ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 11 — January 28, 2013

Patrick Lam *version 1*

Version Control Systems

We've made you use *version control* for labs and assignments, but just as a dropbox. In this lecture, we'll talk about some of the theory of operation behind the use of version control systems.

- Ever wanted to undo your changes to software?
- Ever needed to collaborate with others to develop software?
  - without having to email new versions of files to each other and manually integrate changes?

Version control lets you do that!

Conceptually, version control stores all of the versions of a set of files in a *repository*. You can backtrack to any previous version. You can also investigate previous versions by author, date, or commit comment. Furthermore, you can merge the changes in your local copy with some other copy of the set of files, and commit a new copy, integrating the changes.

Workflow. While there are two models, we'll only talk about the *copy-modify-merge* model.

1. To start working on a project, you check out (*copy*) a version of the project, usually the most recent one, to create your working copy.
2. Next, you *modify* your copy of the projects, test your changes, and commit your changes to the repository.
3. Other developers can then *merge* your changes into their working copies, which contain their changes. They will then commit their changes as appropriate.

Yes, merging works. Have you tried it?

Merging can be a hassle: although it usually succeeds on its own, there is occasionally a *merge conflict*, where the version control software can't figure out what the right thing should be. In that case, you'll have to look at three things: 1) the common ancestor; 2) your change; and 3) the other change. You can then decide what the proper result should be. We'll talk about merge conflicts a bit more below.

Another model is the *lock-modify-unlock* model, but it's obsolete.

Distributed versus centralized. Traditionally, there was one canonical central repository for a software project. Old-school *centralized systems* work on that model. So, you copy your version from the central repository, and commit your changes back to that repository.

Newer-school version control systems are *decentralized*. Every developer can keep a full copy of the project's history on their system, and there doesn't need to be a central repository anymore (although one often exists, in some sense). You can therefore commit changes to the history on your local computer, without network access. I do this for my own copy of the ECE 155 repository, which I maintain with Git, and then copy files to the Subversion repository that's available to you.

Case Study: Subversion

Subversion is a relatively easy-to-use version control system, and you've been using it at least to submit source code for the labs. I'm going to talk about command-line usage; you've been using `subclipse`.

Here's a useful reference for Subversion:

Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato. *Version Control with Subversion*. `http://svnbook.red-bean.com/`. Accessed January 27, 2013.

Getting started. First, you need a copy of a repository. You can create one from scratch and then check it out, or you can check out an existing repository.

To create a repository from scratch, use the `svnadmin` command, e.g.

```
svnadmin create c:\svn\repos
```

where you've previously created the empty `c:\svn` directory. Now you have a repository. Often you won't need to do this, because someone else will already have created a repository for you. You either check out your new repository, or someone else's repository, before working on it.

To check out a repository, use the `svn checkout` command, e.g.

```
svn checkout http://k9mail.googlecode.com/svn/k9mail/trunk/ k9mail-read-only
```

This creates a *working copy*; in this case, the working copy is in the subdirectory `k9mail-read-only`.

Adding files. There's an `import` command, but I don't recommend its use. It is for bringing in a collection of files that weren't previously version controlled. Usually, you want to `svn add` new files, to indicate that they should be in the repository. A leading cause of build breakage is forgetting to `add` new files.

Ignoring files. Tip: editors and compilers always generate bogus extra files that you don't want to commit, like `.obj` files. You can ignore them with `svn propedit svn:ignore .`, and adding appropriate wildcards (e.g. `*.obj`) to the ignore list.

Committing files. Once you're done with your changes, you need to commit them to the repository. Use the `svn commit` command to do so. It will prompt you for a commit message, or complain that it can't get one from you. Good commit messages are really important[1].

[1] `http://who-t.blogspot.com/2009/12/on-commit-messages.html`

Updating your repository. You can pull changes from the repository to your working copy. Use `svn update` to do that. If all goes well, you'll get output like this:

```
plam@noether:~/production/11.aosd.modularity$ svn up
D    example.tex
A    studies.tex
U    introduction.tex
A    sketch.tex
U    main.tex
```

The first letter is important. The bad letter is `C`, denoting a conflict. In that case, you can start by swearing at the fact that you have to resolve conflicts. Then, you need to go look at the offending file, which will have conflict markers. The start of the conflict will have a bunch of `<<<<<<`s, followed by one version of the change, followed by `======`, and then followed by the conflicting version. The end-conflict marker is `>>>>>>`.

You resolve the conflict by going into your editor and figuring out what the file should actually contain. Then you tell svn that you've resolved the conflict, with `svn resolved`. That allows you to commit your merged changes.

Stepping back in time. "Oops! I did it again!"

Oh no, you committed a bogus change! You can look at previous versions by using `svn update` and passing it the `-r` parameter. This parameter takes a revision number. Want to know what's in which revision? Use `svn log` and read the awesome commit messages you wrote earlier. Or, you can pass `-r` a date, e.g. `-r {2011-01-04}`.

Note that `svn update` with a version number just checks out the version you specified. You can't commit that version. If you want to merge a previous version with the latest version, use `svn merge` rather than `svn update`.

Inspecting diffs. A *diff* shows the difference between two versions of a file. I recommend reviewing diffs before commiting changes, to avoid embarassment and to commit minimal diffs (exclude whitespace). Here is an example of a diff.

```
===================================================================
--- Text/abstract.tex (revision 17379)
+++ Text/abstract.tex (working copy)
@@ -1,10 +1,10 @@
 Runtime monitoring enables developers to (1) specify important program
 properties and (2) dynamically validate that these properties hold.
 In recent research, we have found that static analysis techniques can,
-in many cases, verify that runtime monitors never trigger.  In
-this paper, we describe a system which enables developers to visualize
-the remaining cases---potential
-points of failure for runtime monitoring properties. Our system
-graphically displays the automata associated with specified runtime
-monitoring properties and enables developers to inspect (and, if
-necessary, fix) the code at each automaton transition.
+in many cases, verify that runtime monitors never trigger.  In this
+paper, we describe a tool which enables developers to visualize the
+remaining cases---potential points of failure for runtime monitoring
+properties. Our tool graphically displays the automata associated with
+specified runtime monitoring properties and enables developers to
+inspect (and, if necessary, fix) the code at each automaton
+transition.
```

Lines with `+` are new, while lines with `-` went away.

Basic workflow

The basic work cycle of SVN is the following:

- Update your working copy of the repository using `svn update`.
- Edit files. Manipulate set of files with `svn add`, `svn delete`, `svn copy`, and `svn move`.
- Examine changes using `svn status` (to list changed files) and `svn diff`.
- Undo changes, if necessary, using `svn revert`.
- Commit changes using `svn commit`.