A Guide to Git

Andrew Au

Riley Shenk

Bethany Wong

University of Maryland, College Park

ENGL393 Section 0101

Table of Contents

Introduction

If you are a Computer Science major at the University of Maryland who has completed at least CMSC131, then you have been forced to work alone on all of your coding assignments. Consequently, if you were to go to your first internship or job and asked to work with a team of software engineers this would be a completely new experience for you. Though you may not be intimidated at first, you will quickly realize this introduces a lot of complexities to your job that you never experienced while working alone in school. By following this Guide to Git you will learn the tools needed to positively contribute to the group from day one. Please note that the setup is geared specifically towards Mac users. However, git commands are universal for any operating system.

Git is a version control system often used in software development. Now, what is "version control system"? Similar to the concept of Google Docs, a group of people can collaborate on an assignment, update different versions of a document to a common space, and keep track of changes. In git, users are able to take open source code, make changes, and update back into the repository (the common space) for collaborators to view and see previous versions.

This manual will provide step by step instructions on the basics of using git including the following: the installation process, creation of a repository, and necessary linux and git commands. A simple practice project will be incorporated for you to work on with a group of people to make sure you are able to apply the knowledge as you learn. Depending on your programming skills, this manual may take approximately 1-2 hours to complete. You can always reference back to this manual when you want to get started on creating a program with others.

By the end of this manual, you can expect to become proficient in git, have experience elaborating on others' code, and be prepared to create your own personal projects. Ultimately, you will be prepared for the increasingly competitive job market and have valuable experience to boost your resume and to give off a good impression to recruiters.

After interviewing multiple upperclassmen and alumni who have been exposed to programming jobs, they have emphasized that the single best thing you can do for your resume is to get involved in the open-source community through working on existing projects in GitHub. This shows employers that you are genuinely interested in software and that you love coding enough to work on something you are not getting paid for. To do this, you have to know how to collaborate on a team made up of people you have never met before, probably will never meet, and people who potentially aren't even in the same time zone as you. Through completing this guide, you will have the tools necessary to start collaborating on any project that sparks your interest on GitHub.

Please do not use git or GitHub for your school projects because this violates the school's academic integrity policy. Caught doing so will lead to an XF for the course, which will deter recruiters and will revoke your diploma if you graduated and uploaded code from a major requirement course. Further details can be found here.

Git Informed

Git & GitHub

Git and GitHub are

Git is a command line tool created by Linus Torvalds in 2005. A free tool that was very similar to git used to exist and was called BitKeeper. However, the relationship between the BitKeeper owners and the open source community broke down and BitKeeper became a paid service. In the wake of this, Linus Torvalds created an open source version of BitKeeper and called it git. Later, git became an open source project just like his earlier creation: Linux. Contrary to what people typically think, the word git does not stand for anything; git is merely one of the shortest words not yet taken by operating systems for other uses.

Github is a server for git repositories founded by PJ Hyett, Chris Wanstrath, and Tom Preston-Werner. They launched their website in 2008 and has quickly grown to today having 11 million developers contributing towards 28 million different software projects. Github provides a graphical user interface to use git, server space for users to store their code publicly for free, and paid server space to store code privately. Unlike git that limits users to the command line, GitHub is available as a website, mobile app, and desktop application.

The Structure of Git

At a glance, git seems like a fairly simple concept: make changes on a project and publish these changes to a repository for others to elaborate on. However, once you start using git, things may get a bit confusing with all the commands, branches, and stages. In fact, git has been so frustrating for beginners to learn that comics such as **Figure 1**, have begun emerging to make stabs at its usability.

In this section, you will get a better understanding of the basic structures in git including the following features: branching and merging. After understanding the foundation of how everything works, git can be less complex to work with.

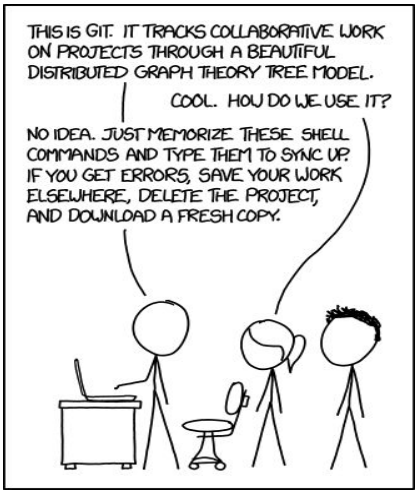

Figure 1: xkcd.com comic in reference to using git.

Branching

The main feature of git that makes it so widely used is its fast and memory efficient branching model. To make a **branch** means to create a deep copy of whatever source code you are currently looking at. The new branch that has been created will be independent of the original source code until you are happy with the changes you have made. This is much different than a tool like Google Docs where the changes you make are made immediately visible to everyone you are sharing your files with.

You initially begin in a *master* branch that is the root of your program and contains the original copies of the source code that everyone sees. When you are ready to start working on the repository you can make as many branches as you would like. **Figure 2**, illustrates this point by showing what it would look like to have a *develop* branch along with individual *topic* branches. When you are happy with the changes made in each of your branches git makes it easy to merge your changes back to the *master* branch for your collaborators to see. Don't worry, you will be guided through how and when to make a branch and merge it back to the *master* branch later in this guide.

Figure 2: A graphic representation of the branching model. Retrieved from git-scm.com.

The advantages of branching consist of the following:

- **Easier to test and experiment code.** Since branches are independent of each other, you can create branches purely for experimenting different code implementations. The best part is that if the code doesn't work, you can simply delete the branch and move on. Otherwise, with a few commands, you can apply your work to the main source code.
- **Efficiently collaborate on programming.** Instead of having to compare which lines you and your partner(s) changed to merge changes or work on the same and only copy of your program, you and your partner(s) can create your own branches with your own copy of the program. After completing your set of changes, git

allows you to easily merge and identify changes among you and your partner(s) work.

- **Organize development.** Branches can be assigned roles, in which each branch can be dedicated to working on a feature of the program. That way, you do not need to worry about making the other features functional before being able to work on another feature.

Merging

We talk a lot about git being able to merge changes together, but how exactly does it work? In order to understand the mechanics of merging, you will first need to understand the levels in git and the different states of your file.

Levels in Git

Git is structured into three general levels that files go through: working directory, staging area, and repository.

- **Working directory.** This would be the branches you are working in. Here, you are creating additional files, implementing new code, or editing existing code.
- **Staging area.** Git refers to this area as the "intermediate area" where changes can be reviewed before committing it to the repository.
- **Repository.** Your "common area" shared with your peers. This is where all your project files are stored and final updates are published to.

Figure 3 shows an example of how the levels relate to each other. Once you complete your changes, you will add the files you want into the staging area for further review. After your are sure you want to continue with those changes, your files leave the staging area once you choose to commit. Committing the files will sync the changes to the existing files in your repository and replace the old files with your new files and add any additional files. By then, your peers will be able to see the changes you have made.

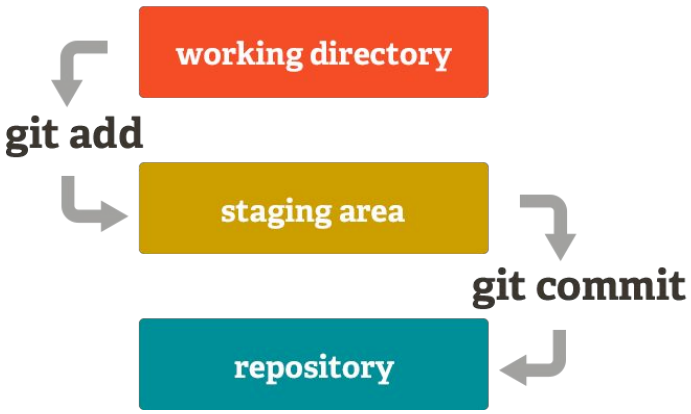

Figure 3: The different levels of git gives you the control to ensure that you add the changes you want and commit all the changes added to the repository. Retrieved from git-scm.com.

States of Your File

Going into further detail, your files can enter four states before getting to the staging area: untracked, unmodified, modified, and staged. The file is either untracked or tracked, but tracked can be divided to unmodified, modified, and staged. All your files are considered as tracked files, specifically unmodified files, when you first get them after cloning a repository since you have not done anything to them yet. **Figure 4** provides a visual representation of the file state cycle and how the states interact with each other.

- **Untracked.** Files in your working directory that were not added to the staging area in the previous commit.
- **Unmodified.** As the title says, these are the files that have not been edited at all.
- **Modified**. As the title says, these are the files that have been edited.
- **Staged.** Once you stage the modified files by adding them, the files are considered as staged and commits the staged changes.

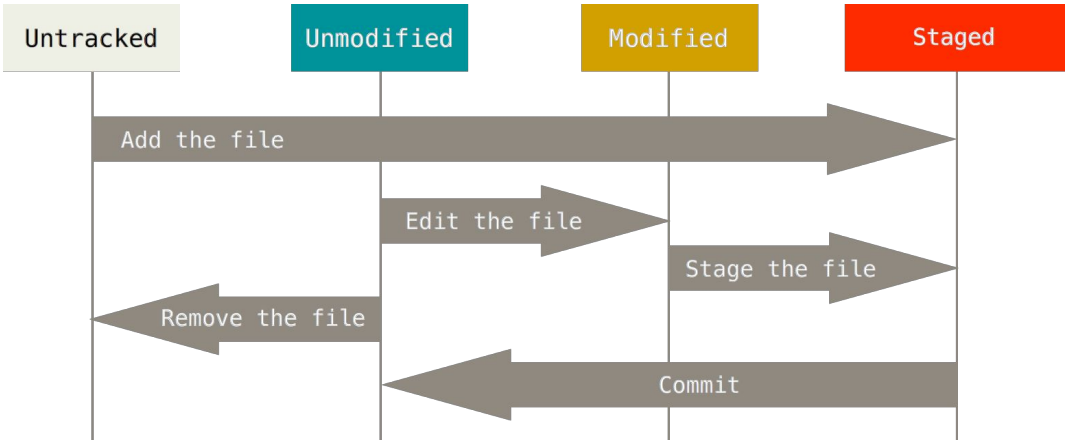

Figure 4: File state life cycle. Retrieved from git-scm.com.

Setting Up Your Computer

Operating System - Version Verification

1. Check your version by clicking on the Apple logo in the upper left corner of your screen.
2. In the drop down menu, click “About This Mac”. Refer to **Figure 5**.

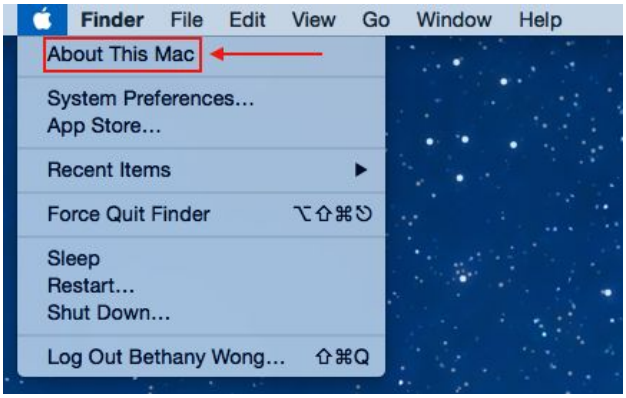

Figure 5: Follow step 1 and 2 to get to your system’s information

3. A window should appear with your system’s information. As identified in **Figure 6**, your Mac OS X version is displayed at the top of the window.

Figure 6: Your system’s information and specs are located here. You can see which version your system is running.

4. Please take note of your version number as it will be used further down.

Terminal

You will be using terminal to enter your commands and interact with files.

1. Search in Spotlight for the application called terminal. You can do so by pressing the following keys: ⌘ + `Space` + "terminal" + `Enter`
2. Otherwise, you can go into your applications to open terminal.

All commands you type into your terminal follow the following format: `$ command`. Please type everything after the `$`.

Ruby

To install Homebrew you must have Ruby installed.

1. Check if you already have Ruby installed by entering the following command.

```
$ ruby --version
```

2. If `command not found` prints, then install Ruby by entering the following command.

```
$ xcode-select --install
```

Homebrew

Homebrew allows you to install software through the terminal.

1. The following command will download a Ruby script that will automatically install Homebrew

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/ins
tall)"
```

2. Review the list of commands that brew has by entering:

```
$ brew -h
```

Note: Commands that you will be using later on are

`$ brew search <software name>` and `$ brew install <software name>`

Git

You will need to install git, if not already installed, to import git commands to your system.

1. If your version of Mac OS X is 10.9 or higher, follow step two. Otherwise, skip to step four.
2. Enter the following command.

```
$ git
```

3. If you do not have git installed already, the git OS X installer window will automatically appear. Please follow the instructions described in the git OS X installer window (**Figure 7**). Afterwards, skip to the GitHub section.

Figure 7: Git OS X installer.

4. Search for git to make sure it is available to install by entering the following command.

```
$ brew search git
```

5. Once you see that git is listed, enter the following command to install git.

```
$ brew install git
```

GitHub

Create an account for GitHub if you do not have one already.

1. Go to github.com.
2. On the right of the frontpage, enter your desired username, email, and desired password.
3. Click “Sign up for GitHub”.

Java

The project you will be going through with your team requires you to have java installed.

1. Enter the following command into your terminal.

```
$ java -version
```

2. If it tells you `java: command not found`, then install java with the following command:

```
$ brew update && cask install java
```

Text Editor

If you have a text editor that you are comfortable with feel free to skip this step. Otherwise, we suggest using Sublime.

Note: For CMSC216, you will need to have a nice text editor to work with for your projects. It may be nice to set this up now.

1. Enter the following command into your terminal. This should show you that Caskroom/cask/sublime-text is available to install.

```
$ brew search sublime-text
```

2. Install sublime by entering the following command:

```
$ brew install sublime-text
```

Guided Project

Welcome to the Real World

You and your group are now going to be guided through a real life simulation where you project to learn from experience how git works then read about it. First, the three of you will need to decide a group leader, this leader will need to follow the instructions specifically set out for them before the group can continue with the project.

Now Hiring

At the Census Bureau, we are currently working on a project that simulates population growth amongst a ethnically diverse community. We originally hired a group of software developers to carry out this project, but due to their incompetence, they were unable to work as a team so we hope you all can. They have already developed a decent skeleton code uploaded on GitHub.

Starting Out

Leader needs to fork and add collaborators. Once that happens the other two need to clone the leader's fork and work on their respective parts.

In this project you are given a human class in which you must implement its functions to make the simulator work properly. Because there should be three team members we recommend that one person does the introduce() and giveBirth() functions, another does marry() and divorce(), while the last does getJob() and leaveJob().

Human Class

This is the class in which you will be implementing functions. Before you start, these are the instance variables that you will be using throughout your implementations:

`public String name`
The name of the current Human instance, this can be a first name or a first and last name.

`private int age`
The current age of the current Human instance, babies are born at age 0, otherwise Humans may be created at any age.

`public int health`
A value that increases and decreases as the Human ages, if this value is less than or equal to 0, the Human dies

`public String ethnicity`
The ethnicity of the current Human instance, babies will be either the mother's or the father's i.e. no mixed ethnicity

`public String gender`
The gender of the current Human instance, baby's gender are randomly decided between male/female, otherwise the human can be created with any gender.

`public Human spouse`
The spouse of the current Human instance, this project does not judge and so allows same sex marriage.

`public LinkedList<Human> children`
A list of the children that belong to the current Human instance.

`public LinkedList<Human> friends`
A list of the friends that the current Human instance has.

`public boolean employed`
The status of employment for the current Human instance, employment is normally determined by if salary is greater than 0.

`public int salary`
The amount of money the current Human instance is making per year.

Human Functions

You must implement the functions described below. Notice that many functions are already given to you acl only need to change what is described below. Functions may be done in any order as they do not depend on one another.

1. `public String introduce()`

   This function will return the *String* "Hi, my name is *$NAME* and I'm *$AGE* years old" where *$NAME* and *$AGE* represents the name and age of the *Human* calling *introduce()*.

2. `public String marry(Human toWed)`

   This function will set the spouse field of both the *Human* calling *marry()* and the *Human, toWed,* to equal each other if both do not yet have a spouse and both *Humans'* age is greater than *18*.

   This function will return one of four *Strings*:
   - "*$NAME* is too young to be married" if either the *Human* calling *marry()* or the *Human toWed's* age is less than *18* where *$NAME* will take priority on the *Human* calling *marry()*.
   - "*$NAME* is trying to marry (himself/herself)!! Unfortunately, (he/she) is still single" where *$NAME* is the *Human* calling *marry()* while (himself/herself) and (he/she) is chosen based on the gender of the *Human* calling *marry()*.
   - "*$NAME* is already married" where *$NAME* will be the name of the *Human* that is already married where *$NAME* will take priority on the *Human* calling *marry()*.
   - "*$NAME1* and *$NAME2* are now married" if the marriage was successful where *$NAME1* is the name of the *Human* calling *marry()* and *$NAME2* is the name of the *Human* being passed in.

3. `public String divorce()`

   This function will set the *spouse* field of both the *Human* calling *divorce()* and his/her spouse to *null* if they are already married.

   This function will return one of two *Strings*
   - "*$NAME1* and *$NAME2* divorced" if the divorce was successful where *$NAME1* is the name of the *Human* calling *divorce()* and *$NAME2* is the name of their spouse.
   - "*$NAME* isn't even married" if the divorce is not successful where *$NAME* will be the name of the *Human* calling *divorce()*.

4. `public String giveBirth(Human baby, ArrayList<Human> kids)`

   This function will welcome a new *Human* into the world; this can only happen when the *Human* calling *giveBirth()* is a female that is married to a male. The new *baby* will need to be added to the the mother and father's *children* list.

   This function will return one of three *Strings*:
   - "*$NAME* has to be a female to give birth" if the *Human* giving birth is a male where *$NAME* will represent the *Human* calling *giveBirth()*.
   - "*$NAME* can only give birth if her spouse is a male" if the spouse of the *Human* giving birth is a female where *$NAME* will represent the *Human* calling *giveBirth()*.
   - "*$NAME1* and *$NAME2* gave birth to a baby named *$NAME3*" if *giveBirth()* is successful where *$NAME1* is the name of the *Human* calling *giveBirth()*, *$NAME2* is the name of the spouse, and *$NAME3* is the name of the *baby*.

5. `public String getJob(int money)`

   This function will set the *employed* and *salary* fields of the *Human* calling *getJob()* to *true* and *money* respectively if that *Human* does not already have a job.

   This function will return one of two *Strings*:
   - "*$NAME* found a job that is paying (him/her) *$SALARY* per year" if `getJob()` is successful where *$NAME* is the name of the *Human* calling *getJob()* and *$SALARY* is the amount of *money* that the job is paying; the *String* being returned should also use only either(him/her) depending on the *Human's* gender.
   - "*$NAME* already has a job" is returned if the *Human* calling *getJob()* is already employed where *$NAME* is the name of the *Human* calling *getJob()*.

6. `public void leaveJob()`

   This function will set the *employed* and *salary* fields of the *Human* calling `leaveJob()` to *false* and *0* respectively if that *Human* already has a job to leave.

   The function will return one of two *Strings*:
   - "*$NAME* has left (his/her) job" if the *Human* is employed where *$NAME* is the name of the *Human* calling *leaveJob()*.
   - "*$NAME* does not have a job (he/she) can leave" if `leaveJob()` is unsuccessful where *$NAME* is the name of the *Human* calling *leaveJob()*. In both cases (his/her) and (he/she) should be chosen depending on the gender of the *Human*.

What's next?

Open your resume right now and add "git" to your list of software/platforms that you are equipt to use!

talk about how to get involved in projects on github. argue that a summer spent contributing to the open source world will be 10x more beneficial than any summer or part time job you can find. Maybe suggest going to monthly meetups to get inspired to stay on track with your goals for contributing to github repos.

Glossary

Terms

Branch

A branch is a version of the repository that exists in parallel with the master branch. Working on a branch does not affect or change the master branch until you need to merge it.

Clone

A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere, or the act of making that copy. Because this is a local copy, you will need to sync with the online copy through the use of push and pull.

Collaborator

Someone who has access to make changes to the main development repository of a project.

Commit

A commit is an individual change to a file (or set of files). Commits will have a unique ID that also store the user who made a commit as well as the changes that were made.

Contributor

Someone attempting to add to a project but does not have permission to do so and has to request permission to make changes from a collaborator.

Fork

A fork is a personal copy of another user's repository that lives on your account. This allows you to be the collaborator of your own repository instead of a contributor of the repository you forked off of.

Git

An open source program for tracking changes in text files.

GitHub

A social and user interface that is built on top of git

Master

The main branch of a repository, eventually all branches will merge back into this one.

Merge

Taking the changes from one branch and applying them to another.

Repository

A project folder that contains all project versions and branches. This is what collaborators and contributors work on

Working Tree

A branch that users are currently working on and therefore have branched off of in order to make changes to and later merge back

Commands

git add $FILENAME1 $FILENAME2 $FILENAME3 ...

Add all files listed to the staging area to be committed.

git branch $BRANCHNAME

Creates a copy of the current branch that you are working in and names it $BRANCHNAME

git checkout

Switches to another branch, normally used to work on different features of a project.

git commit -m $COMMENT

Records the changes of files you've use git add on into your repository, this will not update remote repositories

git diff

Brief description including any flags they need to know about

git merge $BRANCHNAME

Combine the current branch with $BRANCHNAME so that changes made to the current branch are reflected in $BRANCHNAME

git push

Sends the changes that you've made to your current repository to the remote repository that you've pulled from

git status

Show files that have been added to the current index as well as those that are different from the most recent commit on the current branch.

References

"Git." Git. N.p., n.d. Web. 01 Nov. 2015.