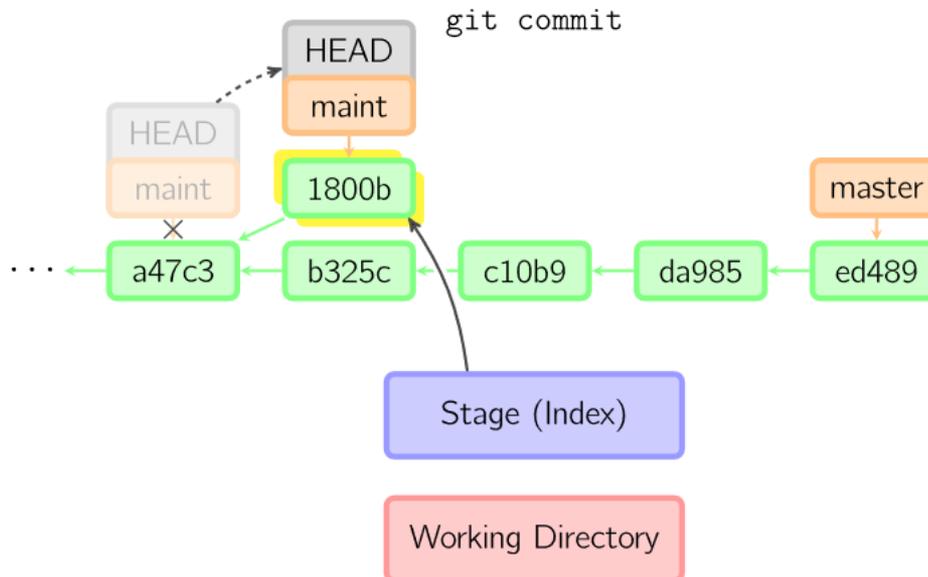
- ◆ Vor dem commit zeigte *master* auf *ed489*
- ◆ Danach wurde ein neuer Commit *f0cec* mit dem Vorläufer *ed489* erstellt
- ◆ ... und die Branch Referenz *master* auf den neuen Commit verschoben.



# git commit

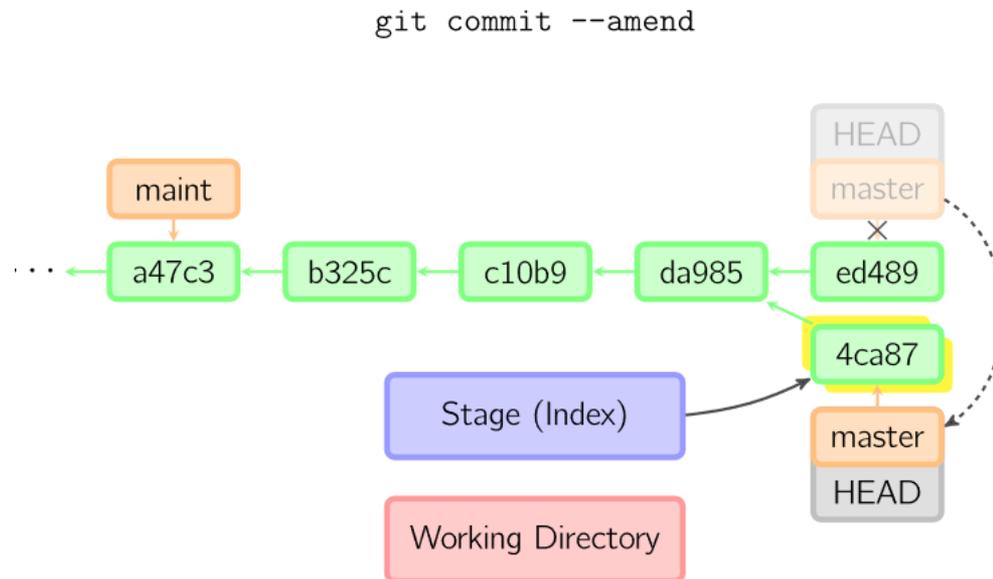
Es funktioniert auch wenn die Spitze des aktuellen Branches auch zu einem anderen Branch gehört:

- Beispiel: Commit `a47c3` ist Spitze von `maint` und hat `b325c` als Kind
  - ◆ Nun commit `1800b` in den `maint` Branch einfügen
  - ◆ Danach divergieren `maint` und `master`.
  - ◆ Um beide zusammen zu führen ist ein `merge` (oder `rebase`) notwendig.



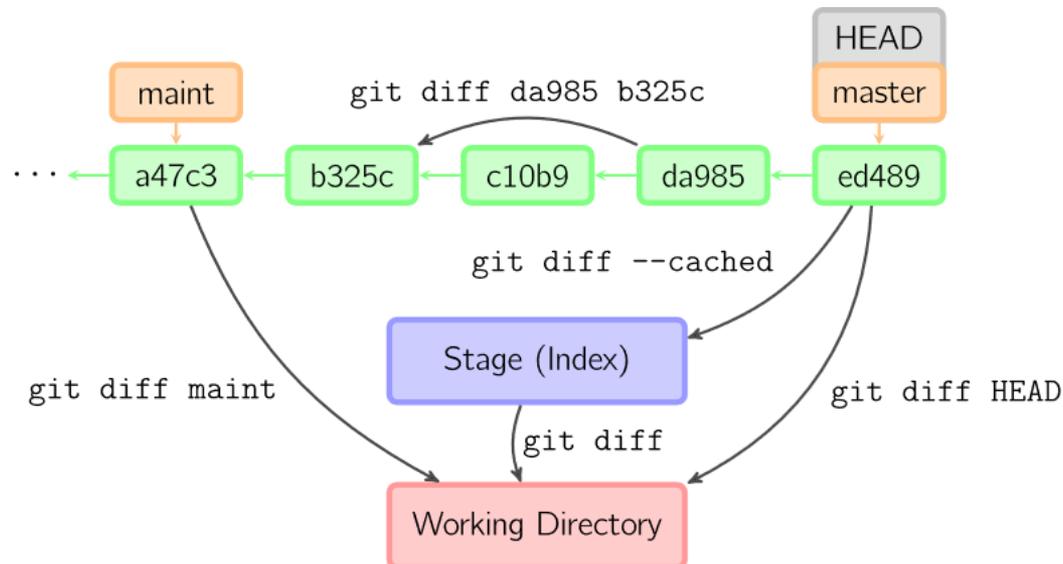
# git commit -- amend

- Die `-- amend` Option ersetzt den vorherigen Commit
  - ◆ weil man zu viel oder zu wenig committed hat ...
  - ◆ ... oder um den Commit-Kommentar zu korrigieren
- Git erstellt einen neuen Commit der den selben Vorgänger hat wie der aktuelle Commit:



# git diff

- **git diff** erlaubt es Unterschiede zwischen Commits zu betrachten
  - ◆ implizit: Stage versus Arbeitsbereich – `git diff`
  - ◆ explizit: Head versus Arbeitsbereich – `git diff HEAD`
  - ◆ explizit: Head versus Stage – `git diff --cached`
  - ◆ explizit: Branch versus Arbeitsbereich – `git diff maint`
  - ◆ explizit: zwei commits – `git diff da985 b325c`
  - ◆ Es können explizit Dateien als Argument angehängt werden. Dies beschränkt die Operation auf diese Dateien.



# Commits ohne vorhergehendes Staging

---

- **git commit --a**

- ◆ Automatisches staging aller Dateien unter Versionskontrolle vor dem commit → Committed alle Änderungen im Arbeitsbereich
- ◆ Oft nicht was man will! ☹️

- **git commit files**

- ◆ Stage und commit der **files** aus Arbeitsbereich



# Arbeiten mit „Branches“

- ▶ **Erzeugen:** branch
- ▶ **Umschalten:** checkout
- ▶ **Abgleichen:** merge, cherry picking, rebase

# Kurzüberblick – Branch, Checkout, Merge

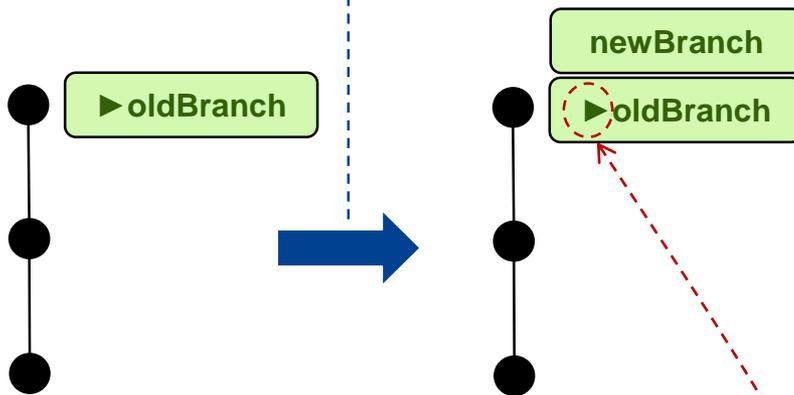
Neben der dezentralen Struktur bietet Git noch eine besondere Stärke: Es ist sehr leicht neue Entwicklungslinien („branches“) zu erzeugen und wieder zusammen zu führen („merge“).

- Typische Vorgehensweise:
  1. Branch erzeugen
  2. Implementieren und Testen (Bugfix, neue Funktionalität, etc.)
  3. Merge
- „Merge“ kann zwischen beliebigen branches stattfinden. Ablauf:
  - ◆ Wechsel in den Ziel-Branch: „`checkout Zielbranch`“
  - ◆ Merge des Quellbranches in aktuellen branch: „`merge Quellbranch`“
- „Merge“ findet automatisch statt als Teil eines „pull“. Dabei wird ein „tracking branch“ des Remote Repositories (von dem das „pull“ durchgeführt wird) in den entsprechenden („getrackten“) branch des lokalen Repositories ge-merged.
- Merge kann zu Konflikt(en) führen

# Branch erzeugen und wechseln

## Add Branch

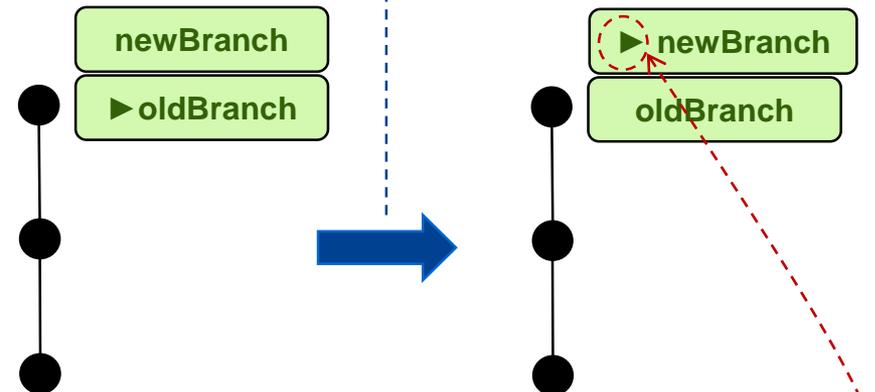
```
git branch newBranch
```



- „newBranch“ erstellt
- Weiterarbeiten in „oldBranch“
  - Nächster „commit“ geht in „oldBranch“

## Checkout Branch

```
git checkout newBranch
```

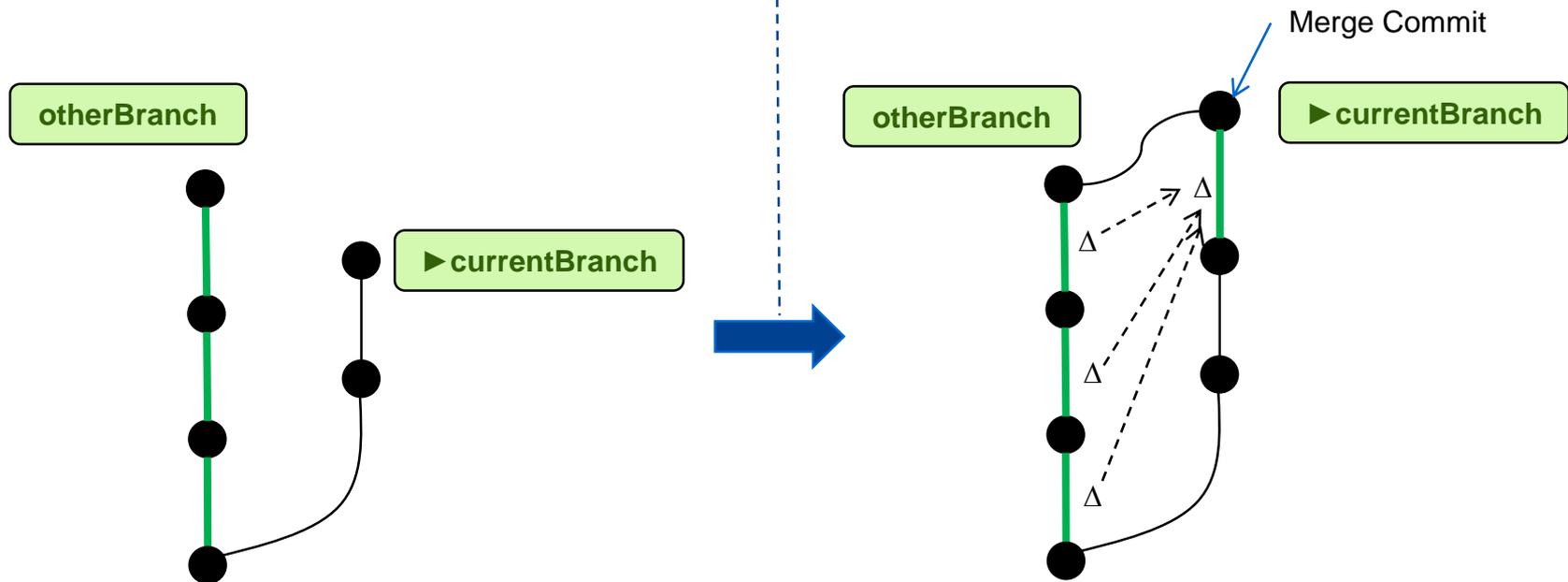


- Wechsel in „newBranch“
  - Nächster „commit“ geht in „newBranch“

# Merge: Branches Synchronisieren

```
git checkout currentBranch
```

```
git merge otherBranch
```

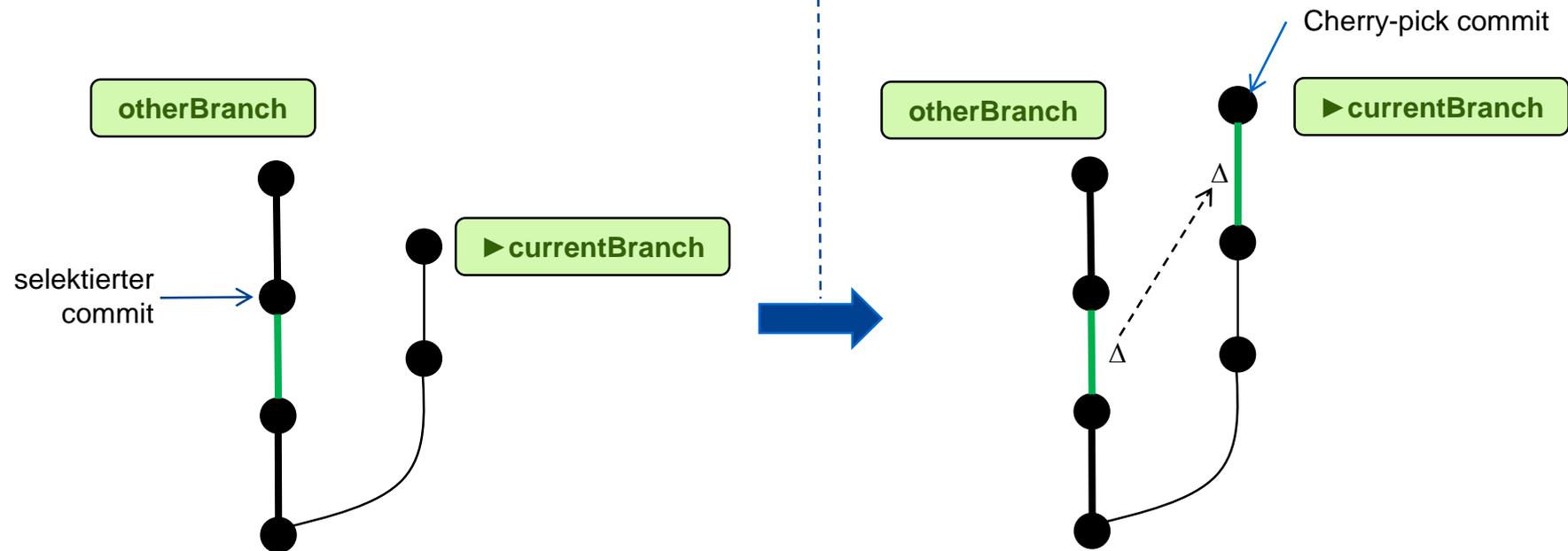


Alle Änderungen vom Verzweigungspunkt der beiden branches bis zum neuesten Commit des Quellbranches werden in den aktuellen Branch übernommen

# Chery Pick: Einzelnes Commit übernehmen

```
git checkout currentBranch
```

```
git cherry-pick commit
```



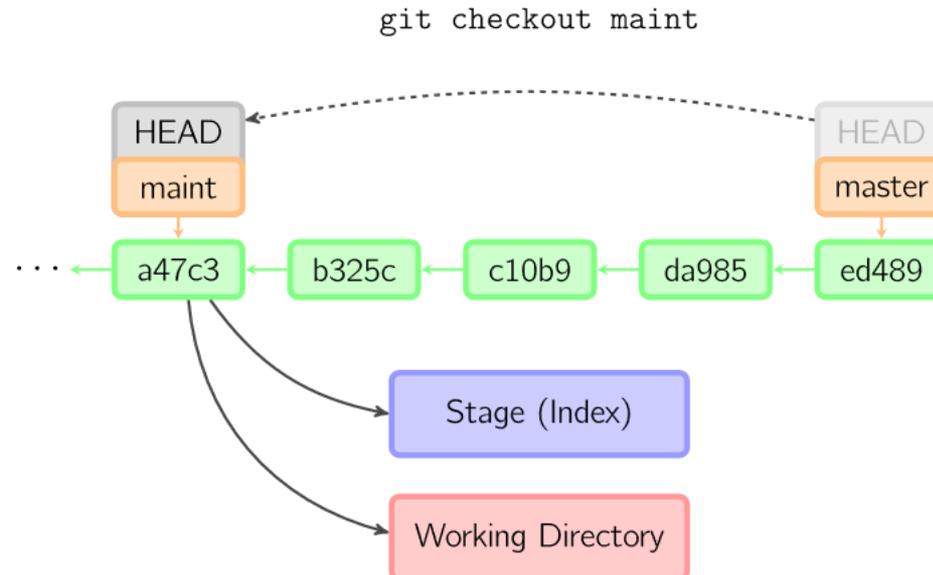
Nur Änderungen des selektierten Commits aus einem anderen Branch werden in den aktuellen Branch übernommen (z.B. um Bugfix selektiv zu übernehmen)

# git checkout **branch**

- ▶ **Hier**: Nutzung von checkout zum Wechsel in einen anderen Branch
- ▶ **Später**: Nutzung von checkout zur Wiederherstellung alter Dateiversionen

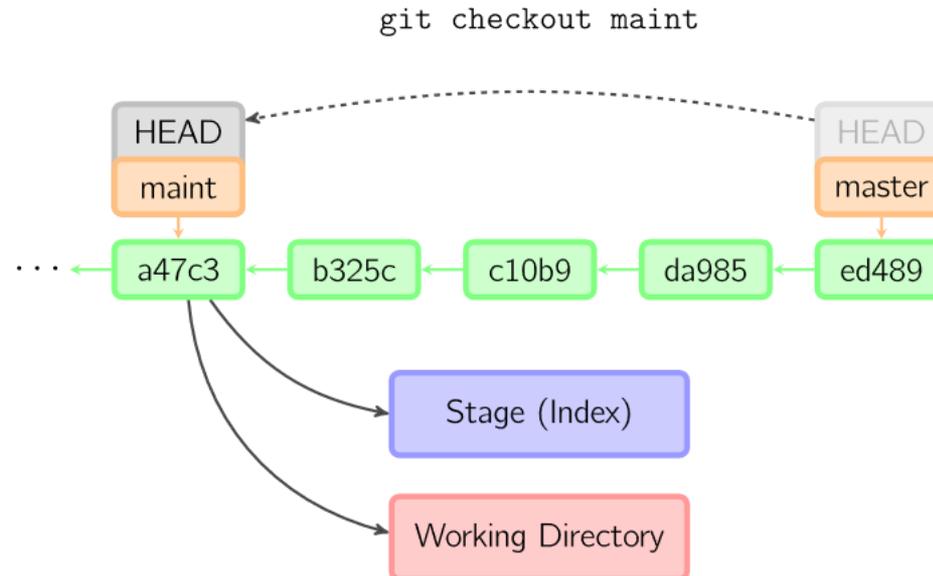
# git checkout branchName

- Ändern des Branches
  - ◆ Checkout mit Name eines (lokalen) Branches
  - ◆ HEAD wird auf diesen Branch gesetzt
  - ◆ Daraufhin werden die Dateien im Index und im Arbeitsverzeichnis mit denen aus dem neuen Head überschreiben
- Beispiel: **git checkout maint**



# git checkout branchName

- Jede Datei die im neuen Commit ( a47c3 s.u.) existiert wird kopiert.
- Jede die im alten Commit (ed489) existiert, aber nicht im neuen, wird im Index und Arbeitsverzeichnis gelöscht.
- Alle weiteren werden nicht angefasst.
  - ◆ Dateien, die nicht unter Versionskontrolle stehen bleiben unberührt



# git merge

Fast Forward Merge

3-Wege-Merge

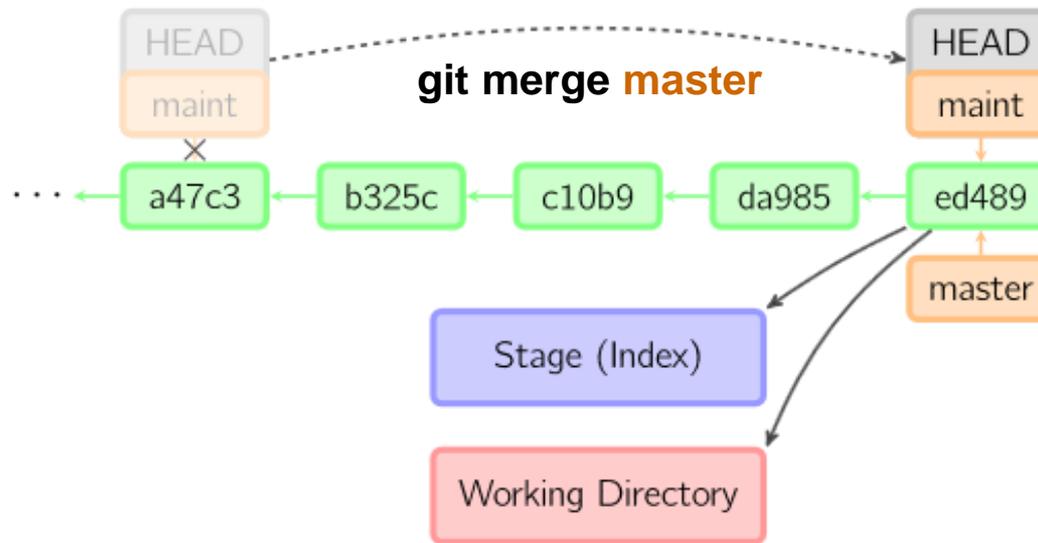
# git merge branchName

## Merge – allgemein

- im aktuellen branch neuen Commit erstellen, der Änderungen von commits aus einem anderen branch übernimmt

## Merge – fast forward

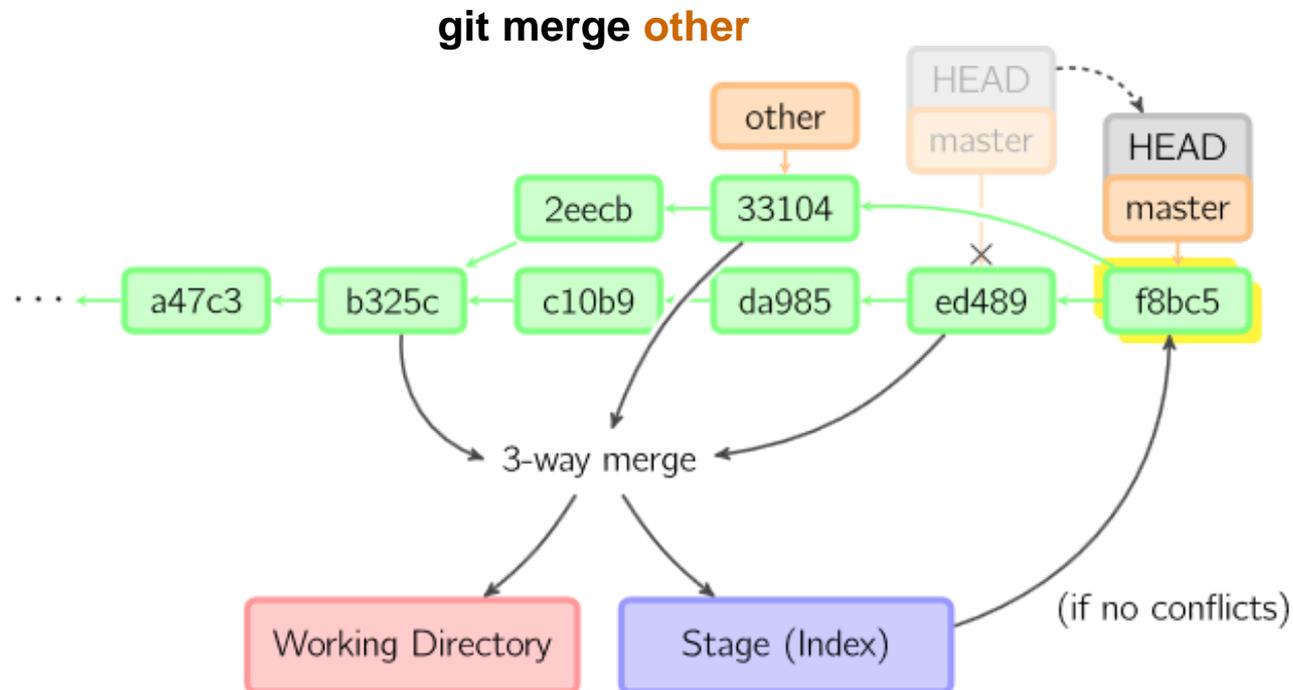
- falls eigener branch (hier **maint**) Vorläufer des anderen ist, einfach eigenen branch-Zeiger weiter setzen:



# git merge branchName

Ansonsten wird ein **rekursiver 3-Wege-Merge** durchgeführt auf Basis

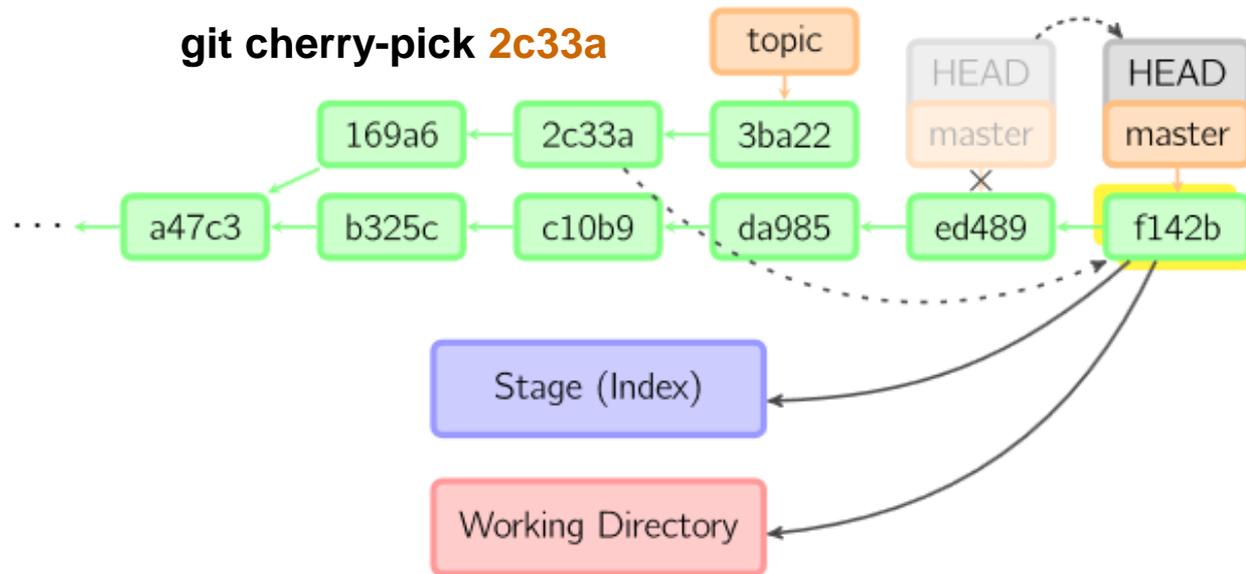
- des aktuellsten Commit im eigenen branch (**ed489**),
- des aktuellsten Commit im anderen Branch (**33104**) und
- ihres gemeinsamen Vorläufers (**b325c**)



# **git cherry-pick**

# git cherry-pick **commitID**

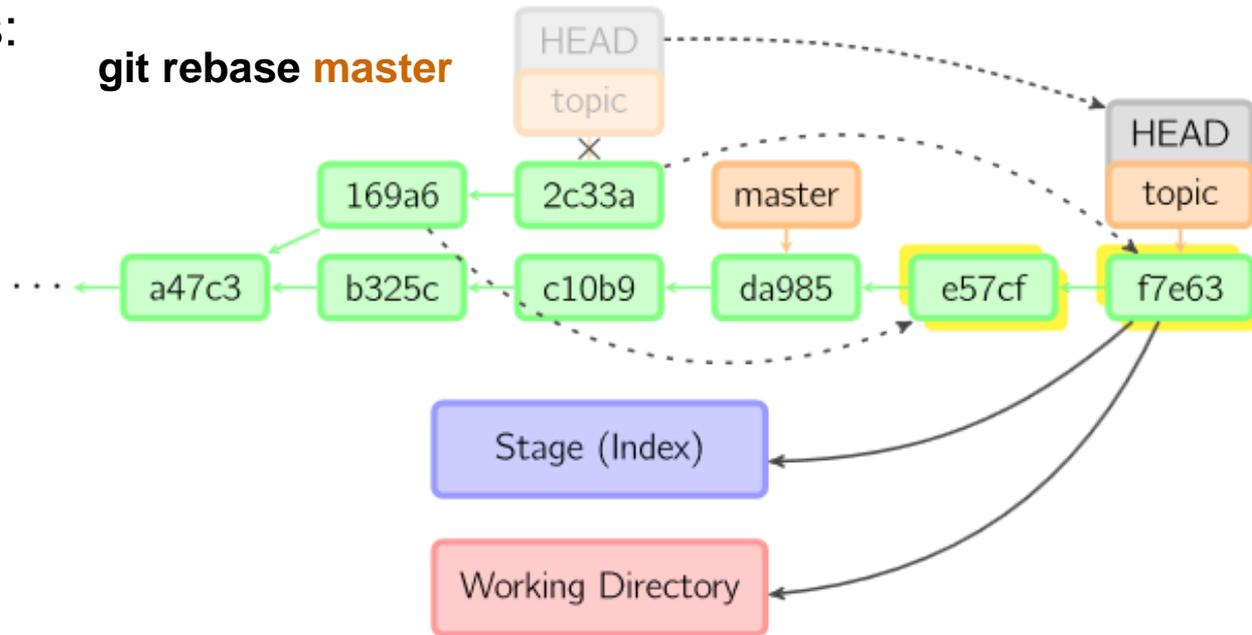
- Der cherry-pick (Kirschen bzw. Rosinen rauspicken) Befehl „merged“ selektiv **genau einen Commit** in den aktuellen Branch:
  - ◆ die Änderungen dieses commits
  - ◆ seinen Kommentar
- Gegebenenfalls entstehen Konflikte (wie bei einem normalen merge)



# **git rebase**

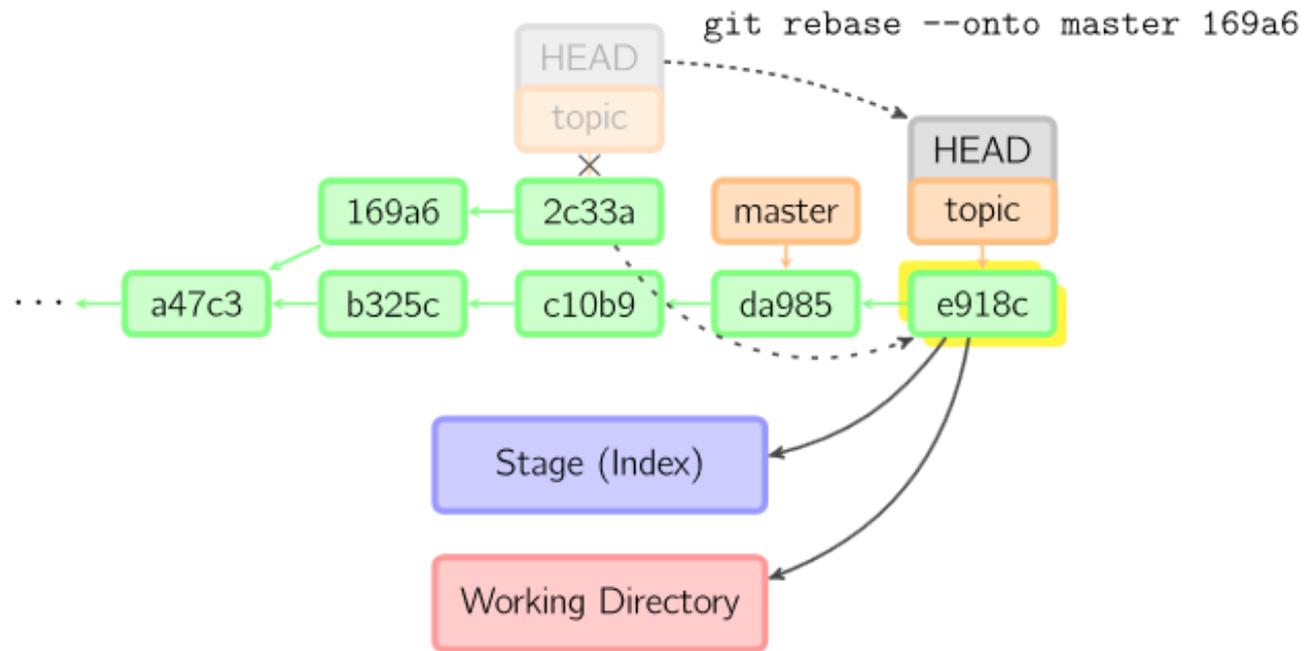
# git rebase branch

- Rebase spielt die Commits des aktuellen Branches auf einen anderen Branch auf.
- Es hinterlässt eine Historie, an der nicht mehr erkennbar ist, dass die übertragenen commits aus einem anderen branch stammten.
- Es ist wie cherry picking **aller** commits seit der Verzweigung der beiden branches:



# git rebase --onto **branch** **commitID**

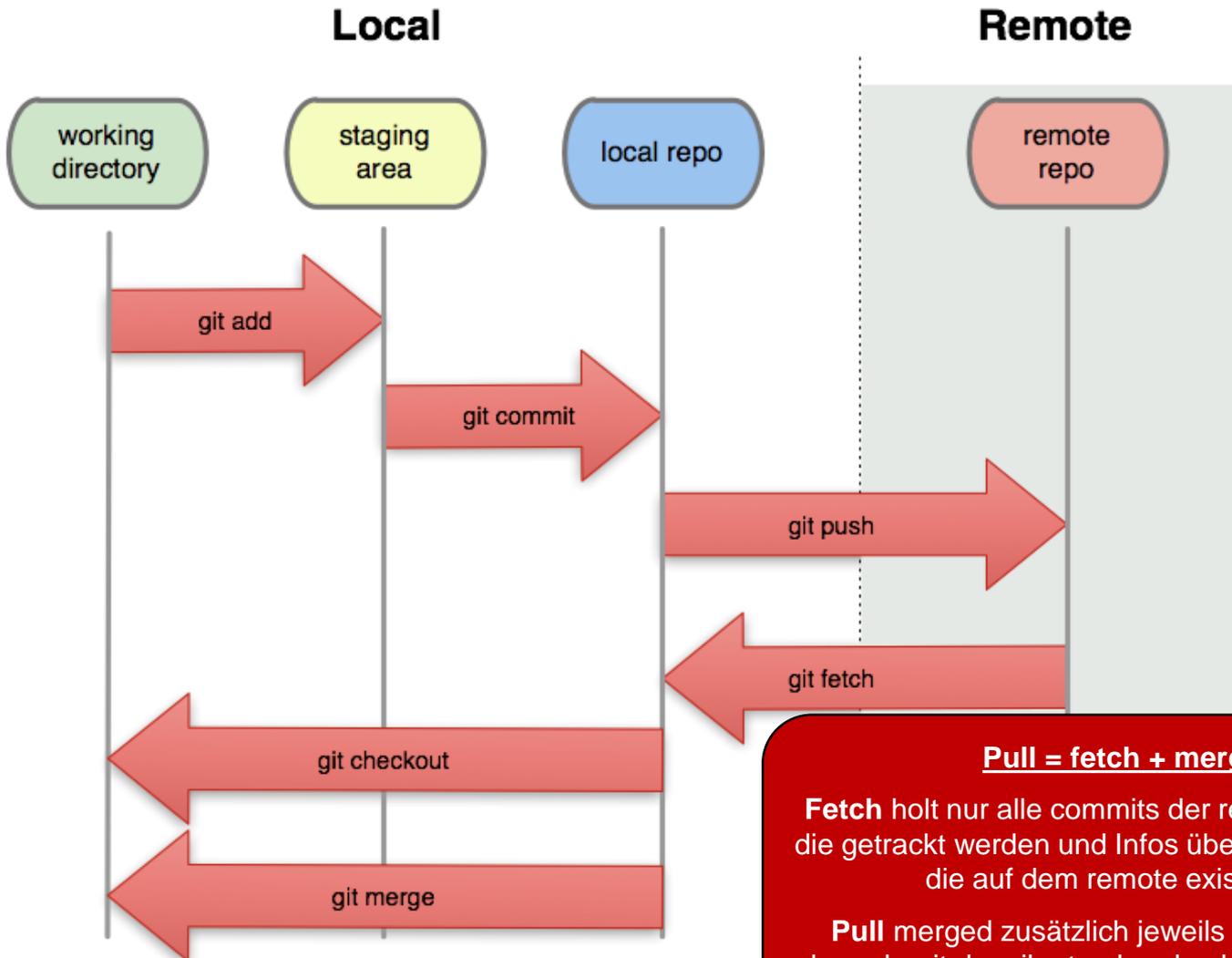
Mit der Option `--onto` werden nur die Commits des aktuellen branches die **neuer als commitId** sind auf **branch** übertragen:



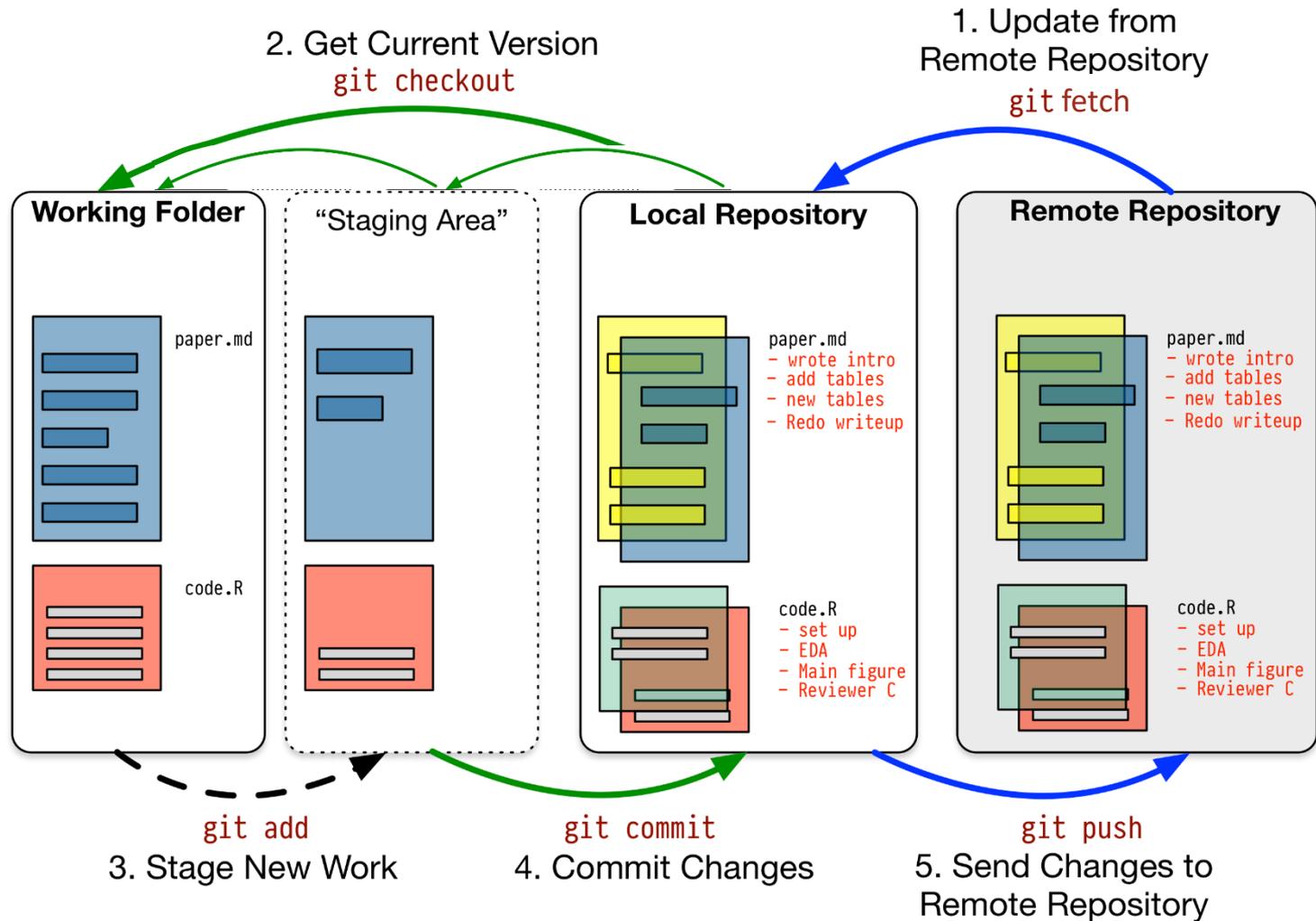
# Verteiltes Arbeiten

- ▶ **Terminologie:** Remotes, origin, tracking
- ▶ **Operationen:** clone, fetch, pull, push

# Gesamtbild: Lokal und verteilt



# Gesamtbild



# Plattformunabhängige GIT-Clients

---

- SmartGIT

- ◆ Kommerzielles Produkt aber für nichtkommerzielle Anwendung frei
- ◆ Stabil, ausgereift
- ◆ Gute dedizierte SCM-GUI
- ◆ <http://www.syntevo.com/smartgithg/>

- Egit

- ◆ Freies Plugin für Eclipse
- ◆ Sehr umfangreich, in Eclipse integriert
- ◆ SCM leider nur eines von vielen Dingen die Eclipse tut
  - ⇒ GUI dementsprechend spartanisch im Vergleich zu SmartGit und anderen
- ◆ Sehr ausführliches Tutorial ([http://wiki.eclipse.org/EGit/User\\_Guide](http://wiki.eclipse.org/EGit/User_Guide))
- ◆ <http://www.eclipse.org/egit/>

# Zusammenfassung und Ausblick

Vergleich der Ansätze  
Weiterführende Informationen

# SCM-Ansätze und Werkzeuge ▶ Vergleich

## Zentral (z.B. SVN)

- Operationen

- ◆ Verteilt

- ⇒ —

- ⇒ —

- ⇒ —

- ◆ Lokal, linear

- ⇒ Checkin

- ⇒ Checkout

- ⇒ Update (incl. 3-Wege-Merge)

- ⇒ Commit

- ◆ Lokal, branch management

- ⇒ Branch

- ⇒ Switch

- ⇒ Merge

- ⇒ —

## Verteilt (z.B. GIT)

- Operationen

- ◆ Verteilt

- ⇒ **Clone**

- ⇒ **Pull (incl. 3-Wege-Merge)**

- ⇒ **Push**

- ◆ Lokal, linear

- ⇒ Create repository

- ⇒ —

- ⇒ — (implizit bei „pull“)

- ⇒ Commit

- ◆ Lokal, branch management

- ⇒ Branch

- ⇒ Switch

- ⇒ Merge

- ⇒ **Cherry pick**

# SCM-Ansätze und Werkzeuge ► Vergleich

---

## Zentral (z.B. SVN)

- Es gibt nur ein Repository
- Interaktion nur über zentralen Server

## Verteilt (z.B. GIT)

- Gleichberechtigte Repositories
- Zentrale Instanz organisatorisch möglich

# Vorteile verteilter Versionskontrolle

---

- Eigene, lokale Versionskontrolle
  - ◆ Nicht erst dann einchecken, wenn wirklich alles läuft
  - ◆ Eigene inkrementelle Historie, Rollbacks auch lokal möglich
- Lokal Arbeiten ist sehr schnell
  - ◆ Diff, Commit und Revert sind rein lokal → kein Netzwerkverkehr nötig
- Branching und Merging ist leicht
  - ◆ Schnelles Merge (kein kopieren)
  - ◆ Weitergehende Konfliktauflösung als bei SVN
- Partielle Integration ist leicht
  - ◆ Entwickler können ihre Änderungen untereinander abgleichen
  - ◆ Änderung eines „zentralen“ Masters erst nach partieller Integration
- Wenig Management erforderlich
  - ◆ Schneller Start (einfach „clone“ oder „create repository“)
  - ◆ Nur der eigene Rechner muss andauernd laufen
  - ◆ Kein Usermanagement erforderlich

# Nachteile verteilter Versionskontrolle

---

- Es gibt keine globalen Versionsnummern
  - ◆ Jedes Repository verwaltet seine eigene Änderungshistorie
  - ◆ Aber man kann Releases mit sinnvollen Namen taggen
- Es gibt keine „neueste Version“
  - ◆ Kann man durch ein dediziertes Repository emulieren
- Ein zentrales Backup ist unbedingt sinnvoll
  - ◆ eventuell hat nicht jeder alle Änderungen „gepullt“, daher kann man sich nicht auf andere Repositories als Backup verlassen

# Listen weiterer SCM Werkzeuge

---

- Wikipedia

- ◆ [http://en.wikipedia.org/wiki/List\\_of\\_revision\\_control\\_software](http://en.wikipedia.org/wiki/List_of_revision_control_software)
- ◆ Sehr umfangreiche Liste, mit oft guten Fortsetzungen zu den einzelnen Werkzeugen

- Wikipedia

- ◆ [http://en.wikipedia.org/wiki/Comparison\\_of\\_revision\\_control\\_software](http://en.wikipedia.org/wiki/Comparison_of_revision_control_software)
- ◆ Versuch eines tabellarischen Vergleichs der Systeme
- ◆ Teilweise gut, aber hoffnungslose Sisyphos-Arbeit...

- Andere

- ◆ <http://www.software-pointers.com/en-configuration-tools.html>
- ◆ <http://www.thefreecountry.com/programming/versioncontrol.shtml>
- ◆ [http://www.dmoz.org/Computers/Software/Configuration\\_Management/Tools/](http://www.dmoz.org/Computers/Software/Configuration_Management/Tools/)

# Vergleiche von SCM Werkzeuge

---

- Wikipedia (allgemein)
  - ◆ [http://en.wikipedia.org/wiki/Comparison\\_of\\_revision\\_control\\_software](http://en.wikipedia.org/wiki/Comparison_of_revision_control_software)
  - ◆ Versuch eines tabellarischen Vergleichs der Systeme
  - ◆ Teilweise gut, aber hoffnungslose Sisyphos-Arbeit...
- Wikipedia (nur open source)
  - ◆ [http://en.wikipedia.org/wiki/Comparison\\_of\\_open\\_source\\_configuration\\_management\\_software](http://en.wikipedia.org/wiki/Comparison_of_open_source_configuration_management_software)
- Andere neutrale Vergleiche
  - ◆ <http://better-scm.berlios.de/comparison/> (Stand 29. Feb. 2008)
- Nicht ganz unparteiische Vergleiche...
  - ◆ [http://www.relisoft.com/co\\_op/vcs\\_compare.html](http://www.relisoft.com/co_op/vcs_compare.html)
  - ◆ [http://www.accurev.com/scm\\_comparisons.html](http://www.accurev.com/scm_comparisons.html)