

Chapter 4

Advanced CVS

Beyond the Basics

It is one thing to know how to make CVS work for you, but it is quite a different thing to actually include CVS in your software development process. In Chapter 2, we already explained the basic function of the typical CVS cycle: checkout, update, commit, update, and commit. In this chapter, we'll elaborate on the cycle and discuss how CVS can be used for software developers to collaborate and manage complex projects. Some of the techniques we introduce here cover new CVS commands, but many merely explain better ways to use commands that you already learned in previous chapters.

CVS as a Communication Device

It comes as a surprise to many to discover that CVS can function as a communications device as well as a record keeper. This section concentrates on how you can use CVS to keep participants informed about what's going on in a project. As is true of other aspects of CVS, these features reward collaboration and cooperation. The participants must want to be informed; if people choose not to use the communications features, there's nothing CVS can do about it.

Watches: Knowing Who's Working on What, When

In its default behavior, CVS treats each working copy as an isolated sandbox. No one knows what you're doing in your working copy until you commit your changes. In turn, you don't know

what others are doing in theirs—except via the usual methods of communication, such as shouting across the cubicles, "Hey, I'm going to work on housekeeping.c now. Let me know if you're editing it, too, so we can avoid conflicts!"

This informality works for projects where people have a general idea of who's responsible for what. However, this process breaks down when a large number of developers are active in all parts of a code base and need to avoid conflicts. In such cases, they frequently have to cross each others' areas of responsibility but can't shout across the cubicles at each other because they're geographically dispersed. Clearly, the CVS **admin –l** command to get reserved checkouts (thereby effectively locking the file in question) is an outdated collaboration mechanism.

Fortunately, CVS provides a mechanism beyond its default behavior. A CVS feature called *watches* provides developers with a way to notify each other about who is working on which files at a given time. By "setting a watch" on a file, a developer can have CVS notify him/her (by email or GSM SMS messages or any other kind of notification) if anyone else starts to work on that file. The **watch** command, therefore, specifies that developers should run **cvs edit** before editing *files*. CVS will create read-only working copies of *files* to remind developers to run the **cvs edit** command before working on them.

To use watches, you must modify one or two files in the repository administrative area, and developers must add some extra steps to the usual checkout/update/commit cycle. The changes on the repository side are fairly simple: You edit the CVSROOT/notify file so that CVS knows how notifications are to be performed. You also have to add lines to the CVSROOT/users file, which supplies external email addresses.

On the working copy side, developers have to tell CVS which files they want to watch so that CVS can send them notifications when someone else starts editing those files. They also need to tell CVS when someone starts or stops editing a file, so CVS can send out notifications to others who might be watching. The following commands are used to implement these extra steps:

- **cvs watch**
- **cvs edit**
- **cvs unedit**

Note

*The **watch** command differs from the usual CVS command pattern in that it requires further subcommands, such as **cvs watch add...**, **cvs watch remove...**, and so on.*

In the following example, we look at how to turn on watches in the repository and then how to use watches for the developers. The two example users, jrandom and mosheb, each have separate working copies of the same project; the working copies can even be on different machines. As usual, all examples assume that the **$CVSROOT** environment variable

has already been set, so there's no need to pass **-d *REPOSITORY*** to any CVS commands.

Enabling Watches in the Repository

First, the CVSROOT/notify file must be edited to turn on email notification. One of the developers can do this, but usually it is the repository administrator who has the permissions to change the repository's administrative files. In any case, the first thing to do is check out the administrative area and edit the notify file:

```
yarkon$ cvs -q co CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/loginfo
U CVSROOT/modules
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg
yarkon$ cd CVSROOT
yarkon$ emacs notify
...
```

When you edit the notify file for the first time, you'll see something like this:

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

All you have to do is remove the initial comment, the # sign, from the beginning of the line. The notify file provides the same flexible interface as the other administrative files, with regular expressions matching against directory names. In our experience, however, this feature is not used very often.

To specify email notification, the line

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

should work on any standard Unix machine. This command causes notifications to be sent as emails with the subject line "CVS notification" (the special expression **ALL** matches any directory, as usual). The expression "CVS notification" has become something of a standard in CVS, but you are free to put in there whatever you like. Having uncommented that line, run **commit** on the notify file so the repository is aware of the change:

```
yarkon$ cvs ci -m "turned on watch notification"
cvs commit: Examining .
Checking in notify;
/usr/local/newrepos/CVSROOT/notify,v  <--  notify
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
yarkon$
```

Editing the notify file in this way might be all that you need to do for watches in the repository. However, if there are remote developers working on the project, you might need to edit the CVSROOT/users file, too. The purpose of the users file is to tell CVS where to send email notifications for those users who have external email addresses. The format of each line in the users file is:

```
CVS_USERNAME:EMAIL_ADDRESS
```

Here's an example:

```
moshe:moshe@moelabs.com
```

The CVS username at the beginning of the line must correspond to a CVS username in CVSROOT/password if one is present and the **pserver** access method is being used. If a CVS username is not present in CVSROOT/password, or if the **pserver** access method isn't being used, the CVS username must correspond to the server-side system username of the person running CVS. Following the colon is an external email address to which CVS should send watch notifications for that user.

In some of the older CVS distributions, the users file was not included. If you happen to install such a CVS distribution (and because it is an administrative file), you must not only create, run **cvs add** on, and commit the users file in the usual way, but also add it to CVSROOT/checkoutlist so that a checked-out copy is always maintained in the repository.

Here is a sample session demonstrating this:

```
yarkon$ emacs checkoutlist
  ... (add the line for the users file) ...
```

```
yarkon$ emacs users
  ... (add the line for mosheb) ...
yarkon$ cvs add users
yarkon$ cvs ci -m "added users to checkoutlist, mosheb to users"
cvs commit: Examining .
Checking in checkoutlist;
/usr/local/newrepos/CVSROOT/checkoutlist,v  <--  checkoutlist
new revision: 1.2; previous revision: 1.1
done
Checking in users;
/usr/local/newrepos/CVSROOT/users,v  <--  users
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
yarkon$
```

It's possible to use expanded-format email addresses in CVSROOT/users, but be careful to encapsulate all white space within either single or double quotes. For example, the following will work

```
mosheb:"Moshe Bar <moshe@somedomain.com>"
```

or

```
mosheb:'Moshe Bar <moshe@somedomain.com>'
```

However, this will not work:

```
mosheb:"Moshe Bar" <moshe@somedomain.com>
```

When in doubt, you should test by running the command line given in the notify file manually. Just replace the **%s** in

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

with what follows the colon in users. If it works when you run it at a command prompt, it should work in the users file, too.

When you finish, the checkout file will look like this:

```
# The "checkoutlist" file is used to support additional version controlled
# administrative files in $CVSROOT/CVSROOT, such as template files.
#
# The first entry on a line is a filename which will be checked out from
# the corresponding RCS file in the $CVSROOT/CVSROOT directory.
# The remainder of the line is an error message to use if the file cannot
```

```
# be checked out.
#
# File format:
#
#       [<whitespace>]<filename><whitespace><error message><end-of-line>
#
# comment lines begin with '#'
    users   Unable to check out 'users' file in CVSROOT.
```

The users file will look like this:

```
mosheb:moshe@somedomain.com
```

With that, the repository is set up for watches. Let's see what developers need to do on their side to be notified.

Using Watches in Development

First, a developer checks out a working copy and adds herself to the list of watchers for one of the files in the project:

```
yarkon$ whoami
jrandom
yarkon$ cvs -q co myproj
U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
yarkon$ cd myproj
yarkon$ cvs watch add hello.c
yarkon$
```

The last command, **cvs watch add hello.c**, tells CVS to notify jrandom if anyone else starts working on hello.c (that is, it adds jrandom to hello.c's watch list). For CVS to send notifications as soon as a file is being edited, the user who is editing it has to announce the fact by running **cvs edit** on the file first. Because of its inherent conceptual design, CVS has no other way of knowing when someone starts working on a file. Once checkout is finished, CVS isn't usually invoked until the next update or commit, which happens after the file has already been edited:

```
ayalon$ whoami
mosheb
ayalon$ cvs -q co myproj
U myproj/README.txt
U myproj/foo.gif
```

```
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
ayalon$ cd myproj
ayalon$ cvs edit hello.c
ayalon$ emacs hello.c
...
```

When the user mosheb runs **cvs edit hello.c**, CVS looks at the watch list for hello.c, sees that jrandom is on it, and sends email to jrandom telling her that mosheb has started editing the file. In the classic CVS tradition of perfection, the email even appears to come from mosheb:

```
From: mosheb
Subject: CVS notification
To: jrandom
Date: Sun, 8 Jul 2001 22:14:43 -0500

myproj hello.c
--
Triggered edit watch on /usr/local/newrepos/myproj
By mosheb
```

Furthermore, every time that mosheb (or anyone) commits a new revision of hello.c, jrandom will receive another email:

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

After receiving these emails, jrandom might want to update hello.c immediately to see what mosheb has done, or perhaps she'll email mosheb to find out why he's working on that file. You might have noticed that nothing forced mosheb to remember to run **cvs edit**—presumably he did it because he wanted jrandom to know what he was up to (if he forgets to do **cvs edit**, his commits will still trigger the notifications). Using **cvs edit** is always a good idea because it triggers the notifications *before* you start to work on a file. The watchers can contact you if they think there might be a conflict, before you've wasted a lot of time.

In a somewhat disputable decision, the developers of CVS assumed that anyone who runs **cvs edit** on a file wants to be added to the file's watch list, at least temporarily, in case someone else starts to edit it. When mosheb ran **cvs edit**, he became a watcher of hello.c. Both he and jrandom would have received notification if a third party had run **cvs edit** on that file (or committed it).

However, CVS also assumes that the person editing the file wants to be on its watch list only while he or she is editing it. These users are taken off the watch list when they've

finished editing. If they prefer to be permanent watchers of the file, they have to run **cvs watch add**. CVS makes a default assumption that someone is finished editing (until the next time, anyway) when he or she commits a file.

Anyone who gets on a file's watch list solely by virtue of having run **cvs edit** on that file is known as a *temporary watcher* and is taken off the watch list as soon as he or she commits a change to the file. If he/she wants to edit it again, **cvs edit** needs to be re-run.

CVS's assumption that the first commit ends the editing session is only a best guess, of course, because CVS doesn't know how many commits the person will need to finish his or her changes. The guess is probably accurate for "one-off" changes—changes where someone just needs to make one quick fix to a file and commit it. For more prolonged editing sessions involving several commits, users should add themselves permanently to the file's watch list:

```
ayalon$ cvs watch add hello.c
ayalon$ cvs edit hello.c
ayalon$ emacs hello.c
...
ayalon$ cvs commit -m "print goodbye in addition to hello"
```

Even after the commit, mosheb remains a watcher of hello.c because he ran **watch add** on it. Obviously, mosheb will not receive notification of his own edits; only other watchers will. CVS is smart enough not to notify you about actions that you took.

Ending an Editing Session

If you don't want to commit but want to explicitly end an editing session, you can do so by running **cvs unedit**:

```
ayalon$ cvs unedit hello.c
```

Be careful, though! This does more than just notify all watchers that you've finished editing—it also offers to revert any uncommitted changes that you've made to the file:

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

If you answer "y," CVS undoes all your changes and notifies watchers that you're not editing the file anymore. If you answer "n," CVS keeps your changes and also keeps you registered as an editor of the file (so no notification goes out—it is as if you never ran **cvs unedit** at all). The possibility of CVS undoing all of your changes at a single keystroke is a bit scary, but the rationale is easy to understand. If you declare to the world that you're ending an editing session, then any changes you haven't committed are probably changes you don't

mean to keep. At least, that's the way CVS sees it. So, be careful!

Controlling What Actions Are Watched

By default, watchers are notified about three kinds of actions: edits, commits, and unedits. However, if you want to be notified only about, say, commits, you can restrict notifications by adjusting your watch with the **-a** flag (*a* for action):

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

Or if you want to watch edits and commits but don't care about unedits, you could pass the **-a** flag twice:

```
yarkon$ cvs watch add -a edit -a commit hello.c
```

Adding a watch with the **-a** flag will never cause any of your existing watches to be removed. If you were watching for all three kinds of actions on hello.c, running

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

has no effect—you'll still be a watcher for all three actions. To remove watches, you should run

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

which is similar to **add** in that, by default, it removes your watches for all three actions. If you pass **-a** arguments, it removes only the watches you specify:

```
yarkon$ cvs watch remove -a commit hello.c
```

This means that you want to stop receiving notifications about commits but to continue to receive notifications about edits and unedits (assuming you were watching edits and unedits to begin with, that is).

There are two special actions you can pass to the **-a** flag:

- **all**—All actions are eligible for watching (edits, commits, and unedits, as of this writing).
- **none**—No actions are eligible for watching.

Because CVS's default behavior, in the absence of **-a**, is to watch all actions, and because watching **none** is the same as removing yourself from the watch list entirely, it's hard to imagine a situation in which it would be useful to specify either of these two special actions. However, **cvs edit** also takes the **-a** option, and in this case, it can be useful to specify **all** or **none**. For example, someone working on a file very briefly might not want to receive any notifications about what other people do with the file. Thus, this command

```
ayalon$ whoami
mosheb
ayalon$ cvs edit -a none README.txt
```

causes watchers of README.txt to be notified that mosheb is about to work on it, but mosheb would *not* be added as a temporary watcher of README.txt during his editing session (which he normally would have been), because he asked not to watch any actions.

Finding Out Who's Watching What

At times, it might be useful in the development process to determine who's watching before you even run **cvs edit**, or to see who is editing what without adding yourself to any watch lists. Or you might want to check your own status because you have forgotten exactly what that status is. (After setting and unsetting a few watches and committing some files, it's easy to lose track of what you're watching and editing.)

CVS provides two commands to show who's watching and who's editing files—**cvs watchers** and **cvs editors**:

```
yarkon$ whoami
jrandom
yarkon$ cvs watch add hello.c
yarkon$ cvs watchers hello.c
hello.c jrandom  edit unedit  commit
yarkon$ cvs watch remove -a unedit hello.c
yarkon$ cvs watchers hello.c
hello.c jrandom  edit commit
yarkon$ cvs watch add README.txt
yarkon$ cvs watchers
README.txt      jrandom edit    unedit  commit
hello.c jrandom edit    commit
yarkon$
```

Notice that the last **cvs watchers** command doesn't specify any files and, therefore, shows watchers for all files that have watchers.

All of the **watch** and **edit** commands have this behavior in common with other CVS commands. If you specify file names, they act on those files. If you specify directory names, they act on everything in that directory and its subdirectories. If you don't specify anything, they act on the current directory and everything underneath it, to as many levels of depth as are available. For example (continuing with the same session):

```
yarkon$ cvs watch add a-subdir/whatever.c
yarkon$ cvs watchers
README.txt      jrandom edit    unedit  commit
hello.c jrandom edit    commit
```

```
a-subdir/whatever.c     jrandom edit    unedit  commit
yarkon$ cvs watch add
yarkon$ cvs watchers
README.txt      jrandom edit    unedit  commit
foo.gif jrandom edit    unedit  commit
hello.c jrandom edit    commit  unedit
a-subdir/whatever.c     jrandom edit    unedit  commit
a-subdir/subsubdir/fish.c       jrandom edit    unedit  commit
b-subdir/random.c       jrandom edit    unedit  commit
yarkon$
```

The last two commands made jrandom a watcher of every file in the project and then showed the watch list for every file in the project, respectively. The output of **cvs watchers** doesn't always line up perfectly in columns because it mixes tab stops with information of varying lengths, but the lines are consistently formatted:

```
[FILENAME] [whitespace] WATCHER [whitespace] ACTIONS-BEING-WATCHED...
```

Now watch what happens when mosheb starts to edit one of the files:

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

The file hello.c has acquired another watcher: mosheb himself. Note that the file name is not repeated but is left as white space at the beginning of the line—this is important if you ever want to write a program that parses **watchers** output.

Because he's editing hello.c, mosheb has a temporary watch on the file that goes away as soon as he commits a new revision of hello.c. The prefix "**t**" in front of each of the actions indicates that these are temporary watches. If mosheb adds himself as a regular watcher of hello.c

```
ayalon$ cvs watch add hello.c
README.txt      jrandom edit    unedit  commit
foo.gif jrandom edit    unedit  commit
hello.c jrandom edit    commit  unedit
       mosheb  tedit   tunedit tcommit edit    unedit  commit
a-subdir/whatever.c     jrandom edit    unedit  commit
a-subdir/subsubdir/fish.c       jrandom edit    unedit  commit
```

```
b-subdir/random.c       jrandom edit    unedit  commit
```

he is listed as both a temporary watcher and a permanent watcher. You might think that the permanent watch status would simply override the temporary and the line would look like this:

```
mosheb  edit    unedit  commit
```

However, CVS can't just replace the temporary watches because it has no way of knowing in what order things happen. Will mosheb remove himself from the permanent watch list before ending his editing session, or will he finish the edits while still remaining a watcher? If the former, the **edit/unedit/commit** actions disappear while the **tedit/tunedit/tcommit** ones remain; if the latter, the reverse will be true.

Anyway, that side of the watch list is usually not of great concern. Most of the time, what you want to do is run

```
yarkon$ cvs watchers
```

or

```
yarkon$ cvs editors
```

from the top level of a project and see who's doing what.

Reminding People to Use Watches

You've probably noticed that the watch features are utterly dependent on the cooperation of all the developers. If someone just starts editing a file without first running **cvs edit**, no one else will know about it until the changes get committed. Because **cvs edit** is an additional step and not part of the normal development routine, people can easily forget to do it.

Although CVS can't force someone to use **cvs edit**, it does have a mechanism for reminding people to do so—the **watch on** command:

```
yarkon$ cvs -q co myproj
U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
yarkon$ cd myproj
yarkon$ cvs watch on hello.c
yarkon$
```

By running **cvs watch on hello.c**, jrandom causes future checkouts of myproj to create hello.c as a read-only file in the working copy. When mosheb tries to work on it, he'll

discover that it's read-only and be reminded to run **cvs edit** first:

```
ayalon$ cvs -q co myproj
U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
ayalon$ cd myproj
ayalon$ ls -l
total 6
drwxr-xr-x   2 mosheb    users        1024 Jul 19 01:06 CVS/
-rw-r--r--   1 mosheb    users          38 Jul 12 11:28 README.txt
drwxr-xr-x   4 mosheb    users        1024 Jul 19 01:06 a-subdir/
drwxr-xr-x   3 mosheb    users        1024 Jul 19 01:06 b-subdir/
-rw-r--r--   1 mosheb    users         673 Jun 20 22:47 foo.gif
-r--r--r--   1 mosheb    users         188 Jul 18 01:20 hello.c
ayalon$
```

When he does so, the file becomes read-write. He can then edit it, and when he commits, it becomes read-only again:

```
ayalon$ cvs edit hello.c
ayalon$ ls -l hello.c
-rw-r--r--   1 mosheb    users         188 Jul 18 01:20 hello.c
ayalon$ emacs hello.c
  ...
ayalon$ cvs commit -m "say hello in Hebrew" hello.c
Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v  <--  hello.c
new revision: 1.12; previous revision: 1.11
done
ayalon$ ls -l hello.c
-r--r--r--   1 mosheb    users         210 Jul 19 01:12 hello.c
ayalon$
```

His edit and commit will send notification to all watchers of hello.c. Note that jrandom isn't necessarily one of them. By running **cvs watch on hello.c**, jrandom did not add herself to the watch list for that file; she merely specified that it should be checked out as read-only. People who want to watch a file must remember to add themselves to its watch list—CVS cannot help them with that.

Turning on watches for a single file might be the exception; it's more common to turn on watches project-wide:

```
yarkon$ cvs -q co myproj
U myproj/README.txt
U myproj/foo.gif
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
yarkon$ cd myproj
yarkon$ cvs watch on
yarkon$
```

This action amounts to announcing a policy decision for the entire project: "Please use **cvs edit** to tell watchers what you're working on, and feel free to watch any file you're interested in or responsible for." Every file in the project will be checked out as read-only, and thus people will be reminded that they're expected to use **cvs edit** before working on anything.

Strangely, although checkouts of watched files make them read-only, updates do not. If mosheb had checked out his working copy *before* jrandom ran **cvs watch on**, his files would have stayed read-write, remaining so even after updates. However, any file he commits after jrandom turns watching on will become read-only. If jrandom turns off watches

```
yarkon$ cvs watch off
```

mosheb's read-only files do not magically become read-write. On the other hand, after he commits one, it will not revert to read-only again (as it would have if watches were still on).

It's worth noting that mosheb could, were he truly devious (as sometimes people claim), make files in his working copy writable by using the standard Unix **chmod** command, bypassing **cvs edit** entirely:

```
ayalon$ chmod u+w hello.c
```

Or, he could get everything in one fell swoop:

```
ayalon$ chmod -R u+w .
```

CVS can do nothing about this. Working copies are, by their nature, private sandboxes—the watch features can open them up to public scrutiny a little bit, but only as far as the developer permits. Only when a developer does something that affects the repository (such as commits) is her privacy unconditionally lost.

The relationship among **watch add**, **watch remove**, **watch on**, and **watch off** probably seems a bit confusing, so we'll summarize the overall scheme. The **add** and **remove** options are about adding or removing users from a file's watch list. They don't have anything to do with whether files are read-only on checkout or after commits. The **on** and **off** options are only about file permissions. They don't have anything to do with who is on a file's watch

list; rather, they are tools to help remind developers of the watch policy by causing working-copy files to become read-only.

All of this may seem a little inconsistent, because the watches go against the basic CVS philosophy of letting people do what they want and then caring about it at commit time. With watches, CVS gives developers convenient shortcuts for informing each other of what's going on in their working copies; however, it has no way to enforce observation policies, and it doesn't have a clear definition of what constitutes an editing session. Nevertheless, watches can be helpful in certain circumstances if developers work with them.

Log Messages and Commit Emails

Commit emails are notices sent out at commit time, showing the log message and files involved in the commit. They usually go to all project participants and sometimes to other interested parties. The details of setting up commit emails were covered in Chapter 3, so we won't repeat them here. We have noticed, however, that commit emails can sometimes result in unexpected side effects to projects, effects that you might want to take into account if you set up commit emails for your project.

First, be prepared for the messages to be mostly ignored. Whether people read them depends, at least partly, on the frequency of commits in your project. Do developers tend to commit one big change at the end of the day, or many small changes throughout the day? In the latter case, the thicker the barrage of tiny commit notices raining down on the developers all day long, the less inclined they will be to pay attention to each message.

This doesn't mean the notices aren't useful, just that you shouldn't count on every person reading every message. It's still a convenient way for people to keep tabs on who's doing what (without the intrusiveness of watches). When the emails go to a publicly subscribable mailing list, they are a wonderful mechanism for giving interested users (and future developers!) a chance to see what happens in the code on a daily basis.

You might want to consider having a designated developer who watches all log messages and has an overview of activity across the entire project (of course, a good project leader will probably be doing this anyway). If there are clear divisions of responsibility—say, certain developers are "in charge of" certain subdirectories of the project—you could (or maybe should?) do some fancy scripting in CVSROOT/loginfo to see that each responsible party receives specially marked notices of changes made in their area. This helps ensure that the developers will read at least the email that pertains to their subdirectories.

A more interesting side effect happens when commit emails aren't ignored. People start to use them as a realtime communications method. Here's the kind of log message that can result:

```
Finished feedback form; fixed the fonts and background colors
on the home page. And now I'm off for a beer! Anyone want to join me?
```

There's nothing wrong with this, and it makes the logs more fun to read later. However, people need to be aware that log messages are not only distributed by email but are also pre-

served forever in the project's history. For example, griping about customer specifications is a frequent pastime among programmers; it's not hard to imagine someone committing a log message like this one, knowing that the other programmers will soon see it in their email:

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

This makes for an amusing email, but what happens if the customer reviews the logs someday? (Similar concerns have led more than one site to set up CVSROOT/loginfo so that it invokes scripts to guard against offensive words in log messages!)

The overall effect of commit emails seems to be that people become less willing to write short or obscure log messages, which is probably a good thing. However, they might need to be reminded that their audience is anyone who might ever read the logs, not just the people receiving commit emails.

Changing a Log Message after It's Been Committed

Just in case someone does commit a regrettable log message, CVS enables you to rewrite logs after they've been committed. It's done with the **-m** option to the **admin** command and allows you to change one log message (per revision, per file) at a time. Here's how it works:

```
yarkon$ cvs admin -m 1.7:"Truncate four-digit years to two in input." date.c
RCS file: /usr/local/newrepos/someproj/date.c,v
done
yarkon$
```

The original, offensive log message that was committed with revision 1.7 has been replaced with a perfectly innocent—albeit duller—message. (Don't forget the colon separating the revision number from the new log message.)

If the bad message was committed into multiple files, you'll have to run **cvs admin** separately for each one, because the revision number is different for each file. Therefore, this is one of the few commands in CVS that requires you to pass a single file name as an argument:

```
yarkon$ cvs admin -m 1.2:"very boring log message" hello.c README.txt foo.gif
cvs admin: while processing more than one file:
cvs [admin aborted]: attempt to specify a numeric revision
yarkon$
```

Confusingly, you get the same error if you pass no file names (because CVS then assumes all the files in the current directory and below are implied arguments):

```
yarkon$ cvs admin -m 1.2:"very boring log message"
cvs admin: while processing more than one file:
cvs [admin aborted]: attempt to specify a numeric revision
yarkon$
```

As is unfortunately often the case with CVS error messages, you have to see things from CVS's point of view before the message makes sense!

Invoking **admin -m** actually changes the project's history, so use it with extreme care. There will be no record that the log message was ever changed—it will simply appear as though that revision had been originally committed with the new log message. No trace of the old message will be left anywhere (unless you saved the original commit email).

Although its name might seem to imply that only the designated CVS administrator can use it, normally anyone can run **cvs admin** as long as he or she has write access to the project in question. However, if a Unix group named **cvs admin** exists on the server, the use of this command is restricted to members of that group (except that the **cvs admin -k** command is still allowed for all). Either way, **cvs admin** is to be used with caution; the ability to change a project's history is mild compared with the other potentially damaging things it can do, like destroying—for good—your hard work.

Getting Rid of a Working Copy

In typical CVS usage, the way to get rid of a working copy directory tree is to remove it as you would any other directory tree:

```
ayalon$ rm -rf myproj
```

However, if you eliminate your working copy this way, other developers will not know that you have stopped using it. CVS provides a command to relinquish a working copy explicitly. Think of **release** as the opposite of **checkout**—you're telling the repository that you're finished with the working copy now. Like **checkout**, **release** is invoked from the parent directory of the tree:

```
ayalon$ pwd
/home/mosheb/myproj
ayalon$ cd ..
ayalon$ ls
myproj
ayalon$ cvs release myproj
You have [0] altered files in this repository.
Are you sure you want to release directory 'myproj': y
ayalon$
```

If there are any uncommitted changes in the repository, the release fails, meaning that it just lists the modified files and otherwise has no effect. Assuming the tree is clean (totally up to date), **release** records in the repository that the working copy has been released.

You can also have **release** automatically delete the working tree for you, by passing the **-d** flag:

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

As of CVS version 1.10.6, the **release** command is not able to deduce the repository's location by examining the working copy (this is because **release** is invoked from above the working copy, not within it). You must pass the **-d *REPOSITORY*** global option or make sure that your **CVSROOT** environment variable is set correctly. (This bug might be fixed in future versions of CVS.)

The Cederqvist claims that if you use **release** instead of just deleting the working tree, people with watches set on the released files will be notified just as if you had run **unedit**. However, we tried to verify this experimentally, and it does not seem to be true.

A Bird's-Eye View of Project History

In Chapter 3, we mentioned the **cvs history** command. This command displays a summary of all checkouts, commits, updates, rtags, and releases done in the repository (at least, since logging was enabled by the creation of the CVSROOT/history file in the repository). You can control the format and contents of the summary with various options.

The first step is to make sure that logging is enabled in your repository. The repository administrator should first make sure there is a history file:

```
yarkon$ cd /usr/local/newrepos/CVSROOT
yarkon$ ls -l history
ls: history: No such file or directory
yarkon$
```

If there isn't one, create it as follows:

```
yarkon$ touch history
yarkon$ ls -l history
-rw-r--r--   1 jrandom   cvs           0 Jul 22 14:57 history
yarkon$
```

Everyone who uses the repository needs to be able to write to this history file; otherwise, they'll get an error every time they try to run a CVS command that modifies that file. The easiest way to enable everyone to write to the history file is simply to make the file world-writable:

```
yarkon$ chmod a+rw history
yarkon$ ls -l history
```

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

Note

*If the repository was created with the **cvs init** command, the history file already exists. You might still have to fix its permissions, however.*

The rest of these examples assume that history logging has been enabled for a while, so that data has had time to accumulate in the history file.

The output of **cvs history** is somewhat terse (it's probably intended to be parsed by programs rather than humans, although it is readable with a little study). Let's run it once and see what we get:

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

Before we examine the output, notice that the invocation included two options: **-e** and **-a**. When you run **history**, you almost always need to pass options telling it what data to report

and how to report it. In this respect, it differs from most other CVS commands, which usually do something useful when invoked without any options. In this example, the two flags meant "everything" (show every kind of event that happened) and "all" (for all users), respectively.

history differs from other commands in another way: Although it is usually invoked from within a working copy, it does not restrict its output to that working copy's project. Instead, it shows all history events from all projects in the repository—the working copy merely serves to tell CVS which repository to retrieve the history data from . (In the preceding example, the only history data in that repository is for the "myproj" project, so that's all we see.)

The general format of the output is:

```
CODE DATE USER [REVISION] [FILE] PATH_IN_REPOSITORY ACTUAL_WORKING_COPY_NAME
```

The code letters refer to various CVS operations, as shown in Table 4.1.

For operations (such as **checkout**) that are about the project as a whole rather than about individual files, the revision and file are omitted, and the repository path is placed between the equal signs.

Although the output of the **history** command was designed to be compact, parsable input for other programs, CVS still gives you a lot of control over its scope and content. The options shown in Table 4.2 control what types of events get reported.

Once you have selected what type of events you want reported, you can filter further with the options shown in Table 4.3.

Bird's-Eye View, with Telescope: The **annotate** Command

Table 4.1 The meaning of the code letters.

Letter	Meaning
O	Checkout
T	Tag
F	Release
W	Update (no user file, remove from entries file)
U	Update (file overwrote unmodified user file)
G	Update (file was merged successfully into modified user file)
C	Update (file was merged, but has conflicts with modified user file)
M	Commit (from modified file)
A	Commit (an added file)
R	Commit (the removal of a file)
E	Export (see Chapter 11)

Table 4.2 Options to filter by event type.

Option	Meaning
-m *MODULE*	Show historical events affecting ***MODULE***.
-c	Show **commit** events.
-o	Show **checkout** events.
-T	Show **tag** events.
-x *CODE(S)*	Show all events of type ***CODE*** (one or more of **OTFWUGCMARE**).
-e	Show all types of events, period.

Table 4.3 Options to filter by user.

Option	Meaning
-a	Show actions taken by all users.
-w	Show only actions taken from within this working copy.
-l	Show only the last time this user took the action.
-u *USER*	Show records for ***USER***.

Whereas the **history** command gives an overview of project activity, the **annotate** command is a way of attaching a zoom lens to the view. With **annotate**, you can see who was the last person to touch each line of a file, and at what revision they touched it:

```
yarkon$ cvs annotate
Annotations for README.txt
***************
1.14         (jrandom  25-Jul-01): blah
1.13         (jrandom  25-Jul-01): test 3 for history
1.12         (mosheb   19-Jul-01): test 2
1.11         (mosheb   19-Jul-01): test
1.10         (jrandom  12-Jul-01): blah
1.1          (jrandom  20-Jun-01): Just a test project.
1.4          (jrandom  21-Jun-01): yeah.
1.5          (jrandom  21-Jun-01): nope.
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.15         (jrandom  25-Jul-01):   /* another test for history */
1.13         (mosheb   19-Jul-01):   /* random change number two */
1.10         (jrandom  12-Jul-01):   /* test */
1.21         (jrandom  25-Jul-01):   printf ("Hellooo, world!\n");
1.3          (jrandom  21-Jun-01):   printf ("hmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
```

```
1.11          (mosheb   18-Jul-01):   /* added this comment */
1.16          (mosheb   25-Jul-01):   /* will merge these changes */
1.18          (jrandom  25-Jul-01):   /* will merge these changes too */
1.2           (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1           (jrandom  20-Jun-01): }
Annotations for a-subdir/whatever.c
***************
1.3           (jrandom  25-Jul-01): /* A completely non-empty C file. */
Annotations for a-subdir/subsubdir/fish.c
***************
1.2           (jrandom  25-Jul-01): /* An almost completely empty C file. */
Annotations for b-subdir/random.c
***************
1.1           (jrandom  20-Jun-01): /* A completely empty C file. */
yarkon$
```

The output of **annotate** is pretty intuitive. On the left are the revision number, developer, and date on which the line in question was added or last modified. On the right is the line itself, as of the current revision. Because every line is annotated, you can actually see the entire contents of the file, pushed over to the right by the annotation information.

If you specify a revision number or tag, the annotations are given as of that revision, meaning that it shows the most recent modification to each line at or before that revision. This is probably the most common way to use annotations—examining a particular revision of a single file to determine which developers were active in which parts of the file.

For example, in the output of the previous example, you can see that the most recent revision of hello.c is 1.21, in which jrandom did something to the line:

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

One way to find out what she did is to diff that revision against the previous one:

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

Another way to find out, while still retaining a file-wide view of everyone's activity, is to

compare the current annotations with the annotations from a previous revision:

```
yarkon$ cvs annotate -r 1.20 hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.15         (jrandom  25-Jul-01):   /* another test for history */
1.13         (mosheb   19-Jul-01):   /* random change number two */
1.10         (jrandom  12-Jul-01):   /* test */
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.3          (jrandom  21-Jun-01):   printf ("hmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.11         (mosheb   18-Jul-01):   /* added this comment */
1.16         (mosheb   25-Jul-01):   /* will merge these changes */
1.18         (jrandom  25-Jul-01):   /* will merge these changes too */
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$
```

Although running **diff** reveals the textual facts of the change more concisely, the annotation might be preferable because it places the changes in their historical context by showing how long the previous incarnation of the line had been present (in this case, all the way back to revision 1.1). That knowledge helps you decide whether to look at the logs to find out the motivation for the change:

```
yarkon$ cvs log -r 1.21 hello.c
RCS file: /usr/local/newrepos/myproj/hello.c,v
Working file: hello.c
head: 1.21
branch:
locks: strict
access list:
symbolic names:
       random-tag: 1.20
       start: 1.1.1.1
       jrandom: 1.1.1
keyword substitution: kv
total revisions: 22;    selected revisions: 1
description:
----------------------------
revision 1.21
date: 2001/07/25 20:17:42;  author: jrandom;  state: Exp;  lines: +1 -1
say hello with renewed enthusiasm
```

```
===================================================================
yarkon$
```

In addition to **-r**, you can also filter annotations using the **-D DATE** option:

```
yarkon$ cvs annotate -D "5 weeks ago" hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$ cvs annotate -D "3 weeks ago" hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.3          (jrandom  21-Jun-01):   printf ("hmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$
```

Annotations and Branches

By default, annotation always shows activity on the main trunk of development. Even when invoked from a branch working copy, it shows annotations for the trunk unless you specify otherwise. (This tendency to favor the trunk is either a bug or a feature, depending on your point of view.) You can force CVS to annotate a branch by passing the branch tag as an argument to **-r**. Here is an example from a working copy in which hello.c is on a branch named "Brancho_Gratuito," with at least one change committed on that branch:

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

  Working revision:    1.10.2.2        Sun Jul 25 21:29:05 2001
  Repository revision: 1.10.2.2        /usr/local/newrepos/myproj/hello.c,v
  Sticky Tag:          Brancho_Gratuito (branch: 1.10.2)
  Sticky Date:         (none)
```

```
  Sticky Options:      (none)

yarkon$ cvs annotate hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.10         (jrandom  12-Jul-01):   /* test */
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.3          (jrandom  21-Jun-01):   printf ("hmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$ cvs annotate -r Brancho_Gratuito hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.10         (jrandom  12-Jul-01):   /* test */
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.10.2.2     (jrandom  25-Jul-01):   printf ("hmmmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.10.2.1     (jrandom  25-Jul-01):   printf ("added this line");
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$
```

You can also pass the branch number itself

```
yarkon$ cvs annotate -r 1.10.2 hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.10         (jrandom  12-Jul-01):   /* test */
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.10.2.2     (jrandom  25-Jul-01):   printf ("hmmmmm\n");
```

```
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.10.2.1     (jrandom  25-Jul-01):   printf ("added this line");
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$
```

or a full revision number from the branch:

```
yarkon$ cvs annotate -r 1.10.2.1 hello.c
Annotations for hello.c
***************
1.1          (jrandom  20-Jun-01): #include <stdio.h>
1.1          (jrandom  20-Jun-01):
1.1          (jrandom  20-Jun-01): void
1.1          (jrandom  20-Jun-01): main ()
1.1          (jrandom  20-Jun-01): {
1.10         (jrandom  12-Jul-01):   /* test */
1.1          (jrandom  20-Jun-01):   printf ("Hello, world!\n");
1.3          (jrandom  21-Jun-01):   printf ("hmmm\n");
1.4          (jrandom  21-Jun-01):   printf ("double hmmm\n");
1.10.2.1     (jrandom  25-Jul-01):   printf ("added this line");
1.2          (jrandom  21-Jun-01):   printf ("Goodbye, world!\n");
1.1          (jrandom  20-Jun-01): }
yarkon$
```

If you do this, remember that the numbers are valid only for that particular file. In general, it's probably better to use the branch name wherever possible.

Using Keyword Expansion

You might recall a brief mention of keyword expansion in Chapter 2. RCS keywords are special words, surrounded by dollar signs, that CVS looks for in text files and expands into revision-control information. For example, if a file contains

```
$Author$
```

then when updating the file to a given revision, CVS will expand it to the username of the person who committed that revision:

```
$Author: jrandom $
```

CVS is also sensitive to keywords in their expanded form, so that once expanded, they continue to be updated as appropriate.

Although keywords don't actually offer any information that's not available by other means,

they give people a convenient way to see revision control facts embedded in the text of the file itself, rather than by invoking some arcane CVS operation.

Here are a few other commonly used keywords:

```
$Date$       ==>  date of last commit, expands to ==>
$Date: 2001/07/26 06:39:46 $

$Id$         ==>  filename, revision, date, and author; expands to ==>
$Id: hello.c,v 1.11 2001/07/26 06:39:46 jrandom Exp $

$Revision$   ==>  exactly what you think it is, expands to ==>
$Revision: 1.11 $

$Source$     ==> path to corresponding repository file, expands to ==>
$Source: /usr/local/newrepos/tossproj/hello.c,v $

$Log$        ==>  accumulating log messages for the file, expands to ==>
$Log: hello.c,v $
Revision 1.2  2001/07/26 06:47:52  jrandom
...and this is the second log message.

Revision 1.1  2001/07/26 06:39:46  jrandom
This is the first log message...
```

The **Log** keyword is the only one of these that expands to cover multiple lines, so its behavior is unique. Unlike the others, it does not replace the old expansion with the new one, but instead inserts the latest expansion, plus an additional blank line, right after the keyword (thereby pushing any previous expansions downward). Furthermore, any text between the beginning of the line and **$Log** is used as a prefix for the expansions (this is done to ensure that the log messages stay commented in program code). For example, if you put this into the file

```
// $Log$
```

it will expand to something like this on the first commit

```
// $Log: hello.c,v $
// Revision 1.14  2001/07/26 07:03:20  jrandom
// this is the first log message...
//
```

this on the second

```
// $Log: hello.c,v $
// Revision 1.15  2001/07/26 07:04:40  jrandom
// ...and this is the second log message...
```

```
//
// Revision 1.14  2001/07/26 07:03:20  jrandom
// this is the first log message...
//
```

and so on:

```
// $Log: hello.c,v $
// Revision 1.16  2001/07/26 07:05:34  jrandom
// ...and this is the third!
//
// Revision 1.15  2001/07/26 07:04:40  jrandom
// ...and this is the second log message...
//
// Revision 1.14  2001/07/26 07:03:20  jrandom
// this is the first log message...
//
```

You probably don't want to keep your entire log history in the file all the time; if you do, you can always remove the older sections when it starts to get too lengthy. It's certainly more convenient than running **cvs log**, and it might be worthwhile in projects where people must constantly read over the logs.

A more common technique is to include **$Revision$** in a file and use it as the version number for the program. This can work if the project consists of essentially one file or undergoes frequent releases, and if it has at least one file that is guaranteed to be modified between every release. You can even use an RCS keyword as a value in program code:

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

CVS expands that keyword just like any other; it has no concept of the programming language's semantics and does not assume that the double quotes protect the string in any way.

A complete list of keywords (there are a few more, rather obscure ones) is given Chapter 11.

Going out on a Limb: How to Work with Branches and Survive

Branches are one of the most important, but also one of the most easily misused, features of CVS. Isolating risky or disruptive changes onto a separate line of development until they stabilize can be immensely helpful. If not properly managed, however, branches can quickly propel a project into confusion and cascading chaos, as people lose track of what changes have been merged when. We do not go into the open source development aspects of branches in this section. That issue will be covered in Chapter 10.

To work successfully with branches, your development group should adhere to these principles:

- *Minimize the number of branches active at any one time*. The more branches under development at the same time, the more likely they are to conflict when merged into the trunk. In practical terms, the way to accomplish this is to merge as frequently as you can (whenever a branch is at a stable point) and to move development back onto the trunk as soon as it's feasible. By minimizing the amount of parallel development going on, everyone is better able to keep track of what's happening on each branch, and the possibility of conflicts on merge is reduced.

Note

This does not mean minimizing the absolute number of branches in the project, just the number being worked on at any given time.

- *Minimize the complexity—that is, the depth—of your branching scheme*. There are circumstances in which it's appropriate to have branches from branches, but they are very rare (you may get through your entire programming life without ever encountering one). Just because CVS makes it technically possible to have arbitrary levels of nested branching, and to merge from any branch to any other branch, doesn't mean you actually want to do these things. In most situations, it's best to have all your branches rooted at the trunk and to merge from branch to trunk and back out again.

- *Use consistently named tags to mark all branch and merge events*. Ideally, the meaning of each tag and its relationship to other branches and tags should be apparent from the tag name. (The point of this will become clearer as we go through the examples later in this chapter.)

With those principles in mind, let's take a look at a typical branch development scenario. We have jrandom on the trunk and mosheb on the branch, but note that there could just as well be multiple developers on the trunk and/or on the branch. Regular development along either line can involve any number of people; however, the tagging and merging are best done by one person on each side, as you'll see.

Merging Repeatedly into the Trunk

Let's assume mosheb needs to do development on a branch for a while to avoid destabilizing the trunk that he shares with jrandom. The first step is to create the branch. Notice how mosheb creates a regular (nonbranch) tag at the branch point first and then creates the branch:

```
ayalon$ pwd
/home/mosheb/myproj
ayalon$ cvs tag Root-of-Exotic_Greetings
cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
```

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

The point of tagging the trunk first is that quite probably you will want someday to retrieve the trunk as it was the moment the branch was created. If you ever need to do that, you'll have to have a way of referring to the trunk snapshot without referring to the branch itself. Obviously, you can't use the branch tag because that would retrieve the branch, not the revisions in the trunk that form the root of the branch. The only way to do it is to make a regular tag at the same revisions from which the branch sprouts. (Some people stick to this rule so faithfully that we considered listing it as "Branching Principle Number 4: Always create a nonbranch tag at the branch point." However, many sites don't do it and they generally seem to do okay, so it's really a matter of taste.) From here on, we will refer to this nonbranch tag as the *branch point tag*.

Notice also that a naming convention is being adhered to: The branch point tag begins with **Root-of-**, then the actual branch name, which uses underscores instead of hyphens to separate words. When the actual branch is created, its tag ends with the suffix **-branch** so that you can identify it as a branch tag just by looking at the tag name. (The branch point tag **Root-of-Exotic_Greetings** does not include the **-branch** because it is not a branch tag.) You don't have to use this particular naming convention, of course, but you should use some convention.

Of course, we are being extra-pedantic here. In smallish projects, where everyone knows who's doing what and confusion is easy to recover from, these conventions don't have to be used. Whether you use a branch point tag or have a strict naming convention for your tags depends on the complexity of the project and the branching scheme. (Also, don't forget that you can always go back later and update old tags to use new conventions by retrieving an old tagged version, adding the new tag, and then deleting the old tag.)

Now, mosheb is ready to start working on the branch:

```
ayalon$ cvs update -r Exotic_Greetings-branch
cvs update: Updating .
cvs update: Updating a-subdir
cvs update: Updating a-subdir/subsubdir
cvs update: Updating b-subdir
ayalon$
```

He makes some changes to a couple of files and commits them on the branch:

```
ayalon$ emacs README.txt a-subdir/whatever.c b-subdir/random.c
...
ayalon$ cvs ci -m "print greeting backwards, etc"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v  <--  README.txt
new revision: 1.14.2.1; previous revision: 1.14
done
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v  <--  whatever.c
new revision: 1.3.2.1; previous revision: 1.3
done
Checking in b-subdir/random.c;
/usr/local/newrepos/myproj/b-subdir/random.c,v  <--  random.c
new revision: 1.1.1.1.2.1; previous revision: 1.1.1.1
done
ayalon$
```

Meanwhile, jrandom is continuing to work on the trunk. She modifies two of the three files that mosheb touched. Just to make things a bit more complicated, let's say she made changes that conflict with mosheb's work:

```
yarkon$ emacs README.txt whatever.c
 ...
yarkon$ cvs ci -m "some very stable changes indeed"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v  <--  README.txt
new revision: 1.15; previous revision: 1.14
done
Checking in a-subdir/whatever.c;
```

```
/usr/local/newrepos/myproj/a-subdir/whatever.c,v  <--  whatever.c
new revision: 1.4; previous revision: 1.3
done
yarkon$
```

The conflict is not apparent yet, of course, because neither developer has tried to merge branch and trunk. So far, everything is fine. Suddenly, jrandom does the merge:

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

Two of the files conflict. No big deal. Quickly, jrandom resolves the conflicts, commits, and tags the trunk as successfully merged:

```
yarkon$ emacs README.txt a-subdir/whatever.c
 ...
yarkon$ cvs ci -m "merged from Exotic_Greetings-branch (conflicts resolved)"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
```

```
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v  <--  README.txt
new revision: 1.16; previous revision: 1.15
done
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v  <--  whatever.c
new revision: 1.5; previous revision: 1.4
done
Checking in b-subdir/random.c;
/usr/local/newrepos/myproj/b-subdir/random.c,v  <--  random.c
new revision: 1.2; previous revision: 1.1
done
yarkon$ cvs tag merged-Exotic_Greetings
cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
cvs tag: Tagging b-subdir
T b-subdir/random.c
yarkon$
```

Meanwhile, mosheb needn't wait for the merge to finish before continuing development, as long as he makes a tag for the batch of changes from which jrandom merged (later, jrandom will need to know this tag name. Branches depend on frequent and thorough communication among developers):

```
ayalon$ cvs tag Exotic_Greetings-1
cvs tag: Tagging .
T README.txt
T foo.gif
T hello.c
cvs tag: Tagging a-subdir
T a-subdir/whatever.c
cvs tag: Tagging a-subdir/subsubdir
T a-subdir/subsubdir/fish.c
cvs tag: Tagging b-subdir
T b-subdir/random.c
ayalon$ emacs a-subdir/whatever.c
 ...
ayalon$ cvs ci -m "print a randomly capitalized greeting"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
```

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

And of course, mosheb should tag those changes once he's finished:

```
ayalon$ cvs -q tag Exotic_Greetings-2
T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
ayalon$
```

While all this is going on, jrandom makes a change in a different file, one that mosheb hasn't touched in his new batch of edits:

```
yarkon$ emacs README.txt
 ...
yarkon$ cvs ci -m "Mention new Exotic Greeting features" README.txt
Checking in README.txt;
/usr/local/newrepos/myproj/README.txt,v  <--  README.txt
new revision: 1.17; previous revision: 1.16
done
yarkon$
```

At this point, mosheb has committed a new change on the branch, and jrandom has committed a nonconflicting change in a different file on the trunk. Watch what happens when jrandom tries to merge from the branch again:

```
yarkon$ cvs -q update -j Exotic_Greetings-branch
RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
rcsmerge: warning: conflicts during merge
RCS file: /usr/local/newrepos/myproj/a-subdir/whatever.c,v
retrieving revision 1.3
retrieving revision 1.3.2.2
Merging differences between 1.3 and 1.3.2.2 into whatever.c
rcsmerge: warning: conflicts during merge
RCS file: /usr/local/newrepos/myproj/b-subdir/random.c,v
```

```
retrieving revision 1.1
retrieving revision 1.1.1.1.2.1
Merging differences between 1.1 and 1.1.1.1.2.1 into random.c
yarkon$ cvs -q update
C README.txt
C a-subdir/whatever.c
yarkon$
```

There are conflicts!

The problem lies in the semantics of merging. In Chapter 3, we explained that when you run

```
yarkon$ cvs update -j BRANCH
```

in a working copy, CVS merges into the working copy the differences between **BRANCH**'s root and its tip. The trouble with that behavior in this situation is that most of those changes had already been included into the trunk the first time that jrandom did a merge. When CVS tried to merge them in again (over themselves, as it were), it naturally registered a conflict.

What jrandom really wanted to do was merge into her working copy the changes between the branch's *most recent* merge and its current tip. You can do this by using two **-j** flags to update (as you might recall from Chapter 2) as long as you know what revision to specify with each flag. Fortunately, mosheb made a tag at exactly the last merge point (hurrah for planning ahead!), so this will be no problem. First, let's make jrandom restore her working copy to a clean state, from which she can redo the merge:

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

Now she's ready to do the merge, this time using mosheb's conveniently placed tag:

```
yarkon$ cvs -q update -j Exotic_Greetings-1 -j Exotic_Greetings-branch
RCS file: /usr/local/newrepos/myproj/a-subdir/whatever.c,v
retrieving revision 1.3.2.1
retrieving revision 1.3.2.2
Merging differences between 1.3.2.1 and 1.3.2.2 into whatever.c
yarkon$ cvs -q update
M a-subdir/whatever.c
yarkon$
```

Now, this is better! The change from mosheb has been included into whatever.c; jrandom can now commit and tag:

```
yarkon$ cvs -q ci -m "merged again from Exotic_Greetings (1)"
Checking in a-subdir/whatever.c;
/usr/local/newrepos/myproj/a-subdir/whatever.c,v  <--  whatever.c
new revision: 1.6; previous revision: 1.5
done
yarkon$ cvs -q tag merged-Exotic_Greetings-1
T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
yarkon$
```

Even if mosheb had forgotten to tag at the merge point, all hope would not be lost. If jrandom knew approximately when mosheb's first batch of changes had been committed, she would probably first try filtering by date:

```
yarkon$ cvs update -j Exotic_Greetings-branch:3pm -j Exotic_Greetings_branch
```

Although useful as a last resort, filtering by date is less than ideal because it selects the changes based on people's recollections rather than dependable developer designations. If mosheb's first mergeable set of changes had happened over several commits instead of in one commit, jrandom might mistakenly choose a date or time that catches some of the changes, but not all of them.

Note

There's no reason why each taggable point in mosheb's changes needs to be sent to the repository in a single commit—it just happens to have worked out that way in these examples. In real life, mosheb will make several commits between tags. He can work on the branch in isolation as he pleases. The point of the tags is to record successive points on the branch where he considers the changes to be mergeable into the trunk. As long as jrandom always merges using two ***-j*** *flags and is careful to use mosheb's merge tags in the right order and only once each, the trunk should never experience the double-merge problem. Conflicts might occur, but they will be the unavoidable kinds that require human resolution—situations in which both branch and trunk made changes to the same area of code.*

The Dovetail Approach: Merging in and out of the Trunk

Merging repeatedly from branch to trunk is good for the people on the trunk, because they see all of their own changes and all the changes from the branch. However, the developer on the branch never gets to incorporate any of the work being done on the trunk.

To allow that, the branch developer needs to add an extra step every now and then (mean-

ing whenever he feels like merging in recent trunk changes and dealing with the inevitable conflicts):

```
ayalon$ cvs update -j HEAD
```

The special reserved tag **HEAD** means the tip of the trunk. The preceding command merges in all of the trunk changes between the root of the current branch (**Exotic_Greetings-branch**) and the current highest revisions of each file on the trunk. Of course, mosheb should tag again after doing this, so that the trunk developers can avoid accidentally merging in their own changes when they're trying to get mosheb's.

The branch developer can likewise use the trunk's merge tags as boundaries, allowing the branch to merge exactly those trunk changes between the last merge and the trunk's current state (the same way the trunk handles merges). For example, suppose jrandom had made some changes to hello.c after merging from the branch:

```
yarkon$ emacs hello.c
 ...
yarkon$ cvs ci -m "clarify algorithm" hello.c
Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v  <--  hello.c
new revision: 1.22; previous revision: 1.21
done
yarkon$
```

Now, mosheb can merge those changes into his branch, commit, and, of course, tag:

```
ayalon$ cvs -q update -j merged-Exotic_Greetings-1 -j HEAD
RCS file: /usr/local/newrepos/myproj/hello.c,v
retrieving revision 1.21
retrieving revision 1.22
Merging differences between 1.21 and 1.22 into hello.c
ayalon$ cvs -q update
M hello.c
ayalon$ cvs -q ci -m "merged trunk, from merged-Exotic_Greetings-1 to HEAD"
Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v  <--  hello.c
new revision: 1.21.2.1; previous revision: 1.21
done
ayalon$ cvs -q tag merged-merged-Exotic_Greetings-1
T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
```

```
ayalon$
```

Notice that jrandom did not bother to tag after committing the changes to hello.c, but mosheb did. The principle at work here is that although you don't need to tag after every little change, you should always tag after a merge or after committing your line of development up to a mergeable state. That way, other people—perhaps on other branches—have a reference point against which to base their own merges.

The Flying Fish Approach: A Simpler Way

A simpler variant of the branch-and-trunk merging process involves the branch developers freezing while the trunk merges, and then the trunk developers creating an entirely new branch that replaces the old one. The branch developers move onto that branch and continue working. The cycle continues until there is no more need for branch development. It goes something like this (in shorthand—we'll assume jrandom@yarkon has the trunk and mosheb@ayalon has the branch, as usual):

```
yarkon$ cvs tag -b BRANCH-1
ayalon$ cvs checkout -r BRANCH-1 myproj
```

Trunk and branch both start working; eventually, the developers confer and decide it's time to merge the branch into the trunk:

```
ayalon$ cvs ci -m "committing all uncommitted changes"
yarkon$ cvs update -j BRANCH-1
```

All the changes from the branch merge in; the branch developers stop working while the trunk developers resolve any conflicts, commit, tag, and create a new branch:

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

Now the branch developers switch their working copies over to the new branch. They know they won't lose any uncommitted changes by doing so, because they were up to date when the merge happened, and the new branch is coming out of a trunk that has incorporated the changes from the old branch:

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

And the cycle continues in that way, indefinitely; just substitute **BRANCH-2** for **BRANCH-1** and **BRANCH-3** for **BRANCH-2**.

We call this the "Flying Fish" technique because the branch is constantly emerging from the

trunk, traveling a short distance, then rejoining it. The advantages of this approach are that it's simple (the trunk always merges in all the changes from a given branch) and the branch developers never need to resolve conflicts (they're simply handed a new, clean branch on which to work each time). The disadvantage, of course, is that the branch people must sit idle while the trunk is undergoing merge (which can take large amounts of time if there are many conflicts to be resolved). Another minor disadvantage is that it results in many little, unused branches lying around instead of many unused nonbranch tags. However, if having many obsolete branches doesn't bother you, and you anticipate fairly trouble-free merges, Flying Fish might be the easiest way to go in terms of mental bookkeeping.

Whichever way you do it, always try to keep the time between merges as short as possible. If the branch and the trunk go too long without merging, they could easily begin to suffer not only from textual drift, but also from semantic drift. Changes that conflict textually are the easiest ones to resolve. Changes that conflict conceptually, but not textually, often prove hardest to find and fix. The isolation of a branch, so freeing to the developers, is dangerous precisely because it shields each side from the effects of others' changes for a time. When you use branches, communication becomes more vital than ever: Everyone needs to make extra sure to review each others' plans and code to ensure that they're all staying on the same track.

Branches and Keyword Expansion: Natural Enemies

If your files contain RCS keywords that expand differently on branch and trunk, you're almost guaranteed to get spurious conflicts on every merge. Even if nothing else changes, the keywords are overlapping, and their expansions won't match. For example, if README.txt contains this on the trunk

```
$Revision: 1.14 $
```

and this on the branch

```
$Revision: 1.14.2.1 $
```

then when the merge is performed, you'll get the following conflict:

```
yarkon$ cvs update -j Exotic_Greetings-branch
RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
rcsmerge: warning: conflicts during merge
yarkon$ cat README.txt
 ...
<<<<<<< README.txt
```

```
key $Revision: 1.14 $
========
key $Revision: 1.14.2.1 $
>>>>>>> 1.14.2.1
 ...
yarkon$
```

To avoid this, you can temporarily disable expansion by passing the **-kk** option (nobody seems to remember what it stands for; "kill keywords," maybe?) when you do the merge:

```
yarkon$ cvs update -kk -j Exotic_Greetings-branch
RCS file: /usr/local/newrepos/myproj/README.txt,v
retrieving revision 1.14
retrieving revision 1.14.2.1
Merging differences between 1.14 and 1.14.2.1 into README.txt
yarkon$ cat README.txt
 ...
$Revision$
 ...
yarkon$
```

Be careful, however: If you use **-kk**, it overrides whatever other keyword expansion mode you might have set for that file. Specifically, this is a problem for binary files, which are normally **-kb** (which suppresses all keyword expansion and line-end conversion). So if you have to merge in binary files from a branch, don't use **-kk**. Just deal with the conflicts by hand instead.

Tracking Third-Party Sources: Vendor Branches

Sometimes a site will make local changes to a piece of software received from an outside source. If the outside source does not incorporate the local changes (and there might be many legitimate reasons why it can't), the site has to maintain its changes in each received upgrade of the software.

CVS can help with this task, via a feature known as *vendor branches*. In fact, vendor branches are the explanation for the puzzling (until now) final two arguments to **cvs import**: the **vendor** tag and **release** tag that we glossed over in Chapter 2.

Here's how it works. The initial import is just like any other initial import of a CVS project (except that you'll want to choose the **vendor** tag and **release** tag with a little care):

```
yarkon$ pwd
/home/jrandom/theirproj-1.0
yarkon$ cvs import -m "Import of TheirProj 1.0" theirproj Them THEIRPROJ_1_0
N theirproj/INSTALL
N theirproj/README
N theirproj/src/main.c
```

```
N theirproj/src/parse.c
N theirproj/src/digest.c
N theirproj/doc/random.c
N theirproj/doc/manual.txt

No conflicts created by this import

yarkon$
```

Then you check out a working copy somewhere, make your local modifications, and commit:

```
yarkon$ cvs -q co theirproj
U theirproj/INSTALL
U theirproj/README
U theirproj/doc/manual.txt
U theirproj/doc/random.c
U theirproj/src/digest.c
U theirproj/src/main.c
U theirproj/src/parse.c
yarkon$ cd theirproj
yarkon$ emacs src/main.c src/digest.c
 ...
yarkon$ cvs -q update
M src/digest.c
M src/main.c
yarkon$ cvs -q ci -m "changed digestion algorithm; added comment to main"
Checking in src/digest.c;
/usr/local/newrepos/theirproj/src/digest.c,v  <--  digest.c
new revision: 1.2; previous revision: 1.1
done
Checking in src/main.c;
/usr/local/newrepos/theirproj/src/main.c,v  <--  main.c
new revision: 1.2; previous revision: 1.1
done
yarkon$
```

A year later, the next version of the software arrives from Them, Inc., and you must incorporate your local changes into it. Their changes and yours overlap slightly. They've added one new file, modified a couple of files that you didn't touch, but also modified two files that you modified.

First you must run **import** again, this time from the new sources. Almost everything is the same as it was in the initial import—you're importing to the same project in the repository on the same vendor branch. The only thing different is the **release** tag:

```
yarkon$ pwd
/home/jrandom/theirproj-2.0
```

```
yarkon$ cvs -q import -m "Import of TheirProj 2.0" theirproj Them THEIRPROJ_2_0
U theirproj/INSTALL
N theirproj/TODO
U theirproj/README
cvs import: Importing /usr/local/newrepos/theirproj/src
C theirproj/src/main.c
U theirproj/src/parse.c
C theirproj/src/digest.c
cvs import: Importing /usr/local/newrepos/theirproj/doc
U theirproj/doc/random.c
U theirproj/doc/manual.txt

2 conflicts created by this import.
Use the following command to help the merge:

       cvs checkout -jThem:yesterday -jThem theirproj

yarkon$
```

Wow—we've never seen CVS try to be so helpful. It's actually telling us what command to run to merge the changes. And it's almost right, too! Actually, the command as given works (assuming that you adjust **yesterday** to be any time interval that definitely includes the first import but not the second), but we prefer to do it by **release** tag instead:

```
yarkon$ cvs checkout -j THEIRPROJ_1_0 -j THEIRPROJ_2_0 theirproj
cvs checkout: Updating theirproj
U theirproj/INSTALL
U theirproj/README
U theirproj/TODO
cvs checkout: Updating theirproj/doc
U theirproj/doc/manual.txt
U theirproj/doc/random.c
cvs checkout: Updating theirproj/src
U theirproj/src/digest.c
RCS file: /usr/local/newrepos/theirproj/src/digest.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into digest.c
rcsmerge: warning: conflicts during merge
U theirproj/src/main.c
RCS file: /usr/local/newrepos/theirproj/src/main.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into main.c
U theirproj/src/parse.c
yarkon$
```

Notice how using the **import** command told us that there were two conflicts, but using the **merge** command tells us there is only one conflict. It seems that CVS's idea of a conflict is a little different when importing than at other times. Basically, **import** reports a conflict if both you and the vendor modified a file between the last import and this one. However, when it comes time to merge, **update** sticks with the usual definition of conflict—overlapping changes. Changes that don't overlap are merged in the usual way, and the file is simply marked as modified.

A quick running of **diff** verifies that only one of the files actually has conflict markers:

```
yarkon$ cvs -q update
C src/digest.c
M src/main.c
yarkon$ cvs diff -c
Index: src/digest.c
===================================================================
RCS file: /usr/local/newrepos/theirproj/src/digest.c,v
retrieving revision 1.2
diff -c -r1.2 digest.c
*** src/digest.c        2001/07/26 08:02:18     1.2
-- src/digest.c        2001/07/26 08:16:15
***************
*** 3,7 ****
-- 3,11 ----
 void
 digest ()
 {
+ <<<<<<< digest.c
   printf ("gurgle, slorp\n");
+ =======
+   printf ("mild gurgle\n");
+ >>>>>>> 1.1.1.2
 }
Index: src/main.c
===================================================================
RCS file: /usr/local/newrepos/theirproj/src/main.c,v
retrieving revision 1.2
diff -c -r1.2 main.c
*** src/main.c  2001/07/26 08:02:18     1.2
-- src/main.c  2001/07/26 08:16:15
***************
*** 7,9 ****
-- 7,11 ----
 {
   printf ("Goodbye, world!\n");
```

```
 }
+
+ /* I, the vendor, added this comment for no good reason. */
yarkon$
```

From here, it's just a matter of resolving the conflicts as with any other merge:

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

Then commit the changes

```
yarkon$ cvs -q ci -m "Resolved conflicts with import of 2.0"
Checking in src/digest.c;
/usr/local/newrepos/theirproj/src/digest.c,v  <--  digest.c
new revision: 1.3; previous revision: 1.2
done
Checking in src/main.c;
/usr/local/newrepos/theirproj/src/main.c,v  <--  main.c
new revision: 1.3; previous revision: 1.2
done
yarkon$
```

and wait for the next release from the vendor. (Of course, you'll also want to test that your local modifications still work!)

New CVS Features

Since first edition of this book was published, quite a few features and bug fixes have found their way to CVS. In this section, we will run through important new features and how to use them.

One of the new commands in the recent 1.11.1 version likely to find lots of fans is **cvs rlog**. Its pendant is **cvs rannotate**. Both commands have been added to allow you to obtain log messages and annotations without having to have a checked-out copy of the file. The **cvs list** command is very useful, but you still can't see all the information that might be useful; therefore, the CVS developers created **cvs rlog** (replacing the old deprecated synonym for log), which displays log entries without a working directory. Doubtless a flame war will now ensue over whether this is an appropriate name. (We think it is, because it goes with the convention already established for **rtag**; on the other hand, **rlog** used to mean something slightly different, but the warning against its usage has been around since at least version 1.10, so no one should be using **rlog** now.)

Also, the ~/.cvspass file has a slightly modified format. CVSROOTs are now stored in a new canonical form—hostnames are now case insensitive and port numbers are always stored in the new format. Until a new login for a particular CVSROOT is performed with the new version of CVS, new and old versions of CVS should interoperate transparently. After that time, an extra login using the old version of CVS might be necessary to allow the new and old versions of CVS to interoperate using the same ~/.cvspass file and CVSROOT. The exception to this rule occurs when the CVSROOTs used with the different versions use case-insensitively different hostnames.

In version 1.10 and higher, a password and a port number can be specified for **pserver** connections in CVSROOT. The new format is:

```
:pserver:[[user][:password]@]host[:[port]]/path
```

Note that passwords specified in a checkout command will be saved in the clear in the CVS/Root file in each created directory, so this is not recommended, except, perhaps, when accessing anonymous repositories and the like.

On the positive side, anonymous read-only access can now be done without requiring a password. On the server side, simply give that user (presumably "anonymous") an empty password in the CVSROOT/passwd file, and then any received password will authenticate successfully.

Starting with version 1.10, it is possible for a single CVS command to recurse into several CVS roots. This includes roots that are located on several servers or that are both remote and local. CVS will make connections to as many servers as necessary.

You Are Now a Guru!

If you read and understood (and better yet, experimented with) everything in this chapter, you can rest assured that there are no big surprises left for you in CVS—at least until someone adds a major new feature, which does happen with some frequency. Everything you need to know to use CVS on a major project has been presented.

Before that goes to your head, let us reiterate the suggestion, first made in Chapter 3, that you subscribe to the **info-cvs@gnu.org** mailing list. Despite having the impoverished signal-to-noise ratio common to most Internet mailing lists, the bits of signal that do come through are almost always worth the wait.