CVS

Lecture 21
CVS: Concurrent Version System

- Classic tool for tracking changes to a project and allowing team access
  - Can work across networks

- Key Idea: Repository
  - The place where originals and all modifications to them are kept
  - A new team member checks out their own, private copy from the repository
  - Everyone can commit changes from their own copy to the repository
  - Everyone can update their own copy with the latest changes in the repository
Motivation

- Team-based development
  - Developers share and extend common code base
  - Team members comply with standards (coding conventions, comment templates,...)
  - Bug fixes applied to deployed version 1.0 while development continues, in parallel on version 2.0
- Every team project needs some kind of code management and versioning system
Key Idea: The Repository

- Repository holds master copy of all files
  - Never edited directly
  - Stores history too
- Developers have local copy in their own workspace
  - All work occurs here
- Update:
  - Bring local copies up to date with repository
- Commit:
  - Send local edits to repository
Conflicts and Merging

- Optimistic team model
  - Anyone can modify any file any time (no locking)
  - Most edits can be safely merged automatically
  - Assumption: real conflicts are rare

Conflicts and Merging

- Optimistic team model
  - Anyone can modify any file any time (no locking)
  - Most edits can be safely merged automatically
  - Assumption: real conflicts are rare

Conflict: requires attention

Error: working version out-of-date

Merge
Tagging, Branching, and Merging

- Repository is a *tree* of versions
  - Development of main product occurs as a series of revisions along trunk
- A *tag* names a particular revision
  - Once tagged, a version is *immutable*
- *Branches* off of trunk or off of other branches
  - Bug fixes of a particular release
  - Exploring different development paths
- Branches can be *merged* back to trunk
  - Speculative direction pans out
A History of Revisions
Overview of Workflow

- Create and initialize the repository
  - Once, by 1 person

- Add repository location to Eclipse
  - Once, by each team member

- Populate repository/local project with content:
  - Once, by 1 person: Put existing local project in repository
  - Once, by every other team member: Check out existing project from repository to local machine

- Synchronizing with the repository
  - Repeated frequently by everyone
    - Update local files from repository
    - Run all unit tests
    - Make changes in local project files
    - Run all unit tests (make sure they pass!)
    - Commit local files to repository
Demo: Create the Repository

- Log in to solaris/linux machine
- Two ways to set the “root” of the repository
  - Environment variable
    
    ```
    $ setenv CVSROOT "/project/c421aa01/CVSREP"
    ```
  - Command line flag (-d)
    
    ```
    -d /project/c421aa01/CVSREP
    ```
- Command:
  - $ cvs init
  - $ cvs -d /project/c421aa01/CVSREP init
- Creates repository root, administrative files
- Never edit anything in the repository directly
- Confirm group permissions are properly set
  
  ```
  $ ls -la
  ```
Create the Repository

$ cd /project/c421aa01/
$ ls
Lab1/    Lab2/
$ cvs -d /project/c421aa01/CVSREP init
$ ls -la
...
drwxrwsr-x 3 brutus c421aa01 80 Nov  7 16:34 CVSREP/
...
Demo: Add a Repository Location

- Open perspective “CVS Repository Exploring”
- Right-click in CVS Repositories view
  - New > Repository Location...
- Fill in fields pointing to initialized repository
  - Host: stdsun.cse.ohio-state.edu
  - Path: /project/c421aa01/CVSREP
  - User: brutus (ie your cse login name)
  - Password: • • • • • • • (ie your solaris password)
  - Connection Type: extssh

- Open source projects typically have repositories that permit anonymous access
  - Use of repository, rather than simply downloading the code from a URL, simplifies staying up-to-date with releases
Demo: Populate the Repository

- Right click on project
  - Team > Share Project...
- Select CVS repository to use
- Enter module name
  - Common practice: Choose CVS module name to be same as (local, Eclipse) project name
- Select files to put in repository
  - Omit generated files (eg .class files in bin)
    - Add these to .cvsignore
  - Include other meta files like Eclipse preferences, .project, .classpath, .cvsignore...
Demo: Populate a Local Project

- File > Import... > Projects from CVS
- Select CVS repository to use
- Check "Use an existing module"
  - Select desired module from list
- "Check out as" wizard
  - Common practice: Choose (local, Eclipse) project name to be same as CVS module name
  - Select HEAD to get latest version
- Package explorer view shows different icons for project and contents
  - Reflects association with a repository
  - eg Marks updated files with ">"
Demo: Synchronize with Repository

- Basic operations, right-click on project
  - Team > Commit…
    - Document commit with brief description (make first line very descriptive)
  - Team > Update
    - Safe merges are done automatically

- Alternative: Team Synchronizing perspective
  - Highlights changes in compare editor view
  - Can commit/update from this perspective
  - For non-automergable conflicts, review conflicts and copy/edit to local file as appropriate
  - When done, choose “Mark as merged” for this file, then commit
When to Update/Commit

- Update before committing
  - Integrates everyone else’s changes
- Update when you are ready for someone else’s work
  - Availability of new modules that may affect your code
- Commit when confident that your work can be used by others
  - *Do not* wait until perfection!
  - *Do* make sure your new version compiles!
Good Practices: Golden Rule

☐ Never break the build
  - Applies (primarily) to trunk, although breaking a multi-developer branch is almost as bad
  - Frequent commits are a good thing, but *your* partial code should not prevent another developer from building and testing *their* modifications

☐ (Almost) Never break a test case
  - Other developers may think their (local) changes are responsible for new errors when they next update
Good Practice: Repository Contents

- **Frameworks**
  - JRE, JUnit, Eclipse, ...
  - Warning: big (binary) resources are very slow

- **Team standards/conventions**
  - Comment templates, javadoc templates, ...
  - Eclipse can export project-specific preferences including templates, coding conventions, etc

- **Small sample application**
  - Vanilla application that uses (minimally) the various frameworks relevant to the product
  - Checklist for workstation configuration and building to help new team members get up to speed quickly
Good Practice: Not In Repository

- Generated code
  - eg Java byte code, javadoc html

- FIXME comments *in trunk*
  - OK for developer branches, but should be resolved before merging into trunk

- TODO comments *in trunk (?)*
  - Team convention whether or not to allow these
  - Good reasons on each side of argument:
    - Useful for bookmarking tasks needing attention (by self or others!)
    - Lazy cruft that will accumulate over project lifespan
  - Advice: the more agile the process, the more permissible TODO comments are in the trunk
  - Always OK for developer branches
Good Practices: Process

- Daily build schedule
  - The “heartbeat” of the project
- Release means: tag + create branch for maintenance
- Always tag before a merge
  - Simplifies roll-back if merge goes horribly wrong
- Adopt team standard style:
  - Tag names (versions, major, minor, bug fixes...)
  - Light comment template (brief 1-liners are best)
Pitfalls

- Incomplete commits
  - Common problem: forgetting to add a new file

- Binary vs ASCII files
  - Binary files must be explicitly marked as such to prevent end-of-line mangling
Shortcomings

- Binary files have no meaningful diffs
  - .pdf, .doc, .jar
- Nontransactional commits
  - operations are file-by-file
  - no guarantee of all-or-nothing commit
- Slow for large binaries
  - large binaries/executables/jars can be provided outside the repository
Alternative: SVN

- “Subversion” (subversion.tigris.org)
- Increasingly popular in open source community
- Repository stored as a series of diffs
  - Faster update and commits
- Support recently added to Eclipse, but still flakey
- Advantages:
  - File attributes are part of stored properties
  - Transactional commits
  - Versions refer to entire project (eg directories, not file by file)
  - No need to explicitly mark binaries
  - Support for renaming resources (vs delete and re-add)
  - Better authentication management for remote access
  - Faster, especially for large binaries
SVN Notes

- Create repository (on stdlogin)
  
  $ umask 7
  $ svnadmin create /project/c421aa01/repos
    --fs-type fsfs
  $ umask 77

- Configure repository (SVN perspective)
  - Create new location
    - URL: svn+ssh://stdlogin.cse.ohio-state.edu/project/c421aa01/repos
  - Create subfolder structure
    - New > Create remote folder
    - Typical subfolders: trunk, branches, tags

- Check in project
  - Java perspective: Team > Share Project > SVN
  - Check “Use Specified Folder Name” and give a URL under trunk folder, like repos/trunk/Sudoku
 Summary

- Model
  - Single, shared repository
  - Individual private working copies
  - Optimistic check-out model (no locking)

- Basic operations
  - Update: brings working copy up to date
  - Commit: sends local changes to repository

- Structure
  - Trunk, tags, branches

- Good practices

- Alternative: SVN