CS108, Stanford
Fall, 2008-09

Handout #37
Osvaldo Jiménez

Source Control CVS

Thanks to Nick Parlante for much of this handout

Source Control

- Any modern software project of any size uses "source control" (also known as "revision control")
- Store all past revisions
    - Can see old versions, see the trail of changes over time
    - Also a form of backup
- Free yourself to delete! If all the sources are in source control, you can delete or change lines without hesitation, knowing you can always get back to the old version (vs. keeping the old lines scattered around but commented out).
- Allow multiple people to edit source base at the same time
- Very Old style: lock/unlock -- only one person at a time can change a file.
- New style: everybody just makes changes on their local copy, "merge" things together when trying to commit to the repository.
- CVS is an extremely popular, open source, source control system. A newer system called "subversion" (SVN) is becoming popular. Subversion is deliberately very similar to CVS, using the same vocabulary, but works better in some ways. In any case, CVS is a great first system to learn.

Source Control Components

- "Repository" -- single, centralized store of all the past revisions of the sources
- "HEAD" version -- the set of files in the repository made of the most recent version of every file
- "Local copy" -- the local copy of the file that you edit, compile, run etc.
- "Update" or "Check out" -- get files from the repository down to your local copy
- "Commit" or "Check in" -- send your local files up to the repository

Leland Setup

- Having a leland account allows you to very easily set up a CVS repository.
- Choose one person to own the cvs repository on their account, and that owner does these steps to set things up. No editing should ever happen by hand to the files in the `.cvsroot` directory. Only access the `.cvsroot` directory through the `cvs` commands.
- The owner creates a cvsroot directory in their account, like this

```
mkdir ~ojimenez/.cvsroot
```

- The owner should set a CVSROOT environment variable in their `.cshrc` like this, pointing to the owner's directory. Then "`source .cshrc`" to set the variable. (Others only need to do this if they want to use command line tools to access cvs from their leland account. Going through the Eclipse/CVS module does not depend on a shell CVSROOT variable)

```
setenv CVSROOT ~ojimenez/.cvsroot

Or bash shell style:
export CVSROOT=~ojimenez/.cvsroot
```

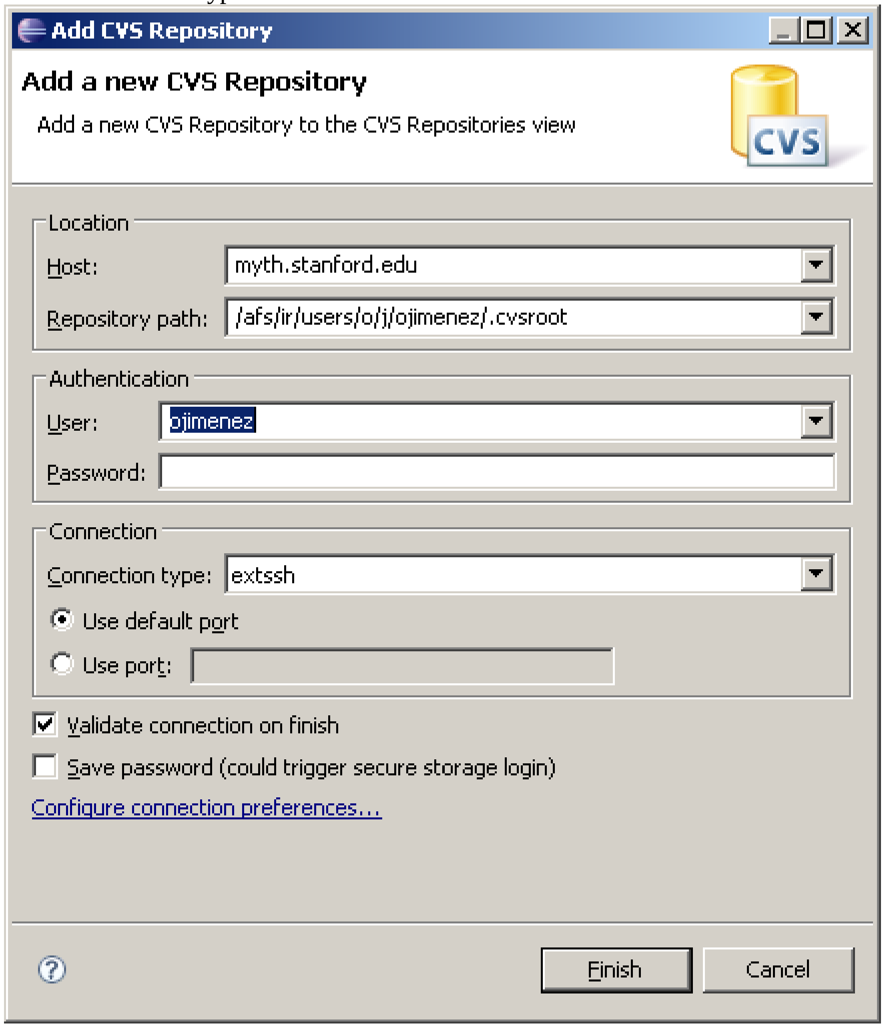
- The repository owner then does a onetime setup with the command

```
cvs init
```

- Then the owner must give the other people, e.g. alice and bob, AFS permission for the repository with the commands
```
  fsr setacl -dir $CVSROOT -acl bob all
  fsr setacl -dir $CVSROOT -acl alice all
```

Eclipse Setup

- In the old days, CVS was done all through the command line, but we will go through Eclipse's nice CVS plugin. (the terminology is the same between the command line and plugin).
- Open the *CVS Repository perspective*
- Right click (or control-click with a one-button mouse) – *new-> Repository Location*
- The repository path is a full leland path like
```
  /afs/ir/users/o/j/ojimenez/.cvsroot
```
- The connection type should be **extssh**


Add CVS Repository

Add a new CVS Repository

Add a new CVS Repository to the CVS Repositories view

Location

Host: myth.stanford.edu

Repository path: /afs/ir/users/o/j/ojimenez/.cvsroot

Authentication

User: ojimenez

Password:

Connection

Connection type: extssh

[✓ Use default port]

Use port:

[✓] Validate connection on finish

[ ] Save password (could trigger secure storage login)

Configure connection preferences...

Initial Add

- One person does this to put the project in cvs
- Right-click the project in java mode, and select *Team->Share Project*
- Select the existing repository location, and go to the next page (or add the repository connection first if needed, as above)

- Select "*Use project name as module name*", and go to the next page
- On this page, see files in project -- want to add/commit them to be under CVS control
- It's ok to add the `.project` and `.classpath` files to cvs
- When it's about to do the commit, it will ask for a change comment like "initial add files"

Others Update/Checkout

- To get the files from the repository, open the perspective: CVS Repositories Exploring
- Add a new CVS Repository, filling in host, username etc. as before (each person uses their own username etc.)
- Under the HEAD revision, right click on the project to Check Out
- Check out as project in workspace -- go back to the Java perspective and there's the code

Binary Files

- CVS does end-of-line conversion which is fine for text files, but which will mess up binary files like .jpeg images
- To fix this, go to *Windows->Preferences. Select Team->File Content preferences* page, and make sure that the file extension (".jpeg" ...) is marked as binary (most of the common binary formats are already there)
- You do not have to store binary files (jpegs) in cvs -- you can just copy them around as files if you prefer, leaving just the .java etc. in cvs.

Team Synchronize View

- Right-click on the project and go to *Team->Synchronize With Repository*. This should switch to the "*Team Synchronizing*" view.
- Or just switch to the Team Synchronize view using the button at the far upper right, or use the *switch perspective* menu
- This view shows the differences between the local copies and the repository
- The arrow to the right represents committing local copies to the repository
- The arrow to the left represents updating changes from the repository to the local copies
- Even if not making any changes, use this view to see what the differences are
- Double click a file to see the "diff" view comparing the local copy to the repository copy

Getting Most Recent Version From Repository (Update)

- Go to synch view to see list differences
- Click the far left "*synchronize*" button to refresh the list of differences
- Click "*Update All Incoming Changes*" (leftward arrow) to pull down new versions, changing the local copies. This is also known as "check out".

Standard Edit, Commit cycle

- Start with up-to-date sources
- Do edits, testing, add features etc. (time passes)
- Before committing your changes, do an *Update* (above) to bring down any recent changes from the repository, so you can make sure those changes mesh with your changes all in your local copy.
- With your local copy working with the latest of everything, it's time to commit.

- To commit those changes to the repository, bring up Synchronization view
- Click the rightward *Commit All Outgoing Changes* button, add a little comment like "fixed XML file saving" or whatever. This is also known as "check in". If the command warns of "conflicts" you need to resolve those first. If you do an update immediately before a commit, you should not see conflicts.
- Only commit code that compiles and runs, since as soon as you commit it, others will get it on their next update.
- Committing something broken is regarded as a real team-programming fuax pas, known as "breaking the build".
- After the commit, the synch view is empty, since you are up to date

Edit/Conflict Sequence Demo

- Both Alice and Bob are up to date
- Bob and Alice both start making changes
- Bob commits first -- it goes through fine
- When Alice tries to update or commit, there are "conflicts"
- This shows up as red markers in the sync view
- There are two ways to resolve...

1. Resolve in Java Editor

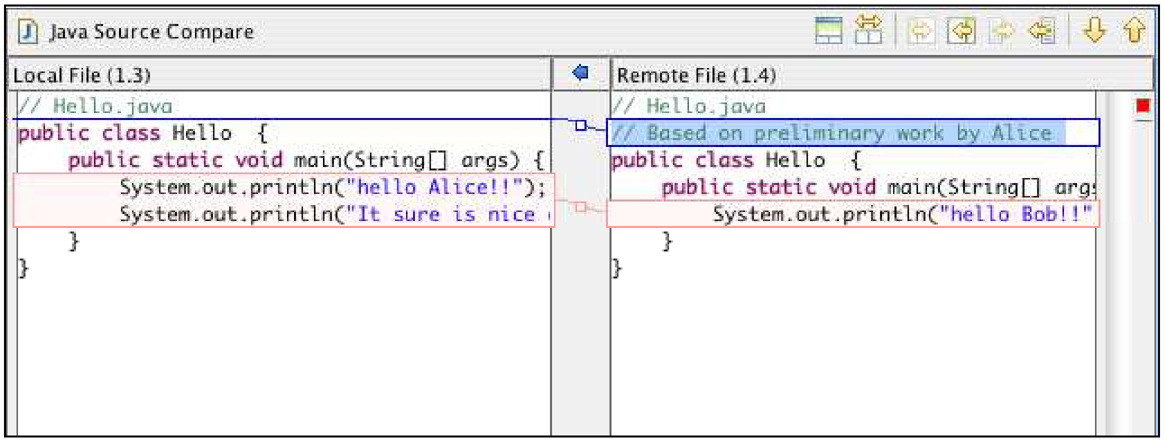
- Go back to the *Java perspective*
- Right click project, *Select Team->Update*
- This copies text down to the Java editor showing both versions for the conflicting lines -- <<<<< followed by your local version, followed by ==== and the version that was up in the repository.

```
<<<<<<< Hello.java
        System.out.println("hello Alice");
=======
        System.out.println("hello Bob");
>>>>>>> 1.8
```

- "Resolve" the two versions, leaving just one "merged" version in the file, and delete all the <<<< markers
- Do not commit it right away -- compile, run, etc. to see that the merged version actually compiles and runs.

2. Resolve in Synch View

- Another way is to look at the changes inside the synch view
- Double click the red/conflict file in the sync view, which shows a diff of the two versions...



```
// Hello.java
public class Hello  {
    public static void main(String[] args) {
        System.out.println("hello Alice!!");
        System.out.println("It sure is nice
    }
}
```

```
// Hello.java
// Based on preliminary work by Alice
public class Hello  {
    public static void main(String[] arg
        System.out.println("hello Bob!!"
    }
}
```

- Left is our local copy, and we see how it is different from the right, repository version
- Click "*Copy all non-conflicting*" button to copy over the lines from the right that do not interfere with the left automatically
- We still have the `println()` commands to resolve manually -- just do edits in the left hand pane and hit save.
- When done, right-click the file in the CVS view and select *Mark As Merged* to signify that is the merged version you want to use going forward
- Go back to Java mode and compile/run to see that the merged version works right and then commit it normally

Revert File to Previous Revision

- To change a file back to a previous version, right click the file and use *Replace With Revision* to get back to an old revision of that file.
- Then compile and run normally to try out that revision.
- Then in the sync view, mark that copy as resolved and commit it.
- This technique works one file at a time -- do not try it on the whole project.

Look at Old Revision (read-only)

- You can temporarily change your whole workspace to show an old version of the project, but you cannot commit changes from there.
- Right click the project and choose *Replace With > Another Branch or Revision*.
- You can choose an old version by date or by a tag name
- Commit operations will not work from here -- replace with the Head revision to get back to normal
- If you want to go back to the past and edit from there, use the technique above, file by file.
- Alternately, in a low-budget way, you can use copy-paste to save old revision text and then paste it into the head version of files that can then be edited.

Other Things To Do

- In the Java perspective, right click a file and select *Replace->Latest from HEAD*
  - This just blows away the current copy with the HEAD version
  - Do this to discard your local version without bothering with doing a resolve
  - This also works for the whole project -- to discard everything local and just pull the latest of everything
- Use the *Show History* command on a file to see its revision history
- Add a tag to mark a moment in time -- tags show up in the histories

Thing Not To Do

- Do not edit the files in `.cvsroot`
- Do not modify the CVS directories or their contents
- Do not mess around with branches

Links

- Try the Eclipse online help -- has a whole cvs section
- Eclipse CVS FAQ -- http://wiki.eclipse.org/index.php/CVS_FAQ
- CVS docs and help: http://www.cvshome.org/

Aside: Distributed Source Control

- Traditional source control systems like CVS, have a central server that contains the one repository of the sources. Clients check out and check in against the central server.
- A few large, complicated projects such as the linux kernel (and the open source repository of the Java) use very new "distributed" source control systems (linux uses "git" and java uses "mercurial" -- these are not trivial systems to learn). In those systems, each computer has its own repository, and they are all peers -- there is no special, central repository. Changes can be sent from any repository to any other repository.