2. Konfigurationsmanagement

„Beim Konfigurationsmanagement handelt es sich um die Entwicklung und Anwendung von Standards und Verfahren zur Verwaltung eines sich weiterentwickelnden Systemprodukts."

[So07]

Lernziele:

- verstehen, warum **Konfigurationsmanagement** (**KM**) wichtig ist
- Einführung in die wichtigsten **Aufgabengebiete** des KMs
- „Open Source" und kommerzielle **CASE-Werkzeuge** für KM kennenlernen
- KM für **verteilte Software-Entwicklung** in „Open Source"-Projekten erleben
- selbständig **KM-Planung** für (kleine) Softwareprodukte durchführen können

2.1 Einleitung

Behandelte Fragestellungen:

- ☹ Das System lief gestern noch; was hat sich seitdem geändert?
- ☹ Wer hat diese (fehlerhafte?) Änderung wann und warum durchgeführt?
- ☹ Wer ist von meinen Änderungen an dieser Datei betroffen?
- ☹ Auf welche Version des Systems bezieht sich die Fehlermeldung?
- ☹ Wie erzeuge ich Version x.y aus dem Jahre 1999 wieder?
- ☹ Welche Fehlermeldungen sind in dieser Version bereits bearbeitet?
- ☹ Welche Erweiterungswünsche liegen für das nächste Release vor?
- ☹ Die Platte ist hinüber; was für einen Status haben die Backups?
- ☹ …

Definition von Software-KM nach IEEE-Standard 828-1988 [IEEE98]:

SCM (Software Configuration Management) constitutes **good engineering practice** for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for **identifying and controlling** documentation, code, interfaces, and databases to support all life cycle phases
- Supporting a chosen **development/maintenance methodology** that fits the requirements, standards, policies, organization, and management philosophy
- Producing **management and product information** concerning the status of baselines, change control, tests, releases, audits etc.

Anmerkung:

- wenig konkrete Definition
- unabhängig von „Software“

Definition nach DIN EN ISO 10007:

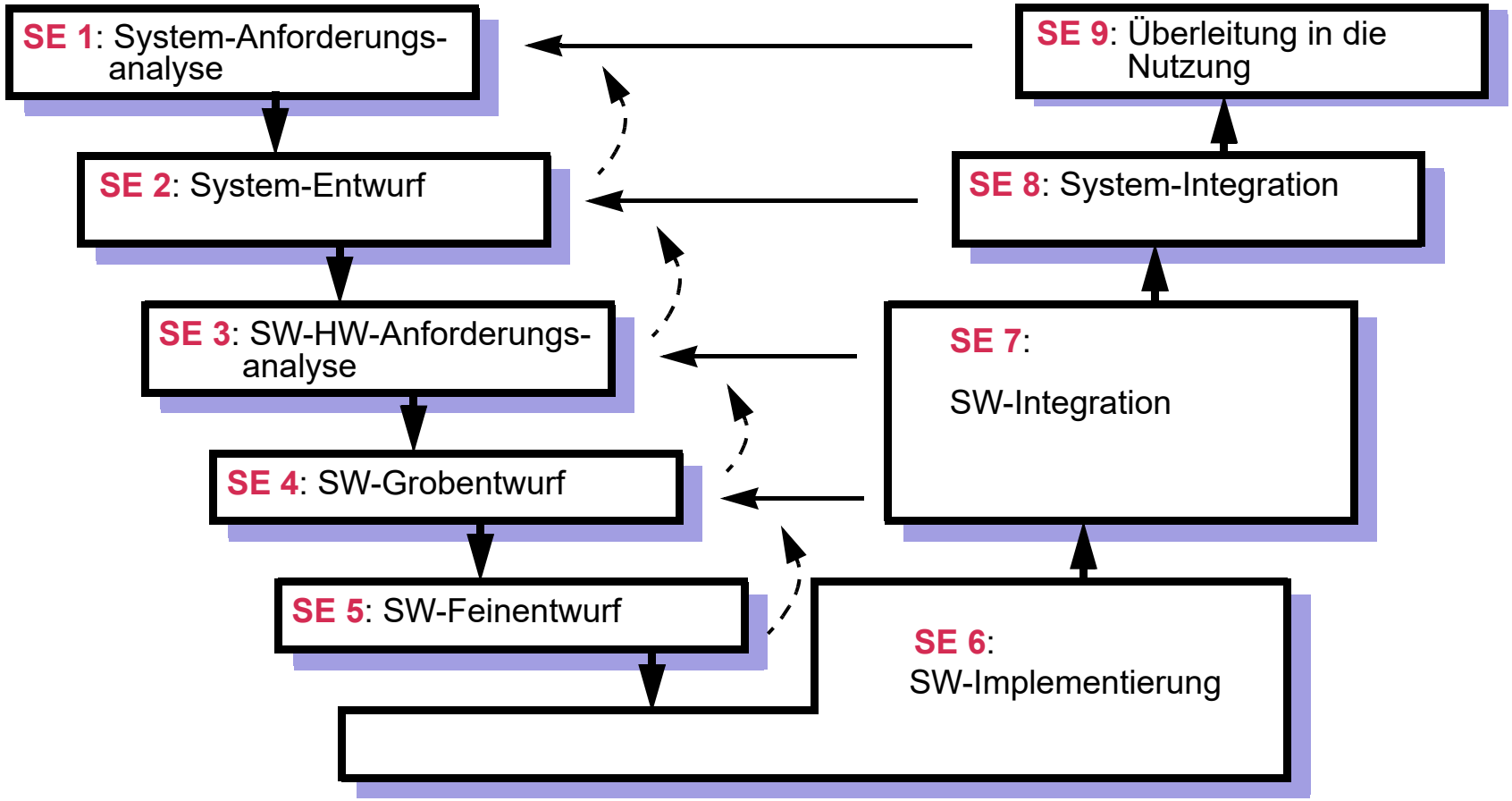
„KM (Konfigurationsmanagement) ist eine Managementdisziplin, die über die gesamte Entwicklungszeit eines Erzeugnisses angewandt wird, um Transparenz und Überwachung seiner funktionellen und physischen Merkmale sicherzustellen. ...
Der KM-Prozess umfasst die folgenden integrierten Tätigkeiten:

- **Konfigurationsidentifizierung**: Definition und Dokumentation der Bestandteile eines Erzeugnisses, Einrichten von Bezugskonfigurationen, ...
- **Konfigurationsüberwachung**: Dokumentation und Begründung von Änderungen, Genehmigung oder Ablehnung von Änderungen, Planung von Freigaben, ...
- **Konfigurationsbuchführung**: Rückverfolgung aller Änderungen bis zur letzten Bezugskonfiguration, ...
- **Konfigurationsauditierung**: Qualitätssicherungsmaßnahmen für Freigabe einer Konfiguration eines Erzeugnisses (siehe Kapitel 3 und Kapitel 4)
- **KM-Planung**: Festlegung der Grundsätze und Verfahren zum KM in Form eines KM-Plans

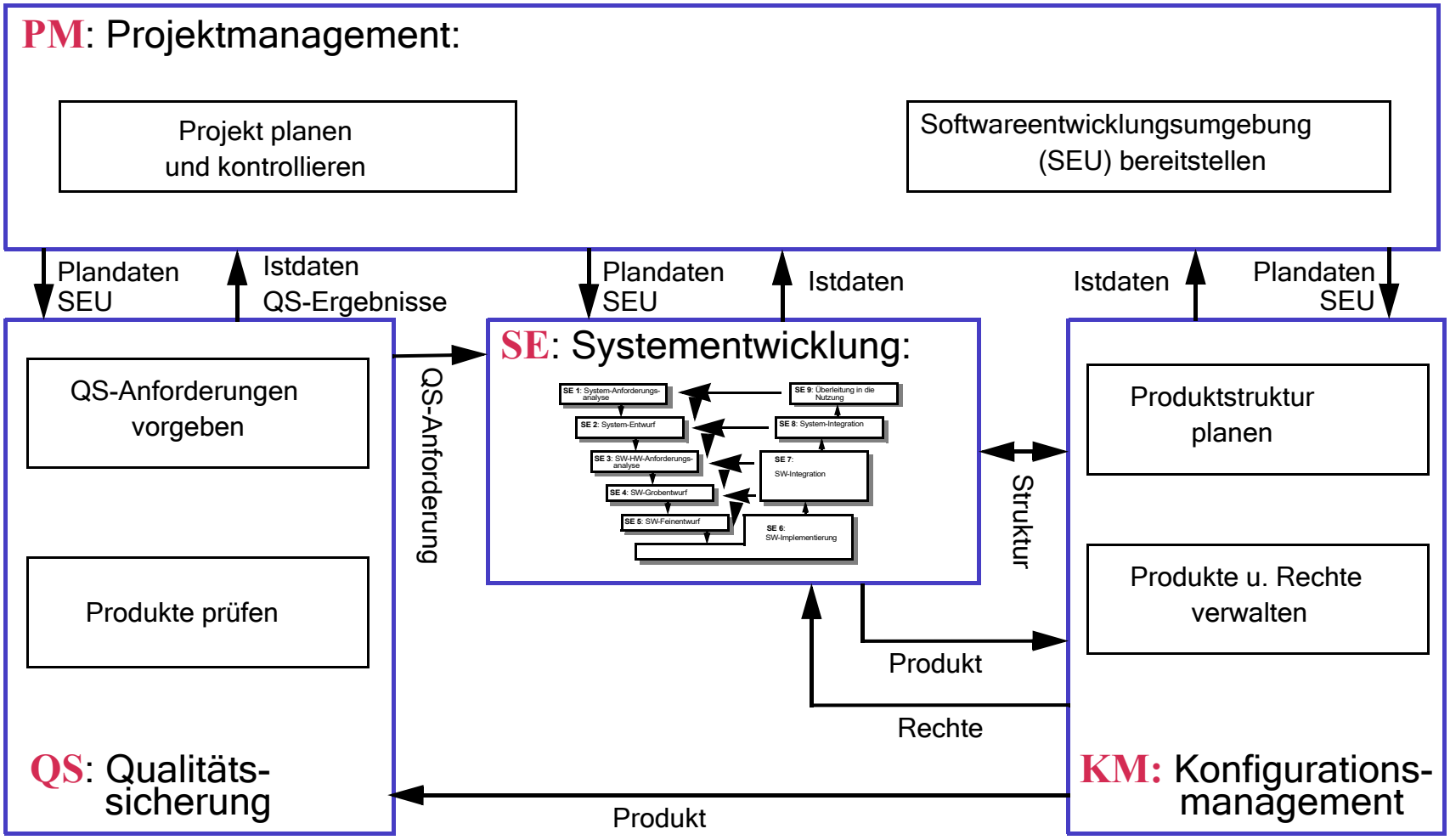
Werkzeugorientierte Sicht auf KM-Aktivitäten:

1. **KM-Planung**: Beschreibung der Standards, Verfahren und Werkzeuge, die für KM benutzt werden; wer darf/muss wann was machen (Abschnitt 2.1)
2. **Versionsmanagement**: Verwaltung der Entwicklungsgeschichte eines Produkts; also wer hat wann, wo, was und warum geändert (Abschnitt 2.2)
3. **Variantenmanagement**: Verwaltung parallel existierender Ausprägungen eines Produkts für verschiedene Anforderungen, Länder, Plattformen (Abschnitt 2.3)
4. **Releasemanagement**: Verwaltung und Planung von Auslieferungsständen; wann wird eine neue Produktversion mit welchen Features auf den Markt geworfen (Abschnitt 2.4)
5. **Buildmanagement**: Erzeugung des auszulieferenden Produkts; wann muss welche Datei mit welchem Werkzeug generiert, übersetzt, ... werden (Abschnitt 2.5)
6. **Änderungsmanagement**: Verwaltung von Änderungsanforderungen; also Bearbeitung von Fehlermeldungen und Änderungswünschen (Feature Requests) sowie Zuordnung zu Auslieferungsständen (Abschnitt 2.6)

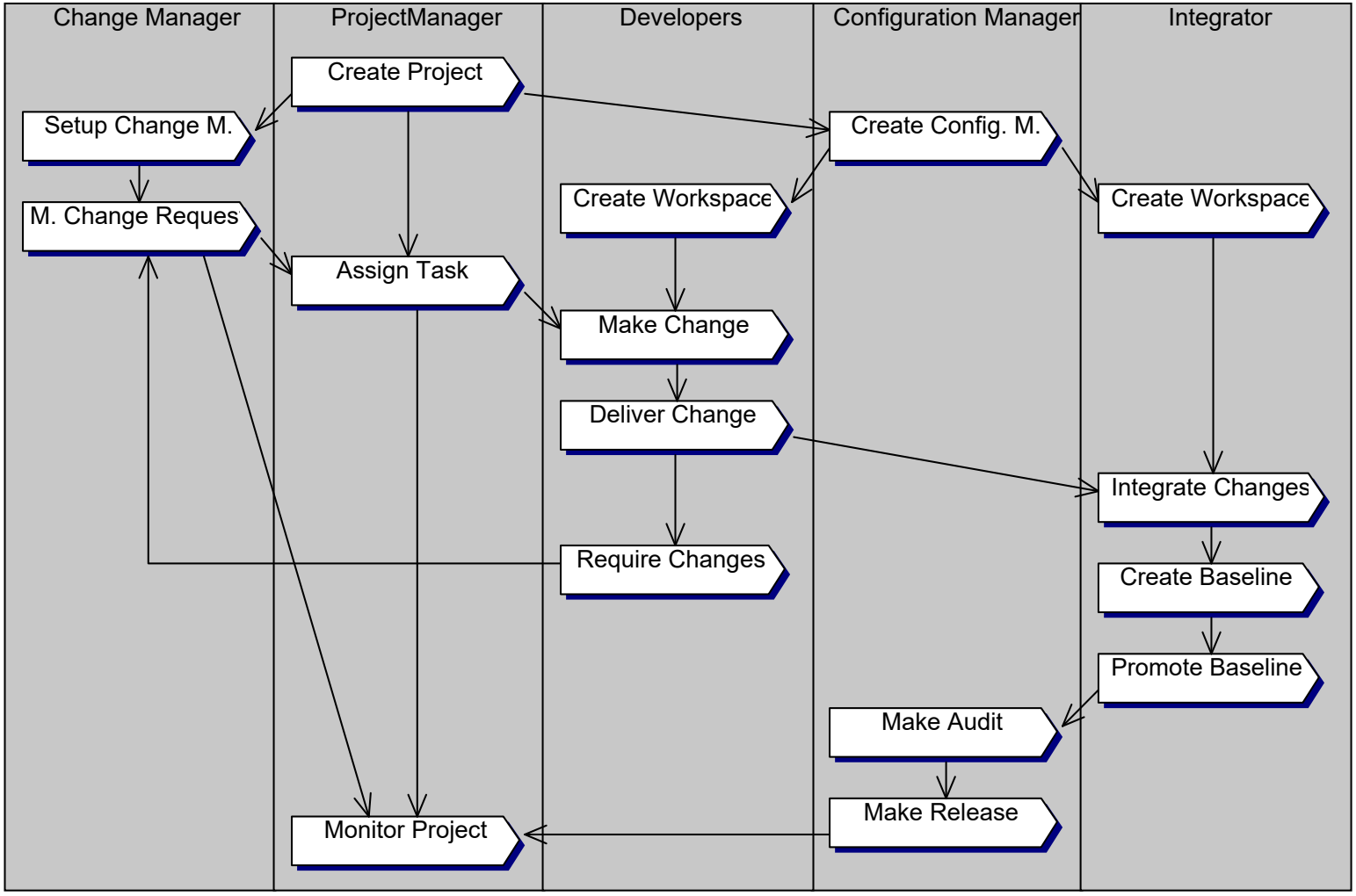
Systementwicklung im V-Modell - zur Erinnerung (siehe SE1):



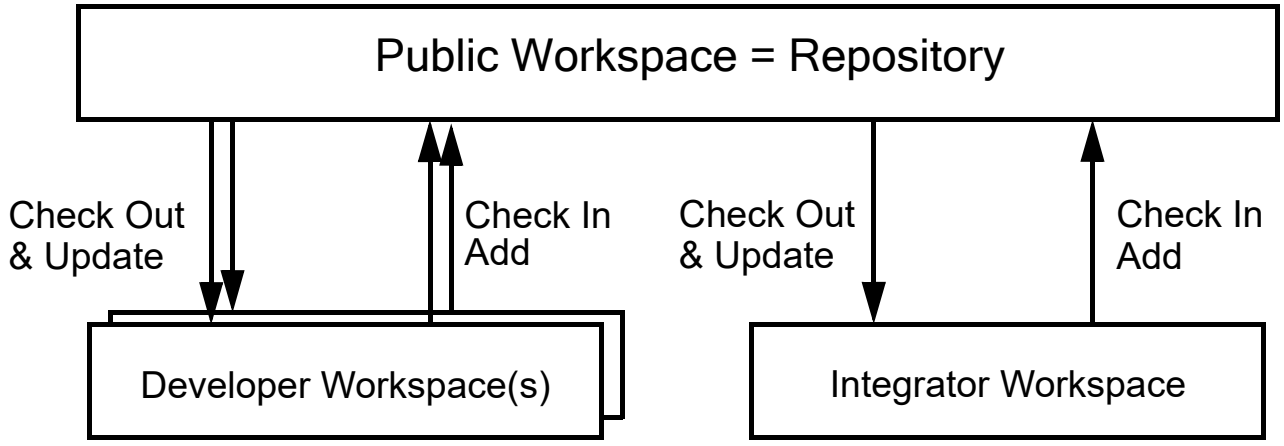
Integration des Konfigurationsmanagements im V-Modell:



Grafische Übersicht über Aufgaben- und Rollenverteilung:



Workspaces für das Konfigurationsmanagement:



- alle Dokumente (Objekte, Komponenten) zu einem bestimmten Projekt werden in einem gemeinsamen Repository (**public workspace**) aufgehoben
- im Repository werden nicht nur aktuelle Versionen, sondern auch alle **früheren Versionen** aller Dokumenten gehalten
- beteiligte Entwickler bearbeiten ihre eigenen Versionen dieser Dokumente in ihrem privaten Arbeitsbereich (private workspace, **developer workspace**)
- es gibt genau einen Integrationsarbeitsbereich (**integrations workspace**) für die Systemintegration

Aktivitäten bei der Arbeit mit Workspaces:

- Personen holen sich Versionen neuer Dokumente, die von anderen Personen erstellt wurden (**checkout**), in ihren privaten Arbeitsbereich.
- Personen passen ihre Privatversionen ggf. von Zeit zu Zeit an neue Versionen im öffentlichen Repository an (**update**).
- Sie fügen (hoffentlich) nur konsistente Dokumente als neue Versionen in das allgemeine Repository ein (**checkin = commit**).
- Ab und an werden neue Dokumente dem Repository hinzugefügt (**add**).
- Jede Person kann alte/neue Versionen frei wählen.

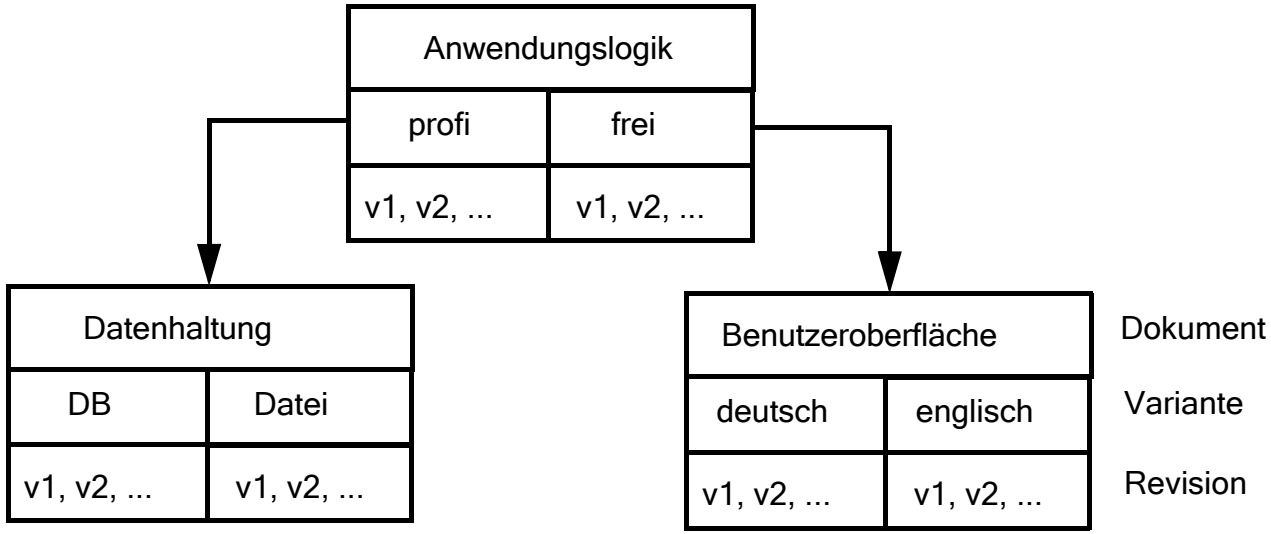
Probleme:

- Wie wird Konsistenz von Gruppen abhängiger Dokumente sichergestellt?
- Was passiert bei gleichzeitigen Änderungswünschen für ein Dokument?
- Wie realisiert man die Repository-Operationen effizient?
- Wie unterstützt man „Offline“-Arbeit (ohne Zugriff auf Repository)?

Weitere Begriffe des Konfigurationsmanagements:

- **Dokument** = Gegenstand, der der Konfigurationsverwaltung unterworfen wird (eine einzelne Datei oder ein ganzer Dateibaum oder ...)
- **(Versions-)Objekt** = Zustand einer Dokument zu einem bestimmten Zeitpunkt in einer bestimmten Ausprägung
- **Varianten** = parallel zueinander (gleichzeitig) existierende Ausprägungen eines Dokuments, die unterschiedliche Anforderungen erfüllen
- **Revisionen** = zeitlich aufeinander folgende Zustände eines Dokuments
- **Konfiguration** = komplexes Versionsobjekt, eine bestimmte Ausprägung eines Programmsystems (oft hierarchisch strukturierte Menge von Dokumenten)
- **Baseline** = eine Konfiguration, die zu einem Meilenstein (Ende einer Entwicklungsphase) gehört und evaluiert (getestet) wird
- **Release** = eine stabile Baseline, die ausgeliefert wird (intern an Entwickler oder extern an bestimmte Kunden oder ...)

Abstraktes Beispiel:



- Anwendungslogik (A), Datenhaltung (D) und Benutzeroberfläche (B) sind drei Pakete (Dokumente), aus denen das Gesamtprodukt besteht
- jedes Paket existiert in zwei Varianten, die jeweils in beliebig vielen zeitlich aufeinander folgenden Revisionen vorliegen können
- es gibt also für das Gesamtprodukt acht mögliche Konfigurationen, falls es für jede Variante genau eine Revision gibt (sonst entsprechend mehr)
- Beispielkonfiguration: { A.profi.v1, D.DB.v2, B.englisch.v2 }

2.2 Versionsmanagement

Versionsmanagement befasst sich (in erster Linie) mit der Verwaltung der zeitlich aufeinander folgenden Revisionen eines Dokuments.

Bekannteste „open source“-Produkte (in zeitlicher Reihenfolge) sind:

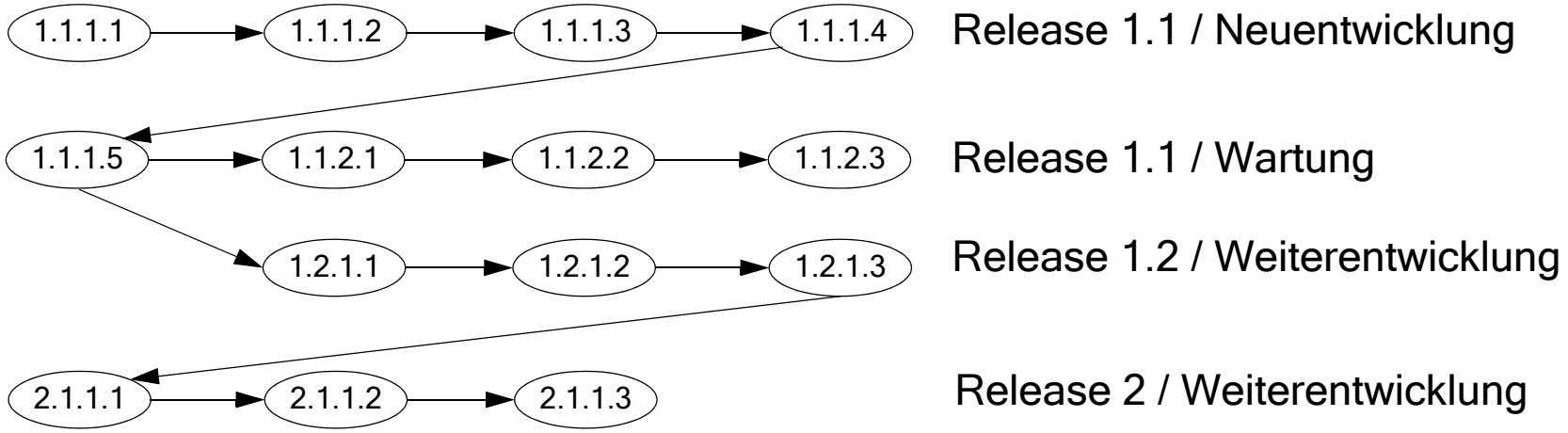
- Source Code Control System **SCCS** von AT&T (Bell Labs):
  - ⇨ effiziente Speicherung von Textdateiversionen als „Patches“
- Revision Control System **RCS** von Berkley/Purdue University
  - ⇨ schnellerer Zugriff auf Textdateiversionen
- Concurrent Version (Control) System **CVS** (zunächst Skripte für RCS)
  - ⇨ Verwaltung von Dateibäumen
  - ⇨ parallele Bearbeitung von Textdateiversionen
- Subversion **SVN** - CVS-Nachfolger von CollabNet initiiert (http://www.collab.net)
  - ⇨ Versionierung von Dateibäumen
- **Git**, Mercurial, ... als verteilte Versionsmanagementsysteme
  - ⇨ jeder Entwickler hat eigene/lokale Versionsverwaltung

Source Code Control System SCCS von AT&T (Bell Labs):

Je Dokument (Quelltextdatei) gibt es eine eigene **History-Datei**, die alle Revisionen als eine Liste jeweils geänderter (Text-)Blöcke speichert:

- ⇨ jeder Block ist ein **Delta**, das Änderungen zwischen Vorgängerrevision und aktueller Revision beschreibt
- ⇨ jedes Delta hat **SCCS-Identifikationsnummer** der zugehörigen Revision: <ReleaseNo>.<LevelNo>.<BranchNo>.<SequenceNo>

Revisionsbäume von SCCS:



Exkurs zu „diff“ und „patch“ - Beispiel:

```
01:PROCEDURE P(a, b: INT);
02:   BEGIN
            ...
10:         IF a < b THEN
11:             ...
12:      END
13:   END P;
```

```
01:PROCEDURE P(v,w: INT);
02:   BEGIN
            ...
10:      IF v < w THEN
11:         ...
12:      ELSE
13:         ...
14:      END
```

```
01:PROCEDURE P(v,w: INT);
02:   BEGIN
            ...
10:      WHILE v < w DO
11:         ...
12:      END
13:END P;
```

diff/patch

diff/patch

```
*** 01,01 *** Hunk 1 ****
!  PROCEDURE P(a, b: INT);
--- 01,01 ---
!  PROCEDURE P(v,w: INT);
**************************
*** 10,11 *** Hunk 2 ****
!         IF a < b THEN
!             ...
--- 10,13 ---
!         IF v < w THEN
!             ...
+         ELSE
```

```
*** 10,13 *** Hunk 1 ****
!         IF v < w THEN
              ...
-         ELSE
-             ...
--- 10,11 ---
!         WHILE v < w DO
              ...
*****************************
```

Erläuterungen zu „diff“ und „patch“:

- „**diff**“-Werkzeug bestimmt Unterschiede zwischen (Text-)Dateien = **Deltas**
- ein Delta (diff) zwischen zwei Textdateien besteht aus einer Folge von „**Hunks**“, die jeweils Änderungen eines Zeilenbereiches beschreiben:
  - ⇨ Änderungen von Zeilen: werden mit „**!**“ markiert
  - ⇨ Hinzufügen von Zeilen: werden mit „**+**“ markiert
  - ⇨ Löschen von Zeilen: werden mit „**-**“ markiert
- reale Deltas enthalten unveränderte **Kontextzeilen** zur besseren Identifikation von Änderungsstellen
- ein **Vorwärtsdelta** zwischen zwei Dateien d1 und d2 kann als „**patch**“ zur Erzeugung von Datei d2 auf Datei d1 angewendet werden
- inverses **Rückwärtsdelta** zwischen zwei Dateien d1 und d2 kann als „patch“ zur Wiederherstellung von Datei d1 auf Datei d2 angewendet werden
- SCCS-Deltas sind in einer Datei gespeichert, deshalb weder Vorwärts- noch Rückwärts- sondern **Inline-Deltas**

Rückwärtsdeltas - Beispiel:

```
01:PROCEDURE P(a, b: INT);
02:   BEGIN
            ...
10:         IF a < b THEN
11:            xxx
12:      END
13:   END P;
```

```
01:PROCEDURE P(v,w: INT);
02:   BEGIN
            ...
10:      IF v < w THEN
11:         xxx
12:      ELSE
13:         yyy
14:      END
```

```
01:PROCEDURE P(v,w: INT);
02:   BEGIN
            ...
10:      WHILE v < w DO
11:         xxx
12:      END
13:END P;
```

diff/patch

diff/patch

```
*** 01,01 *** Hunk 1 ****
!  PROCEDURE P(v,w: INT);
--- 01,01 ---
!  PROCEDURE P(a, b: INT);
**************************

*** 10,13 *** Hunk 2 ****
!         IF v < w THEN
             xxx
-         ELSE
-            yyy
--- 10,11 ---
!         IF a < b THEN
```

```
*** 10,11 *** Hunk 1 ****
!         WHILE v < w DO
             xxx
--- 10,13 ---
!         IF v < w THEN
             xxx
+         ELSE
+            yyy

*****************************
```

Genauere Instruktionen zur Erzeugung von Deltas:

Jedes „diff“-Werkzeug hat seine eigenen Heuristiken, wie es möglichst kleine und/oder lesbare Deltas/Patches erzeugt, die die Unterschiede zweier Dateien darstellen. Ein möglicher (und in den Übungen verwendeter) Satz von Regeln zur Erzeugung von Deltas sieht wie folgt aus:

1. Die Anzahl der geänderten, gelöschten und neu erzeugten Zeilen aller Hunks eines **Deltas** zweier Dateien wird möglichst **klein gehalten**.
2. Jeder Hunk beginnt mit genau **einer** unveränderten **Kontextzeile** und enthält sonst nur geänderte, gelöschte oder neu eingefügte Zeilen (Ausnahme: Dateianfang).
3. Aufeinander folgende Hunks sind also durch jeweils **mindestens eine unveränderte Zeile** getrennt.
4. Optional: Anstelle von Löschen und Neuerzeugen einer Zeile *i* verwendet man die **Änderungsmarkierung „!“**

Durch diese Regeln nicht gelöstes Problem:

Wie erkenne ich, ob eine Änderung in Zeile *i* durch Einfügen einer neuen Zeile oder durch Ändern einer alten Zeile zustande gekommen ist?

Beispiel - alte Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile **und enthält**
3. **sonst nur geänderte, gelöschte oder neu eingefügte Zeilen**.
4. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
5. Änderungsmarkierung.
6. Hunks sind durch mindestens eine unveränderte Zeile getrennt.

Beispiel - neue Version (Zeilennummern nicht Bestandteil der Zeile):

1. Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
3. Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
4. Änderungsmarkierung.
5. **Aufeinander folgende** Hunks sind durch jeweils mindestens eine unveränderte
6. Zeile getrennt.

Falsches Diff-Ergebnis (5 geänderte Zeilen):

*** 01,06 *** Hunk 1 ***

Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
! Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile und enthält
! sonst nur geänderte, gelöschte oder neu eingefügte Zeilen.
! Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
! Änderungsmarkierung.
! Hunks sind durch mindestens eine unveränderte Zeile getrennt.

--- 01,06 ---

Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
! Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
! Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man immer die
! Änderungsmarkierung.
! Aufeinander folgende Hunks sind durch mindestens eine unveränderte
! Zeile getrennt.
****

Richtiges Diff-Ergebnis (2 geänderte Zeilen, 1 neue, 1 gelöschte):

```
*** 01,03 *** Hunk 1 ***
  Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
!  Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile und enthält
-  sonst nur geänderte, gelöschte oder neu eingefügte Zeilen.
--- 01,02 ---
  Die Anzahl der Zeilen aller Hunks eines Deltas wird möglichst klein gehalten.
!  Jeder Hunk beginnt mit genau einer unveränderten Kontextzeile.
**********

***05,06 *** Hunk 2 ***
  Änderungsmarkierung.
!  Hunks sind durch mindestens eine unveränderte Zeile getrennt.
--- 04,06 ---
  Änderungsmarkierung.
!  Aufeinander folgende Hunks sind durch mindestens eine unveränderte
+ Zeile getrennt.
**********
```

Darstellung von Deltas im neueren „Unified Diff“-Format:

```
Index: MyFile
===================================================
- - - MyFile (revision 1)
+++ MyFile (revision 2)
@@ -1,1 +1,1 @@
-  PROCEDURE  P(a, b: INT);
+ PROCEDURE  P(v, w: INT);
@@ -10,2 +10,4 @@
-          IF a < b THEN
+          IF v < w THEN
             xxx
+          ELSE
+            yyy
===================================================
```

Es handelt sich „nur“ um eine etwas andere Art der Darstellung von Unterschieden zwischen verschiedenen Dateien (Dateirevisionen): Änderungen werden als Löschung gefolgt von Einfügung dargestellt, ...

Create- und Apply-Patch in Eclipse:

Die „Create Patch“- und „Apply Patch“-Funktionen in Eclipse benutzen genau das gerade eingeführte „Unified Diff“-Format. Dabei werden bei der Erzeugung von Hunks wohl folgende Heuristiken/Regeln verwendet:

- ein Hunk scheint in der Regel mit drei unveränderten Kontextzeilen zu beginnen (inklusive Leerzeilen).
- zwei Blöcke geänderter Zeilen müssen durch mindestens sieben unveränderte Zeilen getrennt sein, damit dafür getrennte Hunks erzeugt werden

Bei der Anwendung von Patches werden folgende Heuristiken/Regeln verwendet:

- werden der Kontext oder die zu löschenden Zeilen eines Patches so nicht gefunden, dann endet die Patch-Anwendung mit einer Fehlermeldung
- befindet sich die zu patchende Stelle eines Textes nicht mehr an der angegebenen Stelle (Zeile), so wird trotzdem der Patch angewendet
- gibt es mehrere (identische) Stellen in einem Text, auf die ein Patch angewendet werden kann, so wird die Stelle verändert, die am nächsten zur alten Position ist

Eigenschaften von SCCS:

- ❑ für beliebige (Text-)Dateien verwendbar (und nur für solche)
- ❑ Schreibsperren auf “ausgecheckten” Revisionen
- ❑ Revisionsbäume mit manuellem Konsistenthalten von Entwicklungszweigen
- ❑ Rekonstruktionszeit von Revisionen steigt linear mit der Anzahl der Revisionen (Durchlauf durch Blockliste)
- ❑ Revisionsidentifikation nur durch Nummer und Datum

Offene Probleme:

- ☹ kein Konfigurationsbegriff und kein Variantenbegriff
- ☹ keine Unterstützung zur Verwaltung von Konsistenzbeziehungen zwischen verschiedenen Objekten
- ☹ ...

Probleme mit Schreibsperren:

SCCS realisiert ein sogenanntes „pessimistisches“ Sperrkonzept. Gleichzeitige Bearbeitung einer Datei durch mehrere Personen wird verhindert:

- ⇨ ein Checkout zum Schreiben (**single write access**)
- ⇨ mehrere Checkouts zum Lesen (**multiple read access**)

In der Praxis kommt es aber öfter vor, dass mehrere Entwickler dieselbe Datei zeitgleich verändern müssen (oder Person mit Schreibrecht „commit“ vergisst ...)

Unbefriedigende Lösungen:

- ❑ Entwickler mit Schreibrecht macht „commit“ unfertiger Datei, Entwickler mit dringendstem Änderungswunsch macht „checkout“ mit Schreibrecht
  - ⇨ inkonsistente Zustände in Repository, nur einer darf „Arbeiten“
- ❑ weitere Entwickler mit Schreibwunsch „stehlen“ Datei, machen also „checkout“ mit Leserecht und modifizieren Datei trotzdem
  - ⇨ Problem: Verschmelzen der verschiedenen Änderungen

Revision Control System RCS von Berkley/Purdue University

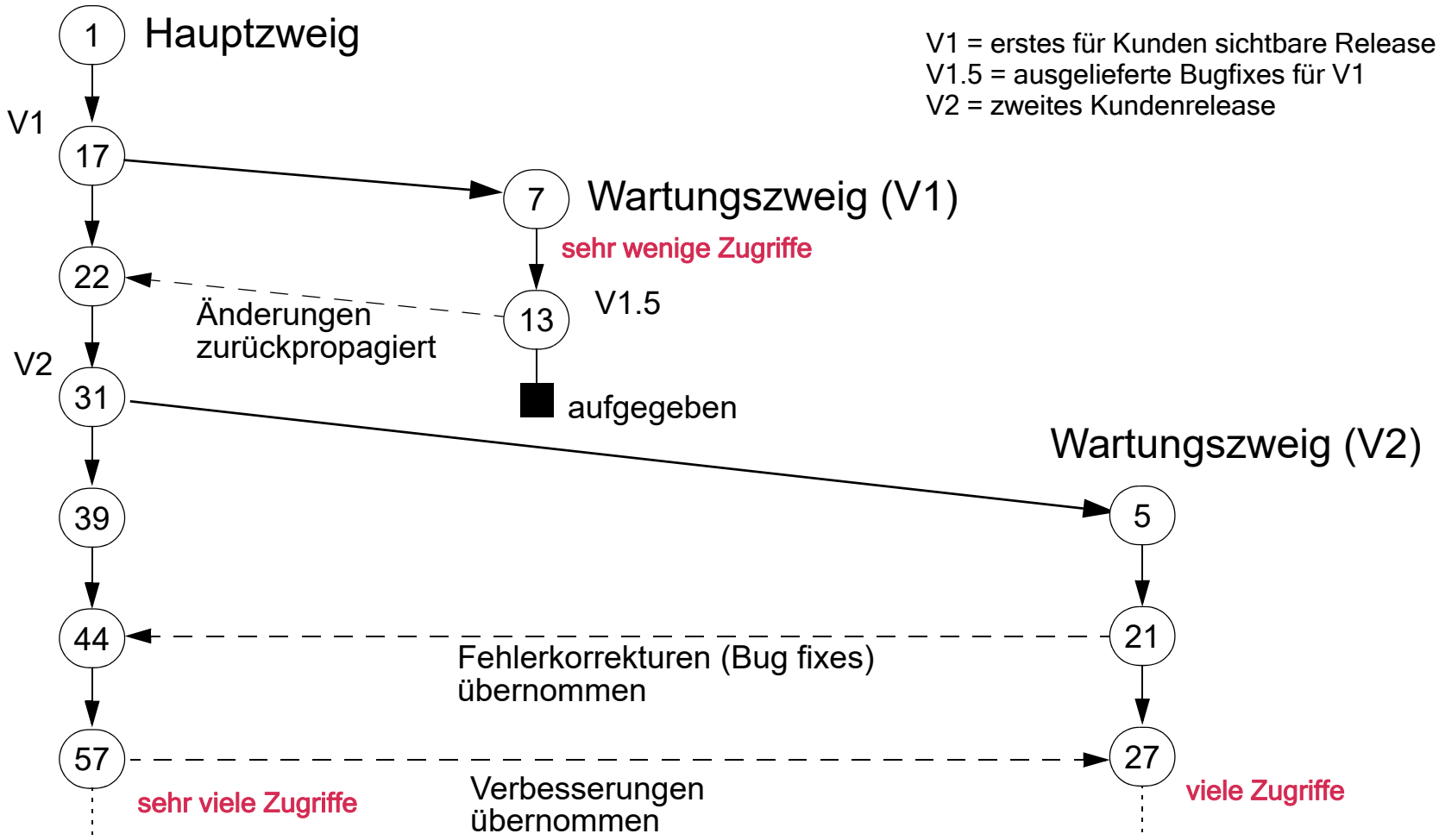
Je Dokument (immer Textdatei) gibt es eine eigene History-Datei, die eine neueste Revision vollständig und andere Revisionen als **Deltas** speichert:

- optionale **Schreibsperren** (verhindern ggf. paralleles Ändern)
- **Revisionsbäume** mit besserem Zugriff auf Revisionen:
  - ⇨ schneller Zugriff auf neueste Revision auf Hauptzweig
  - ⇨ langsamer Zugriff auf ältere Revisionen auf Hauptzweig (mit **Rückwärtsdeltas**)
  - ⇨ langsamer Zugriff auf Revisionen auf Nebenzweigen (mit **Vorwärtsdeltas**).
- Versionsidentifikation auch durch frei wählbare Bezeichner

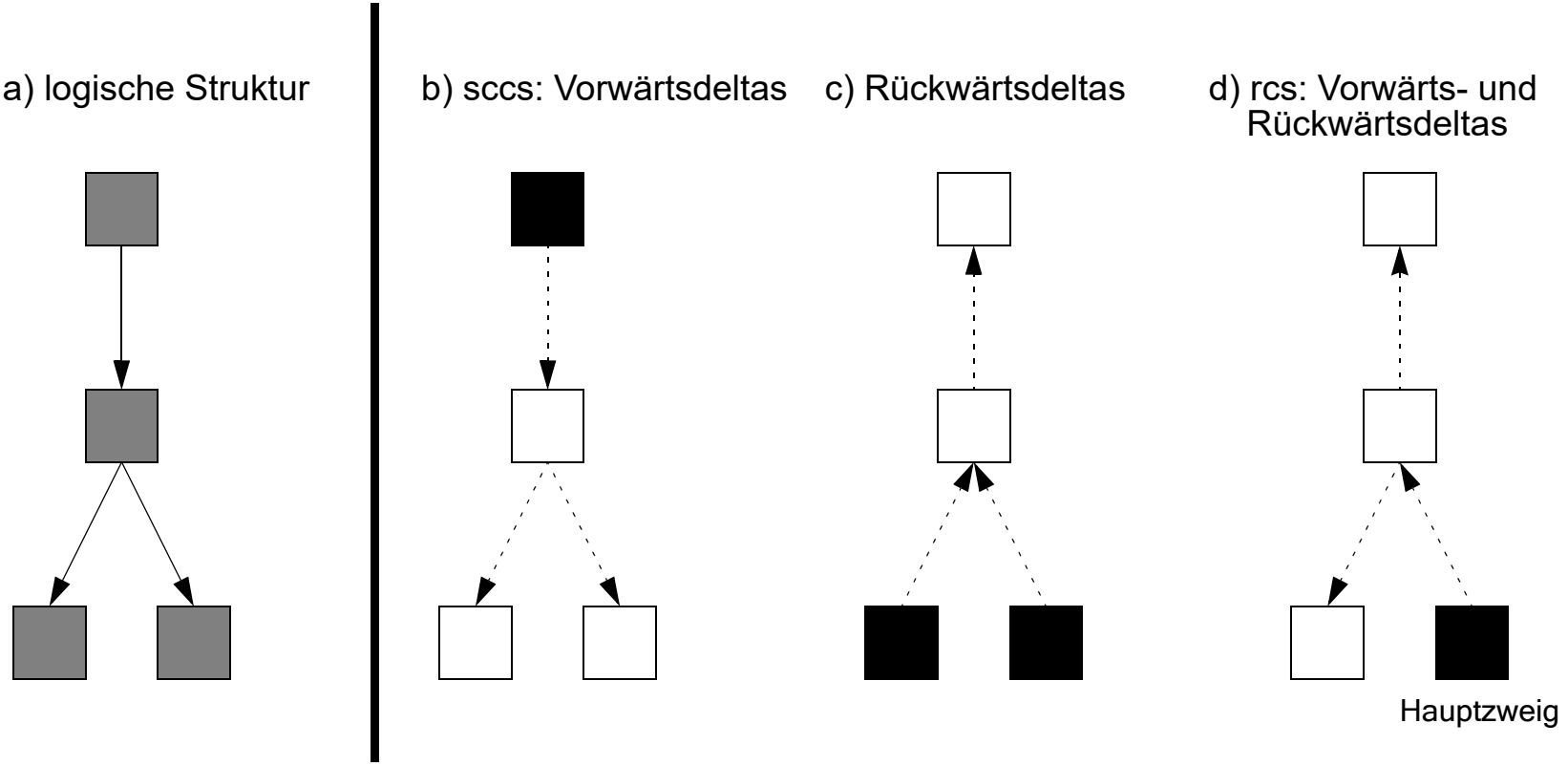
Offene Probleme:

- ☹ kein Konfigurationsbegriff und kein Variantenbegriff
- ☹ ...

Beispielszenario der Softwareentwicklung:



Deltaspeicherung von Revisionen als gerichtete Graphen:

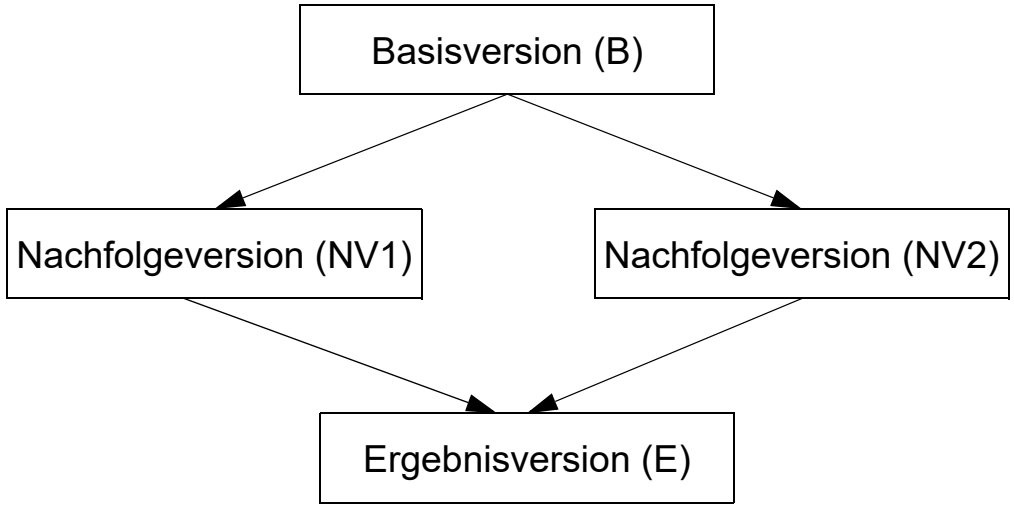


Concurrent Version (Management) System CVS:

Zunächst Aufsatz auf RCS (später Reimplementierung), das Revisionsverwaltung für ganze Directorybäume durchführt und zusätzlich anbietet:

- optimistisches Sperrkonzept mit (fast) **automatischem Verschmelzen** (merge) von parallel durchgeführten Änderungen in verschiedenen privaten Arbeitsbereichen
  - ⇨ Dreiwegeverschmelzen mit manueller Konfliktbehebung
- **Revisionsidentifikation** und damit rudimentäres **Releasemanagement** auch durch frei wählbare Bezeichner
  - ⇨ Auszeichnen zusammengehöriger Revisionen durch symbolische Namen (mit Hilfe sogenannter Tags)
- diverse **Hilfsprogramme**
  - ⇨ wie z.B. „cvsann“, das jeder Zeile einer Textdatei die Information voran-stellt, wann sie von wem zum letzten Mal geändert wurde

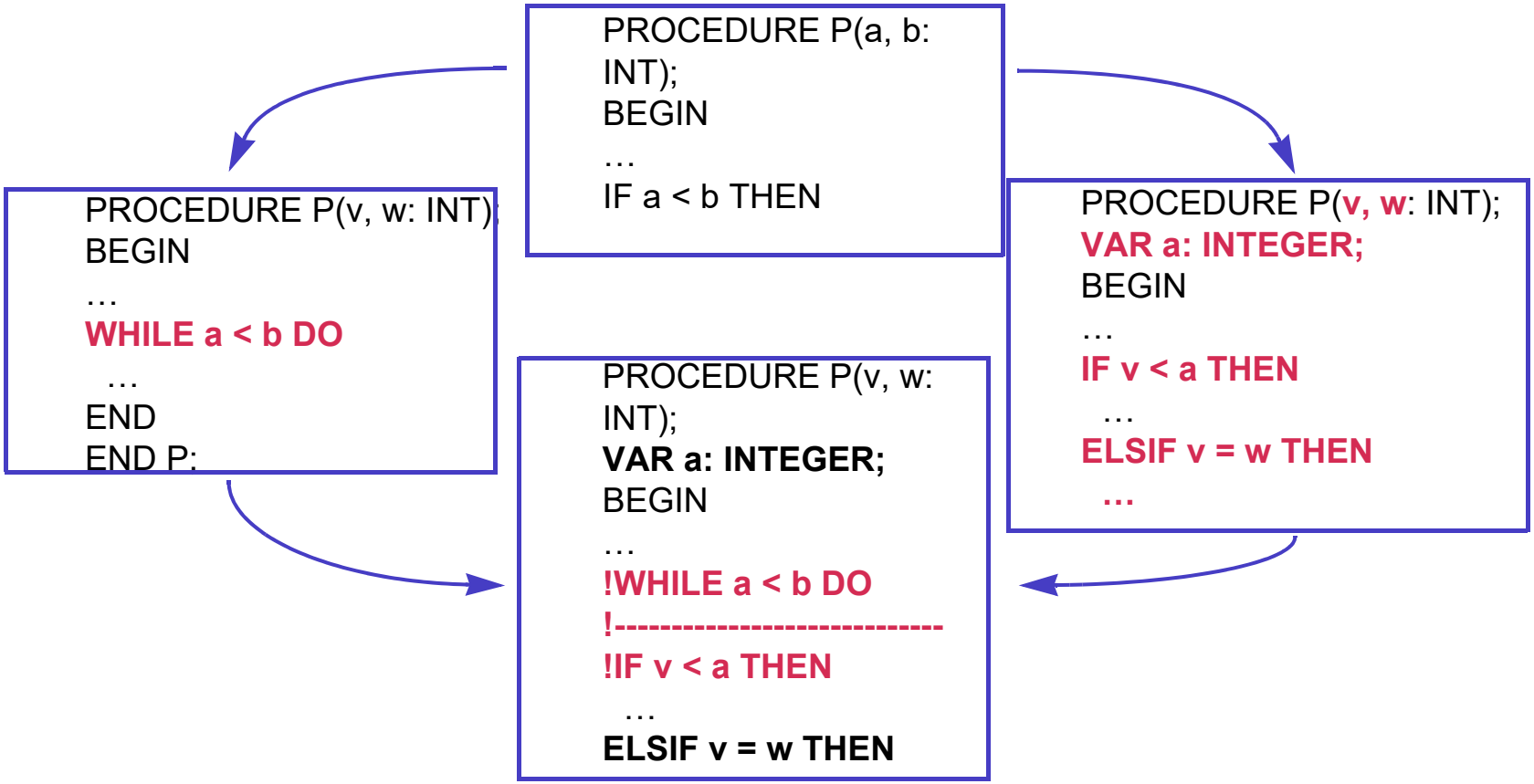
Dreiwegeverschmelzen von (Text-)Dateien:



Verschmelzungsregeln für Revisionen/Varianten:

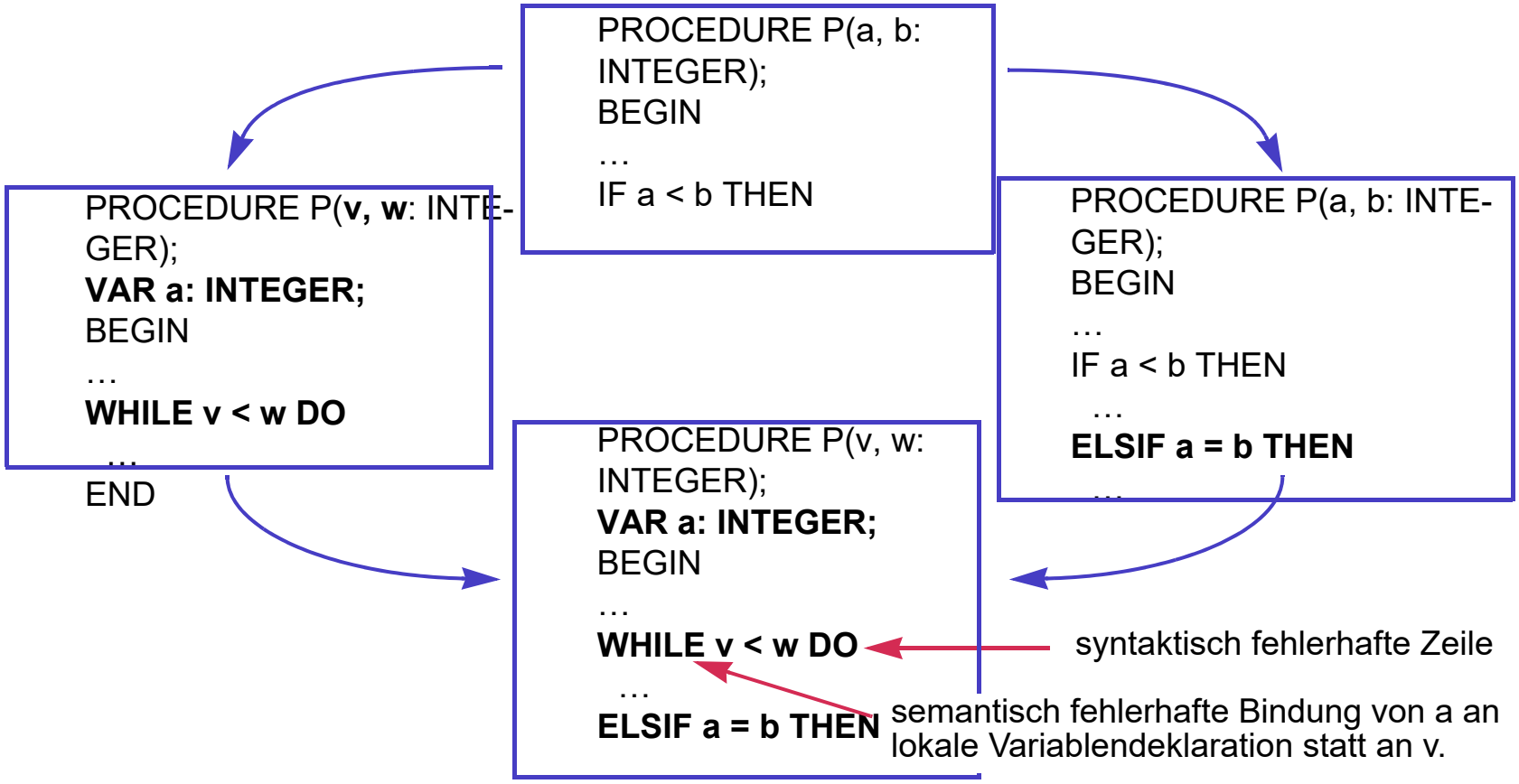
- Textzeile in B, NV1 und NV2 gleich ⇒ Textzeile in E
- Textzeile in B aber nicht in NV x oder/und NV y ⇒ Textzeile nicht in E
- Textzeile in NV x oder/und NV y aber nicht in B ⇒ Textzeile in E
- Textzeile aus B in NV1 und NV2 geändert ⇒ manuelle Konfliktbehebung
  (gilt auch für neue Textzeilen in NV1 und NV2 an gleicher Stelle)

Beispiel für das automatische Verschmelzen:



☹ Gemeldeter Konflikt zwischen Änderungen an derselben Zeile (muss vom Benutzer aufgelöst werden)!

Beispiel für Probleme mit dem automatischen Verschmelzen:



☹ Selbst wenn keine Konflikte gemeldet werden, kann Ergebnis syntaktisch oder semantisch fehlerhaft sein (in der Praxis passiert das aber selten)!

Optimistisches Sperrkonzept mit „Merge“:

- Entwickler A macht checkout einer Revision n
- Entwickler A verändert Revision n lokal zu n1
- Entwickler B macht checkout derselben Revision n
- Entwickler B verändert Revision n lokal zu n2
- Entwickler B macht commit seiner geänderten Revision n2
- Entwickler A versucht commit seiner geänderten Revision n1
  - ⇨ wird mit Fehlermeldung abgebrochen
- Entwickler A macht update seiner geänderten Revision n1
  - ⇨ automatisches merge von n1 und n2 mit Basis n führt zu n’
- Entwickler A löst Verschmelzungskonflikte manuell auf und erzeugt n’’
- Entwickler A macht commit von Revision n’’ (inklusive Änderungen von B)

Synchronisierung mit Watch/Edit/Unedit:

In manchen Fällen will man wegen größerer Umbauten eine Datei(-Revision) „ungestört“ bearbeiten, also zumindest eine Meldung erhalten, wenn andere Entwickler dieselbe Datei bearbeiten (um sie zu warnen):

- watch on schaltet das Beobachten einer Datei ein; beim checkout wird die gewünschte Revision der Datei nur mit Leserecht lokal zur Verfügung gestellt
- watch off ist das Gegenstück zu watch on
- watch add nimmt Entwickler in Beobachtungsliste für Datei auf, für die er vorher ein checkout gemacht hat (Änderungen an Revisionen der Datei werden per Email allen Entwicklern auf Beobachtungsliste gemeldet)
- watch remove entfernt Entwickler von Beobachtungsliste für Datei
- edit verschafft Entwickler Schreibrecht auf lokal verfügbarer Revision einer Datei und meldet das anderen „interessierten“ Entwicklern (enthält watch add)
- unedit nimmt Schreibrecht zurück und meldet das (enthält watch remove)

Identifikation gewünschter Revisionen - Zusammenfassung:

1. Verwendung von **Revisionsnummern** (Identifikatoren): jede Revision einer Datei erhält eine eigene eindeutige Nummer, über die sie angesprochen wird; Nummern werden nach einem bestimmten Schema erzeugt.

2. Verwendung von **Attributen** (Tags): Revisionen werden durch Attribute und deren Werte indirekt identifiziert; Beispiele sind

   ⇨ Kunde (für den Revision erstellt wurde)

   ⇨ Entwicklungssprache (Java, C, ...)

   ⇨ Entwicklungsstatus (in Bearbeitung, getestet, freigegeben, ...)

   ⇨ ...

3. Verwendung von **Zeitstempeln** (Spezialfall der attributbasierten Identifikation): für jede Revision ist der Zeitpunkt ihrer Fertigstellung (commit) bekannt; deshalb können Revisionen über den Zeitraum ihrer Erstellung (jüngste Revision, vor Mai 2003, etc.) identifiziert werden.

Offene Probleme mit Deltaspeicherung und Verschmelzen:

- Berechnung von **Deltas** funktioniert hervorragend für Textdateien (mit Zeilenenden als „Synchronisationspunkte“ für Vergleich)
- Berechnung minimaler Deltas von Binärdateien ist wesentlich schwieriger
- Textverarbeitungsprogramme wie Word oder CASE-Tools besitzen deshalb eigene Algorithmen zur Berechnung (und Anzeige) von Deltas
- **Verschmelzung** von Textdateien funktioniert meist gut (kann aber zu tückischen Inkonsistenzen führen)
- Verschmelzung von Binärdateien oder Grafiken/Diagrammen ist im allgemeinen nicht möglich
- Programme wie Word oder CASE-Tools besitzen deshalb eigene Verschmelzungsfunktionen

Verbleibende Mängel von CVS:

- ☹ zugeschnitten auf Textdateien bei (Deltaberechnung und Verschmelzen)
- ☹ keine Versionierung von Verzeichnisstrukturen (Directories)
- ☹ nicht integriert mit „Build-“ und „Changemanagement“
- ☹ keine gute Unterstützung für geographisch verteilte Software-Entwicklung
- ☹ ...

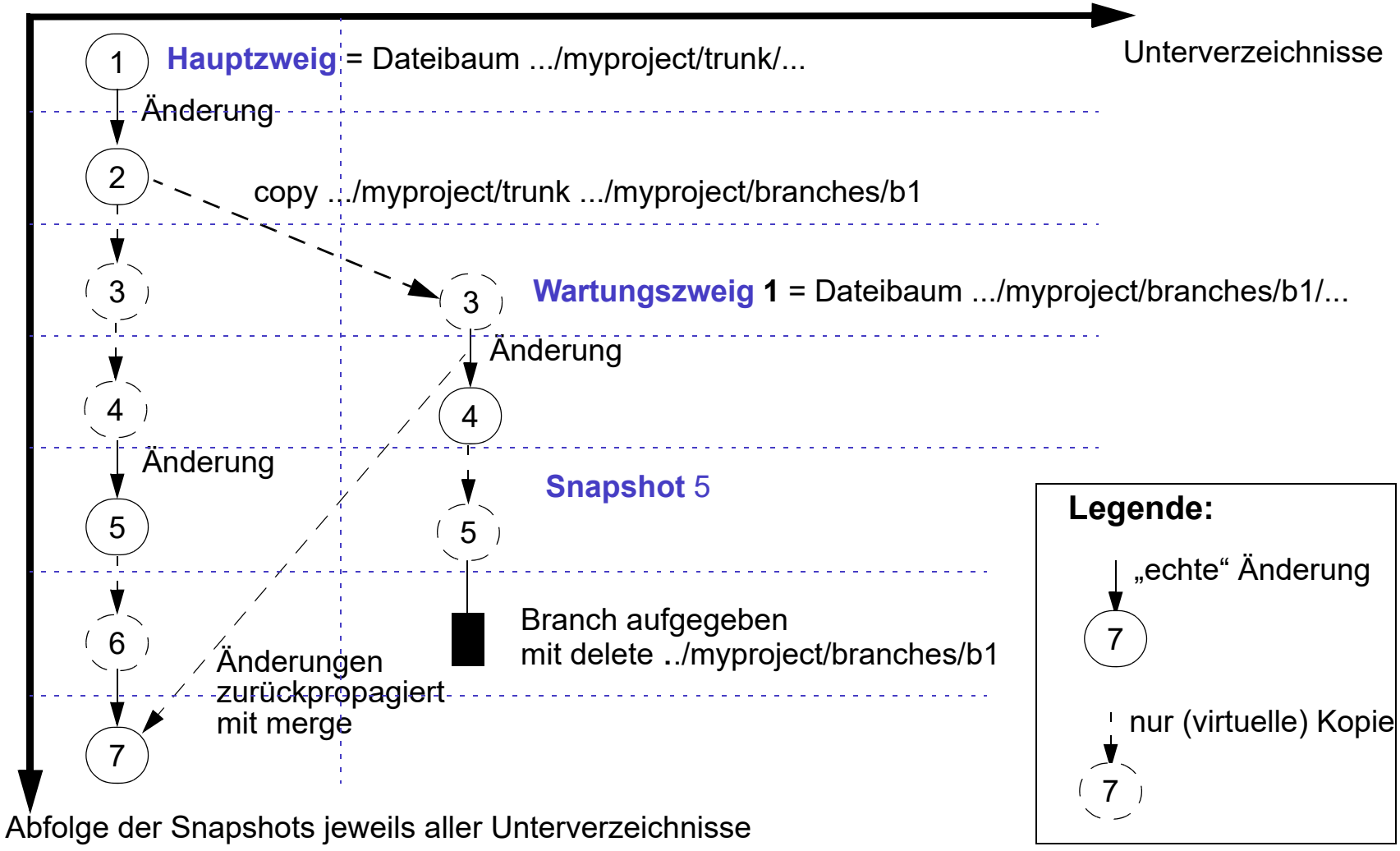
Aber:

- ☺ als „Open Software“ ohne Anschaffungskosten erhältlich
- ☺ Administrationsaufwand hält sich in Grenzen
- ☺ für Build- und Changemanagement gibt es ergänzende Produkte

CVS-Nachfolger Subversion (SVN) [Ca10], [Po09]:

- ☺ immer ganze Dateibäume werden durch commit versioniert (inklusive Erzeugen, Löschen, Kopieren und Umbenennen von Dateien und Unterverzeichnissen)
- ☺ commit ganzer Verzeichnisse ist eine atomare Aktion (auch bei Server-Abstürzen, Netzwerkzusammenbrüchen, ...), die ganz oder gar nicht erfolgt
- ☺ Metadaten (Dateiattribute, Tags) werden als versionierte Objekte unterstützt
- ☺ Deltaberechnung funktioniert für beliebige Dateien (auch Binärdateien); Verschmelzungsoperationen sind aber weiterhin eher auf Textdateien zugeschnitten
- ☺ geographisch verteilte Softwareentwicklung wird besser unterstützt; Dateibäume und Dateien werden durch URLs identifiziert, Datenaustausch kann über http-Protokoll erfolgen
- ☺ Ungewöhnliche, aber einfache/effiziente Behandlung von mehreren Entwicklungszweigen sowie von Systemrevisionen mit bestimmten Eigenschaften (Tags) als „lazy Copies“ (keine echten Kopien, sondern nur neue Verweise/Links)

Beispielszenario in Subversion:



Einsatz von merge in Subversion:

svn merge -r <snapshot1>:<snapshot2> <source> [<wcpath>]

Dieses Kommando berechnet

- ⇨ alle Unterschiede zwischen <snapshot1> und <snapshot2> (als Vorwärts- bzw. Rückwärtsdelta) des Verzeichnisses <source> mit allen Unterverzeichnissen und Dateien
- ⇨ und wendet die Änderungen auf die Dateien im aktuellen Workspace-Verzeichnis (ggf. angegeben durch <wcpath> = Working Copy Path) an
- ⇨ geändert werden dabei einzelne Dateien und ganze (Unter-)Verzeichnisse

Variante des merge-Kommandos:

svn merge <src1>[@<snapshot1>] <src2>[@<snapshot2>] [<wcpath>]

Dieses Kommando wendet das Delta zweier verschiedener Dateien oder Unterverzeichnisse vorgegebener Revisionen (durch Snapshot-Nummer) auf den aktuellen Workspace an. Wird keine Revision angegeben, so wird jeweils die neueste Revision (Head) verwendet.

Beispiele für Verwendung von merge:

- Anwendung aus Sicht von Wartungszweig 1 (propagate changes):

  merge -r 3:4 .../myproject/branches/b1 [wcpath]

  ⇨ propagiert alle Änderungen von Revision 3 auf Revision 4 des Wartungszweiges in die ausgecheckte „Working Copy“ des Hauptzweiges oder eines anderen Wartungszweiges

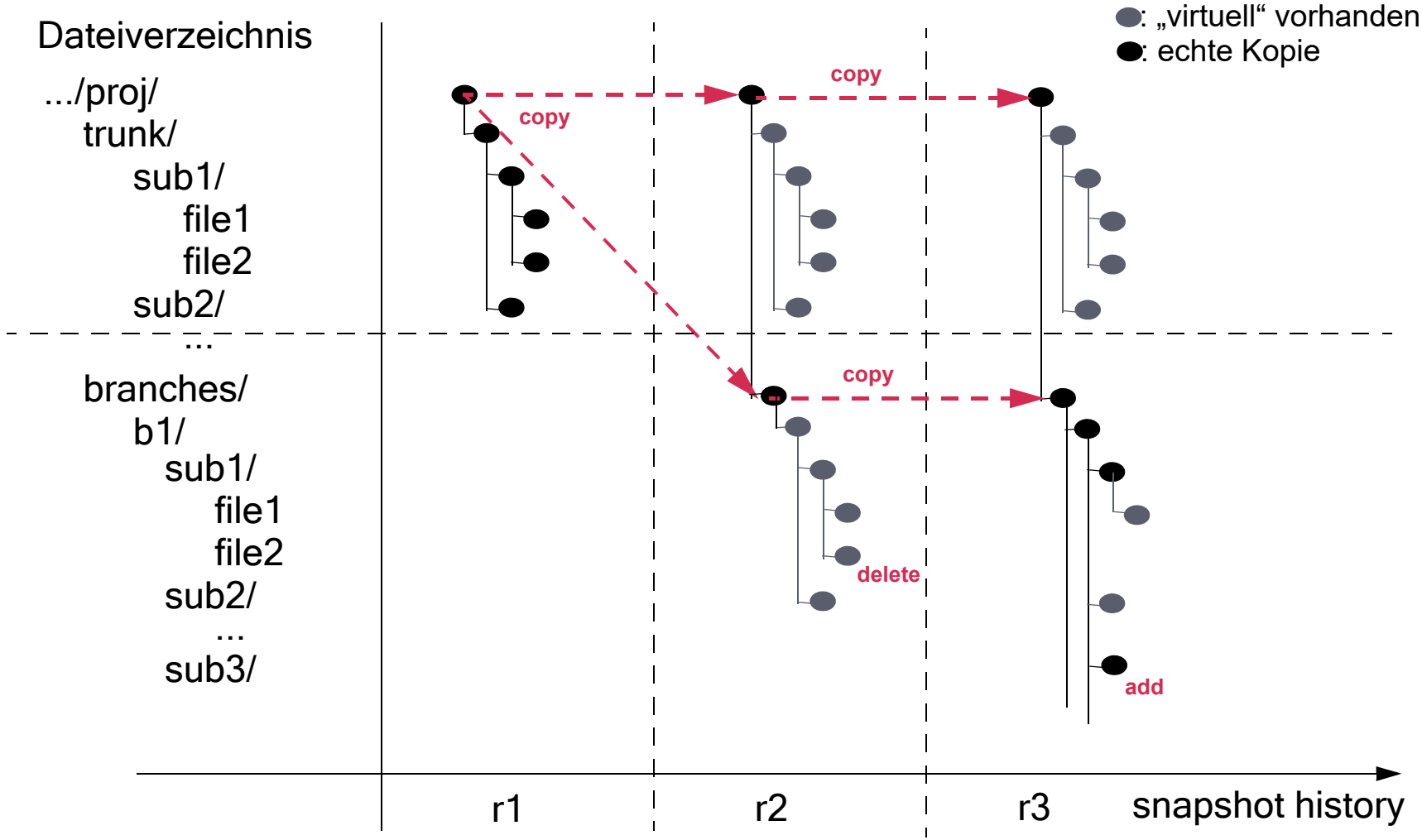
  ⇨ die neue Revision des Hauptzweiges kann anschließend als Snapshot 7 „eingecheckt“ werden

- Anwendung aus Sicht von Wartungszweig 1 (undo changes):

  merge -r 4:3 .../myproject/branches/b1 [wcpath]

  ⇨ eliminiert alle propagierten Änderungen aus Wartungszweig 1 wieder durch Berechnung und Anwendung des inversen Deltas

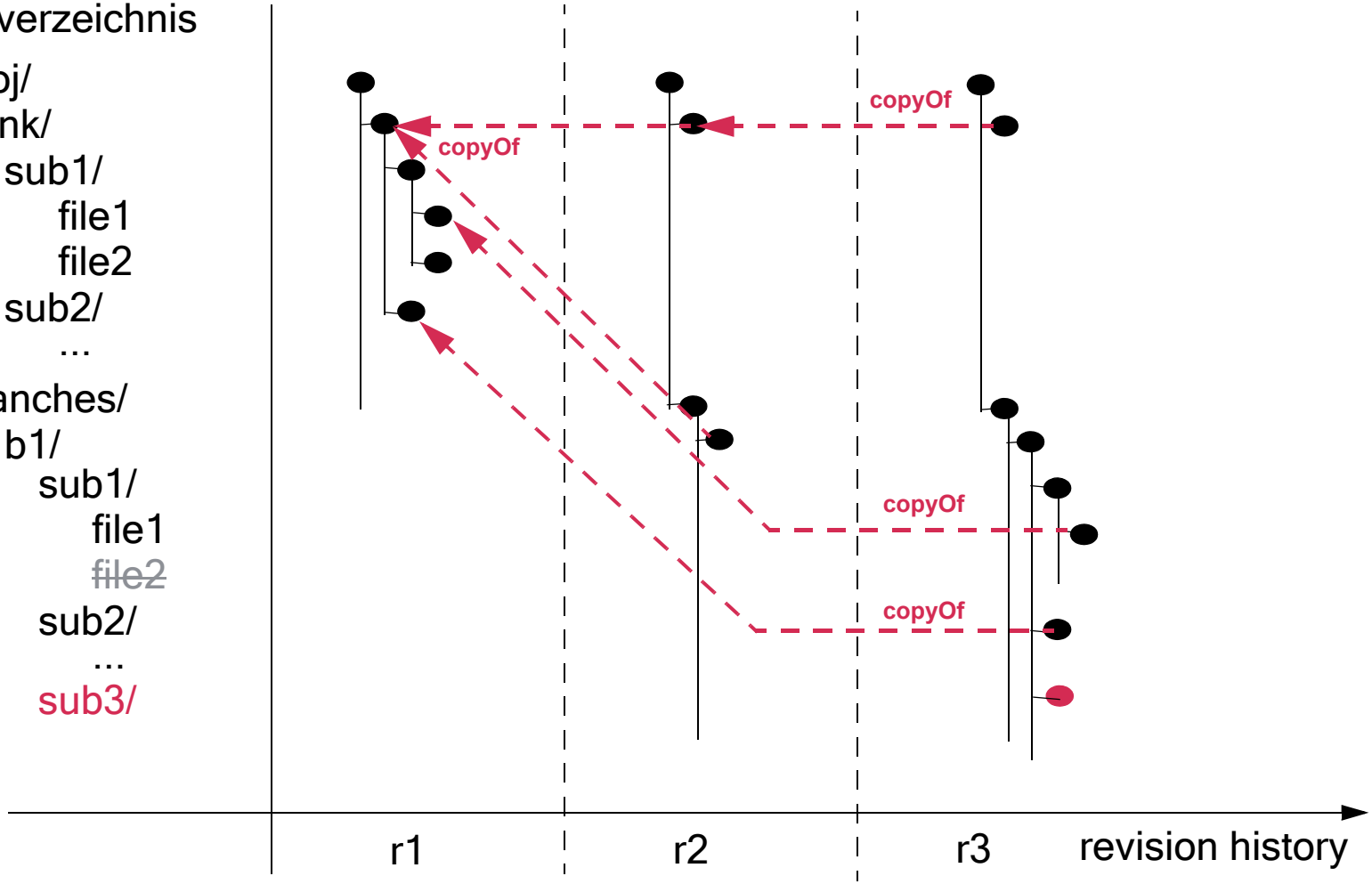
Zweidimensionaler Zustandsraum des Subversion-Repositories:



Zweidimensionaler Zustandsraum des Subversion-Repositories:

Dateiverzeichnis

.../proj/
- trunk/
  - sub1/
    - file1
    - file2
  - sub2/
    - ...
- branches/
  - b1/
    - sub1/
      - file1
      - ~~file2~~
    - sub2/
      - ...
    - sub3/



Delta-Speicherung in svn:

- BDB-Lösung (basierend auf Berkeley Database; älterer Ansatz):
  - ⇨ alles wird in einer Datenbank gespeichert
  - ⇨ Revisionen werden als Rückwärts-Deltas zu Nachfolgern gespeichert (ähnlich wie bei RCS)
- FSFS-Lösung (File System on top of File System; neuerer Ansatz)
  - ⇨ versioniertes Dateisystem, das auf einem „normalen“ Dateisystem aufsetzt
  - ⇨ Revisionen werden als Vorwärts-Deltas zu direkten oder indirekten Vorgängern gespeichert
  - ⇨ die erste Revision ist eine leere Datei
  - ⇨ sogenannter „Skip-List“-Ansatz reduziert Rekonstruktionszeiten durch geschickte Auswahl der Vorgänger

Vorwärts-Delta-Speicherung mit „Skip-List“-Ansatz in Subversion:

Subversion kehrt für die effiziente Speicherung von Revisionen zum „Vorwärtsdelta“-Ansatz von SCCS zurück, benötigt aber bei einer Liste von n aufeinander folgenden Revisionen nur log n Rekonstruktionsschritte. Das funktioniert wie folgt mit r_i = i-te Revision und $d_{i,j}$ = Delta von r_i nach r_j :

- Revision r_0 ist die immer leere Datei.
- Revision r_1 wird durch Anwendung des Deltas $d_{0,1}$ auf r_0 erzeugt.
- Revision r_2 wird durch Anwendung des Deltas $d_{0,2} = d_{0,1}$ & $d_{1,2}$ auf r_0 erzeugt.
- Revision r_3 wird durch Anwendung des Deltas $d_{2,3}$ auf r_2 erzeugt, das seinerseits wiederum durch ... erzeugt wird.
- ...

Achtung:

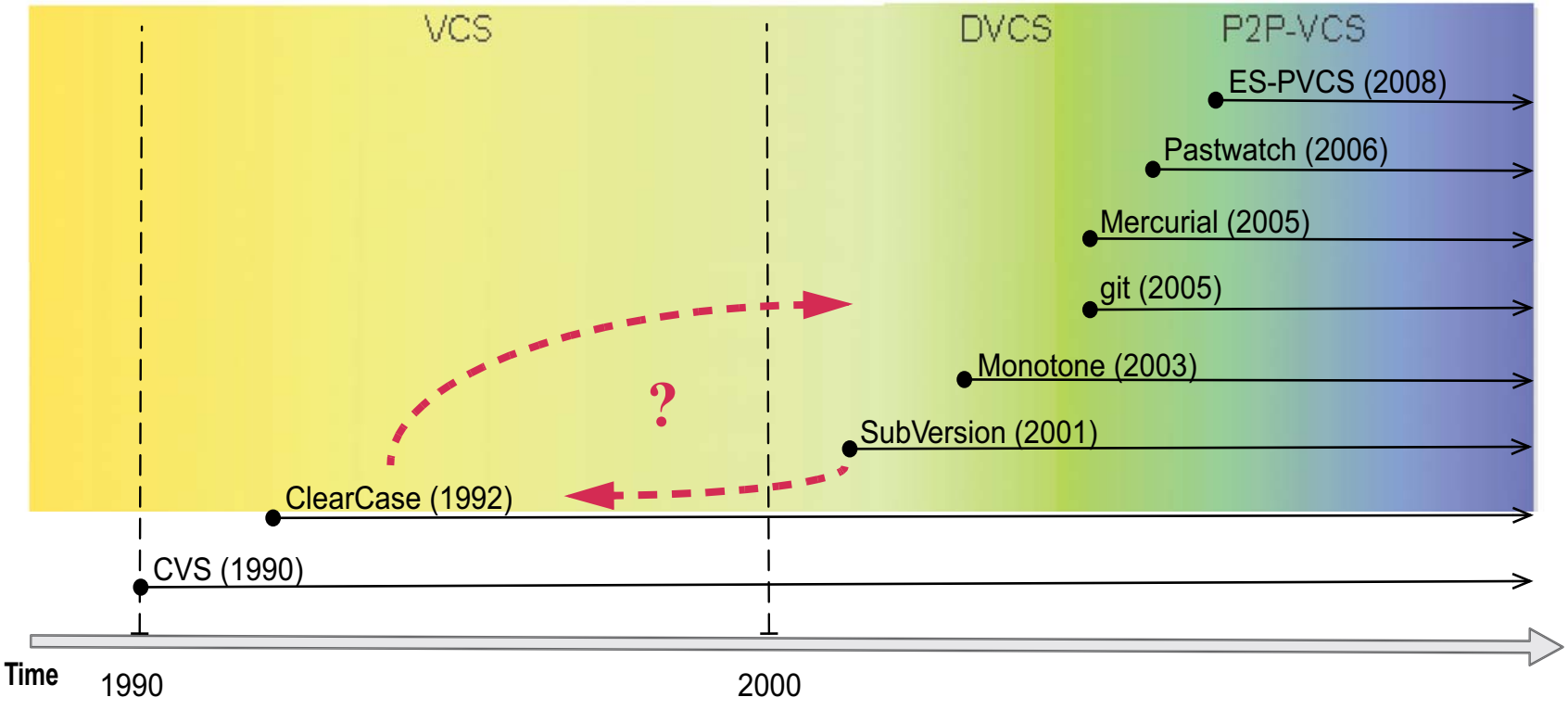
Die Konkatenation $d_{i,j}$ & $d_{j,k}$ zweier Deltas ist oft **deutlich kürzer** als die Summe der Länge der beiden konkatenierten Deltas (aufgrund sich aufhebender Editierschritte).

Vorwärts-Delta-Speicherung mit „Skip-List“-Ansatz - Fortsetzung:

Für die Berechnung des direkten Vorgängers r´ zu einer Revision r wird das letzte 1-Bit der Binärdarstellung von r auf „0“ gesetzt; die sich daraus ergebende Zahl legt r´ fest, auf das Delta $d_{r,r'}$ zur Konstruktion von r angewendet wird (rekursiver Prozess!).

Revisionsnr. von r (Dezimaldarstellung)	Revisionsnr. von r (Binärdarstellung)	Revisionsnr. von r´ (Binärdarstellung)	Revisionsnr. von r´ (fDezimaldarstellung)
0	0000	---	---
1	0001	0000	0
2	0010	0000	0
3	0011	0010	2
4	0100	0000	0
5	0101	0100	4
6	0110	0100	4
7	0111	0110	6
8	1000	0000	0
9	1001	1000	8

Geschichte verteilter Versionierungs-Systeme:



- VCS = Version Control System
- DVCS = Distributed Version Control System

WWW-Seiten einiger DVCS-Systeme:

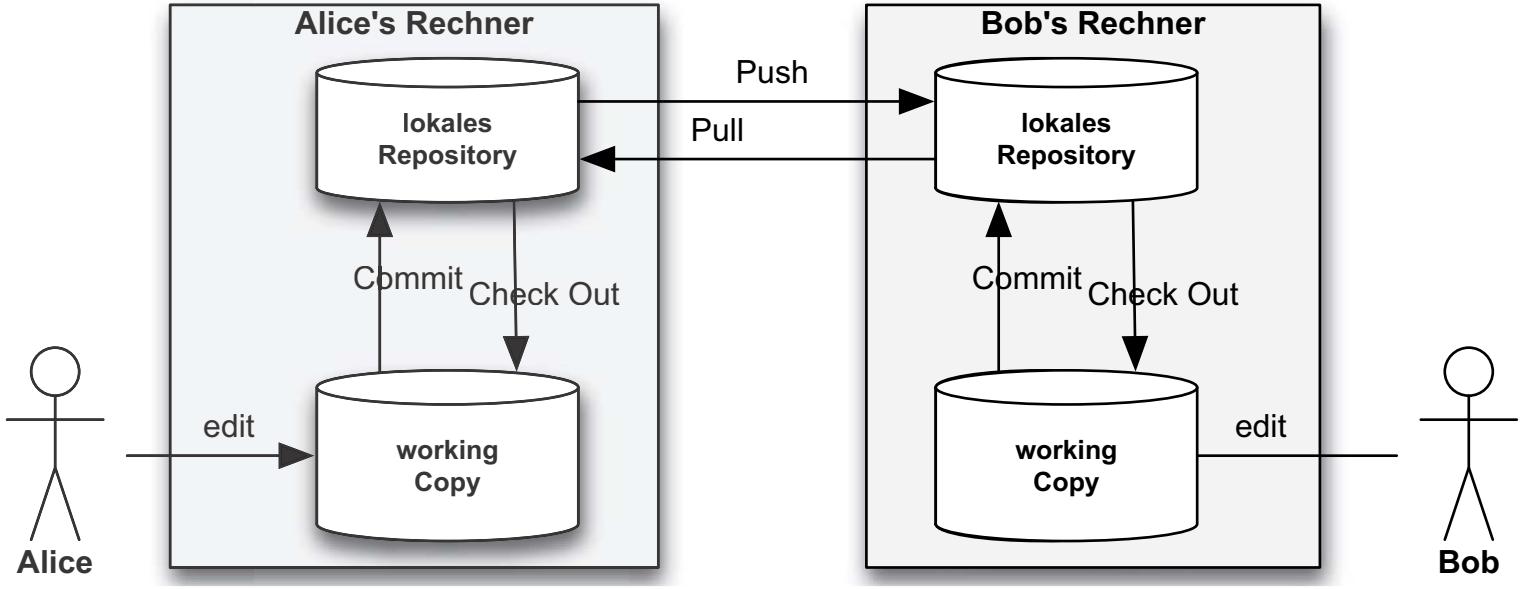
- Monotone: http://monotone.ca/
- Mercurial (aka hg): http://www.selenic.com/mercurial/wiki/
- Bazaar: http://bazaar-vcs.org/
- git: https://git-scm.com/

Anmerkung zu „Urahn“ ClearCase:

Das System ist ein Grenzfall zwischen VCS und DVCS. Es handelt sich zwar um kein reines Client-Server-System mehr, aber die Verteilung auf mehrere Server und deren Kooperation hat doch noch sehr zentralistischen Charakter im Gegensatz zu den „reinen“ DVCS, die auf dem Rechner eines jeden Entwicklers eine eigene Instanz laufen haben.

„Echt“ verteilte Versionierungs-Systeme:

Jeder Benutzer hat sein eigenes vollständiges Repository (statt globale Repositories für Teilgruppen von Entwicklern wie bei ClearCase).



- **Push**: Revisionen zu anderen Repositories aktiv propagieren
- **Pull**: Revisionen von anderen Repositories aktiv holen

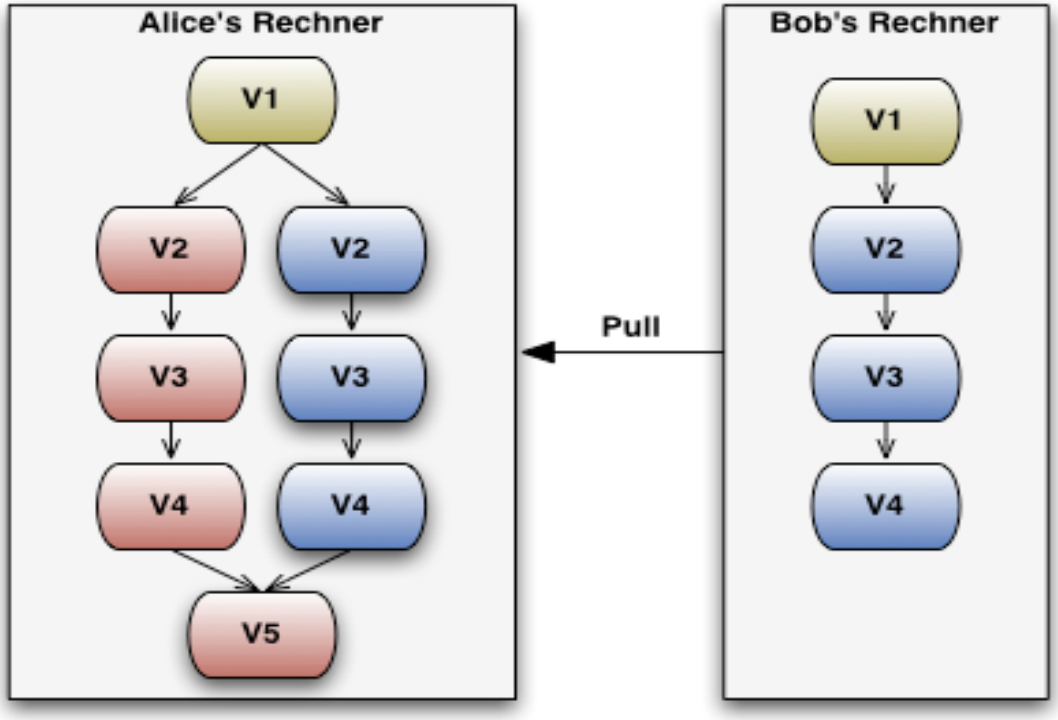
Prinzipien „echt“ verteilter Versionierungs-Systeme wie „git“:

- Jeder Entwickler hat eigenes lokales Versionierungs-Repository mit Snapshots wie bei SVN (Subversion) zusätzlich zu üblichen Arbeitskopien von Dateien
- Check-Out, Commit, ... arbeiten erst mal nur auf eigenen lokalem Repository
- Austausch zwischen Repositories erfolgt mit Push und Pull (zwischen bekannten Personen/Rechnern) via Email oder ssh oder ...
- Bei Push/Pull werden parallele Revisionen angelegt, die dann mit „merge“ zusammengeführt werden (müssen)

Bewertung:

- ☺ Entwickler können auch „offline“ mit eigenem Repository arbeiten
- ☺ Änderungen können zu selbst gewähltem Zeitpunkt integriert werden (merge)
- ☹ Nach Platten-Crash ist lokales Repository weg
- ☹ Hoher Aufwand für Verteilung von Änderungen an andere Entwickler (merge)

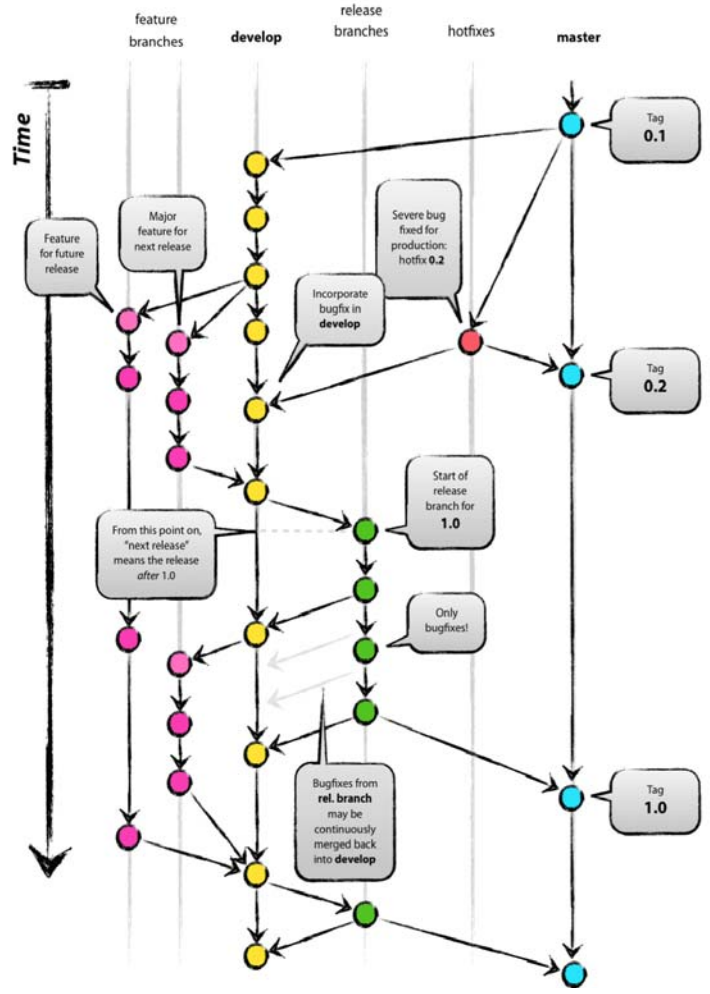
Ablauf der Propagation von Änderungen:



- Änderungen (Revisionen) von Bob werden als paralleler Zweig v. Alice „gepullt“
- Alice „merged“ dann ihre neueste Revision V4 mit neuester Revision V4 von Bob

Mehr zu Versions- und Releasemanagement mit git:

- siehe Folien zum eingeladenen Industrievortrag
- siehe auch:
  A successful Git branching model:
  https://nvie.com/posts/a-successful-git-branching-model/
- Dokumentation von Git:
  https://git-scm.com/doc



Verteilte „Peer-to-Peer“-Versionierung-Systeme:

- **Virtuell** existiert ein **gemeinsames globales Repository** für alle Entwickler
- Es gibt dennoch keinen ausgewählten zentralen Server
- Tatsächlich besitzt jeder Peer ein eigenes Repository (mit allen Dateien oder ausgewählter Menge von Revisionen)
- Das P2P-Versionierungs-System ist „immun“ gegen Ausscheiden einzelner Peers (Revisionen werden auf mehreren Peers gehalten)
- Bei **Netzpartitionierungen** (Spezialfall: einzelner Entwickler arbeitet „offline“) wird je Partition ein eigener Branch angelegt
- Schließen sich Partitionen wieder zusammen, dann werden Branches wieder verschmolzen (im allgemeinen mit Entwicklerhilfe)

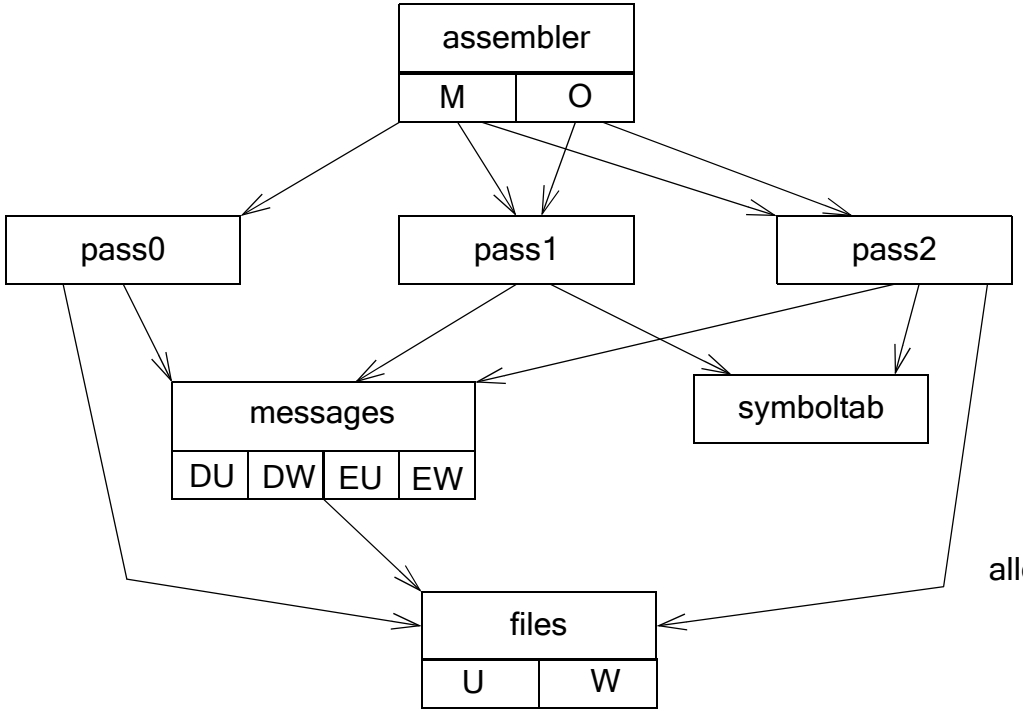
Fazit:

P2P-Versionierungssysteme sollen die Vorteile von VCS und DVCS vereinen!

2.3 Variantenmanagement (Software-Produktlinien)

Das Variantenmanagement befasst sich mit der Verwaltung neben-einander existierender Versionen eines Dokuments, die jeweils eine zeitliche Entwicklungsgeschichte besitzen.

Beispiel:



Legende:

M = mit Makros
O = ohne Makros
D = deutsche Meldungen
E = englische Meldungen
U = für Unix
W = für Microsoft Windows

Modul

Importbeziehung

Denkbare Varianten:

(M,D,U) = mit Makros für Deutsch und Unix
(O,D,U)= ohne Makros für Deutsch, Unix
(M,E,U) = ...

alle Module/Komponenten hängen davon ab

global

Erläuterungen zum Variantenbeispiel (SW-Produktlinie):

- es handelt sich um die Softwarearchitektur eines fiktiven Assemblers, die aus 8 Modulen (Pakete, Komponenten, ...) besteht und in 6 Varianten existiert
- es gibt ein Hauptprogramm assembler, das ein Assemblerprogramm in 2 oder 3 Durchläufen in Maschinencode übersetzt (ohne und mit Behandlung von Makros)
- pass0 expandiert Makros in dem Assemblerprogramm (liest Eingabe aus Datei, schreibt Ausgabe auf Datei)
- pass1 merkt sich Sprungmarken, Konstanten und deren Werte in einer Symboltabelle
- pass2 führt die eigentliche Übersetzung mit Hilfe der Symboltabelle durch
- messages bietet eine Abbildung von Fehlernummern auf Texte in Dateien (Implementierung hängt in kleinen Teilen von Sprache und Betriebssystem ab)
- files bildet die Schnittstelle zum Dateisystem (für Unix und Windows)
- globals sind Konstanten, Funktionen etc., die überall benötigt werden

Software-Produktlinien-Entwicklung = „Variantenmanagement im Großen“:

Eine **Software-Produkt-Linie** (SPL) ist eine Menge „verwandter“ Softwaresysteme

- für eine bestimmte **Anwendungsdomäne**
- die auf Basis einer gemeinsamen **Plattform** (Rahmenwerk)

entwickelt werden. Die Plattform enthält alle SW-Artefakte, die für alle Instanzen der Produktlinie gleich sind. Damit ist die Software-Produktlinien-Entwicklung (SPLE = Software Product Line Engineering) eine logische Verallgemeinerung des Variantenmanagements eines Softwaresystems.

Man unterscheidet beim SPLE zwischen

- der **Domänenentwicklung** (Domain Engineering) der gemeinsamen Plattform (development **for** reuse)
- der **Anwendungsentwicklung** (Application Engineering) von SPL-Instanzen (development **with** reuse)

Domänen- und Anwendungsentwicklung:

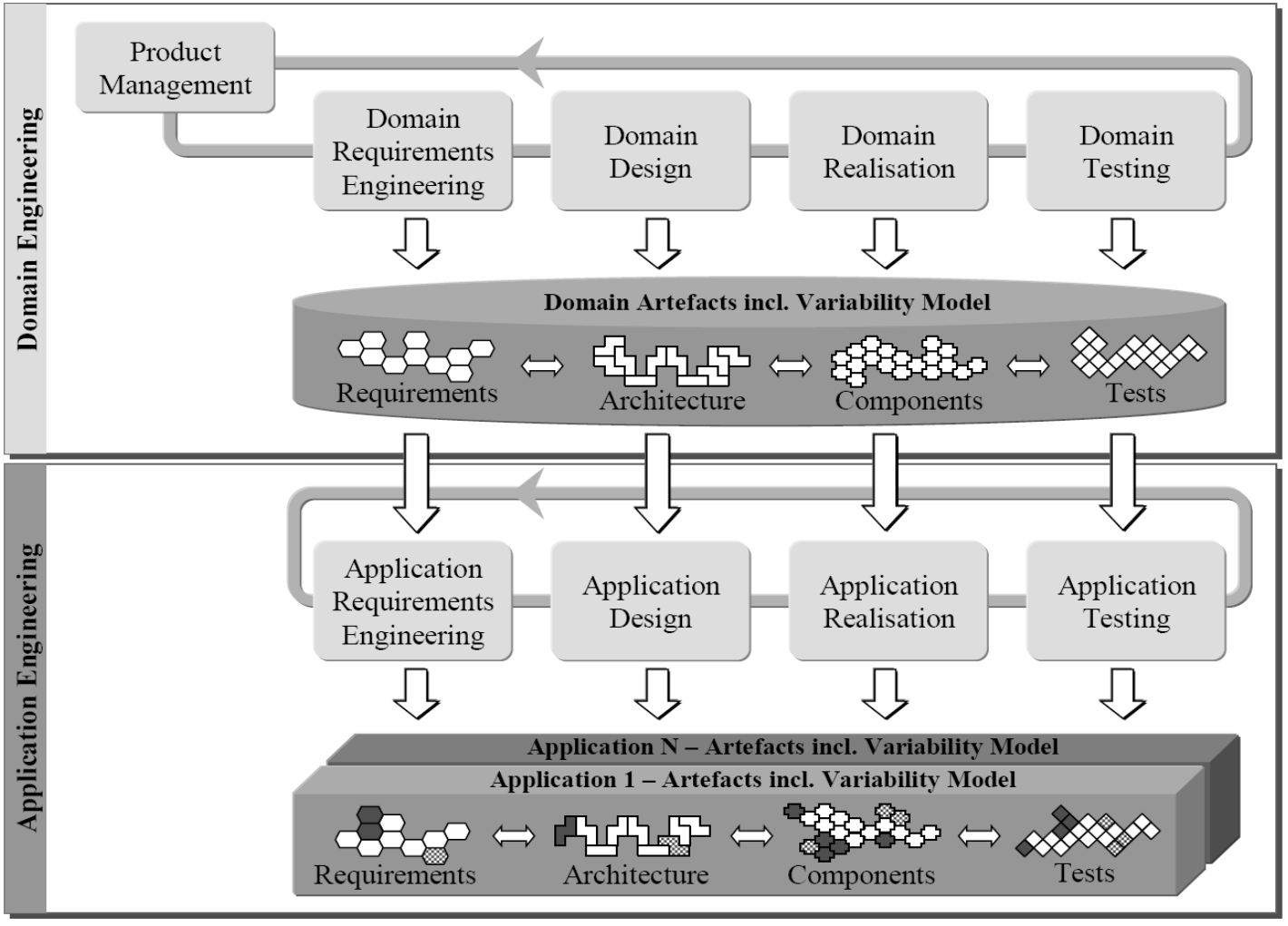


Abb. aus [PBL05]

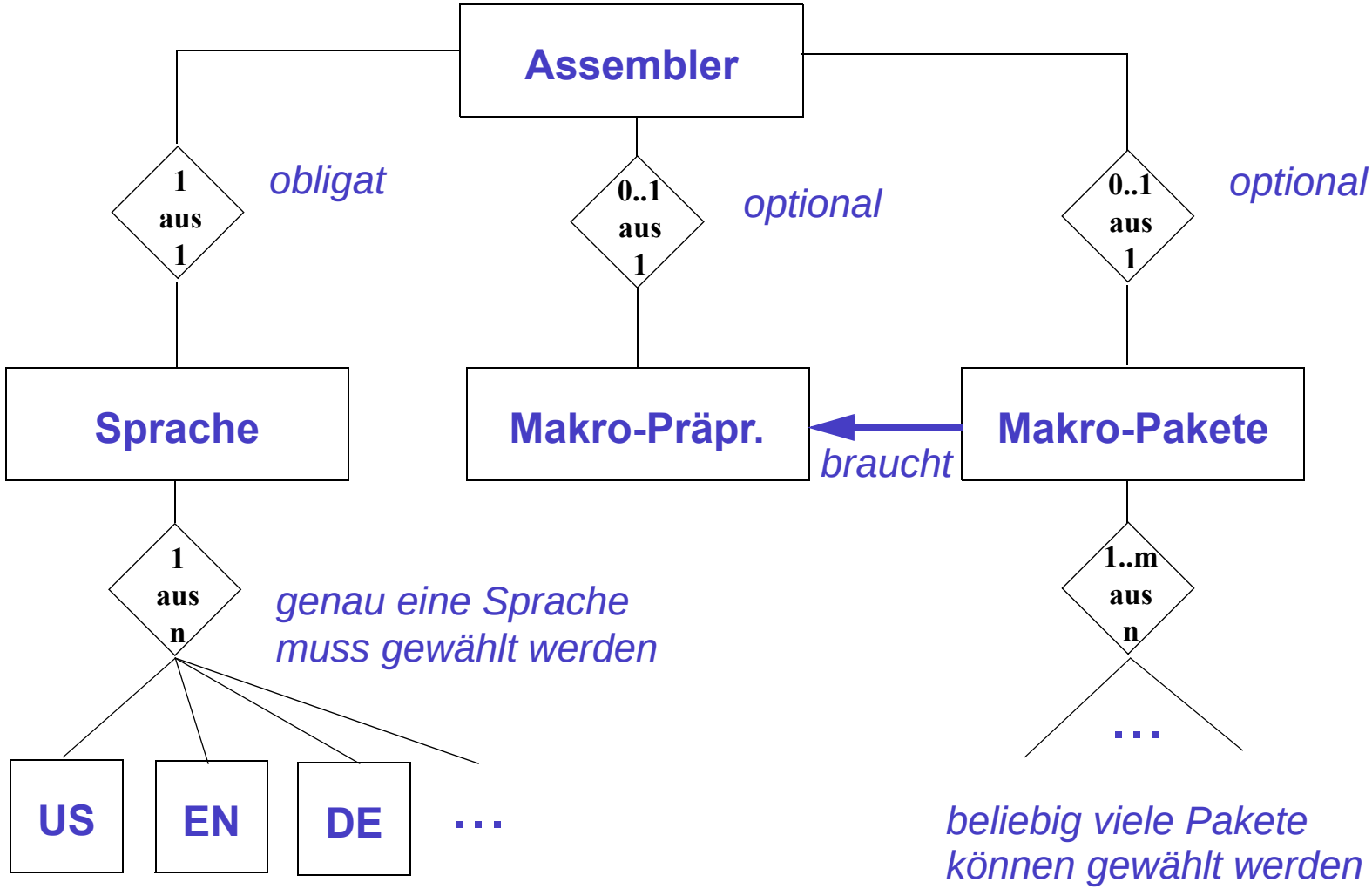
Variabilitätsmodell (Feature-Modell) einer SPL:

So genannte Variabilitäts- oder Feature-Modelle beschreiben alle möglichen Eigenschaften einer Software-Produktlinie und zulässige Kombinationen (Konfigurationen, Varianten, Instanzen) dieser Produktlinie.

Eine Vielzahl von Notationen und Werkzeugen unterstützen die Erstellung solcher Modelle durch

- Festlegung aller auswählbaren Eigenschaften/Merkmale = **Features** der Instanzen der Produktlinie (vor allem die optionalen/alternativen Eigenschaften)
- **hierarchische Zerlegung** von Merkmalen in Untermerkmale
  (das Feature „Dialog-Sprache = Englisch“ wird zerlegt in „US-Englisch“, ...)
- Festlegung von **Auswahl-Optionen** der Art eins-aus-n, m-aus-n, ...
  (eine SPL-Instanz läuft entweder auf Windows oder auf Unix)
- Definition von **Abhängigkeiten** und Ausschlusskriterien einzelner Merkmale in ganz verschiedenen Teilhierarchien

Beispiel eines Variabilitätsmodells (Assembler):



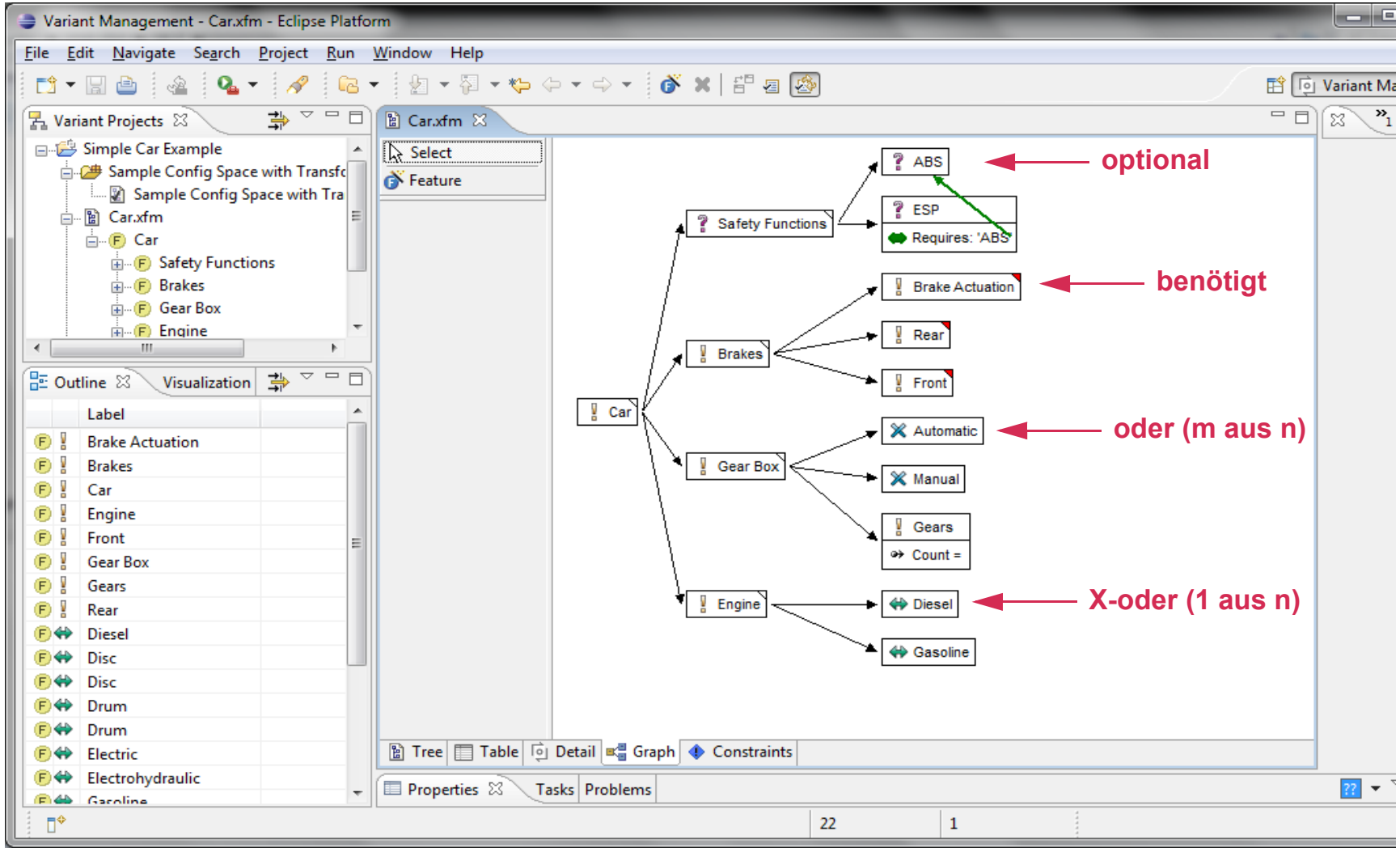
Werkzeugunterstützung für SPL-Entwicklung:

- (grafische) Erstellung von Variabilitätsmodellen
- Unterstützung bei der Auswahl erlaubter Merkmalkombinationen
- Erzeugung von Produktinstanzen (Implementierungen) für bestimmte Merkmalkombinationen
- [Unterstützung beim systematischen Test von SPLs]
- ...

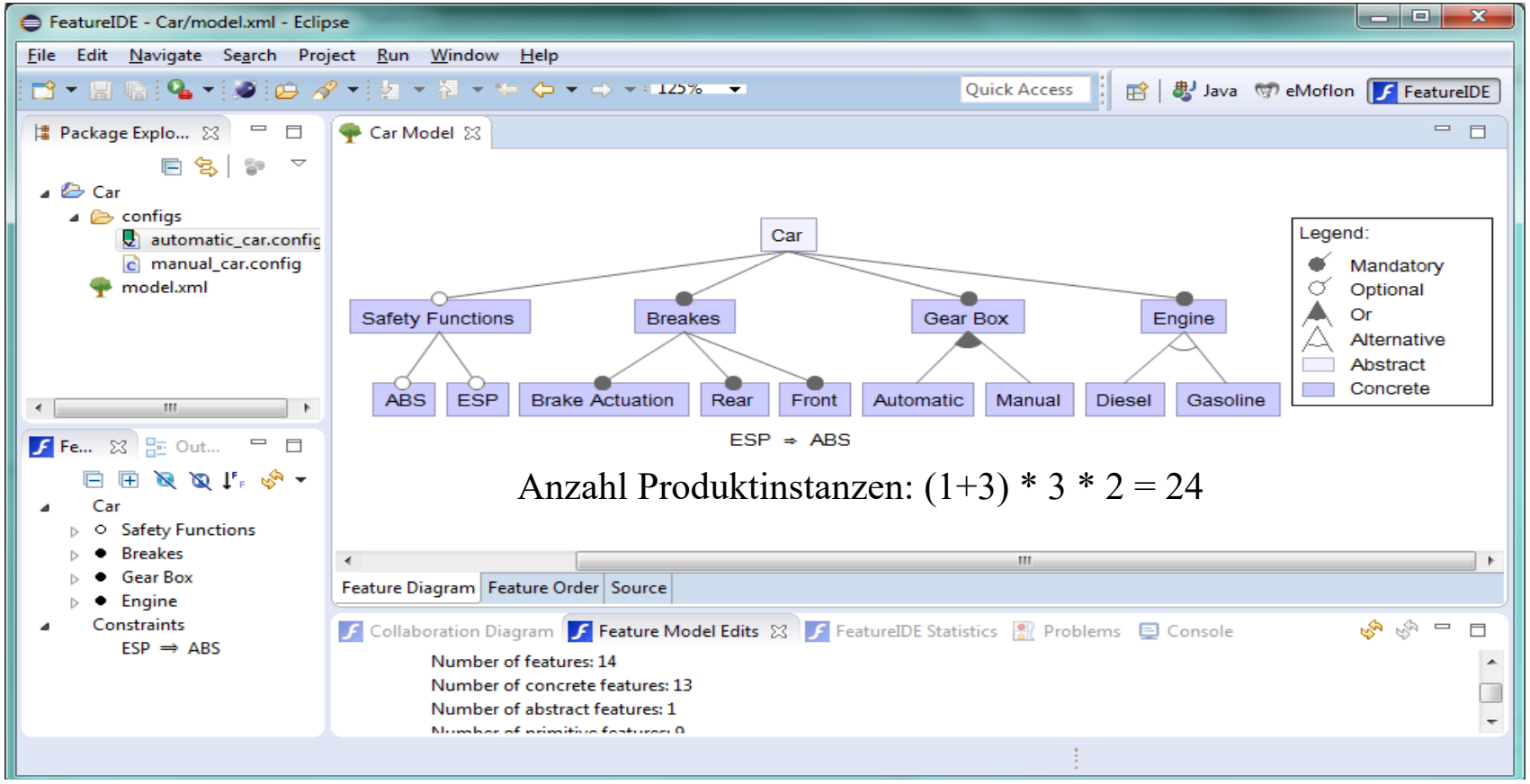
Beispiel-Werkzeuge:

- pure::variants der Firma pure systems (kommerziell)
  (siehe http://www.pure-systems.com/)
- FeatureIDE der Universität Magedburg (public domain)
  (siehe http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/)

Das Produktlinienverwaltungswerkzeug pure::variants:

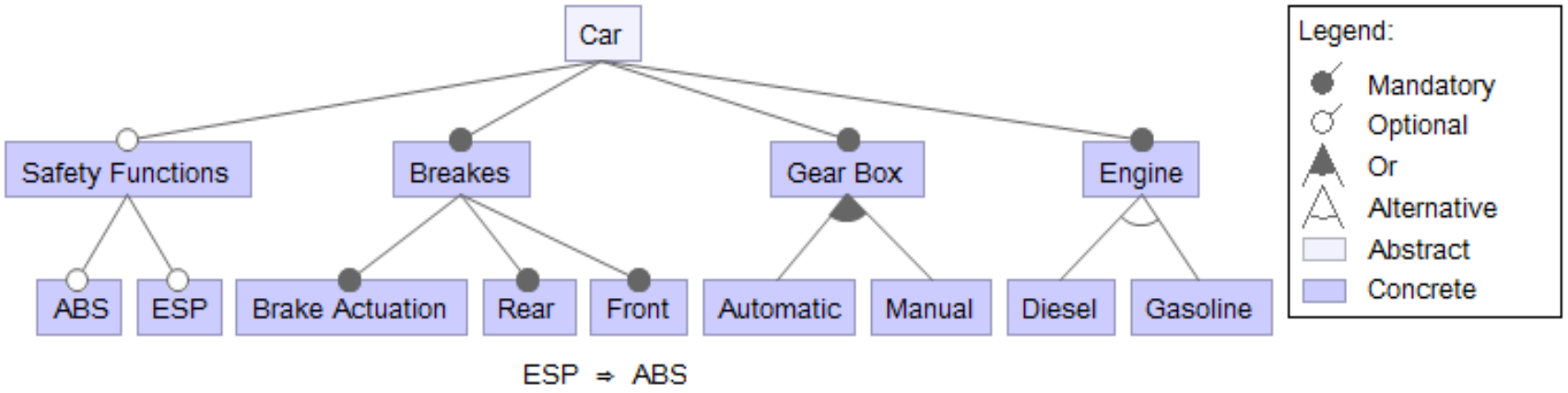


Das Produktlinienverwaltungssystem FeatureIDE:



FeatureIDE ist das „Open Source“-Werkzeug zum Erstellen von Feature-Modellen der internationalen SPL-Forschungsgemeinde.

Grafische Notation von FeatureIDE:



Erläuterung der Legende (Elternfeature der betrachteten Feature sei gewählt):

- Mandatory: ist immer ausgewählt (falls Elternfeature gewählt)
- Optional: kann beliebig an- oder abgewählt werden (falls Elternfeature gewählt)
- Or: mindestens ein Feature ist auszuwählen (falls Elternfeature gewählt)
- Alternative: genau eins muss ausgewählt werden (falls Elternfeature gewählt)
- $X \Rightarrow Y$: wenn X gewählt wird, dann auch Y

Variantenmanagement für Quelltextdateien:

Hierfür gibt es kaum eigene Unterstützung. Im wesentlichen bleiben folgende Möglichkeiten zur Verwaltung verschiedener Varianten eines Dokuments:

- Mit Hilfe der **Versionsverwaltung** wird je Variante/Feature ein eigener Entwicklungszweig gepflegt (mit entsprechendem Tag). Änderungen von einem Zweig werden mit Hilfe der Dreiwegeverschmelzung in andere Zweige propagiert.
- Die verschiedenen Varianten werden alle in einer Datei gespeichert; durch **Bedingungsmakros** (#ifdef Unix ...) werden die für eine Variante benötigten Quelltextteile ein- und ausgeblendet (siehe pure::variants)
- Die verschiedenen Varianten einer Klassenimplementierung werden in Unterklassen ausgelagert (Einsatz von **Vererbung**)
- Benötigte Varianten werden (aus einer domänenspezifischen Beschreibung) **generiert** (mit Quelltexttransformationen, aspektorientierter Programmierung, ...)
- **Achtung**: Auswahl/Konfiguration/Transformation benötigter Quelltextdateien durch das Build-Management (siehe Abschnitt 2.5)

2.4 Releasemanagement

Ein Release ist eine an Kunden ausgelieferte Konfiguration eines (Software-)Systems bestehend aus ausführbaren Programmen, Bibliotheken, Dokumentation, Quelltexte, Installationsskripte, ...

Das Releasemangement dokumentiert ausgelieferte Konfigurationen und stellt deren Rekonstruierbarkeit sicher.

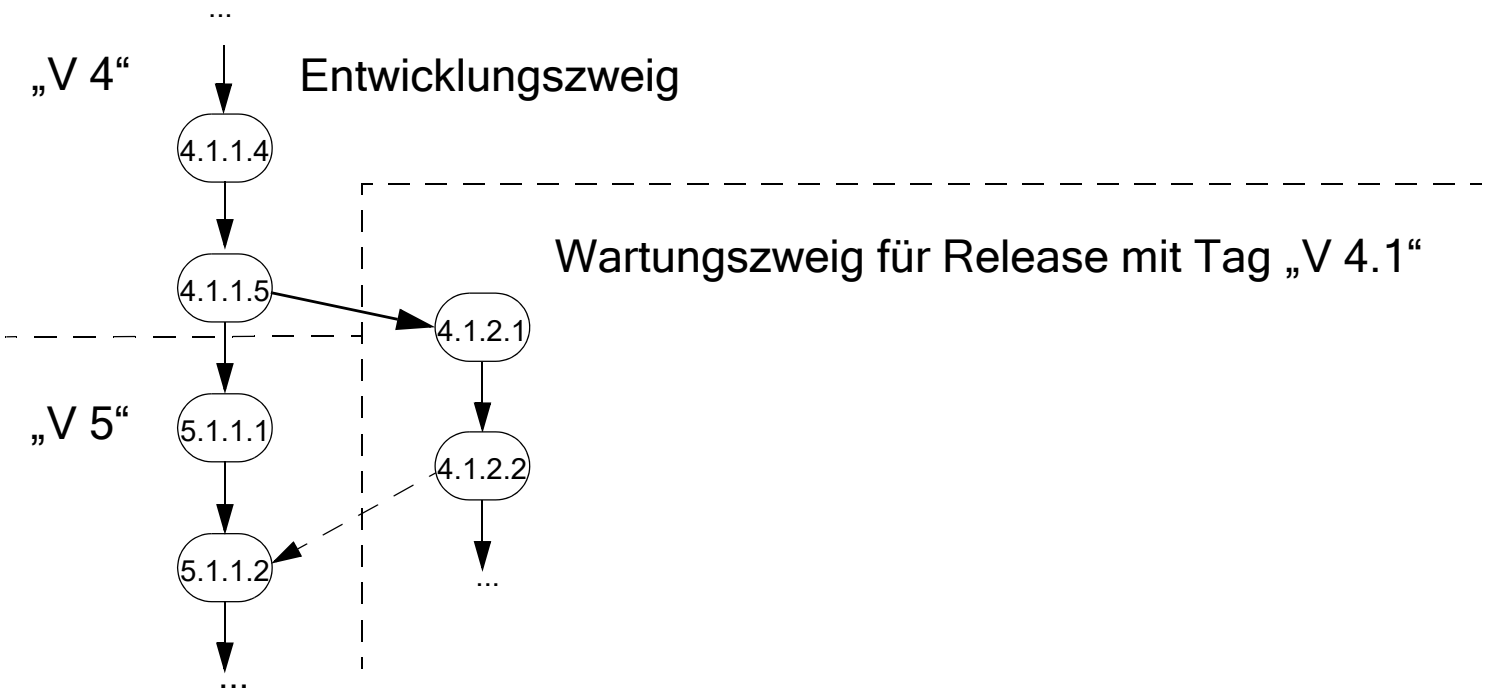
Aufgaben des Releasemanagement:

- Festlegung der (zusätzlichen) Funktionalität eines neuen Releases
- Festlegung des Zeitpunktes der Freigabe eines neuen Releases
- Erstellung und Verbreitung eines Releases (siehe auch Buildmanagement)
- Dokumentation des Releases:
  - ⇨ welche Revisionen welcher Dateien sind Bestandteil des Releases
  - ⇨ welche Compilerversion wurde verwendet
  - ⇨ Betriebssystemversionen auf Entwicklungs- und Zielplattform

Planungsprozess für neues Release:

1. Vorbedingungen für neues Release werden überprüft:
   - ⇨ viel Zeit vergangen (neues Release aus Publicity-Gründen)
   - ⇨ viele Fehler behoben (neues Release mit allen „Patches“)
   - ⇨ viele neue Funktionen hinzugefügt
2. Weiterentwicklung wird eingefroren (freeze):
   - ⇨ **Feature Freeze** (Soft Freeze): nur noch Fehlerkorrekturen und kleine Verbesserungen erlaubt (nur vermutlich nicht destabilisierende Änderungen)
   - ⇨ **Code Freeze** (Hard Freeze): nur noch absolut notwendige Änderungen, selbst „gefährliche“ Fehlerkorrekturen werden verboten
3. Letzte Fehlerkorrekturen und umfangreiche Qualitätssicherungsmaßnahmen (Tests) werden durchgeführt
4. Release wird freigegeben und weiterverbreitet:
   - ⇨ Weiterentwicklung (neuer Releases) wird wieder aufgenommen
   - ⇨ freigegebenes Release muss parallel dazu gepflegt werden

Ältere Standardlösung für parallele Pflege und Weiterentwicklung:

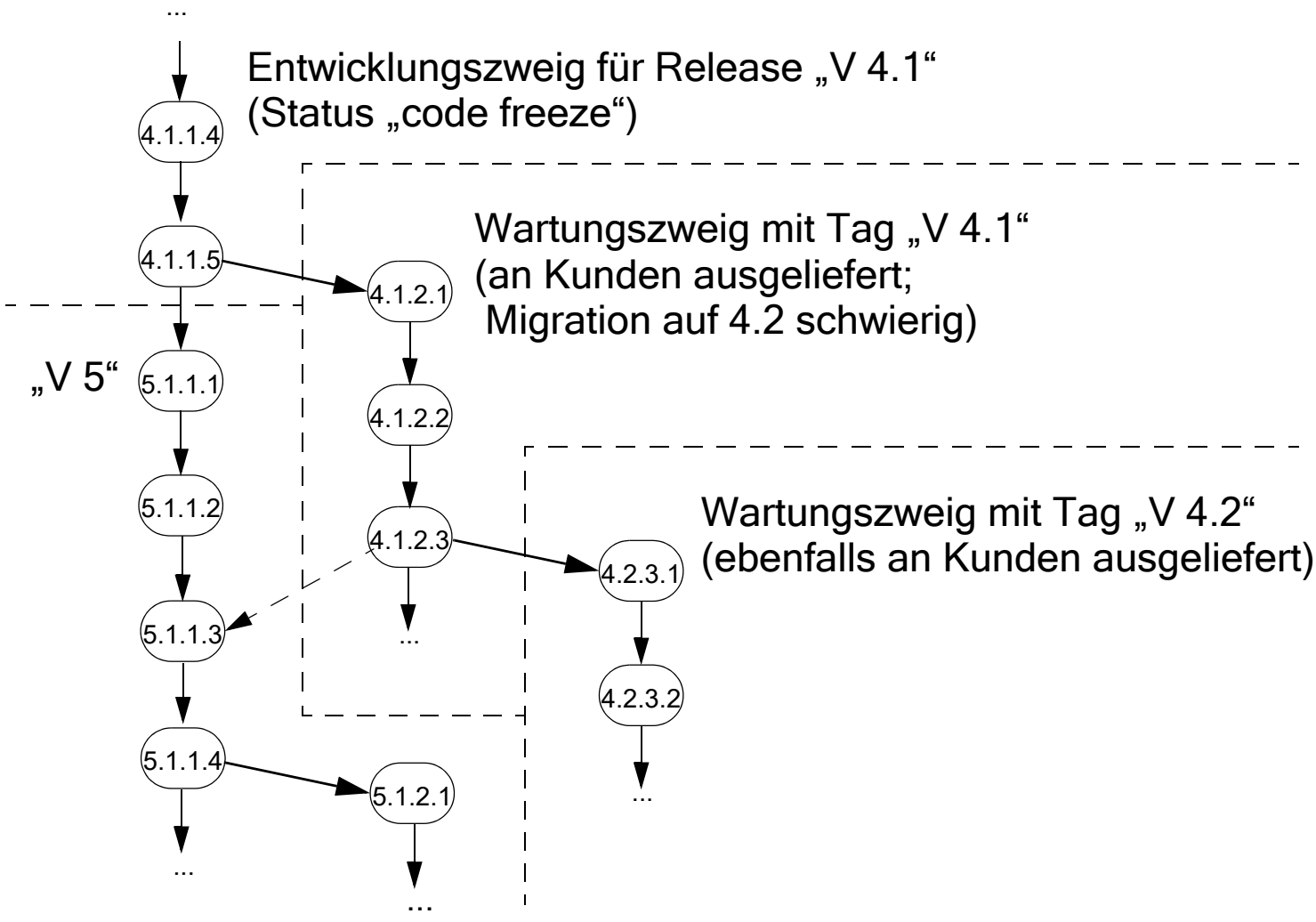


- für Weiterentwicklung des nächsten Releases und Wartung des gerade freigegebenen Releases werden unterschiedliche Revisionszweige verwendet
- alle auf dem Wartungszweig liegenden Revisionen erhalten den Namen des Releases (Versionsnummer) als Tag

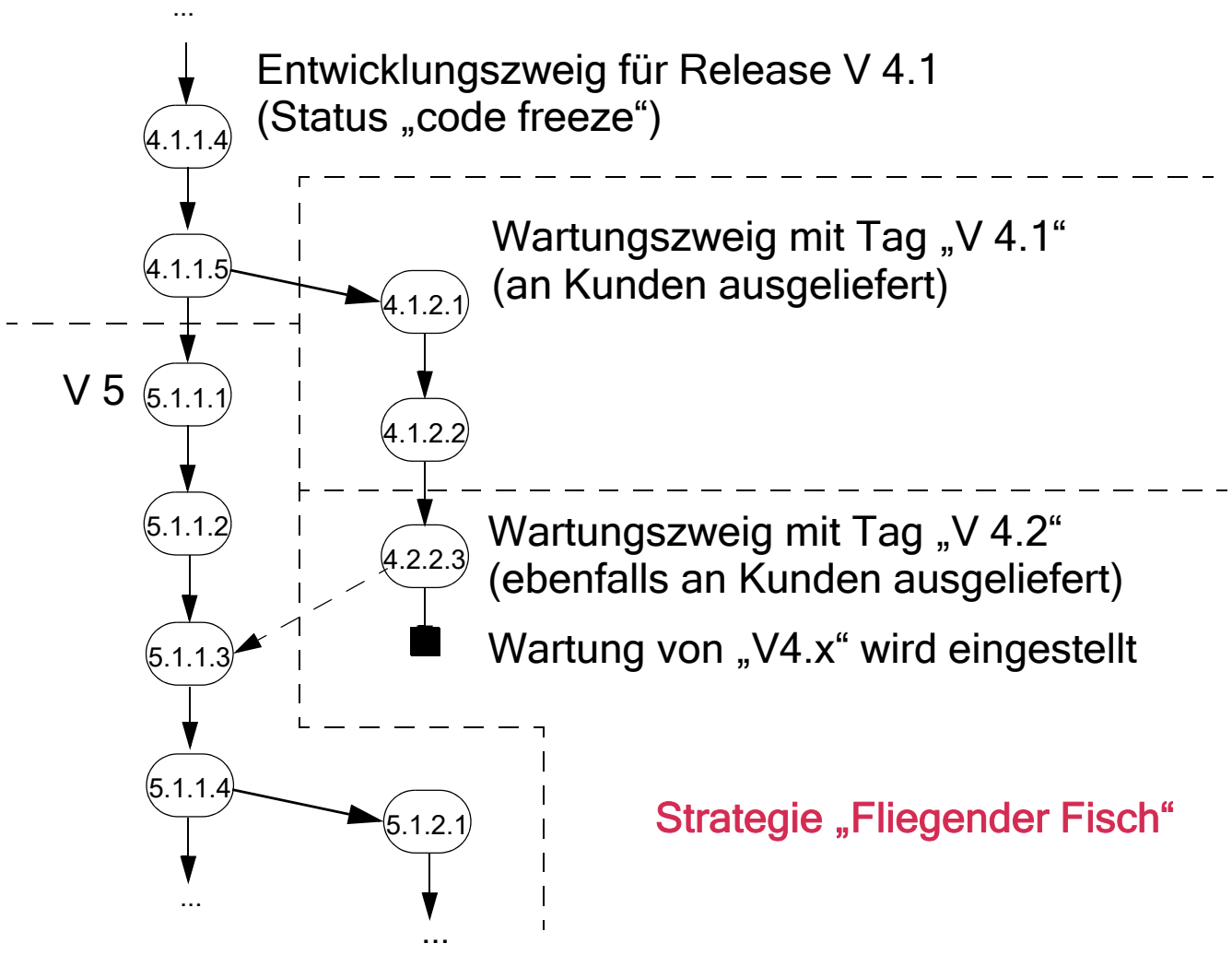
Strategien für die Vergabe von Versions- und Revisionsnummern:

- Software-Releases erhalten üblicherweise zweistellige **Versionsnummern**
- die erste Stelle wird erhöht, wenn sich die Funktionalität signifikant ändert
- die zweite Stelle wird für kleinere Verbesserungen erhöht
- oft wird erwartet, dass x.2, x.4, ... stabiler als x.1, x.3, ... sind
- die (in cvs vierstelligen) internen **Revisionsnummern** eines KM-Systems müssen mit den extern sichtbaren Versionsnummern nichts zu tun haben
- in den vorigen Beispielen galt aber:
  Versionsnummer = ersten beiden Stellen der Revisionsnummer
- oft werden jedoch die ersten beiden Stellen der Revisionsnummern (in cvs) nicht verwendet und bleiben auf „1.1“ gesetzt

Hauptentwicklungszweig und mehrere Wartungsnebenzweige:

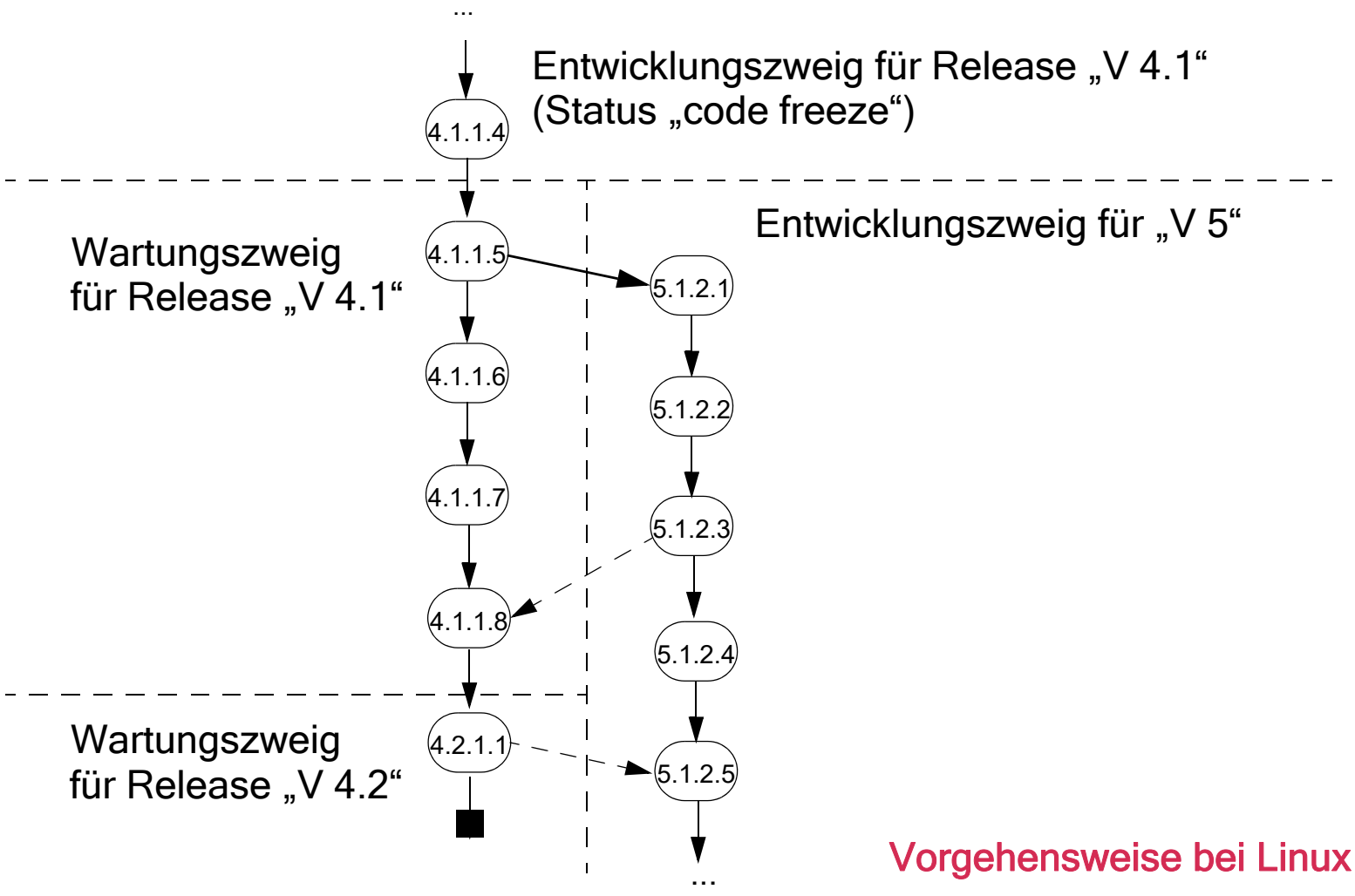


Hauptentwicklungszweig und ein Wartungsnebenzweig:



Strategie „Fliegender Fisch“

Inversion von Wartungs- und Entwicklungszweig:

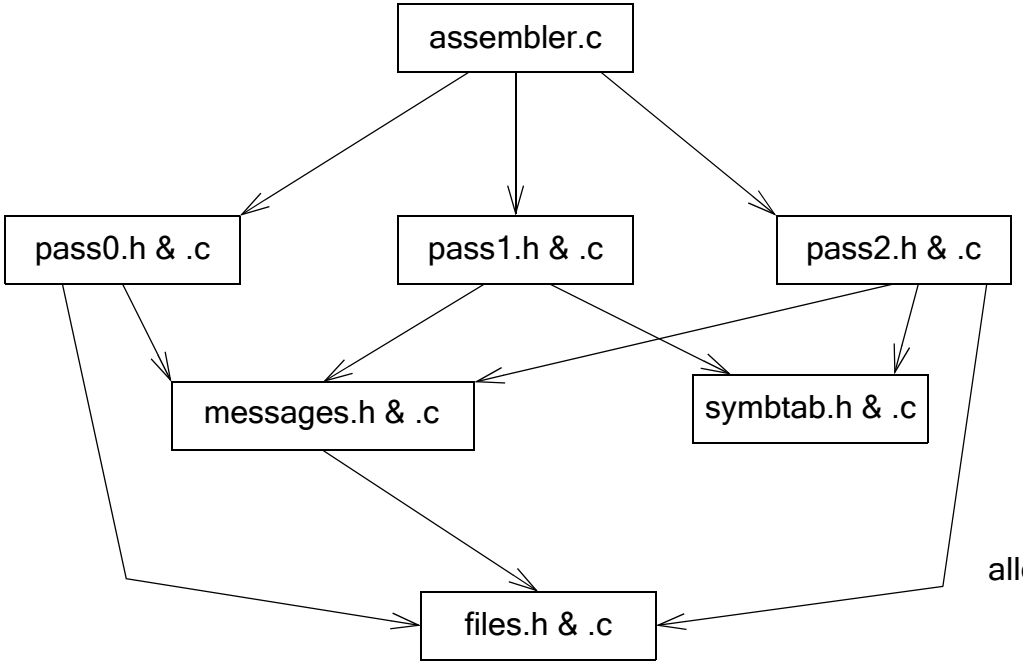


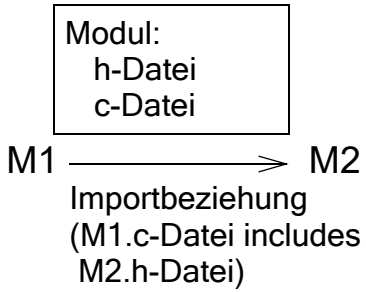
Vorgehensweise bei Linux

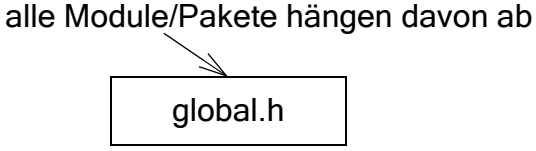
2.5 Buildmanagement

Das Buildmanagement (Software Manufacturing) automatisiert den Erzeugungsprozess von Programmen (Software Releases).

Altes Beispiel - einfache Assemblerimplementierung in C:



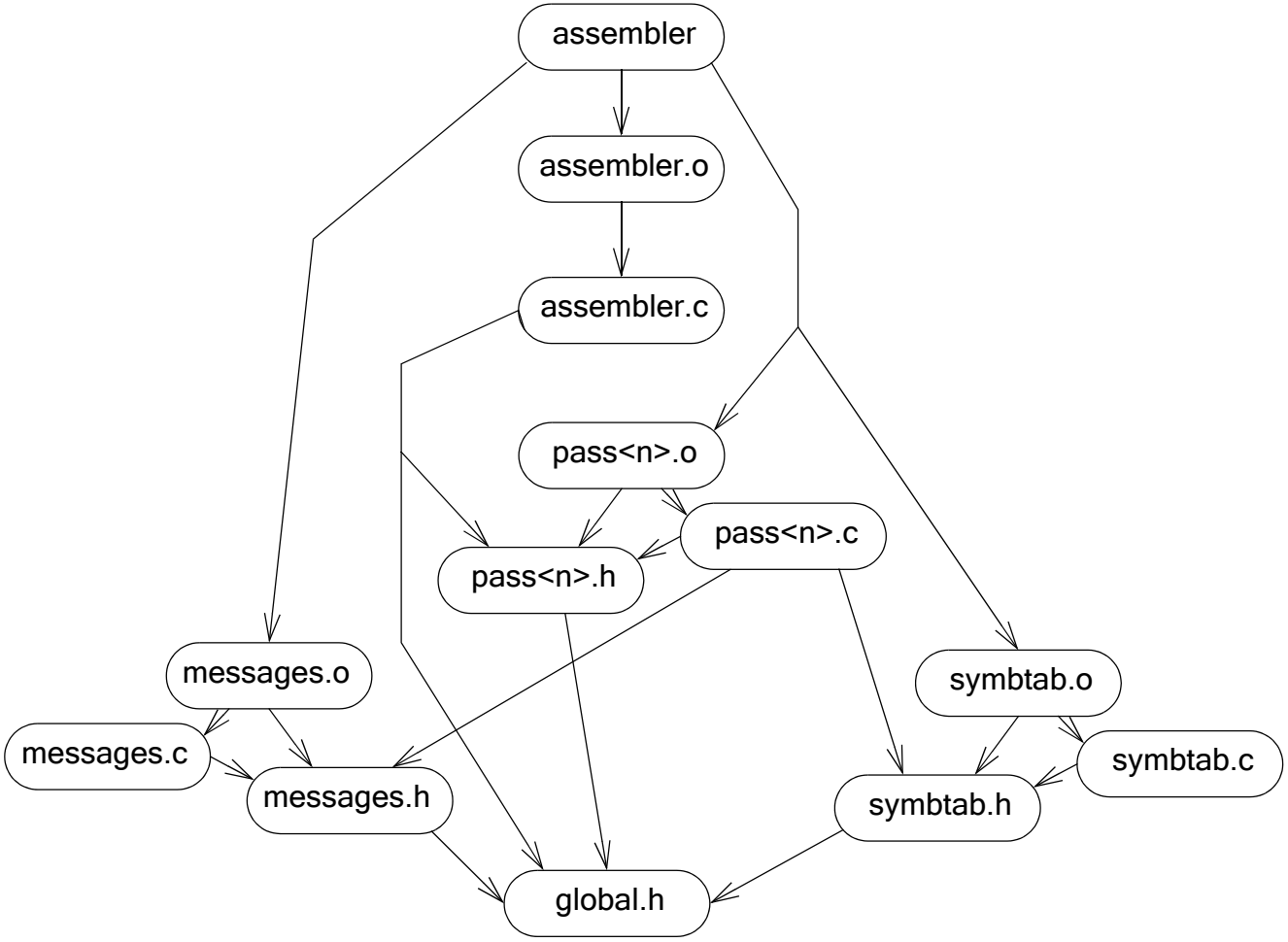




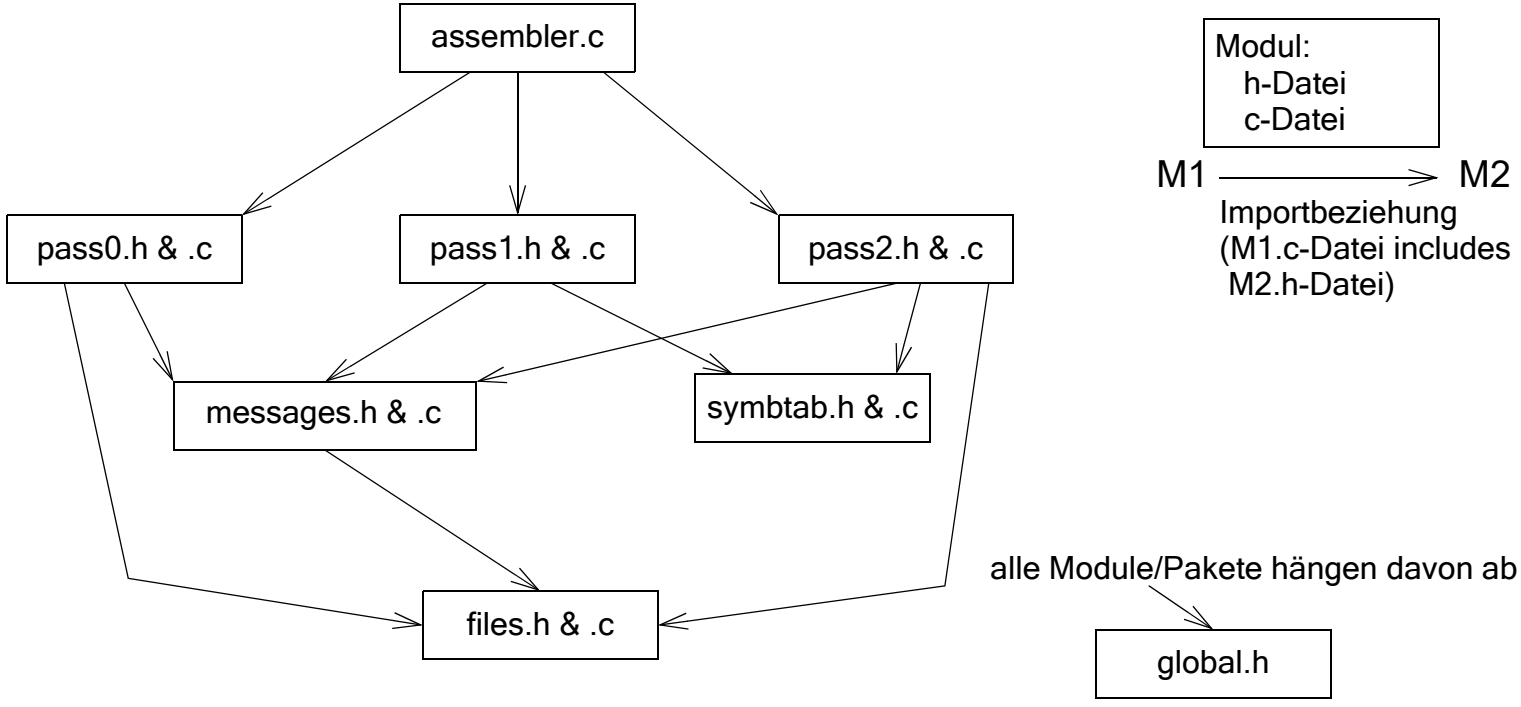
Erläuterungen zum Beispiel - Implementierung in C:

- jedes „normale“ Modul M besteht aus zwei Textdateien:
  - ⇨ die **Header-Datei** M.h beschreibt die Schnittstelle des Moduls, die von anderen Modulen benötigt wird (Konstanten, Typen, Prozedurdefs.)
  - ⇨ die **Implementierungsdatei** M.c enthält die C-Quelltexte der Prozeduren
- das Modul global besteht nur aus einer Textdatei global.h:
  - ⇨ die Datei enthält überall benötigte Konstanten- und Typdefinitionen
  - ⇨ „überall“ = in allen anderen .h- und .c-Dateien
- das Hauptprogramm assembler besitzt keine Schnittstelle für andere Module; es besteht also nur aus einer .c-Datei
- jede Quelltextdatei M.c mit **Suffix .c** wird in eine Objektdatei M.o übersetzt; dabei werden .h-Dateien aller importierten Module verwendet
- alle Objektdateien mit **Suffix .o** werden zu einem ausführbaren Programm zusammengebunden

Altes Beispiel - genauerer Abhängigkeitsgraph:

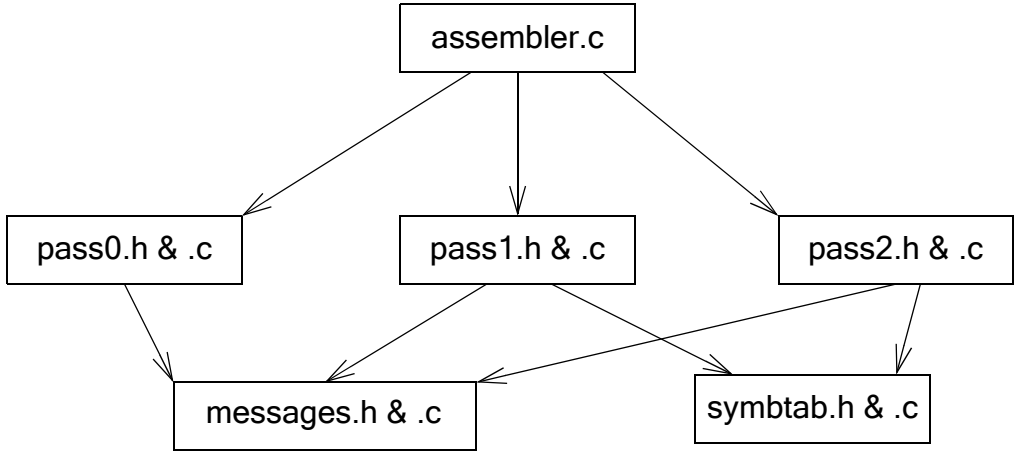


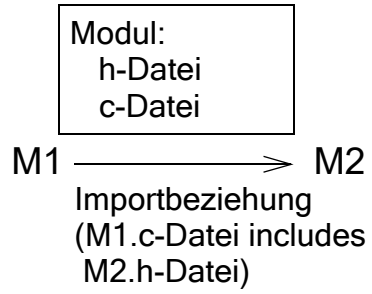
Übersetzungs- und Bindevorgänge - Beispiel 1:



1. die Implementierung einer Prozedur von messages = messages.c ändert sich
2. nur messages.c muss neu in messages.o übersetzt werden: cc -c messages.c
3. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

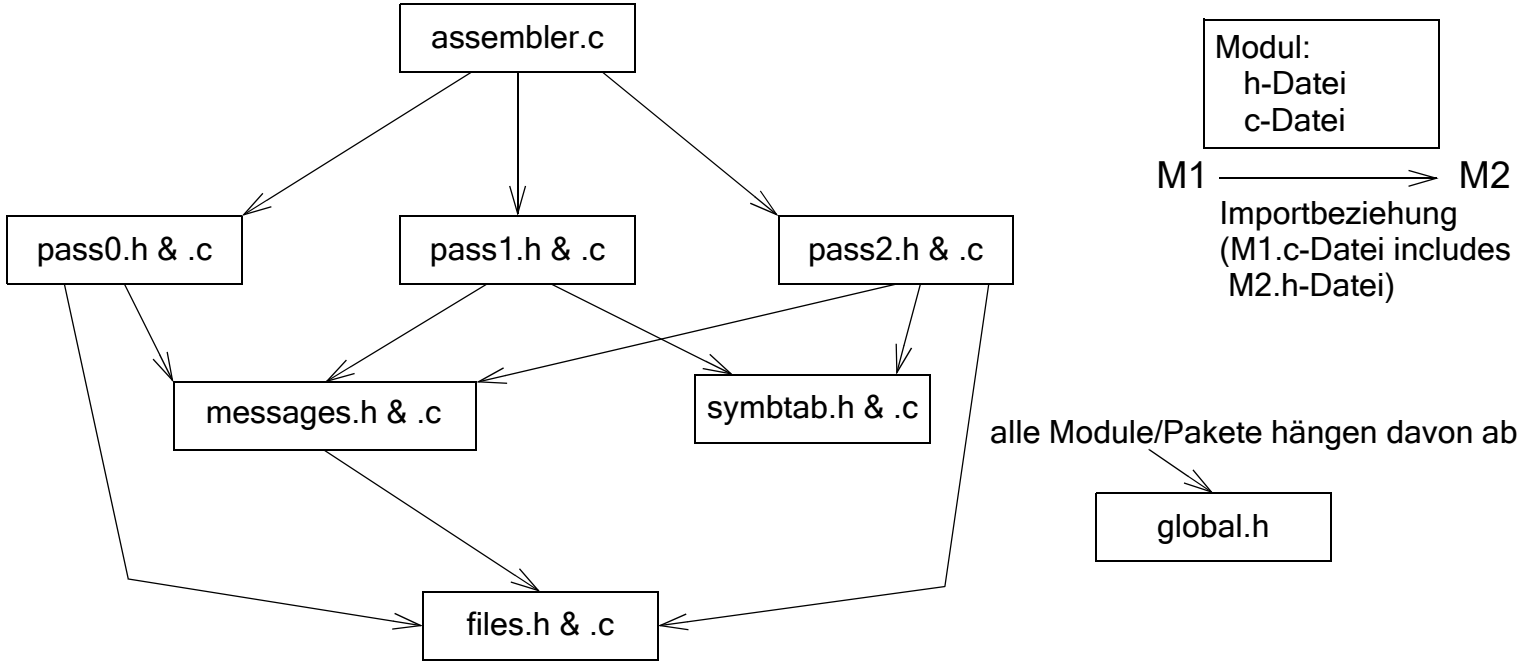
Übersetzungs- und Bindevorgänge - Beispiel 2:





1. Schnittstelle (Typdefinition) von messages = messages.h ändert sich
2. [ggf. muss man auch messages.c ändern]
3. messages.c muss neu in messages.o übersetzt werden:
   cc -c messages.c
4. pass0.c, pass1.c und pass2.c müssen neu übersetzt werden
   cc -c pass0.c | cc -c pass1.c | cc -c pass2.c
5. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

Übersetzungs- und Bindevorgänge - Beispiel 3:



1. in global.h wird eine Konstante geändert
2. alle .c-Dateien müssen in beliebiger Reihenfolge (parallel) neu übersetzt werden:
   cc -c files.c | cc -c messages.c | cc -c symbtab.c | cc -c pass0.c | ...
3. Hauptprogramm assembler muss neu aus allen o-Dateien erzeugt werden:
   cc -o assemb assemb.o pass0.o pass1.o pass2.o messages.o files.o

Werkzeuge für das Build-Management (siehe auch [Po09], [Wie11]):

- Das Werkzeug „**make**“ (im folgenden im Detail präsentiert):
  - ⇨ deklarativer, regelbasierter Ansatz
  - ⇨ es werden Abhängigkeiten zwischen Entwicklungsartefakten definiert (mit zusätzlichen Kommandos zur Erzeugung abgeleiteter Artefakte)
  - ⇨ unabhängig v. Entwicklungsprozess, Programmiersprachen, Werkzeugen
- Das Werkzeug „**Ant**“ (make-Nachfolger im Java-Umfeld):
  - ⇨ gemischt deklarativer/imperativer Ansatz
  - ⇨ es werden Abhängigkeiten zwischen Aufgaben (Tasks) definiert
  - ⇨ Tasks können mit Kontrollstrukturen „ausprogrammiert“ werden
  - ⇨ verwendet XML-Syntax für Konfigurationsdateien
  - ⇨ in Java und vor allem für Java-Projekte entwickelt
  - ⇨ aber: unabhängig von Entwicklungsprozess, Projektstruktur, ...

Werkzeuge für das Build-Management - 2:

- Das Werkzeug „**Maven**“:
  - ⇨ deklarativer Ansatz mit relativ einfachen Konfigurationsdateien
  - ⇨ macht viele Annahmen über Entwicklungsprozess, Projektstruktur, ...
  - ⇨ keine frei definierbaren Regeln für bestimmte Sprachen, Werkzeuge sondern stattdessen Plugins
  - ⇨ Fokus liegt auf der Unterstützung von Java-Projekten
  - ⇨ holt sich Plugins, Libraries, ... aus zentralen Repositories
  - ⇨ oft kombiniert mit „Nexus“ für Einrichten lokaler (Cache-)Repositories
- Das Werkzeug „**Jenkins**“ (früher: „Hudson“):
  - ⇨ ergänzt Build-Werkzeuge wie Ant und Maven
  - ⇨ unterstützt die kontinuierliche Integration/Auslieferung von Produkten
  - ⇨ automatische Integration von Build-, Test-, Reporting-Werkzeugen

Das Programm „make“ zur Automatisierung von Build-Vorgängen:

- make wurde in den 70er Jahren „nebenbei“ von Stuart F. Feldman für die Erzeugung von Programmen unter Unix programmiert
- Varianten von make gibt es heute auf allen Betriebssystemplattformen (oft integriert in oder spezialisiert auf Compiler einer Programmiersprache)
- make werden durch ein sogenanntes makefile Erzeugungsabhängigkeiten zwischen Dateien mitgeteilt
- make benutzt **Zeitmarken** um festzustellen, ob eine Datei noch aktuell ist (Zeitmarke jünger als die Zeitmarken der Dateien, von denen sie abhängt)

Aufbau einer Abhängigkeitsbeschreibung:

```
ziel : objekt1 objekt 2 objekt3 ...
        Kommando zur Erzeugung von ziel aus objekt1 ...
        #Achtung Kommandozeile muss mit <tab> = Tabulator beginnen
```

Abhängigkeitsbeschreibungen für unser Beispiel:

```
assembler : assembler.o \
            pass0.o pass1.o pass2.o pass3.o \
            messages.o symbtab.o \
            files.o
    cc -o assembler   assembler.o pass0.o pass1.o pass2.o pass3.o \
                      messages.o symbtab.o files.o

assembler.o : assembler.c global.h pass0.h pass1.h pass2.h pass3.h
    cc -c assembler.c

pass0.o : pass0.h pass0.c global.h messages.h
    cc -c pass0.c

...
```

Achtung:

Das Zeichen „\“ muss immer letztes Zeichen einer fortgesetzten Zeile sein!

Aufruf von make ohne Parameter nach Änderung von messages.c:

1. der Aufruf sucht eine Datei namens makefile und soll eine aktuelle Version des Ziels der ersten Abhängigkeitsbeschreibung erzeugen
2. erstes Ziel ist assembler, also wird überprüft ob diese Datei noch aktuell ist
3. Dafür wird zunächst rekursiv überprüft, ob alle o-Dateien, von denen assembler abhängt, aktuell sind (wenn sie bereits existieren)
4. ...
5. messages.o hängt nur von Dateien ab, für die es keine eigenen Regeln gibt. Deshalb bricht der rekursive Prozess ab und es wird nur die Zeitmarke von messages.o mit den Zeitstempeln von messages.c, ... verglichen
6. messages.o hat eine ältere Zeitmarke als messages.c und wird deshalb durch Übersetzung neu erzeugt
7. assembler besitzt nun eine ältere Zeitmarke als messages.o und wird deshalb neu mit dem Binder (Linker) erzeugt

Makefiles mit mehreren Zielen:

```
assembler-unix :    assembler.o \
                    pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-unix.o
        cc -o assembler assembler.o pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-unix.o
        # make oder make assembler-unix führt den obigen Bindevorgang aus

assembler-windows : assembler.o \
                    pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-windows.o
        cc -o assembler assembler.o pass0.o pass1.o pass2.o pass3.o \
                    messages.o symbtab.o files-windows.o
        # make assembler-windows führt den obigen Bindevorgang aus

cleanup :
     rm  assembler.o pass0.o pass1.o pass2.o pass3.o \
         messages.o symbtab.o files.o
        # make cleanup löscht immer o-Dateien, da Datei cleanup nicht existiert
```

Besseres Makefile mit selbstdefinierten Makros:

```
CC = cc          # Aufruf des C-Compilers
CFLAGS = -c      # Flag für Übersetzung
LD = cc          # Name des Binders (hier in C-Compiler integriert)
LDFLAGS = -o     # Flags für Binden
DEBUG =          # Debugoption deaktiviert; Aktivierung als -g

assembler :      assembler.o \
                 pass0.o pass1.o pass2.o pass3.o \
                 messages.o symbtab.o \
                 files.o
    ${LD} ${DEBUG} ${LDFLAGS}  assembler assembler.o \
                               pass0.o pass1.o pass2.o pass3.o \
                               messages.o symbtab.o files.o

assembler.o : assembler.c global.h pass0.h pass1.h pass2.h pass3.h
    ${CC} ${DEBUG} ${CFLAGS} assembler.c
```

Weniger redundantes Makefile mit sogenannten internen Makros:

```
...

assembler : $@.o   pass0.o pass1.o pass2.o pass3.o \
                   messages.o symbtab.o files.o
                   ${LD} ${DEBUG} ${LDFLAGS} $@ $^

assembler.o : $*.c  global.h pass0.h pass1.h pass2.h pass3.h
                    ${CC} ${DEBUG} ${CFLAGS} $*.c
```

Erläuterung der seltsamen Sonderzeichen-Makros:

- $@ bezeichnet immer das Ziel einer Regel
- $^ bezeichnet alle Objekte einer Regel rechts vom „:“ (z.B. assembler.o ...)
- $* bezeichnet immer das Ziel einer Regel ohne Suffix
  (assembler.o wird also zu assembler verkürzt)
- $? bezeichnet alle Objekte einer Regel rechts vom „:“, die neuer als das Ziel sind
- $< bezeichnet das erste Objekt rechts vom „:“

Muster-Suche und Substitutionen in make:

Oft benötigt man auf der rechten Seite einer Regel oder in der Kommandozeile alle Dateinamen (im aktuellen Verzeichnis), die einem bestimmten **Namensmuster** entsprechen (also etwa mit einem bestimmten Suffix enden).

```
$(wildcard Muster)
```

liefert bzw. **sucht** alle Dateien im aktuellen Verzeichnis, die dem angegebenen Muster entsprechen. Das Muster kann beispielsweise folgende Form haben:

```
$(wildcard *.c)
```

liefert alle Dateien zurück, die das Suffix „.c“ besitzen (der „*“ matcht alles).

Will man in einer Liste von Dateinamen (durch Leerzeichen getrennte Strings) einen Teilstring durch einen anderen Teilstring **ersetzen** (substituieren) verwendet man:

```
$(patsubst Suchmuster, Ersatzstring, Liste von Wörtern)
```

Folgender Ausdruck ersetzt alle „.c“-Suffixe durch „.o“-Suffixe:

```
$(patsubst %.c, %.o, Liste von Wörtern)
```

Das Zeichen „%“ bezeichnet den gleichbleibenden Teil bei der Ersetzung.

Einsatz von Muster-Suche und -Substitution:

```
...

h-files = $(wildcard *.h)
# alle h-Dateien im aktuellen Verzeichnis werden der Var. h-files zugewiesen

o-files = $(patsubst %.c, %.o, $(wildcard *.c))
# alle c-Dateien im aktuellen Verzeichnis werden gefunden und mit
# ausgetauschtem Suffix (c wird gegen o getauscht) o-files zugewiesen

assembler : ${o-files}
    ${LD} ${DEBUG} ${LDFLAGS}  $@ $^

assembler.o :$*.c  ${h-files}
    ${CC} ${DEBUG} ${CFLAGS} $*.c
```

Muster-Regeln in make:

Manchmal will man Regeln schreiben, die alle Dateien mit einem bestimmten Suffix aus einer anderen Datei mit unterschiedlichem Suffix aber ansonsten gleichem Namen errechnen. So wird beispielsweise jede c-Datei in eine o-Datei ansonsten gleichen Namens übersetzt. Die Suffix-Regel ist für die Definition solcher Abhängigkeiten geeignet:

```
prefix1%suffix1 : prefix2%suffix2
    Kommando
```

Beispiel für Suffix-Regel:

Übersetzung von c-Dateien in o-Dateien geht wie folgt:

```
...

%.o : %.c ${h-files}
    ${CC} ${DEBUG} ${CFLAGS} $<
```

Ein letztes Beispiel für makefiles:

Es soll ein makefile geschrieben werden, das folgende Anforderungen erfüllt:

- beim Aufruf von make ohne Parameter sollen im aktuellen Verzeichnis alle Dateien mit der Endung gif in Dateien mit der Endung jpg übersetzt werden; dafür wird das Kommando convert verwendet
- das Konvertieren soll nur passieren, wenn die jpg-Datei zu einer gif-Datei noch nicht existiert oder einen älteren Zeitstempel besitzt
- alle jpg-Dateien mit zugehöriger gif-Datei werden zu einer Datei (einem Archiv) images.zip zusammengepackt mit dem Kommando zip
- die zip-Datei wird nur erzeugt, wenn vorher mindestens eine jpg-Datei neu erzeugt wurde (nur neue jpg-Dateien werden im Archiv ausgetauscht)
- für ein einfaches makefile können sie davon ausgehen, dass das aktuelle Verzeichnis nur die Dateien tobias.gif, johannes.gif, markus.gif und andy.jpg enthält
- ein „fortgeschrittenes“ makefile muss für beliebig viele gif- und jpg-Dateien funktionieren

Einfaches makefile für bekannte Dateien:

```
default : images.zip

images.zip : tobias.jpg johannes.jpg markus.jpg
        zip -u images.zip tobias.jpg johannes.jpg markus.jpg

tobias.jpg : tobias.gif
        convert tobias.gif tobias.jpg

johannes.jpg : johannes.gif
        convert johannes.gif johannes.jpg

markus.jpg : markus.gif
        convert markus.gif markus.jpg
```

Besseres makefile für beliebige Dateien:

```
jpg-files = $(patsubst %.gif, %.jpg, $(wildcard *.gif))

default : images.zip

images.zip : ${jpg-files}
        zip -u $@ $^

%.jpg : %.gif
        convert $? $@
```

Variante für inkrementelle Aktualisierung von images.zip:

Will man nur die veränderten jpg-Dateien neu in das Archiv aufnehmen, dann ist eine Regel wie folgt zu ändern:

```
images.zip : ${jpg-files}
        zip -u $@ $?
```

Bewertung des Buildmanagements mit make:

- nur ein Bruchteil der Funktionen von make wurde vorgestellt
- ursprüngliches make führt Erzeugungsprozesse sequentiell aus
  - ⇨ GNU make kann Arbeit auf mehrere Rechner verteilen und parallel durchführbare Erzeugungsschritte gleichzeitig starten
- Steuerung durch Zeitmarken ist sehr „grob“:
  - ⇨ **zu häufiges neu generieren**: irrelevante Änderungen (wie Ändern von Kommentaren für Übersetzung) werden nicht erkannt
  - ⇨ **zu seltenes neu generieren**: Änderung von Übersetzerversionen, Übersetzungsoptionen etc. werden nicht erkannt
- kaum Verzahnung mit Versionsverwaltung:
  - ⇨ make wird in aller Regel auf eigenem Repository durchgeführt
  - ⇨ erzeugte/abgeleitete Objekte werden also immer privat gehalten
- makefiles selbst müssen manuell aktuell gehalten werden
  - ⇨ programmiersprachenspezifisches makedepend erzeugt makefiles

Erzeugung von Makefiles:

1. in jedem Quelltextverzeichnis gibt es ein „normales“ Makefile, das den Übersetzungsprozess mit make in diesem Verzeichnis steuert
2. „normale“ Makefiles werden von makedepend durch Analyse von Quelltextdateien erzeugt (in C werden includes von .h-Dateien gesucht)
3. ein „Super“-Makefile sorgt dafür, dass
   - ⇨ makedepend nach relevanten Quelltextänderungen in den entsprechenden Verzeichnissen aufgerufen wird
   - ⇨ in allen „normalen“ Makefiles dieselben Makrodefinitionen verwendet werden (soweit gewünscht)
   - ⇨ geänderte Makefiles oder Makrodefinitionen dazu führen, dass in den betroffenen Verzeichnissen alle abgeleiteten Objekte neu erzeugt werden

Achtung:

- ☞ Makefile-Erzeugung ist eine „Wissenschaft“ für sich, wenn viele verschiedenen Übersetzer und Generatoren in einem Projekt verwendet werden

Generierung von Makefiles mit GNU Autotools [Ca10]:

Die GNU **Autotools** sind mehrere Werkzeuge, die aufbauend auf make das Build-Management, also die Erstellung und Installation von Softwarekonfigurationen für verschiedenste Zielplattformen, erleichtern.

- Fokus liegt auf Softwareprojekten, in denen vor allem C, C++ oder Fortran (77) als Programmiersprachen eingesetzt werden
- die Werkzeuge eignen sich nicht (kaum) für Java-Projekte oder SW-Entwicklung im Windows-Umfeld
- **Automake** unterstützt Generierung „guter“ makefiles aus deutlich kompakteren Konfigurationsdateien (die von allen populären make-Varianten ausführbar sind)
- **Autoconf** unterstützt Konfigurationsprozess von Software durch Generierung von config.h-Skeletten, Konfigurationsskripten, ...
- schließlich gibt es noch **Libtool**, das Umgang mit Bibliotheken erleichtert

Zusammenfassung und Ratschläge:

- Bereits in kleinsten Projekten ist die Automatisierung von Erzeugungsprozessen (Buildmanagement) ein Muss; in einfachen Fällen bietet der verwendete Compiler die notwendige Unterstützung.
- Im Linux/Unix-Umfeld ist „(GNU) make“ das Standardwerkzeug für die Automatisierung von Erzeugungsprozessen.
- In jedem Projekt sollte es genau einen Verantwortlichen für die Pflege von Konfigurationsdateien (Makefiles) und Build-Prozesse geben.
- Für Java-Programmentwicklung gibt es mit Ant ein speziell zugeschnittenes moderneres „Build-Tool“; siehe http://jakarta.apache.org/ant/index.html.
- Noch „moderner“ sind Maven und Jenkins für „Continuous Integration“, also die automatisierte, permanente Erzeugung von Software-Releases.
- Nicht generierte (Anteile von) Konfigurationsdateien müssen selbst unbedingt der Versionsverwaltung unterworfen werden.

2.6 Änderungsmanagement

Ein festgelegter Änderungsmanagementprozess sorgt dafür, dass Wünsche für Änderungen an einem Softwaresystem protokolliert, priorisiert und kosteneffektiv realisiert werden.

Änderungsmanagement frei nach [Li02]:

```
Ausfüllen eines Änderungsantragsformulars;
Analyse des Änderungsantrags;
if Änderung notwendig (und noch nicht beantragt) then
    Bewertung wie Änderung zu implementieren ist;
    Einschätzung der Änderungskosten;
    Einreichen der Änderung bei Kommission;
    if Änderung akzeptiert then
        Durchführen der Änderung für Release ...
    else
        Änderungsantrag ablehnen
else
    Änderungsantrag ablehnen
```

Änderungsmanagement frei nach [Wh00]:

1. ein Änderungswunsch (feature request) oder eine Fehlermeldung (bug report) wird eingereicht (Status **submitted**)
2. ein eingereichter Änderungswunsch wird evaluiert und dabei
   - ⇨ entweder abgelehnt (Status **rejected**)
   - ⇨ oder als Duplikat erkannt (Status **duplicate**)
   - ⇨ oder mit Kategorie und Priorität versehen (Status **accepted**)
3. ein akzeptierter Änderungswunsch wird von dem für seine Kategorie Zuständigen
   - ⇨ für ein bestimmtes Release zur Bearbeitung freigegeben (Status **assigned**)
   - ⇨ oder vorerst aufgeschoben (Status **postponed**)
4. ein zugewiesener Änderungswunsch wird von dem zuständigen Bearbeiter in Angriff genommen (Status **opened**)
5. irgendwann ist die Bearbeitung eines Änderungswunsches beendet und die Änderung wird zur Prüfung freigegeben (Status **released**)
6. erfüllt die durchgeführte Änderung den Änderungswunsch, so wird schließlich der Änderungswunsch geschlossen (Status **closed**)

Funktionalität von Änderungsmanagement-Werkzeugen:

- Änderungswünsche können über Browserschnittstelle übermittelt werden
- Änderungswünsche werden in Datenbank verwaltet
- Betroffene werden vom Statuswechsel eines Änderungswunsches benachrichtigt
- Integration mit Projektmanagement und KM-Management
- Trendanalyse und Statistiken als Grafiken (Anzahl neuer Fehlermeldungen, ...)

Werkzeuge für das Änderungsmanagement:

- „Open Software“-Produkt **Sourceforge** (GForge), das cvs/svn/git/... mit Änderungsmanagementdiensten kombiniert, siehe http://sourceforge.net
- Neuere Alternativen zu Sourceforge: GitHub, GitLab, BitBucket, ...
- „Open Software“-Produkt **Bugzilla** für reines Änderungsmanagement, siehe http://bugzilla.mozilla.org/
- flexibles Projektmanagement-Werkzeug **Redmine** (http://www.redmine.org/) mit Aufgabenverwaltung, Versionsverwaltung (cvs/svn/git/...) etc.

2.7 Zusammenfassung

Mit einem für die jeweilige Projektgröße sinnvollen KM-Management steht und fällt die Qualität eines Softwareentwicklungsprozesses, insbesondere nach der Fertigstellung des ersten Softwarereleases.

Meine Ratschläge für das KM-Management:

- besteht ihr System aus mehr als einer Handvoll Dateien, so sollte ein **Buildmanagementsystem** wie make/Ant/Maven verwendet werden
- haben sie Entwicklungszeit von mehr als ein paar Tagen oder mehr als einem Entwickler, so ist ein **Versionsmanagementsystem** wie svn oder git ein Muss
- haben sie mehr als einen Anwender oder mehr als ein Release der Software, so ist ein **Changemanagementsystem** wie in Bugzilla/Redmine einzusetzen
- Werkzeuge wie Sourceforge, GitHub, GitLab, ... , die Versionsmanagement mit Bugtracking, Wiki-Dokumentation etc. kombinieren, immer einsetzen! Alles weitere hängt von Projektgröße und Kontext ab.

2.8 Zusätzliche Literatur

[Ca10] J. Calcote: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool; William Pollock Publ. (2010)

[Ca10] B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato: Version Control with Subversion, Version 1.1 (2004); http://svnbook.red-bean.com

[CW98] R. Conradi, B. Westfechtel: *Version Models for Software Configuration Management*, ACM Computing Surveys, vol. 30, no. 2, ACM Press (1998), 232-282

[DIN96] DIN EN ISO 10007:1996 Leitfaden für Konfigurationsmanagement

[IEEE88] IEEE Standard 1042-1987: *IEEE Guide to Software Configuration Management*, IEEE Computer Society Press (1988)

[IEEE98] IEEE Standard 828-1998: *IEEE Standard for Software Configuration Management Plans*, IEEE Computer Society Press (1998)

[Po09] G. Popp: *Konfigurationsmangement mit Subversion, Maven und Redmine*, dpunkt.verlag, 3. Auflage (2009)

[PBL05] K. Pohl, G. Böckle, F. van der Linden: *Software Product Line Engineering - Foundations, Principles, and Techniques*, Springer Verlag 2005, 467 Seiten

[Ve00] G. Versteegen: *Projektmanagement mit dem Rational Unified Process*, Springer-Verlag (2000)

[Ve03] G. Veersteegen (Hrsg.), G. Weischedel: *Konfigurationsmanagement*, Springer-Verlag (2003)

[VSH01] G. Versteegen, K. Salomon, R. Heinhold: *Change Management bei Software-Projekten*, Springer-Verlag (2001)

[Ca10] A. Zeller, J. Krinke: *Programmierwerkzeuge: Versionskontrolle - Konstruktion - Testen - Fehlersuche unter Linux*, dpunkt-Verlag (2000)

[Wie11] S. Wiest: *Continuous Integration mit Hudson/Jenkins: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*, dpunkt.verlag (2011)