Version control

CSE 403
Goals of a version control system

• Keep a history of your work
  – Explain the purpose of each change
  – Checkpoint specific versions (known good state)
  – Recover specific state (fix bugs, test old versions)

• Coordinate/merge work between team members (or yourself, on multiple computers)
Varieties of version control system

Centralized VCS
- One repository
- Many working copies

Distributed VCS
- Many repositories
- One working copy per repository
  (More complicated topologies are possible)
Version control history

**Reality**

- #1 (original)
- #2 by A
- #3 by B
- #4 by A
- #5 by A
- #6 by B

**Centralized VCS**

- #1 (original)
- #2 by A
- #4 by A
- #5 by A
- #6 by B
- #3 by B

**Distributed VCS**

- #1 (original)
- #2 by A
- #4 by A
- #5 by A
- #6 by B
- #3 by B
- #7: merge

- Rewrites history
- One commit per dev.

- Preserves history
- Multiple commits, one push per dev.
DVCS history

Working copy can be updated to any revision in the history
Advantages of a DVCS

• checkpoint work without publishing to teammates
• share changes selectively with teammates
• commit, examine history when not connected to the network
• more accurate history
• more effective merging algorithms
• flexibility in repository organization and workflow
• faster performance
A DVCS prohibits some operations

- No update if uncommitted changes exist
  - must commit first
- No push if not ahead of remote
  - must pull & merge first
- No partial update (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)

- Rationale:
  - Maintain more accurate, complete history
  - Keep all users in sync
  - Avoid painful conflicts
  - Avoid loss of work
Coordinating with others

• **pull** incorporates others’ changes
• If you are **behind**, nothing more to do
  – Behind = your history is a prefix of master history
• If you have made changes in parallel, you must **merge**
  – Merge = create a new version incorporating all changes
Two types of merges

• Conflict-free
  – Changes are to different files or different lines of a file
  – “Conflict-free” is a **textual**, not **semantic**, notion: could yield compile errors or test failures

• Conflicting
  – Simultaneous changes to the same lines of a file
  – Requires **manual conflict resolution**
Resolving conflicts

• There are three versions of the file:
• You decide which version to keep or how to merge them
• Many merge tools exist
• Configure your DVCS to use the merge tool that you prefer

• Don’t panic! Instead, think.
• You can always bail out of the merge and start over again (because you have the full local and remote history)
Popular DVCSes

- Mercurial (`hg`)
- Git (`git`)
- Others: Bazaar, DARCS, ...

- Essentially identical functionality
- Mercurial has a better-designed command set
  - more logical, easier to learn and use, errors are less likely
- Git is faster on huge projects
  - you won’t notice a difference on your project
- Git is more popular
- We recommend Mercurial
Hints

• Never use `hg pull`; instead, use `hg fetch`
  – Does: `hg pull`; `hg update`
  – Does if necessary: `hg merge`; `hg commit`

• To use Mercurial just like SVN:
  – `svn update` = `hg fetch`
  – `svn commit` = `hg commit`; `hg push`
Binary files are not diffable

• The history database records changes, not the entire file every time you commit
  – The diff algorithm works line-by-line

• Do not commit generated files
  – Binaries (e.g., .class files), etc.
  – Wastes space in repository
  – Causes merge conflicts

• Avoid binary files (especially simultaneous editing)
  – Word .doc files
Commit often

• Make many small commits, not one big one
• Easier to understand, review, merge, revert
• How to make many small commits:
  – Do only one task at a time
    • commit after each one
  – Do multiple tasks in one clone
    • Commit only a subset of files
    • Error-prone
  – Create a new clone for each simultaneous task
    • Can have as many as you like
  – Create a “branch” for each simultaneous task
    • Somewhat more efficient
    • Somewhat more complicated and error-prone
Synchronize with teammates often

• Fetch often
  – Avoid getting behind the master or your teammates

• Push as often as practical
  – Don’t destabilize the master build
  – Automatic testing on each push is a good idea
More ways to avoid merge conflicts

• Modularize your work
  – Divide work so that individuals or subteams “own” a module
  – Other team members only need to understand its specification
  – Requires good documentation and testing
• Communicate about changes that may conflict
  – But don’t overwhelm the team in such messages
• Consider using a tool to inform you about future conflicts
  – Crystal:
    http://www.cs.washington.edu/homes/brun/research/crystal/