Version Control Using Subversion

What Is Version Control?

Version control is also known as revision control. Version control is provided by a version control system (VCS).
VCS:

- lets you track how your files change over time.
- lets you back up every version of a file.
- allows branching and merging
- lets you revert to earlier versions of a file.
- lets you work with other people on the *same* repository of files
- lets you see who made which changes

What kind of files? Text files (including source code, LaTeX, reStructuredText, etc)
Usually you will *not* put binary files under version control.

Why Subversion?

Collaboration Several people can simultaneously contribute to a single document. Documents are *not* locked!

Availability Documents are securely accessible in a single place: the repository. This is the official master copy.

History All committed versions of a document are maintained forever

Collaboration

- Versions of a document are tracked in a single place, the repository
- Documents are accessible over the internet (via SSH tunnel)
- Changes made to the same document by different users are usually merged automatically.
- Occasionally this must be done manually.
- Hint: use frequent updates and the UMUTC workflow to minimize manual merging. (Update, Modify, Update, Test, Commit)

History

- SVN tracks the entire evolution of a document.
- additions, deletions, and changes to a document are tracked on a line-by-line basis
- incremental changes to a document are committed under a new revision number each time
- the date and time of a new revision is maintained along with the user who committed it

Using Subversion: Server Side

We will not be concerned with the server side.

Creating a Repository

A class depository has already been created for you.

Repository URL has the format:

```
svn checkout
  https://subversion.american.edu/svn/econ450650s2015
  econ450s2015
  --username <userid>
  --password <pw>
```

<userid> substitute your username

<userid> substitute your password

host The host where the repository resides

project The name of the project in the repository

Working Copy

working copy: an ordinary directory on your computer, but SVN adds to special subdirectories named `.svn`.

checkout

- pick a name for your working directory.
- `svn checkout [URL] [working directory name]`

This command gets the latest versions files contained in the repository associated with URL (as described in the previous slide).

Adding and Removing Files

- `svn add [file]`
- `svn delete [file]`

IMPORTANT: files are not actually added to or deleted from the repository until your next `commit`.

IMPORTANT: do not use spaces in filenames (to keep things simple)

Keep Your Code Current

Use the following sequence when you start working on your code.

- update your working copy
- make changes and test them
- update your working copy
- test that your working copy still functions
- commit your changes

Workflow

Follow the UMUTC workflow:

- update
- modify
- update
- test
- commit

Update

You can update all files in your working directory to the latest revision in the repository, or just a single file. The `update` command can also fetch a revision different than the latest revision with the `-r` flag.

- `svn update`
- `svn update [filename]`
- `svn update -r n [filename]`

Here `n` is the desired revision number.

Modify

- Edit your working copy (and save your changes to the file)
- Add or delete files.

Update

- After saving your changes, close the application you are using to edit the file. (This is just to ensure your application does not lock the file and that you end up working on the changed version.)
- Again run `svn update` to ensure your changes are compatible. (See the section on Merging.)
- Open the file in your editor.

Test

- if you will commit code that should function, test that it does

Committing Changes to Existing Files

1. After editing files and saving your changes, but **before** committing them, do another update (following the steps above) to make sure you changes are compatible with any commits that took place while you were working.
2. commit your changes to the project directory:
   `svn commit message="a commit message"`

If you omit the `message`, Subversion will try to start an editor to ask for a commit message. If you also have not set your editor, Subversion will refuse to commit.

Precaution

Make sure (!!) you save your work and close your editor before you update or commit.

Backing Out of a Commit

Suppose you commit code that you should not have: it breaks everything. SVN lets you back out of that commit as follows.

1. extract the old version
2. re-commit the old version

E.g., suppose version 314 was the last "good" version of the code. Return your working directory to that version and recommit as follows:

```
svn update -r314
svn commit -message "Discard a stupid commit."
```

Merging

Suppose you update after you have changed your copy, but the master copy has changed as well. SVN tries to merge the two sets of changes.
Usually the changes are to unrelated areas of the file, and this succeeds.
If the changes overlap, SVN will merge what it can, and then ask you what to do about the rest.
You may choose to

- accept the repository's changes ('tf', or "theirs-full").
- override the repository's changes ('mf', "mine-full")
- postpone the decisions ('p')

Conflicts

Postponement is the safest action, but it will mark affected files in a "conflicted" state and insert blocks that look like this:

```
@@ -1 +1,5 @@
  def foo():
+<<<<<<< .mine
+    bar1();
+=======
+    bar2();
+>>>>>>> .r314
```

Say this lines are in `foo.py`. They mean that your copy of `foo.py` changed your function `foo` to call a function `bar1`, whereas someone already changed the repository, in revision 314, in exactly the same place, with `foo` calling `bar2`. You will have to resolve this conflict before you can commit your changes.

Warning

You can pick your version, or the repository version, or some other resolution. When you finish, run `svn resolved` to tell SVN that you have resolved the conflicts. Then you can commit your changes.

It is a good idea to run `svn diff` after an automatic merge. Automatic merging very seldom fails to be correct, but you want to catch any problem however rare.

Merging by hand is unpleasant. This can usually be avoided by frequent updating. Merging by hand is particulary unpleasant when a lot of code is involved. This can usually be avoided by frequent committing.
If you must make many changes to a file, it is a good idea to warn your teammates, who might have pending modifications that they will want to commit before you act.

Helpful Information

`svn status` gives information about which files are changed or new.

`svn log [file]` displays commit messages for all revisions, in chronological order, along with the associated revision number.

`svn annotate [file]` shows which users made which changes, line by line with the revision number and associated user name.

`svn help [command]` displays help on any command

GUI Subversion Clients

- TortoiseSVN For Windows
- Subclipse For Eclipse IDE
- Netbeans contains a Subversion integration module

Subversion Red Book

Version Control with Subversion. CollinsSussman, Ben and Brian W. Fitzpatrick and C. Michael Pilato. `http://svnbook`.redbean.com/

Example: Adding a New Folder

This example uses the Windows `cmd` shell. Mac and Linux users will just use their `bash` shell instead.

- open a command shell (on Windows: Start > All Programs > Accessories > Command Prompt)
- use the *cd* command to change to the folder holding your working copy `http://www.wikihow.com/Change-Directories-in-Command-Prompt`
- you should always update before making changes, so ask Subversion to update your working copy by entering `svn update`
- use Subversion to create your personal directory by entering `svn mkdir yourNewFolderName`
- finally, ask Subversion to commit your new personal directory to the repository by entering `svn commit -m "committing personal directory"`.
  note: the string folowing `-m` is just a message, but Subversion will require you to include a message when you commit.

TortoiseSVN

Windows users may find it useful to use TortoiseSVN.

1. Download TortoiseSVN (32bit or 64bit, depending on your operating system): `http://tortoisesvn.net/downloads.html` (Do NOT (!!!) use the advertisement links near the top of the page.)
2. Run the TortoiseSVN installer you receive from the download. (You need to have administrator privileges to do this. Unless you turned them off, you should have them, assuming you are on your own computer.) During installation, be sure that you also install the command-line client tools! (The default installation will not include it; you need to add it in the Custom Setup dialog.)

TortoiseSVN ...

1. Create your "working copy" of our repository. This is where you will work on your code. Decide where on your computer you want to keep your code. (Wherever it is now, you will move it to this new place.) For an example, see `http://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-dug-checkout.html` (Replace the first two lines appropriately.) Click OK.
2. Now add an existing code to your new working copy and commit it. (Just drag your files into the new folder, SVN add them, and SVN commit them. `http://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-dug-add.html` `http://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-dug-commit.html`

Locking

If the likelihood of conflicts becomes large enough for some files, users of a Subversion repository can agree to lock those files while working on them. Here is some background: `http://svnbook.red-bean.com/nightly/en/svn.advanced.locking.html`

Distributed Version Control

Subversion uses a centralized model for version control. Distributed version control systems (especially git) have become very popular. Each has advantages and disadvantages.
The arguments over the best approach continue:

- `http://www.ianbicking.org/distributed-vs-centralized-scm.html`
- `http://www.developer.com/open/subversion-1.8-gits-new-features.html`