CSE 303: Concepts and Tools for Software Development

Dan Grossman
Winter 2006
Lecture 19— Version control, shared files, cvs
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., the source code)

- Tools for managing concurrent and potentially conflicting changes from multiple people

- Version numbers, ability to “rollback” or at least see the differences with previous versions.

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

- I use cvs (a version-control system) for papers I write too.

- But cvs is better at plain-text (for detecting differences)
Version-control systems

There are plenty: \texttt{rcs}, \texttt{cvs}, \texttt{subversion}, \texttt{sourceforge}, \texttt{SourceSafe}, ...

The terminology and commands aren’t so standard, but once you know one, the others shouldn’t be difficult.

\texttt{cvs} is actually a layer over \texttt{rcs}.

Weak-point of \texttt{cvs}: renaming files or (worse) directories.
The set-up

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

- Hopefully the repository files are backed up, but that’s not cvs’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 cvs program to perform any operations that need the repository.

One repository may hold many projects. (The repository itself just has a directory structure.)
Questions

• How do you set-up:
  – A repository (init)
  – 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, ...)

Dan Grossman  CSE303 Winter 2006, Lecture 19  5
Common vs. uncommon

Learn the common cases; look up the uncommon ones:

- 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:

```
cvs cvs-options cmd cmd-options filenames
```

Examples:

```
cvs -d ~djg/cvsroot checkout foo
cvs update -P foo
```
Getting started

Set up a repository and project.

- Remember, I have to look up the commands for this.

Accessing the repository:

- From the same machine, just specify the root via a path name (-d).
- After the checkout, the working-copy “remembers” the repository so -d is unnecessary.
- Can access remotely by specifying user-id and machine.
  - Must have cvs and ssh installed on your local machine
  - Will be prompted for password.
  - How I write code with people in other time zones.
  - I recommend you not spend the time to set this up for hw6.
File manipulation

- Add files with `cvs add`.
- Get files with `cvs update`.
- Commit changes with `cvs commit`.
  - Any number of files (no filename means all files in directory and all transitive subdirectories)
  - Added files not really added until commit (unlike directories)

Commit messages are mandatory:

- `-m "a short message"`
- `-F filename-containing-message`
- else an editor pops up
  - `vi` unless you set environment variable `VISUAL` (or `EDITOR`?)
  - learn how to quit `vi :)` (or learn `vi`)
Conflicts

This all works great if there is one working-copy: you keep old versions, can see their differences, etc.

With multiple working-copies there can be conflicts:

1. Your working-copy checks out version 1.7 of foo.
2. You edit foo.
3. Somebody else commits a new version (1.8) of foo.

You cannot commit; you must update foo. What about your changes?

- If you’re nervous, make a copy of foo locally first.
- But cvs will use diff and patch to merge the changes between 1.7 and 1.8 into your working-copy foo.
- Merging is line-based, which is why cvs is better for text files.
- Conflicts indicated in the working-copy file (search for <<<<<<<<).
It’s all just files and diff

There is very little magic to cvs; you can poke around to see how it’s implemented:

• The repository just uses directories and files.
• Files are kept read-only to avoid “mistakes” (cvs command temporarily changes that and changes it back)
• Files are kept in terms of diffs (so small changes lead to small increase in repository size, even for large files)
• Set group permissions appropriately (see chgrp if necessary).
• Hard part of implementation is preventing simultaneous commits and other concurrency errors.

As for the working copy:

• All the “magic” is in the CVS subdirectory.
CVS gotchas

- To get new subdirectories do `update -P` (for hw6, one directory should be plenty).

- 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.

How to “cheat” and throw away somebody else’s changes – don’t!

mv foo bar
  cvs update foo
mv bar foo
  cvs commit -m "muhahaha" foo

cvs just knows about version numbers and diff; it’s not magic.