UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 61B
Fall 2008

P. N. Hilfinger

Using Subversion

1 Introduction

A *version-control system* (also *source-code control system* or *revision-control system*) is a utility for keeping around multiple versions of files that are undergoing development. Basically, it maintains copies of as many old and current versions of the files as desired, allowing its users to retrieve old versions (for example, to "back out" of some change they come to regret), to compare versions of a file, to merge changes from independently changing versions of a file, to attach additional information (change logs, for example) to each version, and to define symbolic labels for versions to facilitate later reference. No serious professional programmer should work without version control of some kind.

SUBVERSION, which we'll be using this semester, is an open-source version-control system that has seen increasing use. The basic idea is simple: SUBVERSION maintains *repositories,* each of which is a set of *versions* (plus a little global data). A version is simply a snapshot of the contents of an entire directory structure—a hierarchical collection of directories (or "folders" in the Mac and PC worlds) that contain files and other directories. A repository's administrator (in this class, that's the staff), can control which parts of the repository any given person or group has access to.

At any given time, you may have *checked out* a copy of all or part of one of these versions into one of your own directories (known as a *working directory*). You can make whatever changes you want (without having any effect on the respository) and then create (*commit* or *check in*) a new version containing modifications, additions, or deletions you've made in your working directory.

If you are working in a team, another member might check out a completely independent copy of the same version you are working on, make modifications, and commit those. You may both, from time to time, *update* your working directory to contain the latest committed files, including changes from other team members. Should several of you have changed a particular file—you in a committed revision, let us say, and others in their working copies—then the others can update their working copies with your changes, *merging* in any of your changes to files you've both modified. After these team members then commit the current state of their working directories, all changes they've made will be available to you on request. If you attempt to commit a file that someone else has modified and committed, then SUBVERSION will require you to update your file (merging these committed changes into it) before it will allow you to commit.

In this class, I may give you various code skeletons to be filled in with your solutions. Sometimes, these skeletons may contain errors I wish to fix, or I might take pity and add a useful method or two after handing out the assignment. By distributing these skeletons to you in the respository, I make it possible for you to merge my changes in the skeletons into your own work, without your having to find our changes and apply them by hand.

Should you mess something up, all the versions you created up to that point still exist unchanged in the respository. You can simply fall back to a previous version entirely, or replace selected files from a previous version. Of course, to make use of this facility, you must check your work in frequently.

Even though the abstraction that SUBVERSION presents is that of numbered snapshots of entire directory trees full of files, its representation is far more efficient. It actually stores *differences* between versions, so that storing a new version that is little changed from a previous one adds little data to the repository. In particular, the representation makes it particularly fast and cheap to create a new version that contains an additional copy of an entire subdirectory (but in a different place and with a different name). No matter how big the subdirectory is, the repository contains only the information of what subdirectory in what version it was copied from. As we'll see, you can use these cheap copies to get the effect of *branching* a project, and of *tagging* (naming) specific versions. These copies have other uses as well. In particular, they allow you to import a set of skeleton files for an assignment without significantly increasing the size of the repository.

What follows is specific to this course. SUBVERSION actually allows much more flexible use than I illustrate here.

2 Preparing to Use the CS61B Server

SUBVERSION is an example of a *client/server* system. That is, running it requires two programs, one of which (the server) accesses the repository and responds to requests from the other program (the client), which is what you run. The client and server need not run on the same machine, allowing remote access to the repository. We have one server program, which the staff will look after, and several alternatives for the client program, depending on how you are working. Later in this document, we'll describe the standard command-line client (called `svn`), which you run in a shell, and the Eclipse client, which is integrated into the Eclipse programming environment.

The staff will maintain a SUBVERSION repository containing subdirectories for each of you on the instructional machines under a staff account (`cs61b-ta`). The SUBVERSION server program manipulates this repository and runs on any of the instructional machines. However, you never run it directly, but rather leave it up to your client to run it. You will be able to run the SUBVERSION client program (`svn`) either from your class account or remotely from any machine where `svn` and SSH are installed.

SSH (the Secure Shell) is another client/server program that allows you to authenticate yourself and communicate securely to an account on (potentially) another machine that is accessible locally or over the Internet. Not only can you use it as its name suggests to create a command shell on a remote machine, but your programs can use it as a utility to create secure and authenticated connections to arbitrary programs running on other machines (a process known as "SSH tunneling"). This is precisely how your SUBVERSION clients will speak to our servers.

Before any of this is possible, however, you will have to create SSH *public and private keys,* a pairs of files that will allow SSH to authenticate connections between our server program and your client programs. The basic idea is that you give our server (or in general any SSH-using program that you wish to talk to) a copy of your SSH public key. Any client program that you run can then use the corresponding SSH private key (which you keep secret) to authenticate itself to the server. There are directions for generating and deploying these keys on the class web page. From this same page, you can initialize your directories in the repository as well.

3 Subversion from the command line

3.1 Setting up a working directory

With SUBVERSION, you are almost always working on a *copy* of something in the repository. Only when you *commit* your copy (usually with the command `svn commit`) does the repository get changed. It is impossible to emphasize this too much—failing to commit changes (either by not issuing the command at all or by failing to notice error messages when you do) is the root of most evil that befalls beginning SUBVERSION users.

In particular, a *working directory* is a copy of some directory in the repository. I suggest that you set up such a directory under your account. In what follows, I'll assume this directory's name is `svnwork` (any name will do) and that it is in your home directory. I'll also assume that your class login is `cs61b-yu` (I need your login because that's the name of the part of the class repository that you're allowed to change). After you've set up your SSH keys (see §2), you can create this directory with the following sequence of commands:

```
% cd    # Go to your home directory
% svn checkout svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk svnwork
Checked out revision 101
```

The part that begins "`svn+ssh:`" is the *URL* (Universal Resource Locator) for a part of the repository. It identifies the protocol for communicating (`svn+ssh`), the account that actually owns the repository files (`cs61b-ta`), a host machine for those files, the directory containing the repository (well, actually, we've set that up to be implicit in our case) and a subdirectory of the repository (`cs61b-yu/trunk`). SUBVERSION will create directory `svnwork` if needed, and will add to it a hidden directory called `svnwork/.svn`. Basically, you'll never have to worry about these `.svn` directories; they contain administrative information that SUBVERSION leaves around for itself for use in later commands, rather like the "cookies" that remote websites are always storing on your disk so that they remember who you are when next you visit.

The part of the repository that you own is the subdirectory `cs61b-yu`. We've set this up for you so that it contains two (initially empty) subdirectories called `trunk` and `tags`. These are traditional names for the directory where you develop your solutions and check in intermediate versions (`trunk`) and the directory where you copy specific, named, versions of trunk directories (`tags`). In real life, you might use the `tags` directories to hold releases of your software. In this class, you'll use it to put versions that you hand in. We can read the entire repository (we own it, after all), and so we will see everything you put in your `tags` directory and automatically treat it as something you've handed in.

You will also want to keep a working copy of files that the staff makes publically available. Here's a good way to do that:

```
% cd    # Go to your home directory
% svn checkout svn+ssh://cs61b-ta@nova.cs.berkeley.edu/staff staff
Checked out revision 101
```

Now the directory `~/staff` contains a copy of our staff files. You can read and write these files, but have no fear: nothing you do to them will change the versions in the repository.

3.2 Starting a project

Suppose that you've just started work on some files for Project #2, and have placed them in the subdirectory `svnwork/proj2` (I'm assuming that you've created `svnwork` as in §3.1.

At the moment, SUBVERSION knows nothing about these files. You can see this by running the `status` subcommand:

```
% cd ~/svnwork
% ls
proj2
% svn status
?     proj2
```

which means "I see this directory called `proj2`, but it does not seem to taken from the repository that this working directory (`svnwork`) came from." As soon as possible, you should put all these files under version control with the `add` subcommand, which might produce the following output:

```
% cd ~/svnwork
% svn add proj2
A     proj2
A     proj2/Main.java
A     proj2/Main.java~
A     proj2/Main.class
A     proj2/doc
A     proj2/doc/INTERNALS
```

The repository *has not changed,* but now SUBVERSION knows that you intend for all these files to eventually be part of the repository, as you'll see if you check the status:

```
% svn status
A     proj2
A     proj2/Main.java
A     proj2/Main.java~
A     proj2/Main.class
A     proj2/doc
A     proj2/doc/INTERNALS
```

Additionally, `svn add` has modified the working directories `proj2` and `proj2/doc` by adding `.svn` directories to them, turning them into official working directories.

You probably did not want to archive copies of `Main.java~` (an Emacs back-up file) or `Main.class` (since it can be reconstructed at any time using the Java compiler). So let's remove them from version control:

```
% svn revert proj2/Main.java~ proj2/Main.class
```

which undoes any changes we've made to these files since last creating a version (in this case, it "unadds" them). At this point, your status is:

```
% svn status
A     proj2
A     proj2/Main.java
?     proj2/Main.java~
?     proj2/Main.class
A     proj2/doc
A     proj2/doc/INTERNALS
```

These particular files are annoying, and it would be better to have SUBVERSION ignore them automatically. We'll explain how to arrange this later.

The procedure I've outlined so far assumes that you start from scratch with your own files. If we have provided a skeleton, you can use it by first copying the skeleton into a working directory, like this:

```
# Make sure your copies of the staff files are up to date.
% svn update ~/staff
% ls ~/staff/proj2
Main.java
doc
% cd ~/svnwork
% svn copy ~/staff/proj2 proj2
A     proj2
```

You now will have a directory named `~/svnwork/proj2` with the same files that we ended up with originally.

So far, you've worked exclusively with your working files. The repository has not changed at all. If you were to erase your files at this point, everything you've done will be lost permanently. So now it's time to record all the changes you've made (specifically, these files and directories you've added) in the repository. As I said above, this is called *committing* the changes:

```
% cd ~/svnwork
% svn commit -m 'Start Project 2'
... various messages
Committed revision 1294.
```

Each version has a unique *revision number*. Because of how we've arranged the CS61B repository, the revision number will reflect the total number of versions created by all members of the class, even though their versions have no impact on yours.

SUBVERSION requires that each version have a *log message* attached to it, which in this case you've supplied with the `-m` option. For other changes, your log messages should be rather more substantial than this. You can also take them from a file:

```
% svn commit -F my-commit-notes
```

If you don't supply either, SUBVERSION will try to invoke your favorite editor. If that is Emacs, for example, you'll find yourself looking at a buffer that contains a list of the files you've added. Add an additional message to the buffer, save it, and exit.

Suppose that at this point, some malicious gremlin executes

```
% rm -rf ~/svnwork
```

You can get it all back with

```
% svn checkout svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk ~/svnwork
```

3.3 The typical work cycle

At this point, you enter the usual cycle of adding and editing files, compiling, and debugging. SUBVERSION will notice any files you change the next time you commit your changes, without any other action by you. Each time you introduce a new source file, tell SUBVERSION about it:

```
% svn add Manager.java
A         Manager.java
```

Occasionally, you'll need to delete a file that you previously committed:

```
% svn delete Munger.java
D         Munger.java
```

or rename it:

```
% svn move Munger.java Mangler.java
A         Mangler.java
D         Munger.java
```

In these last two cases, SUBVERSION will remove or move (rename) your working files appropriately and make a note to itself of these actions for the next commit operation. Neither editing, adding, removing, nor moving a working file has any effect on the repository, which remains unchanged until you commit your changes:

```
% svn commit -m "Add a Manager and mung the Munger"
...
Committed revision 1301.
```

In other words, revision 1301 now contains a snapshot of all the files in the current directory (or its subdirectories) that have changed since you last committed them, minus those you have removed, plus those you have added.

Feel free to commit *frequently,* whenever you think you've finished working on a file, or finished a minor change, or just feel like knocking off or going to lunch. You will not be wasting space because SUBVERSION stores only changes (or *deltas*) from one commit operation to the next.

You may find yourself wanting to work both from your home computer and your instructional account. SUBVERSION will help keep you synchronized. If, for example, you've been working from home, the command

```
% svn update
...
Updated to revision 1350.
```

issued from within a working directory, will bring your latest commit into that working directory. Be careful, though: if you haven't been careful to commit your changes at home, you won't get your most recent work. Likewise, if you did not commit the last changes you made in your instructional account (so that the working files have changes that are not reflected in the repository), you'll get messages about how those changes have been "merged" with the changes you committed from home. Your working copy will still be out of sync with the repository, and you may want to do a commit right away.

3.4 Comparing

One of the advantages to keeping around history is that you can see what you've done and recover what you've lost. After you've done some editing on your files, you can compare them with previous versions. To see what changes you've made to file `Main.java` since you committed your changes:

```
svn diff Main.java
```

You'll get a listing that looks something like this:

```
Index: Main.java
===================================================================
--- f2  (revision 1350)
+++ f2  (working copy)
@@ -1,3 +1,5 @@
-import java.lang.ArrayList;
+/* Project #2: Main program. */
+
+import java.util.ArrayList;
 import java.util.Scanner;

@@ -6,4 +8,5 @@
 int N;
 String v;
+Scanner inp;

 for (String arg : args) {
```

The '+'s indicate lines added in your working copy; the '-'s indicate lines removed in your working copy; and the other lines indicate unchanged lines of context. The remaining lines give the line numbers of these changes in the two different versions.

To see all changes to all files, just leave off any file names:

```
% svn diff
Index: Main.java
...
Index: Manager.java
...
```

You can also compare your working files to previous revisions by number, as in

```
% svn diff -r 1294
...
```

or compare two committed revisions of the current directory, as in

```
% svn diff -r 1294:1300
```

Of course, revision numbers are not the easiest things to deal with, so there are other options for specifying differences. To find the differences between the current working copy and what you had at 1PM, you can write

```
% svn diff -r {13:00}
```

or between the current working copy and noon on the 25th of October:

```
% svn diff -r "{2007-10-25 12:00}"   # You need quotes because of the space
```

or between the last copy you checked in and the previous one (ignoring changes in the working directory):

```
% svn diff -r PREV:BASE Main.java
```

Very often, you're simply interested in knowing what files you've changed in the working directory and not committed yet. Use the 'status' command for this purpose:

```
% svn status
M       Main.java
A       Utilities.java
A       doc/Manual.txt
D       Junk.java
```

This lists changes with a flag indicating modified files (M), added files (A), and deleted files (D).

3.5 Retrieving previous versions

Suppose you've modified `Main.java`, have not committed it, but have decided that you've really messed it up and should simply roll back to the version you committed. This is particularly easy:

```
% svn revert Main.java
Reverted 'Main.java'
```

You can get fancier. Suppose that after committing `Main.java`, you have second thoughts, and want to fall back to the previous version of `Main.java`. Here's a simple way to do so:

```
% svn commit
...
Committed revision 1522.
% svn status
% svn cat -r PREV Main.java > Main.java
% svn status
M       Main.java
% svn diff -r PREV Main.java
No output
```

The `cat` command, like the similarly named command in Unix, lists file contents from the repository. The "`> Main.java`" here is Unix notation for "send the standard output of this command to `Main.java`." Thus, the command overwrites `Main.java` with revision PREV—the version before the last that was checked in. The repository is not changed yet; the `status` commands in this example show that `Main.java` has been modified from the last committed version. You will have to commit this change to make it permanent.

The problem with this particular technique (less important in this class, but you might as well get exposed to real-world considerations) is that it messes up the historical information that SUBVERSION keeps around. The `log` command in SUBVERSION lists all the changes that a given file or directory has undergone. With the short method above, all you'll be told is that the file committed in 1522 is somehow *different* from that in later revisions, rather than being told that the later revision is the *same* as that in revision 1521. So I can get a little fancy and recover the previous version in two steps:

```
% svn delete Main.java
...
% svn commit -m 'Remove Main.java in order to restore from previous version."
...
% svn copy -r 1521 \
>     svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk/proj2/Main.java .
```

(don't forget the dot on the end, meaning "current directory"). Now when you next commit, you'll have the version of `Main.java` as of 1521 and the log will say so.

3.6 URLs and paths

At this point, you've seen two different syntaxes for denoting a file or directory. Just plain Unix paths or Windows file names:

```
svn delete Main.java
```

and entire URLs (*Universal Resource Locators,* just as in your browsers):

```
svn checkout svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk/proj2 ~/svnwork/proj
```

Plain file names generally refer to working files (files in *your* directories); URLs refer to files in the repository. As you can see, URLs are rather tedious to write, which is why you generally want to check out a copy into a working directory and then work from there (SUBVERSION uses those `.svn` directories to keep track of administrative details, including the original URL, so commands can be much shorter).

Sometimes, however, you really need to reference a file or directory in the the repository. The checkout command and copy commands illustrated previously are two examples. Another very important one is producing a copy of a directory in the repository itself. Suppose that you've completed work on Project #2 and are ready to submit it. Suppose, in fact, that you've just finished committing your final version, which you keep in working directory `~/svnwork/proj2`. In this class, we've established the convention that all submissions go in a subdirectory called `submit` of your repository. The full command to submit is

```
% svn copy svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk/proj2 \
>          svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags/proj2-1 \
>  -m "Project 2, first submission"
Committed revision 1600.
```

This does not change your working directory, but it causes the entire directory tree `trunk/proj2` to get copied as a new subdirectory `tags/proj2-1` in the repository. Any future changes to `trunk/proj2` either in your working files or in the repository have no effect on this copy. If you have a reason later to resubmit, just use the same command, but with a slight modification to distinguish submissions:

```
% svn copy svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk/proj2 \
>          svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags/proj2-2 \
>     -m "Project 2, second submission"
Committed revision 1701.
```

This particular application of `svn copy` is an example of *tagging* your Project #2 trunk, and you might refer to the copy in directory `submit` as a *release* of your project.

3.7 Merging

By far the most complex part of the version-control process is the process of *merging:* combining independent changes to a file. The main problem is that there is no automatic way to do it; at some point, a human must intervene to resolve conflicting changes. SUBVERSION's facilities here are rather primitive as these things go, but for our limited purposes, they will probably suffice. In this course, the only time the problem is likely to come up is in cases where you have started from some skeleton files I give you and I later change the skeleton. Assuming you want to take advantage of my changes, you'll want to perform some kind of merge.

In general, each time I change files that I expect you to modify, I will tag the resulting directory, as described above. So, for example, I will keep the current version of the initial files for Project #2

in the repository directory `.../staff/proj2`, and snapshots of each "released" version of the files in directories `.../staff/proj2-0`, `.../staff/proj2-1`, etc., with the highest-numbered tag being a copy of `.../staff/proj2`.

Let's assume that there are two tags for Project #2, `proj2-0` and `proj2-1`, and that when you copied the public trunk version, `proj2-0` was the version you copied. Now that `proj2-1` is out, you want to incorporate its changes. First, commit your current trunk version (this is a safety measure that gives you a convenient way to undo the merge if it gets fouled up somehow). Now merge in the changes between `proj2-0` and the current trunk version like this:

```
% cd ~/svnwork/proj2
% svn merge svn+ssh://cs61b-ta@nova.cs.berkeley.edu/tags/proj2-0 \
>           svn+ssh://cs61b-ta@nova.cs.berkeley.edu/staff/proj2
U      Main.java
C      Manager.java
```

The merge operation modifies your working directory only; the repository is not changed. The message tells you that `Main.java` has been updated with changes that occurred between `proj2-0` and `proj2-1`, and that `Manager.java` has also been updated, but there were some *conflicts*—changes between `proj2-0` and the trunk that overlap changes you made since `proj2-0`. These conflicts are marked in `Manager.java` like this:

```
<<<<<<< .working
     System.out.println ("Welcome to my project");
     initialize ();
=======
     initialize (args);
>>>>>>> .merge-right.r1009
```

The lines between "`<<<<<<< .working` and `=======` are from your version of the file `Manager.java`, and the ones below `=======` up to the `>>>>>>>...` line are from `proj2-1`. In this case, `proj2-0` probably contained `initialize ();` and you added a `println` call, while `v1` added an argument to the call to `initialize`. SUBVERSION decided that was close enough to `proj2-1`'s change to suggest an overlap, and so has asked you to fix the problem. Simply edit the file appropriately. In this case you probably want

```
     System.out.println ("Welcome to my project");
     initialize (args);
```

Now tell `Subversion` that the problem is solved with the command

```
% svn resolved Manager.java
Resolved conflicted state of 'Manager.java'
```

Finally, when all conflicts are resolved, commit your merge just as you would any change to your files.

3.8 Quick guide

Here are some common version-control tasks and the SUBVERSION commands that perform them. In all of the following, I'll assume that

- Your class account is `cs61b-yu`.

- You having chosen to keep working copies of your directories in directory `~/svnwork`.
- You use the "classical" SUBVERSION naming scheme, with one trunk directory that contains one subdirectory per assignment.

We'll use "change directory" (`cd`) commands in these examples just to make clear what directory we're in. Many of these will be redundant.

Set up your working directory `~/svnwork`:

```
% svn checkout svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk ~/svnwork
```

Create a directory for a new project:

```
% cd ~/svnwork
% svn mkdir proj2
% svn commit -m "Set up Project 2"
```

You'd put your work in `~/svnwork/proj2`.

Create a directory for a new project from our template:

```
% cd ~/svnwork
% svn copy svn+ssh://cs61b-ta@nova.cs.berkeley.edu/staff/proj2 proj2
% svn commit -m "Set up project 2"
```

Tell SUBVERSION about a new file:

```
% cd ~/svnwork/proj2
```
create `Main.java`
```
% svn add Main.java
```

This command does *not* change the repository. You still have to commit the change.

Tell SUBVERSION about a whole new subdirectory:

```
% cd ~/svnwork/proj2
```
create directory `util` *and files* `util/IO.java` *and* `util/Manager.java`
```
% svn add util
```

This command does *not* change the repository. You still have to commit the change.

Delete a file or directory:

```
% cd ~/svnwork/proj2
% svn del Main.java
D     Main.java
% svn del util
D     util/IO.java
D     util/Manager.java
D     util
```

These change only your working directory. You must commit in order to change the repository.

Get a brief summary of changes since the last commit:

```
% cd ~/svnwork/proj2
% svn status
M      util/Manager.java
A      util/Tools.java
```

Commit changes:

```
% cd ~/svnwork/proj2
% svn commit -m "Log message"
various messages
Committed revision 2000.
```

This transmits all additions, deletions, renamings, and changes to version-controlled files in your working directory to the repository, and creates a new revision that contains them. Leave off the `-m "Log message"` to get SUBVERSION to use a text editor to compose the message.

Compare your current working files against the repository: To compare against the most recently committed version:

```
% cd ~/svnwork/proj2
% svn diff
```

Against another version by revision number:

```
% svn diff -r 1756
```

Against another version in the repository by time:

```
% svn diff -r '{13:00}'
```

Against another version in the repository by time and date:

```
% svn diff -r '{2007-10-17 13:00}'
```

Compare a particular working file with the repository:

```
% svn diff Main.java
```

You can also use '`-r`' here to look at previous versions of the file in the repository.

Merge corrections in our template into your trunk: First commit the trunk directory. Let's assume that you started from the initial release of the skeleton, which will be called `proj2-0`, and want to merge in the changes we've made to `proj2-1`.

```
% cd ~/svnwork/proj2
% svn merge svn+ssh://cs61b-ta@nova.cs.berkeley.edu/staff/proj2-0 \
>           svn+ssh://cs61b-ta@nova.cs.berkeley.edu/staff/proj2-1
U      Main.java
C      Manager.java
{\it Edit any files marked C to resolve clashes}
% svn resolved Manager.java     # Tell svn about resolution
...
% svn commit -m "Merged in skeleton changes between v0 and v1"
Committed revision 2009.
```

Submit a copy of your trunk: First, make sure your work is committed:

```
% cd svnwork/proj2
% svn commit -m "Project 2, ready to submit"
Committed revision 2010.
```

and now create a submission copy:

```
% svn copy svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/trunk/proj2 \
>          svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags/proj2-0 \
>    -m "Project 2, first submission"
Committed revision 2011.
```

Follow the same procedure for additional submissions of the same project, changing `proj2-0` to `proj2-1`, etc.

Find out what you have submitted: To find out what submissions are actually in the repository (as opposed to merely sitting, uncommitted, in your working directory), use

```
% svn ls svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags
```

A slight variation of this command lists all the files (not just the top-level directories):

```
% svn ls -R svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags
```

although you will probably want to restrict this to a single submission:

```
% svn ls -R svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags/proj2-1
```

Of course, you can see what you actually submitted by just checking it out (I suggest doing so someplace *other* than `svnwork`, to avoid confusion):

```
svn co svn+ssh://cs61b-ta@nova.cs.berkeley.edu/cs61b-yu/tags/proj2-1 DIR
```

4 Subversion from Eclipse

The Eclipse programming environment allows you to plug in a couple of different SUBVERSION clients. We recommend using Subversive. The class website will have directions for obtaining and installing Eclipse and Subversive.

Unfortunately, the various versions of Eclipse—Windows, Mac, Linux, etc.—have gratuitous differences in the order and content of their dialog pages. Therefore, we're going to post detailed instructions on the class web page.