L14: Version Control

At this point, you might feel like Git is just an exotic torture that the 6.005 staff have devised to make it hard for you to know which version of your code you have submitted for grading. In fact, Git and other version control systems are essential tools of the software engineering world. More or less every serious project, whether open source or proprietary in industry, is using version control. Without version control, coordinating a team of programmers, all editing the same project's code, can reach pull-out-your-hair levels of aggravation. Soon you'll begin the first group project in 005, and it will be very useful to learn and apply the lessons of version control.

If version control didn't exist, we would have to invent it

Git may seem confusing because so far we have been using it outside of the intended setting of team projects. The goal of this section of the lecture notes is to tell you a very plausible story about a small development team, and how they over time develop conventions to help them organize their collaboration, avoid losing code, and so on. Each convention they develop will turn out to motivate one of the features of version control systems like Git.

Incidentally, this section will also help you master one of the key cultural literacy components of computer science: using examples featuring characters named Alice and Bob. ;-

Let's start with Alice working on a pset by herself. Version control systems are still extremely useful in such settings.

Alice starts small with one file A.java in her pset.

At the last minute before she needs to hand in her pset code to be graded, she realizes she has made a big change that breaks everything. If only she could go back in time and retrieve a past version!

Of course, a simple discipline of saving backup files would get the job done.
Alice uses her judgment to decide when she has reached some milestone that justifies saving the code. Maybe she concretely saves the versions of `A.java` as `A.1.java`, `A.2.java`, and `A.java`. She follows the convention that the most recent version is just `A.java` to avoid confusing Eclipse. We will call the most recent version the head.

Now when Alice realizes that Version 3 is fatally flawed, she can just copy Version 2 back into the location for her current code. Disaster averted!

But what if Version 3 included some changes that were good and some that were bad? Alice can compare the files manually to find the changes, and sort them into good and bad changes. Then she can copy the good changes into Version 2.

This is a lot of work, and it's easy for the human eye to miss changes. Luckily, there are standard software tools for comparing file text. In the UNIX world, one such tool is `diff`. The high-level point is that we want a way to get a report on what has changed between two versions.

Alice wants to be prepared in case her laptop gets run over by a bus, so she also saves a backup of her work in the cloud, uploading the contents of her working directory whenever she's satisfied with its contents.

Presumably the cloud provider provides its own automatic back-up service that adds further certainty that the data won't be lost. Now Alice can resume work on the pset on a fresh machine, retaining the ability to time-travel back to old versions at will.

Furthermore, she can develop her pset on multiple machines, using the cloud provider as a common interchange point. Alice makes some changes on her laptop and uploads them to the cloud. Then she downloads onto her desktop machine at home, does some more work, and uploads the improved code (complete with old file versions) back to the cloud.
If Alice isn’t careful, though, she can run into trouble with this approach. Imagine that she starts editing `A.java` to create “Version 5” on her laptop. Then she gets distracted and forgets about her changes. Later, she starts working on a new “Version 5” on her desktop machine, including an orthogonal set of improvements. We’ll call these versions “5L” and “5D,” for “laptop” and “desktop,” respectively.

When it comes time to upload changes to the cloud, there is an opportunity for a mishap! Alice might copy all her local files into the cloud, causing it to contain Version 5D only. Later Alice syncs from the cloud to her laptop, potentially overwriting Version 5L, losing the worthwhile changes.

What Alice really wants above is a merge, to create a new version based on the two Version 5's.

OK, considering just the scenario of one programmer working alone, we already have a list of operations that should be supported by a scheme for managing source code: reverting to a past version, comparing two different versions, pushing full version history to another location, pulling history back from that location, and merging versions that are off-shoots of the same previous version.

Now let’s add into the picture Bob, another developer. The picture isn’t too different from what we were just thinking about.

Alice and Bob here are like the two Alices working on different computers. Of course, they no longer share a brain, which makes it even more important to follow a strict discipline in pushing to and pulling from the shared cloud server. The two programmers must somehow coordinate on a scheme for coming up with version numbers. Ideally, the scheme allows us to assign clear names to whole sets of files, not just individual files. (Files depend on
other files, so just thinking about them in isolation allows inconsistencies.) This can be a lot of work, which is one of the reasons why version control is great. But, first, let's consider some other issues Alice and Bob would run into in the manual world.

Merely uploading new source files is not a very good way to communicate to others the high-level idea of a set of changes. So, let's add to each directory a log that records for each version who wrote it, when it was finalized, and what the changes were (in the form of a short human-authored message).

Pushing to another copy of the files now gets a bit more complicated, as we want to merge the logs automatically, which is easier to do than for Java files, since logs have a simpler semantics; but without tool support, Alice and Bob will need to do it manually! We also want to enforce consistency between the logs and the actual sets of available files: for each log entry, it should be easy to extract the complete set of files that were current at the time the entry was made.

Now that we have logs, all sorts of useful operations are enabled. We can look at the log for just a particular file. In other words, we show a view of the log restricted to those coordinated changes that involved modifying some file. We can also use the log to figure out which change contributed each line of code, or, even better, which person contributed each line, so we know who to complain to when the code doesn't work. This sort of operation is a serious pain to do manually! The automated operation in version control systems is called annotate or blame.

In large group projects, or even in simpler settings, it sometimes makes sense for a subset of the developers to go off and work on branches, parallel code universes for, say, experimenting with a new feature. The other developers don't want to pull in the new feature until it is done, even if several coordinated versions are created in the mean time. Even a single developer can find it useful to create a branch, for the same reasons that Alice was originally using the cloud server despite working alone.

Maybe, in our example project, Alice wants to create a branch to experiment with adding some foobazes.
Alice maintains her own separate copy of all the project data, on her own project server. While she's off in branch-land, she only pulls from and pushes to her branch copy, so that Bob isn't bothered with preliminary foobaz-related changes. When Alice feels her changes are in a state ready to share with Bob, she *merges* her latest foobaz version with the latest version in the main shared server, and then *pushes* to the shared server. Bob can now pull and incorporate Alice's foobaz changes without any further work.

In general, it's very useful to have many different shared places for exchanging project state. There may be multiple branch locations at once, each shared by several programmers. With the right permission set-up, any programmer can pull from or push to any location, creating serious flexibility in cooperation patterns.

Now we're ready for the punchline (which was foreshadowed pretty severely): Git does all of these things for you! So do many other version control systems. Traditional *centralized* version control systems, like CVS and Subversion, do a subset of the things we've discussed. They're centralized in that the only cooperation graph they support is one master server and a set of copies of its contents that just communicate directly with the master server. In contrast, *distributed* version control systems like Git and Mercurial allow all sorts of different cooperation graphs, which may be helpful in 6.005 projects because it lets teams (or subsets of teams) experiment easily with alternate versions of code, easily merging versions together as they are decided to be good ideas.

**Git basics**

Like most other version control systems, Git is most fundamentally accessed using a set of *command-line tools*. In 6.005, we try to avoid requiring any command-line savvy of students, but here the command-line perspective just makes everything so much simpler! In a nutshell, a command line is a software interface based on entering text inputs and reading text responses, rather than clicking buttons and so on in a GUI (graphical user interface). Apologies to those readers unfamiliar with command lines; Piazza is a good place to get advice about such things.
It is possible to use Git solely within Eclipse via the EGit interface, but we will stick to the command-line version here. Without further ado, here are the Git commands for the actions that Alice and Bob used in our running example.

The first step is to tell your Git system your name and e-mail address, which will appear in logs.

```
git config --global user.name "Your Name"
git config --global user.email "you@mit.edu"
```

Alice and Bob only followed a convention for files in a directory, but with version control systems we instead work explicitly with repositories, which are principled versions of the directories-with-conventions from the earlier examples. We can create a repository in the foo subdirectory of the current directory with:

```
git init foo
```

Now we start editing files in this new directory. As new files become ready to incorporate in the official record, we add them to the repository.

```
git add A.java
```

In general, after any further modifications to a file, we will need to re-add it, to tell Git to include our changes in the next official version.

After adding all of our initial set of files, we are ready to formalize a single coordinated version of all the files, known as a commit.

```
git commit -m "Log message goes here"
```

Before committing, it's useful to ask Git which changes would go into the commit. The output can also help us catch if we forgot to include some files.

```
git status
```

It's generally a bad sign if any files appear as “Changed but not updated”! You usually want to git add all such files, but sometimes there are files that obviously don't belong in version control. At the root of your repository, you may create a file .gitignore storing patterns like bin and *.class identifying filenames that should be ignored by version control. (Don't forget to git add .gitignore if you want to share this file with others!)

After committing, our version appears in the log, with a long string of letters and numbers assigned to it as a globally unique name. (“Globally unique” means that this name will identify this version, to anyone anywhere who ever gets ahold of it.)

```
git log
```

```
# Output:
commit c792cc884d103e55dd4e7cad910104348fc19b05
Author: Your Name <you@mit.edu>
Date:   Tue Oct 9 15:04:22 2012 -0400
Log message goes here
```
After making further changes to files, you can ask Git to tell you exactly what is now different, compared to the most recent commit.

    git diff

Oh no! We accidentally deleted something important in A.java! Time to restore the most recent official version.

    git checkout A.java

We could also restore to a specific past version.

    git checkout c792cc884d103e55dd4e7cad910104348fc19b05 A.java

Now let's create a remote copy of our repository. First, in our Athena space, we run the following to create a bare Git repository in directory foo. A bare repository doesn't contain a workspace for editing files, but instead stores version information.

    git init --bare foo

Now we can clone to that repository. First, we tell Git that this is our default repository (called origin), and then we push all our changes to it.

    git remote add origin ssh://athena.dialup.mit.edu/mit/user/y/yo/you/foo

    git push --all

More frequently, a new local repository is created as a clone of a remote repository, like so:

    git clone ssh://athena.dialup.mit.edu/mit/user/y/yo/you/foo

A command like this automatically configures the source repository as the origin. Once further changes appear in the source repository, you can integrate them locally with:

    git pull

And, in either scenario for initializing your local repository, pushing changes to origin after one or more commits is as simple as:

    git push

One more basic class of operations has to do with removing a file from version control. If the file hasn't been committed yet:

    git reset A.java

Or if it has been committed, the following both removes the file from your working directory and asks to remove it from version control with the next commit:

    git rm A.java

It's also possible to rename files within version control.

    git mv A.java B.java
When things go wrong....

Each time you run `git pull`, the latest changes from the origin repository will be integrated. Sometimes one of these changes is to a file that you have also modified yourself, in uncommitted changesets. Git tries to be smart and automatically *merge* the changes textually, but sometimes it is not obvious enough how to combine the changes, in which case Git warns you of a conflict and asks you to resolve the changes manually. The files will be marked up with information on the locations of the conflicts, plus data on what was present in each version of the file. After resolving all the changes, add the affected files, commit, and push. Now others can pull your changes without having to think about the merge themselves. If Git gets ornery trying to pull after a merge, the following alternate incantation might help:

```
git pull origin master
```

We also sometimes want the *annotate/blame* operation described in the Alice & Bob example. Here's how we can generate an annotated version of a file:

```
git annotate A.java
```

Output is the file content with information on revision number, responsible person, and time of commit for each line.

Other fun stuff to do with Git

To learn more about Git, try a web search for tutorials. There are plenty out there. Here are a few other features that might be worth learning about:

**Tags**

By default, revisions are named with long strings of random-seeming characters. It can be helpful to assign more human-readable names like “release1.0” or “finallyFixedThatBug.” The *tags* feature supports that style, and tags may be mentioned anywhere that usually needs a long version string.

**Branches**

Git has special support for multiple *branches* (as in the end of the Alice & Bob example) *within a single repository*. You can also just create separate repositories for different branches; the special branch features seem to be just a performance optimization. So, you can trade off between performance/space usage and simplicity.

**Bisecting**

It's easy to introduce a bug in code that goes undetected for a while. Git's *bisection* feature helps you binary-search through the sequence of commits, to find the first one that broke something.