CST316 Software Enterprise: Construction and Transition **Spring 2016**
PRACTICE: Source Code Control using Git

OBJECTIVES:

1. Understand SCC concepts
2. Gain proficiency with git
3. Create and account and learn to use GitHub as a remote repository

SUBMISSION:

There are 2 tasks in this lab, so I require 2 zipfile submissions (1 for each task) to Blackboard. Additionally, at the end of the lab it asks you to create a Github account and place some files there. It also instructs you to add Github user cst316 to the set of collaborators for your personal repository. Please remember to do this as we will inspect your Github as part of your grade!

GRADING CRITERIA:

- Task 1 is worth 50%
- Task 2 is worth 35%
- Task 3 (Github) is worth 15%

PREFACE:

This lab is admittedly more of a tutorial than a lab. It introduces you to Git, a (distributed) version control system, which we will use for your class projects. Along the way there will be specific tasks for you to complete and turn in, but the emphasis here is gaining initial familiarity with the tool and the concepts behind SCC in general and DVCS in particular. Much of the material for this tutorial is abridged from Scott Chacon's excellent online book *Pro Git*, and assorted other presentations and online materials. I highly recommend going back through the first few chapters of the online book (http://git-scm.com/book), especially Chapter 2. The lab covers tools, using Git locally, and using Git with remote repositories. Tools are spread throughout the activities, as I will introduce git, GitHub, and an Eclipse plugin over the course of the activities. Let's start with a discussion of what Git really is.

Overview: Understanding *Content Addressable Filesystems* (or *Managed Directories*)

So you are working on a small team writing source code for a project. In the absence of any tools, you are trying to figure out how to work on the source code tree together. Let's say each of you has an Eclipse project for your work. You have divided up the tasks and are going about your business. What happens when somebody sends an email around saying "hey, I just finished writing the main UI, here are some files that do it: Main.java, Events.java, and Handlers.java; enjoy!" Well, common sense dictates you would copy these files in to the correct place in your directory and move on with coding. But there are a few "what ifs":

1. What if you notice the code has a few minor defects?
2. What if you had already made your own edits to Events.java and Handlers.java?
3. What if copying those files into your workspace caused your project to not compile?
4. What if you copied them in, went about your business, accepted changes this way for weeks/months – but then realized you needed to go back to the version from January 6, 2014 for some reason? Perhaps you edited Main.java and made a whole bunch of changes and now wanted to back to a prior checkpoint?

Well, in the case of #1, you'd hopefully make the corrections and email the files back around. Hmm, is that sustainable?
In the case of #2, you'd probably either curse to yourself while manually trying to merge your stuff with your teammates stuff. Then you would email the result around. What happens though if another team member was doing the same thing to the same files?
#3 can be really frustrating because you have to investigate the differences in your compile environments. Are you using different compiler versions? Did the person forget another file to send around? Is there a 3rd party dependence that was not shared? Did you make a change to your stuff that introduced a dependency that was modified? Have fun figuring that all out!
And for #4 – well you could make a backup manually of each time a file has changed, and put in a directory like "archive" and manipulate filenames – so you might have "Main.Jan6.java.bak Main.Jan12.java.bak Events.Jan9.java.bak" and then use these to restore. But then you have to make sure you always make such backups, and that you keep some metadata with it – why was the backup made, and what other files does it depend on?

Sure, you can look at using shared file systems, like Dropbox, or a hard drive at a team member's house exposed via a hole in their home firewall. But would these problems really change? Nah. What you really need is a *managed directory* tool – something that will track and merge changes. This is what Git does (and note this is not a coding specific problem, and Git is not a coding specific tool!).

So what does Git do? Git creates local compressed copies of files/directories you have placed under its management. The directory you are accustomed to working in is still where you do your work. But now Git creates a "hidden" directory (by default it is just .git) and stores potential changes and accepted changes. As you go about your work Git is there to make sure you can go back to previously checkpointed versions, and can decide when you want to checkpoint new versions.

The #1 thing to always remember about Git is that it is not a client/server tool. It is a local tool managing a local directory. It does not require a remote server at all. Of course, the collaboration scenarios are more interesting when it is used with a remote repository – but keep in mind that this is more a peer-to-peer (P2P) arrangement (notice I said remote repository, not remote server). Essentially the collaboration scenario is one of "how do I synch from my local repository with someone else repository out there, when I want and under the conditions I want and with only the stuff I want?".

How is this different that traditional SCC? Traditional SCC tools have some similarities but some important differences. Like Git, traditional SCC tools like CVS and SVN will create local repositories in hidden directories. But the difference is what is in them. They store changes you have made to your workspace, not the entire set of files and all their changes. So they take up a lot less space, and can be faster when working with the server. What is better about Git then?

- You can work locally most of the time. Your repository is local so you do not need to be connected. You can work on an airplane, while camping in Yosemite, or when your home network goes down.
- Most of your work is faster because it is local.
- You do not have to checkin/checkout as often (in fact ci/co don't really make sense, which is why git uses its own verbs like "pull" "push" and "fetch" as we will see) and merges are easier and more self-selecting.
- You have greater flexibility in how you choose to work with others, both in ways you share your stuff or use their stuff.

OK, let's get to some mechanics so we can see how this works.

TASK 1: Installing git and working with local repositories

Install git

If you did not install git before lab, please do so now. You may install git from http://git-scm.org. While git has a set of common commands, there are differences in the tools you can grab based on different platforms. I would like you to start with a command-line program so you get familiar with issuing explicit commands, then you can use a custom tool (we'll use an Eclipse plugin at the end). You may also want to get a GUI-based tool for later, but for now we will work on the command-line.

Manage a directory and the files in it

On Blackboard is a zipfile of a source code tree to use with this assignment. Unzip the content in an empty directory named "work". You will see it is a simple source tree with a few other things thrown in. The first thing we have to do is ask git to create a local repo for us. At the command line, in the directory in which you unzipped the given file, issue:

```
$ git init
```

git should come back with a message:

```
Initialized empty Git repository in  <directory>/.git/
```

Go ahead and check out the contents of this hidden directory (in DOS "ls –la" = "dir"):

```
$ ls –la .git
```

Pretty unexciting eh? We have a repo with nothing in it. So let's add stuff to it:

```
$ git add *.java
$ git add <the rest of the stuff in that directory>
```

Git won't tell you anything when it does this. So, do we have stuff in the repo? Let's ask git:

```
$ git status
```

A whole bunch of stuff comes back with the header:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
```

and a bunch of "new file: <your file>" lines. What does it mean? Well, git is aware you are adding new files, but says they are not *committed*. In git terms these files are *staged* (note the last line about how to *unstage* a file). So let's commit.

```
$ git commit –m "Initial commit of my stuff" .
```

You should get a message back saying (the hex identifier and some of the numbers may be slightly different, that is OK)

```
[master (root-commit) 9c8baf6] Initial commit of test source
 31 files changed, 4782 insertions(+), 0 deletions(-)
 create mode …
```

Followed by a bunch of the create mode lines, one per each file committed. Now do a git status again, what do you see?

Manage change

The next logical thing to do is to make a change to a file. Open ButtonFrame.java in any text editor (don't go Eclipse yet) and add a few lines and delete a few lines (comment lines are fine). After doing so, let's ask git if it knows what is going on:

```
$ git status
```

You should see:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#     modified:   <whatever file you edited here>
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Note it says "Changes not staged" – it knows you made changes but is telling you that you haven't indicated you want those changes to be managed. It suggests either doing a "git add <file>" to *stage* or a "git checkout <file>" to discard what you have done. Why?

- git add can be confusing because you intuitively think "hey I already added that file to the repo, why do I have to do it again?" Well the answer is you need to add a new copy of the file to the repo, a process called *staging*; since you made changes to it.
- Git checkout is really shorthand for saying "I want my stored copy back"; you are basically saying "I no longer want the edits I was just doing".
- So what you are doing with both options is really just file manipulation between the copy in your working directory and the compressed, binary version in the .git directory somewhere.

Go ahead and add the file and run git status again.

```
$ git add ButtonFrame.java
$ git status
```

Note even after *staging* you can still decide to go back:

```
#   (use "git reset HEAD <file>..." to unstage)
```

Go ahead and try it:

```
$ git reset HEAD ButtonFrame.java
```

You will see it goes back to being changed but not staged. How do we get rid of our changes and go back to the unmodified file?

```
$ git checkout ButtonFrame.java
```

Why does this work? Because checkout basically says "copy it from the git repo into the working directory".

If we can add and modify files, and also undo changes, what about removing? Simple enough:

```
$ git rm ButtonFrame.java
```

This will remove it from the repo and adds the convenience of removing it from the working directory. What is nice is that since git stores objects and their entire histories, you can easily restore by using your reset and checkout commands. For convenience sake you can also move files around (which is also how you rename in Unix) using `git mv <from> <to>` which is really just a convenience for moving it the standard way (Unix `mv`) and then doing a `git rm` followed by a `git add`. Let's go ahead and restore the removed file:

```
$ git reset HEAD ButtonFrame.java
$ git checkout ButtonFrame.java
```

You may want to periodically check the history and other metadata about your files. Use the git show command for this:

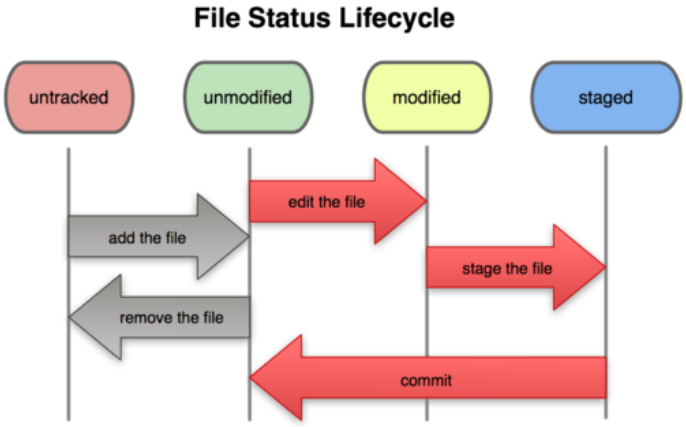
```
$ git show ButtonFrame.java
```

However git show will only show you the transactions on the repo. What about the scenario where you are editing a file but haven't committed yet, and want to check on how it compares to the copy in the repo? Git has its own flavor of the Unix diff command:

```
$ git diff ButtonFrame.java
```

Summary (so far):
We've talked about the following git commands:
init – Initialize a repository
add – add (stage) a file to the repository
checkout – copy file contents from the repo to the working directory
commit – Put a staged file into the repo permanently
diff – compare the content of the file in the working directory to the corresponding object in the repo
reset – unstage files
rm – remove files from the repo/working directory
mv – move/rename files
show – Show metadata, including history, of an object in the repo
status – Summarize the state of the working directory compared to the latest state of the repo for these objects.

A great way to think about the *managed directory* concept is in terms of state machines (UML statecharts) from last semester. A file in a working directory has several states which change based upon the commands above (from ProGit section 2.2):



To see if you really understand what is going on, edit ButtonFrame.java again, and stage it to the repo (remember how?). But don't commit. Instead, after staging, edit the file again and add a new comment line. What state do you think the file is in? Use git status to find out. Can you explain what you see?

Git Branching
Branching and merging are critical features for any SCC system. How the system handles it, what assumptions it makes, and your understanding of the model make a big difference. Do it poorly, and everything gets screwed up and your whole team is mad at you. Do it well, and you can be incredibly productive. The way Git supports branching and merging is perhaps its signature feature.

Since Git stores entire copies of your files as objects together with some metadata, it is able to handle branching and merging in a simple and elegant way. The 1st thing to understand is that Git stores a metadata object called a *commit object* each time someone commits to a repo. As more commits are done over time, these commit objects are chained together in a linked-list like structure (actually it is more a binary tree, with one child being a pointer to the previous commit and the other child being a pointer to the snapshot in time of the repo). A branch in git is merely a named pointer to a commit node in the tree. When you created a repo, git by default created a pointer named *master*. But you can create many such pointers:

```
$ git branch p1
$ git status
$ git checkout p1
$ git status
$ git show p1
$ git show master
```

The git branch p1 command creates a new branch (pointer) named p1. But when you check the status you see it tells you are still on branch master (meaning, "you are pointing at the repository at its state after the last commit"). Doing a git checkout p1 says "I want to point at the repository state using pointer p1". When you check status this time you will see you are using branch (pointer) p1. Given that for the moment *master* and *p1* both point to the same commit, the last 2 show commands should show you the same exact thing (in particular, look at the 1st line of output for each). Git keeps track of which pointer it is using via a special pointer named HEAD (yes, a pointer to a pointer).

OK, at this point HEAD refers to p1, which refers to the last commit. Edit the file ButtonFrame.java and make any simple change you want, and commit. You can check status and all that for p1 and it is as you expect. Now change back to master:

```
$ git checkout master
$ git status
```

All should look well, there is no indication there is anything amiss. But look at the contents of ButtonFrame.java!

```
$ git show p1
$ git show master
$ git show HEAD
```

Which of these looks different and why? HEAD points at master which points at a commit object with a given SHA-1 key. P1 points to a commit object with different SHA-1 key. Why? The pointer referred to by HEAD is automatically moved forward on commit. But any others are not. As we were using p1 at the time of commit (git checkout p1) the commit moved p1 forward but not master. When we switched HEAD back to master (git checkout master) we lost the change to ButtonFrame.java because we lost the commit. Effectively master is one commit behind p1, and git manages our directory and makes sure its state is returned to where master left it.

Now go into ButtonFrame.java and edit a few lines (different from those above), and commit the changes. Repeat the 3 show commands above, are they the same now? Nope, we've add a new commit from the spot where master was pointing, so now we have *divergent branches*. Both master and p1 point to a "latest commit", just not the same one. Let's have more fun and add a pointer p2:

```
$ git checkout –b p2   # This is just shorthand for "git branch p2" followed by "git
checkout p2"
```

Edit ButtonFrame.java and make another change, save, and commit to p2. Master is one commit behind p2, but at least they are sequential. P1 is on a different sequence (commit path), but at least it shares a common ancestor commit with the other two, so perhaps there is hope. Now the question is, how do we bring them back together? (Actually the 1st question is "do we want to bring them together?" which may be an issue for the CCB or higher authority, but we'll assume we do want to bring them back together into one *codeline*).

Merging and Rebasing

The graph structure above is small and contrived, but shows 2 common patterns: 1) branches that are in the same commit path (master and p2), and divergent branches (p1). Reconciling these efficiently depends on what you are doing and where you want to go.

The 1st case is easier. As master and p2 only differ by a commit, reconciling them is as simple as moving the master pointer forward:

```
$ git checkout master
$ git merge p2
```

That's it (you can use your git show to check). In fact it doesn't matter how many commits have been applied to p2 as long as it is not divergent from master – master can "fast-forward" to catch up with p2. But we still have the divergent branch problem with p1. There are 2 ways to deal with this, we'll look at merge first:

```
$ git checkout master
$ git merge p1
```

Hmmph. This doesn't look any different than the other merge. But what happens is a little different. Since these branches diverge, git cannot just fast-forward master. Instead, it has to use information from master, p1, and the common ancestor of them to do the merge. Merge doesn't always work cleanly, you may get a message like this:

```
Auto-merging ButtonFrame.java
CONFLICT (content): Merge conflict in ButtonFrame.java
Automatic merge failed; fix conflicts and then commit the result.
```

This happens when you were editing in the same place in the file and git cannot auto-reconcile the changes. Git status will inform us:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#     both modified:      ButtonFrame.java
#
no changes added to commit (use "git add" and/or "git commit -a")
```

So now what? Well, open up ButtonFrame.java and you will see git has inserted delimiters saying what is different between the files in question, "<<<<<< HEAD" and ">>>>>>> p1". Interesting how it uses HEAD and not master eh? At this point there is no rocket science, you have to manually decide for each conflict across your conflicting files. Choose one, choose the other, combine, delete – it is up to you. Manually edit ButtonFrame.java to merge your changes. Add and commit on master as usual. Note that when you are done you will have a new commit node pointed to by master, distinct from both p1 and p2 (but also descendant of both).

Now let's look at the 2nd option – *rebasing*. To demonstrate, edit ButtonFrame.java in branch p1 again:

```
$ git checkout p1
$ <edit ButtonFrame.java>
$ git add ButtonFrame.java
$ git commit –m "extra commit 1" ButtonFrame.java
$ <edit ButtonFrame.java>
$ git add ButtonFrame.java
$ git commit –m "extra commit 2" ButtonFrame.java
```

So we just added 2 new commits to p1, and we are divergent again. Yes we can go to master and merge. Or another option is to *rebase* – to apply all the changes that were applied by the commits on master to p1 (or the current branch). In essence, you are saying you are accepting the changes made *en masse*.

```
$ git rebase master
$ git checkout master
$ git merge p1
```

What git does is linearize the set of changes (commits) made to the divergent paths of master and p1. It is considered better practice to do it in the non-master branch and then fast-forward master if it is successful, just in case there are problems. Now why would you rebase instead of merge? Basically to linearize the history. With rebase, since commits are basically replayed, it "straightens" the codeline's history, essentially obscuring the fact that a branch ever happened. This helps if you are branching to experiment with something or support a short-term spike, and do not want to cloud the codeline's history.

SUBMISSION FOR TASK 1: Please create a zipfile from the work directory. It should include the hidden .git directory. Call it <asurite>.gitlab.task1.zip where <asurite> is your ASURITE id.

TASK 2: Working with Remote Repositories

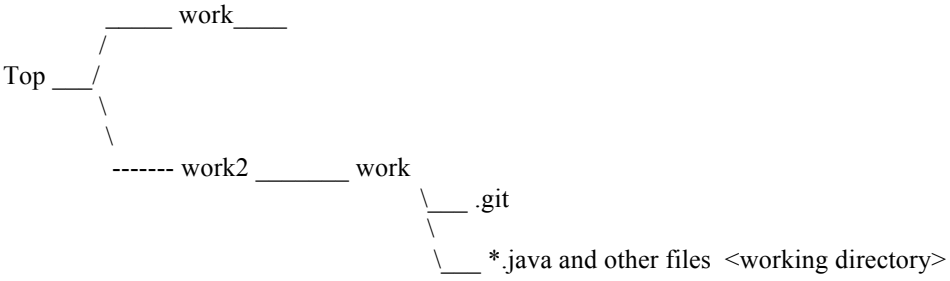
To work on this part, we need to modify your local repository. Go back to the working directory (I'll call it work), and cd to the directory above it (the directory you originally unzipped the given zipfile from). I will call that directory "top", so we have top/work:

```
$ cd <to where top/work is>
$ mkdir work2
$ cd work2
```

We want to create a new repo here *from the state of the existing repo* over in work. git clone does this:

```
$ git clone file:///<full path to>/work
```

git will respond that it did so. But now look at your directory structure:



You have a completely new 2nd repository, created from the first. In this 2nd working directory, edit ButtonFrame.java, and add a comment at the top: "Added a comment for work2 repo version only" and commit. Now do a git status. What is the history of the file?

For the purposes of this lab, we will treat our 2nd repo as the local working repo (top/work2/work) and the 1st repo (top/work) as a remote repo. So what is a remote repo and why do we need it? So far you're working by yourself in a local directory with a local repository. When working with a team or in a community, you need to be able to share your work, and to use contributions shared by others. In this sense there are "servers", but I put that in quotes because there is very little that is server-ish about it. It is really a machine that allows remote access to one or more repositories. The server-side part is in managing access and permissions and multiple repos in convenient ways, and there are a variety of freely available tools (I like gitolite) for doing this. This is really what GitHub is, and we'll get to that in a moment.

But for now, let's pretend top/work on your local machine is a remote "shared" repo, and top/work2/work is where you, as a developer, are doing your coding. Do the following:

```
$ cd top/work2/work
$ git remote –v
```

Look at the results of the remote –v command. Actually we are not pretending top/work is a remote repo, by virtue of *cloning* another repo git automatically added it as a remote repo named *origin*. Note the git remote –v command will give you a verbose listing of the remote repository branches your repo knows about, and under what alias (origin is the default). In terms you already know from above, when you cloned work into work2, you not only got a copy of the files, you also got 2 branches (pointers) – the *master* you are used to seeing from before, and a *remote branch* named *origin/master*. Originally both branches point to the same commit node (yes, when you clone you also get the entire metadata history from the original repo). But, when you start making edits locally and committing changes, you are only doing it against master.

```
$ git status  # you should be in work2/work, verify you are working on master
$ <edit ButtonFrame.java>
$ <add a text file named foo.txt with 1 line of text in it>
$ git add <both files>
$ git commit <both files>
$ git status
```

You should see something a bit different in the status:

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

Note where it says "Your branch is ahead of 'origin/master' by 1 commit." Funky stuff. Git knows you were cloned and maintaining a remote reference, so it is informing you of how your local work is getting out in front of the remote branch. So, how do you synch them up? If these were 2 local branches on the same repo, you would just do a fast-forward merge. But since we have 2 different repos, we have to tell git to *push* our stuff to the remote repo:

```
$ git push origin master
```

If you have problems with the push then you may have not converted top/work properly, or somehow got out of synch. Try doing a "git pull" and then trying the push again. So what is git pull?

```
$ cd <to top/work>
$ git pull
```

And voila! work gets the changes work pushed to it. In this typical pseudo-centralized workflow, the work repository acts as the server, and you as a team member maintain a local working git repository in work2 and periodically synch it via push/pull. There is also a fetch instead of a pull, with the basic difference being fetch grabs the updates but doesn't merge.

SUBMISSION FOR TASK 2: Please create a zipfile from the *top* directory. It should include the hidden .git directory. Call it <asurite>.gitlab.task2.zip where <asurite> is your ASURITE id. Please zip from the right directory!

TASK 3: Using GitHub:

GitHub is merely a remote repository. Well OK it is a little more than that, it has facilities for Wikis, tracking issues, integrating with other tools (stay tuned…). *To use GitHub you need to go to github.com and sign up for an account. Please use your ASURITE id and your ASU email address as your primary email*. If you already have an account you can simply sign up for a new one (please do it this way, using your existing GitHub makes things hard for us to track for class). ***DOING THIS CORRECTLY IS CRITICALLY IMPORTANT FOR THIS CLASS!***

In the middle of the screen after you signup you will see 4 big boxes. The Set Up Git step you've basically done, but they provide some platform-specific convenience features you may like. Let's start with #2 Create a new repository:

1. Click on #2 create a new repository

2. Name the repository lab1git
3. In the description put your name
4. Make sure it is public, and click the "Initialize this repository with a README" checkbox
5. In the Add .gitignore drop-down select Java
6. Press Create repository

You will be taken to the main display page for your new repository. Note there is a README.md and a .gitignore file in the repository already. Click on the .gitignore, you see it conveniently adds file extensions for files you do not want to put in source control. Over time you will want to add other stuff (like Eclipse's .project) as these files live in your working space but are not intended for source control. Otherwise you will get a lot of messages in your git status about having files you haven't added to the repository yet, and it gets quite annoying. It is even more annoying if they are added by a teammate and then you end up with them when you pull – you don't want someone else's .project file, it will have that person's paths in it!

Return to the main display page and get familiar with some of the visual features – you can browse files, commits, branches, etc. One thing you can do as well is clone the repository, see the URL near center-top that looks like https://github.com/<username>/lab1git.git. Return to your command-line window, and in a clean directory issue:

```
$ git clone https://github.com/<username>/lab1git.git
```

And you get a lab1git directory with your .gitignore, README.md, and .git stuff in it. Edit the README.md and add a line. Now:

```
$ git status
$ git remote -v   # Take a look at the output here, instead of file:/// we have https://
$ git add README.md
$ git commit -m "1st change on GitHub" README.md
$ git status
$ git push origin master
```

You may be prompted for the username and password you used when you signed up a minute ago. Because you are using https, it will ask you for credentials each time you push/pull. If you go back to your browser window you will see in a second that README.md has been updated out on GitHub. Whoop Whoop! You will also see that there have been 2 commits, and that you can go back and browse the code after each commit.

Note as well the output of git remote –v. Since we are using a true remote repository this time we need a network-enabled protocol. In reality git can support a number of protocols, including file:///, https://, ssh:// and git:// out of the box. Your team is welcome to setup ssh (http://goo.gl/WVo8R), which can help with the annoying credentials issue, but https should be sufficient for our use.

Let's try a remote branch:

```
$ git checkout -b testbranch
$ <edit README.md and add another line>
$ git commit -a
$ git push origin testbranch
```

Now back in your browser, in the branch dropdown in the upper left (under "Code") you will see testbranch show up. Easy-peasy!

Last thing: On GitHub, go to Settings, click on Collaborators on the left, and add user "cst316". Please do this so we can grade what you did on Github!

Eclipse:

Eclipse has a plugin named Egit (http://www.eclipse.org/egit/) which is supposed to be in the standard bundle, but if it is not you may need to add it. I am a text-based person so I prefer to use the command-line, but you may like Egit's features. Up to you.