Anhang B

Quellcode-Management mit CVS

Obwohl zu CVS ein umfangreiches und gut verständliches Handbuch verfügbar ist, hat es sich dennoch als notwendig erwiesen, die im Entwickleralltag verwendeten Konzepte und Verfahren gesondert und in deutsch zu dokumentieren. Dieser Anhang enthält eine inhaltlich etwas erweiterte und redaktionell überarbeitete Zusammenstellung dieser Dokumentation. Zusätzlich werden für den Verwalter des Repositories relevante Konzepte und Eigenschaften sowie übliche Abläufe beschrieben.

Natürlich ist es in diesem Rahmen nicht möglich, CVS erschöpfend zu behandeln. Für weiterführende Fragen sei auf das CVS–Manual [9] verwiesen.

B.1 Allgemeine Konzepte und Eigenschaften

Das Concurrent Versions System CVS erlaubt es mehreren Entwicklern, am selben Sourcecode zu arbeiten, ohne sich gegenseitig zu behindern. Man kann Projektstände markieren und später auf den Sourcecode, der zum markierten Zeitpunkt aktuell war, zurückgreifen. Außerdem kann eine gemeinsame Codebasis weiterentwickelt werden, während bereits mehrere unabhängig Projektgruppen damit arbeiten. Der weiterentwickelte Code kann dann relativ problemlos nachträglich in die Arbeit der Projektgruppen integriert werden.

B.1.1 Vorbemerkungen

CVS kennt zwei Sorten von Versionen. Zum einen gibt es verschiedene Versionen einer Datei, die sich aus den Änderungen im Laufe der Entwicklung ergeben. Zum anderen gibt es Softwareversionen, die einen bestimmten Projektstand beschreiben. Um Verwirrung zu vermeiden, wird im folgenden der Begriff Version vermieden. Eine Dateiversion wird Revision genannt, eine die ganze Software betreffende Version wird als Release bezeichnet.

B.1.2 Das Repository

Das Repository enthält die zentrale Codebasis des Projekts. Jeder Entwickler holt sich von hier die neuesten Revisionen, arbeitet daran und fügt seine Änderungen anschließend in das Repository ein.

Ein Repository ist ein Verzeichnis, das mehrere Unterverzeichnisse, sog. Module, enthält. In jedem Repository findet man das zentrale Verwaltungsmodul von CVS; es heißt `CVSROOT`. Für das Projekt *AT Humboldt 98* gibt es ein weiteres Modul mit dem Namen `AT_H98`. In diesem Verzeichnis gibt es für jede Sourcecode-Datei eine History-Datei mit der Endung `,v`, in der alle Änderungen seit der Aufnahme dieser Datei in das Repository verzeichnet sind. Wenn im weiteren von Repository die Rede ist, ist das in der Regel dieses Modulverzeichnis gemeint.

Eine Person im Projekt sollte sich um das Repository kümmern. Wir nennen sie im weiteren den Repository-Manager. Niemand außer dem Repository-Manager sollte von Hand Änderungen im Repository vornehmen (was auch normalerweise nicht nötig ist).

B.1.3 Das Arbeitsverzeichnis

Jeder Entwickler hat sein persönliches Arbeitsverzeichnis, auch Arbeitskopie genannt. Im Arbeitsverzeichnis befindet sich normalerweise die aktuellste Revision der Dateien des Repositories. An diesen Dateien kann nun der Entwickler nach Belieben arbeiten.

Sobald der Entwickler einen Stand erreicht hat, an dem er seine Fortschritte den anderen Entwicklern zur Verfügung stellen will, integriert er die Änderungen in das Repository. Dieser Vorgang wird „Einchecken“ genannt.

Das Einchecken schlägt fehl, wenn eine Datei im Repository auf einem neueren Stand ist als die Revision, auf der der Entwickler seine Änderungen gemacht hat. In diesem Fall greift das Merge-Protokoll von CVS. Der Entwickler fügt zunächst die Änderungen zwischen seiner Ausgangsrevision und der aktuellen Revision aus dem Repository in seine Arbeitskopie ein und behebt anschließend evtl. auftretende Konflikte. Dabei unterstützt ihn CVS. Nun kann er von neuem versuchen, seine Änderungen einzuchecken.

B.2 Informationen für den Entwickler

Damit ein Entwickler CVS im Alltag einsetzen kann, muß er die nachfolgenden Aufgaben erledigen können. Dazu genügt die Kenntnis einer Handvoll von CVS-Kommandos. CVS wird über die Kommandozeile aufgerufen. Im Prinzip folgt ein Aufruf diesem Schema:

```
cvs <CVS-Opt.> <CVS-Befehl> <Befehlsopt.> <Befehlsarg.>
```

Alle Parameter bis auf den CVS-Befehl können (und werden in der Praxis oft) entfallen; sie haben folgende Bedeutung:

`CVS-Opt.`	Optionen, die die Ausführung aller CVS-Befehle beeinflussen. Z.B. kann man mit der Option `-d` das Repository angeben.
`CVS-Befehl`	Der eigentliche Befehl, der angibt, welche CVS-Funktion ausgeführt werden soll. Beispiele sind `update` oder `tag`.
`Befehlsopt.`	Optionale Angaben, die die Ausführung des CVS-Befehls beeinflussen. Z.B. kann man mit der Option `-r` des CVS-Befehls `diff` die Repository-Revision angeben, auf er sich beziehen soll.
`Befehlsarg.`	Die Argumente für das Kommando. Z.B. bei `commit` die Dateien, die eingecheckt werden sollen.

B.2.1 Anlegen eines Arbeitsverzeichnisses

Zum Anlegen eines eigenen Arbeitsverzeichnisses muß der Entwickler zuerst die Environment-Variable `CVSROOT` so setzen, daß sie das Repository-Verzeichnis enthält. Obwohl CVS später innerhalb eines existierenden Arbeitsverzeichnisses das zugehörige Repository auch ohne diese Variable findet, ist es ratsam, sie bereits in einer der Shell-Initialisierungsdateien wie `.tcshrc` oder `.bashrc` zu setzen[1]. Ist CVSROOT gesetzt, kann man ein neues Arbeitsverzeichnis anlegen mit

```
> cvs checkout [-d <Verzeichnisname>] <Modulname>
```

Im aktuellen Verzeichnis legt CVS dann ein Verzeichnis *Verzeichnisname* als Arbeitskopie des Repository-Moduls *Modulname* an. Wird die `-d` Option nicht verwendet, so heißt das angelegte Verzeichnis genauso wie das Modul.

In der Arbeitskopie kann der Entwickler nun seine Änderungen vornehmen.

B.2.2 Einchecken von Änderungen

Möchte der Entwickler seine Änderungen in das Repository aufnehmen, so verwendet er dazu den Befehl

```
> cvs commit [ <Datei1> [ <Datei2> <...> ] ]
```

Sind keine Dateien als Parameter angegeben, überprüft CVS das ganze Verzeichnis und alle Unterverzeichnisse auf Änderungen. Wurden geänderte Dateien entdeckt, fragt CVS nach einer Beschreibung der Änderungen. Dazu wird der in der Environment-Variable EDITOR angegebene Editor gestartet. Anschließend erzeugt CVS im Repository eine neue Revision jeder geänderten Datei.

Im Laufe der Entwicklung hat sich folgende Praxis als empfehlenswert herausgestellt.

[1]Außer dem hier beschriebenen `checkout` benötigen einige weitere CVS-Befehle (z.B. `rtag` oder `import`) die Angabe eines Repositories.

- Vor jedem Einchecken sollte das Projekt nochmal testweise kompiliert werden. Es ist überaus ärgerlich, wenn man die neueste Version aus dem Repository holt, und das Programm dann nicht einmal mehr kompiliert. Je nach Fortschritt des Projekts kann man an den einzucheckenden Code verschiedene Qualitätsmaßstäbe anlegen, aber jedenfalls sollte eine Mindestqualität vereinbart und eingehalten werden, die die anderen Entwickler nicht behindert.

- Die beim Einchecken von CVS erfragten Log-Messages sind später überaus hilfreich. Sie sollten also eine Beschreibung enthalten, mit deren Hilfe zumindest der Autor den anderen erklären kann, was er genau gemacht hat.

- Da die Log-Message an das Log jeder eingecheckten Datei angehängt wird, sollte man immer Dateien zusammen einchecken, deren Änderungen auch inhaltlich zusammengehören und mit derselben Log-Message beschrieben werden können.

B.2.3 Das Arbeitsverzeichnis aktualisieren

Nachdem ein anderer Entwickler seine Änderungen in das Repository eingecheckt hat, muß man das eigene Arbeitsverzeichnis aktualisieren, bevor man eigene Änderungen einchecken kann.

Da es möglich ist, daß der andere Entwickler Änderungen in derselben Datei, bei mangelnder Absprache evtl. sogar in derselben Zeile wie man selber vorgenommen hat, können beim Aktualisieren Konflikte auftreten. Diese Konflikte entstehen ausschließlich in der eigenen Arbeitskopie und nicht im Repository. Erst wenn sie zumindest auf syntaktischer Ebene behoben sind, ist es möglich, die eigenen Änderungen einzuchecken.

Folgender Befehl aktualisiert das Arbeitsverzeichnis:

```
> cvs update [-d]
```

Dabei geht CVS alle Dateien durch und meldet Unterschiede zwischen Arbeitskopie und Repository durch Ausgabe einer Zeile pro Datei, die den Dateinamen und davor einen Buchstaben enthält. Die Kenntnis der Bedeutung dieser Buchstaben hilft, das Ergebnis des Updates zu bewerten.

U (Updated) Die Arbeitskopie wurde aktualisiert. U erscheint bei Dateien, die lokal nicht geändert wurden, die aber nicht mehr der aktuellen Revision im Repository entsprechen, sowie bei Dateien, die neu im Repository sind und noch nicht im Arbeitsverzeichnis enthalten sind.

M (Modified) Die lokale Kopie enthält Änderungen gegenüber dem Repository. Ein `commit` wird also eine neue Revision der Datei im Repository erzeugen. Zusätzlich kann es sein, daß sich auch das Repository seit dem letzten Update geändert hat, daß diese Änderungen aber erfolgreich in die lokale Kopie integriert werden konnte. Dieser Vorgang wird „merge“

genannt. Findet ein merge statt, legt CVS eine Backup-Kopie der ursprünglichen Datei an; sie hat den Namen `.#Datei.Revision`, der Name wird beim merge von CVS ausgegeben.

C (Conflict) Sowohl die lokale Kopie als auch das Repository haben sich geändert, und die Änderungen überschneiden sich zeilenweise. In diesem Fall wird ebenfalls eine Backup-Kopie angelegt. In der Datei sind die Überlappungen durch die Zeilen `<<<<<<<<` davor, `--------` dazwischen und `>>>>>>>>` danach markiert. Diese Markierungen müssen entfernt werden, bevor ein `commit` durchgeführt werden kann.

? (Unknown) Die Datei befindet sich im Arbeitsverzeichnis, ist aber nicht unter CVS-Verwaltung. Die Anzeige solcher Dateien kann durch einen entsprechenden Eintrag in der Datei `.cvsignore` unterdrückt werden (siehe [9, Seite 106]).

A (Added) Dies ist lediglich eine Erinnerung, daß die Datei zwar bereits zur Aufnahme ins Repository markiert wurde, bisher aber noch nicht eingecheckt worden ist.

R (Removed) Diese Datei wurde zum Löschen markiert. Ins Repository wird die Löschung aber erst beim nächsten `commit` eingetragen. Auch dies ist nur eine Erinnerung.

Die `-d` Option von `update` führt dazu, daß im Repository vorhandene Verzeichnisse, die im Arbeitsverzeichnis fehlen, neu angelegt werden. Das ist normalerweise nicht der Fall.

Noch einige Bemerkungen zu `update` und Konflikten: CVS kann selbstverständlich nur Konflikte auf syntaktischer Ebene entdecken und melden. Wenn ein Mitarbeiter die Bedeutung eines Parameters ändert, und man selbst die Funktion noch mit den ursprünglichen Parameterbedeutungen aufruft, tritt ein logischer Konflikt auf, den CVS nicht erkennen kann.

Daß syntaktische Konflikte auftreten können, ist eine Folge des „copy-modify-merge“ Schemas von CVS. Dieses Schema ist aber in der Praxis viel besser zu handhaben als der striktere Lock-Mechanismus von RCS, so daß es sich lohnt, mit der Möglichkeit von syntaktischen Konflikten zu leben.

Im Normalfall sollten syntaktische Konflikte ohnehin nicht auftreten. Sie sind praktisch immer ein Anzeichen von mangelnder Koordination zwischen den Entwicklern: zwei Personen fühlen sich gleichzeitig für dasselbe Stück Code verantwortlich. Tritt tatsächlich ein Konflikt auf, so muß nicht nur der Konflikt behoben werden. Vor allem muß das Kommunikationsproblem gelöst werden; Verantwortungsbereiche müssen neu aufgeteilt oder klargestellt werden. Möglicherweise ist der aufgetretene Konflikt nur ein Symptom für ein tiefergehendes Mißverständnis. Daher sollte der Konflikt als Warnung ernstgenommen und die projektplanerische Ursache abgestellt werden.

B.2.4 Dateien neu aufnehmen

Gelegentlich wird ein Entwickler z.B. eine neue Klasse anlegen und dafür auch neue Dateien in die Codebasis aufnehmen wollen. Das ist mit CVS einfach möglich:

```
> cvs add <Datei> [ <Datei2> ... ]
```

Dabei sollte man sich über den Namen der Datei im klaren sein. Nachträgliche Umbenennungen sind etwas umständlich, aber möglich. Dazu benennt man am einfachsten die lokale Kopie um, nimmt die Datei unter dem neuen Namen ins Repository auf und entfernt die Datei mit dem alten Namen aus dem Repository. Dabei werden allerdings die Revisionsnummer und das Log der Datei nicht übernommen. Für Möglichkeiten, das zu verhindern, siehe [9, Seite 55ff].

Ein genereller Hinweis: Es sollten nur die wirklichen Quelldateien ins Repository aufgenommen werden; durch Compiler oder andere Werkzeuge automatisch erzeugte Dateien wie Object-Dateien oder das ausführbare Binary gehören nicht unter CVS-Verwaltung.

B.2.5 Dateien entfernen

Bei einer Konzeptänderung oder auch bei Umbenennungen von Dateien ist es gelegentlich nötig, Dateien wieder aus dem Projekt zu entfernen. Das ist nicht ganz so einfach wie das Hinzufügen, da es ja nach wie vor möglich sein muß, auf alte Releases zurückzugreifen, in denen die Datei noch vorhanden war und gebraucht wurde.

Aus Entwicklersicht funktioniert das Entfernen fast so einfach wie das Hinzufügen:

```
> cvs remove <Datei> [ <Datei2> ... ]
```

Dabei wird allerdings die entsprechende History-Datei im Modulverzeichnis des Repositories nicht gelöscht, sondern nur in einen speziellen Bereich, den sogenannten „Attic“, verschoben. Will man eine Datei tatsächlich endgültig aus dem Repository entfernen, z.B. weil man versehentlich eine Binärdatei eingecheckt hat, so muß man die History-Datei im Repository von Hand löschen. Das sollte man sinnvollerweise den Repository-Manager machen lassen oder es zumindest mit ihm absprechen.

B.2.6 Änderungen anzeigen lassen

CVS unterstützt das Anzeigen von Unterschieden zwischen Arbeitskopie und Repository oder zwischen zwei Revisionen im Repository auf zwei Arten. Man kann sich die Log-Messages anzeigen lassen, und man kann sich die textuellen Differenzen der Dateien auflisten lassen. Ersteres geschieht mit dem Befehl

```
> cvs log [<Datei> [ <Datei2> ... ] ]
```

Eine Liste aller Log-Messages der betreffenden Dateien wird angezeigt. Sind keine Dateien angegeben, werden die Logs aller Dateien angezeigt, auch die der gelöschten und inzwischen im Attic gelandeten.

Die -r Option erlaubt das Einschränken auf bestimmte Revisionen. *Revision* kann hierbei verschiedene Formen annehmen, die im CVS Manual [9, Seite 88] detailliert beschrieben sind. Vor allem kann es entweder eine Revisionsnummer oder ein Name für einen Release sein. In der Praxis haben wir meist folgende Form verwendet, um uns eine Übersicht über alle Änderungen am Projekt seit dem letzten Release zu verschaffen:

```
> cvs log -rv0_08: *.h *.cc
```

In diesem Beispiel werden die Änderungen an allen Headerdateien und Implementationsdateien seit dem Release angezeigt, das wir mit dem symbolischen Namen (auch „tag“ genannt) `v0_08` markiert haben.

Die zweite Art, Änderungen anzeigen zu lassen, ist:

```
> cvs diff [ -r<Revision1> [ -r<Revision2> ] ] <Datei>
```

Ohne Optionen zeigt dieser Befehl die Unterschiede zwischen der Arbeitskopie und der Revision im Repository, auf der die Kopie beruht, an. Gibt man *Revision1* an, wird statt dessen diese Revision mit der Arbeitskopie verglichen. Bei zwei Revisionsangaben werden die Änderungen zwischen diesen angezeigt.

B.3 Informationen für den Repository-Manager

Der Repository-Manager hat im normalen Entwicklungsprozeß praktisch keinen zusätzlichen Aufwand. Er sollte aber dennoch etwas genauer über CVS Bescheid wissen, da er die Verantwortung für das Repository hat.

In der Praxis hat sich gezeigt, daß der Repository-Manager die folgenden Aufgaben erledigen können sollte. Zudem ist es ratsam, wenn er sich das CVS–Manual einmal komplett durchliest, um einen Eindruck davon zu haben, was CVS für Möglichkeiten bietet, und wo er eingreifen kann, um die gewünschten Effekte zu erreichen.

B.3.1 Ein Repository anlegen

Zuallererst muß man ein Repository anlegen. Da es nicht ganz einfach ist, ein in Benutzung befindliches Repository zu verschieben, sollte man sich vorher genau überlegen, wo man es anlegt. Der Platzbedarf stellt wohl meistens kein Problem dar: nach einem halben Jahr Entwicklung war das Modul `AT_H98` im Repository knappe 3 MByte groß.

Zum Anlegen setzt man nur die Environment-Variable CVSROOT auf den kompletten Pfadnamen des zu erzeugenden Repositories und initialisiert das Repository:

```
> setenv CVSROOT /vol/john-vol17/projects/soccer/common/cvsroot
> cvs init
```


B.3.2 Ein Modul in einem Repository anlegen

Normalerweise wird ein Projekt zur Erstellung einer Menge von zusammengehörigen Quelldateien führen, aus denen im wesentlichen ein ausführbares Programm erzeugt wird. Diese Menge von Dateien nennt man im CVS-Jargon „Modul“. Es wird in einem Verzeichnis(baum) im Repository gehalten.

Zur Erstellung eines neuen Projekts legt man zuerst eine Verzeichnisstruktur an, die dem Projekt entspricht:

```
> mkdir Soccer
> mkdir Soccer/doc
> mkdir Soccer/doc/html
> mkdir Soccer/doc/latex
```

Nun importiert man den Verzeichnisbaum ins Repository:

```
> cd Soccer
> cvs import <Modulname> <Vendorbranch-Tag> <Release-Tag>
```

Hierbei haben die drei (notwendigen) Parameter des `import`-Kommandos folgende Bedeutung:

Modulname Z.B. `AT_H99`. So wird das Verzeichnis im Repository heißen, unter dem die Verzeichnisse `doc`, `doc/html` und `doc/latex` angelegt werden. Erzeugt ein Entwickler mit `cvs checkout` ein neues Arbeitsverzeichnis, heißt es standardmäßig so wie das Modul.

Vendorbranch-Tag Dieser Parameter hat ebenso wie der folgende eigentlich nur beim Verfolgen von Drittquellen-Änderungen eine Bedeutung. Daher tragen wir einfach einen beliebigen Wert ein, z.B. `dev`.

Release-Tag Wir tragen einfach einen beliebigen Wert ein, z.B. `start`.

Noch ein wichtiger Hinweis: `import` gehört zu den CVS-Befehlen, die auf keinen Fall mit Ctrl-C abgebrochen werden dürfen. CVS besitzt momentan noch kein echtes Transaktions-Management. Das Repository kann bei einem abgebrochenen import in einen inkonsistenten Zustand versetzt werden. Dieser läßt sich zwar mit CVS-Befehlen wieder reparieren, was aber einige CVS-Kenntnisse voraussetzt und mitunter langwierig sein kann.

Existieren bereits Quelldateien für das unter CVS-Verwaltung zu stellende Projekt, dann kann man einfach diese Quellen importieren. Vorher sollte man sich das Verzeichnis noch genau anschauen und alle automatisch erzeugten Dateien (.o-Dateien, von Editoren angelegte Sicherheitskopien, Binaries, ...) entfernen, da nur die echten Quelldateien importiert werden sollen.

Ein weiteres, beim Einrichten eines Projektes relevantes Thema ist das der Zugriffsrechte auf das Repository. Am einfachsten richtet man dazu eine UNIX-Gruppe ein, in der alle Projektmitarbeiter (und zur Sicherheit der Repository-Manager) enthalten sind. Dieser Gruppe schenkt man nun das Modul im Repository und stellt durch das Setzen des s-Bits bei den Gruppenrechten sicher, daß alle im Repository neu angelegten Dateien auch dieser Gruppe gehören werden:

```
> cd $CVSROOT
> chgrp -R soccteam AT_H99
> chmod -R g+s AT_H99
```

Gibt es mehrere Projekte im Repository, die von verschiedenen Gruppen betrieben werden, so benötigt man eine zentrale Gruppe, in der alle Projektmitglieder aller Gruppen sind. Dieser Gruppe sollte das zentrale Verwaltungsmodul (CVSROOT) gehören.

B.3.3 Tags und Exporte

Um einen Projektstand zu markieren, gibt es die Möglichkeit, die entsprechenden Revisionen aller Dateien mit einem Namen zu versehen. Diesen Namen nennen wir ein „Tag". Besonders, wenn sich in einem Projekt Weiterentwicklungs- und Stabilisierungsphasen abwechseln, bietet es sich an, am Ende einer Stabilisierungsphase ein Tag zu setzen, um später auf den erreichten Stand zurückgreifen zu können. Auch, wenn man ein Release für die Öffentlichkeit (und sei es auch nur eine Gruppe von Alpha-Testern) erzeugt, setzt man vorher ein Tag, da es wichtig ist, beim Beheben von Fehlern zu wissen, auf welchen Software-Stand sich der Bug-Report bezieht.

Das Setzen eines Tags ist nicht schwierig. Am besten bringt man seine Arbeitskopie auf den aktuellsten Stand und markiert dann im Repository die entsprechenden Revisionen:

```
> cd develop/AT_H99
> cvs update
> cvs tag v0_09 .
```

In diesem Beispiel werden alle Dateien im aktuellen Verzeichnis mit dem Tag `v0_09` versehen.

Auf einen derart markierten Projektstand greift man am besten mit dem `export`-Befehl zurück. Ebenso wie bei `cvs checkout` wird damit eine Kopie aus dem Repository gezogen. Allerdings steht diese Kopie nicht mehr unter CVS-Verwaltung. Es werden keine CVS-Verwaltungsverzeichnisse angelegt, und auch `update` und `commit` arbeiten nicht. Das wäre auch nicht sinnvoll, da ja wahrscheinlich bereits Nachfolgerevisionen zu einzelnen Dateien im Repository existieren.

Der nachfolgende Befehl erzeugt unterhalb des Verzeichnisses `develop` im Verzeichnis `currentAlpha` eine Kopie des Projektstandes `v0_09` des Projektes `AT_H99`.

```
> cd develop
> cvs export -d currentAlpha -r v0_09 AT_H99
```

CVS ermöglicht es dennoch, einen Seitenzweig in der Entwicklung zu eröffnen, z.B. um bei einem stabilisierten Release lediglich weitere Bugs zu entfernen, während auf dem Hauptzweig die Entwicklung weitergeht. Wir haben dieses Feature während der Entwicklung von *AT Humboldt 98* nicht eingesetzt. Weiteres dazu siehe [9, Seite 37ff].

B.3.4 Das Verfolgen von Drittquellen-Änderungen

Dieses insgesamt wohl eher selten eingesetzte Feature von CVS hat uns wertvolle Dienste bei der Synchronisation der Kern-Entwicklergruppe mit mehreren auf diesem Kern aufsetzenden „Feature-Teams" geleistet.

Eigentlich ist das Feature dazu gedacht, eigene Patches eines im Sourcecode verfügbaren Programmes in das nächste Release des Programmes hinüberzuretten.

Bei der Entwicklung von *AT Humboldt 98* wurde es eingesetzt, um den Basiscode, in den die einzelnen Praktikumsgruppen ihren Projektcode einbauen, zu aktualisieren. Dazu hat das Kern-Entwicklerteam wöchentlich ein Release herausgebracht, das dann von den Praktikumsgruppen als „Quellcode von Dritten" importiert wurde.

Dabei laufen im einzelnen folgende Schritte ab:

1. Das Kern-Entwicklerteam protokolliert alle wichtigen Änderungen in einer Changes-Datei, setzt ein Tag, exportiert ein Release und packt es in eine .tar.gz-Datei.

2. Der Repository-Manager des Feature-Teams besorgt sich diese .tar.gz-Datei und packt sie in einem import-Verzeichnis aus.

3. Um später noch auf den Stand vor dem Export zugreifen zu können, setzt er ein Tag auf den aktuellen Stand. Dieser Schritt ist nicht notwendig, aber empfehlenswert.

4. Er importiert das Alpha-Release in sein Repository. Dabei verwendet er für alle Releases dasselbe Vendorbranch-Tag und für jedes Release ein anderes Release-Tag. Der Importbefehl, der den Code aus dem aktuellen Verzeichnis als Release `v0_09` von `CODEBASE` in das Modul Soccer aufnimmt, sieht dann so aus:

   ```
   > cd import/v0_09
   > cvs import Soccer CODEBASE v0_09
   ```

   Achtung: Dieser Befehl darf nicht mit Ctrl-C abgebrochen werden[2].

5. Nun müssen in einem Arbeitsverzeichnis die Änderungen der Gruppe mit dem eben importierten Release vereinigt werden. Dazu führt der Repository-Manager in seinem Arbeitsverzeichnis ein spezielles Update durch, das die Änderungen zwischen dem gerade importierten und dem vorhergehenden Release in das Arbeitsverzeichnis schreibt (die beiden `-j` Optionen) und dabei auch inzwischen neu dazugekommene Directories im Arbeitsverzeichnis erzeugt (`-d`):

   ```
   > cd Soccer
   > cvs update -d -jv0_08 -jv0_09
   ```

[2] In einem Fall führte ein Abbruch dazu, daß interne Locks und Sticky-Tags falsch gesetzt waren. Nachfolgende Änderungen wurden dann wie weitere Importe im Repository gespeichert und nicht wie eigene Änderungen. Das Problem fiel erst beim nächsten Import auf.

6. Der Repository-Manager behebt, evtl. zusammen mit den anderen Mitgliedern der Gruppe, die gemeldeten Konflikte und testet das Ergebnis des Imports. Ein abschließendes `cvs commit` schreibt das Endergebnis in das Repository.