Assignment 1 -Source Code Control using Git

Version: May 13, 2025

Objectives

1. Understand SCC concepts
2. Gain proficiency with git
3. Learn to use GitHub as a remote repository

General

There are 2 tasks in this assignment, I require you to upload one zip file to Canvas at the end for task 1. Additionally for task 2 it asks you to create and place some files into your GitHub account. It also instructs you to add GitHub user ser316asu 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! If you do not do the above you will not receive points for these parts.

This assignment is very long based on the text and explanations. In case you already know Git/GitHub you can skip a lot this reading and just follow the code prompts and edits you have to do.

PREFACE

This assignment is admittedly more of a tutorial than a real assignment, it will guide you through many steps. 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 assignment covers tools, using Git locally, and using Git with remote repositories. Tools are spread throughout the activities, as I will introduce Git and GitHub over the course of the activities. Let's start with a discussion of what Git really is.

You can do parts over again or try again without having to start over. It is important however that we do find that you did everything somewhere even if you did have to try 5 times.

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, 2015 for some reason? Perhaps you edited Main.java and made a whole bunch of changes, and now wanted to go 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’s repository out there, when I want, under the conditions I want, and with only the stuff I want?”.

How is this different than 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: Getting to know Git (20 points)

Manage a directory and the files in it

On Canvas is a zipfile of a source code tree to use with this assignment. Extract this zip file, which should lead to a folder "labgit-master". You will see it is a simple source tree with a few other things thrown in (it is not code you need to run). The first thing we have to do is ask git to create a local repo for us for this folder, since we want to version control these files. At the command line, in the directory in which you unzipped the given file (so in labgit-master), issue:

```
$ git init
```

git should come back with a message: Initialized empty Git repository in

<directory>/.git/ You should see this new folder .git, it is a hidden directory.

Go ahead and check out the contents of this hidden directory:

```
$ ls –la . git
```

You can of course also open the folder in your Finder/Explorer (you may have to change your settings to see it), but I would advise you to get "friendly" with your command line. The <> later on stands for "put the right command here or enter names that make sense".

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 to your IDE yet) and add a few lines and delete a few lines (commenting lines are fine). After saving the file, 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”. This deletes the changes you made!

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 (hit ’q’ to get out of this view). 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
```

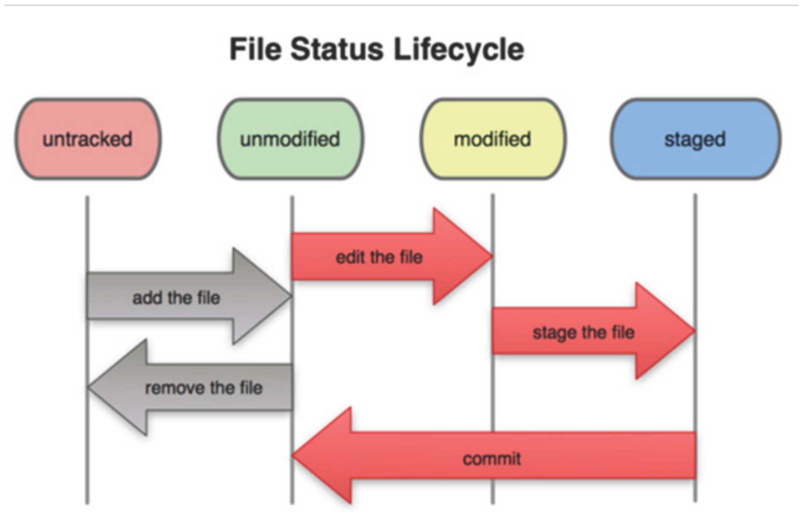
Summary (so far)

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.

If you still have trouble with these go over things again and play with it a bit. That is totally fine for this assignment. You can add, commit, delete as much as you want. Just make sure you still have some things to actually keep working with.

A great way to think about the managed directory concept is in terms of state machines (UML statecharts). 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? (rhetorical just for you).

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.

IMPORTANT: If I do not give you a specific commit message or tell you what to add to a file, always include the branch name and write good commit messages. This is only so we and you can see your changes and where you made them a little easier and is not a git requirement but an assignment requirement. By the way, "unprofessional" commit messages, e.g. "No clue how this sh** works", "this is a dumb assignment" (yes this happened) will be penalized!

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 #shorthand for ”git branch p2” and ”git chechout p2”
```

Edit ButtonFrame.java and make another change, save, and commit to p2 with the commit message. 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).

If you did NOT have a merge conflict as mentioned above I would advise you to go over the steps again and try to create a conflict so that you can practice this!

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 - with the rebase command, you can take all the changes that were committed on one branch (here p1) and replay them on a different branch (here master).

```
$ 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. Rebase might also give you an error that a automatic rebase is not possible (as in Merging and Rebasing). In the git command window will tell you that it is still in rebase and you can either abort your rebase or you can manually edit you code again. If you run into that problem go to your file and edit your file manually. Then do an add, commit and continue your rebase.

Submitting task 1

Please create a zipfile from the work directory. It has to include the hidden .git directory (double check that it does). Call it <asurite>_task1.zip where <asurite> is your ASURITE id.

TASK 2: Using GitHub (20 points)

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. . .). Use your GitHub account for the following assignment. This is to prepare you for what is to come in your project but with the project you will of course have a lot more files and people working on the same repository.

Create a **new** repository in GitHub (do not use the repository of your project!!)

1. Name the repository assign1git
2. In the description put your name

3. Make sure it is public, and click the “Initialize this repository with a README” checkbox
4. Select ‘Add .gitignore : None’ and change None to Java, the type of gitignore file you need to add to your repository.
5. 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 things (like Eclipse’s .project, .class) 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 or build directory, 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>/assign1git.git. Return to your command-line window, and in a clean directory in your file system issue (copy that link from GitHub):

```
            $ git clone https :// github .com/<username>/assign1git . git
```

And you get a assign1git directory with your .gitignore, README.md, and .git in it. Edit your README.md and put "First comment" in it.

In your command line window go to the new assign1git directory created and execute the following commands

Now:

```
$ git status
$ git remote –v # Look at the output , instead of f i l e :/// we have https ://
$ git add README.md
$ git commit –m “1st change on GitHub” README.md
$ git status
$ git push origin main # GitHubs main branch name was changed to main
```

You may be prompted for the username and password, better yet setup SSH to connect to GitHub (read the documentation on how to do this). If you go back to your browser window you will see that README.md has been updated on GitHub. 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 add " testbranch change">
$ git commit –a
$ git push origin testbranch
```

With your commit like this you will get into a text editor. Enter a commit message then hit 'esc', ':wq'. You should ask Google for more information if you need it.

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

Look at the differences between the two README.md files on GitHub.

Got back to your local repo again and look at the README.md. Now do:

```
$ git checkout main
```

Look at the README.md now. You should see that the one version is the "old" one without your changes on the branch and the other one on "testbranch" has your changes.

Now do the following (make sure you are currently on your main branch):

```
$ git checkout —b newbranch
```

Which version of the README.md do you see?

```
$ git checkout testbranch
$ <look at the README.md again you should see the version with your second change>
$ git checkout newbranch
$ <add another line to your Readme>: "This is a change on newbranch ".
$ git add README.md
$ git commit —m "change on newbranch" README.md
$ git push origin newbranch
```

Check your GitHub again and make sure the branch is there.

Now switch around between your 3 branches (main, testbranch, newbranch) all three files should look different now (on Github and locally). This is a scenario which you will encounter a lot during your project.

Now, go to GitHub and select testbranch from Branch dropdown. Now, you should see code for your testbranch. Open README.md and click on edit on the right side (small pencil icon) to edit the README.md. Write "new git branch on GitHub" at the end of the file. At bottom of the page, in commit message write "1st commit of GitHub directly" and select option "Create a new branch for this commit and start a pull request". Name the new branch "newgitbranch" and commit changes. **Exit the Create pull request without creating a pull request.**

On your machine locally run the following commands.

```
$ git checkout newgitbranch
```

Did you see an error? This is a typical scenario when your teammate will create a branch but it is not on your local repositor yet. To do this, execute the following commands.

```
$ git fetch —— all
$ git checkout newgitbranch
```

Was it successful? Now, you can work on this new branch.

Now lets say you want to merge your testbranch into your main. **Make sure you are on your main branch locally.**

```
$ git merge testbranch
$ git push origin main
```

So, now everything from testbranch is in the main branch and our local repo is up to date with the remote repo.

Now, this is basically a situation where 2 team members might have checked out at the same time from main and then both worked locally on a different branch (testbranch and newbranch). One of you got done (testbranch) and merged into main. So this one had the easy merge into the main, since it was a fast forward.

For our second developer, it might get more complicated since newbranch is now not based on the main that he started working on initially. There might be conflicts.

Since you work alone we want to try to simulate this scenario. So lets assume that you just checked out the new main from GitHub (for you, since you are working alone, the local and remote main is the same) and now you would like to put your things from newbranch into main. You could now of course merge into main and hope that there are no conflicts or that you can resolve the conflicts in an easy way. I propose however that you never merge into the main branch if it is not a fast forward. Why? Because in your project you might not only create conflicts in your code but the features you implemented might not work together and just merging into main might break the main branch. You should always try to avoid breaking the main branch.

So a better approach is, to merge your main into your branch (newbranch) and test your changes together with the new main. If you get conflicts here remember what you did in task 1 where you needed to edit the files manually. When everything works then you fast forward the main. The commands would be these:

```
$ git checkout newbranch
$ git merge main
$ git push
```

Now your main should be merged into your newbranch and depending on the changes you made git was able to do the merge or you needed to manage a conflict as in task 1. Newbranch is now ahead of main since it still has changes in it that the main does not have. But newbranch is now based on the current main.

In your project you could now continue your work on newbranch and make sure everything works and then merge your changes into main and push. But if anyone is allowed to merge into main then you might have pushes into main that break the main (or you might push to main accidentally). So lets go a different approach.

A good practice is to protect the main so you cannot merge into the repository's main without a pull request. You can only do this if you have a paid or student account on GitHub. I would advise all of you to get a student account. So this part about the protected branch is optional.

Go to your GitHub: Go to Settings, then Branches and take a look at Protect Branches. Choose main and check "Protect this branch", "Require pull request reviews before merging" and "Do not allow bypassing the above settings". Now save your changes on GitHub.

Now try:

```
$ git checkout main
$ git merge newbranch
$ git push
```

The first two commands should work without a problem but when you push you should get an error message, since you are not allowed to push to main anymore.

Not optional anymore:

Go to GitHub again. Go to "Pull Requests" and create a new Pull Request. (You might want to read up on Pull Request to know more about it, the GitHub documentation is pretty good.) Now, specify that you want to merge newbranch into main. It should tell you that the merge is without conflicts. This should always be your goal, since you always want to fast forward main.

Remember: *Before you merge into main, always merge the current main into your branch to check that everything works. Then when it does you can create a pull request and it should be a fast forward (if no one else changed the main in the meantime).*

Click create PullRequest and add a comment. Make sure this PullRequest has no conflicts, if it does fix it. PullRequests with conflicst should not be approved and merged before resolving them!

Now you should see the request. For this PullRequest a review is required (in your project a team member should review the request).

Since this is an individual assignment you cannot approve your own Pull Request and would thus not be able to merge into main as it is.

To do a PullRequest review: In the Pull Request go to the tab "Files changed". There is a green "Review changes" click on it, write a comment and choose approve (in your project you should of course really review the changes and approve or deny based on that).

In our case you can either should'' add me as a Collaborator (you have to anyway) and send a message through Slack by typing /sparky_appeal into a channel, which will open a dialog. Fill out this dialog and **add the link from your PullRequest** so we can approve your PullRequest by following the link. If you submit the link to the PullRequest via /sparky_appeal your work is done for this task. If you still have the time please merge the Pull Requests (so you know how it works) we will give 1 points extra credit for this.

Submitting Task 2

Last thing: On GitHub, go to Settings, click on Collaborators on the left, and **add user ser316asu as collaborator**, then copy and paste the GitHub link to the submission comment on Canvas. Please do this so we can grade what you did on Github! If you forget this you will be asked to do this again but there will be a 10% penalty for us having to ask to be added.

In case you want to use Git from 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 Eclipse features. Up to you.

Submission for this assignment summary

Task 1: You should submit a zip file (<asurite>_task1.zip) with your folder from task 1 on Canvas (include everything even the hidden files, especially the .git folder without it we cannot grade and you will receive 0 points).

Task 2: You need to add ser316asu as Collaborator to your project on GitHub so we can see it **and paste the GitHub link to the comment section on Canvas in your submission**.