CS 520
Theory and Practice of Software Engineering
Spring 2020
Version Control
February 11, 2020

Thursday (February 13)
- First in-class exercise
- On using git (today is a prelude with useful info)
- Form 4-person teams
  - Use moodle to self-select a team; can do it before Thursday or on Thursday
- At least one person per team needs to bring a laptop

BRING A LAPTOP!

Our Goal
- Learn about different kinds of Version Control Systems (VCSs)
- Overview the basics of git
- Touch some intermediate git topics
- Clear up common points of confusion
  - Branch vs. Fork?
  - Merge vs. Pull Request?
  - Pull vs. Fetch?
  - Fork vs. Clone?

Working in Teams

Large ambitious goals usually require that people work together.

TEAMWORK
What Is VCS?

What are Version Control Systems

A Version Control System (VCS) records changes to a file set over time, making it easy to review or revert to specific versions later.

Why Use VCS?

Why Use Version Control?

[Diagram showing various files and directories]
Why Use Version Control?

- Easy to revert to previous versions
- Work on multiple features in parallel
- Makes collaboration easier
- Narrate the evolution of codebase with messages
- Nice tools such as GitHub (and GitLab (and BitBucket...)) with advanced features such as pipelines, issue tracking, wikis, etc...
- Can store a backup remotely and automatically - easy to keep this up to date!
- Helps keep your working space clean

Who Uses VCS?

- Programmers
- Applications (Microsoft Word, Google Docs, ...)
- Organizations
  - VCS can be used to sync data, not just code
Types of VCS

Centralized version control
- (old model)
- Examples: Concurrent Versions System (CVS)
  Subversion (SVN)

Doing work
- I update my checkout (working copy)
- I edit
- I update my checkout again
- I merge changes if necessary
- I commit my changes to the Master
Problems with centralized VC

• What if I don’t have a network connection?

• What if I am implementing a big change?

• What if I want to explore project history later?

Types of VCS -- Distributed

- Each developer has their own repository:
  - Created by the developer, or
  - Cloned from an existing (remote) repository
- Developers work on their own repos:
  - They can commit, branch, etc.
  - Activity is local unless it is pushed to remote repo
  - Remote activity is not seen until developer fetches from the remote repo
- Examples: Mercurial (hg), git

Distributed version control model

Doing work

• I pull from the Master
• I update my checkout
• I edit
• I commit
• I pull from the Master
• I merge tips if necessary and commit again
• I push my changes to the Master
• Bill and Melinda work at the same time

• At the end, all repositories have the same, rich history

Pros and Cons of Centralized VCS

A Motivating Example: What is this git command?
A Motivating Example: What is this git command?

NAME

git-add - Add file contents to the index

SYNOPSIS

git add 

dry-run | -n

force | -f

interactive | -i

patch | -p

DESCRIPTION

This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically adds the
current content of existing paths as a whole, but with some options it can also
be used to add content only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.

A Motivating Example: What is this git command?

NAME

git-checkout

SYNOPSIS

git checkout 

-q | -f | -m | <branch>

DESCRIPTION

Updates files in the working tree to match the version in the index or the
specified tree. If no paths are given, git checkout will also update HEAD to set
the specified branch as the current branch.

A Motivating Example: What is this git command?

NAME

git-______ - Switch branches or restore working tree files

SYNOPSIS

git ______ 

-q | -f | -m | <branch>

DESCRIPTION

Updates files in the working tree to match the version in the index or the
specified tree. If no paths are given, git ______ will also update HEAD to set
the specified branch as the current branch.

A Motivating Example: What is this git command?

NAME

git-______ - Forward-port local commits to the updated upstream head

SYNOPSIS

git ______ 

-i | --interactive | [options] 

-v <new commit> | --update-commit

--continue | --skip | --abort | --edit-todo

--root | <branch>

DESCRIPTION

If <branch> is specified, git ______ will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If --root is not specifying, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config[1] for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the ______ will abort.
A Motivating Example: What is this git command?

**NAME**
git-rebase

**SYNOPSIS**
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]

**DESCRIPTION**
If <branch> is specified, git rebase will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the rebase will abort.

Our goal with git

Be able to understand the git man-pages

Git Basics

How Git Works

Git Basics -- Tracked vs. Untracked

- untracked file - a file not currently under version control
- tracked file - a file that is under version control

File Status Lifecycle
Git Basics -- Three Main Stages

1. Committed: Everything in the file is currently in the database
2. Modified: Changed the file but have not committed to the database
3. Staged: Marked the file for addition to the database in the next commit

Note that all of the above pertain to tracked files.

Git Basics -- Creating Repositories

Initializing a repository
- `git init`: Create an empty git repository or reinitialize an existing one
  - `--bare`: create a bare repository
  - `[directory]`: `git init` is run inside the provided directory
- `git init` creates a `.git` folder in the directory chosen

Cloning a Repository
- `git clone`: Clone a repository into a new directory
  - `--depth <depth>`: Create a shallow clone with a history truncated to `<depth>` commits
  - `--branch <name>`: Point local HEAD to specific branch (more on HEAD in a bit!)
  - `--origin <name>`: Use `<name>` to keep track of remote repo instead of ‘origin’
- `git clone` basically does:
  - `git init`
  - Points some meta variables at an existing repository
  - Copies the data to the new repo

.git/

- What's in it?
  - `branches/`: ref to recent commit message
  - `config`: configure your git repository
  - `description`: only used by the GitWeb program
  - `hooks/`: contains client or server-side hook scripts
  - `index`: The "staging area"
  - `info/`: keep a global exclude file for your project
  - `objects/`: keeps track of history of HEAD and refs
  - `refs/`: where the actual content is stored
  - `refs/heads`: keeps track of refs and tags
.git/

- What's in it?
  - branches/: most recent commit message
  - config/: configure your git repository
  - hooks/: This contains client or server-side hook scripts
  - index: The “staging area”
  - logs/: keeps track of history of HEAD and refs
  - objects/: where the actual content is stored
  - refs/: keeps track of refs and tags

Git Vocabulary

- index: staging area (located .git/index)
- content
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream
- content: git tracks what's in a file, not the file itself
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream
## Git Vocabulary

- **index**: staging area (located .git/index)
- **content**: git tracks what's in a file, not the file itself
- **tree**: git's representation of a file system.
  - **working tree**: Tree representing what is currently checked out (what you see)
  - **staged**: ready to be committed (in index/will be stored in a commit object)
- **commit**: A set of database entries detailing a snapshot of the working tree
- **ref**
- **branch**
- **HEAD**
- **upstream**
<table>
<thead>
<tr>
<th>Git Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>- index: staging area (located .git/index)</td>
</tr>
<tr>
<td>- content: git tracks what’s in a file, not the file itself</td>
</tr>
<tr>
<td>- tree: git’s representation of a file system.</td>
</tr>
<tr>
<td>- working tree: Tree representing what is currently checked out (what you see)</td>
</tr>
<tr>
<td>- staged: ready to be committed (in index/will be stored in a commit object)</td>
</tr>
<tr>
<td>- commit: A set of database entries detailing a snapshot of the working tree</td>
</tr>
<tr>
<td>- ref: pointer to a commit object</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Git Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>- index: staging area (located .git/index)</td>
</tr>
<tr>
<td>- content: git tracks what’s in a file, not the file itself</td>
</tr>
<tr>
<td>- tree: git’s representation of a file system.</td>
</tr>
<tr>
<td>- working tree: Tree representing what is currently checked out (what you see)</td>
</tr>
<tr>
<td>- staged: ready to be committed (in index/will be stored in a commit object)</td>
</tr>
<tr>
<td>- commit: A set of database entries detailing a snapshot of the working tree</td>
</tr>
<tr>
<td>- ref: pointer to a commit object</td>
</tr>
<tr>
<td>- branch: basically just a (special) ref. Semantically: represents a line of dev</td>
</tr>
<tr>
<td>- HEAD</td>
</tr>
<tr>
<td>- upstream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Git Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>- index: staging area (located .git/index)</td>
</tr>
<tr>
<td>- content: git tracks what’s in a file, not the file itself</td>
</tr>
<tr>
<td>- tree: git’s representation of a file system.</td>
</tr>
<tr>
<td>- working tree: Tree representing what is currently checked out (what you see)</td>
</tr>
<tr>
<td>- staged: ready to be committed (in index/will be stored in a commit object)</td>
</tr>
<tr>
<td>- commit: A set of database entries detailing a snapshot of the working tree</td>
</tr>
<tr>
<td>- ref: pointer to a commit object</td>
</tr>
<tr>
<td>- branch: basically just a (special) ref. Semantically: represents a line of dev</td>
</tr>
<tr>
<td>- HEAD: a ref pointing to branch/commit being worked on (i.e., Working Tree)</td>
</tr>
<tr>
<td>- upstream: complicated, basically “backwards in time” (but not quite!)</td>
</tr>
</tbody>
</table>
Git Basics

Working Locally

Git Basics: Changing Content -- git add

git add does two things:
1. given an untracked file it will
   a. start tracking it
   b. update /.git/index using the current content found in the working tree to prep the content for the next commit (i.e., the content is staged).
2. given a modified unstaged file it will
   a. stage its contents for commit
--patch, -p: start an interactive staging session that lets you choose portions of a file to add to the next commit.

Git Basics: Changing Content -- git commit

git commit updates the Git database with staged content in /.git/index
- Note that staged files can have unstaged changes
- By default this will open an editor for you to enter a commit message
--message=<msg>, -m <msg>: Add <msg> as the commit message. If multiple messages are given, concatenate as separate paragraphs
--patch, -p: Use the interactive patch selection interface to choose which changes to commit (similar to git add -p)
Git Basics: **Making Queries** -- **git status**

**git status** shows the working tree status. This command displays:
- paths that have differences between the index file and the current HEAD
- paths that have differences between the working tree and the index file
- paths in the working tree that are not tracked by git

---short, --s: Give the output in the short-format
--ignored: Show ignored files

Git Basics: **Making Queries** -- **git log**

**git log** inspects commit history with multiple display options
- **git log** is basically a wrapper around **git rev-list** and **git diff-*** (don't worry about these - I sure don't!)

---Some Examples---

**git log**
**git log --graph**
**git log --graph --all**
**git log --graph --all --oneline**

Git Merge
Git Rebase

Changing Commit History with Rebase
- Git rebase lets us change our commit history
- It is a powerful tool, but we will only scratch the surface
Changing Commit History with Rebase

- Git rebase --onto gives us a bit more power
Why use Git Rebase?

Points of Confusion

Fork vs. Clone
Fork vs. Clone

Fork
- Fork is NOT A GIT CONCEPT
  - It was invented by GitHub
  - Fork stores extra information and makes pull requests possible

Clone
- Clone IS A GIT CONCEPT
  - clone extends init
  - exists independent of github

Branch vs. Clone

Branch
- Branch creates a ref

Clone
- Clone creates a new repository

Pull vs. Fetch
**Pull vs. Fetch**

**Fetch**
- Take target branch from a remote repository and store it in `.git/refs/remotes/`
- NOT integrated/merged with local branches!!!

**Pull**
- Fetches remote branch and merges with local branch or repository

---

**Next time: Thursday (February 13)**

- First in-class exercise
- On using git (today was a prelude with useful info)
- Form 4-person teams
  - Use moodle to self-select a team; can do it before Thursday or on Thursday
  - In-class exercises
- At least one person per team needs to bring a laptop

**BRING A LAPTOP!**