

Chapter 2

An Overview of CVS

CVS Basics

In this chapter, we explore the basics of CVS and then go into an in-depth guided tour of everyday CVS usage by looking at some examples. After completing this chapter, you will be able to use CVS's most fundamental functions.

If you've never used CVS (or any version control system) before, it's easy to get tripped up by some of its underlying assumptions. What seems to cause the most initial confusion about CVS is that it is used for two apparently unrelated purposes: record keeping and collaboration. It turns out, however, that these two functions are closely related.

Source revision control systems like CVS became necessary because people wanted to compare a program's current state with how it was at some point in the past. In the normal course of implementing a new feature, for instance, a developer might bring the program into a thoroughly broken state, in which it can sometimes remain until the feature is mostly finished. Unfortunately, this is just the time when someone usually calls to report a bug in the last publicly released version. To debug the problem (which might also exist in the current version of the sources), the program has to be brought back to a usable state.

Restoring the state poses no difficulty if the source code history is kept under CVS. The developer can simply say, in effect, "Give me the program as it was three weeks ago," or perhaps "Give me the program as it was at the time of our last public release." If you've never had this kind of convenient access to historical snapshots

before, you might be surprised at how quickly you come to depend on it. The authors of this book always use revision control in their coding projects—and it has saved them many times.

To understand what this has to do with facilitating collaboration, we need to take a closer look at the mechanism that CVS provides to help numerous people work on the same project.

What CVS Is Not: The Lock-Modify-Unlock Model

Let's take a look at a mechanism that CVS *doesn't* provide (or at least, doesn't encourage): file locking. If you've used other version control systems, you might be familiar with the **lock-modify-unlock** development model, wherein a developer first obtains exclusive write access (a *lock*) to the file to be edited, makes the changes, and then releases the lock to allow other developers access to the file. If someone else already has a lock on the file, they have to "release" it before you can lock it and start making changes (in some implementations, you can "steal" their lock, but that is often an unpleasant surprise for them and not good practice).

This model is workable if the developers know each other, know who's planning to do what at any given time, and can communicate with each other quickly if someone cannot work because of access contention. However, if the developer group becomes too large or too spread out, dealing with all the locking issues begins to chip away at coding time. It quickly becomes a constant hassle that can discourage people from getting real work done.

What CVS Is: The Copy-Modify-Merge Model

CVS takes a more mellow approach. Rather than requiring that developers coordinate with each other to avoid conflicts, CVS enables developers to edit simultaneously, assumes the burden of integrating all the changes, and keeps track of any conflicts. This process uses the **copy-modify-merge** model, which works as follows:

1. The developer downloads a *working copy* (a directory tree containing the files that make up the project) from CVS. This is also known as "checking out" a working copy, like checking a book out of the library.

2. The developer edits freely in his working copy. At the same time, other developers might be busy in their own working copies. Because these are all separate copies, there is no interference. It is as if all of the developers have their own copy of the same library book, and they're all at work scribbling comments in the margins or rewriting certain pages independently.

3. The developer finishes his changes and *commits* them into CVS along with a "log message," which is a comment explaining the nature and purpose of the changes. This is like informing the library of what changes he made to the book and why. The library then incorporates these changes into a "master" copy, where they are recorded for all time.

4. Meanwhile, other developers can ask CVS to query the library to see if the master copy has changed recently. If it has, CVS automatically updates their working copies. (This part is magical and wonderful, and we hope you appreciate it. Imagine how different the world would be if real books worked this way!)

As far as CVS is concerned, all developers on a project are equal. Deciding when to update or when to commit is largely a matter of personal preference or project policy. One common strategy for coding projects is to always update before commencing work on a major change and to commit only when the changes are complete and tested so that the master copy is always in a "runnable" state.

Perhaps you're wondering what happens when developers A and B, each in their own working copy, make different changes to the same area of text and then both commit their changes? This is called a *conflict*, and CVS notices it as soon as developer B tries to commit changes. Instead of allowing developer B to proceed after developer A has committed changes to the same files, CVS announces that it has discovered a conflict and places *conflict markers* (easily recognizable textual flags) at the conflicting location in his copy. That location also shows both sets of changes, arranged for easy comparison. Developer B must sort it all out and commit a new revision with the conflict resolved. Perhaps the two developers will need to talk to each other to settle the issue. CVS only alerts the developers that there is a conflict; it's up to human beings to actually resolve it.

What about the master copy? In official CVS terminology, it is called the project's *repository*. The repository is simply a file tree kept on a central server. Without going into too much detail about its structure (see Chapter 3 for more information), let's look at what the repository must do to meet the requirements of the checkout-commit-update cycle. Consider the following scenario:

1. Two developers, Joseph and Sarah, check out working copies of a project at the same time. The project is at its starting point—no changes have been committed by anyone yet, so all the files are in their original, pristine state.
2. Sarah gets right to work and soon commits her first batch of changes.
3. Meanwhile, Joseph is not doing any work at all.
4. Sarah, feeling quite productive, commits her second batch of changes. Now, the repository's history contains the original files, followed by Sarah's first batch of changes, followed by this set of changes.
5. Meanwhile, developer Joseph remains inactive.
6. Suddenly, developer Robert joins the project and checks out a working copy from the repository. Robert's copy reflects Sarah's first two sets of changes, because they are already in the repository when Robert checks out his copy.
7. Sarah, still feeling very productive, completes and commits her *third* batch of changes.

8. Finally, blissfully unaware of the recent frenzy of activity, Joseph decides it's time to start work. He doesn't bother to update his copy; he just commences editing files, some of which might be files that Sarah has worked in. Shortly thereafter, Joseph commits his first changes.

At this point, one of two things can happen. If none of the files Joseph edited have been edited by Sarah, the commit succeeds. However, if CVS detects that some of Joseph's files are out of date with respect to the repository's latest copies, *and* those files have also been changed by Joseph in his working copy, CVS informs Joseph that he must do an update before committing those files.

When Joseph runs the update, CVS merges all of Sarah's changes into Joseph's local copies of the files. Some of Sarah's work might conflict with Joseph's uncommitted changes, and some might not. Those parts that don't are simply applied to Sarah's copies without further complication; Joseph must resolve the conflicting ones before they can be committed.

If developer Robert does an update now, he'll receive several batches of changes from the repository: Sarah's third commit, then Joseph's first, and then possibly Joseph's second commit (if Joseph had to resolve any conflicts).

Commits Stored as Diffs

In order for CVS to serve up changes in the correct sequence to developers whose working copies might be out of sync by varying degrees, the repository needs to store all commits since the project's beginning. In practice, the CVS repository stores them all as successive diffs. Consequently, even for a very old working copy, CVS is able to calculate the difference between the working copy's files and the current state of the repository, and is thereby able to bring the working copy up to date efficiently. This makes it easy for developers to view the project's history at any point and to revive even very old working copies.

Although, strictly speaking, the repository could achieve the same results by other means, in practice, storing diffs is a simple, intuitive means of implementing the necessary functionality. The process has other added benefits. By using **patch** appropriately, CVS can reconstruct any previous state of the file tree and thus bring any working copy from one state to another. It can allow someone to check out the project as it looked at any particular time. It can also show the differences, in **diff** format, between two states of the tree without affecting anyone's working copy.

As a result, the very features necessary to give convenient access to a project's history are also useful for allowing a group of decentralized developers working separately to collaborate on the project.

Review of Terms

For now, you can ignore the details of setting up a repository, administering user access, and navigating CVS-specific file formats. For the moment, we'll concentrate on how to make changes in a working copy. But first, here is a quick review of terms:

- *Check out*—To request a working copy from the repository. Your working copy reflects the state of the project as of the moment you checked it out; when you and other developers make changes, you must use the **commit** and **update** commands to "publish" your changes and view others' changes.
- *Commit*—To send changes from your working copy into the central repository. Also known as *check in*.
- *Conflict*—The situation when two developers try to commit changes to the same region of the same file. CVS notices and points out conflicts, but the developers must resolve them.
- *Log message*—A comment you attach to a revision when you commit it, describing the changes. Others can page through the log messages to get a summary of what's been going on in a project.
- *Repository*—The master copy where CVS stores a project's full revision history. Each project has exactly one repository.
- *Revision*—A committed change in the history of a file or set of files. A revision is one "snapshot" in a constantly changing project.
- *Update*—To bring others' changes from the repository into your working copy and to show whether your working copy has any uncommitted changes. Be careful not to confuse this with the **commit** operation; they are complementary, not identical, operations. Here's a mnemonic to help you remember: **update** brings your working copy up to date with the repository copy.
- *Working copy*—The copy in which you actually make changes to a project. There can be many working copies of a given project; generally, each developer has his or her own copy.

Other Revision Control Systems

Obviously, CVS is not the only revision control system in use. CVS is not even the only revision control system used for open source projects. Quite a lot of open source software is actually written on Windows platforms.

BitKeeper

BitKeeper is a revision control system, similar, at least at some level, to RCS, CVS, ClearCase, Visual Source Safe, Sun Teamware, and other revision control systems. Some of the features which make BK/Pro stand out in the crowded SCM market include:

- Inherently reliable and scalable through replication
- Change sets provide reproducibility, accountability, and aid in debugging

- Powerful GUI tools shorten development time and remove human error
- Excellent merging tools save engineering time and increase productivity
- Repositories such as branches allow controlled development
- Geographically distributed development works as well as local development
- Disconnected operation with no loss of functionality
- Compressed repositories reduce disk space and increase performance
- Excellent file renaming support allows flexibility as projects grow
- Scalable to thousands of developers
- Multi-protocol (FS/RSH/SSH/HTTP/BKD/SMTP) connectivity: work how and where you want

BK/Pro is a scalable configuration management system, supporting globally distributed development, disconnected operation, compressed repositories, change sets, and repositories as branches.

Distributed means that every developer gets their own personal repository, complete with revision history. The tool also handles moving changes between repositories. SSH, RSH, BKD, HTTP, and/or SMTP can all be used as communication transports between repositories; or, if both are local, the system just uses the file system. For example, this command updates from a remote system to a local file system using ssh:

```
bk pull bitmover.com:/home/bk/bk-3.0.x
```

Change sets are a formalization of a patch file (i.e., one or more changes to one or more files.) Change sets also provide built-in configuration management—the creation of a change set saves the entire state of your repository, both what changed and what didn't, in less than a second.

Other features: file names are revisioned and propagated just like contents; graphical interfaces are provided for merging, browsing, and creating changes; changes are logged to a private or public change server for centralized tracking of work; bug tracking is in the works and will be integrated.

The BitKeeper License

This is actually the most difficult to understand part of BitKpeer. If you're willing to live with the restrictions of a pseudo-open-source license, you can use BitKeeper for free (as in no-cost) under the terms of the BitKeeper License (bk help bkl). The restrictions put on you are that you cannot distribute modified copies that don't pass the BK regression test. Among the things the regression test checks is its ability to send logging information back

to BitMover. This logging information is supposedly only the comments generated from checking in new versions, but I have not verified that to be true.

If you are in a situation where you don't want that information publicly available, then BitMover is in a situation where they'd love to sell you a license. I have no information on the cost of commercial prices of BitKeeper. They have information on their web page that leads me to believe it is on par with the cost of ClearCase (at leats as far as per seat purchase price), which would put it in the $1,000+ range. Of course, you should talk to them about this.

Microsoft VSS

Next to CVS, one of the most widely used source systems for the Windows platform is Microsoft's VSS, or Visual Source Safe. Microsoft VSS helps you manage your projects, regardless of the file type (text files, graphics files, binary files, sound files, or video files) by saving them to a database. VSS easily integrates with Microsoft Access, Visual Basic, Visual C++, Visual FoxPro, and other development tools. If VSS is integrated into your development environment, you do not need to run VSS separately to realize the advantages of source code control.

VSS makes a distinction between text files (files that contain only characters) and binary files (all others). For most operations, you can treat text and binary files exactly the same—VSS uses its highly disk-efficient reverse delta storage on all files, text and binary. The reverse delta is a system that stores incremental changes to a baseline file rather than storing each successive version of the file in its entirety. VSS uses the current version of a file as the baseline, and it saves changes from the previous versions. This results in reduced disk storage requirements and faster access times, because only the current version is stored in the database in its entirety.

When you add a file to VSS, it's automatically assigned a type: text or binary. The default mechanism for creating this assignment is a simple test: VSS scans the file for NULL characters (bytes with value 0). If it finds such a character, VSS identifies the file as binary.

Although generally accurate, this method might on occasion incorrectly assign the text type to a binary file. Therefore, VSS allows you to explicitly set the file type option to Auto-Detect, Binary, or Text. Auto-Detect is the default. A file retains the type it was originally given, unless you explicitly change it.

Here are the significant differences in how VSS treats binary and text files:

- *Storing changes*—Internally, VSS uses different mechanisms for storing changes for text files (which have distinct lines as units of comparison) and binary files (which have no obvious line delimiter). That's why it is important for VSS to correctly identify the type of a file. VSS identifies files as binary if a NULL character exists in the file.
- *The* **show differences** *command*—With a binary file, VSS stores each change as a small record of which bytes moved where; this is useful for reconstructing earlier versions, but not for display. VSS can tell you if the file has changed, but it cannot display how the file has changed.

- *Merging binary files*—VSS cannot perform this operation.
- *Multiple checkouts on binary files*—VSS cannot perform this operation.
- *End-of-line characters*—With a text file, VSS automatically translates end-of-line characters between different operating systems; with a binary file, VSS does not alter the contents of the file except with keyword expansion.

RCS and GNU/RCS

One other source control system is the honorable RCS, which stands for Revision Control System. It is a simple system for keeping track of changes to files. A master copy is kept in a *repository*, and users can check out working copies. Only locked copies can be edited or modified. RCS stores all changes (with comments, author, and timestamp) in the master copy. A revision can be checked out or compared to any other revision or to the current working copy. When a batch of changes has been done and a new revision is wanted, the working copy can be checked in. RCS is a good system for single users or very small teams, because concurrent changes to a file are not allowed. More advanced systems, such as CS-RCS (CS stands for ComponentSoftware), CVS, and Perforce, use the same file format but have extended capabilities for teamwork. Perforce also can work at high speed even for huge projects and provides import tools.

In "traditional" GNU/RCS, the master files are kept in the same directory as the working copy or in a subdirectory called "RCS". This arrangement makes it easy to use RCS with files stored on removable media. CS-RCS is compatible with this arrangement and can also be used in an enhanced mode with a central repository. This repository can be either local (for the free version) or on a server (for the version that is not free), and several remote workstations can access it. The use of a central repository adds to normal RCS capabilities the possibility of grouping files located in different directories—or even on different workstations—into a single "project." This allows operations to be carried out simultaneously on all files—for example, to freeze a release of a system. Perforce also uses a central repository, and in addition works on Windows and several flavors of Unix.

GNU/RCS is free and consists of a series of console applications. CS-RCS is based on GNU/RCS but has a very well integrated front end for the Win32 environment. It is free for single user setup, but it is not free for multi-user network setup with a shared repository. Perforce is commercial software that is aimed at teams of several developers or writers, but it is free for up to two users.

Most implementations of RCS can be used with both text and binary files.

SCCS

SCCS allows several versions of the same file to exist simultaneously, which can be helpful when developing a project requiring many versions of large files. The SCCS commands support Multibyte Character Set (MBCS) characters. The SCCS commands form a complete

system for creating, editing, converting, or changing the controls on SCCS files. An SCCS file is any text file controlled with SCCS commands. All SCCS files have the prefix "s.", which sets them apart from regular text files.

Rather than creating a separate file for each version of a file, the SCCS file system stores only the changes for each version of a file. These changes are referred to as *deltas*. The changes are tracked by the delta table in every SCCS file.

Each entry in the delta table contains information about who created the delta, when they created it, and why they created it. Each delta has a specific SID (SCCS Identification number) of up to four digits. The first digit is the release, the second digit the level, the third digit the branch, and the fourth digit the sequence. No SID digit can be 0, so there cannot be an SID of 2.0 or 2.1.2.0, for example.

Here's an example of an SID number that specifies release 1, level 2, branch 1, sequence 4:

```
SID=1.2.1.4
```

Each time a new delta is created, it is given the next higher SID number by default. That version of the file is built using all the previous deltas. Typically, an SCCS file grows sequentially, so each delta is identified only by its release and level. However, a file can branch and create a new subset of deltas. The file then has a trunk, with deltas identified by release and level, and one or more branches, which have deltas containing all four parts of an SID. On a branch, the release and level numbers are fixed, and new deltas are identified by changing sequence numbers.

After the delta table in an SCCS file, a list of flags starting with @ (the at sign) define the various access and tracking options of the SCCS file. The SCCS flag functions include:

- Designating users who may edit the files
- Locking certain releases of a file from editing
- Allowing joint editing of the file
- Cross-referencing changes to a file

The SCCS file body contains the text for all the different versions of the file. Consequently, the body of the file does not look like a standard text file. Control characters bracket each portion of the text and specify which delta created or deleted it. When the SCCS system builds a specific version of a file, the control characters indicate the portions of text that correspond to each delta. The selected pieces of text are then used to build that specific version.

A Tour of CVS

Now that we have gone over the concepts of source revisioning, we can enter into an introduction to fundamental CVS usage. This is followed by a sample session that covers all

of the most typical CVS operations. As the tour progresses, we'll also start to look at how CVS works internally.

Although you don't need to understand every last detail of CVS implementation to use it, a basic knowledge of how it works is invaluable in choosing the best way to achieve a given result. CVS is more like a bicycle than an automobile, in the sense that its mechanisms are entirely transparent to anyone who cares to look. As with a bicycle, you can just hop on and start riding immediately. However, if you take a few moments to study how the gears work, you'll be able to ride it much more efficiently. (In the case of CVS, we're not sure whether transparency was a deliberate design decision or an accident, but transparency does seem to be a property shared by many free programs. Externally visible implementations have the advantage of encouraging users to become contributing developers by exposing them to the system's inner workings right from the start.)

Our tour takes place in a Unix environment. CVS also runs on Windows and Macintosh operating systems, and Tim Endres of Ice Engineering has even written a Java client (see **www.ice.com/java/jcvs/**), which can be run anywhere Java runs. However, we're going to take a wild guess and assume that the majority of CVS users—present and potential—are most likely working in a Unix command-line environment. If you aren't one of these, the examples in the tour should be easy to translate to other interfaces. Once you understand the concepts, you can sit down at any CVS front end and work with it (trust us, we've done it many times).

The examples in the tour are oriented toward people who will be using CVS to keep track of programming projects. However, CVS operations are applicable to all text documents, not just source code.

The tour also assumes that you already have CVS installed (it's present by default on many of the popular free Unix systems, so you might already have it without knowing it) and that you have access to a repository. Even if you are not set up, you can still benefit from reading the tour. Later in this book, you'll learn how to install CVS and set up repositories.

Assuming CVS is installed, you should take a moment to find the online CVS manual. Known familiarly as the "Cederqvist" (after Per Cederqvist, its original author), it comes with the CVS source distribution and is usually the most up-to-date reference available. It's written in Texinfo format and should be available on Unix systems in the "Info" documentation hierarchy. You can read it either with the command-line **info** program

```
yarkon$ info cvs
```

or by pressing Ctrl+H and then typing "i" inside Emacs. If neither of these works for you, consult your local Unix guru (or see Chapter 3 regarding installation issues). You'll definitely want to have the Cederqvist at your fingertips if you're going to be using CVS regularly.

Invoking CVS

CVS is one program, but it can perform many different actions: updating, committing, branching, diffing, and so on. When you invoke CVS, you must specify which action you want to perform. Thus, the format of a CVS invocation is:

```
yarkon$ cvs command
```

For example, you can type in

```
yarkon$ cvs update
yarkon$ cvs diff
yarkon$ cvs commit
```

and so on. Do not run these commands yet; we are just listing them for a better understanding. We will get to real CVS work very soon.

Both CVS and the command can take options. Options that affect the behavior of CVS, independently of the command being run, are called *global options*; command-specific options are called simply *command options*. Global options always go to the left of the command, and command options go to its right. Therefore, in this line of code

```
yarkon$ cvs -Q update -p
```

-Q is a global option, and **-p** is a command option. (If you're curious, **-Q** means "quietly"—that is, suppress all diagnostic output, and print error messages only if the command absolutely cannot be completed for some reason; **-p** means to send the results of the **update** command to standard output instead of to files.)

Repository Access and the Working Environment

Before you can do anything, you must tell CVS the location of the repository you'll be accessing. This isn't a concern if you already have a working copy checked out—any working copy knows what repository it came from, so CVS can automatically deduce the repository for a given working copy. However, let's assume you don't have a working copy yet, so you need to tell CVS explicitly where to go. This is done with the **-d** global option (the **-d** stands for "directory," an abbreviation for which there is a historical justification, although **-r** for "repository" might have been better), followed by the path to the repository. For example, assuming the repository is on the local machine in /usr/local/cvs (a fairly standard location), you would execute the following code:

```
yarkon$ cvs -d /usr/local/cvs command
```

In many cases, however, the repository is on another machine and must therefore be reached over the network. CVS provides a choice of network access methods; which one you'll use

depends mostly on the security needs of the repository machine—hereinafter referred to as "the server." We cover setting up the server to allow various remote access methods later in this book; here, we'll deal only with the client side.

Fortunately, all the remote access methods share a common invocation syntax. In general, to specify a remote repository as opposed to a local one, you just use a longer repository path. You first name the access method, delimited on each side by colons, followed by the username and the server name (joined with an @ sign), another separator colon, and finally the path to the repository directory on the server.

Let's look at the **pserver** access method, which stands for "password-authenticated server":

```
yarkon$ cvs -d :pserver:jrandom@cvs.foobar.com:/usr/local/cvs login
(Logging in to jrandom@cvs.foobar.com)
CVS password: (enter your CVS password here)
yarkon$
```

The long repository path following **-d** told CVS to use the **pserver** access method, with the username **jrandom**, on the server **cvs.foobar.com**, which has a CVS repository in **/usr/local/cvs**. There's no requirement that the hostname be "cvs.something.com," by the way; that's a common convention, but it could just as easily have been:

```
yarkon$ cvs -d :pserver:jrandom@fish.foobar.org:/usr/local/cvs login
```

The command that was actually run was **login**, which verifies that you are authorized to work with this repository. It prompts for a password, then contacts the server to verify the password. Following Unix custom, **cvs login** returns silently if the login succeeds; it shows an error message if it fails (for instance, because the password is incorrect).

You have to log in only once from your local machine to a given CVS server. After a successful login, CVS stores the password in your home directory, in a file called .cvspass. It consults that file every time a repository is contacted via the **pserver** method, so you have to run **login** only the first time you access a given CVS server from a particular client machine. Of course, you can rerun **cvs login** at any time if the password changes. **pserver** is currently the only access method requiring an initial login like this; with the others, you can start running regular CVS commands immediately.

Once you've stored the authentication information in your .cvspass file, you can run other CVS commands using the same command-line syntax:

```
yarkon$ cvs -d :pserver:jrandom@cvs.foobar.com:/usr/local/cvs command
```

Getting **pserver** to work in Windows might require an extra step. Windows doesn't have the Unix concept of a home directory, so CVS doesn't know where to put the .cvspass file. You'll have to specify a location. It is usual to designate the root of the C: drive as the home directory:

```
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: (enter your CVS password here)
C:\WINDOWS>
```

In addition to **pserver**, CVS supports the **ext** method (which uses an external connection program, such as **rsh** or **ssh**), **kserver** (for the Kerberos security system version 4), and **gserver** (which uses the GSSAPI, or Generic Security Services API, and also handles Kerberos versions 5 and higher). These methods are similar to **pserver**, but each has its own idiosyncrasies.

Of these, the **ext** method is probably the most commonly used. If you can log in to the server with **rsh** or **ssh**, you can use the **ext** method. You can test it like this:

```
yarkon$ rsh -l jrandom cvs.foobar.com
Password: enter your login password here
```

Okay, let's assume you successfully logged in and logged out of the server with **rsh**, so now you're back on the original client machine:

```
yarkon$ CVS_RSH=rsh; export CVS_RSH
yarkon$ cvs -d :ext:jrandom@cvs.foobar.com:/usr/local/cvs command
```

The first line sets (in Unix Bourne shell syntax) the **CVS_RSH** environment variable to **rsh**, which tells CVS to use the **rsh** program to connect. The second line can be any CVS command; you will be prompted for your password so CVS can log in to the server.

For Windows, try this:

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

The rest of the tour will use the Bourne syntax; translate for your environment as necessary.

To use **ssh** (the Secure Shell) instead of **rsh**, just set the **CVS_RSH** variable appropriately:

```
yarkon$ CVS_RSH=ssh; export CVS_RSH
```

Don't get thrown by the fact that the variable's name is **CVS_RSH** but you're setting its value to **ssh**. There are historical reasons for this (the catchall Unix excuse, we know). **CVS_RSH** can point to the name of any program capable of logging you in to the remote server, running commands, and receiving their output. After **rsh**, **ssh** is probably the most common such program, although there are probably others. Note that this program must not modify its data stream in any way. This disqualifies the Windows NT **rsh**, because it converts (or attempts to convert) between the DOS and Unix line-ending conventions. You'd have to get some other **rsh** for Windows or use a different access method.

The **gserver** and **kserver** methods are not used as often as the others and are not covered here. They're quite similar to what we've covered so far; see the Cederqvist manual for details.

If you use only one repository and don't want to type ***-d repository*** each time, just set the **CVSROOT** environment variable (which perhaps should have been named **CVSREPOS**, but it's too late to change that now), like this

```
yarkon$ CVSROOT=/usr/local/cvs
yarkon$ export CVSROOT
yarkon$ echo $CVSROOT
/usr/local/cvs
yarkon$
```

or like this:

```
yarkon$ CVSROOT=:pserver:jrandom@cvs.foobar.com:/usr/local/cvs
yarkon$ export CVSROOT
yarkon$ echo $CVSROOT
:pserver:jrandom@cvs.foobar.com:/usr/local/cvs
yarkon$
```

The rest of this tour assumes that you've set **CVSROOT** to point to your repository, so the examples will not show the **-d** option. If you need to access many different repositories, you should not set **CVSROOT** and should use ***-d repository*** when you need to specify the repository.

Starting a New Project

If you're learning CVS in order to work on a project that's already under CVS control (that is, it is kept in a repository somewhere), you'll probably want to skip down to the next section, "Checking Out a Working Copy." On the other hand, if you want to take existing source code and put it into CVS, this is the section for you. Note that it still assumes you have access to an existing repository; see Chapter 3 if you need to set up a repository first.

Putting a new project into a CVS repository is known as *importing*. The CVS command is

```
yarkon$ cvs import
```

except that it needs some more options (and needs to be in the right location) to succeed. First, go into the top-level directory of your project tree:

```
yarkon$ cd myproj
yarkon$ ls
README.txt  a-subdir/   b-subdir/   hello.c
yarkon$
```

This project has two files—README.txt and hello.c—in the top level, plus two sub-directories—a-subdir and b-subdir—plus some more files (not shown in the example) inside those subdirectories. When you import a project, CVS imports *everything* in the tree, starting from the current directory and working its way down. Therefore, you should make sure that the only files in the tree are ones you want to be permanent parts of the project. Any old backup files, scratch files, and so on should all be cleaned out.

The general syntax of an **import** command is:

```
yarkon$ cvs import -m "log msg" projname vendortag releasetag
```

The **-m** flag (for *message*) is for specifying a short message describing the import. This will be the first log message for the entire project; every commit thereafter will also have its own log message. These messages are mandatory; if you don't give the **-m** flag, CVS automatically starts up an editor (by consulting the **EDITOR** environment variable) for you to type a log message in. After you save the log message and exit the editor, the **import** then continues.

The ***projname*** argument is the project's name (we'll use "myproj"). This is the name under which you'll check out the project from the repository. (What actually happens is that a directory of that name gets created in the repository, but more on that in Chapter 3.) The name you choose now does not need to be the same as the name of the current directory, although in most cases it is.

The ***vendortag*** and ***releasetag*** arguments are a bit of bookkeeping for CVS. Don't worry about them now; it rarely matters what you use. In Chapter 4 you'll learn about the rare circumstances in which they're significant. For now, we'll use a username and **start** for those arguments.

We're ready to run **import**:

```
yarkon$ 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
yarkon$
```

Congratulations! If you ran that command (or something similar), you've finally done something that affects the repository.

Reading over the output of the **import** command, you'll notice that CVS precedes each file name with a single letter—in this case, "N" for "new file." The use of a single letter on the left to indicate the status of a file is a general pattern in CVS command output. We'll see it later in the **checkout** and **update** commands as well.

You might think that, having just imported the project, you can start working in the tree immediately. This is not the case, however. The current directory tree is still not a CVS working copy. It was the source for the **import** command, true, but it wasn't magically changed into a CVS working copy merely by virtue of having been imported. To get a working copy, you need to check one out from the repository.

First, though, you might want to archive the current project tree. The reason is that once the sources are in CVS, you don't want to confuse yourself by accidentally editing copies that aren't in version control (because those changes won't become part of the project's history). You should do all of your editing in a working copy from now on. However, you also don't want to remove the imported tree entirely, because you haven't yet verified that the repository actually has the files. Of course, you can be 99.999 percent certain that it does because the **import** command returned with no error, but why take chances? As every programmer knows, paranoia pays. Therefore, do something like this:

```
yarkon$ ls
README.txt  a-subdir/   b-subdir/   hello.c
yarkon$ cd ..
yarkon$ ls
myproj/
yarkon$  tar zcf was_myproj.tar.gz myproj
yarkon$ rm -fr myproj/
yarkon$ ls
was_myproj/
yarkon$
```

There. You still have the original files, but they're clearly named as an obsolete version, so they won't be in the way when you get a real working copy. Now, you're ready to check out.

Checking Out a Working Copy

The command to check out a project is exactly what you think it is:

```
yarkon$ 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

yarkon$ ls
myproj/      was_myproj/
yarkon$ cd myproj
yarkon$ ls
CVS/        README.txt  a-subdir/   b-subdir/   hello.c
yarkon$
```

Well done! Your first working copy! Its contents are exactly the same as what you imported, with the addition of a subdirectory named "CVS." That's where CVS stores version-control information. Actually, each directory in the project has a CVS subdirectory:

```
yarkon$ ls a-subdir
CVS/        subsubdir/  whatever.c
yarkon$ ls a-subdir/subsubdir/
CVS/    fish.c
yarkon$ ls b-subdir
CVS/      random.c
```

Note

The fact that CVS keeps its revision information in subdirectories named CVS means that your project can never contain subdirectories of its own named CVS. In practice, we've never heard of this being a problem.

Before editing any files, let's take a peek inside CVS:

```
yarkon$ cd CVS
yarkon$ ls
Entries     Repository  Root
yarkon$ cat Root
/usr/local/cvs
yarkon$ cat Repository
myproj
yarkon$
```

Nothing difficult there. The Root file points to repository, and the Repository file points to a project inside the repository. If that's a little confusing, let us explain.

There is a longstanding confusion about terminology in CVS. The word "repository" is used to refer to two different things. Sometimes, it means the root directory of a repository (for example, /usr/local/cvs), which can contain many projects; this is what the Root file refers to. However, other times, it means one particular project-specific subdirectory *within* a repository root (for example, /usr/local/cvs/myproj, /usr/local/cvs/yourproj, or /usr/local/cvs/fish). The Repository file inside a CVS subdirectory takes the latter meaning.

In this book, "repository" generally means Root (that is, the top-level repository), although we might occasionally use it to mean a project-specific subdirectory. If the intended sense can't be figured out from the context, we will include clarifying text.

Note that the Repository file can sometimes contain an absolute, rather than a relative, path to the project name. This can make it slightly redundant with the Root file:

```
yarkon$ cd CVS
yarkon$ cat Root
:pserver:jrandom@cvs.foobar.com:/usr/local/cvs
yarkon$ cat Repository
/usr/local/cvs/myproj
yarkon$
```

The Entries file stores information about the individual files in the project. Each line deals with one file, and there are lines for only files or subdirectories in the immediate parent directory. Here's the top-level CVS/Entries file in myproj:

```
yarkon$ cat Entries
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 2001//
/hello.c/1.1.1.1/Sun Apr 18 18:18:22 2001//
D/a-subdir////
D/b-subdir////
```

The format of each line is

```
/filename/revision number/datestamp//
```

and the directory lines are prefixed with **D**. (CVS doesn't really keep a change history for directories, so the fields for ***revision number*** and ***datestamp*** are empty.)

The datestamps record the date and time of the last update (in Universal Time, not local time) of the files in the working copy. That way, CVS can easily tell whether a file has been modified since the last **checkout**, **update**, or **commit** command. If the file system timestamp differs from the timestamp in the CVS/Entries file, CVS knows (without even having to consult the repository) that the file was probably modified.

If you take a look at the CVS/* files in one of the subdirectories

```
yarkon$ cd a-subdir/CVS
yarkon$ cat Root
/usr/local/cvs
yarkon$ cat Repository
myproj/a-subdir
yarkon$ cat Entries
/whatever.c/1.1.1.1/Sun Apr 18 18:18:22 2001//
```

```
D/subsubdir////
yarkon$
```

you can see that the root repository has not changed, but the Repository file spells out the location of this subdirectory of the project, and the Entries file contains different lines.

Immediately after import, the revision number of every file in the project is shown as 1.1.1.1. This initial revision number is a bit of a special case, so we won't examine it in detail just yet; we'll take a closer look at revision numbers after we've committed some changes.

Making a Change

The project, as it stands, doesn't do much. Here are the contents of hello.c:

```
yarkon$ cat hello.c
#include <stdio.h>

void
main ()
{
   printf ("Hello, world!\n");
}
```

Let's make the first change to the project since importing it; we'll add the line

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

right after the **Hello, world!**. Invoke your favorite editor and make the change:

```
yarkon$ emacs hello.c
  ...
```

This was a fairly simple change, one where you're not likely to forget what you did. However, in a larger, more complex project, it's quite possible that you might edit a file, be interrupted by something else, and return several days later and be unable to remember exactly what you did—or even whether you changed anything at all. This brings us to our first "CVS Saves Your Life" situation: comparing your working copy against the repository.

Finding Out What You (and Others) Did: **update** and **diff**

Previously, we talked about updating as a way of bringing changes down from the repository into your working copy—that is, as a way of getting other people's changes. However, **update** is really a bit more complex; it compares the overall state of the working copy with the state of the project in the repository. Even if nothing in the repository has changed since the **checkout** command was executed, something in the working copy might have changed, and **update** will show that, too:

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

The **M** next to **hello.c** means the file has been modified since it was last checked out, and the modifications have not yet been committed to the repository.

Sometimes, merely knowing which files you've edited is all you need. However, if you want a more detailed look at the changes, you can get a full report in **diff** format. The **diff** command compares the possibly modified files in the working copy to their counterparts in the repository and displays any differences:

```
yarkon$ 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
```

That's helpful, if a bit obscure, but there's still a lot of cruft in the output. For starters, you can ignore most of the first few lines. They just name the repository file and give the number of the last checked-in revision. These are useful pieces of information under other circumstances (we'll look more closely at them later), but you don't need them when you're just trying to get a sense of what changes have been made in the working copy.

A more serious impediment to reading the diff is that CVS is announcing its entry as it goes into each directory during the update. This can be useful during long updates on large projects, because it gives you a sense of how much longer the command will take, but right now it's just getting in the way of reading the diff. Let's tell CVS to be quiet about where it's working, with the **-Q** global option:

```
yarkon$ 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");
```

Better—at least some of the cruft is gone. However, the diff is still hard to read. It's telling you that at line 6, a new line was added (that is, what became line 7), whose contents were:

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

The preceding ">" in the diff tells you that this line is present in the newer version of the file but not in the older one.

The format can be made even more readable, however. Most people find "context" **diff** format easier to read because it displays a few lines of context on either side of a change. Context diffs are generated by passing the **-c** flag to **diff**:

```
yarkon$ 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     2001/04/18 18:18:22     1.1.1.1
--- hello.c     2001/04/19 02:17:07
***************
*** 4,7 ****
--- 4,8 --
  main ()
  {
    printf ("Hello, world!\n");
+   printf ("Goodbye, world!\n");
  }
```

How much clearer could it get? Even if you're not used to reading context diffs, a glance at the preceding output probably makes it obvious what happened: A new line was added (the **+** in the first column signifies an added line) between the line that prints **Hello, world!** and the final curly brace.

We don't need to be able to read context diffs perfectly (that's **patch**'s job), but it's worth taking the time to acquire at least a passing familiarity with the format. The first two lines (after the introductory cruft) are

```
*** hello.c     2001/04/18 18:18:22     1.1.1.1
--- hello.c     2001/04/19 02:17:07
```

and they tell you what is being diffed against what. In this case, revision 1.1.1.1 of hello.c is being compared against a modified version of the same file (thus, there's no revision number for the second line, because the only changes that haven't been committed to the

repository yet are those in the working copy). The lines of asterisks and dashes identify sections farther down in the diff. Later on, a line of asterisks, with a line number range embedded, precedes a section from the original file. Then a line of dashes, with a new and potentially different line number range embedded, precedes a section from the modified file. These sections are organized into contrasting pairs (known as "hunks"), one side from the old file and the other side from the new.

Our diff has one hunk:

```
***************
*** 4,7 ****
--- 4,8 --
  main ()
  {
    printf ("Hello, world!\n");
+   printf ("Goodbye, world!\n");
  }
```

The first section of the hunk is empty, meaning that no material was removed from the original file. The second section shows that, in the corresponding place in the new file, one line has been added; it's marked with a "+". (When the diff quotes excerpts from files, it reserves the first two columns on the left for special codes, such as "+", so the entire excerpt appears to be indented by two spaces. This extra indentation is stripped off when the **diff** command is applied, of course.)

The line number ranges show the hunk's coverage, including context lines. In the original file, the hunk was in lines 4 through 7; in the new file, it's lines 4 through 8 (because a line has been added). Note that the diff didn't need to show any material from the original file because nothing was removed; it just showed the range and moved on to the second half of the hunk.

Here's another context diff, from an actual project of ours:

```
yarkon$ 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   2001/04/13 22:29:44     1.76
--- cvs2cl.pl   2001/04/19 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);
        }
      }
```

The exclamation point shows that the marked line differs between the old and new files. Because there are no "+" or "-" signs, we know that the total number of lines in the file has remained the same.

Here's one more context diff from the same project, slightly more complex this time:

```
yarkon$ 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   2001/04/13 22:29:44     1.76
--- cvs2cl.pl   2001/04/19 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, we 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:
```

This diff has two hunks. In the first, five lines were removed (these lines are shown only in the first section of the hunk, and the second section's line count shows that it has five fewer lines). An unbroken line of asterisks forms the boundary between hunks, and in the second hunk we see that two lines have been added: a blank line and a pointless comment. Note how the line numbers compensate for the effect of the previous hunk. In the original file, the second hunk's range of the area was lines 223 through 228; in the new file, because of the deletion that took place in the *first* hunk, the range is in lines 218 through 225.

If you understand **diff** so far, you already qualify as an expert in many situations.

CVS and Implied Arguments

In each of the CVS commands so far, you might have noticed that no files were specified on the command line. We ran

```
yarkon$ cvs diff
```

instead of:

```
yarkon$ cvs diff hello.c
```

In addition, we ran

```
yarkon$ cvs update
```

instead of:

```
yarkon$ cvs update hello.c
```

The principle at work here is that if you don't name any files, CVS acts on all files for which the command could possibly be appropriate. This even includes files in subdirectories beneath the current directory; CVS automatically descends from the current directory through every subdirectory in the tree. For example, if you modified b-subdir/random.c and a-subdir/subsubdir/fish.c, running **update** may result in this

```
yarkon$ 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
yarkon$
```

or better yet

```
yarkon$ cvs -q update
M hello.c
M a-subdir/subsubdir/fish.c
M b-subdir/random.c
yarkon$
```

Note

*The **-q** flag is a less emphatic version of **-Q**. Had we used **-Q**, the command would have printed out nothing at all, because the modification notices are considered nonessential informational messages. Using the lowercase **-q** is less strict; it suppresses the messages we probably don't want, while allowing more useful messages to pass through.*

You can also name specific files for the update

```
yarkon$ cvs update hello.c b-subdir/random.c
M hello.c
M b-subdir/random.c
yarkon$
```

and CVS will examine only those files, ignoring all others.

It's actually more common to run **update** without restricting it to certain files. In most situations, you'll want to update the entire directory tree at once. Remember, the updates we're doing here show only that some files have been locally modified, because nothing has changed yet in the repository. When other people are working on the project with you, there's always the chance that running **update** will pull some new changes down from the repository and incorporate them into your local files. In that case, you might find it slightly more useful to name which files you want updated.

The same principle can be applied to other CVS commands. For example, with **diff**, you can choose to view the changes one file at a time

```
yarkon$ 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   2001/04/18 18:18:22     1.1.1.1
--- b-subdir/random.c   2001/04/19 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");
! }
```

or to see all the changes at once (hang on to your seat; this is going to be a big diff):

```
yarkon$ 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     2001/04/18 18:18:22     1.1.1.1
--- hello.c     2001/04/19 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   2001/04/18 18:18:22     1.1.1.1
--- a-subdir/subsubdir/fish.c   2001/04/19 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   2001/04/18 18:18:22     1.1.1.1
--- b-subdir/random.c   2001/04/19 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");
! }
```

Anyway, as you can see from these diffs, this project is clearly ready for prime time. Let's commit the changes to the repository.

Committing

The **commit** command sends modifications to the repository. If you don't name any files, a commit will send all changes to the repository; otherwise, you can pass the names of one or more files to be committed (in that case, other files are ignored).

Here, we commit one file by name and two by inference:

```
yarkon$ 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
yarkon$ 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
yarkon$
```

Take a moment to read over the output carefully. Most of what it says is pretty self-explanatory. One thing you may notice is that revision numbers have been incremented (as expected), but the original revisions are listed as 1.1 instead of 1.1.1.1 as we saw in the Entries file earlier.

The explanation for this discrepancy is not very important. It concerns a special meaning that CVS attaches to revision 1.1.1.1. For most purposes, we can just say that files receive a revision number of 1.1 when imported, but the number is displayed—for reasons known only to CVS—as 1.1.1.1 in the Entries file, until the first commit.

Revision Numbers

Each file in a project has its own revision number. When a file is committed, the last portion of the revision number is incremented by one. Thus, at any given time, the various files making up a project might have very different revision numbers. This just means that some files have been changed (committed) more often than others.

(You might be wondering, what's the point of the part to the left of the decimal point, if only the part on the right ever changes? Actually, although CVS never automatically increments the number on the left, that number can be incremented on request by a user. Because this is a rarely used feature, we don't cover it in this book.)

Version vs. Revision

The internal revision number that CVS keeps for each file is unrelated to the version number of the software product of which the files are part. For example, you might have a project composed of three files, whose internal revision numbers on May 3, 2001, were 1.2, 1.7, and 2.48. On that day, you package up a new release of the software and release it as SlickoSoft version 3. This is purely a marketing decision and doesn't affect the CVS revisions at all. The CVS revision numbers are invisible to your customers (unless you give them repository access); the only publicly visible number is the "3" in Version 3. You could have called it version 1,729 as far as CVS is concerned—the version number (or "release" number) has nothing to do with CVS's internal change tracking.

To avoid confusion, we'll use the word "revision" to refer exclusively to the internal revision numbers of files under CVS control. We might still call CVS a "version control system," however, because "revision control system" just sounds too awkward.

In the example project that we've been using, we just committed changes to three files. Each of those files is now revision 1.2, but the remaining files in the project are still revision 1.1. When you check out a project, you get each file at its highest revision so far. Here is what qsmith would see if he checked out myproj right now and looked at the revision numbers for the top-level directory:

```
paste$ 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
paste$ cd myproj/CVS
paste$ cat Entries
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 2001//
/hello.c/1.2/Mon Apr 19 06:35:15 2001//
D/a-subdir////
D/b-subdir////
paste$
```

The file hello.c (among others) is now at revision 1.2, whereas README.txt is still at the initial revision (revision 1.1.1.1, also known as 1.1).

If he adds the line

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

to hello.c and commits it, the file's revision number will be incremented once more:

```
paste$ 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
paste$
```

Now hello.c is revision 1.3, fish.c and random.c still are revision 1.2, and every other file is revision 1.1.

Tip

*The command was given as **cvs ci** instead of **cvs commit**. Most CVS commands have short forms, to make typing easier. For the **checkout**, **update**, and **commit** commands, the abbreviated versions are **co**, **up**, and **ci**, respectively. You can get a list of all of the short forms by running the command **cvs—help-synonyms**.*

You can usually ignore a file's revision number. In most situations, the numbers are just internal bookkeeping that CVS handles automatically. However, being able to find and compare revision numbers is extremely handy when you have to retrieve (or use the **diff** command against) an earlier copy of a file.

Examining the Entries file isn't the only way to discover a revision number. You can also use the **status** command

```
paste$ cvs status hello.c
===================================================================
File: hello.c           Status: Up-to-date

   Working revision:    1.3     Tue Apr 20 02:34:42 2001
   Repository revision: 1.3     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

which, if invoked without any files being named, shows the status of every file in the project:

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

   Working revision:    1.1.1.1 Sun Apr 18 18:18:22 2001
   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 2001
   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 2001
   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 2001
   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 2001
   Repository revision: 1.2     /usr/local/cvs/myproj/b-subdir/random.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

paste$
```

Just ignore the parts of that output that you don't understand. In fact, that's generally good advice with CVS. Often, the one little bit of information you're looking for will be accompanied by reams of information that you don't care about at all, and maybe don't even understand. This situation is normal. Just pick out what you need, and don't worry about the rest.

In the previous example, the parts we care about are the first three lines (not counting the blank line) of each file's status output. The first line is the most important; it tells you the file's name and its status in the working copy. All of the files are currently in sync with the repository, so they all say **Up-to-date**. However, if random.c has been modified but not committed, it might read like this:

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

   Working revision:    1.2     Mon Apr 19 06:35:27 2001
   Repository revision: 1.2     /usr/local/cvs/myproj/b-subdir/random.c,v
   Sticky Tag:          (none)
```

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

The **Working revision** and **Repository revision** tell you whether the file is out of sync with the repository. Returning to our original working copy (jrandom's copy, which hasn't seen the new change to hello.c yet), we see:

```
yarkon$ cvs status hello.c
===================================================================
File: hello.c           Status: Needs Patch

   Working revision:    1.2     Mon Apr 19 02:17:07 2001
   Repository revision: 1.3     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

yarkon$
```

This tells us that someone has committed a change to hello.c, bringing the repository copy to revision 1.3, but that this working copy is still on revision 1.2. The line **Status: Needs Patch** means that the next **update** command will retrieve those changes from the repository and patch them into the working copy's file.

Let's pretend for the moment that we don't know anything about qsmith's change to hello.c, so we don't run **status** or **update**. Instead, we just start editing the file, making a slightly different change at the same location. This brings us to our first conflict.

Detecting and Resolving Conflicts

Detecting a conflict is easy enough. When you run **update**, CVS tells you, in no uncertain terms, that there's a conflict. But first, let's create the conflict. We edit hello.c to insert the line

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

right where qsmith committed this:

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

At this point, the status of our copy of hello.c is

```
yarkon$ cvs status hello.c
===================================================================
File: hello.c           Status: Needs Merge
```

```
   Working revision:    1.2     Mon Apr 19 02:17:07 2001
   Repository revision: 1.3     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

yarkon$
```

meaning that there are changes both in the repository and the working copy, and these changes need to be merged. (CVS isn't aware that the changes will conflict, because we haven't run **update** yet.) When we do the update, we see this:

```
yarkon$ 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
yarkon$
```

The last line of output is the giveaway. The **C** in the left margin next to the file name indicates that although changes have been merged, they conflict. The contents of hello.c now show both changes:

```
#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");
}
```

Conflicts are always shown delimited by *conflict markers*, in the following format:

```
<<<<<<< (filename)
  the uncommitted changes in the working copy
  blah blah blah
=======
```

```
  the new changes that came from the repository
  blah blah blah
  and so on
>>>>>>> (latest revision number in the repository)
```

The Entries file also shows that the file is in a halfway state at the moment:

```
yarkon$ cat CVS/Entries
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 2001//
D/a-subdir////
D/b-subdir////
/hello.c/1.3/Result of merge+Tue Apr 20 03:59:09 2001//
yarkon$
```

The way to resolve the conflict is to edit the file so that it contains whatever text is appropriate, removing the conflict markers in the process, and then to use the **commit** command. This doesn't necessarily mean choosing one change over another; you could decide neither change is sufficient and completely rewrite the conflicting section (or indeed the whole file). In this case, we'll adjust in favor of the first change, but with capitalization and punctuation slightly different from qsmith's:

```
yarkon$ emacs hello.c
  (make the edits...)
yarkon$ cat hello.c
#include <stdio.h>

void
main ()
{
  printf ("Hello, world!\n");
  printf ("BETWEEN HELLO AND GOODBYE.\n");
  printf ("Goodbye, world!\n");
}
yarkon$ 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
yarkon$
```

Finding Out Who Did What (Browsing Log Messages)

By now, the project has undergone several changes. If you're trying to get an overview of what has happened so far, you don't necessarily want to examine every diff in detail. Browsing the log messages would be ideal, and you can accomplish this with the **log** command:

```
yarkon$ cvs log
(pages upon pages of output omitted)
```

The log output tends to be a bit verbose. Let's look at the log messages for just one file:

```
yarkon$ 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: 2001/04/20 04:14:37;  author: jrandom;  state: Exp;  lines: +1 -1
adjusted middle line
----------
revision 1.3
date: 2001/04/20 02:30:05;  author: qsmith;  state: Exp;  lines: +1 -0
added new middle line
----------
revision 1.2
date: 2001/04/19 06:35:15;  author: jrandom;  state: Exp;  lines: +1 -0
print goodbye too
----------
revision 1.1
date: 2001/04/18 18:18:22;  author: jrandom;  state: Exp;
branches:  1.1.1;
Initial revision
----------
revision 1.1.1.1
date: 2001/04/18 18:18:22;  author: jrandom;  state: Exp;  lines: +0 -0
initial import into CVS
=============================================================================
yarkon$
```

As usual, there's a lot of information at the top that you can just ignore. The good stuff comes after each line of dashes, in a format that is self-explanatory.

When many files are sent in the same **commit** command, they all share the same log message—a fact that can be useful in tracing changes. For example, remember when we committed fish.c and random.c simultaneously? It was done like this:

```
yarkon$ 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
yarkon$
```

The effect of this was to commit both files with the same log message: "Filled out C code." (As it happened, both files started at revision 1.1 and went to 1.2, but that's just a coincidence. If random.c had been at revision 1.29, it would have moved to 1.30 with this commit, and its revision 1.30 would have had the same log message as fish.c's revision 1.2.)

When you run **cvs log** on both files, you'll see the shared message:

```
yarkon$ 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: 2001/04/19 06:35:27;  author: jrandom;  state: Exp;  lines: +8 -1
filled out C code
----------------
revision 1.1
```

```
date: 2001/04/18 18:18:22;  author: jrandom;  state: Exp;
branches:  1.1.1;
Initial revision
----------
revision 1.1.1.1
date: 2001/04/18 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: 2001/04/19 06:35:27;  author: jrandom;  state: Exp;  lines: +8 -1
filled out C code
----------
revision 1.1
date: 2001/04/18 18:18:22;  author: jrandom;  state: Exp;
branches:  1.1.1;
Initial revision
----------
revision 1.1.1.1
date: 2001/04/18 18:18:22;  author: jrandom;  state: Exp;  lines: +0 -0
initial import into CVS
=============================================================================
yarkon$
```

From this output, you'll know that the two revisions were part of the same commit (the fact that the timestamps on the two revisions are the same, or very close, is further evidence).

Browsing log messages is a good way to get a quick overview of what's been going on in a project or to find out what happened to a specific file at a certain time. There are also free tools available to convert raw **cvs log** output to more concise and readable formats (such as GNU-style ChangeLog). We don't cover those tools in this tour, but they are introduced in Chapter 10.

Examining and Reverting Changes

Suppose that, in the course of browsing the logs, qsmith sees that jrandom made the most recent change to hello.c

```
revision 1.4
date: 2001/04/20 04:14:37;  author: jrandom;  state: Exp;  lines: +1 -1
adjusted middle line
```

and wonders what jrandom did. In formal terms, the question that qsmith is asking is, "What's the difference between my revision (1.3) of hello.c, and jrandom's revision right after it (1.4)?" The way to find out is with the **diff** command with the **-r** command option to specify both of them:

```
paste$ 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     2001/04/20 02:30:05     1.3
--- hello.c     2001/04/20 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");
  }
paste$
```

When viewed this way, the change is pretty clear. Because the revision numbers are given in chronological order (usually a good idea), the diff shows them in order. If only one revision number is given, CVS uses the revision of the current working copy for the other.

When qsmith sees this change, he instantly decides he likes his way better and resolves to "undo"—that is, to step back by one revision. However, this doesn't mean that he wants to lose his revision 1.4. Although, in an absolute technical sense, it's probably possible to

achieve that effect in CVS, there's almost never any reason to do so. It's preferable to keep revision 1.4 in the history and make a new revision 1.5 that looks exactly like 1.3. That way the **undo** event itself is part of the file's history. The only question is, how can you retrieve the contents of revision 1.3 and put them into 1.5?

In this particular case, the change is a very simple one, so qsmith can probably just edit the file by hand to mirror revision 1.3 and then use the **commit** command. However, if the changes are more complex (as they usually are in a real-life project), trying to re-create the old revision manually will be hopelessly error-prone. Therefore, we'll have qsmith use CVS to retrieve and recommit the older revision's contents.

There are two equally good ways to do this: the slow, plodding way and the fast, fancy way. We'll examine the slow, plodding way first.

The Slow Method of Reverting

This method involves passing the **-p** flag to **update**, in conjunction with **-r**. The **-p** option sends the contents of the named revision to standard output. By itself, this isn't terribly helpful; the contents of the file fly by on the display, leaving the working copy unchanged. However, if you redirect the standard output into the file, the file will hold the contents of the older revision. It's as if the file had been hand-edited into that state.

First, though, qsmith needs to get up to date with respect to the repository:

```
paste$ cvs update
cvs update: Updating .
U hello.c
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
paste$ cat hello.c
#include <stdio.h>

void
main ()
{
  printf ("Hello, world!\n");
  printf ("how are you?\n");
  printf ("Goodbye, world!\n");
}
paste$
```

Next, he runs **update -p** to make sure that revision 1.3 is the one he wants:

```
paste$ 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");
}
```

Oops, there are a few lines of chaff at the beginning. They aren't actually being sent to standard output, but rather to standard error, so they're harmless. Nevertheless, they make reading the output more difficult and can be suppressed with **-Q**:

```
paste$ 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");
}
paste$
```

There—that's exactly what qsmith was hoping to retrieve. The next step is to put that content into the working copy's file, using a Unix redirect (that's what the ">" does):

```
paste$ cvs -Q update -p -r 1.3 hello.c > hello.c
paste$ cvs update
cvs update: Updating .
M hello.c
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
paste$
```

Now when **update** is run, the file is listed as modified, which makes sense because its contents have changed. Specifically, it has the same content as the old revision 1.3 (not that CVS is aware of its being identical to a previous revision—it just knows the file has been modified). If qsmith wants to make extra sure, he can run the **diff** command to check:

```
paste$ 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     2001/04/20 04:14:37     1.4
--- hello.c     2001/04/20 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");
  }
paste$
```

Yes, that's exactly what he wanted: a pure reversion—in fact, it is the reverse of the diff he previously obtained. Satisfied, he commits:

```
paste$ 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
paste$
```

The Fast Method of Reverting

The fast, fancy way of updating is to use the **-j** (for "join") flag with the **update** command. This flag is like **-r** in that it takes a revision number, and you can use up to two **-j**'s at once. CVS calculates the difference between the two named revisions and applies that difference as a patch to the file in question (so the order in which you give the revisions is important).

Thus, assuming qsmith's copy is up to date, he can just do this:

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

When you need to revert only one file, there's not really much difference between the plodding and fast methods. Later in the book, you'll see how the fast method is much better for reverting multiple files at once. In the meantime, use whichever way you're more comfortable with.

Other Useful CVS Commands

At this point, you should be pretty comfortable with basic CVS operation. We'll abandon the tour narrative and introduce a few more useful commands in summarized form.

Adding Files

Adding a file is a two-step process: First you run the **add** command on it, then you run **commit**. The file won't actually appear in the repository until **commit** is run:

Reverting Is Not a Substitute for Communication

In all likelihood, what qsmith did in our example was quite rude. When you're working on a real project with other people and you think that someone has committed a bad change, the first thing you should do is talk to him or her about it. Maybe there's a good reason for the change, or maybe he or she just didn't think things through. Either way, there's no reason to rush and revert. A full record of everything that happens is stored permanently in CVS, so you can always revert to a previous revision after consulting with whoever made the changes.

If you're a project maintainer facing a deadline or you feel you have the right and the need to revert the change unconditionally, then do so—but follow it immediately with an email to the author whose change was reverted, explaining why you did it and what needs to be fixed to recommit the change.

```
yarkon$ cvs add newfile.c
cvs add: scheduling file 'newfile.c' for addition
cvs add: use 'cvs commit' to add this file permanently
yarkon$ 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  <--  newfile.c
initial revision: 1.1
done
yarkon$
```

Adding Directories

Unlike adding a file, adding a new directory is done in one step; there's no need to run **commit** afterward:

```
yarkon$ mkdir c-subdir
yarkon$ cvs add c-subdir
Directory /usr/local/cvs/myproj/c-subdir added to the repository
yarkon$
```

If you look inside the new directory in the working copy, you'll see that a CVS subdirectory was created automatically by the **add** command:

```
yarkon$ ls c-subdir
CVS/
yarkon$ ls c-subdir/CVS
Entries     Repository  Root
yarkon$
```

Now you can add files (or new directories) inside it, as with any other working copy directory.

Removing Files

Removing a file is similar to adding one, except there's an extra step: You have to remove the file from the working copy first:

```
yarkon$ rm newfile.c
yarkon$ cvs remove newfile.c
cvs remove: scheduling 'newfile.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
yarkon$ cvs ci -m "removed newfile.c" newfile.c
Removing newfile.c;
/usr/local/cvs/myproj/newfile.c,v  <-  newfile.c
```

CVS and Binary Files

Until now, we've left unexposed the dirty little secret of CVS, which is that it doesn't handle binary files very well. It's not that CVS doesn't handle binaries at all—it does, just not very well.

All the files we've been working with until now have been plain text files. CVS has some special tricks for text files. For example, when it's working between a Unix repository and a Windows or Macintosh working copy, it converts file line endings appropriately for each platform. For example, Unix convention is to use a linefeed (LF) only, whereas Windows expects a carriage return/linefeed (CRLF) sequence at the end of each line. Thus, the files in a working copy on a Windows machine have CRLF endings, but the files in a working copy of the same project on a Unix machine have LF endings (the repository itself is always stored in LF format).

Another trick is that CVS detects special strings, known as *RCS keyword strings*, in text files and replaces them with revision information and other useful things. For example, if your file contains this string

```
$Revision$
```

CVS will expand on each commit to include the revision number. For example, it might get expanded to:

```
$Revision: 1.3 $
```

CVS will keep that string up to date as the file is developed. (The various keyword strings are documented in Chapters 4 and 11.)

This string expansion is a very useful feature in text files, because it allows you to see the revision number or other information about a file while you're editing it. But what if the file is a JPEG image (with a .JPG extension)? Or a compiled executable program? In those kinds of files, CVS could do some serious damage if it blundered around expanding any keyword string that it encountered. In a binary, such strings can even appear by coincidence.

Therefore, when you add a binary file, you have to tell CVS to turn off both keyword expansion and line-ending conversion. To do so, use **-kb**:

```
yarkon$ cvs add -kb filename
yarkon$ cvs ci -m "added blah" filename
  (etc)
```

Also, in some cases (such as text files that are likely to contain spurious keyword strings), you might want to disable just the keyword expansion. That's done with **-ko**:

```
yarkon$ cvs add -ko filename
yarkon$ cvs ci -m "added blah" filename
  (etc)
```

(In fact, this chapter is one such document, because of the "$Revision$" example shown here.)

Note that you can't meaningfully run **cvs diff** on two revisions of a binary file. The **diff** command uses a text-based algorithm that can report only whether two binary files differ, but not how they differ. Future versions of CVS might provide a way to run **diff** on binary files.

```
new revision: delete; previous revision: 1.1
done
yarkon$
```

Notice that, in the second and third commands, we name newfile.c explicitly even though it doesn't exist in the working copy anymore. Of course, in the commit, you don't absolutely need to name the file, as long as you don't mind the commit encompassing any other modifications that might have taken place in the working copy.

Removing Directories

As we said before, CVS doesn't really keep directories under version control. Instead, as a kind of cheap substitute, it offers certain odd behaviors that in most cases do the "right thing." One of these odd behaviors is that empty directories can be treated specially. If you want to remove a directory from a project, you first remove all the files in it

```
yarkon$ cd dir
yarkon$ rm file1 file2 file3
yarkon$ cvs remove file1 file2 file3
  (output omitted)
yarkon$ cvs ci -m "removed all files" file1 file2 file3
  (output omitted)
```

and then run **update** in the directory above it with the **-P** flag:

```
yarkon$ cd ..
yarkon$ cvs update -P
  (output omitted)
```

The **-P** option tells **update** to "prune" any empty directories—that is, to remove them from the working copy. Once that's done, the directory is said to have been removed; all of its files are gone, and the directory itself is gone (from the working copy, at least, although there is actually still an empty directory in the repository).

An interesting counterpart to this behavior is that when you run a plain **update**, CVS does not automatically bring new directories from the repository into your working copy. There are a couple of different justifications for this, none really worth going into here. The short answer is that from time to time you should run **update** with the **-d** flag, telling it to bring down any new directories from the repository.

Renaming Files and Directories

Renaming a file is equivalent to creating it under the new name and removing it under the old. In Unix, the commands are:

```
yarkon$ cp oldname newname
yarkon$ rm oldname
```

Here's the equivalent in CVS:

```
yarkon$ mv oldname newname
yarkon$ cvs remove oldname
  (output omitted)
yarkon$ cvs add newname
  (output omitted)
yarkon$ cvs ci -m "renamed oldname to newname" oldname newname
  (output omitted)
yarkon$
```

For files, that's all there is to it. Renaming directories is not done very differently: Create the new directory, **cvs add** it, move all the files from the old directory to the new one, **cvs remove** them from the old directory, **cvs add** them in the new one, **cvs commit** so everything takes effect, and then do **cvs update -P** to make the now-empty directory disappear from the working copy. That is to say:

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

Note

The warning message after the third command is telling you that it can't copy olddir's CVS/ subdirectory into newdir because newdir already has a directory of that name. This is fine, because you want olddir to keep its CVS/ subdirectory.

Obviously, moving directories around can get a bit difficult. The best policy is to try to come up with a good layout when you initially import your project so you won't have to move directories around very often. Later, you'll learn about a more drastic method of moving directories that involves making the change directly in the repository. However, that method is best saved for emergencies; whenever possible, it's best to handle everything with CVS operations inside working copies.

Avoiding Option Fatigue

Most people tire pretty quickly of typing the same option flags with every command. If you know that you always want to pass the **-Q** global option or you always want to use **-c** with **diff**, why should you have to type it out each time?

There is help, fortunately. CVS looks for a .cvsrc file in your home directory. In that file, you can specify default options to apply to every invocation of CVS. Here's an example .cvsrc:

```
diff -c
update -P
cvs -q
```

If the leftmost word on a line matches a CVS command (in its unabbreviated form), the corresponding options are used for that command every time. For global options, you just use **cvs**. So, for example, every time this particular user runs **cvs diff**, the **-c** flag is automatically included.

Getting Snapshots (Dates and Tagging)

Let's return to the example of the program that's in a broken state when a bug report comes in. The developer suddenly needs access to the *entire* project as it was at the time of the last release, even though many files might have been changed since then, and each file's revision number differs from the others. It would be far too time-consuming to look over the log messages, figure out what each file's individual revision number was at the time of release, and then run **update** (specifying a revision number with **-r**) on each one of them. In medium- to large-sized projects (tens to hundreds of files), such a process would be too unwieldy to attempt.

CVS, therefore, provides a way to retrieve previous revisions of the project files en masse. In fact, it provides two ways: by date, which selects the revisions based on the time that they were committed, and by tag, which retrieves a previously marked "snapshot" of the project.

Which method you use depends on the situation. The date-based retrievals are done by passing **update** the **-D** flag, which is similar to **-r** but takes dates instead of revision numbers:

```
yarkon$ cvs -q update -D "2001-04-19"
U hello.c
U a-subdir/subsubdir/fish.c
U b-subdir/random.c
yarkon$
```

With the **-D** option, **update** retrieves the highest revision of each file as of the given date, and it will revert the files in the working copy to prior revisions if necessary.

When you give the date, you can—and often should—include the time. For example, the previous command ended up retrieving revision 1.1 of everything (only three files showed changes, because all of the others are still at revision 1.1 anyway). Here's the status of hello.c to prove it:

```
yarkon$ cvs -Q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:          1.1.1.1 Sat Apr 24 22:45:03 2001
   Repository revision:       1.1.1.1 /usr/local/cvs/myproj/hello.c,v
   Sticky Date:               99.04.19.05.00.00
yarkon$
```

However, a glance back at the log messages from earlier in this chapter shows that revision 1.2 of hello.c was definitely committed on April 19, 2001. So why did we now get revision 1.1 instead of 1.2?

The problem is that the date "2001-04-19" was interpreted as meaning "the midnight that begins 2001-04-19"—that is, the very first instant on that date. This is probably not what you want. The 1.2 commit took place later in the day. By qualifying the date more precisely, we can retrieve revision 1.2:

```
yarkon$ cvs -q update -D "2001-04-19 23:59:59"
U hello.c
U a-subdir/subsubdir/fish.c
U b-subdir/random.c
yarkon$ cvs status hello.c
===================================================================
File: hello.c                 Status: Locally Modified
   Working revision:  1.2     Sat Apr 24 22:45:22 2001
   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)
yarkon$
```

We're almost there. If you look closely at the date/time on the **Sticky Date** line, it seems to indicate 4:59:59 A.M., not 11:59 as the command requested (later we'll get to what the "sticky" means). As you might have guessed, the discrepancy is due to the difference between local time and Universal Coordinated Time (also known as Greenwich Mean Time). The repository always stores dates in Universal Time, but CVS on the client side usually assumes the local system time zone. In the case of **-D**, this is rather unfortunate because you're probably most interested in comparing against the repository time and don't care about the local system's idea of time. You can get around this by specifying the GMT zone in the command:

```
yarkon$ cvs -q update -D "2001-04-19 23:59:59 GMT"
U hello.c
yarkon$ cvs -q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.2     Sun Apr 25 22:38:53 2001
   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)
yarkon$
```

There—that brought the working copy back to the final commits from April 19 (unless there were any commits during the last second of the day, which there weren't).

What happens now if you run **update**?

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

Nothing happens at all. However, you know that there are more recent versions of at least three files. Why aren't these included in your working copy?

That's where the "sticky" comes in. Updating with the **-D** flag causes the working copy to be restricted permanently to that date or before. In CVS terminology, the working copy has a "sticky date" set. Once a working copy has acquired a sticky property, it stays sticky until told otherwise. Therefore, subsequent updates will not automatically retrieve the most recent revision. Instead, they'll stay restricted to the sticky date. Stickiness can be revealed by running **cvs status** or by directly examining the CVS/Entries file:

```
yarkon$ cvs -q update -D "2001-04-19 23:59:59 GMT"
U hello.c
yarkon$ cat CVS/Entries
D/a-subdir////
D/b-subdir////
D/c-subdir////
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 2001//D99.04.19.23.59.59
/hello.c/1.2/Sun Apr 25 23:07:29 2001//D99.04.19.23.59.59
yarkon$
```

If you were to modify hello.c and then try to run **commit**

```
yarkon$ cvs update
M hello.c
yarkon$ 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!
yarkon$
```

CVS would not permit the commit to happen because that would be like allowing you to go back and change the past. CVS is all about record keeping and, therefore, will not allow you to do that.

This does not mean CVS is unaware of all the revisions that have been committed since that date, however. You can still compare the sticky-dated working copy against other revisions, including future ones:

```
yarkon$ 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   2001/04/24 22:09:27     1.5
--- hello.c   2001/04/25 00:08:44
***************
*** 3,9 ****
  void
  main ()
  {
    printf ("Hello, world!\n");
-   printf ("how are you?\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");
  }
```

This diff reveals that, as of April 19, 2001, the **how are you?** line had not yet been added. It also shows the modification that we made to the working copy (adding the comment shown in the preceding code snippet).

You can remove a sticky date (or any sticky property) by updating with the **-A** flag (**-A** stands for "reset," don't ask us why), which brings the working copy back to the most recent revisions:

```
yarkon$ cvs -q update -A
U hello.c
yarkon$ cvs status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.5     Sun Apr 25 22:50:27 2001
   Repository revision:       1.5     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                (none)
   Sticky Date:               (none)
   Sticky Options:    (none)
yarkon$
```

Acceptable Date Formats

CVS accepts a wide range of syntax to specify dates. You'll never go wrong if you use ISO 8601 format (that is, the International Organization for Standardization (ISO) standard #8601; see also **www.saqqara.demon.co.uk/datefmt.htm**), which is the format used in the preceding examples. You can also use Internet email dates as described in RFC 822 and RFC 1123 (see **www.rfc-editor.org/rfc/**). Finally, you can use certain unambiguous English constructs to specify dates relative to the current date.

You will probably never need all of the formats available, but here are some more examples to give you an idea of what CVS accepts:

```
yarkon$ cvs update -D "19 Apr 2001"
yarkon$ cvs update -D "19 Apr 2001 20:05"
yarkon$ cvs update -D "19/04/2001"
yarkon$ cvs update -D "3 days ago"
yarkon$ cvs update -D "5 years ago"
yarkon$ cvs update -D "19 Apr 2001 23:59:59 GMT"
yarkon$ cvs update -D "19 Apr"
```

The double quotes around the dates are there to ensure that the Unix shell treats the date as one argument even if it contains spaces. The quotes will do no harm if the date doesn't contain spaces, so it's probably best to always use them.

Marking a Moment in Time (Tags)

Retrieving by date is useful when the mere passage of time is your main concern. However, more often what you really want to do is retrieve the project as it was at the time of a specific event—perhaps a public release, a known stable point in the software's development, or the addition or removal of some major feature.

Trying to remember the date when that event took place or deducing the date from log messages would be a tedious process. Presumably, the event, because it was important, was marked as such in the formal revision history. The method CVS offers for making such marks is known as *tagging*.

Tags differ from commits in that they don't record any particular textual change to files, but rather a change in the developers' attitude *about* the files. A tag gives a label to the collection of revisions represented by one developer's working copy (usually, that working copy is completely up to date so the tag name is attached to the "latest and greatest" revisions in the repository).

Setting a tag is as simple as this:

```
yarkon$ cvs -q tag Release-2001_05_01
T README.txt
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
yarkon$
```

That command associates the symbolic name "Release-2001_05_01" with the snapshot represented by this working copy. Or in other words, *snapshot* means a set of files and associated revision numbers from the project. Those revision numbers do not have to be the same from file to file and, in fact, usually aren't. For example, assuming that **tag** was done on the same myproj directory that we've been using throughout this chapter and that the working copy was completely up to date, the symbolic name "Release-2001_05_01" will be attached to hello.c at revision 1.5, to fish.c at revision 1.2, to random.c at revision 1.2, and to everything else at revision 1.1.

It might help to visualize a tag as a path or string linking various revisions of files in the project. In Figure 2.1, an imaginary string passes through the tagged revision number of each file in a project.

If you pull the string taut and sight directly along it, you'll see a particular moment in the project's history—namely, the moment that the tag was set (Figure 2.2).

As you continue to edit files and commit changes, the tag will not move along with the increasing revision numbers. It stays fixed, "stickily," at the revision number of each file at the time the tag was made.

Given their importance as descriptors, it's a bit unfortunate that log messages can't be included with tags or that the tags themselves can't be full paragraphs of prose. In the preceding example, the tag is fairly obviously stating that the project was in a releasable state as of a certain date. However, sometimes you might want to make snapshots of a more complex state, which can result in ungainly tag names such as:

```
yarkon$ cvs tag testing-release-3_pre-20010525-public-release
```

As a general rule, you should try to keep tags as terse as possible while still including all necessary information about the event that you're trying to record. When in doubt, err on

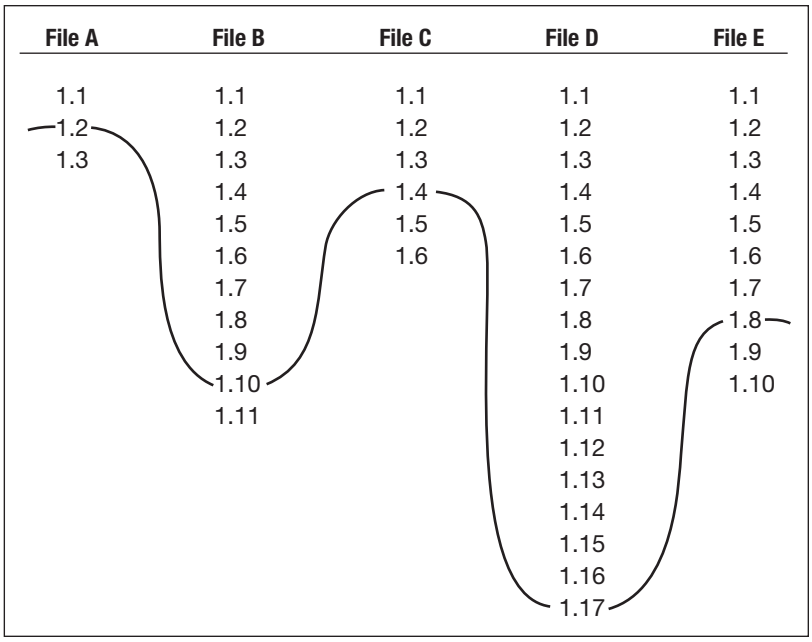

Figure 2.1
How a tag might stand in relation to a project's revision history.

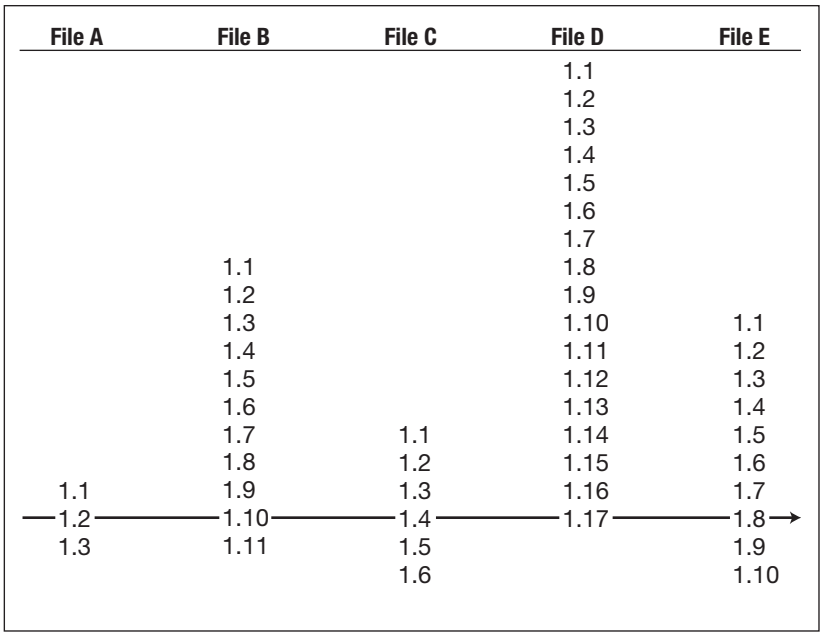

Figure 2.2
The tag is a "straight sight" through a revision history.

the side of being overly descriptive—you'll be glad later when you're able to tell from some verbose tag name exactly what circumstance was recorded.

You've probably noticed that no periods or spaces were used in the tag names. CVS is rather strict about what constitutes a valid tag name. The rules are that it must start with a letter and contain letters, digits, hyphens ("-"), and underscores ("_"). No spaces, periods, colons, commas, or any other symbols may be used.

To retrieve a snapshot by tag name, the tag name is used just like a revision number. There are two ways to retrieve snapshots: You can check out a new working copy with a certain tag, or you can switch an existing working copy over to a tag. Both result in a working copy whose files are at the revisions specified by the tag.

Most of the time, what you are really trying to do is take a look at the project as it was at the time of the snapshot. You might not necessarily want to do this in your main working copy, where you presumably have uncommitted changes and other useful states built up, so let's assume you just want to check out a separate working copy with the tag. Here's how (but make sure to invoke this somewhere other than in your existing working copy or its parent directory!):

```
yarkon$ cvs checkout -r Release-2001_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
```

We've seen the **-r** option before in the **update** command, where it preceded a revision number. In many ways, a tag is just like a revision number because, for any file, a given tag corresponds to exactly one revision number (it's illegal, and generally impossible, to have two tags of the same name in the same project). In fact, anywhere you can use a revision number as part of a CVS command, you can use a tag name instead (as long as the tag has been set previously). If you want to compare a file's current state against its state at the time of the last release, you can use the **diff** command to do this:

```
yarkon$ cvs diff -c -r Release-2001_05_01 hello.c
```

And if you want to revert it temporarily to that revision, you can do this:

```
yarkon$ cvs update -r Release-2001_05_01 hello.c
```

The interchangeability of tags and revision numbers explains some of the strict rules about valid tag names. Imagine if periods were legal in tag names; you could have a tag named "1.3" attached to an actual revision number of "1.47." If you then issued the command

```
yarkon$ cvs update -r 1.3 hello.c
```

how would CVS know whether you were referring to the tag named "1.3," or the much earlier revision 1.3 of hello.c? Thus, restrictions are placed on tag names so that they can always be distinguished easily from revision numbers. A revision number has a period; a tag name doesn't. (There are reasons for the other restrictions, too, mostly having to do with making tag names easy for CVS to parse.)

As you've probably guessed by this point, the second method of retrieving a snapshot—that is, switching an existing working directory over to the tagged revisions—is also done by updating:

```
yarkon$ cvs update -r Release-2001_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
yarkon$
```

The preceding command is just like the one we used to revert hello.c to Release-2001_05_01, except that the file name is omitted because we want to revert the entire project over. (You can, if you want, revert just one subtree of the project to the tag by invoking the preceding command in that subtree instead of starting from the top level, although you hardly ever would want to do that.)

Note that no files appear to have changed when we updated. The working copy was completely up to date when we tagged, and no changes had been committed since the tagging.

However, this does not mean that nothing changed at all. The working copy now knows that it's at a tagged revision. When you make a change and try to commit it (let's assume we modified hello.c):

```
yarkon$ cvs -q update
M hello.c
yarkon$ cvs -q ci -m "trying to commit from a working copy on a tag"
cvs commit: sticky tag 'Release-2001_05_01' for file 'hello.c' is not a branch
cvs [commit aborted]: correct above errors first!
yarkon$
```

CVS does not permit the commit to happen. (Don't worry about the exact meaning of that error message yet—we'll cover branches next in this chapter.) It doesn't matter whether the

working copy got to be on a tag via a **checkout** or an **update** command. Once it is on a tag, CVS views the working copy as a static snapshot of a moment in history, and CVS won't let you change history—at least not easily. If you run **cvs status** or look at the CVS/Entries files, you'll see that there is a sticky tag set on each file. Here's the top-level Entries file, for example:

```
yarkon$ cat CVS/Entries
D/a-subdir////
D/b-subdir////
D/c-subdir////
/README.txt/1.1.1.1/Sun Apr 18 18:18:22 2001//TRelease-2001_05_01
/hello.c/1.5/Tue Apr 20 07:24:10 2001//TRelease-2001_05_01
yarkon$
```

Tags, like other sticky properties, are removed with the **-A** flag to **update**:

```
yarkon$ cvs -q update -A
M hello.c
yarkon$
```

The modification to hello.c did not go away, however; CVS is still aware that the file changed with respect to the repository:

```
yarkon$ 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   2001/04/20 06:12:56     1.5
--- hello.c   2001/05/04 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 */
  }
yarkon$
```

Now that you've reset with **update**, CVS will accept a commit:

```
yarkon$ 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
yarkon$
```

The tag "Release-2001_05_01" is still attached to revision 1.5, of course. Compare the file's status before and after a reversion to the tag:

```
yarkon$ cvs -q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.6     Tue May  4 20:09:17 2001
   Repository revision:       1.6     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                (none)
   Sticky Date:               (none)
   Sticky Options:            (none)
yarkon$ cvs -q update -r Release-2001_05_01
U hello.c
yarkon$ cvs -q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.5     Tue May  4 20:21:12 2001
   Repository revision:       1.5     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                Release-2001_05_01 (revision: 1.5)
   Sticky Date:               (none)
   Sticky Options:            (none)
yarkon$
```

Now, having just told you that CVS doesn't let you change history, we'll show you how to change history.

Branches

We've been viewing CVS as a kind of intelligent, coordinating library. So far, we've seen only how you can examine the past with CVS, without affecting anything. However, CVS also allows you to go back in time to change the past. What do you get then? A CVS *branch* splits a project's development into separate, parallel branches. Changes made on one branch do not affect the other.

Why is this useful?

Let's return for a moment to the scenario of the developer who, in the midst of working on a new version of the program, receives a bug report about an older released version. Assuming the developer fixes the problem, she still needs a way to deliver the fix to the customer. It won't help to just find an old copy of the program somewhere, patch it up without CVS's knowledge, and ship it off. There would be no record of what was done; CVS would be unaware of the fix, and if something was later discovered to be wrong with the patch, no one would have a starting point for reproducing the problem.

It's even more wrong to fix the bug in the current, unstable version of the sources and ship that to the customer. Sure, the reported bug might be solved, but the rest of the code is in a half-implemented, untested state. It might run, but it's certainly not ready for prime time.

Because the last released version is thought to be stable, aside from this one bug, the ideal solution is to go back and correct the bug in the old release—that is, to create an alternate universe in which the last public release includes this bug fix.

That's where branches come in. The developer splits off a branch, rooted in the main line of development (the *trunk*) not at its most recent revisions, but back at the point of the last release. Then he or she checks out a working copy of this branch, makes whatever changes are necessary to fix the bug, and commits them on that branch, so there's a record of the bug fix. Now he or she can package up an interim release based on the branch and ship it to the customer.

The change won't affect the code on the trunk, nor would the developer want it to without first finding out whether the trunk needs the same bug fix or not. If it does, the developer can use the **merge** command to merge the branch changes into the trunk. In a merge, CVS calculates the changes made on the branch between the point where it diverged from the trunk and the branch's *tip* (its most recent state), then applies those differences to the project at the tip of the trunk. The difference between the branch's root and its tip works out, of course, to be precisely the bug fix.

Another good way to think of a merge is as a special case of updating. The difference is that in a merge, the changes to be incorporated are derived by comparing the branch's root and tip, instead of by comparing the working copy against the repository.

The act of updating is itself similar to receiving patches directly from the authors and applying them by hand. In fact, to run the **update** command, CVS calculates the difference (that's "difference" as in the **diff** program) between the working copy and the repository and then applies that diff to the working copy just as the **patch** program would. This mirrors the way in which a developer takes changes from the outside world, by manually applying patch files sent in by contributors.

Thus, merging the bug fix branch into the trunk is just like accepting some outside contributor's patch to fix the bug. The contributor would have made the patch against the last released version, just as the branch's changes are against that version. If that area of code in the current sources hasn't changed much since the last release, the merge will succeed with no problems. If the code is now substantially different, however, the merge will fail with conflict (that is, the patch will be rejected), and some manual fiddling will be necessary. Usually, this is accomplished by reading the conflicting area, making the necessary changes by hand, and committing. Figure 2.3 shows a picture of what happens in a branch and merge.

We'll now walk through the steps necessary to make this picture happen. Remember that it's not really time that's flowing from left to right in the diagram, but the revision history. The branch will not have been made at the time of the release, but is created later, rooted back at the release's revisions.

In our case, let's assume the files in the project have gone through many revisions since they were tagged as "Release-2001_05_01," and perhaps files have been added as well. When the bug report regarding the old release comes in, the first thing we'll want to do is create a branch rooted at the old release, which we conveniently tagged "Release-2001_05_01." One way to do this is to first check out a working copy based on that tag, then create the branch by retagging with the **-b** (branch) option:

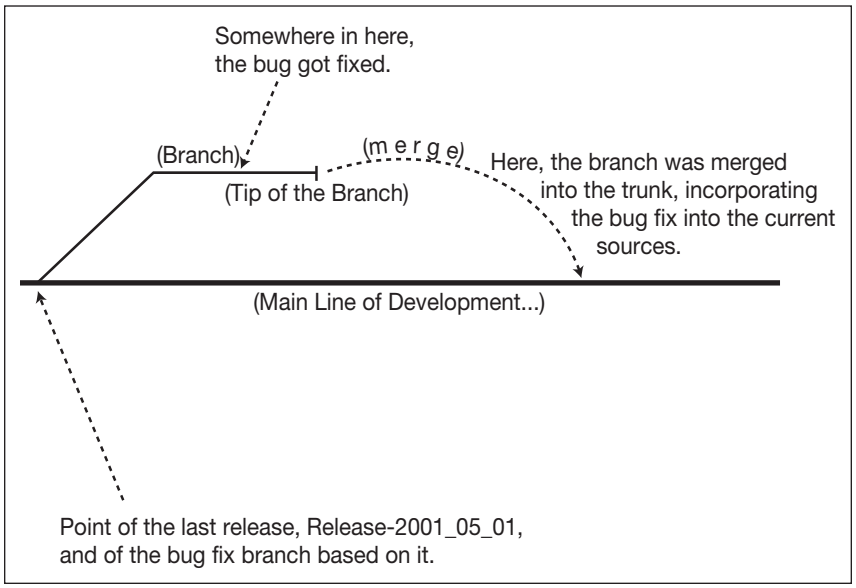

Figure 2.3
Branching and merging.

```
yarkon$ cd ..
yarkon$ ls
myproj/
yarkon$ cvs -q checkout -d myproj_old_release -r Release-2001_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
yarkon$ ls
myproj/      myproj_old_release/
yarkon$ cd myproj_old_release
yarkon$ ls
CVS/      README.txt  a-subdir/   b-subdir/   hello.c
yarkon$ cvs -q tag -b Release-2001_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
yarkon$
```

Take a good look at that last command. It might seem somewhat arbitrary that **tag** is used to create branches, but there's actually a reason for it: The tag name will serve as a label by which the branch can be retrieved later. Branch tags do not look any different from non-branch tags, and they are subject to the same naming restrictions. Some people like to always include the word *branch* in the tag name itself (for example, "Release-2001_05_01-bugfix-branch") so they can distinguish branch tags from other kinds of tags. You might want to do this if you often find yourself retrieving the wrong tag.

(And while we're at it, note the **-d myproj_old_release** option to **checkout** in the first CVS command. This tells **checkout** to put the working copy in a directory called myproj_old_release, so we won't confuse it with the current version in myproj. Be careful not to confuse this use of **-d** with the global option of the same name, or with the **-d** option to **update**.)

Of course, merely running the **tag** command does not switch this working copy over to the branch. Tagging never affects the working copy; it just records some extra information in the repository to allow you to retrieve that working copy's revisions later on (as a static piece of history or as a branch, as the case may be).

Retrieval can be done one of two ways (you're probably getting used to this motif by now). You can check out a new working copy on the branch

```
yarkon$ pwd
/home/whatever
yarkon$ cvs co -d myproj_branch -r Release-2001_05_01-bugfixes myproj
```

or switch an existing working copy over to it:

```
yarkon$ pwd
/home/whatever/myproj
yarkon$ cvs update -r Release-2001_05_01-bugfixes
```

The end result is the same (well, the name of the new working copy's top-level directory might be different, but that's not important for CVS's purposes). If your current working copy has uncommitted changes, you'll probably want to use **checkout** instead of **update** to access the branch. Otherwise, CVS attempts to merge your changes into the working copy as it switches it over to the branch. In that case, you might get conflicts, and even if you didn't, you'd still have an impure branch. It won't truly reflect the state of the program as of the designated tag, because some files in the working copy will contain modifications made by you.

Anyway, let's assume that by one method or another you get a working copy on the desired branch:

```
yarkon$ cvs -q status hello.c
===================================================================
File: hello.c                 Status: Up-to-date
   Working revision:  1.5     Tue Apr 20 06:12:56 2001
   Repository revision:       1.5     /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                Release-2001_05_01-bugfixes
(branch: 1.5.2)
   Sticky Date:               (none)
   Sticky Options:            (none)
yarkon$ cvs -q status b-subdir/random.c
===================================================================
File: random.c                Status: Up-to-date
   Working revision:  1.2     Mon Apr 19 06:35:27 2001
   Repository revision:       1.2 /usr/local/cvs/myproj/b-subdir/random.c,v
   Sticky Tag:                Release-2001_05_01-bugfixes (branch: 1.2.2)
   Sticky Date:               (none)
   Sticky Options:            (none)
yarkon$
```

(The contents of those **Sticky Tag** lines will be explained shortly.) If you modify hello.c and random.c, and commit

```
yarkon$ cvs -q update
M hello.c
M b-subdir/random.c
yarkon$ 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  <--  hello.c
new revision: 1.5.2.1; previous revision: 1.5
done
Checking in b-subdir/random.c;
/usr/local/cvs/myproj/b-subdir/random.c,v  <--  random.c
new revision: 1.2.2.1; previous revision: 1.2
done
yarkon$
```

you'll notice that there's something funny going on with the revision numbers:

```
yarkon$ 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 2001
   Repository revision:       1.5.2.1 /usr/local/cvs/myproj/hello.c,v
   Sticky Tag:                Release-2001_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 2001
   Repository revision:       1.2.2.1 /usr/local/cvs/myproj/b-subdir/random.c,v
   Sticky Tag:                Release-2001_05_01-bugfixes (branch: 1.2.2)
   Sticky Date:               (none)
   Sticky Options:            (none)
yarkon$
```

They now have four digits instead of two!

A closer look reveals that each file's revision number is just the branch number (as shown on the **Sticky Tag** line) plus an extra digit on the end.

What you're seeing is a little bit of CVS's innards. Although you almost always use a branch to mark a project-wide divergence, CVS actually records the branch on a per-file basis. This project had five files in it at the point of the branch, so five individual branches were made, all with the same tag name: "Release-2001_05_01-bugfixes."

Note

Most people consider this per-file scheme a rather inelegant implementation on CVS's part. It's a bit of the old RCS legacy showing through—RCS didn't know how to group files into projects, and even though CVS does, it still uses code inherited from RCS to handle branches.

Ordinarily, you don't need to be too concerned with how CVS is keeping track of things internally, but in this case, it helps to understand the relationship between branch numbers and revision numbers. Let's look at the hello.c file; everything we're about to say about hello.c applies to the other files in the branch (with revision/branch numbers adjusted accordingly).

The hello.c file was on revision 1.5 at the point where the branch was rooted. When we created the branch, a new number was tacked onto the end to make a *branch number* (CVS chooses the first unused even, nonzero integer). Thus, the branch number in this case became "1.5.2." The branch number by itself is not a revision number, but it is the *root* (that is, the prefix) of all the revision numbers for hello.c along this branch.

However, when we ran that first **cvs status** command in a branched working copy, hello.c's revision number showed up as only "1.5," not "1.5.2.0" or something similar. This is because the initial revision on a branch is always the same as the trunk revision of the file, where the branch sprouts off. Therefore, CVS shows the trunk revision number in status output, for as long as the file is the same on both branch and trunk.

Once we had committed a new revision, hello.c was no longer the same on both trunk and branch—the branch version of the file had changed, while the trunk remained the same. Accordingly, hello.c was assigned its first branch revision number. We saw this in the status output after running **commit**, where its revision number is clearly "1.5.2.1."

The same story applies to the random.c file. Its revision number at the time of branching was "1.2," so its first branch is "1.2.2," and the first new **commit** command of random.c on that branch received the revision number "1.2.2.1."

There is no numeric relationship between 1.5.2.1 and 1.2.2.1—no reason to think that they are part of the same branch event, except that both files are tagged with "Release-2001_05_01-bugfixes," and the tag is attached to branch numbers 1.5.2 and 1.2.2 in the respective files. Therefore, the tag name is your only handle on the branch as a project-wide entity. Although it is perfectly possible to move a file to a branch by using the revision number directly

```
yarkon$ cvs update -r 1.5.2.1 hello.c
U hello.c
yarkon$
```

it is almost always a bad idea. You would be mixing the branch revision of one file with nonbranch revisions of the others. Who knows what losses might result? It is better to use the branch tag to refer to the branch and do all files at once by not specifying any particular file. That way you don't have to know or care what the actual branch revision number is for any particular file.

It is also possible to have branches that sprout off other branches, to any level of absurdity. A file with a revision number of 1.5.4.37.2.3.12.1 is depicted graphically by Figure 2.4.

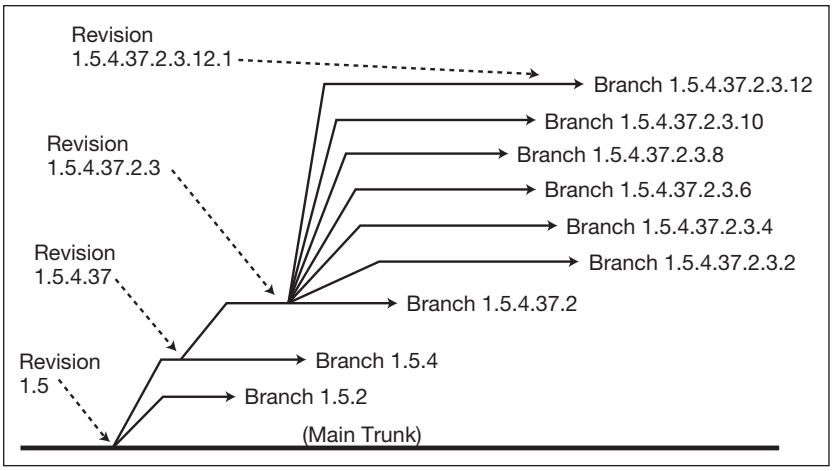

Figure 2.4
A ridiculously high degree of branching.

Admittedly, it's hard to imagine what circumstances would make such a branching depth necessary, but isn't it nice to know that CVS will go as far as you're willing to take it? Nested branches are created the same way as any other branch: Check out a working copy on branch N, run **cvs tag -b *branchname*** in it, and you'll create branch *N*.M in the repository (where "N" represents the appropriate branch revision number in each file, such as "1.5.2.1, and "M" represents the next available branch at the end of that number, such as "2").

Merging Changes from Branch to Trunk

Now that the bug fix has been committed on the branch, let's switch the working copy over to the highest trunk revisions and see if the bug fix needs to be done there, too. We'll move the working copy off the branch by using **update -A** (branch tags are like other sticky properties in this respect) and then using **diff** against the branch we just left:

```
yarkon$ cvs -q update -A
U hello.c
U b-subdir/random.c
yarkon$ cvs -q diff -c -r Release-2001_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   2001/05/05 00:15:07     1.5.2.1
--- hello.c   2001/05/04 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 2001/05/05 00:15:07     1.2.2.1
--- b-subdir/random.c 2001/04/19 06:35:27     1.2
***************
*** 4,8 ****
  void main ()
  {
!   printf ("A random number.\n");
  }
--- 4,8 --
  void main ()
  {
!   printf ("a random number\n");
  }
yarkon$
```

Using the **diff** command shows that **good-bye** is spelled with a hyphen in the branch revision of hello.c, and that the trunk revision of that file has a comment near the end that the branch revision doesn't have. Meanwhile, in random.c, the branch revision has a capital "A" and a period, whereas the trunk doesn't.

To actually merge the branch changes into the current working copy, run **update** with the **-j** flag (the same **j** for "join" that we used to revert a file to an old revision before):

```
yarkon$ cvs -q update -j Release-2001_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
yarkon$ cvs -q update
M hello.c
M b-subdir/random.c
yarkon$ cvs -q ci -m "merged from branch Release-2001_05_01-bugfixes"
Checking in hello.c;
/usr/local/cvs/myproj/hello.c,v  <--  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  <--  random.c
new revision: 1.3; previous revision: 1.2
done
yarkon$
```

This takes the changes from the branch's root to its tip and merges them into the current working copy (which subsequently shows those modifications just as though the files had been hand-edited into that state). The changes are then committed onto the trunk, because nothing in the repository changed when a working copy underwent a merge.

Although no conflicts were encountered in this example, it's quite possible (even probable) that there would be some in a normal merge. If that happens, they need to be resolved like any other conflict and then committed.

Multiple Merges

Sometimes a branch will continue to be actively developed even after the trunk has undergone a merge from it. For example, this can happen if a second bug in the previous public release is discovered and has to be fixed on the branch. Maybe someone didn't get the joke in random.c, so on the branch you have to add a line explaining it

```
yarkon$ pwd
/home/whatever/myproj_branch
yarkon$ 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");
```

```
}
yarkon$
```

and then run **commit**. If that bug fix also needs to be merged into the trunk, you might be tempted to try the same **update** command as before in the trunk working copy to "re-merge":

```
yarkon$ cvs -q update -j Release-2001_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
yarkon$
```

As you can see, that didn't have quite the desired effect—we got a conflict, even though the trunk copy hadn't been modified there and, therefore, no conflict was expected.

The trouble was that the **update** command behaved exactly as described: It tried to take all the changes between the branch's root and tip and merge them into the current working copy. The only problem is that some of those changes had already been merged into this working copy. That's why we got the conflict:

```
yarkon$ pwd
/home/whatever/myproj
yarkon$ 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
}
yarkon$
```

You could go through resolving all such conflicts by hand—it's usually not hard to tell what you need to do in each file. Nevertheless, it is even better to avoid a conflict in the first place. By passing two **-j** flags instead of one, you'll get only those changes from where you last merged to the tip instead of all of the changes on the branch, from root to tip. The first

-j gives the starting point on the branch, and the second is just the plain branch name (which implies the tip of the branch).

The question then is, how can you specify the point on the branch from which you last merged? One way is to qualify by using a date along with the branch tag name. CVS provides a special syntax for this:

```
yarkon$ cvs -q update -j "Release-2001_05_01-bugfixes:2 days ago" \
                      -j Release-2001_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
yarkon$
```

If the branch tag name is followed by a colon and then a date (in any of the usual CVS date syntaxes), CVS will include only changes later than that date. So if you know that the original bug fix was committed on the branch three days ago, the preceding command merges the second bug fix only.

A better way, if you plan ahead, is to tag the branch after each bug fix (just a regular tag—we're not starting a new branch here or anything like that). Suppose after fixing the bug in the branch and committing, you do this in the branch's working copy:

```
yarkon$ cvs -q tag Release-2001_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
yarkon$
```

Then, when it's time to merge the second change into the trunk, you can use that conveniently placed tag to delimit the earlier revision:

```
yarkon$ cvs -q update -j Release-2001_05_01-bugfixes-fix-number-1 \
-j Release-2001_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
yarkon$
```

This way, of course, is much better than trying to recall how long ago you made one change versus another, but it works only if you remember to tag the branch every time it is merged to the trunk. The lesson, therefore, is to tag early and tag often! It's better to err on the side of too many tags (as long as they all have descriptive names) than to have too few. In these

last examples, for instance, there was no requirement that the new tag on the branch have a name similar to the branch tag itself. Although we named it "Release-2001_05_01-bugfixes-fix-number-1," it could just as easily have been "fix1." However, the former is preferable, because it contains the name of the branch and thus won't ever be confused with a tag on some other branch. (Remember that tag names are unique within files, not within branches. You can't have two tags named "fix1" in the same file, even if they refer to revisions on different branches.)

Creating a Tag or Branch without a Working Copy

As stated earlier, tagging affects the repository, not the working copy. That begs the question: Why require a working copy at all when tagging? The only purpose that it serves is to designate which project and which revisions of the various files in the project are being tagged. If you could specify the project and revisions independently of the working copy, no working copy would be necessary.

There is way to do this: the **rtag** command (for "repository tag"). It's very similar to **tag**; a couple of examples will explain its usage. Let's go back to the moment when the first bug report came in and we needed to create a branch rooted at the last public release. We checked out a working copy at the **release** tag and then ran **tag -b** on it:

```
yarkon$ cvs tag -b Release-2001_05_01-bugfixes
```

This created a branch rooted at "Release-2001_05_01." However, because we know the **release** tag, we could have used it in an **rtag** command to specify where to root the branch, not even bothering with a working copy:

```
yarkon$ cvs rtag -b -r Release-2001_05_01 Release-2001_05_01-bugfixes myproj
```

That's all there is to it. That command can be issued from anywhere, inside or outside a working copy. However, your CVSROOT environment variable would have to point to the repository, of course, or you can specify it with the global **-d** option. It works for non-branch tagging, too, but it's less useful that way because you have to specify each file's revision number, one by one. (Or you can refer to it by tag, but then you'd obviously already have a tag there, so why would you want to set a second one on the exact same revisions?)

You now know enough to navigate inside CVS and enough to start working with other people on a project. There are still a few minor features that haven't been introduced, as well as some unmentioned but useful options to features we've already seen. These will all be presented as appropriate in chapters to come, in scenarios that demonstrate both how and why to use them. When in doubt, don't hesitate to consult the Cederqvist manual; it is an indispensable resource for serious CVS users.