SelfLinux-0.10.0

CVS für Fortgeschrittene

Autor: Karl Fogel ()
Formatierung: Johnny Graber (*linux@jgraber.ch*)
Lizenz: GPL

Der folgende Text enthält das Kapitel 4 der deutschen Übersetzung des Buches "Open Source Development with CVS", welche unter der GNU Public License veröffentlicht wurden.

Das SelfLinux-Team

Inhaltsverzeichnis

1 Über die Grundlagen hinaus

Wir haben nun die grundlegenden Konzepte der Benutzung von CVS und der Administration des Archivs behandelt und werden jetzt einen Blick darauf werfen, wie CVS in den gesamten Entwicklungsvorgang integriert werden kann. Der grundlegende CVS-Arbeitszyklus von Checkout (auschecken), Update, Commit, Update, Commit und so weiter ist schon anhand von Beispielen in Kapitel 2 illustriert worden. Im Folgenden werden wir diesen Zyklus noch ausführlicher beschreiben. Außerdem werden wir in diesem Kapitel untersuchen, wie CVS den Entwicklern dabei helfen kann, miteinander zu kommunizieren, wie CVS einen Überblick über die Projektaktivitäten und den Fortschritt des Projekts liefern kann, wie es einen dabei unterstützt, Entwicklungszweige (sog. Branches) abzuspalten und wieder mit der Hauptentwicklungslinie (dem sog. Trunk, also Stamm oder Rumpf) zu vereinen, und schließlich, wie man mit CVS häufig wiederkehrende Vorgänge automatisieren kann. Für die Nutzung mancher dieser Eigenschaften werden wir neue CVS-Kommandos einführen, bei vielen genügt die erweiterte Nutzung der Kommandos, die Sie bereits kennen gelernt haben.

2 CVS als Telefon

Ein Hauptvorteil bei der Benutzung von CVS für ein Projekt liegt darin, dass es sowohl als Kommunikationsmittel als auch als Protokollant dienen kann. Dieser Abschnitt konzentriert sich darauf, wie CVS verwendet werden kann, um die Teilnehmer eines Projekts darüber, was im Projekt vorgeht, auf dem Laufenden zu halten. Die Teilnehmer müssen jedoch informiert werden wollen, denn wenn sich jemand entschließt, diese Kommunikationsfähigkeiten nicht zu verwenden, dann kann auch CVS nicht darüber hinweg helfen.

3 Watches: Wer arbeitet wann woran

Normalerweise behandelt CVS jede Arbeitskopie als isolierte »Sandbox1«. Niemand bekommt mit, was Sie an Ihrer Kopie bearbeiten, solange bis Sie Ihre Änderungen durch einen Commit freigeben. Umgekehrt wissen Sie nicht, was »die anderen« an ihrer Kopie tun, von den herkömmlichen Kommunikationsmitteln, wie »Hey, ich werde jetzt an parse.c arbeiten, sagt mir, wenn ihr etwas daran machen wollt, damit wir Konflikte vermeiden können!« den Gang runter zu brüllen, einmal abgesehen.

Dieses formlose Vorgehen mag für Projekte ausreichen, bei denen jeder eine grobe Ahnung davon hat, wer wofür zuständig ist. Hingegen wird diese Vorgehensweise scheitern, wenn eine große Entwicklergruppe an allen möglichen Teilen des Projekts aktiv ist und Konflikte vermieden werden sollen. In solchen Fällen überschneiden sich die Zuständigkeiten, man kann jedoch nichts »den Gang runter brüllen«, da man über die ganze Welt verteilt arbeitet.

Eine »Watch2« genannte CVS-Funktion gibt den Entwicklern ein Mittel in die Hand, um andere darüber zu informieren, wer gerade an welchen Dateien arbeitet. Indem er ein Watch auf eine Datei setzt, kann ein Entwickler CVS veranlassen, ihn zu benachrichtigen, sobald ein anderer beginnt, ebenfalls an ihr zu arbeiten. Diese Benachrichtigungen werden normalerweise als E-Mail versendet, man kann aber auch andere Benachrichtigungsmethoden einrichten.

Um Watches verwenden zu können, muss man ein oder zwei Dateien im Administrationsteil des Archivs anpassen. Weiterhin ist es erforderlich, dass die Entwickler einige Zusatzschritte in den Checkout/Update/Commit-Zyklus einfügen. Die auf Archivseite nötigen Änderungen sind recht einfach: Man muss CVSROOT/notify so abändern, dass CVS weiß, wie Benachrichtigungen durchgeführt werden sollen. Eventuell muss man auch einige Zeilen in CVSROOT/users einfügen, um die externen E-Mail-Adressen festzulegen.

In ihrer Arbeitskopie müssen die Entwickler CVS mitteilen, welche Dateien beobachtet werden sollen, damit CVS ihnen eine Benachrichtigung schicken kann, wenn jemand anders an diesen Dateien zu arbeiten beginnt. Die Entwickler müssen außerdem CVS informieren, wenn sie damit anfangen oder aufhören, an einer Datei zu arbeiten, damit CVS es denjenigen mitteilen kann, die ein Watch gesetzt haben. Folgende Kommandos sind für diese Zusatzschritte zuständig:

```
cvs watch
cvs edit
cvs unedit
```

Bemerkung
Das Kommando `watch` unterscheidet sich von den normalen CVS-Kommandos dadurch, dass es zusätzliche Unterkommandos benötigt, wie etwa `cvs watch add...`, `cvs watch remove...`, und so weiter.

Im Folgenden werfen wir einen Blick darauf, wie man Watches im Archiv aktiviert, und darauf, wie Watches aus Entwicklersicht eingesetzt werden. Zwei Beispielnutzer, jrandom und qsmith, haben beide ihre eigene Arbeitskopie desselben Projektes, sie können sogar an verschiedenen Computern arbeiten. Wie üblich wird in allen Beispielen davon ausgegangen, dass die Umgebungsvariable `$CVSROOT` bereits gesetzt wurde, sodass es bei keinem CVS-Kommando nötig ist, `-d <ARCHIV>` mit anzugeben.

3.1 Watches im Archiv aktivieren

Zunächst muss CVSROOT/notify angepasst werden, um E-Mail-Benachrichtigungen einzuschalten. Das kann von einem Entwickler erledigt werden, der Administrator muss es nur dann selbst erledigen, wenn kein

Entwickler die Berechtigung hat, die Verwaltungsdateien zu verändern. In beiden Fällen ist zuerst der Administrationsbereich auszuchecken und die Datei `notify` zu editieren:

```
user@linux ~/ # cvs -q co CVSROOT

U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/loginfo
U CVSROOT/modules
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg

user@linux ~/ # cd CVSROOT
user@linux ~/ # emacs notify
user@linux ~/ # ...
```

Falls Sie notify zum ersten Mal öffnen, werden Sie in etwa Folgendes sehen:

Datei notify

```
# The "notify" file controls where notifications from watches set by
# "cvs watch add" or "cvs edit" are sent.  The first entry on a line is
# a regular expression which is tested against the directory that the
# change is being made to, relative to the &#36;CVSROOT.  If it matches,
# then the remainder of the line is a filter program that should contain
# one occurrence of %s for the user to notify, and information on its
# standard input.
#
# "ALL" or "DEFAULT" can be used in place of the regular expression.
#
# For example:
# ALL mail %s -s "CVS notification"
```

Es muss lediglich die letzte Zeile einkommentiert werden, indem das führende Doppelkreuz (#) entfernt wird. Obwohl die Datei notify dieselbe flexible Schnittstelle (reguläre Ausdrücke für die Verzeichnisnamen) wie die anderen administrativen Dateien bietet, werden Sie diese Flexibilität hier wohl nie brauchen. Der einzig vorstellbare Grund, mehrere Zeilen mit jeweils einem regulären Ausdruck für einen bestimmten Teil des Archivs haben zu wollen, wäre, wenn Sie unterschiedliche Benachrichtigungsarten für die verschiedenen Projekte haben möchten. Wie auch immer, eine normale E-Mail ist ein völlig ausreichender Benachrichtigungsmechanismus, er wird für die meisten Projekte verwendet.

Um die E-Mail-Benachrichtigung einzuschalten, wird die Zeile

ALL mail %s -s "CVS notification"

wohl auf jedem normalen UNIX-Rechner funktionieren. Die Anweisung bewirkt, dass Benachrichtigungen als E-Mail versendet werden, mit »CVS notification« als Betreff, wobei der Ausdruck `ALL` wie üblich für alle Verzeichnisse steht. Wenn Sie diese Zeile einkommentiert haben, sollten Sie die Datei notify durch einen Commit dem Archiv zukommen lassen, damit es die Änderung bemerkt5:

```
user@linux ~/ # cvs ci -m "turned on watch notification"

cvs commit: Examining .
Checking in notify;
/usr/local/newrepos/CVSROOT/notify,v <-- notify
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database

user@linux ~/ # floss$
```

Die Anpassung der Datei notify kann auch schon alles sein, was im Archiv zum Aktivieren der Watches getan werden muss. Wenn jedoch auch Entwickler an anderen Rechnern am Projekt teilnehmen, dann muss wahrscheinlich auch noch die Datei CVSROOT/users angepasst werden. Die Aufgabe der Datei users liegt darin, CVS mitzuteilen, wohin die E-Mail-Benachrichtigungen für diejenigen Benutzer gesendet werden müssen, die keine lokale E-Mail-Adresse haben. Das Format jeder Zeile der Datei users:

CVS_BENUTZERNAME:EMAILADRESSE

Zum Beispiel:
qsmith:quentinsmith@ganzweitweg.com

Der CVS-Benutzername am Anfang der Zeile korrespondiert mit einem CVS-Benutzernamen in CVSROOT/password (falls vorhanden und falls die Zugriffsmethode pserver verwendet wird). Ansonsten ist es der Benutzername auf Serverseite der Person, die CVS verwendet. Dem Doppelpunkt folgt die externe E-Mail-Adresse, an die CVS die Watch-Benachrichtigungen für diesen Benutzer schicken soll.

Unglücklicherweise existiert die Datei users (zum Zeitpunkt des Schreibens) nicht in der CVS-Standarddistribution. Weil es sich um eine administrative Datei handelt, genügt es nicht, sie zu erzeugen, sie mit `cvs add` dem Archiv hinzuzufügen und einen Commit auszuführen, sondern man muss sie auch noch in die CVSROOT/checkoutlist eintragen, damit immer eine ausgecheckte Kopie im Archiv vorliegt.

Hier eine Beispielssitzung, die diesen Vorgang illustriert:

```
user@linux ~/ # floss$ emacs checkoutlist

 ... (die Zeile für die Datei users hinzufügen) ...

user@linux ~/ # floss$ emacs users

 ... (die Zeile für qsmith hinzufügen) ...

user@linux ~/ # floss$ cvs add users
user@linux ~/ # floss$ cvs ci -m "added users to checkoutlist, qsmith to
users"

cvs commit: Examining .
Checking in checkoutlist;
/usr/local/newrepos/CVSROOT/checkoutlist,v <-- checkoutlist
new revision: 1.2; previous revision: 1.1
done
Checking in users;
/usr/local/newrepos/CVSROOT/users,v <-- users
new revision: 1.2; previous revision: 1.1
done
```

```
cvs commit: Rebuilding administrative file database

user@linux ~/ #
```

Es ist durchaus möglich, das erweiterte Format für die E-Mail-Adresse in CVSROOT/users zu verwenden, man muss nur darauf achten, dass alle Leerzeichen durch Anführungszeichen gekapselt werden. Zum Beispiel ist Folgendes möglich:

qsmith:"Quentin Q. Smith <quentinsmith@ganzweitweg.com>"

oder auch:

qsmith:'Quentin Q. Smith <quentinsmith@ganzweitweg.com>'

So geht es allerdings nicht:

qsmith:"Quentin Q. Smith" <quentinsmith@ganzweitweg.com>

Probieren Sie im Zweifelsfall das Kommando, wie es in der Datei notify angegeben ist, manuell aus. Es muss lediglich das %s in

mail %s -s "CVS notification"

durch das, was in der Datei users nach dem Doppelpunkt folgt, ersetzt werden. Wenn es auf der Kommandozeile funktioniert, dann sollte es auch in der Datei users funktionieren.

Die Datei checkout sollte jetzt in etwa so aussehen6:

Datei checkout

```
# The "checkoutlist" file is used to support additional version controlled
# administrative files in &#36;CVSROOT/CVSROOT, such as template files.
#
# The first entry on a line is a filename which will be checked out from
# the corresponding RCS file in the &#36;CVSROOT/CVSROOT directory.
# The remainder of the line is an error message to use if the file cannot
# be checked out.
#
# File format:
#
# [&lt;whitespace>]&lt;filename&gt;&lt;whitespace&gt;&lt;error
message&gt;&lt;end-of-line&gt;
#
# comment lines begin with '#'
users Unable to check out 'users' file in CVSROOT.
```

Und die Datei users sieht in etwa so aus:

Datei users

```
qsmith:quentinsmith@ganzweitweg.com
```

Das Archiv ist nun für Watches vorbereitet. Werfen wir einen Blick darauf, was die Entwickler an ihren

Arbeitskopien machen müssen.

Watches bei der Entwicklung verwenden

Zuerst wird ein Entwickler eine Arbeitskopie auschecken und sich selbst auf die Liste der Beobachter (»Watch List«) einer Datei des Projekts setzen:

```
user@linux ~/ # whoami

jrandom

user@linux ~/ # cvs -q co myproj

U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c

user@linux ~/ # cd myproj
user@linux myproj/ # cvs watch add hello.c
user@linux myproj/ #
```

Das letzte Kommando, `cvs watch add hello.c`, teilt CVS mit, dass jrandom benachrichtigt werden soll, wenn jemand beginnt, an hello.c zu arbeiten. Der Benutzer jrandom wird also der Liste der Beobachter hinzugefügt. Damit CVS Benachrichtigungen versenden kann, sobald eine Datei bearbeitet wird, muss der sie bearbeitende Benutzer das CVS mitteilen, indem er zunächst `cvs edit` für die Datei aufruft. CVS hat keine andere Möglichkeit, festzustellen, dass jemand eine Datei zu bearbeiten beginnt. Sobald der Checkout durchgeführt wurde, ist CVS normalerweise bis zum nächsten Update oder Commit nicht mehr beteiligt. Letzteres geschieht aber erst, nachdem die Datei schon bearbeitet wurde:

```
user@linux ~/ # whoami

qsmith

user@linux ~/ # cvs -q co myproj

U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c

user@linux ~/ # paste$ cd myproj
user@linux ~/ # paste$ cvs edit hello.c
user@linux ~/ # paste$ emacs hello.c

...
```

Wenn qsmith `cvs edit hello.c` aufruft, schaut sich CVS die Watch-Liste für hello.c an, sieht, dass jrandom darauf vertreten ist, und schickt jrandom eine E-Mail, die ihm sagt, dass qsmith damit begonnen hat, die

Datei zu bearbeiten. Die E-Mail hat sogar den Anschein, als käme sie von qsmith:

Email an jrandom

```
From: qsmith
Subject: CVS notification
To: jrandom
Date: Sat, 17 Jul 1999 22:14:43 -0500
myproj hello.c
--
Triggered edit watch on /usr/local/newrepos/myproj
By qsmith
```

Außerdem wird jrandom jedes Mal, wenn qsmith (oder sonst jemand) den Commit einer neuen Revision von hello.c ausführt, eine weitere E-Mail erhalten:

Email commit

```
myproj hello.c
--
Triggered commit watch on /usr/local/newrepos/myproj
By qsmith
```

Nach Erhalt dieser E-Mails wird sich jrandom sofort ein Update von hello.c holen wollen, damit sie sehen kann, was qsmith geändert hat; möglicherweise wird sie qsmith eine E-Mail schreiben, um herauszufinden, warum er an der Datei Änderungen vorgenommen hat. Es bleibt zu beachten, dass niemand qsmith gezwungen hat, `cvs edit` auszuführen, vermutlich wollte er also, dass jrandom erfährt, was er vorhatte. Andererseits: Selbst wenn er `cvs edit` vergessen hätte, würde sein Commit dennoch das Verschicken von Benachrichtigungen auslösen. Der Sinn, `cvs edit` zu benutzen, liegt darin, dass Beobachtende benachrichtigt werden, bevor man die Arbeit an einer Datei aufnimmt. Die Beobachtenden können einen dann informieren, wenn sie einen Konflikt kommen sehen.

CVS geht davon aus, dass jeder, der `cvs edit` auf eine Datei anwendet, selbst - zumindest temporär - auf die Liste der Beobachter der Datei gesetzt werden will, für den Fall, dass jemand anders beginnt, Änderungen vorzunehmen. Als qsmith `cvs edit` aufgerufen hat, ist er ein Beobachter von hello.c geworden. Sowohl er als auch jrandom wären benachrichtigt worden, hätte ein Dritter `cvs edit` auf die Datei angewendet oder den Commit einer neuen Revision vorgenommen.

Allerdings glaubt CVS, dass Personen, die eine Datei bearbeiten, nur so lange auf der Beobachtungsliste sein möchten, wie sie daran arbeiten. Benutzer dieser Art werden automatisch von der Beobachtungsliste genommen, wenn sie mit ihren Änderungen fertig sind. Sollten sie es wünschen, dauerhafte Beobachter der Datei zu werden, müssten sie cvs add watch aufrufen. CVS unterstellt, dass jemand mit seiner Arbeit an einer Datei fertig ist, sobald er den Commit der Datei ausgeführt hat - jedenfalls bis zum nächsten Mal.

Jeder, der auf die Beobachtungsliste einer Datei nur dadurch gelangt, dass er `cvs edit` aufruft, wird als temporärer Beobachter geführt und automatisch von der Beobachtungsliste genommen, sobald er die Änderungen an der Datei per Commit abgeschlossen hat. Wenn er danach wieder etwas bearbeiten möchte, muss er `cvs edit` noch einmal aufrufen.

Die Annahme, dass der erstbeste Commit die Arbeitssitzung an der Datei beendet, ist natürlich nur eine Vermutung, denn CVS kann ja nicht wissen, wie viele Commits jemand benötigt, um seine Änderungen abzuschließen. Die Vermutung trifft am wahrscheinlichsten bei so genannten »One-off7«-Korrekturen zu,

Änderungen, bei denen jemand nur schnell eine Kleinigkeit korrigieren möchte und den Commit gleich durchführt. Für längere Arbeitssitzungen an einer Datei, die mehrere Commits umfassen, sollten die Benutzer sich dauerhaft auf die Beobachtungsliste der Datei setzen:

```
user@linux ~/ # cvs watch add hello.c
user@linux ~/ # cvs edit hello.c
user@linux ~/ # paste$ emacs hello.c

...

user@linux ~/ # paste$ cvs commit -m "print hello in Sanskrit"
```

Nun bleibt qsmith auch nach einem Commit auf der Beobachtungsliste von hello.c, da er `watch add` ausgeführt hat. Übrigens wird qsmith keine Benachrichtigung seiner eigenen Änderungen erhalten, die bekommen nur andere Beobachter. CVS ist klug genug, einen nicht über eigene Änderungen zu informieren.

Eine Bearbeitungssitzung abschließen

Wenn Sie kein Commit durchführen wollen, die Bearbeitungssitzung aber explizit beenden wollen, können Sie das mit `cvs unedit` tun:

```
user@linux ~/ # cvs unedit hello.c
```

Doch Vorsicht! Dadurch wird nicht nur allen Beobachtenden gemeldet, dass man mit dem Bearbeiten fertig ist, zusätzlich wird Ihnen noch angeboten, alle Änderungen an der Datei, die Sie noch nicht durch einen Commit bestätigt haben, rückgängig zu machen:

```
user@linux ~/ # paste$ cvs unedit hello.c

hello.c has been modified; revert changes? y

user@linux / #
```

Wenn Sie hier mit »y« (für »yes«) antworten, wird CVS all Ihre Änderungen rückgängig machen und allen Beobachtenden mitteilen, dass Sie die Datei nicht mehr bearbeiten. Wenn Sie mit »n« (»no«) antworten, behält CVS Ihre Änderungen bei und vermerkt Sie weiterhin als Bearbeiter der Datei. (Es wird also keine Mitteilung verschickt - alles verhält sich so, als hätten Sie `cvs unedit` nie aufgerufen.) Es mag ein wenig besorgniserregend erscheinen, dass CVS anbietet, mit einem einzigen Tastendruck alle Änderungen zu verwerfen; die Logik dahinter ist allerdings einleuchtend: Wenn Sie »der Welt« mitteilen, dass Sie eine Bearbeitungssitzung abschließen, dann sind die Änderungen, die Sie noch nicht mit einem Commit bestätigt haben, vermutlich solche, die Sie gar nicht beibehalten wollen. CVS sieht es jedenfalls so. Seien Sie also vorsichtig.

Kontrolle über die zu beobachtenden Aktionen

Beobachter werden normalerweise über drei Aktionsarten informiert: das Bearbeiten einer Datei (edits), den Commit und das Ende der Arbeiten an einer Datei (unedits). Falls Sie jedoch beispielsweise nur über Commits benachrichtigt werden wollen, können Sie die Benachrichtigungen auch mit der `-a`-Option einschränken (a für Aktion):

```
user@linux ~/ # cvs watch add -a commit hello.c
```

Wenn Sie jedoch sowohl das Editieren als auch den Commit einer Datei beobachten wollen, können Sie die `-a`-Option auch zwei Mal angeben:

```
user@linux ~/ # cvs watch add -a edit -a commit hello.c
```

Wenn Sie eine Watch zusammen mit der `-a`-Option setzen, werden schon existierende Watches dadurch nicht entfernt. Wenn Sie schon alle drei Aktionen auf hello.c beobachten, hat der Aufruf

```
user@linux ~/ # cvs watch add -a commit hello.c
```

keinen Effekt - Sie sind weiterhin Beobachter aller drei Aktionen. Um Watches zu entfernen, sollten Sie

```
user@linux ~/ # cvs watch remove hello.c
```

aufrufen, was, wie `add`, normalerweise alle drei Aktionen von der Beobachtung ausschließt. Falls Sie `-a`-Argumente übergeben, werden nur die Watches entfernt, die Sie angeben:

```
user@linux ~/ # floss$ cvs watch remove -a commit hello.c
```

Das bedeutet, dass Sie keine weitere Benachrichtigung über ein Commit bekommen möchten, aber weiterhin über Beginn und Ende des Editierens informiert werden möchten - vorausgesetzt, Sie beobachteten diese schon vorher.

Es gibt zwei spezielle Aktionen, die Sie zusammen mit der -a-Option übergeben können: `all` für alle oder `none` für keine. Da es das vorgegebene Verhalten von CVS ist, wenn `-a` nicht mit angegeben wird, alle Aktionen zu beobachten, und da `none`, also keine Aktionen zu beobachten, dasselbe ist, wie sich ganz von der Beobachtungsliste herunterzunehmen, ist eine Situation, in der Sie eine dieser Sonderoptionen mit angeben wollen, nur schwer vorzustellen. Andererseits ist die `-a`-Option auch für `edit` verwendbar, und in diesem Fall kann es von Nutzen sein, `all` oder `none` anzugeben. Zum Beispiel könnte jemand, der nur sehr kurz an einer Datei arbeitet, keine Benachrichtigungen darüber wünschen, was andere an der Datei ändern. So bewirkt das Kommando

```
user@linux ~/ # whoami
qsmith
user@linux ~/ # cvs edit -a none README.txt
```

dass die Beobachter der Datei README.txt darüber benachrichtigt werden, dass qsmith drauf und dran ist, sie zu bearbeiten, qsmith selbst würde aber nicht als kurzzeitiger Beobachter geführt werden (was normalerweise der Fall wäre), da er darum gebeten hat, keine Aktionen zu beobachten.

Beachten Sie, dass Sie nur die eigenen Beobachtungen mit dem `cvs watch`-Kommando beeinflussen können. Sie können aufhören, eine Datei zu beobachten, aber Sie können fremde Watches nicht ändern.

Wie man herausfindet, wer was beobachtet

Manchmal kann es nützlich sein, zu überprüfen, wer eine Datei beobachtet, bevor man `cvs edit` aufruft, oder man möchte einfach sehen, wer was unter Beobachung hat, ohne sich selbst auf eine Beobachtungsliste zu setzen. Oder man hat einfach vergessen, wie denn der eigene Status nun genau ist. Wenn man einige Watches

gesetzt und wieder zurückgesetzt hat und einige Dateien per Commit zurückgegeben hat, kann man sehr leicht den Überblick darüber verlieren, was man beobachtet und bearbeitet.

CVS bietet zwei Kommandos, mit denen man anzeigen kann, wer Dateien beobachtet und wer Dateien unter Bearbeitung hat: `cvs watchers` und `cvs editors`.

```
user@linux ~/ # whoami

jrandom

user@linux ~/ # cvs watch add hello.c
user@linux ~/ # cvs watchers hello.c

hello.c jrandom edit unedit commit

user@linux ~/ # cvs watch remove -a unedit hello.c
user@linux ~/ # cvs watchers hello.c

hello.c jrandom edit commit

user@linux ~/ # cvs watch add README.txt
user@linux ~/ # cvs watchers

README.txt jrandom edit unedit commit
hello.c jrandom edit commit

user@linux ~/ #
```

Beachten Sie, dass beim letzten `cvs watchers`-Kommando keine Dateien angegeben sind. Darum werden die Beobachter aller Dateien angezeigt - und natürlich nur die Dateien, die Beobachter haben.

Dieses Verhalten haben die `watch`- und `edit`-Kommandos mit anderen CVS-Kommandos gemeinsam. Wenn Dateinamen mit angegeben werden, wirken sie sich auf diese Dateien aus. Geben Sie Verzeichnisnamen an, sind alle Dateien in den Verzeichnissen und in den darin liegenden Unterverzeichnissen gemeint. Wird gar nichts spezifiziert, arbeiten die Kommandos auf dem aktuellen Verzeichnis und auf allem darunter, auf allen darunter liegenden Ebenen. Zum Beispiel (als Fortsetzung derselben Sitzung):

```
user@linux ~/ # cvs watch add a-subdir/whatever.c
user@linux ~/ # cvs watchers

README.txt jrandom edit unedit commit
hello.c jrandom edit commit
a-subdir/whatever.c jrandom edit unedit commit

user@linux ~/ # cvs watch add
user@linux ~/ # cvs watchers

README.txt jrandom edit unedit commit
foo.gif jrandom edit unedit commit
hello.c jrandom edit commit unedit
a-subdir/whatever.c jrandom edit unedit commit
a-subdir/subsubdir/fish.c jrandom edit unedit commit
b-subdir/random.c jrandom edit unedit commit

user@linux / #
```

Durch die letzten beiden Kommandos wurde jrandom Beobachterin aller Dateien des Projekts und hat dann die Beobachtungsliste für alle Dateien im Projekt abgerufen. Die Ausgabe von `cvs watchers` passt nicht immer exakt in die Spalten, da CVS Tabulatoren mit Informationen variabler Länge vermischt, doch das Format ist einheitlich:

```
[DATEINAME] [leerzeichen] BEOBACHTER [leerzeichen] BEOBACHTETE_AKTION ...
```

Folgendes passiert, wenn qsmith eine dieser Dateien editiert:

```
user@linux ~/ # cvs edit hello.c
user@linux ~/ # cvs watchers

README.txt jrandom edit unedit commit
foo.gif jrandom edit unedit commit
hello.c jrandom edit commit unedit
qsmith tedit tunedit tcommit
a-subdir/whatever.c jrandom edit unedit commit
a-subdir/subsubdir/fish.c jrandom edit unedit commit
b-subdir/random.c jrandom edit unedit commit
```

Bei der Datei hello.c ist ein weiterer Beobachter hinzugekommen: qsmith selbst. Beachten Sie, dass der Dateiname am Anfang der Zeile nicht wiederholt wird, sondern durch Leerzeichen ersetzt wird - falls Sie jemals ein Programm schreiben, das die Ausgabe von `watchers` einliest, kann das wichtig sein. Da er hello.c bearbeitet, ist qsmith temporärer Beobachter der Datei - so lange, bis er eine neue Revision von hello.c per Commit erzeugt. Das `t` vor jeder der Aktionen zeigt an, dass es sich nur um temporäre Watches handelt. Wenn sich qsmith nun als normaler Beobachter von hello.c hinzufügt:

```
user@linux ~/ # paste$ cvs watch add hello.c

README.txt jrandom edit unedit commit
foo.gif jrandom edit unedit commit
hello.c jrandom edit commit unedit
qsmith tedit tunedit tcommit edit unedit commit
a-subdir/whatever.c jrandom edit unedit commit
a-subdir/subsubdir/fish.c jrandom edit unedit commit
b-subdir/random.c jrandom edit unedit commit
```

dann wird er sowohl als temporärer als auch als permanenter Beobachter aufgelistet. Man könnte erwarten, dass der permanente Status den temporären einfach überschreibt, sodass die Zeile folgendermaßen aussähe:

```
qsmith edit unedit commit
```

CVS kann aber nicht einfach den temporären Status ersetzen, da es nicht weiß, in welcher Reihenfolge die Aktionen ablaufen: Wird qsmith sich von der permanenten Beobachtungsliste entfernen, bevor er die Bearbeitungssitzung beendet, oder wird er die Änderungen abschließen und trotzdem Beobachter bleiben? Im ersten Fall würden die edit/unedit/commit-Aktionen verschwinden, die tedit/tunedit/tcommit würden aber bleiben. Im zweiten Fall wäre es umgekehrt.

Wie auch immer - dieser Aspekt der Beobachtungslisten ist normalerweise nicht von großem Interesse. Es genügt,

```
user@linux ~/ # floss$ cvs watchers
```

oder

```
user@linux ~/ # floss$ cvs editors
```

von der obersten Verzeichnisebene des Projekts aus aufzurufen, um zu sehen, wer was tut. Man braucht nicht zu wissen, wen welche Aktionen kümmern, wichtig sind die Namen der Personen und der Dateien.

Benutzer an Watches erinnern

Sie haben es wahrscheinlich schon bemerkt: Die Beobachtungs-Features sind völlig von der Kooperation aller Entwickler abhängig. Wenn jemand einfach eine Datei ändert, ohne zuvor `cvs edit` aufzurufen, wird es niemand mitbekommen, bis die Änderungen mit Commit beendet werden. Da `cvs edit` einen zusätzlichen, nicht routinemäßigen Schritt darstellt, wird er leicht vergessen.

Obwohl CVS niemanden zwingen kann, `cvs edit` zu verwenden, hat es dennoch einen Mechanismus, um die Leute wenigstens daran zu erinnern: das Kommando `watch on`:

```
user@linux ~/ # cvs -q co myproj

U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c

user@linux / # cd myproj
user@linux / # cvs watch on hello.c
user@linux / #
```

Durch den Aufruf von `cvs watch on hello.c` bewirkt jrandom, dass zukünftige Checkouts von myproj die Datei hello.c in der Arbeitskopie mit dem Nur-lese-Status erzeugen. Versucht nun qsmith, daran zu arbeiten, wird er feststellen, dass die Datei nur lesbar ist, und wird so daran erinnert, zuerst `cvs edit` aufzurufen:

```
user@linux ~/ # cvs -q co myproj

U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c

user@linux ~/ # cd myproj
user@linux ~/ # ls -l

total 6
drwxr-xr-x 2 qsmith users 1024 Jul 19 01:06 CVS/
-rw-r--r-- 1 qsmith users 38 Jul 12 11:28 README.txt
drwxr-xr-x 4 qsmith users 1024 Jul 19 01:06 a-subdir/
drwxr-xr-x 3 qsmith users 1024 Jul 19 01:06 b-subdir/
```

```
-rw-r--r-- 1 qsmith users 673 Jun 20 22:47 foo.gif
-r--r--r-- 1 qsmith users 188 Jul 18 01:20 hello.c

user@linux / #
```

Sobald er das getan hat, wird die Datei auch beschreibbar werden; er kann sie nun editieren. Wenn er die Änderungen mittels Commit abschickt, wird sie wieder nur lesbar:

```
user@linux ~/ # cvs edit hello.c
user@linux ~/ # ls -l hello.c

-rw-r--r-- 1 qsmith users 188 Jul 18 01:20 hello.c
emacs hello.c
 ...

user@linux / # cvs commit -m "say hello in Aramaic" hello.c

Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v <-- hello.c
new revision: 1.12; previous revision: 1.11
done

user@linux / # ls -l hello.c

-r--r--r-- 1 qsmith users 210 Jul 19 01:12 hello.c

user@linux / #
```

Sein `edit` und sein `commit` wird Benachrichtigungen an alle Beobachter der Datei hello.c versenden. Beachten Sie, dass jrandom nicht notwendigerweise selbst Beobachter der Datei ist. Durch den Aufruf von `cvs watch on hello.c` hat sich jrandom nicht selbst auf die Beobachtungsliste gesetzt, sie hat nur erwirkt, dass hello.c beim Checkout den Nur-lese-Status erhält. Jeder, der eine Datei beobachten will, muss selbst daran denken, sich auf die Beobachtungsliste zu setzen, dabei kann ihm CVS nicht helfen.

Es mag eher die Ausnahme sein, die Beobachtung einer einzelnen Datei einzuschalten. Normalerweise werden Watches für das gesamte Projekt eingeschaltet:

```
user@linux ~/ # cvs -q co myproj

U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c

user@linux ~/ # cd myproj
user@linux / # cvs watch on
user@linux / #
```

Diese Aktion kommt der Ankündigung einer Grundsatzentscheidung für das gesamte Projekt gleich: »Bitte verwenden Sie `cvs edit`, um Beobachtern mitzuteilen, woran Sie gerade arbeiten, und beobachten Sie ruhig jede Datei, die Sie interessiert oder für die Sie sich verantwortlich fühlen.« Jede Datei des Projekts erhält nun

beim Checkout den Nur-lese-Status, auf dass sich die Leute daran erinnern mögen, dass sie `cvs edit` aufzurufen haben, bevor sie irgendwelche Arbeiten daran durchführen.

Obwohl beobachtete Dateien beim Checkout wieder nur lesbar werden, geschieht dies durch Updates seltsamerweise nicht. Hätte qsmith den Checkout seiner Arbeitskopie ausgeführt, bevor jrandom `cvs watch on` aufgerufen hat, blieben seine Dateien schreibbar, selbst nach einem Update. Jedoch werden alle Dateien, die er mit einem Commit dem Archiv zukommen lässt, nachdem jrandrom die Watches eingeschaltet hat, nur lesbar. Schaltet jrandom die Watches ab

```
user@linux ~/ # floss$ cvs watch off
```

werden qsmiths Dateien nicht von Geisterhand wieder beschreibbar. Andererseits werden sie auch nicht nach seinen Commits wieder nur lesbar, was der Fall wäre, wären die Watches noch eingeschaltet.

Es bleibt zu beachten, dass qsmith, wenn er richtig hinterhältig wäre, die Dateien seiner Arbeitskopie auch mit dem normalen Unix-Kommando `chmod` beschreibbar machen und so `cvs edit` völlig umgehen könnte:

```
user@linux ~/ # chmod u+w hello.c
```

oder, wenn er alles auf einen Schlag erledigen wollte:

```
user@linux ~/ # paste$ chmod -R u+w .
```

CVS kann nichts dagegen tun. Die Arbeitskopien sind - bedingt durch ihre Natur - private »Sandkästen", durch die Beobachtungs-Features kann man sie ein klein wenig unter öffentliche Beobachtung stellen, jedoch nur so weit, wie es der Entwickler erlaubt. Nur wenn ein Entwickler etwas ausführt, das das Archiv berührt, wie zum Beispiel ein Commit, gibt er seine Privatsphäre ohne Wenn und Aber auf.

Das Verhältnis zwischen `watch add`, `watch remove`, `watch on` und `watch off` mag leicht verwirrend erscheinen. Vielleicht hilft es, die Systematik noch einmal zusammenzufassen: `add` und `remove` sind dafür da, um Benutzer auf die Beobachtungsliste einer Datei zu setzen oder sie davon zu entfernen; sie haben nichts damit zu tun, ob Dateien beim Checkout das Nur-lese-Attribut erhalten oder ob sie nach einem Commit (wieder) nur lesbar werden. Bei `on` und `off` geht es nur um Dateirechte. Sie haben nichts damit zu tun, wer auf einer Beobachtungsliste ist, es geht lediglich darum, die Entwickler daran zu erinnern, sich an die Beobachtungsübereinkunft zu halten, indem die Dateien der Arbeitskopie zunächst nur die Leseberechtigung erhalten.

Vielleicht wirkt das Ganze ja auch ein wenig inkonsistent. Schließlich laufen Watches dem Grundkonzept von CVS entgegen. Es ist eine Teilabkehr von dem idealisierten Universum, in dem viele Entwickler völlig frei an ihren Arbeitskopien arbeiten, unbemerkt von den anderen, bis sie sich entschließen, etwas per Commit zu veröffentlichen. Mit Watches gibt CVS den Entwicklern eine bequeme Methode, die anderen darüber, was in der eigenen Arbeitskopie vor sich geht, zu informieren; allerdings ohne die Möglichkeit, das für die Arbeit mit Watches richtige Verhalten zu erzwingen. Auch gibt es kein festgelegtes Konzept davon, woraus denn eine Arbeitssitzung nun genau besteht. Nichtsdestotrotz können Watches unter gewissen Umständen nützlich sein, wenn die Entwickler sie verwenden.

Wie Watches im Archiv aussehen

Damit mal wieder ein unnötiges Mysterium ausgerottet wird, werfen wir einen kurzen Blick darauf, wie Watches im Archiv implementiert sind. Nur einen ganz kurzen Blick, denn es ist wirklich kein schöner Anblick:

Wenn man ein Watch setzt

```
user@linux ~/ # pwd

/home/jrandom/myproj

user@linux / # cvs watch add hello.c
user@linux / # cvs watchers

hello.c jrandom edit unedit commit

user@linux / #
```

hält CVS das in einer gesonderten Datei, CVS/fileattr, im zuständigen Unterverzeichnis des Archivs fest:

```
user@linux ~/ # cd /usr/local/newrepos
user@linux / # ls

CVSROOT/ myproj/

user@linux / # cd myproj
user@linux / # ls

CVS/ a-subdir/ foo.gif,v
README.txt,v b-subdir/ hello.c,v

user@linux / # cd CVS
user@linux / # ls

fileattr

user@linux / # cat fileattr

Fhello.c _watchers=jrandom>edit+unedit+commit

user@linux / #
```

Die Tatsache, dass fileattr in einem Unterverzeichnis des Archivs mit Namen CVS abgelegt wird, heißt jetzt aber nicht, dass das Archiv zu einer Arbeitskopie geworden ist. Es ist einfach so, dass der Name CVS schon für die Buchführung in der Arbeitskopie reserviert ist und sich CVS deshalb sicher sein kann, dass es niemals ein Unterverzeichnis mit diesem Namen im Archiv speichern muss.

Ich werde das Dateiformat von fileattr hier nicht formal beschreiben, man kapiert es recht schnell, wenn man zusieht, wie sich die Datei von Kommando zu Kommando verändert:

```
user@linux ~/ # cvs watch add hello.c
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

Fhello.c _watchers=jrandom>edit+unedit+commit

user@linux / # cvs watch add README.txt
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

Fhello.c _watchers=jrandom>edit+unedit+commit
FREADME.txt _watchers=jrandom>edit+unedit+commit
```

```
user@linux / # cvs watch on hello.c
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

Fhello.c _watchers=jrandom>edit+unedit+commit;_watched=
FREADME.txt _watchers=jrandom>edit+unedit+commit

user@linux / # cvs watch remove hello.c
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

Fhello.c _watched=
FREADME.txt _watchers=jrandom>edit+unedit+commit

user@linux / # cvs watch off hello.c
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

FREADME.txt _watchers=jrandom>edit+unedit+commit

user@linux / #
```

Die Informationen über Bearbeitungssitzungen werden ebenfalls in fileattr abgelegt. Folgendes geschieht, wenn qsmith sich als Bearbeiter einträgt:

```
user@linux ~/ # cvs edit hello.c
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

Fhello.c _watched=;_editors=qsmith>Tue Jul 20 04:53:23 1999 GMT+floss\
+/home/qsmith/myproj;_watchers=qsmith>tedit+tunedit+tcommit
FREADME.txt _watchers=jrandom>edit+unedit+commit
```

Letztlich bleibt zu bemerken, dass CVS die Datei fileattr sowie das Unterverzeichnis CVS löscht, wenn für keine der Dateien in einem Verzeichnis noch Beobachter oder Bearbeiter vorhanden sind:

```
user@linux ~/ # cvs unedit
user@linux / # cvs watch off
user@linux / # cvs watch remove
user@linux / # cat /usr/local/newrepos/myproj/CVS/fileattr

cat: /usr/local/newrepos/myproj/CVS/fileattr: No such file or directory
```

Nach dieser kurzen Enthüllung sollte klar sein, dass man die Analyse des fileattr-Formats besser CVS überlässt. Die Hauptmotivation dafür, eine grobe Ahnung von diesem Format zu haben (von der tiefen Befriedigung zu wissen, was hinter den Vorhängen vor sich geht, einmal abgesehen), liegt darin, dass man möglicherweise eine Erweiterung an der Funktionalität der Watches plant oder dass man eventuell auftretende Probleme beheben kann. Ansonsten ist es ausreichend, zu wissen, dass es keinen Grund zur Beunruhigung gibt, wenn plötzlich ein Unterverzeichnis namens CVS in Ihrem Archiv auftaucht. Es handelt sich einfach um den einzig sicheren Ort, an dem CVS Metainformationen wie eben die Beobachtungslisten speichern kann.

4 Log-Nachrichten und Commit-E-Mails

Commit-E-Mails sind beim Commit abgeschickte Benachrichtigungen, welche die Log-Nachrichten und die vom Commit betroffenen Dateien auflisten. Sie gehen normalerweise an alle Teilnehmer des Projekts, manchmal auch an sonstige Interessierte. Da die Details, wie Commit-E-Mails eingerichtet werden, schon von Kapitel 4 abgedeckt worden sind, werde ich sie hier nicht wiederholen. Mir ist allerdings aufgefallen, dass Commit-E-Mails manchmal unerwartete Seiteneffekte auf Projekte haben können, Effekte, die Sie in Betracht ziehen sollten, wenn Sie Commit-E-Mails für Ihr Projekt einsetzen wollen.

Erstens: Rechnen Sie damit, dass die Nachrichten meistens ignoriert werden. Ob sie gelesen werden oder nicht, hängt zumindest zum Teil davon ab, wie häufig Commits in Ihrem Projekt vorkommen. Tendieren die Entwickler eher dazu, täglich eine große Änderung per Commit zu veröffentlichen, oder eher dazu, es über viele kleine Änderungen, verteilt über den Tag, zu tun? Je näher Ihr Projekt dem zweiten Fall ist und je stärker die vielen kleinen Commits den ganzen Tag lang auf die Entwickler herunter prasseln, um so weniger werden sie sich um jede einzelne Nachricht kümmern.

Daraus folgt nicht, dass die Benachrichtigungen keinen Zweck erfüllen, man sollte nur nicht davon ausgehen, dass jeder jede Nachricht liest. Es ist immer noch ein bequemer Weg, die Sachen im Auge zu behalten - wer was macht -, ohne das Aufdringliche, das Watches an sich haben. Gehen die E-Mails an eine öffentlich zugängliche Mailingliste, so hat man einen wundervollen Mechanismus, um interessierten Benutzern (Entwickler in spe!) die Möglichkeit zu bieten, täglich mitzubekommen, was am Quelltext geschieht.

Vielleicht sollten Sie in Betracht ziehen, einen Entwickler abzustellen, die Log-Nachrichten zu verfolgen und den Überblick über das gesamte Projekt zu behalten (ein guter Projektleiter tut das natürlich sowieso). Wenn die Zuständigkeiten klar verteilt sind, beispielsweise wenn bestimmte Entwickler bestimmten Unterverzeichnissen des Projekts zugeordnet sind, könnten Sie ganz besonders schicke Vorkehrungen in CVSROOT/loginfo treffen, sodass jede verantwortliche Partei gesondert markierte Nachrichten darüber erhält, was in ihrem Zuständigkeitsbereich passiert. Das hilft dabei sicherzustellen, dass die Entwickler wenigstens die E-Mails lesen, die zu ihren Unterverzeichnissen gehören.

Ein interessanterer Effekt tritt ein, wenn Commit-E-Mails nicht ignoriert werden. Die Leute fangen an, sie als Echtzeit-Kommunikationsmittel zu verwenden. Dadurch kann sich so etwas ergeben:

> Finished feedback form; fixed the fonts and background colors on the home page. Whew! Anyone want to go to Mon Lung for lunch?

Es ist nichts Falsches daran, die Logs auf diese Art zu »mißbrauchen«. Dadurch wird es interessant, sie später noch einmal zu lesen. Dennoch sollte sich jeder darüber im Klaren sein, dass sich Log-Nachrichten, wie etwa die folgende, nicht nur per E-Mail verbreiten, sondern sich auch in der Projekthistorie verewigen. Über die Spezifikationen eines Kunden zu hadern, mag ein verbreiteter Zeitvertreib unter Programmierern sein; man kann sich leicht vorstellen, dass jemand beim Commit eine Log-Nachricht wie folgende schreibt, wissend, dass die anderen Programmierer sie als E-Mail erhalten:

> Truncate four-digit years to two-digits in input. What the customer wants, the customer gets, no matter how silly & wrong. Sigh.

Kein Zweifel - eine amüsante E-Mail, aber was passiert, wenn der Kunde sich eines Tages die Log-Nachrichten ansieht? (Ich wette, dass ähnliche Befürchtungen schon bei mehr als einer Site dazu geführt haben, CVS/loginfo so einzurichten, dass Mechanismen vorgeschaltet werden, die Anstößiges aus den Log-Nachrichten heraus halten!)

Im Großen und Ganzen scheinen Commit-E-Mails die Leute davon abzuhalten, zu kurze oder unverständliche

Log-Nachrichten zu schreiben, was möglicherweise »eine gute Sache« ist. Jedoch müssen sie hin und wieder daran erinnert werden, dass jeder, der irgendwann einmal die Logs liest, ein potenzieller Adressat ist, nicht nur die Empfänger der E-Mails.

Wie man Log-Nachrichten nach dem Commit ändert

Für den Fall, dass jemand eine Log-Nachricht nach dem Commit bereut, ermöglicht es CVS, die Logs nachträglich zu ändern. Man erledigt dies mit der `-m`-Option, die man zusammen mit dem `admin`-Kommando verwendet (auf das Kommando wird später in diesem Kapitel noch detaillierter eingegangen). Das Kommando erlaubt es, genau eine Log-Nachricht (pro Revision, pro Datei) auf einmal zu ändern. Das funktioniert so:

```
user@linux ~/ # floss$ cvs admin -m 1.7:"Truncate four-digit years to two
in input." date.c

RCS file: /usr/local/newrepos/someproj/date.c,v
done
```

Die ursprüngliche Log-Nachricht, die mit dem Commit von Revision 1.7 abgelegt wurde, ist durch eine völlig unschuldige - wenn auch weniger scharfzüngige - ersetzt worden. (Nicht den Doppelpunkt vergessen, der die Revisionsnummer von der Log-Nachricht trennt.)

Wenn die »falsche« Log-Nachricht beim Commit von mehreren Dateien verwendet wurde, muss man `cvs admin` für jede Datei getrennt aufrufen. Es handelt sich also um eines der wenigen Kommandos, bei denen CVS erwartet, dass nur ein einziger Dateiname als Argument übergeben wird:

```
user@linux ~/ # cvs admin -m 1.2:"very boring log message" hello.c
README.txt foo.gif

cvs admin: while processing more than one file:
cvs [admin aborted]: attempt to specify a numeric revision
```

Lassen Sie sich nicht davon verwirren, dass Sie dieselbe Fehlermeldung erhalten, als wenn Sie gar keine Dateinamen mit angegeben hätten. Das liegt daran, dass CVS in dem Fall alle Dateien, die im aktuellen Verzeichnis und darunter liegen, als implizite Argumente betrachtet:

```
user@linux ~/ # cvs admin -m 1.2:"very boring log message"

cvs admin: while processing more than one file:
cvs [admin aborted]: attempt to specify a numeric revision
```

(Es ist bei CVS-Fehlermeldungen leider häufig der Fall, dass man die Dinge aus der Sicht von CVS betrachten muss, damit sie einen Sinn ergeben!)

Der `admin -m`-Aufruf ändert die Projekthistorie, seien Sie also vorsichtig. Es wird keine Aufzeichnung geben, die besagt, dass die Log-Nachricht jemals verändert wurde - es wird einfach so aussehen, als ob die Revision schon beim ursprünglichen Commit die neue Log-Nachricht erhalten hätte. Die alte Meldung wird nirgends ihre Spuren hinterlassen (außer Sie heben die Original-E-Mail auf).

Obwohl sein Name scheinbar besagt, dass nur der designierte CVS-Administrator es benutzen kann, kann

normalerweise jeder `cvs admin` aufrufen, solange er Schreibzugriff auf das fragliche Projekt hat. Existiert jedoch eine Unix-Benutzergruppe namens cvsadmin, dann ist die Nutzung dieses Kommandos auf die Mitglieder der Gruppe beschränkt. (Mit der Ausnahme, dass immer noch jeder `cvs admin -k` benutzen kann.) Dennoch benutzt man es besser mit großer Vorsicht, denn die Möglichkeit, die Geschichte des Projekts umzuschreiben, ist verglichen mit anderen, potenziell zerstörerischen Fähigkeiten noch harmlos. In Kapitel 9 gibt es noch mehr zu `admin`, zusammen mit Wegen, dessen Benutzung einzuschränken.

5 Wie man eine Arbeitskopie los wird

Bei normaler CVS-Nutzung wird man den Verzeichnisbaum der Arbeitskopie so wie jeden anderen Verzeichnisbaum los:

```
user@linux ~/ # rm -rf myproj
```

Wenn Sie sich Ihrer Arbeitskopie auf diese Art entledigen, werden die übrigen Entwickler allerdings nicht mitbekommen, dass Sie deren Nutzung eingestellt haben. CVS stellt ein Kommando zur Verfügung, mit dem man die Arbeitskopie explizit terminieren kann. Sehen Sie `release`8 als das Gegenstück zu `checkout` an - Sie teilen dem Archiv mit, dass Sie mit Ihrer Arbeitskopie abgeschlossen haben. Wie `checkout` wird `release` vom übergeordneten Verzeichnis im Verzeichnisbaum aufgerufen:

```
user@linux ~/ # pwd

/home/qsmith/myproj

user@linux ~/ # cd ..
user@linux ~/ # ls

myproj
cvs release myproj
You have [0] altered files in this repository.
Are you sure you want to release directory 'myproj': y
```

Falls Sie gegenüber dem Archiv noch nicht per Commit gespeicherte Änderungen in Ihrer Arbeitskopie haben, wird `release` fehlschlagen, soll heißen, dass lediglich die modifizierten Dateien aufgelistet werden und sonst nichts geschieht. Vorausgesetzt der Verzeichnisbaum ist »sauber« (komplett auf dem aktuellen Stand), vermerkt `release` im Archiv, dass die Arbeitskopie freigegeben wurde.

Sie können `release` auch anweisen, den Verzeichnisbaum für Sie zu löschen, indem Sie `-d` mit angeben:

```
user@linux ~/ # ls

myproj

user@linux ~/ # cvs release -d myproj

You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'myproj: y

user@linux ~/ # ls
```

Zum Zeitpunkt von CVS Version 1.10.6 ist das `release`-Kommando nicht in der Lage, die Position des Archivs anhand der Arbeitskopie zu erkennen (da `release` außerhalb der Arbeitskopie aufgerufen wird). Man muss daher entweder die globale Option `-d <ARCHIV>` mit angeben oder sicherstellen, dass die Umgebungsvariable CVSROOT richtig gesetzt worden ist. (Dieser Fehler mag in zukünftigen CVS-Versionen behoben worden sein.)

Im Cederqvist wird behauptet, dass, wenn Sie `release` verwenden, statt den Arbeitsverzeichnisbaum einfach zu löschen, diejenigen benachrichtigt werden, die auf die freigegebenen Dateien ein Watch gesetzt haben, genau so, als hätten Sie `unedit` aufgerufen. Als ich das experimentell überprüfen wollte, habe ich festgestellt, dass das nicht stimmt.

6 Überblick über die Projekthistorie

In Kapitel 4 habe ich kurz das Kommando `cvs history` erwähnt. Dieses Kommando zeigt eine Zusammenfassung aller `checkouts, commits, updates, rtags` und `releases` an, die im Archiv getätigt wurden (vorausgesetzt, Logging war bei Erstellung der Datei CVSROOT/history im Archiv aktiv). Mit den folgenden Kommandos können Sie Inhalt und Erscheinungsbild der Zusammenfassung mit verschiedenen Optionen kontrollieren.

Der erste Schritt liegt darin, sicherzustellen, dass Logging im Archiv eingeschaltet ist. Der Archivadministrator sollte erst einmal sicherstellen, dass es eine Datei history gibt:

```
user@linux ~/ # cd /usr/local/newrepos/CVSROOT
user@linux / # ls -l history

ls: history: No such file or directory
```

und falls es keine gibt, sollte er eine wie folgt erstellen:

```
user@linux ~/ # touch history
user@linux / # ls -l history

-rw-r--r-- 1 jrandom cvs 0 Jul 22 14:57 history
```

Die Datei history soll außerdem von jedem, der das Archiv benutzt, beschreibbar sein, ansonsten wird dieser jedes Mal, wenn er ein CVS-Kommando, das diese Datei verändert, ausführen will, eine Fehlermeldung erhalten. Am einfachsten macht man die Datei von allen beschreibbar:

```
user@linux ~/ # chmod a+rw history
user@linux / # ls -l history

-rw-rw-rw- 1 jrandom cvs 0 Jul 22 14:57 history
```

Bemerkung
Im Fall, dass das Archiv mit dem Kommando cvs init angelegt wurde, existiert die Datei bereits. Um die Zugriffsrechte muss man sich allerdings noch kümmern.

In den folgenden Beispielen wird davon ausgegangen, dass History Logging, also die Aufzeichnung der Projekthistorie, schon eine Weile eingeschaltet war, sodass sich schon einiges an Daten in der Datei history angesammelt hat.

Die Ausgabe von `cvs history` ist etwas knapp geraten (sie ist vermutlich nicht dafür gedacht, von Menschen analysiert zu werden, obwohl sie mit ein wenig Übung gut lesbar ist). Rufen wir das Kommando kurz auf und sehen, was wir bekommen:

```
user@linux ~/ # paste$ pwd

/home/qsmith/myproj
```

```
user@linux ~/ # paste$ cvs history -e -a

O 07/25 15:14 +0000 qsmith myproj =mp= ~/*
M 07/25 15:16 +0000 qsmith 1.14 hello.c myproj == ~/mp
U 07/25 15:21 +0000 qsmith 1.14 README.txt myproj == ~/mp
G 07/25 15:21 +0000 qsmith 1.15 hello.c myproj == ~/mp
A 07/25 15:22 +0000 qsmith 1.1 goodbye.c myproj == ~/mp
M 07/25 15:23 +0000 qsmith 1.16 hello.c myproj == ~/mp
M 07/25 15:26 +0000 qsmith 1.17 hello.c myproj == ~/mp
U 07/25 15:29 +0000 qsmith 1.2 goodbye.c myproj == ~/mp
G 07/25 15:29 +0000 qsmith 1.18 hello.c myproj == ~/mp
M 07/25 15:30 +0000 qsmith 1.19 hello.c myproj == ~/mp
O 07/23 03:45 +0000 jrandom myproj =myproj= ~/src/*
F 07/23 03:48 +0000 jrandom =myproj= ~/src/*
F 07/23 04:06 +0000 jrandom =myproj= ~/src/*
M 07/25 15:12 +0000 jrandom 1.13 README.txt myproj == ~/src/myproj
U 07/25 15:17 +0000 jrandom 1.14 hello.c myproj == ~/src/myproj
M 07/25 15:18 +0000 jrandom 1.14 README.txt myproj == ~/src/myproj
M 07/25 15:18 +0000 jrandom 1.15 hello.c myproj == ~/src/myproj
U 07/25 15:23 +0000 jrandom 1.1 goodbye.c myproj == ~/src/myproj
U 07/25 15:23 +0000 jrandom 1.16 hello.c myproj == ~/src/myproj
U 07/25 15:26 +0000 jrandom 1.1 goodbye.c myproj == ~/src/myproj
G 07/25 15:26 +0000 jrandom 1.17 hello.c myproj == ~/src/myproj
M 07/25 15:27 +0000 jrandom 1.18 hello.c myproj == ~/src/myproj
C 07/25 15:30 +0000 jrandom 1.19 hello.c myproj == ~/src/myproj
M 07/25 15:31 +0000 jrandom 1.20 hello.c myproj == ~/src/myproj
M 07/25 16:29 +0000 jrandom 1.3 whatever.c myproj/a-subdir ==
~/src/myproj
```

Ist doch alles klar verständlich, oder?

Bevor wir die Ausgabe näher untersuchen, sei angemerkt, dass der Aufruf zusammen mit zwei Optionen geschah: `-e` und `-a`. Wenn Sie `history` aufrufen, werden Sie fast immer Optionen mit angeben wollen, die festlegen, welche Daten wie angezeigt werden sollen. Darin unterscheidet es sich von den meisten anderen CVS-Kommandos, die normalerweise bereits dann etwas Sinnvolles tun, wenn sie ganz ohne Optionen aufgerufen werden. In unserem Beispiel bedeuten die Optionen »alles« (jede Art von Ereignis) beziehungsweise »alle« (für alle Benutzer).

Das `history`-Kommando unterscheidet sich von anderen Kommandos auch noch darin, dass, obwohl es normalerweise aus einer Arbeitskopie heraus aufgerufen wird, es seine Ausgabe nicht auf das in der Arbeitskopie enthaltene Projekt beschränkt. Stattdessen zeigt es die gesamte Historie aller Projekte im Archiv an - die Arbeitskopie dient nur dazu, CVS mitzuteilen, welchem Archiv die history-Daten entnommen werden sollen. (Im vorangegangenen Beispiel waren die einzigen history-Daten die vom Projekt myproj, daher sieht man sonst keine.)

Das generelle Format der Ausgabe ist

```
KÜRZEL     DATUM     BENUTZER     [REVISION]     [DATEI]     PFAD_IM_ARCHIV
NAME_DER_ARBEITSKOPIE
```

Die Buchstabenkürzel beziehen sich auf die verschiedenen CVS-Operationen, wie in Tabelle 6.1 dargestellt.

Für Operationen (wie z.B. `checkout`), die sich auf das Projekt als Ganzes anstelle von einzelnen Dateien davon beziehen, werden die REVISION und DATEI weggelassen, stattdessen wird der Pfad des Archivs

zwischen die Gleichheitszeichen gesetzt.

Obwohl die Ausgabe des Kommandos `history` vom Design her als kompakte, interpretierbare Eingabe für andere Programme gedacht ist, gibt CVS einem viel Kontrolle über Umfang und Inhalt. Die in Tabelle 6.2 aufgelisteten Optionen kontrollieren, welche Typen von Ereignissen gemeldet werden. Wenn Sie ausgewählt haben, welche Ereignisse Sie angezeigt bekommen möchten, können Sie noch weitere Filterkriterien der Tabelle 6.3 entnehmen.

Buchstabenkürzel	Bedeutung
O	Auschecken (Checkout) einer Datei
T	Marke (Tag)
F	Freigabe (siehe Release)
W	Update (Benutzerdatei wurde gelöscht, Datei aus entries entfernt. Die Datei war im Archiv gelöscht worden.)
U	Update (Datei hat unveränderte Datei des Benutzers überschrieben)
G	Update (Datei wurde erfolgreich mit einer vom Benutzer veränderten Datei verschmolzen)
C	Update (Datei wurde verschmolzen, aber Konflikt mit einer vom Benutzer geänderten Datei, conflicts)
M	Commit (einer modifizierten Datei)
A	Commit (einer neuen Datei, add)
R	Commit (Löschen einer Datei, remove)
E	Export (siehe Kapitel 9)

Tabelle 6.1 Die Bedeutung der Buchstabenkürzel

Option	Bedeutung
-m MODUL	Zeige Vorgänge, die MODUL betreffen
-c	Zeige Commit-Vorgänge
-o	Zeige Checkout-Vorgänge
-T	Zeige alle Vorgänge, die Marken (Tags) betreffen
-x KÜRZEL	Zeige alle Vorgänge, die vom Typ KÜRZEL sind (mindestens eines aus OTFWUGCMARE, siehe Tabelle 6.1)
-e	Zeige einfach alle Vorgänge

Tabelle 6.2 Optionen, die nach Ereignistyp filtern

Option	Bedeutung
-a	Zeige die Aktionen aller Benutzer
-w	Zeige nur die Aktionen, die aus dieser Arbeitskopie heraus vorgenommen wurden
-l	Zeige nur die letzte solche Aktion dieses Benutzers
-u BENUTZER	Zeige die Einträge für BENUTZER

Tabelle 6.3 Optionen, die Benutzer heraus filtern

7 Detaillierter Überblick über Projektaktivitäten: Anmerkungen

Wenn das `history`-Kommando einem einen groben Überblick über die Projektaktivitäten gibt, dann ist das `annotate`9-Kommando sozusagen das Mikroskop, das es einem ermöglicht, die Details zu erkennen. Mit

`annotate` kann man sehen, wer die letzte Person war, die ihre Finger an den einzelnen Zeilen einer Datei hatte, und bei welcher Revision dies geschah:

```
user@linux ~/ # floss$ cvs annotate

Annotations for README.txt
***************
1.14 (jrandom 25-Jul-99): blah
1.13 (jrandom 25-Jul-99): test 3 for history
1.12 (qsmith 19-Jul-99): test 2
1.11 (qsmith 19-Jul-99): test
1.10 (jrandom 12-Jul-99): blah
1.1 (jrandom 20-Jun-99): Just a test project.
1.4 (jrandom 21-Jun-99): yeah.
1.5 (jrandom 21-Jun-99): nope.
Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.15 (jrandom 25-Jul-99): /* another test for history */
1.13 (qsmith 19-Jul-99): /* random change number two */
1.10 (jrandom 12-Jul-99): /* test */
1.21 (jrandom 25-Jul-99): printf ("Hellooo, world!\n");
1.3 (jrandom 21-Jun-99): printf ("hmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.11 (qsmith 18-Jul-99): /* added this comment */
1.16 (qsmith 25-Jul-99): /* will merge these changes */
1.18 (jrandom 25-Jul-99): /* will merge these changes too */
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
Annotations for a-subdir/whatever.c
***************
1.3 (jrandom 25-Jul-99): /* A completely non-empty C file. */
Annotations for a-subdir/subsubdir/fish.c
***************
1.2 (jrandom 25-Jul-99): /* An almost completely empty C file. */
Annotations for b-subdir/random.c
***************
1.1 (jrandom 20-Jun-99): /* A completely empty C file. */
```

Die Ausgabe von `annotate` lässt sich intuitiv erfassen. Links sind Revisionsnummer, Entwickler und da Datum, zu dem die fragliche Zeile hinzugefügt oder verändert wurde. Rechts sieht man die eigentliche Zeile zur jeweils aktuellen Revision. Da jede Zeile mit Anmerkungen (also Revisionsnummer, Entwickler und Datum) versehen ist, bekommt man den ganzen Inhalt der Datei aufgelistet, aber um die Anmerkungen nach rechts verschoben.

Wenn man eine Revisionsnummer oder eine Marke (Tag) spezifiziert, bekommt man die Anmerkungen, die zu dieser Revision aktuell waren; soll heißen: Es werden die letzten Modifikationen jeder Zeile zu oder bis zu dieser Revision angezeigt. Das ist wahrscheinlich der üblichste Weg, `annotate` zu benutzen: Eine einzige Datei zu einer bestimmten Revision zu untersuchen, um zu erkennen, welche Entwickler an welchen Teilen der Datei aktiv waren.

Zum Beispiel kann man in der Ausgabe aus dem vorangegangenen Beispiel sehen, dass die aktuellste Revision von hello.c 1.21 ist, als jrandom etwas an folgender Zeile änderte:

```
printf ("Hellooo, world!\n");
```

Ein Weg herauszufinden, was sie getan hat, ist, sich den diff10 zwischen dieser Revision und der vorangegangenen anzusehen:

```
user@linux ~/ # cvs diff -r 1.20 -r 1.21 hello.c

index: hello.c
===================================================================
RCS file: /usr/local/newrepos/myproj/hello.c,v
retrieving revision 1.20
retrieving revision 1.21
diff -r1.20 -r1.21
9c9
< printf ("Hello, world!\n");
--
> printf ("Hellooo, world!\n");
```

Eine weitere Möglichkeit, unter Beibehaltung des dateiweiten Überblicks über die allgemeinen Aktivitäten herauszufinden, was geschehen ist, liegt darin, die aktuellen Anmerkungen mit denen der vorigen Version zu vergleichen:

```
user@linux ~/ # floss$ cvs annotate -r 1.20 hello.c

Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.15 (jrandom 25-Jul-99): /* another test for history */
1.13 (qsmith 19-Jul-99): /* random change number two */
1.10 (jrandom 12-Jul-99): /* test */
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.3 (jrandom 21-Jun-99): printf ("hmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.11 (qsmith 18-Jul-99): /* added this comment */
1.16 (qsmith 25-Jul-99): /* will merge these changes */
1.18 (jrandom 25-Jul-99): /* will merge these changes too */
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
```

Obwohl diff die Fakten über die Veränderung des Quelltextes in knapperer Form darstellt, können die Anmerkungen vorzuziehen sein, denn durch sie wird der geschichtliche Kontext hergestellt, indem gezeigt wird, wie lange die vorige Ausführung vorhanden war (in unserem Fall die ganze Zeit, seit Revision 1.1). Dieses Wissen kann Ihnen bei der Entscheidung helfen, ob Sie in die Logs schauen wollen, um die Motivation für die Änderungen herauszufinden:

```
user@linux ~/ # cvs log -r 1.21 hello.c

RCS file: /usr/local/newrepos/myproj/hello.c,v
Working file: hello.c
head: 1.21
branch:
locks: strict
access list:
symbolic names:
 random-tag: 1.20
 start: 1.1.1.1
 jrandom: 1.1.1
keyword substitution: kv
total revisions: 22; selected revisions: 1
description:
----------------------------
evision 1.21
date: 1999/07/25 20:17:42; author: jrandom; state: Exp; lines: +1 -1
say hello with renewed enthusiasm
=============================================
```

Zusätzlich zu -r können Sie die Anmerkungen auch mit der Option `-D DATUM` filtern:

```
user@linux ~/ # cvs annotate -D "5 weeks ago" hello.c

Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.1 (jrandom 20-Jun-99): }
floss$ cvs annotate -D "3 weeks ago" hello.c
Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.3 (jrandom 21-Jun-99): printf ("hmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
```

Anmerkungen und Verzweigungen

Wenn Sie keine weiteren Optionen angeben, zeigt `annotate` immer die Aktivitäten der Stammversion (engl. trunk). (Die Tendenz, die Stammversion so zu bevorzugen, ist entweder ein Bug oder ein Feature, je nach Standpunkt.) Sie können CVS zwingen, die Anmerkungen einer abgezweigten Version auszugeben, indem Sie

die Marke dieses Zweiges als Argument für `-r` übergeben. Hier ein Beispiel einer Arbeitskopie, in der sich hello.c in einer abgezweigten Version namens Brancho_Gratuito befindet und in der mindestens eine Änderung in dem Zweig per Commit vorgenommen wurde:

```
user@linux ~/ # cvs status hello.c

===================================================
File: hello.c Status: Up-to-date
 Working revision: 1.10.2.2 Sun Jul 25 21:29:05 1999
 Repository revision: 1.10.2.2 /usr/local/newrepos/myproj/hello.c,v
 Sticky Tag: Brancho_Gratuito (branch: 1.10.2)
 Sticky Date: (none)
 Sticky Options: (none
floss$ cvs annotate hello.c
Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.10 (jrandom 12-Jul-99): /* test */
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.3 (jrandom 21-Jun-99): printf ("hmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
floss$ cvs annotate -r Brancho_Gratuito hello.c
Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.10 (jrandom 12-Jul-99): /* test */
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.10.2.2 (jrandom 25-Jul-99): printf ("hmmmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.10.2.1 (jrandom 25-Jul-99): printf ("added this line");
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
```

Sie können auch die Nummer der Zweigversion übergeben:

```
user@linux ~/ # cvs annotate -r 1.10.2 hello.c

Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.10 (jrandom 12-Jul-99): /* test */
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
```

```
1.10.2.2 (jrandom 25-Jul-99): printf ("hmmmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.10.2.1 (jrandom 25-Jul-99): printf ("added this line");
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
```

oder auch eine vollständige Revisionsnummer innerhalb der Zweigversion:

```
user@linux ~/ # cvs annotate -r 1.10.2.1 hello.c

Annotations for hello.c
***************
1.1 (jrandom 20-Jun-99): #include <stdio.h>
1.1 (jrandom 20-Jun-99):
1.1 (jrandom 20-Jun-99): void
1.1 (jrandom 20-Jun-99): main ()
1.1 (jrandom 20-Jun-99): {
1.10 (jrandom 12-Jul-99): /* test */
1.1 (jrandom 20-Jun-99): printf ("Hello, world!\n");
1.3 (jrandom 21-Jun-99): printf ("hmmm\n");
1.4 (jrandom 21-Jun-99): printf ("double hmmm\n");
1.10.2.1 (jrandom 25-Jul-99): printf ("added this line");
1.2 (jrandom 21-Jun-99): printf ("Goodbye, world!\n");
1.1 (jrandom 20-Jun-99): }
```

Wenn Sie so vorgehen, vergessen Sie nicht, dass die Nummern nur für diese eine Datei gültig sind. Generell ist es wohl besser, - sofern möglich - den Namen der Zweigversion zu benutzen.

8 Verwendung der Schlüsselwortexpansion

Sie erinnern sich vielleicht daran, dass Schlüsselwortexpansion kurz in Kapitel 2 erwähnt wurde. RCS-Schlüsselwörter sind spezielle Wörter, die in Dollarzeichen eingeschlossen sind und die CVS aus Textdateien heraussucht und zu Revisions-Kontrollinformationen expandiert. Wenn beispielsweise eine Datei

```
$Author$
```

enthält, dann wird CVS das beim Update dieser Datei auf eine bestimmte Revision durch den Benutzernamen derjenigen Person expandieren, die für den Commit der Revision verantwortlich ist:

```
$Author: jrandom $
```

CVS kümmert sich um diese Schlüsselwörter auch in ihrer expandierten Form, sodass sie, selbst wenn sie schon einmal expandiert wurden, auch weiterhin aktualisiert werden.

Obwohl Schlüsselwörter keine Informationen liefern, die nicht auch auf anderen Wegen erreichbar sind, bieten sie doch eine bequeme Möglichkeit, die Fakten über die Revisionskontrolle in die Textdatei einzubetten, sodass man keine obskuren CVS-Operationen durchführen muss.

Hier ein paar weitere gebräuchliche Schlüsselwörter:

`$Date$` ==> Datum des letzten Commit, wird zu ==> `$Date: 1999/07/26 06:39:46 $`

`$Id$` ==> Dateiname, Revision, Datum und Autor, wird zu ==> `$Id: hello.c,v 1.11 1999/07/26 06:39:46 jrandom Exp $`

`$Revision$` ==> genau was Sie denken, wird zu ==> `$Revision: 1.11 $`

`$Source$` ==> Pfad zur korrespondierenden Datei im Archiv, wird zu ==> `$Source: /usr/local/newrepos/tossproj/hello.c,v $`

`$Log$` ==> sammelt Log-Nachrichten für diese Datei an, wird zu ==> `$Log: hello.c,v $`

Revision 1.2 1999/07/26 06:47:52 jrandom ...and this is the second log message.

Revision 1.1 1999/07/26 06:39:46 jrandom This is the first log message...

Das Schlüsselwort `$Log$` ist hierbei das einzige, das zu mehreren Zeilen expandiert wird. Es ersetzt nicht - wie die anderen - die alte Expansion durch eine neue, sondern fügt direkt nach dem Schlüsselwort die neuste Expansion und zusätzlich noch eine Leerzeile ein. So wird die vorige Expansion weiter nach unten geschoben. Außerdem wird noch jeder Text, der zwischen dem Anfang der Zeile und `$Log$` steht, den expandierten Zeilen vorangestellt, damit die Log-Nachrichten im Quelltext einkommentiert werden. Wenn Sie beispielsweise das

```
// $Log$
```

in die Datei schreiben, wird es beim ersten Commit zu so etwas:

```
// $Log: hello.c,v $
// Revision 1.14 1999/07/26 07:03:20 jrandom
// this is the first log message...
//
```

Beim zweiten Commit:

```
// $Log: hello.c,v $
// Revision 1.15 1999/07/26 07:05:34 jrandom
// ...and this is the second log message...
//
// Revision 1.14 1999/07/26 07:03:20 jrandom
// this is the first log message...
```

Und so weiter:

```
// $Log: hello.c,v $
// Revision 1.16 1999/07/26 07:05:34 jrandom
// ...and this is the third!
//
// Revision 1.15 1999/07/26 07:04:40 jrandom
// ...and this is the second log message...
//
// Revision 1.14 1999/07/26 07:03:20 jrandom
// this is the first log message...
//
```

Wenn Sie nicht die gesamte Entwicklung der Log-Datei in Ihrer Datei haben wollen, können Sie die älteren Abschnitte entfernen, wenn es Ihnen zu lang wird. Die von `$Log$` zur Verfügung gestellte Funktionalität ist mit Sicherheit komfortabler, als cvs log zu bemühen, und mag sich bei Projekten lohnen, bei denen die Log-Dateien ständig gelesen werden müssen.

Eine üblichere Technik ist es, `$Revision$` in die Datei mit aufzunehmen und es als Versionsnummer des Programms zu verwenden. Das ist möglich, wenn das Projekt im Wesentlichen aus einer Datei besteht oder häufig neue Versionen veröffentlicht werden und sich eine Datei bei jeder neuen Version garantiert ändert. Sie können sogar die RCS-Schlüsselwörter direkt im Quelltext des Programms benutzen.

```
VERSION = "$Revision: 1.114 $";
```

CVS wird das Schlüsselwort wie jedes andere expandieren, es hat keine Vorstellung von der Semantik der Programmiersprache und geht nicht davon aus, dass die Anführungszeichen die Zeichenkette in irgendeiner Form schützen sollen.

Eine komplette Liste der Schlüsselwörter (es gibt noch ein paar weitere, ziemlich obskure) gibt es in Kapitel 9.

9 Eine prekäre Lage: Wie überlebt man die Arbeit mit Verzweigungen?

Verzweigungen sind gleichzeitig eine der wichtigsten und am leichtesten mißbrauchten Fähigkeiten von CVS. Es kann sehr hilfreich sein, wenn man gefährliche oder störende Änderungen in einer getrennten Entwicklungslinie isoliert, bis sie sich stabilisiert haben. Wenn sie jedoch nicht richtig gemanagt werden, können Verzweigungen ein Projekt ganz schnell in Verwirrung und Chaos stürzen, nämlich wenn die Entwickler den Überblick verlieren, welche Änderungen wann wieder zusammengeführt wurden.

Um erfolgreich mit Verzweigungen arbeiten zu können, sollte sich die Entwicklergruppe an folgende Regeln halten:

Halten Sie die gleichzeitig aktiven Verzweigungen möglichst klein. Je mehr Verzweigungen zur gleichen Zeit entwickelt werden, umso wahrscheinlicher ist es, dass sie Konflikte erzeugen, wenn sie zurück in die Hauptversion einfließen sollen. In der Praxis erreicht man das, indem man die Verzweigungen so häufig wie möglich (sobald eine Zweigversion an einem stabilen Punkt angelangt ist) mit der Hauptlinie verschmilzt und die Entwicklungsarbeit an der Hauptversion fortsetzt. Indem man die parallel laufenden Entwicklungen klein hält, kann jeder besser nachvollziehen, was in welcher Verzweigung vorgeht, und die Wahrscheinlichkeit, dass Konflikte auftreten, wird kleiner.

Bemerkung
Das heißt jetzt aber nicht, dass die absolute Anzahl an Verzweigungen im Projekt klein zu halten ist, lediglich die Zahl der Verzweigungen, an denen gleichzeitig gearbeitet wird, soll klein sein.

Minimieren Sie die Komplexität - also die Tiefe - in Ihrem Verzweigungsplan. Es mag Umstände geben, wo es angemessen ist, Verzweigungen einer Verzweigung zu haben, aber die sind spärlich gesät. (Man kann ein ganzes Leben lang programmieren, ohne jemals auf eine zu stoßen.) Nur weil CVS es ermöglicht, beliebig viele Ebenen von verschachtelten Verzweigungen zu haben und beliebige Verzweigungen zu vereinen, heißt das noch lange nicht, dass Sie das auch wollen. In den meisten Situationen ist es am besten, dass alle Verzweigungen ihre Wurzel in der Hauptversion haben und auch dahin zurückkehren.

Benutzen Sie einheitlich benannte Marken, um all ihre Verzweigungs- und Zusammenführungsereignisse zu kennzeichnen. Im Idealfall sollte die Bedeutung jeder Marke und ihr Verhältnis zu den übrigen Verzweigungen allein durch den Namen offensichtlich sein. (Dieser Punkt wird noch anhand der Beispiele klarer werden.)

Mit diesen Regeln im Kopf wenden wir uns nun einem typischen Szenario mit verzweigter Entwicklungsarbeit zu. Wir werden jrandom an der Hauptversion und qsmith an einer abgezweigten Version haben. Bedenken Sie aber, dass genauso gut mehrere Entwickler an beiden tätig sein könnten. Die normale Entwicklungsarbeit an jedweder Linie kann beliebig viele Personen umfassen, die Benennung und Zusammenführung überlässt man aber am besten genau einer Person auf jeder Seite, wie Sie gleich sehen werden.

10 Häufig mit der Hauptversion verschmelzen

Wir gehen davon aus, dass qsmith für einige Zeit an einer abgezweigten Version arbeiten möchte, damit er nicht die Hauptversion destabilisiert, die er mit jrandom teilt. Der erste Schritt liegt darin, die Verzweigung zu erzeugen. Beachten Sie, wie qsmith zunächst eine normale Marke (nicht verzweigend) am Punkt der Verzweigung erzeugt und erst dann die abgezweigte Version erstellt:

```
user@linux ~/ # pwd

/home/qsmith/myproj

user@linux ~/ # cvs tag Root-of-Exotic_Greetings

cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
cvs tag: Tagging b-subdir
T b-subdir/random.c

user@linux ~/ # cvs tag -b Exotic_Greetings-branch

cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
cvs tag: Tagging b-subdir
T b-subdir/random.c
```

Der Grund, zuerst die Hauptversion zu markieren, liegt darin, dass es eines Tages notwendig sein kann, die Hauptversion so abzurufen, wie sie zum Zeitpunkt der Erstellung der Verzweigung war. Wenn das jemals nötig ist, so hat man die Möglichkeit, eine Momentaufnahme der Hauptversion zu referenzieren, die vor der Verzweigung entstanden ist. Offensichtlich kann nicht die Marke der Verzweigung benutzt werden, da dann die abgezweigte Version abgerufen werden würde und nicht die Revisionen der Hauptversion, welche die Wurzel der Verzweigung bildet. Die einzige Möglichkeit ist die, eine normale Marke an der Revision anzubringen, an der die Verzweigung wurzelt. (Mancher hält sich so strikt an diese Regel, dass ich überlegt habe, sie als Verzweigungsregel Nummer 4 aufzuführen: Erstelle immer eine nichtverzweigende Marke zum Zeitpunkt der Verzweigung. Wie auch immer, auf manchen Servern wird dies nicht getan, und sie scheinen auch ohne auszukommen, sodass es letztlich einfach eine Geschmacksfrage bleibt.) Von nun an werde ich diese nichtverzweigende Marke als Verzweigungspunktmarkierung (Branch Point Tag) bezeichnen.

Beachten Sie auch, dass ein Namensschema eingehalten wird: Die Verzweigungspunktmarkierung fängt mit Root-of- an, gefolgt vom eigentlichen Namen der Verzweigung, wobei Unterstriche statt Bindestriche zur Worttrennung verwendet werden. Wenn die eigentliche Verzweigung angelegt wird, endet ihre Marke mit

-branch, sodass Sie sie schon anhand ihres Namens als Marke eines Zweiges erkennen können. (Die Verzweigungspunktmarkierung Root-of-Exotic_Greetings erhält kein -branch, da sie ja keine Marke eines Zweiges ist.) Sie müssen sich natürlich nicht an dieses spezielle Namensschema halten, solange Sie nur irgendeines verwenden.

Ich bin hier natürlich besonders pedantisch. In kleineren Projekten, bei denen jeder weiß, was von wem getan wird, und bei denen man sich leicht von einer kurzen Phase der Verwirrung erholt, muss man sich nicht unbedingt an diese Regeln halten. Ob Sie nun Verzweigungspunktmarkierungen verwenden oder ein striktes Namensschema für die Marken haben, hängt von der Komplexität des Projektes und vom Verzweigungsschema ab. (Vergessen Sie außerdem nicht, dass Sie sich jederzeit umentscheiden können und alte Marken für eine neues Namensschema aktualisieren können, indem Sie die nach altem Schema markierte Version abrufen, eine neue Marke anbringen und dann die alte Marke löschen.)

Jetzt ist qsmith bereit, an der abgezweigten Version zu arbeiten:

```
user@linux ~/ # cvs update -r Exotic_Greetings-branch

cvs update: Updating .
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
```

Er nimmt einige Änderungen an ein paar Dateien vor und führt einen Commit aus, der die Änderungen in den Zweig hineinbringt:

```
user@linux ~/ # emacs README.txt a-subdir/whatever.c b-subdir/random.c

...

user@linux ~/ # cvs ci -m "print greeting backwards, etc"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v <-- README.txt
new revision: 1.14.2.1; previous revision: 1.14
done
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v <-- whatever.c
new revision: 1.3.2.1; previous revision: 1.3
done
Checking in b-subdir/random.c;
/usr/local/newrepos/myproj/b-subdir/random.c,v <-- random.c
new revision: 1.1.1.1.2.1; previous revision: 1.1.1.1
done
```

In der Zwischenzeit setzt jrandom ihre Arbeit an der Hauptversion fort. Sie ändert auch zwei der drei Dateien, die qsmith geändert hat. Aus reiner Bosheit lassen wir sie einige Änderungen machen, die im Widerspruch zur Arbeit von qsmith stehen:

```
user@linux ~/ # emacs README.txt whatever.c

 ...

user@linux ~/ # cvs ci -m "some very stable changes indeed"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v <-- README.txt
new revision: 1.15; previous revision: 1.14
done
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v <-- whatever.c
new revision: 1.4; previous revision: 1.3
done
```

Der Konflikt zeigt sich natürlich noch nicht, denn keiner der Entwickler hat bisher versucht, den Zweig und die Hauptversion wieder zusammenzuführen. Jetzt nimmt jrandom die Zusammenführung vor:

```
user@linux ~/ # cvs update -j Exotic_Greetings-branch

cvs update: Updating .
RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
rcsmerge: warning: conflicts during merge
cvs update: Updating a-subdir
RCS file: /usr/local/newrepos/myproj/a-subdir/whatever.c,v
retrieving revision 1.3
retrieving revision 1.3.2.1
Merging differences between 1.3 and 1.3.2.1 into whatever.c
rcsmerge: warning: conflicts during merge
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
RCS file: /usr/local/newrepos/myproj/b-subdir/random.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.1.2.1
Merging differences between 1.1.1.1 and 1.1.1.1.2.1 into random.c

user@linux ~/ # cvs update

cvs update: Updating .
C README.txt
cvs update: Updating a-subdir
C a-subdir/whatever.c
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
M b-subdir/random.c
```

Zwei der Dateien haben Konflikte. Keine große Sache, denn mit ihrem üblichen Savoir-faire löst jrandom die

Konflikte auf, führt den Commit aus und markiert die Hauptversion als erfolgreich wieder vereinigt:

```
user@linux ~/ # emacs README.txt a-subdir/whatever.c

 ...

user@linux ~/ # cvs ci -m "merged from Exotic_Greetings-branch (conflicts
resolved)"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v <-- README.txt
new revision: 1.16; previous revision: 1.15
done
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v <-- whatever.c
new revision: 1.5; previous revision: 1.4
done
Checking in b-subdir/random.c;
/usr/local/newrepos/myproj/b-subdir/random.c,v <-- random.c
new revision: 1.2; previous revision: 1.1
done

user@linux ~/ # cvs tag merged-Exotic_Greetings

cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
cvs tag: Tagging b-subdir
T b-subdir/random.c
```

Währenddessen braucht qsmith aber nicht abzuwarten, bis die Zusammenführung abgeschlossen ist, bevor er mit seiner Programmierarbeit fortfährt, vorausgesetzt, er setzt eine Marke für die Änderungen, von denen aus jrandom die Zusammenführung durchführt. (Später wird jrandom den Namen dieser Marke benötigen, generell kommt es bei der Verwendung von Verzweigungen darauf an, dass die Entwickler häufig und ausführlich miteinander kommunizieren):

```
user@linux ~/ # cvs tag Exotic_Greetings-1

cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
```

```
cvs tag: Tagging b-subdir
T b-subdir/random.c

user@linux ~/ # paste$ emacs a-subdir/whatever.c

 ...

user@linux ~/ # paste$ cvs ci -m "print a randomly capitalized greeting"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v <-- whatever.c
new revision: 1.3.2.2; previous revision: 1.3.2.1
done
```

Natürlich sollte qsmith diese Änderungen durch eine Marke kennzeichnen, sobald er fertig ist:

```
user@linux ~/ # paste$ cvs -q tag Exotic_Greetings-2

T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
```

Während all das geschieht, nimmt jrandom an einer anderen Datei, die qsmith bei seinen jüngsten Arbeiten nicht angefasst hat, Veränderungen vor:

```
user@linux ~/ # floss$ emacs README.txt

 ...

user@linux ~/ # floss$ cvs ci -m "Mention new Exotic Greeting features"
README.txt

Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v <-- README.txt
new revision: 1.17; previous revision: 1.16
done
```

Jetzt hat qsmith eine weitere Änderung an der Zweigversion vorgenommen, und jrandom hat eine konfliktfreie Änderung an der Hauptversion vorgenommen. Folgendes geschieht, wenn jrandom erneut versucht, beide zusammenzuführen:

```
user@linux ~/ # floss$ cvs -q update -j Exotic_Greetings-branch

RCS file: /usr/local/newrepos/myproj/README.txt,v
```

```
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
rcsmerge: warning: conflicts during merge
RCS file: /usr/local/newrepos/myproj/a-subdir/whatever.c,v
retrieving revision 1.3
retrieving revision 1.3.2.2
Merging differences between 1.3 and 1.3.2.2 into whatever.c
rcsmerge: warning: conflicts during merge
RCS file: /usr/local/newrepos/myproj/b-subdir/random.c,v
retrieving revision 1.1
retrieving revision 1.1.1.1.2.1
Merging differences between 1.1 and 1.1.1.1.2.1 into random.c

user@linux ~/ # floss$ cvs -q update

C README.txt
C a-subdir/whatever.c
```

Es gibt Konflikte! Haben Sie es erwartet?

Das Problem liegt in der Semantik der Zusammenführung. In Kapitel 2 habe ich gezeigt, dass, wenn Sie

```
user@linux ~/ # floss$ cvs update -j Zweig
```

in einer Arbeitskopie ausführen, CVS die Unterschiede zwischen der Wurzel von ZWEIG und seinem derzeitigen Endpunkt in die Arbeitskopie einbringt. Das Problem mit diesem Verhalten, in dieser Situation, ist, dass die meisten dieser Änderungen schon beim ersten Zusammenführen, das jrandom durchgeführt hat, in die Hauptversion eingeflossen sind. Als CVS versucht hat, diese erneut einzubringen (über sie selbst), hat es natürlich den Konflikt bemerkt.

Was jrandom eigentlich tun wollte, war, die Änderungen zwischen dem letzten Zusammenführen mit dem Zweig und seinem aktuellen Endpunkt in ihrer Arbeitsgruppe zu vereinen. Sie können das, wie Sie sich vielleicht aus Kapitel 2 erinnern, mit zwei -j-Optionen für update bewerkstelligen, vorausgesetzt Sie wissen, welche Revision bei jeder Option anzugeben ist. Glücklicherweise hat jrandom exakt am Punkt der letzten Zusammenführung eine Marke gesetzt (Extralob für Vorausplanung!), sodass das kein Problem ist. Lassen Sie uns zuerst jrandom ihre Arbeitskopie in einem sauberen Zustand wiederherstellen, von wo aus sie dann die Zusammenführung erneut versuchen kann:

```
user@linux ~/ # rm README.txt a-subdir/whatever.c
user@linux ~/ # cvs -q update

cvs update: warning: README.txt was lost
U README.txt
cvs update: warning: a-subdir/whatever.c was lost
U a-subdir/whatever.c
```

Nun ist sie bereit, die Zusammenführung durchzuführen, diesmal mit der von qsmith praktischerweise gesetzten Marke:

```
user@linux ~/ # cvs -q update -j Exotic_Greetings-1 -j
```

```
Exotic_Greetings-branch

RCS file: /usr/local/newrepos/myproj/a-subdir/whatever.c,v
retrieving revision 1.3.2.1
retrieving revision 1.3.2.2
Merging differences between 1.3.2.1 and 1.3.2.2 into whatever.c

user@linux ~/ # cvs -q update

M a-subdir/whatever.c
```

Schon viel besser. Die Änderung von qsmith wurde in whatever.c eingearbeitet; jrandom kann nun den Commit ausführen und eine Marke setzen:

```
user@linux ~/ # cvs -q ci -m "merged again from Exotic_Greetings (1)"

Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v <-- whatever.c
new revision: 1.6; previous revision: 1.5
done

user@linux ~/ # cvs -q tag merged-Exotic_Greetings-1

T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
```

Selbst wenn qsmith vergessen hätte, eine Marke beim Zusammenführen anzubringen, wäre noch nicht alles verloren gewesen. Wenn jrandom ungefähr wüsste, wann qsmiths erste Änderungen stattgefunden haben, könnte sie mittels Datum filtern:

```
user@linux ~/ # cvs update -j Exotic_Greetings-branch:3pm -j
Exotic_Greetings_branch
```

Obgleich nützlich als letzter Ausweg, ist das Filtern nach Datum ein weniger geeignetes Mittel, da die Änderungen auf Grund von persönlichen Erinnerungen anstelle von verlässlichen Vorgaben der Entwickler ausgewählt werden. Wenn der erste Satz Änderungen von qsmith über mehrere Commits anstelle von einem einzigen verteilt gewesen wäre, hätte jrandom leicht ein Datum oder eine Uhrzeit wählen können, das zwar einige der Änderungen auffängt, aber fälschlicherweise nicht alle.

Bemerkung
Es gibt keinen Grund, warum markierbare Punkte in jrandoms Änderungen als ein einziger Commit an das Archiv gesendet werden müssen - es hat sich in diesen Beispielen einfach so ergeben. Im wirklichen Leben könnte qsmith mehrere Commits zwischen den Markierungen vornehmen. Er kann auch völlig isoliert an dem Zweig arbeiten, wenn es ihm so gefällt. Der Sinn der Marken ist, aufeinander folgende Punkte anzuzeigen, zu denen er die Änderungen im Zweig für mit der Hauptversion verschmelzbar hält. Solange jrandom die Verschmelzung immer mit zwei -j-Optionen vornimmt und darauf achtet, die Zusammenführungsmarkierungen von qsmith in der richtigen Reihenfolge und jeweils nur einmal zu verwenden, sollte die Hauptversion niemals

das Problem mit der mehrfach versuchten Verschmelzung erfahren. Konflikte können immer noch auftreten, es werden aber unvermeidbare sein, solche, bei denen sowohl in der Hauptversion als auch im Zweig Änderungen an denselben Stellen im Quelltext vorgenommen wurden. Diese müssen immer manuell aufgelöst werden.

11 Der koordinierte Ansatz: Verschmelzungen von und zur Hauptversion

Für die an der Hauptversion tätigen Entwickler ist es von Vorteil, wenn häufig die Änderungen aus der abgezweigten Version in die Hauptversion eingearbeitet werden, denn so bekommen sie außer ihren eigenen Änderungen auch noch die Änderungen im Zweig mit. Die Entwickler, die den Zweig bearbeiten, sehen aber nichts von den Veränderungen der Hauptversion und können sie so nicht für ihre eigene Arbeit nutzen beziehungsweise sie darauf abstimmen.

Damit das möglich wird, muss der am Zweig tätige Entwickler ab und zu (soll heißen: wann immer er Lust hat, die letzten Änderungen der Hauptversion zu übernehmen und die unvermeidlichen Konflikte zu behandeln) einen zusätzlichen Schritt ausführen:

```
paste$ cvs update -j HEAD
```

Die reservierte Marke HEAD bezeichnet den Kopf, also die Spitze der Hauptentwicklungslinie. Obiges Kommando übernimmt alle Änderungen aus der Hauptversion, die zwischen der Wurzel des aktuellen Zweigs (Exotic_Greetings-branch) und der jeweils aktuellsten Revision aller Dateien der Hauptversion liegen. Natürlich sollte qsmith danach eine Marke anbringen, damit die Entwickler an der Hauptversion vermeiden können, aus Versehen ihre eigenen Änderungen zu übernehmen, wenn sie die Änderungen von qsmith bekommen wollen.

Der Entwickler des Zweigs kann ebenso die Marken der Hauptversion als Begrenzungen verwenden, was dem Zweig erlaubt, genau die Änderungen zu übernehmen, die zwischen dem letzten Zusammenführen und dem aktuellen Zustand der Hauptversion liegen (genau wie die Hauptversion Zusammenführungen durchführt). Zum Beispiel (angenommen jrandom hat nach dem Verschmelzen mit dem Zweig einige Änderungen an hello.c vorgenommen):

```
user@linux ~/ # emacs hello.c

...

user@linux ~/ # cvs ci -m "clarify algorithm" hello.c

Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v <-- hello.c
new revision: 1.22; previous revision: 1.21
done
```

Dann kann qsmith diese Änderungen in den Zweig einarbeiten, den Commit ausführen und, natürlich, eine Markierung anbringen:

```
user@linux ~/ # cvs -q update -j merged-Exotic_Greetings-1 -j HEAD

RCS file: /usr/local/newrepos/myproj/hello.c,v
retrieving revision 1.21
retrieving revision 1.22
Merging differences between 1.21 and 1.22 into hello.c
```

```
user@linux ~/ # cvs -q update

M hello.c
cvs -q ci -m "merged trunk, from merged-Exotic_Greetings-1 to HEAD"
Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v <-- hello.c
new revision: 1.21.2.1; previous revision: 1.21
done

user@linux ~/ # cvs -q tag merged-merged-Exotic_Greetings-1

T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
```

Beachten Sie, dass jrandom nach dem Commit der Änderungen an hello.c keine Markierung gesetzt hat, qsmith hingegen schon. Das Prinzip dahinter ist, dass man nicht nach jeder winzigen Änderung eine Marke anzubringen braucht. Nach dem Zusammenführen oder nach einem Commit einer Entwicklungslinie zu einem verschmelzbaren Zustand hin jedoch schon! So haben andere Entwickler - möglicherweise an anderen Zweigen - einen Bezugspunkt, um ihre eigenen Zusammenführungen abzustützen.

12 Der Ansatz »fliegender Fisch«: Wie's einfacher geht

Es gibt auch eine einfachere, wenn auch leicht beschränkende Variante des Vorangegangenen. Bei ihr frieren die Entwickler am Zweig ihre Arbeit ein, wenn die Hauptversion eine Verschmelzung durchführt, worauf die Entwickler der Hauptversion einen völlig neuen Zweig abspalten, der dann den alten ersetzt. Die Entwickler des alten Zweiges wechseln zu dem neuen über und fahren mit der Arbeit fort. Dieser Zyklus wird wiederholt, bis es keinen Bedarf mehr für den Zweig gibt. Das Ganze geht in etwa so (in Steno - wir setzen wie immer voraus, dass jrandom@floss die Hauptversion bearbeitet und qsmith@paste den Zweig):

```
user@linux floss/ # cvs tag -b BRANCH-1
user@linux paste/ # cvs checkout -r BRANCH-1 myproj
```

Die Arbeit sowohl an der Hauptversion als auch am Zweig wird aufgenommen; irgendwann beratschlagen sich die Entwickler und beschließen, dass es Zeit ist, die Änderungen aus dem Zweig in die Hauptversion einfließen zu lassen:

```
user@linux paste/ # cvs ci -m "committing all uncommitted changes"
user@linux floss/ # cvs update -j BRANCH-1
```

Alle Änderungen werden aus dem Zweig übernommen; die Entwickler am Zweig unterbrechen ihre Arbeit, während die Entwickler der Hauptversion alle Konflikte auflösen, den Commit vornehmen, eine Markierung setzen und einen neuen Zweig erzeugen:

```
user@linux floss/ # cvs ci -m "merged from BRANCH-1"
user@linux floss/ # cvs tag merged-from-BRANCH-1
user@linux floss/ # cvs tag -b BRANCH-2
```

jetzt schalten die Entwickler des (alten) Zweiges ihre Arbeitskopien auf den neuen Zweig um; sie wissen, dass sie keine noch nicht per Commit bestätigten Änderungen verlieren, denn als die Verschmelzung begonnen wurde, waren sie up-to-date, und der neue Zweig stammt aus einer Hauptversion, welche die Änderungen des alten Zweigs übernommen hat:

```
user@linux paste/ # cvs update -r BRANCH-2
```

Das Ganze wird so endlos fortgesetzt, man muss nur BRANCH-1 durch BRANCH-2 und (vorher) BRANCH-2 durch BRANCH-3 ersetzen.

Ich nenne das die Technik »fliegender Fisch«, denn der Zweig »entspringt« mehrfach der Hauptversion, reist ein kurzes Stück und »taucht« dann wieder in sie ein. Der Vorteil dieses Ansatzes ist, dass er einfach ist (die Hauptversion übernimmt immer alle Änderungen des jeweiligen Zweigs) und dass die Entwickler der Zweigversion nie Konflikte auflösen müssen - sie bekommen einfach jedes Mal einen neuen sauberen Zweig ausgehändigt, an dem sie dann arbeiten. Der Nachteil ist, dass die »abgespaltenen« Entwickler natürlich jedes Mal untätig herumsitzen müssen, während ihre Änderungen in die Hauptversion propagiert werden - und das kann beliebig viel Zeit in Anspruch nehmen, abhängig davon, wie viele Konflikte aufgelöst werden müssen. Ein weiterer Nachteil liegt darin, dass dann viele kleine unbenutzte Zweige herumliegen anstelle von vielen unbenutzten nichtverzweigenden Marken. Wie auch immer - wenn es Sie nicht stört, Millionen winziger, überflüssiger Zweige zu haben und Sie weitgehend reibungslose Zusammenführungen zu schätzen wissen, dann ist »fliegender Fisch« der - was die mentale Buchhaltung angeht - leichteste Weg.

Egal wie Sie nun vorgehen, Sie sollten immer versuchen, die Trennung so kurz wie möglich zu halten. Wenn Zweig und Hauptversion zu lange ohne Verschmelzung laufen, können sie schnell nicht nur unter textuellem, sondern auch unter semantischem Auseinanderdriften leiden. Änderungen, die nur textuell in Konflikt stehen, sind sehr leicht aufzulösen. Änderungen, die auf Grund unterschiedlicher Konzepte in Konflikt stehen, erweisen sich häufig als die am schwersten zu findenden und zu behebenden. Die Abspaltung eines Zweiges, die für die Entwickler so befreiend wirkt, ist auch genau deswegen so gefährlich, da sie beide Seiten von den Auswirkungen durch die Änderungen auf der jeweils anderen Seite abschirmt ... eine gewisse Zeit lang. Wenn Sie Verzweigungen verwenden, wird Kommunikation lebensnotwendig: Jeder muss doppelt sicherstellen, dass er die Pläne der anderen kennt und so programmiert, dass alle auf dasselbe Ziel zusteuern.

Verzweigungen und Schlüsselwortexpansion sind natürliche Feinde

Wenn Ihre Dateien RCS-Schlüsselwörter verwenden, die im Zweig und in der Hauptversion unterschiedlich expandiert werden, werden Sie beinahe unter Garantie bei jedem Versuch, die beiden zusammenzuführen, »unberechtigte« Konflikte erhalten. Selbst wenn überhaupt nichts geändert wurde, überlappen die Schlüsselwörter, und ihre Expansionen passen nicht zueinander. Wenn beispielsweise README.txt in der Hauptversion

$Revision: 1.5 $

enthält, im Zweig hingegen

$Revision: 1.5 $

dann werden Sie beim Verschmelzen folgenden Konflikt bekommen:

```
user@linux ~/ # cvs update -j Exotic_Greetings-branch

RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.tx
rcsmerge: warning: conflicts during merge

user@linux ~/ # cat README.txt

....
<<<<<<< README.txt
key $Revision: 1.5 $
=======
key $Revision: 1.5 $
>>>>>>> 1.14.2.1
 ...
```

Um dies zu verhindern, können Sie die Expansion zeitweise unterdrücken, indem Sie beim Zusammenführen die Option -kk mit übergeben (ich weiß nicht, wofür -kk steht, vielleicht »kill keywords«11?):

```
user@linux ~/ # cvs update -kk -j Exotic_Greetings-branch

RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
```

```
user@linux ~/ # floss$ cat README.txt

 ...
$Revision: 1.5 $
 ...
```

Eines müssen Sie allerdings beachten: Wenn Sie -kk verwenden, wird auch jede andere Schlüsselwortexpansion, die Sie vielleicht für die Datei gesetzt haben, außer Kraft gesetzt. Das ist besonders bei Binärdateien ein Problem, die normalerweise auf -kb gesetzt sind (wodurch jede Schlüsselwortexpansion und jede Zeilenendekonvertierung unterdrückt wird). Wenn Sie also Binärdateien aus dem Zweig in die Hauptversion überführen möchten, benutzen Sie -kk nicht. Behandeln Sie stattdessen die auftretenden Konflikte von Hand.

13 Die Quelltexte Dritter verfolgen: Vendor Branches

Manchmal müssen an einer Software eines externen Lieferanten lokale Änderungen vorgenommen werden, die bei jedem erhaltenen Update der Software aktualisiert werden müssen, nämlich wenn der Lieferant die lokalen Änderungen nicht übernimmt. (Und es gibt eine Menge in Frage kommender legitimer Gründe, warum das nicht möglich ist.)

CVS kann einen bei dieser Aufgabe unterstützen. Der dafür zuständige Mechanismus nennt sich »Vendor Branches«, was soviel heißt wie »abgezweigte Version eines externen Lieferanten«. »Vendor Branches« sind die Erklärung für die (bisher) verwirrenden letzten beiden Optionen, die man bei cvs import angeben kann: vendor tag und release tag; beide habe ich in Kapitel 2 unter den Tisch fallen lassen.

Das Ganze funktioniert so: Der allererste Import gleicht jedem anderen initialisierenden Import eines CVS-Projekts (abgesehen davon, dass Sie besondere Sorgfalt bei der Wahl der Lieferantenmarkierung (»vendor tag«) und der Versionsmarke (»release tag«) walten lassen sollten):

```
user@linux ~/ # pwd

/home/jrandom/theirproj-1.0

user@linux ~/ # cvs import -m "Import of TheirProj 1.0" theirproj Them
THEIRPROJ_1_0

N theirproj/INSTALL
N theirproj/README
N theirproj/src/main.c
N theirproj/src/parse.c
N theirproj/src/digest.c
N theirproj/doc/random.c
N theirproj/doc/manual.txt
No conflicts created by this import
```

Dann checken Sie irgendwo eine Arbeitskopie aus, nehmen Ihre lokalen Anpassungen vor und führen commit auf dem Ergebnis aus:

```
user@linux ~/ # cvs -q co theirproj

U theirproj/INSTALL
```

```
U theirproj/README
U theirproj/doc/manual.txt
U theirproj/doc/random.c
U theirproj/src/digest.
U theirproj/src/main.c
U theirproj/src/parse.c

user@linux ~/ # cd theirproj
user@linux ~/ # emacs src/main.c src/digest.c
user@linux ~/ # cvs -q update

M src/digest.c
M src/main.c

user@linux ~/ # floss$ cvs -q ci -m "changed digestion algorithm; added
comment to main"

Checking in src/digest.c;
/usr/local/newrepos/theirproj/src/digest.c,v <-- digest.c
new revision: 1.2; previous revision: 1.1
done
Checking in src/main.c;
/usr/local/newrepos/theirproj/src/main.c,v <-- main.c
new revision: 1.2; previous revision: 1.1
done
```

Ein Jahr später erreicht uns die nächste Version der Software von Them, Inc., und Sie müssen Ihre lokalen Änderungen darin einbauen. Deren und Ihre Änderungen überlappen ein wenig. Them, Inc. hat eine neue Datei hinzugefügt, einige Dateien geändert, die Sie nicht berührt haben, aber auch zwei Dateien verändert, an denen auch Sie Änderungen vorgenommen haben.

Zunächst müssen Sie einen neuen import durchführen, diesmal von den neuen Quelltexten. Nur wenig ist gegenüber dem ersten Import anders: Sie importieren dasselbe Projekt in das Archiv, mit demselben Vendor Branch. Das Einzige, was sich unterscheidet, ist der Release Tag:

```
user@linux ~/ # floss$ pwd

/home/jrandom/theirproj-2.0

user@linux ~/ # cvs -q import -m "Import of TheirProj 2.0" theirproj Them
THEIRPROJ_2_0

U theirproj/INSTALL
N theirproj/TODO
U theirproj/README
cvs import: Importing /usr/local/newrepos/theirproj/src
C theirproj/src/main.c
U theirproj/src/parse.c
C theirproj/src/digest.c
cvs import: Importing /usr/local/newrepos/theirproj/doc
U theirproj/doc/random.c
U theirproj/doc/manual.txt
2 conflicts created by this import.
Use the following command to help the merge:
 cvs checkout -jThem:yesterday -jThem theirproj
```

Himmel! Nie haben wir CVS so hilfsbereit gesehen. Es sagt uns in der Tat, welches Kommando wir eingeben sollen, um die Änderungen zusammenzuführen. Und was es uns sagt, ist sogar fast richtig! Das angegebene Kommando funktioniert - vorausgesetzt, Sie passen yesterday so an, dass es einen beliebigen Zeitraum bezeichnet, der mit Sicherheit den ersten Import einschließt, aber nicht den zweiten. Ich bevorzuge aber leicht die Methode, die den Release Tag verwendet:

```
user@linux ~/ # cvs checkout -j THEIRPROJ_1_0 -j THEIRPROJ_2_0 theirproj

cvs checkout: Updating theirproj
U theirproj/INSTALL
U theirproj/README
U theirproj/TODO
cvs checkout: Updating theirproj/doc
U theirproj/doc/manual.txt
U theirproj/doc/random.c
cvs checkout: Updating theirproj/src
U theirproj/src/digest.c
RCS file: /usr/local/newrepos/theirproj/src/digest.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into digest.c
rcsmerge: warning: conflicts during merge
U theirproj/src/main.c
RCS file: /usr/local/newrepos/theirproj/src/main.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into main.c
U theirproj/src/parse.c
```

Beachten Sie, dass import uns zwei Konflikte gemeldet hat, merge hingegen scheint nur einen Konflikt zu bemerken. Es scheint, als wäre die Vorstellung eines Konflikts, die CVS beim Importieren hat, leicht abweichend von den übrigen Fällen. Grundsätzlich meldet import einen Konflikt, wenn sowohl Sie als auch der Lieferant zwischen dem letzten Import und dem jetzigen eine Datei verändert hat. Wenn es jedoch an das Zusammenführen geht, dann hält update es mit der normalen Definition von »Konflikt«: überlappende Änderungen. Änderungen, die nicht überlappen, werden auf übliche Art zusammengeführt; die Datei wird dann als geändert markiert.

Ein schneller diff bestätigt, dass nur eine der Dateien Konfliktmarkierungen trägt:

```
user@linux ~/ # cvs -q update

C src/digest.c
M src/main.c

user@linux ~/ # cvs diff -c

Index: src/digest.c
===================================================================
RCS file: /usr/local/newrepos/theirproj/src/digest.c,v
retrieving revision 1.2
diff -c -r1.2 digest.c
```

```
*** src/digest.c 1999/07/26 08:02:18 1.2
-- src/digest.c 1999/07/26 08:16:15
***************
*** 3,7 ****
-- 3,11 ----
 void
 digest ()
 {
+ <<<<<<< digest.c
 printf ("gurgle, slorp\n");
+ =======
+ printf ("mild gurgle\n");
+ >>>>>>> 1.1.1.2
 }
Index: src/main.c
===================================================================
RCS file: /usr/local/newrepos/theirproj/src/main.c,v
retrieving revision 1.2
diff -c -r1.2 main.c
*** src/main.c 1999/07/26 08:02:18 1.2
-- src/main.c 1999/07/26 08:16:15
***************
*** 7,9 ****
-- 7,11 ----
 {
 printf ("Goodbye, world!\n");
 }
+
+ /* I, the vendor, added this comment for no good reason. */
```

Jetzt gilt es nur noch, die Konflikte - wie bei jedem anderen Verschmelzen von Zweigen auch - auszuräumen:

```
user@linux ~/ # emacs src/digest.c src/main.c

 ...

user@linux ~/ # cvs -q update

M src/digest.c
M src/main.c

user@linux ~/ # cvs diff src/digest.c

cvs diff src/digest.c
Index: src/digest.c
===================================================================
RCS file: /usr/local/newrepos/theirproj/src/digest.c,v
retrieving revision 1.2
diff -r1.2 digest.c
6c6
< printf ("gurgle, slorp\n");
--
> printf ("mild gurgle, slorp\n");
```

Und noch ein Commit der Änderungen

```
user@linux ~/ # floss$ cvs -q ci -m "Resolved conflicts with import of
2.0"

Checking in src/digest.c;
/usr/local/newrepos/theirproj/src/digest.c,v <-- digest.c
new revision: 1.3; previous revision: 1.2
done
Checking in src/main.c;
/usr/local/newrepos/theirproj/src/main.c,v <-- main.c
new revision: 1.3; previous revision: 1.2
done
```

und dann auf die nächste Version des Lieferanten warten. (Natürlich sollten Sie überprüfen, ob Ihre lokalen Anpassungen noch funktionieren!)

14 Der bescheidene Guru

Wenn Sie alles in diesem Kapitel gelesen und verstanden haben (besser noch: damit experimentiert haben), können Sie beruhigt davon ausgehen, dass CVS keine unangenehmen Überraschungen mehr für Sie bereithält - zumindest solange niemand größere neue Funktionen einbaut, was mit einer gewissen Regelmäßigkeit geschieht. Alles, was Sie über CVS wissen müssen, um ein richtiges Projekt zu überstehen, wurde angesprochen.

Bevor Ihnen das zu Kopf steigt, lassen Sie mich den Vorschlag wiederholen, den ich erstmals in Kapital 4 gemacht habe, nämlich die Mailingliste info-cvs@gnu.org zu abonnieren. Trotz des kümmerlichen Signal-zu-Rauschen-Abstands, der den meisten Internet-Mailinglisten gemeinsam ist, sind die wenigen Signale, die durchkommen, das Warten fast immer wert. Ich war die ganze Zeit, in der ich dieses Kapitel geschrieben habe, auf der Mailingliste (während der übrigen Kapitel auch), und Sie wären erstaunt, wenn Sie wüssten, wie viele wichtige Details ich über das Verhalten von CVS anhand der Diskussionen gelernt habe. Wenn Sie CVS ernsthaft einsetzen wollen - besonders, wenn Sie CVS-Administrator für eine Entwicklergruppe sind -, können Sie sehr stark von dem gesammelten Wissen der anderen ernsthaften Anwender profitieren.

1. Anm. d. Übers.: wörtlich: Sandkasten, also ein persönlicher Spielplatz, der keinen Einfluss auf die restliche Welt hat.
2. Anm. d. Übers.: Watch, Plural watches: wörtlich Beobachtung, im Sinne von unter Beobachtung stellen
3. Anm. d. Übers.: Auf Deutsch und in Kürze der Inhalt: Die Datei notify bestimmt, wohin Benachrichtigungen geschickt werden. Der erste Eintrag in einer Zeile ist ein regulärer Ausdruck, gegen den das Verzeichnis, in dem es zu Änderungen kommt, getestet wird. Wenn es »passt«, dann wird der Rest der Zeile zum Aufrufen eines Filterprogramms verwendet, das seine Eingabe über die Standardeingabe erhält, wobei %s für den zu benachrichtigenden Benutzer steht. Statt eines regulären Ausdrucks kann auch »ALL« oder »DEFAULT« genommen werden, dann gilt es für alle Verzeichnisse
4. Anm. d. Übers.: In etwa »Nachricht von CVS«
5. Anm. d. Übers.: Die verwendete Log-Nachricht »turned on watch notification« kann man mit »habe die Watch-Benachrichtigung aktiviert« übersetzen.
6. Anm. d. Übers.: Kurze Übersetzung des Dateiinhalts: Die Datei checkoutlist dient dazu, weitere Dateien zu ermöglichen, die Teil des administrativen Bereichs von CVS sind. Der erste Eintrag einer Zeile bezeichnet den Namen der Datei, die aus den korrespondierenden RCS-Dateien in `$CVSROOT/CVSROOT` ausgecheckt werden kann. Der Rest der Zeile beinhaltet die Fehlermeldung, die verwendet wird, wenn die Datei nicht ausgecheckt werden kann. [...]
7. Anm. d. Übers.: Etwa: knapp daneben
8. Anm. d. Übers.: Loslassen, abgeben
9. Anm. d. Übers.: to annotate: anmerken
10. Anm. d. Übers.: Von engl. »differences" - Unterschiede, hier: Änderungen
11. Anm. d. Übers.: Schlüsselwörter ausschalten