Dipl.-Ing. Frank Sell
System Engineer Unix

: www.xpile.de
: fsell@xpile.de

Xpile
Software Entwicklung

Tips und Kniffe

im Umgang mit Versionsverwaltungen

Dokument: versionsverwaltung

Erstellungsdatum: 16. August 2005
Letzte Bearbeitung: 14. Januar 2014
Version: 9

Inhaltsverzeichnis

1 Einleitung

Versionsverwaltungen sind bekanntlich ein unverzichtbarer Bestandteil einer jeden Entwicklungsarbeit. Sie gewährleisten, dass alle Versionen eines Files im Repository gesichert sind und mit Hilfe einer konkreten Sicht auch wieder verfügbar sind.

Die zur Zeit gängigsten Versionsverwaltungen sind derzeit ClearCase und CVS. Natürlich gibt es weitere Tools wie z.B. SCCS, RCS und andere. Im folgenden Text wird aber im wesentlichen auf ClearCase-Beispiele zurückgegriffen.

Es wird nochmals kurz auf allgemein anerkannten Regeln eingegangen sowie insbesondere auf immer wieder auftretende Tücken und Schwierigkeiten hingewiesen.

Weiterhin wird ein kurzer Ausblick über den möglichen Workflow von der Anforderung einer Entwicklung, den Entwicklungsprozess im Development-Team bis zur Produktionseinführung durch ein Deployment-Team gegeben.

2 Sinn und Zweck von Versionsverwaltungen

In der Vergangenheit wurden Dokumente, Programmquellen und sonstige Dateien entweder keiner Versionskontrolle unterworfen oder aber verschiedene Stände solcher Dateien wurden mehr oder weniger sinnvoll an unterschiedlichen Stellen auf dem Speichermedium hinterlegt.

Dadurch kommt es mit der Zeit zu einem Zustand, von dem Niemand mehr weiß, was wofür aktuell ist und wo welcher Stand gespeichert ist.

Genau hier sollen Tools zur Versionsverwaltung einsteigen und die Unübersichtlichkeit eindämmen. Es geht also im wesentlichen darum, Änderungen an Dateien so zu protokollieren, dass jederzeit ein beliebiger Stand rekonstruiert werden kann.

Nachfolgend wird gezeigt, wie dieses zu erreichen ist.

2.1 Lineare Versionskontrolle

Die einfachste Art, eine Datei zu versionieren ist es, Änderungen immer linear zu protokollieren. In kleinen Projekten oder bei Dateien, welche sich selten ändern, wird dieses die häufigste Art der Versionierung sein.

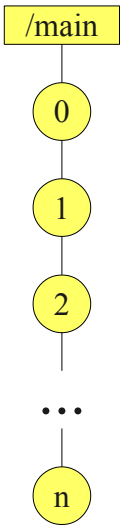

Bild 1: Beispiel für lineare Speicherung in ClearCase

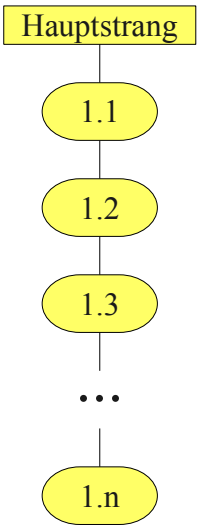

Bild 2: Beispiel für lineare Speicherung in CVS

Leider ist diese Art der Versionierung bei Dateien, welche mehrfach mit unterschiedlichen Inhalten ausgeliefert werden, nicht mehr praktikabel.

Natürlich kann man einfach mehrere Dateien anlegen. Es wird aber z.B. nicht mehr einfach möglich aus einem Masterdokument auf die richtige Datei zu verweisen.

2.2 Parallele Versionskontrolle

Besser ist es, wenn eine Datei mit verschiedenen Inhalten parallel verfügbar ist.

Alle gängigen Tools sind in der Lage Versionen von Files sowohl in einer Linie zu halten, als auch von einem beliebigen Element einer Linie einen Ast (Branch) abzuzweigen.

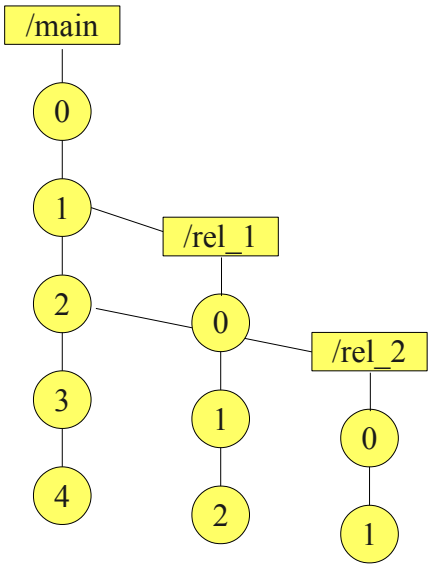

Bild 3: Beispiel für einen ClearCase-Baum

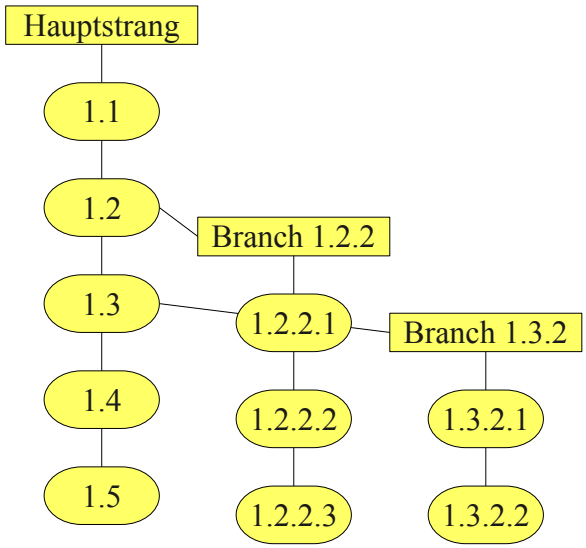

Bild 4: Beispiel für einen CVS-Baum

Durch diese Funktionalität ist es auf einfache Art möglich, Dateien für z.B. verschiedene Kunden zu pflegen.

In der Softwareentwicklung ist dieses Verhalten ein unverzichtbarer Bestandteil zur parallelen Bearbeitung von Quellcode.

3 Labeling der Versionen

Für eine gute Nachvollziehbarkeit, welche Dateien innerhalb eines Arbeitsauftrages bearbeitet wurden oder zu einer Lieferung für einen Kunden zusammengefasst wurden ist es unbedingt nötig, diese zu kennzeichnen.

Dazu bieten die Versionsverwaltungen den Mechanismus des Labelings / Taggings an. D.h. eine bestimmte Version eines Files wird mit einem symbolischen Namen belegt und ist über diesen eindeutig identifizierbar.

Damit braucht sich der Benutzer der Versionsverwaltung keine Gedanken über Versionsstände zu machen. Die Sicht über dies Label / Tags bestimmt, was in der Versionsverwaltung sichtbar ist.

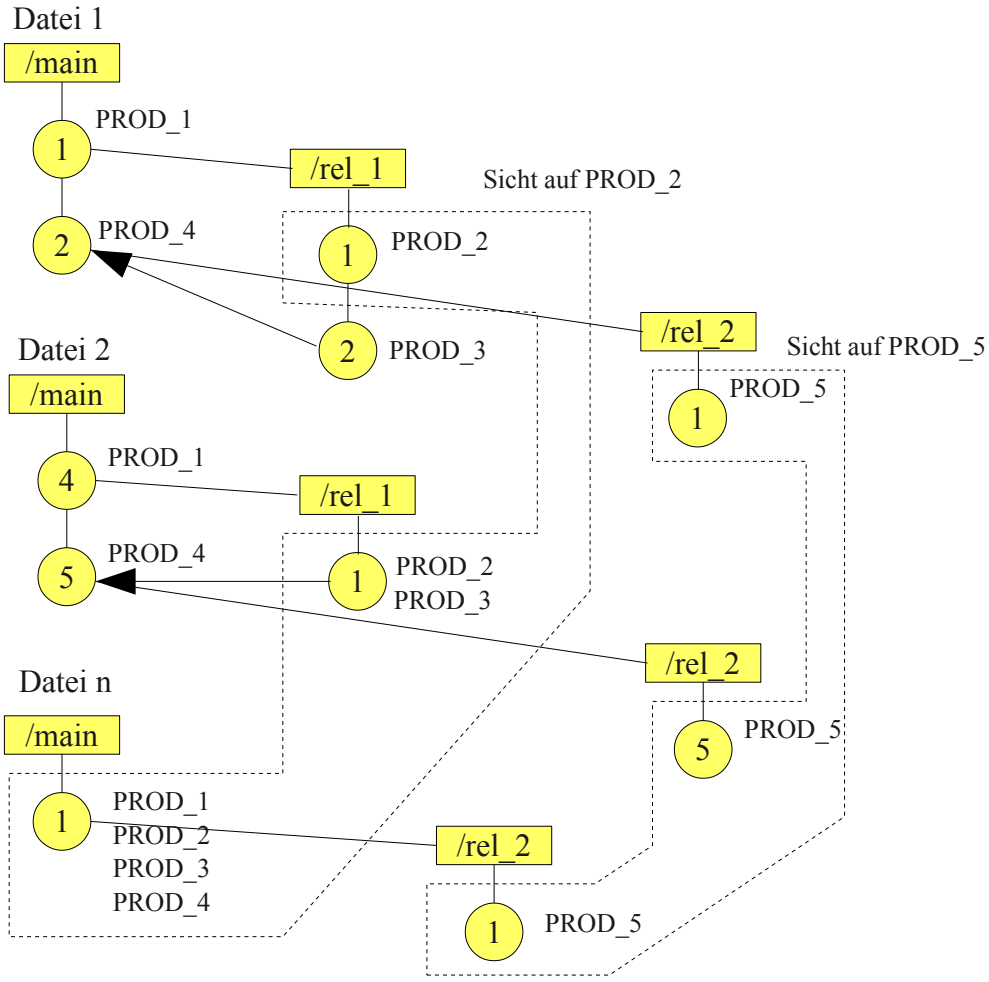

Bild 5: Beispiel für das Labeling von Dateien in ClearCase

Die gestrichelt zusammengefassten Dateistände im obigen Beispiel stellen eine mögliche Sicht auf die Versionsverwaltung dar.

4 Eindeutige Identifizierung von Lieferobjekten

Aus Qualitätsmanagement-Sicht gibt nichts Schlimmeres, als einem Kunden eine Software zu liefern, bei der nicht mehr zu ermitteln ist, aus welchen Bestandteilen sie gebaut wurde.

Dazu gibt es bei den Versionsverwaltungen 'CVS' und 'RCS' den Mechanismus der Keywords.

Leider bietet 'ClearCase' auf den ersten Blick diese Funktionalität nicht.

Umgehen lässt sich dieses Manko durch den Einsatz von Triggern. Doch dazu später.

Um aus einem Executable oder einem Shell-Script Informationen zu gewinnen, bieten sich die Tools 'ident' oder 'what' an.

Das Tool 'ident' sucht im angegebenen File nach RCS-Keyworte der Form: '$keyword: text $' und gibt diese formatiert aus. Das Tool 'what' arbeitet ähnlich. Allerdings sucht dieses Programm nach dem Vorkommen des SCCS-Pattern '@(#)'.

Da nicht auf allen Umgebungen beide Tools verfügbar sind, sollten beide o.g. Pattern in den zu versionierenden Files / Quellen verankert werden.

Um eine brauchbare Information zu erhalten, bieten sich folgende RCS-Keywords an:

$RCSfile$	Name des Files ohne Pfadanteil
$Source$	Name des Files mit Pfadanteil
$Revision$	Revisionnummer des Files
$Header$	Voller Pfadname des Files, Revisionsnummer und diverse andere Angaben
Id	Nur Name des Files, Revisionsnummer und diverse andere Angaben

Für vernünftige Informationen sollten die Keywords: '$Source$' und '$Revision$' preferiert werden. Somit ergibt sich dann folgender Keyword-String:

```
"@(#) $Source$ $Revision$"
```

Das nachfolgende Beispiel für eine C-Quelle veranschaulicht die Funktionalität.

```
static char bla_c_rcsid[] = "@(#) $Source$ $Revision$";
```

wird z.B. expandiert zu

```
static char bla_c_rcsid[] = "@(#) $Source: /opt/cvs/src/bla.c,v $ $Revision: 1.2 $";
```

Wie im obigen Beispiel zu sehen ist, wird die statische Charakter-Variable nicht einfach nur 'rcsid' genannt. Es ist von Vorteil, wenn sich der Dateiname und die Dateiendung mit im Variablenname befindet.

Damit werden von vornherein Konflikte z.B. mit Variablennamen includierter Headerfiles ausgeschlossen.

Für ein Shell-Script würde sich dann folgender Keyword-String anbieten:

```
# @(#) $Source$ $Revision$
```

Jetzt noch ein Wort zu 'ClearCase' und den RCS-Keywords.

Wie bereits oben bemerkt, unterstützt 'ClearCase' von Hause aus die Substitution der Pattern

nicht. Aber man kann sich mit einem Trigger behelfen. Dieser Trigger muss dann vor dem Checkin der entsprechenden Files tätig werden. Entsprechend Lösungen sind bereits verfügbar bzw. können leicht implementiert werden. Dieser Trigger hat dann die Aufgabe, den RCS-String entsprechend anzupassen.

Das obige Beispiel verhält sich dann z.B. so:

```
static char bla_c_rcsid[] = "@(#) $Source$ $Revision$";
```

wird z.B. expandiert zu

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c $ $Revision: main/2 $";
```

natürlich kann man den Trigger so gestalten, dass er das Keyword 'Revision' unterdrückt und statt dessen die Version gleich an das Sourcefile anhängt (wie normalerweise bei ClearCase).

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/2 $";
```

Der Trigger kann auch ab prüfen, ob in dem einzucheckenden File die Keywords richtig eingetragen sind. Und wenn dem nicht so ist, kann der Checkin mit einer entsprechenden Fehlermeldung abgewiesen werden.

Natürlich kann man nicht in jedem File RCS-Keywords verankern. Besonders Binaries vertragen dieses nicht besonders gut. Deshalb ist es notwendig, automatisch zu entscheiden, ob ein Keyword gefordert ist. Sinnvoll ist dieser Triggermechnismus z.B für Makefiles, c/c++-Files, Perl- und Shell-Scripte, sql-Files, Java-Sourcen etc.

Mit einer konsequenten und disziplinierten Anwendung dieser Maßnahme ist eine gute Identifikation der gelieferten Dateien zur Version in der Versionsverwaltung gewährleistet.

Das bringt uns auch gleich auf eine neue Besonderheit der Versionsverwaltungen.

5 Mergebarkeit von Fileversionen

Bei jeder ernsthaft betriebenen Entwicklungsarbeit wird auf die Vorzüge des 'Abbranchen' (siehe Parallele Versionskontrolle) von stabilen Versionen zurückgegriffen.

Nun ist es aber nicht so, dass die Blätter und Äste im Versionsbaum eines Files so liegen gelassen werden.

Im Gegenteil sollte das Bestreben dahin gehen, die Entwicklungsarbeiten wieder auf einen stabilen Stand zurückzuführen, sprich zu mergen.

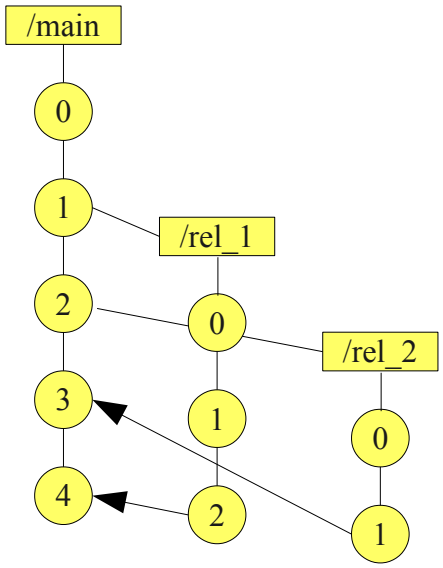

Bild 6: Mergebeispiel für einen ClearCase-Baum

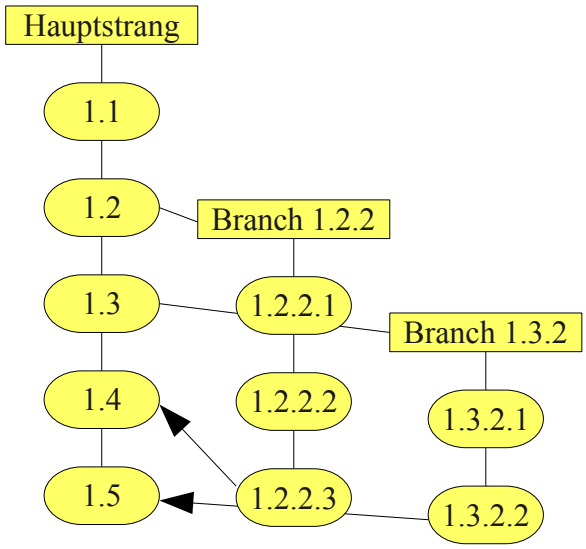

Bild 7: Mergebeispiel für einen CVS-Baum

In den oben gezeigten Bespielen ist die Mergearbeit nicht sonderlich hoch.

Die Versionsverwaltungs-Tools sollten bei dieser Konstellation in der Lage sein, den Merge automatisch durchzuführen.

Anders sieht es allerdings aus, wenn wirklich eine massive parallele Arbeit der Entwicklungsteams vorliegt.

Es kann durchaus sein, das bei paralleler Arbeit in einer Datei an genau gleichen Stellen Änderungen durchgeführt werden. Hier ist der Merge dann nicht mehr trivial.

Eine prädestinierte Stelle ist z.B. eine Zeile, welche RCS – Keywords enthält.

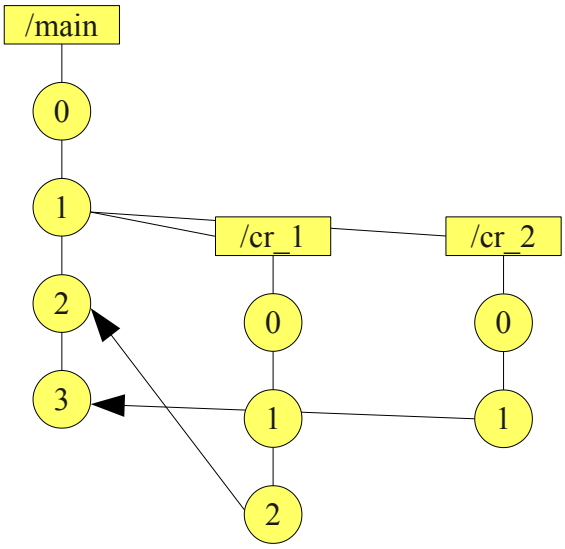

Bild 8: Mergebeispiel für einen ClearCase-Baum bei parallel verlaufenden Entwicklungen

Wie hier zu sehen ist, branchen alle drei Entwicklungslinien von der Version /main/1 ab.

Es wurde hier bewusst auf ein 'ClearCase'-Beispiel zurückgegriffen, weil es unter CVS trotzdem möglich ist, diesen Merge automatisch durchzuführen. Der Schalter -kk beim Aufruf

```
cvs update -kk -j cr_1
```

unterdrückt die Keyword-Expansion und lässt zumindest an dieser Stelle keine Mergekonflikte aufkommen.

Der Merge für den cr_1 lässt sich in der Regel noch automatisch durchführen. Das Versionsverwaltungssystem erkennt lediglich eine Änderung im String

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/1 $";
```

nach

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/cr_1/2 $";
```

und merged diesen korrekt.

Allerdings wird beim Einchecken der Datei der weiter oben beschriebene Trigger-Mechanismus aktiv und ändert den String auf

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/2 $";
```

Der Merge für den cr_2 lässt sich allerdings nun nicht mehr automatisch durchführen.

Das Versionsverwaltungssystem muss jetzt drei unterschiedliche Versionen miteinander vergleichen.

Verglichen werden die Änderung von

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/1 $";
```

nach

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/cr_2/1 $";
```

sowie

```
static char bla_c_rcsid[] = "@(#) $Source: /vob/src/bla.c@@/main/2 $";
```

An dieser Stelle scheitert der automatische Merge.

Hier ist es unbedingt nötig, das normale Verhalten von 'ClearCase' zu manipulieren. Ein Trigger für das Pre-Checkin und Post-Checkout hilft auch hier nicht wirklich.

Vielmehr sollte hier der 'ClearCase'-Typemanager manipuliert werden.

Dieser Typemanager kann für verschiedene Aktionen aufgerufen werden. An dieser Stelle ist es sinnvoll, einen angepassten Manager für die Aktionen 'merge' und 'xmerge' zu benutzen.

Dieser Manager bereinigt die RCS-Keywords vor dem eigentlichen Mergen der Dateien.

In gängigen Implementationen ist es so gelöst, dass sich der angepasste Manager von der Schnittstelle her wie der originale verhält. D.h. es werden alle Aufrufparameter vom 'ClearCase'-System übernommen.

Anschließend wird das originale zu mergende File in ein temporäres File kopiert, wobei alle Keywords und ihre Erweiterungen bis auf sich selbst zurückgeführt werden. Anschließend kann der originale Merge-Manager aufgerufen werden. Dieser behandelt die zu mergenden Files in der gewohnten Art und Weise.

Der Wrap-Manager muss dafür Sorge tragen, dass nicht jede Art von Files in der oben beschriebenen Art und Weise behandelt wird. Diese Verfahrensweise ist nur auf Files anzuwenden, welche auch durch die unter Eindeutige Identifizierung von Lieferobjekten beschriebenen Trigger behandelt werden.

Um dieses Verhalten für bestimmte Files zu erreichen, sollte ein neuer File-Typ im ClearCase bekannt gemacht werden.

Z.B. ein Type 'keyed_text_file':

```
cleartool mkeltype -supertype text_file -manager keyed_text_file_delta \
-c "RCS keyword type" eltype:keyed_text_file@vob:<your-vob>
```

Der dazugehörige Manager 'keyed_text_file_delta' muss dann natürlich auch entsprechend installiert werden. Für nähere Informationen sei auf die ClearCase-Dokumentation verwiesen.

Der Type 'keyed_text_file' kann jetzt auch für bestimmte Files konfiguriert werden. Dazu verfügt ClearCase über einen Magic-File-Mechanismus, der anhand einer Konfigurationsdatei 'default.magic' den richtigen Filetype beim Anlegen eines Files im Repository ermittelt.

Die Datei 'default.magic' kann dann beispielsweise wie folgt erweitert werden:

```
# Erweiterungen fuer die RCS Keywords
keyed_text_file text_file :
            -name "*.[sS][hH]"
          | -name "*.[kK][sS][hH]"
          | -name "*.[bB][aA][sS][hH]"
          | -name "*.[pP][lL]"
          | -name "*.[pP][mM]"
          | -name "*.[sS][qQ][lL]" | -name "*.[pP][lL][sS][qQ][lL]"
          | -name "*.[bB][aA][tT]"
          | -name "*.[tT][xX][tT]"
```

```
| -name "*.[cC][bB][sS]"
| -name "*.[hH]" | -name "*.[hH][pP][pP]"
| -name "*.[cC]" | -name "*.[cC][pP][pP]" | -name "*.[eEpP][cC]"
| -name "*.4[gG][lL]"
| -name "*.[pP][eE][rR]"
| -name "*.java" | -name "*.JAVA" ;
```

An dieser Stelle noch ein Hinweis:

Einige Programme, die im html- oder xml-Umfeld erzeugen Dateien mit Zeilenlänge > 8000 Charakter. Diese Dateien legt ClearCase in der Regel als 'text_file' an. Leider kann der dazu gehörende Mergemanager solche Files nicht mehr mergen. Es bietet sich an, solche Files zum Type 'compressed_file' zu konvertieren oder gleich in diesem Type anlegen zu lassen:

```
# generierte mdl-Files koennen durchaus mehr als 8000 Charakter in der
# Zeile haben
compressed_file : -name "*.mdl" ;
```

Die Einführung eines eigenen File-Typs und des angepassten Mergemangers schützt natürlich nicht vor Mergekonflikten an anderen Stellen. Aber die Wahrscheinlichkeit, das zwei Entwickler das File an genau der gleichen Stelle bearbeiten ist relativ gering.

Falls dieses trotzdem auftreten sollte, müssen die Entwickler entscheiden, welche Änderungen letztendlich relevant sind und gemerged werden sollen.

Auf die Installation dieses Wrap-Typemanagers wird hier nicht weiter eingegangen, sondern auf die 'ClearCase' Dokumentation verwiesen.

6 Paketierung versus Monsterinstallation

An dieser Stelle muss man eine Unterscheidung zwischen relativ kleinen und überschaubaren Vorhaben und Projekten mit mehreren Produkten, an denen eine größere Anzahl von Mitarbeitern involviert ist, treffen.

Bei kleinen Projekten ist es sicherlich nicht von Belang, wenn der Kunde nach einem Releasewechsel oder nach Fehlerbehebungen eine Gesamtlieferung des kompletten Produktes über stellt bekommt.

Anders sieht es aus, wenn Projekte oder Vorhaben eine gewisse Größe überschreiten. D.h. wenn eine Lieferung aus mehreren voneinander unabhängigen Teilkomponenten besteht.

Hier bietet es sich an, unabhängige Komponenten auch voneinander losgelöst liefern zu können, ohne das bei der Auslieferung Rückwirkungen auf andere Komponenten des Gesamtsystems auftreten.

Folgendes Beispiel soll diesen Sachverhalt verdeutlichen.

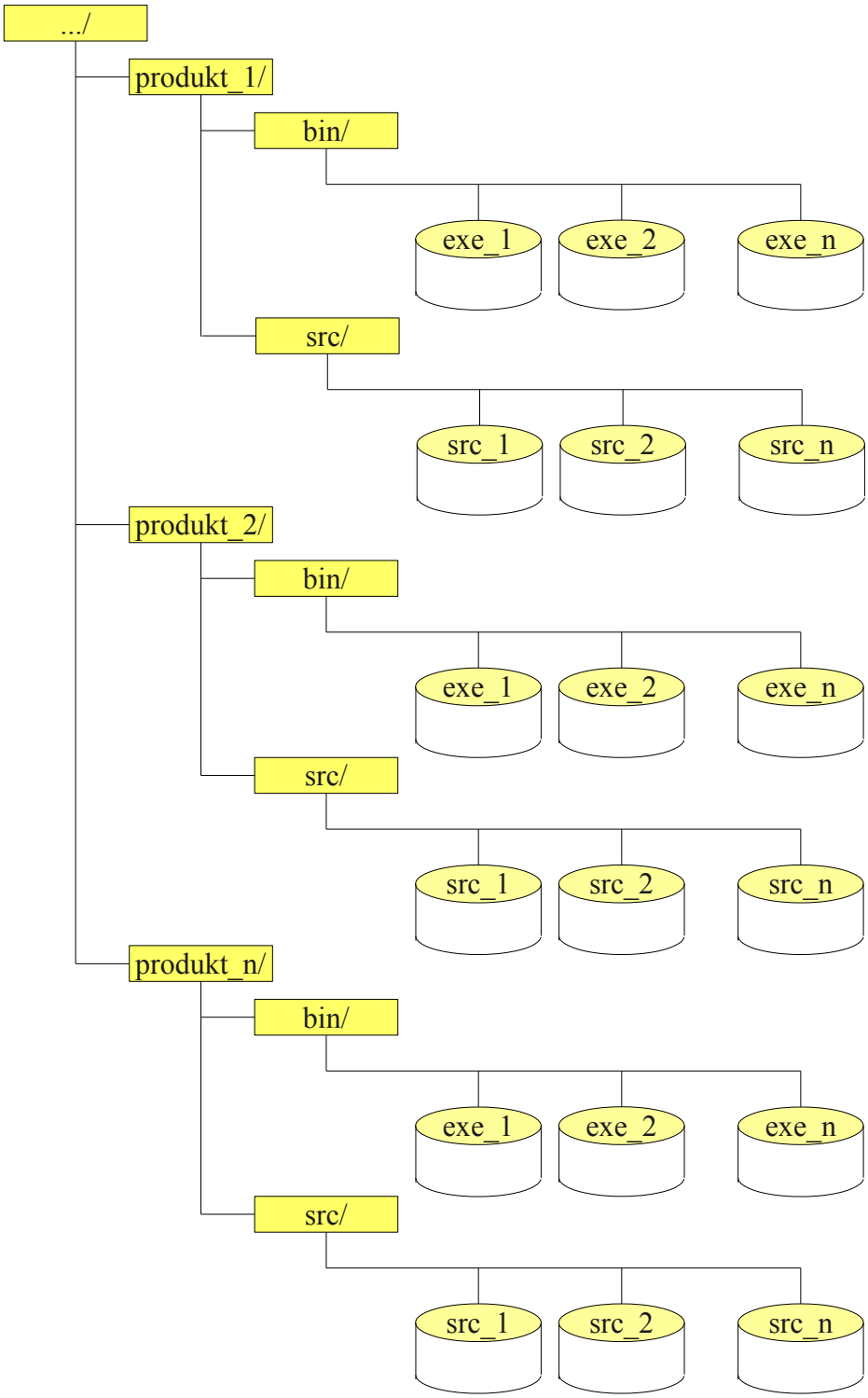

Bild 9: Beispiel für einen Verzeichnisbaum in einem Projekt mit verschiedenen Produkten

Aus dem Bild geht hervor, dass in Versionsverwaltungen durchaus mehrere verschiedene Projekte/Produkte verwaltet werden können.

Im Interesse einer sauberen Arbeit innerhalb der Versionsverwaltung sollte dafür Sorge getragen werden, die Freiheiten der Bearbeitung von Files auf konkrete Projekte einzuschränken.

Dieses birgt allerdings auch die Gefahr, die Projekte so fein zu strukturieren, dass Änderungen in einem Projekt auch Änderungen in einem oder mehreren anderen Projekten nach sich ziehen.

Folgendes Beispiel verdeutlicht dieses Problem.

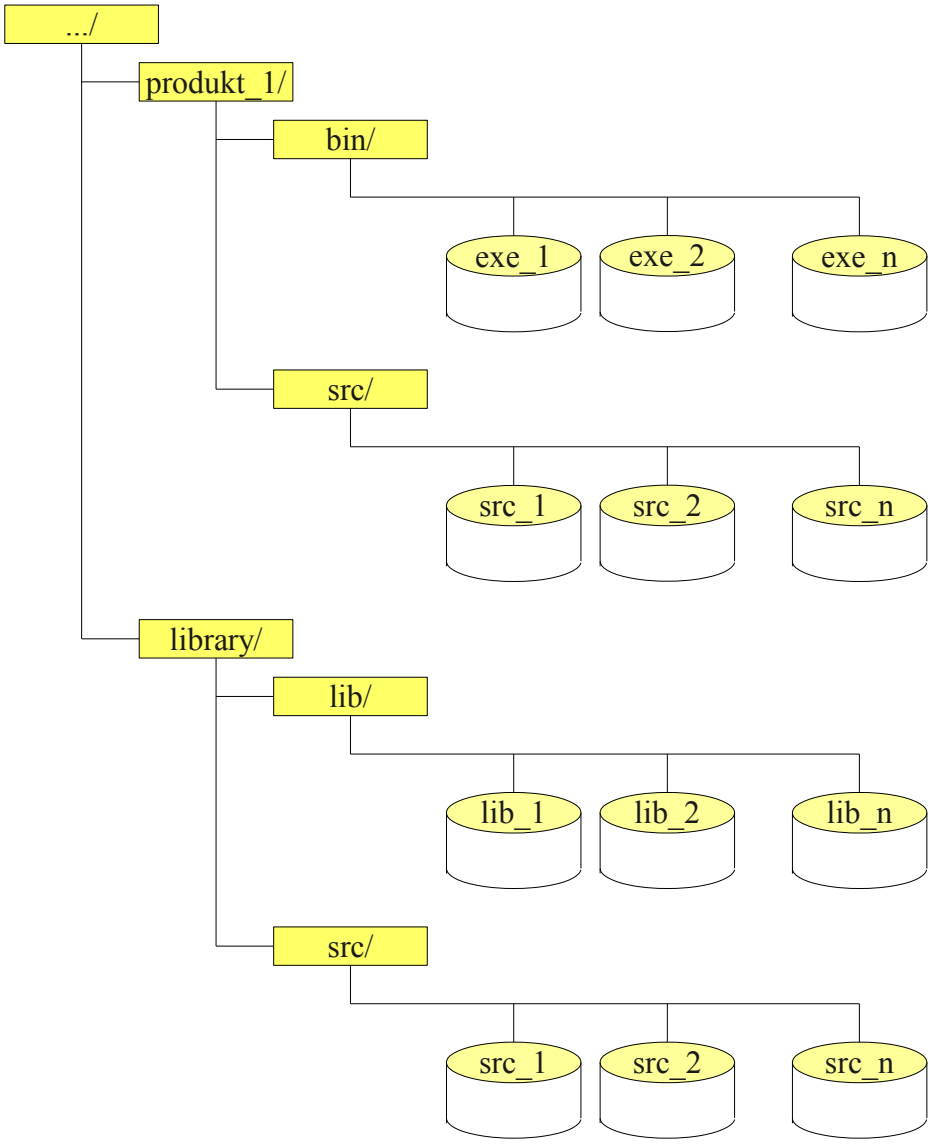

Bild 10: Beispiel für einen Verzeichnisbaum mit einem Projekt und einer Library

Eine Änderung in der Projekt 'library' kann eine Änderung im Projekt 'produkt_1' nach sich ziehen.

Dieses bedeutet, dass der Entwickler gezwungen ist in zwei Projekten gleichzeitig zu arbeiten. Das ist an sich weiter nicht tragisch, bedeutet aber, dass für beide Projekte Arbeitsaufträge generiert werden müssen und diese dann als untereinander abhängig gekennzeichnet werden müssen.

Eine solche Vorgehensweise erfordert eine entsprechend gute Arbeit des Release-Managements.

Es ist also immer ein gesundes Mittelmaß bei der Definition der einzelnen Projekte zu finden.

7 Bau von Config-Specs

Bei größeren Projekten bietet sich natürlich an, die Sicht auf die einzelnen Produkte so zu gestalten, dass immer für das jeweilige Verzeichnis eine konkrete Sicht auf die Versionsverwaltung eingestellt wird.

In einer ClearCase-Umgebung mit mehreren Produkten kann eine mögliche Config-Spec z.B. folgendes Aussehen haben:

```
# Generelles Auschecken zulassen
element * CHECKEDOUT

# kein Auschecken für produkt_1 zulassen
# Das Verzeichnis ist rekursiv mit dem Label PRODUKT_1 versehen
element /vob/produkt_1/... PRODUKT_1 -nocheckout

# Auschecken für produkt_2 zulassen
# Das Verzeichnis ist rekursiv mit dem Label PRODUKT_2 versehen
# Jeder Checkout erzeugt automatisch einen Branch 'notfall_fix_555'
element /vob/produkt_2/... .../notfall_fix_555/LATEST
element /vob/produkt_2/... PRODUKT_2 -mkbranch notfall_fix_555
# falls ein Element noch nicht existieren sollte:
element /vob/produkt_2/... .../main/0 -mkbranch notfall_fix_555

...
```

Diese Config-Spec ist für die Bearbeitung des Projektes 'podukt_2' vorgesehen und lässt auch nur unterhalb des Verzeichnisses 'produkt_2' ein Bearbeiten zu.

Es muss davon ausgegangen werden, dass die Produktionsstände immer sauber durchgelabelt werden. Ansonsten würde bei einer existierenden Datei, welche das Label 'PRODUKT_2' nicht trägt, vom Element 'main/0' abgebrancht. Dieses Element entspricht dann einer leeren Datei.

Ein Checkout eines Files oder Verzeichnisses führt in diesem Beispiel automatisch zum Anlegen der Branch 'notfall_fix_555'.

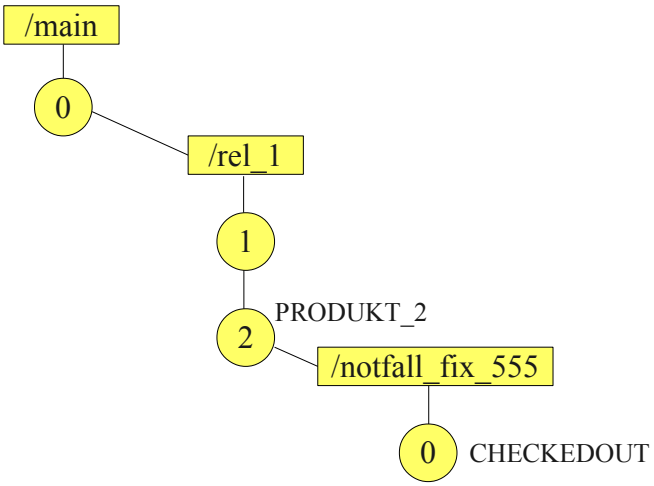

Bild 11: Beispiel für das automatische Abbranchen beim Checkout in ClearCase

Für eine CVS-Umgebung gestaltet sich die Situation etwas anders. CVS kennt nicht den bei

ClearCase genutzten Mechanismus der 'Config-Spec'.

Die korrekte Sicht auf die entsprechenden Verzeichnisse lässt sich nur durch den Aufruf 'cvs update' einstellen.

Weiterhin kennt CVS nicht die Funktionalität des automatischen Abbranchen beim Checkout.

Hier muss für jedes Verzeichnis ein entsprechender Aufruf abgesetzt werden:

```
# Sicht auf Stabile Version produkt_1
cvs update -A -d -P -R -r PRODUKT_1 produkt_1

# Branch erzeugen
cvs rtag -b -r PRODUKT_2 notfall_fix_555 produkt_2
# Sicht auf Branch-Version einstellen
cvs update -A -d -P -R -r notfall_fix_555 produkt_2

...
```

Diese Art der File-Bereitstellung in CVS hindert den Entwickler leider nicht, im Verzeichnis 'produkt_1' Files zu bearbeiten und diese auch wieder in die Versionsverwaltung einzustellen.

Dieses sollte mit geeigneten Mitteln unterbunden werden (siehe Transparente Arbeit mit 'CVS' und 'ClearCase').

8 Merges der Entwicklerbranches in den Produktionsstand

In einer halbwegs gut strukturierten Versionsverwaltung sollte der Versionsbaum auch die Entwicklung widerspiegeln.

Es ist vorteilhaft, von vornherein in Release-Branches zu arbeiten. Weiterentwicklungen bzw. Bugfixe branchen dann von dem zum Zeitpunkt der Bearbeitung letzten Stand des Releases bzw. vom letzten als Produktionsstand gelabelten Element ab.

Soll die Weiterentwicklung oder der Bugfix produktiv werden, wird die abgebranchte Version auf den Release-Branch zurückgemerged.

Bei einem Release-Wechsel ist es vorteilhaft, den entsprechenden neuen Release-Branch wieder bei der Wurzel aufzusetzen und die letzte Version des vorhergehenden Releases zu mergen.

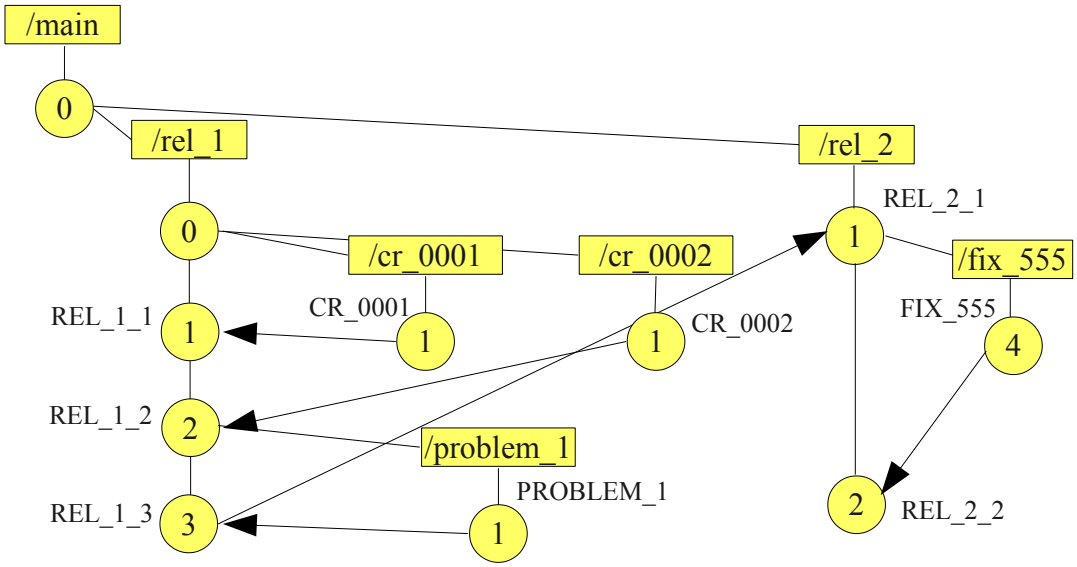

Bild 12: Beispiel für den Entwicklungsverlauf von Dateien in ClearCase

Wie im obigen Beispiel zu sehen ist, erfolgt die Rückführung der Entwicklungen linear zu den Release-Ständen.

Das muss aber nicht immer so sein.

In großen Projekten mit massiv paralleler Entwicklung kann es durchaus sein, dass Release-Stände schneller fortschreiten, als die Entwicklung.

In solchen Fällen ist es vor Freigabe der Entwicklung nötig, die Entwicklungsbranches mit dem letzten Release-Stand zu mergen -- sprich es muss ein Rebase durchgeführt werden.

An dieser Stelle noch einige Besonderheiten:

ClearCase ist bis zur Version 2003.06.00 in bestimmten Fällen nicht in der Lage leere Verzeichnisse zu mergen. Ob das ein Bug oder ein gewolltes Verhalten ist, konnte bisher noch nicht geklärt werden. Es sollte darauf geachtet werden, zumindest ein dummy-File in solch gewollt leeren Verzeichnissen anzulegen.

Eine weitere Besonderheit sind Files, welche ein Produktionssystem beeinflussen ohne aber zu Datenverlusten zu führen. Ein Beispiel dafür sind SQL-Scripte, welche auf einer lebenden Datenbank ausgeführt werden. Diese Scripte dürfen in der Regel nur einmal laufen -- sprich es

handelt sich in der Regel um Updates. Diese unbesehen in einen Produktionsstand zu mergen kann zu katastrophalen Ergebnissen führen.

In der Datenbankentwicklung gibt es deshalb in der Regel eine zweigleisige Entwicklung:

- Quellen, die es ermöglichen, die Datenbank von Grund auf neu aufzubauen (Full-Deployment).
- und solche, die nur Updates enthalten (Update-Deployment).

Bsp: neues Feld in einer Tabelle:

Full-Deployment:

```
create table ...
```

Update-Deployment:

```
alter table ...
```

Weitere Informationen dazu siehe Full- und Update-Deployments am Beispiel von Datenbanken.

9 Rebase der Entwicklung auf aktuellen Produktionsstand

Wie bereits oben angedeutet, können Release-Stände schneller voranschreiten, als ein Entwicklungsauftrag umgesetzt werden kann.

Kommt dieses vor, ist vor Abgabe des Entwicklungsergebnisses unbedingt darauf zu achten, die Änderungen des fortgeschrittenen Release-Standes in die Entwicklung zurückzuführen.

Damit können im Vorfeld viele spätere Mergekonflikte bei der Rückführung der Entwicklungsaufgabe ausgeschlossen werden.

Dieses Synchronisieren (Rebase) der Entwicklung mit dem Releasestand sollte bei jedem neuen Releasestand durchgeführt werden. Man muss es nicht tun, allerdings können bei einem finalen Rebase dann doch erhebliche Änderungen auftreten.

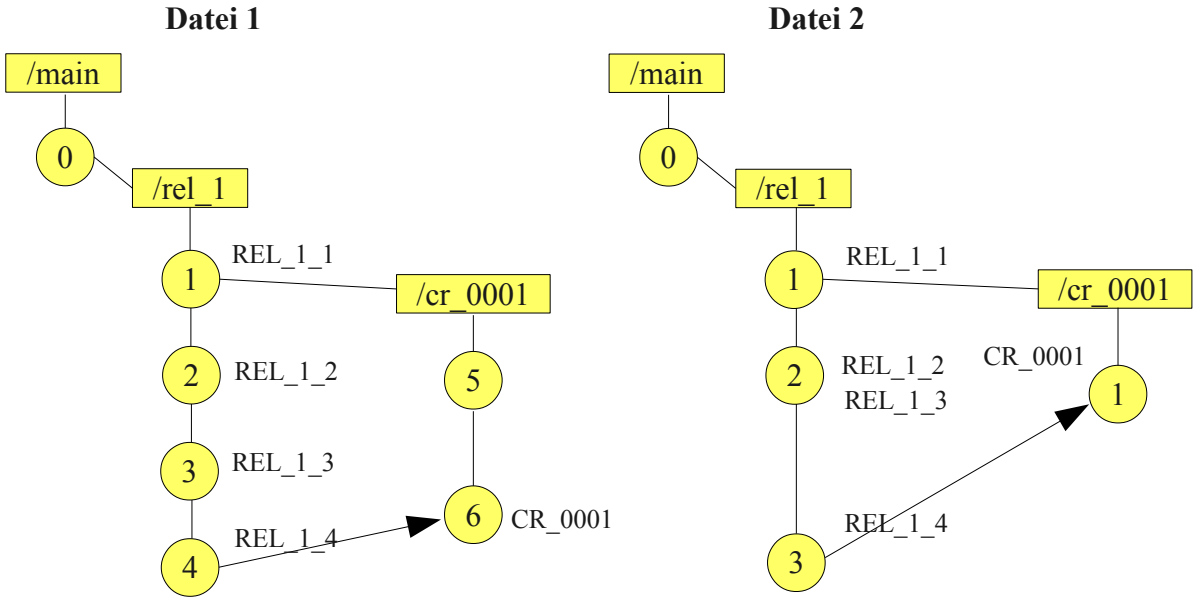

Bild 13: Beispiel für den Rebase von Dateien in ClearCase

Wie im obigen Beispiel zu sehen ist, beginnt die Entwicklung des CR0001 auf dem Release-Stand 'REL_1_1'.

Die Entwicklung wäre mit dem Element 5 des Branches 'cr_0001' eigentlich abgeschlossen.

Da aber der Release-Stand zum Zeitpunkt des Abschlusses bereits zwei Stände weiter ist, müssen die eingeflossenen Änderungen in die Entwicklung übernommen werden.

Dadurch entsteht das Element 6, welches dann auch letztendlich geliefert wird.

Leider hat diese Vorgehensweise einen Haken.

Die Datei 2 im obigen Bespiel wurde zwischenzeitlich auch weiterentwickelt.

Für einen konsistenten Stand wird das Versionsverwaltungssystem diese ebenfalls zurückmergen.
D.h. es wird für diese Datei ein 'cr_0001' Branch angelegt, obwohl sie nicht zwangsläufig Bestandteil der Entwicklung war.

Damit werden dann nach Abschluss der Entwicklung mehr Dateien geliefert, als ursprünglich bearbeitet wurden.

Ein verbessertes 'Rebase' wird im Beispiel unten gezeigt.

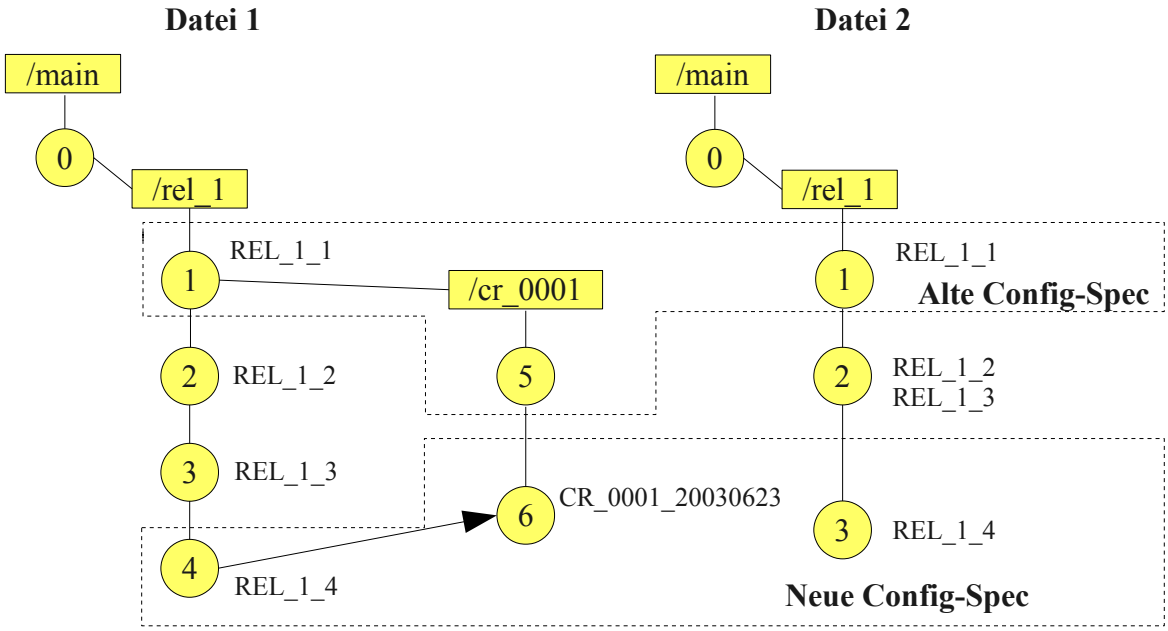

Bild 14: Beispiel für einen verbesserten Rebase von Dateien in ClearCase

Hier wird beim 'Rebase' die ursprüngliche Config-Spec für 'cr_0001' neu erzeugt. Es wird eine neue Sicht auf den letzten Release-Stand erzeugt und vom letzten Element ein Merge nach 'cr_0001' durchgeführt.

Dieses Vorgehen ist zwar wesentlich arbeitsintensiver, hat aber den Vorteil, wirklich nur die Dateien zu liefern, welche auch bearbeitet wurden. Weiterhin wird durch einen rechtzeitigen Rebase gewährleistet, das immer nahe am letzten Release-Stand entwickelt wird.

Der Merge in einen neuen Release-Stand (siehe Merges der Entwicklerbranches in den Produktionsstand) ist dann trivial.

Leider hat auch dieses Verfahren einen entscheidenden Nachteil.

Durch den Rebase erhält der Entwicklungsauftrag eine neue 'Baseline'.

Ursprünglich setzte der Branchtype im obigen Beispiel auf 'REL_1_1' auf. Nach dem Rebase wird auf 'REL_1_4' aufgesetzt.

D.h. es entsteht eine völlig neue Config-Spec.

Für eine Entwicklungsaufgabe, in der mehrere Entwickler involviert sind, bedeutet dieses: Alle beteiligten Entwickler müssen über die Änderung der Config-Spec benachrichtigt werden und diese müssen ihre Views / Arbeitsumgebungen entsprechend umsetzen.

Es sollten folgende Punkte beachtet werden:

- Der Rebase darf nur ausgeführt werden, wenn alle Files des Entwicklungsauftrages in einem gesicherten Stand (eingecheckt) sind.
- Die neue Config-Spec muss in einem für alle an der Entwicklungsarbeit beteiligten Entwickler sichtbaren Verzeichnis gespeichert werden.
- Während des Rebase sollten alle Files für die weitere Bearbeitung gesperrt werden.

- Alle beteiligten Entwickler müssen über die Änderung der Config-Spec informiert werden, damit diese ihre Arbeitsumgebungen entsprechend neu einstellen können.

10 Weiterentwicklungen von Ständen, welche noch nicht in Produktion gegangen sind

In der Entwicklung ist es durchaus nötig, auf Stände aufzusetzen, welche selbst noch in der Entwicklung sind. Dieses ist zwar von der Releaseplanung her zu unterbinden, aber manchmal nicht zu vermeiden.

Dazu bieten sich grundsätzlich zwei unterschiedliche Ansätze an.

1. Die Config-Spec setzt direkt auf die noch auf der in der Entwicklung befindlichen Aufgabe auf.

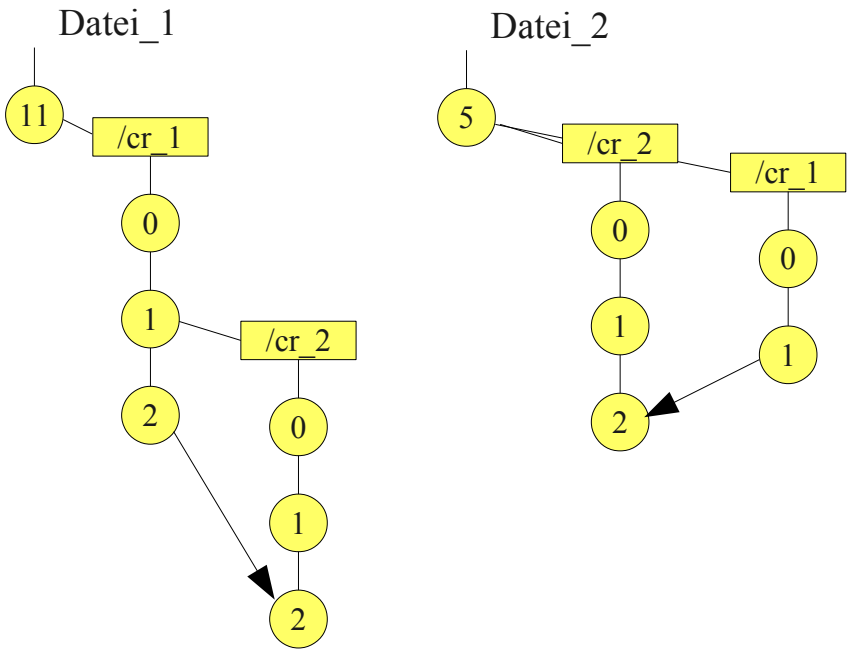

Bild 15: Beispiel für einer Entwicklung basierend auf einer weiteren Entwicklungen

Im obigen Beispiel wird die Aufgabe 'cr_2' direkt auf 'cr_1' aufgesetzt. Datei 1 ist zu diesem Zeitpunkt bereits in der Bearbeitung und Liegt im Element 1 vor. Eine Bearbeitung des 'cr_2' wird nun zu einem Abbranchen vom Element 1 führen. Änderungen im 'cr_1' müssen nachgeführt werden.

Ein anderes Bild wird sich ergeben, wenn 'cr_2' die Datei 2 vor dem 'cr_1' bearbeitet. Auch in diesem Fall sind die Änderungen der führenden Aufgabe zu übernehmen.

2. Es wird eine neue Aufgabe definiert und alle Änderungen der Aufgabe, auf die aufgesetzt wird, werden möglichst zeitnah in die neue Aufgabe integriert.

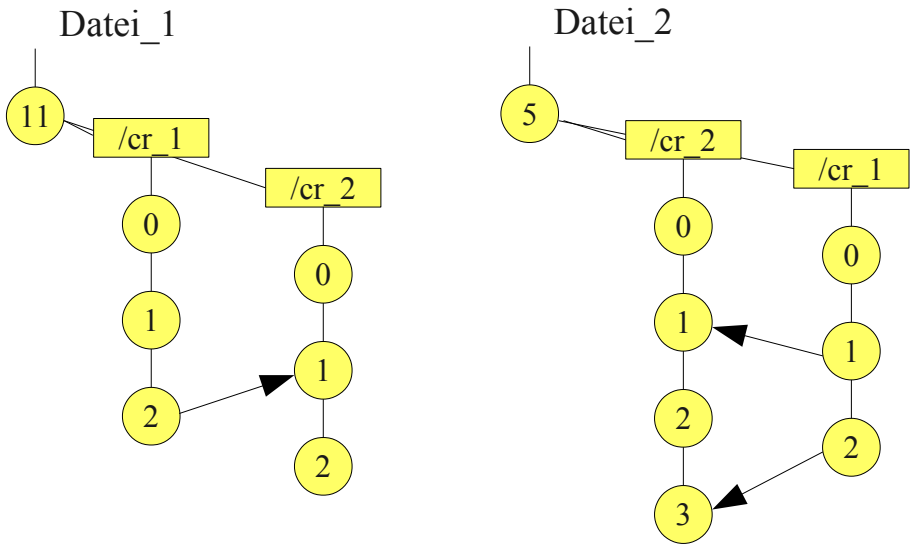

Bild 16: Beispiel für einer Entwicklung basierend auf einer weiteren Entwicklungen aber abgekoppelt von dieser

Bei diesem Beispiel ist es ebenfalls notwendig alle Änderungen des 'cr_1' in den 'cr_2' einzubinden.

Allerdings ist es hier nachträglich noch möglich, den 'cr_1' wieder aus der Entwicklung herauszunehmen (subtraktiver Merge).

Welche Methode jeweils bevorzugt wird, sollte jeder für sich selbst entscheiden.

11 Full- und Update-Deployments am Beispiel von Datenbanken

Wie weiter oben bereits angedeutet, sind speziell Datenbankentwicklungen ein kritisches Thema für eine Versionsverwaltung. Einerseits soll es möglich sein, eine Datenbank von Grund auf neu anzulegen und andererseits ist es fatal, in bestehenden Datenbanken Scripte auszuführen, welche zu Datenverlust führen.

Aus diesen Gründen sollte eine Entwicklung für eine Datenbank immer zweigleisig durchgeführt werden.

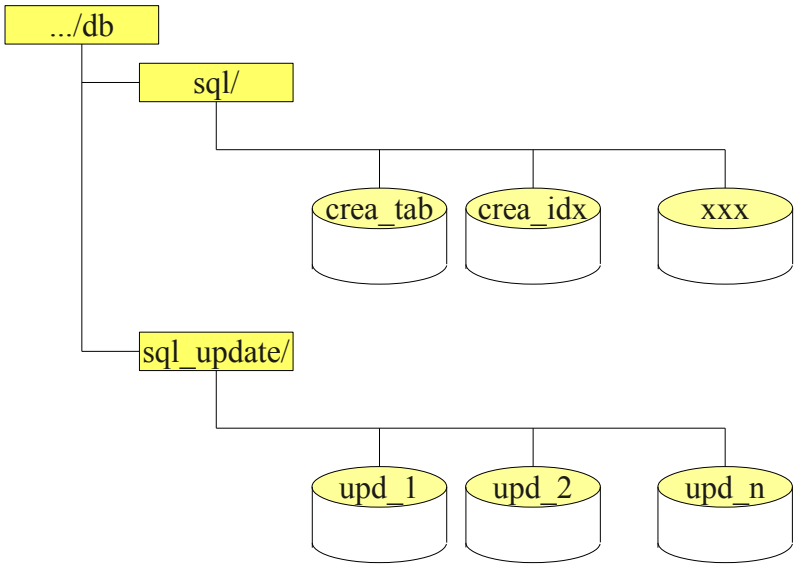

Bild 17: Beispiel für einen Verzeichnisbaum in einem Datenbank-Projekt

Oben ist ein mögliches Beispiel für einen Produktionsstand abgebildet. Im Verzeichnis 'sql' sind alle Files für ein Full-Deployment enthalten. Diese werden auf einer produktiven Datenbank natürlich nicht ausgeführt. Statt dessen werden ausschließlich die Files im Verzeichnis 'sql_update' aufgerufen.

Für eine Weiterentwicklung der Datenbank dürfen jetzt die Files im Verzeichnis 'sql_update' nicht mehr berücksichtigt werden. Für die Entwicklung muss demzufolge ein leeres Verzeichnis zur Verfügung gestellt werden.

Nachfolgend wird versucht, die Entwicklungsschritte anhand eines Datenbank-Files nachzuvollziehen.

Ausgehend von einem Produktionsstand 'prod_20040112' wird für eine Entwicklung 'cr_20040126' abgebrancht.

Zwischenzeitlich wird ein neuer Produktionsstand 'prod_20040210' eingeführt. Für die Entwicklung ist jetzt ein Rebase nötig. Es dürfen aber nur die Files im Verzeichnis 'sql' auf den Produktionsstand gehoben werden.

Dann wird aus dem 'cr_20040126' der neue Produktionsstand 'prod_20040228' gebildet.

Stand	*.../sql*	*.../sql_update*
Prod 2 (Ausgangsstand)	File: crea_tab1.sql Inhalt: **....** **-- entfaellt** **-- cust_descr varchar(128),** **-- neu hinzu** **cust_descr varchar(256),** **....**	File: upd_tab1_cr1.sql Inhalt: **alter table custumer modify** **cust_descr varchar(256);**
CR 3	File: crea_tab1.sql Inhalt: Nach dem Abbranchen noch der gleiche Inhalt wie 'Prod 2' Änderung: **....** **-- entfaellt** **-- cust_descr varchar(128),** **-- neu hinzu** **cust_descr varchar(256),** **....** **-- neu hinzu** **cust_telno char(20),** **....**	Nach dem Abbranchen sind keine Files vorhanden. Neues File: upd_tab1_cr3.sql Inhalt: **alter table customer add** **cust_telno char(20);**
Prod 3 (CR 2 wird in Produktion gebracht)	File: crea_tab1.sql Inhalt: **....** **-- entfaellt** **-- cust_descr varchar(128),** **-- neu hinzu** **cust_descr varchar(256),** **-- neu hinzu** **cust_anrede char(10),** **....**	File: upd_tab1_cr2.sql Inhalt: **alter table custumer add** **cust_anrede char(10);**
REBASE	File: crea_tab1.sql Inhalt: **....** **-- entfaellt** **-- cust_descr varchar(128),** **-- neu hinzu** **cust_descr varchar(256),** **-- neu hinzu** **cust_anrede char(10),** **....** **-- neu hinzu** **cust_telno char(20),** **....**	Es darf nach wie vor nur das File 'upd_tab1_cr3.sql' existieren. Das File 'upd_tab1_cr2.sql' darf keinesfalls gemerged werden, da es sonst beim Produktionsmerge wieder im neuen Produktionsstand auftaucht!

Stand	*.../sql*	*.../sql_update*
Prod 4	File: crea_tab1.sql wird gemerged Inhalt: **....** **-- entfaellt** **-- cust_descr varchar(128),** **-- neu hinzu** **cust_descr varchar(256),** **-- neu hinzu** **cust_anrede char(10),** **....** **-- neu hinzu** **cust_telno char(20),** **....**	Verzeichnis wird geleert und anschließendwird das File upd_tab1_cr3.sql in das Verzeichnis gemerged Inhalt: **alter table customer add cust_telno char(20);**

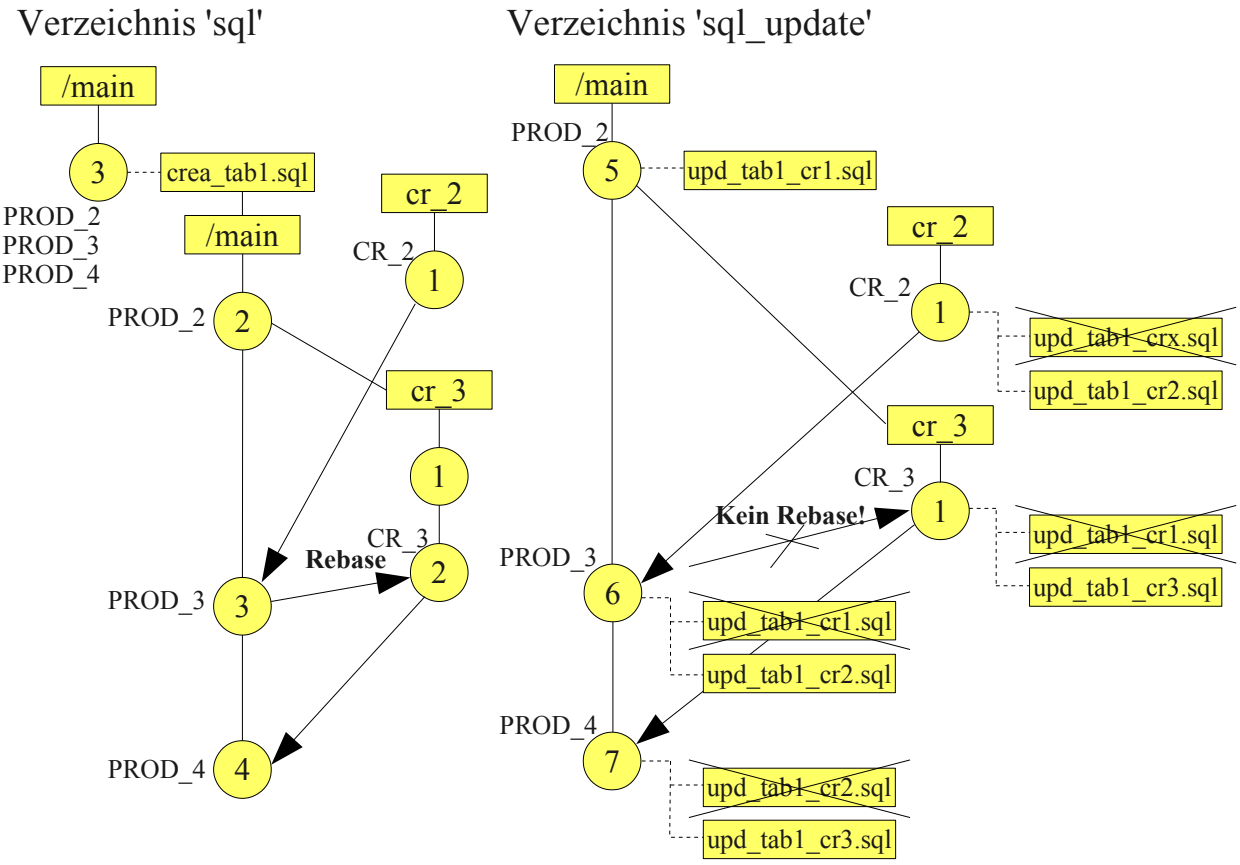

Bild 18: Obiges Entwicklungsbeispiel in zeitlicher Abfolge

12 Transparente Arbeit mit 'CVS' und 'ClearCase'

Bei der Arbeit mit Versionsverwaltungssystemen ist es immer möglich die in diesem Dokument vorgestellten Tips und Verfahren mehr oder weniger komfortabel umzusetzen. Eine solche Vorgehensweise bedingt aber auch ein gutes und fundiertes Wissen um die Möglichkeiten und die Handhabung der Systeme.

Leider zeigt die Praxis, das der Wissensstand um Versionsverwaltungen nicht dem eigentlich gefordertem KowHow entspricht.

Natürlich ist es möglich, in kleineren Projekten jemanden einzusetzen, der sich dann ausschließlich um das Management der Versionsverwaltung kümmert. Je größer aber die Projekte werden und je mehr Entwicklungsaufgaben in immer kürzerer Zeit zu erledigen sind, desto mehr steigt der Aufwand. Das kann sogar soweit gehen, dass die Versionierung einfach nicht mehr handhabbar ist.

Es sollte deshalb angestrebt werden den Teams Tools an die Hand zu geben, die einen Großteil der notwendigen Arbeiten automatisch ausführt.

Folgende Aufgaben sollten diese Tools erfüllen:

1. allgemeine Tools
    - Bereitstellen von konfigurierten Sichten auf die versionierten Dateien (Views).
      Unter konfigurierten Sichten wird hier auch das Bereitstellen der kompletten Umgebung verstanden.
      Dazu gehören z.B. der richtige Compiler, die richtige Library-Version, ...
    - Einfaches Wechseln der Sicht auf die Dateien (Config-Spec).
      Besonderer Wert sollte hierbei auf in einem View ausgecheckte Dateien gelegt werden. Ggf. ist bei nicht abgeschlossenen Arbeiten ein Wechsel der Config-Spec zu unterbinden.
    - Löschen von Sichten auf die versionierten Dateien (Views)
2. Tools für Developer-Teams
    - Bereitstellen von maßgeschneiderten 'Config-Specs' für die Entwicklungsarbeit.
      Wichtig hierbei ist auch eine Angleichung von CVS an das co/ci – Verhalten von ClearCase.
      Es kann auch unter CVS ermittelt bzw. festgehalten werden, wer welche Datei in welcher Version bearbeitet.
    - Einfacher 'rebase' der Entwicklungsarbeit auf einen neuen Produktionsstand.
    - Labeln der fertigen Entwicklung durch Vergabe eines Developer-Labels.
3. Tools für Deployment-Teams
    - Gegenlabeln der Entwickler-Label durch ein Deployment-Label (quittieren der Lieferung)
      Jede Lieferung des Development-Teams wird durch ein Label quittiert, welches dann nicht mehr verschoben wird. Auch notwendige Updates der Lieferung werden so behandelt. Damit ist von Seiten der QS-Anforderungen ein lückenloser Nachweis der Entwicklungsarbeit gegeben. Die Label des Deployment-Teams werden nach der Vergabe gelockt.
    - Integration (Zusammenfassung) mehrere Entwicklungslieferungen zu einer

Gesamtlieferung für z.B. eine Testinstallation

- Merge von getesteten und freigegebenen Entwicklungslieferungen auf einen neuen Produktionsstand.
- Labeln des neuen Produktionsstandes
- Einfache Erzeugung von Config-Specs für spezielle Sichten auf die Versionen zum Zwecke einer Testinstallation bzw. Erzeugung neuer Produktions-Config-Specs.
- Lückenlose Dokumentation und Abspeicherung dieser Config-Specs

13 Möglicher Workflow vom Entwicklungsauftrag bis zur Produktionseinführung

Nachfolgend ist ein möglicher zeitlicher Verlauf einer Entwicklung von der Auftragserstellung über Tests bis zur Produktionseinführung dargestellt.

Es macht dabei in meinen Augen Sinn eine klare Aufgabenverteilung einzuhalten:

- Releasemanagement:
  regulitative Aufgaben (Auftragserstellung, Kontrolle, Terminabstimmung)
  Freigaben
  Planung der Produktionseinführung verschiedener Entwicklungen
  Risikoabschätzungen
- Developmentteam:
  Entwicklung und Modultests
- Deploymentteam:
  Installation der Entwicklungsergebnisse auf Testumgebungen
  Management der Versionsverwaltungssysteme
  Erzeugung und Dokumentation von Config-Specs
- Testteam:
  Test der gelieferten Entwicklungsergebnisse
  Freigabeempfehlungen oder Rückstellung in die Entwicklung

Welche Gewichtung den einzelnen Teams zukommt, soll an dieser Stelle nicht behandelt werden.

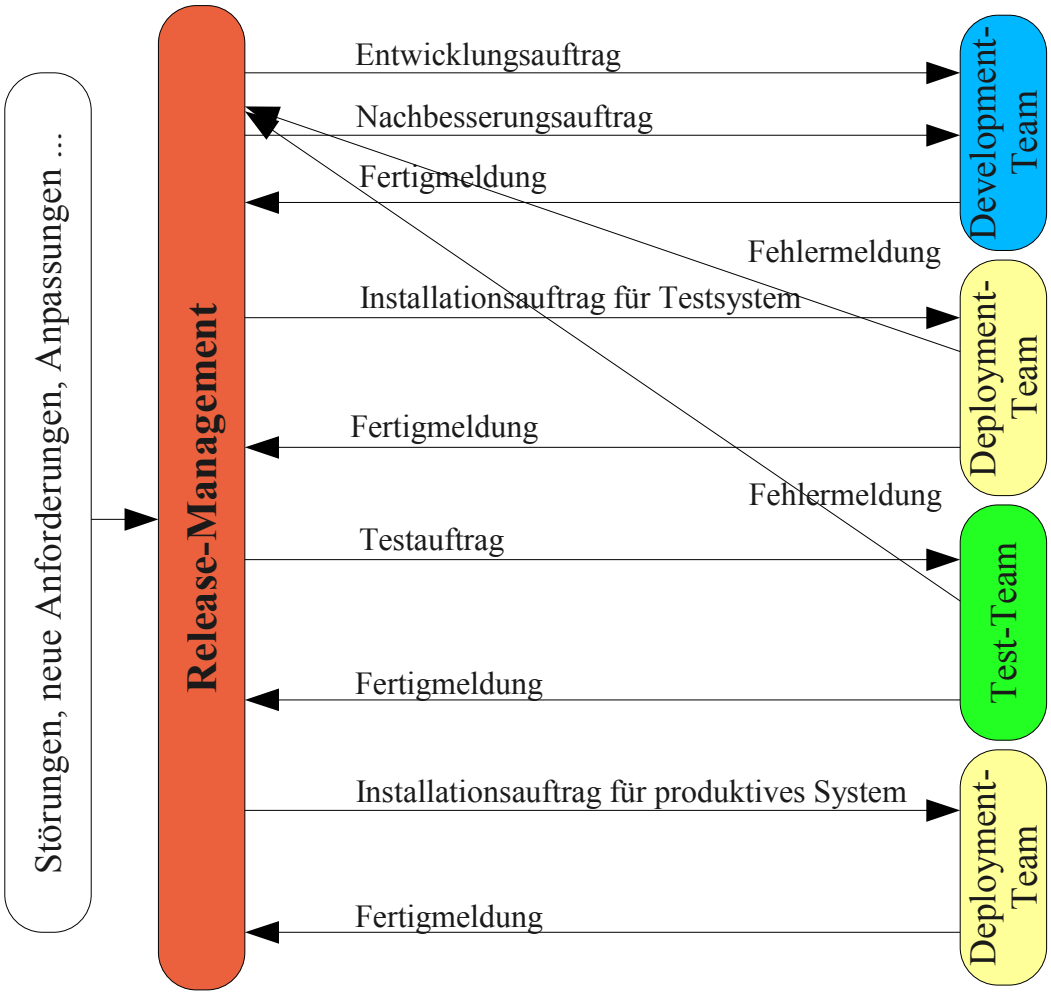

Bild 19: Möglicher Workflow einer Entwicklung