**Reflect: Using diff to Find Bugs**

**Make sure you can access the command line**

If you are on Windows, you’ll need to install some software in order to use a unix-like command line (different from the Windows built-in Command Prompt), as we mentioned previously. For more info, **[check out this page](https://www.udacity.com/wiki/ud775/command-line-instructions" \l "windows-users" \t "_blank)**.

**Choose a text editor**

As noted in the previous video, you should make sure to use a simple text editor like **[Notepad++](http://notepad-plus-plus.org/" \t "_blank)**, **[Sublime](http://www.sublimetext.com/3" \t "_blank)**,**[Atom](https://atom.io/" \t "_blank)**, emacs, vim, etc. and **not** a rich-text editor like Microsoft Word or OpenOffice, so that you can easily look at your files' content on the command line.

If you don’t have one that you like yet, Sublime is a good option that will work on Windows, Mac, and Linux. We have provided more detailed instructions for setting up Sublime than for other editors, and we use Sublime for all the examples in the course. You can download Sublime **[here](http://www.sublimetext.com/3" \t "_blank)**.

**Make sure you can launch your editor from the command line**

It will be helpful to be able to launch your text editor from the command line. See **[here](https://www.udacity.com/wiki/ud775/sublime" \t "_blank)** for instructions on how to do this for Sublime. If you have trouble getting this working, videos at the end of the lesson called "Setting up Your Workspace on Windows" and "Setting Up Your Workspace on Mac" will demonstrate this process, so you can wait until then.

**Set up your course workspace**

Right now, you should create a version-control directory (a more computer-science-y term for “folder”) to hold all your files for this course in an easy to remember location, then create a reflections sub-directory, and within that, create a plain text file called lesson\_1\_reflections.txt for the questions from this lesson.

You can do this by running the following commands in either Git Bash or the terminal (the bits after the # signs are comments, anything after those are not interpreted):

cd ~ # change directories to your home directory mkdir version-control # make version-control directory cd version-control # go to version-control directory mkdir reflections # create reflections directory cd reflections # go to reflections directory subl lesson\_1\_reflections.txt # launch sublime with file called lesson\_1\_reflections.txt (you can replace subl with another editor here if you prefer a different one)

If you prefer, rather than creating the file from scratch, you could download the lesson\_1\_reflections\_prompts.txt file from the Downloadables section, place it in the reflectionsdirectory and rename it, then add your response after the first prompt.

Once you’ve saved your file, if you want to double-check that everything has gone as planned, try these commands:

pwd # print working directory - shows what directory you are in ls # list the files in this directory

If you're having trouble getting this working, videos at the end of the lesson called "Setting up Your Workspace on Windows" and "Setting Up Your Workspace on Mac" will demonstrate parts of this process, so you might want to wait until then. In the mean time, you can create the directories and file using your OS’s Graphical Interface for working with files (Finder, Windows Explorer, etc). It's good to get some practice using the command line, though, since we'll be using it a lot in this course, so once you've watched the workspace instructions, make sure you can complete actions like this on the command line.

You are also welcome to use a different naming scheme, but later in the course we will refer to this file structure and it will be up to you to translate to whatever naming scheme you chose instead!

**Use short lines**

Many command line tools, including Git, are less useful if your files contain very long lines. For example, if you use diff to compare two files that have all their content on the same line, diff will only show you that the two files are different. It will not be able to pinpoint the location of the difference for you.

For this reason, it is a good idea to make sure you keep your lines reasonably short when writing your reflections (or other plain-text files). The exact limit is a matter of personal preference. Many developers use a max line length of 80 to 120 characters. Some editors can automatically insert line breaks for you, but for others, like Sublime, you will need to remember to press enter when you want to create a new line.

**Do the first reflection exercise**

Populate lesson\_1\_reflections.txt with the following question and your thoughts on it:

**How did viewing a diff between two versions of a file help you see the bug that was introduced?**

When you've created your document, written down your thoughts, and saved the file, click "Next" to learn about some systems that can help you create these versions of your files.

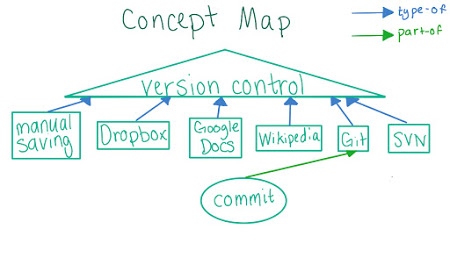
Like Sarah said, there is a lot of research out there on reflection. If you want a quick intro with some neat graphics, check out **[this site](https://sites.google.com/site/reflection4learning/why-reflect" \t "_blank)**.

For a more in-depth look, check out **[this seminal 20-page paper](http://184.182.233.150/rid=1LW06D9V6-26428MK-1Z64/Mezirow's%20chapter,%20How%20Critical%20Refletion%20Triggers%20TL.pdf" \t "_blank)** on the topic.

## Concept Map: diff

Now that you know about git diff, you can add it to the concept map.

### Previous version

First, let’s look at the original map:  


### New addition

Remember that we are trying to avoid cluttering up our map, so try to only add relationships that would not be implied by other relationships already on the map.

If you were to put "git diff" on the map, which of the following existing concept(s) would you directly connect it with?

* commit
* Google Docs
* git
* Version Control

Continue to the quiz to make your selection(s).

## How Often to Commit

Since you can choose when to make a commit, you might be wondering how often to commit your changes. It's usually a good idea to keep commits small. As the diff between two versions gets bigger, it gets harder to understand and less useful. However, you don’t want to make your commits too small either. If you always save a commit every time you change a line of code, your history will be harder to read since it will have a huge number of commits over a short time period.

A good rule of thumb is to make one commit per logical change. For example, if you fixed a typo, then fixed a bug in a separate part of the file, you should use one commit for each change since they are logically separate. If you do this, each commit will have one purpose that can be easily understood. Git allows you to write a short message explaining what was changed in each commit, and that message will be more useful if each commit has a single logical change.

## Commit Size Quiz

To get some practice thinking about how often to commit, on the next screen, mark whether you think the following would be good commit sizes. If not, indicate whether you think this commit is too small and you’d like to wait and commit later, or whether you think it’s too big and you would have committed earlier. This is subjective, so there aren’t any definite right or wrong answers, but just choose the answer you think is best in each case.

* You commit all the changes required to add a new feature, which you’ve been working on for a week. You haven’t committed since you started working on it.
* You find three typos in your README. You fix and commit the first.
* You commit all the changes required to add a new feature, which you’ve been working on for an hour.
* You fix two small bugs in different functions and commit them both at once.

### What is a README?

Many projects contain a file named "README" that gives a general description of what the project does and how to use it. It's often a good idea to read this file before doing anything with the project, so the file is given this name to make users more likely to read it.

|  |  |
| --- | --- |
| GIT Befehl | Beschreibung |
| git clone „path“ | Klont ein Repository |
| git log |  |
| q | Quit git log |
| cd „reponame“ | Wechsel in ein repository (z.B. nach dem clone) |
| git diff „commitid“ „commitid“ | Zeigt Unterschiede zwischen 2 Commits |
| git config --global color.ui auto | Schaltet farbliche markierung in commad git ein |
| Git checkout | Checkout eines bestimmten Commits (überschreibt lokale Dateien mit entsprechendem Inhalt aus Commit 🡪 anders als SVN) |
| Git init | Erstellen eines neuen Repos |
| Git status | Gibt den Status eines Repos zurück |
|  |  |
|  |  |
|  |  |

## Git Errors and Warnings Solution

**Should not be doing an octopus**  
Octopus is a strategy Git uses to combine many different versions of code together. This message can appear if you try to use this strategy in an inappropriate situation.

**You are in 'detached HEAD' state**  
HEAD is what Git calls the commit you are currently on. You can “detach” the HEAD by switching to a previous commit, which we’ll see in the next video. Despite what it sounds like, it’s actually not a bad thing to detach the HEAD. Git just warns you so that you’ll realize you’re doing it.

**Panic! (the 'impossible' happened)**  
This is a real error message, but it’s not output by Git. Instead it’s output by GHC, the compiler for a programming language called Haskell. It’s reserved for particularly surprising errors!

**Takeaway** We hope these errors and warnings amused you as much as they amused us! Now that you know what kind of errors Git can throw, you’re ready to start checking out previous versions of files with Caroline.