Basic Version Control with Subversion

Richard Cobbe

Jan 16, 2007

1 Overview

Version control (VC) is an essential tool for any serious software development project, even if there's only a single developer.

There are two major benefits. First, a version control system keeps a complete history of all files in the project. This makes it easier to reproduce bug reports and serves as a crude backup mechanism: it's easier to recover from deletions or bad changes.

Second, if there are multiple developers working on a project, version control serves as a form of synchronization. With VC, it's much harder for developers to overwrite others' work.

1.1 Basic Version Control Operations

Most VC system are based around the idea of a repository. This is a database that contains the current state of your project. You won't edit the repository directly; indeed, in many cases, the repository will be hosted on a machine to which you don't have shell access.

To work on your code, then, you have to "check out" a copy of the repository to some filesystem you can access, usually somewhere under your home directory. In this working copy, you edit, compile, test, and debug. When it's time to publish your changes to others, you "commit" those changes back to the repository. Similarly, if you're working with someone else, you can update your working copy to include any changes that they've committed.

The repository contains not only the current state of your project, but also all past states that your project has been in. VC systems refer to these states as "revisions" and generally identify them with numbers, but the details of the numbering scheme vary.

2 Subversion

There are lots of version control systems out there, and they don't all agree on the Right Way to do version control. We'll teach you Subversion, a popular VC system, to get you started; it's up to you to learn the others if you ever need to use them.

2.1 Obtaining Subversion

SVN is easily available:

- It's included with most Linux distributions,
- It's available for Mac OS X through fink or darwinports,
- It's available through Cygwin on Windows XP, and

- TortoiseSVN (about which more later) provides a native Windows client.

SVN is already installed on the CCIS Solaris network, and TortoiseSVN is available on the CCIS Windows machines. (SVN may also be available through Cygwin on those machines.)

2.2 Normal Usage

I'll describe this in terms of using the SVN command-line client on a Unix-like machine, though there are other alternatives, which I'll mention later.

In SVN, we use URLs to name a repository (or a file or directory within it). You'll get these URLs from your local SVN administrator. In our case, they're all going to begin with

```
http://svn.barzilay.org/670/
```

and your team's specific part of the repository will be in a subdirectory of that. I'll send you the precise URL for your team once all members of your team have registered for SVN accounts (see below). For now, we'll use the "sandbox" project.

2.2.1 Getting a Working Copy

To get a working copy, use the `svn checkout` command, specifying the repository URL as an argument:

```
svn checkout http://svn.barzilay.org/670/sandbox
```

In many cases, including our repository, this will prompt you for a username and a password. The client caches authentication credentials so you won't have to re-authenticate at each SVN operation.

2.2.2 Publishing Changes

Now that we have a working copy, we can edit the files as we like. One of the files in our sandbox is `Combinatorics.java`, so let's edit that and add some comments like good developers. But these edits aren't visible to anyone else until we commit them to the repository. To do that, we commit our changes:

```
svn commit
```

or, if we only want to commit changes to specific files:

```
svn commit files ...
```

For each commit, SVN requires a log message, which is a human-readable high-level description of the changes that you're committing to the repository. You can specify this on the command line, or you can configure SVN to open an editor where you can edit the log message. By default, SVN is configured to require the message on the command line. To have it bring up an editor, set the `SVN_EDITOR` environment variable to the name of the editor you want SVN to start. You can also do this by setting the `editor-cmd` setting in `~/.subversion/config`, once that file has been created.

2.2.3 Incorporating Others' Changes

We've committed our changes to the repository, but they don't automatically appear in our partner's working copy. For that to happen, our partner has to explicitly ask for the recent changes to the repository:

```
svn update
```

If we update in our second working copy and then examine `Combinatorics.java`, we see that the comments are now there.

2.2.4 Commits, Again

This isn't quite the full story, though. If you're editing `Combinatorics.java`, but your partner has committed changes to the same file while you've been working, you can't commit. You don't want to unconditionally delete their changes without looking at them first, so SVN won't let you simply overwrite their changes with your version.

To fix this, `svn update` before you commit. This ensures that you're working on the most recent revision. Even if you and your partner have both made changes to the same file, SVN just merges your partner's changes into your file without you having to do anything about it.

Well, usually. And that brings us to....

2.2.5 Conflict Resolution

Subversion is pretty smart about merging changes. But what happens if two people make incompatible changes to the same part of the file at the same time?

Let's demonstrate. We'll update both working copies to the most recent revision, then make different changes to each copy of `Combinatorics.java`, but on the same line. We update/commit one copy, and it's fine. But when we update/commit the second copy, we get a conflict warning!

We have to resolve this conflict before we can commit our changes. To help us, SVN has left us lots of notes:

- `Combinatorics.java.mine` is a copy of the file as we had it just before the update.
- `Combinatorics.java.rOLD` is a copy of the file at the revision that we started working with. (That is, the *second* most recent update.)
- `Combinatorics.java.rNEW` is a copy of the file at the revision that we just pulled down with the `update` command.
- `Combinatorics.java` is the working copy of the file, annotated with conflict markers.

If we open `Combinatorics.java` in an editor, we can see the conflict markers. Let's resolve the conflict, save the file, and commit.

But the commit fails! We have to tell SVN that we've resolved the conflict:

```
svn resolved Combinatorics.java
```

This cleans up the notes that SVN created for us and lets SVN know that we've resolved the conflict. Now, we can commit the changed file.

SVN's notion of conflicts only goes so far, though. It doesn't know anything about what the contents of the file *mean*; it only knows which lines have changed. So, let's have our first programmer edit `Combinatorics.java` to change the type of the `fact` method to use an accumulator; it now takes two arguments.

While we're doing this, our second developer decides to add a new function, `perms`, that computes ${}_nP_r$; this will of course use `fact`. Since we're working from an old version, we give it a single argument. When we update, SVN doesn't tell us anything about a conflict, even though the resulting code clearly won't compile.

So, here's the process:

1. Edit.
2. `svn update`
3. Resolve any conflicts.

4. Try to compile the code; fix any problems. You may even want to run test cases here.
5. Commit. (Of course, someone else may have committed while you were fixing problems. In that case, you'll get an out-of-date error, so start again from step 2.)

2.2.6 Other Useful Commands

There are several other useful SVN commands that you'll use on a regular basis:

`svn add`: Add a new file to version control.

`svn delete`: Remove a file from version control (and also your working copy).

`svn move`: Move a file to a new location (or rename it), in the repository and your working copy.

`svn copy`: Copy a file to a new location in the repository (and your working copy).

`svn revert`: Remove all of the changes you've made to the file since the last time you updated it.

Important: just as with edits, the results of these commands aren't immediately visible to your partner. You have to commit them, just like any other change. (The only exception is `svn revert`, which only changes your working copy and doesn't affect the repository.)

2.2.7 Getting Information

SVN provides several commands that allow you to get information about the files in the repository:

`svn status`: Indicates which files you've changed locally since your last commit.

`svn diff`: Shows what changes you've made to files since your last commit.

`svn log`: Displays the log message for the named file(s), defaulting to the current directory, from your last update on back.

When you move or copy a file, using the commands above, SVN keeps track of its original history. So, by default, if you `svn copy` a file and then `svn log` the new file, you'll see the file's revision history all the way back to the point at which it was first added to the repository. (SVN will follow an arbitrary number of copies back.)

2.2.8 Getting More Information

There are two good ways to get additional information. First, you can use the `svn help` command to get basic information. Second, the subversion manual is available in HTML and as a PDF at `http://svnbook.org/` on the web.

2.3 Trying it Out

The best way to learn Subversion is to try it out on a project where you can't cause too much damage. I've created a sandbox within our repository where you can experiment with Subversion; it is located at `http://svn.barzilay.org/670/sandbox/`.

3 Best Practices

By now, you know how the SVN tool works, at a basic level. In this section, I'll discuss some basic ideas that help you use the tool well, with as few surprises to you and your partner as possible.

3.1 What to Commit

It's important to think about what kind of files you should commit to the repository. You must add all of the files that anyone will need to build the project: source code, makefiles, etc.

However, it's a bad idea to commit any file that's automatically generated by part of this build process. First, these often tend to be binary files, and SVN's merge algorithm doesn't work on binary files. Second, you can get into consistency problems; it's easier to just automatically generate these during builds.

3.2 When to Commit

How often should you commit?

More frequent commits are generally a good idea. This way, each revision has smaller changes, and it'll be easier to remove those changes later should you need to. (Although we didn't talk about it, SVN has a command that lets you roll back previous changes made to a file. If you need that, come talk to me and I'll show you how it works.) If, however, you have several changes in one big revision, you can't roll any of those changes back separately.

That said, there are limits. A good rule of thumb is that you should never commit anything that doesn't compile. If you do, and it's right before you go out of town for a week, and your partner updates right after you leave, then he's stuck until you get back.

Some teams in industry even require you to pass all of your test cases before you commit, although I think this is a bit too stringent. (A good happy medium seems to be requiring that all commits compile, then automatically running the test cases. A failed test case doesn't block the commit notification, but the test infrastructure can send out an email alert about the failed tests.)

3.3 Log messages

Log messages should be brief, fairly high-level summaries of what changed. Instead of saying

```
changed %02d to %02x on line 42,
```

you should say something more like

```
Fixed output format bug in function123.
```

The first message is too specific; we can get that information through `svn diff`. On the other hand, you want to say something more detailed than just "`fixed bug.`" Which bug did you fix?

Keep in mind that your log messages are primarily for your own benefit. If you later discover that your fix for a specific bug actually made matters worse, you'll want to roll that back. In order to do that, you'll need the specific revision number(s) for the bugfix in question. Make sure your log messages are complete enough so you can find the right revisions!

4 Using Our Repository

As described above, we've set up a SVN repository that we expect you to use for your projects. This section will tell you what you need to know in order to use our repository.

4.1 Getting an Account

First, our repository requires that you authenticate yourself, as I demonstrated earlier. The first step, of course, is to create your account. To do this, go to `https://csu670.barzilay.org/` and fill out the form. Important: you **must** do this by this coming Friday, the 19th.

It will be most convenient for you if you use your CCIS username with our Subversion repository, but this is not a requirement. In addition, you should follow the normal guidelines and choose a password that's hard to guess. In particular, please don't use a password that you're already using for some other account.

If you lose your password after I've created your SVN account, you must come see me in person.

4.2 Authentication

The first time you connect to the repository, the SVN client will require you to authenticate, assuming that your SVN username is the same as your Unix username. If the two aren't the same, then simply hit enter at the password prompt, and SVN will ask for your username as well.

As discussed above, the SVN client will cache your authentication information, so you shouldn't need to enter your password again on that machine. If you check out your project on a different machine, you will need to re-authenticate.

4.3 Directory Structure

Within the repository, I will create a separate directory for each team. You will have full read and write access within your own team's directory; other teams will of course be off-limits.

Within your team directory, you will see the following directory structure, where `<team>` is replaced by your team's name:

```
<team>/work
<team>/work/README
<team>/work/compile
<team>/work/run
<team>/work/run-tests
<team>/work/BUGS
<team>/work/BUGS/closed
<team>/turnin
```

You should create your project under the `work` directory.

4.3.1 The `README` File

At the top level within your `work` directory, you must create a README file, which must be viewable as plain text. This file must contain the following things:

1. A high-level roadmap of the code. What does each file do? If you've split your project up into subdirectories, what does each subdirectory do?
2. A list of the parts of the project that you attempted.
3. Any known bugs. If, for example, you tried a specific part of the project, and it works except in some really weird cases, tell me about the exceptions.

4.3.2 Scripts

You must create three scripts (or other executables, if you prefer) within your `work` directory. These scripts must be executable, and they should have the following behavior:

`compile`: compiles your project. This script must exist, but it can be a NOP if your language doesn't require a specific compilation step.

`run`: Run your project. This script may assume that I have already run `compile`.

`run-tests`: Runs your project's test cases. Again, this script may assume that I have already run `compile`. The precise output of this script is up to you, but it should finish by printing a line indicating how many test cases it ran, how many succeeded, and how many failed.

If, for instance, you have chosen to implement your projects in C++ with a Makefile, then the `compile` script would simply call `make` with the appropriate arguments.

These scripts must run on the college's Solaris network. They should not make any further assumptions about the environment in which they are run; this includes things like `$PATH` settings.

4.3.3 `BUGS`

We had hoped to have a bug-tracking system for you to use with this course, but it wasn't ready in time. In lieu of that, we're asking you to keep track of bugs in the `BUGS` directory within your project.

Each bug (or to-do item; see below) should be in a separate file, which should look like the following:

```
Summary:        one-line summary of bug
Component:      which part of your project contains the bug
Severity:       how severe: critical, major, normal, minor, trivial
Milestone:      what's the deadline on fixing the bug
Owner:          who's responsible for fixing it
```

After these five lines, you should include a longer description of the bug, ideally including instructions on how to reproduce the problem.

Following the description, you may wish to add notes to the bug report, as you investigate the problem and determine its cause, or as you discuss potential solutions with your partner. (Having records of these discussions can be very helpful in the long run, as you try to remember why this particular piece of code looks like it does, or why you made certain design decisions.)

Once you have resolved a bug, you can move it into the `BUGS/closed` directory, so you don't have to filter those bugs each time you look at the database. For example, to move the `startup-crash.txt` bug into the closed directory, you would execute the following commands from within the `BUGS` directory in your working copy:

```
svn mv crash-on-startup.txt closed/startup-crash.txt
svn commit
```

An important note: while the directory is called `BUGS`, the contents can be more general. Many industrial developers consider bug-tracking systems to be sophisticated to-do lists. So, for example, you could put feature requests in your `BUGS` directory. Indeed, many development teams require developers to start a new project by opening "bug reports" that describe the functionality they intend to implement for that project. When the implementation is complete and tested, they close out the bug report.

4.4 How to Submit a Project

To submit a project, you will copy the contents of your `work` directory to a specified location within the `turnin` directory.[1] We will specify the name of the subdirectory under `turnin` as part of the assignment. I will consider the contents of `turnin/`*`project-name`* as of the project deadline to be your submission.

In the instructions that follow, I'll use these abbreviations:

***<WORK>*:** the location of your working copy. Example: `~/classes/670/work`.

***<TMP>*:** the location of some temporary space in your account. **Important:** this must *not* be part of your working copy! Example: `~/tmp`.

***<TEAM>*:** the name of your team's part of the SVN repository. (I'll email this to you when I create your team's space.) Example: `jabra-ventz`.

***<PRJ>*:** the name of the turnin directory, as specified by the assignment. Example: `project-02`.

To submit a project, follow these steps:

1. Make sure that you've committed all your changes to the repository. Be especially sure that you've added all the necessary files to the repository and committed them.
2. `cd <TMP>`
3. `svn checkout -N http://svn.barzilay.org/670/<TEAM>/turnin`
4. `cd turnin`
5. `svn mkdir <PRJ>`
6. `svn commit -m "Added submission directory for project 3"`
7. `svn merge <PRJ>@HEAD <WORK>@HEAD <PRJ>`

   After this command, *<PRJ>* will be an exact copy of *<WORK>*.
8. `svn commit <PRJ>`
9. You should probably change to the *<PRJ>* directory at this point and make sure your program compiles and runs. Running the test cases is a good idea, too.

   If you find a problem, then make the necessary changes within *<WORK>* and repeat from step 7.

Once *<PRJ>* looks like you want it to, you can delete the *<TMP>*`/turnin` directory that you created above.

Finally, you only need to do this once per team.

5 Caveats and Warnings

Two big warnings:

1. Remember that nobody else can see your changes if you don't commit!

[1] For those of you familiar with SVN's branching capability, we're creating a branch for your submissions, though of course you won't usually make further commits to these branches.

2. The most recent release of Subversion, v1.4, introduced an incompatible change in the format of the working copy. So, while v1.4 can read a working copy created with any version, versions 1.3 and earlier cannot read a working copy created with Subversion 1.4. Additionally, if you use the 1.4 client with a 1.3 working copy, it will update the working copy to the new format, making that working copy unusable by older clients.

   Upshot: The subversion client installed on the Solaris network is version 1.1.3, so it cannot read the new working copy format. (It'll talk to our server with no problems, though.) So, if you want a working copy in your Solaris home directory, make sure you don't use a newer client on it, otherwise it'll be unreadable by the default client. (This might come up if you mount your Solaris home directory on one of the College's Windows machines and use TortoiseSVN on that working directory, for example.)

   So be careful.

6 Other SVN Clients

Several other subversion clients are available for your use, including some that integrate with popular development environments. I'll list some of the most popular here; there are almost certainly others as well. You're on your own for installing, configuring, and using these, though.

- TortoiseSVN (`http://tortoisesvn.tigris.org/`) integrates with the Windows XP explorer. I believe this is already installed on the College's Windows machines.
- Many of the recent Emacsen integrate with SVN, much as they did with CVS. Try `M-x svn-status`.
- There is an Eclipse plugin, Subclipse, that provides SVN integration within Eclipse. See `http://subclipse.tigris.org/` for more details and installation and use instructions.
- If your Subversion repository is available through an `http:` URL, as ours is, you can point any web browser at that URL to browse the repository. This is, of course, read-only access.