# What is Git?

BY CONCORD SPARK TUTORING

# Git is a Version Control System

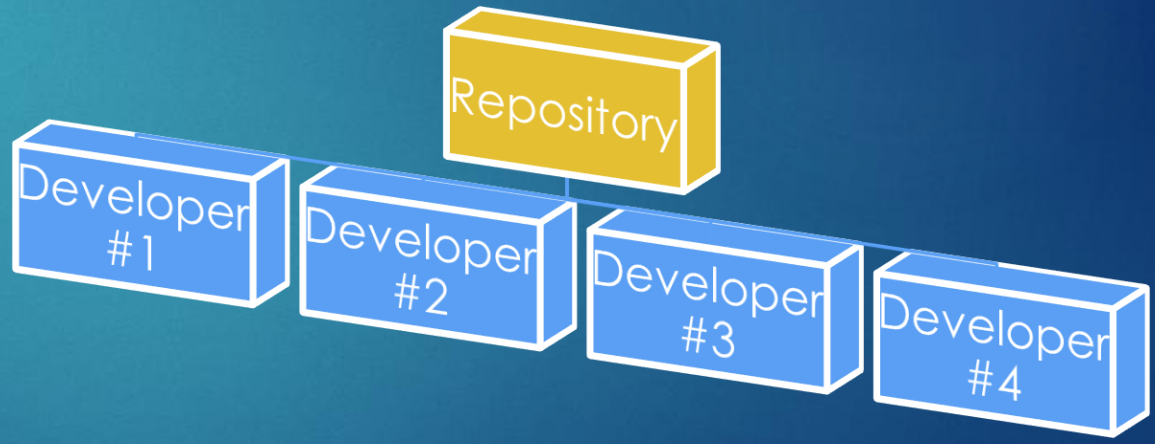
- A version control system allows software developers to track revisions to projects such as software applications
- More specifically, Git is a Distributed Version Control System
- What this means is that the complete codebase and version history is mirrored on every developer's computer rather than being located solely on a single backup location
- Any revisions on one developer's computer is synchronized with all the other developers'
- The full codebase and version history are not limited to residing only in a single, centralized location

'Codebase' refers to the complete collection of source code which makes up the software application project.

# Why not use a Centralized Version Control System?

- The repository is accessed over the internet, so performing actions takes time
- In a centralized system, developers have to "pull" the latest version off the repository which would require an internet connection
- Each change someone saves is automatically available to the whole group - this may be problematic if you're not ready to share the change yet OR if it's bugged in which case, everyone else will get affected
- If the central server crashes, no one can access the repository

In a Centralized Version Control System, the repository is located in one, central, remote location. All developers "pull" the files from this repository and "push" their changes back to it.



# Advantages of a Distributed Version Control System

- Every developer's computer is itself a repository containing the project's full codebase and version history
- This means each developer has access to everything and can make changes without having the need to be connected to the network
- This also means that access over a network is required only when "pushing" or "pulling" changes
- A crash or hardware malfunction on one repository will not affect the other repositories on other developers' computers

# Make no mistake though…

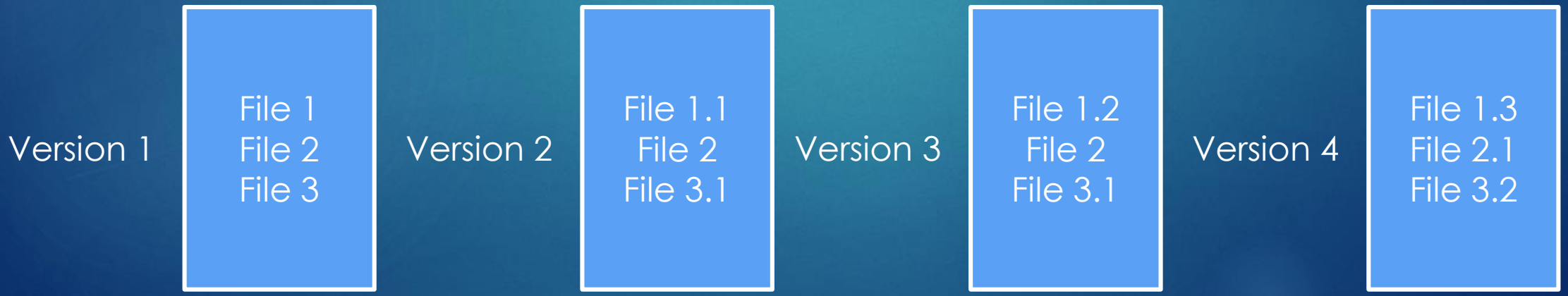
- A Distributed Version Control System can still have a central repository from which each developer can push and pull their changes to/from
- However, even with a central repository, each developer's computer still has a clone of the full repository on their own unit(s) in a distributed system

# Back to Git

- Git is an actively maintained open-source project
- Git is written in C
- Some of the features of Git are its efficiency in terms of its performance and compression of data
- The only time Git will be considerably slower than other VCSs (Version Control Systems) is when its cloning the entire codebase and version history for the first time

# How does Git view data?

- Git takes 'snapshots' of your data
- Every time you commit (save) your data, Git takes a snapshot of what all your files looks like at that point. Git then stores a reference to that snapshot
- Of course, if certain files haven't changed, then Git doesn't take a snapshot of those files. It simply puts a link to those files from the previous version



# Three States

- Git has three main states that your files can have separately over the course of your project development:
  - Modified: A file is modified when you've made changes to it, but have not yet committed anything
  - Staged: A modified file marked as staged means that it will get committed to the database on your next commit snapshot
  - Committed: This means that the data is saved to the database. You're safe.

# Three Main Sections of Git Projects

- The Git directory:
  - This is where all the metadata and the object database for your project is stored.
  - The Git directory is what gets stored onto your computer when you clone a repository from another computer
- Staging Area (The Index):
  - This is simply a file that stores information on what gets saved on the next commit operation.
  - The actual Git term for this is the 'index' but most people refer to it as the 'Staging Area'
  - This index file is located in the Git directory on your computer
- Working Tree (Working Directory)
  - This is a directory containing a single copy of one version of the project for you to modify
  - The files for this directory are pulled from the compressed database in the Git directory and placed on your disk to work on

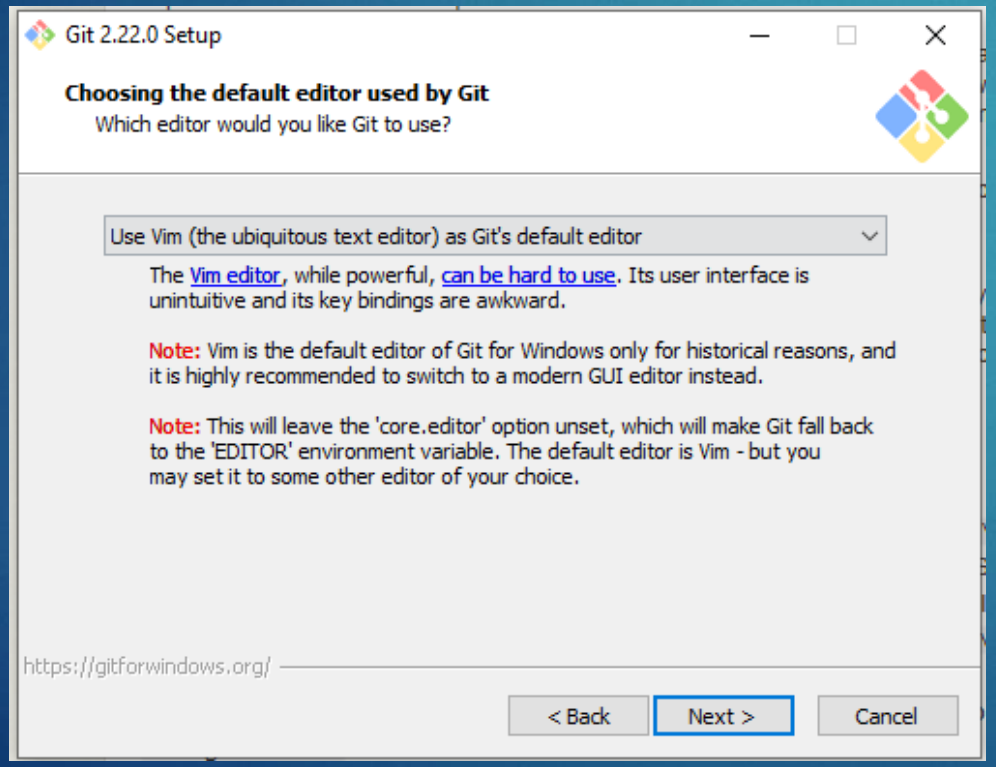
# Installing Git

- You will need to make sure that Git is installed on your system
- This link provides the download for Linux, Mac and Windows
  - https://git-scm.com/downloads
- Git is a command-line program. This means you have to type text-based commands in a command-prompt window
- There are GUIs which allow you to interface with Git via graphical interface, but this often means losing some ability with what you can do with Git as opposed to using the command-line version
- You can also download GitHub Desktop which provides a GUI layer that you can use instead of the command-line
  - https://desktop.github.com/

Go ahead and download GitHub Desktop. However, in these slides, we'll simply deal with the command-line version of Git.

# Installing Git

- Download and run the install file for Git from the link provided on the previous slide
- At one stage during the install process, you will see this:




From the Drop-down box, select 'Visual Studio Code' as the default editor.

As stated by the installer, Vim is hard to use and not recommended.

# Starting up Git on Windows


- This slide assumes you installed Git on Windows in which case you will see three installed items – screenshot on the right.
  - Git Bash opens a command-line window that imitates a bash environment on a Unix system.* This shell allows you to use standard Unix commands as well as all git commands.
  - Git GUI provides a graphical user interface so that you don't need to type commands to use git
  - Git CMD will open up a command-based (Windows Style) prompt where you can use all features of Git. Useful if you're used to the windows cmd prompt.
- For our slides, we'll use Git Bash. Open this up.

Bash is a command-line language on Unix systems (like DOS for Windows)

# Configuring Git

- The first thing to do is to set your username and password, e.g.:
  - git config --global user.name "John Doe"
  - git config --global user.email johndoe@example.com
- The --global command specifies that this information will be applied to anything we do with Git on this system
- After entering those commands in, you can check if it worked by typing:
  - git config --list --show
- Give it a few moments and then you'll see all your settings listed in the window. At the very end – again, you may have to give it a few seconds – you should see your username and email settings.

# Configuring Git Cont.

- During installation, Git asked you to specify the default text editor to use – we're using Visual Studio Code (a lightweight code editor with intellisense)
- However, you can also use the following command to set your default text editor (for example, back to VIM):
  - git config --global core.editor "vim"
- Anytime you need help, you can use the following command to open up the Git help page
  - git --help push
  - The command above looks up the help page for the "push" command and opens it in your browser

In case you're confused on the number of dashes needed before "help" or "global" in the listed git commands, it's two dashes: --

# Creating a Local Git Repository

- The following instructions help you set up a local Git repository on your Windows computer
- First, navigate to the directory of where your project is located. For example, the following command navigates to a directory called 'Concord Spark Tutoring' on the user's desktop:
  - cd /c/users/johndoe/desktop/concord\ spark\ tutoring/
- Note that when changing directories, the folder names aren't case-sensitive.
- Also note that a directory with spaces necessitates the need for a backslash before each space: concord\ spark\ tutoring. This identifies the spaces and subsequent words as part of the directory name
- Once you are in the folder where your project is, type:
  - git init
- If all goes well, you should see a message like the one shown on the next slide

# Creating a Local Git Repository Cont.

```
$ git init
Initialized empty Git repository in C:/Users/josep/Desktop/concord media and design/spark bi
lls/.git/
```

- If you see the message above, then congratulations! You have just created your first local Git repository. This is the Git directory that we talked about on a previous slide.
- The Git directory is a hidden folder, so you won't see it in File Explorer unless you enable the viewing of hidden files and folders.
- Now that you have your repository all set up, it's time to start adding your project files to it so that they can get tracked

# Adding Files and Folders

- To add a file to the Staging Area before committing, simply type:
  - git add README
- To add files of a specific type to the index, type:
  - git add *.cpp
  - Here, the command adds all .cpp files (source files) to the repository
- You can also specify the name of a folder by typing the folder name (remember: spaces must be preceded by a backslash) followed by a forward slash and *:
  - git add my\ project\ folder/*
- Adding these files simply stages them. You must commit them with the following command:
  - git commit –m "initial project version"
  - The –m "initial project version" portion specifies a log message associated with this commit transaction. Normally, you would use this message to describe the changes you're committing to the repository.

Remember: Staging Area and Index mean the same thing: a file that keeps track of files to get saved to the repository.

# Adding Files and Folders Cont.

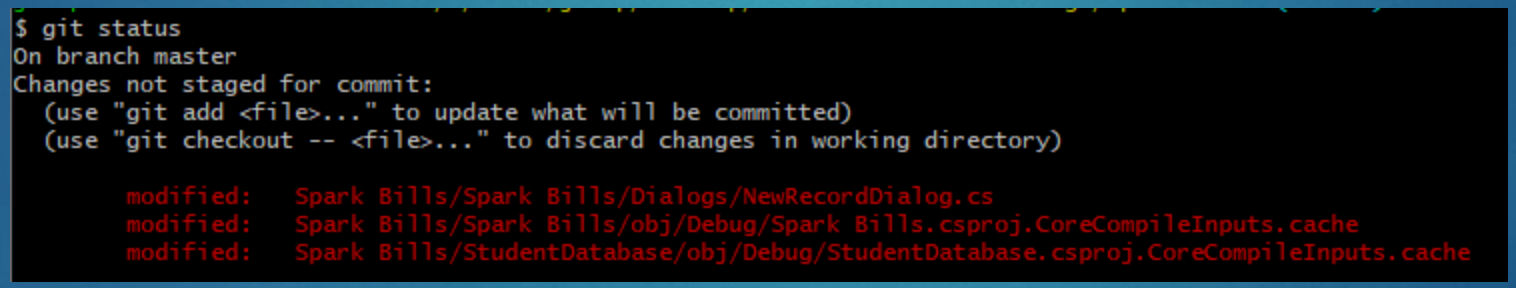
- Once you commit your project files to the Git directory, your Bash terminal should display a listing of all the files pushed to the Git directory
- You can also check the status of your git repository by typing:
  - git status
- If you haven't modified any files yet, you will see:

```
$ git status
On branch master
nothing to commit, working tree clean
```

- If for example, you added a new file to your project, then the status operation will show you the new files listed as 'Untracked'
- It is up to you on whether you want to add these files to the repository or leave them untracked.

# Tracking Your Files

- Now that you've saved your project files to the repository, if you modify them, you can check the status in git using
  - For example, in my project, I modified 'NewRecordDialog.cs'. I then type the below command:
  - git status
  - The following screenshot shows the list of files that have been modified




- Note the message that says 'Changes not staged for commit'
- This simply means that a tracked file has been modified but not yet staged for the next commit operation

# Tracking Your Files Cont.

- Now, let's stage the modified files using the add command
- On our next git status report, we will see the following:

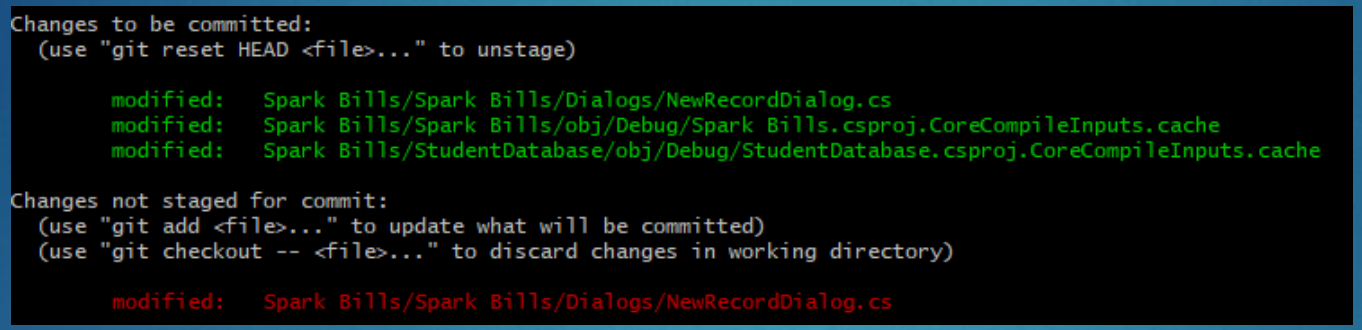
```
josep@DESKTOP-POVCB6R MINGW64 /c/users/josep/Desktop/concord media and design/spark bills (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Spark Bills/Spark Bills/Dialogs/NewRecordDialog.cs
        modified:   Spark Bills/Spark Bills/obj/Debug/Spark Bills.csproj.CoreCompileInputs.cache
        modified:   Spark Bills/StudentDatabase/obj/Debug/StudentDatabase.csproj.CoreCompileInputs.cache
```

- As you can see from the status report, our three files which we've added to the staging area are now ready to go on our next commi operation
- BUT what happens if you choose to modify a file that you've already staged?

Unlike a change directory command, adding files and folders is case-sensitive!!!

# Tracking Your Files Cont.

- Suppose after staging it, I modify the same file (NewRecordDialog.cs) again. After 'git status', this is what I'll see:




- In the screenshot, 'NewRecordDialog.cs' is under both 'Changes to be committed' AND 'Changes not staged for commit'
- Don't be confused. What happens is that Git will save the version of 'NewRecordDialog.cs' that was BEFORE I made the latest change. Once you stage a file, even if you modify it again afterwards, the staged file will still be as it was when you first staged it.
- To save the latest changes, you'll have to 'git add' the latest version of the file to the staging area.

# Tracking Your Files Cont.

- If you have staged some files and then modified them, you can view the exact changes you've made using the following command:
  - git diff
- This will show the difference *in content* between the unstaged, modified files in your working directory and the files that have been staged.
- For example, I added a couple of comments to 'NewRecordDialog.cs' AFTER I had already staged the file. This is what was displayed after a call to 'git diff':

```
private void buttonAdd_Click(object sender, EventArgs e)
{
    //We need to obtain data from the database first
    //This should show up in the next git diff command

}
```

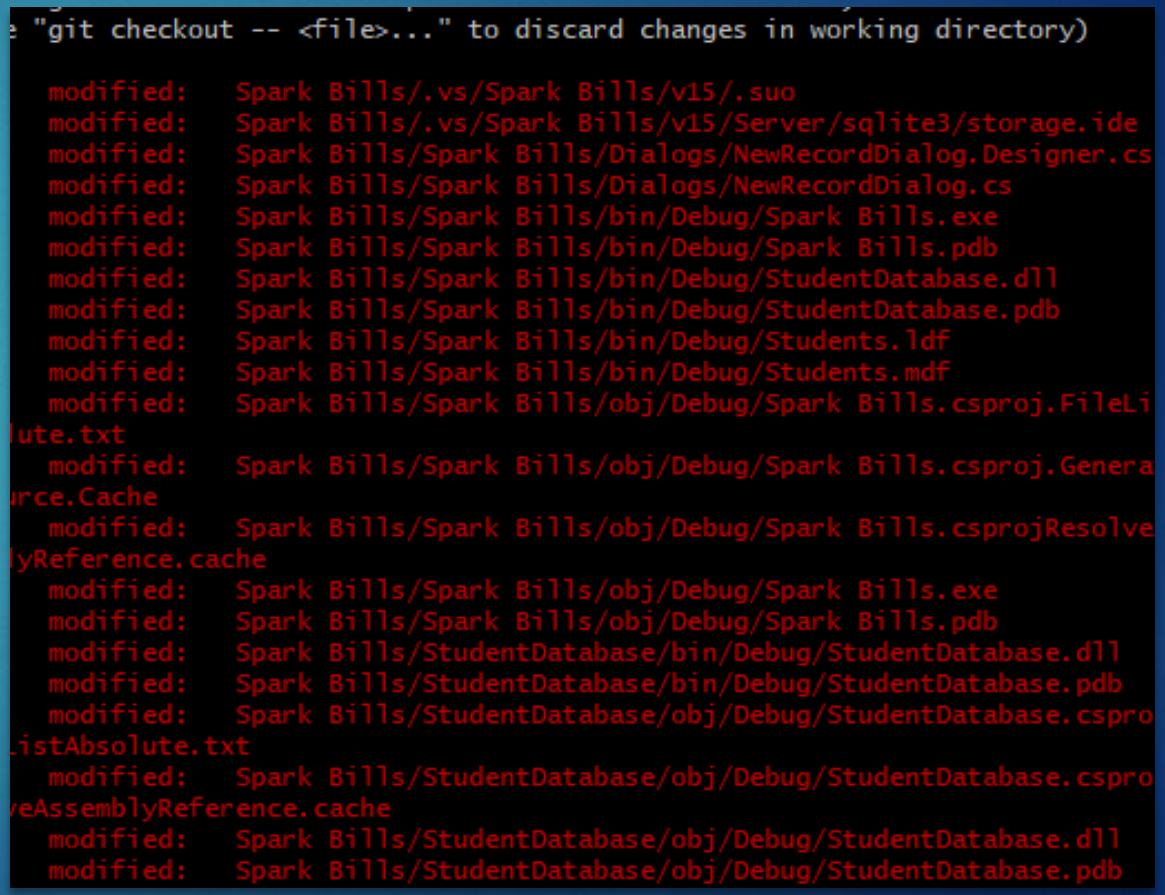
Note: 'git diff' does not show any difference between unstaged files and what's already been committed!

# Committing Files

- If you currently have any files in your staging area, go ahead and commit them. How? Well, there's two variations:
  - git commit
  - git commit –m 'a log message to identify the changes made'
- The first version simply commits the staged files to the repository
- The second specifies a log message that's used to identify this version of the files you're committing. Note that the message comes in single quotes!

# Committing Files Cont.

- Now, oftentimes making a couple of changes to your project can and will affect other files depending on the complexity of your project.
  - For example, updating a database model on my end affected the database model file, a SQL file containing the SQL statements to create the database and a host of other files.
- Running ‘git status’ showed me the report on the right:



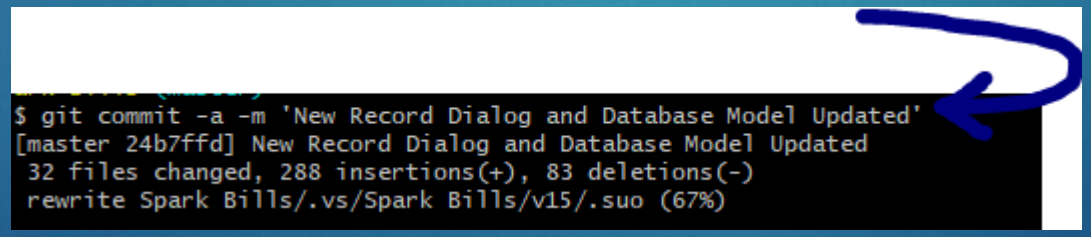
# Committing Files Cont.

- When you run into a situation like this, it'd be nice to have a command that would automatically stage all modified files and commit them in one go.
- In git, this is possible! Simply type:
  - git commit –a
- The –a option specified after the 'commit' command will instruct git to add every *tracked, modified* file to the staging area and then commit it all to the git directory.

It's very important to understand the distinction between a *tracked, modified file* and an *untracked* file. An *untracked* file won't be added to the staged area (even if you've modified it) as part of the 'git commit –a' call. This is because you – on purpose, I'm assuming – have left it out as it is unnecessary to be tracked (e.g. readme or log files).

# Committing Files Cont.

- You can also combine multiple options in a git commit statement like this:
  - git commit –a –m ‘changes to database model were made’
- The above command adds all modified files that are being tracked to the staging area and commits them with the specified log message.
- The below image shows my command:



# Removing Files

- Now, it's time to tackle the opposite of a commit: removals!
- Removals also involve the staging area, or the index file.
- We can remove a file from the git directory and our working directory (i.e. your local file system) by writing:
  - git rm samplefile.txt
  - git commit –m 'Removed samplefile.txt from working and git directory'
- Now, if you simply want to remove a file from the repository but leave it on your local file system, then add --cached:
  - git rm --cached samplefile.txt
  - git commit –m 'Removed samplefile.txt from repository only'

Try it!

Add a new file to your working directory. Next, commit it to the repo. Then, try both rm and rm --cached!

# That's it!

GO TO: WWW.CONCORDSPARK.COM
TO GET A FREE COPY OF THESE SLIDES!