**CS2820 Object Oriented Group Project Design Document**

Daniel Conway, Alex Pang, Seojin Ko

**Use Case Diagram:**





















**Textual Descriptions:**

Commit:

Requirements: Commit will allow the user to save their file to our revision control system with the option of choosing a branch name to give the new version.

The user will use the command to send the required information to the program. Then, our program will store the differences between the checked out version and the current local file as a child of the checked out version. If a desired branch name is given, it will then store the differences and set it as its branch name. Some sort of commit message can be stored with the version in the system, so users can type in a message before the version is saved if the option is given. If there is not past version stored, a new “tree” is created for the file. Finally, a short summary of the version status being added is printed.

Checkout:

User calls checkout and rewrites their local file with the version requested. To return a specific version, checkout starts at the “most recent version” node and traverses the DAG, combining all of the diffs up to the designated checkout node. The system downloads the generated version to user’s local hard drive. User can make revisions and save to local hard drive.

Branch:

A branch is created when commit is used on a version that already has a child.

When it comes to rename, we check whether the name is already existing symbol or not. If it is

existing symbol, send error message and if it is not, rename the branch and print summary.

Merge:

Requirements: User should be able to select nodes in different branches and merge the two versions for a given file.

The user will first give the command. Then, our system will take version1 have the next version be the version2 and add the diff of version1 and version2 to both versions. Order of the versions in the command line should not matter, as our program will set the correct parent automatically for the merge node.

<?xml version="1.0"?>

<FileName name="a.txt">

<current Version="1.1"></current>

<Node Version ="1.1">

<BrName>null</BrName>

<Message>null</Message>

<Parent>null</Parent>

<isBranch>F</isBranch>

<Children>

</Children>

</Node>

</FileName>

We plan to use this format to create our DAG at the start of our program. Our stax\_parser class generated and adds nodes, forming our DAG by adding nodes to the parent designated in the XML. We also keep track of all attributes corresponding to each node.

<?xml version="1.0"?>  
<FileName name="abc.txt">  
<superSecretString Version = "1.1">--- temp.txt 2019-12-21 20:22:52.000000000 -0600  
+++ temp2.txt 2019-12-21 20:22:52.000000000 -0600  
@@ -1,3 +1,3 @@  
-high  
-ho  
-high ho  
\ No newline at end of file  
+apple  
+banana  
+coconut  
\ No newline at end of file  
</superSecretString>

</FileName>

Diff and merge use this as described in the diagrams and textual descriptions. After getting the diff between versions, we can then store diffs by creating a new entry in our diff tracking xml that is also compartmentalized by file name, allowing for the storage of multiple files simultaneously. By doing this we can parse for the version and get the diff we want easily for the given file name. Versions are generated by applying (patching) diffs consecutively down the generated DAG tree to the version the user wanted.