Extracted from:

Pragmatic Version Control

with CVS

Chapter 10

Third-Party Code

All projects rely to some extent on external libraries: C programs use the `libc` library, Java programs use `rt.jar`, and so on. Should these libraries form part of your personal workspace?

To answer that question, ask yourself another. You need to be able to rebuild a working program at some arbitrary time in the future. Will you be able to use the versions of these libraries that will be available then?

If you feel comfortable that the libraries used by your code will be available (and compatible) over the life of your application, then there's no need to do anything special with them; just use them as installed on your machine.

Looking beyond standard language facilities, many projects include other, less stable, libraries in their projects. For example, many Java developers will use the JUnit framework[1] to test their code. Compared to the standard libraries, these frameworks are fairly volatile (as of June 2003, JUnit is already up to version 3.8). Although the changes between versions are mostly compatible, there can be changes that affect your application.[2] As a result, we'd recommend that you

[1] `http://www.junit.org`

[2] For example, we've seen interactions between the Ant build tool and various revisions of JUnit.

include these libraries in your workspace, and (by extension) in your project's repositories.

Having made the decision that you want to include a third party library in your workspace and repository, you now have to decide what to include and where to put it.

The first decision is what files to include. This is relatively easy. If you use the library in the form distributed by the maker, and you feel confident that the library will continue to work unmodified through the life of the application, then storing the binary form of the library is all that is needed. We suggest putting all these libraries in subdirectories of a top-level `vendor/` directory. If the library is architecture-independent (for example a Java `.jar` file), then it can simply sit in a subdirectory called `lib/`. If instead you have libraries that depend on the target architecture (and assuming your application is targeted at more than one architecture) you'll need to have subdirectories below `vendor/` for each architecture and operating system combination. A common naming scheme for these subdirectories is to use *arch-os* where *arch* is the target architecture (i586 for an Intel Pentium, ppc for a PowerPC, and so on) and *os* is the operating system (linux, win2k, and so on). Always remember to use the `-kb` flag when importing or adding a binary file (such as a .DLL (dynamic link library) or other library) to CVS.

Languages such as C and C++ require that you include source header files in application code that uses a particular library. These header files are supplied with the library, and should also be stored in the workspace and repository. We suggest storing them in an `include/` subdirectory beneath `vendor`. Structure the subdirectories of `vendor/include/` in such a way that the compilers can find the libraries' include files naturally. As an example, consider a C library called `datetime` which performs date and time calculations. It comes with a binary library archive, `libdatetime.a`, and two header files, `datetime.h` and `extras.h`. The `datetime.h` header library is intended to be installed at the top level of the include hierarchy, while `extras.h` is expected to be in a subdirectory called `dt/`. That is, a program that used both header files would normally start:

```
proj/                    ← top level of project
└── vendor/
    ├── lib/
    │   └── libdatetime.a
    └── include/
        ├── datetime.h
        └── dt/
            └── extras.h
```

Figure 10.1: SAMPLE REPOSITORY WITH THIRD PARTY LIBRARY

```
#include <datetime>
#include <dt/extras>
// . . .
```

In this case, we'd organize our repository (and our workspace) as shown in Figure 10.1.

Integrating with the Build Environment

If you include vendor libraries or header files in your workspace, you'll need to make sure that your compilers, linkers, and IDEs can get to them. There's a minor problem: you need to make sure that you don't check anything in to the repository that contains absolute path names (as this might not work on some other developer's machine). Instead, you have a couple of options:

1. Arrange your build tools so that all path names are relative to (say) the top level project directory. This is workable if you're using an external build tool such as "make" or "ant," but it can get tricky.

2. Set up some external environment variable to point to the top of the project tree, and make all references in the build relative to this variable. This allows each developer to have different values in the external variable, but then to share a common build environment layout.

   The external variable need not be a true operating system environment variable. The Eclipse IDE, for example, allows each user to set internal variables, and then

to have a common shared build structure that references these variables. This means that all developers can share a common Eclipse build definition, but that developers can still install the source in different locations.

We recommend the second approach.

10.1 Libraries With Source Code

Sometimes a library comes with source code (or is distributed only as source code). If you have both source and binary versions of the library available, which should you store in the repository, and how should you set up your workspace?

The answer is an exercise in risk management. Having the source available means that you are always in the position (technically, at least) to fix bugs and add features, something you can't do with a binary library. This is clearly a good thing. At the same time, including the source code for all the libraries used by your project can slow down builds and complicate the structure of your project. It also gives future maintainers a headache. If there's a bug, do they need to consider potential changes to the library source, or can they concentrate on the code written by your organization?

Our recommendation is to add vendor source to your repository, but to treat it specially. To do this, you have to do a bit of role-playing.

Imagine for a minute that you are the writer of this particular library, and that every now and then you release an updated version of the code to your user base. Being a high-quality library writer, you naturally put all your source in a version control system, and practice all the necessary release control procedures.

Now come back from the role-play (remember, breathe in, breathe out, breathe in, breathe out). In an ideal world, we should be able to hook straight in to our vendor's repository and extract releases directly from there. But we can't, so we have to do the work ourselves. Whenever we receive code, bug fixes, and new releases from a vendor, we have to pretend that

we had generated the code, and handle it in our version control system as if we were the vendor handling it in theirs. This turns out to be simpler than it sounds.

Importing the Initial Source

When we first receive the source code for a third-party library, we need to import it into our repository. We recommend keeping this code separate from the code of your project. If you anticipate importing code from multiple sources over time, it probably makes sense to keep it all under a common top-level directory; we suggest calling it `vendorsrc/` (to differentiate it from `vendor/`, which contains libraries and header files).

To make this more concrete, let's assume that we've decided to use version 4.3 of the GNU readline library in our project (after checking the license terms, of course).

We start by downloading the latest sources from the GNU ftp site. We'll store this in a temporary directory.

```
~>  cd tmp
tmp>  ftp ftp.gnu.org
Connected to ftp.gnu.org.
Name (ftp.gnu.org:dave): ftp
331 Please specify the password.
Password:
230 Login successful. Have fun.
Using binary mode to transfer files.
ftp> cd pub/gnu/readline
250 Directory successfully changed.
ftp> get readline-4.3.tar.gz
local: readline-4.3.tar.gz remote: readline-4.3.tar.gz
961662 bytes received in 00:06 (136.48 KB/s)
ftp> bye
221 Goodbye.
```

We then unpack the archive. This creates a source tree in a subdirectory (which we know from experience will be called `readline-4.3`). We make this our current working directory.

```
tmp>  tar zxf readline-4.3.tar.gz
tmp>  cd readline-4.3
```

We are now in a position to import this source into our repository. We'll store it in the repository under `vendorsrc/fsf/readline`. (Remember, all our third-party code is stored under `vendorsrc/`. In this case, the vendor is the Free Software Foundation, and the "product" is readline.)

```
tmp/readline-4.3>  cvs import -ko -I! -m "load 4.3" \
                       vendorsrc/fsf/readline FSF_RL RL_4_3
N import/aclocal.m4
N import/ansi_stdlib.h
N import/bind.c
N import/callback.c
   :     :
N import/support/shobj-conf
N import/support/wcwidth.c

No conflicts created by this import
```

That's quite a command: we break it down in Figure 10.2 on the next page. The `-ko` flag is important, but subtle. Normally, CVS will expand special keywords (such as `$Author$`) in each of the files it manages. This lets you add annotations to the files. (This isn't a practice we encourage, so we haven't shown it so far in this book.) The problem is that the keywords are expanded every time the file is checked out. If the vendor also uses CVS, and if the vendor has used these tags, then the source you receive will have the vendor's information in these fields. However, if you just import these files as they stand and check them back out, CVS will update the tags, and suddenly your name will appear in the author field. While this may be vaguely satisfying, it will cause problems later when you come to merge in changes with the next vendor release. CVS will notice that these tag lines have changed, and you'll get conflicts when merging with the vendor's code. Specifying the `-ko` option turns off tag expansion for all files in the import, so you won't see this problem.

`-ko` ⇒ *Keywords Off*

The `-I!` is equally subtle; it tells CVS not to ignore any files while importing. When you're working with your own directories, you'll probably want CVS to bypass processing of backup files and the like, but with vendor-supplied files, you're going to want to load everything into the repository.

`-I!` ⇒ *Ignore Nothing!*

The *vendor tag* gives us a way to name the product we're importing. In this case, all the code for readline can be referenced using the tag `FSF_RL`. The *release tag* specifies the code that makes up this particular release. If the FSF comes up with version 4.4 of readline, we'll check it in with a different release tag. This means that we'll always be able to get back to the 4.3 release using the original `RL_4_3` tag.

Having imported this code into the repository, we can delete the temporary directory that we used.

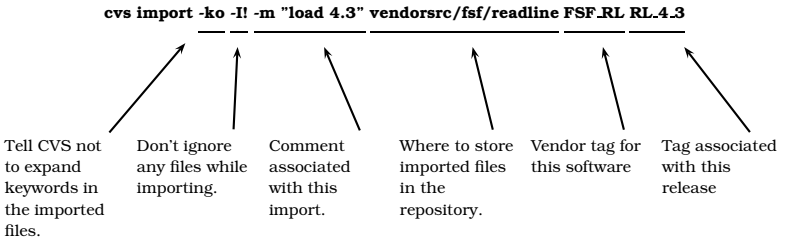

Figure 10.2: A CVS "IMPORT" COMMAND

Importing New Vendor Releases

When a vendor releases a new version of their software, you might want to incorporate it into your repository.[3] Assuming that you haven't made any local changes to the vendor's source code, then this is easy; simply import it again, following the same steps as above:

1. Download the new source, and unpack it in to a temporary directory.
2. Issue a CVS import command, using the same repository location and vendor tag, but with an updated release tag.

For example, if the FSF released readline version 4.4, we could do:

```
tmp>  tar zxf readline-4.4.tar.gz
tmp>  cd readline-4.4
tmp/readline-4.4>  cvs import -m -ko -I! "load 4.4" \
                       vendorsrc/fsf/readline FSF_RL RL_4_4
N import/aclocal.m4
N import/ansi_stdlib.h
N import/bind.c
N import/callback.c
N import/chardefs.h
N import/compat.c
   :     :
N import/support/shobj-conf
N import/support/wcwidth.c

No conflicts created by this import
```

[3] Many teams make the mistake of constantly chasing the latest and greatest vendor releases. This isn't always prudent. If the features added at a particular release don't enhance your application, is it worth the risk of incorporating new code? Sometimes skipping minor releases and only merging major changes is a better idea.

10.2 Modifying Third-Party Code

Sometimes the reason for importing third-party source code is to allow your team to make changes. You may need to add some application-specific functionality, or you might have local bug fixes that you need to apply.

Clearly the ideal solution would be to supply this changed code back to the third party and let them incorporate it into their own copy. That way when they send you the next release, their code will incorporate your changes, and life will be wonderful.

However, that isn't always possible. In these cases, we need to maintain our local changes and (ideally) have them automatically roll forward from each vendor release to the next.

Fortunately for us, CVS makes this relatively easy. In the background, the import mechanism is actually building and managing a simple release tree. It works like this.

When you first import code into CVS, it creates a mainline, and then immediately creates a branch (numbered 1.1.1). It then places the code that you import into this branch (so the first source files will have a revision number of 1.1.1.1). Although this sounds complicated, it's really no different to the description we had of a simple release structure back on page 18. And that isn't a coincidence; behind the scenes CVS is handing these imports as if you were the vendor performing releases. The vendor tag that you give the import command turns out to be the tag given to the release branch, and the release tags given on each import identify the points on that branch where each individual release's code sits. This is illustrated in Figure 10.3 on the next page.

If you check out vendor code, you'll be checking out of the release branch (the branch labeled with the vendor tag). You can verify this; doing a `cvs status` on a file will show a revision number with four levels (so the first revision will be 1.1.1.1). However, there's some magic here. If you edit vendor code and check it back in, CVS will place your changes in the mainline, but the revision number will be 1.2, not 1.1.1.2. CVS reserves the code in the vendor branch for vendor code.

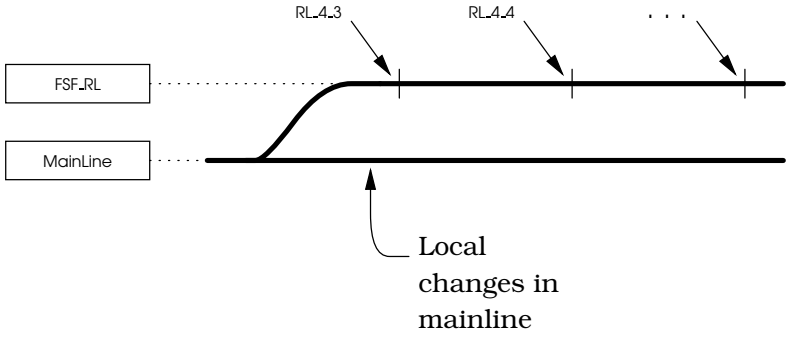

Figure 10.3: IMPORTED THIRD PARTY CODE. CODE IS IMPORTED IN TO A RELEASE BRANCH, LABELED BY THE VENDOR TAG. EACH IMPORT GENERATES A NEW RELEASE TAG IN THAT BRANCH. LOCAL WORK AUTOMATICALLY TAKES PLACE IN THE MAINLINE.

What happens if you edit third-party code, and then a new release comes along? Let's find out. To do this, we'll set up a dummy repository. We'll then pretend to be a vendor (called Acme) and create a couple of simple files. With our project team hat back on, we'll then import these, and check them out into our workspace. We'll then make a change and check it in.

Back in the vendor directory we'll prepare an updated release. We'll then try to import it, and we'll work out how to merge the vendor changes with our own.

Because all this role playing can get confusing, once we get started we'll show the full path of the current directory at the start of each of the sequences of commands. In the prompts themselves, we'll just show the directory name. In general, when we're playing vendor we'll be in the directory:

```
tmp/3rdparty/Acme
```

When we're a client dealing with checked-out vendor files we'll be in the directory:

```
tmp/3rdparty/work/vendorsrc/Acme
```

Step 1: Set up the Repository

We'll do all our work in a directory called `3rdparty`; this will let us clean everything up at the end. The repository goes in a subdirectory called `repository`.

```
# In directory tmp
tmp>  mkdir 3rdparty
tmp>  cd 3rdparty
tmp/3rdparty>  export CVSROOT=~/tmp/3rdparty/repository
tmp/3rdparty>  cvs init
tmp/3rdparty>  ls    # use 'dir' under Windows
repository
```

Step 2: Create the Third-Party Code

We'll create a directory called Acme that contains the third-party code. This directory will be the one we import into CVS. We'll use an editor to create two files, `Color.txt` and `Number.txt` using our favorite editor.

```
# in directory tmp/3rdparty
tmp/3rdparty>  mkdir Acme
tmp/3rdparty>  cd Acme
```

edit files, giving. . .

File `Color.txt`:

```
black
brown
red
orange
yellow
green
```

File `Number.txt`:

```
zero
one
two
three
four
```

Step 3: Import the Vendor Code

We've finished playing vendor for a minute. Now we'll pretend that we've received this code from the vendor and import it in to the repository, storing it in `vendorsrc/Acme`.

```
# In directory tmp/3rdparty/Acme
Acme>  cvs -ko import -m "load" vendorsrc/Acme Acme REL_1_0
N vendorsrc/Acme/Color.txt
N vendorsrc/Acme/Number.txt

No conflicts created by this import
```

Step 4: Set Up The Workspace

We'll now create a workspace and check out this vendor code there.

```
# In directory tmp/3rdparty/Acme
Acme>  cd ..
tmp/3rdparty>  mkdir work
tmp/3rdparty>  cd work
tmp/3rdparty/work>  cvs co vendorsrc/Acme
cvs checkout: Updating vendorsrc/Acme
U vendorsrc/Acme/Color.txt
U vendorsrc/Acme/Number.txt
```

Step 5: Modify The Vendor Code

Part way through our project, we discover a problem in the vendor code; their numbers file uses "zero," but our project standards call for "naught." The vendor ignores our pleas for a change, claiming we are their only customer to use Middle-English numbering (can that be?). So we bite the bullet and make the change ourselves. We edit the file in our workspace, then check it back in.

```
# In directory tmp/3rdparty/work
work>  cd vendorsrc/Acme
Acme>  # ... edit file ...
Acme>  cvs commit -m "Zero becomes naught"
cvs commit: Examining .
Checking in Number.txt;
.../repository/vendorsrc/Acme/Number.txt,v  <--  Number.txt
new revision: 1.2; previous revision: 1.1
done
```

Step 6: The Vendor Makes a Change

Meanwhile, back at Acme Corp, they decide to produce V1.1 of the product. As part of the added value in this new release, they're adding three new numbers to their numbers file. We'll simulate this by going back to our Acme directory (the one at the top level) and editing the file.

```
# In directory tmp/3rdparty/work/vendorsrc/Acme
Acme>  cd ../../../Acme
tmp/3rdparty/Acme>  # ... edit file ...
```

After the edit, the new numbers file contains:

File `Number.txt`:

```
zero
one
two
three
four
five
six
seven
```

This file still has "zero" in it; remember that Acme did not make the change to "naught." That's only in our local copy.

Step 7: Import the New Revision

Acme sends us the new revision, so with our client hats on we import it into CVS.

```
# In directory tmp/3rdparty/Acme
Acme>  cvs import -ko -I! -m "update" vendorsrc/Acme Acme REL_1_1
U vendorsrc/Acme/Color.txt
C vendorsrc/Acme/Number.txt

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

        cvs checkout -jAcme:yesterday -jAcme vendorsrc/Acme
```

CVS was smart enough to recognize that this import was actually updating existing files. The `Color.txt` file updated successfully (in fact it is unchanged) but the `Number.txt` file has a potential conflict; it has been changed by us (as the client) and also by the vendor. CVS was nice enough to suggest the command we could use to fix the situation. Normally, this command would work fine. Unfortunately it won't work for us. To see why, let's look at the command in more detail.

As we saw in Chapter 7.2, the `-j` option is used to merge in changes during checkout or update. In this case we're using two `-j` options. The first option, `-jAcme:yesterday`, tells CVS to look at the Acme branch as it was yesterday, before (in theory) we imported the latest release. The second, `-jAcme` says look at it as it is now. The two together ask CVS to compute the difference; this difference is the changes that the vendor made. These changes are then applied to the current head of our mainline. The net result of all this is that the vendor's changes are used to update our local copy.

Although this incantation would normally work (because few vendors produce more than one release per day), it doesn't work too well in our example, as we didn't even have any vendor code yesterday. Instead, we'll use an alternate form of the `-j` option, which allows us to merge based on release tags.

To do this, change back to our workspace, and issue the following command.

```
# In directory tmp/3rdparty/Acme
Acme>  cd ../work
work>  cvs co -jREL_1_0 -jREL_1_1 vendorsrc/Acme
cvs checkout: Updating vendorsrc/Acme
RCS file: /Users/dave/tmp/3rdparty/repository/vendorsrc/Acme/Number.txt,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 Number.txt
```

Remember that we gave the first import the revision tag of `REL_1_0` and the second the tag `REL_1_1`. This lets us tell CVS to apply the differences between these two releases to our current mainline code. The result can be seen in the tracing that follows the command: CVS merges the vendor's changes in to our local file. Let's look at it and confirm that we have the vendor's three additional numbers, and that our "naught" has not been changed.

```
# In directory tmp/3rdparty/work
work>  cd vendorsrc/Acme
Acme>  cvs status Number.txt
===================================================
File: Number.txt        Status: Locally Modified

   Working revision:    1.2     Result of merge
   Repository revision: 1.2     /Users/dave/tmp/3rdparty/repos...
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

We can also check the file contents.

File `Number.txt`:

```
naught
one
two
three
four
five
six
seven
```

Had there been conflicts between the vendor code and our changes, we'd have seen the normal conflict markers in the file.

Step 8: Save the Merged File

Now that we've merged the changes (and run the tests to confirm the system still works) we can check everything back in to the repository.

```
# In directory tmp/3rdparty/work/vendorsrc/Acme
Acme>  cvs commit -m "Merged 1.1 changes"
cvs commit: Examining .
Checking in Number.txt;
/Users/.../vendorsrc/Acme/Number.txt,v  <--  Number.txt
new revision: 1.3; previous revision: 1.2
done
```

Summary: Modifying Third-Party Code

Managing vendor releases using these simple steps is both straightforward and powerful. CVS automatically maintains a release branch that contains the unmodified code from the vendor, tagged at each release. Our mainline in the repository contains the same code, but with all our local changes. Using the `-j` options allows us to merge the vendor's changes at each release into our local version of their code.

To summarize, the steps are:

- Import the vendor code:

```
cvs import -ko -I! -m "load" \
           vendor_module vendor release_tag
```

- Check out vendor code into a local workspace:

```
cd work
cvs co vendor_module
```

- Make local changes to vendor code and check back in:

```
cvs commit -m "summary of changes"
```

- If the vendor issues a new release, import it into the vendor branch:

```
cvs import -ko -I! -m "update" \
           vendor_module vendor release_tag
```

- Fix conflicts between vendor changes and our changes:

```
cvs co -jrelease_1 -jrelease_2 vendor_module
```

- Save the changes back:

```
cvs commit -m "summary of changes"
```