by Jeffreys Copeland and Haemer

Practical CVS, Part 1

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS' R&D group in Boulder, CO. He's been a software consultant to the Hugo award administrators for several years. He spends his spare time raising children and cats.*

Jeffrey S. Haemer (`jsh@canary.com`) *now works for QMS, too, and is having a great time. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

Last month, we sketched an overview of Revision Control System, or RCS, for you. In the process, we stumbled over, and pointed out, RCS' single-file myopia. In closing, we promised you a discussion of Concurrent Versions System, or CVS, a widely used, freely available extension to RCS, built to handle the file hierarchies that we all use to build products. Here it comes.

Getting Started

The nice thing about RCS is that it's easy to use. For the most part, all you need are two commands, `ci` and `co`, which are used to check files in and check files out, respectively. As you'd expect, dealing with trees requires, unavoidably, more work. Still, CVS tries to mirror RCS' simplicity and doesn't do so bad. We'll illustrate this by beginning the same way we did last month:

```
$ echo "Use the right tool for the job." > jeff
$ cvs ci jeff
cvs commit: No CVSROOT specified!
   Please use the '-d' option
cvs [commit aborted]:
   or set the CVSROOT environment variable.
```

We want to draw your attention to two noteworthy things about our example: First, the command we used, `cvs ci`, is similar to the command we would have used for RCS (there is also a `cvs co`). While these and other features of CVS use easy-to-remember analogs to RCS, `ci` is an *argument* to the command `cvs`, not a command on its own.

(We could tell you that for RCS you need to learn two commands, but for CVS, you only need to learn one. However, if we could say something like this with a straight face, we'd be working in marketing, making a lot more money.)

Second, the above example didn't work. We did something that seemed like it made sense, but it didn't. Normally, we'd shrug and say, "First time for everything," try the exact same thing a couple more times–we're not in marketing, we're in software–and then, when all else failed, read the manual.

In this case, CVS told us what to do. This is an important principle of software design: *Don't just say what's wrong, say how to fix it!* Contrast the SunOS usage message

```
sed: Unknown flag: X
```

with the Linux usage message

```
Usage: sed [-nV] [--quiet] [--silent]
   [--version] [-e script]
   [-f script-file] [--expression=script]
   [--file=script-file] [file...]
```

This principle has a long history in UNIX and has been clearly set out on many occasions. Although we try to use standards-conforming library functions, we steer clear of the POSIX `getopt()` interface because it doesn't force a usage message.

In contrast, our Perl programs routinely have lines like this:

```
getopt('LSMFT') or die $usage;
```

The module `Getopt::Std` doesn't automatically emit "Hey! Don't do that" messages, so we feel better about using it.

So let's try taking the advice CVS offers.

```
$ CVSROOT=/cvs; export CVSROOT
$ cvs ci jeff
cvs commit: cannot open CVS/Entries for reading:
   No such file or directory
cvs commit: nothing known about 'jeff'
cvs [commit aborted]:
   correct above errors first!
```

Progress. Now we're making new mistakes.

We enjoy learning by making mistakes for several reasons. First, we make a lot of mistakes, so it's important to know from the outset what software will let us shoot ourselves in the foot before we do it. By this assay, CVS turns out to be relatively safe. Second, we like to see and decode as many error messages as possible. We know from experience that we'll see them again; if we don't generate them ourselves by accident, someone else will invariably appear at our door demanding to know what they mean. (We're tempted to call all this "Learning by not doing." However, if we yielded easily to temptation, we'd be in politics, making *a lot* more money.)

So what's really going on here? CVS is designed to let you work with collections of files. To do so, you need to keep those collections in a repository. The environment variable `$CVSROOT` points at the root of this repository. You can have more than one repository, but each repository can hold many unrelated collections.

The first time we ran `cvs`, we hadn't designated a repository. Now, we've designated a repository, but the commands `cvs ci` and `cvs co` only work on source code collections that have already been put into the repository. To put a collection of files into the repository, you need to use the command `cvs import`.

We could try importing the file `jeff`, but that wouldn't show off CVS' ability to deal with file *collections*, so let's put in something bigger. First, we'll grab a copy of the entire `/etc` directory:

```
$ cp -r /etc/ .
```

Next, we'll import it:

```
$ cvs import etc
cvs [import aborted]: /cvs/CVSROOT:
   No such file or directory
```

Again, a new error message. This one's telling us that CVS uses a suite of administrative files, all of which it expects to find in the directory `$CVSROOT/CVSROOT`. (We think the choice of names is genuinely horrible. We didn't write CVS; we just use it. If we could think up good names for things, we'd be in advertising and making lots more money.)

We could show you how to create these administrative files by hand, but the command `cvs init` does the job for you (see Figure 1). We'll postpone explaining what each of the files shown in Figure 1 are, but note for now that almost all of them are under RCS control. In this case, "under RCS control" also means "under CVS control." With a nice self-referential twist, CVS allows you to work with its administrative files as a legitimate CVS collection.

What do we mean by "almost all"? `CVSROOT/history`, for example, has no associated RCS file because it's a file containing the entire history of everything done to any repository under `$CVSROOT`. It starts out empty.

```
-rw-rw-r-- 1 jsh rd 0 Jun 2 12:17 cvs/CVSROOT/history
```

We'll look at this file again after we import `etc`. Speaking of which, let's try again:

```
$ cvs import etc
Usage: cvs import [-d] [-k subst]
  [-I ign] [-m msg] [-b branch] [-W spec]
  repository vendor-tag release-tags...
   -d Use the file's modification time
      as the time of import.
   -k sub  Set default RCS keyword
      substitution mode.
   -I ign  More files to ignore (! to reset).
   -b bra  Vendor branch id.
   -m msg  Log message.
   -W spec Wrappers specification line.
```

Figure 1. Creating Administrative Files with cvs init

```
$ cvs init
$ ls -RFC $CVSROOT
CVSROOT/

/cvs/CVSROOT:
checkoutlist     cvswrappers,v    loginfo,v    rcsinfo
checkoutlist,v   editinfo         modules      rcsinfo,v
commitinfo       editinfo,v       modules,v    taginfo
commitinfo,v     history          notify       taginfo,v
cvswrappers      oginfo           notify,v     verifymsg
verifymsg,v
```

We're getting close now. The usage message from `cvs import` tells us that we're just calling it with the wrong arguments.

What are the arguments? `repository` is where to put it under `$CVSROOT`. But where to put *what*? CVS is built to understand trees. By default, operations are performed on whatever directory you're in and all its subdirectories. If we say `cvs import`, for example, we'll actually import everything under our current working directory. This means we have to make sure we do a

```
$ cd etc
```

The arguments `vendor-tag` and `release-tag` are there to let us keep track of large releases of software from other people. Most of the time, we're focused on keeping track of large collections of software that we're developing ourselves and want to release. Sometimes, however, we begin with a code base from somewhere else–a vendor that we've paid to develop something, or even a different branch of our own company. `vendor-tag` lets us identify the source.

What about `release-tag`? In some situations, we need to be able to handle massive updates from the original vendor. If we begin with a source release from Acme Software, for example, work on a customized development of the package for six months, and then get an upgrade from Acme, we want a place to store the upgrade, exactly as supplied by the vendor, before we begin merging the changes into our customized version. With these tags, we can do that.

In our case, we'll call the "vendor" Jeff and the release "initial," like this:

```
cvs import etc Jeff initial
```

Aha! Suddenly, we're editing a file containing the following lines:

```
CVS: ----------------------------------------------------------------------
CVS: Enter Log. Lines beginning with 'CVS:' are removed automatically.
CVS:
CVS: ----------------------------------------------------------------------
```

This is what CVS does to ask you for a log message. It's the same idea as the RCS prompt

```
enter log message, terminated with single '.' or end of file:
>>
```

with three twists:

1. CVS automatically puts you into whatever editor is specified by the variables `$CVSEDITOR` or `$EDITOR`.
2. By default, CVS uses one comment for all imported files. This is a theme that we'll see again in `cvs ci`, and flows from the idea that CVS operations deal with collections, not just individual files.
3. CVS gives you starting text for the comment. Any line that starts with `CVS:` never goes into the RCS file, so none of the lines labeled `CVS:` get saved in any file. However, anything you type, above or below it, will become a log comment for every file you're importing.

The lines in the above example are hard-wired into CVS, but they're just the defaults. In general, the default comment files that the editor brings up come from one of the administrative files. Here's how it works:

The file `$CVSROOT/CVSROOT/rcsinfo`, for example, contains a two-column list of regular expressions and template files. If the name of the directory your file is in matches the regular expression, CVS uses the corresponding template to initialize your comment–a catch-all expression, `DEFAULT`, matches any directory that isn't otherwise matched. If we want to provide forms for fill-in-the-blank-style comments, we can put those forms into template files.

CVS is built to understand trees. By default, operations are performed on whatever directory you're in and all its subdirectories.

After the comments go in, and we exit the editor, CVS fills the screen with a series of lines like this:

```
N etc/passwd
I etc/passwd~
N etc/rmt
I etc/rmt.old
```

The files marked `N` are new files in the repository and the files marked `I` are being ignored. CVS has customizable rules about which files it ignores, but the defaults are so reasonable that we have never had to modify them. After all this is done, we have a repository with an RCS file that corresponds to each file we imported:

```
$ ls $CVSROOT
CVSROOT
etc
$ ls $CVSROOT/etc/pass*
/cvs/etc/passwd,v
```

Working with Files in the Repository

The Source Code Motel: Your files check in, but they never check out. – Anonymous

Seems like it took forever, doesn't it? Well, we *could* have tried reading the manual first, but that wouldn't have been as much fun. Let's review what we did to get started:

1. We created a place to store the repositories, then set `$CVSROOT` to point at it.
2. We initialized the `$CVSROOT` directory using `cvs admin`.
3. We went to the tree of sources we wanted to check in.
4. We said the magic words: `cvs admin etc Jeff initial`.
5. We put in a comment and exited the editor.

Is working with them as much of a hassle as getting them in, or is there a way to get something back out?

Try this:

```
$ cd ..
$ rm -rf etc   # That's 1705 files -- GONE!!!
       # A power tool is not user-friendly
$ cvs co etc   # Whew.  They're back.
cvs checkout: Updating etc
U etc/aliases
U etc/aliases.db
...
cvs checkout: Updating etc/X11
...
cvs checkout: Updating etc/rc.d
U etc/rc.0
...
cvs checkout: Updating etc/rc.d/init.d
U etc/rc/init.d/httpd
```

One command lets us check out the entire repository as many times as we want:

```
$ cd /tmp
$ cvs co etc
cvs checkout: Updating etc
U etc/X11
...
```

Is it really all there? Sure.

```
$ ls -F etc
CVS/
X11/
aliases
aliases.db
...
```

Wait. What is that directory `CVS/`? A little investigation reveals that every directory in a checked-out hierarchy has one, and that they all contain the same files:

```
$ ls CVS
Entries
Repository
Root
$ ls X11/CVS
Entries
Repository
Root
$ ls rc/init.d/CVS
Entries
Repository
Root
```

These files are not in the repository itself, but are administrative files used by CVS to keep track of where the files came from and what versions have been checked out. Here's an example, using the control files in `$CVSROOT/CVSROOT`:

```
$ cvs co CVSROOT
...
$ cat CVSROOT/CVS/Repository
/woodcock/jsh/RS/work/cvs/cvs/CVSROOT
$ cat CVSROOT/CVS/Entries
/checkoutlist/1.1/Mon Jun  2 18:17:04 1997//
/commitinfo/1.1/Mon Jun  2 18:17:04 1997//
/cvswrappers/1.1/Mon Jun  2 18:17:04 1997//
/editinfo/1.1/Mon Jun  2 18:17:04 1997//
/loginfo/1.1/Mon Jun  2 18:17:04 1997//
/modules/1.1/Mon Jun  2 18:17:04 1997//
/notify/1.1/Mon Jun  2 18:17:04 1997//
/rcsinfo/1.1/Mon Jun  2 18:17:04 1997//
/taginfo/1.1/Mon Jun  2 18:17:04 1997//
/verifymsg/1.1/Mon Jun  2 18:17:04 1997//
D
```

The `Repository` and `Root` files are obvious safeguards. If, while you're working, you change your `$CVSROOT` to point somewhere else–deliberately or by accident–the information in these directories ensures that any modifications you've made will be put back in the correct repository. Do this:

```
$ unset $CVSROOT
$ echo >> editinfo; echo >> loginfo
$ cvs ci
```

and you find yourself in the editor facing a screen that looks like this:

```
CVS: ----------------------------------------------------------------------
CVS: Enter Log. Lines beginning with 'CVS:' are removed automatically.
CVS:
CVS: Committing in
CVS:
CVS: Modified Files:
CVS:    editinfo loginfo
CVS: ----------------------------------------------------------------------
```

At this point, you insert a comment, explaining why you've placed a blank line at the end of these files, and exit the editor. When you do, you'll see something like the following:

```
Checking in editinfo;
/woodcock/jsh/RS/work/cvs/cvs/CVSROOT/editinfo,v  <--  editinfo
new revision: 1.7; previous revision: 1.6
done
Checking in loginfo;
/woodcock/jsh/RS/work/cvs/cvs/CVSROOT/loginfo,v  <--  loginfo
new revision: 1.3; previous revision: 1.2
done
cvs commit: Rebuilding administrative file database
```

Thus, the `Repository` and `Root` files let CVS remember where to check these in to, even though `CVSROOT` got

unset after you checked them out.

Understanding the goal of the `Entries` file requires thinking about bigger projects. Imagine, for a moment, a project so large that it has more than one file *and* more than one person working on it. Call those two people "Jeff" and "Jeff." Hmm. Call those two people "Zoe" and "Gillian." Consider the following scenario:

• Each developer checks out a copy of the repository file for project "fezmo."

• Zoe fixes a bug in her copy of the file `fezmo/charlie/dotsero` and checks it in using `cvs ci fezmo` (or just `cvs ci`, if she's in any directory that contains `dotsero` in one of its subdirectories. CVS will find it, realize that it's changed and check it in).

• Gillian fixes a bug in *her* copy of `fezmo/charlie/dotsero` and … what? If there were no source code control at all, Gillian's update could overwrite Zoe's update, wiping out Zoe's work.

If you're used to RCS, you may already be saying, "Only one of them could have checked the file out for editing. That must have been Zoe, so Gillian now has to re-check-out the file for editing." But that wouldn't be practical either, because that means one person would be grabbing and releasing locks on perhaps thousands of files, and Zoe and Gillian, and every other developer, would have to keep track of exactly who had locks on which files at all times.

Under CVS, Gillian, like Zoe, says `cvs ci`, CVS sees that `dotsero` has changed, then, before checking it in, *looks in the* `Entries` *file to see if the version Gillian checked out matches the version at the top of the tree*. Because it doesn't, Gillian gets a message that says there's a problem, and CVS points her toward another command, `cvs update`. This command will help Gillian update her version of all files in the collection, and will help her find and resolve conflicts between her changes and any other changes that have been done since she checked out her base version.

Administrative Files

CVS is easy enough to use that if it only extended RCS to let us handle collections of files and track their revisions, we'd be happy. It turns out, however, that there's a lot more to it. Some of the things CVS provides are commands and options. To show you how to take a look at what commands and options are available, we'll revert back to learning by not doing.

Try typing this:

```
$ cvs -:
```

Note that "`:`" is not a legal option for any UNIX command we know of, including `ls`, so we often use it to get commands to give us a usage message.

(Which reminds us of a joke that reappeared on the Net the other day: A brave knight approached an evil magician at the bridge–you've seen this *Monty Python* movie, so you at least know the form of what's coming next: The knight has to answer three questions to cross the bridge, or he will be cast into the abyss. The magician asks, "What is your name?" The knight answers, "Sir Brian of Bell." The magician asks, "What is your quest?" In a clear, firm voice, the knight answers, "I seek the Holy Grail." The magician, demonstrating just how evil he is, asks, "What four lowercase alphabetic characters are not legal flag arguments to the Berkeley UNIX implementation of `ls`?" Sir Brian, of course, hasn't the foggiest idea, which is the end of that particular knight.)

Try this, just the way the usage message tells you to:

```
$ cvs --help-commands
```

If you want to learn more about any of these commands, you can experiment with them. Or you can always do this:

```
$ man cvs
```

What else? We have mentioned some of the files in `$CVSROOT/CVSROOT`, such as `history` and `rcsinfo`, but there's more. Actually, there's a lot more, so rather than try to tackle it right now, let's wait and talk about it next time. If you want to play around with it in the meantime, you can get the CVS software from `http://www.loria.fr/~molli/cvs-index.html`.

Until then, happy trails.