Using Subversion in CS 131

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 you want either to undo changes[1] or check to see what changed[2], and unless you had enough foresight to make and keep a lot of different backups, this can be tricky.

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 clobber all the work 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 tricky.

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.[3] 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 prevent two people from clobbering each others' changes. RCS requires you to "check out" a file from the repository before you can edit it. While you have a file checked out, the file is "locked" so that no one else can check out that file; once you're done making changes you "check in" the file, the lock is released, and other people can see your changes and make their own.

This sounds reasonable, but it has some drawbacks in practice. You can't have two people working on different parts of the same file at one time. Also, programmers tend to leave files locked when they're done, preventing you from modifying the file (especially annoying if they've gone on vacation).

[1] "I swear it was working yesterday!"

[2] "Why is it behaving differently now? I only added a comment…I think."

[3] To save space, the repository may only keep track of the *differences* between each successive version of the file; by combining a sequence of differences, any particular version of the file can be reconstructed.

Systems like CVS and Subversion take a different approach. Everyone gets a so-called "working copy" of the files in the repository, and everyone can modify any of the files in their copy at any time. Users can *commit* their changes back into the repository, and can *update* their working copy to include changes committed to the repository by others.

How do such systems handle the case where everyone is editing the same file?

1. You cannot commit a file unless you have first updated your working copy to get the most recent changes in the repository.

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 were fixed in different functions) then your working copy automatically reflects all the changes. If the system believes 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 own changes.

This document attempts to explain useful commands for an existing Subversion repository. For more information, including how to create new repositories, see `http://www.subversion.org/`. You can also 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` .

Subversion and Versions

Subversion uses a global *revision* counter for each repository; the counter is incremented each time anyone commits changes to any file. Because the entire class is using (different subdirectories) of a single repository, you may notice that consecutive versions of your files have non-consecutive numbers like 2, 17, and 32; this just reflects other students committing changes to their own files.

Getting Your Working Copy

The command `svn checkout` is used once[4] to create a working copy; it gets the files from the repository and sets up some bookkeeping data.

Specifically, the command for CS 131 will be

```
svn checkout http://svn.cs.hmc.edu/svn/cs131f07
```

This will create a `cs131f07` subdirectory, which itself will contain subdirectories with your files.

The repository uses your normal HMC CS password.

Doing and Undoing Changes

You can edit your working copy your heart's content, but *frequently* (for backup purposes, because you have a question and you want a grutor or the professor to be able to see your code, **and to submit the final version**) you should 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.

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 say, e.g., `svn commit Design.txt` .

If you committed changes in one working copy, and you want to get these changes to appear in another (e.g., your partner's) working copy (or to get any new files supplied with a new homework assignment), the command is

```
svn update
```

For each updated item you get a line containing a flag character and the file name. Possible flags are

[4] A single user can create working copies on different computers, but 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 if you switch 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.

- `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.
- `C`: There were conflicts between changes you made and changes found in the repository.
- `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.

You may also see files prefixed with `?`. This just means the repository hasn't been told about this particular file and isn't tracking changes. Typically, such files should be things like compiled programs that are created using the files in the repository, or backup files created by your editor. If an important source file appears with a `?`, see Extending The Repository below.

If there is a conflict in a file, say `Design.txt`, it must 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 relevant lines, being sure to delete the the `>>>>` and `====` and `<<<<` lines as well. Once you are done, run `svn resolved Design.txt` to tell Subversion the conflicts in that file have been resolved.

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

Alternatively (or any time you realize that you're on the totally wrong track and want to throw away all your changes to start afresh) you can use the command `svn revert Design.txt` to throw away all changes or conflicts in the file `Design.txt` and to get a fresh copy of the file from the repository. Be very sure you don't mind permanently losing your changes before reverting!

To see files the way they were at an earlier time, you can use `svn update -r`*n* to go back to revision *n* of the repository. The command `svn update` will get you

back to the latest version. (If you just want to display the older version without modifying any files, you can use `svn cat -r`*n filename*.)

Extending 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 say that a file or directory in your working copy should be tracked by the repository with the command `svn add` (e.g., `svn add Design.txt`.) Adding a directory automatically adds all of its files; you cannot add a file unless its containing directory was previously added.

You can say 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 no longer will appear in people's working copies.

You can also rename files (so that everyone sees the change when they next `update`), e.g., `svn move design.txt Design.txt .` The advantage of `svn move` over `mv` is that the repository knows what happened; it won't start complaining that your `design.txt` is missing and that an unknown file `Design.txt` has suddenly appeared.

Note 1: None of the command in this subsection change the repository right away! The files are simply "scheduled" for addition, deletion, renaming, or copying; the repository will be updated appropriately the next time you `commit`.

If you change your mind about adding/deleting a file, you can use `svn revert` on the file before committing.

Note 2: If you use `svn update -r`*n* to go back to some earlier revision *n* of the repository, it 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. If you `update` back to the latest version of the repository, all such modifications will be reapplied.

Getting Information

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

The command `svn diff Design.txt` shows you the differences between the repository's version of `Design.txt` and your working copy. You can compare the

working copy against an old version (e.g., `svn diff -r 42 Design.txt` to compare Revision 42 of the file against your working copy) or compare two old versions (e.g., `svn diff -r 3:7 Design.txt` 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 Design.txt` (or its synonym `svn blame`) displays the contents of a file with each line annotated to show where and when that line was last changed/added.

The command `svn status` summarizes how your files differ from the repository (e.g., `M` means you have Modified the file). 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.

Differences from CVS

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

- Subversion revision numbers are per-version-of-the-repository, rather than having 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 do `svn resolved`.