Subversion HowTo

In CS 70, we use a "version-control system" called Subversion to manage collaboration between members of a programming pair, and to handle the distribution and submission of assignment materials, such as program code. This document describes what version-control systems do in general, and how to use Subversion in particular.

Version Control and Subversion

Version control systems like Subversion aim to solve two problems:

1. If you modify the contents of a file (e.g., source code or documents) the old version disappears as soon as you hit "save". Frequently people wish to either undo changes or check to see what changed, and unless you had enough foresight to make backups at the right time(s), restoration can be difficult.

2. If more than one person is working on a project, it can be difficult to figure out who changed what. Worse, if two people try to edit the same file at the same time, whoever hits "save" last will overwrite and lose the changes of the person who hit "save" first. But if everyone has their own copy of the code, trying to get a consistent copy of the code including everyone's changes can be difficult.

Version control systems solve the first problem by maintaining a centralized *repository* of files. When a file is saved ("checked in" or "committed") to the repository, not only is the new version of the file stored, but all previous versions of the file remain as well.[1] Thus, you can always ask the repository for a previous version of the file, or the differences between any two versions of the file.

Different systems use different methods to solve the second problem and prevent two people from accidentally clobbering each others' changes. One of the first revision-control systems, RCS, required you to "check out" a file from the repository before you could edit it. While you had a file checked out, the file was "locked" so that no one else could check out that file; once you were done making changes you "checked in" the file, the lock was released, and other people could see your changes and make their own.

This idea may sound reasonable, but it had some drawbacks in practice. You couldn't have two people working on different parts of the same file at one time. Also, programmers tended to accidentally leave files locked when they were done making changes, preventing anyone else from modifying the file (especially annoying if they had gone on vacation).

Thus, version-control systems like CVS and Subversion take a different approach.

1. To save space, the repository actually stores only the *differences* between each successive version of the file; by combining all this information, any particular version of the file can be reconstructed.

The Working-Copy Model

In Subversion (and its predecessor, CVS, as well as most current version-control systems), we let everyone edit files at the same time, but we do so in a way that avoids having people accidentally trample on each other's work.

Everyone gets their own "working copy" of the files in the repository, and each user can modify any of the files in their working copy at any time. While you work in your working copy, nobody else can see the changes you are making. When you're reading, you can *commit* your changes back into the repository. Similarly, you can *update* your working copy to include changes committed to the repository by others.

Subversion uses the following rules to handle the case where everyone edits the same file independently and then commits their changed file back into the repository:

1. If someone else has changed a file that you have edited, you cannot commit that file until you have updated your working copy.

2. When you update your working copy, the system tries to "merge" changes in the repository with any changes you have made. If the changes are in different parts of a file (i.e., bugs in two separate functions were fixed) then your working copy automatically reflects all the changes. If the system believes that changes in the repository and your working copy overlap, then it reports a *conflict*. Once there are no remaining conflicts (e.g., because you merged the changes by hand), you can commit your changes.

Getting Your Working Copy

The command `svn checkout` creates a working copy; it gets the files from the repository and sets up some bookkeeping data.

Usually, a single user runs `svn checkout` only once—when they first check out their working copy. But sometimes it can be useful to run `svn checkout` more than once. For example, you might want to create working copies on different computers (e.g., knuth and your own machine); to keep both copies up-to-date, be sure to commit all your changes before moving from one computer to another. (Your home directory is shared across all CS machines, so you should never need to create a second working copy just because you're moving from one CS machine to another.) But you could check out multiple working copies in different subdirectories if you really wanted to (assuming you have enough disk space); in fact, doing so can sometimes be useful if you want to look back at an earlier revision of your code.

Storing Your Changes

You can edit your working copy to your heart's content, but you should commit your changes frequently—whenever you get to a reasonable stopping point or milestone. Committing your changes provides you with a backup system, allows the course staff

to see your code when you have questions, and, most importantly, **it is how you submit your assignments**.

To commit your changes, run the command

```
svn commit
```

This command should open an editor (giving you a chance to describe the purpose of the changes you made) and then store the changes back into the central repository.

Before running `svn commit` it is a good idea to run `svn status` to see what files Subversion thinks you have changed. It will produce a listing of files, prefixed by a single-letter code:

`M` You've edited this file, so the changes will be stored in the repository.

`A` You'll be adding this new file to the repository.

`D` You'll be deleting this file from the repository.

`?` The repository does not contain this file—no changes to this file are tracked.

The "`?`" prefix is sometimes harmless and sometimes **very important**. Sometimes a "`?`" appears next to files you don't want to store in the repository such as scratch files, notes you made to yourself, or compiled programs that are created using the files in the repository.[2] If an important source file appears with a "`?`", it means that you've forgotten to add it to the repository—see *Adding Files to the Repository* for how to do so.

Checking In Individual Files

By default, many commands, including `commit`, `update`, and `status`, work on the current directory and its subdirectories. If you want these commands to work on only a particular file or directory you can specify that file; for example, `svn commit foo.c`. In general, you should usually run `svn commit` from the base of your working-copy (e.g., `~/cs70`).

Revision Numbers

Each time a change is committed to the repository, it creates a new *revision* of the repository. Revisions are numbered, using an integer, one greater than the number of the previous revision (the initial revision number for a freshly created empty repository is zero). If you know the revision number for an earlier version of the repository when your files were different, you can go "back in time" and look at those files as they were then (see *Going Back into the Past*).

Because the entire class is using (different subdirectories) of a single repository, you may notice that consecutive versions of your files have nonconsecutive numbers like 2, 17, and 32; this phenomenon just reflects other students committing changes to their own files.

2. In most cases, you don't want to include files generated from other files in the repository (e.g., compiled object files and programs, typeset documents, etc.). If running `make` creates the file, you don't need to check it in.

Retrieving Changes Made Elsewhere

If your partner made changes in their working copy, *and* committed them to the repository, you will want to see those changes in your working copy. To apply those changes to your working copy, run

```
svn update
```

For each updated item, Subversion prints a line containing a flag character and the file name. Possible flags are

- `U` A file you had not changed has been Updated with changes from the repository.
- `G` A file you have changed has had repository changes automatically merGed (i.e., you both edited different parts of the file).
- `C` There were *conflicts* between changes you made and changes found in the repository (i.e., you both edited the same part of the file).
- `A` A file was added to the repository since your last update, and it has been copied into your working copy.
- `D` The repository has been told that a file is no longer needed and so it was deleted from your working copy.

Note: You *must* run `svn update` before you run `svn commit` if your partner has made changes to the repository since you last ran `svn update`. It is sensible to get into the habit of running `svn update` before you start working (so you don't begin with stale files) and before you commit your changes.

Conflicts

If there is a conflict in one of your files it should be hand-edited. The file will be marked with sections where the changes occurred, of the form

```
<<<<<<< .mine
  ...
  lines taken from your working copy
  ...
=======
  ...
  lines taken from the repository
  ...
>>>>>>> [repository version number]
```

You should edit this section to contain a single correct version of the lines, being sure to delete the the `>>>>` and `=====` and `<<<<` lines as well. Also, to help you figure out what to do, Subversion creates some other files. If the conflicting file was `foo.c`, the

revision you began editing was number 15, and the current revision number of the repository is 27, it will create

- `foo.c.r15` — the file as it was when you started editing
- `foo.c.r27` — the file as it is in the repository now
- `foo.c.mine` — the file as you edited it, before it tried to do the merge

Once you are done merging the two versions by hand, run `svn resolved foo.c` to tell Subversion that the conflicts in that file have been resolved and remove these extra files.

Once all conflicts have been resolved, you can run `svn commit`, if desired.

Throwing Away Your Changes

Alternatively (or if you realize that you're on the totally wrong track) you can use the command `svn revert foo.c` to throw away all changes or conflicts in the file `foo.c`, resetting it back to how it was before you changed it. Be very sure that you don't mind permanently losing your changes before reverting!

Adding Files to the Repository

Subversion commands only work on files the repository knows about! If you create a brand-new file in your working copy and `commit`, the new file *will not* be stored in the repository.

You can tell Subversion that a file or directory in your working copy should be tracked by the repository. with the command `svn add` (e.g., `svn add foo.c`). By default, adding a directory automatically adds all of its files; you cannot add a file unless its containing directory has been previously added.

You can tell Subversion that a file or directory is no longer useful to anyone with the command `svn delete`. All old versions of the file *remain* in the repository, but it will not appear in any up-to-date working copies (it will be removed from other people's working copies when they run `svn update`).

You can also rename files (e.g., `svn move foo.c bar.c`). Again, everyone will see the change when they next update their working copy.

Important Notes

None of these commands changes the repository right away! The files are simply "scheduled" for addition, deletion, or renaming; the repository will be updated appropriately the next time you `commit`. If you change your mind about adding, deleting or changing a file, you can use `svn revert` on the file before committing.

Before you make changes to a file that you've renamed or added, it is a very good idea to run `svn commit` to check it in to the repository first.

Getting Information

The command `svn log foo.c` shows you the revision numbers corresponding to each commit of the file `foo.c`, including any comments that were written at the commit.

The command `svn diff foo.c` shows you the differences between the repository's version of `foo.c` and your working copy. You can compare the working copy against an old version (e.g., `svn diff -r 42 foo.c` to compare revision 42 of the file against your working copy) or compare two old versions (e.g., `svn diff -r 3:7 foo.c` to compare the differences between revision 3 and revision 7 of the file).

The output is in Unix `diff` format, which can be a little hard to read at first. The system tries to only show you sections of the files that are different. Lines that appear only in one version are prefixed with `-`, lines that appear only in the other are prefixed with `+`, and lines that appear in both have no prefix. Each such segment has a header explaining which version is the `-` one and which is `+`, and (more cryptically) where in each file the changes occur.

The command `svn annotate foo.c` (or its synonyms `svn praise` and `svn blame`) displays the contents of a file with each line annotated to show where and when that line was last changed or added.

Recall that `svn status` summarizes how your files differ from the repository. By default it only shows information on "interesting" files that can be determined without looking at the repository, but you can say `svn status -uv` for more verbose information on all files. See `svn help status` for more information on interpreting the output.

Going Back into the Past

The command `svn log` will show you what changes you've made (by showing you your log messages) and the revision numbers associated with those changes.

If you just want to display the older version of a file `foo.c` without modifying any files, you can use `svn cat -r` *N* `foo.c`, where *N* is the revision you desire. The contents of the file will be sent to standard output.

If you want to see *all* the files in a directory and its subdirectories the way they were at an earlier time, you can use `svn update -r` *N* to go back to revision *N* of the repository. Doing so will put the files the way they were at that revision: deleted files will reappear, added files will disappear, and renamed files will get their old names (but the only files affected will be the ones stored in the repository). Use `svn update` to return to the present. This command reapplies the changes stored in the repository to update your working copy to the latest revision.

Note that to commit new changes, you must be at the latest version of the repository, not back in the past.

Differences from CVS

For anyone who has used CVS before, some of the biggest differences are

- Subversion revision numbers are per repository (CVS has a separate revision counter for each file);
- Subversion has much better support for moving and renaming files without losing revision histories;
- Subversion makes conflicts harder to accidentally overlook by not letting you `commit` until you run `svn resolved`.

For More Information

You can get help on a particular command (e.g., `update`) by saying `svn help update`, or see a list of all commands by saying just `svn help`.

The entire O'Reilly Subversion book is available online for free at `http://svnbook.red-bean.com/`. The online copy is updated to reflect changes in later releases of Subversion, so it will always have the most recent documentation available.