CVS – Vortrag

René Scholz – Voland@jena.thur.de

8. Dezember 2000

1 Einführung

CVS ist ein Versions-Kontrollsystem (*Concurrent Versions System*). Im Gegensatz zu VSS können viele Entwickler gleichzeitig und unabhängig voneinander an den gleichen Programmquellen arbeiten.

Die Sourcen werden dabei in einem zentralen Verzeichnis auf dem CVS-Server gehalten, dem **Repository**.

Jeder (berechtigte) Benutzer kann sich vom Repository bzw. Teilen davon mittels des `checkout`-Befehls eine lokale Kopie in seinem **Workspace** erstellen und dort bearbeiten.

Die Veränderungen werden mit dem `commit`-Befehl ins Repository zurückgeschrieben. Veränderungen anderer Benutzer am Repository werden mittels des `update`-Befehls in den lokalen Workspace integriert.

Dabei können sog. Konflikte auftreten, der per Hand aufgelöst werden müssen.

CVS unterstützt **Branches** und erlaubt damit mehrere (parallele) Entwicklungslinien der Software.

Die Funktionalität von CVS kann durch Skripte erweitert werden.

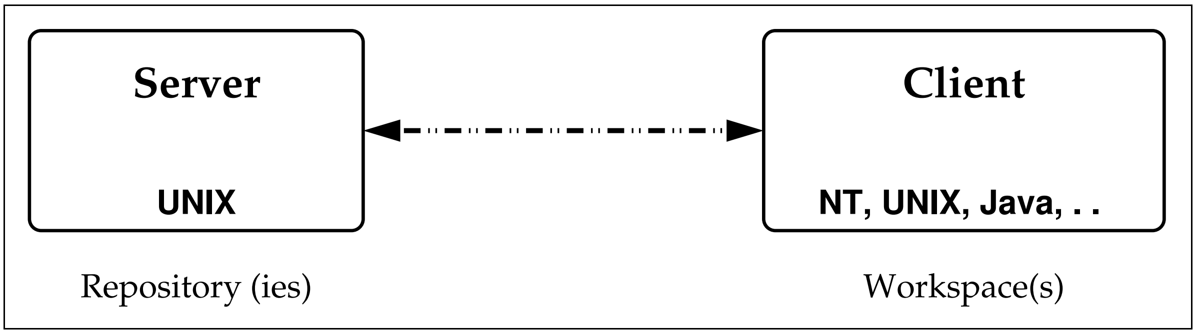
1.1 Was ist CVS *nicht*:

CVS ist kein:

- Ersatz für Software-Management
- Ersatz für Kommunikation der Entwickler
- Bug-Tracking-System
- Entwicklungsystem

2 Architektur

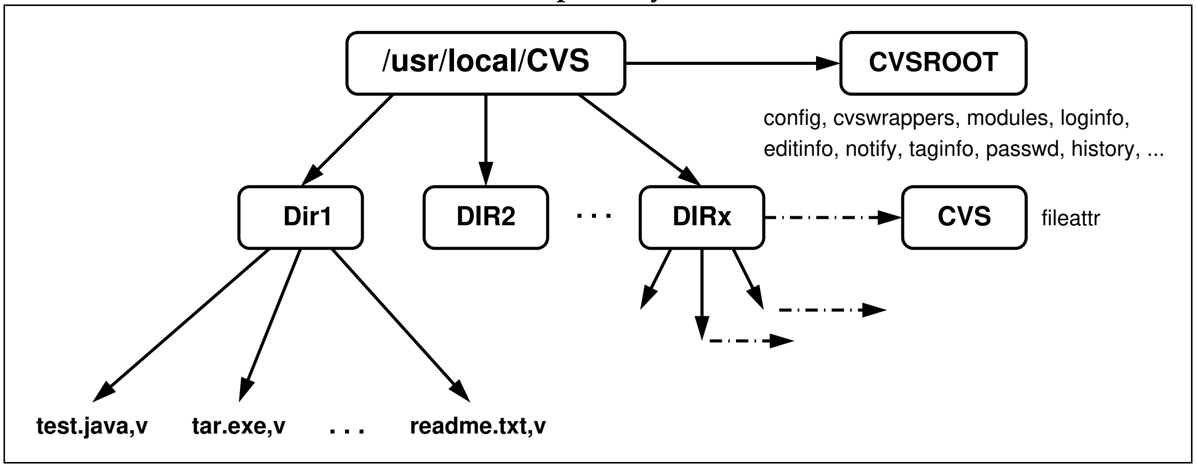
CVS benutzt eine Client-Server-Architektur:



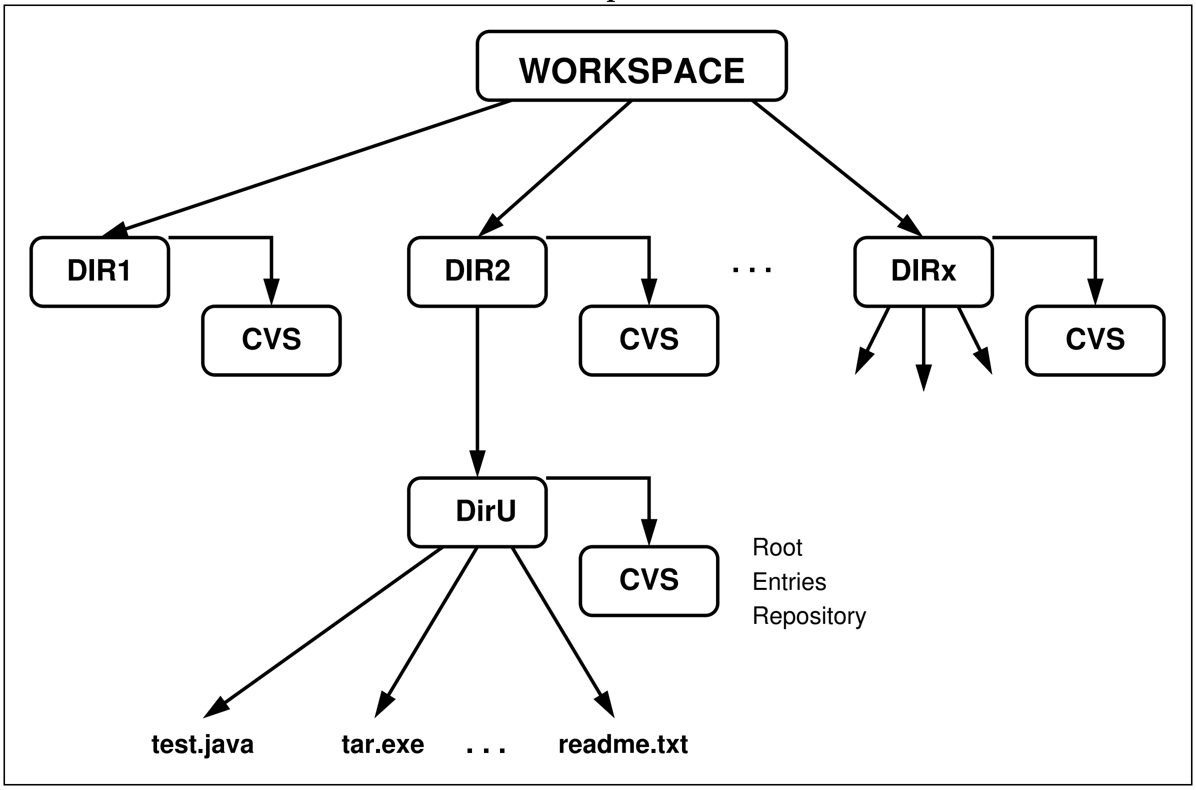
Der CVS-Server befindet sich immer auf einem UNIX-Rechner und wird über den `inetd` gestartet (es gibt inzwischen auch eine frühe Beta der Server-Version für NT, aber es ist generell besser, den Server unter UNIX laufen zu lassen). CVS basiert auf RCS (Dateiformat).

2.1 Verzeichnis-Strukturen

Repository:



Workspace:



CVS (Workspace)

In den `CVS`-Verzeichnissen im Workspace befinden sich *immer* die 3 Dateien `Entries`, `Repository` und `Root`:

- Entries: Hier sind alle Dateien und Unterverzeichnisse im zugehörigen Verzeichnis aufgelistet, Beispiel:

  ```
  D/AS3_Patch_for_SOLARIS7////
  D/doc////
  D/aPages////
  D/hotfixora_ismb2224(ora)////
  D/as_gnu_archive////
  D/as_install////
  D/sac////
  D/src////
  ...
  /QSTART.TXT/1.2/Fri Dec 10 15:44:47 1999//
  /install.eng/1.1.1.1/Fri Dec 10 15:44:48 1999//
  /install.ger/1.1.1.1/Fri Dec 10 15:44:48 1999//
  /install.txt/1.7/Fri Dec 10 16:00:11 1999//
  /read.me/1.1.1.1/Fri Dec 10 15:44:48 1999//

  /readme.txt/1.5/Fri Dec 10 17:45:58 1999//
  ```

- Repository: Gibt die zugehörige Stelle des zugehörigen Verzeichnisses im Repository an, Beispiel:

  ```
  /usr/local/CVSROOT/test/doc
  ```

- Root: Gibt an, wo das Repository zu finden ist (analog Option `-d`), Beispiel:

  ```
  :pserver:cvstest@cv3.zserver.de:/usr/local/CVSROOT
  ```

Es können auch weitere Dateien hier vorhanden sein:

- `Tag`: Hier merkt sich CVS die *sticky* Tags.
- `Notify, Template, ...`

Manche Clients legen hier ebenfalls Dateien an, z. B. jCVS die Datei `jcvs.txt`, in der Meta-Daten über das zugehörigen Projekt gespeichert werden.

2.2 Transportmechanismen

Die Kommunikation zwischen dem Server und einem CVS-Client kann auf unterschiedliche Weisen erfolgen:

- lokal (Client = Server)
- rsh
- ssh
- kerberos

Es ist *keine* ständige Verbindung zwischen Server und Client notwendig, sondern nur für bestimmte CVS-Befehle (`update, commit, add, ...`).
Der Transportmechanismus wird in der Variablen `$CVSROOT` oder mit der Option `-d` angegeben sowie in der Datei `CVS/Root`.
Syntax: `:METHOD:USER@HOSTNAME:/path/to/repository`
Beispiel:

```
:pserver:cvstest2@cv3.zserver.de:/usr/local/CVSROOT
```

3 GUI's und andere Clients

Es gibt verschiedene brauchbare Clients für CVS, sowohl für Unix als auch für andere Betriebssysteme. Das normale CVS-Clientbinary funktioniert auch unter Windows (mit Cygwin), hingegen gibt es dafür noch keinen stabilen CVS-Server.
Hier gibt es eine ausführliche Dokumentation zur Installation verschiedener Clients:
Hier die wichtigsten Vertreter:

- **PCL-CVS**:

  Ist ein (X)Emacs-Paket, welches sich in `dired` einklinkt. Ein CVS-Binary als Client wird benötigt, z. B. aus dem Programm WinCVS.

- **jCVS**:

  Basiert auf Java-1.2, und ist damit auf vielen unterschiedlichen Plattformen lauffähig.
  Lizenz: GPL.

  Screenshot:

.

- **WinCVS**:

  Ist eine native Windows-Applikation. Umfangreicher aber komplizierter als jCVS.
  Kann die unterschiedlichen Zweige (*Branches*) grafisch anzeigen.
  Lizenz: GPL.

  Screenshot:

.

Darüber hinaus gibt es auch mehrere WWW-basierte Interfaces für einen Lesezugriff auf das Repository, z. B. `CVSweb` und `Bonsai` sowie ein `mod_cvs` für den Apache.
Beispiel für CVSweb: `www.htmlwizard.net/cgi-bin/cvsweb.pl`

4 Die CVS-Kommandozeile

Der Syntax für einen CVS-Befehl ist wie folgt:

```
cvs [ cvs_options ] cvs_command [ command_options ] [ command_args ]
```

Dabei gibt es *globale CVS-Optionen*, die vor dem Befehl stehen und Auswirkungen auf alle Befehle haben, und *Kommando-Optionen*, die sich auf den jeweiligen Befehl beziehen.
Viele Kommando-Optionen haben eine gleiche Bedeutung, dieses ist aber nicht zwangsläufig immer der Fall!

Beispiele für Globale Optionen und Kommando-Optionen:

```
$ cvs -q update -P -r 1.5
$ cvs commit -m "Bugs removed" file1.c file2.txt
$ cvs -Q status -l .
$ cvs diff -c -r 1.10 file1.c
```

Eine kurze Hilfe zu einem Befehl gibt es mit cvs -H <Befehl>.
Hilfe zu den globalen Optionen gibt es mit cvs -help-options.
Die Liste aller CVS-Befehle erhält man mit cvs -help-commands.

In der Datei ~/.cvsrc (oder bei anderen Clients auf andere Weise) können Default-Werte für Optionen eingestellt werden, Beispiel:

```
$ cat ~/.cvsrc
cvs -q -z9
update -dAP
diff -u
status -v
checkout -P
```

4.1 Wichtige Globale Optionen

- -d CVS_ROOT_DIRECTORY Angabe des Repositories und des CVS-Servers.
- -e EDITOR Angabe des Editors.
- -H Anzeige der Hilfe zu einem Befehl/Kommando.
- -l Keine Loginformation schreiben.
- -n Befehl nicht ausführen, aber dessen Ausgaben anzeigen.
- -q Quiet.
- -Q Really Quiet.

4.2 Wichtige Kommando-Optionen

- -D DATE_SPEC Verwende die neueste Version, die nicht älter als DATE_SPEC ist.
- -k KFLAG Ändere Keyword-Verwendung. Benutzbar für: add, checkout, diff, import, update.
- -l Nur im lokalen Verzeichnis arbeiten (Default: rekursiv).
- -m MESSAGE Keinen Editor aufrufen, sondern MESSAGE verwenden für Loginformationen. Verwendbar bei: add, commit, import.
- -P Leere Verzeichnisse entfernen.
- -p Dateien (bei checkout und update) auf Standardausgabe schreiben anstelle in die entsprechenden Dateien.

- `-r TAG` Eine bestimmte Version/Revision verwenden, die vorher mit dem Befehl `tag` oder `rtag` definiert wurde. Es gibt symbolische und numerische Tags (sind gleichbedeutend). Die Tag-Angabe ist *sticky* bei den Befehlen `checkout` und `update`.
  Benutzbar für: `checkout, commit, diff, history, export, rdiff, rtag` und `update`.

Weiterhin gibt es spezielle Admin-Optionen mit `cvs admin`.

4.3 Wichtige CVS-Befehle

Hier die am häufigsten benötigten Befehle. Viele dieser Befehle wirken sich aber erst nach einem **commit** auf das Repository aus!
Manche Befehle brauchen eine Netzverbindung zum CVS-Server, einige nicht.
Die genauen Kommando-Optionen erhält man mit `cvs -H <Kommando>`.

init

`cvs -d <directory> init`
Erzeugt ein Repository auf dem Server mit den zugehörigen Dateien in `CVSROOT`. Anschließend muß der CVS-Admin noch Dateirechte ändern und CVS-Logins einrichten.

import

`cvs import [-options] repository vendortag releasetag...`
Importiert ein (sauberes!) Verzeichnis mit Quelldateien in das Repository.

checkout

`cvs checkout [options] modules...`
Erzeugt oder erneuert einen Workspace.

update

`cvs update [-AdflPpR] [-d] [-r tag|-D date] files...`
Bringt einen Workspace auf den aktuellen Stand, indem die Änderungen anderer Benutzer eingefügt werden.
Wenn es neue Verzeichnisse im Repository gibt, muß die Option `-d` benutzt werden!
Mit `-P` werden leere Verzeichnisse nicht in den Workspace eingebracht.

status

`cvs status [-vlR] [files...]`
Zeigt Informationen über den CVS-Status, Tags und weiteres an.

add

`cvs add [-k rcs-kflag] [-m message] files...`
Fügt Dateien zur CVS-Verwaltung hinzu.

remove

`cvs remove [-flR] [files...]`
Entfernt Dateien aus dem Projekt (allerdings nicht vollständig, damit es später wiederhergestellt werden kann).

commit

`cvs commit [-nRlf] [-m msg | -F logfile] [-r rev] files...`
Führt die lokalen Änderungen im Workspace auf dem Repository durch, z. B. nach `add, remove, tag` usw. und bringt auch die Änderungen an den Dateiinhalten ins Repository.

release

`cvs release [-d] directories...`
Löscht einen Workspace, wobei auf ev. Fehler hingewiesen wird.
Ist nicht so gefährlich wie `rm -rf <Workspace>`.

diff

```
cvs diff [-lNR] [rcsdiff-options]
    [[-r rev1 | -D date1] [-r rev2 | -D date2]] [files...]
```
Zeigt Unterschiede zwischen verschiedenen Versionen einer Datei als UNIX-Diff an. GUIs können diesen Output grafisch aufbereiten.

history

`cvs history [-report] [-flags] [-options args] [files...]`
Zeigt an, wer wann was im Repository gemacht hat (`checkouts, commits, rtags, updates, releases`).

log

`cvs log [-lRhtNb] [-r[revisions]] [-d dates] [-s states]`
Zeigt Log-Einträge für Dateien und Verzeichnisse an.

tag

`cvs tag [-lRF] [-b] [-d] [-c] [-r rev|-D date] tag [files...]`
Benennt eine bestimmte Revision des Projektes/Verzeichnisses mit einem lesbaren (und sinnvollen!) Namen.
Mit der Option `-b` werden **Branches** erzeugt.

export

`export [-NRfln] [-r rev | -D date] [-d dir] [-k kopt] module...`
Erzeugt eine Kopie des Projektes ohne die CVS-Dateien, die zum Erstellen von fertigen Distributionen der Software genutzt werden kann.

annotate

`cvs annotate [-lRf] [-r rev|-D date] [files...]`
Zeigt eine Datei an, wobei jeweils die aktuellsten Zeilen angezeigt werden zusammen mit Versionnummer, Datum und Author.

Die meisten Befehle haben auch Kurzformen, z. B. `co` für `checkout` oder `up` für `update` usw.

4.4 Revision Numbers

Jede Datei innerhalb des CVS besitzt eine oder mehrere Revision Numbers (die nichts mit der Versionsnummer der Software zu tun haben). Diese Nummer beginnt meist mit 1.1, die Initial-Revision bei einem Import ist aber immer 1.1.1.1 (die nächste Revision wäre aber wieder 1.2). Bei jeder Änderung wird die letzte Zahl (*minor number*) um 1 erhöht. Eine Erhöhung der Hauptnummer (*major number*) kann man nur per Hand erzwingen.
Anhand der Revisions-Nummer kann man jeden beliebigen Zustand des Projektes wiederherstellen. Bei einem Checkout oder Update des Workspace wird (per default) immer die letzte Revision einer Datei übertragen. Mit der Option `-r` kann man jede beliebige Revision verwenden.

Normalerweise sollten aber vom Nutzer weniger die Revisions-Nummern benutzt werden, sondern es sollten (für sinnvolle Zwischenstände) sog. Tags vergeben werden.

4.5 Tags

Ein Tag ist eine symbolische und lesbare Beschreibung der jeweiligen Revision, Beispiel: `Release-1999_05_01`. Ein Tag wird mit dem Befehl `$ cvs tag` oder `$ cvs -rtag` erzeugt und innerhalb der `,v`-Dateien im Repository gespeichert. Dabei wird der aktuelle Stand der Revisions-Nummern der Dateien im Workspace benutzt. Insbesondere können daher beliebige Revisionen der Dateien verwendet werden, nicht nur die jeweils letzte:

```
        File A      File B      File C      File D      File E
        ------      ------      ------      ------      ------
                                            1.1
                                            1.2
                                            1.3
                                            1.4
                                            1.5
                                            1.6
                                            1.7
                    1.1                     1.8
                    1.2                     1.9
                    1.3                     1.10        1.1
                    1.4                     1.11        1.2
                    1.5                     1.12        1.3
                    1.6                     1.13        1.4
                    1.7         1.1         1.14        1.5
                    1.8         1.2         1.15        1.6
        1.1         1.9         1.3         1.16        1.7
    ----1.2---------1.10--------1.4---------1.17--------1.8------->
        1.3         1.11        1.5         1.18        1.9
                                1.6         1.19        1.10
```

Zur Bezeichnung der Tags sind nur Buchstaben, Zahlen, `-` und `_` erlaubt. Die Bezeichnung sollte möglichst informativ sein, aber auch nicht zu lang.

Man kann zur Identifikation einer bestimmten Version einer Datei wahlweise die Revisions-Nummern oder die jeweiligen Tags verwenden, für CVS macht das keinen Unterschied.

Eine bestimmte Revision eines Projektes (*Snapshot*) kann man dann z. B. folgendermaßen in einem Workspace anlegen: `$ cvs checkout -r Release-1999_05_01 myproj`.
Mit `$ cvs update -r Release-1999_05_01 myproj` kann man einen Workspace auch direkt auf eine bestimmte Revision bringen, bei der dann allerdings der Tag *Sticky* ist, d. h. es können vorerst keine Änderungen an den Dateien im Workspace mehr erfolgen. Mit der Option `-A` bei `update` oder `checkout` kann man die Eigenschaft *Sticky* wieder entfernen.

Man kann einen Tag auch wieder entfernen: `$cvs rtag -d tag modul`. Dabei wird aber nur die symbolische Bezeichnung entfernt, die Revisionen der Dateien bleiben natürlich erhalten.
Der Befehl `rtag` arbeitet direkt auf dem Repository.

4.6 Branches

Mittels der Tags kann man im CVS beliebige Zustände der Vergangenheit wiederherstellen. Um sie auch zu verändern, werden *Branches* benötigt, mit denen der Sourcecode in unterschiedliche Zweige aufgeteilt werden kann, die von einem gemeinsamen Ausgangspunkt ausgehen.

Branches sind z. B. notwendig, wenn man Bugs einer Software fixen will, bei der man über eine ältere stabile Version und eine aktuelle Entwicklerversion verfügt. Der Bug kann zwar in der Entwicklerversion gefixed werden, aber die Software ist zu instabil, um sie an Kunden auszuliefern. Stattdessen wird der Bug bzgl. der älteren Version der Software gefixed.

Zu diesem Zweck wird ein Branch der älteren Version der Software erzeugt und in diesem Branch der Bug gefixed (und das gefixte Paket an den Kunden ausgeliefert).

Achtung: Es ist unbedingt notwendig, daß der Ausgangpunkt des Branches (`*` in der Abb.) mit einem Tag wie z. B. `Release-1999_05_01` versehen wird!

In diesem Branch finden dann auch alle weiteren Bugfixes statt, bis die Entwicklerversion stabil genug zur Auslieferung ist. Dann werden die Änderungen des Branches zur Ausgangsversion in die Entwicklerversion übernommen, für die beiden Zweige findet ein *Merge* statt:

```
            (branch on which bug was fixed)
          .---------------->---------------.
         /                                 |
        /                                  |
       /                                   |
      /                                    |
     /                                     V (<------ point of merge)
====*===================================================================>
            (main line of development - Trunk)
```

Ein solcher Merge kann als Spezialfall eines Updates betrachtet werden, bei dem die Änderungen aus dem Branch kommen, also sowas wie ein größerer Patch (z. B. Linux-Kernel-Patch).

Die Hauptlinie, von welcher der Branch ausgeht, wird als **Trunk** bezeichnet.

Beispiel: Es existiere eine stabile Version mit dem Tag `Release-1999_05_01`.

- Branch erzeugen:

  Zuerst wird ein neuer Workspace angelegt, der genau dem Stand des Tags `Release-1999_05_01` entspricht:
  `$ cvs -q co -d myproj_old_release -r Release-1999_05_01 myproj`

  Dann wird mit der Option `-b` der Branch-Tag erzeugt:

  ```
  $ cd myproj_old_release
  $ cvs -q tag -b Release-1999_05_01-bugfix-branch
  ```

  Da Branch-Tags nicht anders aussehen als normale Tags, ist es sinnvoll, in den Branch-Tag die Bezeichnung `branch` aufzunehmen!

- Workspace updaten:

  Entweder erzeugt man einen neuen Workspace mittels

  ```
  $ cd otherdir
  $ cvs co -d myproj_branch -r Release-1999_05_01-bugfix-branch myproj
  ```

  oder man kann den gerade benutzten Workspace auf den Branch `Release-1999_05_01-bugfix-branch` updaten (es sollten zwischenzeitlich natürlich keine Veränderungen am Workspace stattgefunden haben):

```
$ cvs update -r Release-1999_05_01-bugfix-branch
```

Die Dateien im Workspace könnten dann etwa folgenden Status haben:

```
floss$ cvs -q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.5     Tue Apr 20 06:12:56 1999
   Repository revision:       1.5     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                Release-1999_05_01-bugfix-branch (branch: 1.5.2)
   Sticky Date:               (none)
   Sticky Options:            (none)
floss$ cvs -q status b-subdir/random.c
===================================================================
File: random.c                Status: Up-to-date
   Working revision:  1.2     Mon Apr 19 06:35:27 1999
   Repository revision:       1.2 /usr/local/cvs/myproj/b-subdir/random.c,v
   Sticky Tag:                Release-1999_05_01-bugfix-branch (branch: 1.2.2)
   Sticky Date:               (none)
   Sticky Options:            (none)
```

Die Branch-Tags werden direkt in den `,v`-Dateien im Repository gespeichert.

Die **Branch-Nummer** wird aus der alten Revisions-Nummer der jeweiligen Datei am Verzweigungspunkt gebildet, an welche die nächste freie gerade Zahl ≥ 2 angehängt wird. In obigem Beispiel erhält `hello.c` die Branch-Nummer 1.5.2 und `random.c` die Branch-Nummer 1.2.2. Branch-Nummern haben immer eine ungerade Anzahl von Teilnummern, Revisions-Nummern immer eine gerade.

- Bug fixen:

  Die notwendigen Änderungen an den Dateien werden vorgenommen. Anschließend macht man wie sonst auch einen Update und ein Commit:

```
$ cvs -q update
M hello.c
M b-subdir/random.c
$ cvs ci -m "fixed some old bugs"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v  <--  hello.c
new revision: 1.5.2.1; previous revision: 1.5
done
Checking in b-subdir/random.c;
/usr/local/cvs/myproj/b-subdir/random.c,v  <--  random.c
new revision: 1.2.2.1; previous revision: 1.2
done
```

Dabei werden auch die Revisions-Nummern der Dateien verändert, und zwar werden neue Revisions-Nummern gebildet, die aus der Branch-Nummer gefolgt von der bekannten Minor-Nummer bestehen. Bei Veränderungen der Dateien auf dem Branch wird die Minor-Nummer jeweils incrementiert.

Es ist noch zu beachten, daß zu Beginn, wenn die Dateien auf dem Branch noch nicht verändert wurden, diese noch ihre alte Revisions-Nummer behalten (und nicht etwa 1.5.2.0 o. ä.). Die Initial-Version einer Datei erhält immer die gleiche Revisions-Nummer wie die Datei auf dem Trunk.

Weiterhin ist zu beachten, daß beide Revisions-Nummern 1.5.2.1 und 1.2.2.1 zum selben Branch gehören, d. h. die Zahlen sagen nichts darüber aus, zu welchem Branch eine Datei gehört. Daher wird (fast) immer der Branch-Tag für die Angabe des gewünschten Branches für die CVS-Befehle genutzt.

Es ist auch möglich, Sub-Branches anzulegen, dies sollte aber möglichst vermieden werden, um die Übersicht zu behalten.

- Änderungen im Branch in den Trunk zurück mergen:

  Nachdem nun einige Zeit am Hauptentwicklungszweig (dem Trunk) und dem Branch gearbeitet wurde, wird man die Bugfixes irgendwann in den Trunk übernehmen wollen. Dazu wechselt man wieder zum Hauptentwicklungszweig:

  `$ cvs -q update -A`.

  Jetzt kann man sich noch die gemachten Änderungen des Branches im Vergleich zum Trunk anzeigen lassen:

  `$ cvs -q diff -c -r Release-1999_05_01-bugfix-branch`.

  Um die Änderungen des Branches in den aktuellen Workspace zu übernehmen, wird die Update-Option `-j` (wie *join*) benutzt:

  `$ cvs -q update -j Release-1999_05_01-bugfix-branch`.

  Dabei können natürlich wie sonst auch Konflikte auftreten, die aufgelöst werden müssen (siehe 5.7).

- Mehrfaches Mergen:

  Oft wird auf einem Branch weitergearbeitet (weitere Bugfixes), nachdem bereits ältere Bugfixes vom Branch in den Trunk zurück gemerged wurden.

  Anschließend möchte man die neuen Änderungen ebenfalls wieder in den Trunk einbringen. Wenn man dabei wie im letzten Beispiel verfahren würde, also:

  `$ cvs -q update -j Release-1999_05_01-bugfix-branch`

  würden Konflikte auftreten. Der Grund besteht darin, daß CVS dabei versucht, auch die bereits vorher gemachen Änderungen (seit dem Tag `Release-1999_05_01-bugfix-branch`) zu mergen. Man darf daher nur die neuen Änderungen nach dem letzten Merge einbringen.

  Dies kann man mit zwei Methoden erreichen:

  1. Verwendung eines Tags

  Der empfohlene Weg ist die Vergabe eines Tags nach jedem Bugfix, nachdem der Bug im Workspace gefixed und commited ist:

```
$ cvs -q tag Release-1999_05_01-bugfix-branch-fix1 .
```

Um diesen Bugfix dann in den Trunk zu mergen, benutzt man die Option `-j` zweimal:

```
$ cvs -q update -j Release-1999_05_01-bugfix-branch-fix1 \
                -j Release-1999_05_01-bugfix-branch .
```

2. Verwendung eines Datums

Man kann auch ein Datum benutzen, wenn man den Zeitpunkt kennt, an dem der Bugfix erstellt wurde:

```
$ cvs -q update -j "Release-1999_05_01-bugfix-branch: 2 days ago" \
                -j Release-1999_05_01-bugfix-branch
```

Hierbei werden nur Änderungen im Branch *nach* dem angegebenen Datum beim Merge berücksichtigt.

(es gibt verschiedene Syntaxe zur Zeitangabe im CVS).

Tips zum Arbeiten mit Branches:

- Man kann einen Branch-Tag (oder auch einen normalen Tag) direkt im Repository erzeugen (also ohne einen Workspace zu benutzen):

  ```
  $ cvs rtag -b -r Release-1999 Release-1999-bugfix-branch myproj.
  ```

- Es sollten nicht zu viele *aktive* Branches benutzt werden, da dadurch die Wahrscheinlichkeit für Konflikte mit dem Trunk steigt. Ein Branch sollte in den Trunk zurück gemerged werden, sowie er stabil genug ist.
- Nach Möglichkeit sollten keine Sub-Branches oder noch tiefere Verzweigungen benutzt werden.
- Die Tags sollten konsistent benannt werden und ihre Bedeutung und ihr Bezug zu anderen Tags bzw. Branches anhand dieser Bezeichnung erkennbar sein.
- Die Keyword-Verwendung kann bei Branches zu Konflikten führen (z. B. `$Revision$`).

The Flying Fish Approach:

Für die Entwickler die auf dem Branch arbeiten ist es wichtig, daß sie die Änderungen auf dem Trunk regelmäßig in den Branch übernehmen.
Dazu ist die sog. *Flying Fish* Vorgehensweise am einfachsten und sichersten.

Hierbei wird der Branch nach einiger Zeit in den Trunk zurückgeführt, d. h.

- die Entwicklung auf dem Branch wird eingestellt,
- der Branch in den Trunk gemerged und
- sofort anschließend ein neuer Branch erzeugt.

Die Entwickler auf dem Branch wechseln dann zu diesem neuen Branch und arbeiten dort normal weiter.
Dieser Zyklus wird fortgesetzt, bis keine Branches mehr benötigt werden.

Beispiel: Sei T der/die Entwickler auf dem Trunk und B der/die Entwickler auf dem Branch.

Branch erzeugen und auf den Branch wechseln (B):

```
T$ cvs tag -b BRANCH-1
B$ cvs checkout -r BRANCH-1 myproj
```

Jetzt wird auf dem Trunk und Branch normal gearbeitet. Irgendwann beschließen T und B, daß nun die unterschiedlichen Entwicklungszweige zusammengeführt werden können.
Der Entwickler B commited seine Änderungen und anschließend merged T diese dann vollständig in den Trunk:

```
B$ cvs ci -m "committing all uncommitted changes"
T$ cvs update -j BRANCH-1
```

Nun wartet B solange, bis T alle aufgetretenden Konflikte aufgelöst hat und den commit durchführt sowie einen neuen Tag und einen neuen Branch erzeugt hat:

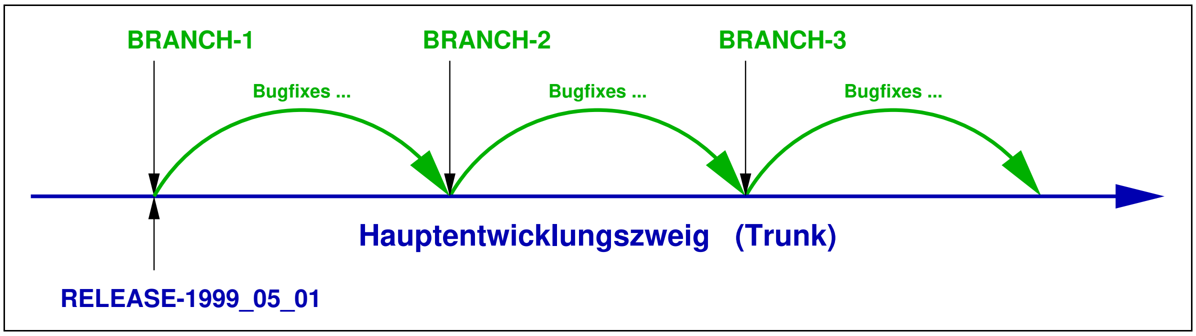
```
T$ cvs ci -m "merged from BRANCH-1"
T$ cvs tag merged-from-BRANCH-1
T$ cvs tag -b BRANCH-2
```

Jetzt wechselt B seinen Workspace auf den neuen Branch (in dem alle vorher von B gemachten Änderungen enthalten sind):

```
B$ cvs update -r BRANCH-2
```

Dieser Zyklus wird immer weiter fortgesetzt, wobei BRANCH-1 mit BRANCH-2 und BRANCH-3 usw. ersetzt wird.

Der Name *Flying Fish* wird in folgender Abbildung verdeutlicht:



Vorteile dieses Verfahrens: Alle Änderungen der Branches gelangen wieder in den Hauptzweig, die Entwickler auf dem Branch müssen keine Konflikte auflösen. Der Hauptnachteil: Die Entwickler auf dem Branch müssen in der Zeit, in der die Merges stattfinden, warten (allerdings sind die Rollen von T und B oft in denselben Personen vereint).

Allgemein kann man sagen, daß die Trennung der Branches vom Hauptzweig nicht zu lange dauern sollte, sonst entstehen nicht nur textuelle Konflikte, sondern es entstehen auch semantische Unterschiede in den Quellcodes.

4.7 Text vs. Binär

Dateien können entweder *Binär* oder als *Text* im Repository gespeichert werden.

Die *Keyword-Expansion* (`$Author$, $Date$, $Id$` usw.) sowie ein `diff` funktioniert nur bei Textdateien.

Beispiele für Keyword-Expansions:

```
$Id$       ==> $Id: install.txt,v 1.11 2000/01/27 16:18:42 cvstest Exp $
$Author$   ==> $Author: cvstest $
$Date$     ==> $Date: 2000/01/27 16:18:42 $
$Revision$ ==> $Revision: 1.11 $
```

Es werden bei den Textdateien immer die Zeilenenden des jeweiligen Betriebssystems benutzt, d. h. die Dateien werden auf dem UNIX-Server ohne CR gespeichert. Beim `checkout` auf einen NT-Rechner wird dort aber CRLF benutzt.

Binäre Dateien müssen als solche gekennzeichnet werden, *bevor* sie ins Repository übertragen werden:

```
cvs add -kb test.exe
cvs ci -m "Binary Testfile added" test.exe
```

Man kann aber mittels der Datei `cvswrappers` in CVSROOT globale Dateitypen festlegen, die binär sein sollen, sowie ob ein Update einer Datei per COPY oder MERGE erfolgen soll.

5 Beispiel-Session

1. **init**
   `$ cvs init -d /export/CVS`
   Wird auf dem CVS-Server aufgerufen (vom CVS-Admin) und erzeugt dort ein Repository im Verzeichnis `/export/CVS`.

2. **import**
   `$ cvs import -m"Importing from vendor VENDOR" test VENDOR VENDOR_0`
   Importiert das Verzeichnis `test` vom "Hersteller" VENDOR ins Repository. Sollte besser auf dem CVS-Server selbst aufgerufen werden, damit keine Fehler bei der Zeilenendenbehandlung o. ä. auftreten, ist aber auch mit Clients möglich.
   Bestimmte Dateitypen werden dabei ignoriert (ist konfigurierbar).

3. **checkout**
   Bevor man mit dem Arbeiten anfangen kann, muß erst ein Projekt in den Workspace gebracht werden:

   ```
   $ cd WORKSPACE
   $ cvs checkout MYPROJECT
   ```

   Hiermit wird das gesamte MYROJECT-Projekt in das aktuelle Verzeichnis `WORKSPACE` kopiert bzw. dort erneuert, falls es schon vorhanden ist.

   Anschließend können die Dateien im Projekt editiert werden:

4. **verändern des Workspace**
   Dazu wechselt man ins Projekt-Verzeichnis und editiert die gewünschten Dateien:

```
$ cd MYPROJECT/.../doc/Solaris
$ emacs *.txt &
$ ...
```

5. **Status anzeigen**
   Man kann sich den aktuellen Status aller Dateien im Workspace anzeigen lassen:

```
$ cvs -Q -l status .
? dummy.file
? new-doc.txt
===================================================================
File: install.eng       Status: Up-to-date

   Working revision:    1.1.1.1
   Repository revision: 1.1.1.1 /export/CVS/test/doc/Solaris/install.eng,v

===================================================================
File: install.ger       Status: Up-to-date

   Working revision:    1.1.1.1
   Repository revision: 1.1.1.1 /export/CVS/test/doc/Solaris/install.ger,v

===================================================================
File: install.txt       Status: Locally Modified

   Working revision:    1.9
   Repository revision: 1.9     /export/CVS/test/doc/Solaris/install.txt,v

===================================================================
File: read.me           Status: Up-to-date

   Working revision:    1.1.1.1
   Repository revision: 1.1.1.1 /export/CVS/test/doc/Solaris/read.me,v

===================================================================
File: readme.txt        Status: Needs Merge

   Working revision:    1.5
   Repository revision: 1.6     /export/CVS/test/doc/Solaris/readme.txt,v
```

   Der Status `Needs Merge` zeigt an, daß in der Zwischenzeit (seit dem letzten Checkout) bereits Änderungen an dieser Datei vorgenommen wurden (und sich jetzt im Repository befinden), die noch mit der lokalen Version zusammengefügt werden müssen. Dabei kann prinzipiell ein Konflikt auftreten.

   Die mit `?` markierten Dateien sind dem CVS unbekannt und müssen erst mit `cvs add` eingefügt werden, falls notwendig.

   Eine weitere Möglichkeit, einen schnellen Überblick über den aktuellen Zustand zu erhalten, ist die Verwendung der Option `-n` beim Update-Befehl:

```
$ cvs -nq update.
```

   Dabei werden folgende Zeichen für die verschiedenen Zustände angezeigt:

? → Datei ist dem CVS unbekannt.
A → Datei lokal per `add` hinzugefügt, aber noch nicht committed.
M → Datei lokal verändert.
C → Datei hat einen Konflikt.
R → Datei lokal per `rm` entfernt, aber noch nicht committed.
U → Datei wurde im Repository verändert seit dem letzten Update.

6. **diff**
   Die Unterschiede zwischen den existierenden Versionen einer Datei kann man mit dem `diff`-Befehl anzeigen:

```
$ cvs diff install.txt
Index: install.txt
===================================================================
RCS file: /export/CVS/test/doc/Solaris/install.txt,v
retrieving revision 1.9
diff -r1.9 install.txt
1,2c1
< bla bla ...
<
---
> Langweilig!
```

   Wenn keine spezielle Revision der Datei verlangt wird, zeigt CVS den Diff der Datei im Workspace gegenüber der letzten Version im Repository an.

   Wenn auf dem Server GNU-diff installiert ist, kann man auch die Option `-c` für `diff` benutzen:

```
$ cvs -Q diff -c -r 1.4 -r 1.5 install.txt
Index: install.txt
===================================================================
RCS file: /export/CVS/test/doc/Solaris/install.txt,v
retrieving revision 1.4
retrieving revision 1.5
diff -c -r1.4 -r1.5
*** install.txt 1999/12/10 17:15:08     1.4
--- install.txt 1999/12/10 17:23:52     1.5
***************
*** 1,4 ****
! eine Zeile die zu einem Konflikt fuehren soll!


  ETWAS NEUES HIER OBEN!
--- 1,4 ----
! ich will hier aber was anderes!


  ETWAS NEUES HIER OBEN!
```

7. **update und Konflikte auflösen**
   ACHTUNG: *Bevor* man nun die Änderungen ins Repository zurückschreibt, muß man erst ein Update des Workspace durchführen, um Änderungen anderer Entwickler zu erhalten und mögliche Konflikte zu erkennen:

   ```
   $ cvs update
   ? dummy.file
   ? new-doc.txt
   cvs server: Updating .
   M install.txt
   RCS file: /export/CVS/test/doc/Solaris/readme.txt,v
   retrieving revision 1.5
   retrieving revision 1.6
   Merging differences between 1.5 and 1.6 into readme.txt
   rcsmerge: warning: conflicts during merge
   cvs server: conflicts found in readme.txt
   C readme.txt
   ```

   Dabei ändert sich nur der Status der Datei `readme.txt`:

```
===================================================================
File: readme.txt        Status: File had conflicts on merge

   Working revision:    1.6
   Repository revision: 1.6     /export/CVS/test/doc/Solaris/readme.txt,v
```

   Der hier entstandene Konflikt muß erst aufgelöst werden. Dazu wird die Datei `readme.txt` erneut in den Editor geladen (da sie durch den Update verändert wurde!):

   ```
   <<<<<<< readme.txt
   *********************************
   =======
   # ------------------------------
   >>>>>>> 1.6
   ```

   Die Trenner <<<<<<< und >>>>>>> zeigen Anfang und Ende des Konfliktes an. Mittels ======= werden die beiden Versionen voneinander getrennt. Die lokalen Änderungen befinden sich oben, die neuen Änderungen aus dem Repository unten.

   Wir ändern nun diesen Konflikt so ab:

   ```
   # eigentlich wird diese Zeile nicht benötigt.
   ```

   und entfernen auch die Konfliktmarkierungen.

8. **commit**
   Anschließend kann man mittels dem Befehl `commit` die Änderungen ins Repository übertragen. Dabei wird `$EDITOR` aufgerufen, um Einträge in die History-Datei des Repository vorzunehmen:

```
$ cvs commit
cvs commit: Examining .
? new-doc.txt
? dummy.file
Checking in install.txt;
/export/CVS/test/doc/Solaris/install.txt,v  <--  install.txt
new revision: 1.10; previous revision: 1.9
done
Checking in readme.txt;
/export/CVS/test/doc/Solaris/readme.txt,v  <--  readme.txt
new revision: 1.7; previous revision: 1.6
done
```

Mit der Option `-m` kann man die Informationen für die History-Datei direkt angeben, ohne extra einen Editor aufzurufen.

Jetzt sind alle Dateien im Workspace auf dem neuesten Stand:

```
$ cvs -Q -l status . | grep Status
File: install.eng       Status: Up-to-date
File: install.ger       Status: Up-to-date
File: install.txt       Status: Up-to-date
File: read.me           Status: Up-to-date
File: readme.txt        Status: Up-to-date
```

9. **add** und **remove**
   Die Dateien `new-doc.txt` und `dummy.file` sollen nun ins CVS aufgenommen werden:

```
$ cvs add -m "Neue Doc-Datei" new-doc.txt
cvs server: scheduling file ‘new-doc.txt’ for addition
cvs server: use ’cvs commit’ to add this file permanently

$ cvs add -m "nur zum Testen" dummy.file
cvs server: scheduling file ‘dummy.file’ for addition
cvs server: use ’cvs commit’ to add this file permanently
```

Der Status dieser beiden Dateien ist anschließend wie folgt:

```
===================================================================
File: dummy.file        Status: Locally Added

   Working revision:    New file!
   Repository revision: No revision control file

===================================================================
...
===================================================================
File: new-doc.txt       Status: Locally Added
```

```
    Working revision:    New file!
    Repository revision: No revision control file

===================================================================
```

Nun teilen wir diese Änderungen dem Repository mit:

```
$ cvs commit -m "2 neue Dateien"
cvs commit: Examining .
RCS file: /export/CVS/test/doc/Solaris/dummy.file,v
done
Checking in dummy.file;
/export/CVS/test/doc/Solaris/dummy.file,v  <--  dummy.file
initial revision: 1.1
done
RCS file: /export/CVS/test/doc/Solaris/new-doc.txt,v
done
Checking in new-doc.txt;
/export/CVS/test/doc/Solaris/new-doc.txt,v  <--  new-doc.txt
initial revision: 1.1
done
```

Um ein neues Verzeichnis ins Repository einzufügen, reicht:

`$ mkdir subdir; cvs add subdir.`

Das Löschen von Dateien geschieht folgendermaßen:

```
$ rm dummy.file
$ cvs rm dummy.file
cvs server: scheduling ‘dummy.file’ for removal
cvs server: use ’cvs commit’ to remove this file permanently

$ cvs status dummy.file
===================================================================
File: no file dummy.file                Status: Locally Removed

   Working revision:    -1.1
   Repository revision: 1.1     /export/CVS/test/doc/Solaris/dummy.file,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

$ cvs commit -m "Dummy-Datei wieder gelöscht"
cvs commit: Examining .
Removing dummy.file;
/export/CVS/test/doc/Solaris/dummy.file,v  <--  dummy.file
new revision: delete; previous revision: 1.1
done
```

Nach dem Löschen befindet sich die Datei nicht mehr im Workspace. Im Repository wird die gelöschte Datei in das spezielle Verzeichnis `Attic` bewegt, das erst beim Löschen einer Datei angelegt wird:

```
$ cvs status dummy.file
===================================================================
File: no file dummy.file                Status: Up-to-date

   Working revision:    No entry for dummy.file
   Repository revision: 1.2     /export/CVS/test/doc/Solaris/Attic/dummy.file,v
```

Solange noch kein `commit` durchgeführt wurde, kann man ein `cvs add` durch ein `cvs rm` wieder rückgängig machen und umgekehrt (dabei sollte man natürlich die Datei `nicht` lokal entfernen).
Man kann mit `cvs rm -f` das lokale Löschen auch gleich durch das CVS erledigen lassen.

10. **Löschen von Verzeichnissen**
    Verzeichnisse werden vom CVS nicht mit-versioniert, Verzeichnisse werden daher auch nicht wirklich gelöscht.

    Als Ersatz werden alle Dateien in einem Verzeichnis gelöscht (`cvs rm -f *; cvs commit`) und dann bei einem Update die Option `-P` (Prune) benutzt, bei der leere Verzeichnisse aus dem Workspace entfernt bzw. nicht angelegt werden.

11. **Umbenennen von Dateien**

    Das Umbenennen einer Datei erfolgt wie folgt:

```
$ mv oldname newname
$ cvs rm oldname
$ cvs add newname
$ cvs ci -m "renamed oldname to newname" oldname newname
```

Das Umbenennen von Verzeichnissen geschieht folgendermaßen:

```
$ mkdir newdir
$ cvs add newdir
$ mv olddir/* newdir
mv: newdir/CVS: cannot overwrite directory
$ cd olddir
$ cvs rm foo.c bar.txt
$ cd ../newdir
$ cvs add foo.c bar.txt
$ cd ..
$ cvs commit -m "moved foo.c and bar.txt from olddir to newdir"
$ cvs update -P
```

Das Umbewegen an andere Stellen ist komplizierter, daher sollte die Verzeichnis-Struktur frühzeitig festgelegt werden, damit solche Umbenennungen nur selten erforderlich sind.

Im Notfall kann man auch Änderungen direkt im Repository durchführen.

12. **Löschen eines Workspace**

    Im Normalfall kann man seinen Workspace einfach mittels `rm -rf` entfernen. Dabei kann es allerdings leicht passieren, daß man einige geänderte Dateien übersieht und daher deren Änderungen verlieren würde.

    Aus diesem Grund gibt es den CVS-Befehl **release**, der den Workspace daraufhin überprüft:

    ```
    $ cvs release myproject
    M Readme
    You have [1] altered files in this repository.
    Are you sure you want to release directory 'myproject':
    ```

    Hier sollte man jetzt `n` eingeben und erst die gemachten Änderungen committen.

    Mit der Option `-d` kann man den Workspace dann auch entfernen lassen, wenn man `y` eingibt:

    ```
    $ cvs release -d myproject
    You have [0] altered files in this repository.
    Are you sure you want to release (and delete) directory 'myproject': y
    $ ls
    $
    ```

6 Watches und Edits

Watches sind ein «Ersatz» für Locks. Normalerweise wissen die Entwickler nichts von den Vorgängen in den Workspaces der anderen Entwickler bis zum Zeitpunkt eines Commits.
Mittels Watches kann man die Kommunikation der Entwickler verbessern und Benachrichtigungen (Notifies) austauschen, wer zu einem bestimmten Zeitpunkt welche Dateien bearbeitet. Die Benachrichtigung erfolgt gewöhnlich per Email, es sind aber auch andere Formen möglich.
Watches müssen in `CVSROOT` in der Datei `notify` aktiviert bzw. eingestellt werden, z. B.:

```
ALL mail %s -s "CVS notification"
```

Die Entwickler müssen einige zusätzliche Schritte während eines checkouts/updates/commits durchführen:

- Sie müssen dem CVS mitteilen, welche Dateien sie beobachten wollen, damit CVS ihnen eine Nachricht zuschicken kann, wenn jemand diese Dateien zu editieren beginnt.
- Sie müssen dem CVS mitteilen, wenn sie selbst eine Datei zu editieren beginnen, damit CVS Benachrichtigungen an alle Entwickler verschicken kann, die diese Dateien beobachten.

Dazu werden die Kommandos `cvs watch add, cvs watch remove, cvs edit` und `cvs unedit` benutzt.
Für jede Datei existiert eine Watch-Liste (bzw. wird bei Bedarf angelegt). Dies ist die Datei `CVS/fileattr` im Repository. Auf dieser Liste können (für jeden Nutzer einzeln) die Aktionen `edit,unedit` und `commit` vermerkt sein (und damit beobachtet werden).
Weiterhin gibt es noch die Kommandos `cvs watch on` und `cvs watch off`.

Beispiel: Die beiden Entwickler A und B haben in ihren Workspaces das gleiche Projekt `myproject` ausgecheckt.

Watch anlegen:

```
A$ cvs -q co myproject
...
A$ cd myproject
A$ cvs watch add hello.c
```

Hiermit fordert A das CVS auf, ihn zu benachrichtigen, wenn jemand die Datei `hello.c` editiert, d. h. der Nutzer A wird auf die Watch-Liste der Datei `hello.c` eingetragen, und zwar (per default) für *alle 3* Aktionen `edit`, `unedit` und `commit`.
Wenn sich A z. B. nur über die Aktion `commit` benachrichtigen lassen will, wird er die Option `-a` benutzen:

```
A$ cvs watch add -a commit hello.c
```

Edit-Vorgang starten:
Der einzige Weg für CVS zu erkennen, wann jemand eine Datei editiert, besteht in der Verwendung des Kommandos `cvs edit`, d. h. der Nutzer B muß dieses Kommando aufrufen:

```
B$ cvs -q co myproject
...
B$ cd myproject
B$ cvs edit hello.c
B$ emacs hello.c
...
```

Wenn B das Kommando `cvs edit hello.c` eingibt, überprüft CVS die Watch-Liste dieser Datei und erkennt, daß A auf dieser Liste steht (bzgl. dieser Aktion `edit`) und schickt diesem dann eine entsprechende Nachricht, etwa:

```
From: B
Subject: CVS notification
To: A
Date: Sat, 17 Jul 1999 22:14:43 -0500

myproject hello.c
--
Triggered edit watch on /usr/local/newrepos/myproject
By B
```

Bemerkung: Es wird hier immer der `From:` des zugehörigen UNIX-Logins des CVS-Nutzers benutzt. Wenn sich mehrere CVS-Nutzer einen UNIX-Login teilen, erscheinen Benachrichtigungsmails dieser Nutzer immer mit dem gleichen `From:` dieses UNIX-Logins. In der letzten Zeile steht aber immer der CVS-Nutzername.
Dies kann vermieden werden, wenn jeder CVS-Nutzer auch einen eigenen UNIX-Login auf dem CVS-Server besitzt (auch per NIS möglich!).

Änderungen committen:
Nachdem B nun die Datei `hello.c` fertig editiert hat, commited er seine Änderungen:

```
B$ emacs hello.c
B$ ...
B$ cvs ci hello.c
```

Dabei wird wieder eine Benachrichtigungsmail an den Nutzer A verschickt, da dieser auch mit der Aktion `commit` auf der Watchliste der Datei `hello.c` steht:

```
From: B
Subject: CVS notification
To: A
Date: Sat, 17 Jul 1999 22:46:22 -0500

myproject hello.c
--
Triggered commit watch on /usr/local/newrepos/myproject
By B
```

Ein `cvs unedit hello.c` ist nicht notwendig, da der Commit den Edit-Vorgang bereits beendet hat. Bei einem zusätzlichen `cvs unedit` geschieht nichts (es werden auch keine Mails verschickt).

Edit-Vorgang beenden:
Man kann einen solchen Edit-Vorgang auch ohne ein Commit beenden, indem man `cvs unedit` benutzt. Dabei bietet CVS dem Nutzer an, die bisher (seit `cvs edit`) gemachten Änderungen zu verwerfen:

```
B$ cvs unedit hello.c
hello.c has been modified; revert changes? y
B$
```

Bei der Antwort `y` werden alle Änderungen verworfen und der Zustand der Datei vor `cvs edit` wiederhergestellt. Gleichzeitig werden Benachrichtigungsmails an alle Watchers der Datei (bzgl. der Aktion `unedit`) verschickt.
Bei der Antwort `n` geschieht gar nichts.

Watch entfernen:
Wenn ein Nutzer keine Benachrichtigungen bzgl. einer Datei mehr erhalten möchte, kann er sich mit `cvs watch remove` wieder von der Watchliste dieser Datei entfernen:

```
A$ cvs watch remove hello.c
```

Watchers und Editors anzeigen:
Man kann jederzeit alle Watcher oder Editors einer Datei bzw. Verzeichnisses anzeigen, auch wenn man nicht selbst auf einer Watchlist steht:

```
A$ cvs watchers
doc/Solaris/hello.c A edit    unedit  commit
        B   commit
doc/Solaris/readme.txt  B   tedit   tunedit tcommit
```

Hierbei sind die Nutzer A und B auf der Watchlist der Datei `hello.c` eingetragen, wobei A alle Aktionen beobachtet, B hingegen nur `commit`.
Der Nutzer B ist zusätzlich noch mit *temporären* Watches auf der Watchlist der Datei `readme.txt`. Temporäre Watches bestehen wenn man `cvs edit` aufruft (sa. 6.1).

6.1 Spezielles zu Watches und Edits

- **Temporäre Watches:**

  CVS geht davon aus, daß ein Editor (jemand, der `cvs edit` aufruft), zumindest temporär auf die Watchliste dieser Datei gehört, da ein anderer Nutzer ebenfalls `cvs edit` aufrufen könnte und eine entsprechende Benachrichtigung daher sinnvoll ist.

  Weiterhin geht CVS davon aus, daß der Nutzer nach dem Ende des Edit-Vorgangs wieder von der Watchliste entfernt werden möchte.

  CVS macht die Annahme, daß mit dem Aufruf des Commit-Befehls ein Edit-Vorgang beendet wird.

  Mit dem Beginn und Ende eines Edit-Vorgangs wird man also vom CVS automatisch auf die Watchliste der entsprechenden Datei gesetzt bzw. entfernt; der Nutzer ist ein *temporary watcher*. Die entsprechenden Aktionen auf der Watchliste heißen dabei: **tedit, tunedit, tcommit**.

  Falls ein Nutzer nicht nur temporär beobachten möchte, muß er den Befehl `cvs watch add` ausführen.

- **Auswahl der Aktionen:**

  Die zu beobachtenden Aktionen können einzeln hinzugefügt oder entfernt werden, Beispiele:

  ```
  $ cvs watch add -a commit hello.c
  $ cvs watch add -a edit -a commit hello.c
  $ cvs watch remove -a commit hello.c
  $ cvs edit -a none README.txt
  ```

- **watch on / off: Erzwingen eines Edits:**

  Aus dem letzen Abschnitt wird klar, daß Benachrichtigungen (außer bei `commit`) nur dann verschickt werden, wenn der entsprechende Nutzer `cvs edit` aufruft. Dieser ist aber nicht dazu gezwungen bzw. hat ev. vergessen, daran zu denken.

  Um einen Nutzer daran zu erinnern, `cvs edit` zu benutzen, existieren die Befehle `cvs watch on` und `cvs watch off`. Durch den Aufruf von `cvs watch on file` wird die Datei `file` beim Anlegen eines Workspaces durch einen Nutzer (mittels `cvs co`) mit **read-only** Dateirechten versehen. Der Nutzer kann dann diese nicht direkt speichern und wird dadurch auf das notwendige `cvs edit` hingewiesen (einige grafische CVS-Clients erzeugen dann entsprechende Nachrichten).
  Achtung: Bei einigen Editoren bemerkt man dies möglicherweise erst am Ende der Editierarbeiten.

  Durch den Aufruf von `cvs edit file` wird diese Datei wieder mit den Rechten **read-write** versehen und die entsprechenden Benachrichtigungen verschickt.
  Der Nutzer hat aber immer die Möglichkeit, die Schreibrechte mittels `chmod` selbst herzustellen und kann diesen Mechanismus also umgehen! Allerdings würde er dann u. U. bewußt gegen ev. bestehende Regeln verstoßen.

  Nach einem abschließenden Commit wird die Datei wieder mit read-only Dateirechten versehen!

Achtung: Durch ein Update werden keine Dateien auf read-only gesetzt! Bei einem Commit hingegen wird eine Datei auf read-only gesetzt, wenn in der Zwischenzeit jemand ein `cvs watch on` aufgerufen hat. Bei `cvs watch off` ist das Verhalten ähnlich.

Zu beachten ist noch, daß `cvs watch on` niemanden auf einer Watchlist einträgt.

Weiterhin wird man oft nicht nur für eine einzelne Datei, sondern gleich für ein ganzes Verzeichnis `cvs watch on` aufrufen.

Die Speicherung der Informationen bzgl. Edits und Watches geschieht im Repository in der Datei `.../CVS/fileattr`. Wenn keine Edits oder Watches gesetzt sind, entfernt CVS diese Datei wieder.

A Wichtiges / Weiteres / Tips

- Per Default arbeiten die CVS-Befehle rekursiv auf dem aktuellen Verzeichnis.
  Abhilfe: Option `-l`
- Textdateien werden immer mit der Zeilenende-Konvention des jeweiligen Systems behandelt, d. h. auf dem UNIX-Server mit `<LF>`, auf den ausgecheckten Dateien unter NT mit `<CRLF>`.
- Binäre Dateien sollten nicht mit `cvs import`, sondern nur mit `cvs add -kb file` hinzugefügt werden, außer wenn die entsprechenden Dateitypen bereits als binär in `cvswrappers` eingetragen sind.
- Auf dem CVS-Server sollte unbedingt GNU-Diff eingesetzt werden, damit die Option `-c` für `cvs diff` benutzt werden kann.
- Es wird kein Locking der Dateien unterstützt, man kann sich allerdings anzeigen lassen, wer eine Datei bearbeitet (Watches, Editors).
- Auf dem Server werden die Dateien nicht im Original gespeichert, sondern als fortlaufende Serie von Diffs aller Zwischenstände.
- Keyword-Expansion: Folgende Keywords können benutzt werden:

  `$Author$ $Date$ $Header$ $Id$ $Revision$ $Source$ $Locker$ $Name$ $RCSfile$ $State$ $Log$.`

  Weitere Keywords können im CVS-Quelltext eingefügt werden.
- **Achtung:** Verzeichnisse können nur geleert werden, aber nicht gelöscht, da das Verzeichnis für die History alter Versionen erhalten bleiben muß.
  Verzeichnisse werden **nicht** mit-versioniert!
- Wenn eine Datei gelöscht wird, kommt diese in ein Verzeichnis `Attic` (wird angelegt).
- Es ist sinnvoll, einen cronjob einzurichten, welcher täglich ein Update des Workspace durchführt.
- Bei einem Update werden neue Verzeichnisse im Repository *nicht* automatisch übernommen. Man sollte daher von Zeit zu Zeit die Kommando-Option `-d` beim Update benutzen oder die Datei `~/.cvsrc` entsprechend anpassen (siehe 4).

- Bevor Sourcen ins Repository committed werden, sollte geprüft werden, ob diese kompilierbar sind.
- Es können sog. Module definiert werden, die Aliase auf Verzeichnisse im Repository darstellen. Diese kann man sich auch anzeigen lassen: `$ cvs co -c`.
- Man kann ein Repository nicht durchbrowsen (außer lokal mit `ls`).
  Abhilfe: `CVSweb` oder `$ cvs co -c`.
- Niemals einfach CVS-Dateien zwischen verschiedenen Workspaces kopieren! Immer `update` und `commit` nutzen.
- Viele CVS-Befehle haben Kurzformen, z. B. `cvs ci` für `commit` oder `cvs co` für `cvs checkout`, die man sich mit `cvs --help-synonyms` anzeigen lassen kann.
- Die (meisten) Admin-Dateien in `${CVSROOT}/CVSROOT` dürfen nicht direkt editiert werden, sondern unterliegen ebenfalls einer CVS-Kontrolle.

B Hooks

Einige der Ereignisse im CVS können (unabhängig von eventuellen Watches) durch Skripte bzw. spezielle Dateien beeinflußt werden:

commitinfo
: Hier können Programme/Skripte angegeben werden, die während eines Commits ausgeführt werden. Dabei kann der Inhalt der zu commitenden Datei untersucht werden.

cvsignore
: Hier aufgeführte Dateien (Wildcards möglich) werden bei einem `update, import` oder `release` ignoriert.
Einige Dateitypen werden auch ohne diese Datei vom CVS bereits per Default ignoriert.

cvswrappers
: Für hier aufgeführte Dateien (Wildcards möglich) können eine Update-Methode (*Copy* oder *Merge*) sowie der Keyword-Ersetzungsmode angegeben werden. Insbesondere können hier global bestimmte Dateitypen als binär definiert werden.

loginfo
: Hier können Programme/Skripte aufgeführt werden, die bei einem `commit` durchgeführt werden sollen, wobei die Log-Message übergeben wird.

modules
: Hier können Aliase für Projekte festgelegt werden (dabei können auch Unterverzeichnisse ausgeschlossen werden) sowie der Aufruf von Programmen/Skripte bei den Aktionen `export, commit, checkout, rtag` oder `update` für Dateien aus einem Modul.

notify
: Kontrolliert, wie die Benachrichtigungen bei Watches durchgeführt werden.

rcsinfo
: Gibt Templates für das Ausfüllen der Log-Messages an.

taginfo
Hier können Programme/Skripte angegeben werden, die während eines Tag-Befehls ausgeführt werden (hier kann z. B. überprüft werden, ob der Tagname bestimmte Konventionen einhält).

users
Bildet Usernamen auf Emailadressen ab.

verifymsg
Wird zusammen mit rcsinfo zum Überprüfen der Log-Messages eingesetzt.

C Weitere Dokumentationen

man cvs: Die Man-Page zu CVS.

info cvs: Das umfangreiche Manual zum CVS von Per Cederqvist.

Usenet: Newsgruppe zu Source Control Systemen: `comp.software.config-mgmt`

Sourcegear: (CVS-Download)
`www.sourcegear.com/CVS/`

***Version Management with CVS* by Per Cederqvist:**
`www.loria.fr/cgi-bin/molli/wilma.cgi/doc.847210383.html`

CVS Datasheet
`www.cvshome.org/cyclic/cyclic-pages/CVS-sheet.html`

***Introduction to CVS* by Jim Blandy (a tutorial):**
`www.cvshome.org/docs/blandy.html`

***The CVS Book* by Karl Fogel:**
`cvsbook.red-bean.com/`

Related tools:
`www.loria.fr/cgi-bin/molli/wilma.cgi/rel/`

Documentation-FAQ:
`www.loria.fr/cgi-bin/molli/wilma.cgi/doc/`

D GUIs und Clients im WWW

WinCVS:
`www.wincvs.org/download.html`

jCVS:
`trustice.com/java/jcvs/`

tkCVS
`www.twobarleycorns.net/tkcvs.html`