CIS Minicourses Shared Lecture

2. Version Control Systems

Several Challenges in Software Development:

1. **Collaboration** - multiple people working on the same codebase
2. **Tracking Changes** - keeping track of what changed and when
3. **Reverting Mistakes** - being able to undo changes

Version Control Systems help to solve all of these problems.

Collaboration

How do you put all of your code in the same place?

How do you "stitch together" the changes that different people made?

How do you actually *share* the code that you've written?

Tracking Changes

When did this feature get added? When did this bug get fixed?

When did this feature stop working? When did the bug appear?

Who has contributed what? Who's responsible for this feature?

Reverting Mistakes

What if you accidentally delete a file? What if you accidentally delete a line?

How do you go back to a version of your program that worked?

Centralized vs. Decentralized

Centralized	Decentralized
One central server holds the repository	Multiple servers/clients can hold different versions of the codebase *
subversion (svn) or cvs	**git** or mercurial
Can only change the codebase when you can connect to the server!	Can work independently.

**some centralization here: usually one instance of the codebase is marked as the special "main" copy*

A Demo (and a Warning)

The command line interface for git is unwieldy and confusing. It can be easy to do the wrong thing. Often you'll use a GUI (GitHub Desktop, something integrated into your IDE) to do this stuff instead, but the terminology is the same.

- `git init` - create a new git repository in the current directory
- `git status` - "what's currently going on in this repository?"
  - What's our branch?
  - List the commits
  - List the changed files
- `git add <filename>` - add a file to the staging area
- `git rm --cached <filename>` - remove a file from the staging area
- `git commit` - commit the changes in the staging area
  - `git commit -m "message"` - commit with a **message** explaining what the purpose of this new "version" is
- `git log` - show the history of commits

An Ontology of Files

- Untracked
  - git only knows of the file's existence, but its history.
- Staged
  - git is tracking the file and is prepared to make a snapshot of its current state
- Unstaged & Changed
  - File has not yet been `add`ed and that can be `restore`d to their previous state
- Committed
  - git has made a snapshot of the file's current state

Have Problems with Committment?

- `git add <filename>` stages a change, but staging before committing lets you observe your changes before preserving them.
- `git diff <filename>` shows the differences between the current version of the file and the version in the staging area.
- `git restore <filename>` restores a file to its previous state.

Bringing in GitHub

- Can create a repo on GitHub, but at first it has nothing to do with the local one on your machine!
- Need to `push` the local repo to the remote one on GitHub:

```
git remote add origin git@github.com:your_username/your_repo.git
git branch -M main
git push -u origin main
```

- Need to do a `push` every time you want your local changes to be reflected on GitHub.

`Push`ing and `Pull`ing

- `git push` - send your local changes to the remote repository
- `git pull` - get the latest changes from the remote repository

`git` is decentralized, so you'll need to do some work to get consistency among all of your repositories.

The Burden Is on the Pusher

When versioning differences exist between your local version and the remote version, you'll need to resolve them before you can push.

- `git pull` will often be able to automatically do the merging.
  - Bring in the line additions/deletions from the remote version to your local version
  - Conflicting versions within the same line will need to be resolved manually.
- Then, you can safely `push` the merged version to the remote repository.

The Manual Merge

- Open the file to review the conflicts and choose how they'll each be resolved.
  - (there's software that can help you do this, but just a text editor will do)
- Save and close the file, then `add` it to the commit.
- Commit the new changes!
- Then, you can finally push.

Avoiding the Manual Merge

Manual merges are a pain to deal with, especially if you're dealing with conflicts in code of any real complexity.

Best practices:

- Delegate tasks so that different people work on different files
- Make frequent, small commits!

Other Mistakes to Avoid

Original:

```
int x = 4;
int y = 5;
int z = x + y;
```

My Change:

```
String x = "Harry";
int y = 5;
String z = x + y;
```

Your Change:

```
int x = 4;
int y = 9;
int z = x + y;
```

⚠️ Make Sure Your Code Compiles Before You Commit!! ⚠️

Branches

- Branches are a way to work on a new feature without affecting the main codebase.
- Multiple branches can co-exist simultaneously, and each can be pushed to/pulled from separately
- To complete the addition of a feature, you can merge the branch back into the main codebase.
    - ⚠️ Of course, merging can be hard! ⚠️

Branches allow us to extend our timeline to have "parallel universes"; we can go "side-to-side" instead of back and forth in the timeline.

Branch Commands

- `git branch` - list all branches
- `git branch <branchname>` - create a new branch
- `git checkout <branchname>` - switch to a different branch
- `git push --set-upstream origin <branchname>` - push, setting the destination branch on the remote repository
- `git merge <branchname>` - merge the changes from the specified branch into the current branch