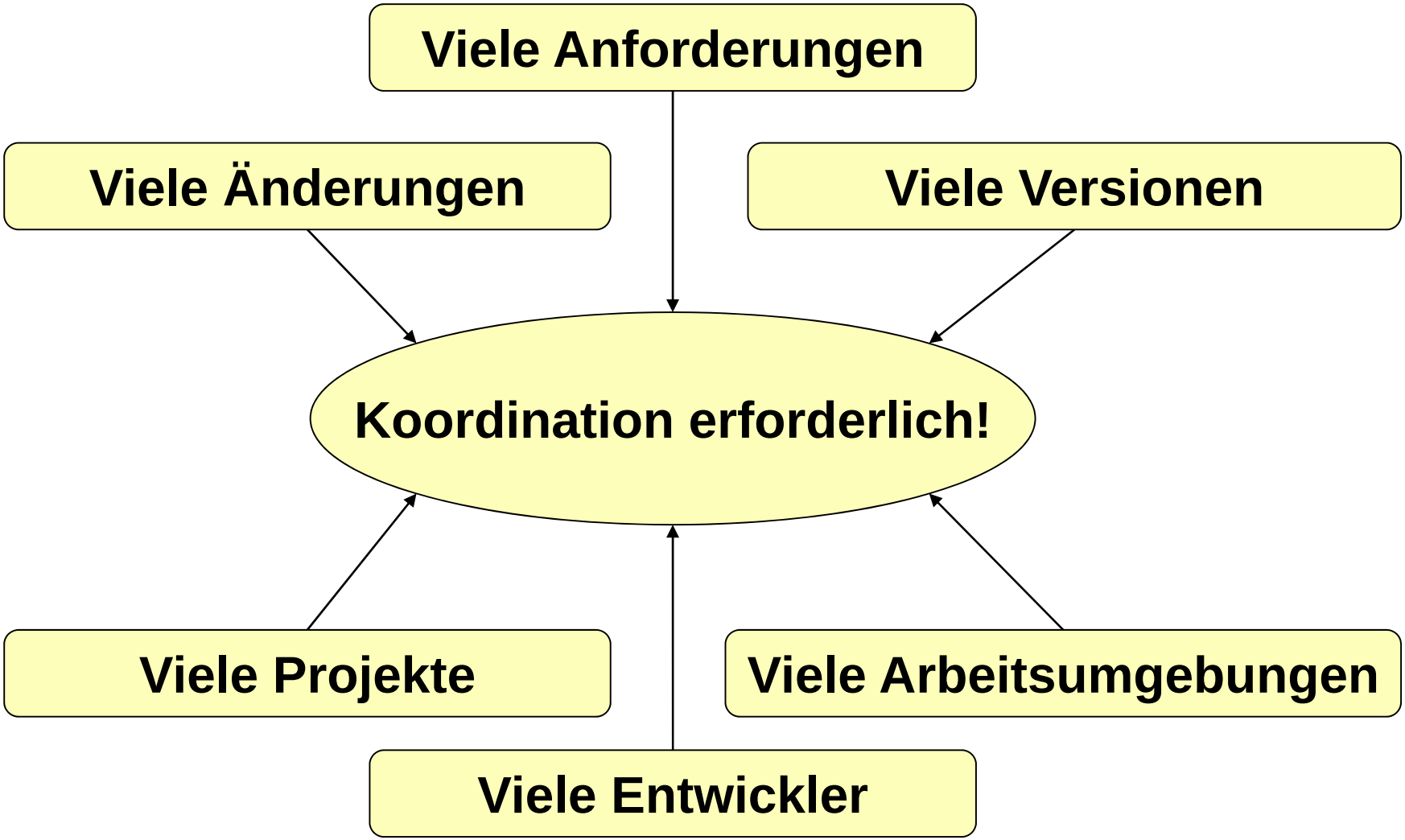
universitätbonn ROOTS

Kapitel 2
Software Configuration Management

Stand: 26. Oktober 2010

Update 26.10.2010: Folie 42+43 korrigiert (Build 3 umfasst Version 3 von Artefakt C).

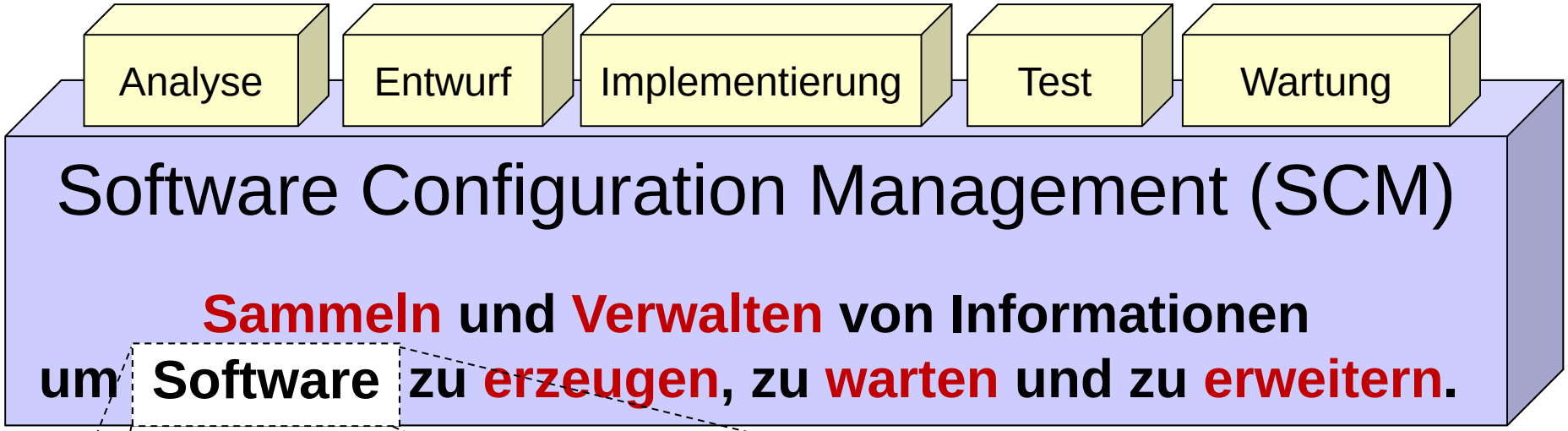
Probleme während der Softwareentwicklung



Warum Software Configuration Management?

Problem	Sie möchten
• Software-Entwicklung ist nicht linear ◆ Man macht Programmierfehler ◆ Man trifft falsche Entwurfsentscheidungen	→ wissen, was Sie wann warum getan haben → zu alter Version zurückgehen
• Software-Entwicklung ist Teamarbeit ◆ Sie und andere arbeiten parallel ◆ ... auf vielen verschiedenen Dateien	→ wissen, wer was wann warum getan hat → Änderungen gemeinsam nutzen
• Verwaltung verschiedener Versionen ◆ Kunde erhält stabile Version, während Entwicklung weitergeht ◆ Bugfixes müssen in alle Versionen integriert werden ◆ Verschiedene Kunden erhalten verschiedene Varianten des Produkts	→ 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 Features

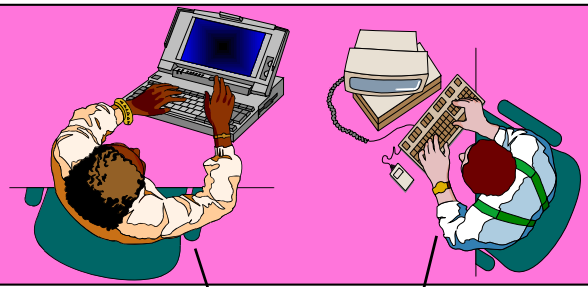
- Managt alle Komponenten des Projekts in einem „Repository“
  - Keine redundanten Kopien
  - Sicher
- Macht Unterschiede zwischen Versionen sichtbar
  - „Was hat sich verglichen mit der gestrigen Version verändert?“
- Erlaubt Änderungen zu identifizieren, auszuwerten, zu diskutieren (!) und sie schließlich 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)
  - Letzte konsistente Version
  - Milestones
  - Releases

SCM Aktivitäten

- “Configuration item identification “
  - Bestimmung der per SCM zu verwaltenden Artefakte
- “Branch management”
  - Verwaltung verschiedener Versionen, die später integriert werden sollen
- “Variant management”
  - Verwaltung von Versionen die nebeneinander existieren sollen
- “Change management“
  - Erfassung, Genemigung und Nachverfolgung von Änderungswünschen
- “Promotion”
  - Erzeugung von Versionen für andere Entwickler
- “Release”
  - Erzeugung von Versionen für Kunden bzw. Benutzer

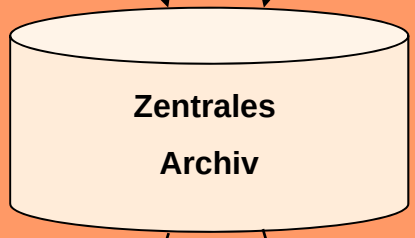
SCM Grundbegriffe

- Arbeitskopie
  - Unter Kontrolle eines Programmierers
  - Nur für ihn sichtbar



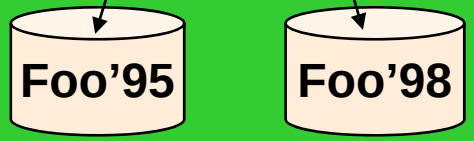
Promotion

- Repository
  - Zentrales Verzeichnis aller “promoteten” Versionen
  - Von allen Teammitgliedern benutzt



Release

- Produkt
  - Externe, veröffentlichte Version



Grundlegende Ansätze

Vault Model

Standard Repository

Virtual File System

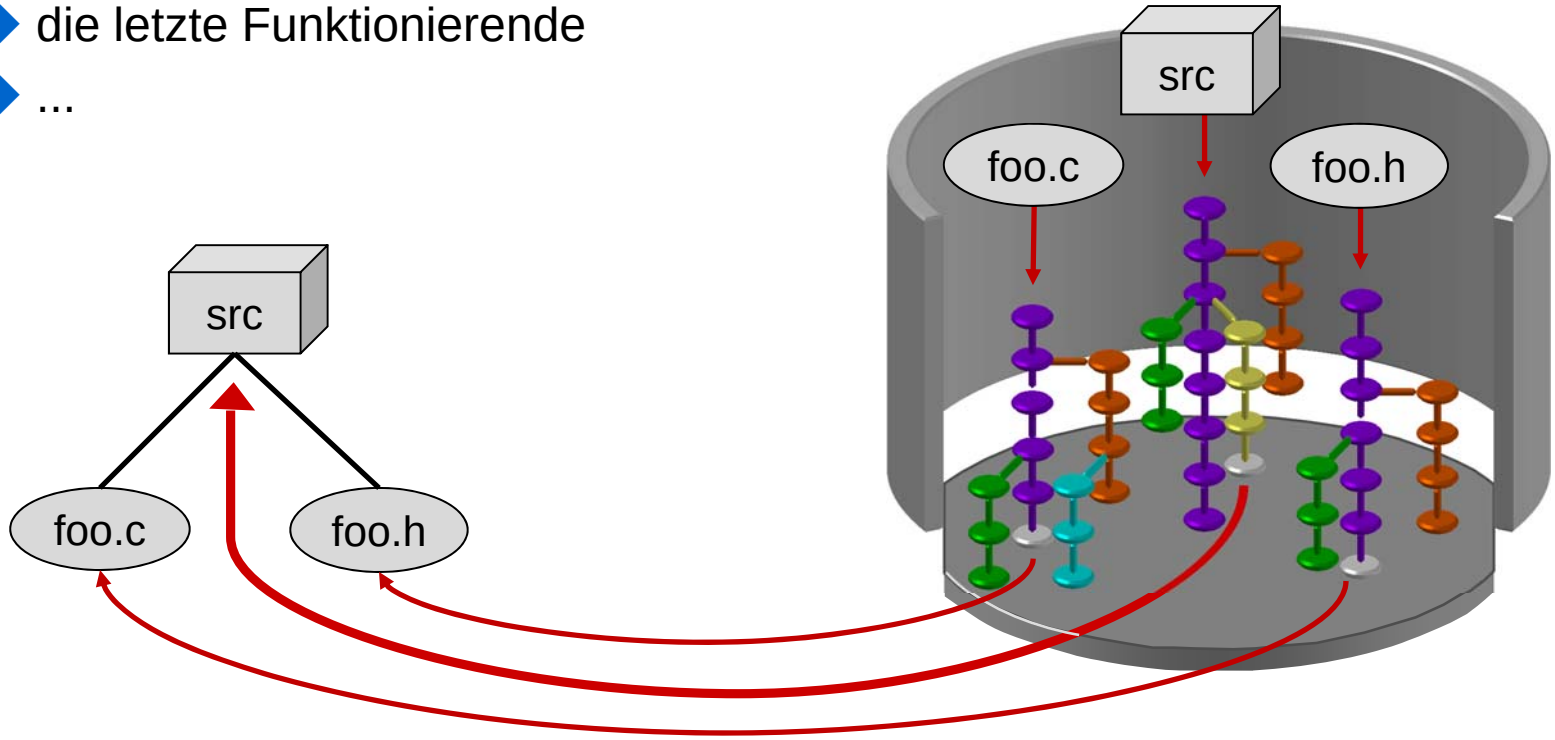
Arbeitsumgebung und Repository

Arbeitsumgebung ("Sandbox")

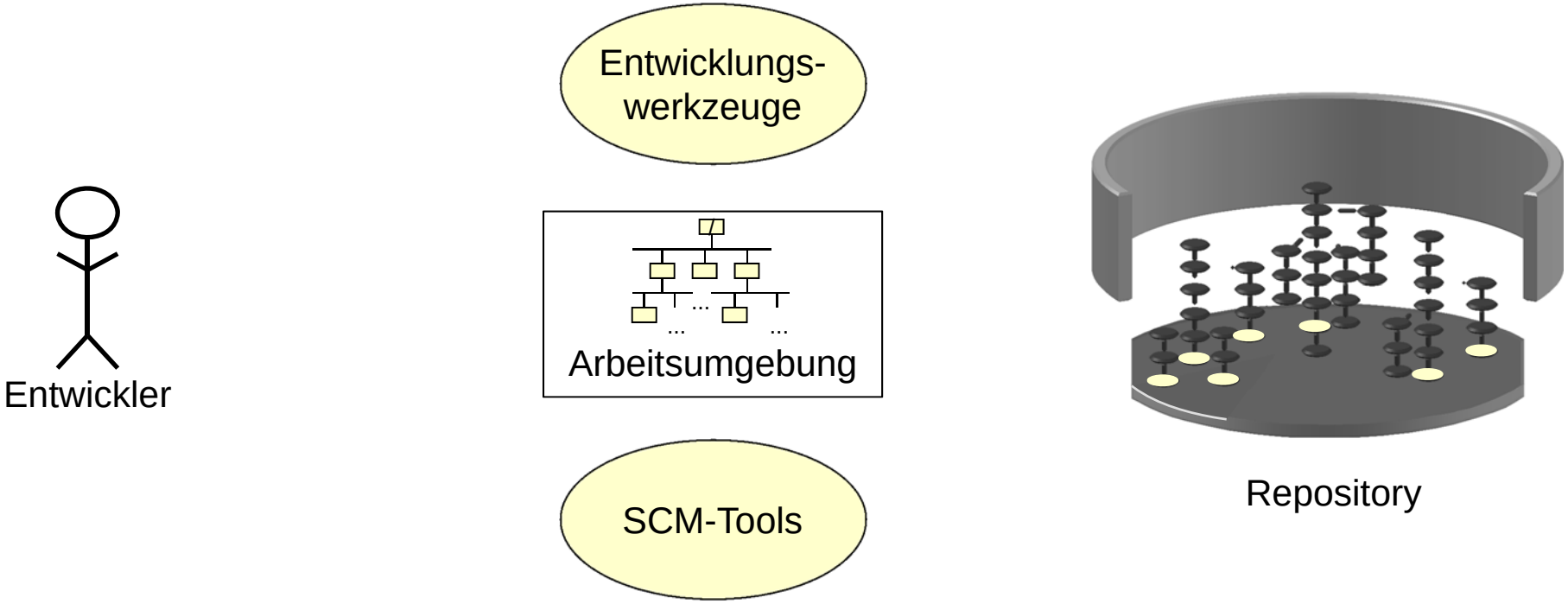
- Enthält nur die aktuell relevante Version eines jeden Artefaktes aus dem Repository
  - die Aktuellste
  - die letzte Funktionierende
  - ...

Repository

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

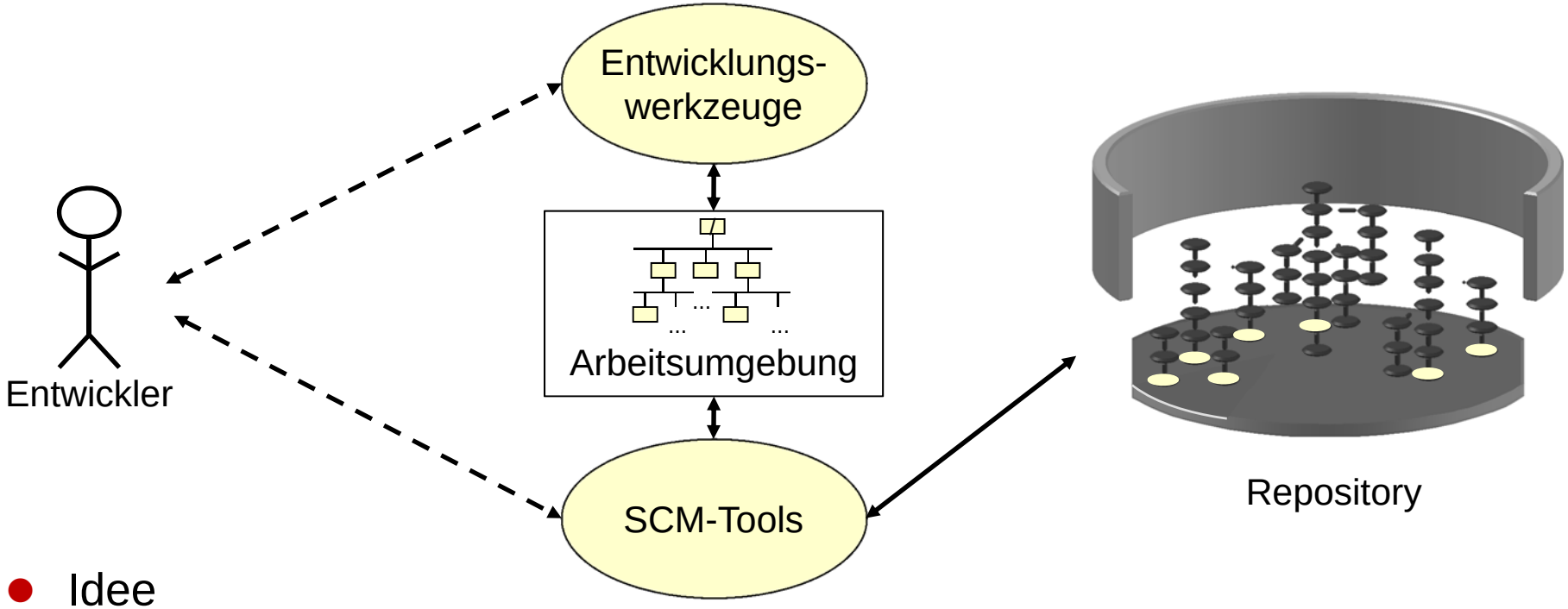


Allgemeines Szenario



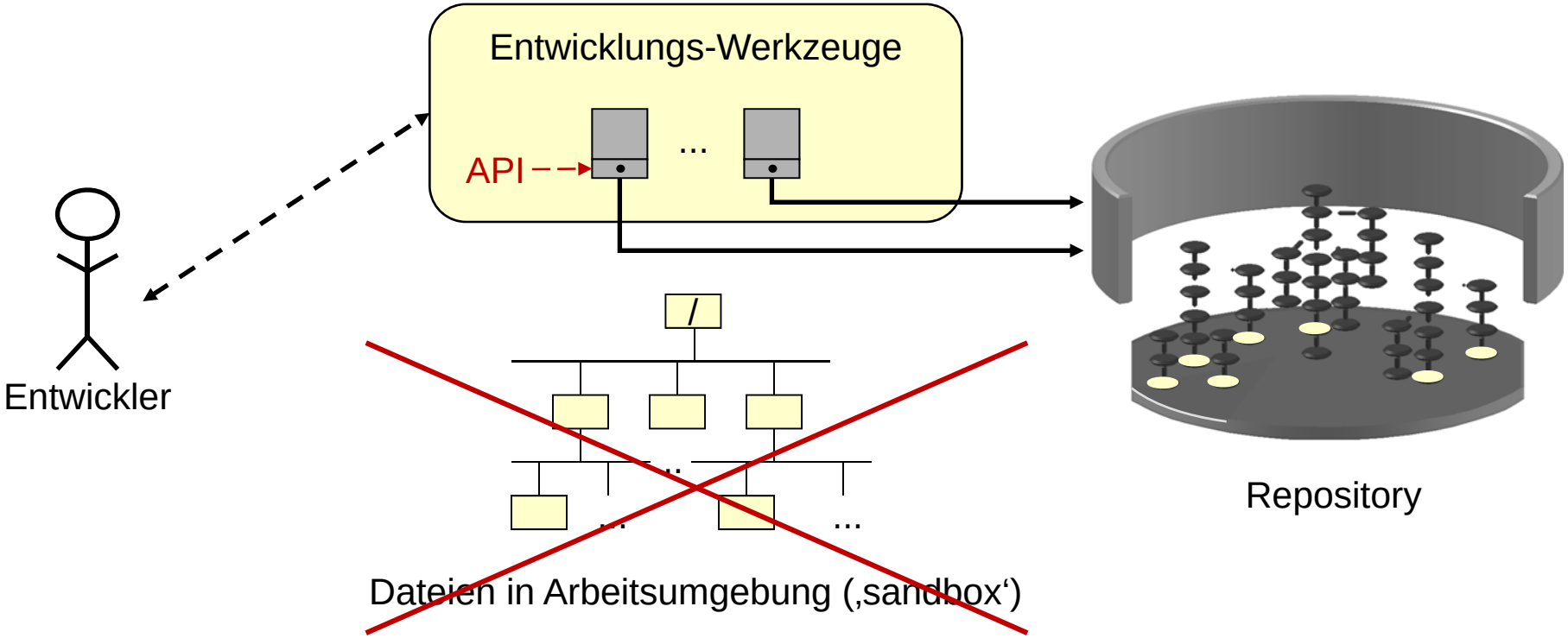
- Frage: Wie interagieren obige Beteiligte?
  - Muss der Entwickler selbst seine Arbeitsumgebungen verwalten oder kann das ein SCM-Tool (größtenteils) automatisch?
  - Ist immer eine lokale Kopie in einer Arbeitsumgebung erforderlich oder können Entwicklungswerkzeuge direkt auf das Repository zugreifen?

SCM-Ansätze: „Vault Model“



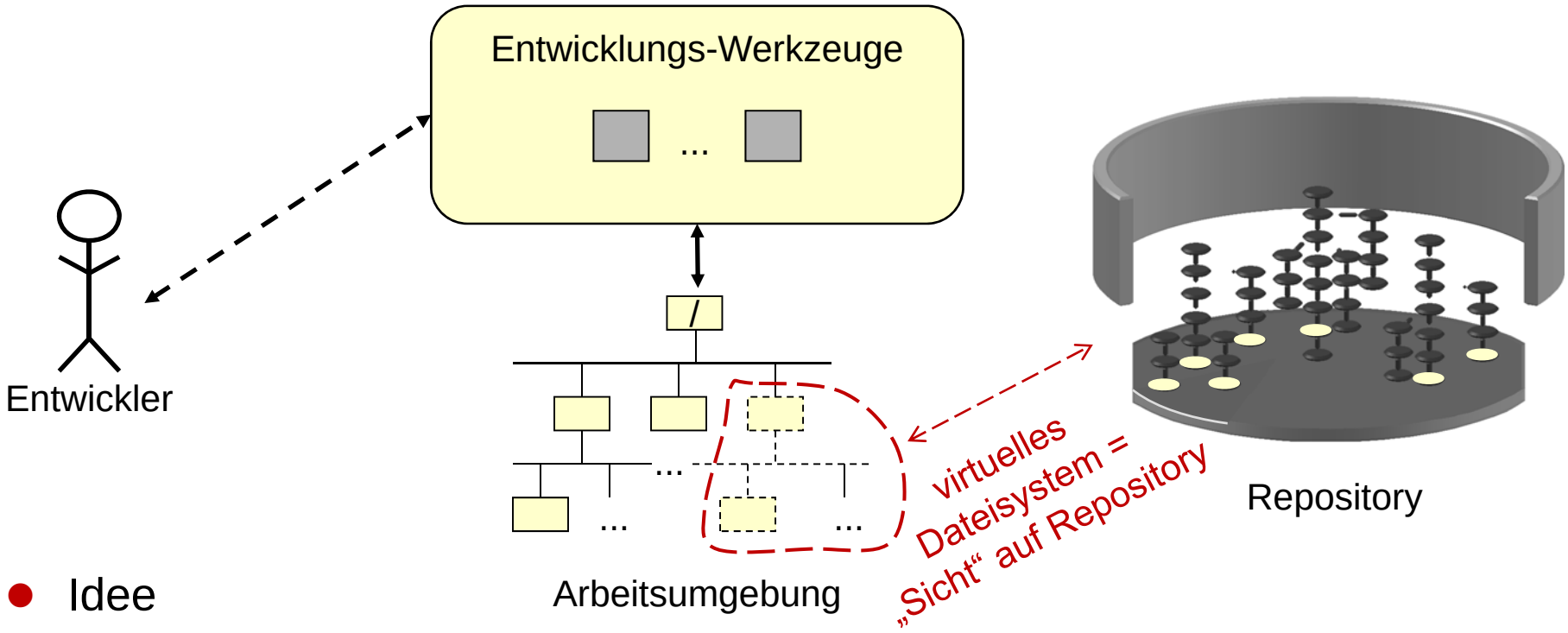
- Idee
  - Dateien kopieren: Arbeitsumgebung ← → Repository
- Probleme
  - Evtl. viele private Kopien in Arbeitsumgebungen außerhalb des Repository
  - Entwickler arbeitet evtl. auf veralteten Dateien
  - Möglicher Datenverlust durch gleichzeitige oder abgebrochene Updates

SCM-Ansätze: "Standard Repository API"



- Idee
  - Alle Entwicklungswerkzeuge greifen direkt auf das Repository zu, über eine standardisierte Schnittstelle
- Problem
  - Alle Werkzeuge müssen angepasst werden -- bei jeder Änderung des Standards!

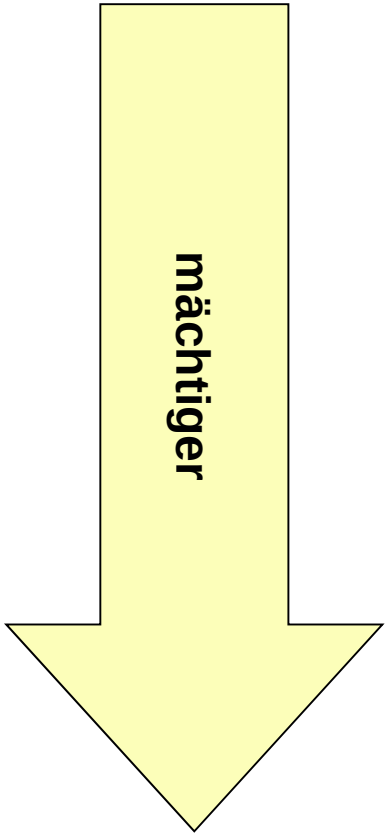
SCM-Ansätze: "Virtual File System" (VFS)



- 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 Binärdatein
  - ⇨ Komfortable graphische Benutzeroberflächen
- ClearCase (high-end, kommerziell)
  - ⇨ Virtuelles Dateisystem basierend auf Datenbank
  - ⇨ Alle Eigenschaften von SVN plus...
  - ⇨ Dynamische Sichten auf Repository
  - ⇨ „Derived Object Sharing“
  - ⇨ Multiple Server, Replikation
  - ⇨ Prozessmodelierung



Arbeiten mit SVN (und CVS)

Check-in und Check-out

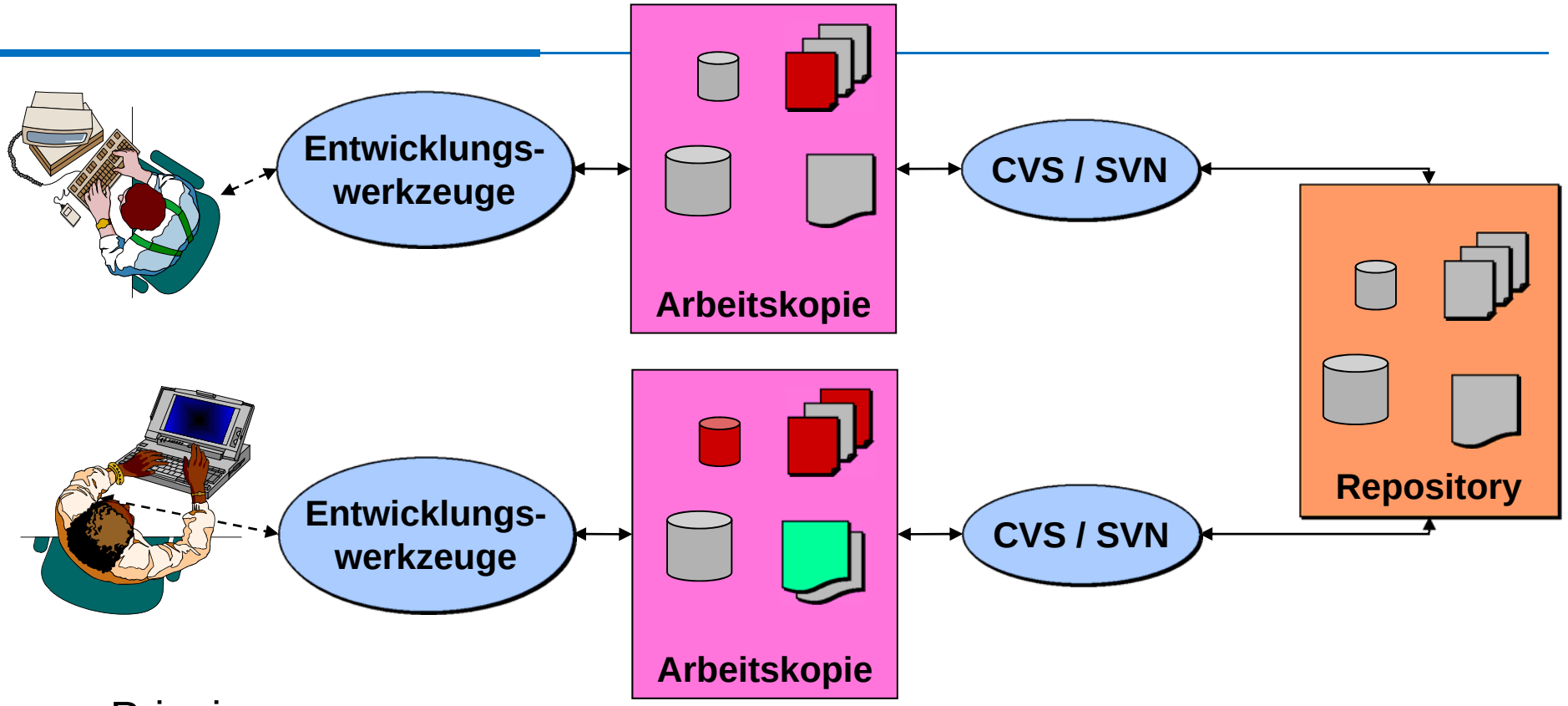
Commit und Update

Synchronisierung

Konfliktbeurteilung durch Dateivergleich

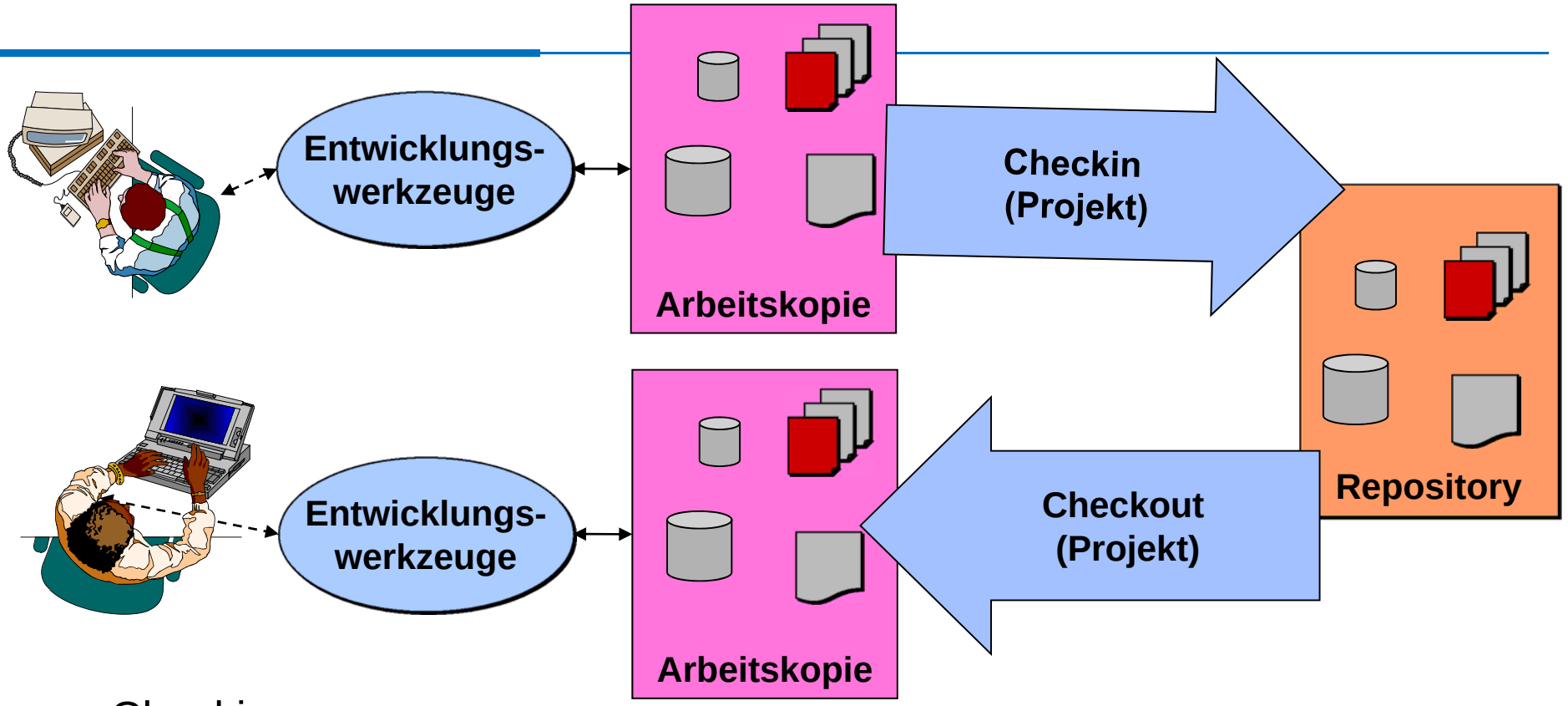
Manuelle Konfliktauflösung

CVS und SVN: “Vault Model”



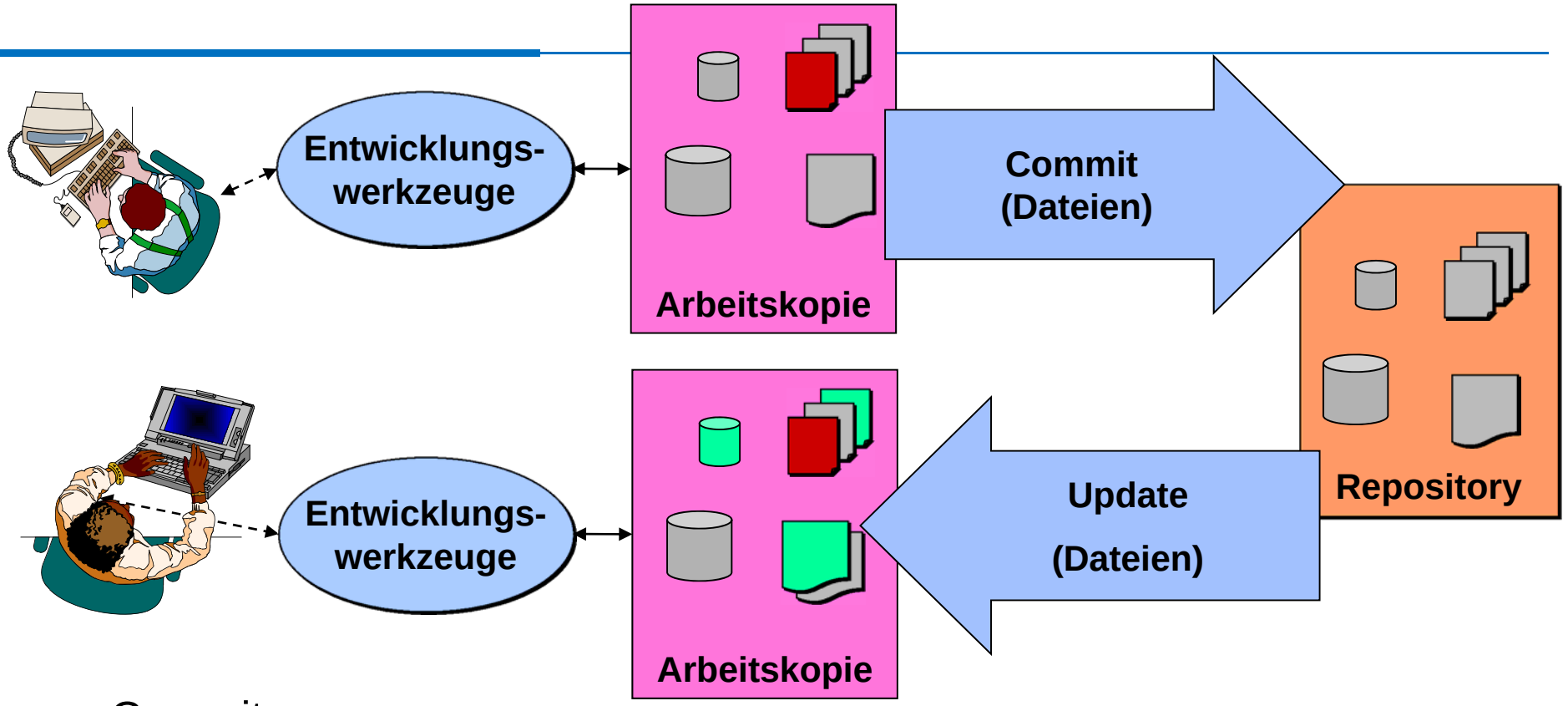
- Prinzip
  - Ein zentrales Sammelbecken („Repository“) aller relevanter Dateien
    - ⇨ Nur „offizielle“ Versionen
  - Viele private Arbeitsumgebungen („Sandbox“) mit Kopien von Dateien
    - ⇨ Auch temporäre, inkonsistente, unfertige, ... Versionen

CVS und SVN: Checkin und Checkout



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

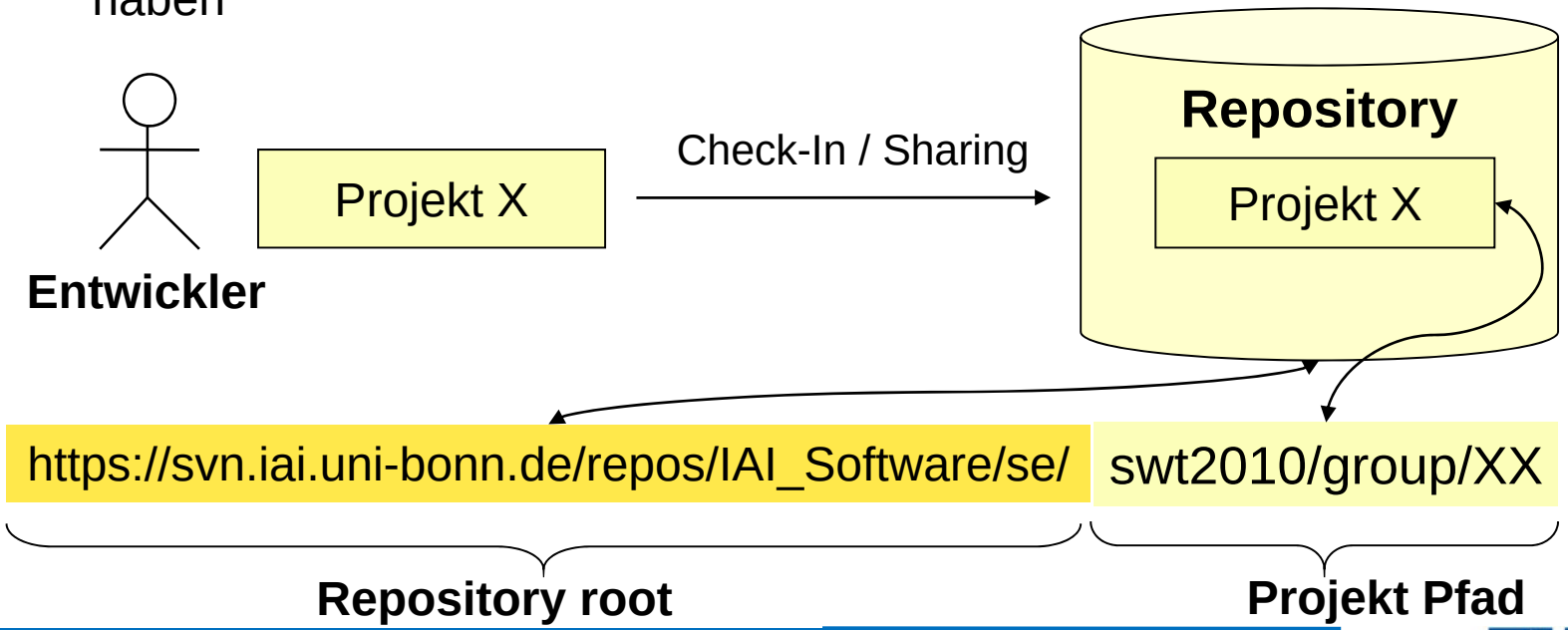
CVS und SVN: Commit und Update



- Commit
  - Transferiert vom Programmierer geänderte Dateien in das Repository
- Update
  - Transferiert geänderte Dateien vom Repository in die Arbeitskopie

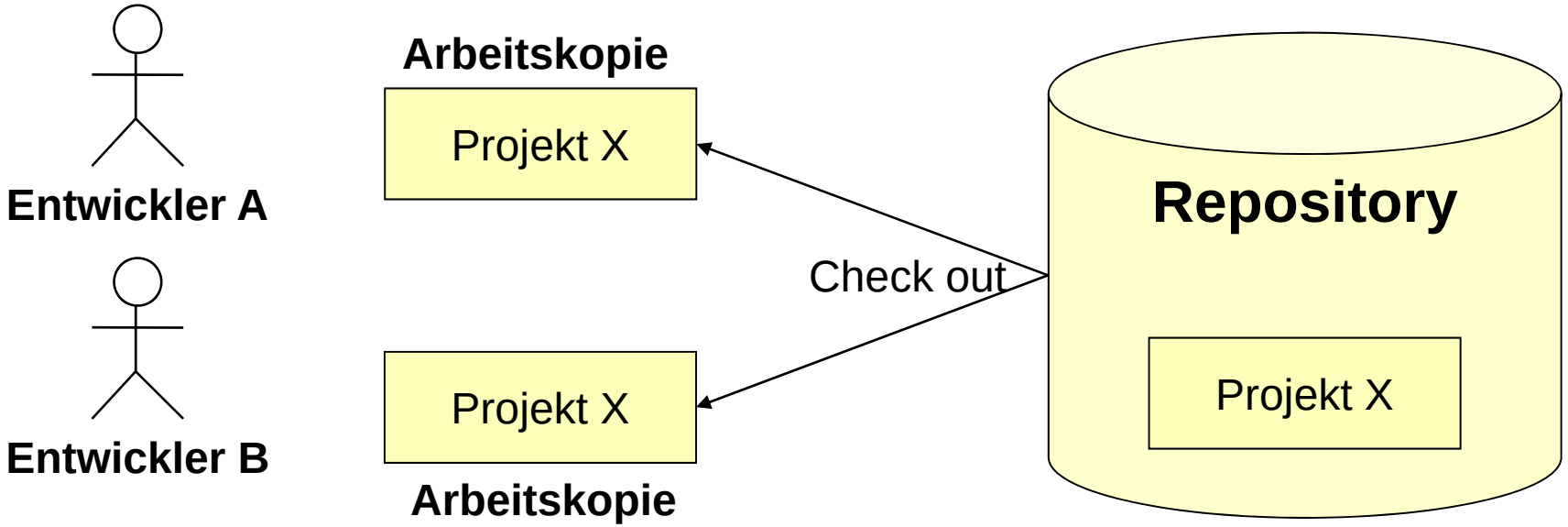
Check-In: Projekt unter Versionskontrolle stellen

- 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



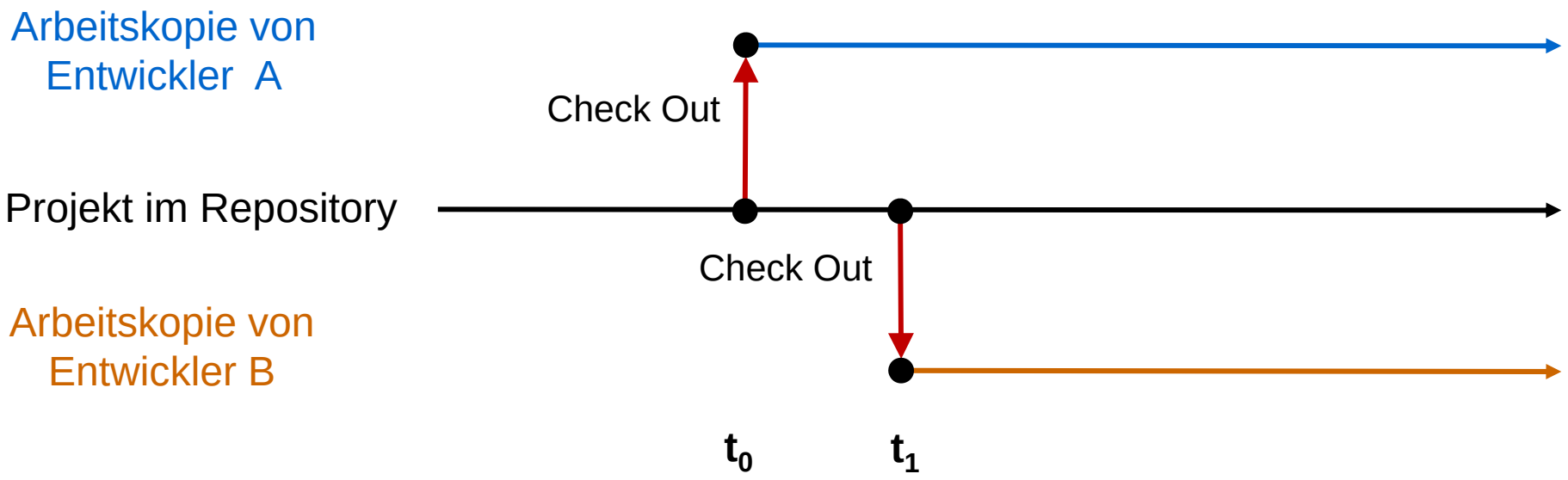
Check-Out: Initialer Projektdownload

- Check-out eines Projektes
  - Entwickler bekommen eine lokale Arbeitskopie
  - Von jetzt an können sie an dem Projekt arbeiten
- Auschecken wird nur einmal pro Projekt gemacht!
  - Neue Version aus Repository holen → siehe „Update“



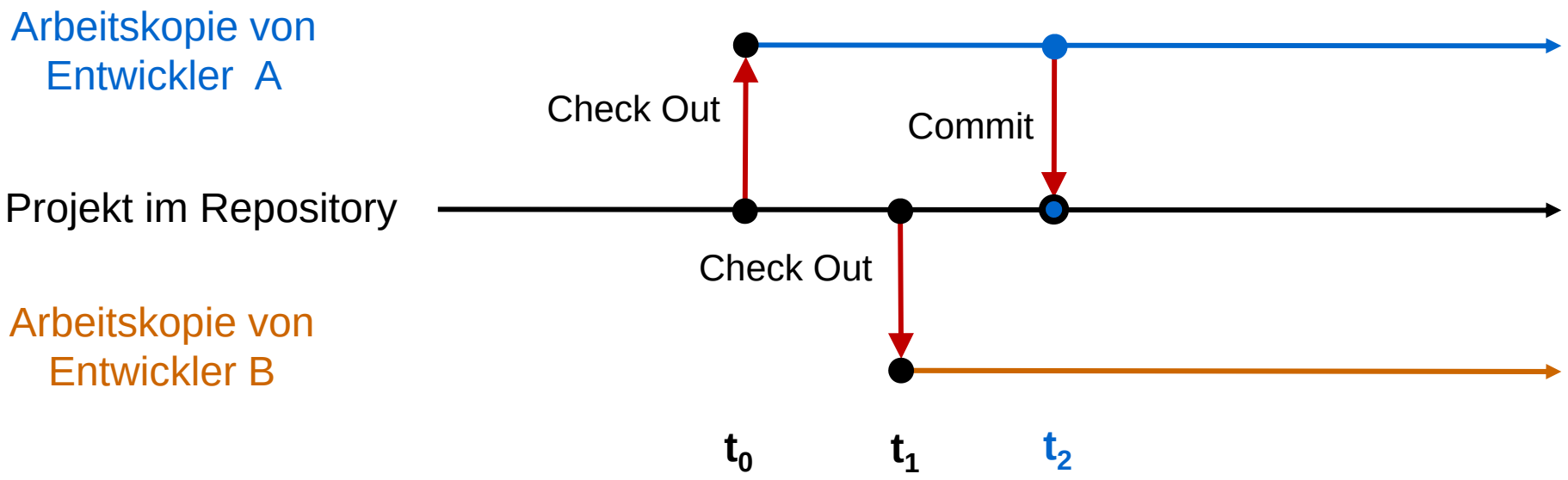
Check-Out: Initialer Projektdownload

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



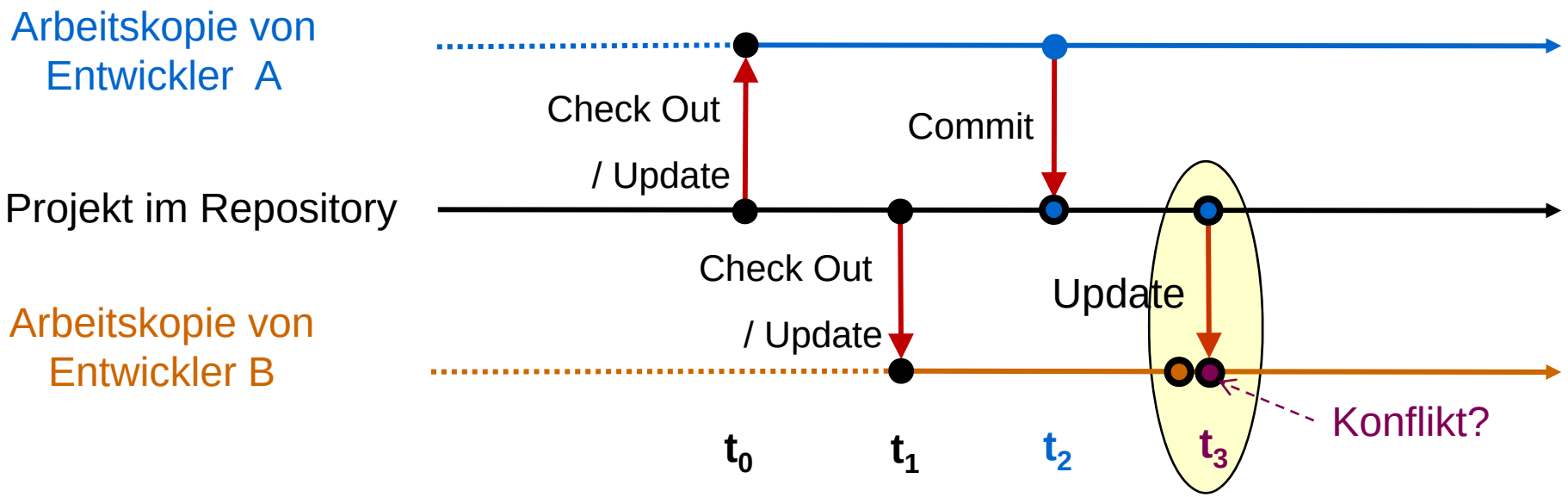
Commit: Änderungen in das Repository übertragen

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



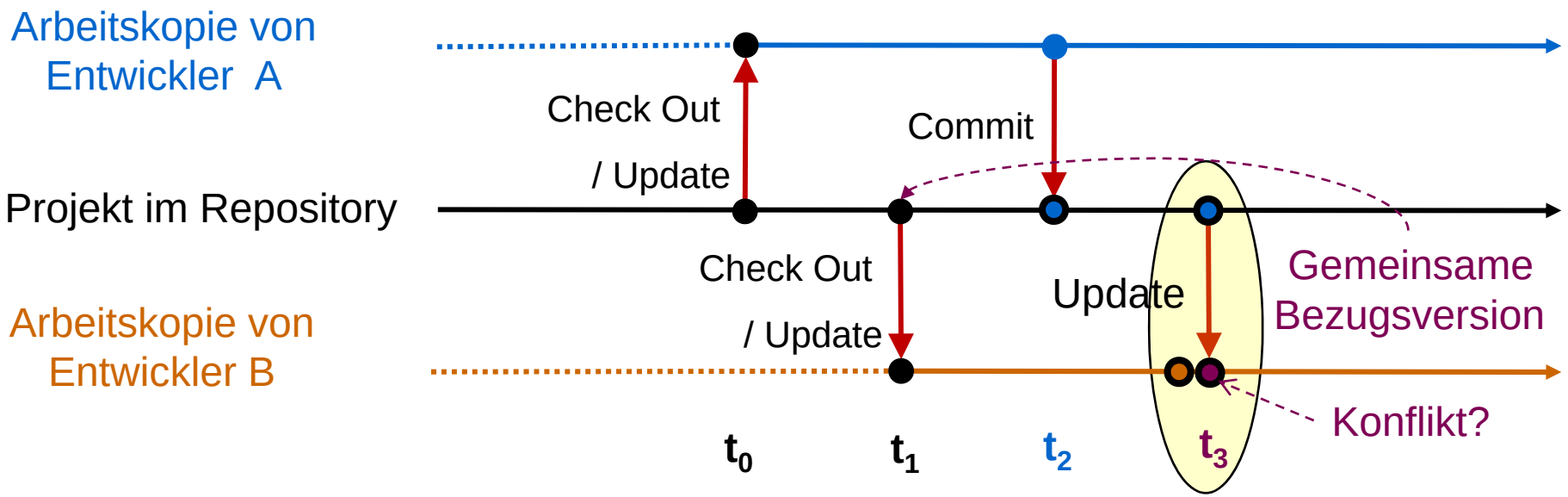
Update: Neueste Änderungen vom Repository übernehmen

- 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
  - Woher weiß der Entwickler, welche Änderungen er übernehmen soll?
  - Besser: Zuerst „Synchronize“ benutzen!



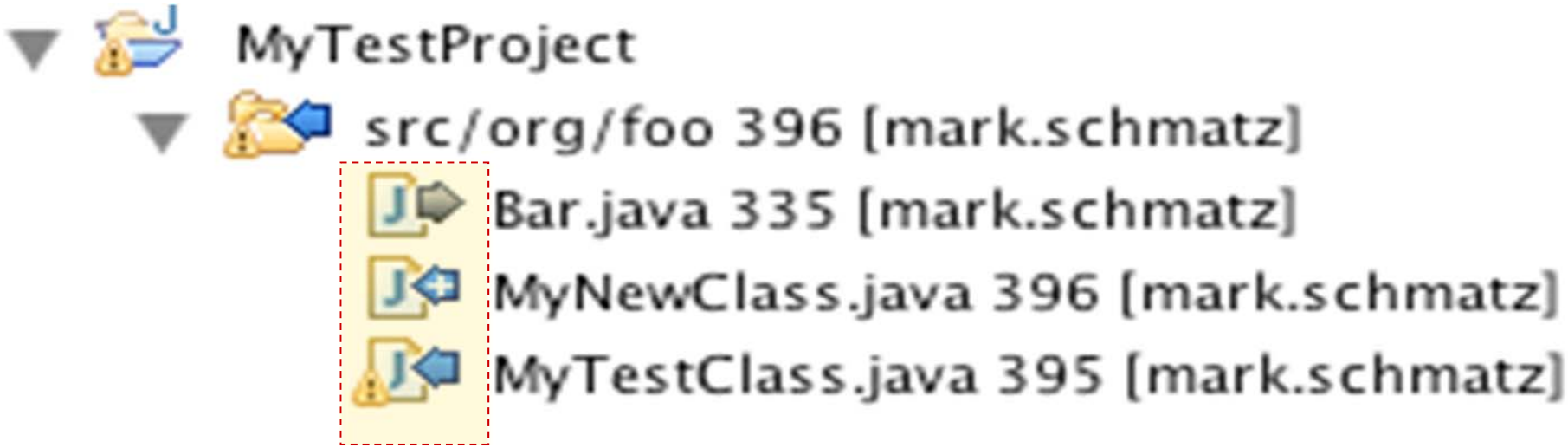
Synchronisation: Versionsvergleich

- 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 durch (eventuell selektiv)

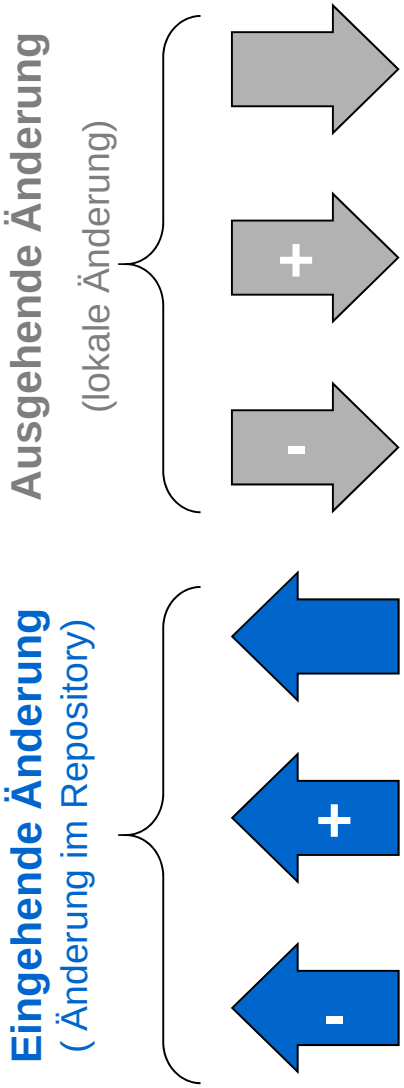


“Synchronize View”: Gesamtübersicht

- Automatisierte Gesamtübersicht
  - Vergleich auf Projektebene: Alle Dateien des Projektes (oder des selektierten Ordners) werden verglichen
  - Zeigt pro Datei ein- und ausgehende Änderungen sowie Konflikte durch entsprechende Symbole



Symbole im „Synchronize View“



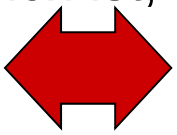
- Lokale Datei ist neuer als ihre Version im Repository
  → Überschreibe Version im Repository mit lokaler Version
- Neue lokale Datei existiert nicht im Repository
  → Füge Datei zum Repository hinzu
- Datei aus Repository wurde lokal gelöscht
  → Datei aus (nächster Version in) dem Repository löschen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

Bedeutung der Symbole im „Synchronize View“ (Fortsezung)

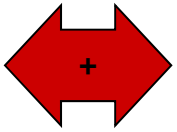
- Die vorherige Folie stellt die Fälle dar, wenn eine Datei gegenüber dem Stand des letzten Abgleichs mit dem Repository nur auf einer Seite verändert wurde
  - Änderung nur lokal → Übernahme ins Repository (ausgehende Änderung)
  - Änderung nur in Repository → Übernahme in lokale Kopie (eingehende Änderung)

- Wenn eine Datei auf beiden Seiten neuer als der Stand beim letzten Abgleich ist, wird ein Konflikt gemeldet:

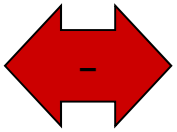
Konflikt



- Dateiinhalt wurde lokal und im Repository verändert
  ↔ Manuelle Konfliktlösung notwendig



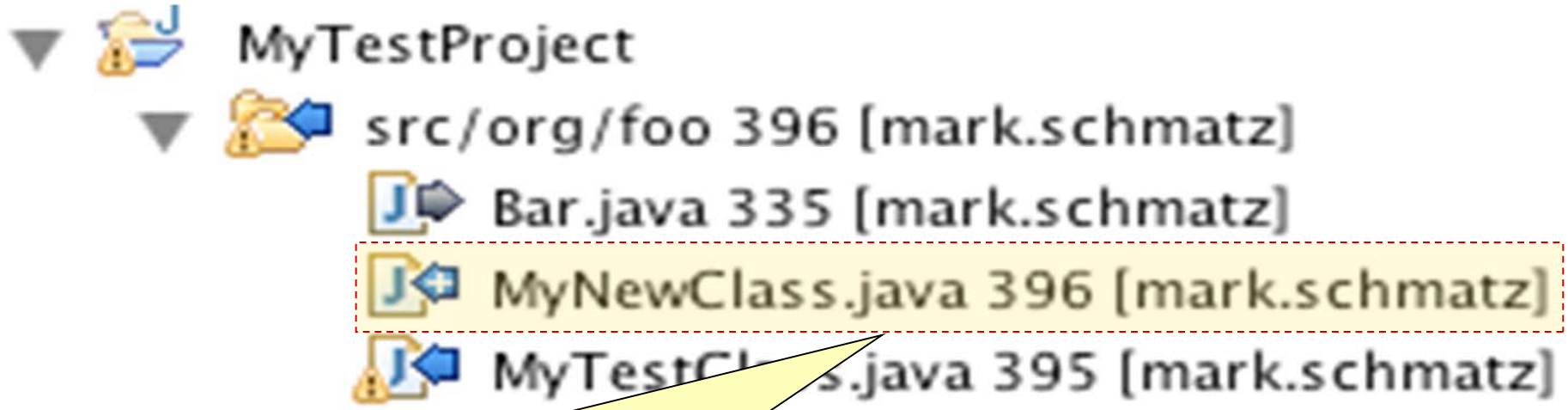
- Lokale veränderte Datei wurde im Repository gelöscht
  ↔ Manuelle Konfliktlösung notwendig



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

Synchronisieren: Konfliktauflösung

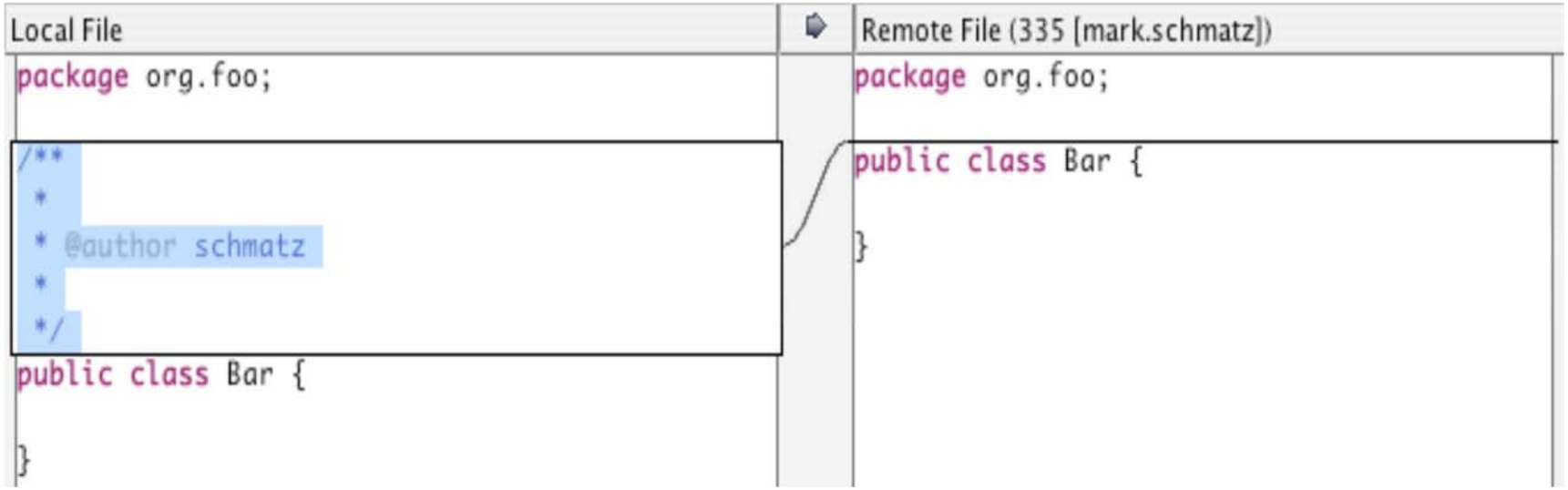
- Konfliktauflösung erfordert Inhaltsvergleich der Dateien
  - Angezeigt in “Side-by-side View” / “Compare Editor” von Eclipse
  - Zeigt Versionsvergleich von Dateien („diff“) übersichtlich an



Beginne Detailvergleich durch Doppelklick auf die entsprechende Datei ...

Synchronisieren: Inhaltsvergleich im „Compare“ Fenster

- Die lokale Arbeitskopie wird mit der Repository Version verglichen
  - Hier ein Fall ohne Konflikte: nur ausgehende Änderung



- SVN- / CVS-Plugin für Eclipse hilft beim Versionsvergleich
  - „Compare View“

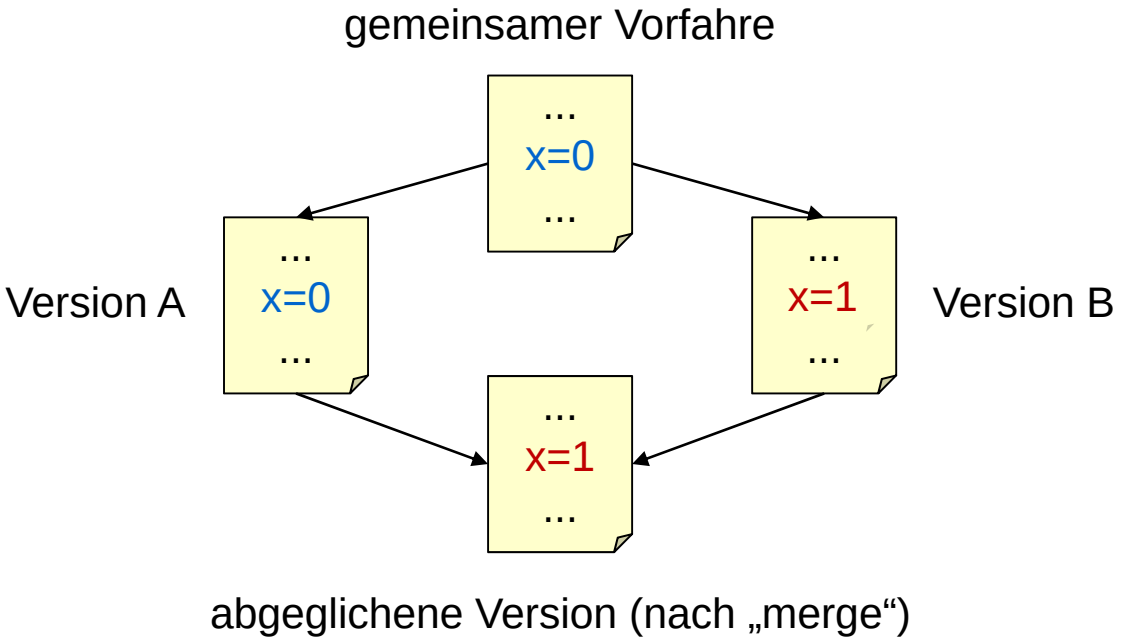
Synchronisieren: Inhaltsvergleich im „Compare Editor“ Fenster

- Hier die Anzeige eines Konfliktes

- 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 mit Hilfe der „Common Ancestor Pane“

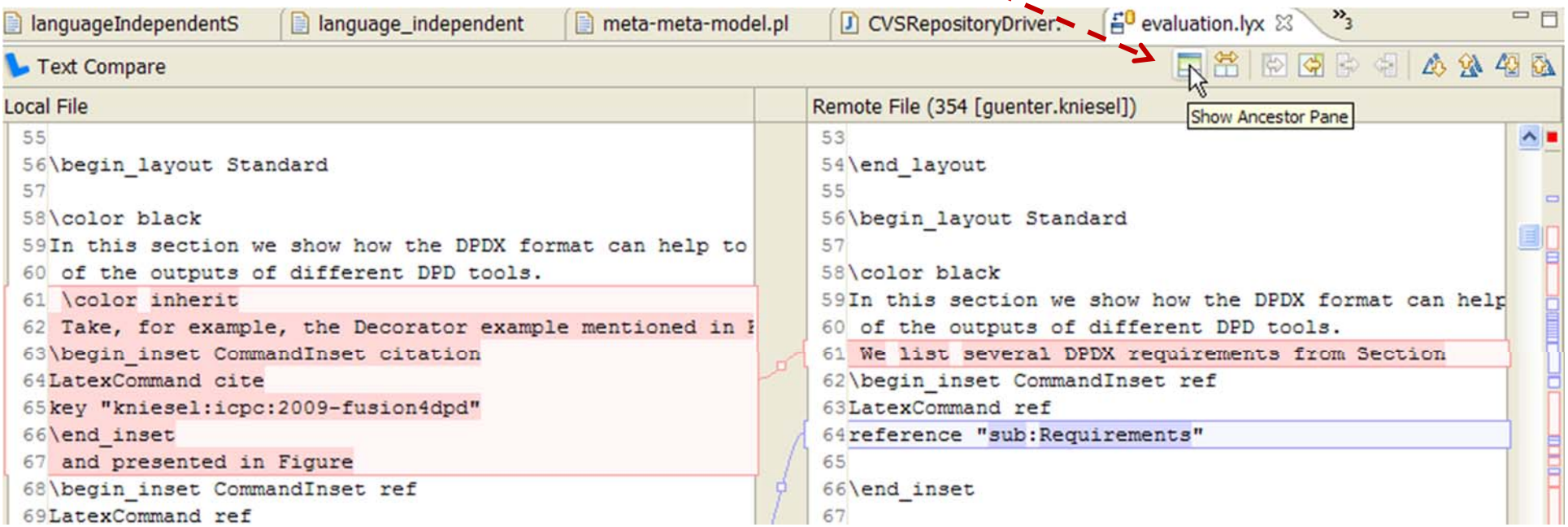
- Anhand des gemeinsamen Vorfahren feststellen was sich geändert hat
- Die Änderung übernehmen



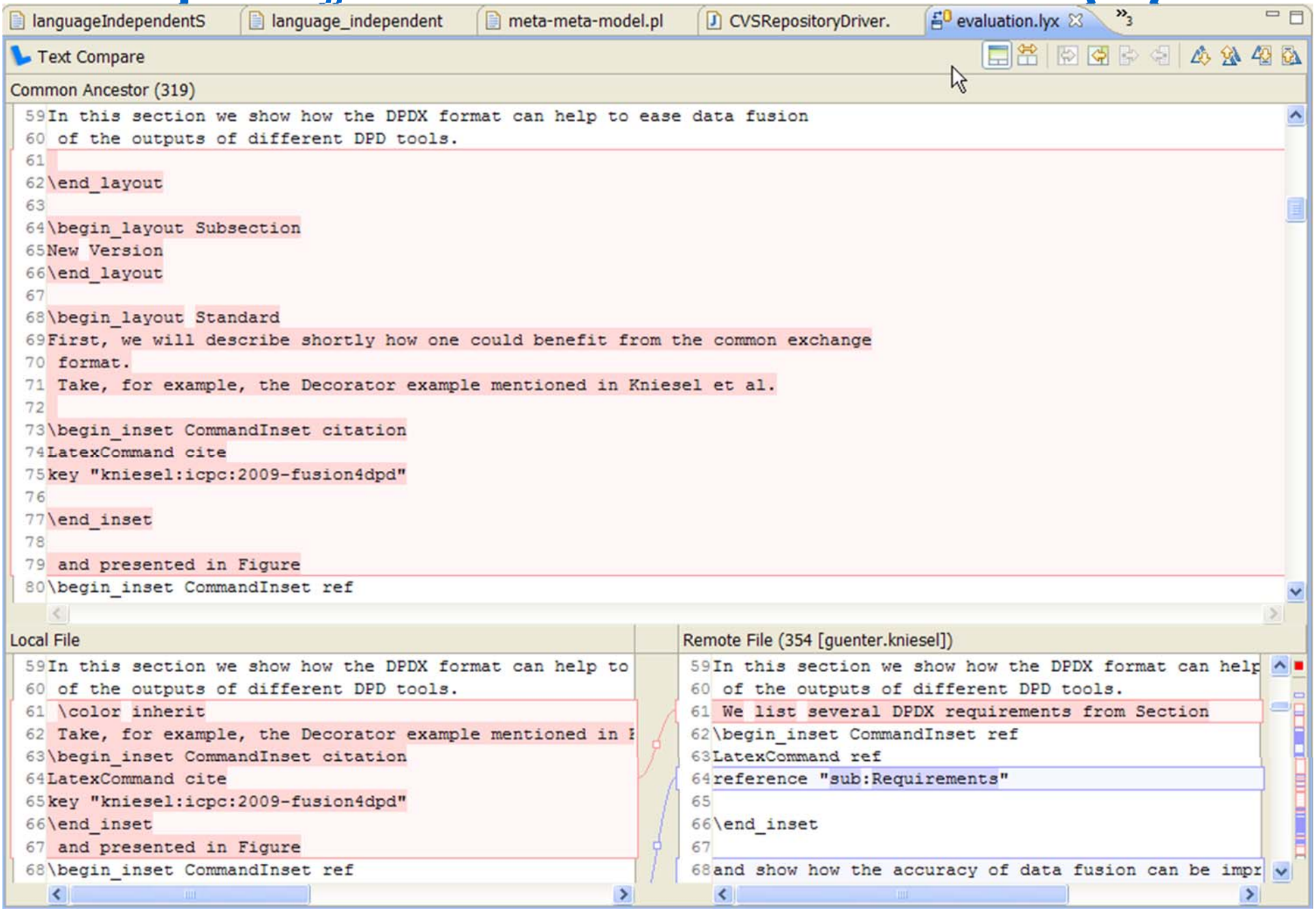
- Mit SVN ist dies ein manueller Prozess: Sie selbst entscheiden
  - für jede Datei mit Konflikten
  - für jeden Konflikt in der Datei

Beispiel: „Common Ancestor Pane“ (1)

- Szenario: Gemeinsam eine Ausarbeitung / Veröffentlichung schreiben
  - Konflikte in Datei “evaluation.lyx”
- Frage: Welcher Stand ist der neueste?
  - Erster Schritt: “Show Ancestor Pane”

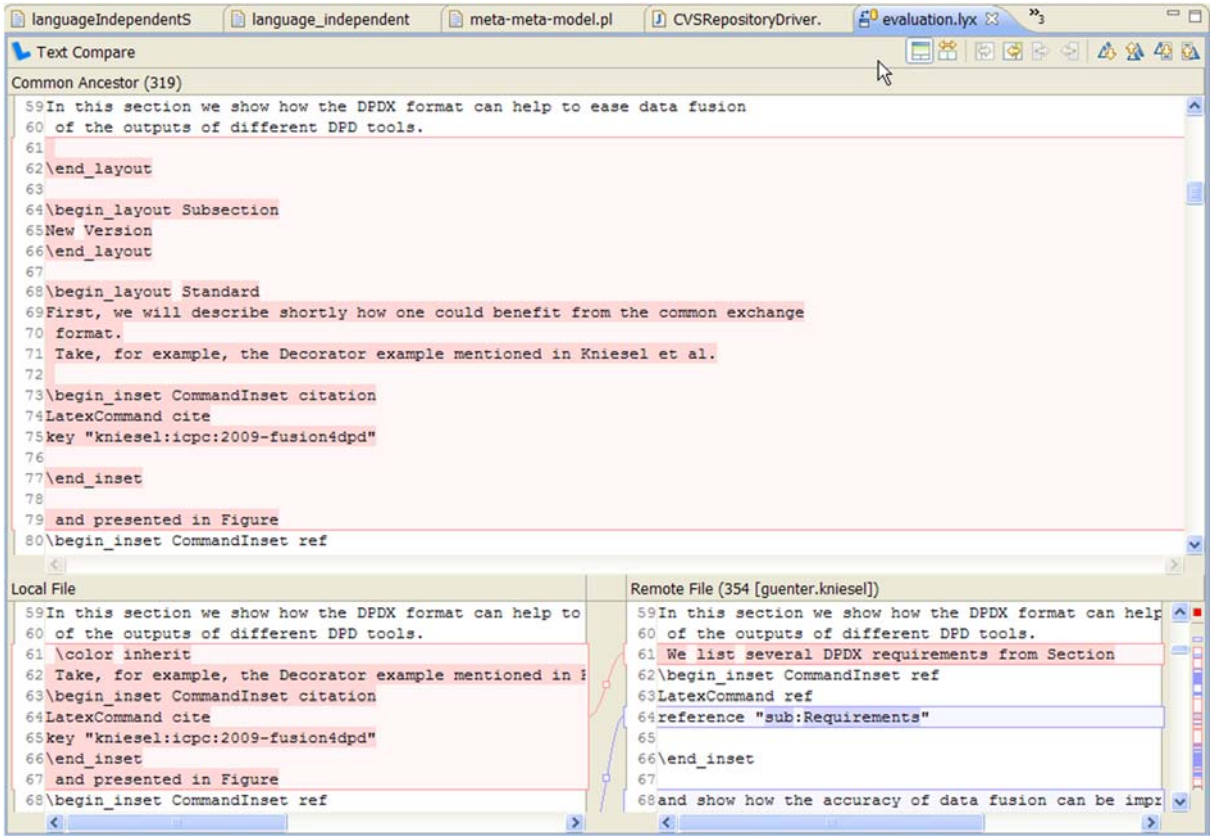


Beispiel: „Common Ancestor Pane“ (2)



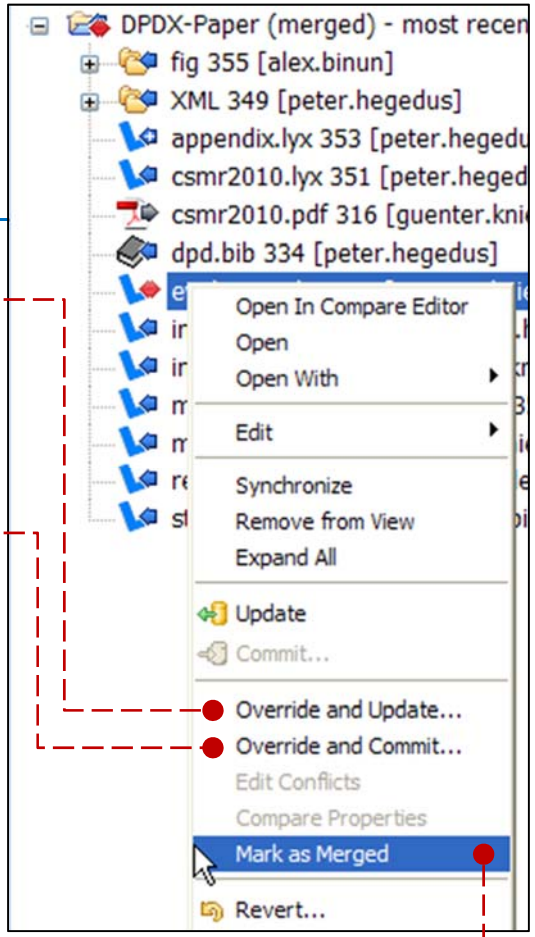
Beispiel: „Common Ancestor Pane“ (3)

- Antwort: Keiner
  - Hier wurde tatsächlich an der gleichen Stelle parallel geändert!
  - ➔ Absprache mit dem Kollegen / Co-Author erforderlich!

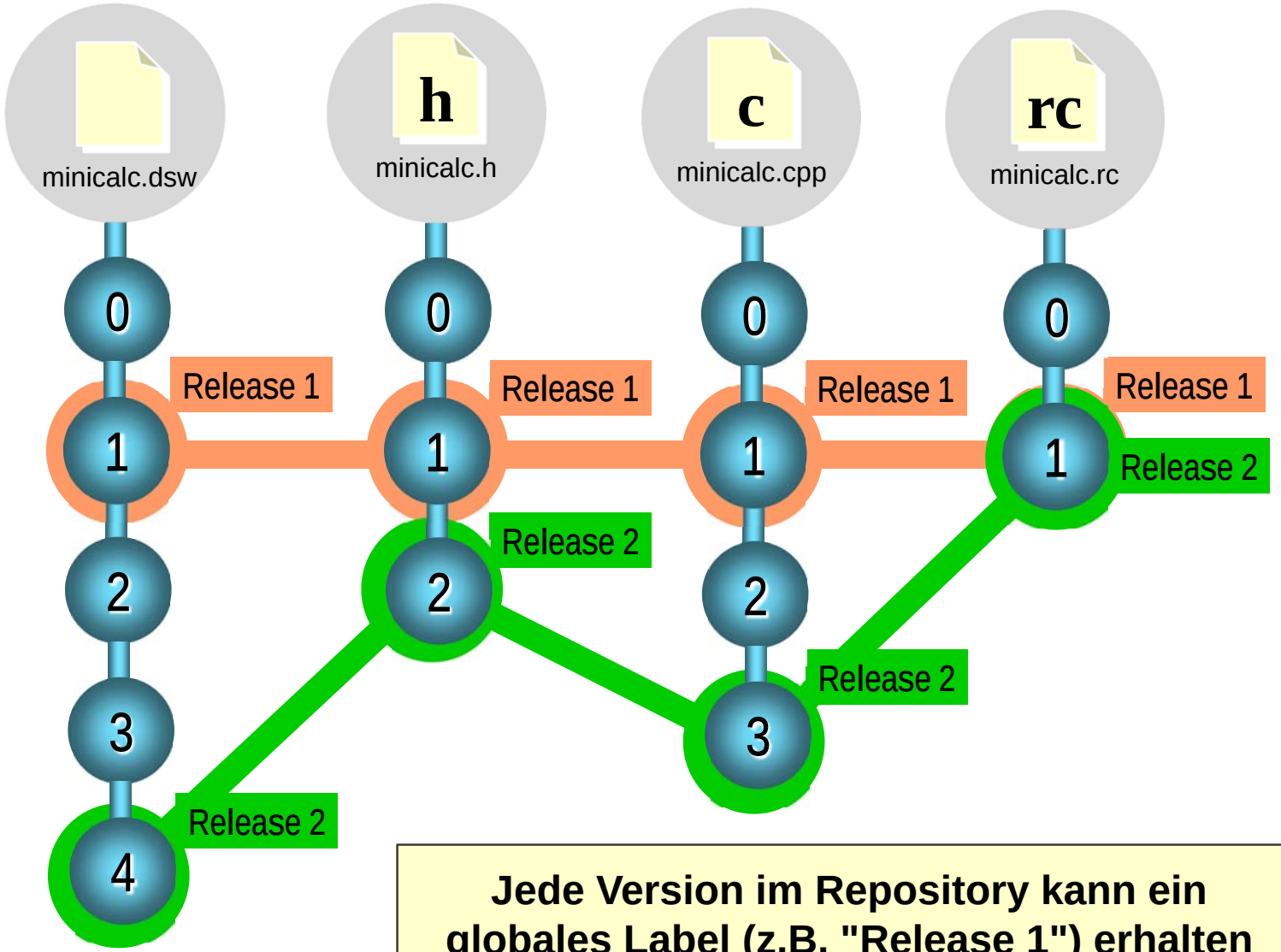


Synchronisieren: Aktionen zur Konfliktauflösung

- „Override and Update“ (Menupunkt)
  - Überschreibt lokale Version mit Repository-Inhalt
- „Override and Commit“ (Menupunkt)
  - Überschreibt Repository-Version mit lokaler Version
  - Nie ohne vorherige Kommunikation mit dem Autor der Repository-Version!
- „Merging“ (Manueller Vorgang)
  - Selektive Übernahme einzelner Änderungen der Datei im „Compare Editor“
  - Nur Übernahme von Repository-Stand in lokale Version!
  - Anschließend „Mark as Merged“ auf lokaler Version (Menupunkt)
    - ⇨ Lokale Version wird nun als aktueller als die aus dem Repository angesehen
    - ⇨ Bei einem „Commit“ würden nun die nicht übernommenen Repository-Inhalte überschrieben!



Tagging (= Baselining)



Jede Version im Repository kann ein globales Label (z.B. "Release 1") erhalten

Vergleich von Subversion (SVN) und CVS

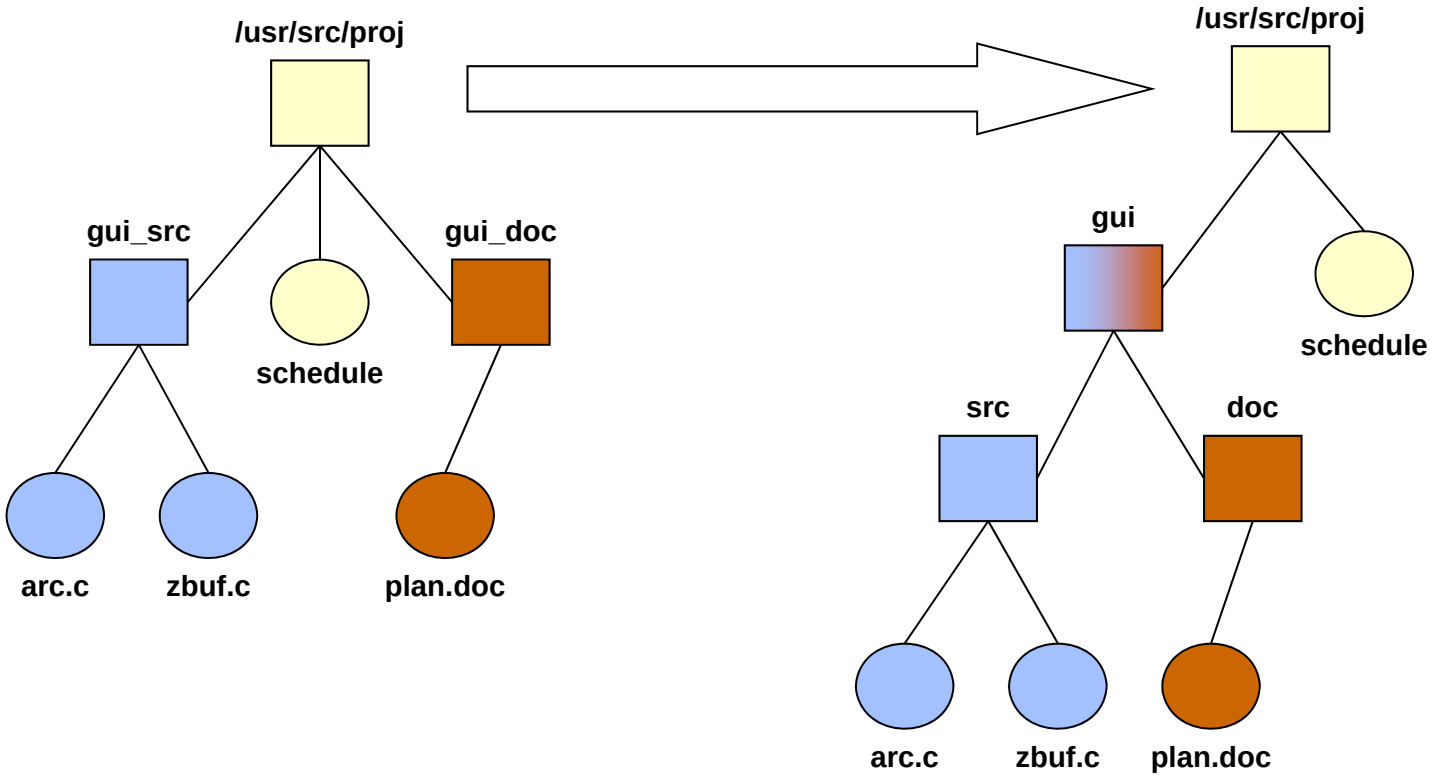
Versionierung von Verzeichnissen

Globale Commit-Nummerierung

Atomare Commit-Aktionen

Subversion kann mehr als CVS: 1. 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: 1. Versionierung von Verzeichnissen

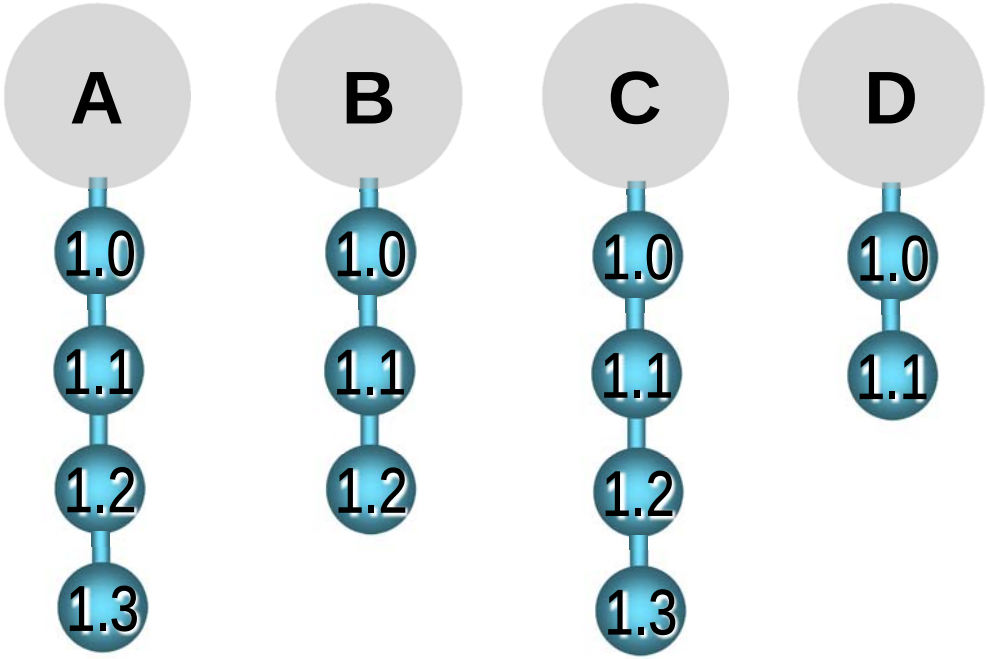
- Es ist in Subversion möglich Dateien und Verzeichnisse zu löschen, umzubenennen und zu verschieben

Warnung

- 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!
- Man sollte tunlichst vermeiden, versehentlich die „.svn“ Ordner innerhalb der Arbeitskopie zu verändern!
  - Sie enthalten die Meta-Informationen für SVN über die zwischen den Abgleichen mit dem Repository geschehenen Änderungen auf Dateiebene

Subversion kann mehr als CVS: 2. Globale Versions-Nummerierung

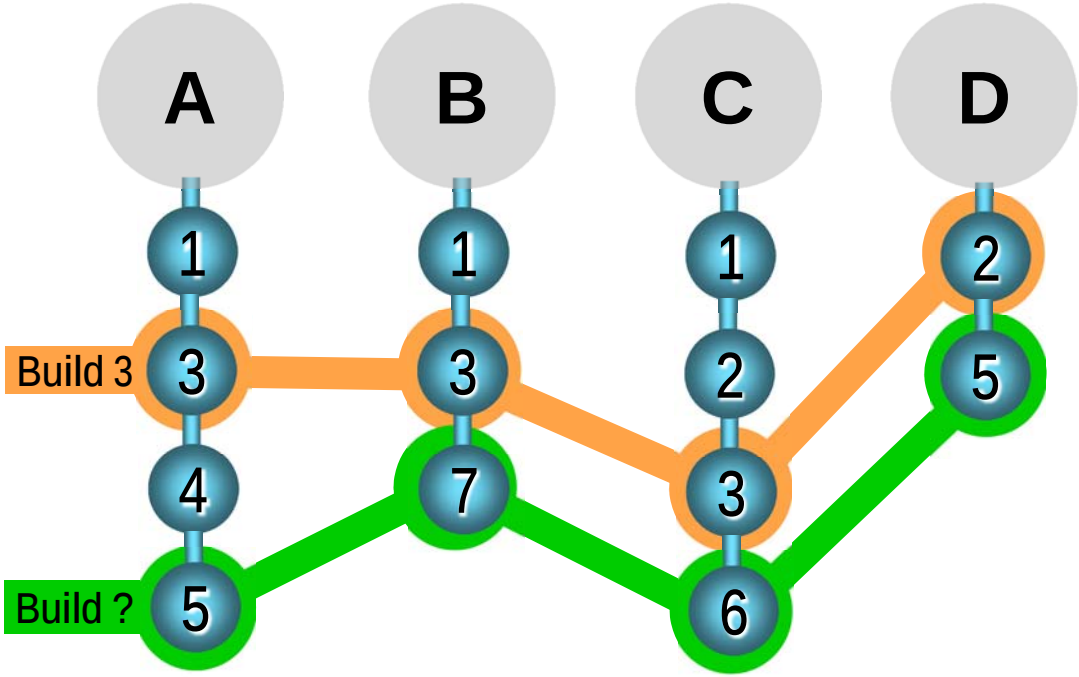
- CVS weist jeder Datei ihre eigenen Versions-Nummern zu



- Problem: Es ist nicht einfach zusehen welche Versionen verschiedener Dateien zusammengehören
  - „Gehört Version 1.0 von *Datei A* wirklich zu Version 1.1 von *Datei B*?“

Subversion kann mehr als CVS: 2. Globale Versions-Nummerierung

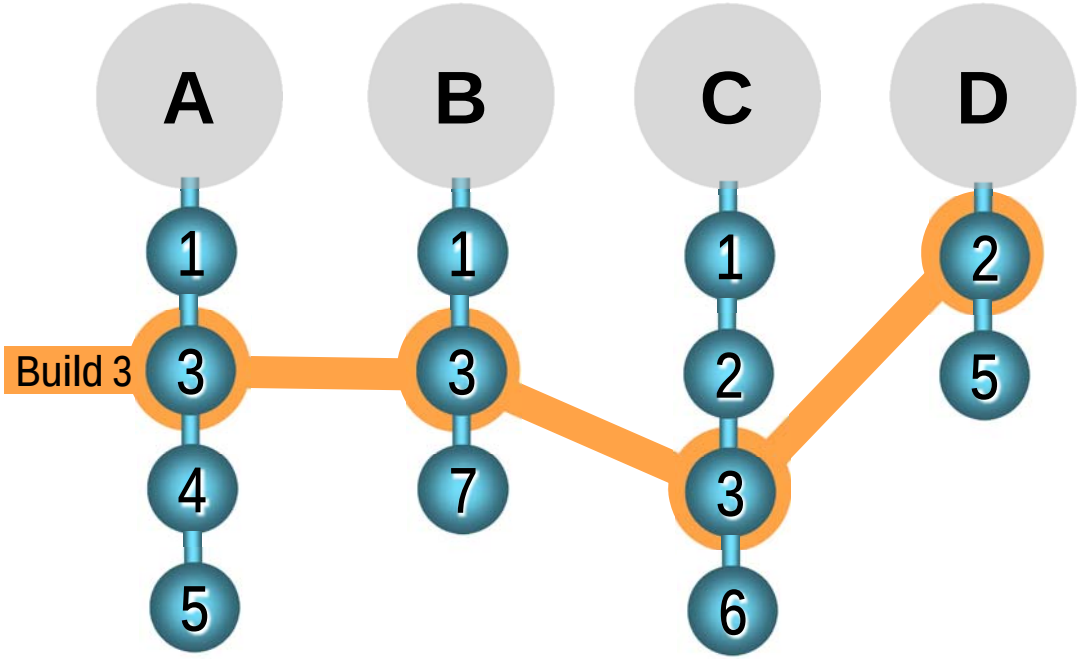
- SVN weist jeder Datei die bei einem Commit verändert wurde die selbe Versions-Nummer zu.



- 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: 2. Globale Versions-Nummerierung

- Ein „Roll back“ auf eine alte Version ist einfach, selbst ohne „Tags“
  - „Gehe zurück zu Version 3“



- Tagging sollte für wichtige Zwischenstände trotzdem genutzt werden
  - „Release 1.0 alpha“ ist leichter zu merken als Version 1093

Subversion kann mehr als CVS: 3. Atomic 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!

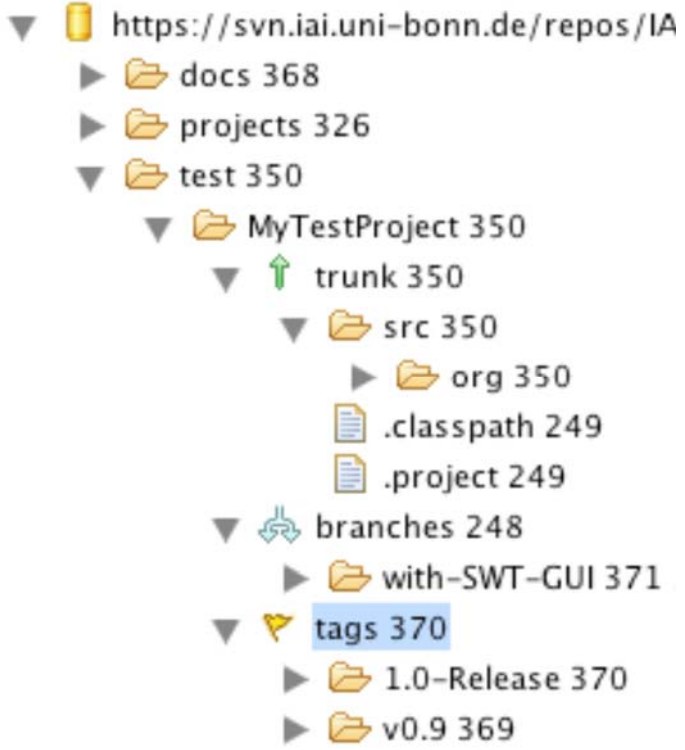
Struktur von Subversion-Repositories

Spezielle Ordner

Optionen

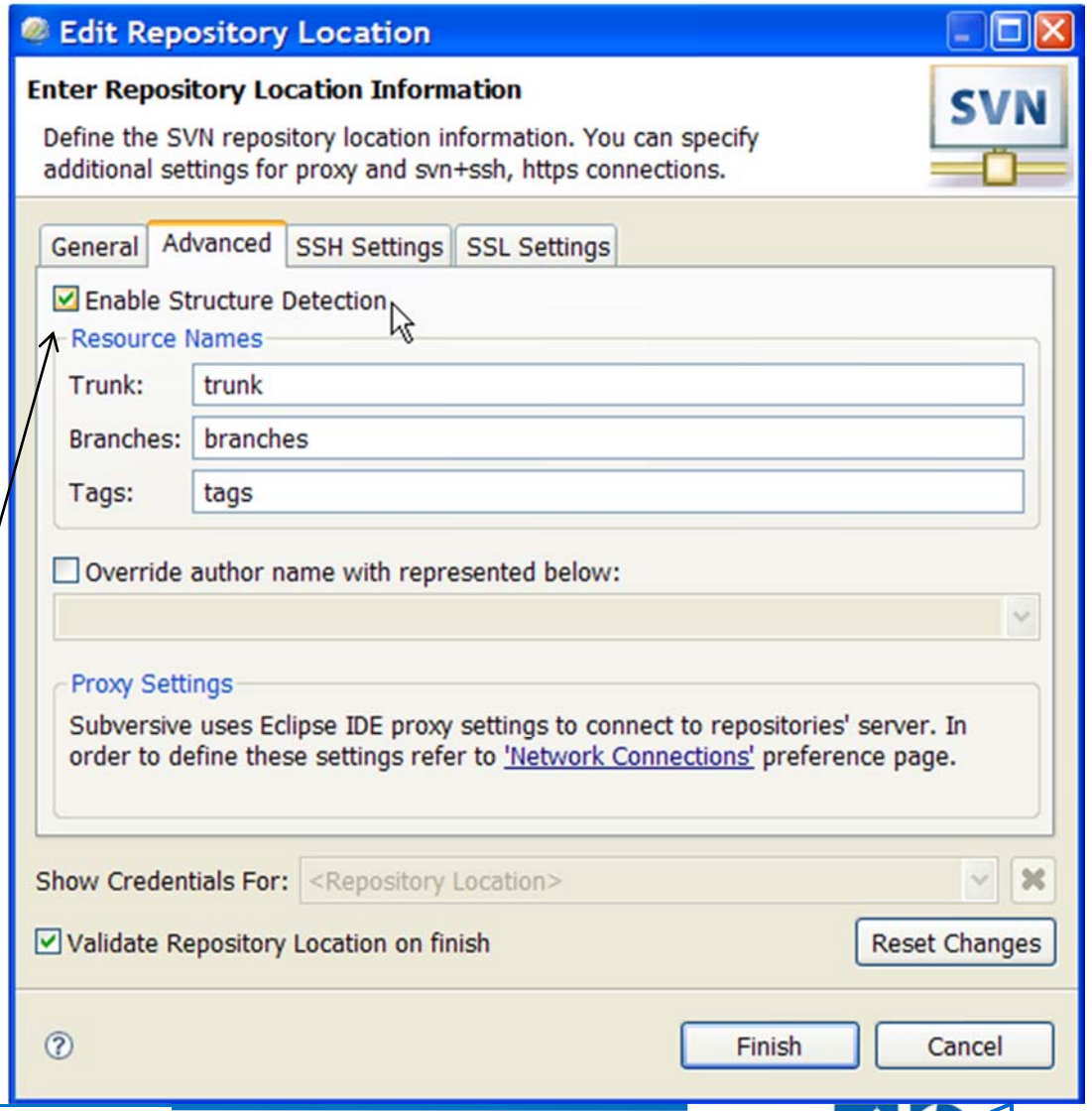
Repository Layout: Spezielle Ordner

- trunk
  - Enthält die aktuelle Entwicklungslinie (wie HEAD in CVS)
  - Also den trunk „auschecken“!
- tags
  - Enthält unveränderliche Versionen des Projekts
    - ⇨ Für Releases, Milestones, Backups
- branches
  - Enthält alternative Entwicklungszweige des Projekts
  - Für Untergruppen eines Teams oder parallele Entwicklung verschiedener Varianten



Repository Layout: Alles nur Konvention!

- Vorherige Folie ist nur Konvention!
  - trunk, branches, tags sind für Subversion normale Ordner
  - Es ist bloß der Subversive-Client, der sie besonders behandelt
  - Ob das geschehen soll, kann als Option angegeben werden
- Man darf die Ordner des Projekts organisieren wie man möchte!



Installation des „Subversive“ Plugins für Eclipse

Hinweise zur Installation des „Subversive“ Plugins in Eclipse finden Sie in einem „Exkurs“-Foliensatz zum aktuellen Kapitel und auf der Vorlesungswebsite.

Wichtige Links

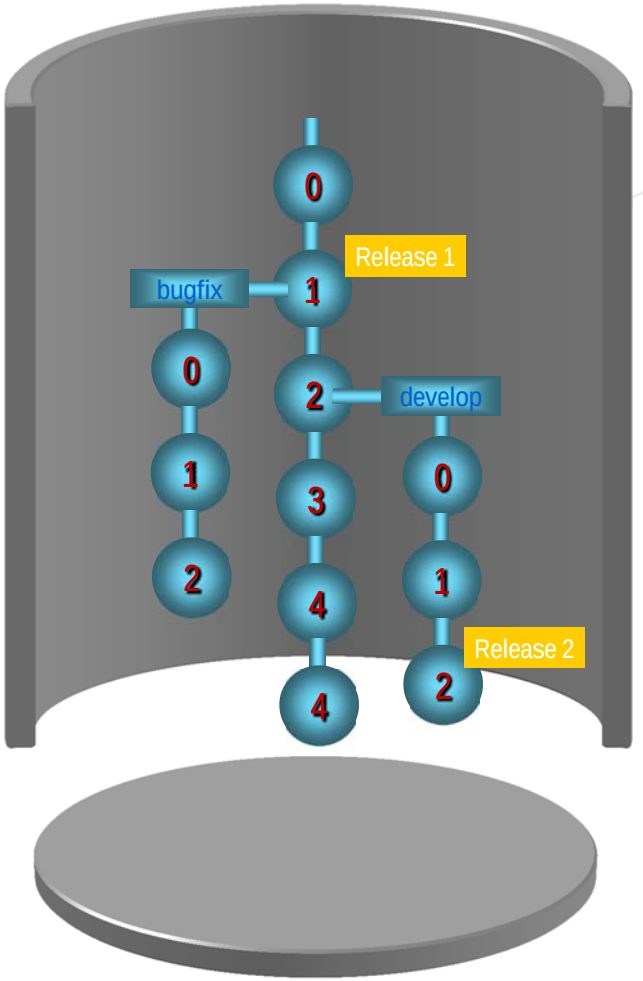
- Subversion Buch
  - http://svnbook.red-bean.com/nightly/en/svn-book.pdf
- Subversion Server
  - subversion.tigris.org
- Subversion Clients als Erweiterung für den Windows Explorer
  - TortoiseSVN (nur für Windows)
- Subversion Clients als Plugins für Eclipse
  - Subversive ← **Wir empfehlen „Subversive“ zu benutzen**
    - ⇨ Homepage: www.polarion.org
    - ⇨ Plugin Download Seite: http://www.polarion.org/projects/subversive/download/1.1/update-site/
  - Subclipse
    - ⇨ www.eclipse.org

Hier geht's weiter am Di, 19.10.2010

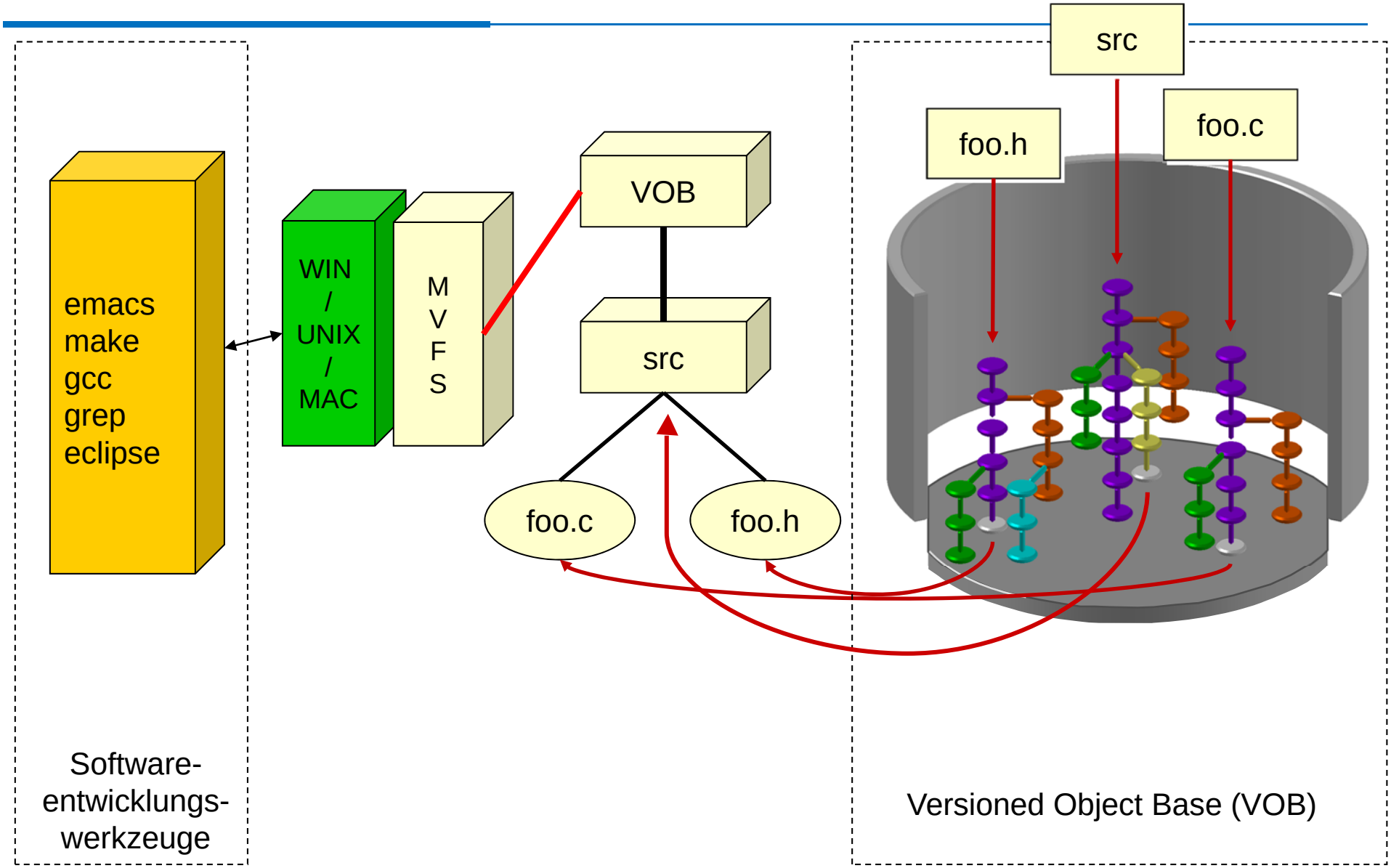
Clear Case

ClearCase VOB (Versioned Object Base)

- Sicheres Repository
  - aufbauend auf objektorientierter Datenbank
  - Zugriff nur mit ClearCase möglich
- Speichert alle versionierten Daten
  - Source code
  - Directories
  - Binaries
  - Wer hat was, wann, warum getan?
- Virtuelles Dateisystem
  - Regelbasierte Konfiguration
- Beliebig viele VOBs im Netzwerk
  - Verteile Datenbank



Virtual File System bei ClearCase



Regelbasierte Arbeitskopie-Verwaltung

- Prinzip: regeldefinierte Auswahl von Objekten / Versionen
  - Sichtdefinition durch benutzerspezifische Regelmenge
  - Regeln agieren als Filter: Auswahl einer bestimmten Objektversion verhindert Zugriff auf alle anderen Versionen des selben Objekts
  - Projektion der ausgewählten Versionen als normaler Verzeichnisbaum

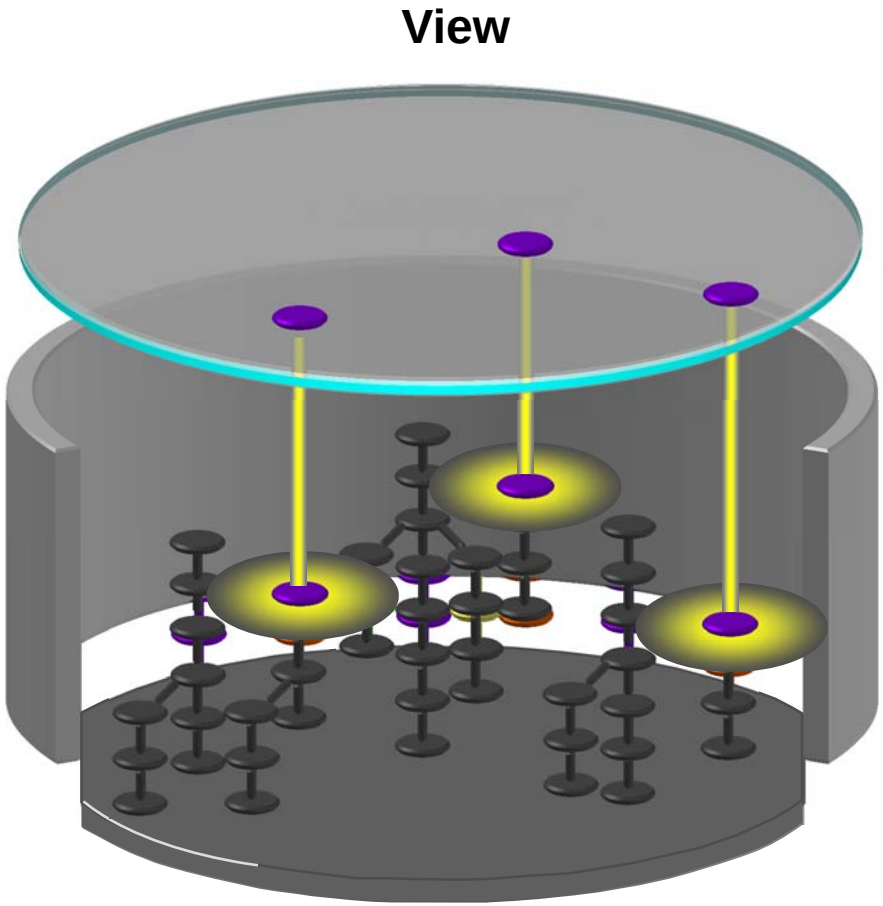
- Statische Regelauswertung
  - Auswertung der Regeln zum Zeitpunkt des Arbeitsbeginns
  - ausgewählte Versionen werden in den privaten Arbeitsbereich kopiert
  - der Benutzer muss am Ende explizit die Synchronisation anstoßen

- Dynamische Regelauswertung (ClearCase)
  - Auswertung der Regeln zum Zeitpunkt des Objektzugriffs
  - Lesevorgänge direkt aus der VOB → längstmöglich aktuellster Stand
  - Objekte werden erst beim ersten Schreiben in den Arbeitsbereich kopiert

Regelbasierte Arbeitskopie-Verwaltung: Regelsyntax und -semantik

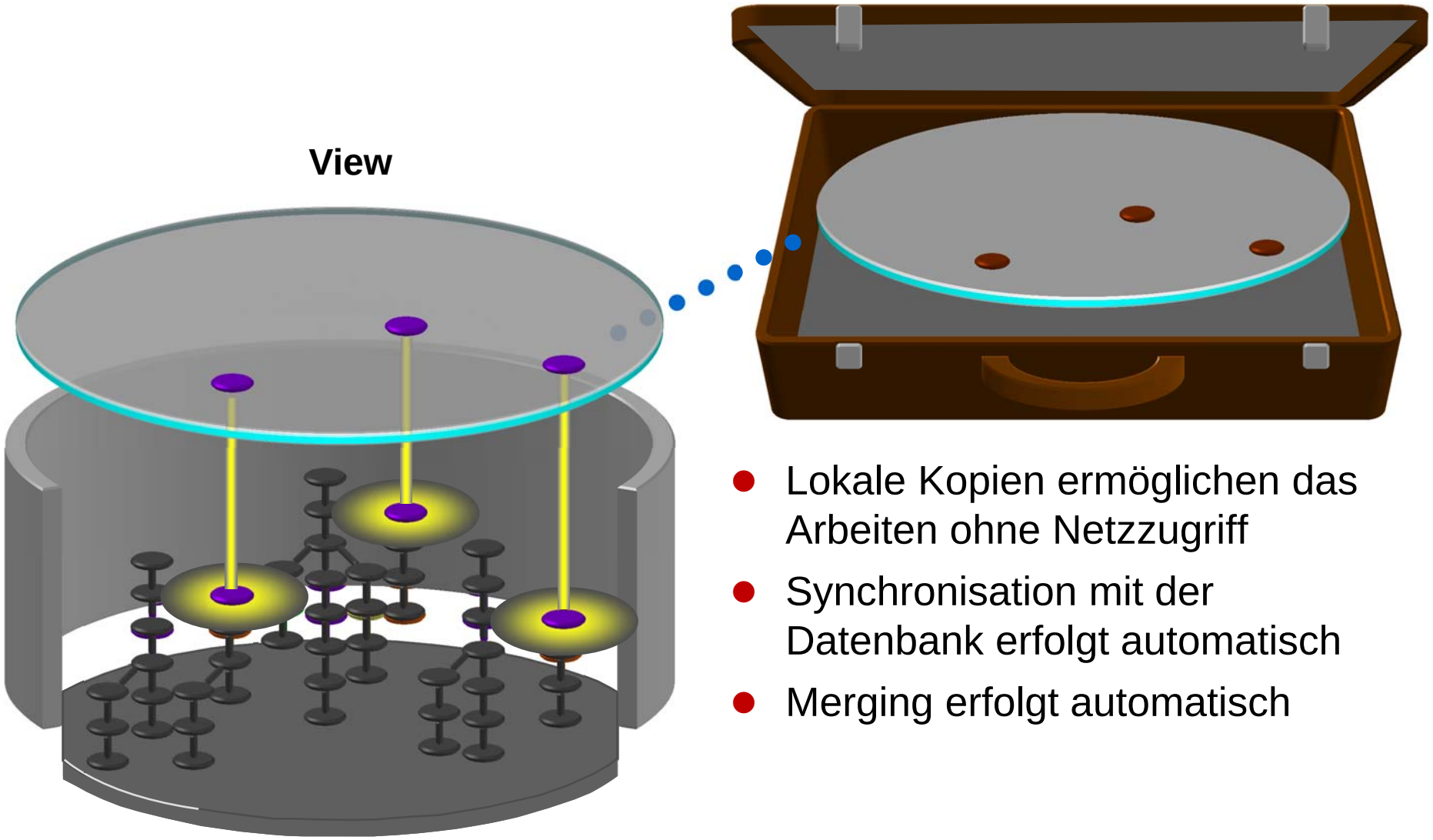
- http://techpubs.sgi.com/library/dynaweb_docs/0620/SGI_EndUser/books/ClrC_UG/sgi_html/ch05.html
- http://www.philforhumanity.com/ClearCase_Support_17.html

ClearCase Views



- Transparentes Arbeiten mit verschiedenen Releases des gleichen Projekts
  - “Zeig mir Version 2.20”
- Arbeiten in Echtzeit
- Sofortiger Zugriff auf den gesamten Datenbestand
- Downloads finden nur bei Zugriff statt
  - Kein komplettes Kopieren!

ClearCase Views: Private Storage

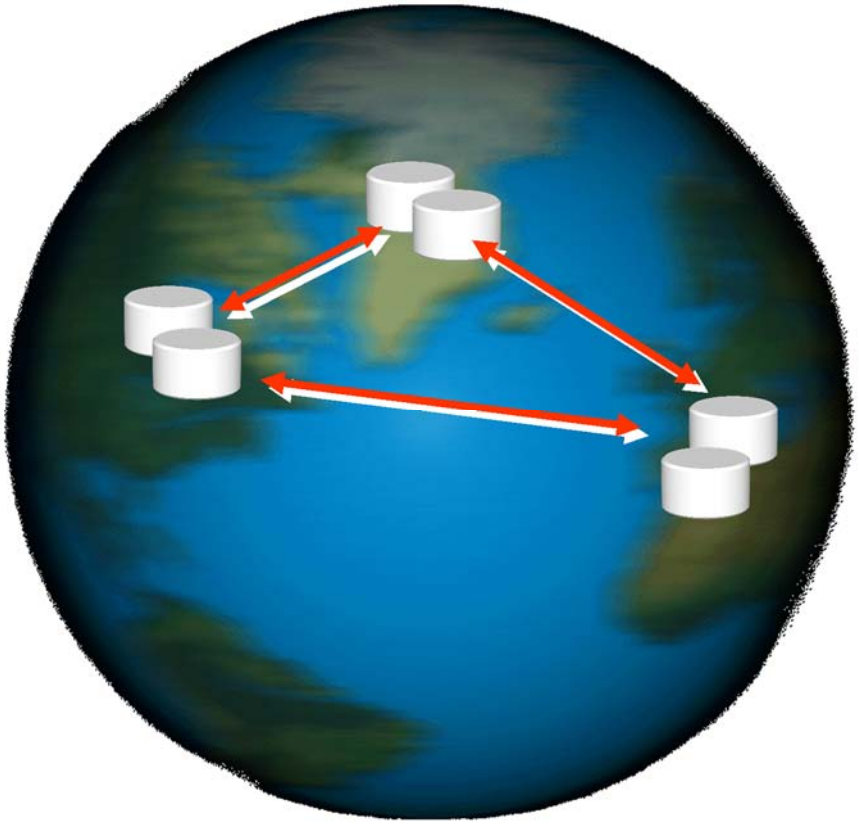


- Lokale Kopien ermöglichen das Arbeiten ohne Netzzugriff
- Synchronisation mit der Datenbank erfolgt automatisch
- Merging erfolgt automatisch

ClearCase Views

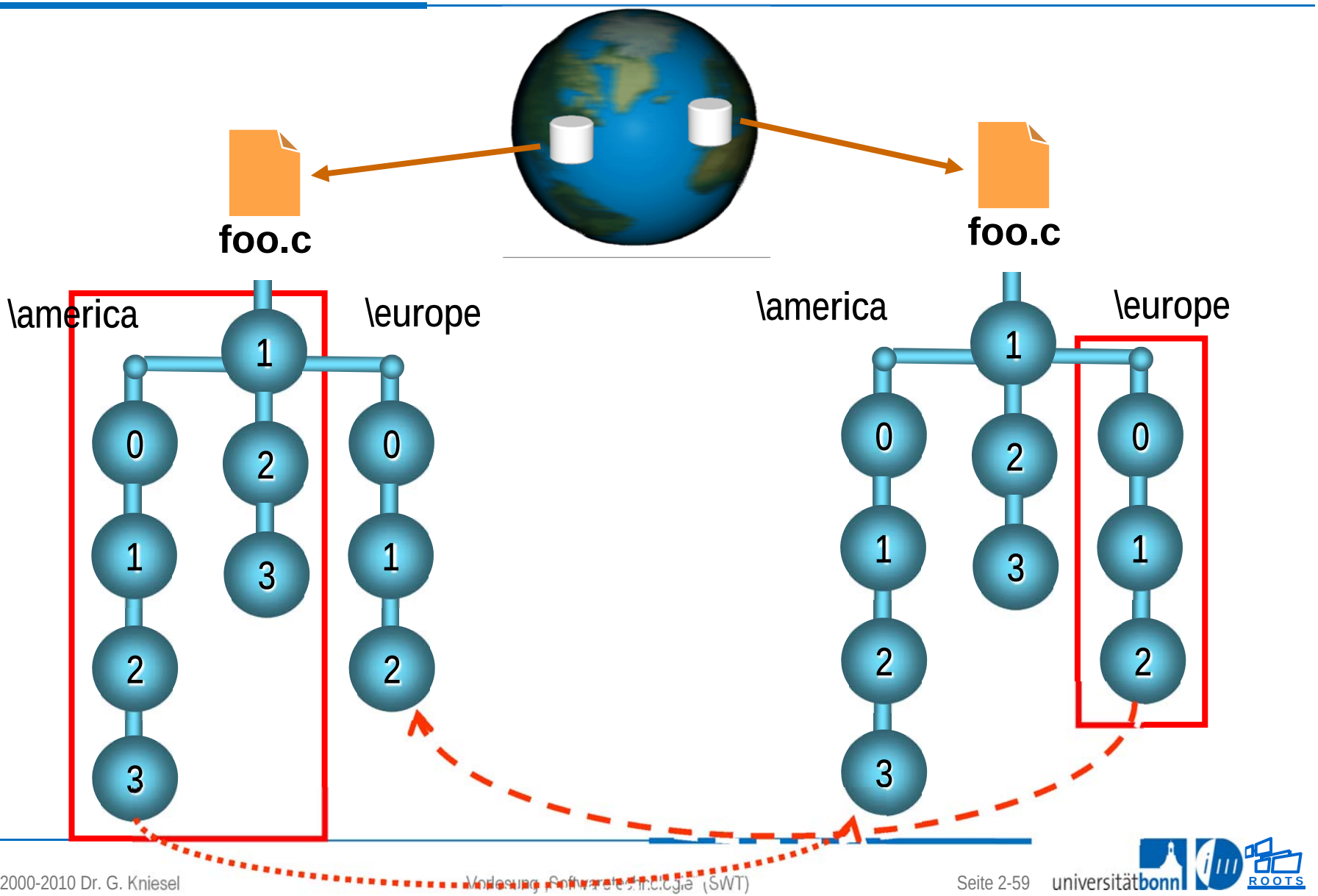
- Alle Objekte im VOB sind schreibgeschützt
- Check-Out
  - Ein Entwickler muß ein check-out-Kommando absetzen, um ein Objekt verändern zu können
  - Dieses wird dann in den privaten Speicherbereich kopiert, ist aber noch unter dem ursprünglichen Namen und Pfad ansprechbar
- Check-In
  - Das check-in-Kommando erzeugt eine neue Objektversion im VOB und löscht die private Kopie
  - Locking: Nur derjenige Entwickler, der das check-out-Kommando abgesetzt hat, darf die neue Objektversion auch wieder einchecken
- Variante: „unreserved check-out“:
  - „first check-in wins“; alle anderen müssen mergen

Verteile Entwicklung



- Paralleles Entwickeln mit geographisch verteilten Projekt-Teams
- Automatische Updates und Kopien der VOBs
- Jeder „branch“ hat einen „Master“-Server → s. nächste Folie

Verteile Entwicklung: „Master“ pro Branch



ClearCase Build Management

Verwaltung der Abhängigkeiten von Artefakten

Reproduzierbare Neugenerierung abgeleiteter Artefakte

Build Management mit herkömmlichen Werkzeugen (make, ant)

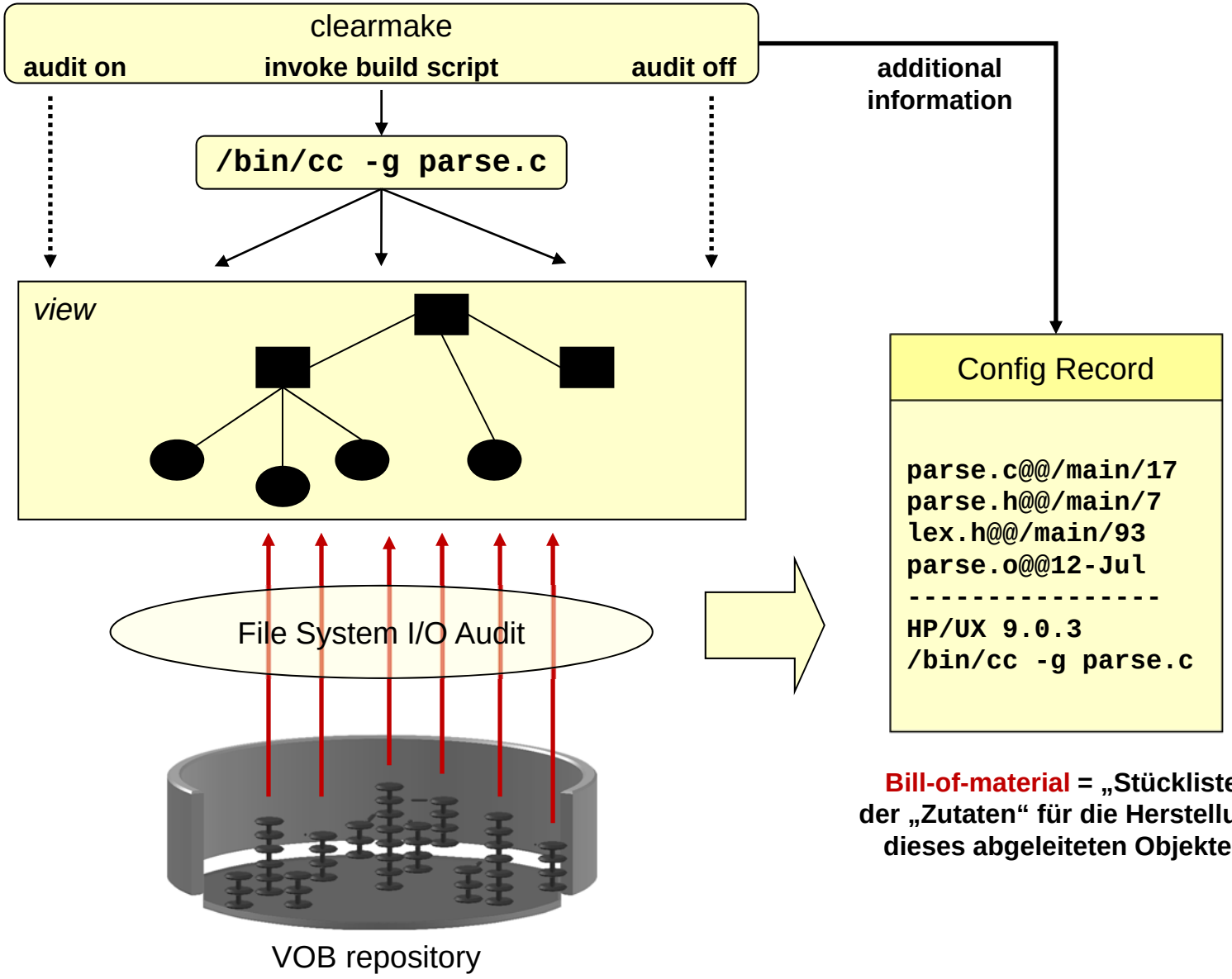
- Keine automatisierte Abhängigkeitsanalyse
  - Abhängigkeiten werden vom Benutzer manuell beschrieben
  - Das ist aber in einer sich schnell ändernden Umgebung sehr aufwendig und fehleranfällig
- Kein „Binary sharing“
  - Das "normale" Make (oder Ant) weiß nichts darüber, dass ein Kollege evtl. ein aufwendig zu erstellendes abgeleitetes Objekt schon erzeugt hat
- Kein „Bill of materials“
  - unklar ob zwei abgeleitete Produkte gleichen Namens (foo.o) wirklich gleich sind, da nicht klar ist, ob sie auch aus den gleichen „Zutaten“ und nach der gleichen „Rezeptur“ erstellt wurden

Build Management mit ClearCase

Verwaltung von Informationen zum reproduzierbaren Erstellen von abgeleiteten Objekten

- Technische Grundlage
  - Durch das VFS ist es möglich, die I/O-Zugriffe vom Compiler oder anderen Werkzeugen auf alle beteiligten Objekte erfassen und protokollieren zu lassen.
  - clearmake erledigt diese Aufgabe
  - Protokollierung der benötigte Daten (Auditing) → Zutatenliste (bill-of-materials)
    - ⇨ Korrekte Versionen aller Quell-Dateien und anderer beteiligter Objekte
    - ⇨ Namen und Versionen der verwendeten Werkzeuge
    - ⇨ benutzte Parametereinstellungen
- Resultierende nützliche Eigenschaften
  - binary sharing: Abgeleitete Objekte können von verschiedenen Benutzern genutzt werden
  - minimal rebuilding: Nur die notwendigsten Operationen zum Erstellen eines abgeleiteten Objekts müssen durchgeführt werden

Build Management mit ClearCase



Bill-of-material = „Stückliste“ der „Zutaten“ für die Herstellung dieses abgeleiteten Objektes.

Zusammenfassung: SCM de Luxe

- Virtuelles Dateisystem
  - → Transparenz der Datenhaltung
- Regelbasierte Konfiguration des Arbeitsbereiches
  - → Flexibilität
- Dynamische Regelauswertung
  - → Aktualität
- Automatisches Merging
  - → Geringer Integrationsaufwand
- Build Management
  - → Reproduzierbare „build“-Ergebnisse, Effizienz durch „minimal rebuilding“
- ECA-Regeln
  - → Workflow-Management
- Verteilung, automatische Spiegelung, verteiltes Building
  - → Unterstützung für internationale Unternehmen

Von zentraler zu verteilter Versionskontrolle

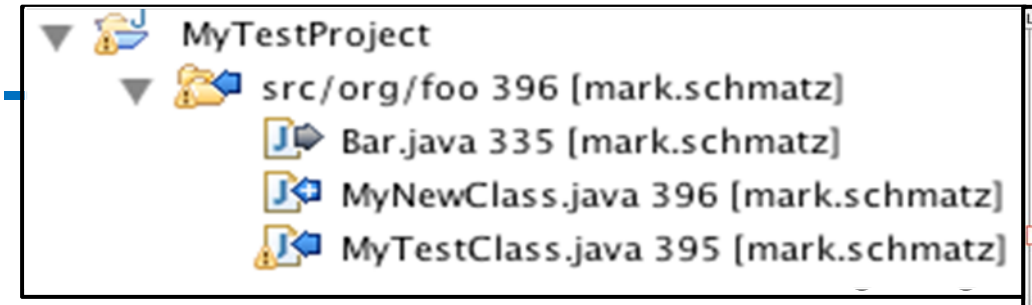
Nachteile zentralisierter Ansätze

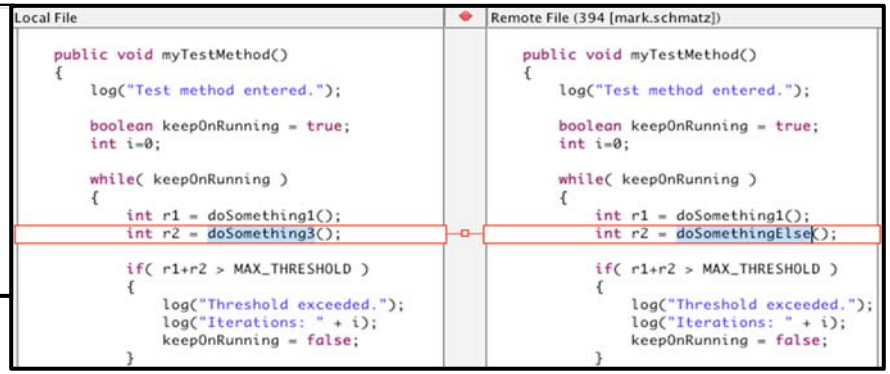
Prinzip der verteilten Versionskontrolle

Vergleich zentrale $\leftrightarrow$ dezentrale Versionskontrolle

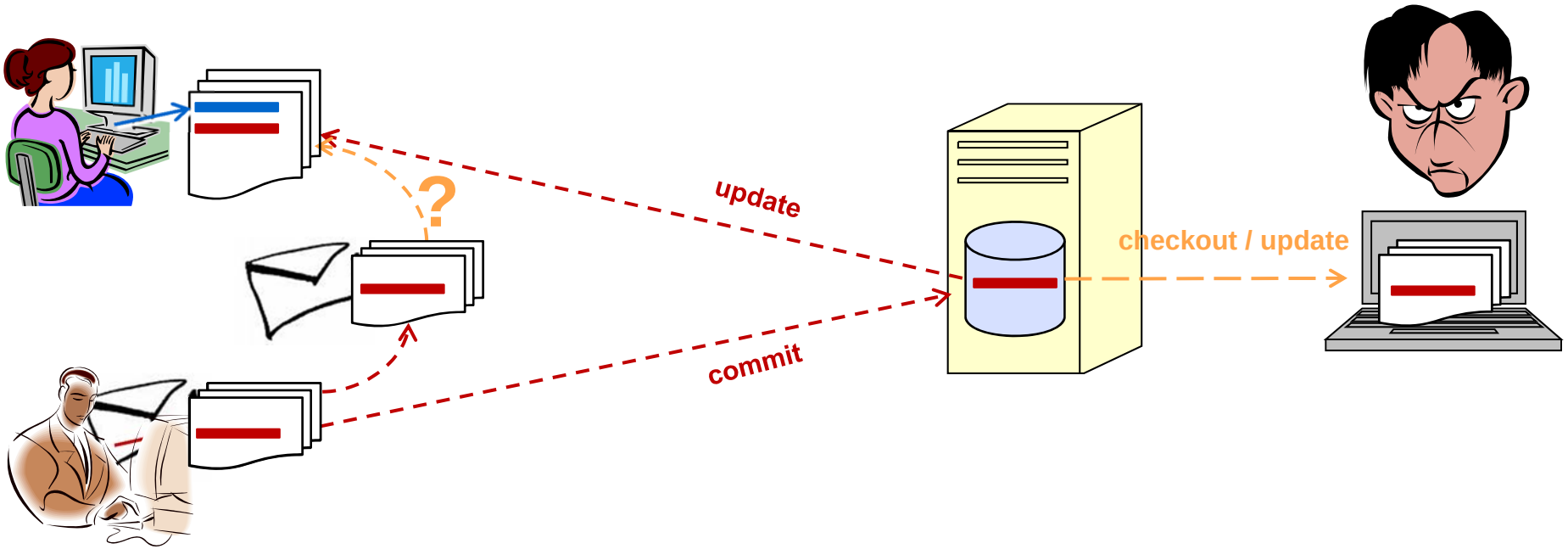
Werkzeuge

Austausch halbfertiger Zwischenstände

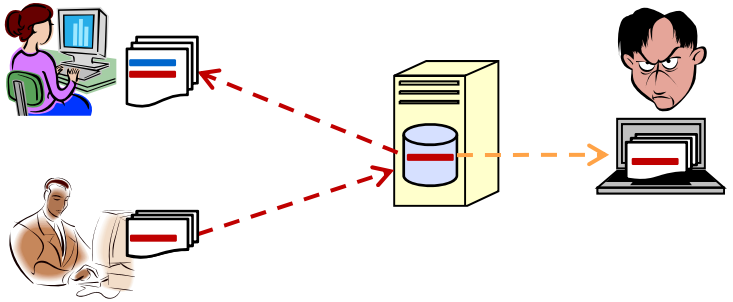




- Ohne SCM: Via e-mail, etc.
  - ⇨ Problem: Keine Unterstützung für Abgleich (Synchronize & Merge)
- Mit SCM: Via commit und update
  - ⇨ Problem: Der inkonsistente Zustand erscheint im Repository → betrifft Andere



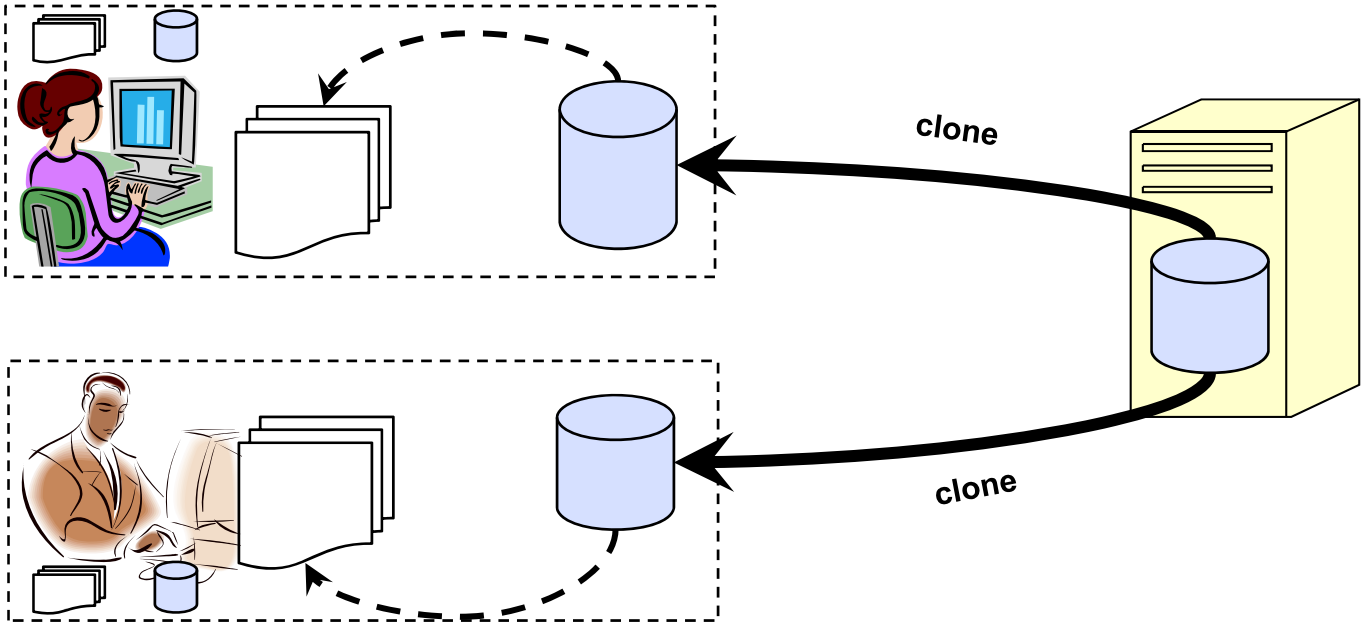
Probleme



- Commit von inkonsistenten Zuständen
  - inkonsistenter Zustand betrifft Andere
  - Regel: Kein commit von inkonsistenten Zustände! Niemals!
- Commit inkonsistenter Zustände verhindern
  - Spezielle „commiter“ Rolle
    - ⇨ Nur sehr erfahrene Entwickler haben das Recht zum „Commit“
    - ⇨ Sie müssen jeden Änderungsvorschlag beurteilen und entscheiden
  - Folgeproblem: Engpass
- 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 sammlen sich umfangreiche Änderungen an
    - ⇨ Rücksetzen auf Repository-Stand entsprechend verlustreich und aufwendig

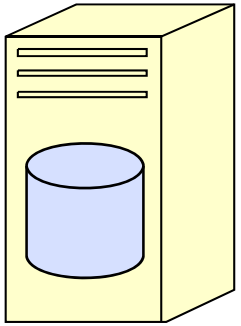
Prinzipien Verteilter Versionskontrolle (Distributed Version Control)

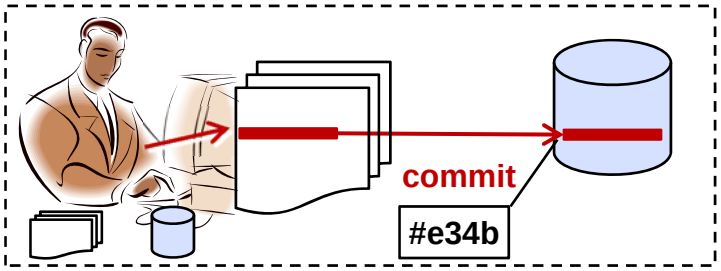
- Multiple, verteilte Repositories
  - Jede(r) hat ein oder mehrere lokale Repositories
    - ⇨ Mehrere → Verschieden Projekte oder verschiedene Aufgaben im Projekt
  - Repository kann geklont werden → inkl. der gesamten Versionshistorie!
    - ⇨ Ab jetzt kann man lokal weiterarbeiten
    - ⇨ Mit allen Funktionalitäten (commit, branch, switch brach, merge, …)



Prinzipien Verteilter Versionskontrolle (Distributed Version Control)

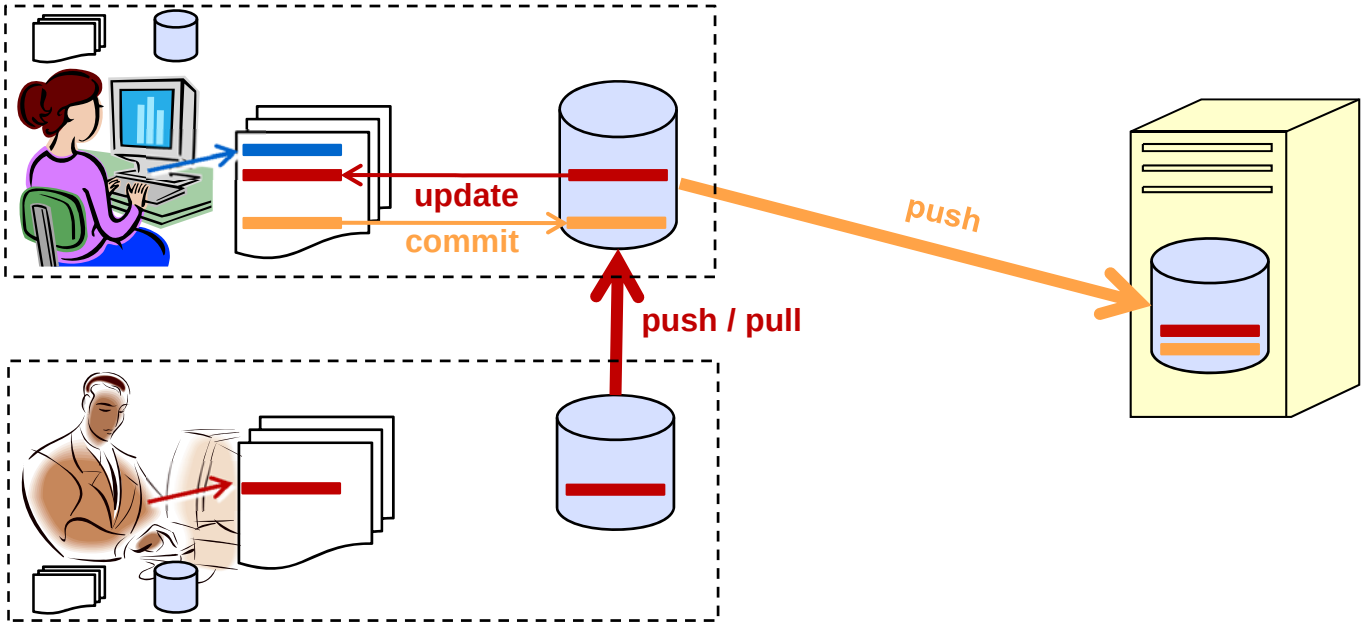
- Checkout, Commit, Brach, Switch Brach, … geschehen lokal
  - Sehr schnell
  - Unabhängig von Netzverbindung und Server
- Fokus auf Gesamt-Repository statt einzelnen Artefakten
  - Commit = Menge von Änderungen („Change set“ / „Patch“)
  - Jedes Commit hat eine eindeutige ID
  - Repository-Zustand =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





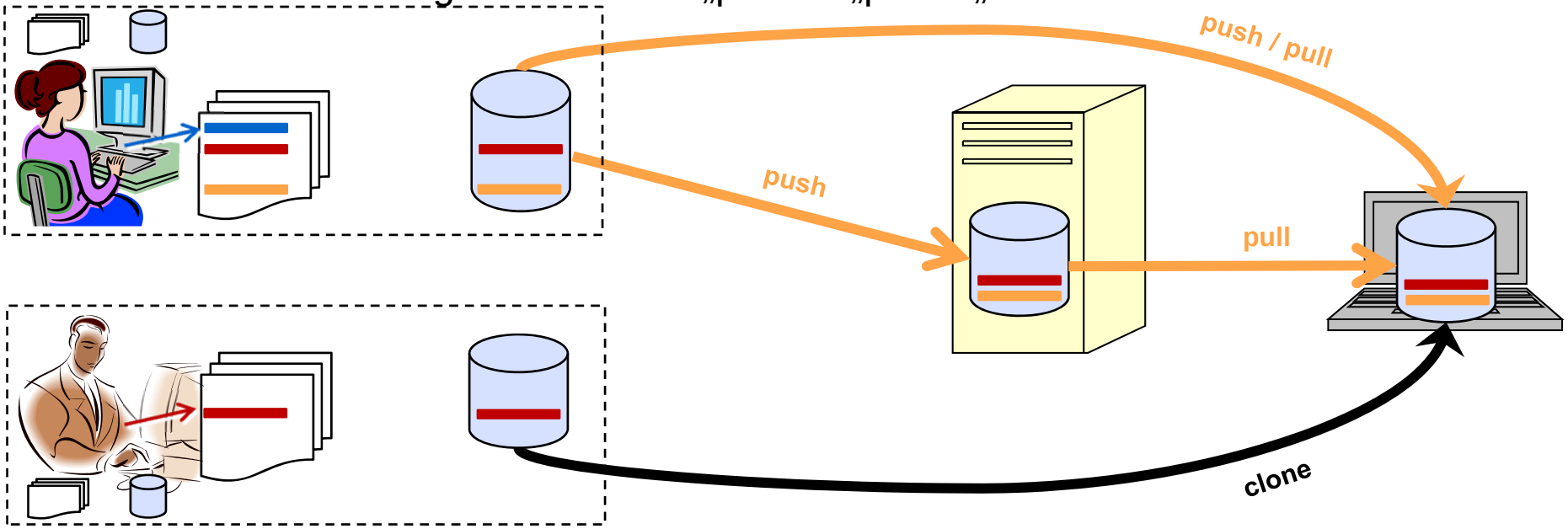
Prinzipien Verteilter Versionskontrolle (Distributed Version Control)

- Repositories können miteinander abgeglichen werden
  - Pull: Holen von Änderungen aus anderem Repository
  - Push: Übertragen von Änderungen in ein anderes Repository
  - Pull beinhaltet bei Bedarf automatisches Merge
    - ⇨ Merge wird als eigene Änderung verwaltet (Hat eigene ID)
    - ⇨ In der Abb. nehmen wir an es sei kein Merge nötig gewesen



Prinzipien Verteilter Versionskontrolle (Distributed Version Control)

- Technische Gleichberechtigung
  - Jedes Repository kann geklont werden und von jedem kann abgeglichen werden
    - ⇨ Egal welches die „ursprüngliche Kopie“ war
- Organisatorische Zentralisierung (wenn gewünscht)
  - Vereinbarung, welches Repository als „Master“ benutzt wird
  - Evtl. durch Zugriffsrechte für „push“ / „pull“ / „clone“ unterstützt



Aktuelle GIT-Clients

- Egit
  - Plugin für Eclipse
  - Platformunabhängig, Open Source
  - Kommendes Release soll auch den Synchronize-View und vieles andere unterstützen, was zur Zeit noch etwas halbherzig funktioniert.
  - http://www.eclipse.org/egit/
- SmartGIT
  - Kommerzielles Produkt aber für nichtkommerzielle Anwendung frei
  - Auf allen Platformen verfügbar
  - Stabil, ausgereift
  - Leider nicht in Eclipse integriert
- Empfehlung
  - Vorerst SmartGIT nutzen aber Fortschritt von Egit verfolgen

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
    - ⇨ Checkout
    - ⇨ —— (implizit)
    - ⇨ Commit
  - Lokal, branch management
    - ⇨ Branch
    - ⇨ Switch
    - ⇨ Merge
    - ⇨ Cherry pick

selektives Merge ausgewählter Commits aus anderem Branch

SCM-Ansätze und Werkzeuge ▸ Vergleich

Zentral (z.B. SVN)

- Konzepte
  - Es gibt nur ein Repository
  - Interaktion nur über zentralen Server
  - Dateiweise Historie
  - Tracking von Umbenennungen
  - Verwaltungsdaten in jedem Ordner

Verteilt (z.B. GIT)

- Konzepte
  - Gleichberechtigte Repositories
  - Zentrale Instanz organisatorisch möglich
  - Repository-weite Historie
  - Änderung hat eindeutige Id
  - Verwaltungsdaten nur im obersten Ordner

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
  - „Branch = Clone“ ▶ Branch im repository oder clone des repository.
  - „Merge = Pull“ ▶ Beim pull wird automatisch ein merge durchgeführt
- 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

- Ein zentrales Backup ist dennoch sinnvoll
  - auch wenn gerne dagegen argumentiert wird
  - eventuell hat nicht jeder alle Änderungen mitbekommen, also kann man sich nicht auf andere Repositories als Backup verlassen
- Es gibt keine wirkliche „neueste Version“
  - Kann man durch ein dediziertes Repository emulieren
- Es gibt keine globalen Versionsnummern
  - Jedes Repository verwaltet seine eigene Änderungshistorie
  - Aber man kann Releases mit sinnvollen Namen taggen

Listen weiterer SCM Werkzeuge

- Wikipedia
  - 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
  - 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/

Vergleiche von SCM Werkzeuge

- Wikipedia (allgemein)
  - 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
- 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.accurev.com/scm_comparisons.html