CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Winter 2009

Lecture 17— Version control, shared files, `svn`

Where are We

Learning tools and concepts relevant to multi-file, multi-person, multi-platform, multi-month projects.

Today: Managing source code

- Reliable backup of hard-to-replace information (i.e., source code)
- Tools for managing concurrent and potentially conflicting changes from multiple people
- Ability to retrieve previous versions

Note: None of this has anything to do with code. Like `make`, version-control systems are typically not language-specific.

- Many people use version control systems for everything they do (code, papers, slides, letters, drawings, pictures, . . .)
  - Traditional systems are best at text files (comparing differences, etc.); newer ones are better with other kinds.

Version-control systems

There are plenty: `scss` (historical), `rcs` (mostly historical), `cvs` (built on top of `rcs`), `subversion`, `git` (much more distributed), `SourceSafe`, ...

The terminology and commands aren't particularly standard, but once you know one, the others shouldn't be difficult — the basic concepts are the same.

`cvs` had the biggest mind-share over the last decade (particularly in the open-source community), but `svn` (subversion) improves on its shortcomings (particularly handling renaming files or directories while retaining version history) and is the current mainstream VCS.

We'll learn the basics of `svn`.

The set-up

There is a svn *repository*, where files (and past versions) are reliably stored.

- Hopefully the repository files are backed up, but that's not `svn`'s problem.

You do *not* edit files in the repository directly. Instead:

- You *check-out* a *working copy* and edit it.
- You *commit* changes back to the repository.

You use the `svn` program to perform any operations that need the repository.

One repository may hold many *projects*. A subversion repository is just a database of projects and files.

Questions

- How do you set-up:
  - A repository (`svnadmin create`)
  - A project in a repository (`import`)
  - A working copy of a project in a repository (`checkout`)
- How do you edit files:
  - Get latest updates of a project (`update`)
  - Add or remove files (`add` or `remove`)
  - Put changes back in repository (`commit`)
- How do you get information about:
  - History of revisions (`log`)
  - Difference between versions (`diff`)
- Other (branches, locks, watches, ...)

Common vs. uncommon

Learn the common cases; look up the uncommon ones.

In production shops:

- You will set up new repositories approx. once every 5 years
- You will add a project approx. once a year
- You will checkout a project approx. once a month
- You will update your working copy and update the repository approx. once a day.

Nonetheless, the command-structure for all these is similar:

`svn` *`svn-options`* `cmd` *`cmd-options filenames`*

Getting started

Set up a repository and project.

- Remember, everyone has to look up the commands for this.

Accessing the repository:

- From the same machine, just specify the root via a path name url.
- After the checkout, the working-copy "remembers" the repository
- Can access remotely by specifying user-id and machine.
  - Must have `svn` and `ssh` installed on your local machine
  - Will be prompted for password or use other ssh authentication.
  - How to write code with other people in other places.
  - Recommendation: Figure out how to use `svn` locally on the same machine first (`attu` for next homework). Remote is easy enough, but adds some extra complexity.

Working with the repository

- Set up a repository (your choice of repository name and location)

  ```
  svnadmin create ~/svnrepos
  ```

- Put a project directory in the repository (use name of your project directory, path to repository)

  ```
  svn import proj file:///homes/iws/me/svnrepos -m ...
  ```

- Check out project to a working directory

  ```
  cd working_directory
  svn checkout file:///homes/iws/me/svnrepos  proj
  ```

  Repository location is remembered in working directory now

File manipulation

- Add files with `svn add`.
- Get files with `svn update` (bring local working copy up to date).
- Commit changes with `svn commit`.
  - Any number of files (no filename means all files in directory and all transitive subdirectories)
  - Added files not really added until commit

Commit messages are mandatory:

- `-m "a short message"`
- `-F filename-containing-message`
- else an editor pops up if you have set the `EDITOR` or `VISUAL` environment variable
- otherwise `svn` complains

Working with files: Examples

- Update local working directory to match repository

```
svn update
```

- Make changes

```
svn add file.c
svn move oldfile.c newfile.c
svn delete obsoletefile
```

- Commit changes

```
svn commit -m ‘‘this is much better’’
```

- Examine your changes

```
svn status
svn diff file.c
svn revert file.c
```

Conflicts

This all works great if there is one working-copy. With multiple working-copies there can be *conflicts*:

1. Your working-copy checks out version 17 of `foo`.
2. You edit `foo`.
3. Somebody else commits a new version (18) of `foo`.

Subversion tries to merge changes automatically; if it can't you must resolve the conflict. If `svn commit` fails:

- Do `svn update` to get repository version and attempt merge
  - "G" means the automatic merge succeeded
  - "C" means you have to resolve the conflict
- Merging is *line-based*, which is why `svn` is better for text files.
- Conflicts indicated in the working-copy file (search for <<<<<<).

SVN gotchas

- Do not forget to add files or your group members will be very unhappy.
- Keep in the repository *exactly what you need to build the application*!
  - Yes: foo.c foo.h Makefile
  - No: foo.o a.out
  - You don't want versions of .o files:
    * Replaceable things have no value
    * They will change a lot when .c files change a little
    * Developers on other machines can't use them

Summary

Another tool for letting the computer do what it's good at:

- Much better than manually emailing files, adding dates to filenames, etc.
- Managing versions, storing the differences
- Keeping source-code safe.
- Preventing concurrent access, detecting conflicts.