Learn to use GitHub

How to work on the Menu_State (project specific)

Basheer Becerra

In this document, I will describe the important functions of GitHub and how to do them in eclipse. At the end of this document, I will talk about how to work on the menustate branch as well as talking about Hierarchal branching. So if you are working on menus, look at that. It will be at the end of the document.

This might become difficult to understand. If you don't know, GitHub is a team used by ACTUAL organizations. For example, when I talked to a StateFarm employer, they said that usually people after their 4 years of college know this stuff, so you should definitely be proud of yourself after this. I will try to simplify it as much as possible then go into more specifics. And don't worry, at the end, I will explain how to do this in Eclipse. For now, just get the concepts as it is crucial to work on this project.

If you need any help with this, try and meet with me in my free hours or after school in club. My free hours are at 7AM, 10AM, 1PM and 2PM and I will be at club meetings Wednesday after school in the Tech Conference Room.

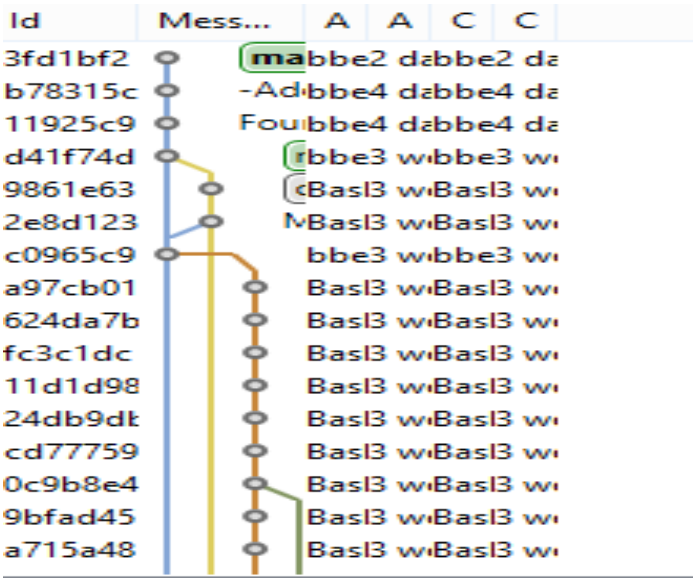
GitHub Concepts

Branches

Branches are just lines of development. They are intended to track the changes for a specific task (as general as holding the code of a whole state [such as gamestate or menustate] or as specific as fixing a small bug). So really the purpose of this is that the project will not collapse if a mistake is made, especially when worked on by multiple people. Also, this adds organization to a project. In every project (not just this one, but for EVERY software) there is a branch named **master** or anything equivalent to represent the *main* branch. Note that the **master** should be UNTOUCHED except for the organizer (which is me in this case). You can create other branches for your own development then you can add your development to the master. This will definitely be explained throughout the document. When creating your own branch, it is better to create it intended for a specific task. I am repeating this because if you do too much other tasks in the branch, it will cause problems in the future (especially when merging which we will talk about later).

Commits

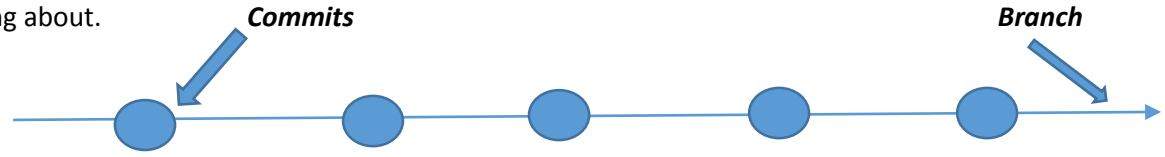
In specific terms, branches are made up of *commits*. Commits are pretty much just snapshots of a code. When you make a change in the code, such as adding a method, you can *commit* that code so it saves within the branch. This is useful if you made any mistakes in the future, you can go back through your commits and revert back to a specific on. (If this happens, please contact me for instructions on how to *rebase* or go back to a commit). So if you look at a *branch* in GitHub, you will pretty much just see a straight line with a bunch of circles in the middle. Each circle represents a *commit* which is chronologically ordered. Here is a screenshot of the history of the project currently.



So really this diagram isn't what I described exactly, since there are multiple lines (or as I'll explain later, multiple branches, we will talk about it once I get into merging).

To get that view, just right click on your master branch, click Show In, and click History, but before you do that, you will need to setup your windows and layout, which we did at the beginning of the project.

I will teach you how to *commit* at the end of these sections. Here is a simpler diagram of what I am talking about.



Push and Pull

Now we will talk about how to send and receive information from your computer to the server. Before we get into pushing and pulling, we will need to understand two important terms: **local** and **remote**. **Local** is just all the data stored on your computer for your own use. **Remote** is data stored on the cloud which is available (or mostly availably if there are restrictions/permission settings) for all to view or use. In simple terms, **pushing** is just sending data from the **local** to the **remote**, and **pulling** is receiving data from the **remote** to the **local**. In other words, you push to upload your code to the remote, and you pull to receive code (or update code) from the remote. There is something called **Fetching** but is not of any use for now. As I said, I will explain how to do this on your Eclipse later.

Merging

Here is the last main concept to understand in GitHub. **Merging** is combing the code between two branches. Why do we do this? Well when two or more branches are finished with their specific task (as what branches are intended for, a specific task) Then they can be **merged.** One example is if you created one branch to, for instance, create a menu for a game. If you finished the menu, you can now **merge** it to the **master** branch. Then once another branch is finished, it can be merged to the **master**. We will

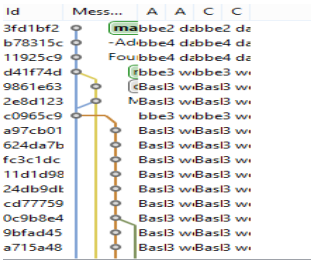
describe more of this in the next section. Sometimes, the branch does not really have to be finished, but it should definitely be working code free with known bugs.

IMPORTANT

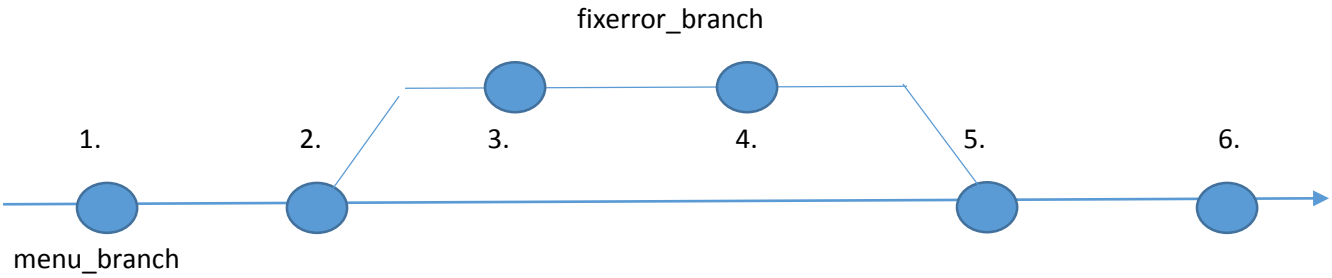
One thing which might be the most important thing in the document is that **YOU SHOULD NOT DIRECTLY MERGE TO MASTER**. Since the master is the main line of code, it must be administered carefully and free of mistakes, otherwise the whole branch system will collapse (this will be administered by me). So here is what you should do when you think you are finished with your branch. You will create a **pull-request** from your branch to the **master**. Pretty much this will submit a request to me asking if it is OK to merge your branch to the master. I will look at all the changes, test your code, and make sure everything is working. If it works, great, I will accept the request to merge, if not, I will decline and ask to change anything specific. The bad thing about GitHub is that people can just merge the branches directly in Eclipse. There are ways to combat this, but takes a lot of setting up. Refer to the **Creating a Pull Request** for instructions.

Explaining the complicated branch diagram

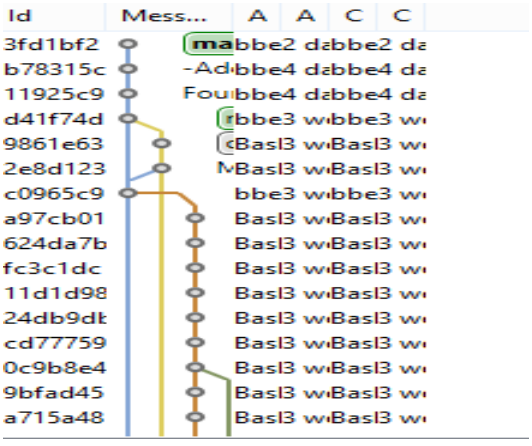
I'll fully explain the branch diagram that I showed above since you now know about merging and commits.



Let me draw a simpler diagram for my ease of description. (next page)



So you will see how this diagram now involves two different branches: **fixerror_branch** and **menu_branch**. I'll interpret what this diagram means. Of course, you see the horizontal lines represent branches and the circles represents commits, but what about the lines that are diagonal and what relationship does it show? For the first one on commit #2, you see a diagonal line from the **menu_branch** to the **fixerror_branch**. What this means is that you **created a "fixerror_branch" branch based of the branch "menu_branch".** So in your new **fixerror_branch**, you can now work independently from the **menu_branch** for your own development. At the time of creation, both of the branches are identical (except for the name) until a first commit is made such as commit #3. So this means that the **fixerror_branch** is one commit ahead of the **menu_branch** (which you might see around the GitHub website). How about the line connecting commit #4 to commit #5? Well when the **fixerror_branch** is finished, it is now ready to put its changes back into the main branch that it was originally based off of. This is where we **merge** the branches. Now the **menubranch** inherits all the changes from the **fixerror_branch** unless there are merge conflicts which usually occur if while the **fixerror_branch** was in development, the **menu_branch** was also in development. So underneath the commit #3 and commit #4, you would see more commits in the **menu_branch**. So what a merge conflict means is that the program is having trouble with both changes. Refer to the **Merge Conflicts** section below in this document for help. Now look at the original screenshot and try to interpret it!



Now I will show you how to do all the actions I had described above, which will be horrible on my part since I have to screenshot and crop all the buttons :/ You're welcome by the way.

Also, if any tabs or icons don't show up, then your window/layout was not setup correctly (or setup at all). Contact me about setting it up or look up Setting up GitHub on Eclipse on YouTube.

Setting up Github (Import Project and Settup Layout)

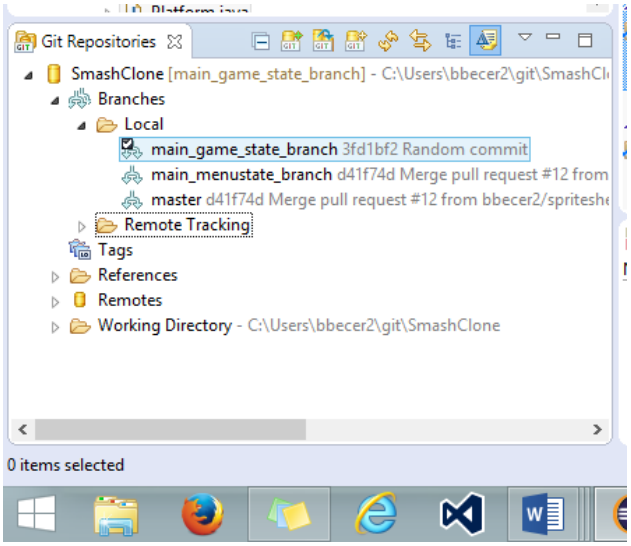
Watch this video.

https://www.youtube.com/watch?v=r5C6yXNaSGo

Create a local branch

How to create a **branch** on the **local** (your computer) (we will figure how to **push** this to the **remote** later)

1. Go to your Git Repositories tab, open the SmashClone Repository, open Branches, and open Local

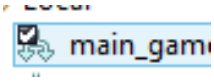


2. Right click on one of the branches that you want to **base** on and click Create Branch. All code from that branch that was right clicked will be copied to the new branch you are creating. This is if you want to create a new line of development based of the branch you are currently in (for the most case).

Develop on a specific branch

It is important that you are on the branch intended otherwise you done screwed up bro. But if it happens, contact me. We may be able to commit your changes, and copy that commit to the intended branch. Otherwise, do these steps.

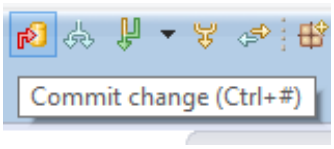
1. Right click on the branch you want to work on and click **CheckOut**
2. A grey checkmark will appear on the branch as shown here. (Frequently check this)



Commit a set of changes

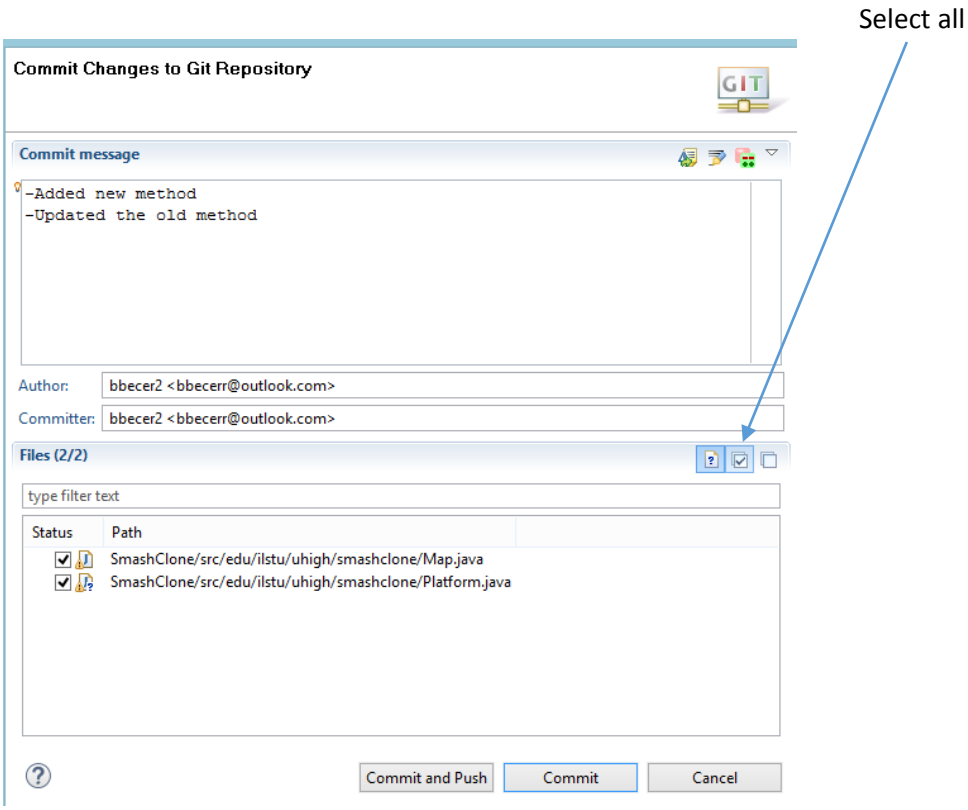
Once you coded like a boss and feel like you have done enough, you can now commit your changes. If you as when it is appropriate to commit, know that the more commits the better, but not too much to where it is chaotic (such as every line of code). Maybe like you added a method or you implemented some small change.

1. COMFIRM that you are checked out on the correct branch.
2. Click this icon on the top toolbar (if you don't see it, your layout setup is wrong, contact me).



(If you hover over it, it will say Commit change

3. Fill in a detailed Commit description and check all files (otherwise none of your changes will be uploaded. You can check the "Select All" button if you are too lazy to click the so many files you updated.



4. Select COMMIT, do not click commit and push (it is safer to do in two separate steps)

5. Click finish or any final steps (IMPORTANT: If an error occurs and states “fast-forward error” This means that your branch or your superbranch (or the branch you were based off of [what u right-clicked and created off of]) is behind the **master.** This occurs when a merge was made to master, so master has changed and as a rule, all code must have every commit of the **master** to make sure everyone’s branches work with the changes. This just ensures that the whole system doesn’t collapse. So you cannot make any changes until you have all the changes from master. Refer to the **Fast-Forward Error** section.

Push

If you want to upload your branch to the remote (which should be after every commit), then you will need to push it. Here are the instructions.

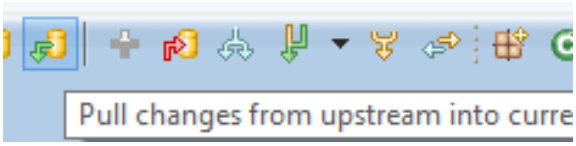
1. Right click your branch you want to push
2. Click Push Branch
3. Done! A confirmation will show display all the commits you made before the last push

Pull

If you want to update any changes to a branch made by someone else from the remote, or if you just want to download a whole branch to your local, you will need to pull a branch. There are a few methods I use to do this.

UPDATE A BRANCH

1. Checkout the branch you want to update
2. Click the **pull** icon on the top toolbar



3. If there are any changes from the remote, then it will download AND merge to the checked out branch (if you are wondering what **fetching** is, it is pretty much pulling without the merging)
4. If there are no changes from the local to the remote, an error (not really an error) will show that it cannot pull since the local branch and the remote branch is identical.

PULL A NEW BRANCH

1. In your Git Repositories window, open Branches, then open Remote. This will contain all the remote branches
2. Double-click on the branch you want to download
3. Click on “Checkout as New Local Branch”
4. Done!

Fast-Forward Error

Fast-Forward error occurs when your branch is behind the master branch, meaning that the master has been updated, but your specific branch did not update with those same changes. This error occurs because since master is the main branch, the specific branch should always be based off of it, so all changes must be pulled by all branches. Here is what to do if you receive the error.

1. Pull any changes on the master from local to remote (Pulling explained above)
2. Now checkout Your specific branch (**NOT THE MASTER, MAKE SURE YOU DO THIS)**
3. Right click on the local master branch and click **merge**. (You can skip step one and merge on the remote master branch under the remote folder, but it is good to keep your local updated)
4. If this is not successful, usually red marks will appear all over your files. This means that there were conflicts in merging two branches (which logically makes sense). Refer to the **Merging Conflicts** section.

Merging Conflicts

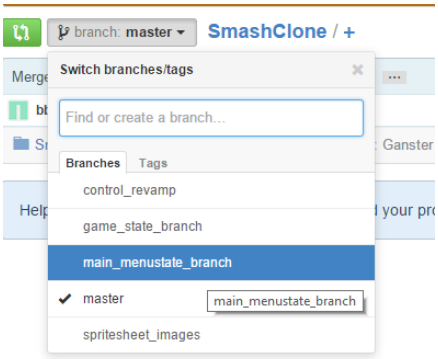
This is due to conflicts that occur when you merge two branches. Usually this should not occur if you merge frequently between the two branches and ensure that each branch is towards a specific task, not a whole lot of tasks unless it is high up in the hierarchal scale (the branch hierarchy is described below). To fix it, please watch this video on youtube:

https://www.youtube.com/watch?v=ZK20jVt7XEc

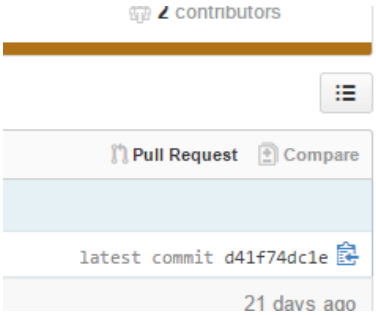
Creating a Pull Request

A pull request is for requesting to merge a branch to another branch. While this can be done directly on eclipse (as shown above), merging to important branches (such as **master** or any main branch worked on by multiple people). Creating a pull request will be done on the github.com website.

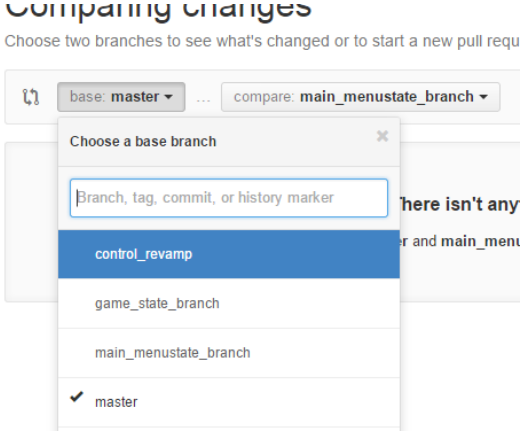
1. Go to www.github.com/bbecer2/SmashClone
2. Select the branch you want to submit a pull request **from**



3. Click Pull Request



4. Ensure that the base that you want to send the pull request to is correct. It can be maste, or it can be a main section branch (EX: main_menustate_branch)



5. Fill out any descriptions and click Submit Pull Request
6. If there is an error saying it cannot automatically merge, this is due to potential merge conflicts. If this occurs, check to see if your branch is updated with any recent changes in master and update your branch. If it still does not work, contact me so we can carefully do a merge conflict resolve. You can look more into **merge conflicts** above.
7. I will then review your pull request and decide if it is finished to merge to the master or the main branch. If not, I will decline with comments on what you need to change.

Main State Work and Hierarchal Branching

For everyone working on the menus, here is just a brief document on what you will be working on. Of course, if you have any questions about this, please try and ask my during club meetings. If you cannot make any club meetings, you can ask me anytime in the day when you see me.

Branching

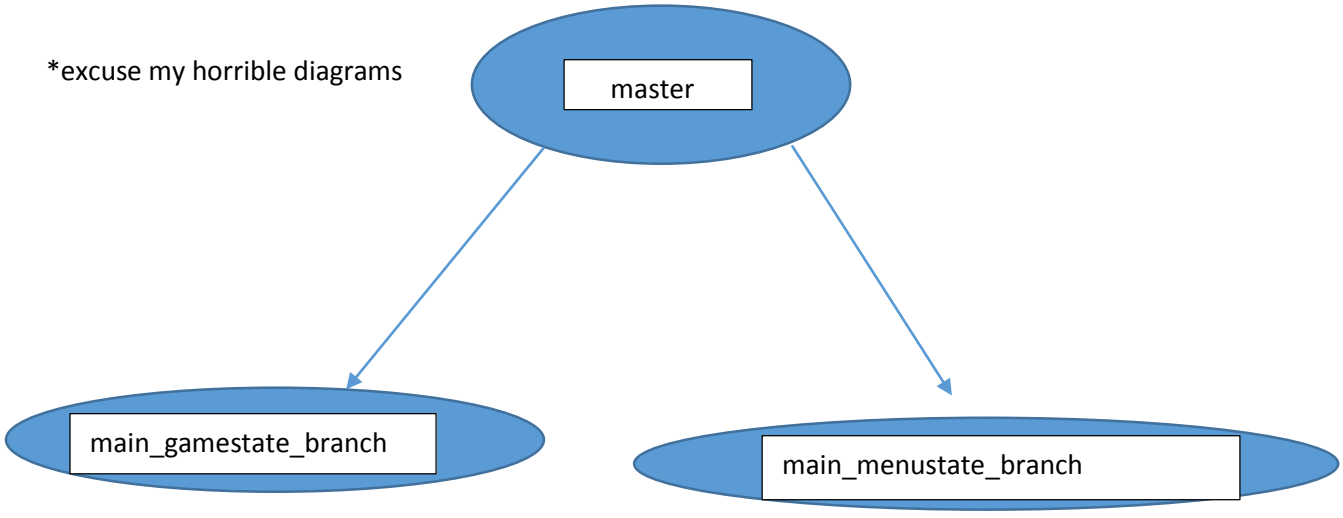
It is important that you understand how the GitHub branching works in order to develop. If you look in our GitHub at www.github.com/bbecer2/SmashClone you will see three main branches that we are using (there are some other ones, but don't worry about them as they are for backup). The branches we have in the remote (*remote: on the server for access by anyone on the team*) include:

master, main_gamestate_branch, main_menustate_branch

(like I said, the couple others *controle_revamp* and *spritesheet_images* won't matter to you.)

These three branches are important in what area you are developing. If you haven't guessed already, you will be working in the ***main_menustate_branch*** since you will be working on menus. One other thing to understand is how the branches are *based.* Describing in words, the ***main_gamestate_branch*** and ***main_menustate_branch*** are based off of ***master.*** This means that when I created these branches, all the code were directly inherited from ***master***; In simple terms, all the code is copied from master to the created branch, so at the time of creation, the branches are **identical** until a commit (*commit: a specified change in code, added to the line of development*) is made. Here is a diagram of the branches I made.

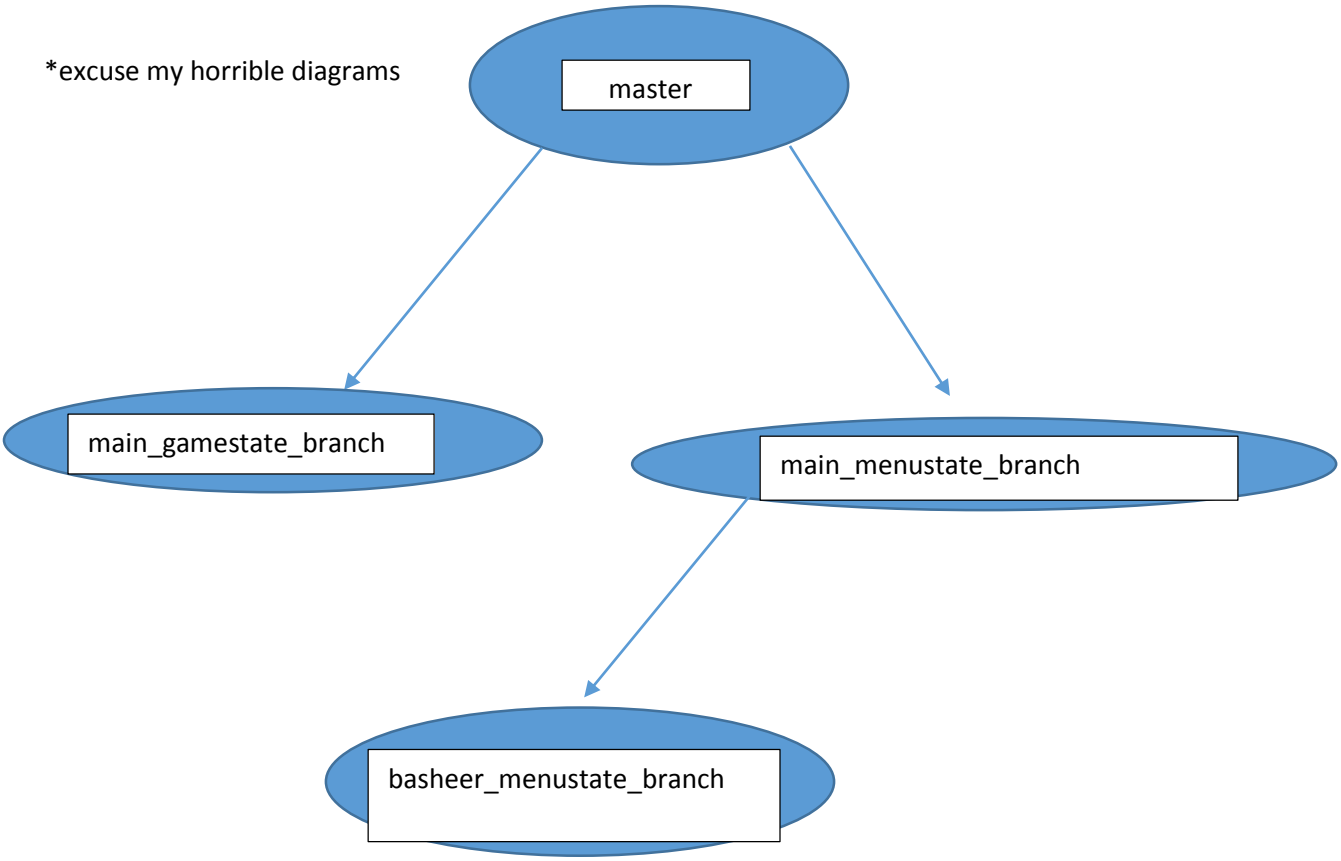
*excuse my horrible diagrams



As you can see, the branches are somewhat based in hierarchal order. Now here is where YOU come into play. Since you will be working in the ***main_menustate_branch***, you will now create a new branch *based* off of the ***main_menustate_branch***. So everything in the branch is now copied to YOUR OWN branch for YOUR OWN development so if you mess anything up, the whole project does not collapse ☺,

but in all seriousness, this is the same reason as creating the separate gamestate and menustate branches. Branching will just create specific lines of development (or just branches) for its own task which in effect will create better management, organization, etc. So here is a diagram for if I was creating my own branch.

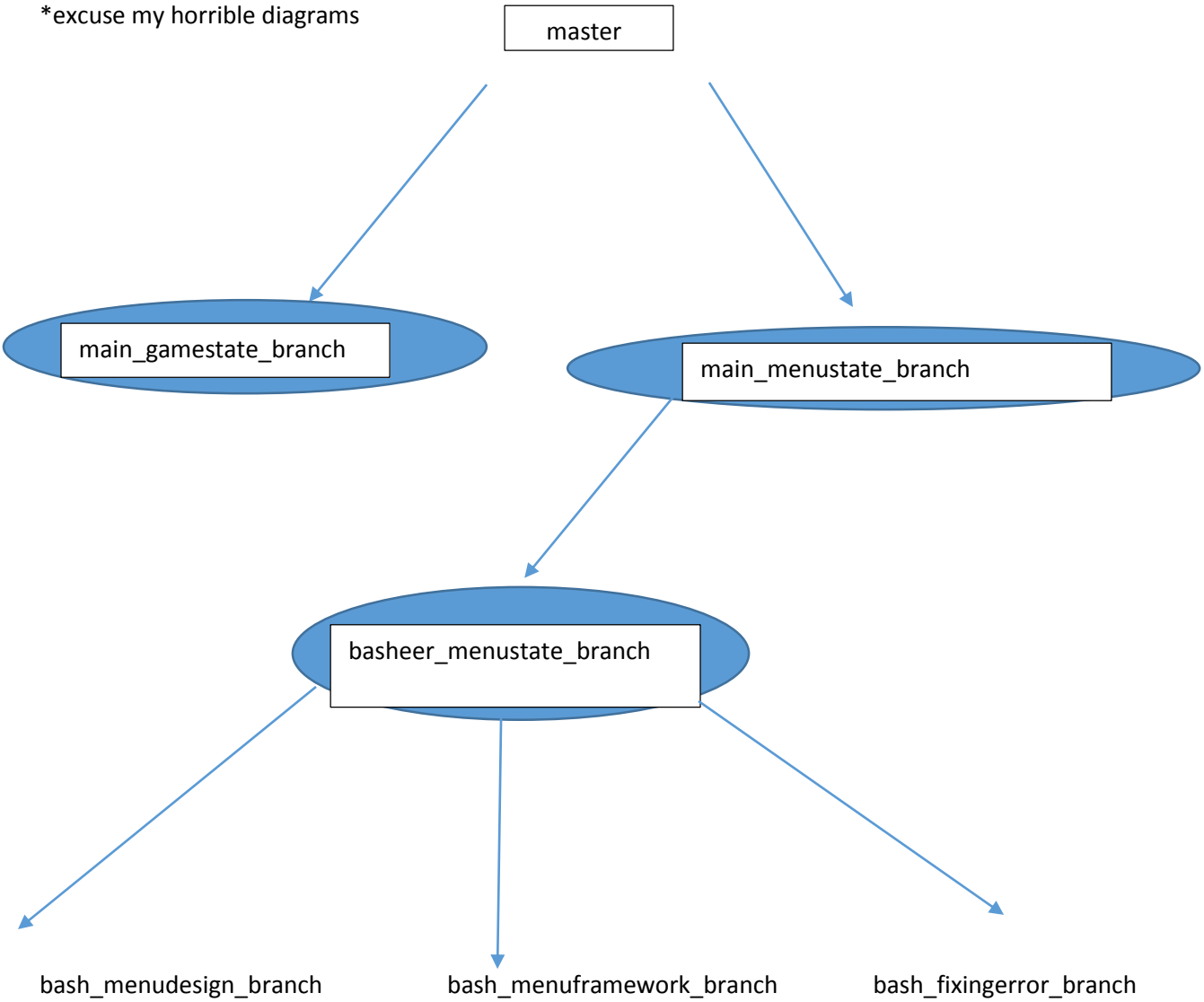
*excuse my horrible diagrams



To do this in Eclipse, Refer to the "Create a Local Branch" far above in the document. You will just right click on the "main_menustate_branch" then click Create Branch. This will create a branch "based" off of the menustate branch.

Please follow the naming of the branches **name_menustate_branch** or generally **name_specifictask_branch.** If you don't name it right, I will change it for you. . . If you wanted to do separate tasks within your own developmental branches, then you will create even MORE branches *based* off of YOUR OWN specific branch NOT the main branch (specifically NOT ***main_menustate_branch***). I will create even another diagram just so I know you will understand.. (next page)

*excuse my horrible diagrams



(The names are just ideas, also I was too lazy to insert circles, its 12AM right now ☹ . . .)

To do this in Eclipse, Refer to the "Create a Local Branch" far above in the document. Instead of Basing off of the main_menustate_branch, you will now bas it off of your own branch you created (that was originally based of the menustate branch). So you will right click on your branch then click create branch. Please refer to "Create a local branch" far above in this document for detailed instructions.

So that is the layout for just ONE developer. Of course there should be multiple strings of these associated branches for each developer. Now what is most important is to maintain these branches. The rule is to make sure that the base is updated and maintained. Personally, I will maintain the ***master, main_gamestate_branch, main_menustate_branch.*** But the branches that you base off of these should be maintained. So any sub branches should be appropriately merged between the super branch (the branch that based the sub branch EX: master is the super branch of the main menustate branch) and all

sub branches should be merged and updated from the super branch. Now you should know that you must create pull requests to the master (requesting permission to merge to master which can be done on GitHub) but since you will be working in the menustate branch, you will need to now create a pull request towards the **main_menustate_branch**. Refer back to the **Creating a Pull Request** section far above on when and how you create a pull request.

Any questions please ask. My free hours are at 7AM, 10AM, 1PM and 2PM. You can also contact me Wednesday after school during Tech Club. You can text me at 309-684-4641. You can also Skype me at bbecerr@outlook.com. Thanks.