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
Next class: Thursday (February 10)

- 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

1. I update my checkout (working copy)
2. I edit
3. I update my checkout again
4. I merge changes if necessary
5. 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 laptop working copy
    - Alice’s Desktop Repository
      - Alice’s Desktop working copy
  - Bob’s Repository
    - Bob’s working copy
  - Yuriy’s Desktop Repository
    - Yuriy’s Desktop working copy
  - Yuriy’s Laptop Repository
    - Yuriy’s laptop 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
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
    git-_____ - _____ 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 _____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**

```markdown
git-checkout - Switch branches or restore working tree files
```

**SYNOPSIS**

```markdown
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
.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 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

Rico Angell committed on Oct 2, 2017

Showing 2 changed files with 108 additions and 64 deletions.

`Themis2.0/test/test_themis.py`

```python
import pytest
import themis
```

```python
def test_get_test_result():
    t = themisThemis(xml_f_name=\"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 = themisThemis(xml_f_name=\"settings.xml\")
    _, p = t.causal_discrimination(l_fields=[\"Sex", \"Race\"])
```

```python
def test_group_discrimination():
    t = themisThemis(xml_f_name=\"settings.xml\")
    _, p = t.group_discrimination(l_fields=[\"Sex", \"Race\", \"Age\", \"Income\"]):
        _, p = t.group_discrimination(l_fields=f)
        print f, \"-> \", p
```

```python
def test_causal_discrimination():
    t = themisThemis(xml_f_name=\"settings.xml\")
    _, p = t.causal_discrimination(l_fields=[\"Sex", \"Race\"])
    print \"\Causal: \"
basic functionality with sanity done

Rico Angell committed on Oct 2, 2017

Showing 2 changed files with 108 additions and 64 deletions.

```python
# Themis2.0/test/test_themis.py
import pytest
import themis

# def test_get_test_result():

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"])  # fix import themis
    print "Sex: ", p

def test_causal_discrimination():
    t = themis.Themis(xml_fname="settings.xml")
    _, p = t.causal_discrimination(l_fields=["Sex", "Race"])
    print "\Causal:
    # for f in t..all_relevant_subs(["Sex", "Race", "Age", "Income"]):
    #    _, p = t.group_discrimination(l_fields=f)
    #    print f, "--> ", p
```

+ from itertools import chain, combinations

1 parent 1c8203 commit 2c2ec8a72ce21615dade3868af3f2d9298528b07a65

Unified Split

View
basic functionality with sanity check done

Rico Angell committed on Oct 2, 2017

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

Message

Author or committer along with date

Parent
Message
Parent
Hash
Author or committer along with date
Git Basics – Git Commit Object

Message

Author or committer along with date

Initially entire file then diffs

Parent

Hash
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-]rerere-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
NOTE) Git (and GitHub) are currently switching the term “master” to “main”
Git Rebase
Git Rebase

NAME

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

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase >] [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase >] --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 (February 10)

- 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)
Final project: Topic selection

● Each team of 2, 3, or 4 will carry out one of the following projects:
  ○ MSR Mining Challenge
  ○ Replication Study (e.g., Automated Test Generation)
  ○ ML Development Toolkits (e.g., Weights & Biases)
  ○ EleNa: Elevation-based Navigation

● The key phases of the project are: topic selection, mid-point presentation, final presentation (and deliverables)

● More details available here: https://people.cs.umass.edu/~hconboy/class/2022Spring/CS520/finalProject.pdf