CS 520
Theory and Practice of Software Engineering
Fall 2022

Version Control

September 20, 2022
Recap: Usability versus UI design

- **Usability**: the effectiveness of users achieving tasks

- The goal of good UI design is to improve usability
Recap: Achieving usability

- Follow UI design principles (e.g., Shneidermann)
- Prototype the user interface (paper or electronic)
- Have users try the user interface and gather data
- Have UI experts (e.g., HCI, Human Factors) review the user interface

Good UI design focuses on the user

not on the developer, not on the system environment
Recap:
Another UI example
Recap: UI design review - Good
Recap: UI design review - Bad
Next class: Thursday (September 22)

- First in-class exercise
- On using git (today is a prelude with useful info)
- Form 2-, 3-, or 4- person teams
  - Use Moodle to self-select a team from now until Thursday at 11:59 PM (a little before midnight)
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?
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.
Motivating example
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 Version Control?

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

- Centralized, e.g., SVN
- Distributed, e.g., Git
Types of VCS -- Centralized

- There exists a single "central" copy of the project
  - All developers commit to this single copy
- Each developer has local working copy(ies)
  - As soon as they commit, the central repo reflects the changes
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 Remote Repo
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 dev pulls from the remote repo
- Examples: Mercurial (hg), git
Distributed version control model

Remote Repository

- Alice's Laptop Repository
- Alice's Desktop Repository
- Bob's Repository
- Yuriy's Laptop Repository
- Yuriy's Desktop Repository

- Alice's laptop working copy
- Alice's Desktop working copy
- Bob's working copy
- Yuriy's laptop working copy
- Yuriy's desktop working copy
Doing work

- I pull from the Remote Repo
- I update my checkout
- I edit
- I commit
- I pull from the Remote Repo
- I merge tips if necessary and commit again
- I push my changes to the Remote Repo
History view (log)

- Bill and Melinda work at the same time
- At the end, all repositories have the same, rich history

*NOTE* Git (and GitHub) switched the term “master” to “main”
Centralized versus Distributed version control
Our goal with git

*Be able to understand the git man-pages*
Motivating Example 1: What is this git command?

NAME

```bash
git-___ - ___ file contents to the index
```

SYNOPSIS

```bash
```

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 ___s the current content of existing paths as a whole, but with some options it can also be used to ___ content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.
Motivating Example 1: What is this git command?

**NAME**

`git-add` - Adds file contents to the index

**SYNOPSIS**

```
```

**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 with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.
Motivating Example 2: 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.
Motivating Example 2: What is this git command?

NAME

    git-checkout - Switch branches or restore working tree files

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.
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
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
Git Basics -- Creating Repositories

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'

- Basically, clone just:
  - calls `init`
  - points some meta variables at an existing repository
  - copies the data to the new repo
.git/

- What's in it?
  - branches/:
  - COMMIT_EDITMSG: most recent commit message
  - config: configure your git repository
  - description: only used by the GitWeb program ([source](#))
  - hooks/: This contains client or server-side hook scripts ([more info](#))
  - index: The "staging area"
  - info/: keeps a global exclude file for your project
  - logs/: keeps track of history of HEAD and refs
  - objects/: where the actual content is stored
  - refs/: keeps track of refs and tags
What's in it?

- **branches/**
  - COMMIT_EDITMSG: most recent commit message
  - config: configure your git repository
  - description: only used by the GitWeb program ([source](#))
- **hooks/**: This contains client or server-side hook scripts ([more info](#))
- **index**: *The "staging area"
- **info/**: keeps a global exclude file for your project
- **logs/**: *keeps track of history of HEAD and refs
- **objects/**: where the actual content is stored in a database
- **refs/**: *keeps track of refs and tags
Git Vocabulary
Git Vocabulary

- index: staging area (located .git/index)
- content
- 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
- 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
- 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
  - 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
- 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
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: pointer to a commit object
- 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: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a line of dev
- 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: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a line of dev
- HEAD: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- 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**: pointer to a commit object
- **branch**: basically just a (special) ref. Semantically: represents a *line of dev*
- **HEAD**: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- **upstream**: complicated, basically "backwards in time" (but not quite!)
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`)
basic functionality with sanity check done

master (#15)

Rico Angell committed on Oct 2, 2017

1 parent 1c82903 commit 2c22c0a72ce221615da3868af326d929852b887a65

Showing 2 changed files with 108 additions and 64 deletions.

```python
def test_group_discrimination():
    t = themis.Theme(xml_fname="settings.xml")
    _p = t.group_discrimination(_fields=['Sex', 'Race'])
    print "Sex and Race": p
    _p = t.group_discrimination(_fields=['Race'])
    print "Race": p
    _p = t.group_discrimination(_fields=['Sex'])
    print "Sex": p
def test_causal_discrimination():
    t = themis.Theme(xml_fname="settings.xml")
    _p = t.causal_discrimination(_fields=['Sex', 'Race'])
    print "\nCausal:
```

```python
+ from itertools import chain, combinations

def test_group_discrimination():
    t = themis.Theme(xml_fname="settings.xml")
    _p = t.group_discrimination(_fields=['Sex', 'Race'])
    print "\nGroup:"
    for f in t._all_relevant_subs(['Sex', 'Race', 'Age', 'Income']):
        _p = t.group_discrimination(_fields=f)
        print f, "--> ", _p
```

```python
+ print "\nCausal:"
```
Git Basics – Git Commit Object

Message

basic functionality with sanity done

Rico Angell committed on Oct 2, 2017

1 parent 1c28901 commit 2c2ec0a72ce21615da3868af32d29d29852887a65

Showing 2 changed files with 108 additions and 64 deletions.

Themis2.0/test/test_themis.py

```python
import pytest
import themis

def test_group_discrimination():
    t = themis.Themis(xml_fname="settings.xml")
    _, p = t.group_discrimination(l_fields=["Sex", "Race"])
    print "Sex and Race: ", p
    _, p = t.group_discrimination(l_fields=["Race"])
    print "Race: ", p
    _, p = t.group_discrimination(l_fields=["Sex"])
    print "Sex: ", p

def test_causal_discrimination():
    t = themis.Themis(xml_fname="settings.xml")
    _, p = t.causal_discrimination(l_fields=["Sex", "Race"])
```

+ from itertools import chain, combinations

```python
import pytest
import themis

def test_group_discrimination():
    t = themis.Themis(xml_fname="settings.xml")
    _, p = t.group_discrimination(l_fields=["Sex", "Race", "Age", "Income"]):
    _, p = t.group_discrimination(l_fields=f)
    print f, "---> ", p

def test_causal_discrimination():
    t = themis.Themis(xml_fname="settings.xml")
    _, p = t.causal_discrimination(l_fields=["Sex", "Race"])
```
basic functionality with sanity check done

Rico Angell committed on Oct 2, 2017

1 parent 1c820d1 commit 2c2ec8a72ce2665da3e38668af32d929852887a65

Showing 2 changed files with 108 additions and 64 deletions.

29 30  Themis2.0/test/test_themis.py
... @@ -1,3 +1,4 @@
 1 import pytest
 2 import themis

1 + from itertools import chain, combinations
2  import pytest
3  import themis

-26,23 +27,23 @@ def test_get_test_result():
 27  
 28  def test_group_discrimination():
 29  t = themis.Temis(xml_fName="settings.xml")
 30  - p = t.group_discrimination(l_fields=['Sex', 'Race'])
 31  - print 'Sex and Race: ', p
 32  - p = t.group_discrimination(l_fields=['Race'])
 33  - print 'Race: ', p
 34  - p = t.group_discrimination(l_fields=['Sex'])
 35  - print 'Sex: ', p

-37  
 38  def test_causal_discrimination():
 39  t = themis.Temis(xml_fName="settings.xml")
 40  - p = t.causal_discrimination(l_fields=['Sex', 'Race'])

+37  
 40  def test_causal_discrimination():
 41  t = themis.Temis(xml_fName="settings.xml")
 42  - p = t.causal_discrimination(l_fields=['Sex', 'Race'])
 43  + print '\nCausal:'
basic functionality with sanity check done

Rico Angell committed on Oct 2, 2017

Git Basics – Git Commit Object

Message

Author or committer along with date

Parent
Git Basics – Git Commit Object

- Message
- Parent
- Hash

Author or committer along with date
Git Basics – Git Commit Object

- Message: Initially entire file then diffs.
- Parent: Author or committer along with date.
- Hash: 1 parent c82093 commit 2c2ec0a72ce21615da5e3888af32d929528087a65

Rico Angell committed on Oct 2, 2017

Showing 2 changed files with 108 additions and 64 deletions.
Git Basics

Making Queries
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 Basics: *Making Queries* -- `git log` ...Some Examples

git log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD
%C(reset) %C(bold green)(%ar)%C(reset) %C(bold cyan)
(committed: %cD)%C(reset) %C(auto)%d%C(reset)%n'
%C(white)%s%C(reset)%n'
%C(dim white)- %an <%ae>
%C(reset) %C(dim white)(committer: %cn <%ce>))%C(reset)'
Git Merge
Git Merge

NAME

git-merge - Join two or more development histories together

SYNOPSIS

[-s <strategy>] [-X <strategy-option>] [-S <keyid>]
[--no-allow-unrelated-histories]
[--no-rereere-autoupdate] [-m <msg>] [<commit>...]

git merge --abort

git merge --continue

DESCRIPTION

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by `git pull` to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.

Assume the following history exists and the current branch is “main”:

```
A---B---C topic
|
D---E---F---G master
```
Git Rebase
Git Rebase

NAME

git-rebase - Reapply commits on top of another base tip

SYNOPSIS

```bash
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase >]

<upstream> [<branch>]

<branch>

--root [<branch>]

```
git rebase --continue | --skip | --abort | --quit | --edit-todo

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(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 rebase will abort.
Changing Commit History with Rebase

- Git rebase lets us change our commit history
- rebase 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?
Why use Git Rebase?

- Clean history
- If you are issuing a pull request it's nicer for folks looking over it to decide to include it
- Clean "narrative"
- Branches can look messy
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 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 class: Thursday (September 22)

- First in-class exercise
- On using git (today is a prelude with useful info)
- Form 2-, 3-, or 4- person teams
  - Use Moodle to self-select a team from now until Thursday at 11:59 PM (a little before midnight)