Git:
Miscellaneous Topics

Lecture 4
Basic Workflow: Overview

1. Configure git (everyone)
2. Create central repo (1 person)
3. Create local repo (everyone)
4. As you work (everyone):
   - Commit locally
   - Fetch/merge as appropriate
   - Push to share
Step 1: Configure Git

- Each team member, in their own VM
  - Set identity for authoring commits
    - $ git config --global user.name "Brutus Buckeye"
    - $ git config --global user.email bb@osu.edu
  - Optional: diff and merge tool (eg meld)
    - $ sudo apt install meld  # to get tool
    - $ git config --global merge.tool meld
    - $ git config --global diff.tool meld
    - # example use:
    - $ git difftool e9d36
Step 2: Initialize Central Rep

- One person, once per project:
- Hosting services (GitHub, BitBucket...) use a web interface for this step
- Or, could use stdlinux instead:
  - Create central repository in group's project directory (/project/c3901aa03)
  - $ cd /project/c3901aa03
  - $ mkdir rep.git  # ordinary directory
  - Initialize central repository as bare and shared within the group
  - $ git init --bare --shared rep.git
Step 3: Create Local Repository

- Each team member, once, in their VM
  - Create local repository by cloning the central repository
  
  ```
  $ git clone ssh://brut@stdlinux.cse.ohio-state.edu//project/c3901aa03/proj1.git mywork
  ```

  - You will be prompted for your (stdlinux) password (every time you fetch and push too)
  - To avoid having to enter your password each time, create an ssh key-pair (see VM setup instructions)
Step 4: Local Development

- Each team member repeats:
  - Edit and commit (to local repository) often
    $ git status/add/rm/commit
  - Pull others' work when can benefit
    $ git fetch origin # bring in changes
    $ git log/checkout # examine new work
    $ git merge, commit # merge work
  - Push to central repository when confident
    $ git push origin master # share
Professional Git

- Commit/branch conventions
- Deciding what goes in, and what stays out of the store
  - Share all the things that should be shared
  - Only share things that should be shared
- Normalizing contents of the store
  - Windows vs linux line endings
Commit/Branch Conventions

- Team strategy for managing the structure of the DAG (ie the store)

- Examples:
  - “Master is always deployable”
    - All work is done on other branches, merged with master only when result compiles
  - “Feature branches”, “developer branches”
    - Each feature developed on its own branch vs. each developer works on their own branch
  - “Favor rebase over merge”
    - Always append to latest origin/branch
Example: Branch-Based Dev
Example: Trunk-Based Dev
What Goes Into Central Repo?

- Avoid developer-specific environment settings
  - Hard-coded file/directory paths from local machine
  - Passwords
  - Better: Use variables (eg $OSU_CSE_LIB) instead

- Avoid IDE-specific files (.settings)
  - But OK to keep .project and .classpath in repo so it is easier to get started by cloning

- Avoid living binaries (docx, pdf)
  - Meaningless diffs

- Avoid generated files
  - Javadoc HTML, .class, .jar, compiled files

- Agree on code formatting
  - Auto-format is good, but only if everyone uses the same format settings!
  - Spaces vs tabs, brace position, etc
Ignoring Files from Working Tree

- Use a .gitignore file in root of project
  - Committed as part of the project
  - Consistent policy for everyone on team

- Example:
  
  ```
  # see github:gitignore/Ruby, /Global/
  # Ignore auto-saved emacs files
  *~
  
  # Ignore bundler config
  /.bundle
  # Ignore the default SQLite database
  /db/*.sqlite3
  # Ignore all logfiles and tempfiles
  /log/*
  /tmp/*
  ```
Problem: End-of-line Confusion

- Differences between OS's in how a "new line" is encoded in a text file
  - Windows: CR + LF (ie "\r\n", 0x0D 0x0A)
  - Unix/Mac: LF (ie "\n", 0x0A)

- Demo: hexdump

- Difference is hidden by most editors
  - An IDE might recognize either when opening a file, but convert all to \r\n when saving

- But difference matters to git when comparing files!

- Problem: OS differences within team
  - Changing 1 line causes every line to be modified
  - Flood of spurious changes masks the real edit
Solution: Normalization

- Git convention: use \n in the store
  - Working tree uses OS's native eol
  - Convert when moving data between the two (e.g., commit, checkout)

- Note: Applies to text files only
  - A “binary” file, like a jpg, might contain these bytes (0x0D and/or 0x0A), but they should not be converted

- How does git know whether a file is text or binary?
  - Heuristics: auto-detect based on contents
  - Configuration: filename matches a pattern
Normalization With .gitattributes

- Use a .gitattributes file in root of project
  - Committed as part of the project
  - Consistent policy for everyone on team

- Example:
  
  ```
  # Auto detect text files and perform LF normalization
  * text=auto

  # These files are text, should be normalized (crlf=>lf)
  *.java text
  *.md text
  *.txt text
  *.classpath text
  *.project text

  # These files are binary, should be left untouched
  *.class binary
  *.jar binary
  ```
Ninja Git

- Temporary storage with stash
- Undoing mistakes in working tree with reset
- Undoing mistakes in store with amend
- DAG surgery with rebase
Advanced: Temporary Storage

- Say you have uncommitted work and want to look at a different branch
- Checkout won't work!
Stash: Push Work Onto A Stack

$ git stash  # repo now clean
$ git checkout ...etc... # feel free to poke around

\[ \text{maint} \quad \text{b} \quad \text{c} \quad \text{d} \quad \text{e} \]

\( \alpha \quad \beta \quad \gamma \quad \delta \quad \varepsilon \)

\( \text{wt} \quad \text{clean} \quad \text{ind} \)
Stash: Pop Work Off the Stack

$ git stash pop  # restores state of wt (and store)

# equivalent to:
$ git stash apply  # restore wt and index
$ git stash drop   # restore store
Advanced: Undoing Mistakes

- Say you want to throw away all your uncommitted work
  - i.e., "Roll back" to last committed state
- Checkout won't work!
Reset: Discarding Changes

$ git reset --hard
$ git clean --dry-run # list untracked files
$ git clean --force # remove untracked files

\[
\begin{align*}
\alpha & \to a \\
\beta & \to b \\
\gamma & \to c \\
\delta & \to d
\end{align*}
\]

\[
\begin{align*}
\text{maint} & \to \text{HEAD} \\
\text{master} & \to \text{HEAD}
\end{align*}
\]

\[\delta \to \text{wt}\]

\[\delta \to \text{ind}\]

replaced to be same as HEAD
Reset: Discarding Commits

$ git reset --hard HEAD~1
# no need to git clean, since wt was already clean

HEAD moved (and attached branch)

now unreachable

replaced to be same as HEAD~1
The Power to Change History

- Changing the store lets us:
  - Fix mistakes in recent commits
  - Clean up messy DAGs to make history look more linear

- Rule: Never change *shared* history
  - Once something has been pushed to a remote repo (e.g., origin), do not change that part of the DAG
  - So: A *push* is really a *commit*ment!
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit

Diagram:
- HEAD
- master
- α → β → γ
- a ← b ← c
- wt
- clean
- ind
Problem 1: Wrong or incomplete commit

Oops! That wasn’t quite right...
Problem 1: Wrong or incomplete commit

Oops! That wasn’t quite right...
Problem 1: Wrong or incomplete commit

Oops! That wasn’t quite right...
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
- Result: Lots of tiny “fix it”, “oops”, “retry” commits
Commit --amend: Tip Repair

- Alternative: Change most recent commit(s)
Commit --amend: Tip Repair

```bash
$ git add --all .
$ git commit --amend --no-edit
# no-edit keeps the same commit message
```

Brand new commit, different hash
Problem 2: As an independent branch is being developed, main also evolves
Problem 2: As an independent branch is being developed, main also evolves
Problem 2: As an independent branch is being developed, main also evolves.

Result: Need periodic merges of main with (incomplete) branch.
Problem 2: As an independent branch is being developed, main also evolves

Result: Need periodic merges of main with (incomplete) branch.
Problem 2: As an independent branch is being developed, main also evolves.

Result: Need periodic merges of main with (incomplete) branch.
Rebase: DAG Surgery

- Alternative: Move commits to a different part of the DAG
Rebase: DAG Surgery

$ git rebase master
# merging master into menu is now a fast-forward
Git Clients and Hosting Services

- Recommended client: Command line!
- Alternative: Various GUIs
  - Linux: gitg, git-gui, git-cola, giggle
  - Win/mac GUI: SourceTree
  - IDEs: RubyMine
- Lots of sites for hosting your repos:
  - GitHub, Bitbucket, SourceForge, Google Code,…
- These cloud services provide
  - Storage space
  - Pretty web interface
  - Issues, bug tracking
  - Workflow with "forks" and "pull requests" to promote contributions from others
Clarity

git != GitHub
Warning: Academic Misconduct

- GitHub is a very popular service
  - But only *public* repo's are free
  - Edu discount gives free *private* repo's
  - 3901 has an account ("organization") for private repo's (see class web site)

- Bitbucket has free private repo's, for small teams (< 5 collaborators)

- Public repo's containing coursework can create academic misconduct issues
  - Problems for poster
  - Problems for plagiarist
Mercurial (hg): Another DVCS

- Slightly simpler mental model
- Some differences in terminology
  - `git fetch/pull` ~=` hg pull/fetch`
  - `git checkout` ~=` hg update`
- Some (minor) differences in features
  - No rebasing (only merging)
  - No octopus merge (`#parents <= 2`)
- But key ideas are identical
  - Repository = working directory + store
  - Send/Receive changes between stores
Summary

- Workflow
  - Fetch/push frequency
  - Respect team conventions for how/when to use different branches

- Central repo is a shared resource
  - Contains common (source) code
  - Normalize line endings and formats

- Advanced techniques
  - Stash, reset, rebase

- Advice
  - Learn by using the command line
  - Beware academic misconduct