CMPT 276 Class 02: Version Control

Dr. Jack Thomas
Simon Fraser University
Fall 2020

Remember this?

Image credit: https://www.amazon.com/Double-Density-MF2-DD-Diskettes-Formatted/dp/B006NNGZ9S

Version Control? Revision Control? Source Control?

- All mean the same thing.
- A system to manage changes to electronic documents (typically, code).

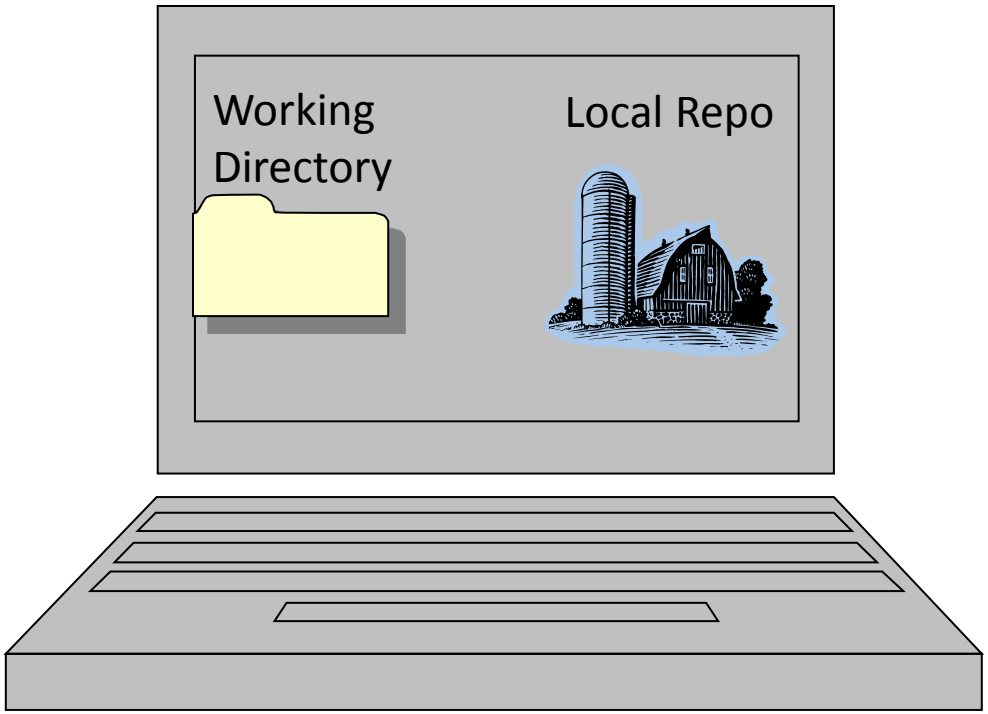
- **Motivation**: Coordinating changes to code from multiple developers. How can we ensure changes are not lost or incompatible?

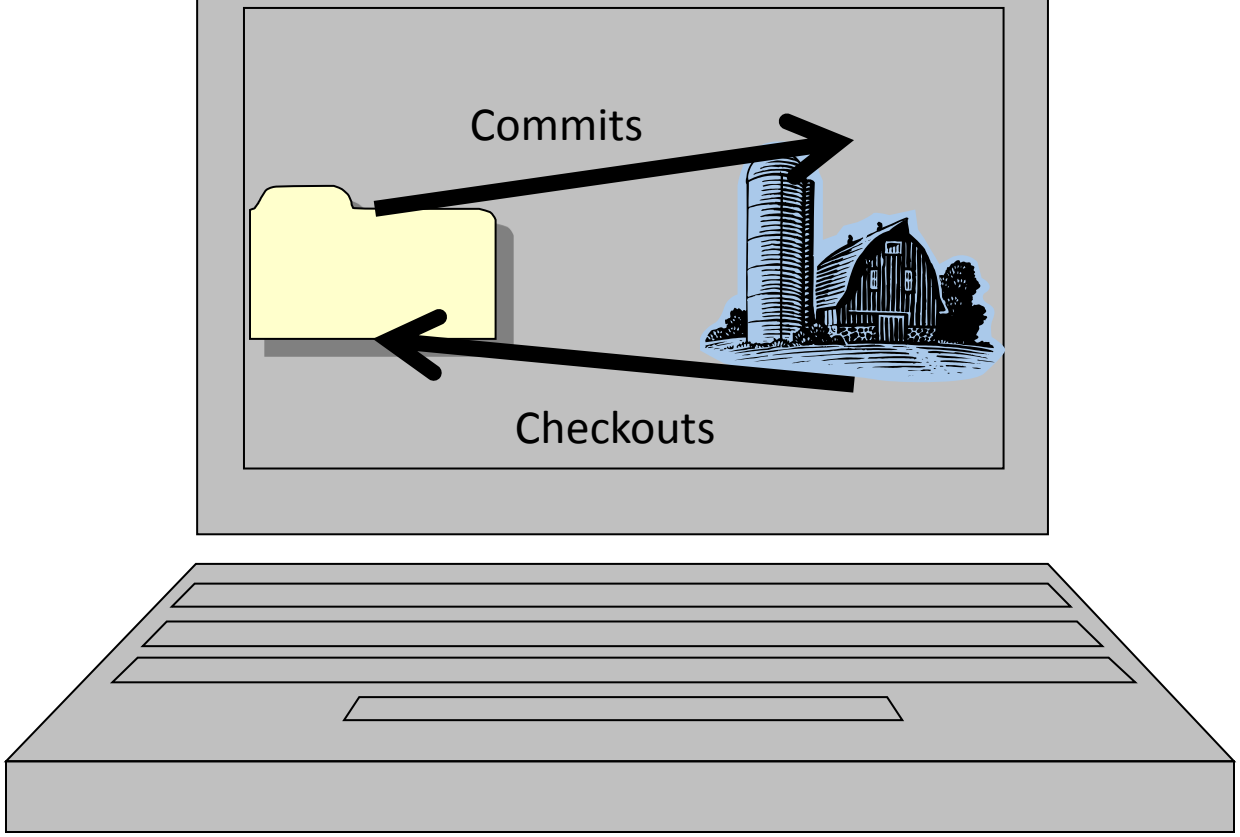
- What does Git stand for?

- Nothing.

Git Basics – The Local Topology

- We start with the working directory on your local computer.
- Then there's the local code repository ("repo")

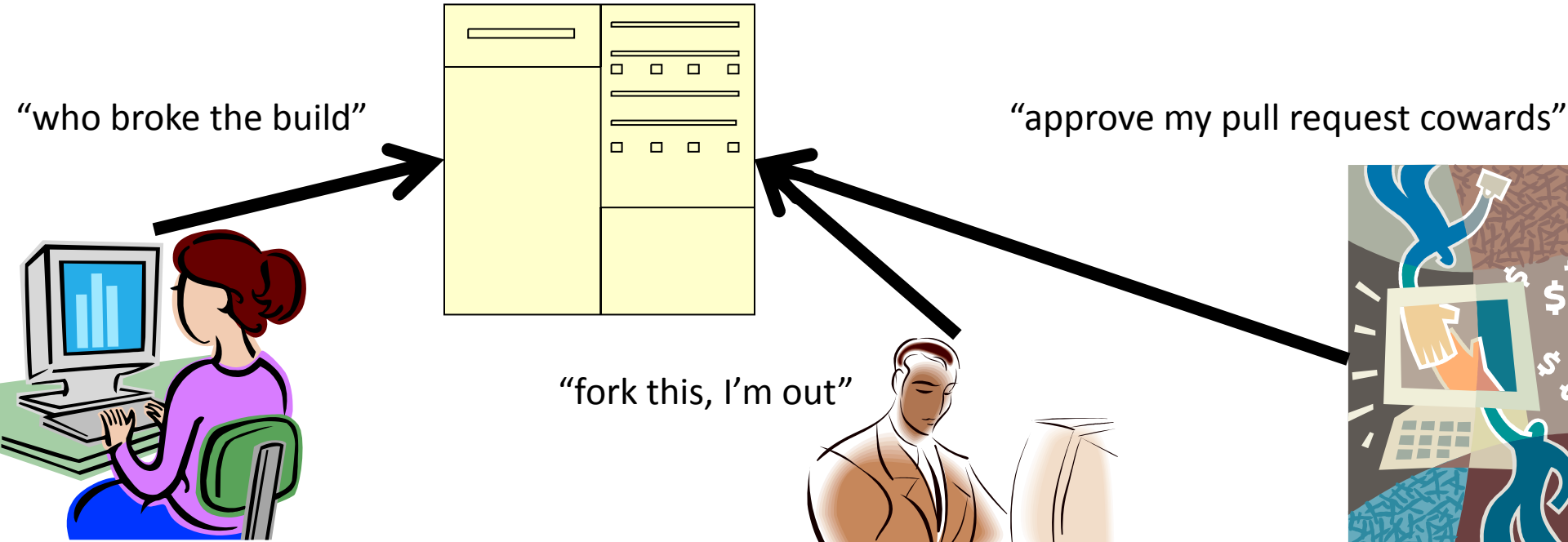




- The **head** is the latest version of the code.
  - The latest code in the repo can be **checked out** into the working directory.
  - You then **commit** the changes to your local repo.

Git Basics – Remote Topology

- That local repository is synchronized with a **remote repository** on a **remote server**, which other developers on other computers can synch their own local repositories to.



Git Is Distributed

- Git has no single centralized master repo, each “local repo” is a full and complete repo.
  - Can work off-line (on a plane) and still commit to the local repo. Later sync up with the remote repo.
- Often the remote repo is a dedicated Git server such as GitHub or GitLab.
  - These systems add extra team collaboration and discussion tools (more later).

Work Flow Step 1: Setup

1. After making a working directory, initialize a git repository in it.

2. Associate your local repo to a remote repo by either:

   a) **Creating** a repo in GitLab (gitlab.cs.sfu.ca) and push some existing code to it
   b) **Cloning** an existing repo to your local PC.

Step 2: When You Make Changes

1. Do some work in working directory
   - Create new files, change files, delete files, etc.
2. **Add** Command
   - Either adding new files to the repo or marking which changed files to update.
3. **Commit** Command
   - Commit all staged changes to local repo.
   - Sometimes called "**Check-in**"
4. **Push** Command
   - Transfer committed changes to remote repo.
5. **Status** Command (optional)
   - View the state of local file changes

Image credit: *The Simpson*, https://frinkiac.com/

Step 3: Other People Are Also Doing Changes

- Other team members will push their own changes to the repo which you then need.
  - May be new / changed / deleted files
- **Pull** Command
  - Get changes from remote repo and apply them to local repo and working directory (move to head).
  - If there are any conflicting changes, may need to do a merge (more later).
- **Log** Command
  - At any time, can view the changes people have made.

Git Tools

- Via Command Line
  - Git is very often accessed via its command-line tools
  - Git commands look like:

```
git clone git@csil-git1.cs.surrey.sfu.ca:myTeam/daProject.git
git commit –m “I just did a whole load of work!”
```

- Via GUI Integrated Tools
  - Abstracts away some low-level details, but low-level understanding is still required to understand it.
  - Can be part of your IDE, like IntelliJ
  - Can be integrated into a file system (TortoiseGit)
- In this course we’ll mostly be using Git via GUI, but try it with the command line too!

Time For The Basic Git Demo!

1. Go to https://csil-git1.cs.surrey.sfu.ca/
2. Click “New Project”, name it something, and select private.
3. Copy the git repository’s URL.
4. Go back to your project and select VCS -> Enable Version Control Integration and choose Git.
5. Right click your project folder, go to Git and +Add.
6. Now go to VCS -> Commit and write a message.
7. Choose Commit and Push and designate the Remote using the git URL you copied.

Basic Git Sequence For Editing Code

1. Open your working directory before making any changes.
2. **Pull** any changes anyone else has made**.**
   - Should bring you up to speed without causing any conflicts.
3. Do your work.
   - Can't **pull** from the remote with uncommitted changes.
4. **Add** and **Commit** changed files, updating the local.
5. **Pull** any changes made while you were working**.**
   - Automatically merges files without conflicting changes.
   - You'll have to manually merge conflicts when required.
6. **Push** your merged result to the remote.
   - Cannot push if others have pushed code: "current branch is behind master", "unable to fast-forward"

Merge Conflict Demo

- What happens when two team members make conflicting changes on the same piece of code?
  1. Can’t pull any more while conflicts exist.
  2. Need to commit my changes.
  3. Now a pull will trigger a merge
  4. Decide every highlighted conflict.
  5. Once you’re done merging, add/commit/push.
- Bonus step: Communicate with your team!

.gitignore Files

- Lists file types to exclude from Git, ensuring only the right kinds of files are excluded.
  - Examples: Exclude .bak, build products, some IDE files, other temporary and working files.
- Useful to avoid accidentally adding files in your working directory you don’t want as part of the repo.
- Just a plain-text file listing names, directories, file extensions, etc., on a line-by-line basis.

.gitignore File Example

```
1  *.iml
2  .gradle
3  /local.properties
4  /.idea/caches
5  /.idea/libraries
6  /.idea/modules.xml
7  /.idea/workspace.xml
8  /.idea/navEditor.xml
9  /.idea/assetWizardSettings.xml
10 .DS_Store
11 /build
12 /captures
13 .externalNativeBuild
14 .cxx
15
```

Commit Messages

- Commit messages must be meaningful!
- Line 1: Short summary (<70 characters)
  - Capitalize your statement
  - Use imperative: "Fix bug..." vs "fixed" or "fixes"
- Line 2: blank
- Line 3+: details; wrap your text every ~70 characters
- Ex:

> Make game state persist between launches and rotation.
>
> Use SharedPreferences to store Game's state. Serialize using Gson library and Bundle for rotation.

- If pair programming during CMPT276, add your partner's user ID at start: "[pair: jackt] Make game state persist ...."

Reverting Changes

- Use **Checkout** to revert files (or the ‘Revert’ button in your IDE)
  - Discards any uncommitted changes to a file.
  - Overwrites the files in working directory with ones from the local repo.
- Be careful when reverting! You’ll lose all uncommitted changes!
  - If in doubt, grab a backup copy of your work folder using ZIP, then revert. Just make sure you don’t commit the backup too!

Delete, Rename

- Deleting a file
  - Delete file normally via the OS/IDE, then Add it to Git. Git records it's now deleted.
  - Will be deleted on everyone else's system when they pull your changes.
- Renaming a file
  - Rename file normally via the OS/IDE, then Add it via its new name. Git will know.
  - Git tracks files by their content, not by their name.

General Principles: Merge v. Lock

- Two competing approaches different teams can take to version control on a project:

1. Checkout-Edit-Merge
   - Allow multiple developers to work on the same code concurrently, and simply manage merge conflicts as they come up.
2. Lock-Edit-Unlock
   - Locking avoids merge conflicts by preventing others from changing the file, but adds pressure to make changes quickly.

Image Credit: *SimCity 2000*, http://deathgenerator.com/#sc2k

Other Features

- Git works **Atomically**
  - No part of a change occurs unless the whole change happens .
  - If there’s conflicts, your push won’t merge some and skip others.
- Making a **Tag**
  - Mark certain versions of certain files as a group. Ex: "Files for Version 1.0 of product".
  - Can then check out the code as it was at a certain point, more structured than just winding back to a particular commit.

Even More Features?

- What about Stashing, Fetching, Rebasing, Branching, Forking…

- Git has many more advanced features, as well as optional settings for the more basic commands, but we'll revisit them as needed.

Team Work

- It Makes The Dream Work
- …Unless you break the build. Don’t check in something that breaks the build.
- Seriously.

How Often Should I Commit?

- Never Not Be Committing
  - Commit little changes to local repo very often (**hourly**)
  - Once some work is more stable, push all the changes at once to remote repo (**daily**)
- Expectations for CMPT 276:
  - Committing / pushing frequently gives visibility to your contributions, making them easier to mark.
  - In professional settings, your commits and pushes would be tailored more to the needs of your team, and likely add up several smaller contributions into one bigger commit representing a more meaningful contribution.

Some Notes On Comment Etiquette

- Don’t write journals in comments in source code

```
// Removed Jan 2002 for V1.01
// cout << "Dave; I wouldn't do that, Dave.\n";
```

  - Put meaningful comments into checkins!
- Don’t leave dead code

```
#if 0
// Unneeded, but left 'cuz someone may want it...
.....
#endif
```

- Don’t sign your code
  - // Written by Dr. Evil

```
....
```

Try Using Git Commands in the Terminal!

- When you create a new project on GitLab and want to push an existing project from your IDE into it, GitLab suggests running the following commands in the terminal:
  - git init
  - git remote add origin https://csil-git1.cs.surrey.sfu.ca/jackt/GitDemo.git
  - git add .
  - git commit -m "Initial commit"
  - git push -u origin master

(Optional) Adding an SSH key

- Alternative to logging into GitLab when pushing.

1. Open Git GUI, go to Help, and select Show SSH Key
2. Generate a key and copy it.
3. Go to your GitLab profile settings and then SSH keys.
4. Paste the SSH key you just generated and then Add it.
5. Start using the SSH link instead of the git URL

Recap – Commit This To Memory!

- Version control is a critical tool for development.
  - Git is a popular, distributed version control system.
- Common Operations:
  - clone, add, commit, push, pull, status, log, merge
- Git Project Management
  - Avoid conflicts with locks, or else merge them as needed.
- Basic Features
  - Use .gitignore, reverts, and Tags. Understand atomic operations.
- Rules to Code By
  - Commit often, write good messages, and don’t break the build!