SelfLinux-0.10.0

Übersicht CVS

Autor: Karl Fogel ()
Formatierung: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)

Der folgende Text enthält die Kapitel 2 der deutschen Übersetzung des Buches "Open Source Development with CVS", welche unter der GNU Public License veröffentlicht wurden.

Das SelfLinux-Team

Inhaltsverzeichnis

1 CVS-Grundlagen

Dieses Kapitel führt in die grundlegenden Konzepte von CVS ein und gibt dann eine tiefer gehende Erläuterung des täglichen CVS-Einsatzes. Nach dessen Lektüre sind Sie auf dem besten Wege, ein CVS-Guru zu werden.

Wenn Sie noch nie CVS (oder ein anderes Versionsverwaltungssystem) benutzt haben, kann es leicht sein, dass Sie durch einige der zu Grunde liegenden Annahmen ins Stolpern geraten. Was anfänglich anscheinend für die meiste Verwirrung sorgt ist, dass CVS für zwei scheinbar unabhängige Aufgaben eingesetzt wird: Historienverwaltung und Zusammenarbeit. Es zeigt sich jedoch, dass diese beiden Funktionen eng miteinander verbunden sind.

Die Historienverwaltung wurde notwendig, weil Benutzer den momentanen Zustand eines Programmes mit dem an einem bestimmten Zeitpunkt der Vergangenheit vergleichen wollten. Zum Beispiel bringt ein Programmierer während der Implementation einer neuen Funktion das Programm in einen ziemlich fehlerhaften Zustand, in dem das Programm auch wahrscheinlich erst einmal bleiben wird, bis diese neue Funktion endgültig fertig implementiert ist. Unglücklicherweise ist genau dies meist der Zeitpunkt, zu dem ein Benutzer die Beschreibung eines Fehlers in der letzten veröffentlichten Version einschickt. Um diesen Fehler zu finden (der auch in der aktuellen Entwicklerversion enthalten sein kann), muss das Programm zuerst wieder in einen benutzbaren Zustand gebracht werden.

Diesen Zustand wieder herzustellen stellt dann kein Problem dar, wenn die Historie zu den Quelltexten mit CVS verwaltet wird. Ein Entwickler kann einfach sagen: »Gib mir den Quelltext, wie er vor drei Wochen war«, oder vielleicht: »Gib mir den Quelltext so, wie er war, als wir die letzte öffentliche Version freigegeben haben.« Wenn Sie bisher noch nie einen so praktischen Zugriff auf frühere Versionen hatten, werden Sie überrascht sein, wie schnell man davon abhängig werden kann. Persönlich verwende ich eine Revisionskontrolle bei allen meinen Programmierprojekten - dies hat mich schon oft gerettet.

Um zu verstehen, was dies mit der Unterstützung der Zusammenarbeit zu tun hat, müssen wir zunächst die Mechanismen etwas näher betrachten, mit denen CVS es ermöglicht, dass mehrere Personen zusammen an einem Projekt arbeiten. Doch zuvor sehen wir uns einen Mechanismus an, den CVS nicht bietet (oder der zumindest nicht zu empfehlen ist): Dateisperren. Wenn Sie bereits andere Versionsverwaltungssysteme benutzt haben, werden Sie schon mit dem Entwicklungsmodell **Sperren-Ändern-Freigeben** vertraut sein, bei dem ein Entwickler zuerst den exklusiven Schreibzugriff auf die zu bearbeitende Datei (eine Sperre) bekommen muss, die Veränderungen vornimmt und dann die Sperre wieder freigibt, damit andere Entwickler auf diese Datei zugreifen können. Wenn jemand anderes bereits eine Sperre für diese Datei gesetzt hat, so muss er diese zuerst wieder freigeben, bevor man selbst eine Sperre setzen und Veränderungen anbringen kann. (In manchen Implementationen kann man diese Sperre auch stehlen, was aber für den anderen eine böse Überraschung und außerdem kein guter Stil ist!)

Dieses System ist dann brauchbar, wenn sich die Entwickler kennen, wissen, wer was zu einem bestimmten Zeitpunkt machen möchte, und, im Falle von Zugriffskonflikten, schnell miteinander kommunizieren können. Wenn jedoch die Entwicklergruppe zu groß wird oder zu weiträumig verstreut ist, knabbert die Verwaltung der Sperren an der eigentlichen Arbeitszeit; dies wird zu einem ständigen Problem und entmutigt viele, wirkliche Arbeit zu leisten.

CVS verfolgt einen ausgereifteren Ansatz. Anstatt von den Entwicklern zu verlangen, sich gegenseitig zu koordinieren, erlaubt CVS den Entwicklern gleichzeitiges Arbeiten, übernimmt die Integration der Veränderungen und behält mögliche Konflikte im Auge. Dieser Prozess benutzt das **Kopieren-Modifizieren-Zusammenfassen-Modell**, das wie folgt funktioniert:

Entwickler A fordert eine Arbeitskopie von CVS an (ein Verzeichnisbaum, der alle Dateien eines Projektes enthält). Dies wird auch **Checking out** einer Arbeitskopie genannt, wie das Ausleihen eines Buches aus einer

Bibliothek.

Entwickler A arbeitet frei an seiner Arbeitskopie. Zum gleichen Zeitpunkt können auch andere Entwickler an ihren eigenen Arbeitskopien fleißig sein. Weil alle Kopien unabhängig voneinander sind, gibt es auch keine Konflikte - es ist so, als hätten alle Entwickler ihre eigene Kopie des gleichen Buches aus der Bibliothek, und sie alle schreiben, unabhängig voneinander, Kommentare an die Ränder oder bestimmte Seiten vollständig neu.

Entwickler A beendet seine Veränderungen und sendet diese mit einer **Log-Nachricht**, also einem Kommentar, der beschreibt, was der Zweck der Veränderungen war, an den CVS-Server (`commit`). Dies ist damit vergleichbar, die Bibliothek darüber zu informieren, welche Veränderungen gemacht wurden und warum. Die Bibliothek lässt diese wiederum in eine Hauptkopie einfließen, wo sie damit für alle Zeit aufgezeichnet werden.

In der Zwischenzeit können andere Entwickler CVS dazu veranlassen, die Bibliothek abzufragen, um herauszufinden, ob die Hauptkopie in jüngster Zeit verändert wurde. Ist dem so, aktualisiert CVS automatisch deren Arbeitskopie. (Dieser Teil grenzt an Magie und ist einfach wunderbar, ich hoffe, Sie wissen dies zu schätzen. Stellen Sie sich vor, wie die Welt wäre, wenn echte Bücher so funktionieren würden!)

Soweit es CVS betrifft, sind alle Entwickler eines Projektes gleich. Zu entscheiden, wann ein `Commit` oder eine Aktualisierung durchgeführt wird, ist eine Sache der persönlichen Einschätzung oder der Projektregeln. Eine übliche Strategie bei Programmierprojekten ist es, immer eine Aktualisierung zu machen, bevor die Arbeit an größeren Veränderungen begonnen wird, und einen `Commit` erst dann zu machen, wenn die Veränderungen vollständig und getestet sind, sodass die Hauptkopie immer in einem funktionsfähigen Zustand ist.

Vielleicht fragen Sie sich, was passiert, wenn die Entwickler A und B in ihren Arbeitskopien unterschiedliche Veränderungen an dem gleichen Stück (Quell-)Text vornehmen und beide ihre Veränderungen mittels `Commit` abschicken? Dies wird Konflikt genannt und von CVS entdeckt, sobald Entwickler B, versucht seine Veränderungen abzuschicken. Anstatt Entwickler B zu erlauben fortzufahren, gibt CVS bekannt, dass es einen Konflikt gefunden hat, und setzt Konfliktmarkierungen (leicht zu erkennende Marken im Text) an die in Konflikt stehenden Stellen im Text seiner Kopie. Diese Stellen beinhalten beide Veränderungen und sind derart angeordnet, dass sie leicht verglichen werden können. Entwickler B muss sich nun alles noch einmal ansehen und eine neue Version abschicken, die den Konflikt auflöst. Vielleicht müssen die beiden Entwickler miteinander reden, um die Sache zu klären. CVS alarmiert nur die Entwickler über die Konflikte; es ist an den Menschen, diese tatsächlich zu lösen.

Was ist nun mit der Hauptkopie? In der offiziellen CVS-Terminologie wird diese das Archiv (Repository) eines Projektes genannt. Das Archiv ist schlicht nur ein Datei-/Verzeichnisbaum, der auf einem Server gespeichert ist. Ohne zu stark in die Tiefe der Struktur zu gehen (siehe jedoch Kapitel 4), werfen wir einen Blick darauf, was das Archiv leisten muss, um den Anforderungen des **Checkout-Commit-Aktualisieren-Zyklus** gerecht zu werden.

Stellen Sie sich folgendes Szenario vor:

Zwei Entwickler, A und B, führen gleichzeitig einen `Checkout` des gleichen Projektes aus. Das Projekt befindet sich noch am Ausgangspunkt - es wurden noch von niemandem Veränderungen per `Commit` an das Archiv geschickt, sodass sich noch alle Dateien in ihrem ursprünglichen Zustand befinden.

Entwickler A beginnt sofort mit seiner Arbeit und führt schon bald den ersten `Commit` seiner Änderungen aus.

In der Zwischenzeit sieht Entwickler B fern.

Entwickler A arbeitet, als ob es kein Morgen gäbe, und führt einen zweiten `Commit` für einen weiteren Satz von Veränderungen aus. Das Archiv enthält nun die Originaldateien, gefolgt von As erstem Satz von Veränderungen, gefolgt von diesem Satz an Veränderungen.

In der Zwischenzeit spielt Entwickler B Videospiele.

Plötzlich schließt sich Entwickler C dem Projekt an und macht einen `Checkout` einer Arbeitskopie aus dem Archiv. Die Kopie von Entwickler C enthält As erste zwei Sätze von Veränderungen, weil diese schon im Archiv enthalten waren, als C für seine Arbeitskopie einen `Checkout` gemacht hat.

Entwickler A arbeitet weiter wie von Geistern besessen, vollendet seinen dritten Satz an Veränderungen und führt abermals einen `Commit` aus.

Zu guter Letzt, nichts ahnend von der jüngsten rasanten Aktivität, entscheidet Entwickler B, dass es Zeit wird, an die Arbeit zu gehen. Er kümmert sich nicht darum, eine Aktualisierung seiner Arbeitskopie durchzuführen; er fängt an, Dateien zu bearbeiten, von denen einige jene Dateien sein könnten, an denen A gearbeitet hat. Kurz darauf führt Entwickler B seinen ersten `Commit` dieser Veränderungen aus.

An diesem Punkt können nun zwei Dinge passieren. Wenn keine der von Entwickler B bearbeiteten Dateien von A bearbeitet wurde, dann ist der `Commit` erfolgreich. Wenn CVS jedoch merkt, dass einige der Dateien von B verglichen mit den aktuellen Dateien des Archivs veraltet sind und diese auch von B in seiner Arbeitskopie verändert wurden, informiert CVS B darüber, dass er eine Aktualisierung durchführen muss, bevor ein `Commit` durchgeführt werden kann.

Wenn Entwickler B die Aktualisierung durchführt, fügt CVS alle Veränderungen von A in Bs lokale Kopien der Dateien ein. Einige von As Veränderungen können mit Bs noch nicht abgeschickten Veränderungen in Konflikt geraten, manche nicht. Die Teile, welche nicht in Konflikt stehen, werden einfach ohne weitere Komplikationen in Bs Kopie eingefügt; die in Konflikt stehenden müssen zuerst von B bereinigt werden, bevor der `Commit` durchgeführt werden kann.

Wenn Entwickler C nun eine Aktualisierung durchführt, bekommt er mehrere Sätze an Veränderungen aus dem Archiv: den dritten `Commit` von A, den ersten von B und vielleicht den zweiten von B (wenn B die Konflikte aufgelöst hatte).

Damit CVS Veränderungen in der richtigen Reihenfolge an die Entwickler verteilen kann, deren Arbeitskopien unter Umständen unterschiedlich stark veraltet sind, muss das Archiv alle `Commits` seit Projektbeginn aufzeichnen. In der Praxis speichert das CVS-Archiv diese als aufeinander folgende `Diffs`. Daher kann CVS auch noch für sehr alte Arbeitskopien den Unterschied zwischen den Dateien der Arbeitskopien und dem aktuellen Stand des Archivs bestimmen und dadurch die Arbeitskopie effizient wieder auf den aktuellen Stand bringen. Für Entwickler ist es dadurch einfach, die Historie des Projektes einzusehen und zu jedem Zeitpunkt sogar sehr alte Arbeitskopien wieder zum Leben zu erwecken.

Obwohl das Archiv genau genommen das gleiche Resultat mit anderen Methoden erreichen könnte, ist das Abspeichern der `Diffs` eine einfache und intuitive Methode, die notwendige Funktionalität zu implementieren. Dieser Prozess hat den zusätzlichen Vorteil, dass CVS durch die korrekte Anwendung von `patch` jeden vorangegangenen Zustand des Verzeichnisbaumes wiederherstellen und damit jede Arbeitskopie von einem Zustand in einen anderen überführen kann. Es erlaubt jedem, einen `Checkout`. Daher des Projektes in einem womöglich vergangenen Zustand zu machen. Es kann ebenso die Unterschiede im `diff`-Format zwischen zwei Zuständen des Projektes aufzeigen, ohne dabei irgendeine Arbeitskopie zu beeinflussen.

Daher sind genau diese Funktionen, die den vernünftigen Zugriff auf die Historie eines Projektes zulassen, auch dafür nützlich, es einer dezentralen, unkoordinierten Entwicklergruppe zu ermöglichen, an einem Projekt zusammenzuarbeiten.

Die Details, wie ein Archiv angelegt wird, Benutzerzugriffe administriert werden und CVS-spezifische Dateiformate gehandhabt werden (diese werden in Kapitel 4 beschrieben), können Sie erst einmal außer Acht lassen. Im Augenblick konzentrieren wir uns darauf, wie Veränderungen an einer Arbeitskopie durchgeführt werden können.

Doch zuerst noch eine kurze Übersicht der Terminologie:

Revision - Eine Veränderung an einer Datei oder Menge von Dateien, die durch einen Commit abgeschlossen wurde. Eine Revision ist eine Momentaufnahme eines sich ständig verändernden Projektes.

Archiv - Die Hauptkopie, in der CVS die vollständige Revisionshistorie eines Projektes speichert. Jedes Projekt hat genau ein Archiv.

Arbeitskopie - Die Kopie, mit der gearbeitet wird und die tatsächlich verändert wird. Es kann mehrere Arbeitskopien eines bestimmten Projektes geben; im Allgemeinen hat jeder Entwickler seine eigene Kopie.

Checkout - Eine Arbeitskopie von dem Archiv anfordern. Die angeforderte Kopie stellt den Zustand des Projektes zu dem Zeitpunkt dar, zu dem die Kopie angefordert wurde; wenn Sie oder andere Entwickler Veränderungen vornehmen, müssen commit und update durchgeführt werden, um die eigenen Veränderungen zu »veröffentlichen« und die der anderen Mitentwickler sehen zu können.

Commit - Senden der eigenen Veränderungen zum zentralen Archiv. Auch Check-in genannt.

Log-Nachricht - Ein Kommentar der bei einem Commit einer Revision angehängt wird und die vorgenommenen Veränderungen beschreibt. Andere Entwickler können durch die Log-Nachrichten blättern und erhalten so die Antwort auf die Frage, was in dem Projekt passiert ist.

Aktualisierung (update) - Veränderungen von anderen Entwicklern vom Archiv in die eigene Arbeitskopie aufnehmen und aufzeigen, ob die eigene Arbeitskopie noch nicht durch `commit` abgeschickte Veränderungen enthält.

Konflikt - Situation, in der zwei Entwickler Veränderungen im gleichen Teil der gleichen Datei per commit abzuschicken versuchen. CVS bemerkt solche Konflikte und benachrichtigt die Entwickler, aber die Entwickler müssen diese selbst auflösen.

2 Ein Tag mit CVS

Der folgende Teil des Buches gibt eine Einführung in die grundlegende Benutzung von CVS, gefolgt von einer beispielhaften Sitzung, welche die typischsten CVS-Operationen beinhaltet. Im Laufe dessen werden wir auch beginnen, die interne Arbeitsweise von CVS zu betrachten.

Obwohl Sie zur Benutzung die Implementation von CVS nicht bis ins kleinste Detail verstehen müssen, ist ein Grundwissen über dessen Funktionsweise unschätzbar wertvoll, um ein bestimmtes Ergebnis zu erzielen. CVS verhält sich eher wie ein Fahrrad als ein Auto, denn seine Mechanismen sind für jeden transparent, der einen aufmerksamen Blick darauf wirft. Wie mit einem Fahrrad kann man einfach aufspringen und sofort anfangen zu fahren. Wenn man sich jedoch einen Augenblick Zeit nimmt, um genauer zu betrachten, wie das Getriebe funktioniert, kann man wesentlich besser fahren. (Im Falle von CVS bin ich mir nicht sicher, ob diese Transparenz ein bewusstes Entwicklungsziel oder ein Unfall gewesen ist, aber es scheint eine Eigenschaft zu sein, die auf viele freie Programme zutrifft. Durchschaubare Implementationen haben den Vorteil, Benutzer dazu zu motivieren, zu dem Projekt beitragende Entwickler zu werden, indem sie von Anfang an mit den internen Prozessen konfrontiert werden.)

Unsere Führung findet in einer Unix-Umgebung statt. CVS läuft auch unter Windows oder dem Macintosh Betriebssystem, und Tim Endres von Ice Engineering hat sogar einen Java-Client geschrieben, der überall dort läuft, wo auch Java läuft. Dennoch wage ich die grobe Schätzung, dass die Mehrheit der CVS-Benutzer wahrscheinlich mit einer Unix Kommandozeilenumgebung arbeiten. Sollten Sie kein solcher sein, so sollten die Beispiele dieser Führung dennoch leicht auf andere Oberflächen übertragbar sein. Haben Sie die Konzepte einmal verstanden, können Sie sich an jede CVS-Oberfläche setzen und damit arbeiten (vertrauen Sie mir, ich habe dies schon oft gemacht).

Die Beispiele der Führung orientieren sich an Benutzern, die CVS für Programmierprojekte einsetzen werden. Trotzdem sind CVS-Operationen auf alle Textdokumente anwendbar, nicht nur auf Quelltexte.

In der Führung wird auch davon ausgegangen, dass Sie CVS bereits installiert haben (es ist bei vielen bekannten freien Unix-Systemen bereits enthalten, wodurch Sie es bereits haben könnten, ohne es zu wissen) und dass Sie Zugriff auf ein Archiv haben. Auch wenn Sie diese Voraussetzungen nicht erfüllen, können Sie dennoch von dieser Führung profitieren. In Kapitel 4 werden Sie lernen, wie man CVS installiert und wie Archive angelegt werden.

Davon ausgehend, dass CVS installiert ist, sollten Sie sich einen Augenblick Zeit nehmen, die Online-Dokumentation zu CVS zu finden. Gewöhnlich als das **Cederqvist** bekannt (nach *Per Cederqvist*, dem ursprünglichen Autor), liegt es dem Quelltextpaket von CVS bei und ist die wohl aktuellste verfügbare Referenz. Der Text ist im Texinfo-Format geschrieben und sollte auf Unix-Systemen in der **Info**-Hierarchie zu finden sein. Sie können dieses entweder mit dem Kommandozeilen `info`-Programm lesen

```
user@linux ~/ # info cvs
```

oder durch die Tastenkombination `CTRL + H` und dann `i` in Emacs. Wenn keines derer bei Ihnen funktioniert, fragen Sie den nächsten Unix-Guru (oder lesen Sie Kapitel 4, Installation). Wenn Sie regelmäßig mit CVS arbeiten wollen, sollten Sie auf jeden Fall das **Cederqvist** zur Hand haben.

3 CVS aufrufen

CVS ist ein einzelnes Programm, kann aber viele verschiedene Aktionen ausführen: `Update`, `Commit`, Verzweigung (Branch), `Diff`, und so weiter. Wenn Sie CVS aufrufen, müssen Sie angeben, welche Aktion Sie ausführen wollen. Daraus folgt das Format für CVS-Aufrufe:

```
user@linux ~/ # cvs Kommando
```

Zum Beispiel

```
user@linux ~/ # cvs update
user@linux ~/ # cvs diff
user@linux ~/ # cvs commit
```

und so weiter. (Aber versuchen Sie nicht, eines dieser Kommandos in dieser Form auszuführen; solange Sie sich noch nicht in einer Arbeitskopie befinden, wird noch nichts passieren, wozu wir aber gleich kommen.)

Sowohl CVS als auch das Kommando können zusätzliche Optionen bekommen. Optionen, die das Verhalten von CVS unabhängig von dem auszuführenden Kommando verändern, heißen globale Optionen; kommandospezifische Optionen heißen einfach Kommandooptionen. Globale Optionen stehen immer links des Kommandos; Kommandooptionen rechts davon. Also ist bei

```
user@linux ~/ # cvs -Q update -p
```

`-Q` eine globale Option und `-p` eine Kommandooption. (Falls es Sie interessiert, `-Q` bedeutet **leise** - also alle Diagnosemeldungen unterdrücken und Fehlermeldungen nur dann ausgeben, wenn das Kommando aus irgendeinem Grund gar nicht verarbeitet werden kann; `-p` bedeutet, das Ergebnis des `Update` auf der Standardausgabe auszugeben, anstatt es in Dateien zu schreiben.)

4 Zugriff auf ein Archiv

Bevor überhaupt irgendetwas ausgeführt werden kann, muss CVS der Ursprungsort des Archivs, auf das zugegriffen werden soll, mitgeteilt werden. Dies trifft dann nicht mehr zu, wenn schon eine Arbeitskopie durch einen `Checkout` geholt wurde - jede Arbeitskopie weiß, aus welchem Archiv sie stammt, wodurch CVS das Archiv automatisch aus einer bestimmten Arbeitskopie ableiten kann. Nehmen wir aber dennoch an, Sie haben noch keine Arbeitskopie und müssen daher CVS explizit mitteilen, wohin es sich wenden soll. Dies geschieht mit der globalen Option `-d` (`-d` steht für **directory**, eine Option, für die es eine historische Begründung gibt, obwohl `-r` für **Repository** vielleicht besser gewesen wäre), gefolgt von dem Pfad zu dem Archiv. Nehmen wir zum Beispiel an, das Archiv befindet sich auf der lokalen Maschine in `/usr/local/cvs` (ein Standardort):

```
user@linux ~/ # cvs -d /usr/local/cvs Kommando
```

In vielen Fällen befindet sich das Archiv jedoch auf einer anderen Maschine, und es muss daher über das Netzwerk zugegriffen werden. CVS stellt mehrere Netzwerkzugriffsmethoden zur Verfügung; welche eingesetzt werden soll, hängt von den Sicherheitsansprüchen des Archivservers ab (im Folgenden **Server** genannt). Den Server für verschiedene Zugriffsmethoden einzurichten, wird in Kapitel 4 beschrieben; an dieser Stelle soll nur der Teil des Clients behandelt werden.

Glücklicherweise haben alle Netzwerkzugriffsmethoden eine gemeinsame Aufrufsyntax. Grundsätzlich muss zur Spezifikation eines nicht lokalen Archivs lediglich ein längerer Pfad zum Archiv angegeben werden. Zuerst wird die Zugriffsmethode angegeben, zu beiden Seiten mit Doppelpunkten abgetrennt, gefolgt von dem Benutzernamen und dem Servernamen (zusammengesetzt mit einem @-Zeichen), einem weiteren Doppelpunkt als Trenner und letztendlich dem Pfad des Archivverzeichnisses auf dem Server.

Betrachten wir die pserver-Zugriffsmethode, die für **passwort-authentisierten Server** steht:

```
user@linux ~/ # cvs -d :pserver:jrandom@cvs.foobar.com:/usr/local/cvs
login

(Logging in to jrandom@cvs.foobar.com)
CVS password: (hier das CVS Passwort eingeben)

user@linux ~/ #
```

Der lange Archivname nach `-d` sagte CVS, die pserver-Zugriffsmethode mit dem Benutzernamen jrandom auf dem Server cvs.foobar.com zu verwenden, der ein CVS-Archiv in `/usr/local/cvs` hat. Übrigens muss der Hostname nicht **cvs.irgendetwas.com** lauten; dies ist eine übliche Übereinkunft, aber es hätte auch einfach folgendermaßen sein können:

```
user@linux ~/ # cvs -d :pserver:jrandom@fisch.foobar.org:/usr/local/cvs
Kommando
```

Das tatsächlich verwendete Kommando war `login`, das verifiziert, ob Sie autorisiert sind, mit dem Archiv zu arbeiten. Das `login`-Kommando fragt Sie anschließend nach einem Passwort und kontaktiert dann den Server, um das Passwort zu verifizieren. Guter Unix-Sitte folgend, hat cvs `login` keine Ausgabe, wenn das Login korrekt ablief; wenn es fehlschlägt, wird eine Fehlermeldung ausgegeben (zum Beispiel weil das Passwort falsch war).

Man muss sich von seiner lokalen Maschine nur einmal bei einem bestimmten CVS-Server anmelden. Nach

einem erfolgreichen Login speichert CVS das Passwort in der Datei `.cvspass` in Ihrem Home-Directory. Diese Datei wird anschließend immer wieder eingelesen, wenn auf ein Archiv mit der pserver-Methode zugegriffen wird, wodurch `login` nur einmal beim ersten Zugriff auf einen bestimmten CVS-Server von einer bestimmten Client-Maschine aus durchgeführt werden muss. Natürlich kann cvs `login` jederzeit wiederholt werden, wenn sich zum Beispiel das Passwort geändert hat.

Bemerkung
pserver ist derzeit die einzige Zugriffsmethode, die ein erstes Login wie dieses benötigt; mit den anderen können direkt normale CVS-Kommandos ausgeführt werden.

Ist einmal die Authentifizierungsinformation in der `.cvspass`-Datei gespeichert, können andere CVS-Kommandos in der gleichen Kommandozeilensyntax ausgeführt werden:

```
user@linux ~/ # cvs -d :pserver:jrandom@cvs.foobar.com:/usr/local/cvs
Kommando
```

Die pserver-Methode unter Windows anzuwenden kann einen weiteren Schritt benötigen. Windows kennt das Unix-Konzept der Home-Verzeichnisse nicht, weshalb CVS nicht weiß, wo die cvspass-Datei abgespeichert werden soll. Hierzu muss explizit ein Verzeichnis angegeben werden. Normalerweise wird das Hauptverzeichnis der Festplatte C: als Home-Verzeichnis angegeben:

HOME-Verzeichnis

```
C:\WINDOWS> set HOME=C:
C:\WINDOWS> cvs -d :pserver:jrandom@cvs.foobar.com:/usr/local/cvs login
(Logging in to jrandom@cvs.foobar.com)
CVS password: (hier Passwort eingeben)
C:\WINDOWS>
```

Jedes Verzeichnis des Dateisystems ist möglich. Netzwerklaufwerke sollten jedoch vermieden werden, da der Inhalt der `.cvspass`-Datei dann für jeden ersichtlich wäre, der Zugriff auf dieses Laufwerk hat.

Zusätzlich zu **pserver** unterstützt CVS die `ext`-Methode (die ein externes Programm zur Verbindung benutzt, bspw. `rsh` oder `ssh`), **kserver** (für das Kerberos-Sicherheitssystem Version 4) und **gserver** (welches das GSSAPI, oder auch Generic Security Services API, benutzt und auch Kerberos Version 5 oder größer verwenden kann). Diese Methoden sind ähnlich wie pserver, haben aber jede für sich ihre Eigenheiten.

Von diesen ist wohl die `ext`-Methode die üblichste. Wenn Sie sich an dem Server über rsh oder ssh anmelden können, können Sie die `ext`-Methode benutzen. Sie können dies folgendermaßen ausprobieren:

```
user@linux ~/ # rsh -l jrandom cvs.foobar.com

Password hier Ihr Login-Passwort eingeben
```

Angenommen, Sie konnten sich mit `rsh` erfolgreich bei dem Server ein- und ausloggen, dann sind Sie nun wieder zurück auf Ihrer ursprünglichen Client-Maschine:

```
user@linux ~/ # CVS_RSH=rsh; export CVS_RSH
user@linux ~/ # cvs -d :ext:jrandom@cvs.foobar.com:/usr/local/cvs
```

```
Kommando
```

Die erste Zeile setzt (in der Syntax der Unix Bourne-Shell) die CVS_RSH-Umgebungsvariable auf rsh, was CVS mitteilt, rsh als das Programm zur Verbindung zu benutzen. Die zweite Zeile kann irgendein CVS-Kommando sein; Sie werden nach Ihrem Passwort gefragt, sodass CVS das Login beim Server durchführen kann.

Wenn Sie eine C-Shell anstatt einer Bourne-Shell benutzen, versuchen Sie Folgendes:

```
user@linux ~/ # setenv CVS_RSH rsh
```

und unter Windows versuchen Sie dies:

c:\WINDOWS

```
C:\WINDOWS> set CVS_RSH=rsh
```

Der Rest dieser Führung verwendet die Bourne-Shell-Syntax; Sie können dies für Ihre Umgebung bei Bedarf umsetzen.

Um ssh (die Secure-Shell) anstatt von rsh zu benutzen, muss nur die Umgebungsvariable CVS_RSH entsprechend gesetzt werden:

```
user@linux ~/ # CVS_RSH=ssh; export CVS_RSH
```

Lassen Sie sich nicht davon verwirren, dass die Variable CVS_RSH heißt, Sie ihren Inhalt aber auf ssh setzen. Es gibt historische Gründe dafür (die allumfassende Unix-Entschuldigung, ich weiß). CVS_RSH kann auf ein beliebiges Programm verweisen, welches das Login auf einer anderen Maschine sowie Kommandos ausführen und deren Ausgabe empfangen kann. Nach rsh ist ssh wohl das verbreitetste dieser Programme, obwohl es sicherlich noch andere gibt. Wichtig ist, dass diese Programme den Datenstrom in keiner Weise verändern dürfen. Dies disqualifiziert die Windows-NT rsh, weil diese zwischen den Unix- und DOS-Zeilenumbrüchen konvertiert (oder es zumindest versucht). Sie müssten sich in diesem Fall eine andere rsh für Windows besorgen oder eine andere Zugriffsmethode benutzen.

Die **gserver**- und **kserver**- Methoden werden nicht so oft wie die anderen benutzt und werden hier nicht behandelt. Diese sind bezüglich dessen, was bisher behandelt wurde, recht ähnlich; Näheres findet sich im Cederqvist.

Verwenden Sie nur ein Archiv und wollen nicht jedes Mal -d repos eingeben, können Sie einfach die CVSROOT -Umgebungsvariable (die vielleicht CVSREPOS hätte genannt werden sollen, doch dafür ist es nun zu spät) wie folgt setzen:

```
user@linux ~/ # CVSROOT=/usr/local/cvs
user@linux ~/ # export CVSROOT
user@linux ~/ # echo $CVSROOT

/usr/local/cvs
```

```
user@linux ~/ #
```

oder vielleicht

```
user@linux ~/ # CVSROOT=:pserver:jrandom@cvs.foobar.com:/usr/local/cvs
user@linux ~/ # export CVSROOT
user@linux ~/ # echo $CVSROOT

:pserver:jrandom@cvs.foobar.com:/usr/local/cvs

user@linux ~/ #
```

Der Rest dieser Führung geht davon aus, dass `CVSROOT` auf das Archiv verweist, sodass die Beispiele die `-d` Option nicht enthalten. Wenn auf mehrere Archive zugegriffen werden soll, sollte die `CVSROOT` -Umgebungsvariable nicht gesetzt werden und anstatt dessen mit `-d repos` das benötigte Archiv angegeben werden.

5 Ein neues Projekt beginnen

Wenn Sie den Umgang mit CVS erlernen wollen, um mit einem Projekt zu arbeiten, das bereits mit CVS verwaltet wird (das heißt, es befindet sich bereits irgendwo in einem Archiv), dann sollten Sie vielleicht mit dem Abschnitt Eine Arbeitskopie auschecken fortfahren. Möchten Sie allerdings existierende Quelltexte unter die Kontrolle von CVS stellen, ist dies der für Sie passende Abschnitt. Beachten Sie, dass immer noch davon ausgegangen wird, dass Sie Zugriff auf ein bereits bestehendes Archiv haben; wenn Sie zuerst eines anlegen müssen, lesen Sie bitte Kapitel 4.

Ein bestehendes Projekt in CVS zu übernehmen, wird importieren genannt. Das CVS-Kommando dazu lautet, wie Sie sich sicherlich bereits gedacht haben,

```
user@linux ~/ # cvs import
```

abgesehen davon, dass es noch ein paar zusätzliche Optionen benötigt (und an der richtigen Stelle im Dateisystem ausgeführt werden muss), um korrekt ausgeführt zu werden. Zuerst wechseln Sie in das Hauptverzeichnis Ihres Projektes:

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

README.txt a-subdir/ b-subdir/ hello.c

user@linux ~/ #
```

Dieses Projekt besteht aus zwei Dateien - `README.txt` und `hello.c` - im Hauptverzeichnis, zuzüglich zwei Unterverzeichnissen - `a-unterverzeichnis` und `b-unterverzeichnis` - und noch einiger weiterer Dateien in den Unterverzeichnissen, die hier nicht angezeigt werden. Wenn ein Projekt importiert wird, importiert CVS alles aus der Verzeichnisstruktur, ausgehend von dem aktuellen Verzeichnis den ganzen Baum hinab. Daher sollten Sie sich vergewissern, dass sich nur solche Dateien in den Verzeichnissen befinden, die auch permanenter Bestandteil des Projektes werden sollen. Jegliche alten Sicherheitskopien, Schmierdateien und so weiter sollten entfernt werden.

Die allgemeine Syntax des `import`-Kommandos ist

```
user@linux ~/ # cvs import -m "log nachr." projname hersteller-marke
versions-marke
```

Die Option `-m` (m = message, Nachricht) spezifiziert eine kurze Nachricht, die den `Import` beschreibt. Dies wird dann die erste Log-Nachricht des gesamten Projektes; jeder nachfolgende `Commit` wird ebenfalls eine eigene Log-Nachricht bekommen. Diese Nachrichten sind verpflichtend; wird die Option `-m` nicht angegeben, startet CVS automatisch einen Texteditor (unter Verwendung der Umgebungsvariablen $EDITOR), damit Sie eine Log-Nachricht eingeben können. Nachdem die Log-Nachricht abgespeichert wurde, wird der Import weiter durchgeführt.

Das nächste Argument der Kommandozeile ist der Projektname (hier wird »myproject« verwendet). Dies ist der Name, anhand dessen ein `Checkout` des Projektes aus dem Projektarchiv durchgeführt wird. (Was tatsächlich passiert ist, dass ein Verzeichnis mit diesem Namen im Archiv angelegt wird, doch mehr dazu in Kapitel 4). Der nun auszuwählende Name muss nicht der gleiche wie der des aktuellen Verzeichnisses sein, obwohl er es in den meisten Fällen wohl sein wird.

Die Argumente hersteller-marke und versions-marke dienen nur als Verwaltungsinformationen für CVS. Sie brauchen sich zu diesem Zeitpunkt nicht darum zu kümmern; es spielt praktisch keine Rolle, was Sie dafür wählen. In Kapitel 6 werden die seltenen Umstände beschrieben, unter denen diese relevant sind. Hier werden wir einen Benutzernamen und **start** für diese Argumente benutzen.

Wir können nun den Import starten:

```
user@linux ~/ # cvs import -m "initial import into CVS" myproj jrandom
start

N myproj/hello.c


N myproj/README.txt
cvs import: Importing /usr/local/cvs/myproj/a-subdir
N myproj/a-subdir/whatever.c
cvs import: Importing /usr/local/cvs/myproj/a-subdir/subsubdir
N myproj/a-subdir/subsubdir/fish.c
cvs import: Importing /usr/local/cvs/myproj/b-subdir
N myproj/b-subdir/random.c

No conflicts created by this import

user@linux ~/ #
```

Herzlichen Glückwunsch! Wenn Sie dieses oder ein ähnliches Kommando ausgeführt haben, haben Sie letztendlich etwas ausgeführt, was das Archiv verändert.

Wenn Sie sich die Ausgabe des `Import`-Kommandos noch einmal durchlesen, werden Sie feststellen, dass CVS den Dateinamen einen einzelnen Buchstaben vorangestellt hat - in diesem Fall **N** für **neue Datei**. Die Verwendung eines einzelnen Buchstabens an der linken Position, um den Status einer Datei anzuzeigen, ist bei den Ausgaben eines Kommandos von CVS üblich. Wir werden dies auch später bei `Checkout` und `Update` sehen.

Sie könnten nun denken, dass Sie, nachdem Sie gerade das Projekt importiert haben, in den aktuellen Verzeichnissen sofort arbeiten können. Dies ist jedoch nicht der Fall. Das aktuelle Verzeichnis ist immer noch keine CVS-Arbeitskopie. Es war die Quelle für das `import`-Kommando, richtig, aber es wurde nicht alleine durch die Tatsache, in CVS importiert worden zu sein, auf magische Art und Weise in eine Arbeitskopie verwandelt. Um eine Arbeitskopie zu erhalten, müssen Sie eine aus dem Archiv auschecken.

Zuerst sollten Sie vielleicht jedoch den aktuellen Projektstamm sichern. Der Grund dafür ist, dass, wenn die Quelltexte einmal im CVS-Archiv liegen, Sie sich nicht selbst dadurch verwirren sollten, indem Sie Kopien von Dateien modifizieren, die nicht der Versionskontrolle unterliegen (und diese Veränderungen daher nicht Teil der Projekthistorie werden). Sie sollten von nun an alle Ihre Arbeiten an einer Arbeitskopie vornehmen. Sie sollten jedoch den gerade importierten Verzeichnisbaum noch nicht entfernen, da Sie noch nicht überprüft haben, ob das Archiv alle Dateien enthält. Natürlich können Sie sich dessen zu 99,999 Prozent sicher sein, weil der `import`-Befehl ohne Fehler ablief, doch warum etwas riskieren? Paranoia zahlt sich aus, wie jeder Programmierer weiß. Daher führen Sie etwa wie folgt aus:

```
user@linux ~/ # ls

README.txt a-subdir/ b-subdir/ hello.c

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

```
user@linux ~/ # ls

myproj/

user@linux ~/ # mv myproj was_myproj
user@linux ~/ # ls

was_myproj/

user@linux ~/ #
```

So. Die Originaldateien sind noch vorhanden, sind aber durch den Namen klar als eine veraltete Version erkennbar, sodass sie nicht im Weg sind, wenn eine richtige Arbeitskopie geholt wird. Nun kann ein `Checkout` durchgeführt werden.

6 Eine Arbeitskopie auschecken

Das Kommando, um einen `Checkout` durchzuführen, ist genau das, was Sie sich sicherlich schon gedacht haben:

```
user@linux ~/ # cvs checkout myproj

cvs checkout: Updating myproj
U myproj/README.txt
U myproj/hello.c
cvs checkout: Updating myproj/a-subdir
U myproj/a-subdir/whatever.c
cvs checkout: Updating myproj/a-subdir/subsubdir
U myproj/a-subdir/subsubdir/fish.c
cvs checkout: Updating myproj/b-subdir
U myproj/b-subdir/random.c

user@linux ~/ # ls

myproj/ was_myproj/

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

CVS/ README.txt a-subdir/ b-subdir/ hello.c

user@linux ~/ #
```

Achtung - Ihre erste Arbeitskopie! Der Inhalt ist genau derselbe wie der, den Sie gerade importiert haben, zuzüglich eines Unterverzeichnisses `CVS`. In diesem werden von CVS Informationen zur Versionskontrolle gespeichert. Genauer gesagt, es existiert nun in jedem Unterverzeichnis des Projektes ein CVS-Unterverzeichnis:

```
user@linux ~/ # ls a-subdir

CVS/ subsubdir/ whatever.c

user@linux ~/ # ls a-subdir/subsubdir/

CVS/ fish.c

user@linux ~/ # ls b-subdir

CVS/ random.c
```

Tipp
Die Tatsache, dass CVS die Informationen zur Versionskontrolle in Unterverzeichnissen namens CVS ablegt, bedeutet, dass Ihr Projekt niemals eigene Unterverzeichnisse mit dem Namen CVS enthalten kann. Ich habe aber praktisch noch nie davon gehört, dass dies ein Problem gewesen wäre.

Bevor Dateien modifiziert werden, lassen Sie uns einen Blick in diese Blackbox werfen:

```
user@linux ~/ # cd CVS
```

```
user@linux ~/ # ls

Entries Repository Root

user@linux ~/ # cat Root

/usr/local/cvs

user@linux ~/ # cat Repository

myproject

user@linux ~/ #
```

Hier ist nichts besonders Mysteriöses. Die Datei `Root` verweist auf das Archiv, und die Datei `Repository` verweist auf ein Projekt innerhalb des Archivs. Lassen Sie es mich erklären, wenn dies auf Anhieb etwas verwirrend erscheint.

Die Terminologie von CVS sorgt seit langem für Verwirrung. Der Begriff »Archiv« wird für zwei unterschiedliche Dinge benutzt. Manchmal ist damit das Hauptverzeichnis eines Archivs gemeint (zum Beispiel `/usr/local/cvs`), das mehrere Projekte enthalten kann; die Datei `Root` verweist dorthin. Doch manchmal ist damit ein projektspezifisches Unterverzeichnis innerhalb des Archiv-Root gemeint (zum Beispiel `/usr/local/cvs/myproject`, `/usr/local/cvs/deinprojekt`, `/usr/local/cvs/Fisch`). Die Datei `Repository` innerhalb des CVS-Unterverzeichnisses hat diese Bedeutung.

Innerhalb dieses Buches bedeutet »Archiv« allgemein Root (also das übergeordnete Hauptarchiv), obwohl es gelegentlich auch ein projektspezifisches Unterverzeichnis bezeichnen kann. Sollte die eigentliche Intention nicht aus dem Kontext hervorgehen, wird dies im Text erklärt.

Beachten Sie, dass die Datei `Repository` manchmal mit einem absoluten Pfad anstatt eines relativen auf das Projekt verweist. Dies ist ein wenig redundant mit der `Root` Datei:

```
user@linux ~/ # cd CVS
user@linux ~/ # cat Root

:pserver:jrandom@cvs.foobar.com:/usr/local/cvs

user@linux ~/ # cat Repository

/usr/local/cvs/myproject

user@linux ~/ #
```

In der Datei `Entries` werden Informationen über die einzelnen Dateien eines Projektes abgelegt. Jede Zeile beschäftigt sich dabei mit einer Datei, und es finden sich dort auch nur Einträge für die Dateien und Unterverzeichnisse des nächst übergeordneten Verzeichnisses. Hier die Haupt-`CVS/Entries`-Datei in myproject:

```
user@linux ~/ # cat Entries

/README.txt/1.1.1.1/Sun Apr 18 18:18:22 1999//
/hello.c/1.1.1.1/Sun Apr 18 18:18:22 1999//
D/a-subdir////
D/b-subdir////
```

Jede Zeile folgt dem Format

/dateiname/revisionsnummer/Zeitstempel//

und die Zeilen der Verzeichnisse werden mit einem **D** eingeleitet. (CVS verwaltet keine Historie über Veränderungen der Verzeichnisse selbst, weshalb die Felder Revisionsnummer und Zeitstempel leer bleiben.)

Die Zeitstempel bezeichnen Datum und Uhrzeit der letzten Aktualisierung der Dateien in der Arbeitskopie (in universeller Zeit, nicht lokaler Zeit). Auf diese Weise kann CVS einfach unterscheiden, ob eine Datei seit dem letzten `checkout`, `update` oder `commit` verändert wurde. Wenn sich der Zeitstempel des Dateisystems von dem in der `CVS/Entries`-Datei unterscheidet, weiß CVS (ohne überhaupt das Archiv zu überprüfen), dass die Datei wahrscheinlich verändert wurde.

Betrachtet man die `CVS/*`-Dateien in den Unterverzeichnissen

```
user@linux ~/ # cd a-subdir/CVS
user@linux ~/ # cat Root

/usr/local/cvs

user@linux ~/ # cat Repository

myproj/a-subdir

user@linux ~/ # cat Entries

/whatever.c/1.1.1.1/Sun Apr 18 18:18:22 1999//
D/subsubdir////

user@linux ~/ #
```

stellt man fest, dass `Root` immer noch auf das gleiche Archiv verweist, `Repository` jedoch auf die Position des Verzeichnisses innerhalb des Projektes zeigt und die `Entries`-Datei andere Einträge enthält.

Direkt nach einem `Import` wird die Revisionsnummer einer jeden Datei des Projektes mit 1.1.1.1 angezeigt. Diese initiale Revisionsnummer ist eine Art Spezialfall, weshalb wir hier nicht näher darauf eingehen; wir werden uns näher mit Revisionsnummern beschäftigen, wenn ein `Commit` von ein paar Veränderungen durchgeführt wurde.

Version kontra Revision

Die von CVS intern verwalteten Revisionsnummern sind unabhängig von der Versionsnummer des Softwareproduktes, von dem diese ein Teil sind. Nehmen wir zum Beispiel ein aus drei Dateien bestehendes Projekt, deren Revisionsnummern am 3. Mai 1999 1.2, 1.7 und 2.48 waren. An diesem Tag wird eine neue Version dieser Software zusammengepackt und als SlickoSoft Version 3 freigegeben. Dies ist eine reine Marketingentscheidung und beeinflusst die CVS-Revisionen überhaupt nicht. Die CVS-Revisionsnummern sind für die Kunden nicht sichtbar (es sei denn, sie haben Zugriff auf das Archiv); die einzig sichtbare Nummer ist **3** in **Version 3**. Soweit es CVS betrifft, hätte die Version auch 1729 lauten können - die Versionsnummer (oder auch **Release**-Nummer) hat nichts mit der internen Verwaltung von Veränderungen durch CVS zu tun.

Um Verwirrung zu vermeiden, werde ich den Begriff **Revision** verwenden, um mich einzig auf die internen

Revisionsnummern von Dateien unter der Kontrolle von CVS zu beziehen. Ich werde trotzdem CVS ein **Versionskontrollsystem** nennen, weil **Revisionskontrollsystem** doch etwas komisch klingt.

6.1 Eine Veränderung einbringen

Das Projekt, in seinem momentanen Zustand, macht noch nicht allzu viel. Hier ist der Inhalt von `hello.c`:

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

#include <stdio.h>
void
main ()
{
printf ("Hello, world!\n");
}
```

Lassen Sie uns nun die erste Veränderung seit dem `Import` anbringen; es wird die Zeile

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

eingefügt, direkt nach `Hello`, world!. Starten Sie Ihren bevorzugten Texteditor und führen die Änderung durch:

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

...
```

Dies war eine recht simple Veränderung, eine, bei der man nicht so schnell vergessen kann, was man getan hat. Bei einem größeren und komplexeren Projekt ist es aber recht wahrscheinlich, dass man eine Datei bearbeitet, von etwas anderem unterbrochen wird und erst einige Tage später wieder dahin zurückkehrt und sich nun nicht mehr daran erinnern kann, was man tatsächlich oder ob überhaupt verändert hat. Dies bringt uns zur ersten Situation **CVS rettet Dein Leben**: die eigene Arbeitskopie mit dem Archiv vergleichen.

6.2 Herausfinden, was man selbst und andere getan haben: update und diff

Zuvor erwähnte ich `Update` als eine Methode, Veränderungen aus dem Archiv in die eigene Arbeitskopie einfließen zu lassen - also als eine Methode, die Veränderungen anderer Entwickler zu bekommen. `Update` ist jedoch etwas komplexer; es vergleicht den Gesamtzustand der Arbeitskopie mit dem Zustand des Projektes im Archiv. Auch wenn nichts im Archiv seit dem letzten `Checkout` verändert wurde, könnte sich dennoch etwas in der Arbeitskopie verändert haben, und `update` zeigt dies dann auch auf:

```
user@linux ~/ # cvs update

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

Das `M` neben `hello.c` bedeutet, dass die Datei seit dem letzten `Checkout` modifiziert wurde und die Veränderungen noch nicht mit `Commit` in das Archiv eingebracht wurden.

Manchmal ist alles, was man möchte, herauszufinden, welche Dateien man bearbeitet hat. Möchte man jedoch einen detaillierteren Blick auf die Veränderungen werfen, kann man einen kompletten Report im `diff`-Format anfordern. Das `diff`-Kommando vergleicht die möglicherweise modifizierten Dateien der Arbeitskopie mit den entsprechenden Gegenstücken im Archiv und zeigt jegliche Unterschiede auf:

```
user@linux ~/ # cvs diff

cvs diff: Diffing .
Index: hello.c
===================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 hello.c
6a7
> printf ("Goodbye, world!\n");
cvs diff: Diffing a-subdir
cvs diff: Diffing a-subdir/subsubdir
cvs diff: Diffing b-subdir
```

Dies hilft schon weiter, auch wenn es durch eine Menge überflüssiger Ausgaben ein wenig obskur erscheinen mag. Für den Anfang können die meisten der ersten paar Zeilen ignoriert werden. Diese benennen nur die Datei des Archivs und zeigen die Nummer der letzten eingecheckten Revision. Unter bestimmten Umständen kann auch das eine nützliche Information sein (wir werden später genauer dazu kommen), sie wird aber nicht gebraucht, wenn man nur einen Eindruck davon bekommen möchte, welche Veränderungen an der Arbeitskopie stattgefunden haben.

Ein größeres Hindernis, den `Diff` zu lesen, stellen die Meldungen von CVS bei jedem Wechsel in ein Verzeichnis während des `Updates` dar. Dies kann während eines langen `Updates` bei großen Projekten nützlich sein, da es einen Anhaltspunkt bietet, wie lange das `Update` wohl noch dauern wird. Doch jetzt sind sie beim Lesen des `Diff` schlicht im Weg. Also sagen wir CVS mit der globalen `-Q`-Option, dass es nicht melden soll, wo es gerade arbeitet:

```
user@linux ~/ # cvs -Q diff

Index: hello.c
===================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 hello.c
6a7
> printf ("Goodbye, world!\n");
```

Besser - zumindest ist ein Teil der überflüssigen Ausgaben weg. Dennoch ist der `Diff` noch schwer zu lesen. Er sagt aus, dass an Zeile 6 eine neue Zeile hinzugekommen ist (was also jetzt Zeile 7 ist), und dass deren Inhalt

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

ist. Das vorangestellte > in dem `Diff` bedeutet, dass diese Zeile in der neuen Version vorhanden ist, nicht aber in der älteren.

Das Format kann jedoch noch lesbarer gemacht werden. Die meisten empfinden das »Kontext«-`Diff`-Format als leichter zu lesen, da es ein paar Kontextzeilen zu beiden Seiten einer Veränderung mit anzeigt. Kontext `Diffs` werden durch die zusätzliche Option `-c` zu `diff` erzeugt:

```
user@linux ~/ # cvs -Q diff -c

Index: hello.c
===================================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.1.1.1
diff -c -r1.1.1.1 hello.c
*** hello.c 1999/04/18 18:18:22 1.1.1.1
--- hello.c 1999/04/19 02:17:07
***************
*** 4,7 ****
---4,8 ----
main ()
{
printf ("Hello, world!\n");
+ printf ("Goodbye, world!\n");
}
```

Nun, das ist Klarheit! Selbst wenn man nicht gewohnt ist, Kontext-`Diffs` zu lesen, macht ein kurzer Blick auf die vorangegangene Ausgabe offensichtlich, was passiert ist: eine neue Zeile wurde zwischen der Zeile, welche **Hello, world!** ausgibt und der abschließenden geschweiften Klammer hinzugefügt (das + in der ersten Spalte der Ausgabe markiert eine hinzugefügte Zeile).

Wir müssen Kontext-`Diffs` nicht perfekt lesen können, dies ist die Aufgabe von `patch`, es lohnt sich aber dennoch, sich die Zeit zu nehmen, um eine zumindest ansatzweise Gewöhnung an dieses Format zu bekommen. Die ersten beiden Zeilen (nach der momentan nutzlosen Einleitung) sind

```
*** hello.c 1999/04/18 18:18:22 1.1.1.1
--- hello.c 1999/04/19 02:17:07
```

und sagen einem, was mit wem »gedifft« wurde. In diesem Fall wurde Revision 1.1.1.1 von `hello.c` mit einer modifizierten Version der gleichen Datei verglichen (daher gibt es keine Revisionsnummer für die zweite Zeile, weil die Veränderungen der Arbeitskopie noch nicht mit einem `Commit` in das Archiv aufgenommen wurden). Die Zeilen mit Sternchen und Strichen markieren Teile im späteren Teil des `Diffs`. Anschließend wird ein Teil der Originaldatei von einer Zeile mit Sternchen und einem eingefügten Zeilennummernbereich eingeleitet. Danach folgt eine Zeile mit Strichen mit möglicherweise anderen Zeilenummernbereichen, die einen Teil der modifizierten Datei einleiten. Diese Bereiche sind in sich kontrastierenden Paaren angeordnet (genannt **Hunks**), die eine Seite von der alten Datei und die andere Seite von der neuen.

Dieser Diff besteht aus einem Hunk:

```
***************
*** 4,7 ****
```

```
--- 4,8 ----
main ()
{
printf ("Hello, world!\n");
+ printf ("Goodbye, world!\n");
}
```

Der erste Teil dieses Hunks ist leer, was bedeutet, dass keine Teile der Originaldatei entfernt wurden. Der zweite Teil zeigt an der entsprechenden Stelle der neuen Datei, dass eine Zeile eingefügt wurde; diese ist mit+ markiert. (Wenn diff Auszüge einer Datei anführt, werden die ersten beiden linken Spalten für spezielle Codes reserviert, wie bspw. das +, sodass der gesamte Auszug um zwei Zeichen eingerückt erscheint. Die zusätzliche Einrückung wird natürlich entfernt, bevor der `Diff` wieder als `Patch` angewendet wird.)

Der Zeilennummernbereich zeigt den Bereich an, den der Hunk einschließt, samt der Zeilen aus dem Kontext. In der Originaldatei umfasste der Hunk die Zeilen 4 bis 7; in der neuen Datei sind dies die Zeilen 4 bis 8 (weil eine Zeile hinzugefügt wurde). Zu beachten ist, dass `diff` keine Ausschnitte aus der Originaldatei angezeigt hat, weil nichts entfernt wurde; es wurde nur der Bereich angezeigt und dann zu der zweiten Hälfte des Hunks übergegangen.

Hier noch ein zweiter Kontext-`Diff` eines meiner Projekte:

```
user@linux ~/ # cvs -Q diff -c

Index: cvs2cl.pl
===================================
RCS file: /usr/local/cvs/kfogel/code/cvs2cl/cvs2cl.pl,v
retrieving revision 1.76
diff -c -r1.76 cvs2cl.pl
*** cvs2cl.pl 1999/04/13 22:29:44 1.76
--- cvs2cl.pl 1999/04/19 05:41:37
***************
*** 212,218 ****
# can contain uppercase and lowercase letters, digits, '-',
# and '_'. However, it's not our place to enforce that, so
# we'll allow anything CVS hands us to be a tag:
! /^\s([^:]+): ([0-9.]+)$/;
push (@{$symbolic_names{$2}}, $1);
}
}
--- 212,218 ----
# can contain uppercase and lowercase letters, digits, '-',
# and '_'. However, it's not our place to enforce that, so
# we'll allow anything CVS hands us to be a tag:
! /^\s([^:]+): ([\d.]+)$/;
push (@{$symbolic_names{$2}}, $1);
}
}
```

Das Ausrufungszeichen zeigt an, dass die markierte Zeile zwischen der neuen und alten Datei unterschiedlich ist. Da keine +- oder --Zeichen vorhanden sind, wissen wir, dass die Gesamtzeilenzahl der Datei gleich geblieben ist.

Hier ist noch ein Kontext-Diff des gleichen Projektes, diesmal ein wenig komplizierter:

```
user@linux ~/ # cvs -Q diff -c

Index: cvs2cl.pl
===================================================
RCS file: /usr/local/cvs/kfogel/code/cvs2cl/cvs2cl.pl,v
retrieving revision 1.76
diff -c -r1.76 cvs2cl.pl
*** cvs2cl.pl 1999/04/13 22:29:44 1.76
--- cvs2cl.pl 1999/04/19 05:58:51
***************
*** 207,217 ****
}
else # we're looking at a tag name, so parse & store it
{
- # According to the Cederqvist manual, in node "Tags", "Tag
- # names must start with an uppercase or lowercase letter and
- # can contain uppercase and lowercase letters, digits, '-',
- # and '_'. However, it's not our place to enforce that, so
- # we'll allow anything CVS hands us to be a tag:
/^\s([^:]+): ([0-9.]+)$/;
push (@{$symbolic_names{$2}}, $1);
}
---- 207,212 ----
***************
*** 223,228 ****
--- 218,225 ----
if (/^revision (\d\.[0-9.]+)$/) {
$revision = "$1";
}
+
+ # This line was added, I admit, solely for the sake of a diff example.
# If have file name but not time and author, and see date or
# author, then grab them:
```

Dieser Diff hat zwei Hunks. Im ersten wurden fünf Zeilen entfernt (diese Zeilen sind nur im ersten Teil des Hunks zu sehen, und die Zeilenanzahl des zweiten Teils weist fünf Zeilen weniger auf). Eine ununterbrochene Zeile von Sternchen markiert die Grenze zwischen Hunks. Im zweiten Hunk ist zu sehen, dass zwei Zeilen hinzugefügt wurden: eine Leerzeile und ein sinnloser Kommentar. Zu beachten ist, wie die Zeilennummern durch die Effekte des ersten Hunks kompensiert werden. In der Originaldatei war der Zeilennummernbereich des zweiten Hunks 223 bis 228; in der neuen Datei, bedingt durch das Entfernen von Zeilen durch den ersten Hunk, ist der Zeilennummernbereich 218 bis 225.

Herzlichen Glückwunsch! Sie sind wahrscheinlich nun Experte im Lesen von Diffs, zumindest soweit Sie es aller Voraussicht nach benötigen werden.

6.3 CVS und implizite Argumente

Sie haben vielleicht bemerkt, dass bei jedem bisher verwendeten CVS-Kommando keine Dateien in der Kommandozeile angegeben wurden. Es wurde

```
user@linux ~/ # cvs diff
```

ausgeführt anstatt

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

und

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

anstatt von

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

Das Prinzip das dahinter steht, ist, dass, wenn keine Dateinamen angegeben werden, CVS das Kommando auf alle Dateien anwendet, die dazu als sinnvoll erscheinen. Dies schließt auch Dateien in Unterverzeichnissen unterhalb des aktuellen Verzeichnisses ein; CVS durchläuft automatisch auch alle Unterverzeichnisse des Verzeichnisbaumes. Wenn zum Beispiel `b-subdir/random.c` und `a-subdir/subsubdir/fish.c` verändert wurden, wäre das Resultat von update folgendes:

```
user@linux ~/ # cvs update

cvs update: Updating .
M hello.c
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
M a-subdir/subsubdir/fish.c
cvs update: Updating b-subdir
M b-subdir/random.c

user@linux ~/ #
```

oder noch besser:

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

M hello.c
M a-subdir/subsubdir/fish.c
M b-subdir/random.c

user@linux ~/ #
```

Bemerkung
Die `-q`-Option ist eine Abschwächung der `-Q`-Option. Hätten wir `-Q` verwendet, hätte das Kommando keinerlei Ausgabe gehabt, da die Hinweise über Modifikationen als nicht essentielle Informationen gehandhabt werden. Die `-q`-Option ist weniger streng; Meldungen, die wir wahrscheinlich sowieso nicht sehen wollten, werden unterdrückt, und bestimmte nützlichere Meldungen werden durchgelassen.

Es können bei einem `Update` auch bestimmte Dateien angegeben werden:

```
user@linux ~/ # cvs update hello.c b-subdir/random.c

M hello.c
```

```
M b-subdir/random.c

user@linux ~/ #
```

Tatsächlich ist es aber üblich, update ohne Angabe bestimmter Dateien zu starten. In den meisten Fällen wird man den gesamten Verzeichnisbaum auf einmal aktualisieren wollen. Zu beachten ist, dass alle bisherigen `Updates` nur zeigten, dass einige Dateien lokal modifiziert wurden, weil sich bisher noch nichts im Archiv verändert hat. Wenn weitere Entwickler mit einem zusammen an dem Projekt arbeiten, ist es immer möglich, dass `update` Veränderungen aus dem Archiv holt und in die lokalen Dateien einfließen lässt. In diesem Fall kann es etwas nützlicher sein, die Dateien zum `Update` explizit zu benennen.

Das gleiche Prinzip kann auch auf andere CVS-Kommandos angewendet werden. Zum Beispiel können die Veränderungen für eine Datei nach der anderen mit `diff` betrachtet werden

```
user@linux ~/ # cvs diff -c b-subdir/random.c

Index: b-subdir/random.c
===================================
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.1.1.1
diff -c -r1.1.1.1 random.c
*** b-subdir/random.c 1999/04/18 18:18:22 1.1.1.1
--- b-subdir/random.c 1999/04/19 06:09:48
***************
*** 1 ****
! /* A completely empty C file. */
--- 1,8 ----
! /* Print out a random number. */
!
! #include <stdio.h>
!
! void main ()
! {
! printf ("a random number\n");
! }
```

oder es können alle Veränderungen auf einmal angezeigt werden (bleiben Sie sitzen, dies wird ein großer `Diff`):

```
user@linux ~/ # cvs -Q diff -c

Index: hello.c
===================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.1.1.1
diff -c -r1.1.1.1 hello.c
*** hello.c 1999/04/18 18:18:22 1.1.1.1
--- hello.c 1999/04/19 02:17:07
***************
*** 4,7 ****
--- 4,8 ----
main ()
{
printf ("Hello, world!\n");
```

```
+ printf ("Goodbye, world!\n");
}
Index: a-subdir/subsubdir/fish.c
===================================================================
RCS file: /usr/local/cvs/myproj/a-subdir/subsubdir/fish.c,v
retrieving revision 1.1.1.1
diff -c -r1.1.1.1 fish.c
*** a-subdir/subsubdir/fish.c 1999/04/18 18:18:22 1.1.1.1
--- a-subdir/subsubdir/fish.c 1999/04/19 06:08:50
***************
*** 1 ****
! /* A completely empty C file. */
--- 1,8 ----
! #include <stdio.h>
! void main ()
! {
! while (1) {
! printf ("fish\n");
! }
! }
Index: b-subdir/random.c
===================================================================
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.1.1.1
diff -c -r1.1.1.1 random.c
*** b-subdir/random.c 1999/04/18 18:18:22 1.1.1.1
--- b-subdir/random.c 1999/04/19 06:09:48
***************
*** 1 ****
! /* A completely empty C file. */
--- 1,8 ----
! /* Print out a random number. */
!
! #include <stdio.h>
!
! void main ()
! {
! printf ("a random number\n");
! }
```

Wie aus den `Diffs` klar hervorgeht, ist dieses Projekt produktionsreif. Machen wir also einen `Commit` der Änderungen in das Archiv.

6.4 Commit durchführen

Das `commit`-Kommando schickt Veränderungen an das Archiv. Werden keine Dateien angegeben, sendet `commit` alle Veränderungen an das Archiv; ansonsten kann einer oder können mehrere Dateinamen für den `Commit` angegeben werden (die anderen Dateien werden in diesem Fall ignoriert).

Hier wird eine Datei direkt und zwei werden indirekt an `commit` übergeben:

```
user@linux ~/ # cvs commit -m "print goodbye too" hello.c

Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v <-- hello.c
```

```
new revision: 1.2; previous revision: 1.1
done

user@linux ~/ # cvs commit -m "filled out C code"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in a-subdir/subsubdir/fish.c;
/usr/local/cvs/myproj/a-subdir/subsubdir/fish.c,v <-- fish.c
new revision: 1.2; previous revision: 1.1
done
Checking in b-subdir/random.c;
/usr/local/cvs/myproj/b-subdir/random.c,v <-- random.c
new revision: 1.2; previous revision: 1.1
done

user@linux ~/ #
```

Nehmen Sie sich einen Augenblick Zeit, um die Ausgaben sorgfältig zu lesen. Das meiste ist selbsterklärend. Es fällt jedoch auf, dass die Revisionsnummern inkrementiert wurden (wie zu erwarten war), die original Revisionen aber mit 1.1 anstatt 1.1.1.1, wie in der anfänglich erwähnten `Entries`-Datei, angezeigt wurden.

Es gibt eine Erklärung für diese Diskrepanz, auch wenn es nicht sonderlich wichtig ist. Dies betrifft die besondere Bedeutung, die CVS der Revision 1.1.1.1 beimisst. In den meisten Fällen kann man sagen, dass Dateien bei einem `Import` die Revisionsnummer 1.1 bekommen, diese aber - aus Gründen, die nur CVS weiß - als 1.1.1.1 in der `Entries`-Datei bis zum ersten `Commit` abgelegt wird.

Revisionsnummern

Jede Datei eines Projektes hat eine eigene Revisionsnummer. Wenn eine Datei wieder durch einen `Commit` zurückgesendet wird, wird die letzte Stelle der Revisionsnummer um eins inkrementiert. Daher haben die verschiedenen Dateien, die ein Projekt bilden, möglicherweise sehr unterschiedliche Revisionsnummern. Das bedeutet lediglich, dass einige Dateien öfter verändert (und durch `Commit` übertragen) wurden als andere.

(Sie werden sich vielleicht fragen, was es nun mit dem linken Teil der Revisionsnummern auf sich hat, wenn immer nur der rechte inkrementiert wird. Tatsächlich wird dieser Teil von CVS nicht automatisch inkrementiert, jedoch kann ein Benutzer dies anfordern. Dies ist eine selten benutzte Funktion und wird daher in diesem Kapitel nicht behandelt.)

Aus dem benutzten Beispielprojekt wurden gerade Veränderungen an drei Dateien durch einen `Commit` abgeschickt. Jede dieser Dateien hat nun die Revisionsnummer 1.2, jedoch haben die restlichen Dateien noch 1.1. Wird ein `Checkout` ausgeführt, werden nur die Dateien mit der höchsten Revisionsnummer geholt. Die nachfolgende Ausgabe zeigt, was der Benutzer qsmith sehen würde, wenn er zum aktuellen Zeitpunkt einen `Ckeckout` von myproject machen würde und die Revisionsnummern des Hauptverzeichnisses ausgibt:

```
user@linux ~/ # cvs -q -d :pserver:qsmith@cvs.foobar.com:/usr/local/cvs
co myproj

U myproj/README.txt
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/CVS
user@linux ~/ # cat Entries

/README.txt/1.1.1.1/Sun Apr 18 18:18:22 1999//
/hello.c/1.2/Mon Apr 19 06:35:15 1999//
D/a-subdir////
D/b-subdir////

user@linux ~/ #
```

Unter anderem hat die Datei `hello.c` nun die Revisionsnummer 1.2, während `README.txt` noch die ursprüngliche Revisionsnummer hat (Revision 1.1.1.1 oder auch 1.1).

Wenn er nun die Zeile

```
printf ("between hello and goodbye\n");
```

in `hello.c` einfügen würde und durch einen `Commit` an das Archiv sendet, wird die Revisionsnummer wiederum um eins erhöht:

```
user@linux ~/ # cvs ci -m "added new middle line"

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: 1.3; previous revision: 1.2
done

user@linux ~/ #
```

Nun hat `hello.c` die Revision 1.3, `fish.c` und `random.c` haben noch 1.2, und alle anderen Dateien haben 1.1.

Bemerkung
Hier wurde das Kommando mit `cvs ci` anstatt `cvs commit` angegeben. Die meisten CVS-Kommandos haben, um die Tipparbeit zu vereinfachen, Kurzformen. Von `checkout`, `update` und `commit` sind die Kurzformen `co`, `up` und `ci`. Eine Liste aller Kurzformen kann mit dem Befehl `cvs --help-synonyms` abgefragt werden.

In den meisten Fällen kann die Revisionsnummer einer Datei ignoriert werden. Meistens werden diese Nummern nur für interne Verwaltungsaufgaben von CVS selbst automatisch verwendet. Dennoch sind Revisionsnummern sehr nützlich, wenn man ältere Versionen einer Datei holen möchte (oder dagegen einen `Diff` macht).

Die Untersuchung der `Entries`-Datei ist nicht die einzige Möglichkeit, Revisionsnummern herauszubekommen. Dazu kann auch das `status`-Kommando verwendet werden.

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

```
===================================================
File: hello.c Status: Up-to-date

Working revision: 1.3 Tue Apr 20 02:34:42 1999
Repository revision: 1.3 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)
```

Dies gibt, wenn es ohne bestimmte Dateinamen aufgerufen wird, den Status aller Dateien eines Projektes aus:

```
user@linux ~/ # cvs status

cvs status: Examining.
=============================================
File: README.txt Status: Up-to-date

Working revision: 1.1.1.1 Sun Apr 18 18:18:22 1999
Repository revision: 1.1.1.1 /usr/local/cvs/myproj/README.txt,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

===============================================
File: hello.c Status: Up-to-date
Working revision: 1.3 Tue Apr 20 02:34:42 1999
Repository revision: 1.3 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

cvs status: Examining a-subdir
============================================
File: whatever.c Status: Up-to-date

Working revision: 1.1.1.1 Sun Apr 18 18:18:22 1999
Repository revision: 1.1.1.1 /usr/local/cvs/myproj/a-subdir/whatever.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

cvs status: Examining a-subdir/subsubdir
============================================
File: fish.c Status: Up-to-date

Working revision: 1.2 Mon Apr 19 06:35:27 1999
Repository revision: 1.2 /usr/local/cvs/myproj/
a-subdir/subsubdir/fish.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

cvs status: Examining b-subdir
============================================
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: (none)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Ignorieren Sie einfach alle Teile, die Sie nicht verstehen. Tatsächlich gilt dies grundsätzlich für CVS. Oft wird die kleine Information, die Sie benötigen, von Unmengen an Informationen flankiert, die Sie entweder gar nicht interessieren oder vielleicht auch gar nicht verstehen. Das ist völlig normal. Suchen Sie sich einfach das heraus, was Sie brauchen, und ignorieren Sie den Rest.

Im vorangegangenen Beispiel besteht der interessante Teil aus den ersten drei Zeilen (die Leerzeile nicht mitgezählt) der Statusausgabe jeder Datei. Die erste Zeile ist die wichtigste; dort stehen der Dateiname und der Status der Datei innerhalb der Arbeitskopie. Zurzeit sind alle Dateien auf dem gleichen Stand mit dem Archiv, daher steht überall der Status Up-to-date. Wenn jedoch `random.c` modifiziert und noch nicht an das Archiv mittels `Commit` übertragen worden wäre, könnte dies so aussehen:

```
===================================================
File: random.c Status: Locally Modified

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: (none)
Sticky Date: (none)
Sticky Options: (none)
```

Working revision und Repository revision zeigen an, ob die Datei mit dem Archiv übereinstimmt. Zurück bei der original Arbeitskopie (die Kopie von jrandom, welche die aktuellen Änderungen von `hello.c` noch nicht hat) wird Folgendes ausgegeben:

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

===================================================
File: hello.c Status: Needs Patch

Working revision: 1.2 Mon Apr 19 02:17:07 1999
Repository revision: 1.3 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Dies besagt nun, dass jemand eine Veränderung an `hello.c` durchgeführt und mittels `Commit` eingefügt hat und damit die Revision des Archivs zu 1.3 wurde, diese Arbeitskopie aber noch Revision 1.2 hat. Die Zeile Status: Needs Patch besagt, dass beim nächsten `Update` die Änderungen vom Archiv geholt und mittels `patch` in die Arbeitskopie eingearbeitet würden.

Nehmen wir aber zunächst an, wir wüssten nichts von den Änderungen, die qsmith an `hello.c` durchgeführt hat und führen daher auch nicht `status` oder `update` aus. Stattdessen wird die Datei ebenfalls bearbeitet und eine geringfügig andere Veränderung an der gleichen Stelle der Datei durchgeführt. Dies führt zu dem ersten Konflikt.

Konflikte erkennen und auflösen

Einen Konflikt zu erkennen ist einfach. Wird update ausgeführt, gibt CVS diesbezüglich sehr eindeutige Meldungen aus. Doch lassen Sie uns zuerst einen Konflikt erzeugen. Dazu bearbeiten wir `hello.c` und fügen folgende Zeile ein:

```
printf ("this change will conflict\n");
```

und zwar genau an der Stelle, an der qsmith

```
printf ("between hello and goodbye\n");
```

einfügte. Zu diesem Zeitpunkt ist der `Status` unserer Kopie von `hello.c`

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

===================================================
File: hello.c Status: Needs Merge

Working revision: 1.2 Mon Apr 19 02:17:07 1999
Repository revision: 1.3 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Das bedeutet, dass sowohl die Version der Datei im Archiv als auch die lokale Arbeitskopie verändert wurde und diese Veränderungen zusammengeführt werden müssen (merge). (CVS weiß noch nicht, dass diese Veränderungen einen Konflikt ergeben werden, da noch kein `Update` durchgeführt wurde.) Wird das `Update` durchgeführt, erscheint folgende Ausgabe:

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

RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in hello.c
C hello.c

user@linux ~/ #
```

Die letzte Zeile ist das kleine Geschenk von CVS. Das `C` in der ersten Spalte neben dem Dateinamen bedeutet, dass die Veränderungen zwar zusammengeführt wurden, aber ein Konflikt entstand. Die Datei `hello.c` beinhaltet nun beide Veränderungen:

```
#include <stdio.h>
void
main ()
{
 printf ("Hello, world!\n");
<<<<<< hello.c
 printf ("this change will conflict\n");
=======
printf ("between hello and goodbye\n");
>>>>>>> 1.3
printf ("Goodbye, world!\n");
}
```

Konflikte werden durch Konfliktmarkierungen in folgendem Format angezeigt:

```
<<<<<< (Dateiname)
die noch nicht durch Commit abgeschickten Änderungen der Arbeitskopie
blah blah lah

=======
die neuen Änderungen aus dem Archiv
blah blah blah
und so weiter
>>>>>>> (letzte Revisionsnummer des Archivs)
```

In der `Entries`-Datei wird ebenfalls vermerkt, dass sich die Datei derzeit in einem nur halbwegs fertigen Zustand befindet:

```
user@linux ~/ # cat CVS/Entries

/README.txt/1.1.1.1/Sun Apr 18 18:18:22 1999//
D/a-subdir////
D/b-subdir////
/hello.c/1.3/Result of merge+Tue Apr 20 03:59:09 1999//

user@linux ~/ #
```

Um den Konflikt zu beseitigen, muss die Datei so bearbeitet werden, dass der entsprechende Quelltext erhalten bleibt, die Konfliktmarkierungen entfernt werden und diese erneute Veränderung mittels `Commit` an das Archiv gesendet wird. Dies bedeutet nicht notwendigerweise, die eine Veränderung gegen die andere abwägen zu müssen; es könnte auch der gesamte Abschnitt (oder gar die gesamte Datei) neu geschrieben werden, weil eventuell beide Veränderungen nicht ausreichend sind. In diesem Fall soll die erste Veränderung den Zuschlag bekommen, jedoch mit etwas anderer Groß- und Kleinschreibung und Punktuation als die Version von qsmith:

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

```
(die Veränderungen anbringen ...)

user@linux ~/ # cat hello.c

#include <stdio.h>
void
main ()
{
printf ("Hello, world!\n");
printf ("BETWEEN HELLO AND GOODBYE.\n");
printf ("Goodbye, world!\n");
}

user@linux ~/ # cvs ci -m "adjusted middle line"

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: 1.4; previous revision: 1.3
done

user@linux ~/ #
```

6.5 Herausfinden, wer was gemacht hat: Log-Nachrichten lesen

Das Projekt hat mittlerweile einige Veränderungen durchgemacht. Möchte man nun einen Überblick darüber bekommen, was bisher geschah, so möchte man wahrscheinlich nicht jeden einzelnen `Diff` im Detail betrachten. Einfach die Log-Nachrichten durchlesen zu können, wäre ideal und kann auch einfach mit dem `log`-Kommando erreicht werden:

```
user@linux ~/ # cvs log

(Seiten über Seiten an Ausgaben ausgelassen)
```

Die Log-Ausgabe ist tendenziell etwas ausführlich. Sehen wir uns die Log-Nachrichten nur für eine Datei an:

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

RCS file: /usr/local/cvs/myproj/hello.c,v
Working file: hello.c
head: 1.4
branch:
locks: strict
access list:
symbolic names:
 start: 1.1.1.1
 jrandom: 1.1.1
keyword substitution: kv
total revisions: 5; selected revisions: 5
description:
```

```
----------------
revision 1.4
date: 1999/04/20 04:14:37; author: jrandom; state: Exp; lines: +1 -1
adjusted middle line
----------------
revision 1.3
date: 1999/04/20 02:30:05; author: qsmith; state: Exp; lines: +1 -0
added new middle line
----------------
revision 1.2
date: 1999/04/19 06:35:15; author: jrandom; state: Exp; lines: +1 -0
print goodbye too
----------------
revision 1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp;
branches: 1.1.1;
Initial revision
----------------
revision 1.1.1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp; lines: +0 -0
initial import into CVS
=============================================
```

Wie üblich, steht am Anfang eine Menge an Informationen, die einfach ignoriert werden kann. Die richtig guten Sachen kommen nach den Zeilen mit den Strichen, und das Format ist eigentlich selbsterklärend.

Wenn mehrere Dateien mit einem `Commit` abgeschickt wurden, erscheint dafür nur eine Log-Nachricht; dies kann beim Nachvollziehen von Veränderungen nützlich sein. Zum Beispiel haben wir zuvor `fish.c` und `random.c` gleichzeitig mit einem `Commit` abgeschickt. Dies geschah wie folgt:

```
user@linux ~/ # cvs commit -m "filled out C code"

Checking in a-subdir/subsubdir/fish.c;
/usr/local/cvs/myproj/a-subdir/subsubdir/fish.c,v <- fish.c
new revision: 1.2; previous revision: 1.1
done
Checking in b-subdir/random.c;
/usr/local/cvs/myproj/b-subdir/random.c,v <- random.c
new revision: 1.2; previous revision: 1.1
done
```

Das Ergebnis war, beide Dateien mit der gleichen Log-Nachricht durch `commit` abzuschicken: »filled out C code.« (So, wie es hier geschah, kamen damit beide Dateien von Revision 1.1 zu 1.2, aber dies ist nur ein Zufall. Hätte `random.c` Revision 1.29 gehabt, wäre daraus mit diesem `Commit` 1.30 geworden, und diese Revision 1.30 hätte die gleiche Log-Nachricht wie `fish.c` in der Revision 1.2 bekommen.)

Wird darauf `cvs log` angewendet, werden die gemeinsamen Log-Nachrichten angezeigt:

```
user@linux ~/ # cvs log a-subdir/subsubdir/fish.c b-subdir/random.c

RCS file: /usr/local/cvs/myproj/a-subdir/subsubdir/fish.c,v
Working file: a-subdir/subsubdir/fish.c
head: 1.2
```

```
branch:
locks: strict
access list:
symbolic names:
start: 1.1.1.1
jrandom: 1.1.1
keyword substitution: kv
total revisions: 3; selected revisions: 3
description:
----------------
revision 1.2
date: 1999/04/19 06:35:27; author: jrandom; state: Exp; lines: +8 -1
filled out C code
----------------
revision 1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp;
branches: 1.1.1;
Initial revision
----------------
revision 1.1.1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp; lines: +0 -0
initial import into CVS
=============================================
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
Working file: b-subdir/random.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
start: 1.1.1.1
jrandom: 1.1.1
keyword substitution: kv
total revisions: 3; selected revisions: 3
description:
----------------
revision 1.2
date: 1999/04/19 06:35:27; author: jrandom; state: Exp; lines: +8 -1
filled out C code
----------------
revision 1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp;
branches: 1.1.1;
Initial revision
----------------
revision 1.1.1.1
date: 1999/04/18 18:18:22; author: jrandom; state: Exp; lines: +0 -0
initial import into CVS
=============================================
```

In dieser Ausgabe kann man sehen, dass die beiden Revisionen Teil des gleichen `Commits` waren. (Die Tatsache, dass die Zeitstempel der beiden Revisionen gleich sind, oder zumindest sehr nahe beieinander liegen, ist ein weiterer Beweis.)

Log-Nachrichten durchzusehen ist eine gute Methode, um einen Überblick darüber zu bekommen, was in einem Projekt vorgegangen oder was mit einer Datei zu einem bestimmten Zeitpunkt passiert ist. Es gibt auch freie

Werkzeuge, um die rohe `cvs log`-Ausgabe in ein kompakteres und lesbareres Format umzuwandeln (wie beispielsweise das GNU ChangeLog-Format); diese Werkzeuge werden an dieser Stelle nicht behandelt, werden aber in Kapitel 10 eingeführt.

6.6 Veränderungen untersuchen und zurücknehmen

Stellen Sie sich vor, dass qsmith in den Log-Nachrichten sieht, dass jrandom die letzten Veränderungen an `hello.c` vorgenommen hat:

```
revision 1.4
date: 1999/04/20 04:14:37; author: jrandom; state: Exp; lines: +1 -1
adjusted middle line
```

und sich fragt, was jrandom getan hat. Formal gesprochen fragt sich qsmith: **Was ist der Unterschied zwischen meiner Revision (1.3) von hello.c und der darauf folgenden von jrandom (1.4)?** Dies kann mit dem `diff`-Kommando herausgefunden werden, indem nun zwei unterschiedliche Revisionen durch die zusätzliche Kommandooption `-r` verglichen werden:

```
user@linux ~/ # cvs diff -c -r 1.3 -r 1.4 hello.c

Index: hello.c
===================================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.3
retrieving revision 1.4
diff -c -r1.3 -r1.4
*** hello.c 1999/04/20 02:30:05 1.3
--- hello.c 1999/04/20 04:14:37 1.4
***************
*** 4,9 ****
 main ()
{
printf ("Hello, world!\n");
! printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
}

--- 4,9 ----
main ()
{
printf ("Hello, world!\n");
! printf ("BETWEEN HELLO AND GOODBYE.\n");
printf ("Goodbye, world!\n");
}
```

Auf diese Weise betrachtet ist die Veränderung sofort klar. Und weil die Revisionsnummern in chronologischer Reihenfolge angegeben werden (grundsätzlich eine gute Idee), wird auch der `Diff` in korrekter Reihenfolge gezeigt. Wird nur eine Revisionsnummer angegeben, benutzt CVS die Revision der aktuellen Arbeitskopie als zweites Argument.

Als qsmith diese Veränderung sieht, entscheidet er sich spontan zu Gunsten seiner eigenen Version und

beschließt, ein `undo` durchzuführen - also eine Revision zurückzugehen.

Dies bedeutet aber nicht, dass er seine Revision 1.4 verlieren möchte. Obwohl es, rein technisch gesprochen, mit CVS sicherlich möglich wäre, diesen Effekt zu erreichen, gibt es dazu meist keinen Grund. Es ist vielmehr sinnvoller, Revision 1.4 in der Historie beizubehalten und eine neue Revision 1.5 zu erzeugen, die genau wie 1.3 aussieht. So wird das `undo`-Ereignis ein Teil der Historie der Datei.

Es bleibt nur die Frage, wie der Inhalt der Revision 1.3 wiederhergestellt werden und Revision 1.5 daraus entstehen kann?

In diesem speziellen Fall könnte qsmith die Datei einfach per Hand bearbeiten, den Stand der Revision 1.3 abbilden und wieder `commit` ausführen. Wenn die Veränderungen jedoch komplexer sind (wie sie es im wahren Leben normalerweise sind), ist der Versuch, die alte Version von Hand wiederherzustellen, hoffnungslos und fehlerträchtig. Daher soll qsmith CVS benutzen, um die ältere wiederherzustellen und erneut durch `commit` an das Archiv zu senden.

Es gibt dafür zwei gleichwertige Wege: den langsamen, mühevollen Weg und den schnellen, schönen Weg. Wir werden den langsamen und mühevollen zuerst betrachten.

Die langsame Methode des Zurücknehmens

Diese Methode verwendet die `-p`-Option für update in Verbindung mit `-r`. Die `-p`-Option sorgt dafür, dass der Inhalt der angegebenen Revision auf der Standardausgabe erscheint. An sich ist dies noch nicht sonderlich hilfreich; der Inhalt der Datei rauscht über den Bildschirm, und die Arbeitskopie bleibt unverändert. Wenn jedoch die Ausgabe in die Datei umgeleitet wird, enthält die Datei wieder den Inhalt der alten Revision. Es ist, als wäre die Datei von Hand zum alten Stand **zurück bearbeitet** worden.

Zuerst muss qsmith jedoch seine Arbeitskopie mit dem Archiv abgleichen:

```
user@linux ~/ # cvs update

cvs update: Updating .
U hello.c
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir

user@linux ~/ # cat hello.c

#include <stdio.h>
void
main ()
{
printf ("Hello, world!\n");
printf ("BETWEEN HELLO AND GOODBYE.\n");
printf ("Goodbye, world!\n");
}
```

Als Nächstes führt er `update -p` aus, um sicherzustellen, dass die Revision 1.3 tatsächlich die ist, die er haben möchte:

```
user@linux ~/ # cvs update -p -r 1.3 hello.c
```

```
===================================
Checking out hello.c
RCS: /usr/local/cvs/myproj/hello.c,v
VERS: 1.3
***************
#include <stdio.h>
void
main ()
{
printf ("Hello, world!\n");
printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
}
```

Huch, da sind noch ein paar überflüssige Zeilen am Anfang der Ausgabe. Diese kamen eigentlich nicht über die Standardausgabe, sondern über Standard-Error, sind also harmlos. Nichtsdestotrotz erschweren diese das Lesen und können mit `-Q` unterdrückt werden:

```
user@linux ~/ # cvs -Q update -p -r 1.3 hello.c

#include <stdio.h>
void
main ()
{
printf ("Hello, world!\n");
printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
}
```

Nun - dies ist genau das, was qsmith bekommen wollte. Der nächste Schritt ist, die Ausgabe mit Hilfe einer Unix-Ausgabeumleitung in die Datei der Arbeitskopie zu bekommen (dies erledigt das >):

```
user@linux ~/ # cvs -Q update -p -r 1.3 hello.c > hello.c
user@linux ~/ # cvs update

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

Wenn nun `update` ausgeführt wird, wird die Datei als modifiziert markiert, was auch Sinn hat, da der Inhalt verändert wurde. Speziell ist der Inhalt der gleiche wie der der älteren Revision 1.3 (nicht dass CVS sich darüber bewusst wäre, dass diese identisch mit einer älteren Revision ist - CVS merkt nur, dass die Datei verändert wurde). Wenn qsmith ganz sichergehen wollte, könnte er einen `Diff` zur Überprüfung machen:

```
user@linux ~/ # cvs -Q diff -c

Index: hello.c
===================================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
```

```
retrieving revision 1.4
diff -c -r1.4 hello.c
*** hello.c 1999/04/20 04:14:37 1.4
--- hello.c 1999/04/20 06:02:25
***************
*** 4,9 ****
main ()
{
printf ("Hello, world!\n");
! printf ("BETWEEN HELLO AND GOODBYE.\n");
printf ("Goodbye, world!\n");
}
--- 4,9 ----
 main ()
{
printf ("Hello, world!\n");
! printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
}
```

Ja, dies ist genau das, was er wollte: eine klare Umkehrung - tatsächlich ist dies das Gegenteil des `Diff`, den er zuvor bekommen hatte. Zufrieden führt er einen `Commit` aus:

```
user@linux ~/ # cvs ci -m "reverted to 1.3 code"

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; previous revision: 1.4
done
```

Die schnelle Methode des Zurücknehmens

Die schnelle, schöne Methode des `Update` ist es, die Option `-j` (für **join**) zu dem `update`-Kommando zu verwenden. Diese Option verhält sich wie die `-r`-Option, da sie zwei Revisionsnummern als Argumente verwendet und bis zu zwei Mal `-j` angegeben werden kann.

CVS bestimmt dann die Unterschiede zwischen den beiden angegebenen Revisionen und wendet diese als Patch auf die fragliche Datei an. (Die Reihenfolge, in der die Revisionen angegeben werden, ist daher von entscheidender Bedeutung.)

Angenommen, die Kopie von qsmith ist aktuell, so folgt daraus, dass er einfach Folgendes ausführen könnte:

```
user@linux ~/ # cvs update -j 1.4 -j 1.3 hello.c

RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.4
retrieving revision 1.3
Merging differences between 1.4 and 1.3 into hello.c
```

```
user@linux ~/ # cvs update

cvs update: Updating .
M hello.c
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir

user@linux ~/ # cvs ci -m "reverted to 1.3 code" hello.c

Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v <-- hello.c
new revision: 1.5; previous revision: 1.4
done
```

Wenn nur eine Datei in einen vorherigen Zustand zurückgeführt werden soll, gibt es eigentlich keinen großen Unterschied zwischen der mühevollen und der schnellen Methode. Sie werden später im Buch sehen, dass die schnelle Methode wesentlich besser dazu geeignet ist, mehrere Dateien gleichzeitig zurückzuführen. In der Zwischenzeit können Sie einfach die Methode verwenden, die Ihnen am besten gefällt.

Zurückführung ist kein Ersatz für Kommunikation

Aller Wahrscheinlichkeit nach war das, was qsmith in diesem Beispiel getan hat, sehr gemein. Arbeitet man mit anderen Leuten an einem wirklichen Projekt und ist man der Meinung, dass jemand eine Veränderung eingebracht hat, die nicht so gut war, sollte man zuerst mit ihm oder ihr darüber reden. Vielleicht gibt es einen guten Grund für diese Veränderung, oder sie oder er hat einfach nicht genau darüber nachgedacht. Auf jeden Fall gibt es keinen Grund, diese sofort rückgängig zu machen. Alle Revisionen werden von CVS permanent gespeichert, und man kann daher jederzeit wieder zu einer älteren Revision zurückkehren, nachdem man sich mit dem entsprechend Verantwortlichen abgesprochen hat.

Wenn Sie ein Projektleiter sind und Abgabefristen einzuhalten haben oder meinen, das Recht und die Notwendigkeit dazu zu haben, dann machen Sie es so - aber schicken Sie direkt anschließend eine E-Mail an den Autor der zurückgenommenen Veränderung, und erklären Sie ihm, warum Sie es getan haben und was Ihrer Meinung nach korrigiert werden müsse, damit die Veränderung wieder einfließen kann.

7 Andere nützliche CVS-Kommandos

Zu diesem Zeitpunkt sollten Sie mit CVS schon recht gut vertraut sein. Daher werde ich an dieser Stelle mit dem Führungsstil aufhören und einige weitere nützliche Kommandos zusammenfassend erläutern.

7.1 Dateien hinzufügen

Dateien hinzuzufügen geschieht in zwei Schritten: Zuerst wird das `add`-Kommando ausgeführt und anschließend das `Commit`. Die Datei wird erst tatsächlich im Archiv erscheinen, wenn das `Commit` ausgeführt wurde:

```
user@linux ~/ # cvs add newfile.c

cvs add: scheduling file 'newfile.c' for addition
cvs add: use 'cvs commit' to add this file permanently

user@linux ~/ # cvs ci -m "added newfile.c" newfile.c

RCS file: /usr/local/cvs/myproj/newfile.c,v
done
Checking in newfile.c;
/usr/local/cvs/myproj/newfile.c,v <ó newfile.c
initial revision: 1.1
done

user@linux ~/ #
```

7.2 Verzeichnisse hinzufügen

Im Gegensatz zum Hinzufügen einer Datei verläuft das Hinzufügen eines Verzeichnisses in einem Schritt; das anschließende `Commit` entfällt hier:

```
user@linux ~/ # mkdir c-subdir
user@linux ~/ # $ cvs add c-subdir

Directory /usr/local/cvs/myproj/c-subdir added to the repository

user@linux ~/ #
```

Betrachtet man nun den Inhalt des neuen Verzeichnisses der Arbeitskopie, so sieht man, dass durch das `add`-Kommando automatisch ein CVS-Unterverzeichnis darin angelegt wurde:

```
user@linux ~/ # ls c-subdir

CVS/

user@linux ~/ # ls c-subdir/CVS

Entries Repository Root

user@linux ~/ #
```

Nun können, wie in jedem anderen Verzeichnis der Arbeitskopie, Dateien (oder neue Unterverzeichnisse) angelegt werden.

CVS und Binärdateien

Bisher habe ich ein kleines schmutziges Geheimnis von CVS ausgelassen, nämlich dass CVS Binärdateien nicht gut verwalten kann (nun, es gibt noch andere kleine schmutzige Geheimnisse von CVS, aber dies zählt bestimmt zu den schmutzigsten). Es ist nicht so, dass CVS Binärdateien gar nicht behandeln könnte; es kann dies nur nicht so elegant.

Alle Dateien, mit denen wir bisher zu tun hatten, waren einfache Textdateien. CVS benutzt einige spezielle Tricks für Textdateien. Zum Beispiel konvertiert CVS automatisch Zeilenumbrüche, wenn zwischen einem Unix-Archiv und Windows oder Macintosh Arbeitskopien ausgetauscht werden. Unter Unix ist zum Beispiel üblich, nur ein Linefeed- (LF-)Zeichen am Ende einer Zeile zu verwenden, wohingegen Windows am Ende einer Zeile ein Carriage Return (CR) und ein Linefeed (LF) erwartet. Daher haben die Dateien einer Arbeitskopie auf einem Windows-Rechner die CRLF-Kombination am Ende der Zeilen, wohingegen die Arbeitskopie des gleichen Projektes auf einem Unix-Rechner nur die LF-Zeilenenden hat (das Archiv selbst hat nur LF-Zeilenenden).

Ein weiterer Trick ist, dass CVS spezielle Zeichenketten, auch RCS-Schlüsselwörter genannt, in Textdateien erkennt und diese durch Revisionsinformationen und andere nützliche Dinge ersetzt. Wenn eine Datei beispielsweise

$Revision: 1.5 $

enthält, ersetzt CVS dies bei jedem `Commit` durch die Revisionsnummer, also könnte es beispielsweise so aussehen:

$Revision: 1.5 $

CVS aktualisiert diese Zeichenkette während der Entwicklung. (Die verschiedenen Schlüsselwörter sind in Kapitel 6 und 10 dokumentiert.)

Diese Wortersetzung ist bei Textdateien sehr nützlich, da man dadurch die Revisionsnummer und andere Informationen direkt beim Bearbeiten sehen kann. Doch was passiert, wenn die Datei ein JPEG-Bild ist? Oder ein übersetztes ausführbares Programm? In dieser Art von Dateien könnte CVS erheblichen Schaden anrichten, wenn es einfach blind alle Schlüsselwörter, die es findet, ersetzt. In Binärdateien können solche Zeichenketten einfach zufällig auftauchen.

Daher muss, wenn eine Binärdatei hinzugefügt werden soll, CVS mitgeteilt werden, sowohl die Schlüsselwortersetzung als auch die Zeilenendenumwandlung zu unterlassen. Dies erfolgt mit der Option `-kb`:

```
user@linux ~/ # cvs add -kb filename
user@linux ~/ # cvs ci -m "added blah" filename

(etc)
```

In manchen Fällen, wie bei Textdateien, die wahrscheinlich verstreute Schlüsselwörter enthalten könnten, kann es sinnvoll sein, nur die Schlüsselwortersetzung auszuschalten. Dies geschieht mit der Option `-ko`:

```
user@linux ~/ # cvs add -ko filename
user@linux ~/ # cvs ci -m "added blah" filename
```

```
(etc)
```

(Tatsächlich wäre dieses Kapitel schon wegen des darin enthaltenen Beispiels **$Revision: 1.5 $** ein Fall für eine solche Textdatei.)

Zu bemerken ist auch, dass kein aussagekräftiger `cvs diff` zwischen zwei Revisionen einer Binärdatei durchgeführt werden kann. `Diff` benutzt einen textbasierten Algorithmus, der bei Binärdateien lediglich die Aussage treffen kann, ob sich diese unterscheiden, nicht aber worin. Zukünftige Versionen von CVS werden vielleicht einen binären `Diff` unterstützen.

7.3 Dateien entfernen

Eine Datei zu entfernen ist ähnlich, wie eine hinzuzufügen, bis auf einen zusätzlichen Schritt: Die Datei muß zuerst aus der Arbeitskopie entfernt werden:

```
user@linux ~/ # rm newfile.c
user@linux ~/ # cvs remove newfile.c

cvs remove: scheduling 'newfile.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently

user@linux ~/ # cvs ci -m "removed newfile.c" newfile.c

Removing newfile.c;
/usr/local/cvs/myproj/newfile.c,v <- newfile.c
new revision: delete; previous revision: 1.1
done

user@linux ~/ #
```

Zu beachten ist, dass bei dem zweiten und dritten Kommando `newfile.c` explizit angegeben wird, obwohl dies in der Arbeitskopie gar nicht mehr existiert. Natürlich muss man dies bei dem `Commit` nicht unbedingt, solange man nichts dagegen hat, dass dann auch weitere Dateien in den `Commit` einbezogen werden.

7.4 Verzeichnisse entfernen

Wie schon zuvor erwähnt, stehen Verzeichnisse nicht unter der Versionskontrolle von CVS. Stattdessen, als eine Art billiger Ersatz, bietet es eine Reihe seltsamer Verhaltensweisen, die meistens **das Richtige** ausführen. Eine dieser Seltsamkeiten ist, dass leere Verzeichnisse besonders behandelt werden können. Soll ein Verzeichnis aus einem Projekt entfernt werden, werden zuerst alle Dateien daraus entfernt:

```
user@linux ~/ # cd dir
user@linux ~/ # rm file1 file2 file3
user@linux ~/ # cvs remove file1 file2 file3

(Ausgabe ausgelassen)

user@linux ~/ # cvs ci -m "removed all files" file1 file2 file3

(Ausgabe ausgelassen)
```

und dann in dem übergeordneten Verzeichnis update mit der `-P`-Option ausgeführt:

```
user@linux ~/ # cd ..
user@linux ~/ # cvs update -P

(Ausgabe ausgelassen)
```

Die `-P`-Option bedeutet für update, leere Verzeichnisse zu reduzieren - diese also aus der Arbeitskopie zu entfernen. Ist dies einmal ausgeführt, kann das Verzeichnis als gelöscht angesehen werden; alle Dateien sind weg und das Verzeichnis selbst ebenfalls (zumindest in der Arbeitskopie, dennoch existiert ein leeres Verzeichnis in dem Archiv).

Ein interessantes Gegenstück zu diesem Verhalten ist, dass CVS bei einem einfachen `update` keine neuen Verzeichnisse aus dem Archiv in die Arbeitskopie einfügt. Es gibt dafür eine Reihe von Begründungen, von denen an dieser Stelle keine besonders erwähnenswert ist. Kurz zusammengefasst kann man sagen, dass Sie von Zeit zu Zeit `update` mit der Option `-p` ausführen sollten, damit neue Verzeichnisse aus dem Archiv in Ihre Arbeitskopie eingefügt werden.

7.5 Dateien und Verzeichnisse umbenennen

Eine Datei umzubenennen ist das Gleiche, wie diese zu löschen und unter einem neuen Namen anzulegen. Unter Unix sind die Befehle dazu:

```
user@linux ~/ # cp oldname newname
user@linux ~/ # rm oldname
```

Und hier ist das CVS-Äquivalent:

```
user@linux ~/ # mv oldname newname
user@linux ~/ # cvs remove oldname

(Ausgabe ausgelassen)

user@linux ~/ # cvs add newname

(Ausgabe ausgelassen)

user@linux ~/ # cvs ci -m "renamed oldname to newname" oldname newname

(Ausgabe ausgelassen)

user@linux ~/ #
```

Bezüglich Dateien ist das alles, was zu tun ist. Verzeichnisse umzubenennen ist nicht sonderlich anders: das neue Verzeichnis anlegen, `cvs add` ausführen, alle Dateien des alten Verzeichnisses in das neue bewegen, `cvs remove` ausführen, um diese aus dem alten Verzeichnis zu entfernen, `cvs add` ausführen, um diese in dem neuen Verzeichnis hinzuzufügen, `cvs commit` ausführen, damit auch alles dem Archiv mitgeteilt wird, und dann `cvs update -P` ausführen, damit das nun leere Verzeichnis auch aus der Arbeitskopie verschwindet. Also:

```
user@linux ~/ # mkdir newdir
```

```
user@linux ~/ # cvs add newdir
user@linux ~/ # mv olddir/* newdir

mv: newdir/CVS: cannot overwrite directory

user@linux ~/ # cd olddir
user@linux ~/ # cvs rm foo.c bar.txt
user@linux ~/ # cd ../newdir
user@linux ~/ # cvs add foo.c bar.txt
user@linux ~/ # cd ..
user@linux ~/ # cvs commit -m "moved foo.c and bar.txt from olddir to
newdir"
user@linux ~/ # cvs update -P
```

Bemerkung
Beachten Sie die Warnmeldung nach dem dritten Befehl. Diese besagt, dass das CVS/Unterverzeichnis von `olddir` nicht in `newdir` kopiert werden kann, da in `newdir` schon ein solches existiert. Dies ist auch gut so, da man sowieso das CVS/ Unterverzeichnis in `newdir` unverändert beibehalten möchte.

Ganz offensichtlich ist das Verschieben von Dateien etwas umständlich. Die beste Methode ist es, schon beim ersten `Import` des Projektes ein gutes Verzeichnislayout zu haben, sodass später möglichst selten ganze Verzeichnisse verschoben werden müssen. Später werden Sie eine etwas drastischere Methode zum Verschieben von Verzeichnissen kennenlernen, welche die Veränderungen direkt im Archiv vornimmt. Diese Methode sollte jedoch für Notfälle aufgehoben werden; nach Möglichkeit sollte alles mit CVS-Operationen innerhalb der Arbeitskopie behandelt werden.

7.6 Optionsmüdigkeit vermeiden

Die meisten Benutzer sind es recht bald leid, zu jedem Befehl immer wieder die gleichen Optionen eingeben zu müssen. Wenn man im Vorhinein weiß, dass immer die globale Option `-Q` oder die Option `-c` im Zusammenhang mit `diff` angegeben werden soll, warum sollte dies dann immer wieder eingegeben werden müssen?

Doch dafür gibt es glücklicherweise Abhilfe. CVS überprüft dazu die Datei `.cvsrc` im Home-Verzeichnis des Benutzers. In dieser Datei können standardmäßige Optionen zu bestimmten Kommandos angegeben werden, die immer ausgeführt werden, wenn CVS aufgerufen wird. Hier eine solche beispielhafte Datei:

.cvsrc

```
diff -c
update -p
cvs -q
```

Entspricht die linke Spalte einem angegebenen CVS-Kommando (in der nicht gekürzten Form), werden die entsprechenden Optionen jedes Mal, wenn CVS verwendet wird, angewendet. Globale Optionen können mit cvs angegeben werden. In diesem Beispiel wird also jedes Mal wenn `diff` ausgeführt wird, die Option `-c` automatisch mit ausgeführt.

7.7 Momentaufnahmen (Zeitstempel und Marken)

Gehen wir noch einmal zu dem Beispiel zurück, in dem ein Programm gerade nicht lauffähig ist, wenn eine bestimmte Fehlerbeschreibung eines Benutzers eintrifft und diese daher nicht überprüft werden kann. Der Entwickler braucht dann plötzlich Zugriff auf das gesamte Projekt in dem Zustand, in dem es war, als die letzte Version freigegeben wurde. Viele Dateien sind seitdem verändert worden, und die meisten Revisionsnummern unterscheiden sich. Es wäre viel zu aufwändig, die gesamten Log-Nachrichten durchzulesen, um herauszufinden, welche Revisionsnummer eine jede Datei zum Zeitpunkt der letzten Freigabe einer Version hatte, um dann `update` (unter Angabe einer Revisionsnummer mittels `-r`) auf jede Datei anzuwenden. Bei mittleren oder großen Projekten (einige Dutzend bis zu Tausende Dateien) wäre dies ein hoffnungsloser Versuch.

CVS bietet daher die Möglichkeit, vorangegangene Revisionen aller Dateien auf einmal zu holen. Tatsächlich gibt es dafür zwei Methoden: nach Datum, dies wählt die zu holende Revision basierend auf dem Datum des Zeitpunkts ihres `Commit` aus, und mit Hilfe von Marken, was eine Momentaufnahme eine Projektes holt, die durch eine Marke gekennzeichnet wurde.

Welche der beiden Methoden zum Einsatz kommt, hängt von der Situation ab. Das datumsbasierte Holen einer Revision wird durch die Option `-D` zu `update` erreicht, die ähnlich zu `-r` ist, aber als Argument ein Datum und nicht eine Revisionsnummer benötigt:

```
user@linux ~/ # cvs -q update -D "1999-04-19"

U hello.c
U a-subdir/subsubdir/fish.c
U b-subdir/random.c

user@linux ~/ #
```

Mit der `-D`-Option holt update die höchste Revision einer jeden Datei eines gegebenen Datums und überführt, wenn nötig, die Dateien der Arbeitskopie in vorangegangene Revisionen.

Zusätzlich zu einem Datum kann, und sollte meistens auch, eine Uhrzeit angegeben werden. Zum Beispiel endete das vorangegangene Beispiel darin, vor allem die Revision 1.1 zu holen (nur drei der Dateien waren verändert, da alle anderen noch Revision 1.1 hatten). Als Beweis hier die Statusausgabe für die Datei `hello.c`:

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

===================================================================
File: hello.c Status: Up-to-date
Working revision: 1.1.1.1 Sat Apr 24 22:45:03 1999
Repository revision: 1.1.1.1 /usr/local/cvs/myproj/hello.c,v
Sticky Date: 99.04.19.05.00.00

user@linux ~/ #
```

Ein Blick in die Log-Nachrichten zeigt jedoch, dass Revision 1.2 von `hello.c` definitiv am 19. April 1999 durch einen `Commit` entstand. Also warum wurde jetzt Revision 1.1 anstatt 1.2 geholt?

Das Problem ist, dass das Datum **1999-04-19** als **Mitternacht beginnend am 19.4.1999** interpretiert wurde - also der erste Zeitpunkt dieses Tages. Das ist wahrscheinlich nicht das, was man möchte. Das Commit der Revision 1.2 fand später an diesem Tag statt. Durch nähere Spezifizierung des Datums kann auch Revision 1.2 geholt werden:

```
user@linux ~/ # cvs -q update -D "1999-04-19 23:59:59"
```

```
U hello.c
U a-subdir/subsubdir/fish.c
U b-subdir/random.c

user@linux ~/ # cvs status hello.c

===================================
File: hello.c Status: Locally Modified
Working revision: 1.2 Sat Apr 24 22:45:22 1999
Repository revision: 1.2 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: 99.04.20.04.59.59
Sticky Options: (none)

user@linux ~/ #
```

Wir sind fast am Ziel. Betrachten wir nun Datum und Uhrzeit in der Zeile Sticky Date näher, scheint dort 4:59:59 a.m. Uhr zu stehen und nicht 11:59 Uhr, wie es durch den Befehl angefordert wurde (wir kommen später dazu, was **sticky** bedeutet). Wie Sie vielleicht schon erraten haben, liegt dieser Unterschied in der Differenz zwischen der lokalen Zeit und der Universal Coordinated Time (auch **Greenwich Mean Time** genannt) begründet. Im Archiv werden alle Zeitstempel immer in Universal Time gespeichert, CVS benutzt jedoch auf der Seite des Clients die lokale Zeitzone. Im Falle von `-D` ist dies etwas unglücklich, da man wahrscheinlich mit den Daten und Zeiten des Archivs vergleichen möchte und die Systemzeit des lokalen Systems egal ist. Dies kann umgangen werden, indem bei dem Befehlsaufruf zusätzlich die GMT-Zeitzone angegeben wird:

```
user@linux ~/ # cvs -q update -D "1999-04-19 23:59:59 GMT"

U hello.c

user@linux ~/ # cvs -q status hello.c

===================================================
File: hello.c Status: Up-to-date
Working revision: 1.2 Sun Apr 25 22:38:53 1999
Repository revision: 1.2 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: 99.04.19.23.59.59
Sticky Options: (none)

user@linux ~/ #
```

Endlich - dies brachte nun die Arbeitskopie auf den letzten, durch `Commit` erreichten Stand vom 19. April (es sei denn, es hätte noch weitere Beiträge durch `Commit` in der letzten Sekunde des Tages gegeben, was aber nicht der Fall ist).

Was passiert, wenn nun update ausgeführt wird?

```
user@linux ~/ # cvs update

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

user@linux ~/ #
```

Es passiert gar nichts. Wir wissen aber, dass es aktuellere Versionen der letzten drei Dateien gibt. Warum sind diese nicht in der Arbeitskopie enthalten?

Genau dies ist die Bedeutung von **sticky** (bindend). Eine Aktualisierung (**Rückterminierung**?) mit der `-D`-Option bewirkt, dass die Arbeitskopie auf dieses vergangene Datum festgelegt wird. In der Terminologie von CVS spricht man davon, dass für diese Arbeitskopie ein **bindendes Datum** gesetzt wurde. Hat eine Arbeitskopie einmal eine bindenden Eigenschaft bekommen, bleibt diese so lange erhalten, bis sie explizit zurückgenommen wird. Daher werden nun folgende `Updates` nicht mehr die aktuellste Version holen. Stattdessen bleiben diese bei diesem bindenden Datum. Bindende Eigenschaften können mit dem `cvs status`-Kommando angezeigt oder direkt in der `CVS/Entries`-Datei nachgelesen werden:

```
user@linux ~/ # cvs -q update -D "1999-04-19 23:59:59 GMT"

U hello.c

user@linux ~/ # cat CVS/Entries

D/a-subdir////
D/b-subdir////
D/c-subdir////
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 1999//D99.04.19.23.59.59
/hello.c/1.2/Sun Apr 25 23:07:29 1999//D99.04.19.23.59.59

user@linux ~/ #
```

Sollte man nun `hello.c` modifiziert haben und einen `Commit` versuchen:

```
user@linux ~/ # cvs update

M hello.c

user@linux ~/ # cvs ci -m "trying to change the past"

cvs commit: cannot commit with sticky date for file 'hello.c'
cvs [commit aborted]: correct above errors first!

user@linux ~/ #
```

so würde CVS das nicht zulassen, da dies so wäre, als erlaube man, in der Zeit zurück zu reisen und die Vergangenheit zu ändern. CVS basiert auf chronologischen Aufzeichnungen und kann dies daher nicht zulassen.

Das bedeutet aber nicht, dass CVS auf einmal nichts mehr von allen anderen Revisionen weiß, die seitdem per `Commit` eingeflossen sind. Man kann immer noch die mit einem bindenden Datum versehene Arbeitskopie mit anderen Revisionen vergleichen, zukünftige eingeschlossen:

```
user@linux ~/ # cvs -q diff -c -r 1.5 hello.c

Index: hello.c
===================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.5
diff -c -r1.5 hello.c
*** hello.c 1999/04/24 22:09:27 1.5
--- hello.c 1999/04/25 00:08:44
```

```
***************
*** 3,9 ****
void
main ()
{
printf ("Hello, world!\n");
- printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
}
--- 3,9 ----
 void
main ()
{
+ /* this line was added to a downdated working copy */
printf ("Hello, world!\n");
printf ("Goodbye, world!\n");
}
```

Der `Diff` zeigt auf, dass, ausgehend vom 19. April 1999, die Zeile **between hello and goodbye** noch nicht hinzugefügt wurde. Er zeigt auch die Modifikation, die wir in der Arbeitskopie gemacht haben (der in dem vorangegangenen Quelltextauszug gezeigte zusätzliche Kommentar).

Das bindende Datum sowie alle anderen bindenden Eigenschaften können mit der Option `-A` (`-A` steht für **reset**, fragen Sie mich nicht, warum) zu `update` entfernt werden, was die Arbeitskopie wieder auf den aktuellsten Stand bringt:

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

U hello.c

user@linux ~/ # cvs status hello.c

===================================
File: hello.c Status: Up-to-date
Working revision: 1.5 Sun Apr 25 22:50:27 1999
Repository revision: 1.5 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Gültige Datumsformate

CVS akzeptiert eine breite Auswahl an Datumsformaten. Mit dem ISO 8691-Format liegt man nie falsch (also Standard #8601 der International Standards Organization, siehe auch www.saqqara.demon.co.uk/datefmt.htm), das auch in den vorangegangenen Beispielen verwendet wurde. Es können auch die E-Mail-Formate für Datum und Uhrzeit verwendet werden, die in RFC 822 und RFC 1123 (siehe www.rfc-editor.org/rfc/) beschrieben sind. Letztendlich können eindeutige Konstruktionen der englischen Datumsformate verwendet werden, um Daten relativ zum aktuellen Datum zu beschreiben.

Sie werden sicherlich nicht alle möglichen Formate benötigen, dennoch hier noch ein paar Beispiele, um Ihnen eine Vorstellung davon zu geben, welche Formate CVS akzeptiert:

```
user@linux ~/ # cvs update -D "19 Apr 1999"
user@linux ~/ # cvs update -D "19 Apr 1999 20:05"
user@linux ~/ # cvs update -D "19/04/1999"
user@linux ~/ # cvs update -D "3 days ago"
user@linux ~/ # cvs update -D "5 years ago"
user@linux ~/ # cvs update -D "19 Apr 1999 23:59:59 GMT"
user@linux ~/ # cvs update -D "19 Apr"
```

Die Anführungszeichen dienen lediglich dazu, der Unix-Shell mitzuteilen, dass es sich jeweils um ein Argument handelt, obwohl Leerzeichen enthalten sind. Die Anführungszeichen stören auch dann nicht, wenn das Datum keine Leerzeichen enthält. Daher ist es sinnvoll, immer welche zu verwenden.

Einen Zeitpunkt markieren (Marken)

Mittels eines Datums Dateien wieder zu holen ist nützlich, wenn lediglich der zeitliche Verlauf von hauptsächlichem Interesse ist. Öfter jedoch soll ein Projekt wieder in einen Zustand gebracht werden, in dem es zu einem bestimmten Ereignis war, beispielsweise dem Zeitpunkt einer Veröffentlichung einer bekanntermaßen stabilen Version oder dem Zeitpunkt, zu dem größere Teile hinzugefügt oder weggenommen wurden.

Sich diese speziellen Daten einfach zu merken oder diese anhand der Log-Dateien wiederzufinden, wäre ein langwieriger und schwieriger Prozess. Vermutlich wurde ein solcher Zeitpunkt, weil er wichtig war, in der Revisionshistorie markiert. CVS bietet dazu das Markieren (engl. **tagging**) an.

Markierungen unterscheiden sich von einem normalen `Commit`, indem keine Veränderungen an den Quelltexten an sich gespeichert werden, sondern lediglich eine Veränderung der Einschätzung der Dateien durch den Entwickler. Eine Markierung zeichnet eine Gruppe von Revisionen, repräsentiert durch die Arbeitskopie eines Entwicklers, besonders aus (für gewöhnlich ist dabei diese Arbeitskopie vollständig aktuell, sodass der Name dieser Markierung der **letzten und höchsten** Revision des Archivs hinzugefügt wird).

Eine Markierung zu setzen ist einfach:

```
user@linux ~/ # cvs -q tag Release-1999_05_01

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

user@linux ~/ #
```

Dieser Befehl verbindet den symbolischen Namen **Release-1999_05_01** mit der Momentaufnahme, die durch die aktuelle Arbeitskopie repräsentiert wird. Formell definiert bezeichnet eine Momentaufnahme eine Gruppe von Dateien und ihre Revisionsnummern innerhalb des Projektes. Diese Revisionsnummern müssen nicht in allen Dateien gleich sein, was sie für gewöhnlich auch nicht sind. Wäre zum Beispiel eine Markierung in dem Beispielprojekt, das wir in diesem ganzen Kapitel verwendet haben, gesetzt worden und die Arbeitskopie wäre auf dem letzten Stand, dann wäre der symbolische Name **Release-1999_05_01** zu `hello.c` mit Revision 1.5, `fish.c` mit Revision 1.2, `random.c` mit Revision 1.2 und allen anderen mit Revision 1.1 hinzugefügt worden.

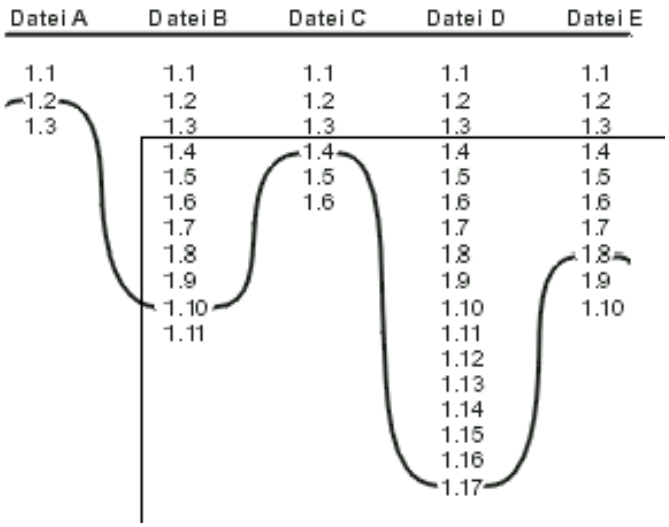


Bild Kap_02-1.png

Wie eine Markierung in Relation zur Revisionshistorie eines Projektes stehen kann.

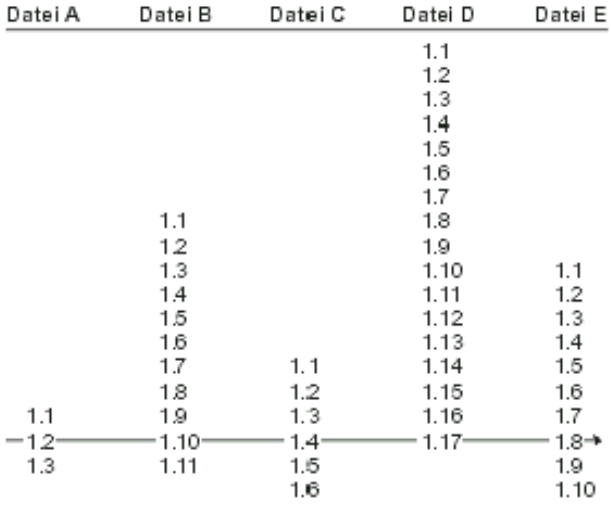


Bild Kap_02-2.png

Eine Markierung ist eine **Gerade Aussicht** durch die Revisionshistorie.

Vielleicht ist es hilfreich, sich eine Markierung als einen Pfad oder eine **Schnur** vorzustellen, welche die verschiedenen Revisionen der Dateien miteinander verbindet.

Wird die Schnur gerade gezogen und sieht man direkt an ihr entlang, so sieht man einen bestimmten Moment aus der Projektgeschichte - nämlich genau den Moment, zu dem die Markierung gesetzt wurde (Abbildung 2.2).

Werden nun weiter Dateien verändert und die Veränderung durch einen `Commit` dem Archiv zur Verfügung gestellt, so wird die Markierung nicht mit den wachsenden Revisionsnummern mit bewegt. Diese bleibt fest, gebunden zu der Revisionsnummer einer jeden Datei, zu der diese Markierung gesetzt wurde.

Durch ihre erläuternde Bedeutung ist es etwas unglücklich, dass Markierungen keine langen Nachrichten oder ganze Paragraphen mit Fließtext enthalten können. In dem vorangegangenen Beispiel besagte die Markierung einfach und offensichtlich, dass sich das Projekt zu diesem Datum in einem zu veröffentlichenden Zustand befand. Manchmal möchte man jedoch komplexere Zustände markieren, was in sehr unvorteilhaften Markierungen mündet:

```
user@linux ~/ # cvs tag testing-release-3_pre-19990525-public-release
```

Als allgemeine Regel sollte gelten, Markierungsnamen so knapp wie möglich zu halten, aber trotzdem alle notwendigen Informationen über das spezielle Ereignis, das aufgezeichnet werden soll, zu enthalten. Seien Sie im Zweifelsfall lieber zu ausführlich - Sie werden es sich später selbst danken, wenn Sie anhand eines Markierungsnamens exakt aussagen können, was denn genau aufgezeichnet wurde (oder werden sollte).

Ihnen ist vielleicht schon aufgefallen, dass keine Punkte oder Leerzeichen in den Markierungsnamen enthalten waren. CVS ist in der Bewertung, was einen gültigen Markierungsnamen darstellt, ziemlich streng. Die Regeln besagen, dass dieser mit einem Buchstaben beginnen muss und Buchstaben, Ziffern, Bindestriche (-) und Unterstriche (_) enthalten darf. Es dürfen keine Leerzeichen, Punkte, Doppelpunkte, Kommas oder andere Symbole verwendet werden.

Um eine Momentaufnahme mittels eines Markierungsnamens aus dem Archiv zu holen, wird dieser wie eine Revisionsnummer benutzt. Es gibt zwei Möglichkeiten, eine Momentaufnahme zu bekommen: Man kann eine neue Arbeitskopie mit einer bestimmten Markierung auschecken (`Checkout`), oder man kann eine existierende Arbeitskopie mit Hilfe der Markierung dahin überführen. Beide Wege führen zu einer Arbeitskopie, deren Dateien den Revisionen entsprechen, die durch die Markierung spezifiziert wurden.

Meistens wird man versuchen, einen Blick in das Projekt in dem Zustand zu werfen, in dem es zum Zeitpunkt der Markierung war. Dies wird man nicht notwendigerweise mit seiner Hauptarbeitskopie machen wollen, die wahrscheinlich noch nicht per `commit` abgeschickte Veränderungen enthält. Nehmen wir also an, es soll eine separate Arbeitskopie per `Checkout` unter Verwendung der Markierung geholt werden. Dies geschieht folgendermaßen (stellen Sie jedoch sicher, dass Sie dies irgendwo anders als in Ihrer existierenden Arbeitskopie oder dem darüber liegenden Verzeichnis ausführen!):

```
user@linux ~/ # cvs checkout -r Release-1999_05_01 myproj

cvs checkout: Updating myproj
U myproj/README.txt
U myproj/hello.c
cvs checkout: Updating myproj/a-subdir
U myproj/a-subdir/whatever.c
cvs checkout: Updating myproj/a-subdir/subsubdir
U myproj/a-subdir/subsubdir/fish.c
cvs checkout: Updating myproj/b-subdir
U myproj/b-subdir/random.c
cvs checkout: Updating myproj/c-subdir
```

Die `-r`-Option wurde bereits für das `update`-Kommando verwendet und spezifizierte dort eine Revisionsnummer. Eine Markierung verhält sich in vielen Belangen wie eine Revisionsnummer, da eine

bestimmte Markierung für eine bestimmte Datei genau einer Revisionsnummer entspricht. (Es ist grundsätzlich nicht möglich, zwei gleich lautende Markierungen innerhalb eines Projektes zu verwenden.) Tatsächlich kann man überall dort, wo auch eine Revisionsnummer benutzt werden kann, einen Markierungsnamen als Teil eines CVS-Befehls verwenden (vorausgesetzt, die Markierung wurde vorher gesetzt). Soll ein `Diff` des aktuellen Zustands einer Datei relativ zu einem Zustand einer veröffentlichten Version gemacht werden, kann dies so erfolgen:

```
user@linux ~/ # cvs diff -c -r Release-1999_05_01 hello.c
```

Und wenn vorübergehend zu dieser Revision zurückgegangen werden soll, geht dies so:

```
user@linux ~/ # cvs update -r Release-1999_05_01 hello.c
```

Die Austauschbarkeit von Revisionsnummern mit Markierungen ist ein Grund für die strengen Regeln der zulässigen Namen. Stellen Sie sich einmal vor, dass Punkte in den Namen erlaubt wären; es könnte dann eine Markierung geben, die **1.3** heißt und zu einer tatsächlichen Revisionsnummer **1.47** gehören soll. Würde dann Folgendes ausgeführt

```
user@linux ~/ # cvs update -r 1.3 hello.c
```

wie sollte CVS dann wissen, ob sich dies nun auf die Markierung **1.3** oder die viel frühere Revision **1.3** von `hello.c` bezieht? Daher werden die Markierungsnamen derart eingeschränkt und können so einfach von Revisionsnummern unterschieden werden. Eine Revisionsnummer hat einen Punkt, eine Markierung nicht. (Es gibt auch für die anderen Einschränkungen Gründe, und die meisten haben damit zu tun, dass dadurch die Markierungsnamen für CVS einfacher zu lesen sind.)

Wie Sie sich sicherlich haben denken können, besteht die zweite Methode, eine Momentaufnahme zu bekommen - also eine bestehende Arbeitskopie in die markierte Revision zu überführen - ebenfalls darin, `update` auszuführen:

```
user@linux ~/ # cvs update -r Release-1999_05_01

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

user@linux ~/ #
```

Der vorangegangene Befehl ist der gleiche wie der, der verwendet wurde, um `hello.c` zu Release-1999_05_01 zurückzuführen, bis darauf, dass der Dateiname ausgelassen wurde, da das gesamte Projekt zurückgeführt werden soll. (Sie können auch, wenn Sie dies wollen, nur einen Zweig der Verzeichnisstruktur des Projektes zurückführen, indem der vorangegangene Befehl in dem entsprechenden Unterverzeichnis anstatt im Hauptverzeichnis ausgeführt wird, obwohl Sie dies wohl nie richtig wollen werden.)

Beachten Sie, dass anscheinend bei dem `Update` keine Dateien verändert wurden. Die Arbeitskopie war völlig aktuell, als die Marke gesetzt wurde, und es wurden seitdem keine Veränderungen vorgenommen.

Dies bedeutet jedoch nicht, dass sich gar nichts verändert hat. Die Arbeitskopie ist nun markiert. Wird nun eine Veränderung gemacht und versucht, diese mit `commit` an das Archiv zu schicken (nehmen wir an, es wurde

`hello.c` verändert):

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

M hello.c

user@linux ~/ # cvs -q ci -m "trying to commit from a working copy on a
tag"

cvs commit: sticky tag 'Release-1999_05_01' for file 'hello.c' is not
a branch
cvs [commit aborted]: correct above errors first!

user@linux ~/ #
```

so lässt CVS dies nicht zu. (Kümmern Sie sich erst einmal nicht um die genaue Bedeutung obiger Fehlermeldung - wir werden Verzweigungen als Nächstes in diesem Kapitel behandeln). Es spielt dabei keine Rolle, ob die Markierung aus einem `Checkout` oder `Update` resultiert. Ist diese einmal markiert, sieht CVS die Arbeitskopie als eine statische Momentaufnahme der Vergangenheit an und erlaubt Ihnen nicht mehr, die Vergangenheit zu ändern, zumindest nicht so einfach. Wird `cvs status` ausgeführt oder wenn Sie sich die `CVS/Entries`-Datei ansehen, werden Sie feststellen, dass für jede Datei eine bindende Markierung (**sticky tag**) gesetzt ist. Zum Beispiel ist dies die Haupt-Entries-Datei:

```
user@linux ~/ # cat CVS/Entries

D/a-subdir////
D/b-subdir////
D/c-subdir////
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 1999//TRelease-1999_05_01
/hello.c/1.5/Tue Apr 20 07:24:10 1999//TRelease-1999_05_01

user@linux ~/ #
```

Markierungen werden genau wie andere bindende Eigenschaften mit der `-A`-Option von `update` entfernt:

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

M hello.c

user@linux ~/ #
```

Die Veränderungen an `hello.c` gehen jedoch nicht verloren; CVS erkennt immer noch, dass die Datei bezüglich des Archivs verändert wurde:

```
user@linux ~/ # cvs -q diff -c hello.c

Index: hello.c
===================================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.5
diff -c -r1.5 hello.c
*** hello.c 1999/04/20 06:12:56 1.5
--- hello.c 1999/05/04 20:09:17
***************
```

```
*** 6,9 ****
--- 6,10 ----
printf ("Hello, world!\n");
printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
+ /* a comment on the last line */
}

user@linux ~/ #
```

Nun, da durch `update` alle bindenden Eigenschaften entfernt wurden, akzeptiert CVS auch wieder einen `Commit`:

```
user@linux ~/ # cvs ci -m "added comment to end of main function"

cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
cvs commit: Examining c-subdir
Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v <- hello.c
new revision: 1.6; previous revision: 1.5
done

user@linux ~/ #
```

Die Markierung **Release-1999_05_01** gehört selbstverständlich immer noch zu Revision 1.5. Vergleichen Sie den Status der Datei vor und nach dieser Umkehrung zu dieser Markierung:

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

===================================================
File: hello.c Status: Up-to-date
Working revision: 1.6 Tue May 4 20:09:17 1999
Repository revision: 1.6 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ # cvs -q update -r Release-1999_05_01

U hello.c

user@linux ~/ # cvs -q status hello.c

===================================================
File: hello.c Status: Up-to-date
Working revision: 1.5 Tue May 4 20:21:12 1999
Repository revision: 1.5 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: Release-1999_05_01 (revision: 1.5)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Nun, da ich Ihnen gesagt habe, dass CVS Sie die Geschichte nicht verändern lässt, zeige ich Ihnen, wie Sie die Geschichte verändern können.

7.8 Verzweigungen

Bisher wurde CVS als eine intelligente und koordinierende Bibliothek betrachtet. Man kann sich CVS aber auch als eine Zeitmaschine vorstellen (Danke schön an *Jim Blandy* für diese Analogie). Bisher haben wir nur gesehen, wie die Vergangenheit mit CVS betrachtet werden kann, ohne darauf irgendeinen Einfluss zu nehmen. Doch wie alle guten Zeitmaschinen erlaubt CVS jedoch auch in der Zeit zurückzugehen und die Vergangenheit zu verändern. Was ist das Resultat daraus? Wie jeder Science Fiction-Fan weiß, ist die Antwort darauf: ein weiteres Universum, parallel zu unserem, aber genau an dem Punkt divergierend, an dem die Vergangenheit verändert wurde. Eine Verzweigung im Sinne von CVS spaltet die Entwicklung eines Projektes in getrennte, parallele Historien. Veränderungen in einem Zweig betreffen den anderen nicht mehr.

Warum ist dies nützlich?

Kehren wir für einen Moment noch einmal zu dem Szenario zurück, in dem ein Entwickler, mitten in der Entwicklung einer neuen Version eines Programmes, eine Fehlerbeschreibung über eine ältere Version bekommt. Angenommen, der Entwickler behebt das Problem, so muss er diese Korrektur immer noch dem Kunden zukommen lassen. Es ist sicherlich nicht sonderlich hilfreich, eine ältere Kopie ausfindig zu machen, darin den Fehler zu beheben, ohne dies CVS wissen zu lassen, und das Ganze dem Kunden zu schicken. Es gäbe anschließend keinerlei Aufzeichnung der durchgeführten Änderung; CVS hätte keine Informationen darüber; und wenn später festgestellt würde, dass auch die Fehlerbehebung einen Fehler hat, hätte niemand einen Ansatzpunkt, um das Problem zu reproduzieren

Noch schlimmer wäre es, den Fehler in der aktuellen und instabilen Version der Quelltexte zu beheben und dies dem Kunden zu schicken. Sicherlich könnte der Fehler behoben sein, doch der Rest des Quelltextes ist in einem instabilen und noch nicht getesteten Zustand. Sie könnte laufen, aber sie ist sicherlich noch nicht produktionsreif.

Weil die letzte veröffentlichte Version, von eben dem einen Fehler abgesehen, als stabil angesehen wird, ist die beste Lösung zurückzugehen und den Fehler in dieser älteren Version zu beheben - also ein weiteres Universum zu schaffen, in dem die letzte veröffentlichte Version die Fehlerbeseitigung beinhaltet.

Dies ist der Punkt, an dem Verzweigungen ins Spiel kommen. Der Entwickler spaltet einen Zweig ab, der in der Hauptentwicklungslinie (engl. **trunk**) verwurzelt ist, aber nicht mit den aktuellen Revisionen, sondern zurück zu dem Zeitpunkt der letzten Veröffentlichung. Dann kann er einen `Checkout` einer Arbeitskopie dieses Zweiges machen, die zur Fehlerbeseitigung notwendigen Veränderungen anbringen und diese durch einen `Commit` wieder CVS mitteilen, sodass davon Aufzeichnungen existieren. Nun kann er eine Zwischenversion zusammenstellen, die auf diesem Zweig basiert, und diese an den Kunden ausliefern.

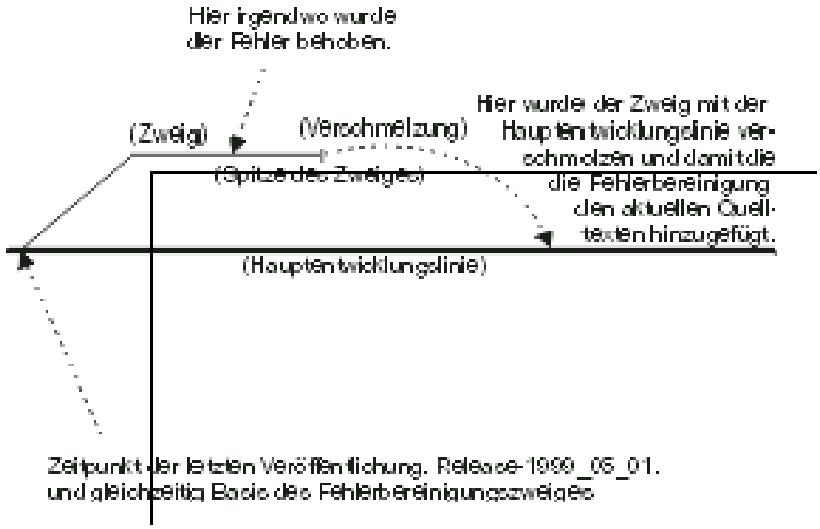
Seine Veränderungen beeinflussen die Quelltexte der Hauptentwicklungslinie nicht, was er auch sicherlich nicht wollte, ohne sich vorher zu vergewissern, dass diese die gleiche Art von Fehlerbereinigung benötigen. Sollte dies aber doch der Fall sein, kann er die Veränderungen des Zweiges wieder in die Hauptentwicklungslinie integrieren (**merge**). Bei einem `Merge` bestimmt CVS die Veränderungen, die seit dem Zeitpunkt der Aufspaltung von der Hauptentwicklungslinie bis zu der aktuellen Spitze (der aktuellste Stand des Zweiges) stattgefunden haben, und bringt diese Veränderung an dem Projekt zum Stand der Spitze des Zweiges an. Der Unterschied zwischen der Wurzel des Zweiges und der Spitze stellt sich, natürlich, als eben die Bereinigung des Fehlers heraus.

Ein `Merge` kann auch als ein Spezialfall des `update` angesehen werden. Der Unterschied beim `Merge` ist, dass die wieder zu integrierenden Veränderungen von Wurzel und Spitze des Zweiges abgeleitet werden und nicht durch den Vergleich einer Arbeitskopie mit dem Archiv.

Der Akt des `Update` an sich ist ähnlich wie die Patches von den jeweiligen Autoren direkt zu bekommen und diese per Hand einzufügen. Tatsächlich bestimmt CVS, um `update` durchzuführen, den Unterschied (also wie mit dem `diff`-Programm selbst) zwischen der Arbeitskopie und dem Archiv und wendet diesen `Diff` auf die Arbeitskopie genau so an, wie es auch das `patch`-Kommando machen würde. Dieses spiegelt die Art und Weise wieder, in der ein Entwickler Veränderungen von außerhalb annimmt, nämlich manuell die von den anderen Autoren erhaltenen `patch`-Dateien einzufügen.

Daher ist das Zusammenführen des Zweiges mit der Fehlerbereinigung mit der Hauptentwicklungslinie wie das Einfügen eines Fehlerbereinigungs-Patches von Dritten außerhalb des Projektes. Ein solcher Dritter würde den Patch gegen die letzte veröffentlichte Version machen, genau wie die Veränderungen des Zweiges gegen diese Version gemacht werden. Wenn sich dieser Bereich der Quelltexte seit der letzten Veröffentlichung nicht stark verändert hat, wird das Zusammenführen ohne Probleme ablaufen. Wenn sich der Quelltext jedoch in einem substanziell anderen Zustand befindet, wird die Zusammenführung mit Konflikten fehlschlagen (der Patch wird also zurückgewiesen), und man wird per Hand daran herumfummeln müssen. Üblicherweise wird dann die betreffende Stelle gelesen, die notwendigen Veränderungen werden per Hand eingefügt und ein `Commit` ausgeführt. Abbildung 2.3 zeigt das Vorgehen bei einer Verzweigung und Zusammenführung.

Wir werden nun die notwendigen Schritte, um das in der Abbildung Gezeigte zu erreichen, durchgehen. Denken Sie daran, dass von links nach rechts betrachtet nicht die Zeit voranschreitet, sondern dies vielmehr die Revisionshistorie widerspiegelt. Die Verzweigung fand nicht zum Zeitpunkt der Veröffentlichung statt, sondern wurde nur etwas später dort angesetzt.



Kap_02-3.png

Verzweigung und Zusammenführung

Nehmen wir für unseren Fall an, dass die Dateien vielen Veränderungen unterworfen waren, bis sie als **Release-1999-05-01** markiert, wurden und sogar einige hinzugekommen sind. Als die Fehlerbeschreibung über die alte veröffentlichte Version hereinkommt, wird das Erste, was wir machen wollen, folgendes sein: einen Zweig zu erzeugen, der auf diesem Veröffentlichungsstand basiert und den wir praktischerweise mit **Release-1999-05-01** markieren.

Ein Weg, dies zu erreichen ist, eine Arbeitskopie basierend auf dieser Markierung per `checkout` zu holen und

diese mit der `-b`-Option erneut zu markieren:

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

myproj/

user@linux ~/ # cvs -q checkout -d myproj_old_release -r
Release-1999_05_01 myproj

U myproj_old_release/README.txt
U myproj_old_release/hello.c
U myproj_old_release/a-subdir/whatever.c
U myproj_old_release/a-subdir/subsubdir/fish.c
U myproj_old_release/b-subdir/random.c

user@linux ~/ # ls

myproj_old_release/

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

CVS/ README.txt a-subdir/ b-subdir/ hello.c

user@linux ~/ # cvs -q tag -b Release-1999_05_01-bugfixes

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

user@linux ~/ #
```

Sehen Sie sich das letzte Kommando gut an. Es mag etwas willkürlich erscheinen, `tag` zur Erzeugung eines Zweiges zu verwenden, doch es gibt einen Grund dafür: Der Markierungsname wird als eine Bezeichnung verwendet, anhand derer der Zweig später aus dem Archiv geholt werden kann. Zweigmarkierungen unterscheiden sich nicht von Nicht-Zweigmarkierungen und unterliegen damit den gleichen Restriktionen für die Namensvergabe. Manche Benutzer fügen das Wort **branch** (engl. Zweig) in den Markierungsnamen ein (zum Beispiel **Release-1999_05_01-bugfix-branch**), sodass die Zweigmarkierungen leichter von anderen Markierungen unterschieden werden können. Vielleicht möchten Sie dies auch, wenn Sie des öfteren die falsche Markierung holen.

(Und wo wir gerade dabei sind, beachten Sie die `-d myproj_old_release`-Option des `checkout`-Befehls beim ersten CVS-Aufruf. Diese sagt `checkout`, die Arbeitskopie in das Verzeichnis namens `myproj_old_release` abzulegen, damit wir diese nicht mit der aktuellen Version in myproj durcheinander bringen. Achten Sie darauf, diese Verwendung von `-d` nicht mit der globalen Option gleichen Namens oder der `-d`-Option von `update` zu verwechseln.)

Natürlich macht die Ausführung dieses `tag`-Kommandos die aktuelle Arbeitskopie nicht automatisch zu einem Zweig. Das Markieren betrifft nie die Arbeitskopie; es zeichnet lediglich einige zusätzliche Informationen im Archiv auf, damit diese Revision der Arbeitskopie später wieder geholt werden kann (als ein fester Punkt in der Geschichte oder als ein Zweig, wie es der Fall sein kann).

Das Holen kann auf zwei Wegen erfolgen (Sie werden sich vielleicht schon daran gewöhnt haben!). Es kann aus

diesem Zweig eine neue Arbeitskopie per `checkout` geholt werden:

```
user@linux ~/ # pwd

/home/whatever

user@linux ~/ # cvs co -d myproj_branch -r Release-1999_05_01-bugfixes
myproj
```

oder eine bereits existierende Arbeitskopie kann dazu gemacht werden:

```
user@linux ~/ # pwd

/home/whatever/myproj

user@linux ~/ # cvs update -r Release-1999_05_01-bugfixes
```

Das Resultat ist das gleiche (nun ja, der Name des übergeordneten Verzeichnisses der neuen Arbeitskopie kann unterschiedlich sein, doch dies ist für die Zwecke von CVS unwichtig). Sollte Ihre Arbeitskopie noch nicht durch `commit` abgeschickte Veränderungen beinhalten, sollten Sie für den Zugriff auf den Zweig besser `checkout` anstatt `update` verwenden. Sonst würde CVS versuchen, Ihre Änderungen in die Arbeitskopie einfließen zu lassen, während es diese zu einem Zweig macht. In diesem Fall könnten Konflikte auftreten, auch wenn nicht, wäre das Ergebnis ein unreiner Zweig. Dieser würde nicht dem tatsächlichen Zustand des Programmes mit der angegebenen Markierung entsprechen, da einige Dateien der Arbeitskopie Ihre Veränderungen beinhalten würden.

Wie auch immer, nehmen wir an, dass Sie auf die eine oder andere Weise eine Arbeitskopie des gewünschten Zweiges bekommen haben:

```
user@linux ~/ # 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-bugfixes
(branch: 1.5.2)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ # 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-bugfixes (branch: 1.2.2)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

(Der Inhalt der Sticky Tag-Zeilen wird gleich erläutert.) Wenn Sie nun `random.c` und `hello.c` modifizieren

und `commit` ausführen

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

M hello.c
M b-subdir/random.c

user@linux ~/ # cvs ci -m "fixed old punctuation 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 <ó 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 <ó random.c
new revision: 1.2.2.1; previous revision: 1.2
done

user@linux ~/ #
```

werden Sie feststellen, dass etwas Lustiges mit den Revisionsnummern vor sich geht:

```
user@linux ~/ # cvs -q status hello.c b-subdir/random.c

===================================================
File: hello.c Status: Up-to-date
Working revision: 1.5.2.1 Wed May 5 00:13:58 1999
Repository revision: 1.5.2.1 /usr/local/cvs/myproj/hello.c,v
Sticky Tag: Release-1999_05_01-bugfixes (branch: 1.5.2)
Sticky Date: (none)
Sticky Options: (none)
===================================================
File: random.c Status: Up-to-date
Working revision: 1.2.2.1 Wed May 5 00:14:25 1999
Repository revision: 1.2.2.1 /usr/local/cvs/myproj/b-subdir/random.c,v
Sticky Tag: Release-1999_05_01-bugfixes (branch: 1.2.2)
Sticky Date: (none)
Sticky Options: (none)

user@linux ~/ #
```

Diese haben nun vier Ziffern anstatt zwei!

Ein näherer Blick zeigt, dass die Revisionsnummer jeder Datei lediglich aus der Zweignummer (wie in der Sticky Tag-Zeile angegeben) und einer extra Ziffer am Ende besteht.

Was Sie hier sehen, ist ein Stück von CVS innerer Arbeitsweise. Obwohl Sie sicherlich immer eine Verzweigung benutzen werden, um eine projektweite Aufspaltung zu markieren, zeichnet CVS dies jedoch auf Basis der einzelnen Dateien auf. Dieses Projekt beinhaltete zum Zeitpunkt der Verzweigung fünf Dateien, und es wurden daher fünf individuelle Zweige erzeugt, alle mit dem gleichen Markierungsnamen: **Release-1999_05_01-bugfixes**.

Bemerkung
Die meisten Benutzer sehen dies als eine eher unelegante Implementierung seitens CVS an. Hier scheint ein Teil des alten RCS-Vermächtnisses durch - RCS konnte Dateien nicht in Projekte gruppieren, und obwohl CVS dies nun kann, benutzt CVS dennoch Programmteile zur Verwaltung der Verzweigungen, die von RCS geerbt wurden.

Für gewöhnlich brauchen Sie sich nicht darum zu kümmern, wie CVS intern arbeitet, doch in diesem Fall ist es hilfreich zu wissen, in welcher Beziehung Zweignummern und Revisionsnummern stehen. Betrachten wir `hello.c`; alles, was ich über `hello.c` aussagen werde, trifft ebenso auf die anderen Dateien des Zweiges zu (Revisions-/Zweignummern entsprechend angepasst).

Zu dem Zeitpunkt, auf dem der Zweig basiert, hatte `hello.c` Revision 1.5. Als der Zweig geschaffen wurde, wurde an das Ende eine neue Ziffer angehängt, um die Zweignummer zu bilden (CVS verwendet dazu die noch nicht benutzte erste, gerade, ganze Zahl, die nicht null ist). Daher wurde die Zweignummer in diesem Fall **1.5.2**. Die Zweignummer an sich ist keine Revisionsnummer, sie ist aber die Wurzel (also das Präfix) aller weiteren Revisionsnummern von `hello.c` innerhalb dieses Zweiges.

Als jedoch das erste Mal der CVS-Status der verzweigten Arbeitskopie abgefragt wurde, wurde als Revisionsnummer nur **1.5** anstatt **1.5.2.0** oder etwas Ähnlichem angezeigt. Dies liegt daran, dass die erste Revisionsnummer innerhalb eines Zweiges immer gleich mit der Revision der Datei in der Hauptentwicklungslinie ist, von welcher der Zweig stammt. Daher zeigt CVS in Statusberichten die Revisionsnummer der Datei in der Hauptentwicklungslinie an, solange die Dateien des Zweiges und der Hauptentwicklungslinie identisch sind.

Als eine neue Revision per `commit` an das Archiv übertragen wurde, war `hello.c` in dem Zweig nicht mehr identisch mit der Hauptentwicklungslinie - der Inhalt der Datei im Zweig wurde verändert, wohingegen der der Hauptentwicklungslinie identisch blieb. Dementsprechend wurde `hello.c` seine erste Zweigrevisionsnummer zugeordnet. Dies ist in der Statusausgabe nach dem `commit`-Kommando zu sehen, die nun die Revisionsnummer **1.5.2.1** zeigt.

Das Gleiche gilt auch für die Datei `random.c`. Deren Revisionsnummer war zum Zeitpunkt der Verzweigung **1.2**, damit ist dessen erste Verzweigung **1.2.2**. Der erste `Commit` von `random.c` innerhalb dieses Zweiges bekam die Revisionsnummer **1.2.2.1**.

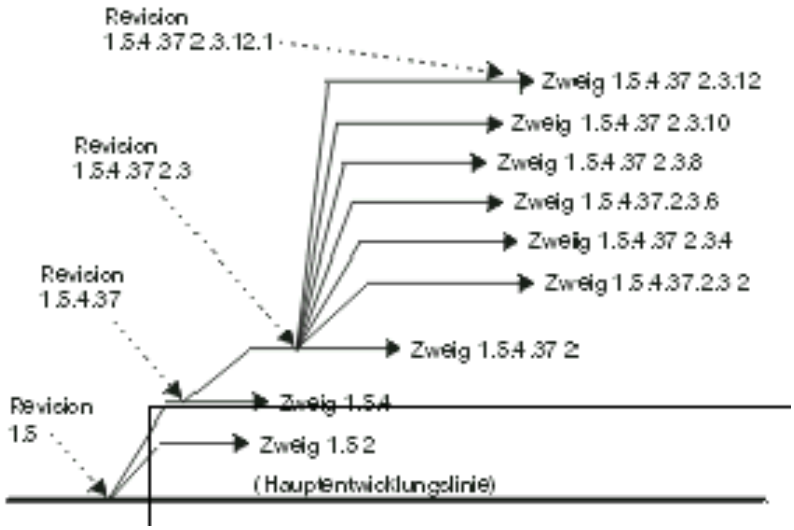
Es gibt keinen numerischen Zusammenhang zwischen **1.5.2.1** und **1.2.2.1** - betrachtet man nur diese Revisionsnummern, gibt es keinen Grund anzunehmen, dass diese Teil der gleichen Verzweigung sind, bis auf dass beide Dateien mit **Release-1999_05_01-bugfixes** markiert sind und diese Markierung zu Verzweigungsnummer 1.5.2 und 1.2.2 der jeweiligen Dateien gehört. Daher ist der Markierungsname die einzige Möglichkeit, den Zweig projektweit zu identifizieren. Obwohl es absolut möglich ist, eine Datei in einen Zweig anhand ihrer Revisionsnummer zu verschieben

```
user@linux ~/ # cvs update -r 1.5.2.1 hello.c

U hello.c

user@linux ~/ #
```

ist davon fast immer abzuraten. Man würde dadurch die verzweigten Revisionen einer Datei mit den nicht verzweigten Revisionen anderer vermischen. Wer weiß, welche Verluste daraus entstehen? Es ist besser, die Zweigmarkierung zur Referenz auf den Zweig zu verwenden und alle Dateien auf einmal zu bearbeiten, als eine bestimmte Datei zu spezifizieren. Auf diese Weise braucht man die tatsächliche Zweigrevisionsnummer einer Datei gar nicht zu wissen oder sich darum zu kümmern.

Es ist auch möglich, dass Zweige sich wieder verzweigen bis zu jeder beliebig absurden Stufe. Eine Datei mit der Revisionsnummer 1.5.4.37.2.3.12.1 wird in Abbildung 2.4 grafisch dargestellt.

Zugegeben sind Umstände, in denen eine solche Verzweigungstiefe notwendig ist, nur schwer vorstellbar, doch ist es nicht schön zu wissen, dass CVS so weit gehen kann, wie man dies selbst möchte? Eingebettete Verzweigungen werden genau wie jeder andere Zweig angelegt: Eine Arbeitskopie eines Zweiges N per `Checkout` holen, `cvs tag -b` Zweigname darin ausführen und damit Zweig N.M im Archiv erstellen (wobei **N** die zugehörige Zweigrevisionsnummer jeder Datei ist, wie beispielsweise **1.5.2.1**, und **M** den nächstmöglichen Zweig am Ende dieser Zahl angibt, wie beispielsweise **2**).



Kap_02-4.png

Ein absurd hochgradiger Verzweigungsgrad

Veränderungen zwischen Zweig und Stamm verschmelzen

Nun, da die Fehlerbereinigung durch `commit` in den Zweig kam, lassen Sie uns aus der Arbeitskopie die höchste Revision der Hauptentwicklungslinie machen und nachsehen, ob diese Fehlerbereinigung dort auch angebracht werden muss. Dazu entfernen wir unsere Arbeitskopie durch `update -A` wieder von dem Zweig (Zweigmarkierungen verhalten sich diesbezüglich wie andere bindende Eigenschaften) und führen dann einen `Diff` gegen die Hauptentwicklungslinie, die wir gerade verlassen haben, durch:

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

U hello.c
U b-subdir/random.c

user@linux ~/ # cvs -q diff -c -r Release-1999_05_01-bugfixes

Index: hello.c
===================================================
RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.5.2.1
retrieving revision 1.6
diff -c -r1.5.2.1 -r1.6
```

```
*** hello.c 1999/05/05 00:15:07 1.5.2.1
--- hello.c 1999/05/04 20:19:16 1.6
***************
*** 4,9 ****
main ()
{
printf ("Hello, world!\n");
! printf ("between hello and good-bye\n");
printf ("Goodbye, world!\n");
}
--- 4,10 ----
main ()
{
printf ("Hello, world!\n");
! printf ("between hello and goodbye\n");
printf ("Goodbye, world!\n");
+ /* a comment on the last line */
}
Index: b-subdir/random.c
===================================================
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.2.2.1
retrieving revision 1.2
diff -c -r1.2.2.1 -r1.2
*** b-subdir/random.c 1999/05/05 00:15:07 1.2.2.1
--- b-subdir/random.c 1999/04/19 06:35:27 1.2
***************
*** 4,8 ****
void main ()
{
! printf ("A random number.\n");
}
--- 4,8 ----
void main ()
{
! printf ("a random number\n");
}

user@linux ~/ #
```

Der `Diff` zeigt, dass good-bye in der verzweigten Revision von `hello.c` mit einem Bindestrich geschrieben wird und dass die Revision der Hauptentwicklungslinie dieser Datei einen Kommentar am Ende enthält, den die andere Revision nicht enthält. Außerdem enthält die Zweigrevision von `random.c` ein großes **A** und einen Punkt, wohingegen die Hauptentwicklungslinie dies nicht beinhaltet.

Um die Veränderungen des Zweiges tatsächlich in die aktuelle Arbeitskopie zu übernehmen, wird das `update`-Kommando mit der Option `-j` verwendet (das gleiche `j` für **join**, das wir auch verwendet haben um eine Datei in eine ältere Revision zurückzuführen):

```
user@linux ~/ # cvs -q update -j Release-1999_05_01-bugfixes

RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.5
retrieving revision 1.5.2.1
Merging differences between 1.5 and 1.5.2.1 into hello.c
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
```

```
retrieving revision 1.2
retrieving revision 1.2.2.1
Merging differences between 1.2 and 1.2.2.1 into random.c

user@linux ~/ # cvs -q update

M hello.c
M b-subdir/random.c

user@linux ~/ # cvs -q ci -m "merged from branch
Release-1999_05_01-bugfixes"

Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v <ó hello.c
new revision: 1.7; previous revision: 1.6
done
Checking in b-subdir/random.c;
/usr/local/cvs/myproj/b-subdir/random.c,v <ó random.c
new revision: 1.3; previous revision: 1.2
done

user@linux ~/ #
```

Dies bringt die Veränderungen von der Wurzel des Zweiges an dessen Spitze und verschmilzt diese mit der aktuellen Arbeitskopie (die daraufhin diese Veränderungen genau so aufweist, als wäre sie von Hand in diesen Zustand gebracht worden). Diese Veränderungen werden dann per `commit` in die Hauptentwicklungslinie eingebunden, da sich nichts im Archiv verändert, wenn merge auf eine Arbeitskopie angewendet wird.

Obwohl in diesem Beispiel keine Konflikte auftraten, ist es möglich (sogar wahrscheinlich), dass welche bei einem normalen `Merge` auftreten. Ist dies der Fall, müssen diese, wie bei jedem anderen Konflikt auch, zuerst aufgelöst und dann wieder per `commit` eingebracht werden.

Mehrfache Verschmelzung

Manchmal wird ein Zweig aktiv weiterentwickelt, obwohl die Hauptentwicklungslinie bereits damit verschmolzen wurde. Dies kann zum Beispiel dann vorkommen, wenn ein zweiter Fehler in der letzten veröffentlichten Version gefunden wird und in dem Zweig behoben werden muss. Vielleicht hat jemand den Witz in `random.c` nicht verstanden, also muss in dem Zweig eine Zeile zu dessen Erklärung eingefügt

```
user@linux ~/ # pwd

/home/whatever/myproj_branch

user@linux ~/ # cat b-subdir/random.c

/* Print out a random number. */
#include <stdio.h>
void main ()
{
printf ("A random number.\n");
printf ("Get the joke?\n");
}

user@linux ~/ #
```

und per `commit` abgeschickt werden. Wenn diese Fehlerbereinigung nun auch in der Hauptentwicklungslinie durchgeführt werden muss, könnte man versucht sein, das gleiche `update`-Kommando wie zuvor in der Arbeitskopie der Hauptentwicklungslinie durchzuführen, um eine **Neuverschmelzung** zu machen:

```
user@linux ~/ # cvs -q update -j Release-1999_05_01-bugfixes

RCS file: /usr/local/cvs/myproj/hello.c,v
retrieving revision 1.5
retrieving revision 1.5.2.1
Merging differences between 1.5 and 1.5.2.1 into hello.c
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.2
retrieving revision 1.2.2.2
Merging differences between 1.2 and 1.2.2.2 into random.c
rcsmerge: warning: conflicts during merge

user@linux ~/ #
```

Wie Sie sehen können, hatte dies nicht den erwarteten Effekt - es wird ein Konflikt angezeigt, obwohl die Kopie der Hauptentwicklungslinie nicht verändert wurde und deshalb keine Konflikte zu erwarten waren.

Das Problem ist, dass das `update`-Kommando genau wie beschrieben arbeitete: Es versuchte alle Veränderungen zwischen der Zweigwurzel und -spitze mit der aktuellen Arbeitskopie zu verschmelzen. Das Problem hier ist, dass einige dieser Veränderungen bereits mit dieser Arbeitskopie verschmolzen wurden. Daher der Konflikt:

```
user@linux ~/ # pwd

/home/whatever/myproj

user@linux ~/ # cat b-subdir/random.c

/* Print out a random number. */
#include <stdio.h>
void main ()
{
<<<<<<<< random.c
printf ("A random number.\n");
=======
printf ("A random number.\n");
printf ("Get the joke?\n");
>>>>>>> 1.2.2.2
}

user@linux ~/ #
```

Man könnte nun alle diese Konflikte durchgehen und von Hand auflösen - es ist gewöhnlich nicht schwer herauszufinden, was in jeder Datei verändert werden muss. Dennoch ist es besser, Konflikte von vornherein zu verhindern. Durch die Angabe von zwei `-j`-Optionen anstatt von einer kann man nur jene Veränderungen bekommen, die nach dem Zeitpunkt der letzten Verschmelzung mit der Spitze stattgefunden haben, anstatt alle Veränderungen des Zweiges von der Wurzel bis zur Spitze. Das erste `-j` gibt dabei den Startpunkt auf dem Zweig und das zweite einfach den Zweignamen an (welcher die Spitze des Zweiges einschließt).

Die Frage ist nur, wie kann der Punkt der letzten Verschmelzung auf dem Zweig spezifiziert werden? Eine

Möglichkeit ist, ein Datum mit dem Namen der Zweigmarkierung anzugeben. CVS bietet dafür eine eigene Syntax:

```
user@linux ~/ # cvs -q update -j "Release-1999_05_01-bugfixes:2 days ago"
\

-j Release-1999_05_01-bugfixes
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.2.2.1
retrieving revision 1.2.2.2
Merging differences between 1.2.2.1 and 1.2.2.2 into random.c

user@linux ~/ #
```

Folgt dem Namen der Zweigmarkierung ein Doppelpunkt und ein Datum (in einem der üblichen CVS-Datumsformate), werden von CVS nur Veränderungen berücksichtigt, die nach diesem Datum stattfanden. Wenn Sie also wissen, dass die ursprüngliche Bereinigung des Fehlers vor drei Tagen per `commit` in den Zweig einfloss, würde das vorstehende Kommando nur die neue Fehlerbereinigung einfließen lassen.

Ein besserer Weg ist, wenn man im Voraus plant, den Zweig nach jeder Fehlerbereinigung zu markieren (nur eine normale Markierung, es soll kein neuer Zweig oder etwas Ähnliches damit begonnen werden). Stellen Sie sich vor, nach der Beseitigung des Fehlers in dem Zweig und anschließendem `commit` führen Sie Folgendes in der Arbeitskopie des Zweiges aus:

```
user@linux ~/ # cvs -q tag Release-1999_05_01-bugfixes-fix-number-1

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

user@linux ~/ #
```

Dann, wenn es Zeit wird, die zweite Veränderung mit der Hauptentwicklungslinie zu verschmelzen, können Sie diese sinnvoll platzierte Markierung verwenden, um die vorangegangenen Revisionen einzuschränken:

```
user@linux ~/ # cvs -q update -j Release-1999_05_01-bugfixes-fix-number-1
\

-j Release-1999_05_01-bugfixes
RCS file: /usr/local/cvs/myproj/b-subdir/random.c,v
retrieving revision 1.2.2.1
retrieving revision 1.2.2.2
Merging differences between 1.2.2.1 and 1.2.2.2 into random.c

user@linux ~/ #
```

Dies ist natürlich wesentlich besser als zu versuchen, sich daran zu erinnern, wann man die eine oder andere Veränderung gemacht hat, funktioniert aber nur, wenn man daran denkt, den Zweig jedes Mal zu markieren, wenn man ihn mit der Hauptentwicklungslinie verschmolzen hat. Die Lehre daraus ist also früh zu markieren, und das oft! Es ist besser, sich durch zu viele Markierungen kämpfen zu müssen (solange diese aussagekräftige Namen haben), als zu wenige zu haben. In den vorangegangenen Beispielen war es zum Beispiel nicht

notwendig, dass die neuen Zweigmarkierungen einen ähnlichen Namen wie der Zweig selbst haben. Obwohl ich diesen **Release-1999_05_01-bugfixes-fix-number-1** genannt habe, könnte er genauso gut **fix1** heißen. Dennoch ist die erste Variante vorzuziehen, weil diese den Namen des Zweiges beinhaltet und so kaum mit einer Markierung eines anderen Zweiges verwechselt werden kann. (Denken Sie daran, dass Markierungsnamen lediglich innerhalb einer Datei einmalig sind, nicht innerhalb von Zweigen. Es kann keine zwei Markierungen **fix1** innerhalb einer Datei geben, auch wenn sich diese auf Revisionen in unterschiedlichen Zweigen beziehen.)

Markierungen und Zweige ohne Arbeitskopie erstellen

Wie vorher schon erwähnt, beeinflusst das Markieren das Archiv, nicht die Arbeitskopie. Dies wirft die Frage auf: Warum wird überhaupt eine Arbeitskopie zum Markieren benötigt? Der einzige Zweck ist, das Projekt und die Revisionen der verschiedenen Dateien innerhalb des Projektes anzugeben, auf die sich die Markierung beziehen soll. Würde man das Projekt und die Revisionen unabhängig von der Arbeitskopie angeben, würde gar keine Arbeitskopie benötigt.

Dies ist auch möglich: mit dem `rtag`-Kommando (**repository tag**, Markieren im Archiv). Dies ist sehr ähnlich zu `tag`; ein paar Beispiele sollen seine Verwendung erläutern. Gehen wir zurück zu dem Augenblick, als der erste Fehler berichtet wurde und wir einen Zweig erzeugen mussten, der zum Zeitpunkt der letzten veröffentlichten Version seine Wurzel hatte. Es wurde dann ein `Checkout` mit der Markierung der Veröffentlichung gemacht und anschließend `tag -b` darauf angewendet:

```
user@linux ~/ # cvs tag -b Release-1999_05_01-bugfixes
```

Dies erstellte einen Zweig, der bei **Release-1999_05_01** verwurzelt ist. Wir kannten allerdings die Markierung der Veröffentlichung und hätten diese mit dem `rtag`-Kommando verwenden können, um die Wurzel der Verzweigung anzugeben, ohne uns um eine Arbeitskopie kümmern zu müssen:

```
user@linux ~/ # cvs rtag -b -r Release-1999_05_01
Release-1999_05_01-bugfixes myproj
```

Das ist alles. Dieses Kommando kann von überall innerhalb oder außerhalb einer Arbeitskopie ausgeführt werden. Die `CVSROOT`-Umgebungsvariable müsste natürlich schon auf das Archiv verweisen, oder man könnte dies mit der `-d`-Option direkt angeben. Dies funktioniert ebenso bei Nicht-Verzweigungsmarkierungen, ist aber weniger nützlich, da dann die Revisionsnummern jeder Datei nacheinander angegeben werden müssten.(Man könnte sich natürlich auch mittels einer Markierung darauf beziehen, doch dann wäre ja schon eine Markierung dieser Revision vorhanden, und warum sollte man dann noch eine hinzufügen?)

Sie wissen nun genug, um mit CVS zurechtzukommen, und vielleicht auch schon genug, um mit anderen Leuten an einem Projekt zu arbeiten. Es gibt immer noch einige weniger wichtige Funktionen, die noch nicht eingeführt wurden, genau wie einige noch nicht erwähnte nützliche Optionen zu Funktionen, die bereits erwähnt wurden. Diese werden alle in entsprechenden, noch kommenden Kapiteln vorgestellt, zusammen mit Szenarien, in denen sowohl gezeigt wird, wie und wann man diese einsetzt. Im Zweifelsfall konsultieren Sie das *Cederqvist*-Handbuch; dies ist eine unverzichtbare Quelle für jeden ernsthaften CVS-Benutzer.

1. Anm. d. Übers.: engl. **directory** = Verzeichnis
2. Anm. d. Übers.: engl. **hunk** = Stück
3. Anm. d. Übers.: engl. **sticky** = bindend, anhaftend, klebrig
4. Anm. d. Übers.: engl. **branch** = Zweig, Verzweigung