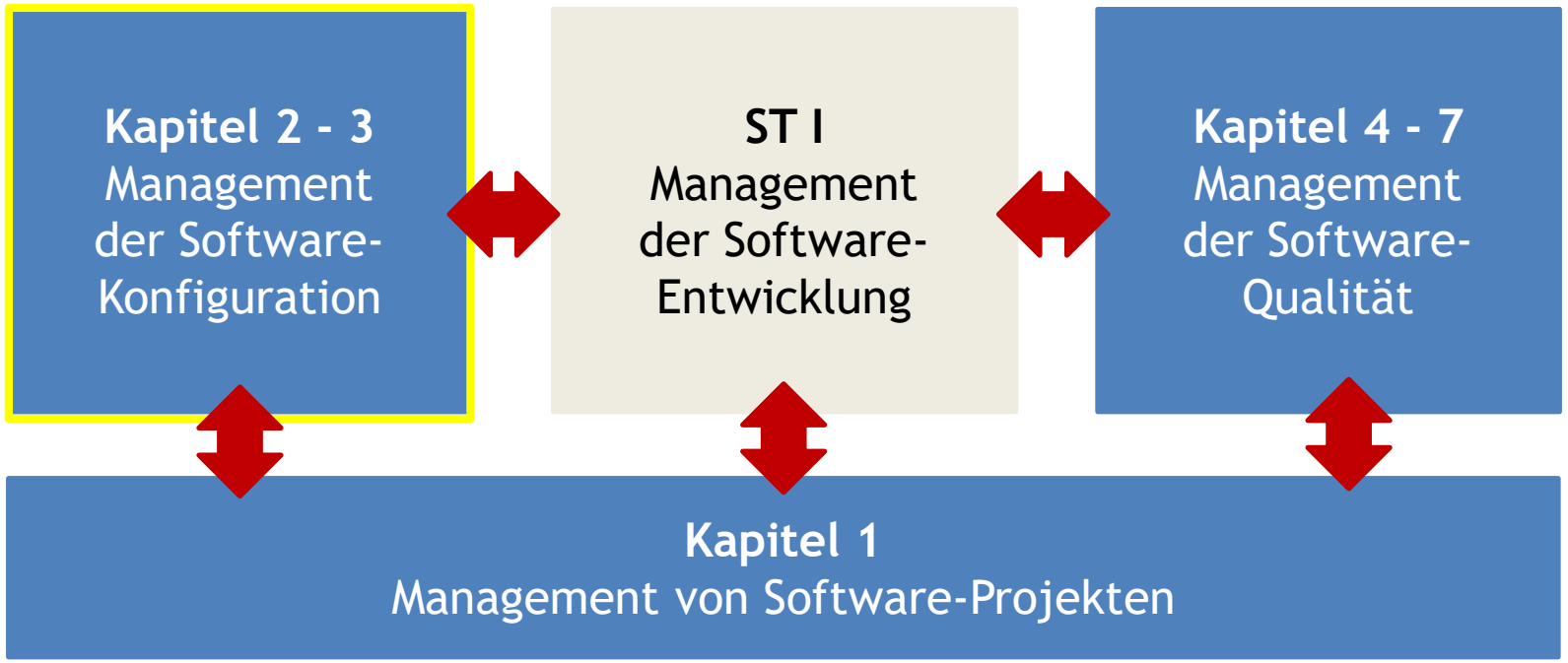
UNIVERSITÄT SIEGEN

# Vorlesung Softwaretechnik II

## Management der Software-Konfiguration: Versionsmanagement

# Aufbau der Vorlesung



# Inhalt

- Einführung: Konfigurationsmanagement
- Versionsmanagement
- (Variantenmanagement - eigenes Kapitel)
- Releasemanagement
- Buildmanagement
- Änderungsmanagement

UNIVERSITÄT SIEGEN

# Einführung: Konfigurationsmanagement

# Behandelte Fragestellungen

# Zitat

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

[Liggesmeyer, 2002]

# Definition: Software Configuration Management

**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.

[IEEE-Standard 828-1988]

# Definition: Software Configuration Management

SCM includes:

- 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.

[IEEE-Standard 828-1988]

# Definition: Konfigurationsmanagement

**KM** (Konfigurationsmanagement) ist eine Managementdisziplin, die über die gesamte Entwicklungszeit eines Erzeugnisses angewandt wird, um Transparenz und Überwachung seiner funktionellen und physischen Merkmale sicherzustellen.

[DIN EN ISO 10007]

# Definition: Konfigurationsmanagement

Der KM-Prozess umfasst:

- **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, …

[DIN EN ISO 10007]

# Definition: Konfigurationsmanagement

- **Konfigurationsbuchführung**: Rückverfolgung aller Änderungen bis zur letzten Bezugskonfiguration, ...
- **Konfigurationsauditierung:** Qualitäts-sicherungsmaßnahmen für Freigabe einer Konfiguration eines Erzeugnisses.
- **KM-Planung**: Festlegung der Grundsätze und Verfahren zum KM in Form eines KM-Plans.

[DIN EN ISO 10007]

# Arbeitsgebiete des Konfigurationsmanagements

- **Versionsmanagement**: Verwaltung der Entwicklungsgeschichte eines Produkts:
  *Wer hat wann, wo, was und warum geändert?*
  (Schwerpunkt dieses Kapitels!)
- **Variantenmanagement**: Verwaltung parallel existierender Ausprägungen eines Produkts: *Welche verschiedenen Features, Nutzer-anforderungen, Länder, Plattformen werden unterstützt?*
  (Eigenes Kapitel!)

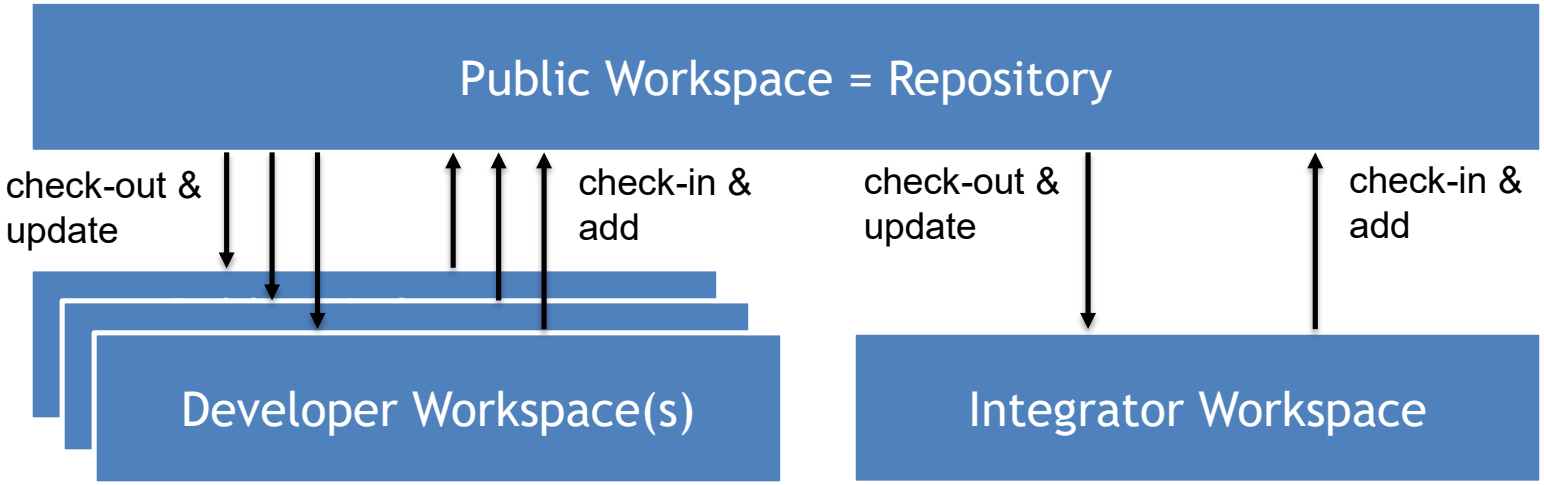
# Arbeitsgebiete des Konfigurationsmanagements

- **Releasemanagement**: Verwaltung und Planung von Auslieferungsständen:
  *Wann wird eine neue Produktversion mit welchen Features auf den Markt geworfen?*
- **Buildmanagement**: Erzeugung des auszuliefernden Produkts:
  *Wann muss welche Datei mit welchem Werkzeug generiert, übersetzt, ... werden?*

# Arbeitsgebiete des Konfigurationsmanagements

- **Änderungsmanagement**: Verwaltung von Änderungsanforderungen und Zuordnung zu Auslieferungsständen:
  *Wie, wann und durch wenn werden Fehlermeldungen (Bug Reports) und Änderungswünsche (Feature Requests) bearbeitet?*

# Konfigurationsmanagement mit Workspaces

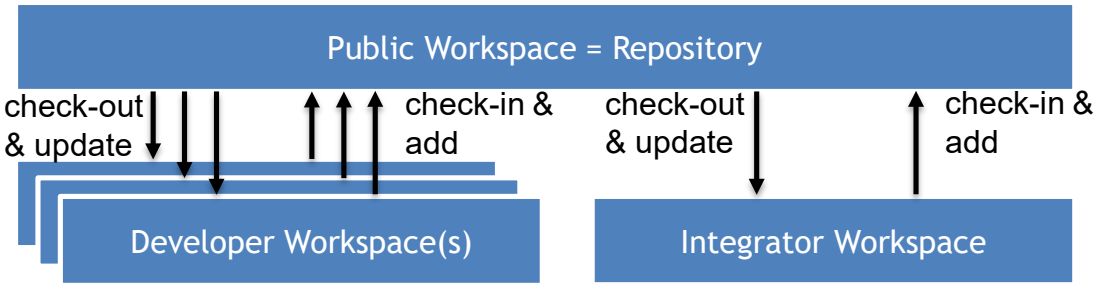



Alle Dokumente (Objekte, Komponenten) zu einem bestimmten Projekt werden in einem gemeinsamen Repository (**public workspace**) aufgehoben.

# Aufbau von Workspaces

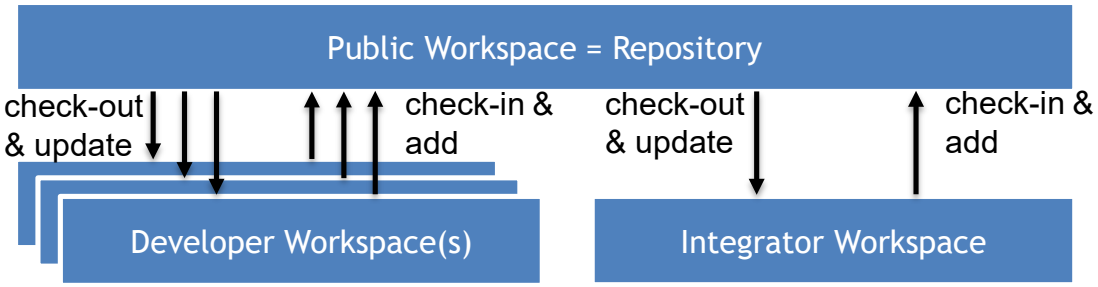
- Im Repository werden nicht nur aktuelle Versionen, sondern auch alle **früheren Versionen** aller Dokumenten gehalten.
- Beteiligte Entwickler bearbeiten ihre eigenen Versionen dieser Dokumente in ihrem privaten Arbeitsbereich (**private / developer workspace**).
- Es gibt genau einen Integrationsarbeitsbereich (**integration workspace**) für die Systemintegration.

# Aktivitäten beim Arbeiten mit Workspaces




- **Checkout:** Personen holen sich Versionen neuer Dokumente, die von anderen Personen erstellt wurden, in ihren privaten Arbeitsbereich.
- **Update:** Personen passen ihre Privatversionen ggf. von Zeit zu Zeit an neue Versionen im öffentlichen Repository an.

# Aktivitäten beim Arbeiten mit Workspaces




- **Checkin/Commit:** Personen fügen (hoffentlich) nur konsistente Dokumente als neue Versionen in das allgemeine Repository ein.
- **Add:** Personen fügen neue Dokumente dem Repository hinzu.

# Probleme beim Umgang mit Workspaces

- 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)?

# Begriffe des Konfigurationsmanagements

- **Dokument:** Gegenstand, der der Konfigurationsverwaltung unterworfen wird (eine einzelne Datei, ein ganzer Dateibaum, …)
- **(Versions-)Objekt:** Zustand eines Dokumentes zu einem bestimmten Zeitpunkt in einer bestimmten Ausprägung.
- **Varianten:** Gleichzeitig existierende Ausprägungen eines Dokuments, die unterschiedliche Anforderungen erfüllen.

# Begriffe des Konfigurationsmanagements

- **Revisionen:** Zeitlich aufeinander folgende Zustände eines Dokuments.
- **Konfiguration:** Komplexes Versionsobjekt, eine bestimmte Ausprägung eines Programmsystems (oft hierarchisch strukturierte Menge von Dokumenten).
- **Baseline:** Konfiguration, die zu einem Meilenstein (Ende einer Entwicklungsphase) gehört und evaluiert (getestet) wird.

# Begriffe des Konfigurationsmanagements

- **Release:** Eine stabile Baseline, die ausgeliefert wird (intern an Entwickler oder extern an bestimmte Kunden oder ... ).

UNIVERSITÄT SIEGEN

# Versionsmanagement

# Definition: Versionsmanagement

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

# Versionsverwaltungssysteme

- Zu jedem Dokument (z.B. Quelltextdatei) wird eine **History** gespeichert (nicht die Versionen selbst!)
- Die History beschreibt die Liste jeweils geänderter (Text-)**Blöcke** des Dokuments.
- Jeder Block der History ist ein **Delta**, das Änderungen zwischen Vorgängerrevision und aktueller Revision beschreibt.
- Jedes Delta hat eine **Identifikationsnummer** der zugehörigen Revision.

# Operationen von Versionsverwaltungssystemen

**Diff:**

- Bestimmung von Unterschieden (Deltas) zwischen (Text-)Dateien.
- Ein Delta (diff) zwischen zwei Textdateien besteht aus einer Folge von **Hunks** (Stücken), die jeweils Änderungen eines Zeilenbereiches beschreiben.

# Operationen von Versionsverwaltungssystemen

Syntax für Deltas:

- **Geänderte** Zeilen werden mit „**!**“ markiert.
- **Hinzugefügte** Zeilen werden mit „**+**“ markiert.
- **Gelöschte** Zeilen werden mit „**-**“ markiert.
- **Unveränderte** Kontextzeilen werden ohne weitere Markierung zur besseren Identifikation des Kontextes der Änderungsstellen aufgeführt.

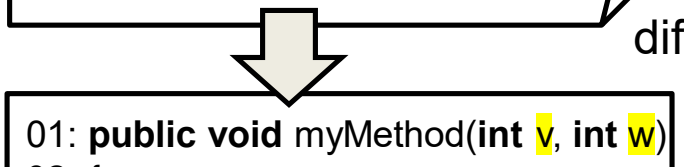
# Operationen von Versionsverwaltungssystemen

**Patch:**

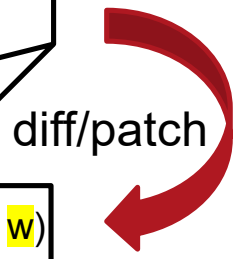
- Ein **Vorwärtsdelta** zwischen Dateien d1 und d2 kann zur *Erzeugung* von Datei d2 auf Datei d1 angewendet werden.
- Ein (inverses) **Rückwärtsdelta** zwischen zwei Dateien d1 und d2 kann zur *Wiederherstellung* von Datei d1 auf Datei d2 angewendet werden.

# Beispiel: Diff & Patch - Erste Revision

```
01: public void myMethod(int a, int b)
02: {
    ...
10:     if (a < b) {
11:         xxx
12:     }
13: }
```



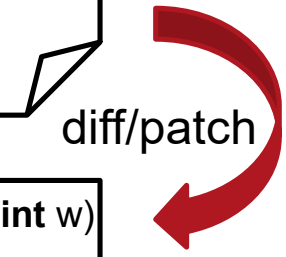
```
01: public void myMethod(int v, int w)
02: {
    ...
10:     if (v < w) {
11:         xxx
12:     } else {
13:         yyy
14:     }
15: }
```




```
*** 01,01 *** Hunk 1 ****
!   public void myMethod(int a, int b)
--- 01,01 ---
!   public void myMethod(int v, int w)
**************************
*** 10,11 *** Hunk 2 ****
!       if (a < b) {
            xxx
--- 10,13 ---
!       if (v < w) {
            xxx
+       } else {
+           yyy
**************************
```

# Beispiel: Diff & Patch - Zweite Revision

```
01: public void myMethod(int v, int w)
02: {
    ...
10:     if (v < w) {
11:         xxx
12:     } else {
13:         yyy
14:     }
15: }
```

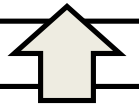



```
01: public void myMethod(int v, int w)
02: {
    ...
10:     while (v < w) {
11:         xxx
12:     }
13: }
```

```
*** 10,13 *** Hunk 1 ****
!    if (v < w) {
         xxx
-    } else {
-        yyy
--- 10,11 ---
!   while (v < w)
         xxx
***************************
```

# Beispiel: Erstes Rückwärtsdelta

```
01: public void myMethod(int v, int w)
02: {
    ...
10:     if (v < w) {
11:         xxx
12:     } else {
13:         yyy
14:     }
15: }
```



```
01: public void myMethod(int v, int w)
02: {
    ...
10:     while (v < w) {
11:         xxx
12:     }
13: }
```


```
*** 10,11 *** Hunk 1 ****
!   while (v < w)
        xxx
--- 10,13 ---
!   if (v < w) {
        xxx
+   } else {
+       yyy
**************************
```

# Beispiel: Zweites Rückwärtsdelta

```
01: public void myMethod(int a, int b)
02: {
    ...
10:     if (a < b) {
11:         xxx
12:     }
13: }
```

```
01: public void myMethod(int v, int w)
02: {
    ...
10:     if (v < w) {
11:         xxx
12:     } else {
13:         yyy
14:     }
15: }
```


```
*** 01,01 *** Hunk 1 ****
!   public void myMethod(int v, int w)
--- 01,01 ---
!   public void myMethod(int a, int b)
**************************
*** 10,13 *** Hunk 2 ****
!       if (v < w) {
            xxx
-       } else {
-           yyy
--- 10,11 ---
!       if (a < b) {
            xxx
**************************
```

# Vorwärtsdelta vs. Rückwärtsdelta

Historie = Aktuelle Version + Rückwärtsdeltas:

- Schneller Zugriff auf neuere Versionen.
- Langsame Wiederherstellung älterer Versionen.

Historie = Initiale Version + Vorwärtsdeltas:

- Langsamer Zugriff auf neuere Versionen.
- Schnelle Wiederherstellung älterer Versionen.

# Ziele bei der Erzeugung von Deltas

Werkzeuge zur „diff“-Berechnung verwenden verschiedene Heuristiken, um

- möglichst **kleine** und/oder
- **gut lesbare**

Deltas/Patches zu erzeugen.

# Regeln zur Erzeugung von Deltas

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 „!“**

# Diskussion des Beispiels

- Unveränderte Kontextzeilen am Beginn der Hunks wurden im Beispiel weggelassen.
  - Erster Hunk: Änderungszeile ist erste Zeile
  - Zweiter Hunk: aus Platzgründen
- Dieses Delta ist so klein wie möglich, aber eventuell schwer lesbar?
- Um deutlicher lesbar zu machen, dass beide Hunks die gleiche Methode betreffen, könnten sie zu einem Hunk zusammengefasst werden (aber dann zusätzliche unveränderte „Zwischenzeilen“ 02 – 09 enthalten).

```
*** 01,01 *** Hunk 1 ****
!   public void myMethod(int a, int b)
--- 01,01 ---
!   public void myMethod(int v, int w)
**************************
*** 10,11 *** Hunk 2 ****
!       if (a < b) {
            xxx
--- 10,13 ---
!       if (v < w) {
            xxx
+       } else {
+           yyy
**************************
```

# Dilemma bei der Erzeugung von Deltas

- Das Diff-Werkzeug erhält als einzige Eingabe die Vorgänger- und Nachfolge-Revision eines Dokuments ohne weitere Informationen über angewendete Änderungsoperationen/-stellen.
- Problem: Wie kann erkannt werden, ob eine Änderung in Zeile *i* durch **Einfügen einer neuen Zeile** oder durch **Ändern einer alten Zeile** zustande gekommen ist?

# Beispiel: Revision von Textdokumenten

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.

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.

Hinweis: Zeilennummern nicht Bestandteil der Texte

# Beispiel: Nutzloses Diff-Ergebnis

```
*** 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.

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

5 geänderte Zeilen!

# Beispiel: Nützliches Diff-Ergebnis

```
*** 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.

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

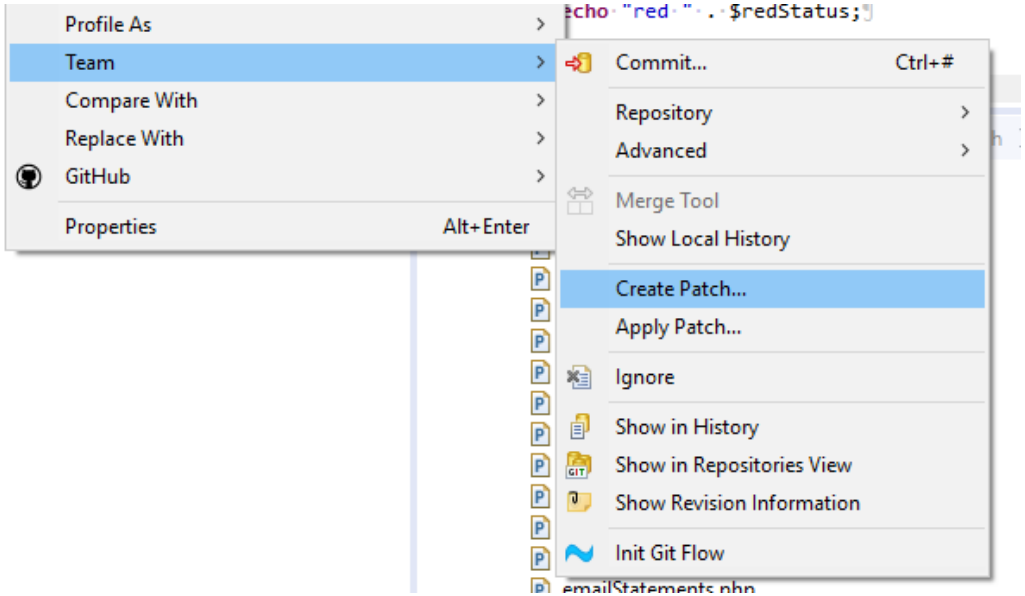
2 geänderte Zeilen
1 neue Zeile
1 gelöschte Zeile

# Unified Diff Format (GNU)

```
Index: MyFile
===================================================================
MyFile (revision 1)+++ MyFile (revision 2)
@@ -1,1 +1,1 @@
- public void myMethod(int a, int b)
+ public void myMethod(int v, int w)
@@ -10,2 +10,4 @@
-  if (a < b) {
+ if (v < w) {
        …
+ } else {
        …
===================================================================
```

Keine Entsprechung von „!“ in UDF, stattdessen Kombination aus „-“ und „+“

# Patches in Eclipse




„Create Patch“- und „Apply Patch“-Funktionen in Eclipse (SVN/GIT) basieren auf UDF.

# Patches in Eclipse

Regeln zur Erzeugung von Hunks:

- Ein Hunk beginnt zumeist mit *drei* unveränderten Kontextzeilen (inklusive Leerzeilen).
- Zwei Blöcke geänderter Zeilen müssen durch mindestens *sieben* unveränderte Zeilen getrennt sein, damit dafür getrennte Hunks erzeugt werden.

# Patches in Eclipse

Regeln zur Anwendung von Hunks:

- Werden der Kontext oder die zu löschenden Zeilen eines Patches nicht gefunden, dann endet die Anwendung mit einer Fehlermeldung.
- Befindet sich die zu patchende Stelle eines Textes nicht mehr an der angegebenen Stelle (Zeile), so wird trotzdem der Patch (an der neuen Stelle) 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.

# Versionsmanagement für Teamentwicklung

In der Praxis kommt es vor, dass **mehrere** Entwickler eines Teams **dieselbe** Datei „zeitgleich“ verändern. Um inkonsistente Zustände zu verhindern, werden Sperrkonzepte für den koordinierten schreibenden Zugriff benötigt.

# Pessimistisches Sperrkonzept

Verhindern gleichzeitiger Bearbeitung einer Datei durch mehrere Personen:

- Zu jedem Zeitpunkt nur maximal ein Checkout zum Schreiben (**single write access**).
- Zu jedem Zeitpunkt beliebig viele Checkouts zum Lesen (**multiple read access**).

Probleme:

- Hoher „manueller“ Koordinationsaufwand.
- Geringer Parallelisierungsgrad.

# Optimistisches Sperrkonzept

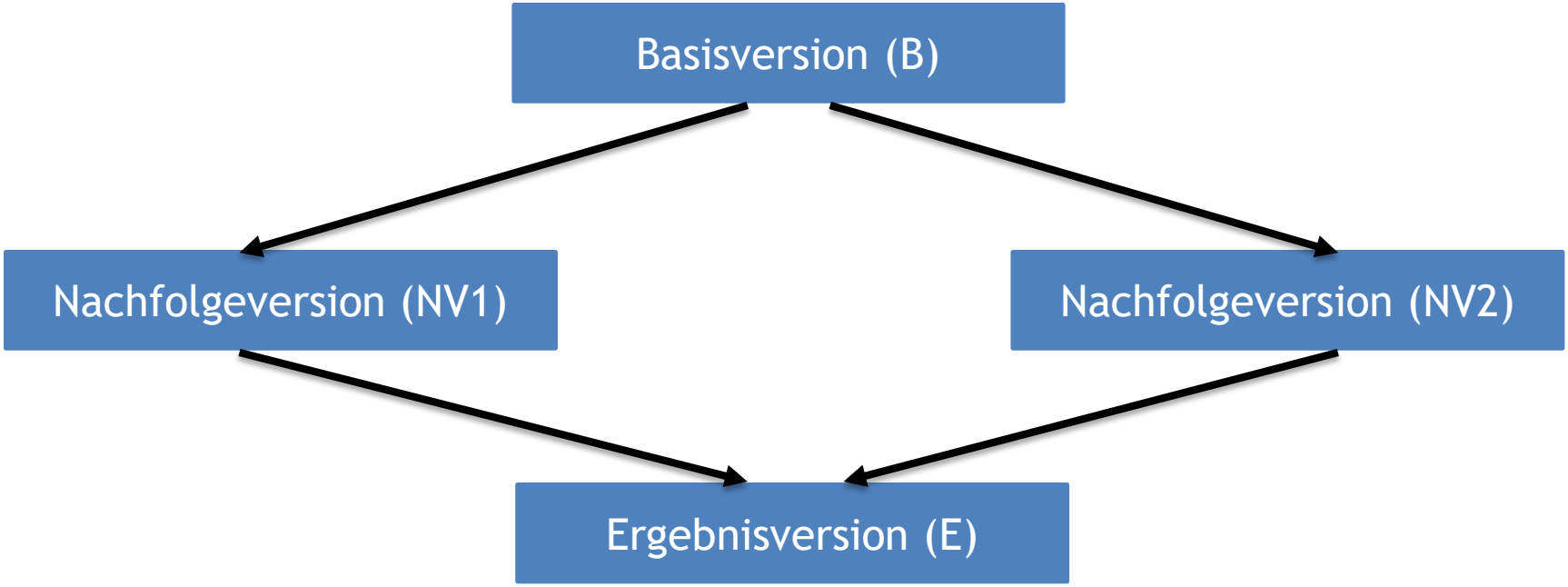
Erlaubt gleichzeitige Bearbeitung einer Datei durch mehrere Personen in verschiedenen privaten Arbeitsbereichen. Parallel durchgeführte Änderungen werden im öffentlichen Workspace integriert.

# Operationen von Versionsverwaltungssystemen

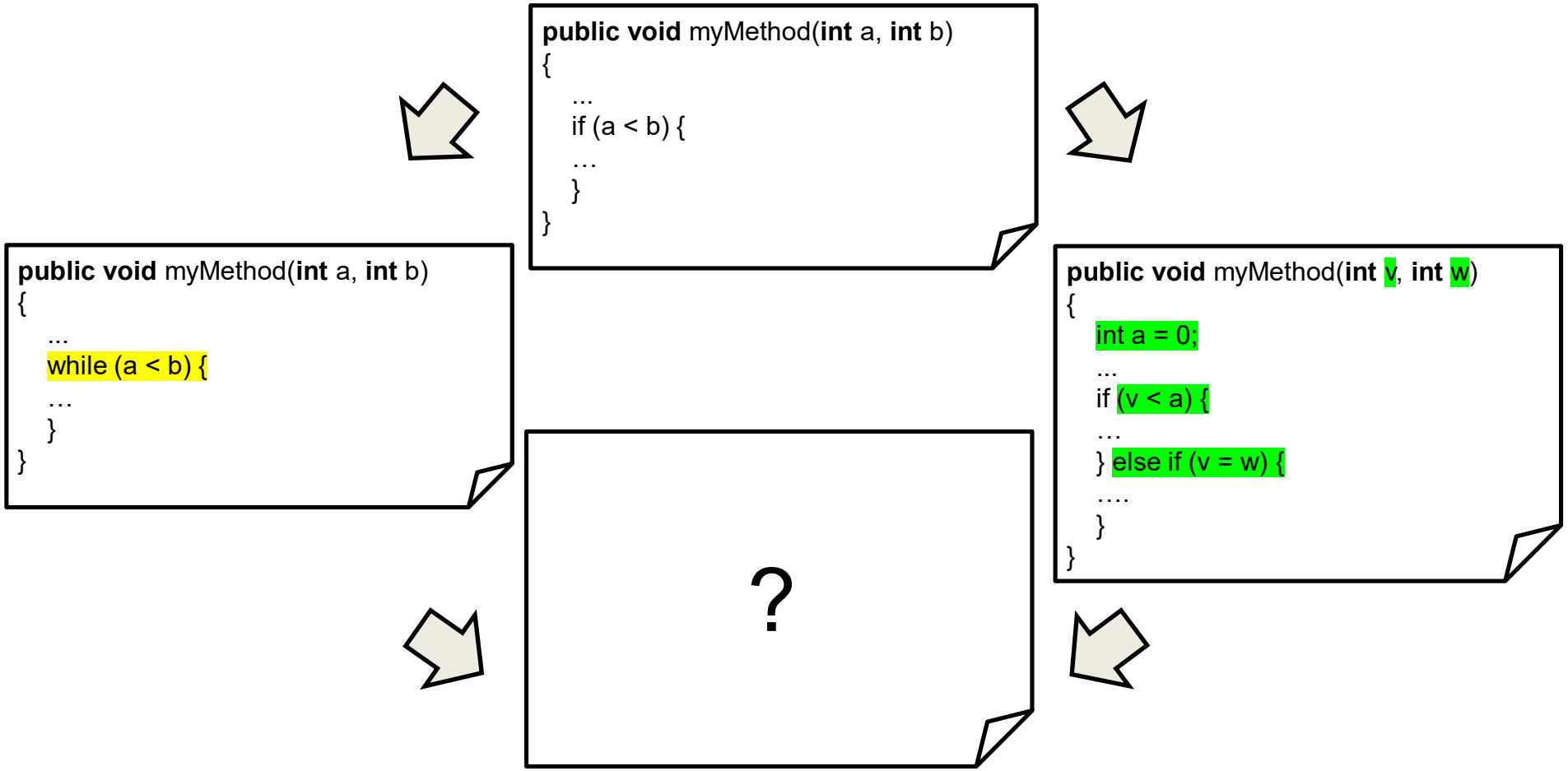
**Merge**:

- **Semi-Automatisches Verschmelzen** von parallel durchgeführten Änderungen in verschiedenen privaten Arbeitsbereichen.
- Dreiwegeverschmelzen mit **manueller Konfliktbehebung.**

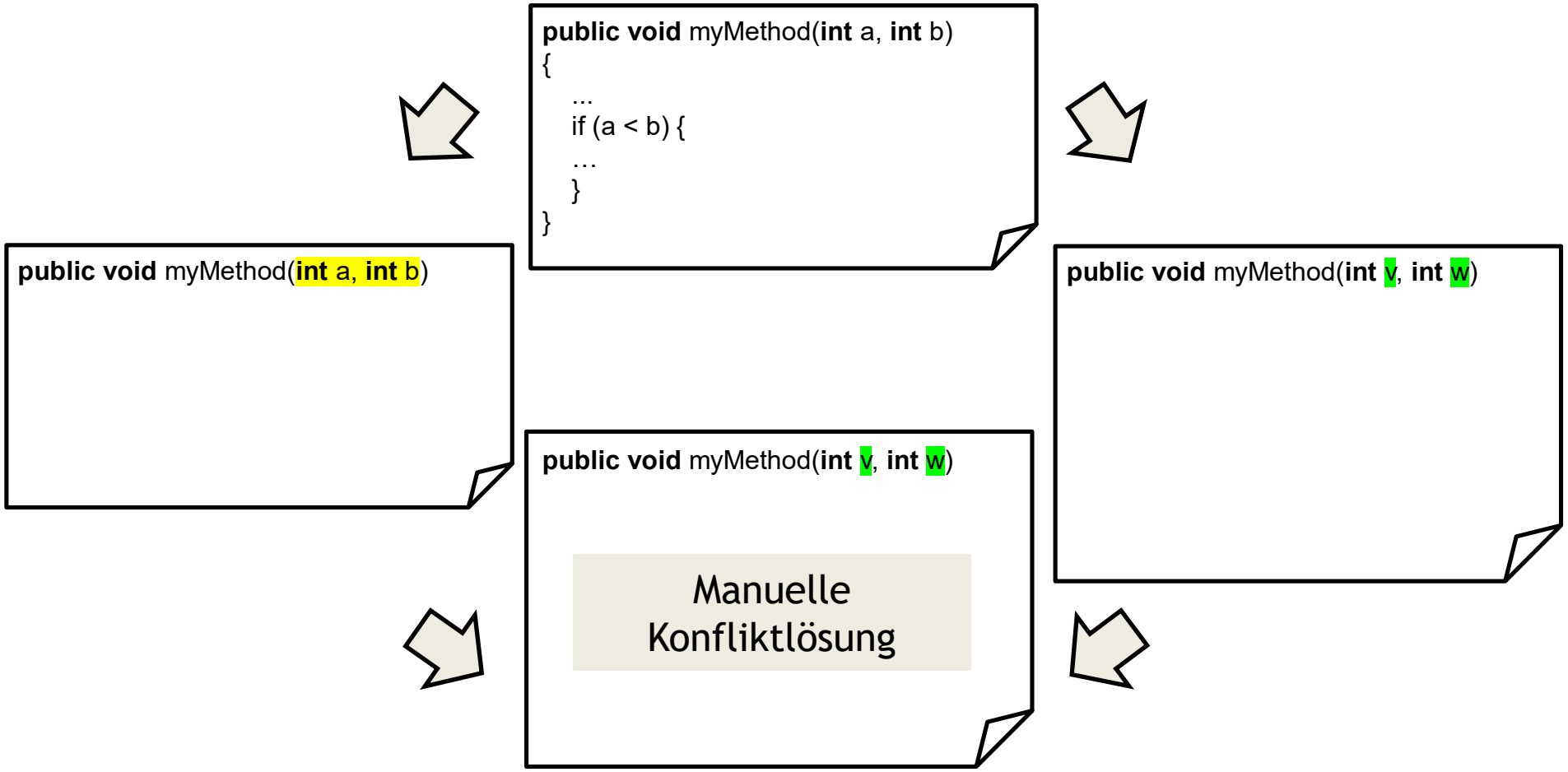
# Dreiwegeverschmelzen von (Text-)Dateien



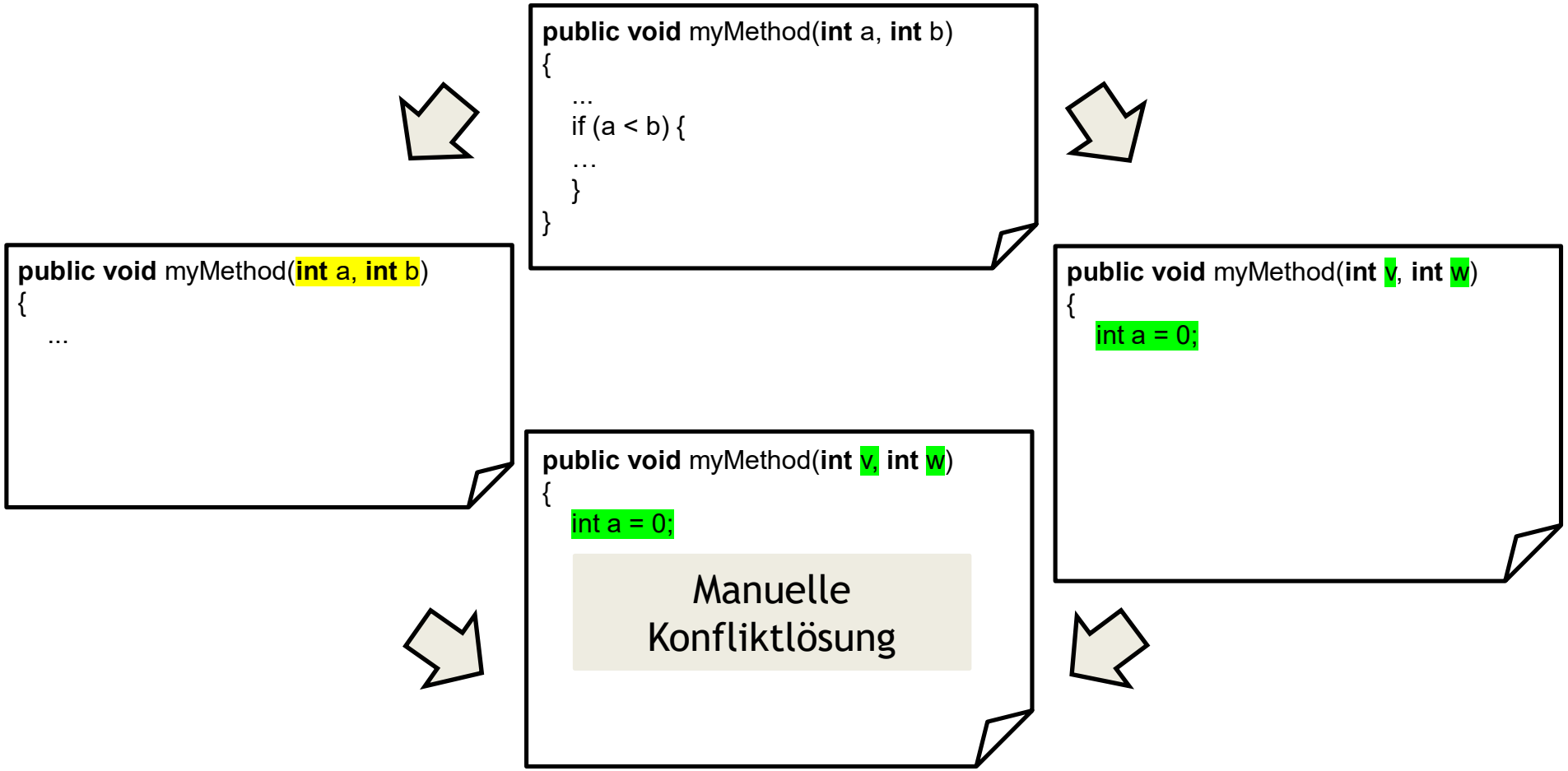
# Beispiel: Automatisches Verschmelzen



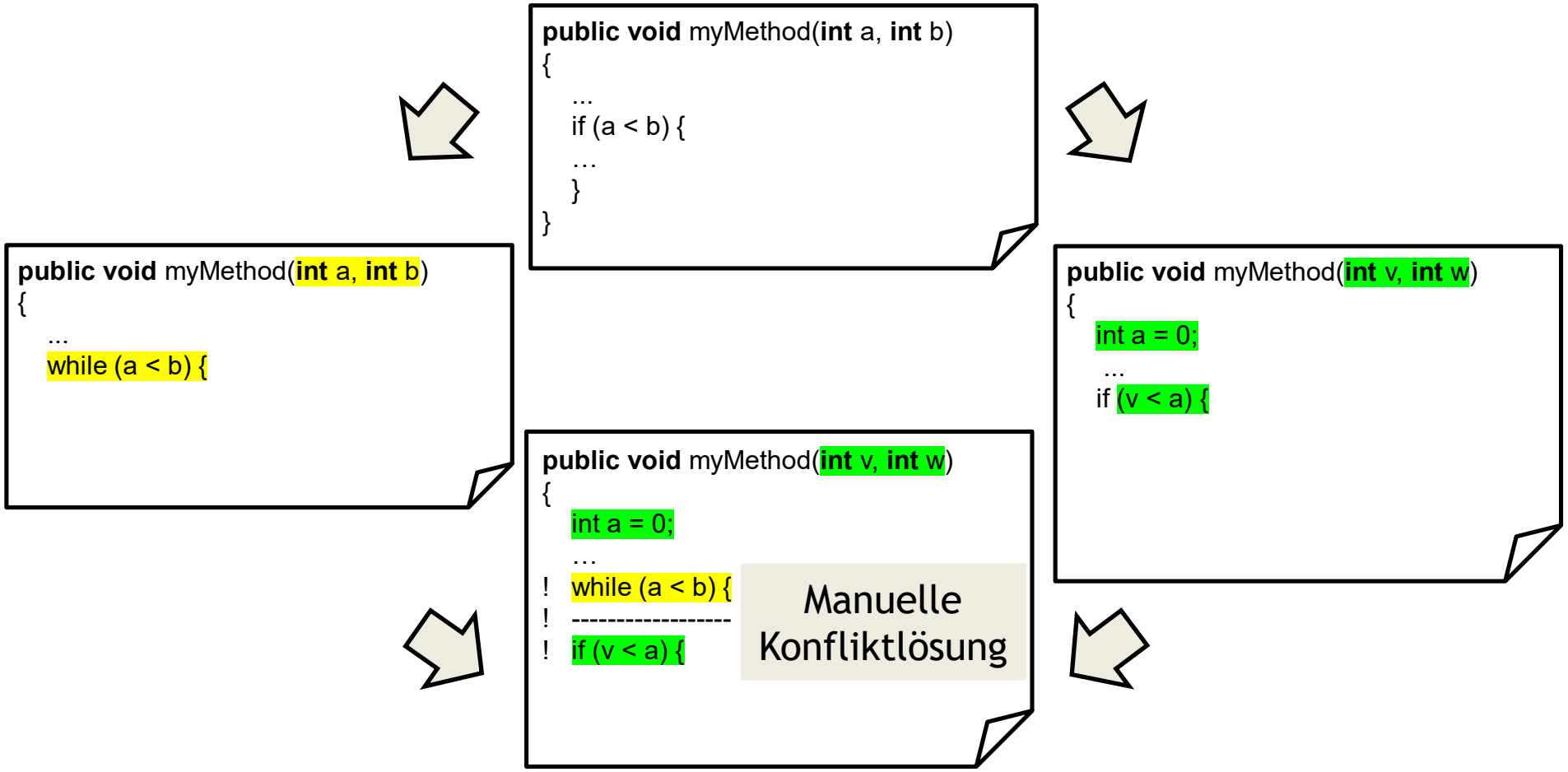
# Beispiel: Schritt 1



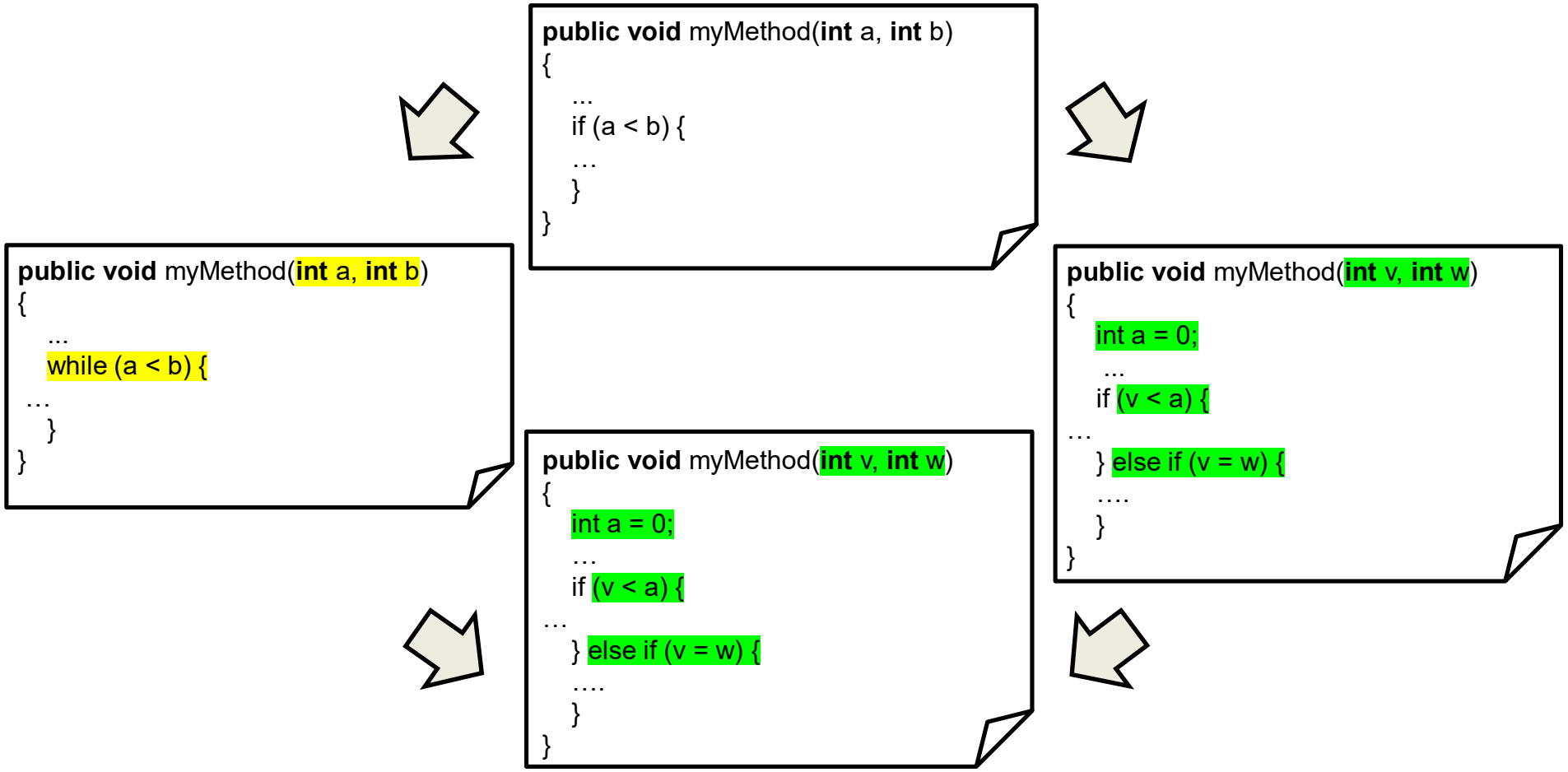
# Beispiel: Schritt 2



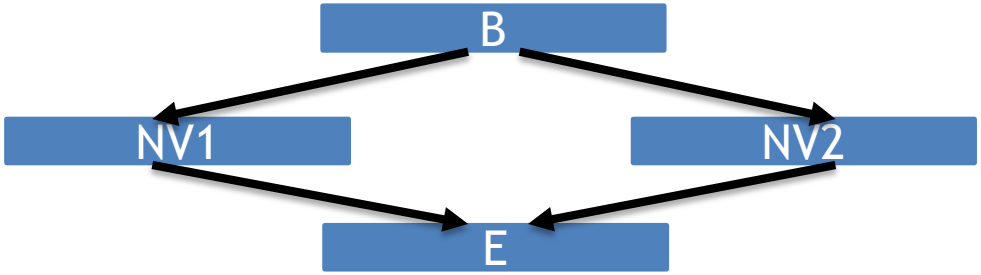
# Beispiel: Schritt 3



# Beispiel: letzter Schritt



# Verschmelzungsregeln




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

# Beispiel: Verschmelzungskonflikte

```
public void myMethod(int a, int b)
{
   ...
   if (a < b) {
   …
   }
}
```

```
public void myMethod(int v, int w)
{
   int a = 0;
   …
   while (v < w) {
…
   }
}
```

```
public void myMethod(int a, int b)
{
   ...
   if (v < a) {
   …
   } else if (a = b) {
   ….
   }
}
```

```
public void myMethod(int v, int w)
{
   int a = 0;
   …
   while (v < a) {
…
   } else if (a = b) {
   ….
   }
}
```

Syntaktisch fehlerhafte Zeile!

Semantisch fehlerhafte Bedingung!

UNIVERSITÄT SIEGEN

# 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).

# Synchronisierung mit Watch/Edit/Unedit

- **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 anderen Entwicklern 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**).

# Diskussion: Diff / Patch / Merge

- Berechnung von **Deltas** funktioniert hervorragend für Textdateien (mit Zeilenenden als „Synchronisationspunkte“ für Vergleiche).
- Berechnung (minimaler) Deltas von Binärdateien ist wesentlich schwieriger.
- Textverarbeitungsprogramme wie Word oder Modellierungstools besitzen häufig eigene Algorithmen zur Berechnung von Deltas.

# Diskussion: Diff / Patch / Merge

- **Verschmelzung** von Textdateien funktioniert meist ebenfalls gut (kann aber zu tückischen Inkonsistenzen führen).
- Verschmelzung von Binärdateien oder Grafiken/Diagrammen ist im allgemeinen nicht möglich.
- Programme wie Word oder Modellierungstools besitzen häufig eigene Verschmelzungs-funktionen.

# Versionierungswerkzeuge (Auswahl)

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:
  - Versionierung und (rekursives) Verschmelzen von Dateibäumen.
- **Git**, Mercurial, …
  - Verteilte Versionsmanagementsysteme.
  - Jeder Entwickler hat eigene/lokale Versionsverwaltung.

UNIVERSITÄT SIEGEN

# Releasemanagement

# Definition: Release

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

# Definition: Releasemanagement

Das Releasemangement dokumentiert ausgelieferte Konfigurationen und stellt deren Rekonstruierbarkeit sicher.

# Aufgaben des Releasemanagememts

- Festlegung der (zusätzlichen) Funktionalität eines neuen Releases.
- Festlegung des Zeitpunktes der Freigabe eines neuen Releases.
- Erstellung und Verbreitung eines Releases (siehe auch Buildmanagement).
- Dokumentation eines Releases.

# Dokumentation eines Releases

- Welche Revisionen welcher Dateien sind Bestandteil des Releases?
- Welche Compilerversion wird verwendet?
- Welche Betriebssystemversion wird auf der Entwicklungs- und Zielplattform (mindestens) vorausgesetzt?
- …

# Planung eines Releases

Vorbedingungen prüfen:

- Ist seit dem letzten Release viel Zeit vergangen? (neues Release aus Publicity-Gründen)
- Wurden seit dem letzten Release viele Fehler behoben? (neues Release mit allen „Patches“)
- Wurden seit dem letzten Release viele neue Funktionen hinzugefügt?

# Planung eines Releases

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 erlaubt (selbst „gefährliche“ Fehlerkorrekturen verboten).

# Planung eines Releases

Kurz vor der Release-Freigabe:

- Umfangreiche Qualitätssicherungsmaßnahmen (Tests) durchführen.
- Letzte Fehlerkorrekturen durchgeführt.

# Planung eines Releases

Release wird freigegeben und weiterverbreitet:

- Weiterentwicklung (neuer Releases) wird wieder aufgenommen.
- Freigegebenes Release muss parallel dazu gepflegt werden (erhält z.B. eigenen Wartungsbranch parallel zum Entwicklungsbranch).

# Vergabe von Versionsnummern

Software-Releases erhalten üblicherweise **zweistellige Versionsnummern** der Form „x.y“:

- Die erste Stelle „x“ wird erhöht, wenn sich die Funktionalität signifikant ändert.
- Die zweite Stelle „y“ wird für kleinere Verbesserungen erhöht.
- Oft wird erwartet, dass x.2, x.4, ... stabiler als x.1, x.3, ... sind.

# Vergabe von Revisionsnummern

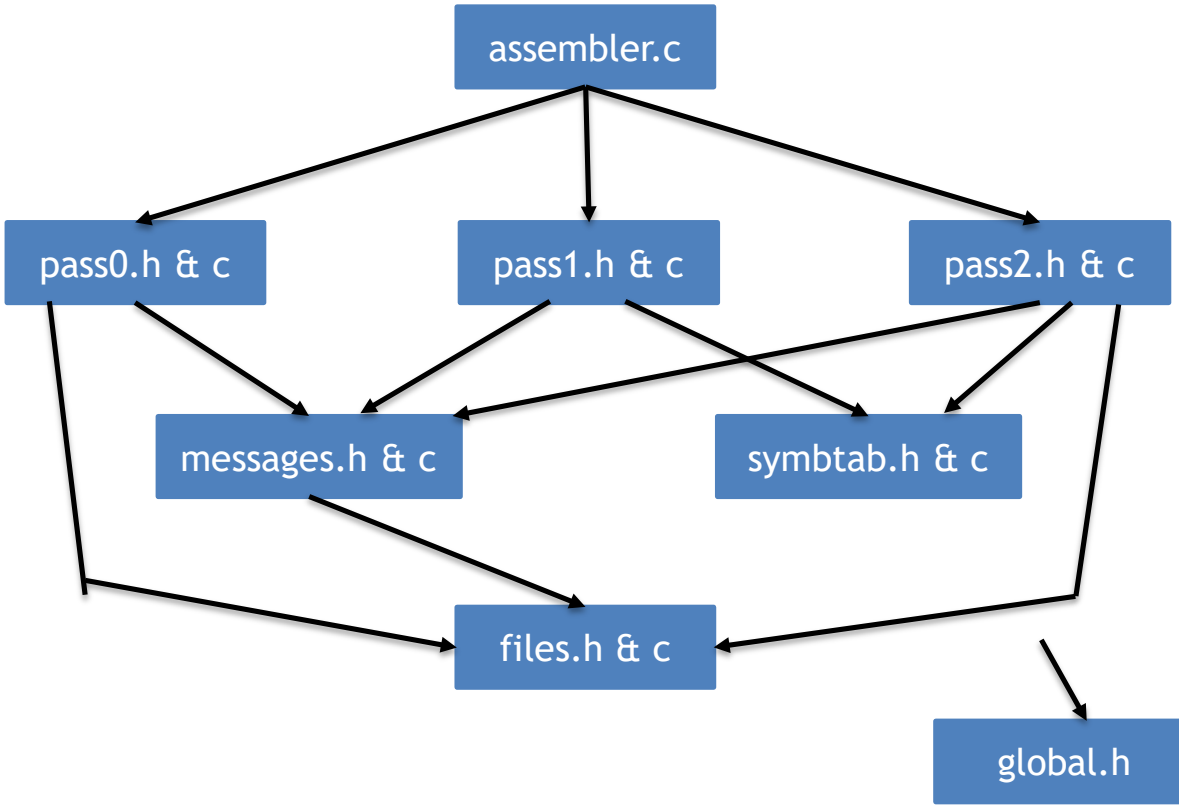
- Die internen Revisionsnummern eines KM-Systems müssen mit den extern sichtbaren Versionsnummern nichts zu tun haben.
- Mögliche Konvention: Versionsnummer = die ersten beiden Stellen der Revisionsnummer.
- Oft werden aber auch die ersten beiden Stellen der Revisionsnummern gar nicht verwendet und bleiben auf „1.1“ gesetzt.

# Buildmanagement

# Definition: Buildmanagement

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

# Klassisches Beispiel: Assembler-Bau in C




(von allen Modulen inkludiert)

**Module:**

Header-Datei (.h)
Quellcode-Datei (.c)

**Importbeziehung:**


(M1.c includes M2.h)

# Erläuterungen zum Beispiel

Jedes „normale“ Modul *M* besteht aus zwei Textdateien:

- **Header-Datei** *M.h* beschreibt die Schnittstelle des Moduls, die von anderen Modulen benötigt wird (Konstanten, Typen, Prozedurdefinitionen).
- **Implementierungsdatei** *M.c* enthält die C-Quelltexte der Prozeduren.

# Erläuterungen zum Beispiel

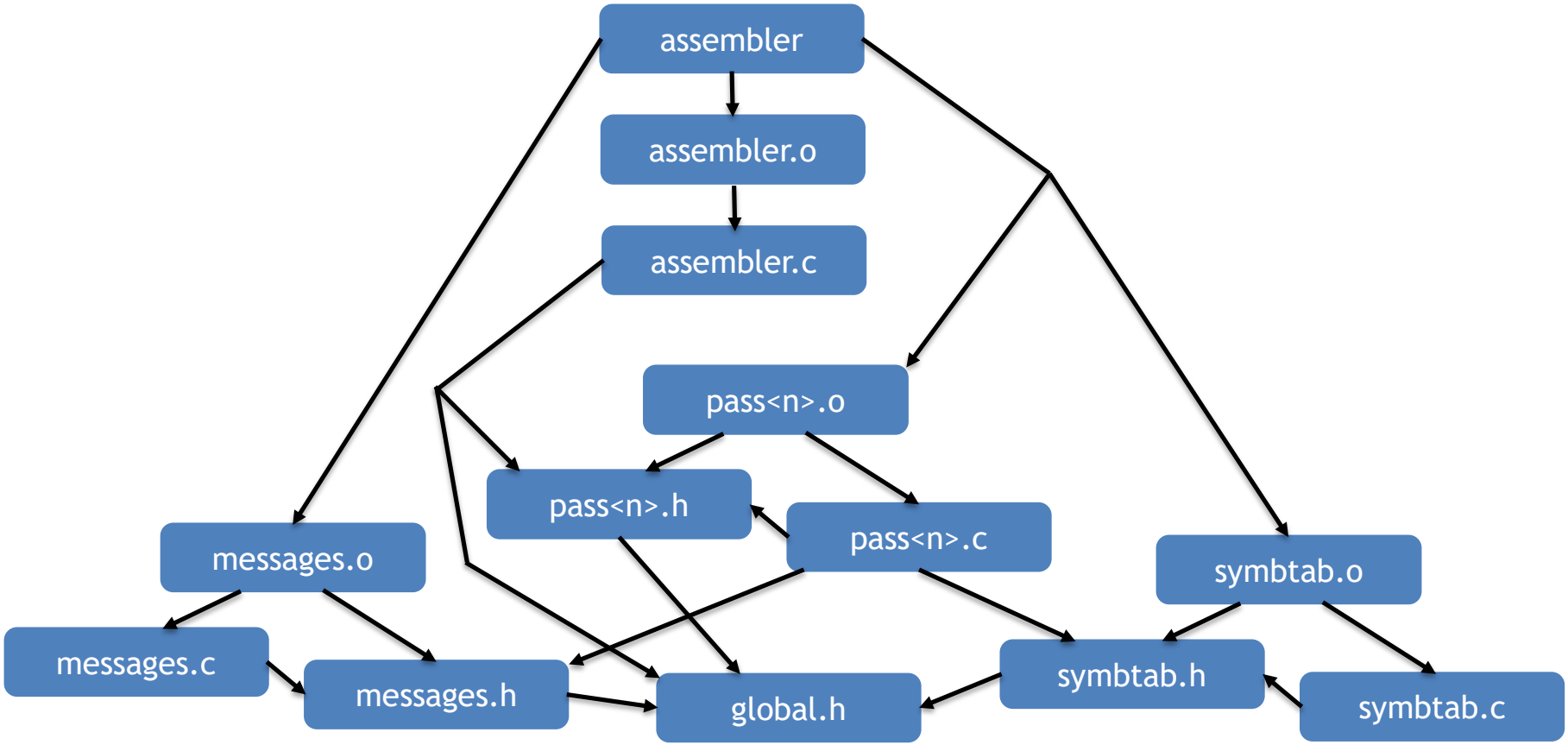
Das Modul *global* besteht nur aus einer Textdatei *global.h*:

- Die Datei enthält überall benötigte Konstanten- und Typdefinitionen.
- „überall“ heißt hier in allen anderen .h- und .c-Dateien.

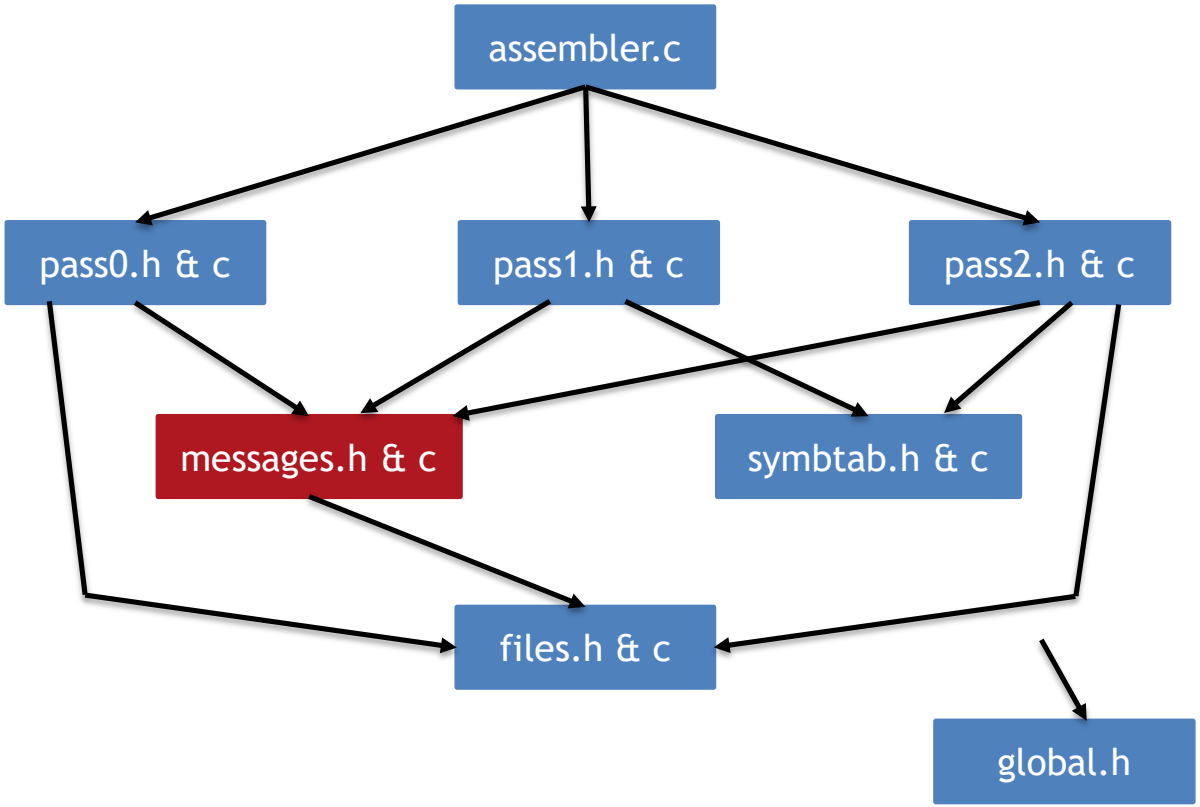
# Erläuterungen zum Beispiel

- 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

# Beispiel: Verfeinerter Abhängigkeitsgraph



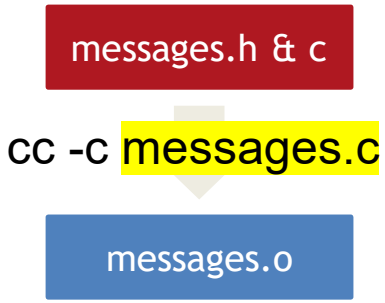
# Beispiel 1: Übersetzungs- und Bindevorgang



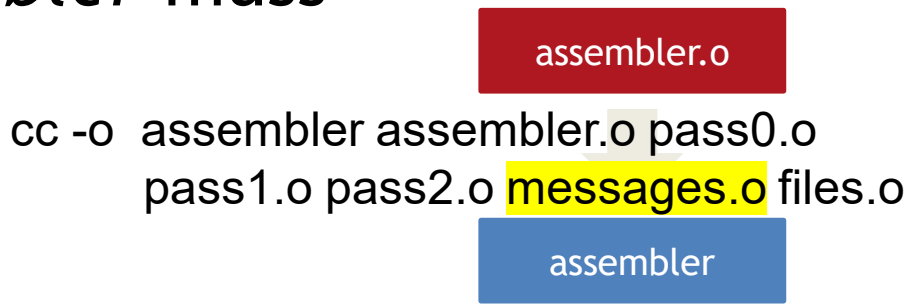

**Szenario 1:**
Die Implementierung von *messages.c* ändert sich.

# Beispiel 1: Übersetzungs- und Bindevorgang

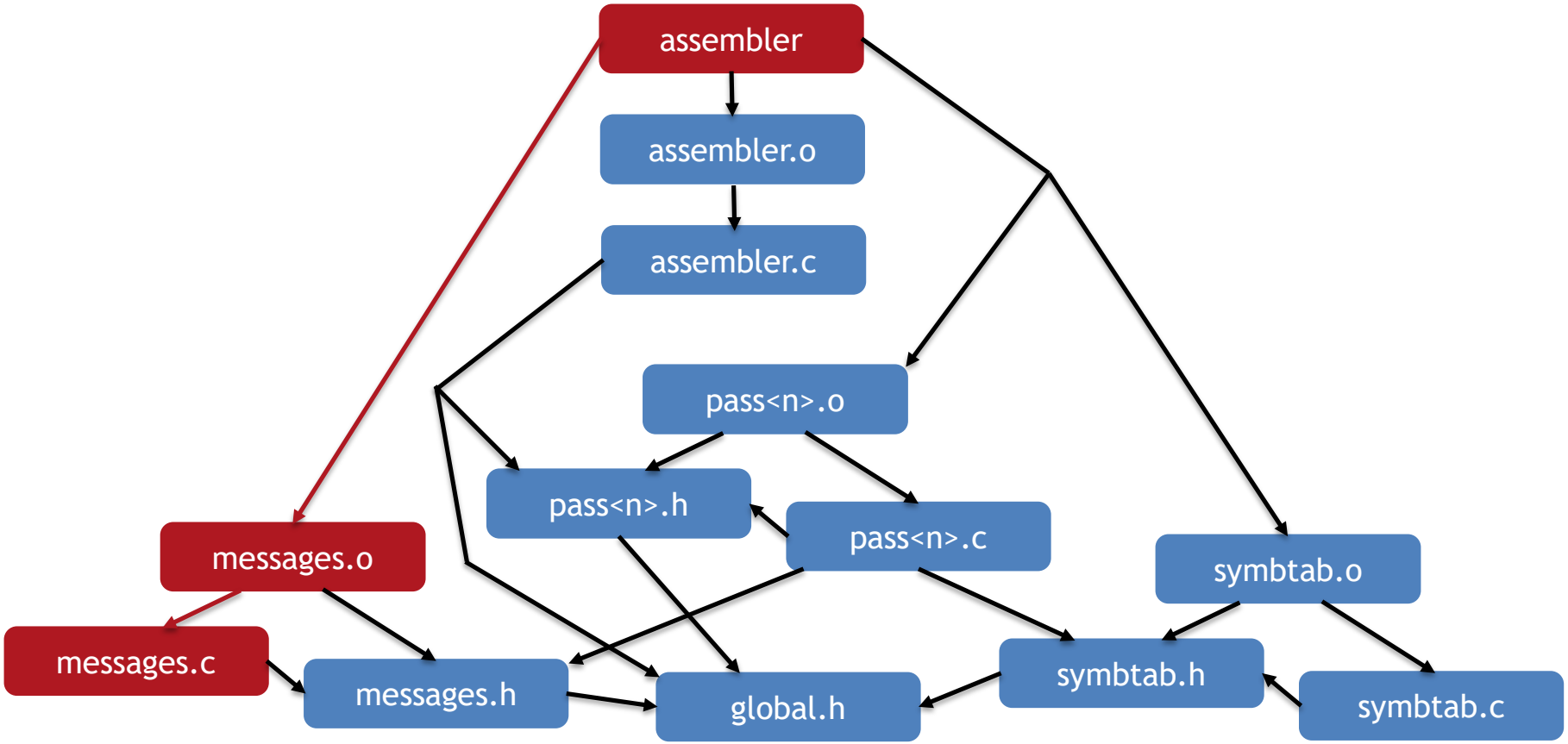
1. Datei *messages.c* muss neu in *messages.o* übersetzt werden.



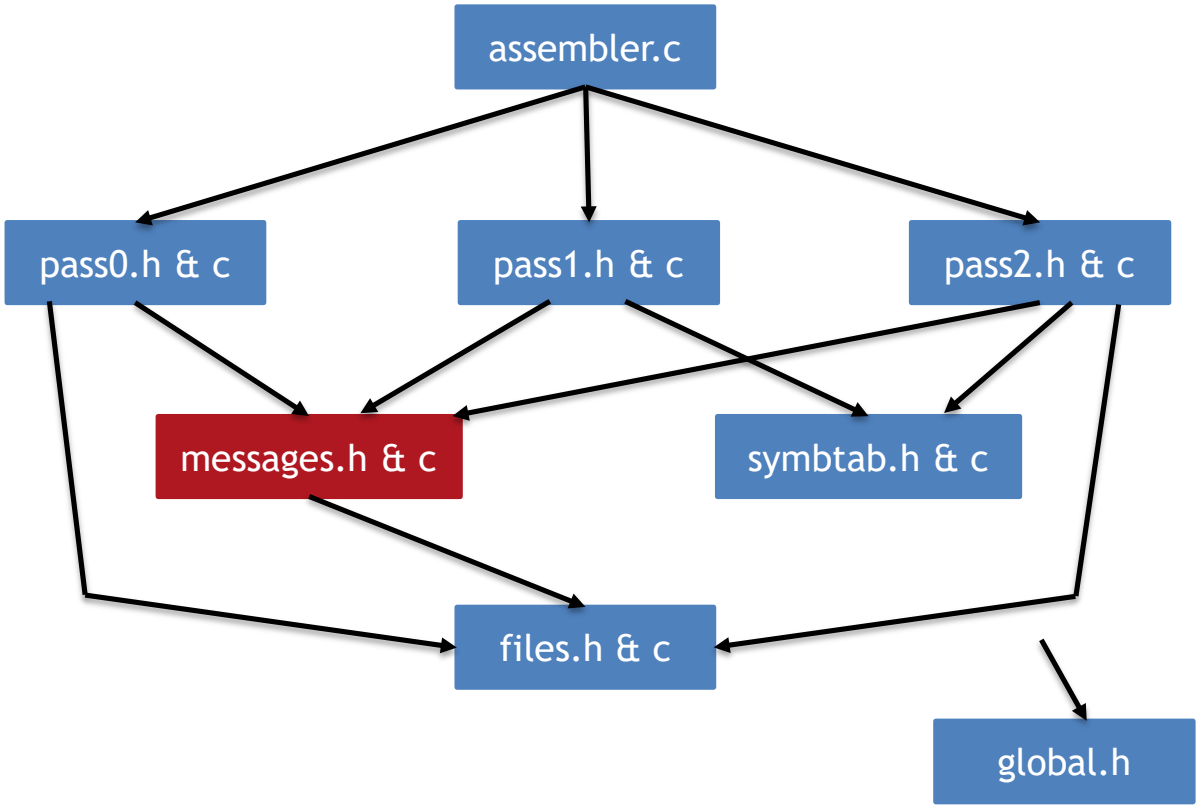

2. Hauptprogramm *assembler* muss neu erzeugt werden:



# Beispiel 1 im Abhängigkeitsgraph

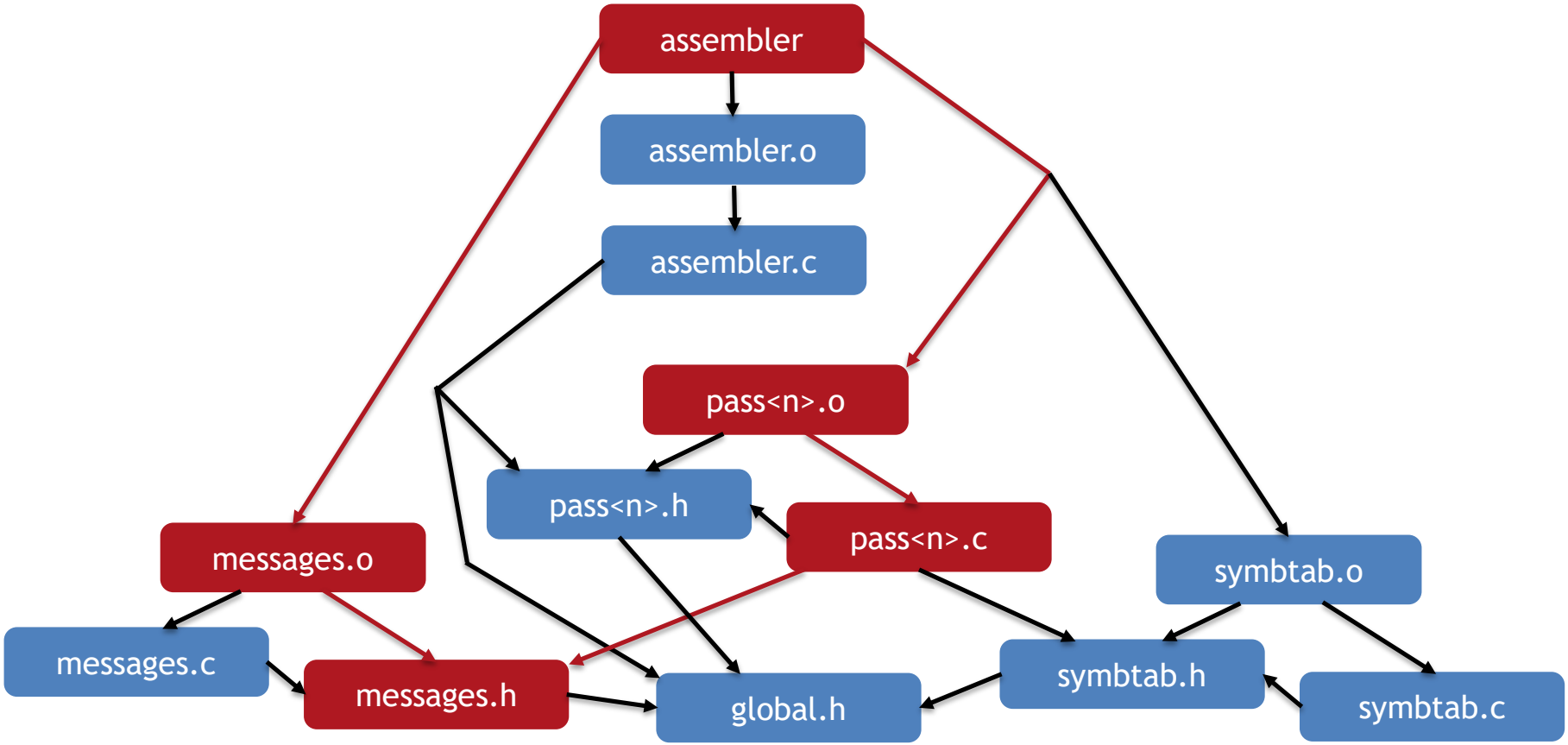


# Beispiel 2: Übersetzungs- und Bindevorgang




**Szenario 2:**
Die Schnittstelle *messages.h* ändert sich (dadurch evtl. auch *messages.c*).

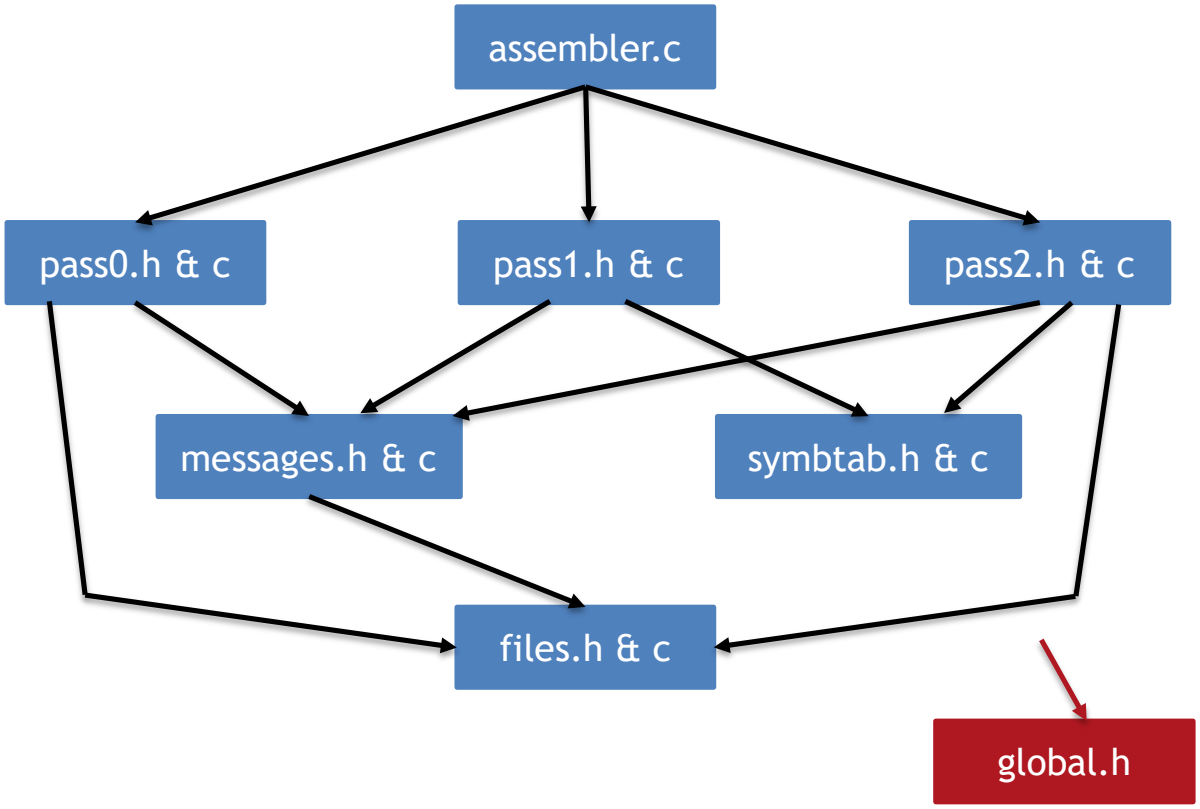
# Beispiel 2 im Abhängigkeitsgraph



# Beispiel 2: Übersetzungs- und Bindevorgang

- *messages.c* muss neu in *messages.o* übersetzt werden: *cc -c messages.c*
- *pass0.c*, *pass1.c* und *pass2.c* müssen neu übersetzt werden: *cc -c pass0.c | cc -c pass1.c | cc -c pass2.c*
- 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*

# Beispiel 3: Übersetzungs- und Bindevorgang




**Szenario 3:**
In *global.h* wird eine Konstante geändert.

# Beispiel 3: Übersetzung- und Bindevorgang

- 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 | ...*
- 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 Buildmanagement

**Ur-Build-Werkzeug „make“:**

- Deklarativer, regelbasierter Ansatz.
- Es werden Abhängigkeiten zwischen Entwicklungsartefakten definiert (mit zusätzlichen Kommandos zur Erzeugung abgeleiteter Artefakte)
- Unabhängig vom Entwicklungsprozess, Programmiersprachen, Werkzeugen.

# Werkzeuge für das Buildmanagement

**Java-Werkzeug „ant“:**

- Gemischt deklarativer/imperativer Ansatz.
- Es werden Abhängigkeiten zwischen Aufgaben (Tasks) definiert; Tasks können mit Kontrollstrukturen „ausprogrammiert“ werden.
- Verwendet XML-Syntax für Konfigurationsdateien.
- Schwerpunkt Java-Projekte, aber: unabhängig von Entwicklungsprozess, Projektstruktur, ...

# Werkzeuge für das Buildmanagement

**„Maven“:**

- Deklarativer Ansatz mit relativ einfachen Konfigurationsdateien.
- Viele Annahmen über Entwicklungsprozess, Projektstruktur, … Anpassung über Plugins
- Fokus auf Java-Projekten mit globalen Repositories.
- Oft kombiniert mit „Nexus“ für Einrichten lokaler (Cache-)Repositories.

# Werkzeuge für das Buildmanagement

„**Jenkins**“ (früher: „Hudson“):

- Ergänzung zu Build-Werkzeugen wie Ant und Maven.
- Unterstützt kontinuierliche Integration/Freigabe von Produkten.
- Automatische Integration von Build-, Test-, Reporting-Werkzeugen.

# Werkzeuge für das Buildmanagement

**„Gradle“**:

- Vergleichbar zu „ant“ und „Maven“.
- Domänenspezifische Sprache zur Abhängigkeitsbeschreibung, daraus generierte Gradle-Skripte sind direkt ausführbarer Code.
- Ebenfalls Task- und Plugin-Konzept.
- Schwerpunkt auf effiziente inkrementelle und parallele Build-Prozesse für große Projekte.

# Buildvorgänge mit „make“ automatisieren

- *make* wurde in den 70er Jahren „nebenbei“ von Stuart F. Feldman für die Erzeugung von Programmen unter Unix programmiert.
- *make* gibt es heute in vielen Varianten auf allen gängien 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 (Ist Zeitmarke jünger als die Zeitmarken der Dateien, von denen sie abhängt?).

UNIVERSITÄT SIEGEN

# Aufbau einer Abhängigkeitsbeschreibung

```
ziel : objekt1 objekt 2 objekt3 ...
        Kommando zur Erzeugung von ziel aus objekt1...
      # Kommentarzeile: Achtung Kommandozeile muss mit
      # Tabulator eingerückt.
```

# Beispiel: Abhängigkeitsbeschreibung in „makefile“

```
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
...
```

(Hinweis: Umbruch einer fortgesetzten Zeile durch „\“)

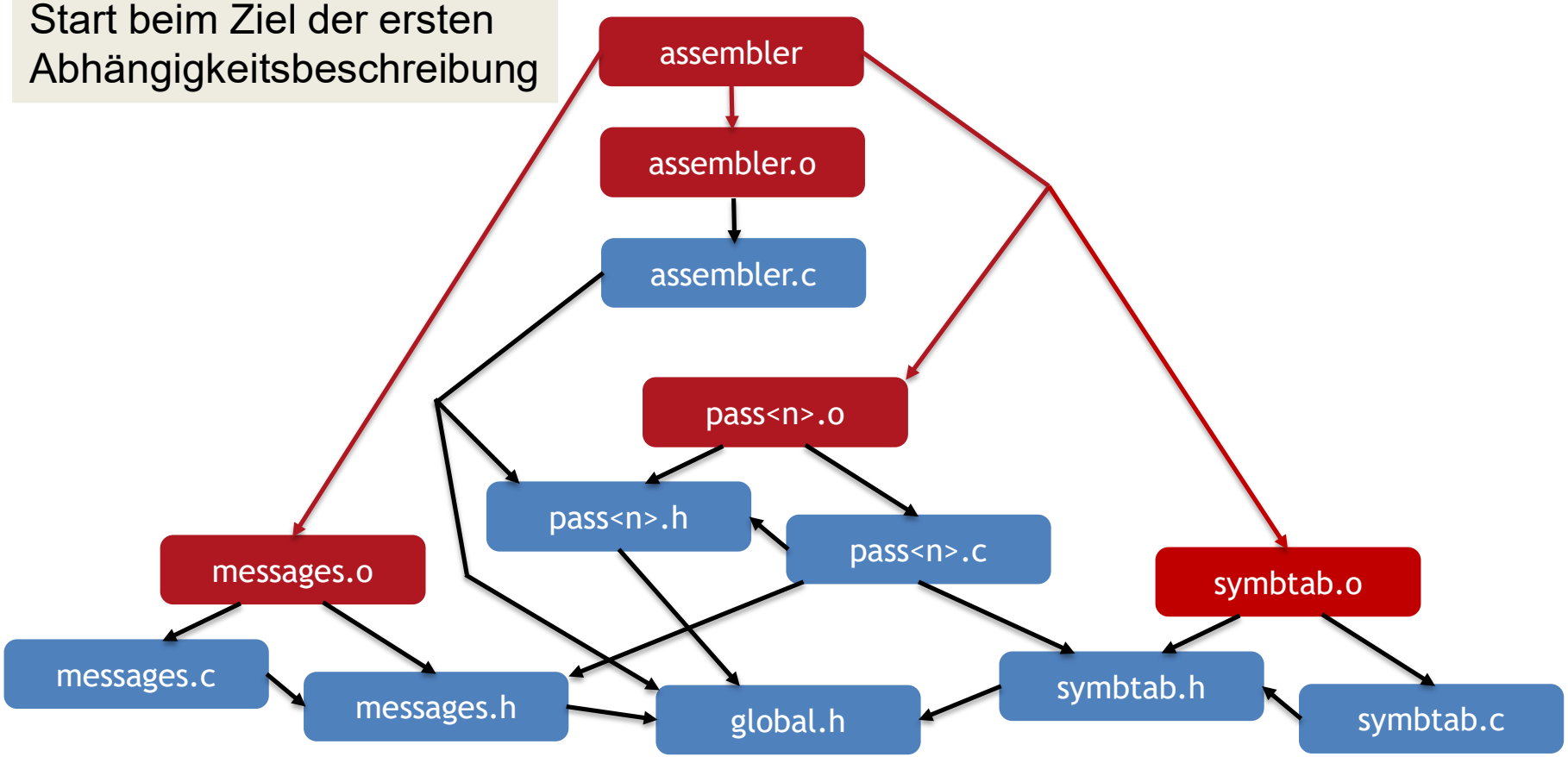
# Beispiel: Aufruf von „make“

Aufruf von *make* nach Änderung von *messages.c*

1. Suche nach Datei namens *makefile*
2. Erzeugen der aktuellen Version des Ziels der **ersten** Abhängigkeitsbeschreibung (hier: *assembler*).
3. Prüfen, ob Ziel noch aktuell ist, durch **rekursive** Prüfung ob alle *o-Dateien*, von denen Ziel abhängt, aktuell sind (falls sie existieren, sonst generieren).

# Beispiel: Rekursive Suche im Abhängigkeitsgraphen

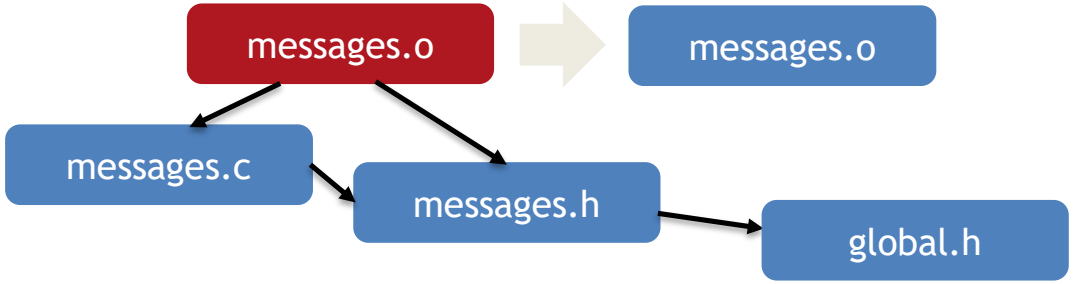
Start beim Ziel der ersten Abhängigkeitsbeschreibung



# Beispiel: Aufruf von „make“ ohne Parameter

1. Suche der der *makefile*
2. Prüfen, ob erstes Ziel (*assembler*) in der *makefile* noch aktuell ist.
3. Prüfen, ob alle *o-Dateien*, von denen *assembler* (direkt) abhängt, aktuell sind / existieren.
   - *messages.o:* hängt nur von Dateien ab, für die es keine eigenen Regeln gibt. Rekursiver Prozess bricht ab und es wird die Zeitmarke von *messages.o* mit den Zeitstempeln von *messages.c*, ... verglichen.

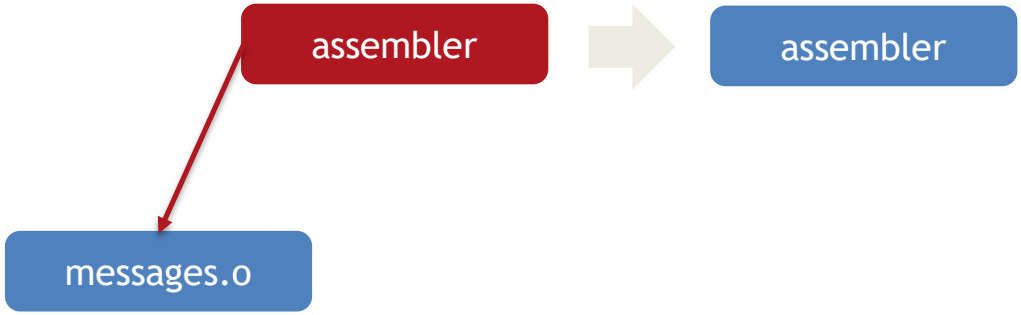
*messages.o* hat eine ältere Zeitmarke als (geänderte) *messages.c* und wird durch den Compiler neu erzeugt.



# Beispiel: Aufruf von „make“ ohne Parameter

– Prüfen, ob Dateien, die von messages.o abhängen, noch aktuell sind.

*assembler* besitzt nun eine ältere Zeitmarke als *messages.o* und wird neu mit dem Binder (Linker) erzeugt.



# Beispiel: 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 assembler-unix erzeugt Unix-Variante

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 erzeugt Windows-Variante

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
```

# Makefiles mit Makros

- ${Makroname} wird durch die Makrodefinition ersetzt.
- Makros können **selbstdefiniert** oder **vordefiniert** sein:

```
– 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
```

UNIVERSITÄT SIEGEN

# Beispiel: Makefiles mit selbstdefinierten Makros

```
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
```

# Weitere Makros in Makefiles

„$@“ bezeichnet das Ziel einer Regel.

Beispiel:

```
assembler:   $@.o pass0.o pass1.o pass2.o pass3.o \
             messages.o symbtab.o files.o
```



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

# Weitere Makros in Makefiles

„$^“ bezeichnet alle Objekte einer Regel rechts vom „:“.

Beispiel:

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

# Weitere Makros in Makefiles

„$*“ bezeichnet das Ziel einer Regel ohne Suffix.

Beispiel:

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



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

# Weitere Makros in Makefiles

„$?“ bezeichnet alle Objekte einer Regel rechts vom „:“, die **neuer** als das Ziel sind.

„$<„ bezeichnet das erste Objekt rechts vom „:“.

# Mustersuche

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).

# Mustersuche

„$(wildcard Muster)“ liefert bzw. **sucht** alle Dateien im aktuellen Verzeichnis, die dem angegebenen Muster entsprechen. Das Muster kann beispielsweise folgende Form haben.

Beispiel:

„$(wildcard *.c)“ liefert alle Dateien zurück, die das Suffix „c“ besitzen („*“ matcht wie üblich alles).

# Substitution

Häufig will man in einer Liste von Dateinamen jeweils einen Teilstring/Muster durch einen anderen Teilstring/Muster ersetzen bzw. **substituieren.**

# Substitution

„$(patsubst Suchmuster, Ersatzstring, Liste von Wörtern)“ ersetzt in „Liste von Wörtern“ jedes auftreten von „Suchmuster“ durch „Ersatzstring“.

Beispiel:

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

- Ersetzt alle „c“-Suffixe durch „o“-Suffixe.
- „%“ bezeichnet den gleichbleibenden Teil bei der Ersetzung.

# Beispiel: Suchmuster und Substitution

```
...
h-files = $(wildcard *.h)
# alle h-Dateien im aktuellen Verzeichnis werden dem Makro 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
```

# Suffix-Regeln

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.

# Suffix-Regel

„prefix1%suffix1 : prefix2%suffix2
    Kommando“

Beispiel: Übersetzung von c-Dateien in o-Dateien.

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

# Übungsaufgabe: makefile

Zu schreiben ist folgende makefile:

- Im aktuellen Verzeichnis sollen alle Dateien mit der Endung *.gif* in Dateien mit der Endung *.jpg* übersetzt werden (Kommando *convert*).
- Konvertierung nur dann durchführen, 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 (Kommando *zip*).
- Die *zip*-Datei nur erzeugen, wenn mindestens eine *jpg*-Datei neu erzeugt wurde (nur neue *jpg*-Dateien werden im Archiv ausgetauscht).

**Einfache Variante:** Das aktuelle Verzeichnis enthält immer nur die Dateien *tobias.gif*, *johannes.gif*, *markus.gif* und *andy.jpg*.

**Schwierige Variante:** Das makefile muss für beliebig viele *gif*- und *jpg*-Dateien im Verzeichnis funktionieren.

# Lösung: Einfache Variante

```
default:            images.zip

images.zip:         tobias.jpg johannes.jpg markus.jpg
        zip -u images.zip tobias.jpg johannes.jpg markus.jpg   # update files

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
```

# Lösung: Schwierige Variante

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

default : images.zip

images.zip : ${jpg-files}
        zip -u $@ $?        # nur die geänderten Dateien neu packen

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

# Diskussion: „make“

- Hier nur ein Bruchteil der Funktionen vorgestellt.
- *makedepend* zum Erzeugen von programmiersprachenspezifischen makefiles.
- *GNU make* kann Arbeit auf mehrere Rechner verteilen und parallel durchführbare Erzeugungsschritte gleichzeitig starten.
- *GNU Autotools* enthält weitere Hilfswerkzeuge zum Buildmanagement basierend auf *make*.
- …

# Diskussion: „make“

Steuerung durch Zeitmarken 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.

# Diskussion: „make“

Kaum Verzahnung mit Versionsverwaltung:

- *make* wird in aller Regel auf eigenem Repository durchgeführt.
- Erzeugte/abgeleitete Objekte werden also immer privat gehalten.

UNIVERSITÄT SIEGEN

# Änderungsmanagement

# Definition: Änderungsmanagement

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

# Änderungsmanagementprozess

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)**

# Änderungsmanagementprozess

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).**

# Änderungswerkzeuge

- „Open Software“-Produkt **Sourceforge** (GForge) kombiniert SVN/GIT/... mit Änderungsmanagement (http://sourceforge.net)
- „Open Software“-Produkt **Bugzilla** für reines Änderungsmanagement (http://bugzilla.mozilla.org/)
- Flexibles Projektmanagement-Werkzeug **Redmine** mit Aufgabenverwaltung und Versionsverwaltung SVN/GIT/… (http://www.redmine.org/)
- Neuere Alternativen: **GitHub**, **GitLab**, **BitBucket**,…

# Fazit: Checkliste für Software-Projekte

- Besteht das System aus mehr als einer Handvoll Dateien, so sollte ein **Buildmanagementsystem** wie make/Ant/Maven verwendet werden.
- Ist die Entwicklungszeit mehr als ein paar Tage oder/und umfasst das Projekt mehr als einen Entwickler, so sollte ein **Versionsmanagement-system** genutzt werden.

# Fazit: Checkliste für Software-Projekte

- Gibt es mehr als einen Anwender oder mehr als ein Release der Software, so ist ein **Änderungsmanagementsystem** einzusetzen.
- Freie Werkzeuge wie Sourceforge, GitHub, GitLab, ..., die Versionsmanagement mit Bugtracking, Wiki-Dokumentation etc. kombinieren, sind immer zu empfehlen.

# Prüfungsstoff

- Problemstellungen und Ziele der verschiedenen Aufgabengebiete des Konfigurations-managements kennen und erläutern.
- Wichtige Werkzeuge für die Aufgabengebiete kennen, bewerten und auswählen.
- Die Arbeitsweise der Operationen *diff*/*merge* und des Werkzeugs *make* grundlegend beherrschen.
- Überlegungen zur Planung des Konfigurations-managements für (kleine) Projekte durchführen können.

# Literatur


- ***DIN EN ISO 10007:*** Leitfaden für Konfigurationsmanagement (1996).
- ***IEEE Standard 1042-1987: IEEE Guide to Software Configuration Management*,** IEEE Computer Society Press (1988).
- G. Veersteegen (Hrsg.), G. Weischedel: ***Konfigurationsmanagement***, Springer (2003)
- G. Versteegen, K. Salomon, R. Heinhold: ***Change Management bei Software-Projekten***, Springer (2001)
- A. Zeller, J. Krinke: ***Programmierwerkzeuge: Versionskontrolle - Konstruktion - Testen - Fehlersuche unter Linux*,** dpunkt-Verlag (2000)