6.005 — Software Construction on MIT OpenCourseWare (https://ocw.mit.edu/ans7870/6/6.005/s16/) | OCW 6.005 Homepage (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/) **Spring 2016**

Reading 27: Team Version Control

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Review Git basics and the commit graph
- Practice multi-user Git scenarios

Git workflow

You've been using Git for problem sets and in-class exercises for a while now. Most of the time, you haven't had to coordinate with other people pushing and pulling to and from the same repository as you at the same time. For the group projects, that will change.

In this reading, prepare for some in-class Git exercises by reviewing what you know and brushing up on some commands. Now that you're more comfortable with Git basics, it's a good time to go back and review some of the resources from the beginning of the semester.

Review **Inventing version control (../05-version-control/#inventing_version_control)** : one developer, multiple developers, and branches.

If you need to, review **Learn the Git workflow (../../getting-started/#git)** from the *Getting Started* page.

Viewing commit history

Review **2.3 Viewing the Commit History (//git-scm.com/book/en/Git-Basics-Viewing-the-Commit-History)** from *Pro Git* .

You don't need to remember all the different command-line options presented in the book! Instead, learn what's possible so you know what to search for when you need it.

Clone the example repo from ***Version Control* (../05-version-control/#copy_an_object_graph_with_git_clone)** : **`https://github.com/mit6005/sp16-ex05-hello-git.git`**

Use `log` commands to make sure you understand the history of the repo.

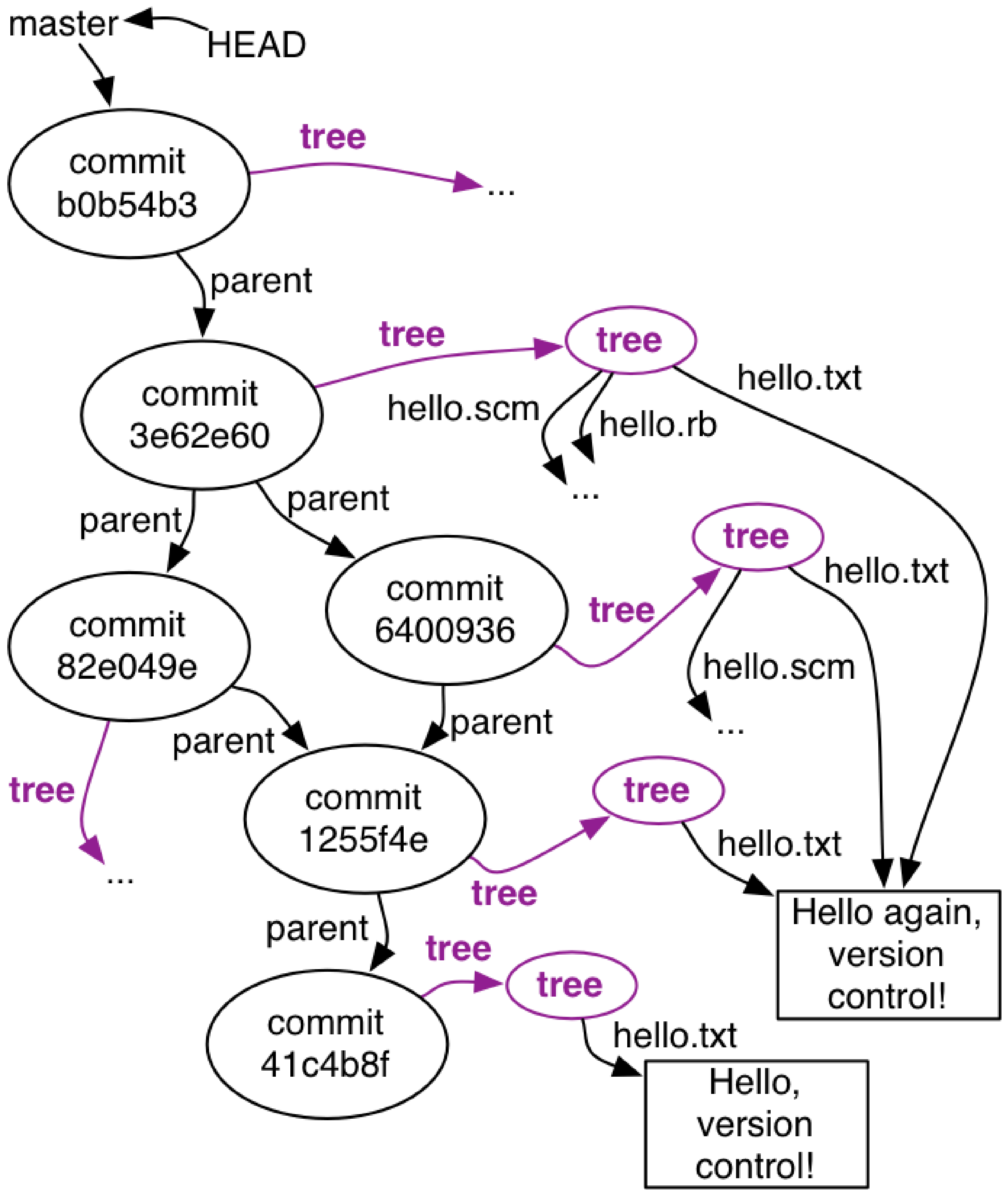
Graph of commits

Recall that the history recorded in a Git repository is a directed acyclic graph. The history of any particular branch in the repo (such as the default `master` branch) starts at some initial commit, and then its history may split apart and come back together, if multiple developers made changes in parallel (or if a single developer worked on two different machines without committing-pushing-pulling before the switch).

Here's the output of `git lol` (../../getting-started/#config-git) for the example repository, which shows an ASCII-art graph:

```
* b0b54b3 (HEAD, origin/master, origin/HEAD, master) Greeting in Java
*   3e62e60 Merge
|\
| * 6400936 Greeting in Scheme
* | 82e049e Greeting in Ruby
|/
* 1255f4e Change the greeting
* 41c4b8f Initial commit
```

And here is a diagram of the DAG:



In the `ex05-hello-git` example repo, make sure you can explain where the history of `master` splits apart, and where it comes back together.

Review **Merging (../05-version-control/#merging)** from the *Version Control* reading.

You should understand every step of the process, and how it relates to the result in the example repo.

Review the **Getting Started section on merges (../../getting-started/#merges)** , including merging and merge conflicts.

Using version control in a team

Every team develops its own standards for version control, and the size of the team and the project they're working on is a major factor. Here are some guidelines for a small-scope team project of the kind you will undertake in 6.005:

- **Communicate.** Tell your teammates what you're going to work on. Tell them that you're working on it. And tell them that you worked on it. Communication is the best way to avoid wasted time and effort cleaning up broken code.
- **Write specs.** Necessary for the things we care about in 6.005, and part of good communication.
- **Write tests.** Don't wait for a giant pile of code to accumulate before you try to test it. Avoid having one person write tests while another person writes implementation (unless the implementation is a prototype you plan to throw away). Write tests first to make sure you agree on the specs. Everyone should take responsibility for the correctness of their code.
- **Run the tests.** Tests can't help you if you don't run them. Run them before you start working, run them again before you commit.
- **Automate.** You've already automated your tests with a tool like JUnit, but now you want to automate running those tests whenever the project changes. For 6.005 group projects, we provide Didit as a way to automatically run your tests every time a team member pushes to Athena. This also removes "it worked on my machine" from the equation: either it works in the automated build, or it needs to be fixed.
- **Review what you commit.** Use `git diff --staged` or a GUI program to see what you're about to commit. Run the tests. Don't use `commit -a` , that's a great way to fill your repo with `println` s and other stuff you didn't mean to commit. Don't annoy your teammates by committing code that doesn't compile, spews debug output, isn't actually used, etc.
- **Pull before you start working.** Otherwise, you probably don't have the latest version as your starting point — you're editing an old version of the code! You're guaranteed to have to merge your changes later, and you're in danger of having to waste time resolving a merge conflict.
- **Sync up.** At the end of a day or at the end of a work session, make sure everyone has pushed and pulled all the changes, you're all at the same commit, and everyone is satisfied with the state of the project.

We don't recommend using features like branching or rebasing for 6.005-sized projects.

We do strongly recommend working together in the same place at the same time, especially if this is your first group software engineering experience.

Team project