Information Security Laboratory

02-Git
Sang Kil Cha
SCIENCE

ENGINEERING
Git?

• Created by Linus Torvalds in 2005

• *Distributed version control* system
Version Control System (VCS)?

A system that records changes to a set of files over time*.  

[Diagram showing Version 1, Version 2, Version 3, and a Repository, with arrows between versions]

Why Use VCS?
The Simplest VCS

- Create a ZIP file for each version
- Keep all the ZIP files on a storage or a database

What’s wrong?
RCS (1982)

• Store only diffs
• Work on single files
  − RCS creates a *local* DB file for each individual file

What’s wrong?
CVS (1985)

• Concurrent Version System
• Keep track of changes in a set of files
• Use client-server model
 CVS (1985)

In the world of open source software, the Concurrent Version System (CVS) has long been the tool of choice for version control. And rightly so. CVS itself is free software, and its non-restrictive *modus operandi* and support for networked operation – which allow dozens of geographically dispersed programmers to share their work – fits the collaborative nature of the open-source world very well. CVS and its semi-chaotic development model have become cornerstones of open-source.

- Ben Collins-Sussman
Problems with CVS

• Commits are not atomic
  − Interrupted commits can corrupt the repository

• For more, see “Bye-bye CVS. I've been Subverted”
  http://archive.oreilly.com/pub/post/byebye_cvs_ive_been_subverted.html
SVN (2000)

- SubVersion (SVN)
- Atomic commits
- Easier to handle conflicts than CVS
- Client-server model (centralized VCS) like CVS

Centralized model (SVN) vs. Distributed model (Git)?
Centralized VCS

Server DB

- Version 1
- Version 2
- Version 3

→

Client A
- Version 2
- Version 3

Client B
Distributed VCS

Server DB
- Version 1
- Version 2
- Version 3

Client A
- Version 1
- Version 2
- Version 3

Client B
- Version 1
- Version 2
- Version 3
Distributed VCS

• Fast
  – Most actions are done locally

• You can work offline
  – You don’t need to create a single large change set

• Small group of developers can share changes before showing them to everyone
Disadvantage of Distributed VCS?

• Slow initialization

• Requiring larger disk space than centralized VCS
Git (2005) Design Principles

• Be simple and fast

• Support fully distributed, non-linear development

• Must be able to handle large projects like the Linux kernel
Traditional VCS (CVS, SVN, etc.)

Checkins Over Time

Version 1

File A

Δ1

Version 2

Δ1

Version 3

Δ2

Version 4

Version 5

File B

Δ1

Δ2

File C

Δ1

Δ2

Δ3

Git’s View on VCS Data

Checkins Over Time

Version 1
- File A
- File B
- File C

Version 2
- A1
- B
- C1

Version 3
- A1
- B
- C2

Version 4
- A2
- B1
- C2

Version 5
- A2
- B2
- C3
Using Git
A File in Git Has 4 States

1. **Untracked** state: the file is not under version control
2. **Committed** state: the file is stored in your local repository
3. **Modified** state: the file is modified, but not committed yet
4. **Staged** state: the file is modified, and marked to be committed in the next commit snapshot

Check the current state of files:

$ git status
Creating/Cloning a Repo

$ git init

$ git clone <addr>
   -git clone git://git.kernel.org/pub/scm/git/git.git
Initial Configuration

- `/etc/gitconfig`: per system configuration
- `~/.gitconfig`: per user configuration
- `.git/config`: per repository configuration

```
$ git config --list
$ git config --global user.name “Sang Kil Cha”
$ git config --global user.email \  
  “sangkilc@kaist.ac.kr”
```
Initial Configuration (Cont’d)

On Unix:
$ git config --global core.editor vim

On Windows:
$ git config --global core.editor \ 
  "'C:/Program Files (x86)/Vim/vim80/gvim.exe' --nofork"

or

$ git config --global core.editor \ 
  "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"
Add, Diff, and Commit

Begin tracking files by adding them:
$ git add <file name(s)>

Modify the staged file, and check the status:
$ git status # checking the status
$ git diff # see the diff
$ git commit # and commit
Commit (Cont’d)

$ git commit

[master 42f00d] Your commit message appears here.
2 files changed, 2 insertions(+)
create mode 100644 XYZ

Branch Commit hash
See the History

$ git log

$ git log --oneline --decorate --graph --all
State Transition of Git

- Untracked
- Committed
- Modified
- Staged

- git add
- Editing
- git add
- git rm & commit
- git commit
- git commit
Commit Hash for Integrity

• Each individual commit (snapshot) has a “globally” unique identifier called **commit hash**

• 160-bit hex value (SHA1) of
  − The tree of files and directories in the commit
  − The parent commit hash(es)
  − The commit message
  − The author information (name, email, etc.)
  − Etc.
Undo State Modification

# Unstaging a staged file:
$ git reset HEAD <file name>

# Unmodifying a modified file:
$ git checkout -- <file name>
Tagging

Allows us to annotate a specific point in the revision history.

# List all the available tags
$ git tag

# Create an annotated tag
$ git tag -a v1.0 -m "Version 1.0"
$ git show v1.0 # show the tag along with the commit
Tagging (Cont’d)

# Create a lightweight tag (without annotation)
$ git tag v1.0

# Tag a specific commit
$ git tag -a v1.0 <commit hash>

# Push tags to upstream
$ git push origin v1.0
Branching

Each commit log has a name we call branch name
$ git merge develop
$ git branch -d develop
HEAD

• HEAD is a symbolic reference to a currently checked out branch
• We can move HEAD with (git checkout)
Tags vs. Branches

• They both point to a specific commit
• The main difference is that tags do not move, while branches can move
  – When we make a new commit to a master branch, master will point to the new commit
Remote Repository

# Show the current remote repositories:
$ git remote

# Add a remote repository:
$ git remote add <name> <URL>

This name is an alias of the URL

# Fetch all the information from a remote (name)
$ git fetch <remote name>
Remote Repository (Cont’d)

# Push to *upstream*

$ git push <remote name> <branch name>
HEAD

master

origin/master

origin/develop

HEAD

master

origin/master

foobar
Fast-Forward Merge

$\text{git fetch origin} \; \&\& \; \text{git merge origin/master}$
or
$\text{git pull}$
$ git fetch origin && \
  git merge origin/master \\
or \\
$ git pull
Non Fast-Forward Merge

1. $ git fetch origin &&
2. `git merge origin/master`
3. or
4. $ git pull

Diagram:
- HEAD
- master
- origin/master
- 1
- 2
- 4
- 5
$ git fetch origin &&
  git merge origin/master
or
$ git pull
Cherry-Pick

HEAD

master

foo/master

$ git cherry-pick foo/master~1
Commit hash differs!

$ git cherry-pick foo/master~1
Rebasing

$ git fetch origin &&
$ git rebase origin/master
Pretend local changes were made after origin/master

$ git fetch origin &&
$ git rebase origin/master
Reset vs. Checkout

- Checkout changes the current branch (where the HEAD points to)

- Reset changes which commit the current branch is pointing to
Reset vs. Checkout

HEAD

master

develop

1

2

4

$ git checkout develop
Reset vs. Checkout

$ git checkout develop
Reset vs. Checkout

$ git reset develop
Reset vs. Checkout

HEAD

master

develop

2

1

4

$ git reset develop
Engineering Practice with Git
Commit Often
Perfect Later
Publish Once
You (Developer) Break Things Up!

• Break the idea into small pieces, and commit each of the pieces separately
  – Without understanding the concept, you cannot do this

• Example: PE Parser
  – Single large commit:
    ▪ Add a PE parser
  – Multiple (hundreds of) commits:
    ▪ ...
    ▪ Separate logic for PE types in the header
    ▪ Check the magic number of a PE file
    ▪ Create PE header types
When Commits are Small

• Git blame becomes much useful
  - On a given line, you see a commit message: “Add this feature” vs. “Add updated X library for module Y”

• Other developers can see the history
  - Some stupid mistakes are shown, but clearly shows how logically we fixed the problem
  - We can make it beautiful later on
Software Engineering ~ Sausage Making

• Sausage making is a messy business
• Software engineering is the same
  − The end result may be a tasty program, but anyone looking at the process of how it was created (through inspection of the commits) may end up with a sour taste in their mouth.

Quoted from https://sethrobertson.github.io/GitBest Practices/#sausage_metaphor
Another Analogy: Movie Making

• Scenes are shot out of temporal order
• Cuts are very short
• Edit, splice every bit together to form individual cuts and scenes
Commit Guideline

• Commit early and often

• One good strategy is to create a typed template (with blank functions), and then fill them one by one (i.e., a commit per each function)

• If your modification is above 100 lines, then think if it is possible to break them into *multiple logical changes*
Hiding the Sausage Making Process?

• You can hide by only showing the last product
  − Merge commits later when we push to the master branch

• Use git add –p

• General workflow:
  
git add –p
  git diff –cached
  git commit
How to Write a Git Commit Message

This lecture is based on an online blog: http://chris.beams.io/posts/git-commit/
Which Looks Better?

$ git log --oneline -S --author cbeams --before "Fri Mar 26 2009"

e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal in r814. @Ignore-ing the
testCglibClassesAreReloadedJustInTimeForEnhancement() method as it turns out this was one of the culprits in the recent build breakage. The
classloader hacking causes subtle downstream effects, breaking unrelated tests. The test method is still useful, but should only be run on a manual
basis to ensure CGLIB is not prematurely classloaded, and should not be run as part of the automated build.
2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingVisitor to revision 794 + eliminated ConfigurationPostProcessorTests until
further investigation determines why it causes downstream tests to fail (such as the seemingly unrelated ClassPathXmlApplicationContextTests)
147709f Tweaks to package-info.java files
22b25e0 Consolidated Util and MutableAnnotationUtils classes into existing AsmUtils
7f96f57 polishing

$ git log --oneline -S --author pwebb --before "Sat Aug 30 2014"

5ba3d86 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
Concise and Consistent

Well maintained projects such as *Linux kernel* has the logs look like the latter
Why Commit Messages Matter?

• Useful commands
  – Git blame, revert, rebase, bisect, etc.

• Maintainable software
  – Long-term success of the project

• A hassle at first soon becomes habit, and eventually a source of pride and productivity

Keeping a commit message convention is as important as following coding convention
The Principle

• Diff shows *what* changed
• Log (Commit message) shows *why* changed

Every developer needs to regularly read commit messages
  – Do *git log*, and read them often!
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that).

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet

If you use an issue tracker, put references to them at the bottom, like this:

Resolves: #123
See also: #456, #789
Five Rules (in this Course)

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Wrap the body at 72 characters
1. Separate Subject from Body

• What if a commit is so simple to write the body text?
  − Don’t add the body. A single summary line is fine.

• Example commit message:
  Fix typo in introduction to user guide

• You don’t need to say what the typo was: you can see “git diff”
Good Example

$ git log
commit 42e769bdf4894310333942ffc5a15151222a87be
Author: Kevin Flynn <kevin@flynnsarcade.com>
Date:   Fri Jan 01 00:00:00 1982 -0200

Derezz the master control program

MCP turned out to be evil and had become intent on world domination. This commit throws Tron's disc into MCP (causing its deresolution) and turns it back into a chess game.
$ git log --oneline
42e769 Derezz the master control program

$ git shortlog
Kevin Flynn (1):
   Derezz the master control program

Alan Bradley (1):
   Introduce security program "Tron"

Ed Dillinger (3):
   Rename chess program to "MCP"
   Modify chess program
   Upgrade chess program
2. Limit the Subject Line to 50 chars

• Not a hard limit, but just a rule of thumb
  – 72 chars is the hard limit

• Shorter subject is more readable

• Forces authors to think about the most concise way to explain the commit
3. Capitalize the Subject Line

Accelerate to 88 miles per hour

vs.

accelerate to 88 miles per hour
4. Don’t End the Subject Line with “.”

Accelerate to 88 miles per hour

vs.

Accelerate to 88 miles per hour.

But, do use punctuation marks in the body!
5. Wrap the Body at 72 chars

- The same reason we use the 80 char limit for source code
- But, we give 8 char room for git to indent text while keeping everything under 80 chars overall
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that).

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet

If you use an issue tracker, put references to them at the bottom, like this:

Resolves: #123
See also: #456, #789
Activity #1

• Install LaTex on the host machine (not in the VM)
• Visit: https://www.latex-tutorial.com/tutorials/
• Finish up to “step 8”, using your public Github repository
• Deadline: before the next class (Wed. 3/8)