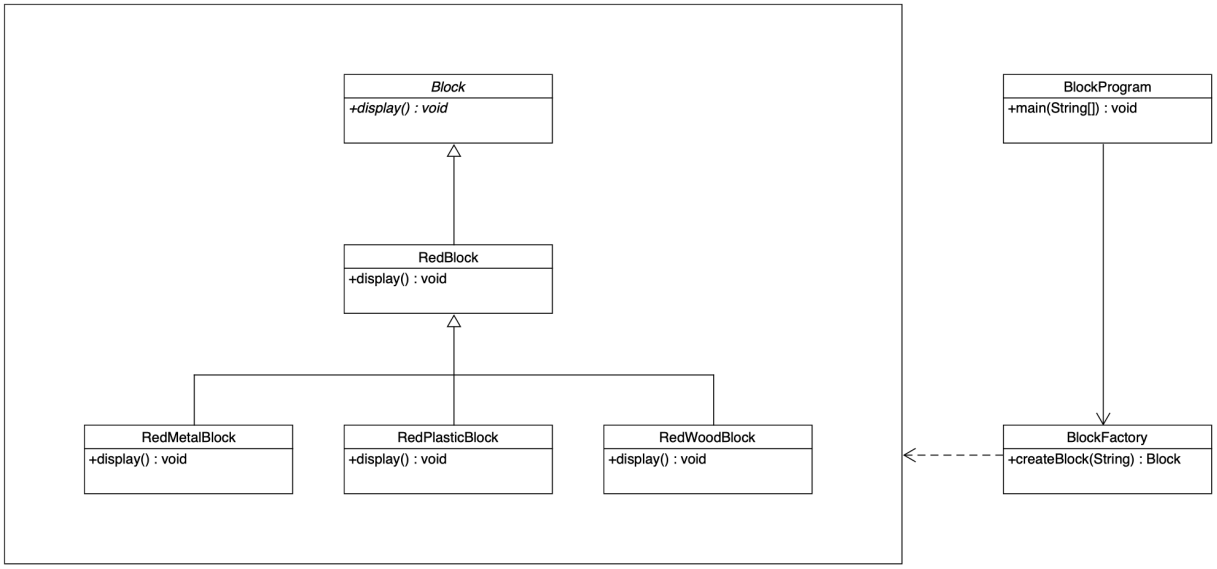
ENSE 375 GitHub Report: Practical Example of Revision Control

Jacob Sauer
22 February 2021

One of the most powerful aspects of the Git revision control framework in software development is the ability to use multiple branches. If several branches are used for a Git-stored project, then one branch can be reserved for stable and well-tested code, while the others can be delegated to the developers as experimentation spaces. Generally, the "main" or "master" branch stores the stablest version of a development team's code. If code is to be deployed to production, it must originate from this master branch. Furthermore, the master branch cannot be edited directly. Instead, any modifications to the code must be performed in a separate branch, and when these changes are finalized, a merge between the separate branch and the master branch must be requested. The multiple-branching system is extremely beneficial for three primary reasons. First, it ensures that the code stored in the master branch is always stable and production-ready. Next, it enables modifications to be performed while ensuring that existing functionalities will be upheld. Finally, it eliminates all risks associated with multi-developer programming, as each member of a development team can edit code in their own branch. As such, multiple-branching is a quintessential asset of development.

To demonstrate these benefits of GitHub revision control, we chose to build a small application with two stages of deployment. The first stage, represented by the class diagram below, drew on the ideals of the Factory design pattern.



The createBlock() method in BlockFactory.java creates an object of type Block, an abstract class, using constructors for one of three concrete classes: RedMetalBlock, RedPlasticBlock, and RedWoodBlock, each of which derive from RedBlock, which derives from Block. The main() function in BlockProgram.java creates six such Block objects - two of each subclass - and then prints the colour and material properties of each object using the display() method, which is first declared in Block.java and later overridden by each derived class.

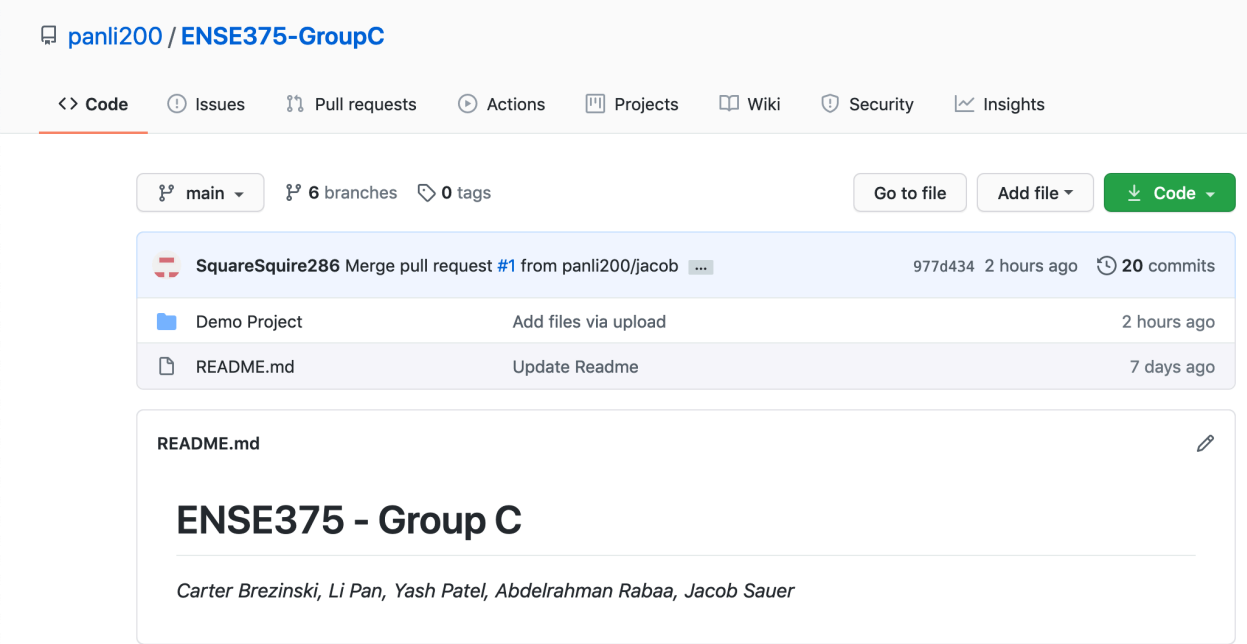
These files were first written in Emacs, committed to the cloned ENSE375-GroupC repository on my personal computer, and pushed to the "jacob" branch of the remote repository. Testing was then conducted, upon which numerous compile-time errors were discovered and debugged. After fixing these compile-time errors, I compared the expected output of BlockProgram.main() to the observed output, and found that four RedMetalBlocks were erroneously created, rather than two RedMetalBlocks and two RedWoodBlocks. This error is displayed in the image below:

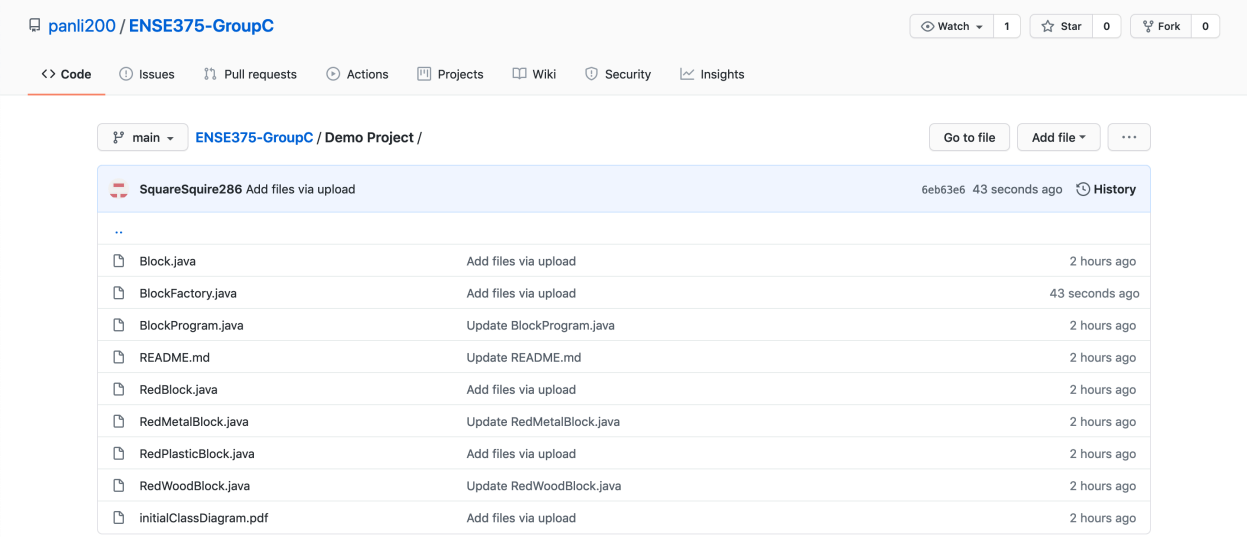
```
[MacBook-Pro-3:Demo Project jacob$ javac Block.java BlockFactory.java BlockProgra]
 m.java RedBlock.java RedMetalBlock.java RedPlasticBlock.java RedWoodBlock.java
[MacBook-Pro-3:Demo Project jacob$ java BlockProgram                             ]
 Red Plastic Block created
 Red Plastic Block created
 Red Metal Block created
 Red Metal Block created
 Red Metal Block created
 Red Metal Block created
```

I traced this issue to a typing error in BlockFactory.createBlock(), in which I accidentally instructed the method to return a new RedMetalBlock for String inputs of both "Metal" and "Wood." Instead, a new RedWoodBlock needed to be returned for the latter String input.

```
import java.lang.IllegalArgumentException;

public class BlockFactory
{
        public Block createBlock(String material)
        {
                if (material == "Plastic")
                        return new RedPlasticBlock();

                else if (material == "Metal")
                        return new RedMetalBlock();

                else if (material == "Wood")
                        return new RedMetalBlock(); // should be RedWoodBlock()

                else throw new IllegalArgumentException("No class definition for the name Red" + material + "Block");
        }
}
```

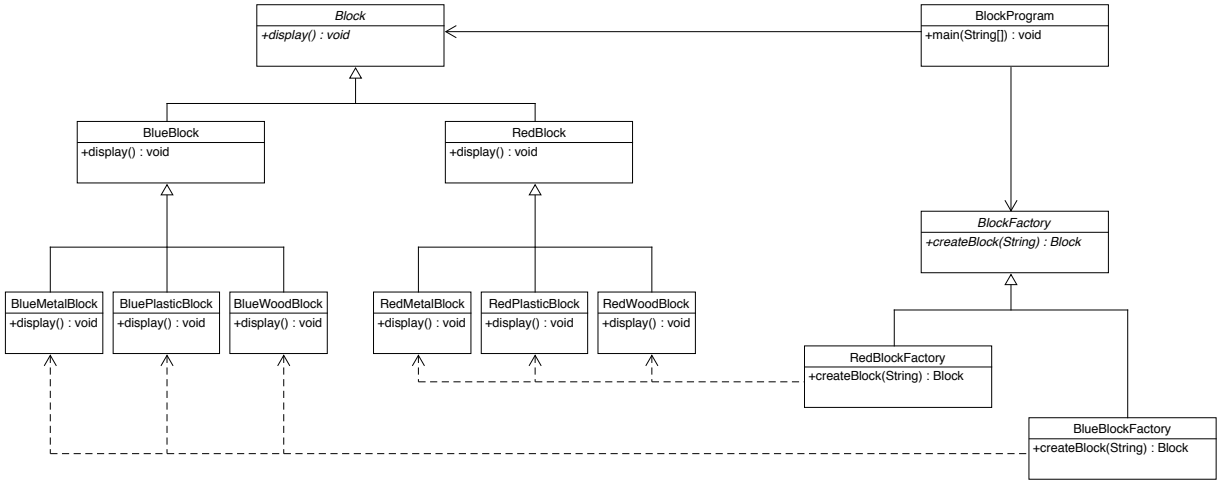
Upon fixing this error and observing the expected output, I committed every updated file to the "jacob" branch, and then merged with the master branch.





At this point in the development process, the master branch of our group's GitHub repository contained a fully functional BlockProgram application, which could instantiate a Block array of RedMetalBlocks, RedPlasticBlocks, and RedWoodBlocks in accordance with the Factory design pattern. However, the application was not yet complete; we envisioned a second stage of deployment oriented around the Abstract Factory design pattern, which would add variants of each Block-deriving class with blue as the primary colour. Therefore, the expected functionality of this second stage of deployment was the ability to create six different types of blocks, rather than only three.

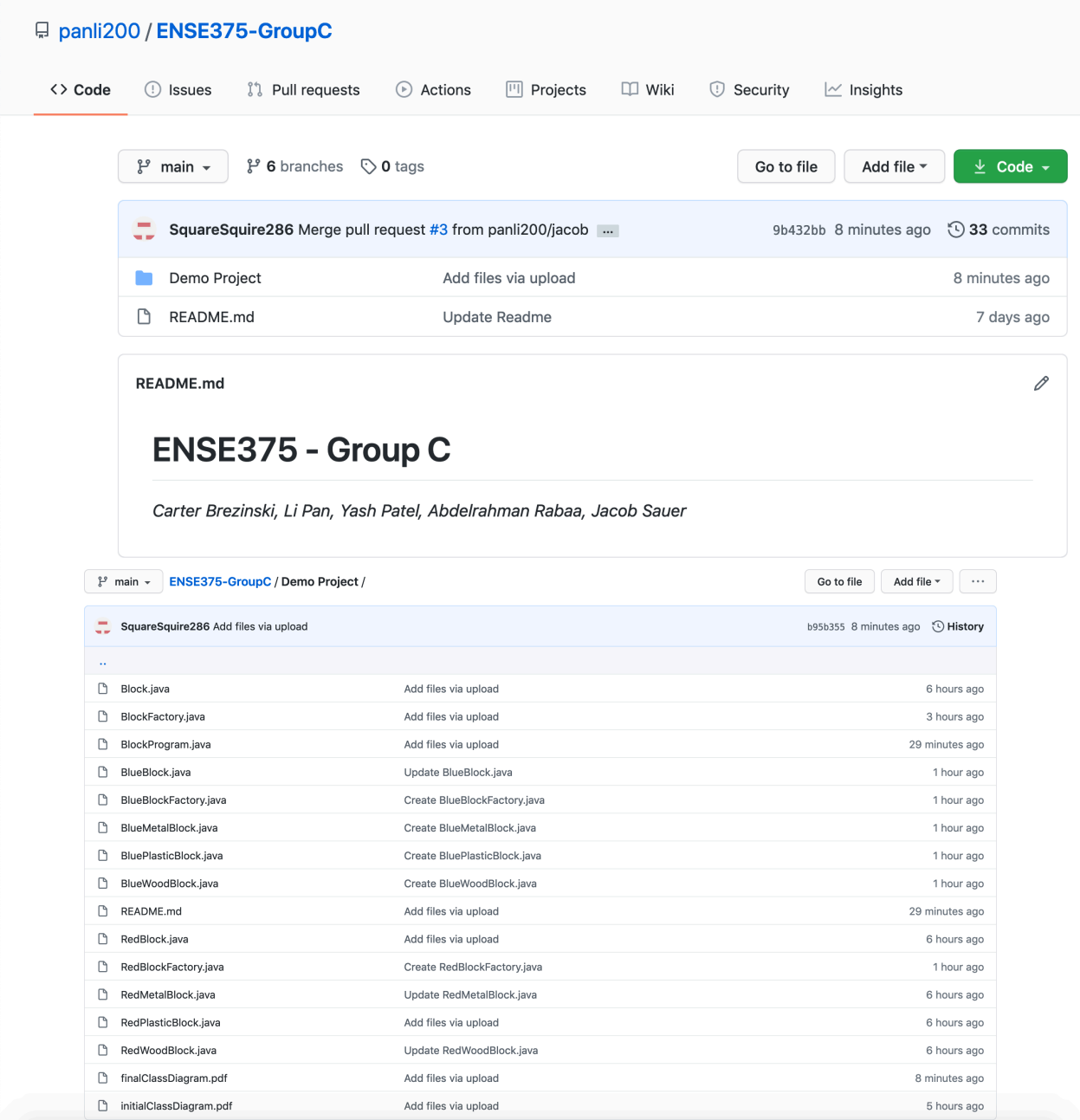
The classes BlueBlock, BlueMetalBlock, BluePlasticBlock, and BlueWoodBlock needed to be written to extend the Block class; the classes RedBlockFactory and BlueBlockFactory needed to be written to extend the now-abstract BlockFactory class; and the code in BlockProgram.main() needed to be modified. The updated class diagram for this application is displayed below:



Once again, I created all of the new classes in Emacs, committed them to my local repository, and then pushed them to the remote repository's "jacob" branch. In using the "jacob" branch, I was able to ensure that none of the fully functional code in the master branch would be affected by these additions and modifications. I conducted all tests on the code in my local repository, ultimately identifying only one syntactical error, and performed a final commit and push upon observing the new expected output from BlockProgram:

```
[MacBook-Pro-3:Demo Project jacob$ javac Block.java BlockFactory.java BlockProgra]
 m.java RedBlock.java RedMetalBlock.java RedPlasticBlock.java RedWoodBlock.java
[MacBook-Pro-3:Demo Project jacob$ java BlockProgram                            ]
 Blue Plastic Block created
 Blue Plastic Block created
 Blue Metal Block created
 Blue Metal Block created
 Blue Wood Block created
 Blue Wood Block created
 Red Plastic Block created
 Red Plastic Block created
 Red Metal Block created
 Red Metal Block created
 Red Wood Block created
 Red Wood Block created
```

Lastly, since BlockProgram.main()'s observed output of six pairs of Block objects matched the expected output, it was appropriate to merge the code in "jacob" with the code in the master branch.



The above example was extremely simple in scope, but nonetheless demonstrated several crucial advantages of revision control. First, the fully functional code in the master branch was always protected. Consequently, had this application been deployed into production prior to completion of the Abstract Factory stage, no complications would have arisen with respect to code functionality. Secondly, in altering the application's design pattern from Factory to Abstract Factory, both modification of existing classes and creation of new classes was necessary. The use of the non-finalized "jacob" branch for these changes was critical, as it ensured that existing Factory functionalities would be temporarily preserved until the Abstract Factory features could

be tested and verified. Writing the code for the BlueBlock derived classes and refactoring BlockProgram.java in the "jacob" branch were, therefore, risk-free.

In general, revision control is an essential strategy for software development teams. It provides opportunities for safe refactoring and extension of existing modules, and enables multiple developers to perform such tasks simultaneously, without sacrificing the integrity of the master branch's code.