To Ponder

If 6 cats kill 6 rats in 6 minutes, how many cats will be needed to kill 100 rats in 50 minutes?

(Hint: it’s not 50 cats)
Git: (Distributed) Version Control

Lecture 2
The Need for Version Control

- Track evolution of a software artifact
  - Development is often non-linear
    - Older versions need to be supported
    - Newer versions need to be developed
  - Development is non-monotonic
    - May need to undo some work, go back to an older version, or track down when a mistake was introduced

- Facilitate team-based development
  - Multiple developers working on a common code base
  - How can project be edited simultaneously?
Key Idea: A Repository

- **Repository** = working tree + store + index
  - Warning: "Repository" often (incorrectly) used to mean just the store part!
- **Working tree** = project itself
  - Ordinary directory with files & subdirectories
- **Store** = history of project
  - Hidden directory: don’t touch!
- **Index** = virtual snapshot
  - Gateway for moving changes in the working tree into the store (aka “stage”, “cache”)
- **History** = DAG of commits
  - Each commit represents a complete snapshot of the entire project
File Structure of a Repository

~/.mashup/

|-- css/
|   |-- buckeye-alert-resp.css
|   |-- demo.css
|-- demo-js.html
|-- Gemfile
|-- Gemfile.lock
|-- .git/
|   |-- HEAD
|   |-- index
|   |-- ...etc...
|-- .gitignore
|-- Rakefile
|-- README.md
|-- ...etc...
Conceptual Structure

store
~/mashup/.git/

working tree
~/mashup/

index
~/mashup/.git/index
A History of Commits

- Commit b
- Revision $\beta$
- d's parent is c
- Working tree
  ~mashup/

- Store
  ~mashup/.git/

- Index
  ~mashup/.git/index
History is a DAG

- Every commit (except the first) has 1 or more parents

Initial commit has no parents

- e has 1 parent
- i has 2 parents

store
### Example View of DAG

<table>
<thead>
<tr>
<th>Graph</th>
<th>Rev</th>
<th>Branch</th>
<th>Description</th>
<th>Age</th>
<th>UTC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>default</td>
<td>Merge</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 17:01:24</td>
</tr>
<tr>
<td>31</td>
<td>default</td>
<td>added text, more gam</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 17:00:25</td>
</tr>
<tr>
<td>30</td>
<td>default</td>
<td>added another marker</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 16:55:45</td>
</tr>
<tr>
<td>29</td>
<td>default</td>
<td>added tester file just</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 16:51:39</td>
</tr>
<tr>
<td>28</td>
<td>default</td>
<td>Merge</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 08:34:40</td>
</tr>
<tr>
<td>27</td>
<td>default</td>
<td>Wrote a new open lab</td>
<td></td>
<td>4 months</td>
<td>2012-05-30 08:32:35</td>
</tr>
<tr>
<td>26</td>
<td>default</td>
<td>minor changes to initi</td>
<td></td>
<td>6 months</td>
<td>2012-04-18 17:07:20</td>
</tr>
<tr>
<td>25</td>
<td>default</td>
<td>Merge</td>
<td></td>
<td>6 months</td>
<td>2012-04-18 05:27:02</td>
</tr>
<tr>
<td>24</td>
<td>default</td>
<td>Added lab plan WIP</td>
<td></td>
<td>6 months</td>
<td>2012-04-18 05:21:09</td>
</tr>
<tr>
<td>23</td>
<td>default</td>
<td>Revised summary.txt</td>
<td></td>
<td>6 months</td>
<td>2012-04-04 17:17:44</td>
</tr>
<tr>
<td>22</td>
<td>default</td>
<td>Merge</td>
<td></td>
<td>6 months</td>
<td>2012-03-28 12:23:42</td>
</tr>
<tr>
<td>21</td>
<td>default</td>
<td>Added minutes and s</td>
<td></td>
<td>6 months</td>
<td>2012-03-28 12:20:02</td>
</tr>
<tr>
<td>20</td>
<td>default</td>
<td>added notes on Obser</td>
<td></td>
<td>7 months</td>
<td>2012-03-06 21:36:29</td>
</tr>
<tr>
<td>19</td>
<td>default</td>
<td>Added App Inventor</td>
<td></td>
<td>7 months</td>
<td>2012-03-06 04:02:33</td>
</tr>
</tbody>
</table>
Example View of DAG

$ git log --oneline --graph

* 1618849 clean up css
*  d579fa2 merge in improvements from master
|\ |  * 0f10869 replace image-url helper in css
*  b595b10 add buckeye alert notes
*  | a6e8eb3 add raw buckeye alert download
|\ * b4e201c wrap osu layout around content
*  e9d3686 add Rakefile and refactor schedule loop
*  515aaa3 create README.md
*  eb26605 initial commit
Commit

- Each commit is identified by a hash
  - 160 bits (i.e., 40 hex digits)
  - Practically guaranteed to be unique
  - Can use short prefix of hash if unique

$ git show --name-only
16188493c252f6924baa17c9b84a4c1baaed438b
Author: Paul Sivilotti <user.pags@server.fake>
Date:   Mon Mar 31 15:30:50 2014 +0200

    clean up css

    source/stylesheets/_site.css
History is a DAG

- A better picture would label each commit with its hash (prefix)

- But in these slides we abbreviate the hash id's as just: 'a', 'b', 'c'...
Nomenclature: Branch

- *Branch*: a pointer to a commit
- Different from "branch" in DAG's shape
Nomenclature: HEAD

- **HEAD**: a special reference, (usually) points to a branch
Nomenclature: HEAD

- Useful to think of HEAD as being "attached" to a particular branch
$ git log --oneline --graph --decorate

* 1618849 (HEAD -> master) clean up css
* d579fa2 (alert) merge in improvements from master
|\  
| * 0f10869 replace image-url helper in css
* | b595b10 add buckeye alert notes
* | a6e8eb3 add raw buckeye alert download
|/
* b4e201c wrap osu layout around content
* e9d3686 add Rakefile and refactor schedule loop
* 515aaa3 create README.md
* eb26605 initial commit
A "Clean" Repository

$ git status
On branch master
nothing to commit,
working directory clean
Edit Files in Working Tree

- Add files, remove files, edit files...

Diagram:

- Files: a, b, c, d
- Branches: maint, master
- HEAD: now differs from index
- wt
- ind
- δ
- ε
Edit Files in Working Tree

- Add files, remove files, edit files...

$ git status
On branch master
Changes not staged for commit:
  modified:  css/demo.css
```bash
$ git add --all .
```
Add: Working Tree → Index

$ git add --all .

$ git status
On branch master
Changes to be committed:
  modified: css/demo.css
Commit: Index → Store

$ git commit

HEAD advanced (with attached branch!)
new commit added to store

parent is old HEAD
DAG extended

Store changed!

unaffected (but now clean)
The (New) State of Repository

Diagram:

- Nodes: a, b, c, d, e
- Edges: α → β, β → γ, γ → δ, δ → ε
- Labels: maint, master
- Branches: wt, ind

The diagram illustrates the state of a repository with nodes representing files or branches and arrows indicating the relationships between them.
Creating a New Branch

$ git branch fix
Checkout: Changing Branch

$ git checkout fix

Store *unaffected* (apart from HEAD)
Same DAG, branches
Checkout: Changing Branch

$ git checkout maint

- HEAD moved

Advice: checkout *only* when wt is clean
Edit Files in Working Tree

- Add files, remove files, edit files...

Diagram:
- HEAD
- maint
- master
- wt
- now differs from index
- fix
- ind
Add & Commit: Update Store

$ git add --all .
$ git commit
Merge: Bringing History together

- Bring work from another branch into current branch
  - Implemented features, fixed bugs, etc.
- Updates current branch, not other
Merge – Case 1: Ancestor

- HEAD is an ancestor of other branch
Fast-Forward Merge

$ git merge master
Merge – Case 2: No Conflicts

\[ \alpha \xrightarrow{\beta} \beta \xrightarrow{\gamma} \gamma \xrightarrow{\delta} \delta \xrightarrow{\theta} \theta \xrightarrow{\varphi} \varphi \]

- HEAD
- master
- wt
Merge Automatically Commits

$ git merge maint

![Diagram showing the git merge process with nodes labeled a, b, c, d, e, f, g, and connecting arrows labeled α, β, γ, δ, ε, θ, and μ.](image)
Merge – Case 3: Conflicts Exist

$ git merge maint

HEAD
master

maint

files with conflicts marked

files that could be merged automatically
Merge: Resolve Conflicts

$ emacs somefile

files with conflicts resolved
Merge with Conflicts: Add

$ git add somefile
Merge with Conflicts: Commit

$ git commit
Summary

- Repository = working tree + store
  - Store contains history
  - History is a DAG of commits
  - References, tags, and HEAD

- Commit/checkout are local operations
  - Former changes store, latter working tree
This is Git. It tracks collaborative work on projects through a beautiful distributed graph theory tree model.

Cool. How do we use it?

No idea. Just memorize these shell commands and type them to sync up. If you get errors, save your work elsewhere, delete the project, and download a fresh copy.
Git:  
*Distributed Version Control*
Demo

- Prep: Empty (but initialized) repo
- Linear development:
  - Create, edit, rename, ls -la files
  - Git: add, status, commit, log
- Checkout (time travel, detach HEAD)
- Branch (re-attach HEAD)
- More commits, see split in history
- Merge
  - No conflict
  - Fast-forward
What Does "D" Stand For?

- Distributed version control
  - Multiple people, distributed across network
- Each person has their own repository!
  - Everyone has their own store (history)!
  - Big difference with older VCS (eg SVN)
- Units of data movement: changeset
  - Communication between teammates is to bring stores in sync
  - Basic operators: fetch and push
Sarah's Repository

Head
master

wt

a ← b ← c ← d ← e

Sarah
And Matt's Repository
Some Shared History
Fetch: Remote Store → Local

sarah$ git fetch mt

new changesets added to store
Remote Repository Unchanged

Matt

HEAD
master

a ← b ← f ← g
Workflow: Merge After Fetch

sarah$ git merge mt/master
Remote Repository Unchanged

Matt

HEAD
master

a ← b ← f ← g
$ git log --oneline --graph --decorate --all

* 1618849 (HEAD-> master, origin/master) clean up css
* d579fa2 (alert) merge in improvements from master
* 0f10869 replace image-url helper in css
* b595b10 (origin/alert) add buckeye alert notes
* a6e8eb3 add raw buckeye alert download
* b4e201c wrap osu layout around content
* e9d3686 add Rakefile and refactor schedule loop
* 515aaa3 create README.md
* eb26605 initial commit
Your Turn

Show the state of Matt's repository after each of the following steps

- Fetch (from Sarah)
- Merge
Sarah and Matt's Repositories
Some Shared History

Sarah

Matt

HEAD    mt/master
master

HEAD    master

a ← b ← c ← d ← e

a ← b ← f ← g
Your Turn: Fetch

matt$ git fetch sr
Your Turn: Fetch

matt$ git fetch sr

Sarah

Matt
Fetch: After

matt$ git fetch sr
Your Turn: Merge

matt$ git merge sr/master
Your Turn: Merge

```
matt$ git merge sr/master
```
Merge: After

matt$ git merge sr/master
Pull: Fetch then Merge

- A "pull" combines both fetch & merge
  
matt$ git pull sr

- Advice: Prefer explicit fetch, merge
  
  - After fetch, examine new work
    
    $ git log --all #see commit messages
    $ git checkout #see work
    $ git diff #compare

  - Then merge

  - Easier to adopt more complex workflows (e.g., rebasing instead of merging)
Push: Local Store $\rightarrow$ Remote

- Push sends local commits to remote store
- Usually push one branch (at a time)
  
sarah$ git push mt fix
  
  - Advances Matt's fix branch
  - Advances Sarah's mt/fix remote branch

- Requires:
  1. Matt's fix branch must not be his HEAD
  2. Matt's fix branch must be ancestor of Sarah's

- Common practices:
  1. Only push to bare repositories (bare means no working tree, ie no HEAD)
  2. Get remote store's branch into local DAG (ie fetch, merge, commit) before pushing
Remote's Branch is Ancestor

Sarah

Matt

HEAD

fix

mt/fix

wt

HEAD

master
Sarah$ git push mt fix

Matt
Push: After

sarah$ git push mt fix

Matt

HEAD
master

working tree unaffected!
Commit/Checkout vs Push/Fetch

Local
- working directory
- staging area
- local repo

Remote
- remote repo

- git add
- git commit
- git push

- git checkout
- git merge

- git fetch
Common Topology: Star

- $n$-person team has $n+1$ repositories
  - 1 shared central repository (bare!)
  - 1 local repository / developer

- Each developer *clones* central repository
  - Cloning creates a remote called "origin"
  - Default source/destination for fetch/push

- Variations for central repository:
  - Everyone can read and write (ie push)
  - Everyone can read, but only 1 person can write (responsible for pulling and merging)
Common Topology: Star

Bare repository (no working tree)

Source: http://nvie.com/posts/a-successful-git-branching-model/
Workflow: Overview

- Configure git (everyone)
- Create central repo (1 person)
- Clone repo (everyone)
- As you work (everyone):
  - Commit locally
  - Fetch/merge as appropriate
  - Push to share
Workflow: Configure Git

- Each team member, in their own VM
  - Set identity for authoring commits
    
    $ git config --global user.name "Brutus Buckeye"
  
    $ git config --global user.email bb@osu.edu
  
  - Optional: diff and merge tool (eg meld)
    
    $ sudo apt-get install meld # to get tool
  
    $ git config --global merge.tool meld
  
    $ git config --global diff.tool meld
  
    # example use:
  
    $ git diffftool e9d36
Workflow: Initialize Central Rep

☐ One person, once per project:
  ■ Create central repository in group's project directory (/project/c3901aa03)
    
    $ ssh brut@stdlinux.cse.ohio-state.edu
    $ cd /project/c3901aa03
    $ mkdir proj1.git  # ordinary directory
  ■ Initialize central repository as bare and shared within the group
    
    $ git init --bare --shared proj1.git

☐ Note: Hosting services (eg GitHub, BitBucket) have a web interface for this step
Workflow: Initialize Repository

- Each team member, once, in their VM
  - Create local repository by cloning the central repository
    
    ```
    $ git clone ssh://brut@stdlinux.cse.ohio-state.edu//project/c3901aa03/proj1.git mywork
    ```
  
  - You will be prompted for your (stdlinux) password (every time you fetch and push too)
  
  - To avoid having to enter your password each time, create an ssh key-pair (see VM setup instructions)
Ignoring Files from Working Tree

☐ Use a .gitignore file in root of project

```bash
# Ignore auto-saved emacs files
*~

# Ignore bundler config
./bundle

# Ignore the default SQLite database
/db/*.sqlite3

# Ignore all logfiles and tempfiles
/log/*
/tmp/*
```
Workflow: Local Development

- Each team member repeats:
  - Edit and commit (to local repository) often
    
    ```
    $ git status/add/rm/commit
    ```
  - Pull others' work when can benefit
    
    ```
    $ git fetch origin # bring in changes
    $ git log/checkout # examine new work
    $ git merge, commit # merge work
    ```
  - Push to central repository when confident
    
    ```
    $ git push origin master # share
    ```
Git Clients and Hosting Services

- Recommended client: Command line!
- Alternative: Various GUIs
  - Linux: gitg, git-gui, git-cola, giggle
  - Win/mac GUI: SourceTree
  - IDEs: RubyMine
- Lots of sites for hosting your repos:
  - GitHub, Bitbucket, SourceForge, Google Code,...
- These cloud services provide
  - Storage space
  - Pretty web interface
  - Issues, bug tracking
  - Workflow with "forks" and "pull requests" to promote contributions from others
Clarity

\[ \text{git} \neq \text{GitHub} \]
Warning: Academic Misconduct

- GitHub is a very popular service
  - But only *public* repo's are free
  - Edu discount gives free *private* repo's
  - 3901 has an account ("organization") for private repo's (see class web site)

- Bitbucket has free private repo's, for small teams (< 5 collaborators)

- Public repo's containing coursework can create academic misconduct issues
  - Problems for poster
  - Problems for plagiarist
Summary

- Push/fetch to share your store with remote repositories
  - Neither working tree is affected
- Branches in history are easy to form
  - Committing when HEAD is not a leaf
  - Fetching work based on earlier commit
- Advice
  - Learn by using the command line
  - Beware academic misconduct
Group Formation

- Groups of 3 (or 4)
- Exchange contact information
- Each person choose a primary technical area:
  - HTML/CSS
  - JavaScript
  - Ruby
- Group constraints on choices:
  - Each technology must be represented
  - No more than 2 people per technology
- Choose a secondary interest as well
  - “Don’t Care” is fine (as primary or secondary)
Advanced: Undoing Mistakes

- Say you want to throw away all your uncommitted work
  - ie "Roll back" to last committed state
- Checkout won't work!
Reset: Discarding Changes

$ git reset --hard
$ git clean --dry-run # list untracked files
$ git clean --force # remove untracked files
Reset: Discarding Commits

```bash
$ git reset --hard HEAD~1
# no need to git clean, since wt was already clean
```

HEAD moved (and attached branch)

now unreachable
Mercurial (hg): Another DVCS

- Slightly simpler mental model
- Some differences in terminology
  - git fetch/pull ~ hg pull/fetch
  - git checkout ~ hg update
- Some (minor) differences in features
  - No rebasing (only merging)
  - No octopus merge (#parents <= 2)
- But key ideas are identical
  - Repository = working directory + store
  - Send/Receive changes between stores