DAVID N. BLANK-EDELMAN

practical Perl tools: polymorphously versioned

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration* (with a second edition due out very soon). He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

TODAY WE'RE GOING TO TALK ABOUT how Perl can improve life in the land of the version control system (VCS). I think you would be hard pressed to find anyone doing a serious amount of programming these days who doesn't use a VCS of some sort. But for those of you who are new to the biz, let me spend a paragraph or two bringing you up to speed.

The basic idea behind any VCS is that there is value in tracking all copies of the data, even and especially the intermediate copies, that make up some project. By tracking I mean "recording who worked on what piece of data, when, and what precisely they did." If this can be done well, it does wonders toward coordinating work being done by a number of people working on the same project so that the end result is congruent.

On top of this, two immensely helpful side effects emerge:

1. You can determine who did what to something and when (especially crucial for debugging).
2. You can revert to a previous working version when a new version causes something to break or recover when something was deleted in error.

The obvious thing for you to do is to keep all of the Perl code you write under some sort of version control system, but that's not what we're going to talk about in this column. If that was the only message of this column, I could simply say "do it" and then let you get back to flossing the cat or whatever you had planned to do today. Instead, we're going to look at how to use Perl to automate and augment several VCS packages. The particular VCS packages we're going to use in this column are those I'm most familiar with: RCS, Subversion, and Git. I'm going to assume you already have a little familiarity with them when I write about them in this column.

Note: This selection is not meant as a slight to any of the other interesting VCS packages, of which there are many (see http://en.wikipedia.org/wiki/Comparison_of_revision_control_software for a comparison). Rabid users of Mercurial, Bazaar, DARCS, Monotone, etc., are more than welcome to write me to let me know how much I'm suffering on a daily basis by not using their favorite system. (I'm always looking for new cool tools.) I'd also be remiss if I didn't mention the one Perl-based VCS, SVK (http://svk.bestpractical.com/view/HomePage). I have not used it, but it looks as though it has some very impressive distributed VCS features.

Automating the Revision Control System (RCS)

If you have only used the newer VCS packages, the notion of using RCS might seem a bit anachronistic. This may be like using a cotton gin for system administration but I assure you that RCS still has its place. Unlike the other systems we're going to look at, it has a few properties that come in handy in the right situations:

1. RCS is lightweight and fairly portable (there are RCS ports for most operating systems).
2. RCS is *file*-based (unlike SVN, which is directory-based, and git, which is content-based). This works in your favor if you just need to keep a file or two under version control.
3. RCS largely assumes strict locking will be used. Sometimes it makes no sense to have two people be able to work on the same file at once and have their changes merged in the end. System configuration files are one such example where there's good reason to serialize access.
4. Files under RCS almost always live right in the same place they are archived versus some nebulous and nonspecific "working directory." (/etc/services has to be in /etc; it does you no good if it lives just in a working directory somewhere else on the system.)

RCS is a good place to start our Perl discussion because it offers a simple example of the basic modus operandi we're going to see for each of these systems. To work with RCS from Perl, you use a Perl module called (surprise) Rcs.

First we load the module and let it know where it should expect to find the RCS binaries installed on your system. Most of the VCS modules are actually wrappers around your existing VCS binaries. Although that may be a little wasteful in terms of resources (e.g., Perl has to spin up some other program), this is more than made up for by the portability it provides. The module author does not have to maintain glue code to some C library that could change each time a new version is released or distribute libraries with the module that will also stale quickly.

Here's the start of all of our RCS code:

```
use Rcs;

Rcs->bindir('/usr/bin');
```

Once we've got that set up, we ask for a new Rcs object through which we'll perform all of our Rcs operations. To do anything we first have to tell the object just what file we're going to manipulate:

```
my $rcs = Rcs->new;
$rcs->file('/etc/services');
```

At this point we can start issuing Rcs commands using object methods named after the commands. Let's say we wanted to check a file out for use, modify it, and then check it back in again. That code would look like this:

```
$rcs->co('-l'); # check it out locked

# do something to the file
...

$rcs->ci('-u',  # check it back in, but leave unlocked version in situ
  '-m'
     . 'Modified by '
     . ( getpwuid($<) )[6] .   ' ('
     . ( getpwuid($<) )[0] .   ') on '
     . scalar localtime );
```

The last line of this code is in some ways the most interesting. For version control to be really useful, it is important to provide some sort of information each time a change is written back to its archive. At the bare minimum, I recommend making sure you log who made the change and when. Here's what the log messages might look like if you used this code:

```
revision 1.5
date: 2009/05/19 23:34:16;  author: dnb;  state: Exp;  lines: +1 -1
Modified by David N. Blank-Edelman (dnb) on Tue May 19 19:34:16 2009
----------------------------
revision 1.4
date: 2009/05/19 23:34:05;  author: eviltwin;  state: Exp;  lines: +1 -1
Modified by Divad Knalb-Namlede (eviltwin) on Tue May 19 19:34:05 2009
----------------------------
revision 1.3
date: 2009/05/19 23:33:35;  author: dnb;  state: Exp;  lines: +20 -0
Modified by David N. Blank-Edelman (dnb) on Tue May 19 19:33:16 2009
```

Eagle-eyed readers might note that the VCS itself should be recording the user and date automatically. That's true, except (a) sometimes your code runs as another user (e.g., root) and you want the uid and not the effective uid logged, and (b) the VCS records the time it hit the archive, but more interesting is probably the time the change was made. If your code takes a while before it gets to the part where it performs the VCS operation, the time information you care about might not get recorded.

This is the very basics for RCS use. The Rcs module also has methods such as revisions()/dates() to provide the list of revisions of a file and rcsdiff() to return the difference between two revisions. With methods like this you could imagine writing code that would analyze an RCS archive and provide information about how a file has changed over time. If you ever wanted to be able to search for a string found any time in a file's history, it would be fairly easy to write code to do that, thanks to this module.

Automating Subversion (SVN)

There are a few modules that allow you to operate on Subversion repositories in a similar fashion to the one we just saw (i.e., using an external program to automate operations versus calling the SVN libraries directly). The two I'd recommend you consider using are SVN::Agent and SVN::Class. I'm going to show you one example from each because they both have their strengths. SVN::Agent is the simpler of the two:

```
use SVN::Agent;

# SVN::Agent looks for the svn binaries in your path
$ENV{PATH} = '/path/to/svnbins' . ':' . $ENV{PATH};

# this assumes we've already got a working dir with files in it,
# if not, we could use the checkout() method
my $svn = SVN::Agent->load({ path => '/path/to/working_dir' });

$svn->update;  # update working dir with latest info in repos

print "These are the files that are modified:\n";
print join("\n",@{$svn->modified});

$svn->add('services')';  # add the file services to the changes list

$svn->prepare_changes;

$svn->commit('Files checked in by'
```

```
    . ( getpwuid($<) )[6] .  ' ('
    . ( getpwuid($<) )[0] .  ') on '
    . scalar localtime );
```

Most of that code should be fairly straightforward. The one line that is less than obvious is the call to prepare_changes. SVN::Agent keeps separate lists of the modified, added, deleted, etc., files in the working directory. When we said `$svn->add('services')` we added the services file to the added list. To give you the flexibility to choose which items should be committed to the repository, SVN::Agent keeps a separate changes list of files and dirs to be committed. This list starts out empty. The prepare_changes method copies the other lists (added, modified, etc.) in the change list so that the commit() method can do its stuff.

The second SVN module, SVN::Class, is interesting because it is essentially an extension of the excellent Path::Class module. Path::Class is a worthy replacement for the venerable File::Spec module. I'm sure we'll see it again in later columns, but, briefly, it provides an OS-independent, object-oriented way to work with file/directory names and filesystem paths. SVN::Class extends it by adding on the same sort of methods you'd expect in an SVN module [e.g., add(), commit(), delete()]. If you are using Path::Class in your programming, this will lend a consistent feel to your programs. Here's a very simple example:

```
use SVN::Class;

my $svnfile = svn_file('services');

$svnfile->svn('/usr/bin/svn'); # explicitly set location of svn command

$svnfile->add;

# ... perform some operation on that file, perhaps using the Path::Class
#       open() method

my $revision = $svnfile->commit('File checked in by'
      . ( getpwuid($<) )[6] .  ' ('
      . ( getpwuid($<) )[0] .  ') on '
      . scalar localtime );

die "Unable to commit file 'services':" . $svnfile->errstr
  unless $revision;
```

SVN::Path does not have the same sort of interface to collectively commit() items as SVN::Agent, which may or may not be a plus in your eyes. It could be argued that an interface that forces you to actively call a commit() object for every file or directory makes for clearer code (versus using some backend data structure). However, SVN::Class does have some methods for querying the objects it uses (e.g., repository information, author of a file). I'd recommend picking the module that suits your style and the particular task.

Automating Git

For the last peek at automating a VCS from Perl we're going to look at Git, the wunderkind that has been storming the open source world. In fact, Perl development itself is now conducted using Git (and for a fun geek story, read about the transition at http://use.perl.org/articles/08/12/22/0830205.shtml).

Driving Git from Perl via Git::Wrapper is as simple as using it from the command line. You start in a similar fashion as the other wrappers we've seen:

```
use Git::Wrapper;

my $git = Git::Wrapper->new('/path/to/your/repos');
```

From this point on the $git object offers methods with the exact same name of each of the standard Git commands. If you look at the code of the module itself, you'll see that it has virtually no internal knowledge of how Git works. This means you have to understand Git's commands and semantics really well, because you'll get virtually no help from the module. That's a plus if you think Git does things perfectly and the module should get out of the way, but a minus if you were hoping for some (syntactic) sugar-coated methods to make your life easier.

Probably the best way to learn this module is to first get a good handle on Git itself from either the official doc [1] or a good book such as *Pragmatic Version Control Using Git* [2].

Automate All (Many) of Them

If you spotted a certain commonality among these modules (or perhaps repetition in my description of them), you are not alone. Max Kanat-Alexander decided to see if he could take a lesson from DBI and create VCI, the generic Version Control Interface. Just like DBI, where Tim Bunce created one interface for performing the operations generic to any number of databases, VCI tries to provide a similar framework for the various VCI packages. The distribution ships with submodules to provide support for Bazaar, Cvs, Git, Mercurial, and Subversion. The documentation is full of warnings about the alpha nature of the effort, but it is worth your consideration, especially if you have to switch between VCS packages on a regular basis.

VCS Augmentation

To end this column I want to briefly mention that Perl can be useful not only for automating VCS packages but also for augmenting them. Leaving aside SVK, the most extreme example (it builds upon parts of Subversion to make it into a whole new beast), there are a number of excellent modules and scripts that make working with these packages easier. Here are just a few to get you started:

- SVN::Access makes maintaining the Subversion repository access control file easy. This is handy if you programmatically provision SVN repositories.
- SVN::Mirror can help keep a local repository in sync with a remote one.
- App::SVN::Bisect provides a command-line tool to make bisecting a repository (searching for a particular change by splitting the commits in half, and then in half again, and then in half again) easy.
- App::Rgit executes a command on every Git repository found in a directory tree.
- Github::Import allows for easy importing of a project into the Git community repository hub github.com.

If you haven't thought of using Perl with your favorite VCS package before, hopefully this column has given you some ideas on how to head in that direction. Take care, and I'll see you next time.

REFERENCES

[1] http://git-scm.com/.

[2] T. Swicegood, *Pragmatic Version Control Using Git* (Pragmatic Bookshelf, 2008).