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

CS 164
Spring 2010

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 enjoys considerable 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 repository) 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 repository, 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 repository. 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.

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. In this document, we'll describe the standard command-line client (called `svn`), which you run in a shell. The staff will maintain a SUBVERSION repository containing subdirectories for each of you on the instructional machines under a staff account (`cs164-ta`). The SUBVERSION server program manipulates this repository and runs on one of the instructional machines. You will be able to run the SUBVERSION client program (`svn`) either from your class account or remotely from any machine where `svn` is installed.

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

2 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 `mywork` (any name will do) and that it is in your home directory. I'll also assume that your class login is `cs164-yu` (I need your login because that's the name of the part of the class repository that you're allowed to change). You can create this directory with the following sequence of commands:

```
$ cd    # Go to your home directory
$ svn checkout svn+ssh://cs164-ta@torus.cs.berkeley.edu/cs164-yu mywork
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 (`cs164-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 (`cs164-yu`). SUBVERSION will create directory `mywork` if needed, and will add to it a hidden directory called `mywork/.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 talk to them.

You would soon get heartily tired of typing "`svn+ssh://cs164-...`," but fortunately, you have two things working for you:

- Once you have set up a working directory, the hidden `.svn` contains the information about the repository's URL (the `svn info` command will show it to you. Therefore, you will not have to use the full URL too often.
- Feeling your pain, we have used the marvelous facilities provided by modern operating system command processors to define shorthand notation. Specifically, on the instructional machines, we have defined `$MYREPOS` to be short for

  `svn+ssh://cs164-ta@torus.cs.berkeley.edu/cs164-yu`,

  so that the command to create your `mywork` directory is simply

  ```
  svn co $MYREPOS mywork
  ```

The part of the repository that you own is the subdirectory `cs164-yu`. We've set this up for you so that it contains one (initially empty) subdirectory called `tags`, which is the traditional name for the directory where you copy specific, named, versions of a project you are developing. 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.

The staff keeps a subdirectory of the repository containing its publicly available files (such as the skeleton we provide for homeworks and projects). For example, the skeleton for project #2 is named `svn+ssh://cs164-ta@torus.cs.berkeley.edu/staff/proj2`. Again, we have a shorthand: `$STAFFREPOS/lab2`.

Finally, all members of your team will have access to a team directory in the repository, named for your team under the URL `svn+ssh://cs164-ta@torus.cs.berkeley.edu/`*TEAM*, where *TEAM* is the name of your team. For this, the shorthand is `$TEAMREPOS`. You can set up a working directory for this just as you did for your own personal work:

```
svn co $TEAMREPOS ~/teamwork
```

This is a working directory—a *copy* of the repository directory with your own changes. When a teammate changes the repository, your working directory will not change until you specifically pull in the updates, as we'll discuss.

3 Starting a project

Suppose that you've just started work on some files for homework #2, and have placed them in the subdirectory `mywork/hw2` (I'm assuming that you've created `mywork` as in §2. We'll call the `hw2` directory in the repository the *trunk* version of `hw2`—the one you do development on. Your submissions will be snapshots of this directory. In other "real-life" uses of Subversion, you'll see these trunk directories stored in subdirectories named `trunk`, but we won't bother with that here.

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

```
$ cd ~/mywork
$ ls
hw1
hw2
$ svn status
?     hw2
```

which means "I see this directory called `hw2`, but it does not seem to taken from the repository that this working directory (`mywork`) came from." Because `hw1` and everything in it are up-to-date, `svn status` says nothing about it. As soon as possible, you should put all the files in `hw2` under version control with the `add` subcommand, which might produce the following output:

```
$ cd ~/mywork
$ svn add hw2
A     hw2
A     hw2/hw2.txt
A     hw2/prob1
A     hw2/prob1.y
```

The repository *has not changed,* but now the SUBVERSION client 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     hw2
A     hw2/hw2.txt
A     hw2/prob1
A     hw2/prob1.c
A     hw2/prob1.y
```

Additionally, `svn add` has modified the working directory `hw2` by adding an `.svn` directory, turning it into official working directories.

If you'd been working on it before this step, your directory might contain files such as `hw2.txt~` (an Emacs back-up file) or `prob1.o` (the result of compilation), which you probably do not need to archive (they are either useless or reconstructable). On the instructional machines, we've configured your version of the client to ignore these and several other files. For those of you who want to set up SUBVERSION clients at home, take a look at the contents of the file `~/.subversion/config` to see what we did (look for '`global-ignores`'). On the other hand, in the example above, it happens that the files `prob1` and `prob1.c` should not be archived either (they are generated automatically from `prob1.y`. At this point, you can remove them from version control:

```
$ svn revert hw2/prob1.c hw2/prob1
```

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

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:

```
$ cd ~/mywork
$ svn copy $STAFFREPOS/hw2 hw2
A     hw2
```

(As described previously, `$STAFFREPOS` is a shorthand defined on the instructional machines. The full command is

```
$ svn copy svn+ssh://cs164-ta@torus.cs.berkeley.edu/staff/hw2 hw2
```

which should work for you on systems other than the instructional computers.) You now will have a directory named `~/mywork/hw2` 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 would 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 ~/mywork/hw2
$ svn commit -m 'Start homework #2'
... various messages
Committed revision 1294.
$ svn update
At revision 1294.
```

Each version has a unique *revision number.* Because of how we've arranged the CS164 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. In this course, you'll probably always want to add that final `svn update` command, as we did above, so that your working directory and the repository are completely in sync. Let's not go into an explanation of why SUBVERSION's designers made this step optional just now.

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
$ svn update
```

Both of these alternatives have problems. The `-F` option is somewhat inconvenient, and `-m` encourages cryptic and uninformative error messages. If you don't supply either, SUBVERSION will try to invoke your favorite editor. By default, we've set up the class accounts to make that editor be `emacsclient` or `emacs`. This means that if you don't already have a running Emacs, you'll suddenly get one, displaying a buffer in which to compose a log message (save the message and exit when done). If you do have a running Emacs, it will sprout a new buffer for the log mesage (in this case, finish the message with the `C-x#` command).

Suppose that at this point, some malicious gremlin executes

```
$ rm -rf ~/mywork
```

You can get it all back with

```
$ svn checkout $MYREPOS ~/mywork
```

4 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 prob2.y
A         prob2.y
```

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

```
$ svn delete garbage.c
D         garbage.c
```

or rename it:

```
$ svn move garbage.c trash.c
A         garbage.c
D         trash.c
```

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 problem 2 solution rename garbage=>trash"
...
Committed revision 1301.
$ svn update
```

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 on the instructional machines, 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.

5 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 `prob2.y` since you committed your changes:

```
svn diff prob2.y
```

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

```
Index: prob2.y
===================================================================
--- f2  (revision 1350)
+++ f2  (working copy)
@@ -1,3 +1,5 @@
-/* Problem 2 */
+/* Problem #2 */
+
+%token NUM
 %token ID

@@ -6,4 +8,5 @@
 int N;
 string v;
+FILE* inp;

 for (int i = 0; i < N; i += 1) {
```

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: prob2.y
...
Index: trash.c
...
```

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 version you last checked out to or committed from the working directory (BASE), ignoring any uncommitted changes to the working file since then, and the version of the file before it last changed in the repository (PREV):

```
$ svn diff -r PREV:BASE prob2.y
```

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       prob1.y
A       prob2.y
A       doc
A       doc/Manual
D       junk
```

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

6 Retrieving previous versions

Suppose you've modified `prob2.y`, 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 prob2.y
Reverted 'prob2.y'
```

You can get fancier. Suppose that after committing `prob2.y`, you have second thoughts, and want to fall back to the previous version of `prob2.y`. Here's a simple way to do so:

```
$ svn commit
...
Committed revision 1522.
$ svn update
At revision 1522.
$ svn status
$ svn cat -r PREV prob2.y > prob2.y
$ svn status
M       prob2.y
$ svn diff -r PREV prob2.y
```
No output

The `cat` command, like the similarly named command in Unix, lists file contents from the repository. The "`> prob2.y`" here is Unix notation for "send the standard output of this command to `prob2.y`." Thus, the command overwrites `prob2.y` 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 `prob2.y` 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 prob2.y
...
$ svn copy -r 1521 $MYREPOS/hw2/prob2.y .
```

(don't forget the dot on the end, meaning "current directory"). At this point, `svn status` will show

```
R  +    prob2.y
```

meaning that you have replaced `prob2.y` with a new file ('`R`') and that this replacement has some history that the log will reflect ('`+`'). Now when you next commit, you'll have the version of `prob2.y` as of 1521 and the log will say so.

7 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://cs164-ta@torus.cs.berkeley.edu/cs164-yu/proj1 ~/mywork/proj1
        or
svn checkout $MYREPOS/proj1 ~/mywork/proj1
```

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 SUBVERSION uses those `.svn` directories to keep track of administrative details, including the original URL, allowing commands to be much shorter.

Sometimes, however, you really need to reference a file or directory in 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 Homework #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 `~/mywork/hw2`. In this class, we've established the convention that all submissions go in a subdirectory called `tags` of your repository. The command to submit is

```
$ svn copy $MYREPOS/hw2 $MYREPOS/tags/hw2-1 -m "Homework 2, first submission"
Committed revision 1600.
```

This does not change your working directory, but it causes the entire directory tree `hw2` in the repository to get copied as a new subdirectory `tags/hw2-1` in the repository. Any future changes to `proj1` 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 $MYREPOS/hw2 $MYREPOS/tags/hw2-2 -m "Homework 2, second submission"
Committed revision 1701.
```

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

In order to encourage disciplined use of Subversion, we've configured the repository to refuse submissions by other means, such as by creating and committing a directory under `tags` as if it were part of the trunk.

8 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. Suppose that you have started homework #2 from some skeleton files I gave 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 Homework #2 in the repository directory `.../staff/hw2`, and snapshots of each "released" version of the files in directories `.../staff/hw2-1`, `.../staff/hw2-1`, etc., with the highest-numbered tag being a copy of `.../staff/hw2`.

Let's assume that there are two tags for Homework #2, `hw2-1` and `hw2-2`, and that when you copied the public version, `hw2-1` was the version you copied. Now that `hw2-2` 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 `hw2-1` and the current trunk version like this:

```
$ cd ~/mywork/hw2
$ svn merge $STAFFREPOS/hw2-1 $STAFFREPOS/hw2
U      prob1.y
C      prob2.y
```

The merge operation modifies your working directory only; the repository is not changed. The message tells you that `prob1.y` has been updated with changes that occurred between `hw2-1` and `hw2`, and that `prob2.y` has also been updated, but there were some *conflicts*—changes between `hw2-1` and the trunk that overlap changes you made since `hw2-1`. These conflicts are marked in `prob2.y` like this:

```
<<<<<<< .working
     printf ("Hello, world!\n");
     initialize ();
=======
     initialize (args);
>>>>>>> .merge-right.r1009
```

The lines between "`<<<<<<< .working` and `=======` are from your working version of the file `prob2.y`, and the ones below `=======` up to the `>>>>>>>...` line are from `hw2-2`. In this case, `hw2-1` probably contained `initialize ();` and you added a `printf` call, while the `hw2-2` tag added an argument to the call to `initialize`. SUBVERSION decided that was close enough to the new `hw2`'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 prob2.y
Resolved conflicted state of 'prob2.y'
```

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

9 Working with Teams

For projects, your team will also have a directory in the repository, as indicated in §2. The commands are analogous, but we suggest that you put the work under a different working directory, which we've been calling `teamwork`, created with the command:

```
svn co $TEAMREPOS ~/teamwork
```

You can now create subdirectories such as `proj1`, just as you did for homework, with commands such as

```
$ cd teamwork
$ svn copy $STAFFREPOS/proj1 proj1
...
$ svn commit proj1 -m "Start project 1"
```

If one of your partners has done this already, you can get a working copy of this work by typing instead

```
$ cd teamwork
$ svn update proj1
```

This command will bring in any directories in your team repository that others have committed.

Unfortunately, things get confusing if two partners independently try to initialize the project (with `svn copy`, say). In that case, the second partner will get error messages when trying to commit his `proj1` working directory. To fix the problem, the second partner can undo his work with

```
$ cd ~/teamwork
$ # Tell svn to forget about proj1
$ svn revert --recursive proj1
$ # Get rid of the proj1 files in the working directory
$ rm -rf proj1
$ # Now get the partner's version
$ svn update proj1
```

From here, you might work in the team repository directly, with a cycle like this:

```
$ cd proj1
edit, add, remove files
$ svn commit
$ svn update
edit, add, remove files
$ svn commit
$ svn update
...
```

Each `svn update` will bring in and attempt to merge any work that your partners have committed. Occasionally, a partner will have committed a file you've been modifying, in which case `svn commit` will ask you to `svn update` first to merge in those changes and resolve conflicts.

As for homework assignments, teams will submit versions by copying to the tags directory:

```
$ cd teamwork
$ svn commit proj1
$ svn copy $TEAMREPOS/proj1 $TEAMREPOS/tags/proj1-1 -m "Submit version 1"
```

9.1 Working in a branch

Of course, working like this can cause problems with the partners stepping on each others' toes and causing merge conflicts. Therefore a somewhat better way to work is to modify your own *branch* of the project. For example, you might do the following in your personal working directory:

```
$ cd mywork
$ svn copy $TEAMREPOS/proj1 $MYREPOS/proj1.0 -m "Record branching point"
$ svn copy $MYREPOS/proj1.0 $MYREPOS/proj1-sync \
      -m "Record last time I synced with the team"
$ svn copy $MYREPOS/proj1.0 proj1
$ svn commit proj1
```

and then work on your piece of the project in `mywork/proj1` rather than `teamwork/proj1`, so that your partners will never cause any conflicts. The reason for keeping an extra copy of the team project you started with in `$MYREPOS/proj1.0` around will be clear in a bit.

Every now and then, you'll want to pull in changes that your partners have made to the team version. You do this by merging changes like this:

```
$ cd ~/mywork/proj1
$ svn commit     # Always commit before merging.
$ svn update     # ... and sync with the repository.
$ svn copy $TEAMREPOS/proj1 $MYREPOS/proj1-next -m "Record next sync point"
$ svn merge $MYREPOS/proj1-sync $MYREPOS/proj1-next
resolve any conflicts
$ svn commit
...
$ svn update
```

This procedure will fetch any changes in the team version into your own personal branch without modifying the team's repository. It also records (in `$MYREPOS/proj1.1`, the precise version of the team's repository that you last merged. To set things up so that you can use the same procedure next time, finish up with

```
svn remove $MYREPOS/proj1-sync -m "Remove old sync point"
svn move $MYREPOS/proj1-next $MYREPOS/proj1-sync -m "And record new sync point"
```

When you are satisfied that your work on the project is ready to include in the team's version, apply the same process in reverse.

```
$ cd ~/mywork
$ svn commit proj1  # Always commit stuff first!
$ cd ~/teamwork/proj1
$ svn update
$ # Merge changes since the last time we merged into the team project.
$ svn merge $MYREPOS/proj1.0 $MYREPOS/proj1
resolve any conflicts
$ svn commit
$ cd ~/mywork
$ svn remove $MYREPOS/proj1.0 $MYREPOS/proj1 $MYREPOS/proj1-sync -m "Remove our branch"
$ svn update
```

Now we've merged our work into the team's directory and removed our branch. We can now start the whole cycle over again.

10 Miscellaneous

Ignoring files. One problem with `svn status` and `svn add` is that there will often be files that you do not want `svn` to track, but that both of these commands will report or process by default. We already mentioned that we have set up the defaults in your instructional accounts to ignore certain *generated files* such as `parser.y~` or `parser.o`, based on their file-name suffixes. Sometimes, however, these default rules can't cover everything. For example, in one of our skeletons, a source file named `parser.y` is used as input to a program that generates a C++ file `parser.cc`, which is eventually linked into an executable file named `apyc`. We don't want to ignore all C++ source files, needless to say, and the name `apyc` has nothing in particular to distinguish it.

To deal with this and other problems, SUBVERSION provides a way of associating *properties* with archived files and directories. Each versioned item can have any number of named properties, each with a given string value. For the current problem, there is a property applied to directories called `svn:ignore`, whose value is a string containing "glob patterns" (Unix file-name patterns) of filenames to be ignored in that directory (only in that directory; the property does not apply to subdirectories). The command

```
$ svn propedit svn:ignore DIR
```

will invoke your favorite editor (as for a log message) and allow you to compose a list of ignored files. For this example, such a list might be

```
parser.cc
apyc
*.out
```

The last line calls for ignoring all files in the directory *DIR* that end in `out` (just to show how patterns work). When you finish creating this list and exit, SUBVERSION will apply this property to the directory *DIR*, and it will get saved in the repository as well when next you commit.

Depth of updates. The particular structure we've used for your directories this semester has a drawback. If you execute

```
$ cd ~/mywork
$ svn update
```

you'll not only update any working directories you currently have, but also the `tags` directory, which will cause you to download copies of all the tags you've committed. In general, you don't want that, since by convention, you don't change tagged data. To avoid the problem, use the command

```
$ cd ~/mywork
$ svn update --set-depth=empty tags
```

This will set your `tags` working directory (it does *not* affect the repository) so that future `svn updates` do not check out the subdirectories of `tags`.

11 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 `cs164-yu`.

- You having chosen to keep working copies of your directories in directory `~/mywork`.
- You use the naming scheme described above, with 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 directories `~/mywork` and `~/teamwork`:

```
$ svn checkout $MYREPOS ~/mywork
$ svn checkout $TEAMREPOS ~/teamwork
```

Create a directory for a new assignment:

```
$ cd ~/mywork
$ svn mkdir hw2
$ svn commit hw2 -m "Set up Homework 2"
$ svn update
```

You'd put your work in `~/mywork/hw2`.

Create a directory for a new assignment from our template:

```
$ cd ~/mywork
$ svn copy $STAFFREPOS/hw2 hw2
$ svn commit hw2 -m "Set up homework 2"
$ svn update
```

Tell SUBVERSION about a new file:

```
$ cd ~/mywork/hw2
create prob2.y
$ svn add prob2.y
```

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

Tell SUBVERSION about a whole new subdirectory:

```
$ cd ~/teamwork/proj1
create directory include and files include/arith.h and util/tree.h
$ svn add util
```

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

Delete a committed file or directory:

```
$ cd ~/teamwork/proj1
$ svn del junk.c
D     junk.c
$ svn del util
D     util/tree.c
D     util/bark.c
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 ~/teamwork/proj1
$ svn status
M      include/tree.h
A      include/bark.h
```

Commit changes:

```
$ cd ~/teamwork/proj1
$ svn commit -m "Log message"
various messages
Committed revision 2000.
$ svn update
```

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 ~/teamwork/proj1
$ 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.c
```

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 `proj1-1`, and want to merge in the changes we've made to `proj1-2`.

```
$ cd ~/mywork/proj1
$ svn merge $STAFFREPOS/proj1-1 $STAFFREPOS/proj1
U      main.cc
C      parser.y
Edit any files marked C to resolve clashes
$ svn resolved parser.y # Tell svn about resolution
...
$ svn commit -m "Merged in skeleton changes between v1 and v2"
Committed revision 2009.
```

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

```
$ cd myteam/proj1
$ svn commit -m "Project 1, ready to submit"
Committed revision 2010.
$ svn update
```

and now create a submission copy:

```
$ svn copy $MYREPOS/proj1 $MYREPOS/tags/proj1-1 -m "Project 1, first submission"
Committed revision 2011.
```

Follow the same procedure for additional submissions of the same project, changing `proj1-1` to `proj1-2`, 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 one of

```
$ svn ls $MYREPOS/tags
$ svn ls $TEAMREPOS/tags
```

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

```
$ svn ls -R $TEAMREPOS/tags
```

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

```
$ svn ls -R $TEAMREPOS/tags/proj1-1
```

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

`svn co $TEAMREPOS/tags/proj1-1` *DIR*

where *DIR* is a new directory name.