CST316 Software Enterprise: Construction and Transition — Spring 2014
PRACTICE: Source Code Control using Git — January 29, 2014

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.

Because of the nature of a tutorial-as-lab, this document is longer than usual and has specific activities I ask you to track. Pay attention to when I ask you to create snapshots of your work and set them aside for submission. 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. Several of you have some understanding of SCC already, through either Git or a different SCC tool like SVN. Some of you may understand the tool but perhaps are not certain of the conceptual underpinnings. Others of you may have no exposure to SCC at all. That is fine, I will attempt to compare and contrast for the newly enlightened as well as the lightly seasoned practitioner. You will see subheadings that explain the difference between Git and traditional SCC tools. If you are new to the whole SCC thing, you should still read this stuff but don't be too concerned if it doesn't make complete sense.

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.

Install git

If you did not install git before lab, please do so now. 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).

Use the guidance at http://git-scm.com/book/en/Getting-Started-Installing-Git for your platform to install. Git. Windows users, it is the case that the Git bash shell is the best command line tool, and while it is Unix-based I recommend using it instead of the classic DOS tool (though you can stick with this if you really want to). Make sure you have git 1.8.2 or later.

Manage a directory and the files in it

Grab the files from the github link on our moodle site. In a clean directory named "work", unjar the contents. 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 unjarred the given file:

```
$ 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:

```
$ 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>
```

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

```
[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? Do a "du –s –h .git" again, what do you get? Do a "du –s –h ." what do you get? Note how the .git directory is smaller than the rest of the working directory yet it has an entire copy of your stuff plus some metadata. Git does some compression when storing objects to save space.

<u>Submit checkpoint #1</u>: jar/zip up your working directory (including the .git subdirectory) to a file named <asurite>.lab1-1.jar[zip]

<u>Manage change</u>
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"; it is like copying "Main.Jan6.java.bak" to "Main.java" – 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. 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
```

And what do you have to do after this to discard altogether? Go ahead and do it, then edit the file again. You should notice the changes have vanished. Make some minor changes and repeat the steps above, except let's add and commit changes at the same time:
```
$ git commit –a
```

This is a convenience that will add and commit all changes in your working directory in one step. I don't normally use it because I don't want to commit something accidentally, but in this case we are only manipulating one file.

If we can add and modify files, 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`.

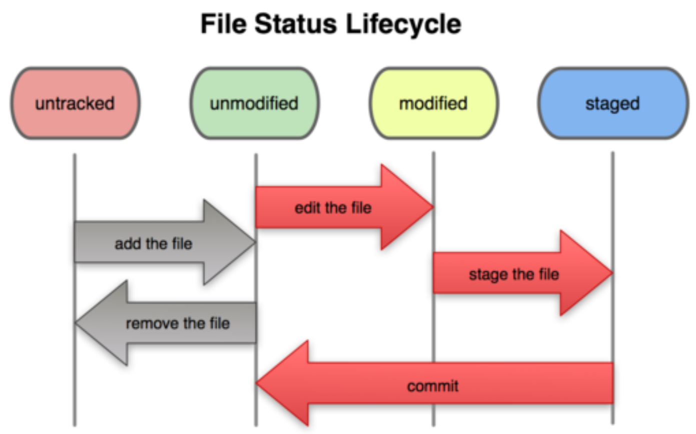
You may want to periodically check the history and other metadata about your files. Use the git show command for this:
```
$ git show ButtonFrame.java
```

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
```

<u>Submit checkpoint #2</u>: jar/zip up your working directory (including the .git subdirectory) to a file named <asurite>.lab1-2.jar[zip]

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 our 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.

How do traditional SCC systems do it? Traditional SCC systems stores the deltas between its local workspace and the last changes stored (committed) to a centralized server. The number of overlapping changes can grow quickly if team members are generating new code but not synchronizing often with the central repository. This is the reason for the "check in early, check in often" mantra that has long been considered a best practice – if you don't then there is to great a chance your changes will overlap with someone else's, and you have to figure the mess out (merge).

Since Git stores entire copies of your files as objects together with some metadata, it is able to handle branching and merging in a more 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*).

Submit checkpoint #3: jar/zip up your working directory (including the .git subdirectory) to a file named <asurite>.lab1-3.jar[zip]

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.

How is this different that traditional SCC? In traditional systems it is up to the developer to determine from which point to merge, and often this is a source of the problem with bad merges (typically something gets lost). Git takes care of this by identifying that nearest ancestor and attempting the merge from there. It 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).

Submit checkpoint #4: jar/zip up your working directory (including the .git subdirectory) to a file named <asurite>.lab1-4.jar[zip]

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.

How is this different that traditional SCC? If you've ever seen a tag history for a highly branched open source project you would understand why this is so helpful! Over time a fair amount of crud can build up (rapidly) in your traditional SCC repository. Lots of branches get created, some are cleanly merged and removed, others live a long and illustrious life, and others kinda just hang around for reasons unknown, but everybody is too scared to remove it. Each legacy branch represents another decision for change management per change, and obscures the history of the codeline. *Rebasing* is a unique concept that I highly encourage for short-term concurrent work – use a branch to do your stuff, but it isn't always necessary to keep that branch history around. Basically we want to conform to the best practice in your notes – "branch only on incompatible policy" – yet sometimes we do it for productivity and convenience. Rebasing allows is to have the best of both worlds.

Submit checkpoint #5: jar/zip up your working directory (including the .git subdirectory) to a file named <asurite>.lab1-5.jar[zip]

After creating this submission jar, you can delete all branches except the master using git branch –d <name>.

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 work1) top/work1:

```
$ cd top/work1
$ <delete all the files and directories except the .git directory>
$ mv .git/* .
$ rmdir .git
$ <edit the file config and change the bare property to true>
```

What is this zaniness? It converts your non-bare repo into a bare repo. Notice how this "bare" repository has all the stuff you saw in the .git subdirectory before. Bare repos are a strange and confusing bird at first; basically just remember they are used as a place where multiple developers will be synching against remotely, and nobody is doing any actual work in (which is why the git stuff is in the directory and there are no code files).

Now, from your work1 directory, go up one directory (cd ..), and create a second working directory (mkdir jekyll). After doing this you should have this structure:

```
         _____ work1____
        /               \___ git stuff
Top ___/
        \
         ------- jekyll
```

cd to jekyll. We want to create a new repo here *from the state of the existing repo* over in work1. git clone does this:

```
$ git clone file:///<path to>/work1
```

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

```
         _____ work1____
        /               \___ git stuff
Top ___/
        \
        \
         ------- jekyll _______ work1
                                \___ .git
                                 \
                                  \___ *.java and other files
```

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 this jekyll repo version only" and commit. Now do a git status. What is the history of the file?

OK, so why did we create 2 repositories for essentially the same stuff? Short answer: to prove a point for this lab. In reality I cannot think of a good use case for creating a 2nd local repo for the same project (if you need to manage multiple codelines you will use branches, see above). For now, we will treat our 2nd repo as the local working repo (top/jekyll/work1) and the 1st repo (top/work1) 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. Now again, going back to our original discussion, you could just use a dropbox folder or email or whatever, but instead why not just use git? Repositories can exist "out there on the Internet" and you may work with any number of them. You can even expose repos on your laptop to others, though it is more common to create repositories in well-known places under well-known labels for (well-known) people. 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/work1 on your local machine is a remote "shared" repo, and top/jekyll/work1 is where you, as a developer, are doing your coding. Do the following:

```
$ cd top/jekyll/work1
$ git remote –v
```

Look at the results of the remote –v command. Actually we are not pretending top/work1 is a remote repo, by virtue of *cloning* another repo git automatically added it as a remote repo named *origin*. That seems pretty crazy, so let's do it again: create a directory top/hyde, and clone work1 again:

```
$ cd top
$ mkdir hyde
$ cd hyde
$ git clone file:///<absolute path to top/work1>
$ cd work1
$ git remote –v
```

Now we have 3 repos, which should be organized like this:

```
          _____ work1____
         /               \___ git stuff
Top ___/
         \
         |\
         | \____ jekyll________ work1
         |                           \___ .git
         |                            \
         \                             \___ *.java and other files
          \_____ hyde _______ work1
                                     \___ .git
                                      \
                                       \___ *.java and other files
```

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 work1 in each of jekyll and hyde, 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 hyde/work1, 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/work1 properly, or somehow got out of synch. Try doing a "git pull" and then trying the push again. So what is git pull?

```
$ cd top/jekyll/work1
$ git pull
```

And voila! Jekyll gets the changes Hyde pushed to the (bare) repository. In the typical pseudo-centralized workflow, the shared remote bare repository acts as the server, and each team member maintains a local working git repository and periodically synchs 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.

So you may be wondering – what is the deal with this "bare" repository business. You can read this: http://www.saintsjd.com/2011/01/what-is-a-bare-git-repository/ or I will try to summarize. Keep in mind always that the .git directory is keeping not only metadata but compressed binary copies of all objects. When you push into a remote repository that is not bare (has a copy of all the working files) you would be writing new stuff to that .git directory *but not to the working directory*. So, the working directory would seem all out of sorts with the .git metadata. Git 1.8.2 will reject the push. There are workarounds of course. You can do a reset, create a private branch, or set a configuration property in the remote repository. But it is an unusual scenario for one to need to keep a working copy of the source files in a shared repository – too many folks can get you out of synch too quickly – so often you just create a bare repository from the start (using git init –bare <directory>) and push stuff to it from afar.

Submission #6: Before continuing, cd to top and make a jarfile of all 3 directories underneath and call it <asurite>.lab1-6.jar[zip]

Remote branching

In the above we pushed/pulled to/from local/remote branches. There were 3 distinct repositories, in top/work1, top/Jekyll/work1, and top/hyde/work1. In each distinct repository there is a branch called *master*. Further, in Jekyll and hyde there are references to remote branches under the name *origin*, each of which point to top/work1's *master* branch.

Now consider this scenario: Jekyll and Hyde are on a team with Holmes and Watson. But Jekyll and Hyde are currently working on an issue collaboratively and want to share work through the shared repository. Yet, they do not want to interfere with the work of the rest of the team (Holmes and Watson). They need a branch they can refer to that is in a shared spot but just for them – they need a branch in the top/work1 repository. How do you do this? Simple, create a local branch and push it to the shared repository:

```
$ cd top/Jekyll/work1
$ git checkout -b sidestuff
$ <edit ButtonFrame.java>
$ git status
$ git commit -a
$ git status
$ git push origin sidestuff   # This pushes the new branch to the remote repository
(origin)
```

Now Jekyll becomes Mr. Hyde;

```
$ cd top/hyde/work1
$ git pull
$ git checkout sidestuff
# look at ButtonFrame.java, it should have your sidestuff-only changes
$ git checkout master
# look at ButtonFrame.java, it does not have your sidestuff-only changes
```

Submission #7: Before continuing, cd to top and make a jarfile of all 3 directories underneath and call it <asurite>.lab1-7.jar[zip]

Using GitHub:

GitHub is merely a remote bare repository. Well OK it is a little more than that, it has facilities for Wikis, tracking issues, integrating with other tools (like Scrumwise and Jenkins, stay tuned…). To use GitHub you need to go to github.com and sign up for an account.

GitHub provides a lot of material for newbies, like the bootcamp link you will see. In the middle of the screen after you signup (and after each time you sign in) you will see the 4 big boxes. The Set Up Git step you've basically already done, but they provide some platform-specific convenience features you may enjoy. 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 and your partner's 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 will 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!

Now collaborate with a partner. In the Setting menu of your lab1git project (see upper right), you have a Collaborators link. Add your lab partner to yours, and vice-versa. Clone each other's repos, make a change to README.md, and push. Double easy-peasy!

For your projects one of the team members should initialize the codeline under a public repository, and add the rest of his/her team to it as Collaborators. Also add me, user cst316.

Eclipse:

Eclipse has a plugin named Egit (http://www.eclipse.org/egit/) which is supposed to be in the standard bundle, but I did not have it so 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.

SUBMISSION:

I asked for 7 submission points throughout the lab. Jar (zip) the 7 submissions up into one jar and submit via Moodle. The individual jars should not be large, so it should all be within the upload limit of 2MB.

Leave your git lab1git repository intact. Make sure the repository description includes your partner's name. I will assume you created your git account name the same as your asurite it; if not please indicate what it is in a readme.txt in your jarfile submission.